UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Guía metodológica introductoria a la ciencia de datos

Santiago Calderón Piedras

Universidad Distrital Francisco José de Caldas
Facultad de ingeniería, Proyecto curricular de Ingeniería Industrial
Bogotá D.C., Colombia
2020

Guía metodológica introductoria a la ciencia de datos

Santiago Calderón Piedras
Cód. 20121015043

Pasantía presentada como requisito parcial para optar al título de:

Ingeniero Industrial

Director interno:

Doctor José Nelson Pérez Castillo

Director externo:

Doctor Nelson Enrique Vera Parra

Universidad Distrital Francisco José de Caldas

Facultad de ingeniería, Proyecto curricular de Ingeniería Industrial

Bogotá D.C., Colombia

2020

A mi madre, por la infinita paciencia, amor y el constante apoyo en todo momento.

Resumen

La ciencia de datos en la actualidad se ha convertido en un campo multidisciplinario muy importante para las empresas que coleccionan grandes cantidades de datos y que comprenden que el análisis de estos las lleva a descubrir información muy útil para la toma de sus decisiones.

Teniendo en cuenta la relevancia actual de la ciencia de datos, se plantea una guía de acercamiento a dicho campo por medio del aprendizaje del lenguaje de programación *Python* apoyado en la distribución Anaconda, abarcando capítulos temáticos en los que se explica mediante ejemplos, procesos característicos en el tratamiento de datos tales como: Importación de datos, preparación de los datos y análisis estadístico de los mismos a través de herramientas básicas del *machine learning* (aprendizaje automático), con las cuales se puede obtener información al aplicar dichas herramientas a los datos para inferir o predecir sobre los mismos.

Para apoyar el objetivo primordial del aprendizaje, se presentan definiciones y fórmulas matemáticas de los modelos o algoritmos empleados en el aprendizaje automático sin tocar las demostraciones o la complejidad que hay detrás de estos. Otros elementos que se encuentran para el apoyo del aprendizaje son propuestas de ejercicios en cada unidad y videos anexos para retroalimentar los temas trabajados en cada unidad, con el fin de reforzar y recordar los conocimientos adquiridos.

Palabras clave: Lenguaje de programación, Análisis de datos, Estadística, Aprendizaje automático.

Abstract

Nowadays data science has become into a very important multidisciplinary field for companies that collect large amounts of data and which understand that the analysis of these data leads to discover very useful information for making their decisions.

Taking into account the current relevance of data science, a guide to approach this field is proposed through the learning of the Python programming language supported by the Anaconda distribution, covering thematic chapters in which it is explained by examples, characteristic processes in the processing of data such as data import, data preparation and statistical analysis through the basic tools of machine learning, with which it is possible to obtain information by applying these tools to the data to infer or predict about them.

To support the primary objective of learning, definitions and mathematical formulas of the models or algorithms used in machine learning are presented without touching the demonstrations or the complexity behind them. Other elements that are found to support learning are proposed exercises in each unit and attached videos to feed back the topics worked on each unit, in order to reinforce and remember the knowledge acquired.

Keywords: Programming Languages, Data handling, Statistics, Machine learning.

Contenido

Lista de figuras

Pág.

Lista de gráficos

Lista de tablas

Pág.

Introducción

El desarrollo de la tecnología, la aparición del concepto de *Big data* (Datos a gran escala) ha causado que los analistas de datos requieran una combinación entre ciencias de la computación, estadística-matemática y análisis de negocios para encontrar soluciones prácticas, precisas, robustas, de forma científica y sistemática a bajo costo en la toma de decisiones.

En consecuencia, sí el trabajo en torno a datos de gran escala denota una práctica científica, esto da lugar a la aparición del concepto de científico de datos como una nueva profesión que es muy valorada en el siglo XXI (Ajay, 2018, p. 6). Por ende, una forma de acercarse a este campo es por medio del aprendizaje de lenguajes de programación aplicados al análisis de datos para fortalecer las capacidades del analista, con el fin de que pueda generar valor en la toma de decisiones en las organizaciones cuando se maneja una gran cantidad de datos.

En la actualidad la tenencia datos tiene un gran valor en el contexto empresarial, tanto que se han denominado en el siglo XXI como el “Nuevo oro”. Existen muchas empresas cuyo valor recae en la cantidad de datos que poseen, como lo son aquellas que crean redes sociales como *Facebook Inc*. o *Instagram Inc*., toda vez que contienen valiosa información sobre el comportamiento, gustos y relaciones entre los usuarios registrados, lo que puede ser utilizado por otras empresas para ofrecer productos o servicios basados en dicha información.

Para el análisis de los datos se requiere comprender lenguajes de programación que contengan módulos o extensiones para el tratamiento estadístico de la información. El lenguaje más conocido para esto es “R”, un lenguaje de programación enfocado en estadística que contiene gran cantidad de paquetes con modelos estadísticos y por ende es muy utilizado en la enseñanza de la ciencia de datos. Además de “R” se encuentra *Python*, el cual contiene módulos para el cálculo matemático, tratamiento de datos, así

como un módulo de aprendizaje automático llamado "*Scikit"* para el análisis de datos y la minería de datos.

A través de los ejemplos de código ejecutado en *Python*, se espera que la guía sea una forma de aprender a trabajar con este lenguaje de programación, enfocada en el análisis de datos, de tal manera que el usuario pueda ejecutar diferentes modelos estadísticos y algoritmos para predecir o inferir sobre un conjunto de datos. La teoría que existe detrás de los modelos y algoritmos es bastante compleja, por ello se aclara que el objetivo de esta guía no es explicar o demostrar los modelos en los que se basa, pero sí en la comprensión y en la ejecución de los resultados.

No obstante, gran parte del código que se presenta puede quedar obsoleto, debido a actualizaciones posteriores que se hagan al lenguaje de programación; por ello es importante estar atento a las mismas y conocer de antemano sí se va a reemplazar funciones o se van a simplificar.

El concepto de ciencia de datos

Longbing Cao (2017) en un apartado de su artículo *"Data science: a comprehensive overview"* resume la evolución del concepto de análisis de datos hasta llegar al concepto ciencia de datos en donde dicho concepto fue mencionado por primera vez en el estudio "*Concise Survey of Computer Methods"* cuyo autor es Peter Naur, define la ciencia de datos como "la ciencia encargada de tratar datos una vez han sido establecidos, sin que esto implique describir lo que estos podrían representar; ya que ello dependía de los campos de conocimiento interesados en ello."

Sin embargo, antes de la aparición del concepto de "ciencia de datos", el concepto utilizado era el de "análisis de datos" el cual fue trabajado por John Tukey quien mencionaba que era una ciencia intrínsecamente empírica. Más adelante se incorporaría el procesamiento de datos y el análisis exploratorio de los mismos junto con la estadística descriptiva, pero faltaba más énfasis para formular y aplicar pruebas de hipótesis. Por tal razón, se necesitaba una expansión de las técnicas estadísticas, lo que sería la aproximación a la ciencia de datos.

De este modo, el taller "*Knowledge Discovery in Databases*" simplificado como "KDD" dio a origen al concepto de "*Data Driven Discovery*", a partir del cual se divulgaron los conceptos de minería de datos y de extracción de conocimientos, definidos como el proceso de encontrar conocimiento interesante escondido entre los datos.

Así las cosas, la unión entre la estadística descriptiva y el análisis de datos, junto con la minería de datos, la extracción de conocimiento y el aprendizaje automático formaron la analítica de datos definida como "un campo multidisciplinario basado en la evaluación cualitativa y cuantitativa de los datos con el objetivo de crear conclusiones o la comprensión exploratoria-predictiva y para extraer y probar hipótesis de la información para la toma de decisiones". La aplicación de la analítica de datos en diferentes campos como el social, comportamiento, negocios, riesgo o dominios específicos y su teoría son el punto clave de la ciencia de datos.

Data science (ciencia de datos) es un término acuñado por muchos autores. Algunas de las definiciones más representativas se presentan a continuación:

Al respecto, Claus Weihs y Katja Ickstadt (2018) expresan:

"La ciencia de datos como una disciplina científica está influenciada por la informática, ciencia computacional, matemáticas, investigación de operaciones y estadística, así como las ciencias aplicadas […]. A pesar de ser un término fundado por estadísticos, en la Figura pública de la ciencia de datos, la importancia de la ciencia computacional y las aplicaciones en negocios están más frecuentemente ligadas, en especial en la era los datos a gran escala […].La estadística en la ciencia de datos es una de las disciplinas más importantes para proveer herramientas, métodos para encontrar una estructura, dar una visión más profunda de los datos y es la disciplina más importante para analizar y cuantificar la incertidumbre".

Otra definición de ciencia de datos es dada por Longbing Cao (2017) en la que define:

"Desde una perspectiva disciplinaria en que la ciencia de datos es un nuevo campo interdisciplinario que sintetiza y se construye desde la estadística, informática, computación, comunicación, gestión y sociología para estudiar los datos y sus entornos

(incluyendo dominios y otros aspectos contextuales como el organizacional y aspectos sociales) con el fin de transformar los datos en ideas y decisiones, siguiendo un pensamiento y una metodología del dato al conocimiento y de éste a la sabiduría, con lo cual define la siguiente fórmula para definir ciencia de datos":

$$Ciencia\ de\ datos = (Estadística + Informática + Computación + Comunicación + Sociología + Gestión)|(Datos + Entorno + Pensamiento).$$

Vale la pena aclarar que en la formula anterior, el autor en mención, explica que el operador "|" indica "condicionadas por".

Por su parte, Ajay Ohri (2018) Expresa: "La ciencia de datos recae en la intersección entre la programación, la estadística y el análisis de negocios. Es el uso de herramientas de programación con técnicas estadísticas para analizar datos de modo sistemático y científico" (p. 3).

La ciencia de datos está ligada a muchos otros conceptos como por ejemplo el de datos a gran escala. Al respecto, Jeffrey S. Saltz y Neil Dewar (2019) asociaron este concepto con el de ciencia de datos como:

"La ciencia de datos es una disciplina emergente que involucra el análisis de datos para resolver problemas y desarrollar ideas. Los datos a gran escala están relacionados con la ciencia de datos, porque en los datos a gran escala, los conjuntos de datos o tablas de datos son tan grandes y/o complejos que el análisis de datos con las técnicas tradicionales no es viable".

En el mismo sentido, Saltz y Stanton, citados por Saltz y Dewar (2019), explican que incluye "la colección, preparación, análisis, visualización, gestión y conservación de grandes colecciones de información".

Por su parte, Takemura (2018) definiendo la ciencia de datos recalca la importancia que tiene como ventaja competitiva para las empresas:

La ciencia de datos es la metodología de procesar, analizar y extraer valor de los datos a gran escala y sus practicantes son llamados científicos de datos.
Las compañías que tienen datos y habilidades en ciencia de datos tienen una ventaja competitiva. Esta es la razón del crecimiento de los gigantes de internet y estas compañías están contratando muchos científicos de datos e invirtiendo grandemente en algoritmos para manipular y analizar datos a gran escala.

Otro aspecto importante que resalta Takemyra (2018) en la ciencia de datos dice que:

"Otra importante diferencia entre la estadística tradicional y la ciencia de datos es la creciente importancia de los datos no estructurados en la ciencia de datos, como datos de texto, en imágenes y datos en sonidos. Los mensajes de texto son importantes fuentes de información con respecto a cómo las personas piensan y actúan".

Lenguaje de programación *Python*

Python es un lenguaje de programación dinámico, frecuentemente visto como un lenguaje de *scripting* (guiones), pero es también un lenguaje de programación de propósito general con el que se realizan desde simples guiones hasta grandes servidores web.

Según (Rossum, 2009), *Python* se usa para desarrollar aplicaciones para interfaces graficas de usuario, programación de bases de datos, programación web del lado del cliente y del servidor y para testear aplicaciones

La fundación (Python Software Foundation, 2019) presenta un resumen ejecutivo de su lenguaje, en el que se encuentra:

Python es un lenguaje de programación interpretado, orientado a objetos, de alto nivel con semánticas dinámicas. […] Las simples reglas de la sintaxis facilitan su legibilidad y la reducción del costo de mantenimiento del programa. *Python* soporta módulos y paquetes que favorecen la modularidad del programa y la reutilización de código. El interpretador de *Python* y la extensa librería estándar están disponibles en formato fuente o binario sin cargo para todas las plataformas principales, y se pueden distribuir libremente.

Adicionalmente, la librería estándar de *Python* permite escoger entre una gran cantidad de módulos de acuerdo a las necesidades del usuario, permitiendo añadir más funcionalidades sin necesidad de escribir más código.

Python también está diseñado para facilitar el análisis y la visualización de datos a partir de complejas aplicaciones científicas y matemáticas. Por ejemplo, proporcionando soluciones específicas a problemas de datos a gran escala sin requerir de demasiado tiempo o esfuerzo (Mindfire Solutions, 2017).

Distribución Anaconda

La distribución anaconda es la plataforma más conocida en el mundo para trabajar ciencia de datos, aprendizaje de maquina e inteligencia artificial con *Python* y “R”. Contiene más de 1500 paquetes para los propósitos mencionados anteriormente dentro de los que se encuentran *scikit*, *numpy, pandas, matplotlib etc.* (Anaconda Inc, 2019).

Con esta distribución se puede utilizar *Jupyter Notebook* que va más allá de una consola hacia un enfoque de computación interactiva a través de una aplicación basada en la web que realiza todo el proceso de la computación: desarrollo, documentación, ejecución de código y comunicación de los resultados del mismo.

Combina tiene dos componentes que son: ser una aplicación web basada en un navegador y construirse bajo documentos *notebooks* que son la representación de todo el contenido (Jupyter Team, 2015).

Jupyter utiliza como núcleo *IPython* el cual es definido por *Jupyter Team* (2019) como una extensión al lenguaje de programación Python, que permite un trabajo interactivamente conveniente y eficiente. Estas extensiones están implementadas en el núcleo *IPython* y están disponibles en todas las ramas de *IPython (Notebook, Terminal, Console y Qt Console)* cuando se ejecuta este núcleo.

Objetivos de la pasantía

Objetivo principal

Generar una guía metodológica introductoria a la ciencia de datos.

Objetivos específicos

- Definir y contextualizar la guía metodológica elaborando y desarrollando las unidades temáticas particulares en concordancia con el objetivo general.

- Introducir y aplicar el lenguaje de programación Python para hacer un acercamiento a la ciencia de datos.

- Enseñar y aplicar fundamentos de estadística en la ciencia de datos.

- Elaborar videos prácticos de apoyo para el aprendizaje de la ciencia de datos.

- Generar ejemplos y propuestas de ejercicios como complemento necesario para el aprendizaje.

1. Capítulo 1. Python Básico

Este capítulo se enfoca en introducir las operaciones y construcción de líneas de código elementales a la hora de aprender un lenguaje de programación como *Python* y se aprenderán elementos básicos en programación como:

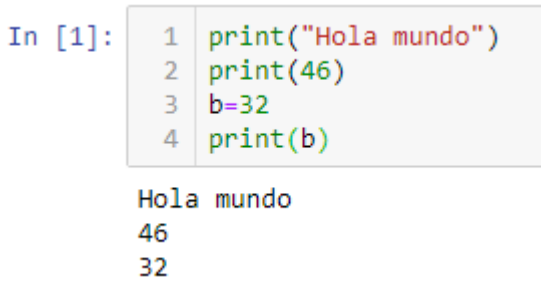
- Imprimir cadenas de texto.
- Realizar operaciones matemáticas.
- Declarar y utilizar de diferentes tipos variables.
- Utilizar estructuras de datos como listas, tuplas y diccionarios.
- Utilizar las estructuras de control "*for"*, *"while"* e "*if".*
- Definir y aplicar funciones.
- Manipular listas.
- Realizar operaciones del algebra lineal con arreglos de dos dimensiones.

1.1 Impresión de resultados

A la hora de aprender un lenguaje de programación, lo más elemental es mostrar un resultado en pantalla. Para presentar resultados se puede utilizar la función "*print"*, la cual no solamente imprime caracteres, sino que también puede evaluar una expresión antes de imprimir un resultado.

Al utilizar texto, éste se debe incluir entre comillas dobles o simples. Otros tipos de datos como variables y números no se colocan entre comillas ya que serían convertidos en caracteres. Es importante no confundir con las comillas gramaticales. Aunque es claro el uso diferenciad de ambas. A continuación, se muestra un ejemplo en el que se incluye un texto, un número y una variable.

Figura 1-1. Uso de la función "*print"* con caracteres, valores numéricos y variables.



Fuente: Elaboración propia con *Jupyter Notebook.*

Esta función se usa para mostrar los resultados de operaciones en la medida que el practicante cree funciones para aplicar modelos o fórmulas o algoritmos, de este modo mostrará en pantalla múltiples salidas y podrá acompañarlos con texto para hacer más claros los resultados.

1.1.1 *Python* como calculadora: operadores aritméticos

Las operaciones básicas en *Python* se realizan con los operadores aritméticos que se muestran en la siguiente tabla:

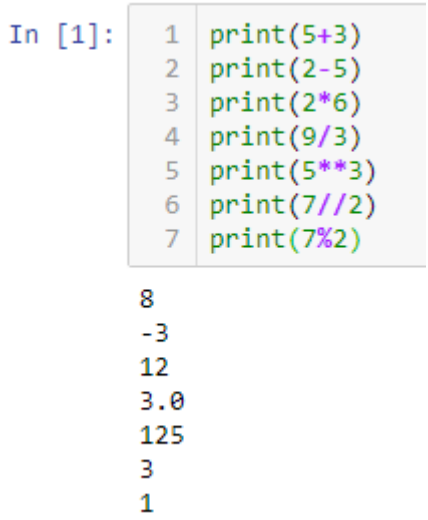
Tabla 1. Operadores aritméticos básicos en *Python.*

Suma	+
Resta	-
Multiplicación	*
División	/
Potencia	**
Residuo	%
Parte entera truncada	//

Fuente: Elaboración propia a partir de los datos de Mueller John. 2018, p. 118.

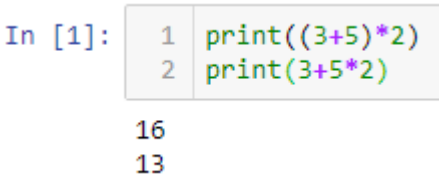
Con estos operadores se pueden ejecutar las operaciones aritméticas básicas. Es importante tener en cuenta que se puede modificar la precedencia de las operaciones usando paréntesis. Existe un orden de precedencia específico en las operaciones por lo que es conveniente utilizarlos en caso de no conocer la precedencia.

Figura 1-2. Ejemplos con los operadores aritméticos de Python.



Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 1-3. Modificación de precedencia en operaciones matemáticas.



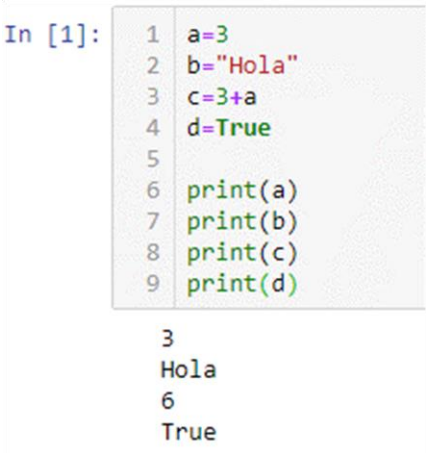
Fuente: Elaboración propia mediante *Jupyter Notebook.*

1.2 Definición de variables

Las variables en *Python* se crean a partir del uso de otro tipo de operadores denominados de asignación. El operador empleado para definir una variable es el igual (=). Para hacerlo

se debe dar a la variable un nombre, luego emplear el operador mencionado y finalmente asignar un valor numérico deseado, una cadena de caracteres, un valor lógico, o incluso otra variable.

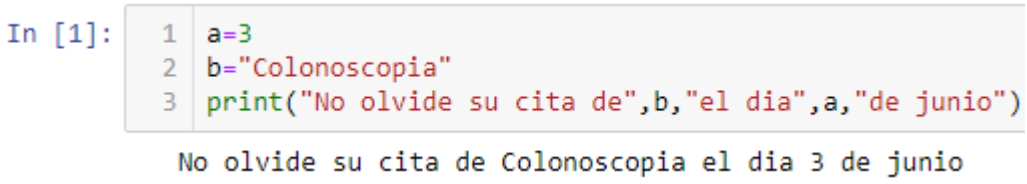
Figura 1-4. Ejemplos de asignación de variables.



Fuente: Elaboración propia mediante *Jupyter Notebook.*

Es común desear imprimir el valor de una variable junto con texto, para ello se deben separar con comas dentro de la función "*print":*

Figura 1-5. Impresión de variables y texto.



Fuente: Elaboración propia con *Jupyter Notebook.*

De las imágenes anteriores se encuentra que las variables pueden ser de varios tipos, tales como lo son numéricas, caracteres o valores lógicos.

1.2.1 Tipo de dato numérico

En cuanto a los tipos de datos numéricos, se encuentran las variables de tipo "*integer"* (entero*)* y de tipo "*float"* (*flotante)*.

Un ejemplo de dato tipo entero es 100, mientras que uno de tipo flotante sería 100.0. Un dato entero no posee números decimales mientras que uno flotante sí.

Los números guardados como enteros utilizarán menos espacio comparado con los números guardados como flotantes, esto quiere decir que los números flotantes pueden almacenar mayor cantidad de dígitos, comparado con un número declarado como entero. El tipo de variable puede modificarse mediante las funciones "*int"* para hacerla entera o "*float"* para hacerla flotante*.*

1.2.2 Tipo de dato carácter

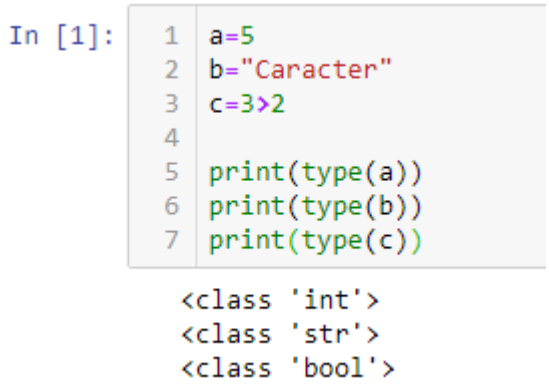
Los datos de tipo "*string"* (carácter) son cadenas de caracteres colocadas entre comillas que para un ordenador están representadas por números en memoria (Mueller, 2018, p. 110). Un dato numérico puede ser convertido a carácter mediante la función "*str".*

1.2.3 Tipo de dato lógico

Los datos de tipo "*boolean"* (dato lógico) sólo puede contener uno de estos dos valores: "*True"* o "*False"*. El dato lógico es utilizado cuando se comparan valores para determinar si condiciones dadas son verdaderas o falsas.

Python en su librería estándar cuenta con la función "*type"* que permite conocer el tipo de un determinado objeto:

Figura 1-6. Uso de la función "*type"* para conocer el tipo de varias variables.



Fuente: Elaboración propia con *Jupyter Notebook.*

A partir de las variables se puede almacenar información, solicitarla o modificarla con el fin de analizarla.

1.3 Estructuras de datos básicas

Una forma más adecuada de almacenar la información es a través de listas, "*tuplas"* y diccionarios, en ellos, la variable se compone de un conjunto ordenado de datos. Para crear estas estructuras se debe dar nombre a una variable, utilizar el operador de declaración igual y dependiendo de si se usan paréntesis, corchetes o llaves se puede crear:

1.3.1 Listas

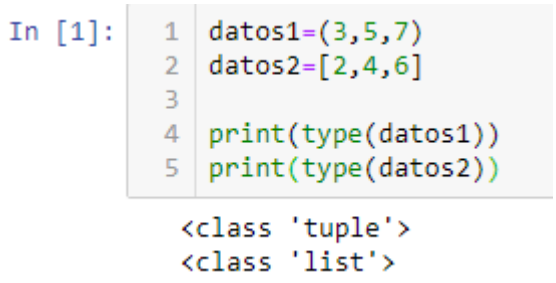
Una "*list"* (lista) es un conjunto de datos que pueden ser de diferente tipo, es decir, que puede contener números enteros, decimales, caracteres, valores booleanos entre otros, cuyos elementos son modificables o mutables y que se encuentran indexados, es decir que los datos se enumeran comenzando desde 0. Para crear listas se usan corchetes ([ ]).

1.3.2 Tuplas

En esta estructura, los datos se separan por comas y se encuentran entre paréntesis. Las tuplas se caracterizan porque no se pueden modificar, son inmutables. Es decir, no se puede añadir, quitar datos ni modificar los que contiene.

Sin embargo, se pueden extraer partes de las tuplas para crear nuevas tuplas o reemplazarlas con otra tupla. Las tuplas también están indexadas y además puede contener objetos mutables, por lo que puede existir una lista inmersa en una tupla.

Figura 1-7. Ejemplo de una tupla y una lista.



Fuente: Elaboración propia con *Jupyter Notebook.*

1.3.3 Diccionarios

Un "*dict"* (diccionario), es considerado en otros lenguajes como arreglos asociativos. En lugar de estar indexados por números, los diccionarios se indexan por medio de "*keys"* (llaves) que comprenden tipos inmutables como caracteres o números (Python Software Foundation, 2019).

