UNIVERSIDAD NACIONAL DE COLOMBIA

UIFCE
UNIDAD DE INFORMÁTICA

Guía práctica de econometría básica con Python

Fabián Alejandro Triana Alarcón
Economista, estudiante de Maestría en Ciencias Económicas
Universidad Nacional de Colombia

2019

Econometría básica con Python.

Fabián Alejandro Triana Alarcón

Universidad Nacional de Colombia
Facultad de Ciencias Económicas
2019

Prefacio

Esta obra constituye un manual para la realización de análisis econométrico básico con el uso del lenguaje de programación **Python**; no comprende, como parte de su contenido, un cuerpo *teórico* sólido que pueda tomarse como base adecuada para la enseñanza de la Econometría y tan solo se limita a ser un referente *práctico* de la aplicación de las técnicas y modelos propios de la materia. Dado el enfoque *aplicado* de este trabajo, el lector no encontrará en éste un texto que le sirva de guía teórica y le permita fortalecer sus conocimientos sobre la Econometría, pues no se trata del objetivo aquí planteado; para tal propósito, se sugiere recurrir a los manuales especializados generalmente empleados como material de referencia en los cursos de pregrado en Economía.

El contenido que se presenta en esta obra está especialmente dirigido a estudiantes de Economía, Estadística y carreras afines, a los profesionales cuyas labores se vean asociadas al ámbito de la Econometría y a toda persona, con cierto conocimiento previo, que sienta afinidad por la materia y esté interesada en mejorar sus habilidades en el uso de herramientas informáticas útiles en el desarrollo de ejercicios de econometría aplicada. Dado el mínimo nivel de profundidad con el que se abordan los fundamentos teóricos sobre los que se estructura el análisis econométrico en esta obra, y su marcado enfoque práctico, se espera que el lector posea unos buenos conocimientos previos (a nivel introductorio) sobre la materia, de modo que no le resulte complejo seguir los ejercicios expuestos y pueda alcanzar sus objetivos de aprendizaje.

Además de un conocimiento básico previo sobre Econometría, se espera que el lector esté familiarizado con el uso de herramientas informáticas, de forma que comprenda, sin mayores problemas, las indicaciones dadas para el desarrollo de los ejercicios planteados y las instrucciones brindadas para la instalación y uso de los programas empleados. Asimismo, resulta necesario que el lector posea cierto conocimiento del idioma inglés, en cuanto la mayoría de funciones, métodos y atributos empleados son representados por medio de palabras en inglés, las cuales, dado el muy alto nivel del lenguaje de programación que se empleará, resultan muy informativas y facilitan enormemente la comprensión de las tareas que se llevan a cabo.

Deseo resaltar mi interés en que este trabajo se constituya como referente no teórico y facilite el aprendizaje de la Econometría dado el tratamiento práctico que pretende lograr; asimismo, que permita el acercamiento de toda persona interesada en la materia, con cierto nivel de conocimiento, sin importar el ámbito de su formación, y que promueva el aprovechamiento de herramientas tecnológicas por parte de todos. Aunque cierto conocimiento o experiencia previa con lenguajes de programación son bienvenidos, y resultarán muy útiles para la comprensión de la obra por parte del lector, estos *no* son indispensables pues, con el tratamiento dado a la información en esta obra, pretendo que las exposiciones presentadas sean lo más ilustrativas y claras posibles, de modo que todo lector se sienta cómodo con el estudio de este trabajo y alcance sus metas de aprendizaje satisfactoriamente.

El Autor,
Fabián Alejandro Triana Alarcón

Índice general

Capítulo 1

Instalación de los programas y preparación del entorno

A lo largo de este trabajo se pretende desarrollar ejercicios prácticos, con propósitos ilustrativos, que sirvan de referente para el uso del lenguaje de programación Python como herramienta útil para el análisis econométrico y que permitan al lector familiarizarse con el mismo y fortalecer sus habilidades en el manejo de éste. Esta obra se basa en el uso de **Python 3**, con distribución **Anaconda** y trabajando sobre **Jupyter Notebook**; aunque estos nombres resulten extraños para el lector, especialmente el novato, es adecuado señalar que éste no debería preocuparse, ya que se trata de conceptos sin mayor complejidad (para nuestros propósitos) y que, a continuación, se pretenden aclarar, de modo que sienta entera confianza en el estudio de esta obra.

Python

Python es un lenguaje de programación orientado a objetos, interpretado, de alto nivel y con tipado dinámico (Python Software Foundation,s.f), creado por Guido van Rossum, un científico de la computación y programador holandés, y que apareció en su versión inicial en 1991. Python es de *muy* alto nivel, lo que sugiere que su sintaxis resulta relativamente fácil de comprender y lo que se refleja en el hecho de que su proceso de aprendizaje es más bien sencillo en comparación con el que corresponde a otros lenguajes de programación.

Se ha señalado que trabajaremos con Python 3, sin embargo, no se ha informado al lector sobre la diferencia entre Python, a secas, y Python **3**. Esto no debería causarle intranquilidad, pues la distinción es bien simple: Python es el lenguaje de programación, mientras que Python **3** es una versión de dicho lenguaje (el cual también está disponible en la versión Python 2). En este trabajo, se hará uso de Python 3 ya que es la versión que evidencia una tendencia creciente a la adopción generalizada y se caracteriza por un desarrollo activo.

El uso que se hace de Python a lo largo de esta obra es *aplicado* y *específico*, es decir, se emplea con propósitos muy claramente delineados, ejecutando tareas útiles para abordar el análisis econométrico que se pretende llevar a cabo; en ningún momento este trabajo trata de ser un manual de programación o algo que se le parezca, por lo cual el lector no debe considerar que con el estudio de esta obra aprenderá a programar, pues tal objetivo excede, y por mucho, el alcance de esta guía. Al lector interesado en la programación como tal, se sugiere consultar otro tipo de recursos que le resulten útiles a tal propósito, en cuanto en esta obra se recurre a Python únicamente como herramienta y no como el tema central a abordar.

Tal y como apenas se ha señalado, Python es una herramienta que utilizaremos para desarrollar análisis econométrico; sin embargo, puede que tal planteamiento no resulte del todo claro al lector, quien es natural que se cuestione ¿cómo se hará uso de Python para llevar a cabo análisis econométrico? La respuesta a tal interrogante no implica mayor complejidad: tan solo se hará uso de las funciones, métodos y atributos que nos ofrecen Python y sus librerías especializadas (las cuales abordaremos posteriormente) para manejar los

datos, extraer la información requerida, construir los modelos, realizar las estimaciones y ejecutar las pruebas estadísticas necesarias. Aunque tales tareas sí comprenden operaciones con cierto nivel de dificultad e implican cálculos elaborados, el usuario no tendrá que realizar un esfuerzo extraordinario, de hecho, ni siquiera tendrá que definir una función de dificultad significativa por su propia cuenta o escribir código "complejo", en cuanto Python y sus librerías especializadas brindarán los instrumentos que requeriremos a lo largo de nuestra labor.

Anaconda

Aunque es posible instalar Python usando la distribución oficial del sitio web de Python Software Foundation, https://www.python.org/, no se recurrirá a tal medio, en cuanto se hará uso de la distribución **Anaconda**. La explicación para tal decisión se encuentra en el hecho de que la distribución Anaconda instala automáticamente múltiples librerías (muy útiles para las labores que se adelantarán), incluye un gestor de paquetes propio, *conda*, el sistema de gestión de paquetes estándar de Python (*pip*) y, además, nos da acceso a **Anaconda Navigator**, una interfaz gráfica con la que podemos ingresar a **Jupyter Notebook**, que es donde, en último término, escribiremos el código Python con el que llevaremos a cabo nuestro análisis econométrico.

Anaconda Distribution es un producto de la empresa estadounidense **Anaconda, Inc.**, (anteriormente conocida como Continuum Analytics) y consiste en una distribución Python gratuita y de código abierto ampliamente utilizada en el campo de la ciencia de datos y *machine learning*. Para mayor información, se sugiere consultar el sitio web de la documentación oficial https://docs.anaconda.com/anaconda/, aunque, para nuestros propósitos, esto no resultará ni siquiera necesario, en cuanto aquí se darán las instrucciones requeridas por el lector para seguir correctamente el desarrollo de los temas tratados.

Jupyter Notebook

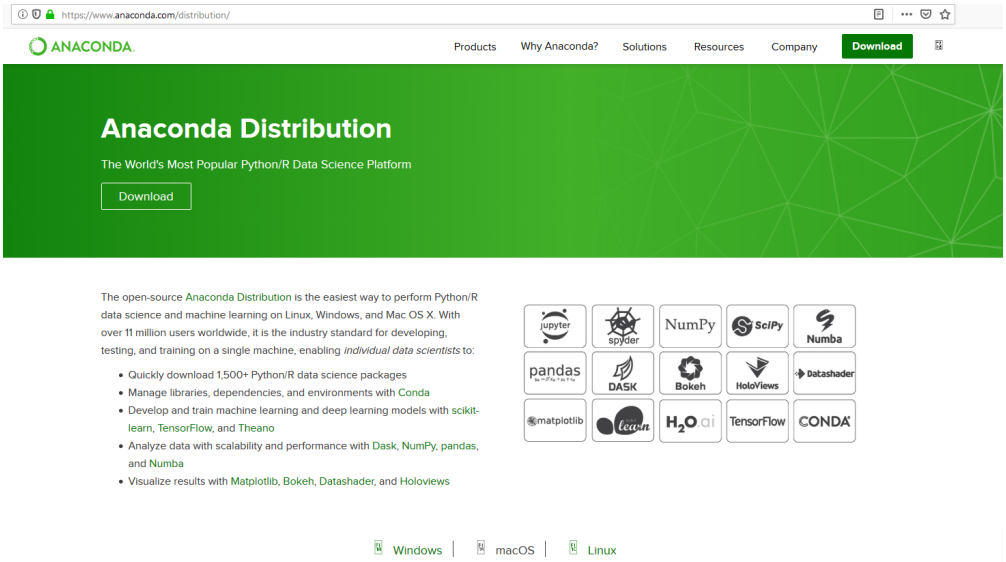
Jupyter Notebook es una aplicación web de código abierto que permite crear y compartir documentos que contienen código, gráficos, ecuaciones y texto (Project Jupyter, 2019). Soporta varios lenguajes de programación, entre ellos **Ju**lia, **Pyt**hon y **R**, cuenta con una interfaz atractiva, de fácil manejo y muy amigable con el usuario, y hace parte de Project Jupyter, una iniciativa surgida en 2014 a partir de IPython y dirigida por Fernando Pérez, físico colombiano de la Universidad de Antioquia, profesor asistente del Departamento de Estadística de UC Berkeley e investigador del Berkeley Institute for Data Science.

La elección de Jupyter Notebook para ser la aplicación en la que se desarrollará nuestro análisis econométrico radica en lo agradable de su interfaz y su manejo extremadamente simple e intuitivo, lo que nos permitirá escribir código, visualizar resultados, elaborar gráficos y realizar comentarios en un mismo documento, enriqueciendo vastamente nuestro análisis y facilitando la comprensión del contenido creado (de hecho, este libro ha sido escrito en Jupyter Notebook, sobre LaTeX y `markdown`, con edición posterior en Overleaf). Para mayor información sobre Project Jupyter, se invita a consultar el sitio web oficial del proyecto: https://jupyter.org/

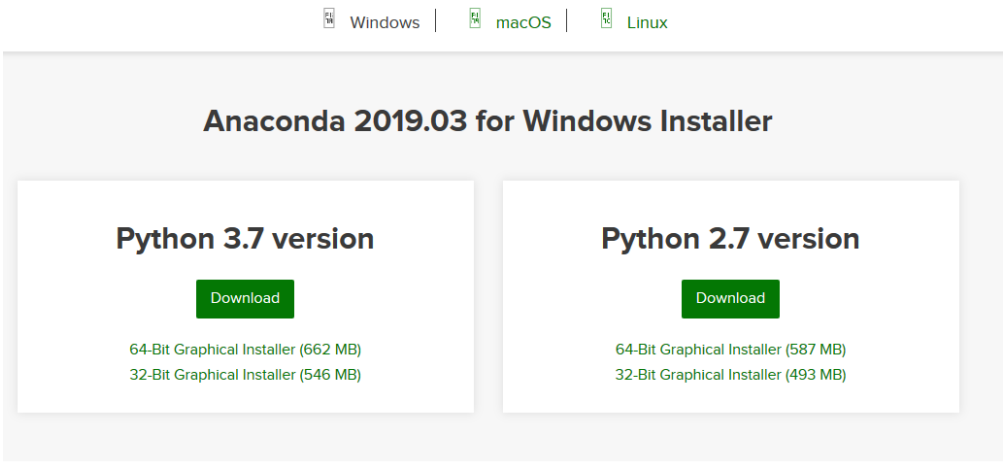
Instalación y primeros pasos

Al comienzo de este capítulo se señaló que "esta obra se basa en el uso de Python 3, con distribución Anaconda y trabajando sobre Jupyter Notebook"; ya se ha dado al lector una breve descripción de cada uno de estos términos, sin embargo, aún no se le ha brindado un conjunto de instrucciones concretas para poder acceder al "material" con el cual trabajaremos. Por lo tanto, en esta sección, se procederá a presentar una descripción detallada de los pasos a seguir para instalar los programas requeridos y preparar el entorno, de modo que se pueda dar inicio efectivo al contenido principal de esta guía práctica de econometría básica con Python.

Lo primero que debemos hacer es contar con la distribución Anaconda. En caso de no tenerla instalada previamente (lo que es muy probable, especialmente para un usuario no familiarizado con el tema), debemos acceder al sitio web oficial https://www.anaconda.com/distribution/. Al ingresar a tal dirección, nos encontraremos con una página como la siguiente:



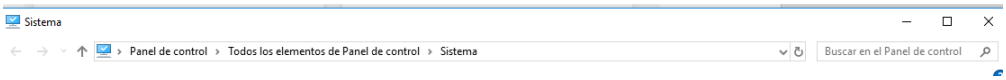
Podemos observar que en la parte inferior se encuentra una sección en la que está escrito `Windows | macOS | Linux`. En este punto, debemos hacer clic sobre el nombre correspondiente al sistema operativo de nuestro computador; esto, con el propósito de acceder al Instalador Anaconda para nuestro sistema operativo. En este caso particular, seleccionaremos `Windows`, sin embargo, el lector tiene que ser cuidadoso, en cuanto debe seleccionar el sistema operativo de su computador (el cual, puede no ser Windows). Al seleccionar el sistema operativo correspondiente, podremos visualizar lo siguiente:



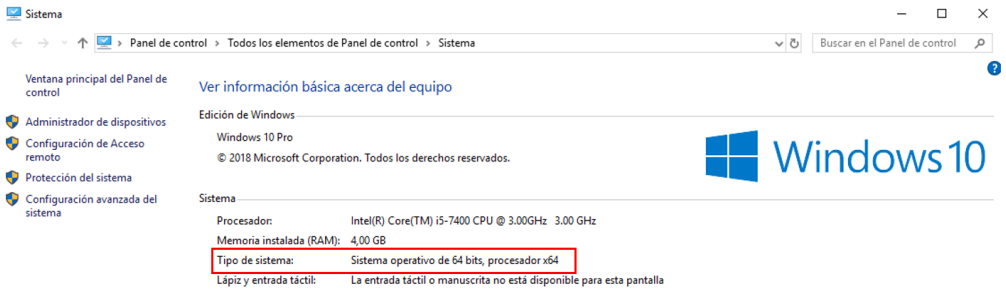
Observamos que hay dos instaladores: uno para Python 3 (versión 3.7[1] específicamente) y otro para Python 2 (versión 2.7[2] específicamente); el que debemos seleccionar, como se ha señalado con anterioridad, es el de

[1] La versión más reciente al momento de publicación de esta obra.
[2] La versión más reciente al momento de publicación de esta obra.

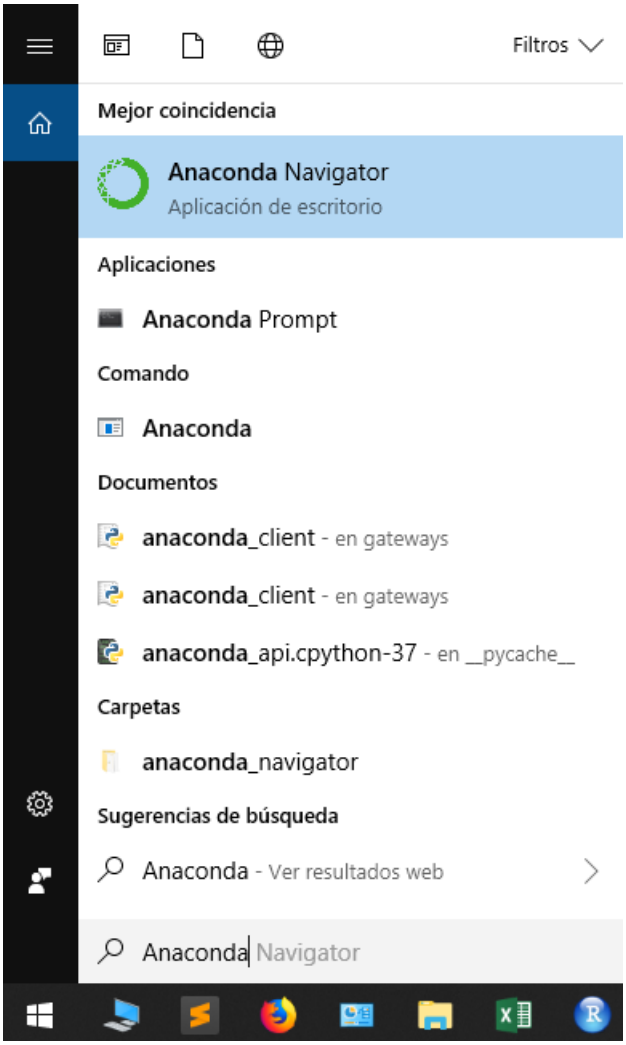
Python 3. En cuanto al Graphical Installer específico, éste depende del procesador de nuestro computador: si se trata de un procesador de 64 bits, naturalmente descargaremos el 64-Bit Graphical Installer, y si se trata de un procesador de 32 bits, nuestra elección será el 32-Bit Graphical Installer. Para el lector que desconozca el sistema de su computador, éste puede ser consultado, en Windows, en las propiedades del equipo o en:



Allí, encontrará, entre otros datos, la información correspondiente al sistema; información similar a la que se presenta a continuación:

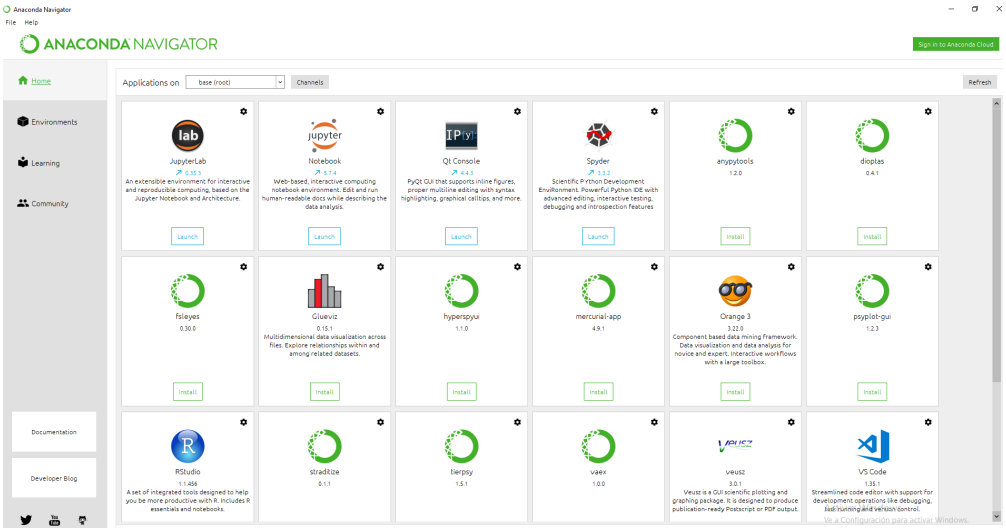


Una vez seleccionado el Graphical Installer correspondiente, solo se debe guardar el archivo y, una vez descargado por completo, se debe ejecutar. Este proceso es muy simple (pues tan solo consiste en seguir instrucciones muy concisas), por lo cual no se expondrá paso a paso, en cuanto hacerlo resultaría superfluo. Una vez descargado e instalado el programa, accederemos a Anaconda Navigator; para tal propósito, recurriremos a una búsqueda desde el botón de inicio de Windows, tal y como se señala a continuación:



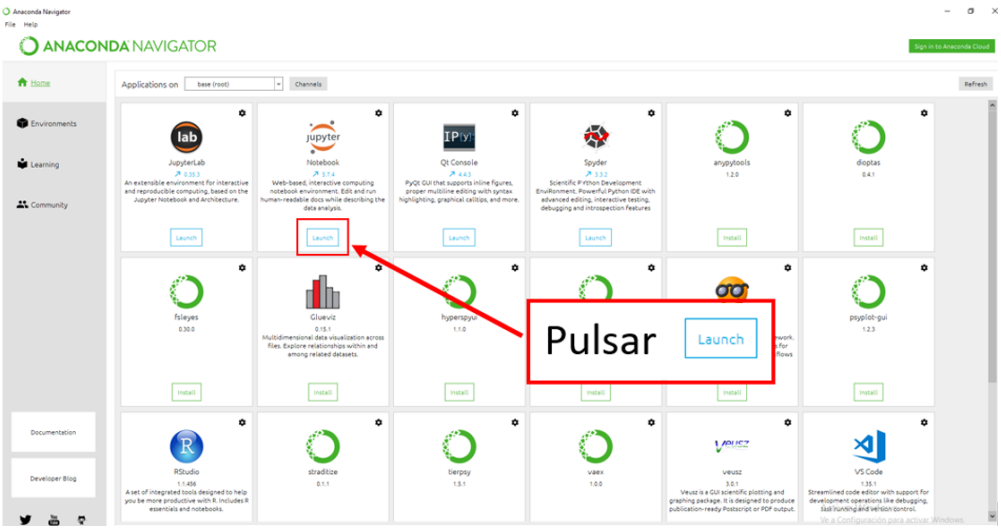
Haremos clic sobre **Anaconda Navigator** (Aplicación de escritorio) y esperaremos que sea cargada por completo (esto puede llegar a tomar un poco de tiempo, por lo que el usuario no debe inquietarse). Recibiremos entonces el siguiente mensaje:

Sobre este mensaje, simplemente debemos pulsar en `Ok` o en `Ok, and don't show again`. Una vez realizado este paso, tendremos acceso a Anaconda Navigator, donde encontraremos una pantalla de inicio similar a la que se presenta a continuación:



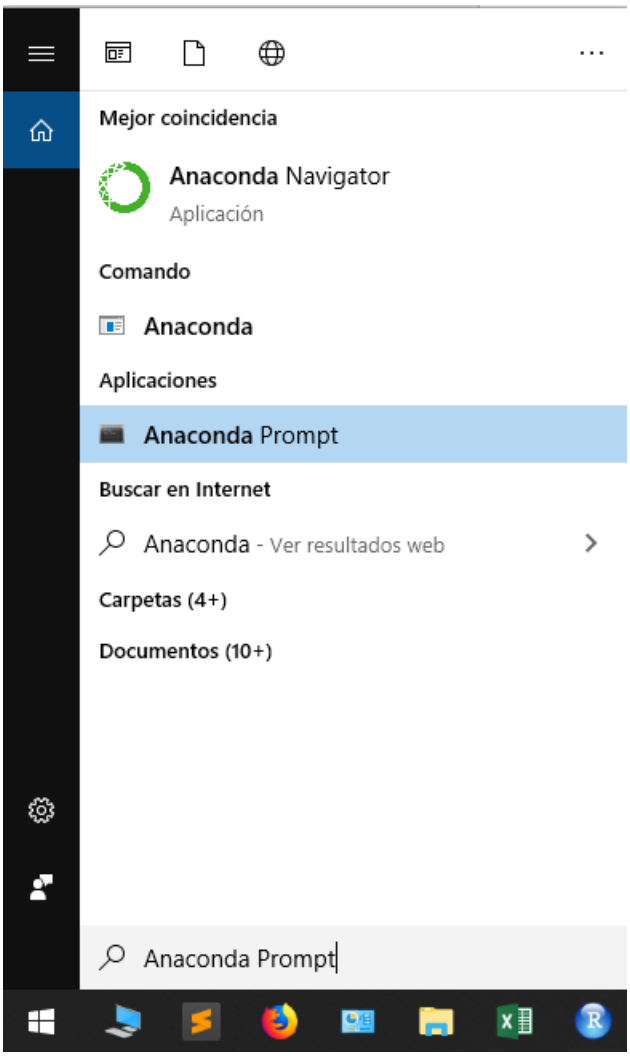
El lector recordará que usaremos Python 3, con distribución Anaconda, trabajando sobre Jupyter Notebook. En este punto, si se han seguido las instrucciones dadas, ya contamos con la distribución Anaconda para Python 3 en nuestro computador; lo único que hace falta es tener acceso a **Jupyter Notebook**, lo que es, a decir verdad, el paso más simple de todos: tan solo consiste en hacer clic.

El lector puede observar que Jupyter Notebook es una de las aplicaciones presentes en la pantalla de inicio de Anaconda Navigator; para ingresar a ésta, tan solo debe pulsar en `Launch`:



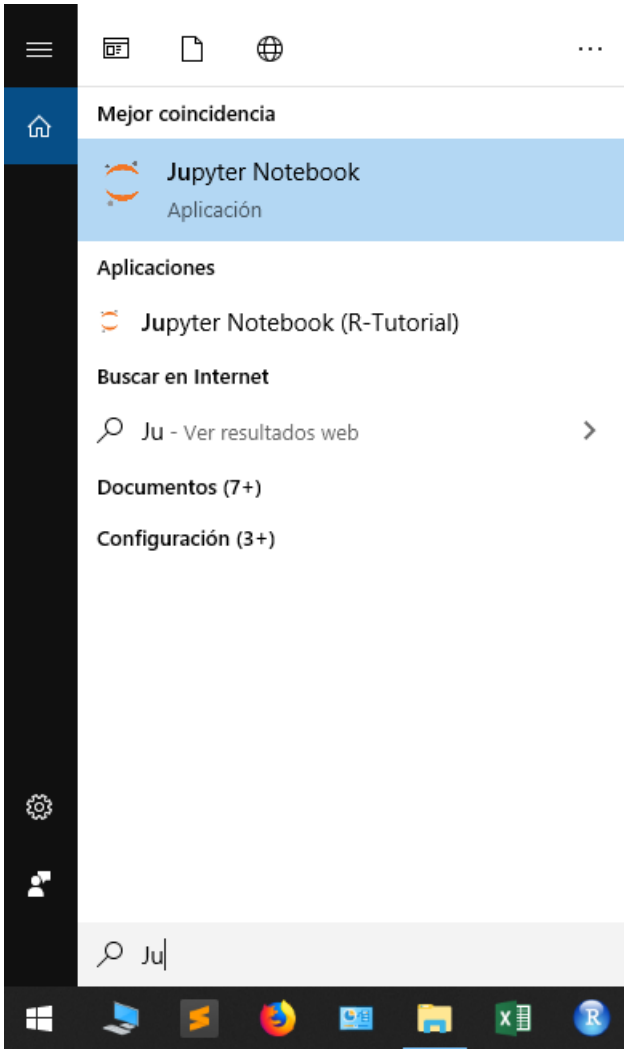
Una vez pulsado, seremos automáticamente redirigidos a *localhost*. El proceso de redirección puede demorar un poco, por lo cual, se recomienda al usuario ser paciente; una vez culminado este proceso, estaremos en Jupyter Notebook, donde finalmente podremos iniciar con nuestro análisis econométrico y entrar en contacto efectivo con Python.

La descripción dada hasta ahora acerca del proceso a seguir para acceder a Jupyter Notebook se ha presentado teniendo en consideración la instalación inmediatamente previa de Anaconda; sin embargo, una vez se cuenta con ésta, no es necesario ingresar a Jupyter Notebook a través de Anaconda Navigator. Una ruta más breve, y con una ejecución más veloz, es el acceso desde **Anaconda Prompt**; basta con hacer la búsqueda y pulsar sobre la aplicación correspondiente:

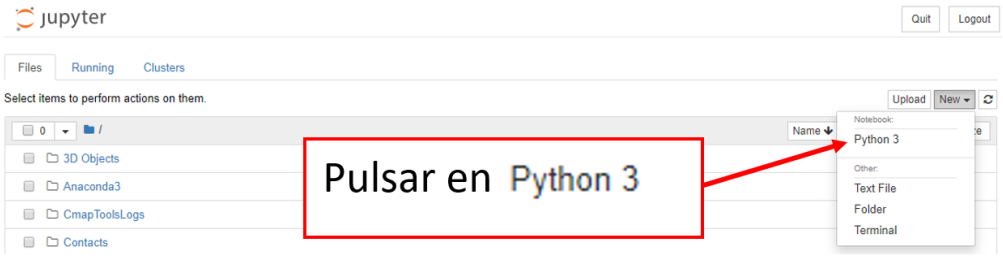


Una vez se ha hecho clic, se debe escribir `jupyter notebook` en Anaconda Prompt y pulsar la tecla `Enter (Intro)`, después de lo cual se abrirá Jupyter Notebook y aparecerá en Anaconda Prompt información adicional sobre el proceso ejecutado.

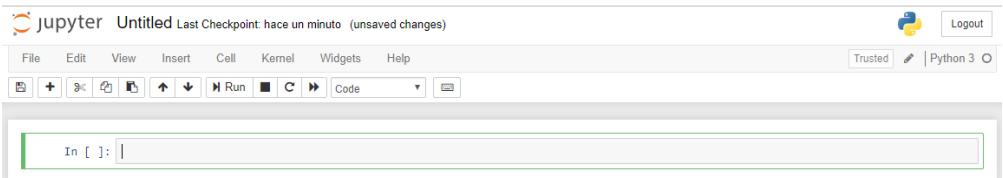
Por último, el modo más sencillo (una vez se tiene instalada Anaconda) y, naturalmente, el más evidente para acceder a Jupyter Notebook es buscando directamente la aplicación y pulsando sobre el ícono correspondiente:



Cualquiera de los 3 métodos de acceso señalados anteriormente es completamente válido y debe permitir al usuario ingresar a Jupyter Notebook; estando en esta aplicación, se debe hacer clic en la pestaña `New` que se encuentra en la sección superior derecha, y pulsar entonces sobre Python 3, como se indica a continuación:

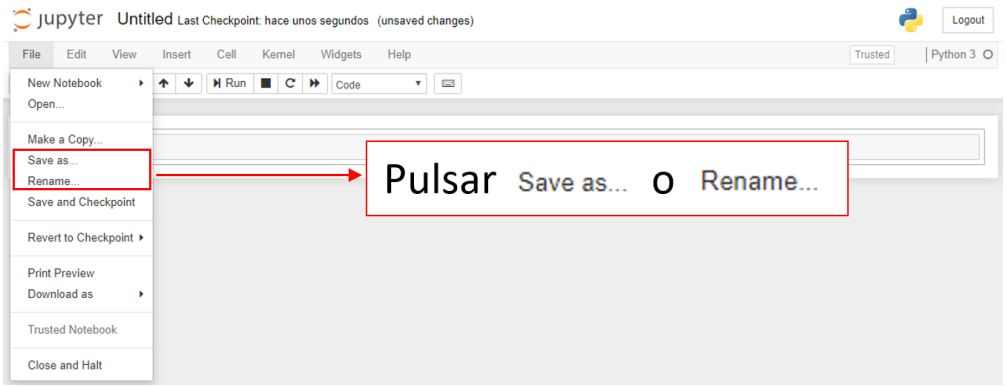


Una vez hemos hecho clic sobre Python 3, se abrirá una nueva pestaña en el navegador, que tendrá una apariencia como la siguiente:

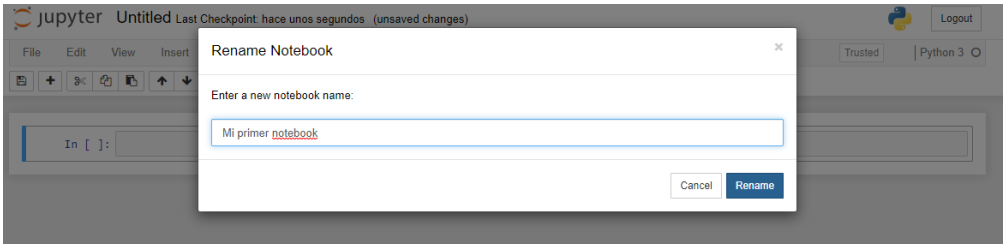


Es en este punto donde empieza nuestro verdadero trabajo, pues es ahora cuando podremos comenzar a escribir código Python y llevar a cabo el análisis econométrico que tenemos planeado.

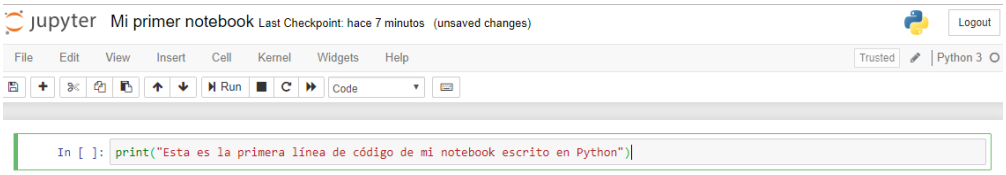
El lector observará que la apariencia del *notebook* de Jupyter es bastante agradable, con cierta similitud a un editor de texto tradicional o un programa ofimático común. En realidad, la sensación de seguridad que transmite esta apariencia 'familiar' efectivamente se ve reflejada en facilidad de manejo: vemos un entorno no sobrecargado y la mayoría de comandos se explican por sí mismos; al usuario no completamente ajeno a la informática le resultará verdaderamente sencillo el uso del *notebook* de Jupyter. Tal vez el lector ha notado que el logo de Python se encuentra en la parte superior derecha del *notebook*, y que cerca de éste está escrito `Python 3`: esto nos indica que el código escrito en el *notebook* corresponde a código Python. Asimismo, observamos que, en la parte superior del *notebook*, al lado del logo de Jupyter Notebook, se encuentra la palabra `Untitled`; éste es el nombre que ha sido asignado por defecto a nuestro *notebook*, el cual, si lo deseamos, podemos cambiar por el de nuestra preferencia haciendo clic sobre `Untitled`, o recurriendo a las opciones `Save as` y `Rename` del menú `File`:



Modificaremos el nombre de nuestro notebook y lo llamaremos, con fines ilustrativos, "Mi primer notebook":

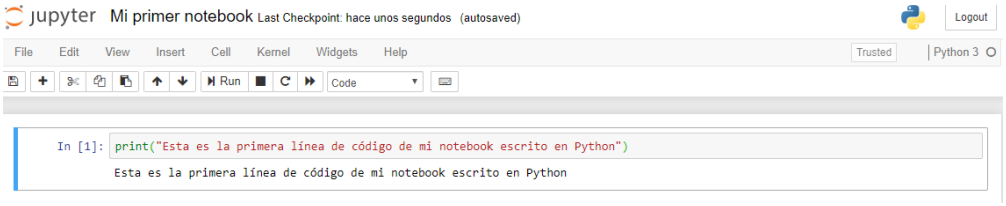


Ahora, procederemos a escribir nuestra primera línea de código; para esto, emplearemos la función predefinida `print(...)`. El lector puede observar que en el *notebook* hay una celda sin ningún contenido; es aquí donde debemos escribir nuestro código. Para este ejemplo específico, el argumento de la función `print(...)` será "Esta es la primera línea de código de mi notebook escrito en Python":



A esta altura, el usuario ha escrito su primera línea de código Python en el *notebook*, pero, ¿este código ha tenido algún efecto? ¿se ha ejecutado alguna acción? La respuesta es simplemente no. Nuestra línea de código

no ha generado ningún resultado ya que tan solo la hemos escrito; aún no la hemos ejecutado. Para ejecutar el código, debemos pulsar el botón `Run`, ubicado en la parte superior, debajo de la ficha `Cell`, o, de forma alternativa y más rápida, emplear la combinación de teclas `Ctrl + Enter`.
Al ejecutar el código, obtendremos lo siguiente:

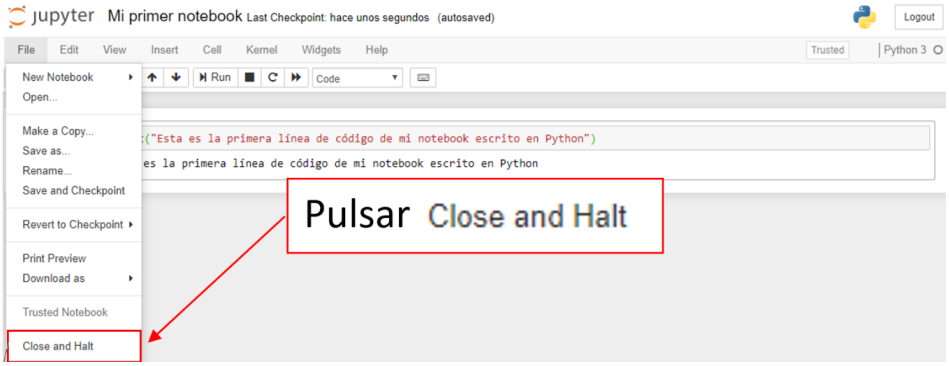


Podemos observar que debajo de nuestra celda ha sido impreso el mensaje contenido en el código y que en los corchetes que se encuentran al lado de la celda de código, los cuales estaban vacíos, ha aparecido el número uno. Este número indica que es la primera ejecución de código que hemos realizado en el *notebook*; si ingresamos de nuevo a la celda y ejecutamos otra vez el código (con `Run` o la combinación `Ctrl + Enter`), observaremos que ya no está el número uno sino el número dos (se invita al lector a comprobarlo por su propia cuenta).

Ya hemos creado un *notebook* de Jupyter, le hemos asignado un nombre y hemos escrito y ejecutado algo de código dentro de éste. Estas son algunas de las instrucciones básicas que el usuario, sin excusa alguna, debe conocer; por último, guardaremos los cambios realizados en el *notebook* y procederemos a detenerlo y cerrarlo. Conociendo este conjunto de operaciones, el usuario tendrá la preparación fundamental requerida para abordar exitosamente el verdadero contenido en el que se especializa esta obra.

El proceso empleado para guardar los cambios efectuados en el *notebook* es muy sencillo y prácticamente idéntico al utilizado con cualquier programa ofimático común, por lo que no representará dificultad alguna a cualquier usuario con conocimiento básico de informática. Basta con usar la combinación de teclas `Ctrl + S`, o hacer clic sobre la ficha de `Save and Checkpoint` (identificada con la forma de un disquete) de la cinta de opciones.

Aunque el *notebook* puede cerrarse cerrando la pestaña del navegador en la que se encuentra, tal procedimiento resulta no recomendable. Para cerrar correctamente el *notebook*, se debe pulsar sobre `Close and Halt` del menú `File`, como se indica a continuación:

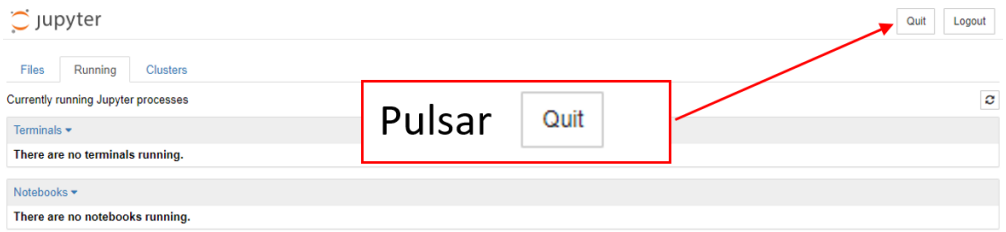


Habiendo realizado tal proceso, la pestaña del navegador se cerrará automáticamente y seremos redirigidos a la pantalla de inicio de Jupyter Notebook, donde debemos encontrar el notebook que hemos apenas creado, guardado y cerrado: 

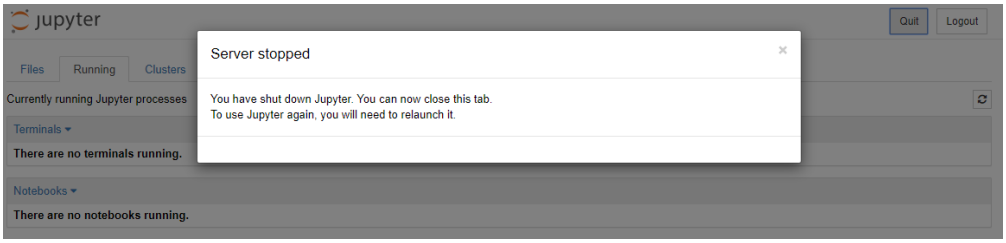
Podemos observar que nuestro notebook "Mi primer notebook"tiene un tamaño de 814 B y ha sido guardado con extensión `.ipynb`; esto se debe a que los *notebooks* de Jupyter son archivos que se almacenan automáticamente con dicha extensión. Para acceder de nuevo al `notebook` basta con hacer clic sobre su nombre; de este modo, se abrirá una nueva pestaña en el navegador en la cual se encontrará el `notebook`.