Figura 1-8. Ejemplo de un diccionario.

```
In [2]:  1 altura={"Diego":183,"Diana":176, "Andrés":174}
         2 type(altura)

Out[2]: dict
```

Fuente: Elaboración propia con *Jupyter Notebook.*

1.4 La función *input*

En el aprendizaje de la programación se trabaja con información o entradas provenientes de un usuario. El usuario ingresa un dato o valor, posteriormente un código lo captura y lo utiliza dentro de una función para arrojar un resultado.

La función "*input"* permite que el usuario ingrese una entrada al programa, la cual siempre será guardada como tipo carácter. En caso de querer usarlo como dato de tipo numérico se debe utilizar la función "*int"* aplicada a la entrada para convertirlo en un dato de tipo entero o de usar la función "*float"* se convertiría en flotante. Un dato puede convertirse a tipo carácter usando la función "*str".*

Es conveniente guardar la información que el usuario va a ingresar en una variable para manipularla posteriormente. La función "*input"* abre un cuadro de texto en el que el usuario podrá ingresar datos.

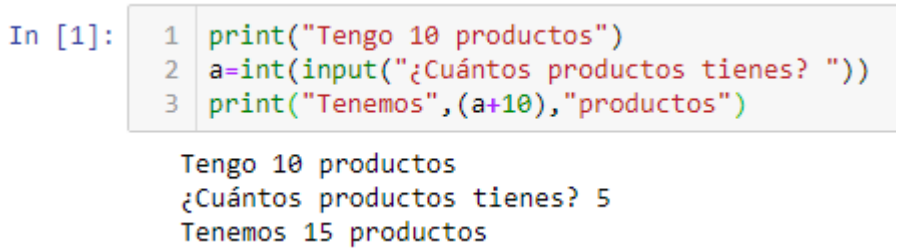
Figura 1-9. Ejemplo de la función "*Input"*.

```
In [1]:  1 a=input("Escribe tu nombre: ")
         2 print("Te llamas",a)

Escribe tu nombre: Daniel
Te llamas Daniel
```

Fuente: Elaboración propia con *Jupyter Notebook*.

Figura 1-10. Otro ejemplo de la función “*Input”*.



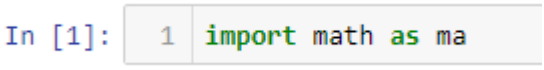
Fuente: Elaboración propia con *Jupyter Notebook*.

1.5 Otras funciones matemáticas

Antes de realizar otras operaciones matemáticas más complejas es necesario entender que *Python* puede cargar módulos, que son partes de *software* con funcionalidades específicas. Los módulos de *Python* suelen contener funciones, clases o variables definidas e implementadas (Learn Python Org., 2019).

Uno de estos módulos es el “*math”,* con el cual se pueden aplicar funciones trigonométricas, hiperbólicas, logarítmicas entre otras. Para hacer uso de este módulo primero debe ser importado utilizando la declaración “*import* “ que permite acceder al código de otros módulos. Se escribe posteriormente otra declaración llamada *as* para instar el módulo con otro nombre de modo que sea más sencillo llamarlo, esto proviene realmente de que Python soporta programación orientada a objetos.

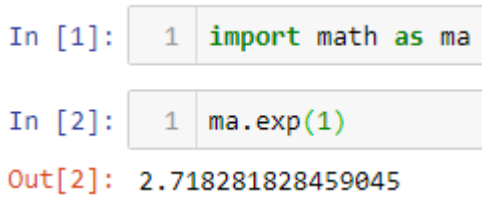
Figura 1-11. Importación del módulo “*math”*.



Fuente: Elaboración propia con *Jupyter Notebook*.

Al hacer esta importación, ya es posible utilizar las funciones que contiene el módulo “*math”* de *Python.* Primero se llama el módulo, seguido se coloca un punto y luego la función que se desee aplicar teniendo en cuenta que entre los paréntesis se coloca la variable o el valor de entrada.

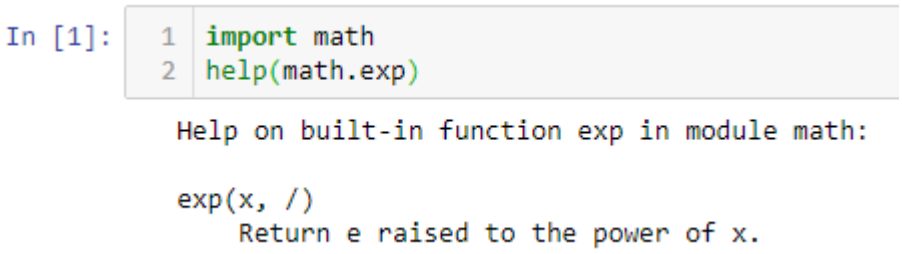
Figura 1-12. Importación de módulo *math* y uso de su función "*exp*".



Fuente: Elaboración propia con *Jupyter Notebook*.

En este caso se aplicó la función "*exp"* a 1 con lo que se obtiene el número de *Euler* elevado a la 1. En caso de necesitar ayuda sobre esta función, *Python* en su librería estándar (aquella que siempre está disponible) cuenta con la función "*help"* que toma un argumento usualmente tipo carácter el cual es buscado como el nombre de un módulo, función, clase, método, palabra clave u otros (Python Software Foundation, 2019).

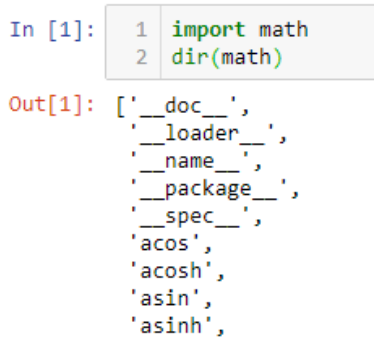
Figura 1-13. Uso de la función "*help"* para obtener información de la función *"exp*" del módulo "*math".*



Fuente: Elaboración propia con *Jupyter Notebook.*

Para conocer todas las demás funciones contenidas en el módulo "*math"* puede utilizarse la función "*dir"* contenida en la librería estándar, en la cual, si se coloca el nombre de un módulo se obtienen los atributos del mismo:

Figura 1-14. Uso de la función "*dir"* para desplegar los atributos de un módulo.



Fuente: Elaboración propia con *Jupyter Notebook.*

1.6 Otros tipos de operadores

Los operadores son símbolos utilizados en informática para que el compilador ejecute operaciones sobre unos operandos para producir un resultado (tutorialspoint, 2019). Cabe anotar que además de los operadores aritméticos existen los operadores relacionales y lógicos.

1.6.1 Operadores relacionales

Los operadores relacionales comparan los valores de dos expresiones y devuelven un resultado booleano. Los operadores relacionales en *Python* se muestran en la siguiente tabla:

Tabla 2. Operadores relacionales en Python

Operador	Significado
==	Es igual a
!=	Es diferente de
>	Es mayor que
<	Es menor que
>=	Es mayor o igual que
<=	Es menor o igual que

Fuente: Elaboración propia a partir de los datos de Mueller John, 2018, p. 119.

1.6.2 Operadores lógicos

Los operadores lógicos ejecutan operaciones en las que usualmente se evalúan dos operandos de tipo lógico y devuelve un resultado de tipo lógico igualmente. Los operadores lógicos en Python se muestran en la siguiente tabla:

Tabla 3. Operadores lógicos en *Python*

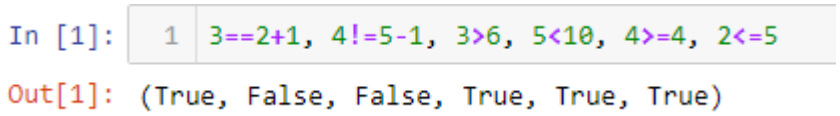
Operador	Significado
And	Y (determina si ambos operandos son verdaderos)
Or	O (determina si alguno de los operandos es verdadero
Not	No es (Niega el valor de un operando)

Fuente: Elaboración propia a partir de los datos de Mueller John, 2018, p. 119.

Los operadores relacionales y lógicos son esenciales ya que son la base del control y gestión de los datos, en especial para comparar datos y modificar información en una variable (Mueller, John, 2018, p. 115).

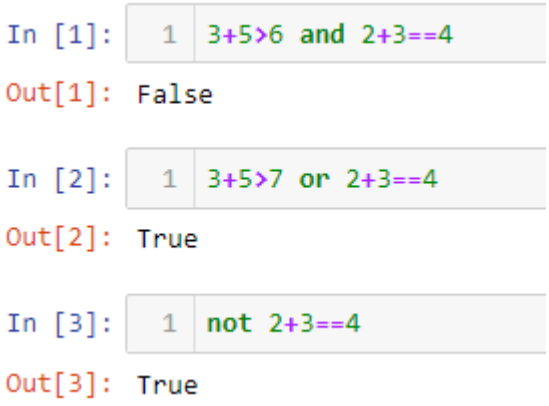
Cabe anotar que el uso de los operadores se da cuando se trabajan estructuras de control para ejecutar tareas, ya que se evalúan condiciones dadas antes de continuar con la ejecución de las mismas.

Figura 1-15. Ejemplo de operadores relacionales.



Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 1-16. Ejemplo de operadores relacionales con operadores lógicos.



Fuente: Elaboración propia con *Jupyter Notebook.*

1.7 Estructuras de control

Según (Fisher, 1970) "por estructuras de control se hace referencia a entornos de programación u operaciones que especifican la secuencia y la interpretación de reglas para programas o partes de programas." A continuación, se trabajarán tres estructuras de control básicas: la declaración "*if*", el ciclo "*while*" y el ciclo "*for*".

1.7.1 Declaración *if- elif-else*

La declaración "*if"* es una herramienta para crear decisiones en los lenguajes de programación. Al usar dicha declaración se puede dar paso a que se cumplan determinadas acciones de acuerdo a una condición que se establece: sí se cumple determinada condición, se realiza determinada acción. Si no, se realiza otra.

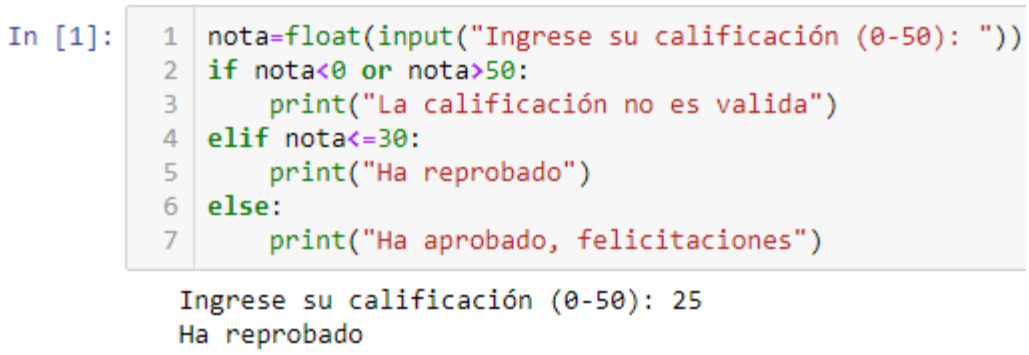
Para usar esta declaración se escribe "*if",* seguido de la condición y después de ésta se colocan dos puntos ":" con el fin de especificar el fin de la condición. Posteriormente se crea un nuevo renglón con la tecla "*enter"* en donde se escribe el código que se ejecutará en caso de que la condición se cumpla.

Es importante distinguir la sangría que aparece después de presionar "*enter"* una vez se haya escrito los dos puntos ":" que marcan el final de la condición.

Cabe resaltar que *Python* no usa llaves para establecer hasta qué punto el código escrito se encuentra dependiente de la condición, por lo que hay que entender bien la profundidad de las sangrías para entender a qué condición pertenecen.

La condición "*elif"* que significa "*else if"*, permite añadir más condiciones para evaluar. Finalmente se añade la cláusula "*else"* al mismo nivel de la cláusula *if* para ejecutar la tarea alternativa en caso de que las previas no se hayan cumplido (Mueller, 2018, p. 136).

Figura 1-17. Ejemplo de declaraciones *"If-elif-else"*.



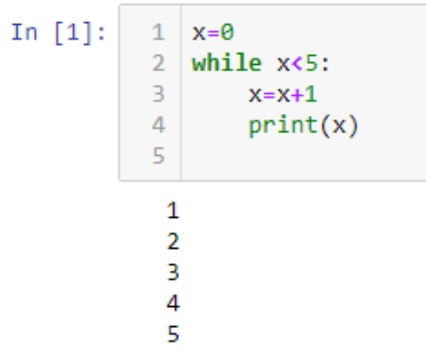
Fuente: Elaboración propia con *Jupyter Notebook.*

En el ejemplo anterior, se evalúa sí la variable nota se encuentra fuera de los rangos de calificación. Sí se encuentra dentro de los rangos válidos para una calificación, posteriormente se evalúa. Si es menor que 30, de ser verdadero como en el ejemplo, se ejecuta la línea siguiente. De no ser verdadero se ejecutará la línea después del "*else"*. Siempre se ejecutan las líneas contenidas en la declaración "*else"* cuando las líneas contenidas en las declaraciones "*if"* no se cumplen (Mueller, 2018, p. 142).

1.7.2 Ciclo While

Según (Fisher, 1970) el ciclo "*while"* es una expresión iterativa de bucle que causa la ejecución repetitiva de una expresión en la medida que una condición determinada se cumpla o no. En *Python* este ciclo se realiza escribiendo "*while*"*,* seguido de la condición y finalmente se especifica el final de la condición colocando dos puntos ":".

Figura 1-18. Uso del ciclo "*while"* para imprimir los números del 1 hasta el 4.



Fuente: Elaboración propia con *Jupyter Notebook.*

Otro ejemplo en el que se utiliza el ciclo "*while"* sería el siguiente:

Figura 1-19. Uso del ciclo *while* para crear un juego de adivinanza.

```
In [1]:
import random
from random import randint
intentos=1
x=randint(1,9)
numero=int(input("Adivina un número entero entre 0 y 10: "))
while numero!=x:
    numero=int(input("Fallaste. Intenta de nuevo: "))
    intentos=intentos+1
print("Has adivinado el numero. Usaste",intentos, "intentos")
```

```
Adivina un número entero entre 0 y 10: 2
Fallaste. Intenta de nuevo: 9
Fallaste. Intenta de nuevo: 8
Fallaste. Intenta de nuevo: 3
Has adivinado el numero. Usaste 4 intentos
```

Fuente: Elaboración propia con *Jupyter Notebook.*

En la Figura anterior (1-19) se importa el módulo "*random"* y de éste se extrae sólo uno de sus atributos usando la declaración "*from"* (Mueller, 2018, p. 205).

El atributo extraído es "*randint"* el cual genera un número entero aleatorio con base en una distribución uniforme. Luego, los números colocados en la función se incluyen (intervalo cerrado), con lo cual se tiene un número aleatorio de entre 0 y 10.

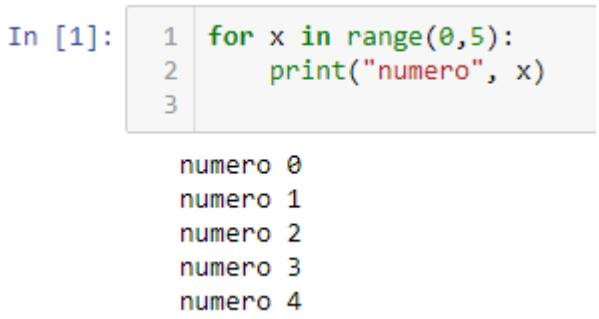
El ciclo "*while"* se encarga entonces de pedir al usuario un número, siempre y cuando éste número no sea igual al número aleatorio generado con la función "*randint"*. Finalmente, con la variable intentos se lleva la cuenta de las veces que el usuario ha ingresado un valor.

1.7.3 Declaración *for*

Es una declaración común en los lenguajes de programación. Se trata de un ciclo que se repite una determinada cantidad de veces por medio de sucesiones que se pueden especificar por medio de la función "*range"*, letras u otras alternativas más convenientes (Mueller, 2018, p. 152).

Para usar esta declaración se escribe “*for”*, seguido de una variable que se irá convirtiendo en los valores que se establezcan, según lo que se especifique como sí se tratara de un contador y al final se utilizan dos puntos”:”.

Figura 1-20. Ejemplo del ciclo “*for”* en “*Python”.*



Fuente: Elaboración propia con *Jupyter Notebook.*

La función “*range (0,5)”* genera una lista de números que va desde 0 hasta 4 (intervalo abierto por derecha). El valor inicial de x es 0.

Luego se ejecuta lo que se encuentre en el ciclo. Teniendo en cuenta ese valor y cuando finalice, se regresa a la línea en la que se encuentra el “*for”.* Posteriormente la variable *x* toma el valor de 1 proveniente de la función “*range”* y vuelve a ejecutarse lo que exista dentro del ciclo “*for”*.

En este ejemplo se está imprimiendo el valor de *x* a medida que cambia de valor al recorrer los números del 0 hasta el 4.

La función “*range”* es útil para crear las iteraciones en los ciclos “*for”*. Debe tenerse en cuenta que no se obtiene una lista como tal con la función “*range”*, sino que es necesario transformar el resultado a una lista, sí se desea usar como un vector de números mediante la función “*list”*.

El ciclo “*for”* también se usa frecuentemente para recorrer valores contenidos en una lista o vector y con ello realizar tareas de búsqueda y/o modificaciones.

Dentro de la declaración "*for*" se suelen incluir declaraciones "*if*" y se usan más declaraciones para controlar la ejecución del ciclo, que son "*Continue*", "*Pass*" y "*Break*":

- *"Break"*: Detiene la ejecución del ciclo
- *"Continue"*: No se ejecuta la iteración actual pero si las iteraciones siguientes
- *"Pass":* Permite la ejecución de la iteración actual y las siguientes.

La siguiente figura muestra un ciclo "*for*" con "*if*" sin las declaraciones mencionadas anteriormente:

Figura 1-21. Ejecución de un ciclo "*for-if*" sin otras declaraciones.

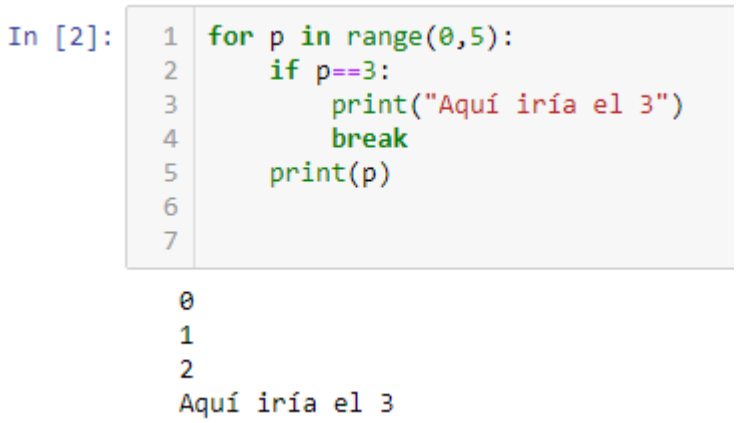
```
In [1]:
for p in range(0,5):
    if p==3:
        print("Aquí iría el 3")
    print(p)

0
1
2
Aquí iría el 3
3
4
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Con la declaración "*break*", colocada dentro de la declaración "*if*" el ciclo se cancela una vez se ejecuta lo especificado cuando la variable *p* es igual a 3. Se ve que *p* no toma el valor de 4 y no se imprime:

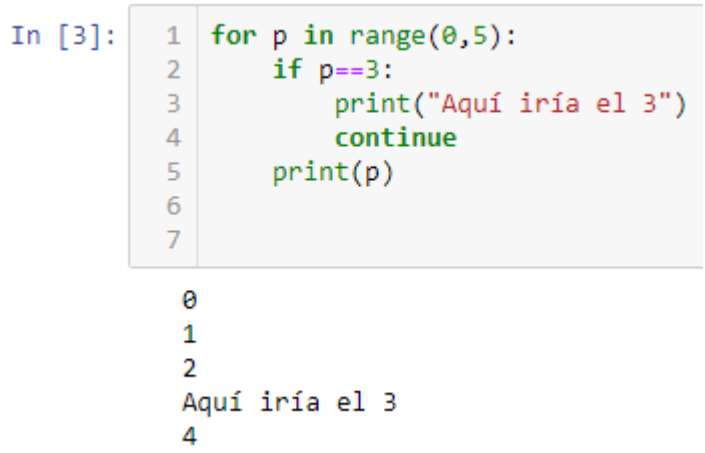
Figura 1-22. Uso de la declaración *break* en un ciclo "*for".*



Fuente: Elaboración propia a partir de Mueller, 2018, p. 154.

Cuando se utiliza la declaración "*continue"*, se impide que se ejecuten las líneas siguientes de código para la iteración actual porque inmediatamente continúa con la siguiente iteración, por ello no se imprime en pantalla el número 3. Es útil para evitar que se ejecuten acciones en determinado instante de una iteración (Mueller, 2018, p. 156).

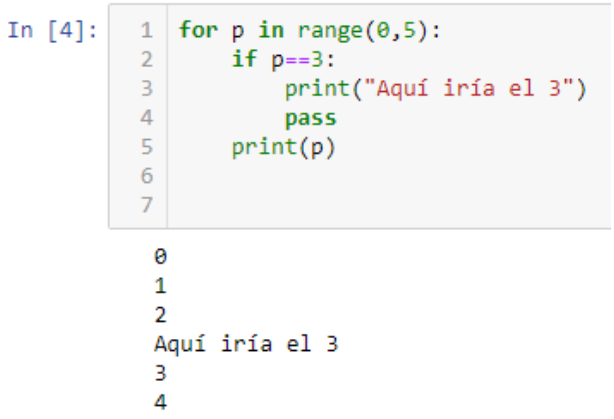
Figura 1-23. Uso de la declaración "*continue"* en un ciclo "*for".*



Fuente: Elaboración propia a partir de Mueller, 2018, p. 156s.

La declaración "*pass"* permite la ejecución de la iteración actual y las siguientes. No difiere del resultado del ciclo sin estas declaraciones, pero a veces suele requerirse para evitar espacios en blanco o código incompleto que puedan causar errores (Piraeus, 2017).

Figura 1-24. Uso de la declaración "*pass"* en el ciclo "*for".*

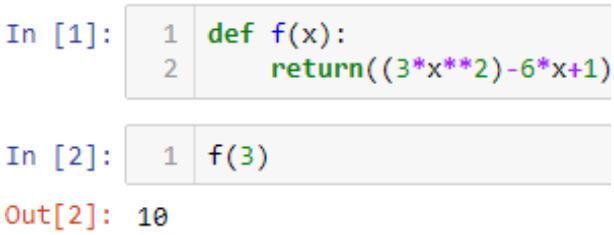


Fuente: Elaboración propia a partir de Mueller, 2018, p. 157.

1.8 Definición de funciones

Las funciones son parte fundamental para crear algoritmos que simplifiquen tareas repetitivas. Una función depende de variables para crear un resultado específico. Las funciones en *Python* se crean escribiendo la declaración "*def"* seguido de un espacio. Luego se escribe el nombre que se le desee dar a la función y junto a ésta se añade entre paréntesis los parámetros o variables de los que dependerá la función. Finalizando la línea se deben colocar dos puntos ":". Posteriormente se pulsa "*enter"* y a partir de esa línea con sangría se digitarán todas las tareas que realizará la función.

Figura 1-25. Una función cuadrática evaluada en x=3.



Fuente: Elaboración propia con *Jupyter Notebook.*

En este ejemplo, en lugar de utilizar la función “*print”*, se utilizó la declaración “*return”*, la cual devuelve el resultado de una función que puede ser empleado en otras funciones o ciclos, si el objetivo fuese solamente mostrar el resultado, podría utilizarse la función “*print”*, en lugar de la declaración “*return”.*

Figura 1-26. Uso de una función para resolver ecuaciones de segundo grado con la formula cuadrática.

```
1  import math
2  def fcuadratica(a,b,c):
3      solucion1=(-b+(math.sqrt((b**2)-(4*a*c))))/(2*a)
4      solucion2=(-b-(math.sqrt((b**2)-(4*a*c))))/(2*a)
5      return(solucion1,solucion2,
6             (f(solucion1),f(solucion2)))
```

```
1  fcuadratica(3,-6,1)
```

```
(1.816496580927726, 0.18350341907227405, (0.0, -4.440892098500626e-16))
```

Fuente: Elaboración propia con *Jupyter Notebook.*

En este ejemplo se crea una función que resuelve ecuaciones de segundo grado por medio de la formula cuadrática y se confirma su validez al evaluar los valores obtenidos mediante la función *f* creada anteriormente.

Es importante reconocer que el trabajo con funciones apoya el tratamiento de datos con estadística y facilita las tareas repetitivas.

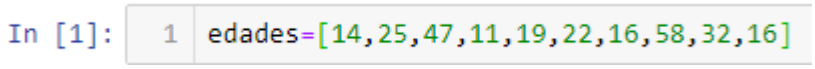
1.9 Otras operaciones en listas

Las listas corresponden a secuencias de datos que van enumeradas desde 0. A través de la librería estándar de *Python* existen varias herramientas con las cuales se pueden trabajar las listas (tutorialspoint, 2019).

Se debe recordar que las listas en Python se crean utilizando corchetes ([ ]) y sus datos se separan con comas.

Suponga que se tiene la siguiente lista:

Figura 1-27. Ejemplo de una lista con edades de personas.

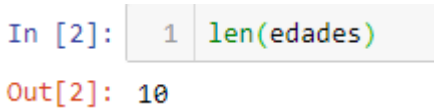


Fuente: Elaboración propia con *Jupyter Notebook.*

Se pueden aplicar las siguientes funciones, métodos para obtener información o modificar la existente como lo indica la fundación de *Python* (Python Software Foundation, 2019):

- *len()*: Devuelve la cantidad de datos de la lista, es decir su longitud.

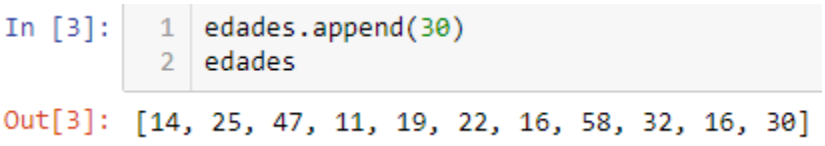
Figura 1-28. Longitud de una lista.



Fuente: Elaboración propia con *Jupyter Notebook.*

- *.append():* Agrega un nuevo dato al final de la lista.

Figura 1-29. Dato añadido al final con función "*append"*.



Fuente: Elaboración propia con *Jupyter Notebook*.

- *.insert(x,y):* añade a la lista el valor y en la posición x. Se debe recordar que la numeración empieza desde 0.

Figura 1-30. El número 40 se inserta en la posición 2 con el método "*insert*".

```
In [4]:  1 edades.insert(2,40)
         2 edades
Out[4]: [14, 25, 40, 47, 11, 19, 22, 16, 58, 32, 16, 30]
```

Fuente: Elaboración propia con *Jupyter Notebook.*

- *.remove():* elimina el dato especificado. De no encontrarse, se emite un error.

Figura 1-31. Número eliminado con "*remove".*

```
In [5]:  1 edades.remove(11)
         2 edades
Out[5]: [14, 25, 40, 47, 19, 22, 16, 58, 32, 16, 30]
```

Fuente: Elaboración propia con *Jupyter Notebook.*

- *index():* devuelve el índice del valor especificado, es decir su posición en la lista.

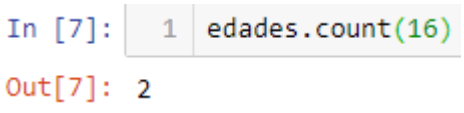
Figura 1-32. Posición del número 16 en la lista.

```
In [6]:  1 edades.index(16)
Out[6]: 6
```

Fuente: Elaboración propia con *Jupyter Notebook.*

- *.count():* cuenta las veces que se encuentra un valor en la lista.

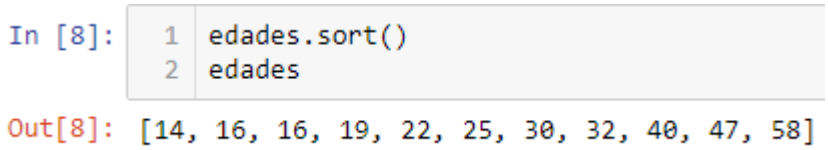
Figura 1-33. Ejemplo de la función *count*.



Fuente: Elaboración propia con *Jupyter Notebook.*

- *.sort():* ordena los datos de forma creciente.

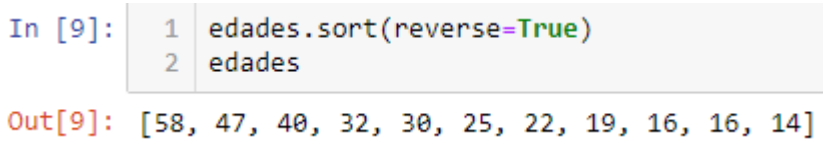
Figura 1-34. Lista ordenada crecientemente con función *sort*.



Fuente: Elaboración propia con *Jupyter Notebook.*

- *.sort (reverse=True):* ordena los datos de forma decreciente.

Figura 1-35. Lista ordenada decrecientemente con función *sort* y argumento reverse.



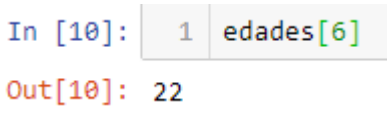
Fuente: Elaboración propia con *Jupyter Notebook.*

- [ ]: devuelve el dato que tiene la numeración especificada. Se puede especificar por intervalos usando dos puntos ":", teniendo en cuenta que el intervalo es cerrado por izquierda y abierto por derecha.

  Si no se especifica alguno de los límites y se deja en blanco, se tomarán el resto de los valores. Es necesario recordar que en "*Python"* las numeraciones comienzan desde 0: es decir que el primer dato está numerado con 0.

  Además, con los corchetes es posible reemplazar valores de la lista si se realiza una asignación (=).

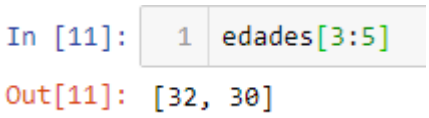
Figura 1-36. Obtención del dato enumerado 6.



Fuente: Elaboración propia con *Jupyter Notebook.*

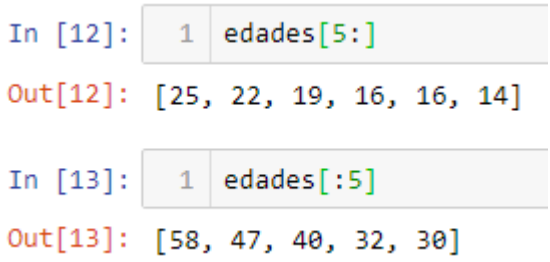
La obtención de un intervalo se puede realizar como se muestra en las siguientes imágenes:

Figura 1-37 .Obtención intervalo de la lista.



Fuente: Elaboración propia con *Jupyter Notebook.*

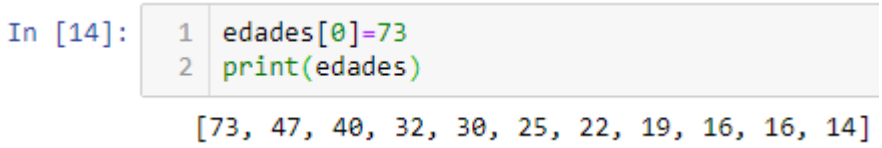
Figura 1-38. Obtención de intervalos desde un punto hasta el final.



Fuente: Elaboración propia con *Jupyter Notebook.*

El reemplazo de un valor de la lista se efectúa con una asignación:

Figura 1-39. Reemplazo de un dato de la lista.

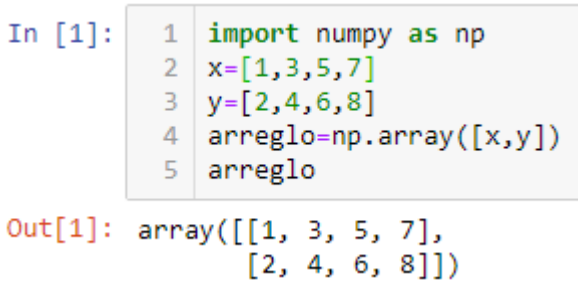


Fuente: Elaboración propia con *Jupyter Notebook.*

1.10 Arreglos de dos dimensiones

Los arreglos son conjuntos de datos organizados que pueden ser una o múltiples dimensiones. Para acceder a estos es se recurre al módulo "*numpy"* o *Python* numérico, con el cual se tendrá una forma de crear los arreglos necesarios a la hora de organizar datos en un "*dataframe"* (tabla de datos). Los arreglos se crean a partir de una lista de números como argumento.

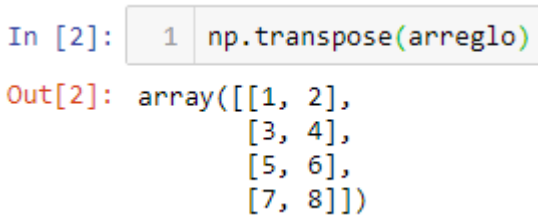
Figura 1-40. Creación de un arreglo “*numpy.ndarray”.*



Fuente: Elaboración propia con *Jupyter Notebook.*

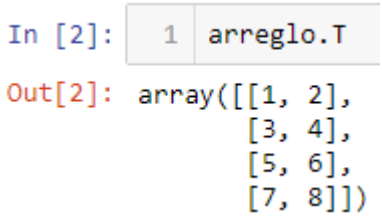
Utilizando la función “*transpose”* se puede transponer el arreglo para obtener las filas como columnas o desde el arreglo con el atributo.”*T”.*

Figura 1-41. Traspuesta de una matriz con función “*transpose”.*



Fuente: Elaboración propia con *Jupyter Notebook.*

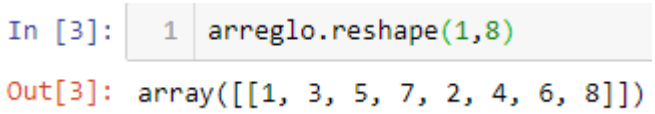
Figura 1-42. Traspuesta de una matriz mediante el atributo .”*T”*.



Fuente: Elaboración propia con *Jupyter Notebook.*

Para cambiar las dimensiones de la matriz sin que cambien sus datos, se puede utilizar el método "*reshape"* contenido en los arreglos "*numpy.ndarray"* colocando como argumento las dimensiones a las que se quiere cambiar el arreglo. En la siguiente figura el arreglo resultante tiene dimensiones *1x8:*

Figura 1-43. Arreglo transformado a dimensiones 1x8.



Fuente: Elaboración propia con *Jupyter Notebook.*

Con los arreglos se pueden realizar operaciones del algebra lineal como las de la siguiente tabla y se muestran con imágenes ejemplos de estas:

Tabla 4. Algunas operaciones matriciales en Python

Sumas o restas (las matrices deben ser del mismo tamaño)	+ ó -
Multiplicación por elemento	*
Multiplicación matricial	@ ó *A*.dot(*B*)
Multiplicación por escalar α	*α*A*
Matriz idéntica	numpy.identity(n)
Matriz inversa	numpy.linalg.inv(*A*)
Determinante de una matriz	numpy.linalg.det(*A*)
Solución a un sistema de ecuaciones lineales (solo matrices cuadradas)	numpy.linalg.solve(*A*,*b*)
Potenciación de matriz cuadrada	numpy.linalg.matrix_power(*A*,*n*)

Fuente: Elaboración propia con la información de The SciPy community, 2019.

Suponga la creacion de los siguientes arreglos. Se aplicarán las operaciones mencionadas en la tabla anterior como ejemplo:

Figura 1-44. Un ejemplo de creación de arreglos.

```
In [1]: import numpy as np
        A=np.array([[2,3,5],[5,8,7],[3,2,1]])
        B=np.array([[4,3,2],[6,2,7],[1,1,1]])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 1-45.Ejemplo de suma de matrices.

```
In [2]: A+B

Out[2]: array([[ 6,  6,  7],
               [11, 10, 14],
               [ 4,  3,  2]])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 1-46. Ejemplo de una resta de matrices.

```
In [3]: B-A

Out[3]: array([[ 2,  0, -3],
               [ 1, -6,  0],
               [-2, -1,  0]])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 1-47. Ejemplo de una multiplicación por elementos.

```
In [4]: A*B

Out[4]: array([[ 8,  9, 10],
               [30, 16, 49],
               [ 3,  2,  1]])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 1-48. Ejemplos de multiplicación matricial.

```
In [5]: 1 A@B

Out[5]: array([[31, 17, 30],
               [75, 38, 73],
               [25, 14, 21]])

In [6]: 1 A.dot(B)

Out[6]: array([[31, 17, 30],
               [75, 38, 73],
               [25, 14, 21]])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 1-49. Ejemplo de multiplicación por escalar.

```
In [7]: 1 (1/2)*A

Out[7]: array([[1. , 1.5, 2.5],
               [2.5, 4. , 3.5],
               [1.5, 1. , 0.5]])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 1-50. Creación de una matriz idéntica de tamaño 3x3.

```
In [8]:  1 np.identity(3)

Out[8]: array([[1., 0., 0.],
               [0., 1., 0.],
               [0., 0., 1.]])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 1-51. Inversa de una matriz.

```
In [9]:  1 np.linalg.inv(A)

Out[9]: array([[ 0.17647059, -0.20588235,  0.55882353],
               [-0.47058824,  0.38235294, -0.32352941],
               [ 0.41176471, -0.14705882, -0.02941176]])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 1-52. Determinante de una matriz.

```
In [10]:  1 np.linalg.det(B)

Out[10]: -9.000000000000002
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 1-53. Solución a un sistema de ecuaciones lineales.

```
In [11]:  1 C=np.array([[3,-2,5],[-2,6,1],[1,4,-3]])
          2 b=np.array([64,-13,-30])
          3 np.linalg.solve(C,b)
Out[11]: array([ 5., -2.,  9.])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 1-54. Potenciación de una matriz.

```
In [12]:  1 np.linalg.matrix_power(A,3)
Out[12]: array([[ 376,  494,  486],
                [ 871, 1133, 1094],
                [ 263,  333,  314]])
```

Fuente: Elaboración propia con *Jupyter Notebook*.

1.11 Ejercicios de la unidad 1

1. Suponga la siguiente variable que almacena datos que representan la existencia de características en cierto hotel:

C= ["piscina", "gimnasio", "golf", "sauna", "turco", "jacuzzi", "teatro", "spa"]

Crear un código que permita al usuario digitar una característica y determinar si se encuentra disponible o no en dicho hotel.

2. Suponga que cada valor de la siguiente lista contiene la ganancia obtenida en dólares con un producto específico de una microempresa de confecciones para determinado mes:

I= [698, 359, 145, 78, 975, 215, 742, 366, 398, 475, 155, 799, 214, 569, 47, 874, 102, 653, 214, 775, 945, 485, 726, 583, 321, 547, 629, 887, 249, 399, 641, 222, 155, 729]

Ordenar la lista y extraer las tres ganancias más altas y las tres ganancias más bajas del mes.

3. Crear un código que imprima los 20 primeros valores de la serie o sucesión de Fibonacci.
4. En cadenas de "*Markov"*, la matriz de transición de n pasos se calcula potenciando la matriz de transición n veces. Si se calcula el límite cuando n tiende a infinito de la matriz de transición a la n se obtienen las probabilidades de estado estable denotadas con π.

   ¿Cuáles son las matrices de transición de 2, 10, 100 pasos para la siguiente matriz P?

$$P = \begin{bmatrix} 7/9 & 2/9 \\ 3/7 & 4/7 \end{bmatrix}$$

5. Los primeros 6 términos de una determinada sucesión son:

$$\{2,3,5,9,17,33...\}$$

Encontrar la sucesión y crear un código que imprima los tres primeros términos de la misma y posteriormente pedir al usuario que ingrese el siguiente término de la sucesión. De encontrarlo, finalizar imprimiendo un mensaje de éxito, de lo contrario, que se imprima el siguiente término de la sucesión hasta un máximo de 12 términos, después de esto se finaliza el código con un mensaje de fracaso.

6. Crear una función que genere los 10 primeros números aleatorios con base en el método congruencial con los parámetros:

a. m=100, a=71 c=43 x_0=29
b. m=100, a=47, c=25, x_0=19
c. m=100, a=13, c=16, x_0=151

Donde x_0 es la semilla, a es el multiplicador, c es el incremento y m es el módulo:

$$x_{n+1} = (ax_n + c)(modulo\ m) \qquad i = (0, 1, 2, \dots) \tag{1}$$

Y el número aleatorio es:

$$R_i = \frac{x_i}{m} \tag{2}$$

Es decir que el primer valor x_1 para el primer ejercicio es igual al residuo de (71) (29)+43 dividido entre 100 que equivale a 2. Luego, el primer número aleatorio es la división de este último número entre 100, es decir 0,02, el segundo sería 0,85 y así sucesivamente.

2. Capítulo 2. Tablas de datos

Las tablas de datos son la forma más común de organizar los datos en forma de matrices para posteriormente tratarlos. Cuando los conjuntos de datos se convierten en una tabla de datos, a partir del uso del uno de los módulos de *Python* conocido como *pandas*, se dispone de un objeto - con valiosos métodos - que facilitan el tratamiento y aplicación de funciones a los datos para analizarlos, para obtener información relevante de gran ayuda para la toma de decisiones.

En este capítulo se mostrará como:

- Importar el módulo *pandas* para trabajar tablas de datos.
- Crear tablas de datos.
- Importar tablas de datos.
- Visualizar tablas de datos.
- Modificar la forma y los elementos de las tablas de datos.
- Filtrar y ordenar tablas de datos.

2.1 Obtención de tablas de datos

Dos de las formas de disponer de tablas de datos son: crear una tabla de datos o importarla desde un *"dataset"* (conjunto de datos) proveniente de un archivo externo. El módulo *pandas* está equipado con varias funciones para importar diferentes tipos de archivos. En esta guía se importarán archivos CSV (*Comma-separated values*), JSON y EXCEL.

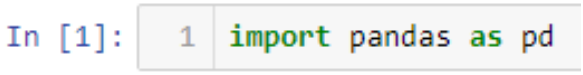
2.1.1 Creación de tablas de datos

Una vez se tenga un arreglo o una variable de tipo diccionario, estos pueden ser convertidos en tabla de datos que son la forma estándar de manejar, de organizar datos en *Python*.

Para ello se necesita la librería *pandas* que contiene este tipo de objeto y además facilita la importación de datos. Todos los objetos clases y métodos del módulo pandas pueden encontrarse en el siguiente link:

https://pandas.pydata.org/pandas-docs/stable/reference/index.html.

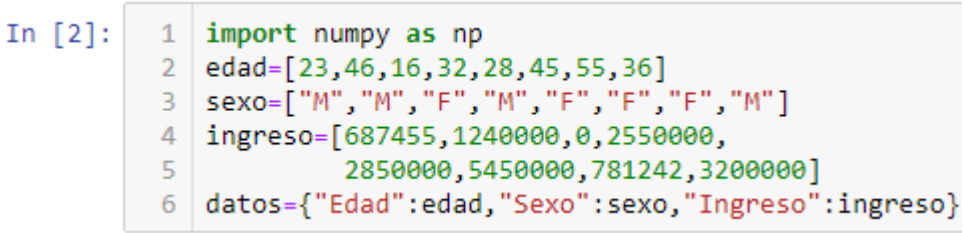
Figura 2-1. Importación del módulo *"pandas".*



Fuente: Elaboración propia con *Jupyter Notebook.*

Suponga que se va a crear una tabla de datos utilizando una variable tipo de tipo diccionario llamada "datos", en la cual las llaves corresponderán a los nombres de las columnas o variables y que comprenden cada una de las listas que se crean a continuación:

Figura 2-2. Creación de una variable *"dict"* a partir de listas.



Fuente: Elaboración propia con *Jupyter Notebook.*

Se utilizará la librería *pandas* para crear una tabla de datos mediante la clase “*DataFrame”* a la que se le llamará “tabla” a partir del diccionario creado anteriormente:

Figura 2-3. Creación de una tabla de datos *pandas* a partir del diccionario "datos".

```
In [3]:  1 tabla=pd.DataFrame(datos)
         2 tabla
```

Out[3]:

	Edad	Sexo	Ingreso
0	23	M	687455
1	46	M	1240000
2	16	F	0
3	32	M	2550000
4	28	F	2850000
5	45	F	5450000
6	55	F	781242
7	36	M	3200000

Fuente: Elaboración propia con *Jupyter Notebook.*

Es fundamental utilizar la función “*dir”* para conocer funciones, atributos y métodos que sirven para modificar la tabla de datos. Por ejemplo: se pueden cambiar los nombres de las columnas desde el atributo “*columns”* de la tabla de datos. Debido a que las columnas son inmutables al ser de tipo *“index”,* se tiene que crear una nueva lista con los nombres de las columnas y asignar el atributo *“columns”* de la tabla de datos la lista creada.

Figura 2-4. Reemplazo de los nombres de las columnas de la tabla de datos.

```
In [4]:  1 tabla.columns=["Edad","Sexo","Sueldo"]
         2 tabla
```

Out[4]:

	Edad	Sexo	Sueldo
0	23	M	687455
1	46	M	1240000
2	16	F	0
3	32	M	2550000
4	28	F	2850000
5	45	F	5450000
6	55	F	781242
7	36	M	3200000

Fuente: Elaboración propia con *Jupyter Notebook.*

Con este resultado ya se tiene la principal estructura de datos de la librería *pandas*, la cual se puede manipular para obtener información a partir de los datos que contiene.

Los datos generalmente provienen de sistemas que capturan datos y los guardan en formatos específicos. La importación de datos permite trabajar con datos generados por otros agentes reduciendo sustancialmente el tiempo empleado para disponer de una tabla de datos. Existen muchos tipos de archivos externos que contienen datos, los más relevantes podrían ser aquellos provenientes de bases de datos.

2.1.2 Importación de datos desde direcciones de internet

Python dispone de varias funciones en la librería *pandas* que permiten leer diferentes tipos de archivos. Las funciones que llevan a cabo estas tareas se caracterizan porque se escriben como *“pandas.read”* seguido de un guion bajo y luego el tipo de archivo a leer.

El argumento entre paréntesis puede ser la dirección de internet en la cual se encuentra el archivo o la ruta de un archivo en el directorio de trabajo actual con el nombre del archivo, cualquiera que sea entre comillas.

Todas las funciones e información relacionada con la importación de datos con la librería *pandas* se encuentran en:

https://pandas.pydata.org/pandas-docs/41ínea41n/0.23.4/api.html#input-output.

A continuación, se trabajarán tres tipos de archivos que pueden ser importados mediante el módulo *pandas*:

- CSV: Es un archivo conformado por solo texto, sin ningún formato. Corresponde a "*Comma Separated Values"* (Beau, 2019), en donde los elementos de las filas se separan mediante comas.
- XLS o XLSX: son los archivos generados por Microsoft Excel.
- JSON (*JavaScript Object Notation*): "es un formato ligero para el intercambio de datos. Utiliza convenciones que son ampliamente utilizadas por programadores de la familia de lenguajes C, C++, C#, Java, JavaScript, Perl, Python y muchos otros" (JSON.org, 2017).

En primer caso se importarán conjuntos de datos de "R", otro lenguaje de programación para el análisis estadístico y creación de gráficos. Uno de dichos conjuntos de datos se llama *"traffic*" en el cual se realizó un experimento en Suecia para evaluar los efectos de los límites de velocidad en los accidentes automovilísticos en 92 días de 1961 y 1962 en donde para bloques de días, se aplicaba o no un límite de velocidad y se contaban la cantidad de accidentes automovilísticos por día (Ripley, 2019).

Para ello se debe tener en cuenta el tipo de archivo y su URL. Se importará un archivo CSV desde la siguiente URL:

https://vincentarelbundock.github.io/Rdatasets/csv/MASS/Traffic.csv.

La concatenación con símbolos se hace para fines ilustrativos.

Figura 2-5. Importación del módulo *pandas*.

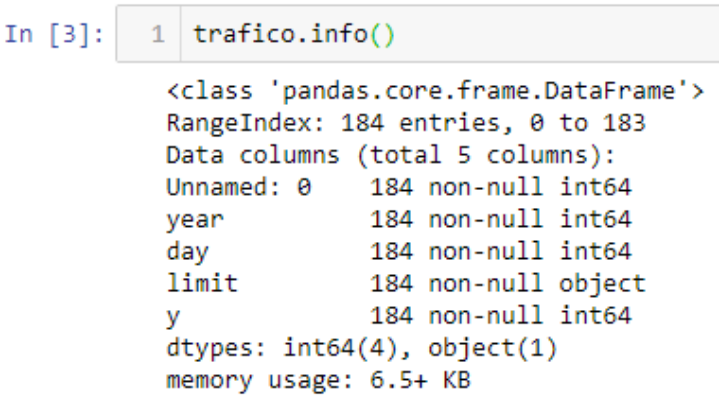
Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 2-6. Lectura de un archivo CSV encontrado en internet.

Fuente: Elaboración propia con *Jupyter Notebook.*

Ahora se dispone de una tabla de datos llamada “tráfico”. Primero es necesario conocer las características principales de esta tabla de datos. Para ello es útil utilizar el método “*info”.*

Figura 2-7. Características de una tabla de datos obtenidas con el método “*info”*.



Fuente: Elaboración propia con *Jupyter Notebook.*

De esta salida, se encuentran 184 entradas con 5 columnas que son: *“Unnamed”, “year”, “day”, “limit” y “y”,* de las cuales 4 columnas son de tipo entero y una de tipo objeto.

La columna “*limit”* especifica sí para el día correspondiente, se aplicó o no el límite de velocidad y la columna “y” es la cantidad de accidentes en el día correspondiente.

Al tener una tabla de datos de mayor tamaño, *Python* simplifica su visualización mostrando su parte inicial y final. En caso de querer visualizar todos los datos de la tabla de datos se debe utilizar la función general “*set_option display”* para filas especificando la cantidad de datos que se desean visualizar:

Figura 2-8. Modificación de la cantidad de filas mostradas de una tabla de datos.

```
In [4]:  1  pd.set_option('display.max_rows', 500)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

En este caso se ha especificado mostrar un máximo de 500 filas en las tablas de datos. Otras opciones están disponibles para columnas, uso de memoria entre otras (The pandas proyect, 2019).

Si por el contrario se desea ver unos cuantos datos para tener una idea general de la tabla de datos, se pueden usar los métodos *“head o tail”* de una tabla de datos para mostrar las cinco primeras o últimas filas respectivamente o se puede especificar también una cantidad a mostrar.

Figura 2-9. Despliegue de primeras filas de una tabla de datos.

```
In [5]: 1 trafico.head()
```

Out[5]:

	Unnamed: 0	year	day	limit	y
0	1	1961	1	no	9
1	2	1961	2	no	11
2	3	1961	3	no	9
3	4	1961	4	no	20
4	5	1961	5	no	31

Fuente: Elaboración propia con *Jupyter Notebook*.