Es importante resaltar que cada vez que se abre el *notebook* se está "iniciando una nueva sesión", por lo que ninguna línea de código ha sido ejecutada y, aunque es posible visualizar el *output* de las líneas de código, el *output* que se visualiza no es el resultado de ejecuciones de la "sesión" actual. Esto es importante ya que, si se pretende ejecutar una línea de código particular que requiere la ejecución previa de otra línea específica y esta última no se ha ejecutado, se obtendrá un error.

Ya se ha señalado el proceso para ingresar a Anaconda Navigator y a Jupyter Notebook (mencionando 3 alternativas), ahora es momento de abordar el proceso contrario: cómo salir. Aunque es posible salir de Jupyter Notebook con tan solo cerrar el navegador en el que se encuentra, tal proceder resulta inadecuado. El método correcto para salir de Jupyter Notebook es cerrar (apropiadamente) los notebooks activos y, en seguida, cerrar Jupyter Notebook haciendo clic sobre la ficha `Quit` que se encuentra en la sección superior derecha:



Una vez pulsada, obtendremos el siguiente mensaje:



Ahora sí, podemos cerrar con entera tranquilidad el navegador. En cuanto a Anaconda Navigator, después de haber realizado el proceso apenas descrito, si está abierta, basta con cerrarla como si fuera un programa ofimático cualquiera, con tan solo pulsar la ficha con la `x` en el extremo superior derecho.

Para volver a acceder a Jupyter Notebook tan solo se requiere repetir las instrucciones necesarias que se han indicado a lo largo de este capítulo, las cuales, en realidad, no son más que acceder a Anaconda Navigator y, desde esta aplicación, ingresar a Jupyter Notebook (o simplemente acceder desde Anaconda Prompt o Jupyter Notebook directamente) siendo redirigido a *localhost*.

Con este capítulo se cierra el tratamiento de los programas a emplear; ya es hora de entrar en la verdadera materia que aborda esta obra, por lo que, a partir del siguiente capítulo, se desarrollará el propósito de esta guía práctica: análisis econométrico con Python.

Capítulo 2

Exploración de los datos

Resulta imposible llevar a cabo un análisis sin tener datos que analizar; esto no es más que un absurdo. Así, si queremos realizar un ejercicio de análisis econométrico, debemos contar con la información requerida para poder llevarlo a cabo; el lector no debe preocuparse por este comentario, en cuanto, además de la guía práctica que se le ofrece, también se le brinda la información con la cual se construye los modelos y se estructura el análisis.

Para cada uno de los ejercicios que se desarrollarán en esta obra se indicará la fuente de la cual se ha obtenido la información empleada en el análisis econométrico correspondiente. Asimismo, es posible obtener los datasets utilizados, en los mismos formatos que se emplean en esta obra, contactando al autor de la misma (basta con enviar un correo a la dirección **fatrianaa@unal.edu.co**), quien le dará acceso a la información requerida, o, alternativamente, acceder al sitio web de la Unidad de Informática y Comunicaciones de la Facultad de Ciencias Económicas de la Universidad Nacional de Colombia.

Esta obra, dado su carácter práctico y la utilidad que puede llegar a significar para los estudiantes de pregrado en Economía, se apoya, principalmente, en *Econometría* de Gujarati y Porter (2010) e *Introducción a la econometría. Un enfoque moderno* de Wooldridge (2010), textos muy sencillos ampliamente empleados como guía en los cursos básicos de Econometría, a los que en este trabajo se recurre, en múltiples ocasiones, para desarrollar los ejercicios ilustrativos planteados y con los que se pretende que el lector pueda contrastar resultados, de modo que evidencie que los procedimientos realizados son correctos y llegan a las soluciones esperadas.

El dataset que se utilizará para la regresión lineal simple por Mínimos Cuadrados Ordinarios que se tratará en el próximo capítulo es el empleado en el Ejemplo 7.1, *Mortalidad infantil en relación con el PIB per cápita y la tasa de alfabetización de las mujeres*, de Gujarati y Porter (2010). Se invita al lector a consultar el ejemplo, de modo que pueda verificar los resultados obtenidos.

La fuente de la información usada por Gujarati y Porter (2010) es "Chandan Mukherjee, Howard White y Marc Whyte, Econometrics and Data Analysis for Developing Countries, Routledge, Londres, 1998, p. 456". Los datos usados para el ejercicio en Python que se lleva a cabo en esta obra se obtuvieron del fichero de datos de Gujarati en **gretl**.

En este punto empieza nuestro trabajo con Python. Lo primero que debe hacer el usuario es acceder a Jupyter Notebook (a través de Anaconda Navigator o por el medio de su preferencia) y crear un nuevo notebook. En este notebook es donde se escribirá el código requerido y se realizará el análisis econométrico planteado.

Preparación del entorno

La primera celda en la que el usuario escribirá y ejecutará código tendrá el siguiente contenido:

```
In [1]: # Importamos las librerías requeridas:
        import numpy as np
        import pandas as pd
        import statsmodels.api as sm
        import matplotlib.pyplot as plt
        import seaborn as sns
        %matplotlib inline
        plt.style.use("seaborn-white")
```

¿De qué tratan estas líneas de código? El lector notará que lo primero que se ha escrito, a modo de comentario (los comentarios van precedidos de #), es "Importamos las librerías requeridas"; esto es lo que, efectivamente, se consigue al ejecutar esta celda de códigos. Básicamente, lo que logramos es preparar nuestro entorno de trabajo al obtener acceso al conjunto de herramientas que requeriremos para nuestro análisis; tales herramientas hacen parte de **librerías**.

Una librería simplemente es un conjunto de funciones y métodos diseñados para realizar tareas específicas, agrupados en un único espacio (la librería), y que se agrega a Python "básico" con el objetivo de alcanzar resultados que con sus funciones integradas (nativas) no pueden lograrse fácilmente. Al lector familiarizado con el lenguaje de programación **R** basta con decir que una librería Python es, esencialmente, equivalente a un paquete de R.

Las librerías que importamos son *NumPy*, *pandas*, *Statsmodels*, *Matplotlib* y *Seaborn*; cada una de estas tiene un rol particular a desempeñar en nuestro trabajo, el cual se presentará, junto a breves descripciones, en un momento. Algo que debe notar el lector es que la importación de las librerías no consiste tan solo en "importar las librerías", sin especificar algún tipo de instrucción adicional: la librería *NumPy* no se importa simplemente como `import numpy` sino que se importa como `import numpy as np`. ¿A qué se debe esto?, ¿qué efecto genera?

El proceso de importación con `as`, que es una palabra reservada de Python, se lleva a cabo con el propósito de asignar un **alias** a las librerías importadas. El lector puede estar preguntándose: si la librería ya tiene un nombre, ¿para qué se le quiere asignar un alias? La respuesta al interrogante es muy sencilla: tan solo por practicidad. Al importar la librería *NumPy* con el alias de `np`, tal librería queda identificada con este nombre (np); por lo tanto, cada vez que se necesite emplear una función o método de *NumPy* basta con hacer referencia a `np` y no a `numpy`.

Tal vez la utilidad de `as` resulte más evidente en el caso de la librería *Matplotlib*: en vez de hacer referencia a `matplotlib.pyplot` en cada ocasión que se requiera una función o método proveniente de éste, basta con hacer referencia a `plt`. Así, en vez de usar 17 caracteres, tan solo se requiere emplear 3; aunque al lector esto le parezca trivial en este momento, con el tiempo verá que la importación con `as` resulta tremendamente práctica y facilita enormemente el trabajo.

Las librerías importadas son *NumPy*, *pandas*, *Statsmodels*, *Matplotlib* y *Seaborn*. ¿Por qué se importa este grupo específico de librerías y no otras? Para comprenderlo, se ofrece una breve descripción de cada una de estas, de modo que el lector tenga claridad sobre el propósito de las mismas.

- *NumPy* es una librería Python para computación científica que provee un conjunto de rutinas para la ejecución de procedimientos sobre objetos, como vectores y matrices, los cuales incluyen, entre otros, operaciones matemáticas y lógicas, álgebra lineal básica, simulación aleatoria y estadística fundamental (The SciPy community, 2019).

- *pandas* es una librería de código abierto que ofrece herramientas de alto desempeño y fáciles de usar para el manejo y análisis de datos en Python. Se trata de una librería que permite llevar a cabo un proceso óptimo de análisis de información directamente en Python, sin tener que recurrir a un lenguaje con mayor especificidad, como R. *pandas* es empleada tanto en ámbitos académicos como comerciales, en campos

diversos que incluyen las finanzas, la neurociencia, la estadística y la economía (Pandas, s.f). El creador y *Benevolent Dictator for Life* de *pandas* es el matemático y desarrollador estadounidense Wes McKinney.

- *Statsmodels* es una librería para análisis econométrico y estadístico en Python que provee funciones para la estimación de múltiples modelos estadísticos, así como la realización de pruebas y análisis de información estadística. Los resultados generados por las funciones de la librería son contrastados con los de paquetes estadísticos reconocidos, de modo que se pueda garantizar su exactitud. La librería tiene su origen en el módulo `models` de `scipy.stats` y en 2009, tras un proceso de corrección, prueba y mejora, se lanzó *Statsmodels* de forma independiente (Perktold, Seabold, y Taylor, 2018).

- *Matplotlib* es una librería de gráficos 2D usada en Python para la generación de imágenes de alta calidad. Con `Matplotlib` es posible generar histogramas, gráficos de barras, diagramas de dispersión, entre otros, mediante el uso de tan solo unas cuantas líneas de código. El módulo *pyplot* provee una interfaz similar a `MATLAB`, útil para gráficos sencillos, que permite un control completo de las distintas propiedades. *Matplotlib* es una contribución de John Hunter, un neurobiólogo estadounidense, y de sus múltiples colaboradores (The Matplotlib development team, 2018).

- *Seaborn* es una librería para la elaboración de gráficos estadísticos en Python. Está basada en *Matplotlib* y sus funciones de construcción de gráficos se orientan a datasets, por lo que está estrechamente integrada con las estructuras de datos de *pandas*. El propósito de *Seaborn* es constituir la visualización en un elemento clave en la exploración y comprensión de información (Waskom, 2018).

El lector, teniendo en consideración las breves descripciones apenas presentadas, ya debería tener una idea del papel que desempeña cada una de las librerías mencionadas en el análisis que se pretende llevar a cabo; sin embargo, si éste no es el caso y la información previamente expuesta le resulta un poco confusa, en palabras sencillas la utilidad de cada una de las librerías es la siguiente:

- **NumPy:** Análisis numérico.
- **pandas:** Manejo de datos.
- **Statsmodels:** Construcción de los modelos y realización de las estimaciones.
- **Matplotlib:** Elaboración de gráficos (generales).
- **Seaborn:** Funciones adicionales para la elaboración de gráficos (especializados).

Una vez ejecutadas las líneas de código de la primera celda, se habrán importado las librerías requeridas y tendremos a disposición el conjunto de herramientas de las cuales haremos uso para llevar a cabo el análisis econométrico. Ahora, procederemos a cargar la información con la que trabajaremos.

Importación de los datos

La importación de los datos se lleva a cabo con funciones de la librería *pandas*. Dependiendo del tipo de archivo en el cual se encuentre almacenada la información del dataset, deberemos recurrir a una función particular y un conjunto de parámetros específicos.

El dataset que usaremos corresponde a un archivo `.txt`, por lo que se recurrirá a la función `read_csv(...)` de *pandas* y se señalarán los valores correspondientes al parámetro *delimiter*.

- **Nota:** El parámetro obligatorio de la función `read_csv(...)` es el *filepath_or_buffer* y corresponde a la ruta del archivo, es decir, la ubicación del archivo que contiene el dataset, expresada como *string*.

Para nuestro caso en particular, el siguiente código es el que importa el dataset:

```
In [2]: data = pd.read_csv("C:/Users/FCE/Documents/Econometrics/GujaratiPorter71.txt",
                           sep = " ", delimiter="\t")
```

El lector debe notar que no se ha escrito `read_csv(...)` sino `pd.read_csv(...)`. Esto se debe a que `read_csv(...)` no es una función "nativa" de Python sino que pertenece a *pandas*, por lo cual es necesario especificar de dónde proviene. Dado que la importación se realizó con `import pandas as pd`, el código `pd.read_csv(...)` informa que la función `read_csv(...)` pertenece a *pandas*. Si la importación no se hubiera llevado a cabo con el uso de un alias, se debería haber recurrido a `pandas.read_csv(...)` para conseguir el resultado.

- **Nota:** El usuario debe tener en cuenta que el valor que asignará al parámetro *filepath_or_buffer* no será el mismo que el del código del ejemplo anterior, pues éste depende de la ubicación del archivo *GujaratiPorter71.txt* (o el archivo en el que haya guardado el dataset) en su computador.

Una vez ejecutada la línea de código de esta celda, el dataset debería haber sido importado correctamente. Para comprobarlo, se recurre al método `.head()`, aplicado al objeto que contiene el dataset (en este caso se le ha asignado el nombre `data`):

```
In [3]: data.head()
```

```
Out[3]:
        CM  FLR  PGNP   TFR
0  128   37  1870  6.66
1  204   22   130  6.15
2  202   16   310  7.00
3  197   65   570  6.25
4   96   76  2050  3.81
```

Podemos observar que el proceso de importación ha sido exitoso y la información del dataset ha sido almacenada correctamente en un `DataFrame` de *pandas*. Ahora, podemos continuar tranquilamente con el desarrollo de nuestro análisis, en cuanto hemos concluido satisfactoriamente el primer paso.

Estadística descriptiva

Antes de empezar a examinar los valores de las variables del dataset e identificar las posibles relaciones que hay entre estas, es recomendable hacer un reconocimiento del propio dataset, es decir, obtener información sobre sus dimensiones, el tipo de datos que contiene, etc. Para conocer las dimensiones del dataset puede recurrirse al atributo `.shape`:

```
In [4]: data.shape
```

```
Out[4]: (64, 4)
```

El resultado de la aplicación del atributo `.shape` es una tupla Python (...) en la que se informa el número de filas y de columnas (en ese orden) que tiene el `DataFrame`; así, nuestro dataset contiene 64 filas y 4 columnas.

Ahora, sabemos que son 4 columnas, pero ¿cómo se llaman? ¿cuál es el nombre de cada una de estas 4 columnas? Cuando se ha aplicado el método `.head()` se han visualizado las primeras observaciones del dataset, incluyendo el encabezado en el que se encuentran los nombres de las columnas; sin embargo, si específicamente se quiere saber el nombre de las columnas, puede emplearse el atributo `.columns`:

```
In [5]: data.columns

Out[5]: Index(['CM', 'FLR', 'PGNP', 'TFR'], dtype='object')
```

Con toda la información obtenida se sabe que el dataset con el que se trabajará tiene 64 filas y 4 columnas, y que los nombres de las columnas son 'CM', 'FLR', 'PGNP' y 'TFR'. Sin embargo, aún se desconoce qué tipo de datos contienen dichas columnas, ¿se trata de palabras? ¿de números? ¿solo números enteros? Para descubrirlo, puede emplearse el atributo `.dtypes`:

```
In [6]: data.dtypes

Out[6]: CM        int64
        FLR       int64
        PGNP      int64
        TFR     float64
        dtype: object
```

La información generada por el atributo `.dtypes` indica que tres variables (columnas) contienen datos de tipo `int64` (número entero) y una de tipo `float64` (número decimal).

Finalmente, una manera de obtener la información que se halla con la aplicación de los tres atributos apenas descritos es el método `.info()`, el cual permite visualizarlos en un único espacio:

```
In [7]: data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 64 entries, 0 to 63
Data columns (total 4 columns):
CM      64 non-null int64
FLR     64 non-null int64
PGNP    64 non-null int64
TFR     64 non-null float64
dtypes: float64(1), int64(3)
memory usage: 2.1 KB
```

El resultado de la aplicación de este método es tan solo un resumen de la información que hemos obtenido previamente. Adicionalmente, se presenta el dato sobre el uso de memoria asociado, que en este caso es de 2.1 Kb; el tipo de objeto al que corresponde el dataset, el cual es un `DataFrame` de *pandas*; y el rango del índice, el cual corresponde al intervalo $[0, 63]$ (es muy importante tener en cuenta que la indización en Python inicia desde 0 y **no** desde 1).

Para obtener estadística descriptiva de las variables numéricas del dataset se puede recurrir al método `.describe()`:

```
In [8]: data.describe()

Out[8]:                CM        FLR          PGNP        TFR
        count   64.000000  64.000000     64.000000  64.000000
        mean   141.500000  51.187500   1401.250000   5.549687
        std     75.978067  26.007859   2725.695775   1.508993
```

```
min      12.000000   9.000000    120.000000   1.690000
25%      82.000000  29.000000    300.000000   4.607500
50%     138.500000  48.000000    620.000000   6.040000
75%     192.500000  77.250000   1317.500000   6.615000
max     312.000000  95.000000  19830.000000   8.490000
```

Tal y como observamos, el método `.describe()` nos permite obtener información estadística básica sobre las variables numéricas, lo que incluye, por ejemplo, el promedio, los valores máximo y mínimo y la desviación estándar. Así, se puede evidenciar, por ejemplo, que el valor mínimo de la variable 'CM' es 12, el valor máximo de la variable 'PGNP' es 19,830 y el promedio de la variable 'TFR' corresponde a 5.55.

Al aplicar el método `.describe()` sobre el `DataFrame` se obtiene, por defecto, información estadística básica de todas las variables numéricas presentes en éste. Si solo se está interesado en la información de una variable en particular, esta puede ser seleccionada por medio del uso de los corchetes [ ], indicando su nombre (entre comillas) dentro de estos.

Así, si, por ejemplo, solo nos interesa la variable 'CM', el código a emplear será el siguiente:

```
In [9]: data["CM"].describe()

Out[9]: count     64.000000
        mean     141.500000
        std       75.978067
        min       12.000000
        25%       82.000000
        50%      138.500000
        75%      192.500000
        max      312.000000
        Name: CM, dtype: float64
```

Si, por el contrario, lo que nos interesa es un grupo de variables y no solo una, podemos seleccionarlo haciendo uso de los corchetes [ ] e incluyendo los nombres de las variables de interés (entre comillas, separados por comas) dentro de una **lista** Python [...]:

```
In [10]: data[["CM", "PGNP", "FLR"]].describe()

Out[10]:                CM          PGNP        FLR
         count   64.000000     64.000000  64.000000
         mean   141.500000   1401.250000  51.187500
         std     75.978067   2725.695775  26.007859
         min     12.000000    120.000000   9.000000
         25%     82.000000    300.000000  29.000000
         50%    138.500000    620.000000  48.000000
         75%    192.500000   1317.500000  77.250000
         max    312.000000  19830.000000  95.000000
```

A esta altura se sabe que el dataset contiene información sobre 4 variables numéricas (3 de valores enteros y 1 de valores no enteros), contando con datos de 64[1] individuos (entidades) para cada una de estas. Ya se ha

[1] En este caso, la información de cada individuo (entidad) corresponde a una única fila, por lo que, como el dataset tiene 64 filas, sabemos que hay información sobre 64 individuos (entidades). Sin embargo, en otros casos, como los paneles de datos, la información del mismo individuo (entidad) no corresponde a una única fila, por lo que hay que ser cuidadosos en tales situaciones.

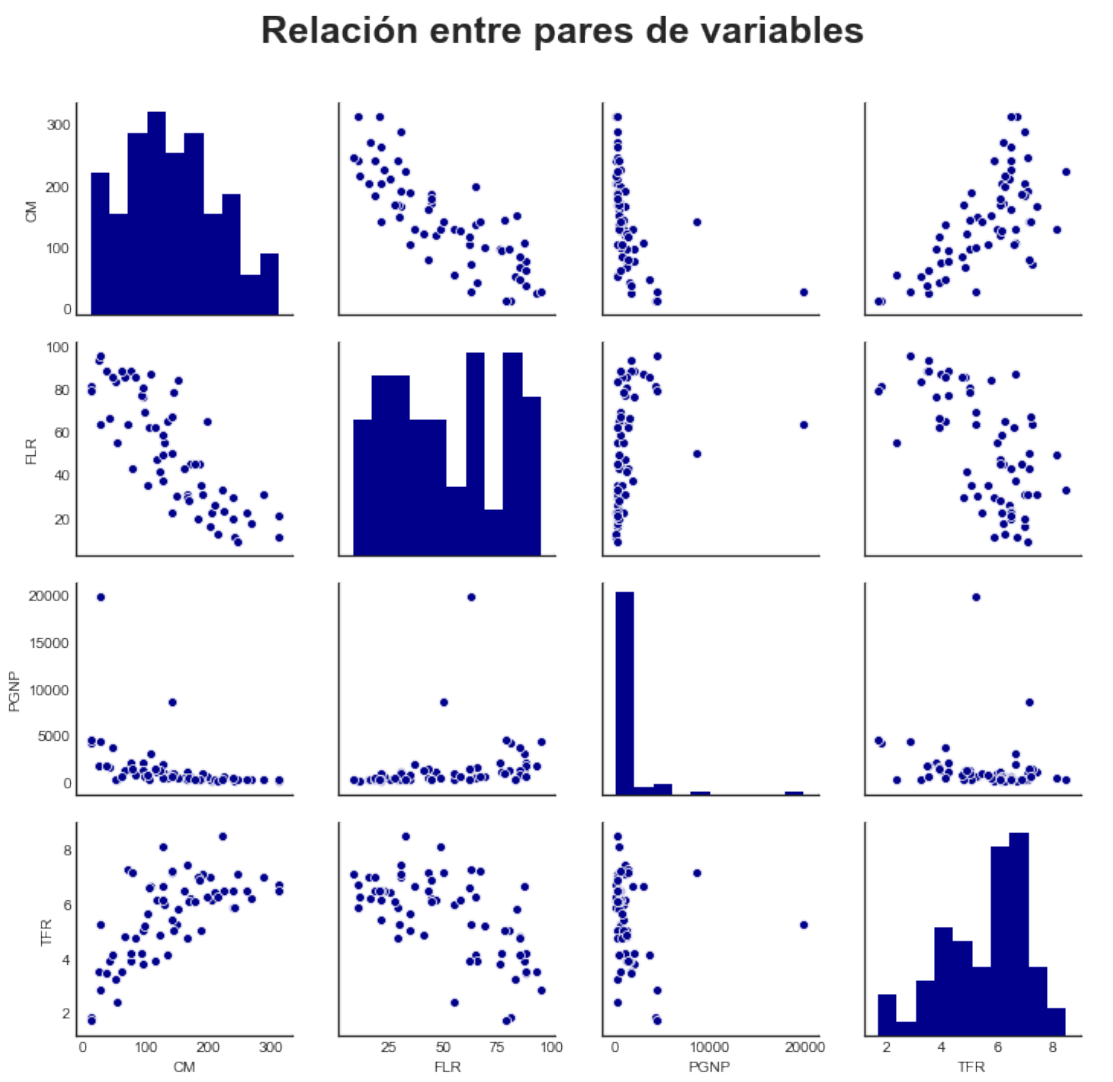
realizado un muy breve reconocimiento de los datos, por lo que se tiene un mayor nivel de familiarización con los mismos y es posible identificar potenciales relaciones a examinar.

Hasta el momento, se ha llevado a cabo un proceso de reconocimiento de los datos por medio del cual se ha obtenido información puramente numérica: el número de filas y columnas, la media y la desviación estándar de cada variable, etc. Esto nos da cierta idea sobre los datos, sin embargo, con frecuencia, "una imagen vale más que mil palabras"[2]; es posible que un gráfico tenga un inmenso poder comunicativo y contribuya enormemente a obtener una mejor comprensión acerca de los datos con los que se trabajará.

La función `pairplot(..)` de *Seaborn* resulta particularmente útil para el contexto en el que nos encontramos; esta función crea una cuadrilla en la que podremos visualizar la relación que existe entre pares de variables (por medio de diagramas de dispersión) y la distribución de cada una de estas (por medio de histogramas):

```
In [11]: g = sns.pairplot(data, plot_kws = {"color": "darkblue"},
                          diag_kws = {"color":"darkblue"})
         g.fig.suptitle("Relación entre pares de variables",
                        fontsize = 25,
                        fontweight = "bold")
         plt.subplots_adjust(top=0.9)
         g.fig.text(1,-.02,
                    "Elaboración:",
                    fontsize = 13, fontweight = "bold",
                    ha = "right")
         g.fig.text(1,-.05,
                    "Triana, F.\n(2019)",
                    fontsize = 12, ha = "right")
         plt.show()
```

[2] O, en vez de palabras, más bien números en este caso.



Elaboración:
Triana, F.
(2019)

En la diagonal se observa la distribución de cada una de las variables, mientras que en los diagramas de dispersión se encuentra la relación entre pares de estas; así, se puede observar que, por ejemplo, el ingreso se concentra en valores inferiores a 5000 y existe una relación negativa entre la tasa de alfabetización de las mujeres y la mortalidad infantil.

En estos momentos ya se cuenta con información básica sobre las variables del dataset y hemos adelantado una breve inspección visual de las relaciones que se presentan entre estas; ahora, se puede hacer uso de este conocimiento para obtener expresiones más concretas de dichas relaciones, examinándolas desde un punto de vista un tanto más técnico: en el siguiente capítulo se abordará el tema de la regresión lineal por el método de Mínimos Cuadrados Ordinarios.

Capítulo 3

Regresión lineal por Mínimos Cuadrados Ordinarios

La regresión lineal es el tema introductorio que generalmente se aborda en los cursos de Econometría de pregrado en Economía; de acuerdo a Greene (2003), "el modelo de regresión lineal es la herramienta más útil del kit de herramientas de los econometristas" (p. 7). Este tema suele ser el primer contacto que tiene el estudiante con materia de contenido propiamente econométrico y tiende a ser abordada con cierto detalle, de modo que éste tenga claridad suficiente sobre la misma. Es común que en los cursos se presente la regresión lineal, junto a los fundamentos teóricos aplicables a la misma, con el apoyo de un texto guía de un autor reconocido y que se haga énfasis apreciable en los supuestos asociados, de modo que el estudiante adquiera un nivel de conocimiento adecuado sobre ésta.

En esta obra, considerando su carácter práctico y teniendo en cuenta la advertencia realizada en el prefacio de que los fundamentos teóricos se abordarán con un nivel de profundidad mínimo, se estará limitado a exponer la idea general de la regresión lineal en términos muy superficiales; esto, con el propósito de enfocarse en el objetivo planteado y de no hacer sentir incómodo al lector proveniente de otras ramas de estudio o con una formación econométrica no tan sólida, al no ahondar en las bases técnicas y las formalidades matemáticas sobre las que se cimienta la regresión lineal y sus métodos de estimación.

> El análisis de regresión trata del estudio de la dependencia de una variable (*variable dependiente*) respecto de una o más variables (*variables explicativas*) con el objetivo de estimar o predecir la media o valor promedio poblacional de la primera en términos de los valores conocidos o fijos (en muestras repetidas) de las segundas. (Gujarati y Porter, 2010, p.15)

En términos simples, el análisis de regresión estudia la relación que existe entre la variable dependiente y la(s) variable(s) explicativa(s). Para el caso de la regresión lineal, tal relación puede ser expresada como una función lineal en los parámetros, pero ¿de qué parámetros se está hablando?

Para resolver el anterior interrogante, lo primero que se hará será considerar una regresión bivariada, en la cual la variable dependiente (que se identifica con y) puede ser expresada como una función[1] (con linealidad en los parámetros) de la variable explicativa (que se identifica con x). Concretamente:

$$y_i = \beta_0 + \beta_1 x_i + u_i$$

Esta expresión corresponde a la **Función de Regresión Poblacional** (FRP) (Gujarati y Porter, 2010) y los parámetros a los que se ha hecho referencia son simplemente los β; como podemos ver, estos β no se ven afectados por algún tipo de transformación que modifique su relación *lineal* con la variable dependiente: a esto

[1] No se trata exactamente de una función, pues, como lo señalan Fahrmeier, Kneib, y Lang (2007), una característica fundamental de los análisis de regresión es que la relación entre la variable dependiente y las variables explicativas no corresponde a una función en sentido estricto, dado que se ve afectada por perturbaciones aleatorias (p. 19).

es que se hace referencia con *linealidad* en los parámetros, la cual, no necesariamente debe aplicar a las variables explicativas (Greene, 2003; Gujarati y Porter, 2010). En cuanto a u, ésta corresponde al término de error, el cual recoge el efecto de las variables no incluidas explícitamente en el modelo y que afectan y.

Ahora, en la práctica se desconoce la Función de Regresión Poblacional, por lo que se recurre a la **Función de Regresión Muestral** (FRM) como una aproximación a esta. Al lector interesado en la cuestión y que desee conocer una explicación al respecto, se sugiere consultar el Capítulo 2, *Análisis de regresión con dos variables: algunas ideas básicas*, de "Gujarati, D.N. y Porter, D.C. (2010). *Econometría*".

La Función de Regresión Muestral es:

$$y_i = \hat{\beta}_0 + \hat{\beta}_1 x_i + \hat{u}_i$$

Esta expresión es muy parecida a la de la Función de Regresión Poblacional, pero presenta una diferencia fundamental: los términos de la Función de Regresión Muestral tienen correspondencia directa con los términos incluidos en la expresión de la Función de Regresión Poblacional, pero no son exactamente los mismos, en cuanto se trata de sus **estimadores**. Así, el estimador del parámetro k se identifica como $\hat{k}$.

Tal y como lo señalan Gujarati y Porter (2010), "el objetivo principal del análisis de regresión es estimar la Función de Regresión Poblacional con base en la Función de Regresión Muestral" (p. 44). En este sentido, lo que se busca es hallar los valores de los $\hat{\beta}$ que sean más cercanos a los verdaderos valores de los β.

Para realizar la estimación, es frecuente el uso de dos métodos: **Mínimos Cuadrados Ordinarios** y **Máxima Verosimilitud**. El método de Mínimos Cuadrados Ordinarios, o **MCO**, es el más común y tiende a ser abordado como tema fundamental en los cursos de econometría a nivel introductorio.

Resaltando, una vez más, que el nivel de profundidad con el que se abordan las bases teóricas en esta obra es ínfimo, tan solo se presentará (con el perdón de los especialistas, al no tratar con el detalle justo la cuestión) la idea general del método de MCO: el objetivo es, básicamente, minimizar la suma de los residuos (termino $\hat{u}$ en la Función de Regresión Muestral) cuadrados.

Para el caso de una regresión bivariada, lo que se busca con el método MCO es el menor valor posible de $\sum_{i=1}^{n} \hat{u}_i^2$, teniendo en cuenta que:

$$\sum_{i=1}^{n} \hat{u}_i^2 = \sum_{i=1}^{n} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2$$

Así, los estimadores $\hat{\beta}_0$ y $\hat{\beta}_1$ con los que se halla la menor suma de los residuos al cuadrado son los estimadores MCO, los cuales, con el cumplimiento de un conjunto de supuestos específicos, evidencian propiedades estadísticas muy atractivas.

Al lector interesado en conocer el proceso por medio del cual se obtienen los valores de los $\hat{\beta}$ y familiarizarse con las bases matemáticas correspondientes, se le invita a consultar el Capítulo 3, *Modelo de regresión con dos variables: problema de estimación*, de "Gujarati, D.N. y Porter, D.C. (2010). *Econometría*" y el Capítulo 2, *El modelo de regresión simple*, de "Wooldridge, J.M. (2010). *Introducción a la econometría. Un enfoque moderno*".

Ya se ha presentado, muy brevemente, una idea general de la regresión lineal y el método de MCO. Ahora, es momento de iniciar de forma efectiva con la labor práctica; a saber, análisis econométrico con el uso de Python.

Lo primero que se hará es llevar a cabo una estimación, por Mínimos Cuadrados Ordinarios, de un modelo de regresión lineal simple. El modelo de regresión lineal simple no es más que una regresión lineal bivariada, es decir, en la que solo intervienen dos variables explícitas: la variable dependiente como función lineal en los parámetros de la variable explicativa. Esto es, simplemente, un modelo al que corresponde una FRM de la forma $y_i = \hat{\beta}_0 + \hat{\beta}_1 x_i + \hat{u}_i$, la cual debe resultar familiar al lector, pues es la que se ha tratado previamente.

Regresión lineal simple

Para llevar a cabo la regresión lineal simple, utilizaremos el mismo dataset del capítulo previo y requeriremos de las mismas herramientas empleadas en éste, por lo que procederemos a importar las respectivas librerías y los datos:

```
In [1]: import numpy as np
        import pandas as pd
        import statsmodels.api as sm
        import matplotlib.pyplot as plt
        import seaborn as sns
        %matplotlib inline
        plt.style.use("seaborn-white")

In [2]: data = pd.read_csv("GujaratiPorter71.txt",
                           sep = " ", delimiter="\t")
```

En el capítulo previo se llevó a cabo un breve reconocimiento de los datos. Ahora, se procederá a la realización de una Regresión Lineal Simple por el método de Mínimos Cuadrados Ordinarios, teniendo en consideración lo siguiente:

- El `DataFrame` contiene información de las siguientes variables:

  'CM'. Esta variable hace referencia a *Child Mortality* y corresponde a la mortalidad infantil. Se trata del número de fallecimientos, en un año, de niños con una edad inferior a 5 años por cada 1000 nacidos vivos.

  'FLR'. Esta variable hace referencia a *Female Literacy Rate* y corresponde a la tasa de alfabetización de las mujeres.

  'PGNP' hace referencia a *Per cápita Gross National Product* y corresponde al PIB per cápita en 1980.

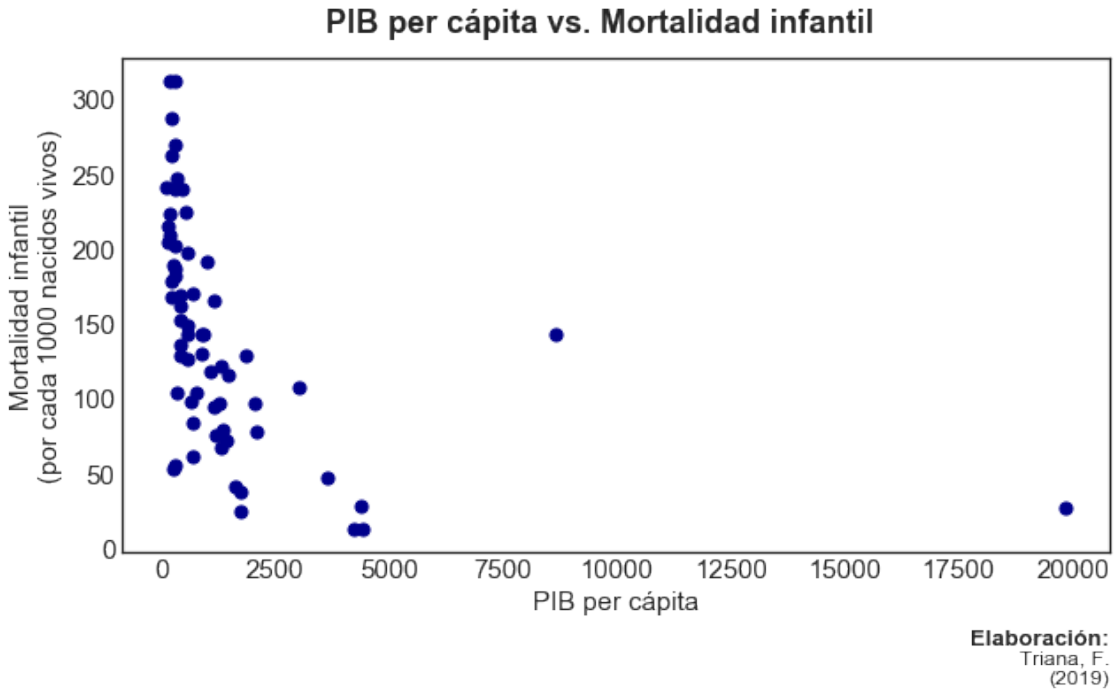
  'TFR' hace referencia a *Total Fertility Rate* y corresponde a la tasa de fecundidad total.

- En la regresión lineal simple que se adelantará, la variable dependiente será 'CM' (mortalidad infantil) y la variable explicativa será 'PGNP' (PIB per cápita).

Se procederá a examinar visualmente la relación que existe entre las variables de interés. Para esto, se construirá un diagrama de dispersión (*scatterplot*) con el uso del método `.scatter()`, aplicado a un `Axes` de *Matplotlib*. El código a emplear es el siguiente:

```
In [3]: fig, ax = plt.subplots(figsize = (10, 5))
        fig.suptitle("PIB per cápita vs. Mortalidad infantil", fontsize = 18,
                     fontweight = "bold")
        ax.scatter(x = data["PGNP"], y = data["CM"], s = 50, color = "darkblue")
        ax.set_xlabel("PIB per cápita", fontsize = 15)
        ax.set_ylabel("Mortalidad infantil\n(por cada 1000 nacidos vivos)",
                      fontsize = 15)
        plt.subplots_adjust(top=0.9)
        plt.tick_params(labelsize = 15)
        fig.text(.9,-.02,
                 "Elaboración:",
```

```
            fontsize = 13, fontweight = "bold",
           ha = "right")
fig.text(.9,-.08,
         "Triana, F.\n(2019)",
         fontsize = 12, ha = "right")
plt.show()
```



Aunque es un bloque de código algo extenso, el lector debe centrar su atención tan solo en la primera, especialmente la tercera, y la última línea de código de la celda, pues son las que contienen la "esencia" del gráfico; las demás, solamente hacen referencia a detalles de los cuales se puede prescindir. No se hará énfasis en la explicación de este código, en cuanto no es el asunto principal que busca tratarse y la mayoría de líneas "se explican por sí mismas". Al lector que, sin embargo, esté interesado en este asunto, se recomienda el estudio detallado de *Matplotlib*.

A partir del gráfico se puede observar que la relación entre las variables es aparentemente negativa. Sin embargo, esto tan solo es una impresión que se genera a partir de la observación y puede ser errada, razón por la cual se examinará cuantitativamente dicha relación buscando comprobar si la conjetura es correcta; para tal propósito se llevará a cabo una regresión lineal por el método de MCO.