Figura 2-10. Despliegue de últimas filas de una tabla de datos.

```
In [6]: 1 trafico.tail(3)
```

Out[6]:

	Unnamed: 0	year	day	limit	y
181	182	1962	90	yes	14
182	183	1962	91	yes	15
183	184	1962	92	yes	9

Fuente: Elaboración propia con *Jupyter Notebook*.

Como bien se había mencionado, existen otros tipos de archivos que se pueden importar a *Python*. La información de los tipos de archivos que se pueden importar y su explicación se encuentra en el "*input/output"* de la documentación del módulo "*pandas"*.

Por ejemplo, se importará el conjunto de datos "*iris*" que contiene 50 mediciones del largo del sépalo, ancho de sépalo, largo del pétalo y ancho de pétalo de cada una de las tres especies de la planta iris.

La importación de estos datos desde archivos tipo JSON y Excel es muy similar.

A continuación, se muestran los links en los que se encuentra este conjunto de datos en formato EXCEL y JSON:

- https://web.stanford.edu/~ashishg/msande111/excel/iris.xls, en formato XLS de EXCEL.

- https://pkgstore.datahub.io/machine-learning/iris/iris_json/data/23a7b3de91da915b506f7ca23f6d1141/iris_json.json, en formato JSON.

La lectura de estos conjuntos de datos seria la que se muestra en las siguientes imágenes:

Figura 2-11. Importación de un conjunto de datos de formato *XLS.*

```
In [7]:  1 irisExcel=pd.read_excel("https://web.stanford.edu"+
         2                         "/~ashishg/msande111/excel/iris.xls")
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 2-12. Importación de un conjunto de datos de formato *JSON.*

```
In [8]:  1 irisJSON=pd.read_json("https://pkgstore.datahub.io/"+
         2 "machine-learning/iris/iris_"
         3 +"json/data/23a7b3de91da915b506f7ca23f6d1141/iris_json.json")
```

Fuente: Elaboración propia con *Jupyter Notebook.*

2.1.3 Importación de un archivo guardado

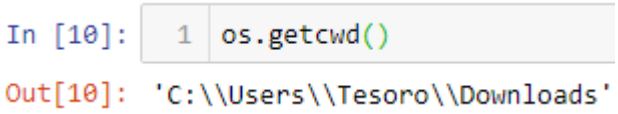
Otro caso es la importación de datos desde un archivo descargado disponible en el disco duro. Para hacer esto es necesario conocer el directorio en el cual se está trabajando y colocar el archivo allí o cambiar el directorio de trabajo a otro destino y colocar el archivo en ese nuevo destino. Esto se lleva a cabo con el módulo "*os"*.

Figura 2-13. Importación del módulo "*os".*

Fuente: Elaboración propia con *Jupyter Notebook.*

Para conocer el directorio de trabajo actual, se utiliza la función "*getcwd"*:

Figura 2-14. Obtención del directorio de trabajo actual.



Fuente: Elaboración propia a partir de la fuente Ohri Ajay, 2018, pp. 36-38.

Aquí se ha obtenido el directorio de trabajo actual. Para importar un archivo, se debe colocar en la ruta obtenida teniendo en cuenta el tipo de archivo y entre comillas se coloca el nombre del archivo con su extensión.

Suponga que se desea cambiar el directorio a la carpeta escritorio, entonces se deberá hacer lo siguiente:

Figura 2-15. Cambio del directorio de trabajo actual.

```
In [11]:  1  os.chdir('C:\\Users\\Tesoro\\Desktop')

In [12]:  1  os.getcwd()

Out[12]: 'C:\\Users\\Tesoro\\Desktop'
```

Fuente: Elaboración propia a partir de la fuente Ohri Ajay, 2018, pp. 36-38.

Una vez asegurado el nuevo directorio de trabajo, se importará un archivo cualquiera colocándolo en la ruta definida:

Figura 2-16. Importación de un archivo del disco duro.

```
In [13]:  1  censo=pd.read_excel("C:\\Users\\Tesoro\\Desktop\\
          2                      Base de datos.xlsx")
```

Fuente: Elaboración propia con *Jupyter Notebook.*

2.2 Modificación de tablas de datos

Durante el tratamiento de datos puede surgir el requerimiento de modificar las tablas de datos para añadir nuevas variables que se necesiten para determinada entrada de función, o corregir información que haya sido capturada incorrectamente proveniente de un conjunto de datos importado.

El conocimiento de estas operaciones puede ayudar a solventar muchos problemas que surgen cuando se necesita analizar tablas de datos.

2.2.1 Añadir columnas o filas

Muchas veces puede resultar necesario añadir columnas a una tabla de datos para añadir variables que se necesiten. Para añadir columnas se recurre al método "*assign"* de las tablas de datos, en el cual se coloca como argumento el nombre de la variable o de la columna y se asigna un vector que contiene los datos cuya cantidad debe ser igual a la de las otras variables o columnas de la tabla de datos. Suponga que se desea añadir a la tabla obtenida de la Figura 2-4, una nueva variable llamada "Estado Civil", la cual puede tomar los valores: soltero "S", casado "C", divorciado "D" o viudo "V":

Figura 2-17. Creación de una lista para añadir a la tabla de datos.

```
In [5]:  1  Estado=["S","C","S","C","C","D","V","S"]
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Luego se procede a añadir esta variable a la tabla de datos:

Figura 2-18 Columna añadida a una tabla de datos con el método “*assign”.*

```
In [6]: 1 tabla=tabla.assign(Estado_Civil=Estado)
        2 tabla
```

Out[6]:

	Edad	Sexo	Sueldo	Estado_Civil
0	23	M	687455	S
1	46	M	1240000	C
2	16	F	0	S
3	32	M	2550000	C
4	28	F	2850000	C
5	45	F	5450000	D
6	55	F	781242	V
7	36	M	3200000	S

Fuente: Elaboración propia con *Jupyter Notebook.*

Otra posible forma de añadir una variable consiste en referenciar la columna de una tabla de datos teniendo en cuenta que la referencia sería el nombre de la nueva columna entre comillas y luego se hace una asignación:

Figura 2-19. Otro método para añadir una columna a una tabla de datos:

```
In [7]: 1 tabla["Estado_Civil"]=Estado
        2 tabla
```

Out[7]:

	Edad	Sexo	Sueldo	Estado_Civil
0	23	M	687455	S
1	46	M	1240000	C
2	16	F	0	S
3	32	M	2550000	C
4	28	F	2850000	C
5	45	F	5450000	D
6	55	F	781242	V
7	36	M	3200000	S

Fuente: Elaboración propia con *Jupyter Notebook.*

La otra posible necesidad es añadir una nueva fila. Se utiliza para ello el método "*append"*, el cual permite concatenar tabla de datos por filas. Esto quiere decir que los datos que se quieran añadir a la tabla de datos deben estar contenidos en otra tabla de datos que contenga los mismos nombres de las columnas de la tabla de datos principal. Es necesario asignar el nombre de la tabla de datos a este resultado para conservar los cambios.

A continuación, se muestra un ejemplo para añadir una fila a partir de una nueva tabla de datos creada desde un diccionario:

Figura 2-20. Creación de una tabla de datos para ser concatenada.

```
In [8]:
fila_nueva={"Edad":[18],
            "Sexo":["M"],
            "Sueldo":[1200000],
            "Estado_Civil":["S"]}
x=pd.DataFrame(fila_nueva)
x
```

Out[8]:

	Edad	Sexo	Sueldo	Estado_Civil
0	18	M	1200000	S

Fuente: Elaboración propia con *Jupyter Notebook.*

Luego se procede a usar el método "*append"* para concatenar las tablas de datos. El argumento "*ignore_index"* se usa para evitar que este vector fila entre con su numeración, la cual seria 0 ya que es el primero y único en su tabla de datos.

Figura 2-21. Concatenación de tablas de datos como método para agregar filas.

```
In [9]:  1 tabla=tabla.append([x],ignore_index=True)
         2 tabla
```

Out[9]:

	Edad	Sexo	Sueldo	Estado_Civil
0	23	M	687455	S
1	46	M	1240000	C
2	16	F	0	S
3	32	M	2550000	C
4	28	F	2850000	C
5	45	F	5450000	D
6	55	F	781242	V
7	36	M	3200000	S
8	18	M	1200000	S

Fuente: Elaboración propia con *Jupyter Notebook.*

Igualmente es posible añadir filas refiriéndose a estas como si ya existieran en la tabla de datos. Por ejemplo: usando la localización basada en etiquetas *"loc"* que se explica más adelante, se puede añadir la misma fila como se muestra a continuación:

Figura 2-22. Otro método para añadir filas a una tabla de datos.

```
In [10]:  1 tabla.loc["8",["Edad",
          2                  "Sexo",
          3                  "Sueldo",
          4                  "Estado_Civil"]]=[18,"M",1200000,"S"]
```

Fuente: Elaboración propia con *Jupyter Notebook.*

2.2.2 Eliminar filas o columnas

Suele ser necesario también, eliminar columnas y filas de las tablas de datos. En estos casos se recurre al método *"drop"*.

Este método tiene un argumento con el cual se escoge entre eliminar filas indexadas o columnas: 0 para eliminar por filas o 1 para eliminar columnas (The pandas proyect, 2019).

Ahora, si se desea eliminar una fila, primero se coloca el número con el cual esta indexada la fila y como segundo argumento se coloca "0", esto último con el fin de hacer referencia a las filas o se puede dejar vacío ya que es el argumento por defecto. Para eliminar una columna en cambio, como primer argumento se debe colocar entre comillas el nombre de la columna y como segundo argumento "1".

En el siguiente ejemplo se aplicarán los métodos descritos, pero serán guardados en otra variable llamada "tabla_prueba" para no alterar la tabla de datos con la que se ha venido trabajando:

Figura 2-23. Tabla de datos obtenida al eliminar una fila de la tabla original.

```
In [11]:  1 tabla_prueba=tabla.drop(2,0)
          2 tabla_prueba
```

Out[11]:

	Edad	Sexo	Sueldo	Estado_Civil
0	23	M	687455	S
1	46	M	1240000	C
3	32	M	2550000	C
4	28	F	2850000	C
5	45	F	5450000	D
6	55	F	781242	V
7	36	M	3200000	S
8	18	M	1200000	S

Fuente: Elaboración propia con *Jupyter Notebook.*

A partir de esta operación se encuentra que en la enumeración de la nueva tabla ha desaparecido la numero “2” y esto puede dar lugar a inconvenientes en el tratamiento futuro de los datos, ya que se esperaría una numeración correcta.

Una forma de arreglarlo sería creando nuevamente esta indexación utilizando las funciones “*range* y *len”* junto en el atributo “*index”* de la tabla de datos:

Figura 2-24. Una forma de corregir la indexación causada por la eliminación de una fila.

```
In [12]:  1 tabla_prueba.index=range(0,len(tabla_prueba),1)
          2 tabla_prueba
```

Out[12]:

	Edad	Sexo	Sueldo	Estado_Civil
0	23	M	687455	S
1	46	M	1240000	C
2	32	M	2550000	C
3	28	F	2850000	C
4	45	F	5450000	D
5	55	F	781242	V
6	36	M	3200000	S
7	18	M	1200000	S

Fuente: Elaboración propia con *Jupyter Notebook*.

Para eliminar una columna, se recurre al método “*drop”* mencionado anteriormente, utilizando como segundo argumento el número “1” para referirse a las columnas.

Luego, para trabajar con una tabla de datos modificada, esta debe ser guardar en una nueva variable o se puede guardar con el mismo nombre para sobrescribirlo. En la siguiente figura no se guarda la tabla de datos modificada:

Figura 2-25. Obtención de una tabla de datos al eliminar una de las columnas.

```
In [13]: 1 tabla.drop("Sexo",1)
```

Out[13]:

	Edad	Sueldo	Estado_Civil
0	23	687455	S
1	46	1240000	C
2	16	0	S
3	32	2550000	C
4	28	2850000	C
5	45	5450000	D
6	55	781242	V
7	36	3200000	S
8	18	1200000	S

Fuente: Elaboración propia con *Jupyter Notebook*.

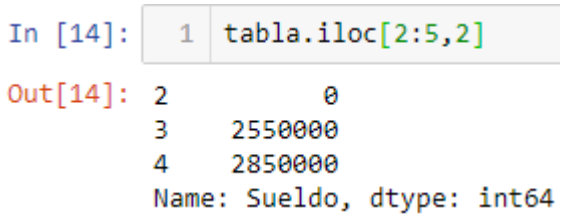
2.2.3 Modificación de valores o datos con *"iloc" o "loc"*

Cuando se trabajan tablas de datos, puede surgir la necesidad de hacer modificaciones, ya que la información proveniente que se importa puede tener algún error o fue digitada incorrectamente. Para resolver dicho inconveniente, se necesita conocer la ubicación de los datos en la tabla. Para acceder a los datos se utiliza localizaciones basadas en enteros o en etiquetas. Se comenzará con el atributo "*iloc"* de las tablas de datos que hace referencia a la localización basada en enteros.

- *.iloc* o localización basada en posiciones enteras: (The pandas proyect, 2019). La localización basada en posiciones enteras usa como números como entradas. Para una tabla de datos en dos dimensiones, se tienen dos entradas, la primera para las filas y la segunda para las columnas.

Similar a las listas, se deben colocar los números o los intervalos que se desean obtener como una subtabla de la tabla de datos original. Se debe recordar que la numeración empieza desde cero y los intervalos son cerrados a la izquierda y abiertos a la derecha.

Figura 2-26. Parte de una tabla de datos obtenida mediante localización entera.



Fuente: Elaboración propia con *Jupyter Notebook.*

En la figura anterior la primera entrada correspondiente a las filas comprende las 2,3 y 4. La fila 5 no se cuenta porque el intervalo es abierto por derecha. En las columnas el número 2 especifica la tercera columna, ya que el 2 desde una numeración que empieza desde 0 sería la tercera posición. Se observa que el tipo de objeto es entero, luego pueden usarse para operaciones matemáticas.

- *.loc*: es la localización basada en "*labels"* (etiquetas), es decir por el nombre que tienen las filas o las columnas. En este caso las filas están enumeradas, por lo que se deben colocar estos números para obtener filas y seguido el nombre de las columnas, que en este caso las columnas ya tienen un nombre dado por las llaves.

Figura 2-27. Parte de una tabla de datos obtenida con localización por etiquetas.

```
In [15]: 1 tabla.loc[[2,4,6],["Edad","Sexo"]]
```

Out[15]:

	Edad	Sexo
2	16	F
4	28	F
6	55	F

Fuente: Elaboración propia con *Jupyter Notebook.*

Si se desea encontrar columnas por su nombre se debe colocar entre corchetes la lista con los nombres de las columnas que se desean visualizar:

Figura 2-28. Obtención de determinadas columnas de la tabla de datos con todas sus filas.

```
In [16]: 1 tabla[["Sueldo","Edad"]]
```

Out[16]:

	Sueldo	Edad
0	687455	23
1	1240000	46
2	0	16
3	2550000	32
4	2850000	28
5	5450000	45
6	781242	55
7	3200000	36
8	1200000	18

Fuente: Elaboración propia con *Jupyter Notebook.*

Suponga que se quieren los datos de edad de las filas 2,3 y 4. En resumen, con las dos alternativas trabajadas se pueden obtener como sigue:

Figura 2-29. Ejemplo de localización basada en etiquetas

```
In [17]:  1  tabla["Edad"][2:5]

Out[17]: 2    16
         3    32
         4    28
         Name: Edad, dtype: int64
```

Fuente: Elaboración propia con *Jupyter Notebook*

Figura 2-30. Ejemplo de localización basada en enteros.

```
In [18]:  1  tabla.iloc[2:5,0]

Out[18]: 2    16
         3    32
         4    28
         Name: Edad, dtype: int64
```

Fuente: Elaboración propia con *Jupyter Notebook*.

En la Figura 2-29, a partir de la tabla *de datos* se accede a la variable “Edad” y de esta al intervalo de filas que se desea encontrar. En la Figura 2-30, se usa la localización basada en enteros teniendo en cuenta que la columna “Edad” es la primera columna en la tabla *de datos* original. Una vez se comprendan estas dos formas de localización, se pueden modificar datos de una tabla realizando una asignación desde una localización cualquiera.

Por ejemplo:

Figura 2-31. Cambio de los datos de una tabla usando localización entera.

```
In [19]:
1 tabla_prueba.iloc[2:5,2]=[1000000,2000000,3000000]
2 tabla_prueba
```

Out[19]:

	Edad	Sexo	Sueldo	Estado_Civil
0	23	M	687455	S
1	46	M	1240000	C
2	32	M	1000000	C
3	28	F	2000000	C
4	45	F	3000000	D
5	55	F	781242	V
6	36	M	3200000	S
7	18	M	1200000	S

Fuente: Elaboración propia con *Jupyter Notebook*.

Una modificación más interesante podría contener ciclos de control. El siguiente código recorre todos los elementos de la columna edad y en cada elemento se evalúa sí el valor equivale a 32, de ser verdadero lo cambia a 50, de lo contrario no ejecuta ninguna operación:

Figura 2-32. Modificaciones de una tabla de datos mediante el uso de ciclos de control.

In [20]:
```
for i in range(0,len(tabla_prueba),1):
    if tabla_prueba.loc[i,"Edad"]==32:
        tabla_prueba.loc[i,"Edad"]=50
tabla_prueba
```

Out[20]:

	Edad	Sexo	Sueldo	Estado_Civil
0	23	M	687455	S
1	46	M	1240000	C
2	50	M	1000000	C
3	28	F	2000000	C
4	45	F	3000000	D
5	55	F	781242	V
6	36	M	3200000	S
7	18	M	1200000	S

Fuente: Elaboración propia con *Jupyter Notebook*.

2.2.4 Modificación del tipo de variable de una tabla de datos

En algunos casos se presenta que alguno de los elementos de la tabla de datos cuente con un tipo de dato que no se pueda manipular de una forma esperada. Por ejemplo:

Un número guardado como carácter no puede ser utilizado en operaciones matemáticas hasta que este sea un dato de tipo numérico. Suponga la creación de la tabla de datos trabajada en el capítulo por medio de arreglos:

Figura 2-33. Tabla de datos creada desde arreglos.

```
In [1]:
import pandas as pd
import numpy as np
edad=[23,46,16,32,28,45,55,36]
sexo=["M","M","F","M","F","F","F","M"]
ingreso=[687455,1240000,0,2550000,
         2850000,5450000,781242,3200000]
informacion=np.array([edad,sexo,ingreso])
data=pd.DataFrame(informacion)
data=data.T
data.columns=["Edad","Sexo","Ingreso"]
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Aparentemente la tabla de datos es correcta. Sin embargo, al evaluar uno de sus elementos, se pone en evidencia que el tipo de dato no es el apropiado para las variables “Edad” e “Ingreso”.

Figura 2-34. Un dato de ingreso se encuentra en formato carácter.

```
In [2]: type(data.Ingreso[5])
Out[2]: str
```

Fuente: Elaboración propia con *Jupyter Notebook*.

En este caso, no se especificó el tipo de dato a manejar en el arreglo y por defecto se colocó como tipo carácter. Esto impide aplicar funciones matemáticas como por ejemplo calcular la media de ingreso. Dicho inconveniente se podría evitar colocando en el arreglo el tipo de dato como objeto (*dtype=object*).

Ahora que la tabla de datos está creada se pueden usar las dos siguientes funciones para cambiar el tipo de dato carácter a numérico:

Figura 2-35. Cambio de tipo de dato de una variable de la tabla de datos.

```
In [3]:  1 tabla_2.Ingreso=pd.to_numeric(tabla_2.Ingreso,
         2                               downcast="float")
```

Fuente: Elaboración propia con *Jupyter Notebook.*

La figura anterior muestra la función general "*to_numeric"* del módulo *pandas*. El argumento "*downcast"* permite escoger el tipo de dato al cual se quiere convertir los datos, en ese caso se convirtió a tipo flotante. Otra forma de modificar el tipo de dato es la siguiente:

Figura 2-36. Otra forma de cambiar el tipo de dato en una tabla de datos.

```
In [4]:  1 tabla_2.Edad=tabla_2.Edad.astype(int)
```

Fuente: Elaboración propia con *Jupyter Notebook*.

En este caso se usó el método "*astype"* de la tabla de datos para transformar los datos a tipo entero. Con el nuevo tipo de datos se pueden aplicar operaciones matemáticas a los datos de Edad e Ingreso.

2.3 Filtros y ordenamiento de tablas de datos

Cuando se dispone de demasiada información, surge el requerimiento de mostrar solo partes interesantes de las tablas de datos en las cuales se desee realizar un análisis más específico. Por ejemplo, encontrar el valor de una variable cuando otra variable toma un

determinado valor puede mostrar información valiosa. El proceso de ordenar los datos también es muy utilizado en ciertas funciones o en el cálculo de parámetros.

2.3.1 Ordenamiento

Operaciones elementales como ordenar un conjunto de datos pueden realizarse con el método "*sort_values"* junto con su atributo "*ascending",* que toma por defecto un valor lógico verdadero.

De colocarse falso, los datos se ordenan de forma descendente (The pandas proyect, 2019). Remontándose a la tabla de datos de la Figura 2-4 se pueden ordenar sus elementos como se muestra en la siguiente figura:

Figura 2-37. Ordenamiento ascendente de una tabla de datos.

```
In [5]: 1 tabla.sort_values(["Sueldo"])
```

Out[5]:

	Edad	Sexo	Sueldo
2	16	F	0
0	23	M	687455
6	55	F	781242
1	46	M	1240000
3	32	M	2550000
4	28	F	2850000
7	36	M	3200000
5	45	F	5450000

Fuente: Elaboración propia con *Jupyter Notebook.*

Sí se añade el argumento "*ascending"* con el valor lógico falso se obtienen los datos ordenados en forma decreciente:

Figura 2-38. Datos ordenados decrecientemente en una tabla de datos.

```
In [6]: 1 tabla.sort_values(["Edad"],ascending=False)
```

Out[6]:

	Edad	Sexo	Sueldo
6	55	F	781242
1	46	M	1240000
5	45	F	5450000
7	36	M	3200000
3	32	M	2550000
4	28	F	2850000
0	23	M	687455
2	16	F	0

Fuente: Elaboración propia con *Jupyter Notebook.*

2.3.2 Selecciones condicionadas

En el tratamiento de datos es necesario filtrar información de acuerdo a unas condiciones dadas que pueden separar los datos para obtener información precisa y relevante.

Para filtrar en *Python* es práctico utilizar la función "*query"* ya que facilita aplicar múltiples condiciones (Ajay, 2018, p. 95).

A continuación se encuentra disponible con las tablas de datos del módulo *pandas*. La sintaxis para este procedimiento es diferente y poco deducible toda vez que el argumento correspondiente a las condiciones va entre comillas.

Figura 2-39. Subtabla resultante de un par de condiciones.