Para crear el modelo, se emplea la función `OLS(...)` de *Statsmodels*. Los argumentos que se incluyen corresponden, respectivamente, a la variable dependiente y a la variable explicativa. El resultado será asignado a un objeto que se denominará `MiModeloSimple`:

```
In [4]: MiModeloSimple = sm.OLS(data["CM"], data["PGNP"])
```

El modelo ya ha sido creado, es decir, ya se ha definido su estructura; sin embargo, aún no se ha llevado a cabo ninguna estimación a partir de éste. Para efectuar la estimación se recurrirá al método `.fit()` y se almacenarán los resultados generados en un objeto al que se denominará `ResultadosSimple`:

```
In [5]: ResultadosSimple = MiModeloSimple.fit()
```

Al ejecutar esta línea de código no se obtiene algún resultado visible. Esto se debe a que lo que se ha conseguido con la ejecución es guardar los resultados generados en un objeto, sin indicar la realización de alguna acción en particular con dicho objeto. Ahora, si se desea visualizar los resultados del modelo, puede simplemente usarse la función integrada `print(...)`, empleando como argumento la aplicación del método `.summary()` sobre el objeto que almacena los resultados de dicho modelo. Así:

```
In [6]: print(ResultadosSimple.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                     CM   R-squared:                       0.056
Model:                            OLS   Adj. R-squared:                  0.041
Method:                 Least Squares   F-statistic:                     3.710
Date:                xxx, xx xxx xxxx   Prob (F-statistic):             0.0586
Time:                        xx:xx:xx   Log-Likelihood:                -413.92
No. Observations:                  64   AIC:                             829.8
Df Residuals:                      63   BIC:                             832.0
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
PGNP           0.0124      0.006      1.926      0.059      -0.000       0.025
==============================================================================
Omnibus:                        7.668   Durbin-Watson:                   0.755
Prob(Omnibus):                  0.022   Jarque-Bera (JB):                7.941
Skew:                          -0.561   Prob(JB):                       0.0189
Kurtosis:                       4.312   Cond. No.                         1.00
==============================================================================

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

Lo que se obtiene con la ejecución del código es un resumen de la instancia de resultados en el que se presenta la información más relevante acerca de estos: en la parte superior se incluye el método de estimación empleado, el número de observaciones, los grados de libertad (de los residuos y del modelo) y el R^2 (estándar y ajustado), entre otros datos relevantes. En la sección del medio se encuentran los coeficientes, junto a sus respectivos errores estándar, estadísticos t e intervalos de confianza. En la última sección están las advertencias, las cuales resultan, frecuentemente, muy útiles y pueden ayudar a identificar potenciales problemas.

Se puede observar que el coeficiente de la variable explicativa es positivo, lo que contradice la conclusión visual de que la relación, aparentemente, era negativa; algo que resulta, además, poco lógico, pues se esperaría que los países con ingreso per cápita más alto tuvieran una tasa de mortalidad infantil menor. Ante esta situación extraña vale la pena preguntarse sobre su causa: ¿cómo puede explicarse tal inconsistencia? La respuesta es extremadamente simple: el lector debe notar que para este modelo solo se ha calculado el coeficiente correspondiente a la variable 'PGNP' y no se ha considerado la existencia del parámetro de intercepto, es decir, se ha llevado a cabo una regresión *a través del origen*.

En vez de considerar $y_i = \beta_0 + \beta_1 x_i + u_i$, se ha asumido que la FRP es de la forma $y_i = \beta_1 x_i + u_i$.

La función `OLS(...)` de *Statsmodels* no asume por defecto que el modelo a estimar incluye una constante como variable explicativa y, por tanto, no realiza la estimación del coeficiente correspondiente al parámetro de intercepto. Para conseguir la estimación de dicho parámetro es necesario especificar que una constante debe añadirse al conjunto de variables explicativas del modelo; esto se logra, por ejemplo, empleando la función `add_constant(...)` de *Statsmodels*, tomando como argumento el objeto correspondiente a las variables explicativas incluidas. Así:

```
In [7]: MiModeloSimple2 = sm.OLS(data["CM"], sm.add_constant(data["PGNP"]))
```

Para realizar la estimación y visualizar los resultados correspondientes, basta con repetir la misma estructura de los códigos empleados previamente (en el caso del modelo sin término del intercepto). Así:

```
In [8]: ResultadosSimple2 = MiModeloSimple2.fit()
        print(ResultadosSimple2.summary())
```

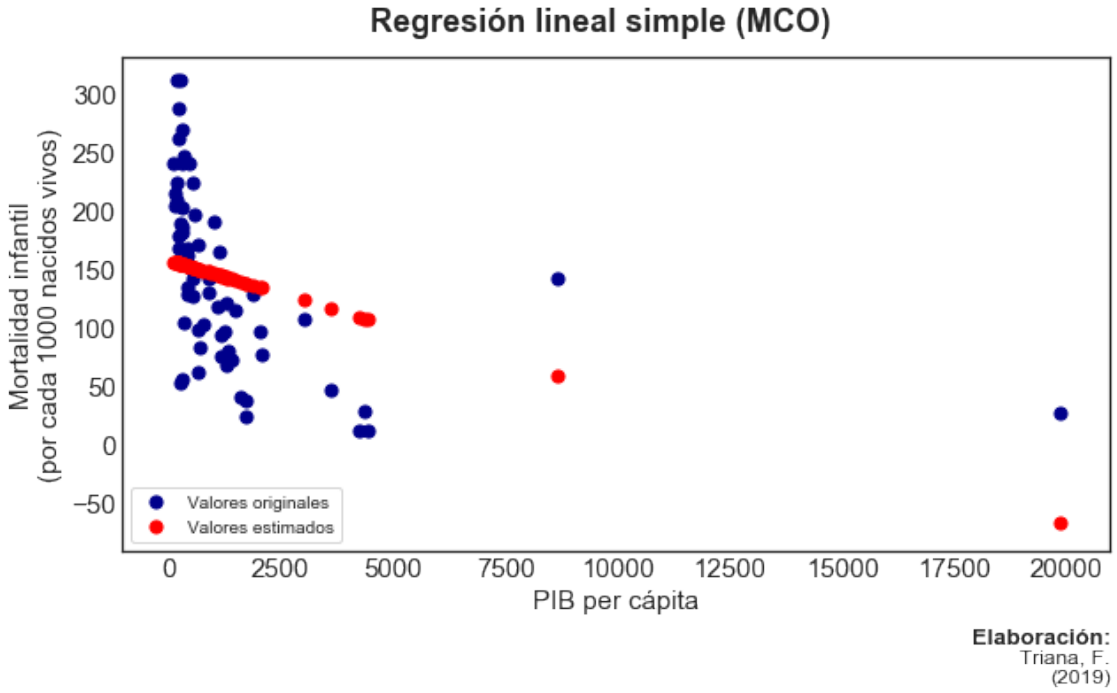
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                     CM   R-squared:                       0.166
Model:                            OLS   Adj. R-squared:                  0.153
Method:                 Least Squares   F-statistic:                     12.36
Date:                xxx, xx xxx xxxx   Prob (F-statistic):           0.000826
Time:                        xx:xx:xx   Log-Likelihood:                -361.64
No. Observations:                  64   AIC:                             727.3
Df Residuals:                      62   BIC:                             731.6
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        157.4244      9.846     15.989      0.000     137.743     177.105
PGNP          -0.0114      0.003     -3.516      0.001      -0.018      -0.005
==============================================================================
Omnibus:                        3.321   Durbin-Watson:                   1.931
Prob(Omnibus):                  0.190   Jarque-Bera (JB):                2.545
Skew:                           0.345   Prob(JB):                        0.280
Kurtosis:                       2.309   Cond. No.                     3.43e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 3.43e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Ahora, se puede observar que el coeficiente correspondiente a la variable explicativa es negativo, tal y como se había presumido en el examen visual, y que ésta es estadísticamente significativa. Lo que se hará enseguida es graficar los valores originales y los valores estimados por el modelo, de modo que puedan contrastarse y

sea posible un examen visual de la bondad de ajuste de éste. Para esto, nuevamente se hará uso del método `.scatter(...)` aplicado sobre un `Axes` de *Matplotlib*.

```
In [9]: fig, ax = plt.subplots(figsize = (10,5))
        fig.suptitle("Regresión lineal simple (MCO)", fontsize = 18,
                     fontweight = "bold")
        ax.scatter(data["PGNP"], data["CM"], s = 50,
                   label = "Valores originales",
                   color = "darkblue")
        ax.scatter(data["PGNP"], ResultadosSimple2.predict(), s = 50,
                   label = "Valores estimados",
                   color = "red")
        ax.set_xlabel("PIB per cápita", fontsize = 15)
        ax.set_ylabel("Mortalidad infantil\n(por cada 1000 nacidos vivos)",
                   fontsize = 15)
        ax.legend(frameon = True, loc = "lower left")
        plt.subplots_adjust(top=0.9)
        plt.tick_params(labelsize = 15)
        fig.text(.9,-.02,
                 "Elaboración:",
                  fontsize = 13, fontweight = "bold",
                ha = "right")
        fig.text(.9,-.08,
                 "Triana, F.\n(2019)",
                 fontsize = 12, ha = "right")
        plt.show()
```



Se observa que la bondad de ajuste del modelo no es la mejor. ¿A qué se debe este resultado desalentador? Si el lector examina el gráfico anterior podrá notar que la parte correspondiente a los datos originales presenta un muy ligero parecido con el gráfico de una función de la forma $y = f(x) = \frac{a}{x}$, con $a > 0$, $x > 0$ y representación del valor de x en el eje de las abscisas y del valor de y en el eje de las ordenadas. Este parecido sugiere que la forma funcional $y_i = \beta_0 + \beta_1 x_i + u_i$ tal vez no sea la más adecuada para modelar una relación con estos datos, en cuanto y puede asemejarse más a una función de x de la forma $f(x) = \frac{a}{x}$ $(a \neq 0)$ que a una de la forma $f(x) = ax$ $(a \neq 0)$. Teniendo en cuenta esto, tal vez la forma $y_i = \beta_0 + \beta_1 \left(\frac{1}{x}\right) + u_i$ genere mejores resultados.

- **Nota:** Un modelo con forma funcional $y_i = \beta_0 + \beta_1 \left(\frac{1}{x}\right) + u_i$ es denominado modelo recíproco. (Se invita al lector a consultar el Capítulo 6, *Extensiones del modelo de regresión lineal con dos variables*, de Gujarati y Porter (2010), para obtener mayor información al respecto).

Para trabajar con un modelo recíproco se debe tener en cuenta que la estimación no se hará a partir de los valores de x sino de los valores de $\frac{1}{x}$. Evidentemente, esto implica que el modelo es **no** *lineal* en la variable x, pero, ¿esto significa que lo que se llevará a cabo es una regresión no lineal? La respuesta es no. Aunque el modelo ya no es lineal en la variable x, el lector debe recordar que la linealidad considerada hace referencia a los parámetros y **no** a las variables, por lo cual, a pesar de la no linealidad de la variable, sigue tratándose de un modelo de regresión lineal, en cuanto la linealidad en los parámetros no se ha visto afectada.

Se ha creado el modelo tomando 'PGNP' como variable explicativa, ahora, ésta no será incluida, sino que se recurrirá a su recíproco: '1/PGNP'. Los valores de 'PGNP' están incluidos en el dataset que se ha importado, sin embargo, los valores de '1/PGNP' son desconocidos. ¿Cómo se puede emplear una variable de cuyos valores no se tiene información?

Aunque se desconocen los valores de '1/PGNP', aún se tiene la posibilidad de obtenerlos en tanto los valores de 'PGNP' son conocidos y tan solo se debe realizar una operación matemática sencilla. El cálculo de estos valores, a pesar de ser simple, puede resultar tedioso dada la cantidad de los mismos y puede llevar a cometer errores si se ejecuta de forma "manual"; por fortuna, *pandas* resulta muy útil en este contexto, pues evita la realización de tal labor "manual" para el cálculo de cada uno de los valores, al permitir aplicar operaciones matemáticas básicas de Python que permiten obtener el resultado fácilmente.

Se creará la variable correspondiente al recíproco de la variable original, ya que ésta no hace parte del dataset. Para crear dicha variable, simplemente se realiza una asignación a una `Series` de *pandas*, señalando el nombre del `DataFrame` correspondiente y, a continuación, especificando el nombre que quiere darse a la variable nueva (escrito entre comillas) dentro de corchetes [ ]. Así:

```
In [10]: data["1/PGNP"] = 1/data["PGNP"]
```

Se verifica la correcta ejecución del proceso, recurriendo al método `.head()`:

```
In [11]: data.head()
```

```
Out[11]:
          CM  FLR  PGNP   TFR    1/PGNP
0  128   37  1870  6.66  0.000535
1  204   22   130  6.15  0.007692
2  202   16   310  7.00  0.003226
3  197   65   570  6.25  0.001754
4   96   76  2050  3.81  0.000488
```

Como el lector puede observar, la última columna lleva el nombre que se ha asignado a la nueva variable ("1/PGNP") cuyos valores son los resultados de tomar, fila por fila, el número uno y dividirlo por el valor correspondiente de la columna 'PGNP'. Para comprobar la exactitud de la operación realizada, siéntase en la libertad de tomar la información de una fila cualquiera y ejecutar el cálculo correspondiente.

Ahora, estimaremos el modelo $y_i = \beta_0 + \beta_1 \left(\frac{1}{x_i}\right) + u_i$, teniendo en cuenta que $\frac{1}{x}$ corresponde a la variable recién creada '1/PGNP', y observaremos si se genera una mejora en el ajuste respecto al obtenido con la estimación de $y_i = \beta_0 + \beta_1 x_i + u_i$. Para la construcción y estimación del modelo y visualización de los resultados, emplearemos, nuevamente, la función `OLS(...)` de *Statsmodels* y los métodos `.fit()` y `.summary()`:

```
In [12]: MiModeloSimple3 = sm.OLS(data["CM"], sm.add_constant(data[["1/PGNP"]]))
         ResultadosSimple3 = MiModeloSimple3.fit()
         print(ResultadosSimple3.summary())
```

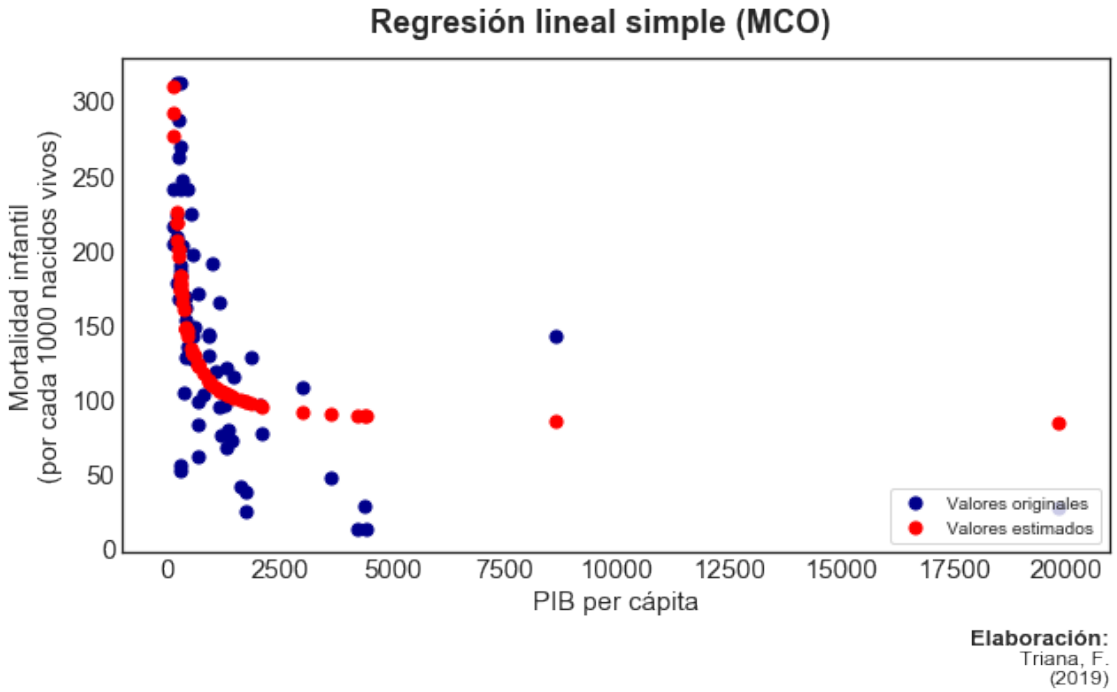
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                     CM   R-squared:                       0.459
Model:                            OLS   Adj. R-squared:                  0.450
Method:                 Least Squares   F-statistic:                     52.61
Date:                xxx, xx xxx xxxx   Prob (F-statistic):           7.82e-10
Time:                        xx:xx:xx   Log-Likelihood:                -347.79
No. Observations:                  64   AIC:                             699.6
Df Residuals:                      62   BIC:                             703.9
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         81.7944     10.832      7.551      0.000      60.141      103.447
1/PGNP     2.727e+04   3759.999      7.254      0.000    1.98e+04    3.48e+04
==============================================================================
Omnibus:                        0.147   Durbin-Watson:                   1.959
Prob(Omnibus):                  0.929   Jarque-Bera (JB):                0.334
Skew:                           0.065   Prob(JB):                        0.846
Kurtosis:                       2.671   Cond. No.                         534.
==============================================================================

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

Ahora, procedemos a graficar los valores originales y los valores estimados, usando el método `.scatter(...)` de un `Axes` de *Matplotlib*, de modo que podamos observar si efectivamente se genera una mejora en el ajuste:

```
In [13]: fig, ax = plt.subplots(figsize = (10,5))
         fig.suptitle("Regresión lineal simple (MCO)", fontsize = 18,
                      fontweight = "bold")
         ax.scatter(data["PGNP"], data["CM"], s = 50,
                    label = "Valores originales",
                    color = "darkblue")
         ax.scatter(data["PGNP"], ResultadosSimple3.predict(), s = 50,
```

```
            label = "Valores estimados",
            color = "red")
ax.set_xlabel("PIB per cápita", fontsize = 15)
ax.set_ylabel("Mortalidad infantil\n(por cada 1000 nacidos vivos)",
            fontsize = 15)
ax.legend(frameon = True, loc = "lower right")
plt.subplots_adjust(top=0.9)
plt.tick_params(labelsize = 15)
fig.text(.9,-.02,
         "Elaboración:",
         fontsize = 13, fontweight = "bold",
         ha = "right")
fig.text(.9,-.08,
         "Triana, F.\n(2019)",
         fontsize = 12, ha = "right")
plt.show()
```



Como podemos observar, se presenta una mejora clara, en cuanto el ajuste parece ser más adecuado; la mejora es visualmente evidente, sin embargo, también se ha presentado un incremento apreciable del coeficiente de determinación, o R^2, el cual ha pasado de 0.166 a 0.459, como el lector puede comprobar en las tablas de resumen de cada regresión. Así, para este caso particular, la forma funcional del modelo recíproco parece ser más apropiada que la forma funcional tradicional.

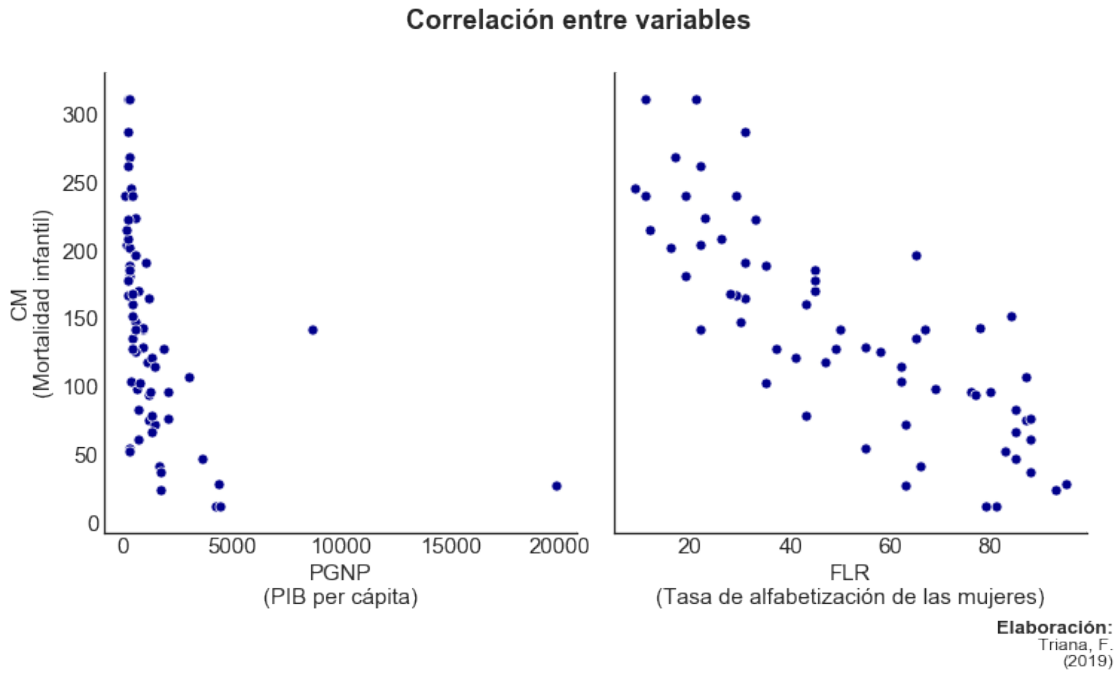
Regresión lineal múltiple

Ya hemos realizado una regresión lineal usando una única variable explicativa; ahora, emplearemos varias explicativas. En un modelo de regresión lineal múltiple se tiene que la FRP es de la forma $y_i = \beta_0 + \beta_1 x_{1i} + ... + \beta_k x_{ki} + u_i$, de modo que se incluyen k variables explicativas (sin contar la constante de β_0) y se deben estimar $k + 1$ parámetros (los de las variables y el término del intercepto).

En el modelo de regresión lineal múltiple que emplearemos, la variable dependiente será 'CM' (mortalidad infantil) y las variables explicativas serán 'PGNP' (PIB per cápita) y 'FLR' (tasa de alfabetización de las mujeres). Lo primero que haremos es examinar gráficamente la relación que existe entre las variables explicativas y la variable dependiente; esta vez, recurriremos a la función `pairplot(...)` de *Seaborn*, en cuanto esta nos permite graficar relaciones de pares en conjuntos de datos.

La función `pairplot(...)` de *Seaborn* incluye múltiples argumentos, sin embargo, para nuestros propósitos, además de (obviamente) el parámetro obligatorio *data*, los argumentos importantes son *x_vars* y *y_vars*; el primero permite especificar las variables que se quiere graficar en el eje de las abscisas y el segundo las que se quiere graficar en el eje de las ordenadas. Para nuestro caso particular, `y_vars = 'CM'` y `x_vars =['PGNP', 'FLR']` (el lector debe notar que, dado que se trata de varias variables, deben incluirse dentro de una **lista Python** [...]).

```
In [14]: g = sns.pairplot(data, x_vars = ["PGNP", "FLR"],
                          y_vars = "CM", height = 5,
                          plot_kws = {"color": "darkblue", "s": 50})
         g.fig.suptitle("Correlación entre variables", fontsize = 18,
                        fontweight = "bold", y = 1.08)
         g.axes.flat[0].set_xlabel("PGNP\n(PIB per cápita)", fontsize = 15)
         g.axes.flat[0].set_ylabel("CM\n(Mortalidad infantil)", fontsize = 15)
         g.axes.flat[1].set_xlabel("FLR\n(Tasa de alfabetización de las mujeres)",
                                   fontsize = 15)
         for ax in g.axes.flat:
             ax.tick_params(labelsize = 15)
         g.fig.text(1,-.09,
                    "Elaboración:",
                     fontsize = 13, fontweight = "bold",
                     ha = "right")
         g.fig.text(1,-.15,
                     "Triana, F.\n(2019)",
                     fontsize = 12, ha = "right")
         plt.show()
```



A partir de nuestro gráfico podemos observar que la relación que existe entre las variables explicativas y la variable dependiente es, aparentemente, negativa para ambas. Ahora, es momento de cuantificar tales relaciones. Debemos asignar las variables a objetos, de modo que resulte menos tediosa la construcción del modelo y contemos con un código más "limpio". Para seleccionar variables de un `DataFrame` se deben emplear los corchetes '[ ]', teniendo en consideración lo siguiente:

- Cuando se trata de una sola variable, basta con escribir su nombre (entre comillas) dentro de los corchetes.
- Cuando se trata de varias variables, se debe escribir sus nombres (entre comillas, separados por comas) dentro de una **lista Python** [...] y emplear tal lista dentro de los corchetes.

La variable dependiente será asignada al objeto `Y` y las variables explicativas serán asignadas al objeto `X`, así:

```
In [15]: Y = data["CM"]
         X = data[["PGNP", "FLR"]]
```

Como ya hemos asignado las variables a los objetos correspondientes, ahora podemos emplear tales objetos como argumentos de la función `OLS(...)` de *Statsmodels* para la construcción del modelo, recordando que sus argumentos son, respectivamente, la variable dependiente y las variables explicativas:

```
In [16]: MiModelo = sm.OLS(Y, X)
```

Llevamos a cabo la estimación, con el método `.fit()`, y guardamos los resultados generados en un objeto que se denominará `Resultados`:

```
In [17]: Resultados = MiModelo.fit()
```

Finalmente, visualizamos los resultados empleando la función integrada (nativa) `print(...)`, usando como argumento la aplicación del método `.summary()`:

```
In [18]: print(Resultados.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                     CM   R-squared:                       0.387
Model:                            OLS   Adj. R-squared:                  0.368
Method:                 Least Squares   F-statistic:                     19.61
Date:                xxx, xx xxx xxxx   Prob (F-statistic):           2.52e-07
Time:                        xx:xx:xx   Log-Likelihood:                -400.07
No. Observations:                  64   AIC:                             804.1
Df Residuals:                      62   BIC:                             808.5
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
PGNP          -0.0060      0.006     -0.974      0.334      -0.018       0.006
FLR            1.8838      0.325      5.796      0.000       1.234       2.534
==============================================================================
Omnibus:                        7.742   Durbin-Watson:                   1.601
Prob(Omnibus):                  0.021   Jarque-Bera (JB):                2.919
Skew:                           0.161   Prob(JB):                        0.232
Kurtosis:                       2.005   Cond. No.                         62.1
==============================================================================

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

Se puede evidenciar que no se ha estimado el término del intercepto, es decir, se ha efectuado una regresión a través del origen. Esto, como se mencionó con anterioridad, se debe a que la función `OLS(...)` de *Statsmodels* no asume por defecto que se debe incluir una constante en el conjunto de variables explicativas. Para que la estimación se realice con un intercepto es necesario especificar que una constante debe ser empleada como regresora; para tal propósito, se puede emplear la función `add_constant(...)` de *Statsmodels*, tomando como argumento el objeto que contiene las demás variables explicativas.

```
In [19]: MiModelo2 = sm.OLS(Y, sm.add_constant(X))
```

La estimación y la visualización de resultados se llevan a cabo empleando los métodos que se han usado con anterioridad (`.fit()` y `.summary()`):

```
In [20]: Resultados2 = MiModelo2.fit()
         print(Resultados2.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                     CM   R-squared:                       0.708
Model:                            OLS   Adj. R-squared:                  0.698
Method:                 Least Squares   F-statistic:                     73.83
Date:                xxx, xx xxx xxxx   Prob (F-statistic):           5.12e-17
Time:                        xx:xx:xx   Log-Likelihood:                -328.10
No. Observations:                  64   AIC:                             662.2
Df Residuals:                      61   BIC:                             668.7
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        263.6416     11.593     22.741      0.000     240.460     286.824
PGNP          -0.0056      0.002     -2.819      0.006      -0.010      -0.002
FLR           -2.2316      0.210    -10.629      0.000      -2.651      -1.812
==============================================================================
Omnibus:                        0.732   Durbin-Watson:                   2.186
Prob(Omnibus):                  0.693   Jarque-Bera (JB):                0.559
Skew:                           0.228   Prob(JB):                        0.756
Kurtosis:                       2.949   Cond. No.                     6.77e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 6.77e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

El lector puede observar que los resultados que hemos alcanzado han sido obtenidos con el empleo de **métodos**, los cuales son, esencialmente, funciones asociadas a un objeto específico, aplicables directamente sobre éste.

`.fit()` y `.summary()` son **métodos** aplicables a objetos generados por la función `OLS(...)`, sin embargo, estos no son los únicos métodos propios de tal tipo de objetos. Para conocer los métodos y atributos de un objeto en Python, se emplea la función integrada (nativa) `dir(...)`, la cual toma como argumento un objeto de Python y devuelve sus atributos y métodos dentro de una lista Python '[...]'.

Para conocer los métodos y atributos del objeto `Resultados2`, el código a emplear es el siguiente:

```
In [21]: dir(Resultados2)

Out[21]: ['HC0_se',
          'HC1_se',
          'HC2_se',
          'HC3_se',
          '_HCCM',
          '__class__',
```

```
'__delattr__',
'__dict__',
'__dir__',
'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattribute__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__le__',
'__lt__',
'__module__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'_cache',
'_data_attr',
'_get_robustcov_results',
'_is_nested',
'_wexog_singular_values',
'aic',
'bic',
'bse',
'centered_tss',
'compare_f_test',
'compare_lm_test',
'compare_lr_test',
'condition_number',
'conf_int',
'conf_int_el',
'cov_HC0',
'cov_HC1',
'cov_HC2',
'cov_HC3',
'cov_kwds',
'cov_params',
'cov_type',
'df_model',
```

```
'df_resid',
'diagn',
'eigenvals',
'el_test',
'ess',
'f_pvalue',
'f_test',
'fittedvalues',
'fvalue',
'get_influence',
'get_prediction',
'get_robustcov_results',
'initialize',
'k_constant',
'llf',
'load',
'model',
'mse_model',
'mse_resid',
'mse_total',
'nobs',
'normalized_cov_params',
'outlier_test',
'params',
'predict',
'pvalues',
'remove_data',
'resid',
'resid_pearson',
'rsquared',
'rsquared_adj',
'save',
'scale',
'ssr',
'summary',
'summary2',
't_test',
't_test_pairwise',
'tvalues',
'uncentered_tss',
'use_t',
'wald_test',
'wald_test_terms',
'wresid']
```

Cada uno de los objetos de esta lista Python [...] es un método o atributo del modelo que hemos construido. Así, si deseamos conocer el coeficiente de determinación múltiple o R^2, el cual se identifica con el atributo `.rsquared`, podemos emplear el siguiente código:

```
In [22]: print("El R2 del modelo es:", Resultados2.rsquared)

El R2 del modelo es: 0.7076654981900712
```

Si estamos interesados en los coeficientes estimados, podemos recurrir al atributo `.params`, así:

```
In [23]: print("Los coeficientes del modelo son:\n", Resultados2.params)

Los coeficientes del modelo son:
 const    263.641586
PGNP      -0.005647
FLR       -2.231586
dtype: float64
```

Asimismo, los métodos permiten ejecutar acciones específicas sobre el objeto. A modo de ejemplo, el método `.get_robustcov_results()` permite obtener una nueva instancia de resultados asumiendo por defecto covarianza robusta.

```
In [24]: Resultados2_robustos = Resultados2.get_robustcov_results()
         print(Resultados2_robustos.summary())

                            OLS Regression Results
==============================================================================
Dep. Variable:                     CM   R-squared:                       0.708
Model:                            OLS   Adj. R-squared:                  0.698
Method:                 Least Squares   F-statistic:                     93.06
Date:                xxx, xx xxx xxxx   Prob (F-statistic):           2.94e-19
Time:                        17:34:52   Log-Likelihood:                -328.10
No. Observations:                  64   AIC:                             662.2
Df Residuals:                      61   BIC:                             668.7
Df Model:                           2
Covariance Type:                  HC1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        263.6416     11.983     22.002      0.000     239.681     287.602
PGNP          -0.0056      0.001     -4.788      0.000      -0.008      -0.003
FLR           -2.2316      0.194    -11.516      0.000      -2.619      -1.844
==============================================================================
Omnibus:                        0.732   Durbin-Watson:                   2.186
Prob(Omnibus):                  0.693   Jarque-Bera (JB):                0.559
Skew:                           0.228   Prob(JB):                        0.756
Kurtosis:                       2.949   Cond. No.                     6.77e+03
==============================================================================

Warnings:
[1] Standard Errors are heteroscedasticity robust (HC1)
```

```
[2] The condition number is large, 6.77e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Se invita al lector a explorar los demás métodos y atributos disponibles, de modo que pueda conocer toda la información adicional con la que puede profundizar su análisis.

En este punto se cierra el tercer capítulo de esta obra; el lector ya ha tenido un acercamiento a la regresión lineal simple y múltiple en Python con estimación por Mínimos Cuadrados Ordinarios; asimismo, ha observado cómo llevar a cabo un análisis exploratorio mínimo y cómo elaborar gráficos básicos. Ahora, tal y como se mencionó en este capítulo, dados ciertos supuestos, los estimadores de MCO exhiben propiedades estadísticas atractivas; tales supuestos corresponden al tema abordado en el siguiente capítulo.

Capítulo 4

Verificación de supuestos

Los estimadores de MCO presentan una serie de propiedades estadísticas atractivas dados unos supuestos; tal afirmación se sustenta en el Teorema de Gauss-Markov, el cual, siguiendo a Gujarati y Porter (2010), puede expresarse como: "Dados los supuestos del modelo clásico de regresión lineal, los estimadores de mínimos cuadrados, dentro de la clase de estimadores lineales insesgados, tienen varianza mínima, es decir, son MELI" (p. 72).

MELI (BLUE en inglés) hace referencia a "Mejores Estimadores Lineales Insesgados" y es la calificación que adquieren los estimadores de MCO cuando se cumple con los supuestos del modelo clásico de regresión lineal, también conocido como modelo de Gauss o modelo estándar de regresión lineal.

El modelo clásico de regresión lineal plantea, de acuerdo a Gujarati y Porter (2010), los siguientes supuestos:

1. Se trata de un modelo de regresión lineal, es decir, lineal en los parámetros.
2. Valores fijos de x o valores de x independientes del término de error. Esto significa que se requiere covarianza 0 entre u_i y cada variable x_{ji}. Es decir $cov(u_i, x_{ji}) = 0 \; \forall \; i, j$
3. Valor medio de la perturbación u_i igual a 0. $E(u_i) = 0 \; \forall \; i$
4. Homoscedasticidad o varianza constante de u_i. Esto es, $var(u_i) = \sigma^2 \; \forall \; i$
5. No autocorrelación, o correlación serial, entre las perturbaciones. Es decir, $cov(u_i, u_j) = 0 \; \forall \; i \neq j$
6. El número de observaciones (n) debe ser mayor que el de parámetros por estimar. Si se tiene k variables explicativas (sin incluir el intercepto), entonces debe estimarse $k + 1$ parámetros (para una regresión con término del intercepto), por lo tanto, en tal caso, $n > k + 1$.
7. Debe haber variación en los valores de las variables x.
8. No debe haber colinealidad exacta entre las variables x.
9. No hay sesgo de especificación, es decir, el modelo está correctamente especificado.

En este capítulo se verifica el cumplimiento de algunos de estos supuestos con el uso de diversas herramientas. Algo que debe tener en consideración el lector es que algunos de los supuestos anteriormente señalados corresponden específicamente a la estructura (ecuación) del modelo (por ejemplo, el supuesto 1) mientras que otros hacen referencia explícita a las características del término de error (por ejemplo, el supuesto 4 y el supuesto 5). Así, para tener mayor claridad, primero examinaremos los supuestos sobre la estructura del modelo y posteriormente abordaremos los supuestos que se refieren al término de error, por lo que no se seguirá el mismo orden en el que los supuestos han sido enunciados.

Supuestos sobre la estructura del modelo

El modelo clásico de regresión lineal asume el cumplimiento de una serie de supuestos sobre la estructura (ecuación) del modelo. Es posible verificar el cumplimiento de algunos de estos supuestos en Python utilizando funciones propias de algunas librerías especializadas como *Statsmodels*.

Estimaremos un modelo de regresión lineal múltiple y verificaremos el cumplimiento de los supuestos sobre la estructura del mismo. El dataset que utilizaremos será, al igual que en el anterior capítulo, el del Ejemplo 7.1, *Mortalidad infantil en relación con el PIB per cápita y la tasa de alfabetización de las mujeres*, de Gujarati y Porter (2010).

Preparación del entorno

La primera celda en la que el usuario escribirá y ejecutará código tendrá el siguiente contenido:

```
In [1]: import numpy as np
        import pandas as pd
        import statsmodels.api as sm
        import statsmodels.stats.api as sms
        import matplotlib.pyplot as plt
        import seaborn as sns
        %matplotlib inline
        plt.style.use("seaborn-white")
```

El lector ya debe saber que el anterior bloque de código permite cargar las herramientas necesarias para llevar a cabo las acciones requeridas en el análisis econométrico planteado; en caso de que no tenga claridad al respecto, se le pide consultar la sección "Preparación del entorno" del anterior capítulo, en donde se brinda una explicación sobre tal cuestión.

El código de esta celda es exactamente igual al empleado en la preparación del entorno del capítulo anterior, exceptuando por lo siguiente: utilizaremos, adicionalmente, el módulo `stats` de la librería *Statsmodels*, el cual se importará con el alias de `sms`.

La importación de `statsmodels.stats` es fundamental para nuestra labor, pues es donde están contenidas la mayoría de funciones que emplearemos, sin las cuales no sería posible verificar fácilmente el cumplimiento de los supuestos del modelo clásico de regresión lineal. El usuario debe ser cuidadoso de, en caso de reutilizar el bloque de código de preparación del entorno del capítulo anterior, incluir correctamente la línea de importación de `statsmodels.stats`.

Importación de los datos

La importación de los datos se lleva a cabo con funciones de la librería *pandas*. Dependiendo del tipo de archivo en el cual se encuentre almacenada la información del dataset, deberemos recurrir a una función particular y un conjunto de parámetros específicos.

Para nuestro caso concreto, el siguiente código es el que importa el dataset:

```
In [2]: data = pd.read_csv("GujaratiPorter71.txt",
                           delimiter="\t")
```

Al lector que no entienda esta línea de código, se sugiere, nuevamente, dirigirse al Capítulo 2, en donde encontrará una explicación que seguramente aclarará sus dudas, o al Capítulo 1 de Triana y Galindo (2019).

Una vez ejecutada la línea de código de esta celda, el dataset debería haber sido importado correctamente. Para comprobarlo, recurrimos al método `.head()`, aplicado al objeto que contiene el dataset (en este caso le hemos asignado el nombre `data`):

```
In [3]: data.head()
```

```
Out[3]:      CM  FLR  PGNP   TFR
        0  128   37  1870  6.66
        1  204   22   130  6.15
        2  202   16   310  7.00
        3  197   65   570  6.25
        4   96   76  2050  3.81
```

Podemos observar que el proceso de importación ha sido exitoso y la información del dataset ha sido almacenada correctamente en un `DataFrame` de *pandas*. Ahora, podemos continuar tranquilamente con el desarrollo de nuestro análisis, en cuanto hemos concluido satisfactoriamente el primer paso. Procederemos a verificar el cumplimiento de algunos de los supuestos sobre la estructura del modelo.

Supuesto de número de observaciones mayor a número de parámetros (Supuesto #6)

Uno de los supuestos sobre la estructura del modelo es que el número de observaciones con el cual se realiza la estimación debe superar el número de parámetros (β) a ser estimados. Si pretendemos trabajar con todas las observaciones del `DataFrame`, basta con examinar sus dimensiones para determinar la cantidad máxima de parámetros que podrían estimarse. Las dimensiones de un `DataFrame` corresponden al número de filas y columnas por los cuales está formado y pueden conocerse muy fácilmente (recuerde que ya se ha realizado la misma operación en el segundo capítulo): basta con emplear el atributo `.shape`, así:

```
In [4]: data.shape
```

```
Out[4]: (64, 4)
```

El resultado que obtenemos es una tupla Python (...) que informa que nuestro `DataFrame` es de tamaño 64×4, por lo que posee 64 filas y 4 columnas. Considerando que cada fila corresponde a una observación y cada columna a una variable, el número máximo de parámetros a estimar es 63.

¿Qué sucede si el número de observaciones es inferior al número de parámetros a estimar? "Si hay menos de k observaciones entonces $\boldsymbol{X}$ no puede ser de rango completo" (Greene, 2003, p. 14). Debido a esto, cuando el número de parámetros a estimar excede el número de observaciones disponibles para hacer la estimación, se obtiene que la varianza es infinita, por lo cual el método MCO no puede emplearse (James, Witten, Hastie, y Tibshirani, 2013).

A modo de ejemplo, se realizará una regresión de prueba, con fines puramente ilustrativos, en la que se usarán solo 3 observaciones para estimar 4 parámetros. Para esto, se seleccionarán únicamente las tres primeras observaciones del `DataFrame` recurriendo a los corchetes [ ]:

```
In [5]: Modelo_de_Prueba = sm.OLS(data["CM"][0:3],
                                  sm.add_constant(data[["FLR", "PGNP", "TFR"]][0:3]))
        Resultados_de_Prueba = Modelo_de_Prueba.fit()
        print(Resultados_de_Prueba.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                     CM   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                    nan
Method:                 Least Squares   F-statistic:                    0.000
Date:                xxx, xx xxx xxxx   Prob (F-statistic):              nan
Time:                        xx:xx:xx   Log-Likelihood:                 66.160
No. Observations:                   3   AIC:                            -126.3
Df Residuals:                       0   BIC:                            -129.0
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          3.2453        inf          0        nan         nan         nan
FLR            2.0501        inf          0        nan         nan         nan
PGNP          -0.0692        inf         -0        nan         nan         nan
TFR           26.7721        inf          0        nan         nan         nan
==============================================================================
Omnibus:                        nan   Durbin-Watson:                   0.854
Prob(Omnibus):                  nan   Jarque-Bera (JB):                0.511
Skew:                        -0.678   Prob(JB):                        0.774
Kurtosis:                     1.500   Cond. No.                         753.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The input rank is higher than the number of observations.
```

Como el lector puede observar, los errores estándar son infinitos, y los *p-values* e intervalos de confianza no están definidos, por lo que no resulta útil emplear los estimadores de MCO cuando el número de observaciones es inferior al número de parámetros que se pretende estimar.