```
In [7]: 1 tabla.query('Edad>40 or Sexo=="F"')
```

Out[7]:

	Edad	Sexo	Sueldo
1	46	M	1240000
2	16	F	0
4	28	F	2850000
5	45	F	5450000
6	55	F	781242

Fuente: Elaboración propia con *Jupyter Notebook.*

En el ejemplo anterior se muestran las filas en las que se cumple que la edad es mayor que 40 años o en donde el sexo es femenino.

2.4 Ejercicios de la unidad 2

1. Importar el conjunto de datos "*Traffic"* tratado en esta unidad y modificarlo de tal manera que se elimine la columna 1 y se cambien los nombres de las columnas restantes por: "*año", "día", "límite" y "no_accidentes".*

2. Filtrar la tabla de datos para obtener el año y el día en el que se presentaron solo 7 accidentes. ¿Se aplicó el límite de velocidad para tal día?

3. La mediana es una medida de posición que se obtiene de los datos ordenados de forma creciente. Previo ordenamiento, corresponde al dato por encima del cual se encuentra una mitad de las observaciones y por debajo del cual se encuentra la otra mitad de las observaciones. Ordenar y utilizar la localización basada en enteros *(iloc)* para encontrar la mediana de accidentes presentados cuando se aplicó el límite de velocidad y cuando no se aplicó el límite de velocidad.

4. Encontrar en cuántos días se presentaron 12 o menos accidentes para:

    a. Los días en los que se aplicó el límite de velocidad.
    b. Los días en los que no se aplicó el límite de velocidad.

5. Usar la función "*max"* para encontrar la cantidad máxima de accidentes presentadas en el estudio y filtrar de acuerdo a esta para responder:

¿Se aplicó el límite de velocidad en el día en el que ocurrieron la mayor cantidad de accidentes de tránsito?

6. Crear un código que reemplace en los límites el resultado *"yes"* por *"si".*

3. Capítulo 3. Estadística

Disponiendo de una tabla de datos, el objetivo primordial radica en obtener información de esos datos organizados. La estadística se convierte en la herramienta adecuada para analizar los datos recolectados, describirlos e inferir sobre ellos.

En este capítulo se mostrará como:

- Importar el módulo *"matplotlib.pyplot"* para crear gráficos.
- Distinguir variables cualitativas y cuantitativas.
- Crear diagramas de barra, diagramas de torta, graficar líneas entre puntos, histogramas de frecuencia y diagramas de caja.
- Aplicar medidas de tendencia central y de dispersión.
- Aplicar medidas entre grupos de datos como la covarianza y la correlación.
- Elaborar pruebas de hipótesis para la media.

3.1 El concepto de estadística y sus objetivos

De acuerdo con Canavos (1998) "La estadística es el estudio de fenómenos aleatorios. El aspecto más importante de la estadística es la obtención de conclusiones basadas en datos experimentales, lo que se conoce como inferencia estadística"

En la estadística siempre se manejan los conceptos de población y muestra. La población consiste en todo el grupo de individuos u objetos que contienen una característica que se desea evaluar. Por su parte, la muestra es un subconjunto de la población que debe ser representativa de la población, es decir que refleje todas sus características y no caiga en una agrupación presente en una población. El propósito de la estadística consiste en cálculos estadísticos de la muestra llamados parámetros y hace inferencias sobre la

población de la cual se hizo la muestra, con el fin de derivar conclusiones. Esto indica que la estadística es inductiva porque va de lo particular a lo general. La estadística es una ciencia porque comprende una medida de confiabilidad en sus proposiciones, por lo que existe una cabida para el error, dado que expresa una probabilidad de que las inferencias sean correctas (Canavos, 1998).

3.2 Descripción grafica de los datos

La visualización de datos en gráficos es una herramienta útil para mejorar la comprensión de la información, detectar tendencias, límites o patrones en los datos. Dependiendo del tipo de dato se emplean diferentes herramientas gráficas para asociar las variables empleadas. Se distinguen dos grupos de tipos de datos que son los datos cualitativos y los datos cuantitativos.

3.2.1 Variables cualitativas

Las variables cualitativas son aquellas que se representan con cualidades, categorías o modalidades, no con números. Dentro de las variables cualitativas existen dos clasificaciones que son

- Variable cualitativa nominal: las cualidades no pueden ordenarse de alguna forma (ejemplo: sexo, color).

- Variable cualitativa ordinal: Las cualidades pueden ser ordenadas (ejemplo: nivel de estudios, rango de autoridad).

Teniendo una tabla de datos con variables cualitativas se puede contar la cantidad de veces que se repite determinada cualidad para una variable. Esto se denomina *frecuencia absoluta o frecuencia de repetición*.

La frecuencia absoluta se puede calcular en *Python* haciendo uso del atributo "*value_counts"* de los objetos tipo *"pandas.series."* Se importará el siguiente conjunto de datos contenido en los conjuntos de datos de "*R"* y cuyo nombre es *"salaries*", el cual

contiene los salarios de profesores en una universidad de los Estados Unidos (vincentarelbundock, 2019).

El conjunto de datos se encuentra disponible en la siguiente dirección:

https://vincentarelbundock.github.io/Rdatasets/csv/carData/Salaries.csv.

Figura 3-1. Importación del conjunto de datos *salaries.*

```
In [1]:
1 import pandas as pd
2 import numpy as np
3 salaries=pd.read_csv("https://vincentarelbundock.github.io"
4                     +"/Rdatasets/csv/carData/Salaries.csv")
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 3-2. Descripción de la tabla de datos *salaries.*

```
In [2]:
1 salaries.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 397 entries, 0 to 396
Data columns (total 7 columns):
Unnamed: 0       397 non-null int64
rank             397 non-null object
discipline       397 non-null object
yrs.since.phd    397 non-null int64
yrs.service      397 non-null int64
sex              397 non-null object
salary           397 non-null int64
dtypes: int64(4), object(3)
memory usage: 17.1+ KB
```

Fuente: Elaboración propia con *Jupyter Notebook*.

Antes de aplicar el atributo “*value_counts”*, se eliminará la columna *“Unnamed*: 0” ya que no tiene ningún propósito y se sobrescribirá la tabla de datos para guardar dicha modificación. Además, se reemplazarán los puntos por guiones en los nombres de las columnas para evitar posibles errores a la hora de hacer referencias sobre estos:

Figura 3-3. Eliminación de la columna "*Unnamed:* 0" de la tabla de datos *salaries.*

```
In [3]:  1  salaries=salaries.drop("Unnamed: 0",1)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Figura 3-4. Adecuación del nombre de las columnas de la tabla de datos *salaries.*

```
In [4]:  1  nombre_columnas=["rank","discipline",
         2                  "yrs_since_phd",
         3                  "yrs_service",
         4                  "sex",
         5                  "salary"]
         6  salaries.columns=nombre_columnas
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Ahora se aprovechará el método “*value_counts”* para contar la cantidad de profesores que son mujeres y la cantidad de profesores que son hombres, es decir, la frecuencia absoluta.

Figura 3-5. Frecuencia absoluta para el sexo en la tabla "*salaries".*

```
In [5]:  1 salaries.sex.value_counts()

Out[5]: Male      358
        Female     39
        Name: sex, dtype: int64
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Con este resultado se evidencia que la mayoría de profesores en esa universidad son hombres.

Además de la frecuencia absoluta, existe la frecuencia relativa, calculada como el cociente entre el número de observaciones de una cualidad y el total de observaciones de la variable cualitativa evaluada.

Figura 3-6. Frecuencias relativas para la variable sexo de la tabla *salaries.*

```
In [6]:  1 salaries.sex.value_counts()/len(salaries.sex)

Out[6]: Male      0.901763
        Female    0.098237
        Name: sex, dtype: float64
```

Fuente: Elaboración propia con *Jupyter Notebook.*

A partir de estos datos se puede graficar un diagrama de barras para visualizar la cantidad de hombres y mujeres en esta tabla de datos. Para acceder a los gráficos, se va a importar y trabajar con el módulo "*matplotlib.pyplot"* con el cual se pueden graficar diagramas de barras y muchas otras gráficas. Toda la referencia de la interfaz de programación de la aplicación para esta librería se encuentra en:

https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html#module-matplotlib.pyplot.

Figura 3-7. Importación del módulo *"matplotlib.pyplot"*.

```
In [7]:  1  import matplotlib.pyplot as plt
```

Fuente: Elaboración propia con *Jupyter Notebook*.

Una vez cargado el módulo, se utilizará la función *bar* para graficar un diagrama de barras como se muestra en la siguiente figura:

Figura 3-8. Código empleado para crear un diagrama de barras.

```
In [8]:  1  plt.bar(["Masculino"],salaries.sex.value_counts()[0],
         2          color="blue",
         3          label="Masculino")
         4  plt.bar(["Femenino"],salaries.sex.value_counts()[1],
         5          color="pink",
         6          label="Femenino")
         7  plt.xlabel("Sexo")
         8  plt.ylabel("Frec. absoluta")
         9  plt.title("Diagrama de barras \n del sexo de los profesores")
        10  plt.legend()
        11  plt.show()
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Para cada barra se utiliza la función "*plt.bar".* En el primer argumento se coloca el vector que contiene el nombre que se le colocará a la barra, el segundo argumento es la frecuencia absoluta tomada desde el método "*value_counts"*, en donde se obtuvo que el primer valor corresponde a la frecuencia absoluta para los hombres y el segundo es para las mujeres.

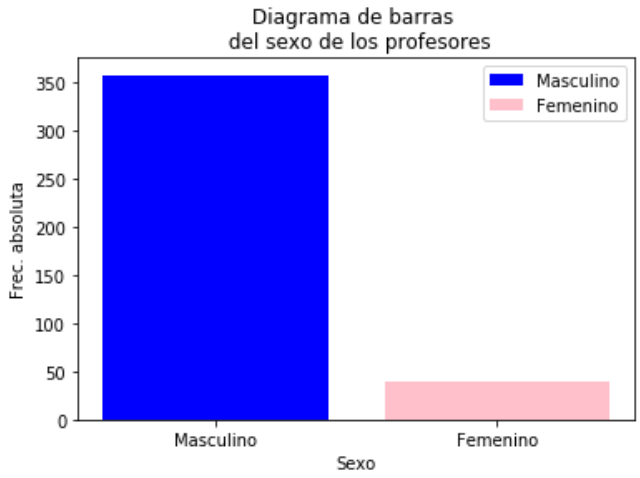
El parámetro *color* le da un color a la barra, estos colores se pueden encontrar en el diccionario "*matplotlib.color.cnames"*.

El parámetro "*label"* es el nombre que aparecerá en la leyenda del gráfico, la cual se introduce con la función "*legend"*.

Con la función "*xlabel"* se le da el nombre al eje horizontal del gráfico y con la función "*ylabel"* se le da el nombre al eje vertical. La función "*title"* se le da el nombre al gráfico y finalmente la función "*show"* lo despliega.

El resultado del código de la Figura 3-8 es el que sigue:

Gráfico 3-1. Diagrama de barras del sexo de la tabla *"salaries".*



Fuente: Elaboración propia con *Jupyter Notebook.*

Otro tipo de gráfico muy usado para representar estos casos es el diagrama circular, el cual, en vez de utilizar barras, divide un círculo en áreas que representan la frecuencia de la clase. La función para construir diagramas circulares presente en el módulo importado se denomina "*pie".*

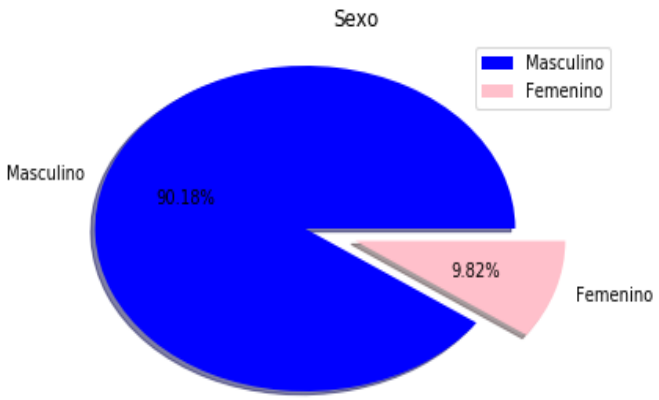
Figura 3-9. Código empleado para crear un diagrama circular.

```
In [9]:
plt.pie(salaries.sex.value_counts(),
        labels=["Masculino","Femenino"],
        colors=["blue", "pink"],
        explode=[0.25,0],
        shadow=True,
        autopct="%1.2f%%")
plt.legend()
plt.title("Sexo")
plt.show()
```

Fuente: Elaboración propia apoyada con Sentdex, 2015

En el primer argumento de esta función se colocan las frecuencias absolutas para la variable sexo. El argumento "*explode"* crea un explosionado del gráfico, el argumento "*shadow"* le da un aspecto tridimensional al gráfico y el argumento "*autopc"* coloca en el diagrama los porcentajes de la cualidad respecto al total de elementos de la variable, en el que se puede especificar la cantidad de números decimales a mostrar. El gráfico resultante es el siguiente:

Gráfico 3-2. Diagrama circular para la variable sexo de la tabla "*salaries"*.



Fuente: Elaboración propia apoyada con Sentdex, 2015.

Otra herramienta muy característica para los datos cualitativos es la *tabla de contingencia*, empleada en probabilidad, la cual es una tabla multidimensional que se construye a partir de variables cualitativas para conocer la asociación entre estas. Para crear una tabla de contingencia se utiliza la función general del módulo pandas "*crosstab"*. Como argumentos se tienen las variables que se van a tabular y se añade el parámetro "*margins"* para añadir los totales por filas y columnas.

Figura 3-10. Tabla de contingencia para las variables cualitativas de la tabla *salaries.*

```
In [10]:
1 pd.crosstab(salaries.sex,[salaries.discipline,
2                           salaries["rank"]],
3                           margins=True)
```

Out[10]:

discipline	A			B			All
rank	AssocProf	AsstProf	Prof	AssocProf	AsstProf	Prof	
sex							
Female	4	6	8	6	5	10	39
Male	22	18	123	32	38	125	358
All	26	24	131	38	43	135	397

Fuente: Elaboración propia con *Jupyter Notebook.*

3.2.2 Datos cuantitativos

Los datos cuantitativos toman valores numéricos y se dividen en variables cuantitativas discretas o variables cuantitativas continuas.

a. Las variables cuantitativas discretas toman valores aislados, visto desde la programación son variables que toman valores enteros. Un ejemplo de esto sería la cantidad de hijos de un individuo.

b. Las variables cuantitativas continuas vistas desde un punto de vista de programación serían valores numéricos de punto flotante y un ejemplo sería la cantidad de líquido en determinada botella de gaseosa.

Cuando se trabaja con variables cuantitativas, estas se agrupan en intervalos de igual o diferente longitud dependiendo de los datos, así como una cantidad de intervalos apropiada. Se calcula la frecuencia absoluta como la cantidad de datos que hay en cada intervalo y se grafica para producir un histograma de frecuencia.
Si se utiliza la frecuencia relativa en lugar de la frecuencia absoluta, se obtiene la distribución de frecuencia relativa con la cual se puede observar tendencias en los datos.

Para crear un histograma se debe entonces manejar un número de intervalos apropiado. Existen reglas para calcular la cantidad de intervalos y estas se pueden especificar en las funciones de *Python*.

Para dibujar un histograma se utiliza la función "*hist"* donde se coloca la serie o lista de datos, los intervalos se definen con el argumento "*bins"*, y se añaden otros parámetros que se ven a continuación.

El siguiente código muestra un histograma para los años transcurridos desde el PhD:

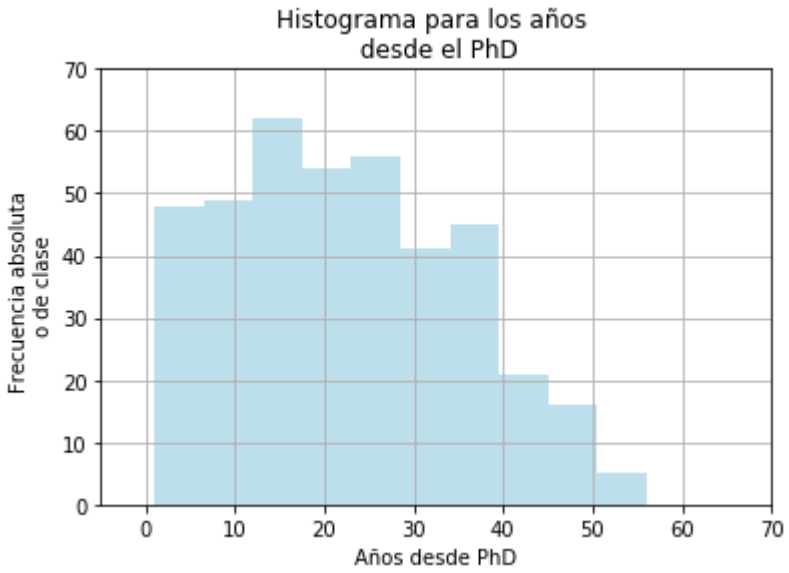
Figura 3-11. Código para generar histograma de frecuencias.

In [11]:
```
plt.hist(salaries.yrs_since_phd,
         color="lightblue",
         alpha=0.8)
plt.xlabel("Años desde PhD")
plt.ylabel('Frecuencia absoluta \n o de clase')
plt.title("Histograma para los años \n desde el PhD")
plt.axis([-5,70,0,70])
plt.grid(True)
plt.show()
```

Fuente: Elaboración propia con *Jupyter Notebook.*

En este caso, el parámetro "*alpha"* es un número que modifica la opacidad del gráfico, la función *axis* cambia los límites en el gráfico en cada eje en el orden "*xmin"*, "*xmax"*, "*ymin"* y "*ymax",* la función "*grid"* añade una cuadricula al gráfico. El resultado generado por la figura anterior se muestra a continuación:

Gráfico 3-3. Histograma de frecuencias absolutas para los años desde el PhD.



Fuente: Elaboración propia con *Jupyter Notebook.*

Dado que no se especificó el argumento "*bins"*, la función *"hist"* genera por defecto 10 intervalos de la misma longitud. Como se había mencionado antes, existen varias reglas empleadas para crear los intervalos y aquellas que se pueden especificar en esta función se pueden encontrar en:

https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram_bin_edges.html.

También es posible especificar el número de intervalos con un número entero, los cuales tendrían la misma longitud o mediante una lista se pueden especificar los límites de los

intervalos, los cuales son cerrados por izquierda y abiertos por derecha, exceptuando el último intervalo que es cerrado en ambos extremos.

La función "*hist"* en el módulo "*matplotlib.pyplot"* que se ha utilizado para graficar el histograma usa la función "*numpy.histogram"*, del cual resultan dos arreglos: El primero la frecuencia absoluta dados 10 intervalos de la misma longitud (en caso de que no se especifique un argumento para los intervalos) y el segundo arreglo contiene los límites de los intervalos según la regla que se adoptada para crearlos como se muestra en la siguiente figura:

Figura 3-12. Obtención de frecuencias absolutas y límites de intervalos con "*np.histogram".*

```
1  np.histogram(salaries.yrs_since_phd)
```

```
(array([48, 49, 62, 54, 56, 41, 45, 21, 16,  5], dtype=int32),
 array([ 1. ,  6.5, 12. , 17.5, 23. , 28.5, 34. , 39.5, 45. , 50.5, 56. ]))
```

Fuente: Elaboración propia con *Jupyter Notebook.*

En el siguiente ejemplo se muestran tres alternativas diferentes para crear los intervalos y para cada una se crea un gráfico.

La longitud y cantidad de intervalos depende de la distribución de los datos, y por ende no es definitivo utilizar siempre una forma de agrupación para crear los intervalos de cualquier conjunto de datos. Dependerá entonces, de qué tan grande sea la diferencia entre el dato mayor y el menor (rango) y de si los datos están muy concentrados en una parte de la distribución.

Figura 3-13. Código para crear tres histogramas con diferentes intervalos.

In [13]:

```
plt.subplot(2, 1, 1)
plt.hist(salaries.yrs_since_phd,
         bins=12,
         facecolor="gray")
plt.title("Histograma de los años \n desde PhD")
plt.ylabel("Frec. absoluta"),
plt.axis([-5,70,0,70])
plt.grid(True)
plt.show()

plt.subplot(2, 1, 1)
plt.hist(salaries.yrs_since_phd,
         bins="sturges",
         facecolor="lightblue")
plt.ylabel('Frec. absoluta')
plt.axis([-5,70,0,70])
plt.grid(True)
plt.show()

plt.subplot(2, 1, 1)
plt.hist(salaries.yrs_since_phd,
         bins=list(range(0,70,5)),
         facecolor="orange")
plt.xlabel("Años desde PhD")
plt.ylabel("Frec. absoluta")
plt.axis([-5,70,0,70])
plt.grid(True)
plt.show()
```

Fuente: Elaboración propia con *Jupyter Notebook*.

La función "*subplot"* se emplea usa para mostrar varios gráficos en una sola salida.

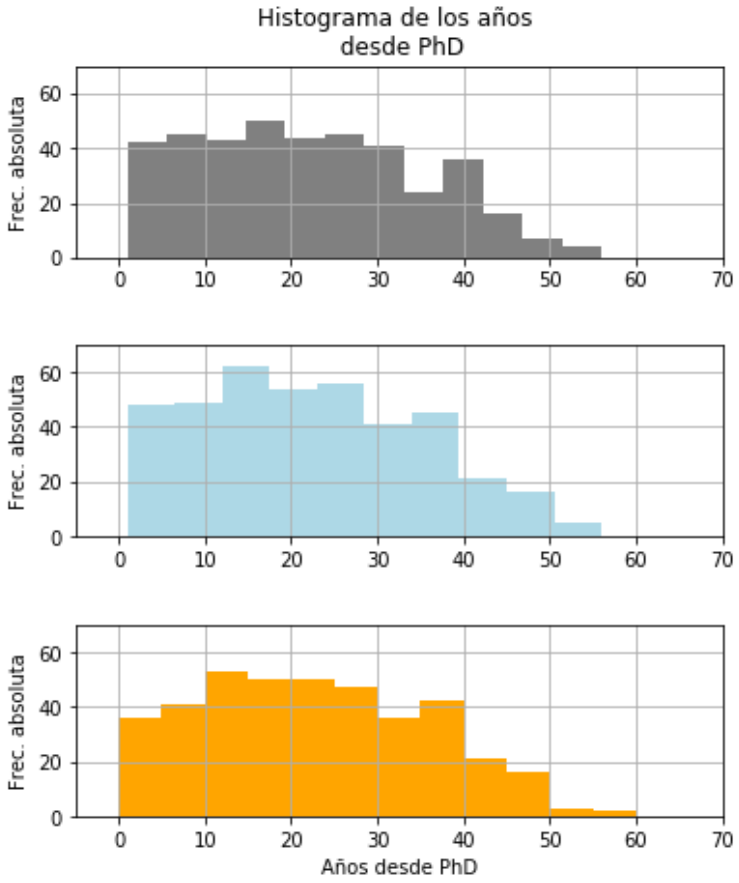
El primer argumento de esta función modifica el tamaño del eje vertical, el segundo argumento es igual que el primero, pero para el eje horizontal, y el último argumento es una indexación. En la primera función se establecieron 12 intervalos de igual longitud, en la segunda función se establecieron los intervalos mediante la regla de "*Sturges"*, en la cual se define que el número de intervalos es igual a 1 más el cociente entre logaritmo en base 10 de la muestra y el logaritmo en base 10 de 2.

Para la última gráfica se establecieron los límites con una lista creada desde un rango que contiene los números 0, 5, 10… ,70.

En este caso se toma el primer intervalo desde 0 cerrado hasta 5 abierto. El segundo sería de 5 cerrado a 10 abierto y así sucesivamente, exceptuando el último intervalo que es cerrado por ambos extremos. Es decir [65,70], dando lugar a un total de 14 intervalos.

Con esta última opción es posible crear intervalos de diferente longitud. Todos los gráficos son diferentes por la forma de aplicar los intervalos:

Gráfico 3-4. Tres histogramas creados con diferente regla para las clases.



Fuente: Elaboración propia con *Jupyter Notebook*.

También es posible crear un histograma utilizando el método "*hist"* de los objetos "*pandas.DataFrame"* el cual se basa en el módulo "*matplotlib.pyplot.hist"*:

Figura 3-14. Código para un histograma desde un método.

In [14]:
```
plt.figure(figsize=(9,5))
salaries.yrs_service.hist(bins="sturges",
                          color="yellow",
                          xlabelsize=15,
                          ylabelsize=15)
plt.xlabel("Años de servicio", fontsize=15)
plt.ylabel("Frec. absoluta", fontsize=15)
plt.title("Histograma para los años de servicio", fontsize=15)
plt.xticks(np.arange(0, 70, 5))
plt.show()
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Se han añadido otras líneas. La función "*plt.figure"* con el argumento "*figsize"* permite modificar las dimensiones del gráfico para que sea más grande o más pequeño. El argumento "*xlabelsize"* cambia el tamaño de la numeración para el eje x, así como "*ylabelsize"* lo hace para la numeración del eje y. La función "*xticks"* cambia la división del eje x mediante un rango "*numpy".* También se puede modificar la del eje y mediante la función "*yticks"*.

El resultado obtenido es el siguiente:

Gráfico 3-5. Histograma para los años de servicio de la tabla *salaries.*

Fuente: Elaboración propia con *Jupyter Notebook.*

Claramente se evidencia que hay pocos profesores que llevan aproximadamente entre 12 a 17 años de servicio, comparado con aquellos que llevan entre 0 y 10 años de servicio o entre 18 y 23 años de servicio. Cabe anotar que también es notable que la distribución está sesgada.

Para obtener la frecuencia relativa se debe dividir la frecuencia absoluta entre el número total de observaciones para cada clase. Retomando la variable años desde el PhD, la frecuencia relativa sería la siguiente:

Figura 3-15. Obtención de frecuencias relativas para los años desde el PhD.