Ciertamente, el ejemplo que se acaba de presentar es muy sencillo; sin embargo, es posible que el dataset a emplear tenga un gran número de variables explicativas, incluso muy superior a la cantidad de observaciones disponibles, por lo cual el Supuesto #6, a pesar de su apariencia inocente, no debe ser tomado a la ligera.

Cuando existen muchas más variables explicativas que observaciones, se puede recurrir a diversas técnicas y se pueden llevar a cabo procesos de selección de variables; sin embargo, tales procedimientos exceden el alcance de esta obra y no serán abordados en la misma.

Supuesto de variación en los valores de las variables explicativas (Supuesto #7)

El supuesto #7 indica que las variables explicativas deben ser "variables"; es decir, que los valores de las variables explicativas deben cambiar y no pueden corresponder a la misma constante. Intuitivamente, para que una variable explicativa sea "explicativa" sus cambios deben asociarse[1] a cambios en la variable dependiente;

[1] Esto no quiere decir que haya una relación causal. El hecho de que el movimiento de una variable se asocie al movimiento de otra no significa que ésta esté causando el cambio de la otra.

si el valor de una variable explicativa no cambia, resulta difícil identificar la manera como está asociada con la variable dependiente.

Identificar el cumplimiento de este supuesto es bastante sencillo: "si la desviación estándar muestral de las x_i es cero, entonces el supuesto no se satisface; si no es así, este supuesto se satisface" (Wooldridge, 2010, p. 48).

Para verificar en Python el cumplimiento del supuesto de variabilidad en las variables, se puede emplear, por ejemplo, el método `.apply()`, utilizando como argumento la función `std(...)` de *NumPy*:

```
In [6]: pd.DataFrame(data.apply(np.std, axis = 0),
                     columns = ["Desviación estándar"])
```

```
Out[6]:       Desviación estándar
        CM              75.382151
        FLR             25.803873
        PGNP          2704.317439
        TFR              1.497158
```

- **Nota:** La función `DataFrame(...)` de *pandas* se usa en este código con fines puramente estéticos, para obtener una presentación más agradable del resultado. La parte realmente importante, que genera el contenido en el que se está interesado, es `data.apply(np.std, axis = 0)`.

Como se evidencia, todas las desviaciones estándar son diferentes de 0, por lo que hay variabilidad en las variables (si la desviación estándar de alguna variable fuese 0, no se trataría de una variable, sino de una constante).

Supuesto de No Multicolinealidad (Supuesto #8)

Uno de los supuestos sobre la estructura del modelo es que no se presenta colinealidad exacta entre las variables explicativas. Al lector que desconozca la definición de colinealidad, se señala que ésta, de acuerdo a Dormann et al. (2013), "describe la situación en la cual dos o más variables predictoras en un modelo estadístico están linealmente relacionadas" (p. 28).

Un método para detectar colinealidad entre variables es calcular el coeficiente de correlación simple entre pares de estas; si alguno de estos coeficientes es mayor, en valor absoluto, a 0.9, entonces hay síntoma de colinealidad muy fuerte.

Para examinar la correlación entre las variables numéricas de un `DataFrame` de *pandas*, es posible emplear el método `.corr()`. Así:

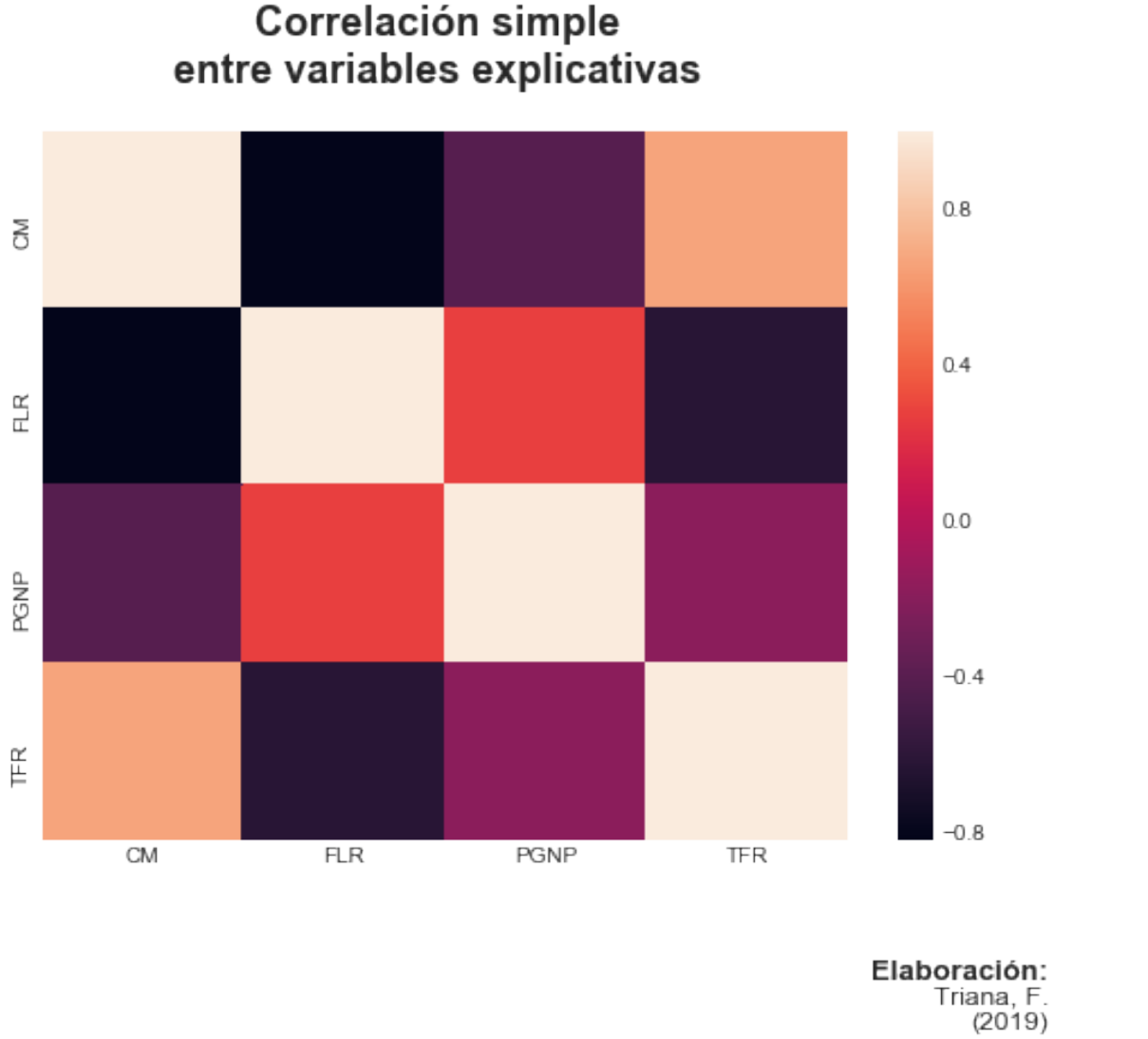
```
In [7]: data.corr()
```

```
Out[7]:             CM       FLR       PGNP       TFR
        CM    1.000000 -0.818285 -0.407697  0.671135
        FLR  -0.818285  1.000000  0.268530 -0.625954
        PGNP -0.407697  0.268530  1.000000 -0.185718
        TFR   0.671135 -0.625954 -0.185718  1.000000
```

Si el usuario prefiere una representación un tanto más "rica" visualmente, puede recurrir a la función `heatmap(...)` de *Seaborn*, la cual dibuja una matriz codificada por color. El argumento a pasar a la función, para este caso específico, es la matriz de correlaciones, la cual, tal y como hemos apenas visto, es obtenida con la aplicación del método `.corr()` sobre el `DataFrame`.

El código a emplear es el siguiente:

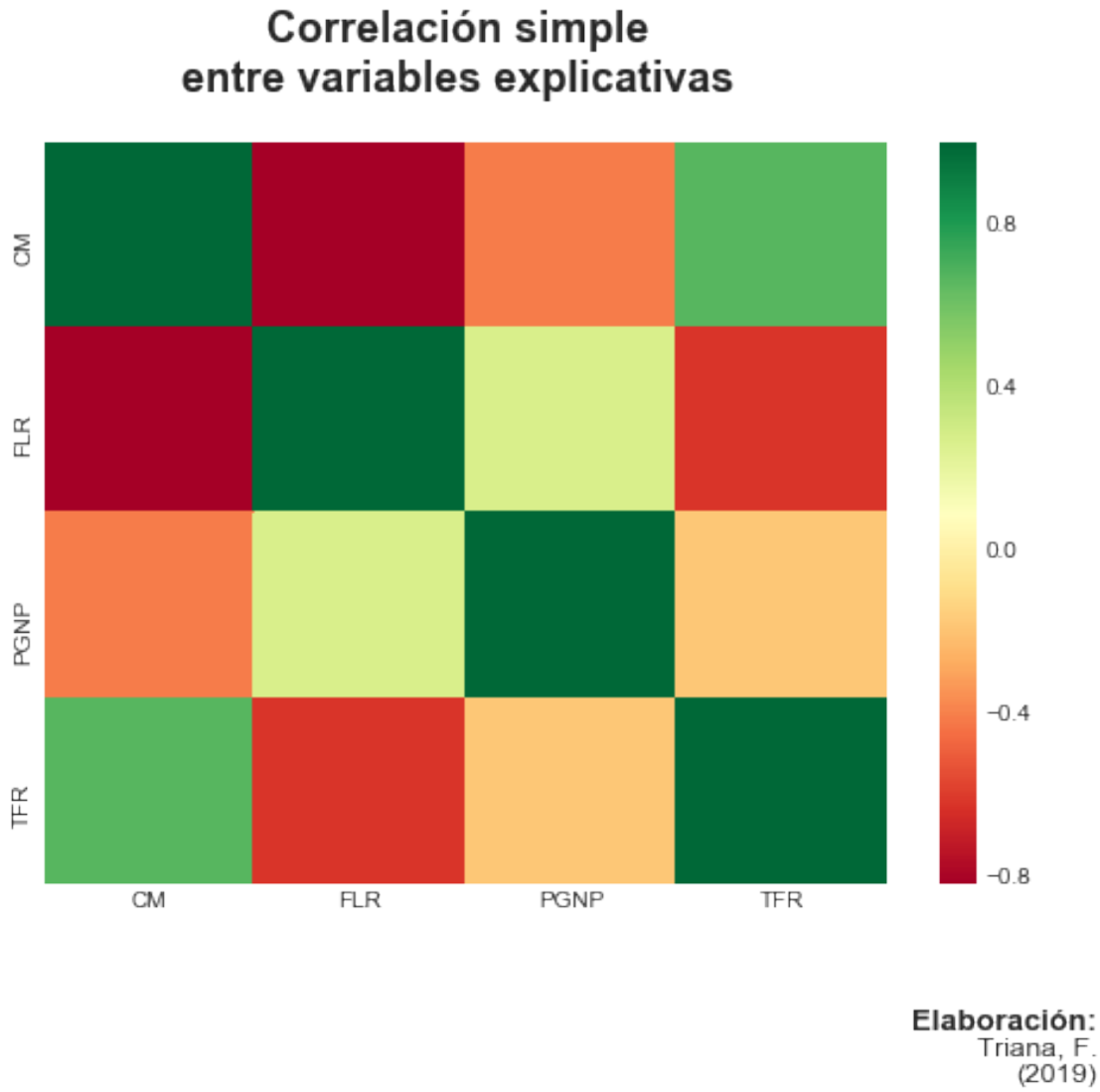
```
In [8]: fig, ax = plt.subplots(figsize = (8, 6))
        fig.suptitle("Correlación simple\nentre variables explicativas",
                     fontsize = 18,
                     fontweight = "bold", x = 0.43)
        sns.heatmap(data.corr(), ax = ax)
        plt.subplots_adjust(top = 0.85)
        fig.text(.9,-.02,
                 "Elaboración:",
                  fontsize = 13, fontweight = "bold",
                ha = "right")
        fig.text(.9,-.07,
                 "Triana, F.\n(2019)",
                 fontsize = 12, ha = "right")
        plt.show()
```



- **Nota:** El lector debe notar que, dado que *Seaborn* es una librería basada en *Matplotlib* y que la función `heatmap(...)` de *Seaborn* devuelve un `Axes` de una figura de *Matplotlib*, el código empleado en el bloque anterior es código común de esta última librería. Dado que *Matplotlib* desempeña un rol auxiliar en este trabajo, no se dedicarán esfuerzos a examinar detalladamente la estructura de su código, por lo que al lector interesado en ésta se sugiere consultar recursos adicionales.

Podemos observar que la escala de color usada por defecto por la función `heatmap(...)` de *Seaborn* no parece ser la más adecuada. Es posible modificar esto incluyendo un valor específico para el parámetro *cmap*, asignando la escala que nos parezca la más indicada. A modo de ejemplo, en el siguiente código se empleará la escala 'RdYlGn' (rojos, amarillos, verdes):

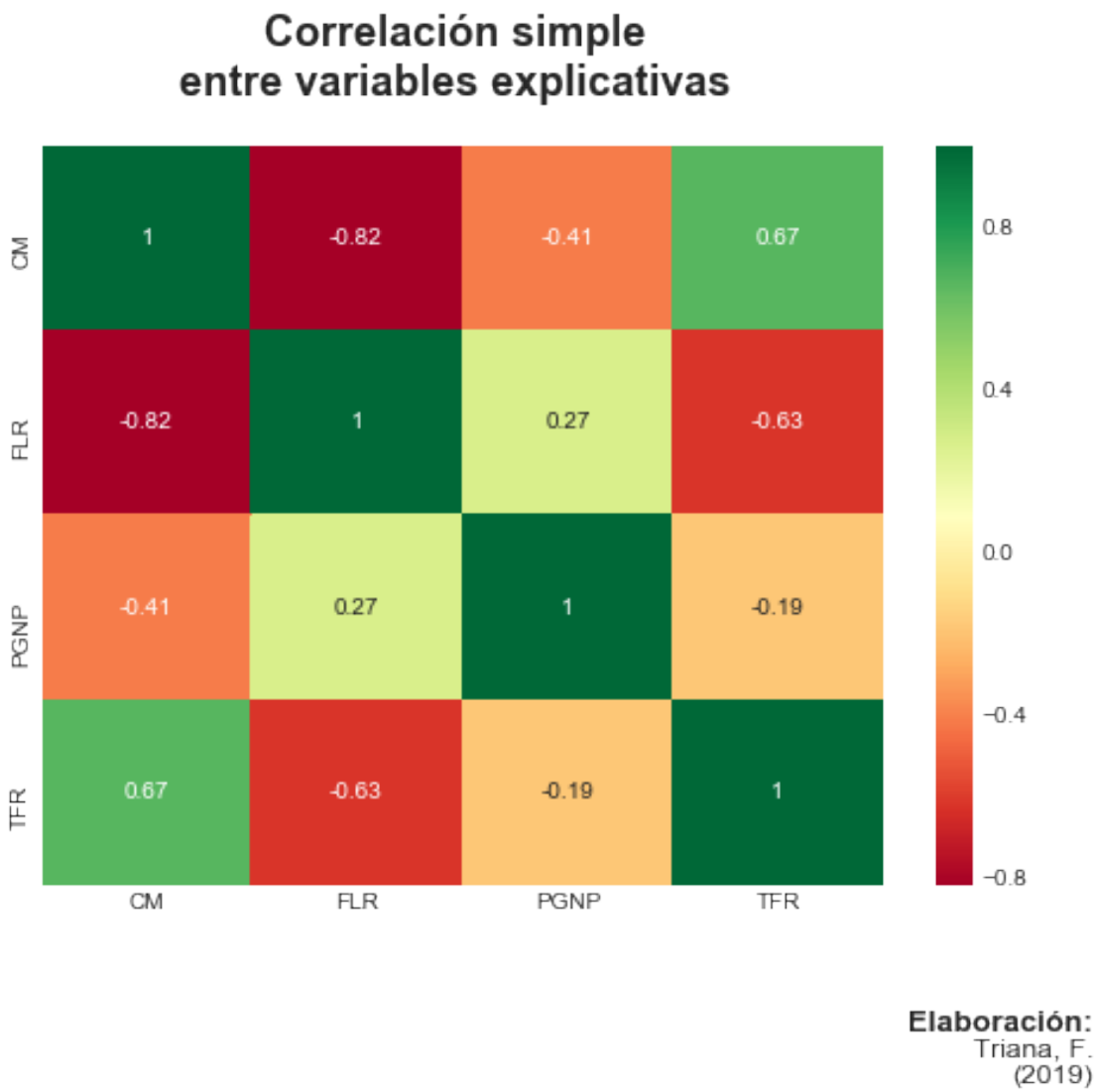
```
In [9]: fig, ax = plt.subplots(figsize = (8, 6))
        fig.suptitle("Correlación simple\nentre variables explicativas",
                     fontsize = 18,
                     fontweight = "bold", x = 0.43)
        sns.heatmap(data.corr(), ax = ax, cmap = "RdYlGn")
        plt.subplots_adjust(top = 0.85)
        fig.text(.9,-.02,
                 "Elaboración:",
                  fontsize = 13, fontweight = "bold",
                ha = "right")
        fig.text(.9,-.07,
                 "Triana, F.\n(2019)",
                 fontsize = 12, ha = "right")
        plt.show()
```



Si consideramos que nuestro gráfico aún no es lo suficientemente informativo, podemos agregar el coeficiente de correlación simple correspondiente a cada una de las celdas. Para alcanzar tal objetivo, debemos incluir el parámetro *annot* en la función `heatmap(...)` de *Seaborn* y asignarle el valor de `True` (*annot* es un parámetro de tipo booleano). Así:

```
In [10]: fig, ax = plt.subplots(figsize = (8, 6))
         fig.suptitle("Correlación simple\nentre variables explicativas",
                      fontsize = 18,
                      fontweight = "bold", x = 0.43)
         sns.heatmap(data.corr(), ax = ax, cmap = "RdYlGn",
                     annot = True)
         plt.subplots_adjust(top = 0.85)
         fig.text(.9,-.02,
                  "Elaboración:",
```

```
            fontsize = 13, fontweight = "bold",
          ha = "right")
fig.text(.9,-.07,
          "Triana, F.\n(2019)",
          fontsize = 12, ha = "right")
plt.show()
```



Así, notamos que la correlación más fuerte se presenta entre las variables 'CM' y 'FLR', con un coeficiente superior a 0.8 (en valor absoluto); sin embargo, tal magnitud del coeficiente no debe ser una preocupación mayor, en cuanto el Supuesto #8 se refiere a colinealidad entre variables *explicativas* y la variable 'CM' es la variable *dependiente*, mientras que 'FLR' es una variable explicativa.

El Supuesto #8 hace referencia a colinealidad *exacta*; esto es cuando una variable explicativa tiene una relación lineal exacta con otra. Cuando se presenta colinealidad exacta, los coeficientes de regresión son indeterminados y los errores estándar son infinitos (Gujarati y Porter, 2010).

En nuestro caso específico, ninguna de las variables explicativas presenta una correlación muy fuerte con alguna otra variable explicativa distinta a ella misma, por lo tanto, no se presenta multicolinealidad perfecta y es posible estimar los parámetros.

Otra técnica útil para detectar multicolinealidad es por medio del Factor de Inflación de Varianza (VIF, por sus siglas en inglés). "VIFs altos reflejan un incremento en las varianzas de los coeficientes de regresión estimados debido a colinealidad entre variables predictoras, en comparación con las que se obtendrían cuando las predictoras son ortogonales" (Murray, Nguyen, Lee, Remmenga, y Smith, 2012).

Así, en caso de que no haya multicolinealidad, o ésta sea leve, se espera que la magnitud de los VIFs sea pequeña y entre más fuerte sea la colinealidad más alto será el VIF; Gujarati y Porter (2010), basados en el trabajo de Kleinbaum, Kupper y Muller (1988), sugieren una regla práctica: "si el FIV [VIF] de una variable es superior a 10 (esto sucede si R_j^2 excede 0.90), se dice que esa variable es muy colineal" (p.340).

¿Cuáles son los VIFs de las variables del dataset con el que se está trabajando? Para obtener estos valores, se hará uso de la función `variance_inflation_factor(...)` del módulo `stats.outliers_influence` de la librería *Statsmodels*.

La función `variance_inflation_factor(...)` recibe dos argumentos: la matriz que contiene las variables explicativas (*exog*) y un índice (*exog_idx*) que señala qué variable es a la que corresponde el VIF calculado. Esta función calcula el VIF para la variable señalada; sin embargo, es de interés conocer el VIF de cada una de las variables explicativas.

Es posible aplicar la función `variance_inflation_factor(...)` de forma "manual" tantas veces como variables de interés haya; para nuestro caso, hay 3 variables explicativas, por lo que la función solo debe aplicarse 3 veces, una operación que no es excesivamente tediosa. Sin embargo, si se tuviera un número apreciable de variables explicativas, resultaría molesto repetir la operación en múltiples ocasiones, por lo que resulta útil poder "automatizar" tal tarea. Una opción para lograr esto es utilizar un ciclo `for`:

```
In [11]: from statsmodels.stats.outliers_influence import variance_inflation_factor
         vif = []
         exog = sm.add_constant(data[data.columns[1:4]])
         rango = sm.add_constant(data[data.columns[1:4]]).shape[1]
         for i in range(rango):
             vif.append(variance_inflation_factor(exog.values, i))
         pd.DataFrame({'VIF': vif[1:]}, index= data.columns[1:4])

Out[11]:            VIF
         FLR   1.711845
         PGNP  1.078306
         TFR   1.645150
```

Así, se puede observar que ninguna de las variables explicativas exhibe un VIF elevado, por lo que no se sugiere que alguna de estas variables sea muy colineal. Este resultado está en línea con lo que se obtuvo en la matriz de correlaciones, en la cual se encontró que ninguno de los coeficientes de correlación simple era muy alto.

Supuesto de No sesgo de especificación (Supuesto #9)

Uno de los supuestos sobre el modelo es que éste debe estar correctamente especificado, es decir, que la forma funcional que adopta es la correcta y que las variables incluidas son las adecuadas.

Visualmente, es posible examinar si la forma funcional adoptada es la adecuada al graficar, por medio de un diagrama de dispersión, los valores estimados contra los residuos; se espera que el comportamiento sea

relativamente estable, con los puntos distribuidos simétricamente alrededor del cero, sin presentar patrones específicos.

Dado que aún no se ha creado el modelo ni llevado a cabo la estimación, no se cuenta con una instancia de resultados a emplear. Por lo tanto, lo primero que se hará es realizar la construcción y estimación correspondientes:

```
In [12]: Y = data["CM"]
         X = data[["PGNP", "FLR"]]
         Modelo = sm.OLS(Y, sm.add_constant(X))
         Resultados = Modelo.fit()
         print(Resultados.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                     CM   R-squared:                       0.708
Model:                            OLS   Adj. R-squared:                  0.698
Method:                 Least Squares   F-statistic:                     73.83
Date:                xxx, xx xxx xxxx   Prob (F-statistic):           5.12e-17
Time:                        xx:xx:xx   Log-Likelihood:                -328.10
No. Observations:                  64   AIC:                             662.2
Df Residuals:                      61   BIC:                             668.7
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        263.6416     11.593     22.741      0.000     240.460     286.824
PGNP          -0.0056      0.002     -2.819      0.006      -0.010      -0.002
FLR           -2.2316      0.210    -10.629      0.000      -2.651      -1.812
==============================================================================
Omnibus:                        0.732   Durbin-Watson:                   2.186
Prob(Omnibus):                  0.693   Jarque-Bera (JB):                0.559
Skew:                           0.228   Prob(JB):                        0.756
Kurtosis:                       2.949   Cond. No.                     6.77e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 6.77e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

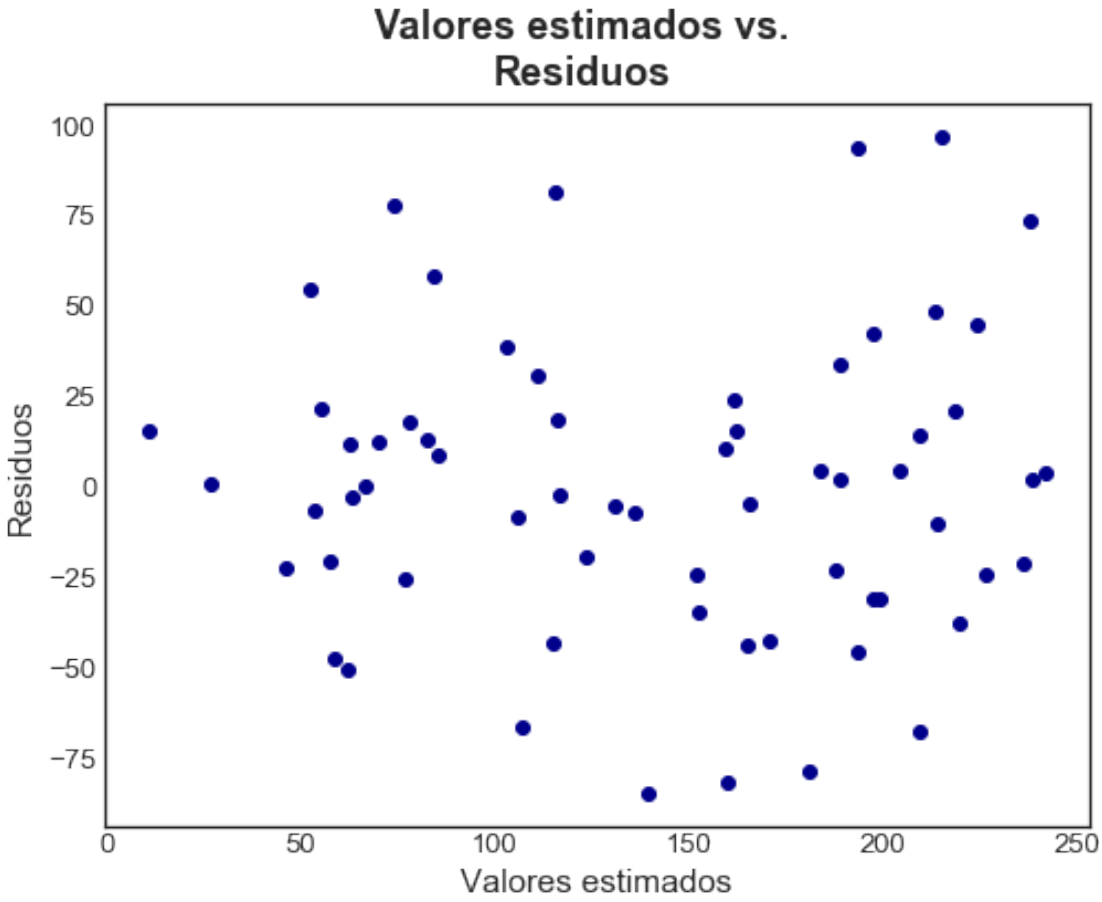
Ya se cuenta con el modelo y sus resultados, pero aún no hemos asignado los residuos a un objeto, por lo que llevaremos a cabo tal asignación, de modo que se simplifique el código empleado. Asignaremos los residuos, los cuales son un atributo de la instancia de resultados (el usuario puede verificarlo con la función `dir(...)`), al objeto `Residuos`, así:

```
In [13]: Residuos = Resultados.resid
```

Del mismo modo, es posible asignar los valores estimados, que se obtienen con la aplicación del método `.predict()` a un objeto específico:

```
In [14]: Valores_estimados = Resultados.predict()

In [15]: fig, ax = plt.subplots(figsize = (8,6))
         fig.suptitle("Valores estimados vs.\nResiduos",
                      fontsize = 18,
                      fontweight = "bold")
         ax.scatter(Valores_estimados, Residuos,
                    color = "darkblue")
         ax.set_xlabel("Valores estimados", fontsize = 15)
         ax.set_ylabel("Residuos", fontsize = 15)
         ax.tick_params(labelsize = 13)
         fig.text(.9,-.02,
                  "Elaboración:",
                   fontsize = 13, fontweight = "bold",
                  ha = "right")
         fig.text(.9,-.07,
                  "Triana, F.\n(2019)",
                  fontsize = 12, ha = "right")
         plt.show()
```



Elaboración:
Triana, F.
(2019)

Como se observa, no parece que se presente un patrón específico y los puntos están simétricamente distribuidos alrededor del cero, por lo que el análisis visual sugiere que la forma funcional adoptada no es errónea.

Otro aspecto a considerar respecto a la estructura del modelo es que ésta se mantenga a lo largo de la estimación; es decir, que no sea inestable. Para verificar la estabilidad en los parámetros, o no cambio estructural, puede emplearse la prueba CUSUM basada en residuos de MCO, la cual toma como hipótesis nula no cambio estructural.

La función empleada en Python para la prueba CUSUM basada en residuos de MCO es `breaks_olsresid(...)` de *statsmodels.stats*; sus argumentos incluyen los residuos del modelo estimado por MCO y el parámetro *ddof*, con valor por defecto igual a 0 y referente al número de grados de libertad empleados en la corrección de la varianza del error.

Ahora, emplearemos el objeto `Residuos` como argumento de la función `breaks_olsresid(...)`; los resultados obtenidos con la ejecución de la función serán almacenados en un objeto al que denominaremos `ResultadosTest`, así:

```
In [16]: ResultadosTest = sms.breaks_cusumolsresid(Residuos)
         print(ResultadosTest)
```

```
(0.5191974341185455, 0.9503227705917948, [(1, 1.63), (5, 1.36), (10, 1.22)])
```

Podemos ver que el resultado que obtenemos de nuestro código es tan solo una **tupla Python** (...) que contiene un conjunto de datos; esto se debe a que *statsmodels.stats* no tiene integrado un modo de presentación más "elaborado" del resultado.

Para mejorar la presentación de los resultados del Test se puede hacer uso de una `Series` de *pandas*, de modo que podamos organizar adecuadamente la información disponible. Procederemos a crear dicha `Series` asignándole como valores los componentes del resultado del test y como *index* una **lista Python** [...] con sus respectivos nombres. Así:

```
In [17]: Nombres = ["Estadístico", "p-value del estadístico", "Valores críticos"]
         pd.Series(ResultadosTest, index = Nombres)

Out[17]: Estadístico                                    0.519197
         p-value del estadístico                        0.950323
         Valores críticos           [(1, 1.63), (5, 1.36), (10, 1.22)]
         dtype: object
```

Asumiendo un nivel de significancia de 5 %, como $p - value > \alpha$ entonces no se rechaza la hipótesis nula y se concluye que no se presenta cambio estructural (a un nivel de confianza de 95 %).

Ya se han evaluado, muy brevemente, algunos de los supuestos del modelo clásico de regresión lineal sobre la estructura del modelo. Ahora, se examinará el cumplimiento de los supuestos referentes al término de error.

Supuestos sobre el término de error

Se dará continuidad al modelo empleado en la sección "Supuestos sobre la estructura del modelo", por lo que no se llevará a cabo la "Preparación del entorno" y la "Importación de los datos", en cuanto estas ya se han efectuado de forma exitosa previamente. Al lector que no haya seguido la información presentada, se le sugiere dirigirse al inicio de este capítulo y ejecutar las dos primeras celdas de código; una vez hecho esto, debería poder continuar con la exposición presentada sin inconvenientes.

Valor medio de la perturbación igual a cero (Supuesto #3)

Los residuos del modelo pueden ser positivos o negativos, es decir, es posible que el valor estimado por el modelo sea inferior al verdadero o superior al mismo, pero la distribución debe ser relativamente simétrica. No se espera que todos los residuos sean positivos ni que todos sean negativos, pues se estaría realizando una subestimación o sobreestimación sistemática; lo adecuado es que exista balance entre los residuos positivos y negativos, de modo que su valor promedio sea cero.

Este supuesto es relativamente fácil de evaluar, pues tan solo debemos calcular el valor promedio de los residuos, para lo que la función `mean(...)` de *NumPy* resulta indicada:

```
In [18]: np.mean(Residuos)

Out[18]: 1.254552017826427e-13
```

El argumento que recibe la función `mean(...)` de *NumPy* es el vector de residuos, el cual, convenientemente, hemos asignado de forma previa al objeto `Residuos`. El resultado obtenido es prácticamente cero, pues

se trata de un número, expresado en notación científica, que es muy pequeño; esto sugiere que el Supuesto #3 se cumple para este ejemplo.

Si bien hemos utilizado una **función**, puede resultar útil señalar al lector que también existe un **método**, con el que es posible calcular el promedio, el cual, como es de esperarse, es `.mean()`:

```
In [19]: Residuos.mean()

Out[19]: 1.254552017826427e-13
```

Homoscedasticidad (Supuesto #4)

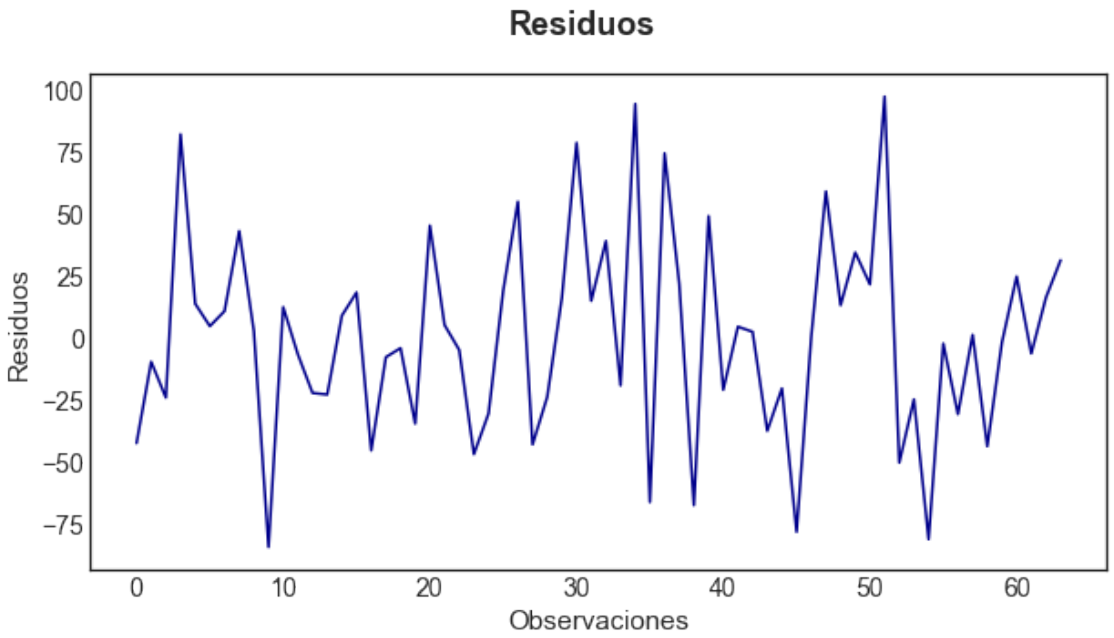
El supuesto de homoscedasticidad en el término de error indica, esencialmente, que éste debe exhibir varianza constante. Existen pruebas estadísticas formales para verificar el cumplimiento de este supuesto, sin embargo, lo primero que haremos será emplear métodos informales, de modo que el lector tenga una idea un tanto más clara de la cuestión a tratar.

Examinaremos la presencia de heteroscedasticidad (esto es, varianza no constante o "no-homoscedasticidad") informalmente por medio de un análisis visual. Lo que haremos es graficar los residuos del modelo, con el objetivo de observar si su comportamiento exhibe un patrón particular que pueda señalar la presencia de heteroscedasticidad. (Véase la Sección 11.5, del Capítulo 11, *Heteroscedasticidad: ¿qué pasa si la varianza del error no es constante?*, de Gujarati y Porter (2010))

Para realizar el gráfico correspondiente usaremos el método `.plot()` sobre un `Axes` creado con la función `subplots(...)` de *Matplotlib*, empleando como argumento los residuos del modelo; estos residuos han sido previamente asignados a un objeto, por lo que es este objeto el que se empleará como argumento. Al lector que no haya seguido el ejercicio realizado en la sección "Supuestos sobre la estructura del modelo", se le invita a consultarla y ejecutar las celdas de código correspondientes, con las que podrá generar los objetos requeridos a continuación.

El contenido de la celda de código para la generación del gráfico es el siguiente:

```
In [20]: fig, ax = plt.subplots(figsize = (10,5))
         fig.suptitle("Residuos", fontsize = 18,
                      fontweight = "bold")
         ax.plot(Residuos, color = "darkblue")
         ax.set_xlabel("Observaciones", fontsize = 15)
         ax.set_ylabel("Residuos", fontsize = 15)
         ax.tick_params(labelsize = 14)
         fig.text(.9,-.02,
                  "Elaboración:",
                  fontsize = 13, fontweight = "bold",
                  ha = "right")
         fig.text(.9,-.08,
                  "Triana, F.\n(2019)",
                  fontsize = 12, ha = "right")
         plt.show()
```



Elaboración:
Triana, F.
(2019)

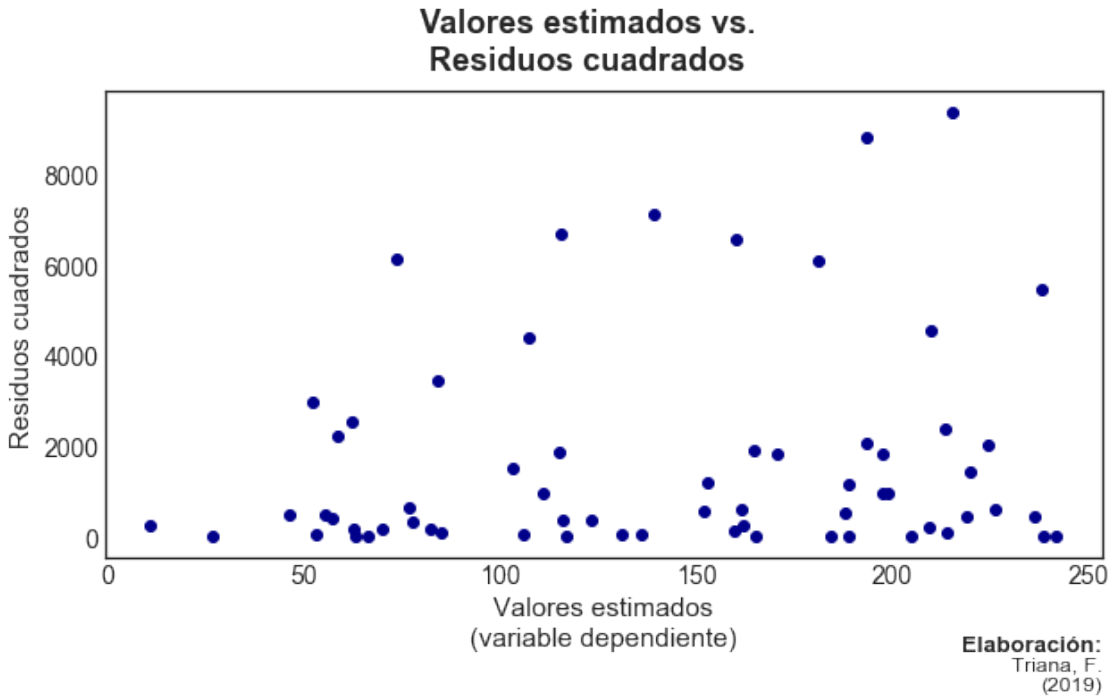
Como se puede observar, los residuos no exhiben un patrón particular y la amplitud de sus cambios no parece variar significativamente, por lo que, a primera vista, no hay evidencia de heteroscedasticidad. Sin embargo, tal conclusión puede resultar apresurada.

Un análisis similar puede llevarse a cabo empleando los valores estimados y los residuos **al cuadrado**, lo que permite controlar por la naturaleza de los residuos (positivos o negativos) al basarse en su magnitud y no en su signo. Se espera que no se presente ningún patrón particular, de modo que la varianza de los residuos sea constante y estos no dependan de la magnitud de la variable regresada.

Para ejecutar este análisis utilizaremos el método `.scatter(...)` empleando como argumentos el vector de valores estimados y el de residuos al cuadrado. Ya contamos con el primero, pero solo se tiene información de los residuos y no de los residuos **cuadrados**; sin embargo, basta con aplicar el operador `**` para obtener dichas magnitudes, ya que éste es el que permite en Python realizar potenciación siguiendo la estructura `base**exponente`:

```
In [21]: fig, ax = plt.subplots(figsize = (10,5))
         fig.suptitle("Valores estimados vs.\nResiduos cuadrados",
                      fontsize = 18,
                      fontweight = "bold")
         ax.scatter(Valores_estimados, Residuos**2,
                    color = "darkblue")
         ax.set_xlabel("Valores estimados\n(variable dependiente)",
                       fontsize = 15)
         ax.set_ylabel("Residuos cuadrados", fontsize = 15)
         ax.tick_params(labelsize = 14)
         fig.text(.9,-.02,
                  "Elaboración:",
```

```
            fontsize = 13, fontweight = "bold",
            ha = "right")
fig.text(.9,-.08,
            "Triana, F.\n(2019)",
            fontsize = 12, ha = "right")
plt.subplots_adjust(top = 0.85)
plt.show()
```



De acuerdo al resultado obtenido, no parece que los residuos sean heteroscedásticos, aunque hay una ligera impresión de que van creciendo con la magnitud de la variable dependiente; para llegar a una conclusión "válida", el análisis visual no es suficiente: se requiere de pruebas estadísticas formales.

Una de las pruebas estadísticas generalmente empleadas para verificar la presencia de heteroscedasticidad es el Test Breusch-Pagan de Multiplicadores de Lagrange (Greene, 2003), el cual, básicamente, toma como hipótesis nula homoscedasticidad.

La función empleada en Python para el Test **Breusch-Pagan** de Multiplicadores de Lagrange para heteroscedasticidad es `het_breuschpagan(...)` de *statsmodels.stats*; sus argumentos incluyen los residuos del modelo y el conjunto de variables que pueden causar la heteroscedasticidad.

Ya se cuenta con un objeto que contiene los residuos, sin embargo, aún no se han asignado las variables explicativas del modelo a un objeto particular. Como las variables explicativas consisten en un **atributo** de la instancia de resultados del modelo, podemos asignarlas a un objeto aplicando dicho atributo, así:

```
In [22]: Explicativas = Resultados.model.exog
```

Ya que contamos con los objetos a emplear como argumentos, podemos hacer uso de la función `het_breuschpagan(...)` de *statsmodels.stats*. Siguiendo la estructura que hemos empleado para las pruebas de la sección anterior, el código a ejecutar es el siguiente:

```
In [23]: ResultadosTest = sms.het_breuschpagan(Residuos, Explicativas)
         Nombres = ["Estadístico LM", "p-value del estadístico LM", "Estadístico F",
                    "p-value del estadístico F"]
         pd.Series(ResultadosTest, index = Nombres)

Out[23]: Estadístico LM                1.591582
         p-value del estadístico LM    0.451224
         Estadístico F                 0.777832
         p-value del estadístico F     0.463904
         dtype: float64
```

Asumiendo un nivel de significancia de 5 %, como $p - value > \alpha$ entonces no se rechaza la hipótesis nula y se concluye que se presenta homoscedasticidad en el término de error (trabajando con un α de 0.05).

- **Nota 1:** Este test equivale al generado por la función `bptest(...)` de **R** (paquete *lmtest*).
- **Nota 2:** Este test equivale al generado por el código *estat hettest, rhs fstat* de **Stata**.

Otra de las pruebas estadísticas comúnmente empleadas para verificar la presencia de heteroscedasticidad es el Test **White** de Multiplicadores de Lagrange, el cual hace uso de regresiones auxiliares, toma, esencialmente, como hipótesis nula homoscedasticidad, y es capaz de identificar formas más generales de heteroscedasticidad que el Test de Breusch-Pagan (Verbeek, 2004).

La función empleada en Python para el Test White de Multiplicadores de Lagrange para heteroscedasticidad es `het_white(...)` de *statsmodels.stats*; sus argumentos incluyen los residuos del modelo y el conjunto de variables a emplear en las regresiones auxiliares.

Los argumentos empleados en la función `het_white(...)` de *statsmodels.stats* corresponden a los mismos utilizados en la función `het_breuschpagan(...)` de dicho módulo y, dado que estos han sido asignados previamente a objetos específicos, basta con reutilizar tales objetos.

Siguiendo la estructura que se ha empleado para las demás pruebas estadísticas realizadas, el código a ejecutar es el siguiente:

```
In [24]: ResultadosTest = sms.het_white(Residuos, Explicativas)
         Nombres = ["Estadístico LM", "p-value del estadístico LM",
                    "Estadístico F", "p-value del estadístico F"]
         pd.Series(ResultadosTest, index = Nombres)

Out[24]: Estadístico LM                2.356718
         p-value del estadístico LM    0.797902
         Estadístico F                 0.443486
         p-value del estadístico F     0.816252
         dtype: float64
```

Los resultados del Test White de Multiplicadores de Lagrange para heteroscedasticidad reafirman la conclusión del Test Breusch-Pagan realizado con anterioridad. Asumiendo un nivel de significancia de 5 %, como $p - value > \alpha$ entonces no se rechaza la hipótesis nula y se concluye que se presenta homoscedasticidad en el término de error (a un nivel de confianza de 95 %).

- **Nota:** Este test equivale al generado por el código *estat imtest, white* de **Stata**.

No autocorrelación, o correlación serial, entre las perturbaciones (Supuesto #5)

El supuesto de no autocorrelación entre las perturbaciones indica que estas no siguen patrones sistemáticos y no están correlacionadas entre sí; tal supuesto puede justificarse para el caso de datos transversales, pero tiende a incumplirse cuando se trabaja con series de tiempo, en cuanto las observaciones sucesivas usualmente están fuertemente correlacionadas (Véase la Sección 3.2, del Capítulo 3, *Modelos de regresión con dos variables: problema de estimación*, de Gujarati y Porter (2010)).

Este supuesto normalmente se verifica con pruebas estadísticas, algunas de las cuales se tratan a continuación. Una prueba estadística generalmente empleada para verificar la presencia de autocorrelación de orden 1 es el Test **Durbin-Watson**, el cual toma, esencialmente, como hipótesis nula no autocorrelación (de orden 1) y genera un estadístico cuyo valor se contrasta con los valores críticos correspondientes (Chatterjee y Simonoff, 2013) para establecer una conclusión.

La función empleada en Python para el Test Durbin-Watson es `durbin_watson(...)` de *statsmodels.stats*; sus argumentos incluyen los residuos del modelo.

Siguiendo la misma estructura de las otras pruebas estadísticas realizadas, y teniendo en cuenta que los residuos ya se asignaron a un objeto específico, el código a ejecutar es el siguiente:

```
In [25]: ResultadosTest = sms.durbin_watson(Residuos)
         Nombres = ["Estadístico DW"]
         pd.Series(ResultadosTest, index = Nombres)

Out[25]: Estadístico DW    2.186159
         dtype: float64
```

El valor obtenido es 2.1861, entonces, ¿se presenta autocorrelación de orden 1? El lector debe notar que, a diferencia de otras pruebas estadísticas, en el caso del Test Durbin-Watson no se reporta un *p-value* con el cual decidir si se rechaza o no la hipótesis nula, entonces, ¿cómo se determina la conclusión?

El estadístico Durbin-Watson es contrastado con valores críticos a partir de los cuales se establecen zonas de autocorrelación positiva, no autocorrelación, indeterminación y autocorrelación negativa; dependiendo de la zona en la que se ubique el valor del estadístico, se obtiene una conclusión específica.

Para este caso en concreto, asumiendo un nivel de significancia de 5 %, los valores de d_L y d_U para 2 variables explicativas (excluyendo el término constante) y 65 observaciones son, respectivamente, 1.536 y 1.662. El valor de 2.1861 pertenece al intervalo $(d_U, 4 - d_U)$, el cual corresponde a la zona de No Autocorrelación, por lo que se concluye que no se presenta autocorrelación de primer orden (trabajando con un α de 0.05).

- **Nota:** Este test equivale al generado por la función `dwtest(...)` de **R** (paquete *lmtest*). El usuario debe tener en cuenta que, en la función de R, a diferencia de la de Python, el argumento a emplear no son los residuos sino el propio modelo.

El Test Durbin-Watson es una prueba estadística que permite identificar autocorrelación de primer orden, sin embargo, si se quiere examinar correlación serial de orden superior ésta ya no puede emplearse y generalmente se recurre al Test **Breusch-Godfrey**, el cual toma, esencialmente, no autocorrelación como hipótesis nula y se basa en la ejecución de una regresión auxiliar en la que se incluyen rezagos del término de error como variables explicativas (Kleiber y Zeileis, 2008).

La función empleada en Python para el Test Breusch-Godfrey de Multiplicadores de Lagrange para autocorrelación de los residuos es `acorr_breusch_godfrey(...)` de *statsmodels.stats*; sus argumentos incluyen la instancia de resultados (¡no los residuos!) del modelo y el número de rezagos, *nlags*, a incluir en la regresión auxiliar.

Siguiendo la misma estructura de las otras pruebas estadísticas realizadas, y teniendo en cuenta que la instancia de resultados del modelo ya se asignó a un objeto específico, el código a ejecutar es el siguiente:

```
In [26]: ResultadosTest = sms.acorr_breusch_godfrey(Resultados, nlags = 1)
         Nombres = ["Estadístico LM", "p-value del estadístico LM", "Estadístico F",
                    "p-value del estadístico F"]
         pd.Series(ResultadosTest, index = Nombres)

Out[26]: Estadístico LM                0.728605
         p-value del estadístico LM    0.393336
         Estadístico F                 0.690933
         p-value del estadístico F     0.409143
         dtype: float64
```

Los resultados del Test Breusch-Godfrey de Multiplicadores de Lagrange para autocorrelación de los residuos reafirman la conclusión del Test Durbin-Watson realizado con anterioridad. Asumiendo un nivel de significancia de 5 %, como $p - value > \alpha$ entonces no se rechaza la hipótesis nula y se concluye que no se presenta correlación serial de primer orden en el término de error (trabajando con un nivel de significancia de 5 %).

Ya se ha indicado que el Test Breusch-Godfrey es una prueba general de autocorrelación, en cuanto permite identificar correlación serial de orden superior a 1. En el código anterior hemos asignado el valor de 1 al parámetro *nlags* de la función `acorr_breusch_godfrey(...)` de *statsmodels.stats*, por lo que hemos examinado la presencia de autocorrelación de primer orden, al igual que en el Test Durbin-Watson. Ahora, procederemos a examinar la presencia de autocorrelación hasta de orden 4, por lo que *nlags* tomará el valor de 4. Basta con reutilizar la última celda de código que hemos empleado y modificar el valor del parámetro *nlags*, así:

```
In [27]: ResultadosTest = sms.acorr_breusch_godfrey(Resultados, nlags = 4)
         Nombres = ["Estadístico LM", "p-value del estadístico LM", "Estadístico F",
                    "p-value del estadístico F"]
         pd.Series(ResultadosTest, index = Nombres)

Out[27]: Estadístico LM                1.211387
         p-value del estadístico LM    0.876220
         Estadístico F                 0.274927
         p-value del estadístico F     0.892978
         dtype: float64
```

Para el caso de autocorrelación hasta de orden 4, asumiendo un nivel de significancia de 5 %, como $p - value > \alpha$ entonces no se rechaza la hipótesis nula y se concluye que no hay correlación serial hasta de orden 4 (trabajando con un α de 0.05).

- **Nota:** Este test equivale al generado por la función `bgtest(...)` de **R** (paquete *lmtest*).

Otras pruebas estadísticas comúnmente empleadas para examinar la presencia de autocorrelación en el término de error son el Test **Box-Pierce** y el Test **Ljung-Box**, los cuales están estrechamente relacionados, toman como hipótesis nula, básicamente, no autocorrelación y, al igual que el Test Breusch-Godfrey, son útiles para determinar la existencia de autocorrelación hasta de orden p.

La función empleada en Python para el Test Ljung-Box es la misma que para el Test Box-Pierce. Se trata de `acorr_ljungbox(...)` de *statsmodels.stats*; sus argumentos incluyen los residuos del modelo, el número de rezagos deseados (*lags*) y una variable booleana para indicar si se quiere obtener únicamente los resultados del Test Ljung-Box (efectuado por defecto) o si también se desea visualizar los resultados del Test Box-Pierce (`boxpierce = True o False`).

Siguiendo la misma estructura de las otras pruebas estadísticas realizadas, y teniendo en cuenta que los residuos ya se asignaron a un objeto específico, el código a ejecutar es el siguiente:

```
In [28]: ResultadosTest = sms.acorr_ljungbox(Residuos, lags = 1, boxpierce = True)
         Nombres = ["Estadístico LB", "p-value del estadístico LB", "Estadístico BP",
                    "p-value del estadístico BP"]
         pd.Series(ResultadosTest, index = Nombres)

Out[28]: Estadístico LB                [0.7549153999366602]
         p-value del estadístico LB    [0.3849244343516546]
         Estadístico BP                [0.7206010635759029]
         p-value del estadístico BP    [0.3959468201677663]
         dtype: object
```

Tanto el *p-value* correspondiente al estadístico del Test Ljung-Box como el correspondiente al Test Box-Pierce son superiores a 0.05, por lo que no se rechaza la hipótesis nula y se concluye que no se presenta correlación serial de primer orden (observe que `lags = 1`).

Al igual que en el Test Breusch-Godfrey, el Test Ljung-Box y el Test Box-Pierce pueden emplearse para examinar la existencia de correlación serial hasta de orden p. Para examinar hasta p orden de autocorrelación, basta con asignar el valor de p al parámetro *lags* de la función `acorr_ljungbox(...)` de *statsmodels.stats*.

Siguiendo la misma estructura de las otras pruebas estadísticas realizadas, y teniendo en cuenta que los residuos ya se asignaron a un objeto específico, el código a ejecutar para evaluar correlación serial hasta de orden 2 (por ejemplo) es el siguiente:

```
In [29]: ResultadosTest = sms.acorr_ljungbox(Residuos, lags = 2, boxpierce = True)
         Nombres = ["Estadístico LB", "p-value del estadístico LB", "Estadístico BP",
                    "p-value del estadístico BP"]
         pd.Series(ResultadosTest, index = Nombres)

Out[29]: Estadístico LB                [0.7549153999366602, 1.080970233422208]
         p-value del estadístico LB    [0.3849244343516546, 0.5824656200187786]
         Estadístico BP                [0.7206010635759029, 1.0268949980623265]
         p-value del estadístico BP    [0.3959468201677663, 0.5984289353087118]
         dtype: object
```

El lector debe notar que la función `acorr_ljungbox(...)` de *statsmodels.stats* genera p valores para el estadístico y *p-value* correspondientes, los cuales presenta dentro de una lista Python "[...]", en cuanto ejecuta la prueba respectiva para cada uno de los rezagos incluidos.

- **Nota:** Este test equivale al generado por la función `Box.test(...)` de **R** (paquete *stats*). A diferencia de Python, en R la prueba que se ejecuta por defecto es la de Box-Pierce, por lo que si se quiere llevar a cabo el Test Ljung-Box, tal preferencia debe señalarse explícitamente en el parámetro *type* de la función.

Supuesto de Normalidad

Uno de los supuestos sobre el término de error es que tiene distribución de probabilidad normal. Es posible que el lector cuidadoso haya notado que este supuesto no se incluyó en el listado de supuestos del modelo clásico de regresión lineal, presentado en la primera parte de este capítulo. Esto se debe a que el supuesto de normalidad en la distribución del término de error no hace parte del modelo clásico de regresión lineal, por lo que no es necesario para garantizar las propiedades de los estimadores de MCO de linealidad, insesgadez y varianza mínima. Entonces, ¿por qué se menciona?

El supuesto de normalidad en la distribución del término de error hace parte del modelo clásico de regresión lineal **normal**, y ve justificada su existencia por la necesidad de llevar a cabo inferencia estadística. Mientras los demás supuestos presentados a inicio de este capítulo son requeridos para que los estimadores de MCO sean MELI (o BLUE en inglés), tales supuestos no son suficientes para garantizar que la inferencia estadística sea válida.

> Debido a que el método de MCO no hace ninguna suposición respecto de la naturaleza probabilística de u_i, resulta de poca ayuda para el propósito de hacer inferencias sobre la FRP mediante la FRM, a pesar del teorema de Gauss-Markov. Este vacío puede llenarse si se supone que las u siguen una determinada distribución de probabilidad. Por razones que mencionaremos en seguida, en el contexto de regresión se supone, por lo general, que las u tienen la distribución de probabilidad normal. (Gujarati y Porter, 2010, p. 98)

El supuesto indica que el término de error tiene distribución normal con media cero y varianza constante y que, además, la covarianza entre u_i y u_j $(i \neq j)$ es cero. Así, el supuesto puede resumirse en que $u_i \sim NID(0, \sigma^2)$, en otras palabras: el término de error es normal e independientemente distribuido.

Una de las pruebas estadísticas más populares empleadas para verificar el cumplimiento de normalidad en la distribución del término de error es el Test **Jarque-Bera**, el cual se basa en la asimetría y la curtosis (Würtz y Katzgraber, 2009) y toma como hipótesis nula, básicamente, distribución normal.

La función empleada en Python para el Test Jarque-Bera para normalidad es `jarque_bera(...)` de *statsmodels.stats*; sus argumentos incluyen los residuos del modelo.

Siguiendo la misma estructura de las otras pruebas estadísticas realizadas, y teniendo en cuenta que los residuos ya se asignaron a un objeto específico, el código a ejecutar para evaluar normalidad en la distribución del término de error es el siguiente:

```
In [30]: ResultadosTest = sms.jarque_bera(Residuos)
         Nombres = ["Jarque-Bera", "p-value", "Asimetría", "Curtosis"]
         pd.Series(ResultadosTest, index = Nombres)

Out[30]: Jarque-Bera    0.559405
         p-value        0.756009
         Asimetría      0.227575
         Curtosis       2.948855
         dtype: float64
```

Asumiendo un nivel de significancia de 5 %, como $p - value > \alpha$ entonces no se rechaza la hipótesis nula y se concluye que se presenta normalidad en la distribución del término de error (trabajando con un α de 0.05).

- **Nota:** Este test equivale al generado por la función `jarque.bera.test(...)` de **R** (paquete *tseries*). El usuario debe tener en cuenta que el argumento empleado en la función de R, al igual que la de Python, corresponde a los residuos del modelo.

Otras pruebas ampliamente utilizadas para evaluar normalidad son la de **Anderson-Darling**, la cual pertenece a la clase cuadrática de los estadísticos basados en Función de Distribución Empírica, y la de **Kolmogorov-Smirnov**, también construida a partir de dicha función (Razali y Wah, 2011). La primera prueba es implementada en Python con la función `normal_ad(...)` de *statsmodels.stats*:

```
In [31]: ResultadosTest = sms.normal_ad(Residuos)
         Nombres = ["Anderson-Darling", "p-value"]
         pd.Series(ResultadosTest, index = Nombres)
```

```
C:\Users\FCE\Anaconda3\lib\site-packages\statsmodels\stats\_adnorm.py:66:
FutureWarning: Using a non-tuple sequence for multidimensional indexing
is deprecated; use `arr[tuple(seq)]` instead of `arr[seq]`.
In the future this will be interpreted as an array index, `arr[np.array(seq)]`,
which will result either in an error or a different result.
  S = np.sum((2*i[sl1]-1.0)/N*(np.log(z)+np.log(1-z[sl2])), axis=axis)
```

```
Out[31]: Anderson-Darling    0.329856
         p-value             0.508773
         dtype: float64
```

Reforzando la conclusión del Test Jarque Bera, el resultado de Anderson-Darling sugiere que la distribución de los residuos es normal.

Respecto a la otra prueba, la de **Kolmogorov-Smirnov**, esta es implementada con la función `kstest_normal(...)` del módulo `diagnostic` de *statsmodels.stats*. La estructura a emplear para presentar los resultados es la misma utilizada en las demás pruebas estadísticas que se han realizado:

```
In [32]: ResultadosTest = sms.diagnostic.kstest_normal(Residuos)
         Nombres = ["Estadístico KS", "p-value"]
         pd.Series(ResultadosTest, index = Nombres)
```

```
Out[32]: Estadístico KS    0.081275
         p-value           0.200000
         dtype: float64
```

Al igual que con Jarque-Bera y Anderson-Darling, los resultados del test de **Kolmogorov-Smirnov** indican que no se rechaza la hipótesis nula y, por tanto, no se rechaza que la distribución del término de error sea normal.

Las dos últimas pruebas realizadas se basan en la **Función de Distribución Empírica**, la cual, de acuerdo a Vaart (1998) es el estimador natural de la distribución subyacente, F, si esta no es conocida. Para visualizar la distribución empírica en Python podemos hacer uso de funciones de *Matplotlib* y para obtener los valores correspondientes a dicha función basta con hacer unas cuantas operaciones sencillas.

Vaart (1998) define, para una muestra aleatoria $X_1, ..., X_n$, la Función de Distribución Empírica de la siguiente manera:

$$F_n(t) = \frac{1}{n}\sum_{i=1}^{n} 1\{X_i \leq t\}$$

A partir de esta definición es muy simple determinar los puntos con los que se calculará la Función de Distribución Empírica. En primer lugar, debemos organizar los valores de la variable de interés de menor a mayor, para lo que se utiliza la función `sort(...)` de *NumPy*:

```
In [33]: x = np.sort(Residuos)
```

A cada uno de estos valores x corresponde una "probabilidad" de ser menor o igual; es decir, si tenemos el conjunto {1,2,2,3,5,7,8,9,10,10} la probabilidad de que un elemento sea menor o igual a 2 es 30 % (pues tres de los diez elementos son menores o iguales a dos), la probabilidad de que un elemento sea menor o igual a 7 es 60 % (pues seis de los diez elementos son menores o iguales a siete). Esta lógica es muy fácil de implementar en el código: tan solo se debe determinar el número de elementos, n, y dividir una secuencia $1, ..., n$ entre este número, de modo que se obtengan las probabilidades correspondientes:

```
In [34]: n = x.size
         y = np.arange(1, n+1) / n
```

Ya se tienen los pares (x, y) para graficar la Función de Distribución Empírica, pero ésta solo nos resulta útil si podemos compararla con algún referente: una distribución **teórica**. La distribución normal se construye con dos parámetros: la media y la desviación estándar. Si se generan datos a partir de una distribución normal con media igual al promedio de los residuos y desviación estándar igual a la de estos, dichos datos sirven para graficar la distribución téorica (pues se sabe que efectivamente se trata de una distribución normal), la cual puede usarse para comparar con la distribución empírica; si el comportamiento es muy similar, visualmente se sugiere que la distribución de los residuos es normal.

Veamos lo anteriormente descrito en la práctica. Para generar valores aleatorios provenientes de una distribución normal se puede usar la función `normal(...)` del módulo `random` de *NumPy*. A esta función se debe especificar la media (*loc*), la desviación estándar (*scale*) y el número de valores a generar (*size*):

```
In [35]: Residuos_teoricos = np.random.normal(loc = np.mean(Residuos),
                                               scale = np.std(Residuos),
                                               size = 10000)
```

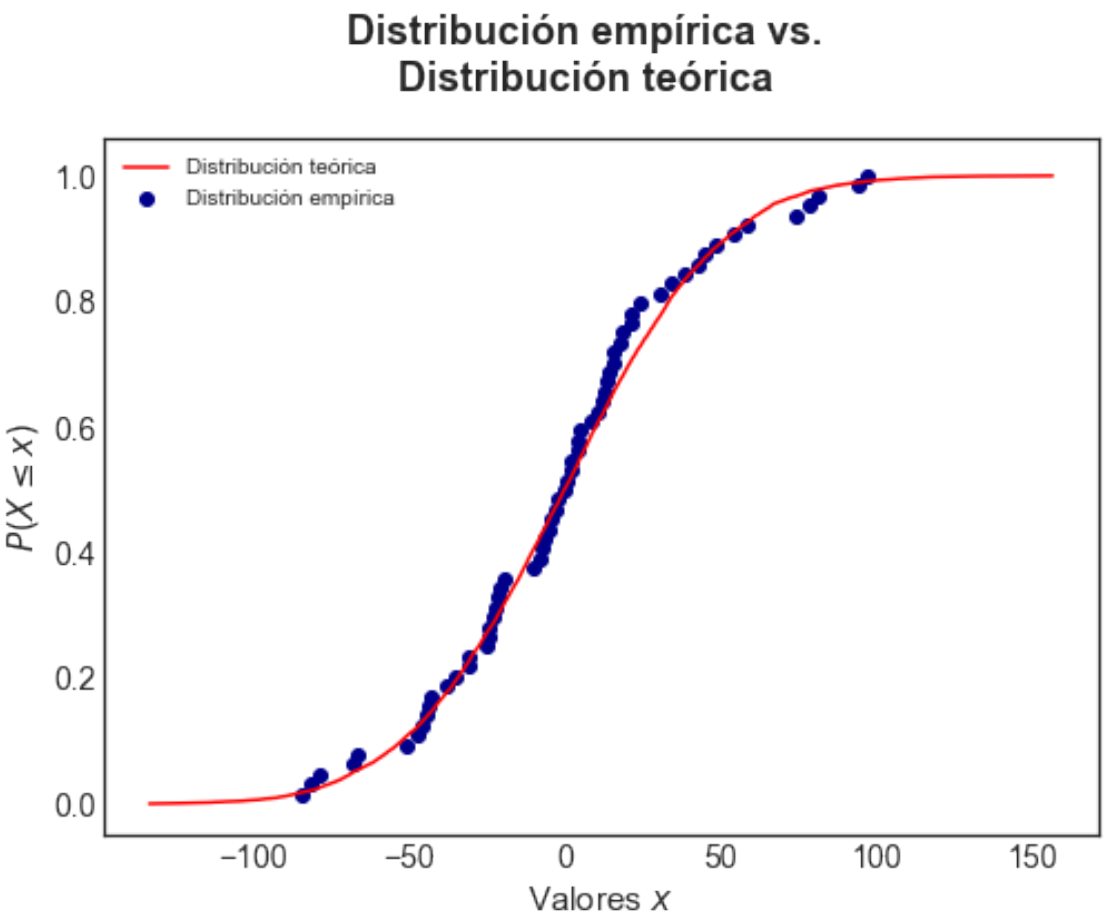
A partir de estos residuos teóricos puede definirse la función de distribución "teórica", para lo que basta emplear las mismas operaciones usadas en la distribución empírica:

```
In [36]: x_teor = np.sort(Residuos_teoricos)
         n_teor = x_teor.size
         y_teor = np.arange(1, n_teor+1) / n_teor
```

Finalmente, con la función `subplots(...)` de *matplotlib.pyplot* y los métodos `.scatter()` y `.plot()`, puede visualizarse la relación entre las funciones previamente definidas:

```
In [37]: fig, ax = plt.subplots(figsize = (8,6))
         fig.suptitle("Distribución empírica vs.\n"
                      "Distribución teórica",
                      fontsize = 18,
                      fontweight = "bold")
         ax.scatter(x, y, color = "darkblue",
                    label = "Distribución empírica")
         ax.plot(x_teor, y_teor, color = "red",
                 label = "Distribución teórica")
         ax.set_xlabel(r"Valores $x$",
                       fontsize = 15)
         ax.set_ylabel("$P(X\leq x)$", fontsize = 15)
         ax.tick_params(labelsize = 14)
         ax.legend()
         fig.text(.9,-.02,
                  "Elaboración:",
                  fontsize = 13, fontweight = "bold",
                  ha = "right")
         fig.text(.9,-.08,
                  "Triana, F.\n(2019)",
                  fontsize = 12, ha = "right")
```

```
plt.subplots_adjust(top = 0.85)
plt.show()
```



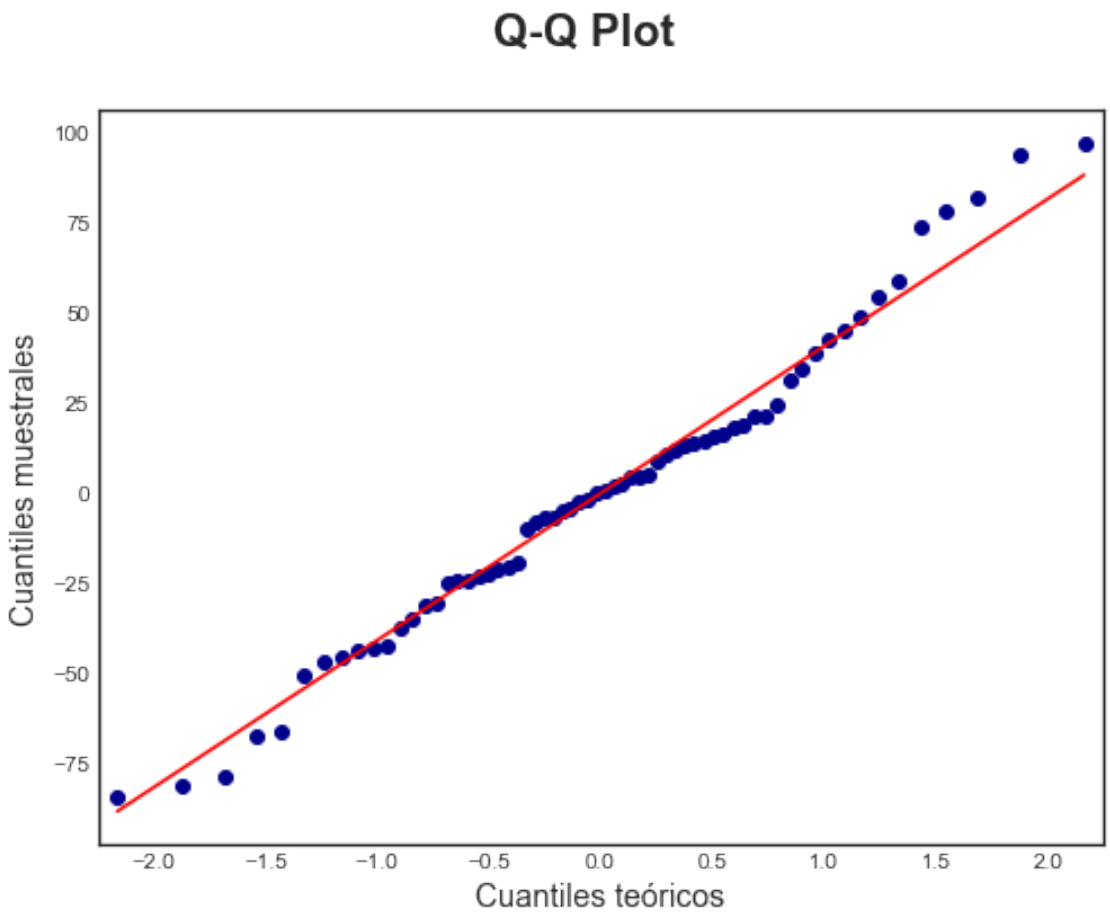
Elaboración:
Triana, F.
(2019)

El lector puede observar que la distribución de los residuos no dista mucho de la apariencia que corresponde a la distribución normal. Los puntos correspondientes a la función de distribución empírica (los azules) se sitúan muy cerca de la línea de referencia (la distribución teórica).

Finalmente, otro método gráfico ampliamente utilizado, seguramente más común que el de la distribución empírica, para evaluar la normalidad, es el gráfico de cuantil-cuantil, o *Q-Q Plot*. Este gráfico puede construirse con la función `qqplot(...)` de *Statsmodels*, pasando como argumento el objeto que contiene los datos a partir de los cuales se quieren establecer los cuantiles. Por defecto esta función trabaja con la distribución normal, razón por la cual no debe introducirse alguna modificación:

```
In [38]: fig, ax = plt.subplots(figsize = (8,6))
         fig.suptitle("Q-Q Plot", size = 20,
                      fontweight = "bold")
         sm.qqplot(Residuos, ax = ax, line = "s",
```

```
          color = "darkblue")
ax.set_xlabel("Cuantiles teóricos", fontsize = 14)
ax.set_ylabel("Cuantiles muestrales", fontsize = 14)
fig.text(.9,-.02,
         "Elaboración:",
         fontsize = 13, fontweight = "bold",
         ha = "right")
fig.text(.9,-.08,
         "Triana, F.\n(2019)",
         fontsize = 12, ha = "right")
plt.show()
```



Como se observa, la mayoría de puntos se sitúa muy cerca de la línea roja de referencia, indicador de que la distribución de los datos no dista mucho de la teórica (la normal). Este resultado concuerda con el obtenido en el análisis de la distribución empírica y con los resultados de las tres pruebas estadísticas empleadas para verificar el cumplimiento del supuesto de normalidad.

En este capítulo se abordó muy brevemente, y sin examinar detalladamente los fundamentos teóricos asociados, el cumplimiento de algunos de los supuestos del modelo clásico de regresión lineal (normal, si consideramos el último supuesto planteado) usando herramientas de librerías Python especializadas. El modelo al que corresponden tales supuestos se ilustró, por medio de un ejemplo particular, en el Capítulo 3, en el cual se dio una ligera aproximación (práctica) al tema fundamental que generalmente se aborda en los cursos de nivel introductorio de Econometría. Ahora, se aumentará ligeramente la dificultad procediendo a examinar un tema un tanto más complejo (en un sentido más teórico que práctico) al que se dedicará el próximo capítulo.

Capítulo 5

Mínimos Cuadrados en 2 Etapas

Uno de los supuestos (el #2 específicamente) del modelo clásico de regresión lineal planteados a inicios del capítulo previo, establece que la covarianza entre las variables explicativas y el término de error es cero, es decir, que toda x (siguiendo la notación empleada con anterioridad) es independiente del término de error. Cuando las variables explicativas cumplen esta condición, son denominadas **exógenas** y cuando violan tal supuesto se consideran **endógenas**.

Cuando alguna(s) de las variables explicativas del modelo es (son) endógena(s) los estimadores obtenidos por el método de Mínimos Cuadrados Ordinarios pierden su calidad de MELI. Para solucionar tal inconveniente se recurre al método de **Variables Instrumentales**, en el cual se emplean variables no incluidas en la ecuación original para estimar las variables explicativas endógenas incluidas en ésta y tratar su endogeneidad.

Cuando alguna(s) variable(s) explicativa(s) y el término de error se correlacionan, se violan condiciones de los supuestos del modelo clásico de regresión lineal, por lo que los estimadores de MCO pierden algunas de sus características atractivas. Para solucionar el inconveniente generado por la endogeneidad de algún(os) covariante(s) se requiere de información adicional, la cual se obtiene de una variable observable exógena no incluida.

Tomando un modelo de regresión lineal de la forma

$$y_{1i} = \beta_0 + \beta_1 x_i + \beta_2 y_{2i} + u_i$$

en donde y_1 es la variable dependiente, x es una variable explicativa exógena (no se correlaciona con el término de error) y y_2 es una variable explicativa endógena (se correlaciona con el término de error), si se tiene una variable z que cumple con las condiciones siguientes:

1. $cov(z, u) = 0$
2. $cov(z, y_2) \neq 0$

entonces z es una variable instrumental (o instrumento) para y_2.

La primera condición establecida hace referencia a la exogeneidad del instrumento y la segunda a la relevancia del mismo; dichas condiciones son útiles para garantizar la "calidad" de las Variables Instrumentales, las cuales son importantes ya que permiten tratar la endogeneidad y superar el inconveniente generado por ésta. Al lector interesado en profundizar sobre este tema se sugiere consultar el Capítulo 15, *Estimación con variables instrumentales y mínimos cuadrados en dos etapas*, de Wooldridge (2010).

Hasta este punto, el lector debe saber que la correlación entre variables explicativas y el término de error es un problema para la obtención de estimadores con las mejores propiedades por el método de Mínimos Cuadrados Ordinarios y que las variables instrumentales pueden emplearse para tratar la endogeneidad de la(s)

variable(s) explicativa(s); sin embargo, aún no se ha señalado el proceso específico por medio del cual se hace uso de los instrumentos.

Uno de los métodos empleados para obtener estimadores de Variables Instrumentales es el de **Mínimos Cuadrados en 2 Etapas (MC2E)**, el cual recibe tal denominación dado que el proceso que comprende está estructurado en dos "etapas" bien definidas en las que se llevan a cabo regresiones específicas con objetivos concretos.

Tomando el modelo de regresión lineal señalado anteriormente, y siguiendo a Wooldridge (2010), la primera etapa del método MC2E consiste en realizar la regresión $\hat{y}_{2i} = \hat{\pi}_0 + \hat{\pi}_1 x_i + \hat{\pi}_2 z_i$, de donde se obtienen los valores ajustados $\hat{y}_2$. La segunda etapa es una regresión por Mínimos Cuadrados Ordinarios de $y_{1i} = \beta_0 + \beta_1 x_i + \beta_2 \hat{y}_{2i} + u_i$ (nótese que en esta regresión se emplea los valores de $\hat{y}_2$ en vez de los valores de y_2). Así, de lo que se trata, básicamente, es de estimar la variable explicativa endógena a partir de variables exógenas y, posteriormente, emplear la variable estimada como variable explicativa, junto a las demás exógenas del modelo original (no se incluyen los instrumentos), para estimar la variable dependiente.

Ya se ha dado una muy mínima descripción de Variables Instrumentales y el método de Mínimos Cuadrados en 2 Etapas, por lo que el lector debe tener una idea general del tema que ahora se pretende abordar de manera práctica; labor a la cual, a continuación, se da inicio efectivo con Python.

El dataset que se utilizará para ilustrar el uso de variables instrumentales y el método MC2E es el empleado en el Ejemplo 15.5, *Rendimientos de la educación para la mujer trabajadora*, de Wooldridge (2010). El dataset se ha obtenido desde Stata, con el comando `bcuse`, y ha sido exportado como archivo *.csv* para su uso en Python. Se invita al lector a consultar el ejemplo, de modo que pueda verificar los resultados obtenidos.

Preparación del entorno

La primera celda en la que el usuario escribirá y ejecutará código tendrá el siguiente contenido:

```
In [1]: import numpy as np
        import pandas as pd
        import statsmodels.api as sm
        from linearmodels.iv import IV2SLS
        import matplotlib.pyplot as plt
        %matplotlib inline
        plt.style.use("seaborn-white")
```

El lector ya debe saber que el anterior bloque de código permite cargar las herramientas necesarias para llevar a cabo las acciones requeridas en el análisis econométrico planteado; en caso de que no tenga claridad al respecto, se le pide consultar la sección 'Preparación del entorno' del segundo capítulo, en donde se brinda una explicación sobre tal cuestión.

El código de esta celda es exactamente igual al empleado en la preparación del entorno del Capítulo 2, exceptuando por lo siguiente: 1) no se importa la librería *Seaborn* y, 2) más importante aún, se importa la función `IV2SLS(...)` del módulo `iv` de la librería *linearmodels*; tal función es fundamental para el tema a abordar en este capítulo y su importación es **obligatoria**.

El usuario debe ser muy **cuidadoso** al ejecutar la celda de código anterior, pues si lo hace inmediatamente, sin tener en cuenta la información que se presenta a continuación, el proceso no será completamente exitoso; esto, por la razón que enseguida se presenta. La librería *linearmodels* **no** viene integrada por defecto en la **Distribución Anaconda** (la cual es la que estamos empleando), por lo cual es necesario instalarla por nuestra propia cuenta para poder hacer uso de las herramientas que brinda.

Para instalar una librería basta con dirigirse al Símbolo del Sistema ('cmd'); estando ahí, tan solo se debe escribir `conda install` y, a continuación, el nombre de la librería en la que se está interesado. En nuestro

caso, el código a emplear es `conda install linearmodels`. De forma alternativa puede emplearse `!conda install linearmodels` directamente en el *notebook*.

- **Nota:** El código anterior hace uso de `conda`, el gestor de paquetes propio de la distribución Anaconda, el cual recurre al repositorio Anaconda; sin embargo, es posible que el proceso de instalación de la librería a través de `conda` no resulte exitoso, en cuyo caso se puede utilizar el sistema de gestión de paquetes estándar de Python, `pip`, el cual trabaja directamente con Python Package Index (PyPi). El código para la instalación de *linearmodels* a través de `pip` es `pip install linearmodels` en 'cmd' o `!pip install linearmodels` en el *notebook*; ejecutando dicho código debería llevarse a cabo de forma exitosa la instalación de la librería *linearmodels*.

La importación de *linearmodels* es fundamental para nuestra labor, pues es donde están contenidas las funciones que emplearemos para hacer uso de variables instrumentales y aplicar el método de Mínimos Cuadrados en 2 Etapas; sin esta librería resultaría excesivamente complejo el proceso de estimación y la realización de las labores requeridas para llevar a cabo un análisis econométrico "aceptable". El usuario debe verificar la instalación previa de *linearmodels* (en caso de no estar instalada, llevar a cabo el proceso de instalación correspondiente, sea con `conda` o con `pip`) y, en caso de reutilizar el bloque de código de preparación del entorno del segundo capítulo, debe ser cuidadoso y asegurarse de incluir correctamente la línea de importación de la función `IV2SLS(...)` del módulo `iv` de *linearmodels*.

Una vez ejecutada la celda de código de la preparación del entorno, sin la generación de algún tipo de error, puede continuarse con entera tranquilidad a la fase siguiente, la cual tiene una importancia fundamental en cuanto es la que permite contar con la "materia prima" para llevar a cabo el análisis: los datos.

Importación de los datos

La importación de los datos se lleva a cabo con funciones de la librería *pandas*. Dependiendo del tipo de archivo en el cual se encuentre almacenada la información del dataset, deberemos recurrir a una función particular y un conjunto de parámetros específico.

Para nuestro caso concreto, el siguiente código es el que importa el dataset:

```
In [2]: data = pd.read_csv("MROZ.csv")
```

Al lector que no entienda esta línea de código, se le sugiere dirigirse al Capítulo 2, en donde encontrará una explicación que seguramente aclarará sus dudas.

Una vez ejecutada la línea de código de esta celda, el dataset debería haber sido importado correctamente. Para comprobarlo, recurrimos al método `.head()`, aplicado al objeto que contiene el dataset (en este caso le hemos asignado el nombre `data`):

```
In [3]: data.head()
```

```
Out[3]:    inlf  hours  kidslt6  kidsge6  age  educ    wage  repwage  hushrs  husage  \
        0     1   1610        1        0   32    12  3.3540     2.65    2708      34
        1     1   1656        0        2   30    12  1.3889     2.65    2310      30
        2     1   1980        1        3   35    12  4.5455     4.04    3072      40
        3     1    456        0        3   34    12  1.0965     3.25    1920      53
        4     1   1568        1        2   31    14  4.5918     3.60    2000      32

           …     faminc     mtr  motheduc  fatheduc  unem  city  exper   nwifeinc  \
```

```
0   …      16310  0.7215        12         7    5.0     0      14  10.910060
1   …      21800  0.6615         7         7   11.0     1       5  19.499981
2   …      21040  0.6915        12         7    5.0     0      15  12.039910
3   …       7300  0.7815         7         7    5.0     0       6   6.799996
4   …      27300  0.6215        12        14    9.5     1       7  20.100060

      lwage  expersq
0  1.210154      196
1  0.328512       25
2  1.514138      225
3  0.092123       36
4  1.524272       49

[5 rows x 22 columns]
```

Podemos observar que el proceso de importación ha sido exitoso y la información del dataset ha sido almacenada correctamente en un `DataFrame` de *pandas*. Ahora, podemos continuar tranquilamente con el desarrollo de nuestro análisis, en cuanto hemos concluido satisfactoriamente el primer paso. Procederemos a usar variables instrumentales empleando el método de Mínimos Cuadrados en 2 Etapas, sin embargo, el usuario debe tener en cuenta la información que se le brinda a continuación.

El dataset que acabamos de importar contiene algunas observaciones faltantes (esto se debe a la fuente original de la información y no a una preferencia del autor de esta obra, aunque veremos que tal situación nos resultará algo enriquecedora al mejorar nuestra capacidad exploratoria y de manejo de datos). Estos valores faltantes pueden generar problemas posteriormente, por lo que procederemos a identificarlos y darles un tratamiento apropiado.

Para examinar cuántas filas presentan algún campo con un valor faltante y conocer su ubicación, emplearemos los métodos `.isna()` y `.sum()`; el primero nos permite identificar valores faltantes (por medio de un criterio booleano) y el segundo permite ejecutar una suma que, teniendo en cuenta el tipo de datos generados por el primer método, permite conocer la cantidad de los mismos.