```
1  fr=np.histogram(salaries.yrs_since_phd)[0]/len(salaries)
2  fr
```

```
array([0.1209068 , 0.12342569, 0.15617128, 0.13602015, 0.14105793,
       0.10327456, 0.11335013, 0.05289673, 0.04030227, 0.01259446])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Posteriormente, con la función "*cumsum"* del módulo "*numpy"* se pueden calcular las frecuencias relativas acumuladas:

Figura 3-16. Frecuencias relativas acumuladas para los años desde el PhD.

```
1  Fr=np.cumsum(fr)
2  Fr
```

```
array([0.1209068 , 0.24433249, 0.40050378, 0.53652393, 0.67758186,
       0.78085642, 0.89420655, 0.94710327, 0.98740554, 1.        ])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Si se desea el graficar la frecuencia relativa se puede digitar el siguiente código utilizando el parámetro "*weight"*. Suponga que se desea obtener la frecuencia relativa para los años desde el PhD.

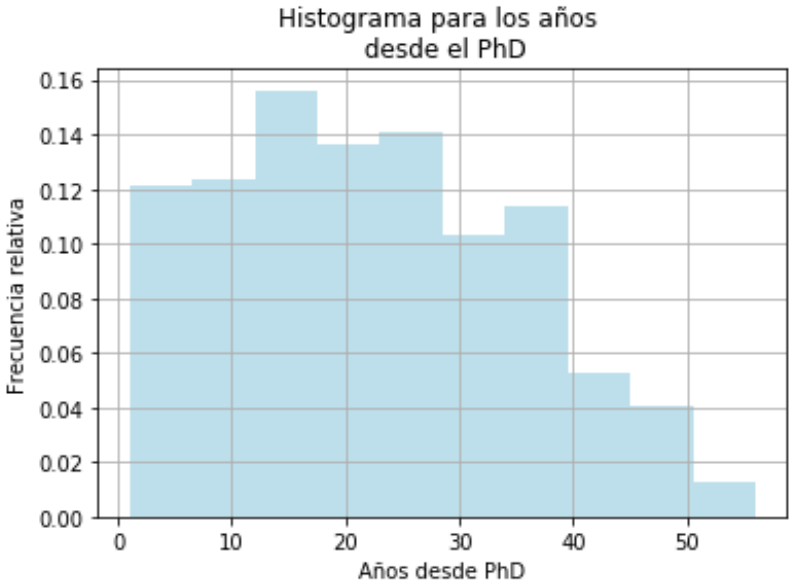
Figura 3-17. Código empleado para graficar un histograma de frecuencias relativas.

In [17]:
```
1  plt.hist(salaries.yrs_since_phd,
2           weights=(np.zeros_like(salaries.yrs_since_phd)+1/
3                    salaries.yrs_since_phd.size),
4           color="lightblue",
5           alpha=0.8)
6  plt.xlabel("Años desde PhD")
7  plt.ylabel('Frecuencia relativa')
8  plt.title("Histograma para los años \n desde el PhD")
9  plt.grid(True)
10 plt.show()
```

Fuente: Elaboración propia con *Jupyter Notebook.*

El parametro "*weights"* le da un peso igual a cada elemento del vector. El resultado es el siguiente:

Gráfico 3-6. Histograma de frecuencias relativas para los años desde el PhD.



Fuente: Elaboración propia con *Jupyter Notebook.*

La *frecuencia relativa acumulada* según George Canavos (1998):

> […] es la proporción de observaciones cuyos valores son menores o iguales al límite superior de la clase o, en forma equivalente, menores que el límite inferior de la siguiente clase […] el principal uso de la distribución acumulativa es lo que comúnmente se conoce como *cuantiles*. Con respecto a una distribución de frecuencia relativa acumulada, se define un cuantil como el valor bajo el cual se encuentra una determinada proporción de los valores de la distribución. (p. 7)

Para obtener la distribución de frecuencia relativa acumulada se usa el parámetro booleano "*cumulative"* al gráfico generado con el código de la Figura 3-17. Otra opción es usar el código de la Figura 3-11 añadiendo el parámetro anterior junto con el parámetro booleano "*density"* al histograma, lo cual genera como resultado el valor de la función de densidad de probabilidad para el intervalo, normalizada de tal forma que el área bajo la curva sea 1 (SciPy org, 2019).

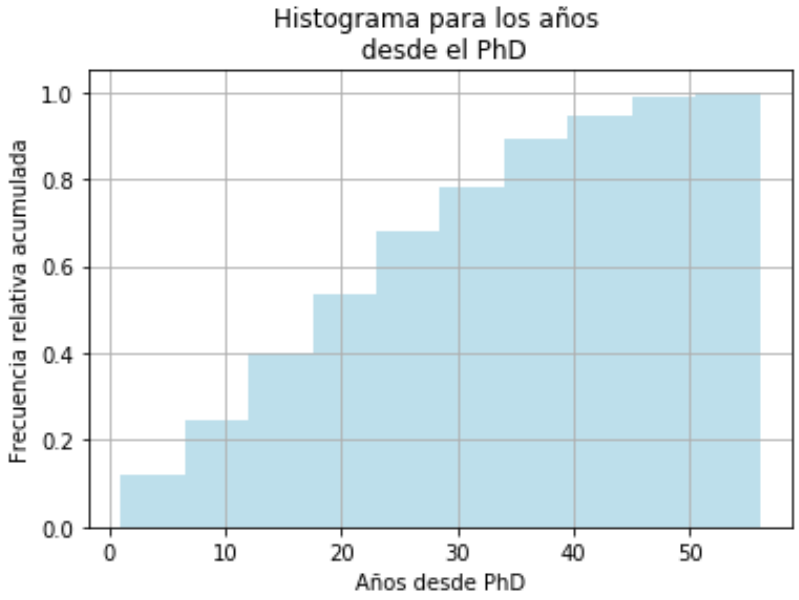
Figura 3-18. Código para crear un histograma de frecuencias relativas acumuladas.

```
In [18]:
plt.hist(salaries.yrs_since_phd,
         cumulative=True,
         density=True,
         color="lightblue",
         alpha=0.8)
plt.xlabel("Años desde PhD")
plt.ylabel('Frecuencia relativa acumulada')
plt.title("Histograma para los años \n desde el PhD")
plt.grid(True)
plt.show()
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Con base en lo anterior se genera el siguiente gráfico:

Gráfico 3-7. Histograma de frecuencias relativas acumuladas.



Fuente: Elaboración propia con *Jupyter Notebook.*

La distribución de frecuencia relativa acumulada se presenta mejor graficando una curva. Recordando que los cuantiles son el valor por debajo del cual se encuentra una determinada cantidad de datos, se puede calcular los siguientes valores:

Figura 3-19. Obtenciones de intervalos y frecuencias relativas acumuladas para los años desde el PhD.

```
1 histograma=np.histogram(salaries.yrs_since_phd)
2 x=list(histograma[1])
3 y=histograma[0]/len(salaries.yrs_since_phd)
4 y=np.concatenate(([0],y))
5 y_acum=np.cumsum(y)
6 print(x)
7 print(y_acum)
```

```
[1.0, 6.5, 12.0, 17.5, 23.0, 28.5, 34.0, 39.5, 45.0, 50.5, 56.0]
[0.         0.1209068  0.24433249 0.40050378 0.53652393 0.67758186
 0.78085642 0.89420655 0.94710327 0.98740554 1.        ]
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Se extrajeron los límites de los intervalos teniendo en cuenta que por defecto se generan 10 intervalos de la misma longitud. También se extrajeron los valores de la frecuencia absoluta y de esta se calculó la frecuencia acumulada con la función "*cumsum"*.

Teniendo los dos arreglos "*x"* y "*y_acum"* calculados en la figura anterior se puede graficar la distribución de frecuencia relativa acumulada:

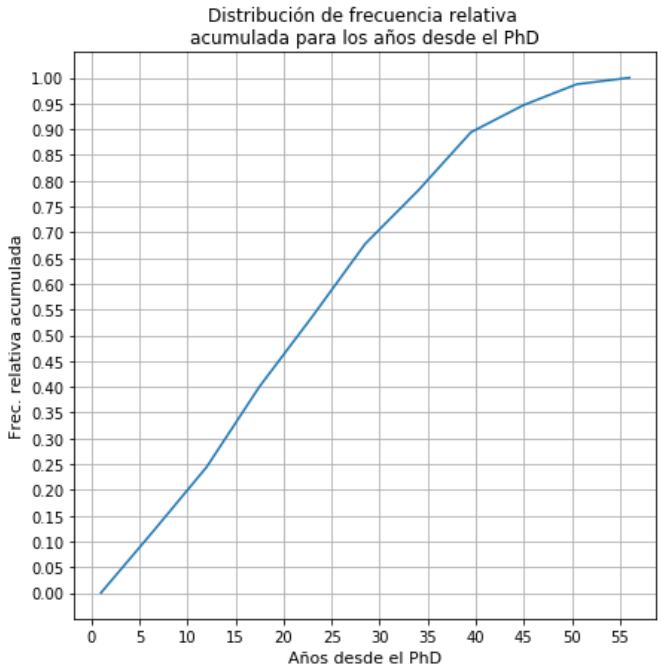
Figura 3-20. Código para crear un gráfico de la distribución de frecuencia relativa acumulada.

```
In [20]:
1  plt.figure(figsize=(7,7))
2  plt.plot(x,y_acum)
3  plt.grid(True)
4  plt.xticks(np.arange(0,60,5))
5  plt.yticks(np.arange(0,1.05,0.05))
6  plt.xlabel("Años desde el PhD", fontsize=11)
7  plt.ylabel("Frec. relativa acumulada", fontsize=11)
8  plt.title("Distribución de frecuencia relativa \n"
9  +"acumulada para los años desde el PhD")
10 plt.show()
```

Fuente: Elaboración propia con *Jupyter Notebook.*

El anterior código produce el siguiente gráfico:

Gráfico 3-8. Distribución de frecuencia relativa acumulada para los años desde el PhD.



Fuente: Elaboración propia con *Jupyter Notebook.*

Se debe tener en cuenta que se han manejado los límites de los intervalos (que fueron calculados por defecto) como *límites de escritura,* puesto que reflejan el mismo grado de precisión que el de las observaciones presentadas. La otra alternativa es usar *límites verdaderos* que reflejan la unidad más pequeña que se emplea para tomar las observaciones (Canavos, 1998, p. 6).

3.3 Medidas numéricas descriptivas

Según Canavos (1998):

> Son medidas numéricas empleadas para describir conjuntos de datos […] Existen dos medidas de interés para cualquier conjunto de datos: la localización de su centro y su variabilidad. La tendencia central de un conjunto de datos es la disposición de estos para agruparse ya sea alrededor del centro o de ciertos valores numéricos. La variabilidad de un conjunto de datos es la dispersión de las observaciones en el conjunto. Existen principalmente tres medidas de tendencia central: la media, la mediana y la moda (pp. 11-12).

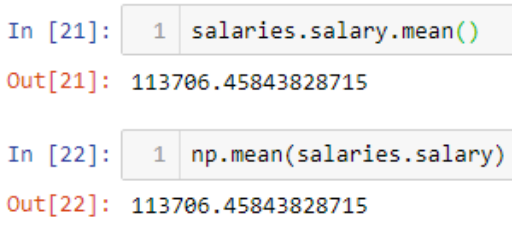
3.3.1 Media de una muestra

La *media* de un conjunto de observaciones x_i se define como:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \qquad (3)$$

Desde *Python* se puede calcular mediante el método *mean* de las series o tablas de datos o a través de la función *mean* del módulo "*numpy"*, por ejemplo:

Figura 3-21. Cálculo de una media de un conjunto de observaciones.



Fuente: Elaboración propia con *Jupyter Notebook.*

3.3.2 Mediana de una muestra

Es el valor para el cual, cuando todas las observaciones se ordenan de manera creciente, la mitad de estas es menor que este valor y la otra mitad mayor. Si la cantidad de datos es par, la mediana se calcula como el promedio de los dos valores que se encuentran en la mitad del conjunto ordenado. Adicionalmente la mediana se puede calcular desde la distribución acumulada puesto que es el percentil cincuenta. Esta medida es útil porque no se encuentra influida por valores extremos (Canavos, 1998, p. 12).

La mediana para un conjunto de n observaciones ordenadas de menor a mayor se calcula como:

$$\tilde{X} = \begin{cases} x_{\frac{n+1}{2}} & si\ n\ es\ impar \\ \dfrac{\left(x_{\left(\frac{n}{2}\right)} + x_{\left(\frac{n}{2}\right)+1}\right)}{2} & si\ n\ es\ par \end{cases} \qquad (4)$$

En Python, el módulo "*numpy"* cuenta con la función *median* :

Figura 3-22. Cálculo de la mediana de un conjunto de observaciones.

```
In [23]:  1  np.median(salaries.yrs_since_phd)
Out[23]: 21.0
```

Fuente: Elaboración propia con *Jupyter Notebook.*

La mediana que es también el percentil cincuenta, se puede calcular con el método "*quantile"* de las tablas de datos. Existen varios parámetros para la interpolación, la interpolación por defecto es lineal.

Figura 3-23. Cálculo de percentiles con el método "*quantile".*

```
In [24]:  1  salaries.yrs_since_phd.quantile([0.1,0.5,0.9])
Out[24]: 0.1     5.0
         0.5    21.0
         0.9    40.0
         Name: yrs_since_phd, dtype: float64
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Cuando se utiliza la mediana es muy común utilizar un "*boxplot"* (diagrama de caja), creado por John W. Tukey, en el cual se encuentran varias características de los datos que son: el máximo valor, el tercer cuartil, la mediana, el primer cuartil, el rango intercuartil (RIC), el mínimo valor y valores atípicos sospechosos o no sospechosos.

Los valores atípicos sospechosos son aquellos que están al menos 1,5 veces el RIC unidades por encima del tercer cuartil o 1,5 veces el RIC unidades por debajo del primer cuartil. Los valores atípicos son los que están al menos 3 veces el RIC unidades por encima que el tercer cuartil o aquellos que se encuentran al menos 3 veces el RIC unidades por

debajo del primer cuartil (Kirkman, 1996). Este gráfico ayuda a identificar valores extremos que podrían ser falsos o que fueron digitados incorrectamente.

Suponga que se desea elaborar el diagrama de caja para los salarios de la tabla de datos "salaries". Para tal efecto. el módulo "*matplotlib.pyplot"* dispone de la función "*boxplot"* para graficar dicho diagrama. Para crear un gráfico con más detalles se recomienda utilizar el módulo "*seaborn"*.

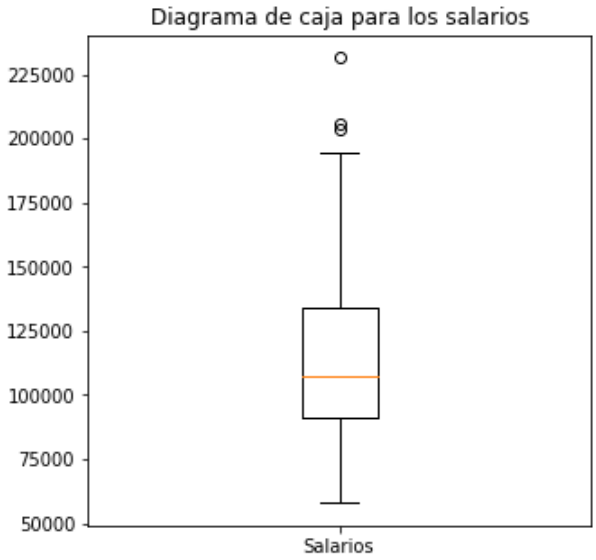
Figura 3-24. Código empleado para graficar un diagrama de caja.

```
plt.figure(figsize=(5,5))
plt.boxplot(salaries.salary,
            labels=['Salarios'])
plt.title("Diagrama de caja para los salarios")
plt.show()
```

Fuente: Elaboración propia con *Jupyter Notebook.*

El gráfico resultante del código anterior es el siguiente:

Gráfico 3-9. Diagrama de caja para los salarios.



Fuente: Elaboración propia con *Jupyter Notebook.*

3.4 Medidas de variabilidad

Las medidas de variabilidad se utilizan para medir la dispersión o cercanía entre los datos de una muestra. Tres medidas de variabilidad se enumeran a continuación:

3.4.1 Rango de una muestra

Es una medida de dispersión sencilla calculada con diferencia entre el dato mayor y el dato menor de la muestra. El rango entonces no tiene en cuenta todos los datos y pierde relevancia cuando se presentan datos extremos.

$$R = x_{max} - x_{min} \quad (5)$$

En *Python* se pueden usar las funciones *"max"* y *min* para calcular el rango de una muestra como se muestra a continuación:

Figura 3-25. Rango de los salarios en la tabla *salaries.*

```
In [25]:  1 R=max(salaries.salary)-min(salaries.salary)
          2 R

Out[25]: 173745
```

Fuente: Elaboración propia con *Jupyter Notebook.*

3.4.2 Varianza de una muestra

La *varianza muestral* ha sido definida por Canavos (1998) como: "La varianza de las observaciones x_i es, en esencia, el promedio del cuadrado de las distancias entre cada observación y la media del conjunto de observaciones. La varianza se denota por s^2" (p. 15).

$$s^2 = \sum_{i=1}^{n}(x_i - \bar{x})^2/(n-1) \quad (6)$$

3.4.3 Desviación estándar de una muestra

La medida de dispersión más usada es la *desviación estándar muestral* que es la raíz cuadrada positiva de la varianza muestral y se denota por "*s*", luego la desviación estándar muestral para un conjunto de observaciones se calcula como:

$$s = \sqrt{\sum_{i=1}^{n}\frac{(x_i-\bar{x})^2}{(n-1)}} \quad (7)$$

Estas medidas de dispersión se pueden calcular con funciones del módulo "*statistics*", presente en los módulos numéricos, matemáticos de *Python* y con el módulo "*numpy".*

Figura 3-26. Cálculo de la varianza muestral de los salarios.

```
In [26]:  1 import statistics
          2 statistics.variance(salaries.salary)

Out[26]: 917425865.0519298
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Con la librería "*numpy"* se dispone de la función "*var"* que calcula la *varianza poblacional* de la muestra. Para calcular la varianza muestral, aquella dividida entre los grados de libertad de la varianza $(n-1)$ desde la varianza poblacional, se puede multiplicar la varianza muestral por la cantidad de datos n dividido entre los grados de libertad de la varianza $(n-1)$.

Figura 3-27. Cálculo de la varianza muestral con el módulo "*numpy"*.

```
In [27]:  1 np.var(salaries.salary)*(len(salaries)/(len(salaries)-1))

Out[27]: 917425865.0519297
```

Fuente: Elaboración propia con *Jupyter Notebook.*

La desviación estándar muestral se calcula al utilizar la función "*stdev"* del módulo "*statistics"* o se obtiene sacando la raíz cuadrada de la varianza muestral obtenida en la Figura 3-27.

Figura 3-28. Cálculo de la desviación estándar muestral.

```
In [28]:  1  statistics.stdev(salaries.salary)
Out[28]: 30289.038694747804
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Una función en las tablas de datos que permite calcular medidas estadísticas descriptivas como las anteriores es "*describe"*. Dicha función calcula la media, la desviación estándar muestral, el valor mínimo, el valor máximo los cuartiles y la cantidad o cuenta.

Figura 3-29. Uso de la función *describe* para obtener medidas descriptivas.

```
In [30]:  1  salaries.describe()
```

Out[30]:

	yrs_since_phd	yrs_service	salary
count	397.000000	397.000000	397.000000
mean	22.314861	17.614610	113706.458438
std	12.887003	13.006024	30289.038695
min	1.000000	0.000000	57800.000000
25%	12.000000	7.000000	91000.000000
50%	21.000000	16.000000	107300.000000
75%	32.000000	27.000000	134185.000000
max	56.000000	60.000000	231545.000000

Fuente: Elaboración propia con *Jupyter Notebook.*

En la tabla de datos "*salaries*" se presentan datos cualitativos que pueden ser aislados para obtener un estadístico más específico. Por ejemplo, existen tres tipos de cargos de profesores que son profesor, profesor asistente y profesor asociado.
Un cálculo interesante podría ser el salario promedio para cada cargo con el objetivo de comparar y obtener el cargo que más salario devenga en promedio.

Este tipo de agrupaciones se pueden realizar mediante el método "*groupby"* de las tablas de datos especificando el nombre de la variable que se va a dividir y con el método "*agg"* se añaden operaciones o funciones que se desean calcular para cada columna o fila según se especifique. Claramente se evidencian diferencias en cada variable según el rango del profesor.

Figura 3-30. Media y desviación estándar para cada rango de profesor.

```
1 salaries.groupby("rank").agg(["mean","std"])
```

	yrs_since_phd		yrs_service		salary	
	mean	std	mean	std	mean	std
rank						
AssocProf	15.453125	9.652584	11.953125	10.100180	93876.437500	13831.699844
AsstProf	5.104478	2.541381	2.373134	1.495811	80775.985075	8174.112637
Prof	28.300752	10.108830	22.815789	11.590493	126772.109023	27718.674999

Fuente: Elaboración propia apoyado en Ohri Ajay, 2018, p. 122

3.5 Covarianza y correlación

La covarianza entre dos variables es una medida de la naturaleza de la relación entre variables aleatorias (Walpole, Myers, & Myers, 1998). La covarianza para dos variables aleatorias X y Y es:

$$\sigma_{XY} = E(XY) - \mu_X \mu_Y \quad (8)$$

Donde E es la esperanza matemática o media y μ_X y μ_Y son las medias de las respectivas variables aleatorias. Si la covarianza es positiva, cuando una variable aleatoria incrementa, la otra variable también incrementa, pero si la covarianza es negativa, cuando incrementa una variable aleatoria, la otra variable disminuye.

La covarianza entre las variables o columnas de una tabla de datos se calcula con la función "*cov"* de los objetos "*pandas.DataFrame"*. La covarianza que calcula es muestral. En la siguiente Figura se calcula la covarianza para las variables numéricas de la tabla de datos "*salaries"*:

Figura 3-31. Covarianzas entre las variables de la tabla *salaries.*

```
1 salaries.cov()
```

	yrs_since_phd	yrs_service	salary
yrs_since_phd	166.074854	152.465085	1.636405e+05
yrs_service	152.465085	169.156655	1.318693e+05
salary	163640.549735	131869.303387	9.174259e+08

Fuente: Elaboración propia con *Jupyter Notebook*.

La covarianza puede mostrar la relación entre las variables aleatorias, sin embargo, no representa el grado en el que están relacionadas, ya que la covarianza depende de la escala (Walpole, Myers, & Myers, 1998).

Para evitar dicho inconveniente se crea el *coeficiente de correlación* el cual es libre de escala. Este coeficiente es ampliamente utilizado en estadística para encontrar relaciones lineales entre las variables. Para calcular los coeficientes de correlación entre las variables de una tabla de datos se emplea el método "*corr"* como se muestra en la siguiente figura:

Figura 3-32. Coeficientes de correlación entre las variables de la tabla "*Salaries".*

```
1 salaries.corr()
```

	yrs_since_phd	yrs_service	salary
yrs_since_phd	1.000000	0.909649	0.419231
yrs_service	0.909649	1.000000	0.334745
salary	0.419231	0.334745	1.000000

Fuente: Elaboración propia con *Jupyter Notebook.*

3.6 Pruebas de hipótesis estadísticas

Según Walpole (1998) "Una hipótesis estadística es una aseveración o conjetura con respecto a una o más poblaciones". Es el uso de los datos contenidos en una muestra aleatoria con el fin de proporcionar evidencia que apoye o no una hipótesis. Dicha decisión se toma con base en una probabilidad.

Las hipótesis que se plantean son la hipótesis nula H_0 que es aquella que se desea probar y la hipótesis alternativa H_1 que se acepta cuando la hipótesis nula se rechaza.

Ahora, la hipótesis nula siempre específica el valor exacto de un parámetro mientras que la hipótesis alterna plantea que el parámetro tome valores mayores, menores o diferentes al del valor exacto. Como las decisiones se basan en probabilidades, existe el error. Se distingue el error de tipo I, denotado por α y cuyo valor usualmente se establece como 0.05, cuando se rechaza la hipótesis nula siendo esta verdadera y el error de tipo 2, que se denota con β, se da cuando se acepta la hipótesis nula siendo esta falsa.

La veracidad de una hipótesis sólo se puede conocer cuando se examina toda la población, lo cual puede ser tedioso o costoso.

Dependiendo de los parámetros conocidos de la población, la cantidad de datos de la muestra recogida y de la distribución de estos datos, se realizan diferentes tipos de pruebas de hipótesis para inferir sobre un parámetro desconocido de la población.

La estadística paramétrica se basa sobre la distribución normal, mientras que la estadística no paramétrica se basa en otras distribuciones. A continuación, se trabajan pruebas suponiendo normalidad en la distribución de los datos.

3.6.1 Pruebas de medias

Existen dos tipos de pruebas para probar medias que son la *prueba Z* y la *prueba t*. La prueba Z se utiliza cuando se conoce la desviación estándar de la población y la prueba t se emplea cuando no se conoce la desviación estándar poblacional, en cuyo caso se emplea la desviación estándar muestral para la prueba de hipótesis. *Python* dispone en el módulo "*scipy.stats"* de muchas funciones estadísticas y distribuciones de probabilidad para aplicar pruebas *t*. Suponga la siguiente lista de alturas en centímetros de los alumnos de una universidad:

Figura 3-33. Lista e importación del módulo estadístico "*scipy.stats".*

```
In [1]:
1  a=[165,178,178,166,181,171,168,
2     173,176,178,177,175,172,166]
3  import scipy.stats as ss
4  import numpy as np
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Se plantea que la altura media de los estudiantes de la universidad es diferente de 175cm. ¿Existe evidencia para tal afirmación con un nivel de significancia de 0,05? Dicha pregunta se puede responder realizando una prueba t de dos colas, dado que no se conoce la

desviación estándar de la altura de todos los estudiantes de la universidad y que la afirmación establece que la altura es mayor o menor que 175cm.

Así las cosas, se formula la hipótesis nula $H_0: \mu = 175cm$ contra la hipótesis alternativa $H_1: \mu \neq 175cm$. Empleando la función *ttest_1samp* de "*scipy.stats"* se realiza una prueba de dos colas.

En caso de presentarse una prueba a una cola, se debe tener en cuenta que el estadístico de prueba es el mismo, pero cambia la región de aceptación porque esta área comprendería todo el valor de α y no solo la mitad de dicho valor, el cual se emplearía en una prueba de dos colas.

Luego, utilizando la función mencionada, se obtienen dos valores: el estadístico de prueba y el *valor p*. Cuando el valor *p* es menor o igual que la significancia α, se rechaza H_0, de lo contrario no se puede rechazar H_0 (Canavos, 1998, p. 326).

La *función ttest_1samp* tiene dos argumentos principales que son la muestra y la media esperada. El planteamiento anterior se resuelve entonces como se muestra en la siguiente figura:

Figura 3-34. Ejemplo de prueba t de una sola media.

```
1 prueba_t=ss.ttest_1samp(a,175)
2 prueba_t
```

```
Ttest_1sampResult(statistic=-1.320998010086657, pvalue=0.20928322669463575)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

De la figura anterior se tiene que el valor *p* es mayor que la significancia, por lo tanto, no hay suficiente evidencia para rechazar que la media de altura de los alumnos de la universidad sea diferente de 175 cm.