```
In [4]: data.isna().sum()

Out[4]: inlf          0
        hours         0
        kidslt6       0
        kidsge6       0
        age           0
        educ          0
        wage        325
        repwage       0
        hushrs        0
        husage        0
        huseduc       0
        huswage       0
        faminc        0
        mtr           0
        motheduc      0
        fatheduc      0
```

```
unem          0
city          0
exper         0
nwifeinc      0
lwage       325
expersq       0
dtype: int64
```

Se puede observar que las únicas variables en las que se presentan valores faltantes son 'wage' y 'lwage', con 325 datos clasificados como `Na` para cada una. Esta situación puede dar lugar a ciertos problemas, por lo cual, tratando de minimizar la posibilidad de inconvenientes durante la fase de estimación, solo se empleará los registros en los cuales existe un valor específico (no `Na`) para la variable 'lwage' (la cual tiene el rol de variable dependiente en el modelo). ¿Cómo podremos conseguir tal objetivo, seleccionar únicamente registros con valores no faltantes?

Para obtener un `DataFrame` que solamente contenga registros en los que no haya valores faltantes para la variable 'lwage' se hará uso de la selección por medio de corchetes "[ ]" y del método `.notna()` (aplicado sobre la variable de referencia); esto, con el propósito de tener en cuenta únicamente registros que cumplan la condición señalada. El resultado obtenido se asignará al objeto `data` (que ya existe), con el objetivo de contar con un único dataset y evitar posibles confusiones. El código a ejecutar es el siguiente:

```
In [5]: data = data[data["lwage"].notna()]
```

Ahora, procedemos a verificar que hemos conseguido lo que nos hemos propuesto y en nuestro `DataFrame` no se presentan valores faltantes; de nuevo, emplearemos los métodos `.isna()` y `.sum()`:

```
In [6]: data.isna().sum()
```

```
Out[6]: inlf        0
        hours       0
        kidslt6     0
        kidsge6     0
        age         0
        educ        0
        wage        0
        repwage     0
        hushrs      0
        husage      0
        huseduc     0
        huswage     0
        faminc      0
        mtr         0
        motheduc    0
        fatheduc    0
        unem        0
        city        0
        exper       0
        nwifeinc    0
        lwage       0
        expersq     0
        dtype: int64
```

Observamos que todas las variables contenidas en el `DataFrame` carecen de valores faltantes, por lo que ningún registro está incompleto. Este `DataFrame`, que tiene el nombre de `data`, es el que emplearemos para trabajar con variables instrumentales y aplicar el método de Mínimos Cuadrados en 2 Etapas, el cual, abordaremos a continuación desde dos enfoques: 1) proceso manual y 2) proceso automático.

Proceso manual

El método de Mínimos Cuadrados en 2 Etapas, tal y como su nombre lo indica, está formado por dos etapas de estimación claramente definidas: la primera consiste en regresar la(s) variable(s) explicativa(s) endógena(s) contra el(los) instrumentos y la(s) variable(s) explicativa(s) exógena(s); la segunda etapa consiste en emplear la variable estimada en la primera etapa en lugar de la variable explicativa endógena en el modelo original. Al lector que tal descripción le parezca confusa, se sugiere remitirse a la primera parte de este capítulo o consultar el Capítulo 15, *Estimación con variables instrumentales y mínimos cuadrados en dos etapas*, de Wooldridge (2010), donde este tema es tratado en detalle.

El proceso "manual" recibe tal denominación en cuanto debemos llevar a cabo las regresiones de la primera y la segunda etapa por nuestra propia cuenta, especificando el conjunto de variables a emplear en cada una de estas. Tal procedimiento no es recomendable y la mayoría de programas y lenguajes que soportan análisis estadístico (Python no es la excepción) cuenta con funciones que realizan la ejecución "automáticamente", evitando que el usuario deba llevar a cabo las tareas por su cuenta.

A pesar de lo señalado en el párrafo anterior, dado el enfoque práctico de esta guía y la utilidad que puede tener para el usuario el llevar a cabo el proceso paso por paso en la comprensión del tema tratado, procederemos a aplicar el método MC2E 'manualmente' (el proceso automático se aborda en la siguiente sección, por lo que al lector que no esté interesado en ejecutar ambas etapas por su propia cuenta se le sugiere consultar dicha sección directamente, sin dedicar su tiempo al estudio de la presente).

Lo primero que debemos hacer para llevar a cabo correctamente las regresiones de la primera etapa y la segunda etapa es agrupar las variables disponibles de forma apropiada, en cuanto cada una de estas desempeña un rol específico. Así, procederemos a asignar la variable dependiente a un objeto específico, la variable explicativa endógena a otro objeto en particular y, del mismo modo, las variables explicativas exógenas y los instrumentos a otros objetos concretos. El código a ejecutar es el siguiente:

```
In [7]: Y1 = data["lwage"] # Variable dependiente
        Y2 = data["educ"] # Variable endógena
        X = data[["exper", "expersq"]] # Variables exógenas
        Z = data[["motheduc", "fatheduc"]] # Instrumentos
```

Primera etapa

Durante la primera etapa se regresa la variable explicativa endógena contra las variables explicativas exógenas y los instrumentos. El lector puede observar que las variables explicativas exógenas y los instrumentos han sido asignados a objetos distintos, por lo que, con el propósito de simplificar el código, ahora se asignarán conjuntamente a un único objeto. Para conseguir tal objetivo, basta con emplear la función `concat(...)` de *pandas*, usando como argumento una lista Python "[...]" en la que se incluyan los `DataFrames` que contienen las variables explicativas exógenas y los instrumentos, y asignando el valor de 1 al parámetro *axis* (para concatenar los `DataFrames` "horizontalmente"). El resultado generado por la aplicación de la función sobre la lista recibirá el nombre (algo extenso, pero muy informativo) de `RegresorasPrimeraEtapa`, así:

```
In [8]: frames = [X, Z]
        RegresorasPrimeraEtapa = pd.concat(frames, axis = 1)
```

Verificamos que el proceso se ha llevado a cabo exitosamente, recurriendo al método `.head()`:

```
In [9]: RegresorasPrimeraEtapa.head()
```

Out[9]:

	exper	expersq	motheduc	fatheduc
0	14	196	12	7
1	5	25	7	7
2	15	225	12	7
3	6	36	7	7
4	7	49	12	14

Ahora, dado que nuestras variables están almacenadas de forma conveniente, construimos el modelo correspondiente a la primera etapa, tomando como variable dependiente la variable explicativa endógena ('educ' en notación original, `Y2` en nuestro entorno) y como variables explicativas las variables explicativas exógenas ('exper', 'expersq' en notación original) y los instrumentos ('motheduc', 'fatheduc' en notación original). Nótese que las variables explicativas exógenas y los instrumentos están contenidos en un único objeto (`RegresorasPrimeraEtapa`).

El código a ejecutar el siguiente:

```
In [10]: PrimeraEtapa = sm.OLS(Y2, sm.add_constant(RegresorasPrimeraEtapa))
         ResultadosPrimeraEtapa = PrimeraEtapa.fit()
         print(ResultadosPrimeraEtapa.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   educ   R-squared:                       0.211
Model:                            OLS   Adj. R-squared:                  0.204
Method:                 Least Squares   F-statistic:                     28.36
Date:                xxx, xx xxx xxxx   Prob (F-statistic):           6.87e-21
Time:                        xx:xx:xx   Log-Likelihood:                -909.72
No. Observations:                 428   AIC:                             1829.
Df Residuals:                     423   BIC:                             1850.
Df Model:                           4
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          9.1026      0.427     21.340      0.000       8.264       9.941
exper          0.0452      0.040      1.124      0.262      -0.034       0.124
expersq       -0.0010      0.001     -0.839      0.402      -0.003       0.001
motheduc       0.1576      0.036      4.391      0.000       0.087       0.228
fatheduc       0.1895      0.034      5.615      0.000       0.123       0.256
==============================================================================
Omnibus:                       10.903   Durbin-Watson:                   1.940
Prob(Omnibus):                  0.004   Jarque-Bera (JB):               20.371
Skew:                          -0.013   Prob(JB):                     3.77e-05
Kurtosis:                       4.068   Cond. No.                     1.55e+03
==============================================================================
```

```
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.55e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Al lector que ha seguido atentamente esta obra, la anterior celda de código no debería resultarle extraña, pues no es más que una regresión por Mínimos Cuadrados Ordinarios; si el código presentado le resulta confuso, se le sugiere consultar el tercer capítulo de este trabajo, en donde la regresión por MCO es el tema central.

Hemos empleado instrumentos como covariantes en una regresión, sin embargo, aún desconocemos si dichos instrumentos cumplen las condiciones para desempeñar satisfactoriamente tal papel; todavía no hemos evaluado la "calidad" de las variables instrumentales utilizadas, por lo que procederemos a examinarla.

Para que las variables instrumentales empleadas cumplan la condición de relevancia (véase la primera parte de este capítulo), los parámetros asociados a los instrumentos deben ser estadísticamente distintos de cero. Como puede observarse en la instancia de resultados del modelo, tanto para la variable 'motheduc' como para la variable 'fatheduc' se cumple dicha condición, pues los coeficientes asociados son estadísticamente significativos, el *p-value* respectivo es inferior a 0.05 (nivel de significancia elegido) y el intervalo de confianza no contiene el 0.

La conclusión del párrafo anterior es que los instrumentos cumplen la condición de relevancia, sin embargo, para probar formalmente la validez de dicha conclusión puede emplearse una Prueba F para significancia de un subconjunto de parámetros. La Prueba F se ejecuta en Python por medio del método `.f_test()`, aplicado sobre la instancia de resultados del modelo y empleando como argumento las condiciones a evaluar (que pueden expresarse como un 'string').

En este caso, asignaremos la hipótesis a un objeto que se empleará como argumento en el método `.f_test()`; esto, con el propósito de simplificar el código. Así:

```
In [11]: hipotesis = "(fatheduc = 0), (motheduc = 0)"
         Prueba_F = ResultadosPrimeraEtapa.f_test(hipotesis)
         print(Prueba_F)

<F test: F=array([[55.40030043]]), p=4.268908724630835e-22, df_denom=423, df_num=2>
```

El resultado generado por el método `.f_test()` es un objeto de tipo `ContrastResults()`, el cual cuenta con sus propios atributos. Algunos de estos atributos corresponden al estadístico y al *p-value*, los cuales son los valores de interés para nuestros propósitos. Estos atributos son objetos de tipo `ndarray` de *NumPy*, sin embargo, lo que nos interesa no es el objeto como tal sino su contenido, el cual debe expresarse como un dato de tipo `float`.

Para realizar la extracción de los valores correspondientes, se recurre al uso de la selección por medio de corchetes "[]" y del método `.item()`, indicando la posición correspondiente al elemento de interés; asimismo, emplearemos una `Series` de *pandas* para obtener una presentación un tanto más "agradable" del resultado. El código a ejecutar es el siguiente:

```
In [12]: Estadístico = Prueba_F.statistic[0].item(0)
         pvalue = Prueba_F.pvalue.item(0)
         Nombres = ["Estadístico", "p-value"]
         pd.Series([Estadístico, round(pvalue, 3)], index = Nombres)
```

```
Out[12]: Estadístico    55.4003
         p-value         0.0000
         dtype: float64
```

Asumiendo un nivel de significancia de 5 %, como $p - value < \alpha$ entonces se rechaza la hipótesis nula y se concluye que los parámetros asociados a los instrumentos son estadísticamente distintos de cero (trabajando con un α de 0.05). Se invita al lector a contrastar estos resultados con los presentados en Wooldridge (2010), de modo que pueda comprobar que son exactamente iguales y que el proceso desarrollado es correcto.

Hasta este punto ya se ha llevado a cabo la regresión de la primera etapa y se ha verificado que los instrumentos empleados cumplen la condición de relevancia, por lo que es posible continuar con la segunda etapa del método MC2E, la cual se aborda en la siguiente sección.

Segunda etapa

Durante la segunda etapa se regresa la variable dependiente de la ecuación original contra las variables explicativas exógenas y la variable estimada en la primera etapa (la cual reemplaza la variable explicativa endógena que inicialmente causaba los inconvenientes) (Nótese que en esta etapa no se hace uso de los instrumentos).

Durante la primera etapa estimamos la variable explicativa endógena en función de las variables explicativas exógenas y los instrumentos; sin embargo, aún desconocemos los valores de dicha variable, los cuales son los que se emplearán en la segunda etapa. Para obtener tales valores, empleamos el método `.predict()` aplicado a la instancia de resultados del modelo de la primera etapa (a la que hemos dado el conveniente nombre de `ResultadosPrimeraEtapa`).

Los resultados generados por la aplicación del método `.predict()` serán asignados a una nueva variable (que llamaremos `predicted_educ`) del `DataFram data`, de modo que podamos emplearlos con facilidad posteriormente. El código a ejecutar es el siguiente:

```
In [13]: data["predicted_educ"] = ResultadosPrimeraEtapa.predict()
```

Verificamos que la nueva variable haya sido efectivamente creada, recurriendo al método `.head()`:

```
In [14]: data.head()
```

```
Out[14]:    inlf  hours  kidslt6  kidsge6  age  educ    wage  repwage  hushrs  husage  \
         0     1   1610        1        0   32    12  3.3540     2.65    2708      34
         1     1   1656        0        2   30    12  1.3889     2.65    2310      30
         2     1   1980        1        3   35    12  4.5455     4.04    3072      40
         3     1    456        0        3   34    12  1.0965     3.25    1920      53
         4     1   1568        1        2   31    14  4.5918     3.60    2000      32

                …      mtr  motheduc  fatheduc  unem  city  exper   nwifeinc  \
         0      …   0.7215        12         7   5.0     0     14  10.910060
         1      …   0.6615         7         7  11.0     1      5  19.499981
         2      …   0.6915        12         7   5.0     0     15  12.039910
         3      …   0.7815         7         7   5.0     0      6   6.799996
         4      …   0.6215        12        14   9.5     1      7  20.100060

              lwage  expersq  predicted_educ
         0  1.210154      196       12.756017
         1  0.328512       25       11.733558
```

```
2  1.514138      225       12.771979
3  0.092123       36       11.767683
4  1.524272       49       13.914615

[5 rows x 23 columns]
```

Podemos observar que la última columna corresponde a la variable `predicted_educ`, la cual es la que hemos creado previamente; los datos que contiene corresponden a los valores estimados durante la primera etapa.

Ya contamos con todas las variables a emplear en la regresión de la segunda etapa. Ahora, al igual que durante la primera etapa, agruparemos las regresoras correspondientes en un único objeto. Nuevamente, crearemos una lista Python "[...]" cuyo contenido corresponderá a las regresoras de la segunda etapa, es decir, las variables explicativas exógenas del modelo original y la variable endógena estimada en la primera etapa; esta lista se empleará como argumento de la función `concat(...)` de *pandas* (recuerde la asignación del valor de 1 al parámetro *axis*). Así:

```
In [15]: FramesSegundaEtapa = [X, data["predicted_educ"]]
         RegresorasSegundaEtapa = pd.concat(FramesSegundaEtapa, axis = 1)
```

Verificamos que el proceso se haya llevado a cabo exitosamente, recurriendo al método `.head()`:

```
In [16]: RegresorasSegundaEtapa.head()

Out[16]:    exper  expersq  predicted_educ
         0     14      196       12.756017
         1      5       25       11.733558
         2     15      225       12.771979
         3      6       36       11.767683
         4      7       49       13.914615
```

Ahora, tal y como hemos señalado repetidamente, construimos el modelo tomando como variable dependiente la variable dependiente de la ecuación original (y_1, siguiendo la notación empleada a inicios del capítulo) y como variables explicativas las variables explicativas exógenas de la ecuación original (las x, siguiendo la notación inicial) y la variable estimada en la primera etapa (¡$\hat{y}_2$ y no y_2!).

Dado que hemos agrupado las variables explicativas de la regresión de la segunda etapa en un único objeto (es lo que hemos hecho en la última celda de código), basta con emplear dicho objeto como argumento en la construcción del modelo (la cual se lleva a cabo con la ya conocida función `OLS(...)` de *Statsmodels*).

El código (que ya debe resultar familiar) para la construcción y estimación del modelo de la segunda etapa es el siguiente:

```
In [17]: SegundaEtapa = sm.OLS(Y1, sm.add_constant(RegresorasSegundaEtapa))
         ResultadosSegundaEtapa = SegundaEtapa.fit()
         print(ResultadosSegundaEtapa.summary())

                            OLS Regression Results
==============================================================================
Dep. Variable:                  lwage   R-squared:                       0.050
Model:                            OLS   Adj. R-squared:                  0.043
Method:                 Least Squares   F-statistic:                     7.405
```

```
Date:                xxx, xx xxx xxxx   Prob (F-statistic):           7.62e-05
Time:                        xx:xx:xx   Log-Likelihood:                -457.17
No. Observations:                 428   AIC:                             922.3
Df Residuals:                     424   BIC:                             938.6
Df Model:                           3
Covariance Type:            nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
const            0.0481      0.420      0.115      0.909      -0.777       0.873
exper            0.0442      0.014      3.136      0.002       0.016       0.072
expersq         -0.0009      0.000     -2.134      0.033      -0.002   -7.11e-05
predicted_educ   0.0614      0.033      1.863      0.063      -0.003       0.126
==============================================================================
Omnibus:                       53.587   Durbin-Watson:                   1.959
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              168.354
Skew:                          -0.551   Prob(JB):                     2.77e-37
Kurtosis:                       5.868   Cond. No.                     4.41e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 4.41e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Así concluye la aplicación del método MC2E en Python de forma manual. El lector debe tener en cuenta que, aunque los parámetros estimados "manualmente" son correctos, los errores estándar no lo son (véase la página 522 de Wooldridge (2010)), por lo que no es recomendable ejecutar las regresiones de primera y segunda etapa por cuenta propia. Para evitar este inconveniente, la mayoría de paquetes y programas estadísticos ofrecen la posibilidad de ejecutar el método MC2E de forma automática, generando resultados correctos. El lenguaje de programación **Python** no es la excepción, pues su librería *linearmodels* ofrece una función diseñada específicamente para tal tarea, como veremos a continuación.

Proceso automático

Para evitar tener que hacer las regresiones de las dos etapas del método MC2E por nuestra propia cuenta, la librería *linearmodels* de Python nos da la posibilidad de hacer uso de sus funciones para realizar "automáticamente" dichas regresiones. La función específica empleada para tal propósito es `IV2SLS(...)` (acrónimo para *Instrumental Variables 2 Stages Least Squares*), cuyos argumentos incluyen la variable dependiente (*dependent*), las regresoras exógenas (*exog*), las regresoras endógenas (*endog*) y los instrumentos (*instruments*).

Para la estimación, como resulta familiar, se usa el método `.fit()`, y para el resumen de resultados el atributo `.summary`.

El código correspondiente a nuestro ejemplo es:

```
In [18]: Modelo2Etapas = IV2SLS(dependent = Y1, exog = sm.add_constant(X),
                                 endog = Y2, instruments = Z)
```

```
ResultadosModelo2Etapas = Modelo2Etapas.fit()
print(ResultadosModelo2Etapas.summary)
```

```
                          IV-2SLS Estimation Summary
==============================================================================
Dep. Variable:                   lwage   R-squared:                      0.1357
Estimator:                     IV-2SLS   Adj. R-squared:                 0.1296
No. Observations:                  428   F-statistic:                    18.611
Date:                xxx, xxx xx xxxx   P-value (F-stat)                0.0003
Time:                        xx:xx:xx   Distribution:                  chi2(3)
Cov. Estimator:                robust

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
const          0.0481     0.4278     0.1124     0.9105     -0.7903      0.8865
exper          0.0442     0.0155     2.8546     0.0043      0.0138      0.0745
expersq       -0.0009     0.0004    -2.1001     0.0357     -0.0017  -5.997e-05
educ           0.0614     0.0332     1.8503     0.0643     -0.0036      0.1264
==============================================================================

Endogenous: educ
Instruments: motheduc, fatheduc
Robust Covariance (Heteroskedastic)
Debiased: False
```

El lector puede observar cómo la aplicación del método MC2E resulta tremendamente sencilla con el uso de la función `IV2SLS(...)` de *linearmodels.iv* y que todas las tareas llevadas a cabo en el proceso manual han sido ejecutadas perfectamente en tan solo ¡3 líneas de código!, sin tener que crear variables adicionales o emplear múltiples grupos de las mismas, obteniendo, además, resultados completamente correctos.

Mínimos Cuadrados Ordinarios vs. Mínimos Cuadrados en 2 Etapas

Por último, se hará una regresión común y corriente por el método de Mínimos Cuadrados Ordinarios, sin tener en cuenta que la regresora endógena es endógena sino considerando todas las regresoras como exógenas. Esto se hará únicamente con propósitos ilustrativos, para examinar cuáles habrían sido los resultados.

Nuevamente, emplearemos la función `concat(...)` de *pandas* para agrupar las variables explicativas (exógenas y endógena) en un único `DataFrame`. Asimismo, para no extendernos innecesariamente, llevaremos a cabo la creación y estimación del modelo en la misma celda de código:

```
In [19]: FramesMCO = [X, Y2]
         RegresorasMCO = pd.concat(FramesMCO, axis = 1)
         ModeloMCO = sm.OLS(Y1, sm.add_constant(RegresorasMCO))
         ResultadosMCO = ModeloMCO.fit(cov_type = "HC0")
         print(ResultadosMCO.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  lwage   R-squared:                       0.157
Model:                            OLS   Adj. R-squared:                  0.151
Method:                 Least Squares   F-statistic:                     27.56
Date:                xxx, xx xxx xxxx   Prob (F-statistic):           2.68e-16
Time:                        xx:xx:xx   Log-Likelihood:                -431.60
No. Observations:                 428   AIC:                             871.2
Df Residuals:                     424   BIC:                             887.4
Df Model:                           3
Covariance Type:                  HC0
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.5220      0.201     -2.601      0.009      -0.915      -0.129
exper          0.0416      0.015      2.734      0.006       0.012       0.071
expersq       -0.0008      0.000     -1.940      0.052      -0.002    8.28e-06
educ           0.1075      0.013      8.170      0.000       0.082       0.133
==============================================================================
Omnibus:                       77.792   Durbin-Watson:                   1.961
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              300.917
Skew:                          -0.753   Prob(JB):                     4.54e-66
Kurtosis:                       6.822   Cond. No.                     2.21e+03
==============================================================================

Warnings:
[1] Standard Errors are heteroscedasticity robust (HC0)
[2] The condition number is large, 2.21e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

El lector puede observar que los resultados obtenidos por el método MCO difieren en cierta medida de los correspondientes al método MC2E y que los errores estándar estimados por MC2E *manualmente* son diferentes a los obtenidos *automáticamente*. Esto puede verificarlo al comparar las tablas de resumen de cada uno de los modelos elaborados; sin embargo, tal ejercicio puede resultar tedioso, en cuanto cada una de estas tablas está situada en una posición distinta dentro del *notebook* y comparar información entre estas puede resultar agotador. Afortunadamente, la librería *linearmodels* brinda una herramienta para facilitar esta labor, como veremos enseguida.

Comparación de modelos

Resulta muy práctico poder comparar los modelos estimados visualizando sus correspondientes resúmenes de instancias de resultados de forma simultánea; tal posibilidad puede materializarse gracias a la función `compare(...)` que brinda la librería *linearmodels* en su módulo `iv`. Lo primero que debemos hacer para utilizar la función `compare(...)` es contar con acceso a ésta, por lo que procedemos a importarla:

```
In [20]: from linearmodels.iv import compare
```

Habiendo ejecutado la celda de código anterior, habremos importado la función `compare(...)`, por lo que podremos hacer uso efectivo de la misma. Como argumento utilizaremos un **diccionario Python** "{...}", en el que emplearemos como claves los nombres que asignaremos a cada modelo y como valores las instancias de resultados de los modelos correspondientes.

El usuario debe tener en cuenta que para que la función `compare(...)` de *linearmodels.iv* no genere un error, todos los modelos a comparar deben corresponder a instancias de resultados del tipo generado por la función `IV2SLS(...)`. En este caso, como solo se utilizó la función `IV2SLS(...)` para el último modelo, no es posible hacer una comparación con los demás. ¿Cómo puede solucionarse tal inconveniente? La respuesta es muy simple en realidad: todos los modelos que hemos creado pueden construirse haciendo uso de la función `IV2SLS(...)`, tan solo hay que ser cuidadosos e incluir los argumentos correctos.

La función `IV2SLS(...)` tiene parámetros específicos para la variable dependiente, la(s) variable(s) exógena(s), la(s) variable(s) endógena(s) y el(los) instrumento(s). Así, para una estimación común y corriente por el método de MCO, basta con asignar `None` al parámetro *endog* y al parámetro *instruments*; para una estimación por MC2E manual basta con emplear las regresoras de la segunda etapa en el parámetro *exog* y asignar `None` a los parámetros *endog* e *instruments*; para una estimación por MC2E automática tan solo debe emplearse los mismos argumentos que en el modelo elaborado previamente en la sección "Proceso automático". El código (se sugiere al lector examinarlo con detenimiento) para estimar los modelos correspondientes, empleando para todos la función `IV2SLS(...)`, es el siguiente:

```
In [21]: ResultadosMCO = IV2SLS(dependent = Y1, exog = sm.add_constant(RegresorasMCO),
                                 endog = None, instruments = None).fit()
         ResultadosManual = IV2SLS(dependent = Y1,
                                   exog = sm.add_constant(RegresorasSegundaEtapa),
                                   endog = None, instruments = None).fit()
         ResultadosAutomatico = IV2SLS(dependent = Y1, exog = sm.add_constant(X),
                                       endog = Y2, instruments = Z).fit()
```

Ahora, como todas las instancias de resultados han sido generadas por un modelo de la misma clase, podemos emplear la función `compare(...)` para comparar los resultados obtenidos y contrastar los modelos, así:

```
In [22]: print(compare({"MCO": ResultadosMCO,
                        "2 Etapas Manual": ResultadosManual,
                        "2 Etapas Automático": ResultadosAutomatico}))

                              Model Comparison
================================================================================
                     2 Etapas Automático 2 Etapas Manual            MCO
--------------------------------------------------------------------------------
Dep. Variable                      lwage           lwage         lwage
Estimator                        IV-2SLS             OLS           OLS
No. Observations                     428             428           428
Cov. Est.                         robust          robust        robust
R-squared                         0.1357          0.0498        0.1568
Adj. R-squared                    0.1296          0.0431        0.1509
F-statistic                       18.611          17.111        82.671
P-value (F-stat)                  0.0003          0.0007        0.0000
==================           ==========      ==========    ==========
const                             0.0481          0.0481       -0.5220
```

```
                            (0.1124)        (0.1071)     (-2.6010)
exper                         0.0442          0.0442       0.0416
                            (2.8546)        (2.7045)     (2.7344)
expersq                      -0.0009         -0.0009      -0.0008
                           (-2.1001)       (-1.9627)    (-1.9402)
educ                          0.0614                       0.1075
                            (1.8503)                     (8.1697)
predicted_educ                                0.0614
                                             (1.7553)
====================  =============  ============= =============
Instruments                    motheduc
                               fatheduc
------------------------------------------------------------------

T-stats reported in parentheses
```

La salida generada por la ejecución del código de la celda anterior nos permite visualizar de forma simultánea los resultados de los diferentes modelos construidos, facilitando la comparación de los mismos. El lector puede observar que en la parte inferior de la salida se indica `T-stats reported in parentheses`, lo que señala que el número reportado entre paréntesis, debajo de cada uno de los coeficientes, corresponde al valor t; sin embargo, en múltiples ocasiones lo que se reporta con correspondencia a un coeficiente no es el valor t sino el error estándar. ¿Es posible reportar el error estándar en lugar del valor t? La respuesta es sí: la función `compare(...)` posee un parámetro *precision*, con valor por defecto *tstats*, que permite especificar el estimador de precisión (dentro de los disponibles) a incluir en la salida.

Para reportar errores estándar en lugar de valores t, basta con asignar "std_errors" al parámetro *precision* de la función `compare(...)`, como se evidencia a continuación:

```
In [23]: print(compare({"MCO": ResultadosMCO,
                        "2 Etapas Manual": ResultadosManual,
                        "2 Etapas Automático": ResultadosAutomatico},
                       precision = "std_errors"))
```

```
                          Model Comparison
=========================================================================
                    2 Etapas Automático 2 Etapas Manual          MCO
-------------------------------------------------------------------------
Dep. Variable                     lwage          lwage        lwage
Estimator                       IV-2SLS            OLS          OLS
No. Observations                    428            428          428
Cov. Est.                        robust         robust       robust
R-squared                        0.1357         0.0498       0.1568
Adj. R-squared                   0.1296         0.0431       0.1509
F-statistic                      18.611         17.111       82.671
P-value (F-stat)                 0.0003         0.0007       0.0000
==================          ==========     ==========   ==========
const                            0.0481         0.0481      -0.5220
                               (0.4278)       (0.4492)     (0.2007)
```

```
exper                          0.0442          0.0442          0.0416
                              (0.0155)        (0.0163)        (0.0152)
expersq                       -0.0009         -0.0009         -0.0008
                              (0.0004)        (0.0005)        (0.0004)
educ                           0.0614                          0.1075
                              (0.0332)                        (0.0132)
predicted_educ                                 0.0614
                                              (0.0350)
====================       ============    ============    ============
Instruments                    motheduc
                               fatheduc
------------------------------------------------------------------------

Std. Errors reported in parentheses
```

Nótese que los valores reportados entre paréntesis han cambiado y que en la sección inferior de la salida aparece `Std. Errors reported in parentheses` en lugar del `T-stats reported in parentheses` del caso previo. De este modo, lo que ahora se reporta en correspondencia a cada coeficiente es su error estándar y no su valor t.

En este punto finaliza el tema de variables instrumentales y el método de Mínimos Cuadrados en 2 Etapas en Python. El lector podrá observar que, hasta el momento, solo se ha trabajado con variables dependientes continuas y que las regresiones llevadas a cabo en los distintos ejemplos son lineales; esto cambiará levemente en el próximo capítulo, en donde se dará paso a una muy breve exposición de otro tipo de modelos.

Capítulo 6

Variable dependiente discreta - Caso binario

En los capítulos previos se ha trabajado con modelos de regresión *lineal* empleando una variable *continua* como variable dependiente; ahora, nos alejaremos un poco de los modelos con tales características, explorando regresiones no lineales con variables dependientes *discretas*.

En los primeros modelos que abordaremos, la variable dependiente es una variable categórica, con solo dos categorías, codificada con los valores 0 y 1. Dicha variable solo toma estos dos valores, los cuales no tienen un significado por sí mismos (a diferencia del caso de variables continuas), en cuanto su función es de codificadores: toma el valor de 1 cuando la observación pertenece a un grupo específico y de 0 cuando no pertenece a tal grupo. En palabras un tanto más simples: la variable toma el valor de 1 cuando se cumple una condición (pertenencia a una clasificación específica) y de 0 cuando no se cumple dicha condición.

Es posible que el lector no tenga completa claridad sobre las características de las variables que emplearemos y que la descripción anterior no le haya resultado del todo satisfactoria; en tal caso, puede que algunos ejemplos sean de ayuda:

- Si contamos con un grupo formado por personas de ambos sexos, en dicho grupo habrá hombres y habrá mujeres; si empleamos una variable 'femenino' para realizar una clasificación por sexo, esta variable tomará el valor de 1 cuando la información corresponda a una mujer y de 0 cuando corresponda a un hombre. 'femenino' es una variable binaria, pues toma un valor cuando una condición se cumple (1 cuando la condición de sexo femenino se cumple) y toma otro valor cuando la condición no se cumple (0 cuando la condición de sexo femenino no se cumple).

- Si tenemos un grupo formado por hombres, es posible que algunos de estos hombres estén casados mientras que otros no, así, podemos emplear una variable binaria 'casado' para clasificarlos en el grupo correspondiente. La variable 'casado' toma el valor de 1 cuando corresponde a un hombre casado y toma el valor de 0 cuando corresponde a un hombre no casado.

- En un grupo de estudiantes universitarios algunos de estos habrán perdido asignaturas mientras que otros no. De este modo, podemos emplear una variable binaria que tome el valor de 1 cuando corresponda a un estudiante que ha perdido asignaturas y de 0 cuando corresponda a un estudiante que no ha perdido asignaturas.

El lector puede observar que la variable binaria toma el valor de 1 cuando se cumple la condición específica y de 0 en caso contrario; tal asignación de valores no tiene sentido numérico y no afecta la regresión. Tranquilamente se puede asignar el valor de 0 cuando la condición se cumple y de 1 en caso contrario; sin embargo, por convención, y mayor facilidad en la interpretación, se emplea 1 para el cumplimiento de la condición y 0 para el no cumplimiento de la misma.

Hasta este punto el lector tiene una idea general de lo que es una variable binaria, sin embargo, aún no se le ha informado cómo se empleará dicha variable en la regresión. Es muy importante señalar que en los modelos que abordaremos a continuación, lo que se estima en realidad es la probabilidad de que la variable dependiente tome el valor de 1 (al lector interesado en conocer el porqué, se recomienda consultar el Capítulo 15, *Modelos de regresión de respuesta cualitativa*, de Gujarati y Porter (2010)); dicha probabilidad está dada por los parámetros ($\boldsymbol{\beta}$) y las variables explicativas ($\boldsymbol{x}$).

En términos un tanto más técnicos:

$$Prob[y_i = 1|\boldsymbol{x_i}] = P_i = F(\boldsymbol{x_i^T \beta})$$

en donde $\boldsymbol{x_i^T}$ corresponde al vector transpuesto de variables explicativas y $\boldsymbol{\beta}$ al vector de parámetros. Así, los coeficientes que se obtienen durante la estimación se asocian al impacto que tiene determinada variable explicativa sobre la probabilidad de que la variable dependiente tome el valor de 1; en otras palabras, los coeficientes se relacionan (no necesariamente cuantifican de forma directa, debido a la forma funcional adoptada, como veremos más adelante) a la variación en la probabilidad de que la variable dependiente tome el valor de 1.

Algunos de los modelos empleados para trabajar con variables de respuesta binaria son: 1) Modelo Lineal de Probabilidad, 2) Modelo Logit y 3) Modelo Probit. Es posible construir y estimar fácilmente cada uno de estos modelos en Python con la ayuda de librerías especializadas. Tales labores son las que llevaremos a cabo a continuación, no sin antes realizar una muy mínima exposición de las ideas detrás de cada uno de los modelos tratados.

El dataset que utilizaremos para desarrollar los distintos modelos será el empleado en el Ejemplo 15.7, *Fumar o no fumar*, de Gujarati y Porter (2010). Este conjunto de información se ha obtenido del sitio web econometrics.com de SHAZAM Analytics Ltd.

Modelo lineal de probabilidad

El Modelo Lineal de Probabilidad (o MLP) es, recurriendo a una definición no-técnica, básicamente, un modelo de regresión en el que la variable dependiente es una variable binaria y la estimación se lleva a cabo por el Método de Mínimos Cuadrados Ordinarios.

Siguiendo la notación que hemos empleado a lo largo de la obra, en el MLP se tiene:

$$Prob[y_i = 1] = \beta_0 + \beta_1 x_{1i} + ... + \beta_k x_{ki} + u_i$$

en donde los estimadores de los $\boldsymbol{\beta}$ se obtienen por el método de Mínimos Cuadrados Ordinarios.

Dado que se trata de una regresión común y corriente por MCO, la estimación de un MLP en Python no debe resultar extraña al lector, pues basta con utilizar las mismas funciones empleadas para el modelo de regresión lineal múltiple tratado en el Capítulo 3. De hecho, los modelos de variable de respuesta binaria que se abordan en este capítulo no deberían representar mayor inconveniente al usuario, en cuanto su construcción es relativamente parecida a la del modelo de regresión lineal y no se requiere recurrir a librerías distintas a las que hemos empleado en capítulos previos.

Preparación del entorno

La primera celda en la que el usuario escribirá y ejecutará código tendrá el siguiente contenido:

```
In [1]: import numpy as np
        import pandas as pd
```

```
import statsmodels.api as sm
import statsmodels.stats.api as sms
import matplotlib.pyplot as plt
```

El anterior bloque de código permite cargar las herramientas necesarias para llevar a cabo las acciones requeridas en el análisis econométrico planteado; en caso de que el lector no tenga claridad al respecto, se le pide consultar la sección "Preparación del entorno" del Capítulo 2, en donde se brinda una explicación sobre tal cuestión.

Importación de los datos

La importación de los datos se lleva a cabo con funciones de la librería *pandas*. Dependiendo del tipo de archivo en el cual se encuentre almacenada la información del dataset, deberemos recurrir a una función particular y un conjunto de parámetros específico.

El dataset que importaremos proviene de un sitio web, no tiene encabezado y está separado por espacios, por lo cual debemos especificar cuidadosamente cada uno de los argumentos requeridos. Concretamente, el siguiente código es el que debe ejecutarse para importar el dataset:

```
In [2]: data = pd.read_csv("http://www.econometrics.com/comdata/gujarati/data_15.7.shd",
                           header = None,
                           delim_whitespace = True,
                           names = ["Fumador", "Edad", "Escolaridad", "Ingreso",
                                    "Pcigs79"])
```

Al lector que no esté familiarizado con el proceso de importación de los datos, se sugiere dirigirse al Capítulo 2, en donde encontrará una explicación que seguramente aclarará sus dudas. Al lector que sí esté familiarizado con dicho proceso, se hace notar que, a diferencia de otras ocasiones, se usan argumentos adicionales, requeridos dadas las características del dataset original: el parámetro *header* permite especificar si se incluye o no un encabezado; el parámetro *delim_whitespace* permite indicar si el delimitador es espacio en blanco (es un parámetro booleano); el parámetro *names* permite asignar los nombres a las columnas del `DataFrame` por medio de una lista Python "[...]".

Una vez ejecutada la línea de código de esta celda, el dataset debería haber sido importado correctamente. En caso contrario, si no puede importarse el dataset directamente desde el sitio web, es posible cargarlo desde el computador (como hemos hecho en todos los capítulos previos).

El código a emplear, en caso de que el proceso de importación desde el sitio web falle, es el siguiente:

```
In [3]: data = pd.read_csv("Gujarati157.csv",
                           index_col = 0)
```

Habiendo ejecutado cualquiera de las celdas de código anteriores, la información del dataset a emplear debería haberse importado correctamente. Para comprobarlo, recurrimos al método `.head()`, aplicado al objeto que contiene el dataset (en este caso le hemos asignado el nombre `data`):

```
In [3]: data.head()
```

Out[3]:

	Fumador	Edad	Escolaridad	Ingreso	Pcigs79
0	0	21	12.0	8500	60.6
1	1	28	15.0	12500	60.6
2	0	67	10.0	12500	60.6
3	0	20	12.0	12500	60.6
4	1	32	12.0	20000	60.6

Podemos observar que el proceso de importación ha sido exitoso y la información del dataset ha sido almacenada correctamente en un `DataFrame` de *pandas*. Ahora, podemos continuar tranquilamente con el desarrollo de nuestro análisis, en cuanto hemos concluido satisfactoriamente el primer paso.

Preparación de los datos

Antes de asignar las variables a los objetos correspondientes, se examinará la estadística descriptiva, para lo que basta con emplear el método `.describe()`:

```
In [4]: data.describe()
```

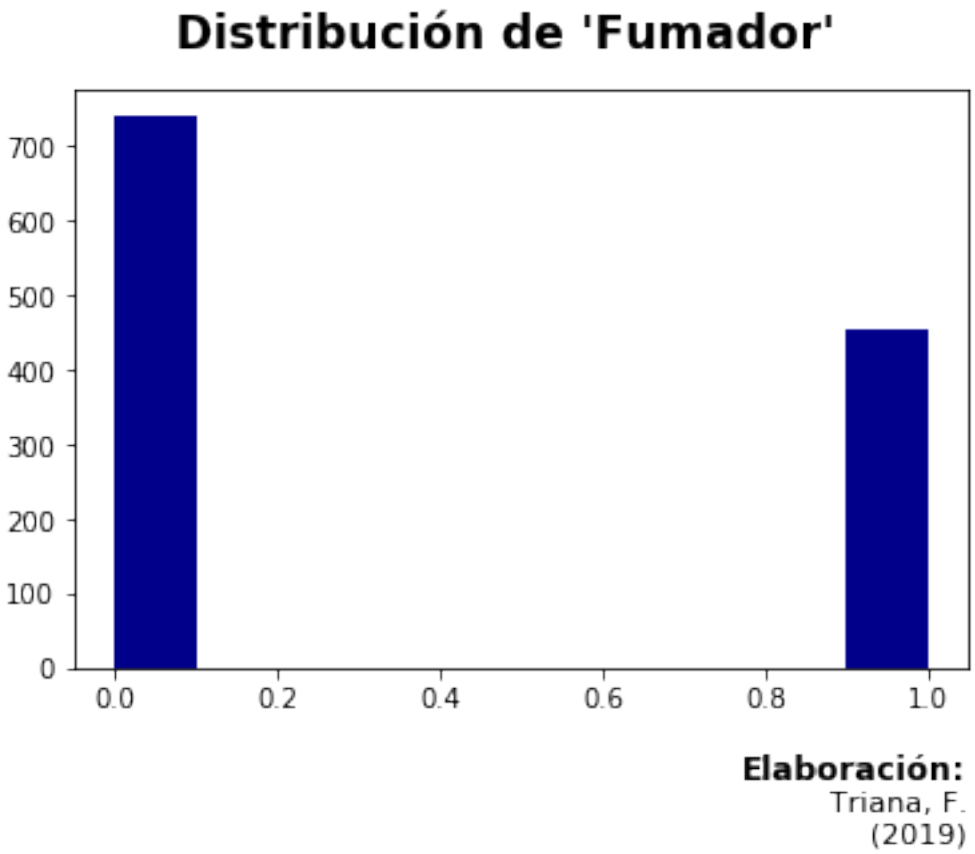
```
Out[4]:            Fumador         Edad  Escolaridad       Ingreso      Pcigs79
        count  1196.000000  1196.000000  1196.000000   1196.000000  1196.000000
        mean      0.380435    41.806856    12.221154  19304.765886    60.984950
        std       0.485697    17.056941     3.275847   9083.511331     4.848666
        min       0.000000    17.000000     0.000000    500.000000    46.300000
        25%       0.000000    27.000000    10.000000  12500.000000    59.225000
        50%       0.000000    39.000000    12.000000  20000.000000    62.100000
        75%       1.000000    56.000000    13.500000  30000.000000    63.800000
        max       1.000000    88.000000    18.000000  30000.000000    69.800000
```

Centre su atención en la columna correspondiente a la variable dependiente ('Fumador'). Esta variable es binaria y solo toma el valor de 0 o el valor de 1; aunque su promedio es 0.38, no existe ninguna observación para la cual el valor de 'Fumador' sea 0.38, de hecho, no existe ninguna observación para la que el valor de esta variable sea distinto de 0 o de 1. A pesar de esto, el promedio de una variable binaria ofrece información valiosa: señala cierta idea de la manera en que están distribuidas las categorías.

Como solo existen ceros y unos, el promedio de este tipo de variable es igual a la suma de los unos dividida por el número total de observaciones (n); por lo tanto, el promedio de una variable binaria corresponde al porcentaje de observaciones que cumplen con la condición específica. Para este ejemplo en concreto, que el promedio de 'Fumador' sea 0.38 significa que el 38 % de los individuos examinados son fumadores y, por ende, el 62 % restante no lo son. Para tener mayor claridad sobre esta distribución de clases puede que un breve examen visual resulte apropiado.

Dado que la dependiente es una variable discreta con solo dos categorías, todos los datos de dicha variable estarán agrupados en únicamente dos puntos (0 y 1) y, de acuerdo a la estadística descriptiva, la cantidad de datos que corresponde a 1 será un poco más de la mitad de la que corresponde a 0, por lo que la barra de 1 será algo más alta que $1/2$ de la barra correspondiente a 0. Examinemos esto por medio de un histograma, usando la función `subplots(...)` de *matplotlib.pyplot* y el método `.hist()`:

```
In [5]: fig, ax = plt.subplots()
        fig.suptitle("Distribución de 'Fumador'", size = 17,
                     fontweight = "bold")
        ax.hist(data["Fumador"], color = "darkblue")
        fig.text(.9,-.02,
                 "Elaboración:",
                 fontsize = 12, fontweight = "bold",
                 ha = "right")
        fig.text(.9,-.1,
                 "Triana, F.\n(2019)",
                 fontsize = 11, ha = "right")
        plt.show()
```



El lector que haya seguido esta obra hasta este punto habrá notado que en los capítulos previos hemos almacenado diversas variables en objetos específicos. Esto lo hemos hecho para agrupar dichas variables de acuerdo a su rol particular en la regresión, tratando de simplificar el código a utilizar posteriormente. Ahora, para los distintos modelos que abordaremos en este capítulo se utilizarán las mismas variables, agrupadas del mismo modo para todos. Procederemos a realizar la agrupación por medio de los corchetes "[ ]" y las listas Python "[...]", tal y como hemos hecho en capítulos previos:

```
In [6]: Y = data["Fumador"]
        X = data[["Edad", "Escolaridad", "Ingreso", "Pcigs79"]]
```

El primer modelo a examinar es el Modelo Lineal de Probabilidad, el cual es, básicamente, un modelo de regresión lineal con variable dependiente binaria y estimación por el método de Mínimos Cuadrados Ordinarios. La regresión lineal y el método MCO se trataron en el Capítulo 3 de esta obra, por lo que a quien no tenga claridad al respecto se le sugiere consultar dicho capítulo.

Para la construcción del modelo y estimación de los coeficientes se emplean (como lo hemos hecho en ocasiones previas) la función `OLS(...)` de *Statsmodels* y el método `.fit()`; para la visualización de los resultados, la función `print(...)` y el método `.summary()`. El código a ejecutar es el siguiente:

```
In [7]: MLP = sm.OLS(Y, sm.add_constant(X))
        ResultadosMLP = MLP.fit()
        print(ResultadosMLP.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                Fumador   R-squared:                       0.039
Model:                            OLS   Adj. R-squared:                  0.036
Method:                 Least Squares   F-statistic:                     12.01
Date:                xxx, xx xxx xxxx   Prob (F-statistic):           1.43e-09
Time:                        xx:xx:xx   Log-Likelihood:                -809.19
No. Observations:                1196   AIC:                             1628.
Df Residuals:                    1191   BIC:                             1654.
Df Model:                           4
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          1.1231      0.188      5.963      0.000       0.754       1.493
Edad          -0.0047      0.001     -5.701      0.000      -0.006      -0.003
Escolaridad   -0.0206      0.005     -4.465      0.000      -0.030      -0.012
Ingreso     1.026e-06   1.63e-06      0.629      0.530   -2.18e-06    4.23e-06
Pcigs79       -0.0051      0.003     -1.799      0.072      -0.011       0.000
==============================================================================
Omnibus:                       37.223   Durbin-Watson:                   1.944
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              173.523
Skew:                           0.449   Prob(JB):                     2.09e-38
Kurtosis:                       1.364   Cond. No.                     2.91e+05
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 2.91e+05. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Uno de los aspectos más importantes a considerar en los modelos de variable dependiente binaria es el **efecto marginal** de cada una de las variables explicativas; tal efecto corresponde a la variación en la probabilidad (de que la variable dependiente tome el valor de 1) respecto a la variación en una variable explicativa. En términos un poco más técnicos, el efecto marginal de una variable x_j corresponde al valor de $\frac{\partial F(\boldsymbol{x_i}^T\boldsymbol{\beta})}{\partial x_{ji}}$, en donde x_j es una variable que hace parte del vector $\boldsymbol{x}$.

Así, para el Modelo Lineal de Probabilidad, los efectos marginales no son más que los coeficientes, pues:

$$F(\boldsymbol{x_i}^T\boldsymbol{\beta}) = \beta_0 + \beta_1 x_{1i} + ... + \beta_k x_{ki}$$

$$\frac{\partial F(\boldsymbol{x_i}^T\boldsymbol{\beta})}{\partial x_{ji}} = \beta_j$$

Por lo tanto, para conocer los efectos marginales del Modelo Lineal de Probabilidad en Python, basta con conocer los coeficientes asociados a las variables. Estos coeficientes corresponden a uno de los atributos (que pueden consultarse con la función `dir(...)`) de la instancia de resultados, y se identifica como `.params`.

Podemos asignar la aplicación de este atributo a un objeto específico y visualizar su contenido utilizando la función `print(...)`. Así:

```
In [8]: MLPMargEff = ResultadosMLP.params
        print("Efectos Marginales MLP\n", MLPMargEff)

Efectos Marginales MLP
 const          1.123089
Edad           -0.004726
Escolaridad    -0.020613
Ingreso         0.000001
Pcigs79        -0.005132
dtype: float64
```

Así, observamos, por ejemplo, que un incremento de un año en la edad está asociado, en promedio, a una disminución de 0.47 puntos porcentuales en la probabilidad de ser fumador, y que un aumento de 1 año en la escolaridad está asociado, en promedio, a una disminución de 2 puntos porcentuales en la probabilidad de ser fumador.

En realidad, para nuestros propósitos, no hay más aspectos a tratar con detenimiento en el Modelo Lineal de Probabilidad; basta con conocer su proceso de construcción y estimación y la obtención de los efectos marginales. Ahora, tal y como se señaló a inicios del capítulo, nos alejaremos de los modelos de regresión lineal y exploraremos otras posibilidades: la primera, el **Modelo Logit**.

Modelo Logit

El Modelo Lineal de Probabilidad, aunque muy simple, no tiende a ser ampliamente usado en la práctica; la razón para tal situación se encuentra, principalmente, en el hecho de que este modelo puede llevar a la obtención de probabilidades carentes de sentido (por ejemplo, mayores a 1 o que violan el axioma de que la probabilidad no puede ser negativa).

Aunque no se mencionó en la sección anterior, el MLP puede resultar muy problemático, en cuanto puede generar probabilidades menores a 0 o mayores a 1, lo que le resta atractivo y conveniencia. Para superar las deficiencias del Modelo Lineal de Probabilidad se recurre a modelos de regresión no lineales, de los cuales los más populares son el **Logit** y el **Probit**.

En el Modelo Logit, la probabilidad de que la variable dependiente tome el valor de 1 se expresa como una función de la siguiente forma:

$$P_i = F\left(\boldsymbol{x}_i^T \boldsymbol{\beta}\right) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_{1i} + \ldots + \beta_k x_{ki})}}$$

Alternativamente, tomando $Z_i = \beta_0 + \beta_1 x_{1i} + \ldots + \beta_k x_{ki}$, la función puede expresarse de la siguiente manera:

$$P_i = F(Z_i) = \frac{1}{1 + e^{-Z_i}}$$

Otro modo de escribir la expresión anterior es $P_i = \frac{e^{Z_i}}{1+e^{Z_i}}$, por lo que $1 - P_i = \frac{1}{1+e^{Z_i}}$. Así, se tiene que:

$$\frac{P_i}{1 - P_i} = e^{Z_i}$$

Aplicando el logaritmo a la anterior expresión se obtiene:

$$L = \ln\left(\frac{P_i}{1-P_i}\right) = Z_i = \beta_0 + \beta_1 x_{1i} + ... + \beta_k x_{ki}$$

L se denomina **logit** y corresponde al logaritmo de la razón de las probabilidades. Así, "mientras el MLP supone que P_i está linealmente relacionado con $\boldsymbol{x_i}$, el modelo logit supone que el logaritmo de la razón de probabilidades está relacionado linealmente con $\boldsymbol{x_i}$" (Gujarati y Porter, 2010, p. 555).

La estimación de los parámetros ($\boldsymbol{\beta}$) en el Modelo Logit es llevada a cabo por medio del método de **Máxima Verosimilitud** (*Maximum Likelihood*) o MV. Al lector interesado en conocer con mayor detalle el Modelo Logit y su proceso de estimación, se sugiere consultar el Capítulo 15, *Modelos de regresión de respuesta cualitativa*, de Gujarati y Porter (2010).

Uno de los puntos a considerar en el Modelo Logit es que la interpretación de los coeficientes no resulta de mayor utilidad, en cuanto estos cuantifican el impacto que tiene el cambio en una variable explicativa sobre el *logaritmo de la razón de probabilidades*, algo que, a primera vista, no tiene gran valor ilustrativo. Debido a esto, se recurre a los efectos marginales para cuantificar el impacto de las variables explicativas.

Como resulta evidente, el efecto marginal de una variable explicativa en un Modelo Logit, a diferencia del caso del Modelo Lineal de Probabilidad, no corresponde al coeficiente asociado a dicha variable, sino que está dado por:

$$\frac{\partial F\left(\boldsymbol{x_i^T\beta}\right)}{\partial x_{ji}} = \beta_j \frac{e^{Z_i}}{\left(1+e^{Z_i}\right)^2}$$

Así, se observa que el efecto marginal de una variable explicativa depende del conjunto de variables explicativas y no está dado directamente por el coeficiente asociado a ésta. Asimismo, es importante notar que en la expresión anterior se ha hallado el efecto marginal de la variable x_j para la observación i, pero debe tenerse en consideración que el dataset contiene información de múltiples observaciones y los valores de las variables son diferentes entre dichas observaciones, por lo que no existe un único valor para el efecto marginal de la variable x_j sino n (número de observaciones) valores para éste.

Dado que no resulta muy práctico conocer la magnitud del efecto marginal de una variable para cada una de las observaciones, es necesario contar con una medida "general" del efecto marginal de determinada variable, pero ¿cómo se obtiene dicha medida "general"? Una respuesta posible a este interrogante es: existen varios "caminos". A continuación, se presentan los más populares.

El primer camino (conocido como efectos marginales en la media) para obtener una medida "general" del efecto marginal de una variable es utilizar la observación promedio para calcularlo; es decir, emplear el promedio de cada variable como valor de la variable correspondiente dentro de Z.

Definiendo el vector que contiene los valores promedio de las variables explicativas como $\bar{\boldsymbol{x}}$, y teniendo en cuenta que, consecuentemente, $\bar{\boldsymbol{x}}^T\boldsymbol{\beta} = \bar{Z}$, el efecto marginal de la variable x_j está dado por:

$$\frac{\partial P}{\partial x_j} = \beta_j \frac{e^{\bar{Z}}}{\left(1+e^{Z}\right)^2}$$

Así, el efecto marginal "general" de determinada variable corresponde al efecto marginal de dicha variable calculado con los valores correspondientes a la observación promedio.

El otro camino (conocido como efectos marginales promedio) para obtener una medida "general" del efecto marginal de una variable es obtener el promedio de los efectos marginales individuales correspondientes a dicha variable; es decir, calcular el efecto marginal de la variable x_j para cada una de las observaciones i (en total se tienen n observaciones), y hallar el promedio de dichos efectos marginales. Concretamente, el efecto marginal de la variable x_j viene dado por:

$$\frac{\partial P}{\partial x_j} = \frac{\sum_{i=1}^{n} \frac{\partial F(x_i^T \beta)}{\partial x_{ji}}}{n} = \beta_j \frac{\sum_{i=1}^{n} \frac{e^{Z_i}}{(1+e^{Z_i})^2}}{n}$$

Así, el efecto marginal "general" de determinada variable corresponde al promedio de los efectos marginales individuales para dicha variable.

Hasta este punto se ha dado una descripción, muy breve, de la idea general sobre la que trata el Modelo Logit; ahora, es momento de aplicar a la práctica los conocimientos adquiridos, llevando a cabo la construcción y estimación de un modelo logit en Python.

Modelo Logit en Python

El dataset que se empleará para la construcción y estimación del Modelo Logit ha sido importado previamente, en la sección de este capítulo que aborda el Modelo Lineal de Probabilidad; asimismo, la preparación del entorno y de los datos se ha llevado a cabo exitosamente con anterioridad. Dado que la construcción y estimación de modelos logit se consigue con funciones de *Statsmodels* y no requieren de instrumentos provenientes de librerías distintas a las ya importadas, se cuenta con todas las herramientas para poder trabajar adecuadamente. Al usuario que no haya seguido el ejercicio correspondiente a la sección del MLP, se invita a ejecutar las celdas de código #1, #2 y #6, las cuales son indispensables para la labor que ahora se pretende desarrollar.

Para construir un modelo logit en Python se emplea la función `Logit(...)` de *Statsmodels*; sus argumentos incluyen la variable dependiente, las variables explicativas y el parámetro *missing* (con valor por defecto `None`) que permite indicar la acción a seguir en caso de que existan valores faltantes.

Para la estimación se usa, al igual que en los modelos que hemos tratado en capítulos previos, el método `.fit()`; para la visualización del resumen de resultados el método `.summary()` (aplicado sobre la instancia de resultados) y la función `print(...)` (tomando como argumento la aplicación del método `.summary()`). El código correspondiente a nuestro caso es:

```
In [9]: ModeloLogit = sm.Logit(Y, sm.add_constant(X))
        ResultadosLogit = ModeloLogit.fit()
        print(ResultadosLogit.summary())

Optimization terminated successfully.
         Current function value: 0.644516
         Iterations 5
                           Logit Regression Results
==============================================================================
Dep. Variable:                Fumador   No. Observations:                 1196
Model:                          Logit   Df Residuals:                     1191
Method:                           MLE   Df Model:                            4
Date:                xxx, xx xxx xxxx   Pseudo R-squ.:                 0.02975
Time:                        xx:xx:xx   Log-Likelihood:                -770.84
converged:                       True   LL-Null:                       -794.47
                                        LLR p-value:                 1.341e-09
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.7451      0.829      3.311      0.001       1.120       4.370
Edad          -0.0209      0.004     -5.577      0.000      -0.028      -0.014
```

```
Escolaridad    -0.0910      0.021     -4.402      0.000      -0.131      -0.050
Ingreso       4.72e-06   7.17e-06      0.658      0.510   -9.33e-06    1.88e-05
Pcigs79        -0.0223      0.012     -1.789      0.074      -0.047       0.002
==============================================================================
```

El lector debe recordar que la interpretación directa de los coeficientes en el Modelo Logit no resulta particularmente útil a primera vista, en cuanto estos coeficientes cuantifican el impacto de determinada variable sobre el *logaritmo de la razón de probabilidades* y no directamente sobre la probabilidad de que la variable dependiente tome el valor de 1.

Para conocer el impacto de una variable específica en un modelo logit, tal y como se señaló a inicio de este capítulo, se recurre a los efectos marginales. Estos efectos se obtienen por medio del método `.get_margeff()`, aplicado a la instancia de resultados del modelo. El lector debe tener en cuenta que para obtener una medida "general" del efecto marginal de una variable específica se señaló la existencia de dos caminos (que no son los únicos): 1) obtener el efecto marginal usando la observación promedio y 2) obtener el promedio de los efectos marginales individuales.

Para obtener los efectos marginales utilizando el primer "camino", basta con asignar el valor 'mean' al parámetro *at* del método `.get_margeff()`; para obtenerlos por el segundo "camino", basta con asignarle el valor de 'overall'. El parámetro *at* del método `.get_margeff()` corresponde a 'overall' por defecto, por lo que los efectos marginales promedio son los que se obtienen en caso de que no se especifique el valor de *at*. Los efectos marginales promedio (segundo camino) son:

```
In [10]: LogitMargEff = ResultadosLogit.get_margeff()
         print(LogitMargEff.summary())
```

```
        Logit Marginal Effects
====================================
Dep. Variable:                Fumador
Method:                          dydx
At:                           overall
==============================================================================
                dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Edad           -0.0047      0.001     -5.864      0.000      -0.006      -0.003
Escolaridad    -0.0206      0.005     -4.539      0.000      -0.030      -0.012
Ingreso      1.069e-06   1.62e-06      0.659      0.510   -2.11e-06    4.25e-06
Pcigs79        -0.0051      0.003     -1.798      0.072      -0.011       0.000
==============================================================================
```

Esta tabla, además del valor específico de los efectos marginales, también presenta los intervalos de confianza correspondientes e indica qué método es empleado para calcularlos (`At:`). Así, observamos, por ejemplo, que un incremento de un año en la edad está asociado, en promedio, a una disminución de 0.47 puntos porcentuales en la probabilidad de ser fumador y que un aumento de 1 año en la escolaridad está asociado, en promedio, a una disminución de 2 puntos porcentuales en la probabilidad de ser fumador.

Para los efectos marginales en la media (primer camino), tan solo se necesita asignar 'mean' al parámetro *at*. El código es el siguiente:

```
In [11]: LogitMargEff = ResultadosLogit.get_margeff(at = "mean")
         print(LogitMargEff.summary())
```

```
        Logit Marginal Effects
=====================================
Dep. Variable:                Fumador
Method:                          dydx
At:                              mean
==============================================================================
                dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Edad           -0.0049      0.001     -5.598      0.000      -0.007      -0.003
Escolaridad    -0.0213      0.005     -4.411      0.000      -0.031      -0.012
Ingreso      1.107e-06   1.68e-06      0.658      0.510   -2.19e-06     4.4e-06
Pcigs79        -0.0052      0.003     -1.790      0.073      -0.011       0.000
==============================================================================
```

Así, observamos, por ejemplo, que un incremento de un año en la edad está asociado, en promedio, a una disminución de 0.49 puntos porcentuales en la probabilidad de ser fumador y que un aumento de 1 año en la escolaridad está asociado, en promedio, a una disminución de 2.13 puntos porcentuales en la probabilidad de ser fumador.

Modelo Probit

Al igual que el Modelo Logit, una alternativa popular al MLP es el Modelo **Probit**. En este modelo, la probabilidad de que la variable dependiente tome el valor de 1 viene dada por lo siguiente:

$$P_i = F\left(\boldsymbol{x_i^T\beta}\right) = \Phi\left(\boldsymbol{x_i^T\beta}\right) = \int_{-\infty}^{x_i^T} \phi(z)dz$$

Así, $F\left(\boldsymbol{x_i^T\beta}\right)$ corresponde a la **Función de Distribución Acumulada** de la Distribución **Normal Estándar**. Al igual que con el Modelo Logit, las probabilidades obtenidas no son inferiores a 0 o superiores a 1 y el método de estimación de los parámetros es el de Máxima Verosimilitud (MV). Al lector interesado en conocer con mayor detalle el Modelo Probit y su proceso de estimación, se le recomienda consultar el Capítulo 15, *Modelos de regresión de respuesta cualitativa*, de Gujarati y Porter (2010).

Como en el caso del Modelo Logit, la interpretación directa de los coeficientes asociados a las variables explicativas no resulta de gran utilidad y los efectos marginales correspondientes difieren de estos. Para el Modelo Probit, el efecto marginal de la variable x_j viene dado por:

$$\frac{\partial F\left(\boldsymbol{x_i^T\beta}\right)}{x_{ji}} = \phi\left(\boldsymbol{x_i^T\beta}\right)\beta_j$$

en donde $\phi(.)$ es la Función de **Densidad de Probabilidad** de la Distribución Normal Estándar. Como puede observarse, los efectos marginales en el Modelo Probit, al igual que en el Modelo Logit, no están dados directamente por el coeficiente asociado a la variable, sino que dependen de los valores de las variables explicativas. Así, en un dataset con n observaciones se obtiene n valores para el efecto marginal de la misma variable; para obtener una medida "general", pueden seguirse los dos caminos previamente señalados en el caso del Modelo Logit.

En el primer camino se emplean los valores promedio de las variables con el propósito de hallar un único efecto marginal para determinada variable. Definiendo, nuevamente, el vector que contiene los valores promedio de las variables explicativas como $\bar{\boldsymbol{x}}$, el efecto marginal de la variable x_j está dado por:

$$\frac{\partial P}{\partial x_j} = \phi\left(\bar{x}_i^T \boldsymbol{\beta}\right)$$

Así, el efecto marginal 'general' de determinada variable corresponde al efecto marginal de dicha variable calculado con los valores correspondientes a la observación promedio.

En el segundo camino, se calculan los n efectos marginales correspondientes a la misma variable y simplemente se obtiene su promedio. Concretamente:

$$\frac{\partial P}{\partial x_j} = \beta_j \frac{\sum_{i=1}^{n} \phi\left(x_i^T \boldsymbol{\beta}\right)}{n}$$

Así, el efecto marginal "general" de determinada variable corresponde al promedio de los efectos marginales individuales para dicha variable.

Hasta este punto se ha dado una descripción muy breve de la idea general sobre la que trata el Modelo Probit; ahora, es momento de aplicar a la práctica los conocimientos adquiridos, llevando a cabo la construcción y estimación de un modelo probit en Python.

Modelo Probit en Python

El dataset que se empleará para la construcción y estimación del Modelo Probit es el mismo utilizado para el MLP y el Modelo Logit. Dado que la construcción y estimación de modelos probit se consigue con funciones de *Statsmodels* y no requieren de instrumentos provenientes de librerías distintas a las ya importadas, se cuenta con todas las herramientas para poder trabajar adecuadamente. Al usuario que no haya seguido el ejercicio correspondiente a la sección del MLP o a la sección del Modelo Logit, se le invita a ejecutar las celdas de código #1, #2 y #6, las cuales son indispensables para la labor que ahora se pretende desarrollar.

Para construir un modelo probit en Python se emplea la función `Probit(...)` de *Statsmodels*; sus argumentos incluyen la variable dependiente, las variables explicativas y el parámetro *missing* (con valor por defecto 'none') que permite indicar la acción a seguir en caso de que existan valores faltantes.

Para la estimación se usa, al igual que en los modelos que hemos tratado en capítulos previos, el método `.fit()`; para la visualización de resultados, el método `.summary()` (aplicado sobre la instancia de resultados) y la función `print(...)` (tomando como argumento la aplicación del método `.summary()`). El código correspondiente a nuestro caso es:

```
In [12]: ModeloProbit = sm.Probit(Y, sm.add_constant(X))
         ResultadosProbit = ModeloProbit.fit()
         print(ResultadosProbit.summary())

Optimization terminated successfully.
         Current function value: 0.644304
         Iterations 5
                          Probit Regression Results
==============================================================================
Dep. Variable:                Fumador   No. Observations:                 1196
Model:                         Probit   Df Residuals:                     1191
Method:                           MLE   Df Model:                            4
Date:                xxx, xx xxx xxxx   Pseudo R-squ.:                 0.03007
Time:                        xx:xx:xx   Log-Likelihood:                -770.59
converged:                       True   LL-Null:                       -794.47
                                        LLR p-value:                1.052e-09
```

```
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          1.7019      0.511      3.333      0.001       0.701       2.703
Edad          -0.0130      0.002     -5.655      0.000      -0.017      -0.008
Escolaridad   -0.0562      0.013     -4.450      0.000      -0.081      -0.031
Ingreso      2.72e-06    4.4e-06      0.619      0.536    -5.9e-06    1.13e-05
Pcigs79       -0.0138      0.008     -1.792      0.073      -0.029       0.001
==============================================================================
```

El lector debe recordar que la interpretación directa de los coeficientes en el Modelo Probit no resulta particularmente útil a primera vista y que para conocer el impacto de una variable específica en un modelo probit, tal y como se señaló a inicio de este capítulo, se recurre a los **efectos marginales**.

Los efectos marginales se obtienen por medio del método `.get_margeff()`, aplicado a la instancia de resultados del modelo. El lector debe tener en cuenta que para obtener una medida "general" del efecto marginal de una variable específica se señaló la existencia de dos caminos (que no son los únicos): 1) obtener el efecto marginal usando la observación promedio y 2) obtener el promedio de los efectos marginales individuales.

Para obtener los efectos marginales utilizando el primer "camino" basta con asignar el valor 'mean' al parámetro *at* del método `.get_margeff()`; para obtenerlos por el segundo "camino", basta con asignarle el valor de 'overall'. El parámetro *at* del método `.get_margeff()` corresponde a 'overall' por defecto, por lo que los efectos marginales promedio son los que se obtienen en caso de que no se especifique el valor de *at*.

Los efectos marginales promedio (segundo camino) son:

```
In [13]: ProbitMargEff = ResultadosProbit.get_margeff()
         print(ProbitMargEff.summary())

       Probit Marginal Effects
=====================================
Dep. Variable:                Fumador
Method:                           dydx
At:                           overall
==============================================================================
                dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Edad          -0.0048      0.001     -5.899      0.000      -0.006      -0.003
Escolaridad   -0.0207      0.005     -4.565      0.000      -0.030      -0.012
Ingreso     1.002e-06   1.62e-06      0.619      0.536   -2.17e-06    4.18e-06
Pcigs79       -0.0051      0.003     -1.800      0.072      -0.011       0.000
==============================================================================
```

La información que se presenta en esta tabla es la misma que se obtiene en el caso del modelo Logit. Así, observamos, por ejemplo, que un incremento de un año en la edad está asociado, en promedio, a una disminución de 0.48 puntos porcentuales en la probabilidad de ser fumador y que un aumento de 1 año en la escolaridad está asociado, en promedio, a una disminución de 2 puntos porcentuales en la probabilidad de ser fumador.

Para los efectos marginales en la media (primer camino) solo es necesario modificar el valor del parámetro *at*; el código es el siguiente:

```
In [14]: ProbitMargEff = ResultadosProbit.get_margeff(at = "mean")
         print(ProbitMargEff.summary())
```

```
       Probit Marginal Effects
=====================================
Dep. Variable:                Fumador
Method:                          dydx
At:                              mean
==============================================================================
                dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Edad           -0.0049      0.001     -5.666      0.000      -0.007      -0.003
Escolaridad    -0.0213      0.005     -4.455      0.000      -0.031      -0.012
Ingreso      1.032e-06   1.67e-06      0.619      0.536   -2.24e-06     4.3e-06
Pcigs79        -0.0052      0.003     -1.792      0.073      -0.011       0.000
==============================================================================
```

Empleando este método, observamos, por ejemplo, que un incremento de un año en la edad está asociado, en promedio, a una disminución de 0.49 puntos porcentuales en la probabilidad de ser fumador y que un aumento de 1 año en la escolaridad está asociado, en promedio, a una disminución de 2.13 puntos porcentuales en la probabilidad de ser fumador.

Comparación de modelos

Resulta útil poder comparar los resultados de los distintos modelos tratados teniendo a disposición su información en un único espacio; es decir, contando con una tabla de resumen en la que se presente simultáneamente la información básica de la instancia de resultados correspondiente a cada modelo.

Es posible comparar resultados de modelos creados con funciones de *Statsmodels* empleando la función `summary_col(...)` del módulo `iolib.summary2` de *Statsmodels*. Esta función permite resumir múltiples instancias de resultados presentándolas una al lado de la otra; algunos de sus argumentos son la lista de instancias de resultados a comparar (*results*), la lista de nombres a asignar al resumen de cada instancia (*model_names*) y el parámetro *stars* (de tipo booleano), que indica si se quiere o no visualizar la significancia por medio de asteriscos (*).

El primer paso a seguir para poder comparar las instancias de resultados de los modelos es importar la función que permite llevar a cabo tal tarea. La línea de importación de la función `summary_col(...)` de *Statsmodels.iolib.summary2* es la siguiente:

```
In [15]: from statsmodels.iolib.summary2 import summary_col
```

Ahora, teniendo a disposición la función requerida, basta con especificar los argumentos correspondientes y ejecutar el código para obtener la tabla de resumen (que denominaremos `MiTabla`):

```
In [16]: MiTabla = summary_col(results = [ResultadosMLP, ResultadosLogit,
                                           ResultadosProbit],
                               stars = True,
                               model_names = ["Modelo MLP", "Modelo Logit",
                                              "Modelo Probit"])
         print(MiTabla)
```

```
================================================
            Modelo MLP Modelo Logit Modelo Probit
------------------------------------------------
const       1.1231***  2.7451***    1.7019***
            (0.1884)   (0.8292)     (0.5106)
Edad        -0.0047*** -0.0209***   -0.0130***
            (0.0008)   (0.0037)     (0.0023)
Escolaridad -0.0206*** -0.0910***   -0.0562***
            (0.0046)   (0.0207)     (0.0126)
Ingreso     0.0000     0.0000       0.0000
            (0.0000)   (0.0000)     (0.0000)
Pcigs79     -0.0051*   -0.0223*     -0.0138*
            (0.0029)   (0.0125)     (0.0077)
================================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

El lector debe recordar que la interpretación directa de los coeficientes de los modelos Logit y Probit no resulta particularmente útil y que no es posible comparar su magnitud con la de los correspondientes al Modelo Lineal de Probabilidad, en tanto cuantifican relaciones distintas. Debido a esto, se ha recurrido a los efectos marginales para poder obtener una medida comparable, entre modelos, del efecto de una variable explicativa sobre la probabilidad de que la variable dependiente tome el valor de 1. La tabla de resumen que hemos apenas creado presenta los coeficientes de cada modelo y **no** los efectos marginales, ¿esto quiere decir que no es posible comparar la información presentada?

Aunque no es posible comparar los coeficientes de los modelos MLP, Logit y Probit directamente y, por tanto, utilizamos los efectos marginales, el lector debe tener en cuenta que existe una relación aproximada entre las magnitudes de dichos coeficientes.

Siguiendo a Katchova (2013), los coeficientes de los modelos MLP, Logit y Probit se caracterizan por las siguientes relaciones:

$$\beta_{Logit} \simeq 4\beta_{MLP}$$

$$\beta_{Probit} \simeq 2{,}5\beta_{MLP}$$

$$\beta_{Logit} \simeq 1{,}6\beta_{Probit}$$

Así, basta con aplicar las transformaciones correspondientes a los coeficientes de los modelos Logit y Probit para obtener medidas directamente comparables con los coeficientes del Modelo Lineal de Probabilidad; sin embargo, debe resaltarse que dichas transformaciones se basan en relaciones *aproximadas*. Por tanto, si se quiere comparar los modelos, lo más indicado es basar tal comparación en los efectos marginales.

Statsmodels no ofrece una función o método (a la fecha de publicación de esta obra y al conocimiento del autor) que permita comparar los efectos marginales de varios modelos; sin embargo, esta situación no debe desalentar al usuario, en cuanto aún es posible obtener una tabla de resumen que presente los efectos marginales. Ahora, algo muy importante que debe tener en cuenta es que la labor asociada a la consecución de este objetivo puede resultar un tanto tediosa (o incluso complicada para el novato) y debe desarrollarse cuidadosamente.

El objetivo que pretendemos es obtener una tabla de resumen en la que se presenten los efectos marginales correspondientes al Modelo Lineal de Probabilidad, al Modelo Logit y al Modelo Probit, por lo que los datos indispensables que necesitamos son los valores de tales efectos marginales. Lo primero que haremos es crear `DataFrames` de *pandas* para almacenar los datos correspondientes.

La información de los efectos marginales del Modelo Logit y del Modelo Probit se ha almacenado con anterioridad en objetos específicos; ahora, extraeremos únicamente las magnitudes de dichos efectos marginales (con el atributo `.margeff`, aplicado al objeto que contiene la información de los efectos marginales respectivos) y las emplearemos como valores del parámetro *data* en la función `DataFrame(...)` de *pandas*. Asimismo, para el parámetro *index* haremos uso de una lista Python "[...]" que contenga los nombres de las variables correspondientes y asignaremos al parámetro *columns* el nombre del respectivo modelo. El procedimiento es el mismo para los modelos Logit y Probit, como se evidencia a continuación:

```
In [17]: Coeficientes = ["Edad", "Escolaridad", "Ingreso", "Pcigs79"]
         EfectosLogit = pd.DataFrame(LogitMargEff.margeff, index = Coeficientes,
                                     columns = ["Logit"])
         EfectosProbit = pd.DataFrame(ProbitMargEff.margeff, index = Coeficientes,
                                      columns = ["Probit"])
```

Ya contamos con los datos de los efectos marginales de los modelos Logit y Probit, pero, ¿qué hay de los efectos marginales del MLP? El lector debe recordar que los efectos marginales del Modelo Lineal de Probabilidad corresponden directamente a los coeficientes, por lo que, a diferencia de los modelos Logit y Probit, su obtención se logra con la aplicación del atributo `.params`. El resultado de la aplicación de este atributo difiere del que se obtiene con la aplicación del método `.get_margeff()` de los modelos Logit y Probit, por lo que el procedimiento para extraer las magnitudes de los efectos marginales (coeficientes) será un tanto distinto al que hemos desarrollado previamente.

Lo primero que el usuario debe notar es que el atributo `.params` genera un resultado en el que se incluye el coeficiente correspondiente al término del intercepto, sin embargo, para los modelos Logit y Probit, como resulta lógico, no se reporta un efecto marginal correspondiente a éste. Por lo tanto, con el propósito de comparar los efectos marginales de las mismas variables, se ignorará el término del intercepto (β_0).

Para realizar la "extracción" de las magnitudes de todos los coeficientes exceptuando el término del intercepto se hace uso de los corchetes "[ ]", aplicados al objeto que contiene los parámetros del MLP (previamente lo almacenamos como una `Series` de *pandas*). Dentro de los corchetes se empleará la secuencia `1:5`; esto, con el propósito de ignorar el término del intercepto. Adicionalmente, el nombre del modelo se asignará al parámetro *columns*.

- **Nota:** Al lector que lo desconozca, se le informa que una `Series` de *pandas* indiza (por defecto) sus observaciones con números enteros desde 0 y que la selección por corchetes "[ ]" con enteros es incluyente en el primer valor y excluyente en el último, por lo que el empleo de una secuencia `1:5` dentro de corchetes "[ ]" de selección significa "ignorar la primera observación (que está identificada con 0) e incluir todas las demás hasta la indizada con el número 4 (no incluye la que tiene 5 como índice)".