Otra prueba de hipótesis muy empleada es la prueba de la diferencia de medias, en la cual se tienen dos muestras de distribuciones normales independientes con sus respectivas medias y desviaciones estándar; por lo que el objetivo consiste en determinar sí la diferencia entre las medias es equivalente a algún valor. Este tipo de pruebas, por ejemplo, puede ayudar a determinar si diferentes muestras provienen de una misma población.

La función "*ttest_ind"* del módulo "*scipy.*stats", realiza una prueba *t* bilateral para la diferencia de medias de dos muestras independientes y cuenta con un argumento lógico denominado "*equal_var",* con el cual se puede colocar la presunción de que las muestras tienen la misma varianza o no.

Esta función evalúa sí la diferencia de medias de las muestras es 0 o, dicho en otras palabras, si son iguales, con una significancia del 5%. Suponga que hay dos máquinas que llenan botellas con gaseosa. El analista quiere determinar si las maquinas llenan la misma cantidad de gaseosa.

Figura 3-35. Mililitros de gaseosa llenados por dos máquinas.

```
In [3]:  1 M1=[344,348,347,346,359,352,342,350,353,352]
         2 M2=[351,348,352,353,350,347,354,350,357,352]
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Mediante la función mencionada en el párrafo anterior se establecen las hipótesis $H_0: \mu_1 - \mu_2 = 0$ y $H_1: \mu_1 - \mu_2 \neq 0$. La función trabaja una prueba bilateral e igualmente devuelve los valores del estadístico de prueba y el valor *p* como se evidencia en la siguiente figura:

Figura 3-36. Prueba t de dos medias bilateral suponiendo varianzas iguales.

```
1  ss.ttest_ind(M1,M2)
```

```
Ttest_indResult(statistic=-1.1531037426034083, pvalue=0.2639510150912356)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Puesto que el valor *p* es mucho mayor que la significancia α no hay evidencia para rechazar la hipótesis nula luego se acepta H_0.

3.7 . Ejercicios de la unidad 3

1. Importe el conjunto de datos “kc_house_data.csv” disponible en Kaggle, el cual contiene los precios de venta y características de casas en el condado King, incluyendo Seattle y de casas vendidas en el periodo comprendido entre mayo de 2014 y mayo de 2015 (harlfoxem, 2016). Utilice la función *describe* para explorar los datos. ¿Existe algún valor atípico en los máximos o mínimos de las características de las casas?

2. Graficar el histograma de frecuencias con un número de intervalos equivalente al valor redondeado de la raíz cuadrada de la cantidad de observaciones del conjunto de datos “*kc_house_data.csv”*, para las siguientes características: precio, alcobas, baños, pies cuadrados de la vivienda, pies cuadrados de la vivienda, pies cuadrados del lote, año en el que la casa fue construida, pisos y grado. ¿La variable pies cuadrados de la vivienda “*sqft_living”* tiene una distribución normal de cola alargada?

3. Calcule la matriz de correlaciones entre las variables continuas de la tabla de datos “kc_house_data” y determine: ¿Cuál es la variable que está más correlacionada con el precio?

4. Grafique los diagramas de caja de la cantidad de pies cuadrados del lote para cada cantidad de pisos. Utilice una escala logarítmica en los pies cuadrados del lote para simplificar la visualización de los datos ¿Cambia la cantidad de pies cuadrados de la vivienda cuando esta tiene más pisos?

5. Utilice la función “*groupby”* para agrupar las viviendas por el año en el que fueron construidas y obtenga la mediana de los pies cuadrados de la vivienda para cada año. Grafique el año en el que fueron construidas las viviendas contra la mediana de cada grupo. ¿Existen tendencias al alza o a la baja de la cantidad de pies cuadrados de la vivienda con el transcurso de los años?

6. Retome la tabla de datos “*salaries”*. Suponiendo normalidad y varianzas desiguales, ¿Existe evidencia que rechace que la diferencia de medias entre los salarios de los profesores con rango asistente y asociado es igual a 0?, ¿cuál es el valor del estadístico t? ¿cuál es el valor p?

4. Capítulo 4. Introducción al aprendizaje automático

Según (Kim, 2017) siempre que se trabaje ciencia de datos se presenta el campo del "*machine learning"* (aprendizaje automático). El aprendizaje automático consiste en aplicar técnicas para descifrar modelos que hay detrás de los datos, teniendo en cuenta que los datos pueden ser estructurados o no estructurados. Los datos con los cuales se crean los modelos denominan "*training data"* (datos de entrenamiento), los cuales deben ser imparciales para que el modelo sea representativo. Los datos con los que se valida el modelo se denominan "*test data"* (datos de prueba), los cuales usualmente son datos tomados que no fueron empleados durante el entrenamiento del modelo.

En este capítulo se aprenderá a:

- Importar el módulo "*sklearn"* y otros sub módulos que contienen modelos y algoritmos de aprendizaje automático.
- Elaborar modelos de regresión básicos para predecir variables cuantitativas.
- Elaborar modelos de clasificación básicos para predecir variables cualitativas.
- Utilizar la validación cruzada de *"k"* iteraciones para medir la calidad de los modelos.
- Elaborar modelos básicos de agrupamiento para encontrar grupos en los datos.

Para trabajar aprendizaje automático en *Python* se trabajan dos módulos que son "*sklearn"* y "*statsmodels.api"*. Estos módulos contienen una inmensa variedad de herramientas y algoritmos utilizados en este tipo de inteligencia artificial.

Existen 2 tipos de aprendizaje automático principales, que son el aprendizaje supervisado y el aprendizaje no supervisado. Este capítulo trabaja algunos de los modelos de estos dos

tipos de aprendizaje automático. Las demostraciones y funciones en las que se basan son bastante complejas por lo que el objetivo del capítulo es simplemente mostrar de dónde proviene el modelo, su aplicación y la comprensión de los resultados para resolver problemas.

La gran mayoría de los modelos pertenecen al campo del aprendizaje supervisado. Primero se ejecutaran modelos de aprendizaje supervisado y al final se trabajan dos modelos de aprendizaje no supervisado. El aprendizaje supervisado se caracteriza porque existe una respuesta para cada variable predictora presente. Luego el modelo deseado debe relacionar estos dos valores con el fin de producir predicciones de futuras observaciones o para comprender la relación que hay entre dichos valores (James, Witten, Hastie, & Tibshirani, 2013).

En cuanto al aprendizaje no supervisado, no existe una respuesta a las variables presentes, por lo que el objetivo consiste en descubrir grupos en los datos. Dentro del campo del aprendizaje supervisado se trabajarán los modelos de regresión, clasificación y de árboles de decisión, mientras que en el campo del aprendizaje no supervisado se aplicará "*clustering"* (*agrupamiento*). Adicionalmente se trabajará un modelo de validación conocido como "*K-fold cross validation"*, que permite medir que tan efectivo es un modelo entrenado a la hora de predecir datos diferentes a los datos de entrenamiento.

Como se mencionó al inicio del capítulo, los modelos se entrenan a partir de la selección de un grupo de datos llamado datos de entrenamiento. Si se dispone de un solo conjunto de datos, se utilizaría una parte como datos de entrenamiento y los sobrantes se utilizarían como datos de prueba.

A la hora de ajustar un modelo a los datos, se presenta el conocido "*trade-off"* entre la "*bias"* y la varianza. La "*bias"* se refiere al error existente al aproximar un problema de la vida real (James, Witten, Hastie, & Tibshirani, 2013, p. 35).

Existen muchas variables que influyen en las observaciones y que no se asumen en los modelos.

La varianza se refiere a la variación de las predicciones de un modelo cuando se ajusta con diferentes datos de entrenamiento. Si las predicciones varían demasiado, se dice que el modelo tiene mucha varianza. Esta característica es común en los modelos que son demasiado ondulantes y cuando sucede, se dice que el modelo está sobreentrenado, además que dada la elevada varianza, puede comprobarse que los valores predichos varían mucho comparados con los datos de prueba.

Sin embargo, los modelos con alta varianza tienen poca "*bias",* ya que la complejidad de un problema de la vida real implica muchas relaciones. Un modelo con baja varianza, como por ejemplo un modelo de regresión lineal, donde los valores predichos provienen de una recta y no de una curva, no cambia considerablemente cuando se ajusta con diferentes datos de entrenamiento, por lo cual es más consistente a la hora de compararlo con los datos de prueba.

No obstante, la complejidad de la vida real escasamente puede representarse a través de la linealidad, por lo que un modelo de regresión lineal tiene una elevada "*bias"*. Este es el "*trade-off"* característico de los modelos de aprendizaje automático, en donde el objetivo consiste en encontrar un modelo que presente baja varianza y baja "*bias"*.

Existen otros métodos para encontrar modelos con dichas características, tales como la regularización, pero no se trabajaran en esta guía.

A continuación se presentan los modelos básicos de aprendizaje automático y la ejecución de los mismos mediante los módulos "*sklearn"* y "*statsmodels.api"* de *Python*. Los modelos inicialmente se trabajaran empleando todas las observaciones como datos de entrenamiento. En el tema de validación, se introduce el uso de datos de entrenamiento y de prueba.

4.1 Regresión lineal

El modelo de regresión es una herramienta principal empleada para predecir una variable cuantitativa o inferir sobre la relación entre una variable de respuesta y una variable predictora, implicando un supuesto de linealidad entre las variables.

La regresión es un análisis paramétrico, porque consiste en encontrar parámetros que explican la relación entre las variables predictoras y las variables de respuesta.

4.1.1 Regresión lineal simple

La regresión lineal simple se basa en los mínimos cuadrados. En este tipo de regresión se maneja solo una variable predictora X para predecir un valor Y asumiendo una relación lineal (James, Witten, Hastie, & Tibshirani, 2013).

Para tal efecto, se busca minimizar la suma de los cuadrados de las diferencias entre el valor de entrenamiento y_i y el valor pronosticado $\hat{y}$, buscando los coeficientes o parámetros $\hat{\beta}_0$ y $\hat{\beta}_1$ de una recta. La verdadera función Y de la que provienen todos los datos y a la cual se desea aproximar es desconocida, por esta y otras razones se establece un término de error ϵ. El modelo de regresión lineal simple se expresa entonces como:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x + \epsilon \quad (9)$$

Existen dos métodos para realizar una regresión lineal en *Python*, el primero de ellos consiste en recurrir al módulo "*sklearn"*. Esta librería contiene una gran cantidad de métodos de aprendizaje automático, construido a partir de las librerías "*Numpy"*, "*SciPy"* y "*matplotlib"* para el análisis de datos y la minería de datos (Pedregosa, et al., 2011).

De este módulo se importa el paquete "*linear_model"* y además se importarán otros módulos necesarios para graficar y adecuar los datos a las clases que ejecutan los modelos de regresión:

Figura 4-1. Importación de los módulos *sklearn*, *linear_model* y otros.

```
In [1]:
1 import pandas as pd
2 import numpy as np
3 import sklearn
4 from sklearn import linear_model
5 import matplotlib.pyplot as plt
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Para elaborar los modelos de regresión lineal se utilizará el conjunto de datos llamado "*kc_house_data.csv"* disponible en Kaggle, que contiene los precios de venta de casas en el condado King incluyendo Seattle y de casas vendidas en el periodo comprendido entre mayo de 2014 y mayo de 2015 (harlfoxem, 2016).

Luego, se importa de cualquier manera el conjunto de datos mencionado, con el nombre deseado, en este caso se le denominará "casas":

Figura 4-2. Importación del conjunto de datos "*kc_house_data."*

```
In [2]:
1 casas=pd.read_csv("C:\\Users\\Tesoro\\BPPD\\kc_house_data.csv")
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Ahora que se dispone de la tabla de datos "casas", se continúa creando una clase "*LinearRegression"*, que corresponde al modelo de regresión por mínimos cuadrados, a partir del submódulo "*linear_model"* a la cual se le dará el nombre de *"regr*". Esta clase incluye por defecto un intercepto.

Figura 4-3. Creación de una clase *"LinearRegression".*

```
In [3]: 1 regr=sklearn.linear_model.LinearRegression()
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Teniendo este objeto se puede aplicar el método de los mínimos cuadrados por medio de la función "*fit"* que contiene dicho objeto. Esta función no marcha si la variable predictora es un arreglo de una dimensión, ya que viene programada para utilizarse en modelos de regresión múltiple, por lo que solicita transformar el arreglo de una dimensión en un arreglo de dos dimensiones. Para realizar una regresión lineal simple se necesita entonces de un arreglo de una columna y "*n"* filas.

Esta última tarea se puede realizar utilizando el método "*reshape"* de los objetos "*numpy.ndarray"*, esto quiere decir que se debe transformar la serie proveniente de la tabla de datos en un arreglo "*numpy"* para poder emplear dicho método.

Suponga que se desea calcular el precio de venta de una casa usando como variable predictora la cantidad de pies cuadrados de la vivienda por medio de una regresión lineal simple. Entonces, la transformación de la serie pies cuadrados de la vivienda sería la que sigue:

Figura 4-4. Uso del método "*reshape"* en un arreglo "*numpy"* para modificar sus dimensiones.

```
In [4]: 1 X=np.array(casas.sqft_living)
        2 X=X.reshape(-1,1)
        3 Y=np.array(casas.price)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Teniendo las dos variables anteriores creadas se procede a utilizar el método "*fit*" para ajustar una línea recta a los datos:

Figura 4-5. Uso del método "*fit*" de la clase "*LinearRegression*" para aplicar el modelo de regresión lineal.

```
In [5]: 1 regr.fit(X,Y)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Una vez ejecutado, se pueden conocer los coeficientes $\hat{\beta}_0$ y $\hat{\beta}_1$ del modelo de regresión mediante los atributos de la clase *"regr"* creada, donde $\hat{\beta}_0$ es el intercepto de la recta con el eje de las ordenadas, $\hat{\beta}_1$ es su pendiente, definida como la cantidad que incrementa el valor de $\hat{y}$ por unidad de incremento de x.

Los coeficientes obtenidos, el valor del coeficiente de determinación R^2 y la predicción de algunos valores se muestran en la siguiente figura:

Figura 4-6. Resultados de un modelo de regresión lineal simple.

```
1 print("Coeficiente β1: ",regr.coef_)
2 print("Intercepto β0: ",regr.intercept_)
3 print("R^2: ",regr.score(X,Y))
4 print("Y_estimado([[6000],[8000]]): ",regr.predict([[6000],[8000]]))
```

```
Coeficiente β1:  [280.6235679]
Intercepto β0:  -43580.74309447373
R^2:  0.49285321790379316
Y_estimado([[6000],[8000]]):  [1640160.66429021 2201407.80008511]
```

Fuente: Elaboración propia con *Jupyter Notebook.*

El coeficiente de determinación R^2 explica el porcentaje de variación en la variable predicha que puede ser explicada a través de las variables predictoras de un modelo especifico (Statistics How To, 2020).

Gráficamente, se puede dibujar un "*scatter plot"* (gráfico de dispersión) con ayuda del módulo "*matplotlib.pyplot"* y la recta de regresión obtenida para visualizar el resultado. El gráfico de dispersión es otro tipo de gráfico que se presenta a continuación.

Para graficar la recta de regresión con "*matplotlib.pyplot"* fue necesario predecir los valores de las casas para cada valor de la superficie de las mismas, utilizando el método *"predict"* de la clase *"regr"* creada.

Figura 4-7. Predicción de valores con el modelo de regresión lineal simple.

```
In [7]:  1  Y_predicho=regr.predict(X)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Teniendo los valores predichos se puede crear el gráfico como sigue:

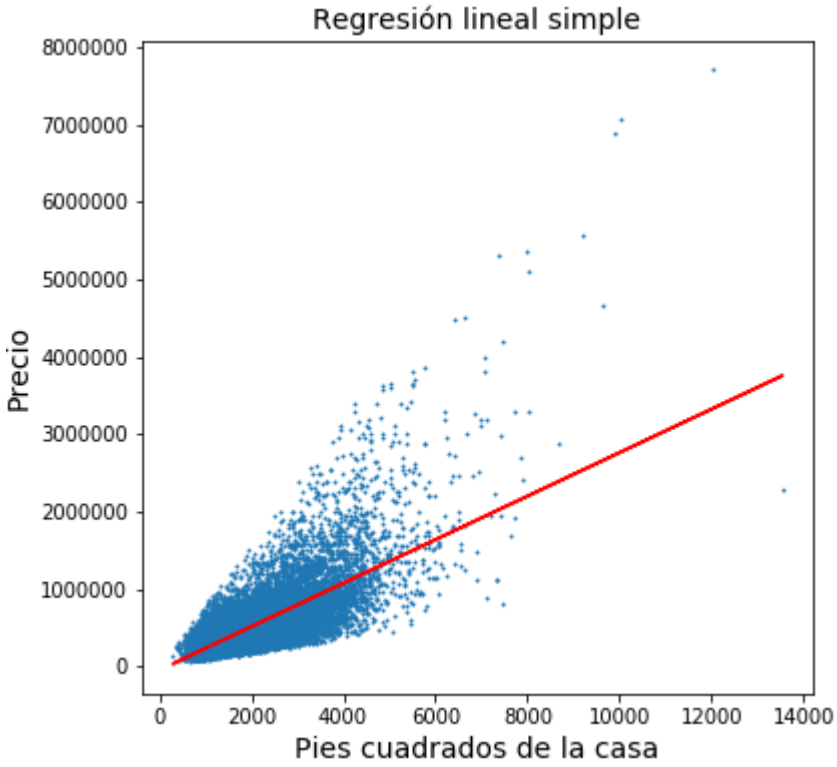
Figura 4-8. Código que genera un diagrama de dispersión junto con recta de regresión.

```
In [8]:  1  plt.figure(figsize=(6,6))
         2  plt.scatter(X,Y,s=1)
         3  plt.plot(X,Y_predicho, color="red")
         4  plt.title("Regresión lineal simple", fontsize=14)
         5  plt.xlabel("Pies cuadrados de la casa", fontsize=14)
         6  plt.ylabel("Precio", fontsize=14)
         7  plt.show()
```

Fuente: Elaboración propia con *Jupyter Notebook.*

El gráfico resultante con el código anterior es el siguiente:

Gráfico 4-1. Gráfico de regresión lineal simple.



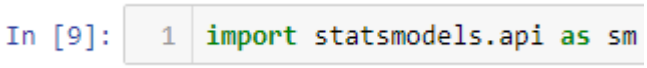
Fuente: Elaboración propia con *Jupyter Notebook.*

En el gráfico anterior se visualiza una gran dispersión de los datos y por ello es difícil encontrar una función que los represente. El modelo de regresión lineal simple no es suficientemente apropiado para este caso, lo cual se evidencia mediante el coeficiente de determinación R^2, una medida empleada para evaluar la calidad del modelo para replicar resultados que adquiere valores entre 0 y 1. Entre más cercano sea a 1, mejor se están replicando los datos.

Este coeficiente de determinación se calcula restando a 1 el cociente entre la suma de los residuos cuadrados y la suma total de los cuadrados. (James, Witten, Hastie, & Tibshirani, 2013, p. 70).

El otro método para realizar la regresión lineal es a través del módulo "*statsmodels.api"*, utilizando la clase "*OLS" (ordinary least squares*) para el modelo simple de mínimos cuadrados. Para tal efecto, se importa entonces el modulo mencionado:

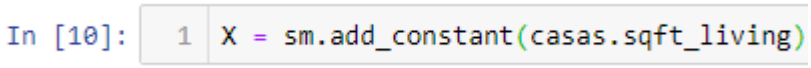
Figura 4-9. Importación del módulo "S*tatsmodels.api".*



Fuente: Elaboración propia con Jupyter Notebook.

Es necesaria una transformación en la variable predictora dado que no hay un intercepto incluido por defecto. Se añade una constante con la función "*add_constant"* del módulo "*statsmodels.api"*

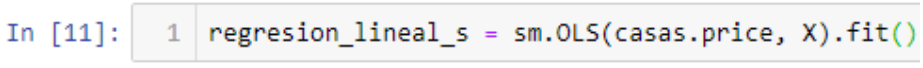
Figura 4-10. Agregación de una constante a una serie.



Fuente: Elaboración propia con *Jupyter Notebook.*

Luego se usa la función "*fit"* contenida en la clase *OLS* para obtener los cálculos de la regresión lineal. Dichos cálculos se almacenan en una clase. Se llamará a la clase que contiene los resultados como *"regresion_lineal_s":*

Figura 4-11. Uso de la clase *OLS* y su función "*fit"*



Fuente: Elaboración propia con *Jupyter Notebook.*

Finalmente con el método "*summary"* se despliegan los resultados. Se puede ver el coeficiente de determinación y otros estadísticos importantes como el estadístico *t* y el valor *p*. El valor *p* se basa en una hipótesis nula que establece que el coeficiente β_1 es igual a 0, sugiriendo que no existe una relación entre la variable predictora y la variable de respuesta y una hipótesis alternativa que establece que el valor β_1 es diferente de 0 indicando que si existe una relación entre dichas variables (James, Witten, Hastie, & Tibshirani, 2013).

Una parte de los resultados del modelo de regresión se muestran en la siguiente figura:

Figura 4-12. Resumen del modelo de regresión lineal simple obtenido con el módulo *OLS.*

```
In [12]: 1 regresion_lineal_s.summary()
```

Out[12]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.493
Model:	OLS	**Adj. R-squared:**	0.493
Method:	Least Squares	**F-statistic:**	2.100e+04
Date:	Wed, 28 Aug 2019	**Prob (F-statistic):**	0.00
Time:	20:07:21	**Log-Likelihood:**	-3.0027e+05
No. Observations:	21613	**AIC:**	6.005e+05
Df Residuals:	21611	**BIC:**	6.006e+05
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	-4.358e+04	4402.690	-9.899	0.000	-5.22e+04	-3.5e+04
sqft_living	280.6236	1.936	144.920	0.000	276.828	284.419

Fuente: Elaboración propia con *Jupyter Notebook.*

A partir de la tabla se evidencia que el valor p tiende a 0, por lo que existe evidencia suficiente para rechazar la hipótesis nula y se debe aceptar la hipótesis alternativa, es decir que si existe relación entre las variables del modelo propuesto.

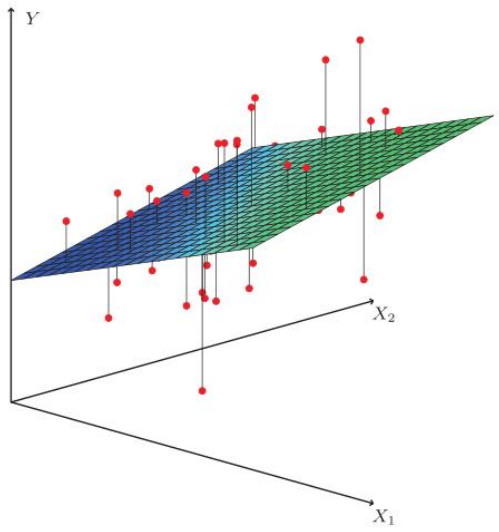
4.1.2 Regresión lineal múltiple

Es una extensión del modelo de regresión lineal simple. En la regresión lineal múltiple, se manejan dos o más variables predictoras e igualmente se asume una relación lineal entre estas y la variable predicha o de respuesta.

El modelo tiene la forma:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \hat{\beta}_3 x_3 + \cdots + \hat{\beta}_p x_p + \epsilon \quad (10)$$

Gráfico 4-2. Interpretación grafica de un modelo de regresión lineal múltiple con dos variables predictoras.



Fuente: (James, Witten, Hastie, & Tibshirani, 2013, p. 73)

Nuevamente se busca encontrar los coeficientes que minimizan la suma de los cuadrados de las diferencias entre el valor de entrenamiento y_i y el valor pronosticado $\hat{y}$.

Para elaborar una regresión lineal múltiple en *Python* se necesitará una tabla de datos que contenga las variables predictoras y una serie o arreglo que contenga las variables de respuesta. Suponga que se desea realizar una regresión lineal múltiple utilizando las variables predictoras “pies cuadrados de la vivienda”, “número de baños” y “número de alcobas” para predecir el precio de una casa:

Figura 4-13. Creación y ajuste de un modelo de regresión lineal múltiple mediante el módulo “*sklearn”.*

```
In [13]:
1 sub_casas=casas[["sqft_living",
2                  "bathrooms",
3                  "bedrooms"]]

In [14]:
1 regr_LM=sklearn.linear_model.LinearRegression()

In [15]:
1 regr_LM.fit(sub_casas,casas.price)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Los atributos de la clase obtenida muestran una mínima mejora en el coeficiente de determinación R^2, es decir que este modelo es apenas mejor que el modelo de regresión lineal simple trabajado anteriormente.

Figura 4-14. Coeficientes, intercepto y coeficiente de determinación del modelo planteado.

```
In [16]:
1 print(regr_LM.coef_)
2 print(regr_LM.intercept_)
3 print(regr_LM.score(sub_casas,casas.price))

[   309.39239013   7932.71222266 -57860.8943206 ]
74847.14080128318
0.5069197973159074
```

Fuente: Elaboración propia con *Jupyter Notebook.*

De manera similar utilizando el módulo "*statmodels.api"* se puede realizar una regresión lineal múltiple. Nuevamente es necesario añadir el intercepto mediante una constante:

Figura 4-15. Creación y ajuste de un modelo de regresión lineal múltiple mediante el módulo *"statsmodels.api"*

```
In [17]: 1 RLM=sm.add_constant(sub_casas)

In [18]: 1 resultado_RLM=sm.OLS(casas.price,RLM).fit()

In [19]: 1 resultado_RLM.summary()
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Los resultados del resumen del modelo anterior se muestran como sigue:

Figura 4-16. Resultados del modelo de regresión múltiple obtenido mediante modulo "*statsmodels.api"*.