El código a ejecutar para el caso del MLP es el siguiente:

```
In [18]: EfectosMLP = pd.DataFrame(MLPMargEff[1:5], columns = ["MLP"])
```

Ya se cuenta con los `DataFrame` que contienen las magnitudes de los efectos marginales de los modelos Logit, Probit y MLP. Ahora, es momento de agrupar estos `DataFrames` en uno solo; para lograrlo, se emplea la función `concat(...)` de *pandas*; en sus argumentos se incluye una lista Python "[...]" que contiene los `DataFrame` a concatenar y el parámetro *axis* con valor de 1, de modo que la concatenación se logre ubicando un `DataFrame` al lado de otro:

```
In [19]: pd.concat([EfectosMLP, EfectosLogit, EfectosProbit], axis = 1)
```

```
Out[19]:                    MLP     Logit    Probit
         Edad        -0.004726 -0.004890 -0.004920
         Escolaridad -0.020613 -0.021334 -0.021340
         Ingreso      0.000001  0.000001  0.000001
         Pcigs79     -0.005132 -0.005234 -0.005235
```

Ahora se cuenta con una única tabla de resumen en la que se presentan los efectos marginales de los distintos modelos: magnitudes que sí son directamente comparables. Podemos observar que los efectos estimados no difieren significativamente: por ejemplo, el efecto marginal de la variable *Escolaridad* es de -2.06 p.p en el MLP, de -2.13 p.p en el modelo Logit y de -2.13 p.p en el modelo Probit; para la variable *Edad* el efecto marginal va desde -0.4726 p.p (MLP) hasta -0.492 p.p (Probit).

En este punto cierra este capítulo, en el cual hemos tratado los modelos con variable dependiente binaria; ahora, extenderemos un tanto el alcance de los modelos de variable dependiente discreta al incluir regresadas que no toman únicamente dos valores, es decir, variables **no** binarias.

Capítulo 7

Variable dependiente discreta - Caso no binario

En el capítulo previo se trabajó con modelos en los que la variable dependiente era discreta y, además, solo podía tomar dos valores; en este capítulo, se dará continuidad al tema de la variable dependiente discreta, pero se eliminará la restricción de que ésta solo puede tomar dos valores, es decir, abordaremos el tema de variables discretas no binarias.

Las variables dependientes con las que trabajaremos serán categóricas, pero no estarán limitadas a solamente dos categorías, por lo que no tomarán únicamente dos valores (0 y 1) sino tantos valores como alternativas tenga dicha variable.

La breve exposición de los fundamentos teóricos del modelo a desarrollar en este capítulo está basada, en su totalidad, en Katchova (2013, 2), por lo que, en caso de ser necesario, se invita al lector a consultar dicha fuente para tener mayor claridad al respecto.

Las variables dependientes a emplear son discretas no binarias, por lo que toman valores de un conjunto finito compuesto por más de dos opciones. Es posible que algunos ejemplos tengan cierto valor para ilustrar la naturaleza de las mencionadas variables:

- Cuando una persona desea consumir una cerveza, tiene a su disposición un conjunto **finito** y bien definido de marcas de cerveza que puede adquirir. Estas marcas pueden codificarse por medio de números, asignando un número distinto a cada una, por lo que la variable resultante corresponde a una variable discreta no binaria (asumiendo que existen más de dos marcas de cerveza a disposición), en cuanto existen múltiples, pero limitadas, posibilidades.

- Cuando alguien asiste a una sala de cine, tiene a su disposición distintos "combos" de comida. Existe un conjunto bien definido de "combos" y su cantidad es limitada; de este modo, se puede asignar un número distinto a cada combo, con el propósito de identificarlo. Así, la variable resultante de la codificación de los "combos" es una variable discreta no binaria (asumiendo que existen más de dos combos).

A una variable discreta con m categorías (o alternativas) se aplica un proceso de codificación empleando m números; estos números no tienen significado por sí mismos, en cuanto su magnitud no es interpretable y su función es exclusivamente de codificadores. Asimismo, debe notarse que el orden en el que se asignan los números es libre, en cuanto estos carecen de significado propio y no tienen naturaleza ordinal, por lo que no establecen un orden definido para las alternativas de las variables.

La variable tiene múltiples categorías, pero es importante señalar que a determinado individuo (entidad) solo corresponde una única alternativa; es decir, cada individuo o entidad, de las múltiples alternativas a disposición, puede elegir únicamente una. Retomando los ejemplos anteriores, para el caso de las cervezas, el individuo solo puede elegir una marca para la cerveza que consumirá en este momento y, para el caso de los combos de comida, la persona solo debe elegir un único combo.

Para este tipo de modelos, la información se presenta en dos formatos específicos: *wide* y *long*. En el formato *wide*, la información para cada individuo (entidad) se presenta en una única fila, por lo tanto $y = j\ (j = 1, 2, 3, \ldots, m)$; en el formato *long*, la información para cada individuo (entidad) se presenta en m filas, cada una de las cuales corresponde a una de las alternativas de la variable dependiente, la cual toma el valor de 1 una única vez y de 0 $m - 1$ veces para cada individuo (entidad).

Recurriendo a una descripción un tanto más ilustrativa, a continuación, se presenta la misma información, en formato *wide* y en formato *long*. Retomando el ejemplo de las cervezas, tomando en consideración solo 5 marcas (codificadas de la siguiente manera: $Aguila: 1, Poker: 2, C.Colombia: 3, Corona: 4, Heineken: 5$), la siguiente información se presenta en formato ***wide***:

Individuo	Marca elegida	y	Ingreso del individuo	$Águila	$Póker	$C.Colombia	$Corona	$Heineken
Individuo 1	Póker	2	COP 2,300,000	COP 2,600	COP 3,100	COP 3,300	COP 4,200	COP 4,100
Individuo 2	C. Colombia	3	COP 3,000,000	COP 3,300	COP 3,600	COP 4,000	COP 5,700	COP 5,500
Individuo 3	Corona	4	COP 2,700,000	COP 3,000	COP 3,300	COP 3,600	COP 5,000	COP 4,800
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
Individuo n	Águila	1	COP 2,500,000	COP 3,000	COP 3,400	COP 3,900	COP 4,300	COP 4,500

Esta misma información, en formato ***long***, corresponde a la siguiente tabla:

Individuo	Marca elegida	Alternativas	y_j	Ingreso del individuo	$ Alternativa
Individuo 1	Póker	Águila	$y_1 = 0$	COP 2,300,000	COP 2,600
Individuo 1	Póker	Póker	$y_2 = 1$	COP 2,300,000	COP 3,100
Individuo 1	Póker	C. Colombia	$y_3 = 0$	COP 2,300,000	COP 3,300
Individuo 1	Póker	Corona	$y_4 = 0$	COP 2,300,000	COP 4,200
Individuo 1	Póker	Heineken	$y_5 = 0$	COP 2,300,000	COP 4,100
Individuo 2	C. Colombia	Águila	$y_1 = 0$	COP 3,000,000	COP 3,300
Individuo 2	C. Colombia	Póker	$y_2 = 0$	COP 3,000,000	COP 3,600
Individuo 2	C. Colombia	C. Colombia	$y_3 = 1$	COP 3,000,000	COP 4,000
Individuo 2	C. Colombia	Corona	$y_4 = 0$	COP 3,000,000	COP 5,700
Individuo 2	C. Colombia	Heineken	$y_5 = 0$	COP 3,000,000	COP 5,500
Individuo 3	Corona	Águila	$y_1 = 0$	COP 2,700,000	COP 3,000
Individuo 3	Corona	Póker	$y_2 = 0$	COP 2,700,000	COP 3,300
Individuo 3	Corona	C. Colombia	$y_3 = 0$	COP 2,700,000	COP 3,600
Individuo 3	Corona	Corona	$y_4 = 1$	COP 2,700,000	COP 5,000
Individuo 3	Corona	Heineken	$y_5 = 0$	COP 2,700,000	COP 4,800
. . .	. . .	. . .	. . .	. . .	. . .
Individuo n	Águila	Águila	$y_1 = 1$	COP 2,500,000	COP 3,000
Individuo n	Águila	Póker	$y_2 = 0$	COP 2,500,000	COP 3,400
Individuo n	Águila	C. Colombia	$y_3 = 0$	COP 2,500,000	COP 3,900
Individuo n	Águila	Corona	$y_4 = 0$	COP 2,500,000	COP 4,300
Individuo n	Águila	Heineken	$y_5 = 0$	COP 2,500,000	COP 4,500

Así, el lector puede apreciar que para el formato *wide* $y = j$, mientras que para el formato *long*:

$$y_j = \begin{cases} 1, & \text{si } y = j \\ 0, & \text{si } y \neq j \end{cases}$$

Algo importante que debe mencionarse es que en los modelos de variable dependiente discreta no binaria, al igual que en los modelos de variable dependiente discreta binaria, lo que interesa no es el valor de la variable dependiente como tal sino la probabilidad de que esta tome un valor en específico. Así, en el Modelo Logit, puesto que la variable dependiente era binaria, solo existían dos posibilidades, por lo que se estimaba la probabilidad de que la variable dependiente tomará el valor de 1, y la probabilidad de que esta tomará el valor de 0 simplemente correspondía a uno menos la probabilidad de que tomará el valor de 1. Sin embargo, en el modelo Logit Multinomial (que es el que trabajaremos en adelante) no existen únicamente dos alternativas, entonces, ¿qué probabilidad es la que se estima?

El lector debe saber que en el modelo Logit Multinomial lo que se busca es estimar la probabilidad de que se elija la alternativa j dentro de las m alternativas que tiene la variable dependiente. Para los modelos de variable dependiente discreta no binaria, la probabilidad de que el individuo i elija la alternativa j está dada por:

$$P_{ji} = Prob\left(y_i = j\right) = F_j\left(\boldsymbol{x_i}\gamma\right)$$

Dependiendo de la forma funcional que adopte F_j, se obtendrá un modelo distinto: Logit Multinomial, Probit Multinomial, Logit Condicional, Logit Ordenado, etc. En esta obra, únicamente se abordará el Modelo Logit Multinomial; esto, debido a la disponibilidad de herramientas efectivas y de fácil manejo en las librerías Python especializadas.

Antes de proceder a la labor práctica en Python, es importante que el lector tenga en consideración los tipos de variables explicativas que se emplean en los modelos de variable dependiente discreta no binaria, en cuanto su clasificación resulta relevante; las variables explicativas pueden ser *alternative-invariant regressors* (también llamados *case-specific regressors*) o *alternative-variant regressors* (también llamados *alternative-specific regressors*).

Las variables explicativas de tipo *alternative-invariant regressors* no tienen correspondencia directa con determinada alternativa; los valores de estas variables varían entre individuos (entidades) pero no varían entre alternativas para el mismo individuo (entidad). Retomando el ejemplo de la cerveza que hemos tratado con anterioridad, la variable *Ingreso* es una variable *alternative-invariant* en cuanto tiene el mismo valor para el mismo individuo y no cambia dependiendo de la alternativa elegida; para el Individuo 1, el ingreso sigue siendo COP 2,300,000 sin importar cuál sea la marca de cerveza que elija; para el Individuo 2, el ingreso sigue siendo COP 3,000,000 sin importar cuál sea la marca de cerveza que elija, etc.

Las variables explicativas de tipo *alternative-variant regressors* tienen correspondencia directa con determinada alternativa, por lo que su valor depende de ésta; los valores de estas variables varían entre alternativas y también pueden variar entre individuos. Una vez más, recurriendo al ejemplo del consumo de cerveza, el precio es una variable *alternative-variant*, en cuanto tiene distintos valores para distintas alternativas; asimismo, se presenta cierta variación entre individuos. Siendo un tanto más explícitos, el lector puede observar que el precio de 'Águila' no es el mismo que el de 'Corona' o el de 'Póker'; a esto se hace referencia con *alternative-variant*, pues, aunque se trata de la misma característica (precio), su valor depende de la alternativa correspondiente.

Un aspecto adicional a considerar es que las variables explicativas tipo *alternative-invariant regressors* se emplean en el Modelo Logit Multinomial y las variables explicativas tipo *alternative-variant regressors* se utilizan en el Modelo Logit Condicional y Logit Mixto. En la labor práctica que adelantaremos con Python, tal y como se ha señalado de forma previa, abordaremos *únicamente* el Modelo Logit Multinomial, por lo que solo se emplearán variables explicativas tipo *alternative-invariant* durante el proceso de construcción y estimación.

Retornando a la descripción del Modelo Logit Multinomial, la forma funcional adoptada es la siguiente:

$$P_{ji} = Prob\left(y_i = j\right) = \frac{e^{w_i^T \gamma_j}}{\sum_{h=1}^{m} e^{w_i^T \gamma_h}}$$

en donde P_{ji} es la probabilidad de que el individuo (entidad) i elija la alternativa j, $\boldsymbol{w_i}$ es el vector de variables *alternative-invariant* para el individuo (entidad) i y γ_j es el conjunto de coeficientes correspondientes a la alternativa j. Para lograr la estimación de la probabilidad, uno de los conjuntos de coeficientes se normaliza a 0, de modo que la *interpretación* de los demás se realiza con referencia a la *categoría* ***base*** (a la que corresponde el conjunto de coeficientes 0).

Un punto a destacar es que la magnitud de los coeficientes del Modelo Logit Multinomial, al igual que de los Logit y Probit, no debe interpretarse directamente; estos coeficientes tan solo indican si aumenta (cuando el coeficiente es positivo) o disminuye (cuando el coeficiente es negativo) la probabilidad de que se elija la alternativa j, *en comparación* con la categoría base, pero no cuantifican la variación.

Para contar con una medida que cuantifique el impacto de determinada variable sobre la probabilidad de elegir la alternativa j, se recurre a los efectos marginales, los cuales, para el Modelo Logit Multinomial, están dados por lo siguiente:

$$P_{ji} = \frac{e^{\boldsymbol{w_i^T}\gamma_j}}{e^{\boldsymbol{w_i^T}\gamma_{alt1}} + e^{\boldsymbol{w_i^T}\gamma_{alt2}} + ... + e^{\boldsymbol{w_i^T}\gamma_{altm}}}$$

en donde $altj$ hace referencia a la alternativa j ($alt1$ se refiere a la alternativa 1, $alt2$ se refiere a la alternativa 2 y así sucesivamente). Ahora, definiendo $\boldsymbol{w_i^T\gamma_{alth}}$ como z_{hi} , se tiene:

$$P_{ji} = \frac{e^{z_{ji}}}{e^{z_{1i}} + e^{z_{2i}} + ... + e^{z_{mi}}}$$

Por lo tanto, como $\boldsymbol{z_{hi} = w_i^T\gamma_{alth}} = \gamma_{h0} + \gamma_{h1}w_{1i} + ... + \gamma_{hr}w_{ri}$ [el primer subíndice (en los coeficientes) hace referencia a la alternativa a la que corresponden los coeficientes (m alternativas), el segundo subíndice de los coeficientes (primer subíndice para las variables) hace referencia a la variable *alternative-invariant* (r variables *alternative-invariant*) y el segundo subíndice de las variables hace referencia al individuo (entidad)], entonces se tiene que:

$$\frac{\partial P_{ji}}{\partial w_f} = \frac{\gamma_{jf}e^{z_{ji}}\left(\sum_{h=1}^{m} e^{z_{hi}}\right) - e^{z_{ji}}\left(\sum_{h=1}^{m}\gamma_{hf}e^{z_{hi}}\right)}{\left(\sum_{h=1}^{m} e^{z_{hi}}\right)^2}$$

$$\frac{\partial P_{ji}}{\partial w_f} = \frac{\gamma_{jf}e^{z_{ji}}\left(\sum_{h=1}^{m} e^{z_{hi}}\right)}{\left(\sum_{h=1}^{m} e^{z_{hi}}\right)^2} - \frac{e^{z_{ji}}\left(\sum_{h=1}^{m}\gamma_{hf}e^{z_{hi}}\right)}{\left(\sum_{h=1}^{m} e^{z_{hi}}\right)^2}$$

$$\frac{\partial P_{ji}}{\partial w_f} = \frac{e^{z_{ji}}}{\sum_{h=1}^{m} e^{z_{hi}}}\left(\gamma_{jf} - \frac{\sum_{h=1}^{m}\gamma_{hf}e^{z_{hi}}}{\sum_{h=1}^{m} e^{z_{hi}}}\right)$$

$$\frac{\partial P_{ji}}{\partial w_f} = P_{ji}\left(\gamma_{jf} - \sum_{h=1}^{m} P_{hi}\gamma_{hf}\right)$$

Debe notarse que los efectos marginales no necesariamente poseen el mismo signo que los coeficientes; por lo tanto, los coeficientes indican (no miden) si la probabilidad de elegir la alternativa j aumenta o disminuye *en comparación* con la categoría base, y los efectos marginales cuantifican la variación en la probabilidad de elegir determinada alternativa ante la variación en una variable explicativa (*alternative-invariant*) específica.

Al igual que con los modelos Logit y Probit estudiados en el capítulo anterior, en el Modelo Logit Multinomial, para una misma variable se obtienen n efectos marginales distintos (hay n individuos (entidades)). Por lo tanto, para obtener una medida general del efecto marginal de una variable particular, pueden emplearse los dos "caminos" expuestos en el Capítulo 6 de esta obra (al lector que no tenga claridad al respecto, se sugiere consultar dicho capítulo, en cuanto la lógica aplicada al actual es la misma).

Es importante señalar que los efectos marginales de la misma variable deben sumar cero entre las alternativas, es decir:

$$\sum_{j=1}^{m} \frac{\partial P_j}{\partial w_f} = 0$$

en donde $\frac{\partial P_j}{\partial w_f}$ es el efecto marginal de la variable w_f sobre la probabilidad de elegir la alternativa j y existen m alternativas. Asimismo, es importante traer a colación que sin importar la alternativa que se elija como categoría base, aunque los coeficientes cambien, los efectos marginales serán los mismos.

Hasta este punto se ha dado una muy breve exposición de los fundamentos teóricos sobre los que se estructura el modelo que ahora pretendemos abordar de manera práctica en Python. Es posible que la estructura matemática sobre la que se construye el Modelo Logit Multinomial no sea totalmente comprensible para el lector; sin embargo, puede que la labor práctica le sea de ayuda para tener mayor claridad sobre determinados aspectos.

Modelo Logit Multinomial en Python

El dataset que se utilizará para la labor práctica en Python es el empleado en Katchova (2013, 3) (los datasets usados por la profesora Katchova provienen, en su versión original, de Herriges y King (1999) y Cameron y Trivedi (2005)). El dataset se ha obtenido del sitio web de *Econometrics Academy* https://sites.google.com/site/econometricsacademy/. Lo que se pretende estudiar es cómo afecta el ingreso ('income'), que es una variable de tipo *alternative-invariant*, a la elección del tipo de pesca ('mode') de los individuos.

Las funciones para trabajar con modelos Logit Multinomial pertenecen a la librería *Statsmodels*, la cual viene integrada en la distribución **Anaconda**; asimismo, las demás librerías que brindan soporte para el análisis econométrico que se adelantará (*pandas* y *NumPy*) también vienen integradas en esta distribución, por lo cual no se requiere de la instalación de librerías adicionales por parte del usuario.

Preparación del entorno

La primera celda en la que el usuario escribirá y ejecutará código tendrá el siguiente contenido:

```
In [1]: import numpy as np
        import pandas as pd
        import statsmodels.api as sm
```

El anterior bloque de código permite cargar las herramientas necesarias para llevar a cabo las acciones requeridas en el análisis econométrico planteado; en caso de que el lector no tenga claridad al respecto, se le pide consultar la sección "Preparación del entorno" del Capítulo 2, en donde se brinda una explicación sobre tal cuestión.

Importación de los datos

La importación de los datos se lleva a cabo con funciones de la librería *pandas*. Dependiendo del tipo de archivo en el cual se encuentre almacenada la información del dataset, deberemos recurrir a una función particular y un conjunto de parámetros específicos.

Para nuestro caso concreto, el siguiente código es el que importa el dataset:

```
In [2]: data = pd.read_csv("multinomial_fishing1.csv")
```

Una vez ejecutada la línea de código de esta celda, el dataset debería haber sido importado correctamente. Para comprobarlo, recurrimos al método `.head()`, aplicado al objeto que contiene el dataset (en este caso le hemos asignado el nombre `data`):

```
In [3]: data.head()
```

```
Out[3]:       mode    price  catchrate  d.beach  d.pier  d.private  d.charter  \
        0  charter  182.930     0.5391        0       0          0          1
        1  charter   34.534     0.4671        0       0          0          1
        2  private   24.334     0.2413        0       0          1          0
        3    pier   15.134     0.0789        0       1          0          0
        4  private   41.514     0.1082        0       0          1          0

           price.beach  price.pier  price.private  price.charter  catch.beach  \
        0      157.930     157.930        157.930        182.930       0.0678
        1       15.114      15.114         10.534         34.534       0.1049
        2      161.874     161.874         24.334         59.334       0.5333
        3       15.134      15.134         55.930         84.930       0.0678
        4      106.930     106.930         41.514         71.014       0.0678

           catch.pier  catch.private  catch.charter    income  mode1
        0      0.0503         0.2601         0.5391  7.083332      4
        1      0.0451         0.1574         0.4671  1.250000      4
        2      0.4522         0.2413         1.0266  3.750000      3
        3      0.0789         0.1643         0.5391  2.083333      2
        4      0.0503         0.1082         0.3240  4.583332      3
```

Podemos observar que el proceso de importación ha sido exitoso y la información del dataset ha sido almacenada correctamente en un `DataFrame` de *pandas*. Ahora, podemos continuar tranquilamente con el desarrollo de nuestro análisis, en cuanto hemos concluido satisfactoriamente el primer paso. Procederemos a construir y estimar un modelo logit multinomial, el cual toma como variable dependiente una variable discreta no binaria; por lo tanto, resulta de utilidad conocer las alternativas que contiene dicha variable.

Para conocer las alternativas de la variable dependiente basta con seleccionarla (empleando los corchetes '[]') y aplicarle el método `.unique()`, el cual permite conocer los valores únicos que tiene un objeto. El código a ejecutar es:

```
In [4]: data["mode"].unique()
```

```
Out[4]: array(['charter', 'private', 'pier', 'beach'], dtype=object)
```

El resultado que obtenemos es un `ndarray` de *NumPy*, en el cual se nos informa que los valores únicos (alternativas) que toma la variable dependiente son 'charter', 'private', 'pier' y 'beach'; por lo tanto, contamos con cuatro alternativas y, consecuentemente, obtendremos tres sets de coeficientes (el lector ya debería saber el porqué).

Para nuestro caso es muy sencillo saber el número de alternativas con las que se cuenta, pues basta con enumerarlas; sin embargo, si la variable tuviera muchas alternativas resultaría tedioso contarlas una a una. *pandas* ofrece una solución efectiva a dicho inconveniente: en vez de aplicar el método `.unique()` se utiliza el método `.nunique()`, el cual cuenta el número de valores únicos. Para comprobar que efectivamente se tienen 4 alternativas, se ejecuta el siguiente código:

```
In [5]: data["mode"].nunique()

Out[5]: 4
```

Preparación de los datos

El lector que haya seguido esta obra hasta este punto habrá notado que en los capítulos previos hemos almacenado diversas variables en objetos específicos. Esto lo hemos hecho para agrupar dichas variables de acuerdo a su rol particular en la regresión y tratando de simplificar el código a utilizar posteriormente. Ahora, en este ejercicio, aunque tan solo se emplearán dos variables, se asignarán las variables a objetos específicos para simplificar un tanto la sintaxis y facilitar la comprensión del código. Procederemos a realizar la asignación extrayendo las variables por medio de los corchetes "[ ]", tal y como hemos hecho en capítulos previos:

```
In [6]: Y = data["mode"]
        X = data["income"]
```

Construcción y estimación

Para la construcción y estimación del modelo se emplean la función `MNLogit(...)` de *Statsmodels* y el método `.fit()`; para la visualización de los resultados, la función `print(...)` y el método `.summary()`. El código a ejecutar es el siguiente:

```
In [7]: ModeloLogitMN = sm.MNLogit(Y, sm.add_constant(X))
        ResultadosLogitMN = ModeloLogitMN.fit()
        print(ResultadosLogitMN.summary())

Optimization terminated successfully.
         Current function value: 1.249704
         Iterations 5
                          MNLogit Regression Results
==============================================================================
Dep. Variable:                   mode   No. Observations:                 1182
Model:                        MNLogit   Df Residuals:                     1176
Method:                           MLE   Df Model:                            3
Date:                xxx, xx xxx xxxx   Pseudo R-squ.:                 0.01374
Time:                        xx:xx:xx   Log-Likelihood:                -1477.2
converged:                       True   LL-Null:                       -1497.7
                                        LLR p-value:                 6.093e-09
==============================================================================
mode=charter       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const            1.3413      0.195      6.896      0.000       0.960       1.723
income          -0.0316      0.042     -0.756      0.450      -0.114       0.050
------------------------------------------------------------------------------
 mode=pier       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.8142      0.229      3.561      0.000       0.366       1.262
income        -0.1434      0.053     -2.691      0.007      -0.248      -0.039
------------------------------------------------------------------------------
```

```
mode=private       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const            0.7389      0.197      3.756      0.000       0.353       1.125
income           0.0919      0.041      2.260      0.024       0.012       0.172
==============================================================================
```

- **Nota:** La información con la que trabaja la función `MNLogit(...)` de *Statsmodels* debe estar contenida en un dataset de formato *wide*. En caso de contar con un formato *long*, el dataset debe reorganizarse de modo que, antes de emplear su información en la función `MNLogit(...)` de *Statsmodels*, su formato sea *wide*.

El lector debe recordar que se mencionó que, para la estimación, el set de coeficientes de una alternativa se normaliza a cero y que la interpretación de los coeficientes obtenidos se realiza con referencia a la categoría base, sin embargo, ¿cuál es la categoría base?

Para este ejemplo concreto, la categoría base es la alternativa 'beach', pero ¿por qué? La respuesta se encuentra en la lógica que aplica la función `MNLogit(...)` de *Statsmodels*: cuando se pasa como variable dependiente una variable no codificada por números (el lector debe observar que la variable *mode* no está codificada), la función ejecuta la codificación automáticamente de forma alfabética y toma como categoría base la alternativa que ocupa la primera posición en dicho orden. Así, como las alternativas de la variable *mode* son 'charter', 'private', 'beach' y 'pier', la categoría base es 'beach', pues es la que ocupa la primera posición si se organizan alfabéticamente tales alternativas.

Ahora, la interpretación de los coeficientes se debe realizar con respecto a la alternativa 'beach'. Así, se tiene que *en comparación con* pesca en la playa ('beach'), un ingreso más alto está asociado a una menor probabilidad de pesca en bote alquilado ('charter') o en el muelle ('pier') y a una mayor probabilidad de pesca en bote privado ('private').

Hemos dicho que un mayor ingreso está asociado, en *comparación* con pesca en la playa, a una probabilidad *mayor* o *menor* de determinadas alternativas, pero ¿de cuánto es esta mayor o menor probabilidad? Recuerde que los coeficientes **no** cuantifican la relación, tan solo indican su sentido; para obtener una medida del impacto de determinada variable sobre la probabilidad de elegir una alternativa específica se emplean los efectos marginales.

Para obtener los efectos marginales de un modelo logit multinomial basta con aplicar el método `.get_margeff()` a la instancia de resultados de dicho modelo; para visualizar estos efectos marginales, tan solo se requiere el empleo del método `.summary()` y de la función `print(...)`, así:

```
In [8]: LogitMNMargEff = ResultadosLogitMN.get_margeff()
        print(LogitMNMargEff.summary())
```

```
       MNLogit Marginal Effects
=====================================
Dep. Variable:                   mode
Method:                          dydx
At:                           overall
==============================================================================
  mode=beach      dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
income           0.0002      0.004      0.044      0.965      -0.007       0.008
------------------------------------------------------------------------------
mode=charter      dy/dx    std err          z      P>|z|      [0.025      0.975]
```

```
------------------------------------------------------------------------------
income          -0.0112      0.006     -1.876      0.061      -0.023       0.000
------------------------------------------------------------------------------
 mode=pier       dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
income         -0.0208      0.005     -4.040      0.000      -0.031      -0.011
------------------------------------------------------------------------------
mode=private       dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
income          0.0318      0.005      6.039      0.000       0.021       0.042
==============================================================================
```

De este modo, se tiene que un incremento de una unidad en el ingreso está asociado a que la pesca en la playa ('beach') sea 0.02 puntos porcentuales más probable; a que la pesca en bote alquilado ('charter') sea 1.12 puntos porcentuales menos probable; a que la pesca en el muelle ('pier') sea 2.08 puntos porcentuales menos probable; y a que la pesca en bote privado ('private') sea 3.18 puntos porcentuales más probable. (El lector puede comprobar que la suma de estos efectos marginales es cero)

El lector debe tener en cuenta que existen varias maneras de obtener una medida general del efecto marginal de determinada variable; los efectos marginales que son calculados por defecto por el método `.get_margeff()` son los efectos marginales promedio. Si lo que se quiere conocer son los efectos marginales en la media, se debe asignar el valor de 'mean' al parámetro *at* del método `.get_margeff()`, así:

```
In [9]: LogitMNMargEff = ResultadosLogitMN.get_margeff(at = "mean")
        print(LogitMNMargEff.summary())
```

```
       MNLogit Marginal Effects
=====================================
Dep. Variable:                   mode
Method:                          dydx
At:                              mean
==============================================================================
  mode=beach       dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
income        7.496e-05      0.004      0.019      0.985      -0.008       0.008
------------------------------------------------------------------------------
mode=charter       dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
income          -0.0120      0.006     -1.977      0.048      -0.024      -0.000
------------------------------------------------------------------------------
 mode=pier       dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
income         -0.0207      0.005     -4.239      0.000      -0.030      -0.011
------------------------------------------------------------------------------
mode=private       dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
income          0.0326      0.006      5.727      0.000       0.021       0.044
==============================================================================
```

Ahora, se tiene que un incremento de una unidad en el ingreso está asociado a que la pesca en la playa ('beach') sea 0.008 p.p más probable; a que la pesca en bote alquilado ('charter') sea 1.2 p.p menos probable; a que la pesca en el muelle ('pier') sea 2.07 p.p menos probable; y a que la pesca en bote privado ('private') sea 3.26 p.p más probable. (Nuevamente, el lector puede comprobar que la suma de estos efectos marginales es cero).

Si el lector contrasta los resultados obtenidos con los de Katchova (2013, 3) notará que los que ella obtiene son diferentes; esto se debe a que en el ejercicio desarrollado por la profesora Katchova no se toma como categoría base 'beach' sino 'charter', ¿es posible que nosotros hagamos lo mismo? La respuesta es sí.

La función `MNLogit(...)` de *Statsmodels* toma como categoría base la alternativa que ocupa la primera posición en orden alfabético o numérico y no es posible modificar tal lógica en su funcionamiento. Sin embargo, si se recodifican las alternativas de modo que la que se desee tener como categoría base ocupe la primera posición, entonces la función `MNLogit(...)` de *Statsmodels* sí asumirá dicha alternativa como la categoría base.

A modo de ejemplo, recurriendo a herramientas de *pandas*, se puede codificar una variable categórica usando el método `.map(...)` y empleando un **diccionario Python** "{...}" como argumento; dentro de dicho diccionario las **claves** corresponderán a las alternativas y los **valores** a los códigos asignados a cada una.

Por ejemplo, a continuación se crea una variable 'mode2' que corresponde a la variable 'mode' codificada de modo que la alternativa 'charter' se encuentre en la primera posición en orden numérico y, por ende, sea tomada como categoría base por la función `MNLogit(...)` de *Statsmodels*:

```
In [10]: data["mode2"] = data["mode"].map({"charter":"1_charter",
                                            "beach":"2_beach",
                                            "pier":"3_pier",
                                            "private":"4_private"})
```

Empleando la misma estructura de código que hemos utilizado con anterioridad, se obtiene el siguiente modelo:

```
In [11]: Y = data["mode2"]
         ModeloLogitMN = sm.MNLogit(Y, sm.add_constant(X))
         ResultadosLogitMN = ModeloLogitMN.fit()
         print(ResultadosLogitMN.summary())

Optimization terminated successfully.
         Current function value: 1.249704
         Iterations 5
                          MNLogit Regression Results
==============================================================================
Dep. Variable:                  mode2   No. Observations:                 1182
Model:                        MNLogit   Df Residuals:                     1176
Method:                           MLE   Df Model:                            3
Date:                xxx, xx xxx xxxx   Pseudo R-squ.:                 0.01374
Time:                        xx:xx:xx   Log-Likelihood:                -1477.2
converged:                       True   LL-Null:                       -1497.7
                                        LLR p-value:                 6.093e-09
==============================================================================
  mode2=2_beach       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const              -1.3413      0.195     -6.896      0.000      -1.723      -0.960
```

```
income              0.0316      0.042      0.756      0.450      -0.050       0.114
------------------------------------------------------------------------------------
mode2=3_pier       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------
const           -0.5271      0.178     -2.965      0.003      -0.876      -0.179
income          -0.1118      0.044     -2.541      0.011      -0.198      -0.026
------------------------------------------------------------------------------------
mode2=4_private       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------
const              -0.6024      0.136     -4.426      0.000      -0.869      -0.336
income              0.1235      0.028      4.426      0.000       0.069       0.178
====================================================================================
```

Como se puede apreciar, las magnitudes de los coeficientes han cambiado y, además, ahora su interpretación no se debe hacer con referencia a 'beach' sino a 'charter'. Así, se tiene que, *en comparación con* pesca en bote alquilado ('charter'), un ingreso más alto está asociado a una mayor probabilidad de pesca en la playa ('beach') y en bote privado ('private') y a una menor probabilidad de pesca en el muelle ('pier'). ¿Qué hay de los efectos marginales?

El lector debe recordar que, sin importar cuál sea la categoría base, el efecto marginal de una variable específica, a diferencia del coeficiente, siempre será el mismo. Para comprobarlo, basta con aplicar el método `.get_margeff()` a la instancia de resultados del modelo:

```
In [12]: LogitMNMargEff = ResultadosLogitMN.get_margeff()
         print(LogitMNMargEff.summary())
```

```
       MNLogit Marginal Effects
=====================================
Dep. Variable:                  mode2
Method:                          dydx
At:                           overall
====================================================================================
mode2=1_charter      dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------
income             -0.0112      0.006     -1.876      0.061      -0.023       0.000
------------------------------------------------------------------------------------
mode2=2_beach      dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------
income            0.0002      0.004      0.044      0.965      -0.007       0.008
------------------------------------------------------------------------------------
mode2=3_pier      dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------
income          -0.0208      0.005     -4.040      0.000      -0.031      -0.011
------------------------------------------------------------------------------------
mode2=4_private      dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------
income              0.0318      0.005      6.039      0.000       0.021       0.042
====================================================================================
```

Se puede observar que los efectos marginales obtenidos del modelo con categoría base 'charter' son exactamente iguales a los obtenidos del modelo con categoría base 'beach' (puede examinarlos uno por uno si así lo desea). Recordando que el método `.get_margeff()` calcula por defecto efectos marginales *promedio*, si lo que se desea conocer son los efectos marginales *en la media*, tan solo se requiere asignar el valor de 'mean' al parámetro *at*:

```
In [13]: LogitMNMargEff = ResultadosLogitMN.get_margeff(at = "mean")
         print(LogitMNMargEff.summary())
```

```
       MNLogit Marginal Effects
=====================================
Dep. Variable:                  mode2
Method:                          dydx
At:                              mean
================================================================================
mode2=1_charter      dy/dx    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------------
income             -0.0120      0.006     -1.977      0.048      -0.024      -0.000
--------------------------------------------------------------------------------
mode2=2_beach      dy/dx    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------------
income          7.496e-05      0.004      0.019      0.985      -0.008       0.008
--------------------------------------------------------------------------------
mode2=3_pier       dy/dx    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------------
income           -0.0207      0.005     -4.239      0.000      -0.030      -0.011
--------------------------------------------------------------------------------
mode2=4_private      dy/dx    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------------
income              0.0326      0.006      5.727      0.000       0.021       0.044
================================================================================
```

Nuevamente, los efectos marginales en la media son exactamente iguales para el modelo con categoría base 'beach' y para el modelo con categoría base 'charter', igualdad que se mantiene para cualquier otra alternativa de la variable dependiente (empleando, obviamente, los mismos datos).

En este punto se cierra este capítulo, con el cual se concluye esta breve obra, la cual ha buscado ofrecer ejemplos ilustrativos de análisis econométrico de nivel básico con el lenguaje de programación **Python** y en la que se han presentado algunas de las herramientas más relevantes para dicho ejercicio, tratando de brindar al lector breves, pero útiles, descripciones del funcionamiento de cada una de estas.

Referencias

- Chatterjee, S., y Simonoff, J. (2013). *Handbook of Regression Analysis*. Estados Unidos: John Wiley & Sons, Inc.
- Dormann, C.F., Elith, J., Bacher, S., Buchmann, C., Gudrun, C., Carré, G., García, J.R., Gruber, B., Lafourcade, B., Leitão, P.J., Münkemüller, T., McClean, C., Osborne, P.E., Reineking, B., Schröder, B., Skidmore, A.K., Zurell, D., y Lautenbach, S. (2013). Collinearity: a review of methods to deal with it and a simulation study evaluating their performance. *Ecograpghy*, 36, 27-46. doi: 10.1111/j.1600-0587.2012.07348.x
- Fahrmeier, L., Kneib, T., y Lang, Stefan. (2007). *Regression. Modelle, Methoden und Anwendungen*. Springer.
- Greene, W.H. (2003). *Econometric Analysis*. New Jersey, Estados Unidos: Pearson Education, Inc.
- Gujarati, D.N. y Porter, D.C. (2010). *Econometría*. México: McGraw-Hill/Interamericana Editores, S.A. de C.V.
- James, G., Witten, D., Hastie, T., y Tibshirani, R. (2013). *An Introduction to Statistical Learning with Applications in R*. doi: 10.1007/978-1-4614-7138-7
- Katchova, A. (2013). *Econometrics – Probit and Logit Models*. Recuperado de: https://docs.google.com/file/d/0BwogTI8d6EEidjg2OGl4b3dxVmc/edit
- Katchova, A. (2013,2). *Econometrics – Multinomial Probit and Logit Models*. Recuperado de: https://docs.google.com/file/d/0BwogTI8d6EEiQWVIaV9NbjVGeHM/edit
- Katchova, A. (2013, 3). *Econometrics – Multinomial Probit and Logit Models, Conditional Logit Model, Mixed Logit Model Examples*. Recuperado de: https://docs.google.com/file/d/0BwogTI8d6EEiLVZ0N1h3LTBLYlk/edit
- Kleiber, C., y Zeileis, A. (2008). *Applied Econometrics with R*. Estados Unidos: Springer Science+Business Media, LLC.
- Murray, L., Nguyen, H., Lee, Y-F., Remmenga, M., y Smith, D.W. (2012). Variance Inflation Factors in Regression Models with dummy variables. *Conference on Applied Statistics in Agriculture*. doi: 10.4148/2475-7772.1034
- Pandas. (s.f). *Python Data Analysis Library*. Recuperado de: https://pandas.pydata.org/
- Perktold, J., Seabold, Skipper., y Taylor, J. (2018). *Welcome to Statsmodels's Documentation*. Recuperado de: https://www.statsmodels.org/stable/index.html
- Project Jupyter. (2019). *The Jupyter Notebook*. Recuperado de: https://jupyter.org/

- Python Software Foundation. (s.f.). *What is Python? Executive Summary*. Recuperado de: https://www.python.org/doc/essays/blurb/
- Razali, N., y Wah, Y. (2011). Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. *Journal of Statistical Modeling and Analytics*, 2(1), 21-33. Recuperado de: http://www.de.ufpb.br/ulisses/disciplinas/normality_tests_comparison.pdf
- The Matplotlib development team. (2018). *Matplotlib*. Recuperado de: https://matplotlib.org/#
- The SciPy community. (2019). *What is NumPy*? Recuperado de: https://www.numpy.org/devdocs/user/whatisnumpy.html
- Vaart, A.W. van der. (1998). *Asymptotic Statistics*. Reino Unido: Cambridge University Press.
- Verbeek, M. (2004). *A guide to modern econometrics*. Inglaterra: John Wiley & Sons Ltd.
- Waskom, M. (2018). *An introduction to seaborn*. Recuperado de: https://seaborn.pydata.org/introduction.html
- Wooldridge, J.M. (2010). *Introducción a la econometría. Un enfoque moderno*. México: Cengage Learning Editores, S.A. de C.V.
- Würtz, D., y Katzgraber, H.G. (2009). Precise finite-sample quantiles of the Jarque-Bera adjusted Lagrange multiplier test. *ETH Econohysics Working and White Papers Series*. Recuperado de: https://www.rmetrics.org/sites/default/files/2009-02-jarqueberaTest.pdf