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.507
Model:	OLS	**Adj. R-squared:**	0.507
Method:	Least Squares	**F-statistic:**	7405.
Date:	Thu, 29 Aug 2019	**Prob (F-statistic):**	0.00
Time:	16:37:00	**Log-Likelihood:**	-2.9996e+05
No. Observations:	21613	**AIC:**	5.999e+05
Df Residuals:	21609	**BIC:**	6.000e+05
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	7.485e+04	6913.667	10.826	0.000	6.13e+04	8.84e+04
sqft_living	309.3924	3.087	100.231	0.000	303.342	315.443
bathrooms	7932.7122	3510.556	2.260	0.024	1051.764	1.48e+04
bedrooms	-5.786e+04	2334.607	-24.784	0.000	-6.24e+04	-5.33e+04

Fuente: Elaboración propia con *Jupyter Notebook*.

De esta manera se pueden emplear diversas variables para predecir un valor. La regresión lineal múltiple debe ser manejada previniendo que entren variables que sean colineales, ya que puede resultar en incoherencias en los coeficientes asociados a las variables predictoras.

Otra variante del modelo de regresión es la regresión lineal múltiple con predictores cualitativos en los que se utilizan variables ficticias.

4.1.3 Regresión lineal múltiple con predictores cualitativos

Además de variables predictoras cuantitativas, es posible incorporar cualitativas a un modelo de regresión por medio de variables ficticias de la forma:

$$x_{i1} = \begin{cases} 1 & si\ el\ i\ésimo\ dato\ tiene\ la\ cualidad\ 1 \\ 0 & si\ el\ ésimo\ \ dato\ no\ tiene\ la\ cualidad\ 1 \end{cases} \quad (11)$$

La variable ficticia anterior entra al modelo:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_p x_p \quad (12)$$

Luego entonces, las variables artificiales son las que habilitan o no los valores de $\hat{\beta}_i$.

Suponiendo una variable categórica de tres niveles como por ejemplo el rango del profesor de la tabla *salaries*, el modelo resultante sería $\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2$ y las variables ficticias serían:

$$x_{i1} = \begin{cases} 1 & si\ el\ i\ésimo\ profesor\ es\ profesor\ asistente \\ 0 & si\ el\ i\ésimo\ profesor\ no\ es\ profesor\ asistente \end{cases}$$

$$x_{i2} = \begin{cases} 1 & si\ el\ i\ésimo\ profesor\ es\ profesor\ asociado \\ 0 & si\ el\ i\ésimo\ profesor\ no\ es\ profesor\ asociado \end{cases}$$

(13)

Para la suposición mostrada, un profesor solo puede tener un rango. Teniendo en cuenta esto, un profesor con rango profesor tendrá las dos variables artificiales creadas anteriormente con valor de 0 y su valor pronosticado será el valor de $\hat{\beta}_0$.

Si un profesor es asistente, el valor de x_{i1} toma el valor de 1 y habilita $\hat{\beta}_1$ por lo que el valor pronosticado será de $\hat{\beta}_0 + \hat{\beta}_1 x_1$. Viendo esta expresión se entiende que $\hat{\beta}_1$ es la cantidad pronosticada por encima o por debajo del pronóstico de un profesor con rango profesor.

Retomando la tabla de datos de las casas en venta en el condado King, suponga que se desea incorporar la variable categórica grado al modelo, la cual es una descripción de la calidad de las mejoras de la casa y que comprende valores entre 1 y 13.

Ahora, para crear el modelo de regresión lineal con predictores cualitativos, se necesita crear las variables artificiales por medio de una función del módulo "*pandas"* llamada "*get_dummies"*, que se aplica a la variable cualitativa:

Figura 4-17. Creación de variables ficticias para el grado de las casas de la tabla "casas".

```
In [20]:  1  grado_vart=pd.get_dummies(casas.grade)
          2  grado_vart.head()
```

Out[20]:

	1	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	0	0
2	0	0	0	0	1	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0	0	0

Fuente: Elaboración propia con *Jupyter Notebook.*

Para que el modelo de regresión lineal múltiple con predictores cualitativos sea el modelo descrito en la ecuación 12, se debe eliminar una columna del conjunto de datos de la Figura 4-17, ya que existe multicolinealidad.

La variable que se elimina entonces, estará representada por el valor que tome el intercepto. Suponga que se elimina la columna "1" y se añade una constante, con el fin de aplicar la regresión con el modulo "*statsmodels.api*." El conjunto de datos requerido es el siguiente:

Figura 4-18. Preparación de las variables predictoras para un modelo de regresión lineal múltiple con predictores cualitativos con "*statsmodels.api".*

```
grado_vart=grado_vart.drop(1,1)
grado_vart_sm=sm.add_constant(grado_vart)
grado_vart_sm.head()
```

	const	3	4	5	6	7	8	9	10	11	12	13
0	1.0	0	0	0	0	1	0	0	0	0	0	0
1	1.0	0	0	0	0	1	0	0	0	0	0	0
2	1.0	0	0	0	1	0	0	0	0	0	0	0
3	1.0	0	0	0	0	1	0	0	0	0	0	0
4	1.0	0	0	0	0	0	1	0	0	0	0	0

Fuente: Elaboración propia con *Jupyter Notebook.*

Una vez listas las variables predictoras, se procede a crear y ajustar el modelo, teniendo en cuenta que se busca predecir el precio de las casas a partir del grado.

Figura 4-19. Creación y ajuste de un modelo de regresión múltiple con predictores cualitativos.

```
1 regr_cualit=sm.OLS(casas.price,grado_vart_sm).fit()
2 regr_cualit.summary()
```

Fuente: Elaboración propia con *Jupyter Notebook.*

El resumen muestra un coeficiente R^2 de 0.520, el intercepto y los valores de los 11 coeficientes $\hat{\beta}_i$. Cada valor de $\hat{\beta}_i$ sumado con el intercepto es el valor pronosticado para una casa de acuerdo a su respectivo grado.

Figura 4-20. Predicción del precio de una casa cuyo grado es 12.

```
1 regr_cualit.predict([1,0,0,0,0,0,0,0,0,0,0,1])
array([3709615.38461538])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Desde el módulo "*sklearn"*, el modelo de regresión lineal múltiple con predictores cualitativos se puede llevar a cabo teniendo en cuenta que no es necesario añadir una constante, ya que la incluye por defecto. Sólo es necesario eliminar una columna de la tabla que contiene las variables artificiales.

Figura 4-21. Creación y ajuste de un modelo de regresión lineal múltiple con predictores cualitativos con "*sklearn".*

```
1 regr_cual_sklearn=linear_model.LinearRegression()
```

```
1 regr_cual_sklearn.fit(grado_vart,casas.price)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Los atributos de la clase contienen los coeficientes y el intercepto, como se muestra en la siguiente figura:

Figura 4-22. Visualización de los resultados de un modelo de regresión lineal múltiple con predictores cualitativos.

```
1 regr_cual_sklearn.coef_
```

```
array([  63666.66666719,   72381.0344833 ,  106523.97107492,
        159919.63739014,  260590.26233213,  400852.76565644,
        631513.18585142,  929771.07407462, 1354841.72681762,
       2049222.00000055, 3567615.38461593])
```

```
1 regr_cual_sklearn.intercept_
```

```
141999.99999945838
```

Fuente: Elaboración propia con *Jupyter Notebook.*

En la medida en que el coeficiente R^2 sea más grande, se tiene un modelo que representa mejor los datos. La meta entonces es crear un modelo cuyo coeficiente sea lo más cercano a 1, mediante la combinación de técnicas y análisis descriptivo para entender los datos y representarlos.

El siguiente tema consiste en emplear métodos para predecir variables cualitativas.

4.2 Clasificación

Diferente a los modelos de regresión, los modelos de clasificación son modelos de aprendizaje supervisado que se utilizan para predecir variables cualitativas a partir de las variables predictoras.

4.2.1 Regresión Logística

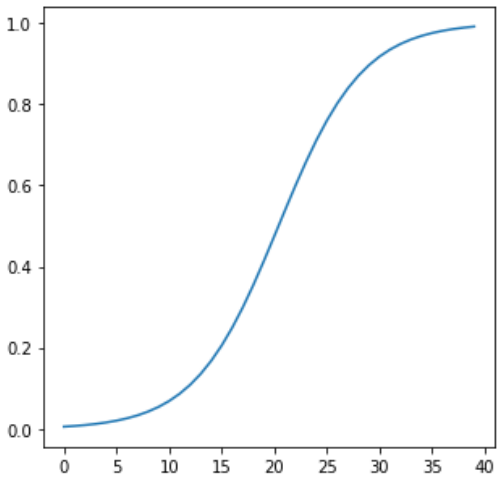
La regresión logística se utiliza para predecir una variable cualitativa que presente solo dos niveles. Por ejemplo: en la predicción del sexo de un individuo o una variable cuyas alternativas usualmente son “si” o “no”.

La regresión logística modela la probabilidad de que una observación pertenezca a determinada categoría o nivel (James, Witten, Hastie, & Tibshirani, 2013, p. 130).

Se construye a partir de la “*función logística”* cuyos valores de respuesta se encuentran entre 0 y 1 y que se define mediante la ecuación:

$$p(X) = \frac{e^{\beta_0+\beta_1 X}}{1+e^{\beta_0+\beta_1 X}} \qquad (14)$$

Gráfico 4-3. Forma típica de una función logística.



Fuente: Elaboración propia con *Jupyter Notebook.*

Según (James, Witten, Hastie, & Tibshirani, 2013), en este tipo de análisis se buscan los valores de los coeficientes β_i mediante la *estimación por máxima verosimilitud*, la cual maximiza la *función de verosimilitud* que ajustan el modelo:

$$\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X \qquad (15)$$

Para crear un modelo de regresión logística en *Python* se necesita que en los datos de entrenamiento, la variable de respuesta, se encuentre en forma binaria.

Retomando la tabla de datos "*salaries"* trabajada en el capítulo 3, se guardará esta tabla en una variable llamada "*salaries_log"* y se modifican los datos de la variable "sexo" de tal forma que 0 corresponda a mujer y 1 corresponda a hombre como se muestra a continuación:

Figura 4-23. Tabla *salaries* con la variable sexo en forma binaria.

	rank	discipline	yrs_since_phd	yrs_service	sex	salary
0	Prof	B	19	18	1	139750
1	Prof	B	20	16	1	173200
2	AsstProf	B	4	3	1	79750
3	Prof	B	45	39	1	115000
4	Prof	B	40	41	1	141500

Fuente: Elaboración propia con *Jupyter Notebook.*

Teniendo la tabla anterior en *Python,* suponga que se desea crear un modelo que determine la probabilidad de que un profesor sea hombre dado los años transcurridos desde su PhD.

Para el modelo de regresión logística se sugiere utilizar las funciones del módulo "*statsmodels.api"*, pero también se puede llevar a cabo con el módulo "*sklearn".* Para elaborar una regresión logística con *"statsmodels.api"* se añade el intercepto a una tabla de datos que contenga los años desde el PhD.

Figura 4-24. Creación de una tabla con una constante y los años desde el PhD.

```
1  años_desde_phd=sm.add_constant(salaries_log.yrs_since_phd)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Se utiliza la clase "*Logit"* correspondiente al modelo de regresión logística para crear otra clase a la que se le denominará *"regr_log"* y se ajusta mediante la función "*fit"* como sigue:

Figura 4-25. Creación y ajuste de un modelo de regresión logística.

```
In [3]:  1  regr_log=sm.Logit(salaries_log.sex,
         2                   años_desde_phd).fit()

Optimization terminated successfully.
         Current function value: 0.309387
         Iterations 7
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Los coeficientes del modelo se pueden encontrar en el método "*summary"* de la clase creada.

Figura 4-26. Resumen del modelo de regresión logística empleado.

```
In [4]: 1 regr_log.summary()
```

Out[4]:

Logit Regression Results

Dep. Variable:	sex	**No. Observations:**	397
Model:	Logit	**Df Residuals:**	395
Method:	MLE	**Df Model:**	1
Date:	Mon, 09 Sep 2019	**Pseudo R-squ.:**	0.03675
Time:	20:00:58	**Log-Likelihood:**	-122.83
converged:	True	**LL-Null:**	-127.51
		LLR p-value:	0.002203

	coef	std err	z	P>|z|	[0.025	0.975]
const	1.3683	0.307	4.457	0.000	0.767	1.970
yrs_since_phd	0.0433	0.015	2.898	0.004	0.014	0.073

Fuente: Elaboración propia con *Jupyter Notebook.*

La interpretación de los coeficientes no es tan sencilla como en los modelos de regresión lineal, es importante recordar el modelo en el que se basa mostrado en la ecuación 15.

Del mismo modo se puede medir la probabilidad de que un profesor sea de sexo masculino mediante el método "*predict"* del modelo creado, en donde se ve que la probabilidad de que el individuo sea de sexo masculino es generalmente muy alta, ya que la mayoría son hombres.

Figura 4-27. Calculo de la probabilidad de que un profesor sea de sexo masculino dado que lleva 14 años desde su PhD.

```
In [5]: 1 regr_log.predict([[1,14]])
Out[5]: array([0.87813066])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Mediante el módulo "*sklearn"*, la regresión logística se puede encontrar en la clase "*LogisticRegression"* incluida en el módulo "*linear_model"*. Esta clase dispone de varios argumentos para especificar el algoritmo de solución, regularización, cantidad máxima de iteraciones, entre otras.

El modelo típico de regresión logística emplea el algoritmo de optimización "*newton-cg"* y se sugiere minimizar la regularización a través del argumento "*C"*, teniendo en cuenta que este es un valor inverso de la tendencia a la regularización. De este modo se crea un modelo de regresión logística como el siguiente:

Figura 4-28. Creación de un modelo de regresión logística con el módulo "*linear_model* de "*sklearn".*

```
1 regr_log_sklearn=linear_model.LogisticRegression(solver='newton-cg',
2                                                  C=1e5)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Acto seguido, el modelo se ajusta a los datos con el método "*fit*". Para un modelo de una sola variable predictora, es necesario que dicha variable se encuentre en un arreglo de dos dimensiones, tal como se presentó en el modelo de regresión lineal simple. El ajuste del modelo con una variable predictora que cumple los requisitos dados es:

Figura 4-29. Modelo de regresión logística simple de *sklearn* ajustado.

```
1 regr_log_sklearn.fit(X,salaries_clas.sex)
```

```
LogisticRegression(C=100000.0, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
          solver='newton-cg', tol=0.0001, verbose=0, warm_start=False)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

En los atributos del modelo ajustado se encuentran los coeficientes del modelo y demás funciones como las de predicción o cálculo de probabilidades.

Figura 4-30. Coeficientes del modelo de regresión logística simple planteado desde el modulo "*sklearn".*

```
1 regr_log_sklearn.coef_
```

```
array([[0.04332537]])
```

```
1 regr_log_sklearn.intercept_
```

```
array([1.36829079])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Cabe anotar que el modelo de regresión logística también puede modelarse para varias variables predictoras, sean cualitativas o cuantitativas.

4.2.2 Regresión logística múltiple con variables cuantitativas y cualitativas

El modelo de regresión logística para múltiples variables predictoras proviene de la función:

$$p(X) = \frac{e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}}{1+e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}} \quad (16)$$

Del mismo modo emplea la estimación por máxima verosimilitud para calcular los coeficientes β_i del modelo:

$$\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p \quad (17)$$

Suponga ahora que se desea calcular la probabilidad de que un profesor sea de sexo masculino de acuerdo a las variables años desde el PhD, salario y disciplina. Empleando el modulo "*statsmodels.api"*, la variable rango debe transformarse en variables ficticias empleando la función "*pd.get_dummies"* y se debe eliminar una de sus columnas. Luego se debe juntar esta tabla de datos resultante con la tabla que contiene las demás variables predictoras en una sola tabla de datos y añadir una constante.

Figura 4-31. Preparación de datos para el modelo de regresión logística múltiple.

```
rango_vart=pd.get_dummies(salaries_clas["discipline"])
rango_vart=rango_vart.drop('A',1)
```

```
sub_salaries_log=salaries_clas[["yrs_since_phd","salary"]]
```

```
sub_salaries_log=pd.concat([sub_salaries_log,rango_vart],1)
```

```
sub_salaries_log=sm.add_constant(sub_salaries_log)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Teniendo listas las tablas de datos para entrenar el modelo, para crear un modelo con "*statsmodels"*, nuevamente se emplea la clase "*Logit"* para crear un modelo de regresión logística y se ajustan los datos extraídos anteriormente a través de su función "*fit"*:

Figura 4-32. Modelo de regresión logística múltiple.

```
regr_log_mult=sm.Logit(salaries_clas.sex,
                       sub_salaries_log).fit()
```

```
Optimization terminated successfully.
         Current function value: 0.305847
         Iterations 7
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Los resultados del modelo de regresión logística múltiple se visualizan con el método "*summary"* como se muestra a continuación:

Figura 4-33. Resultados del modelo de regresión logística múltiple.

```
1 regr_log_mult.summary()
```

Logit Regression Results

Dep. Variable:	sex	**No. Observations:**	397
Model:	Logit	**Df Residuals:**	393
Method:	MLE	**Df Model:**	3
Date:	Tue, 17 Sep 2019	**Pseudo R-squ.:**	0.04777
Time:	21:51:37	**Log-Likelihood:**	-121.42
converged:	True	**LL-Null:**	-127.51
		LLR p-value:	0.006780

	coef	std err	z	P>|z|	[0.025	0.975]
yrs_since_phd	0.0298	0.017	1.735	0.083	-0.004	0.063
salary	1.187e-05	8.04e-06	1.476	0.140	-3.89e-06	2.76e-05
A	0.3134	0.712	0.440	0.660	-1.082	1.709
B	0.3862	0.788	0.490	0.624	-1.158	1.931

Fuente: Elaboración propia con *Jupyter Notebook.*

Los resultados del modelo planteado, muestran valores "*p"* significativamente altos para las disciplinas de los profesores, lo que permite concluir que no existe relación entre la disciplina de un profesor y su sexo.

El valor "*p"* para la variable correspondiente al salario muestra valores "*p"* que sugieren que tampoco existe suficiente evidencia para rechazar que el coeficiente respectivo es 0.

De otro lado, la variable años desde el PhD, podría aceptarse con un rango de significancia ligeramente mayor, toda vez que se encuentra por encima del valor usual de α equivalente a 0.05. Estos valores demuestran la baja calidad del modelo planteado, pues el sexo de un profesor no depende de las variables predictoras empleadas.

Este modelo también se puede aplicar empleando la regresión logística de "*sklearn"*, teniendo en cuenta que no es necesario añadir el intercepto.

La regresión logística, como se mencionó al inicio, es un modelo que sólo puede predecir categorías binarias. Para predecir variables cualitativas con más de dos niveles existen otros modelos más apropiados como lo es el "*análisis de discriminante lineal"* que se verá a continuación.

4.2.3 Análisis de discriminante lineal y cuadrático para variables cualitativas

El "*discriminant analysis"* (análisis de discriminante) es una herramienta empleada para clasificación y reducción de dimensionalidad. Al respecto, el análisis de discriminante lineal para clasificación de una variable que tenga dos categorías, descrito por James, Witten, Hastie, & Tibshirani, (2013) es:

> […] los resultados del clasificador de análisis de discriminante lineal provienen de asumir que las observaciones dentro de cada clase provienen de una distribución normal con una vector clase específico y una varianza común σ^2, incorporando las estimaciones de estos parámetros en el *clasificador de Bayes*. […] (p. 142).

El teorema de Bayes establece que:

$$Pr(Y = y|X = x) = \frac{\pi_k f_k(x)}{\sum_{l=1}^{K} \pi_l f_l(x)} \quad (18)$$

En donde π_k es la probabilidad *a priori* de que una observación tomada aleatoriamente provenga de la $k$va categoría de la variable de respuesta Y. La función de densidad de X para una observación que provenga de la $k$va clase corresponde a $f_k(X) = \Pr(X = x|Y = k)$ (James, Witten, Hastie, & Tibshirani, 2013, p. 139).

El análisis de discriminante lineal estima $f_k(x)$ suponiendo que esta sea normal o gaussiana (James, Witten, Hastie, & Tibshirani, 2013, p. 139), de donde se obtiene que una asignación se realiza a la clase cuyo valor

$$\delta_k(x) = x\frac{\mu_k}{\sigma^2} - \frac{{\mu_k}^2}{2\sigma^2} + \ln(\pi_k) \quad (19)$$

Sea mayor, a través de la estimación de los parámetros

$$\hat{\mu}_k = \frac{1}{n_k}\sum_{i:y_i=k} x_i \quad (20)$$

$$\hat{\sigma}^2 = \frac{1}{n-K}\sum_{k=1}^{K}\sum_{i:y_i=k}(x_i - \hat{\mu}_k)^2 \quad (21)$$

Si la variable de respuesta tiene más de dos categorías, se asume que las observaciones provienen de una *distribución normal multivariante*, con un vector específico medio y una matriz de covarianzas común y esta función se incorpora al teorema de Bayes. (James, Witten, Hastie, & Tibshirani, 2013, p. 143).

De este modo, la asignación se hace a clase que presente el valor mayor.

$$\delta_k(x) = x^T\sum^{-1}\mu_k - \frac{1}{2}\mu_k^T\sum^{-1}\mu_k + \ln(\pi_k) \quad (22)$$

En donde $\sum$ es un parámetro de la función de densidad multivariante normal correspondiente a la matriz cuadrada de covarianzas de X (James, Witten, Hastie, & Tibshirani, 2013, p. 143)

El análisis de discriminante lineal es hecho por *Python* desde la clase "*LinearDiscriminantAnalysis"*, disponible a través del módulo "*sklearn.discriminant_analysis"*.

Similar a los modelos trabajados anteriormente, se usa el método "*fit*" para ajustar el modelo a los datos. Retomando la tabla "*salaries*", suponga ahora que se desea determinar el rango de un profesor según los años desde el PhD y el salario. El modelo se puede programar como se muestra a continuación:

Figura 4-34. Creación y ajuste de un modelo de análisis de discriminante lineal.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
X=salaries_clas[["yrs_since_phd", "salary"]]
Y=salaries_clas["rank"]
ADL=LinearDiscriminantAnalysis()
ADL.fit(X,Y)
```

```
LinearDiscriminantAnalysis(n_components=None, priors=None, shrinkage=None,
              solver='svd', store_covariance=False, tol=0.0001)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

La clase ajustada tiene varios atributos relacionadas con el modelo como las clases de las medias, probabilidades a priori, ajustes, coeficientes, entre otros parámetros.

Teniendo el modelo ajustado se pueden predecir respuestas con la función "*predict*". Una forma de evaluar el modelo es a través de la "*matriz de confusión*" en la que se constata los valores predichos por un modelo de clasificación.

La diagonal de la matriz, muestra la cantidad de predicciones que son correctas teniendo en cuenta las observaciones verdaderas. Para calcular la matriz de confusión se necesita importar un módulo que contiene numerosas herramientas para medir la calidad de la predicción de los modelos, denominado "*metrics*", incorporado en el módulo "*sklearn*".

La función obliga a predecir todos los valores con el modelo para contrastarlos con los valores verdaderos. La función que calcula la matriz de confusión es "*confusión_matrix*", la cual toma como argumentos al conjunto de variables de respuesta y al conjunto de

variables predichas. Otro argumento importante consiste en especificar el orden de las categorías para la matriz, por medio del argumento "*labels"*. El procedimiento anterior se muestra a continuación:

Figura 4-35. Predicción y cálculo de una matriz de confusión.

```
Y_predicho=ADL.predict(X)
from sklearn import metrics
sklearn.metrics.confusion_matrix(Y,Y_predicho,
     labels=["Prof", "AsstProf", "AssocProf"])
```

```
array([[255,   0,  11],
       [  0,  66,   1],
       [ 16,  15,  33]], dtype=int64)
```

Fuente: Elaboración propia con *Jupyter Notebook*.

Mediante la matriz de confusión se evidencia que el modelo funciona bastante bien a la hora de predecir el rango de los profesores. Otras variantes del modelo de discriminante es el análisis de discriminante cuadrático, en donde se asume que cada clase tiene su propia matriz de covarianzas (James, Witten, Hastie, & Tibshirani, 2013, p. 149).

Desde el módulo "*sklearn"* se puede importar la clase "*QuadraticDiscriminantAnalysis"* y aplicar de forma similar las funciones:

Figura 4-36. Modelo de análisis de discriminante cuadrático y matriz de confusión.

```
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
ADC=QuadraticDiscriminantAnalysis()
ADC.fit(X,Y)
Y_predicho=ADC.predict(X)
sklearn.metrics.confusion_matrix(Y,Y_predicho, labels=[
    "Prof", "AsstProf", "AssocProf"])
```

```
array([[251,   0,  15],
       [  0,  66,   1],
       [ 18,  10,  36]], dtype=int64)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

El modelo cuadrático ha pronosticado mejor el rango profesor asociado, pero a cambio pronosticó erróneamente más profesores asociados que en realidad son profesores con rango profesor.

Otro modelo de clasificación más sencillo es el de los *"k"* vecinos más próximos que se enuncia a continuación.

4.2.4 K vecinos más próximos

Según James, Witten, Hastie, & Tibshirani (2013) *"K nearest neighbors"* (k vecinos más próximos) es un modelo que:

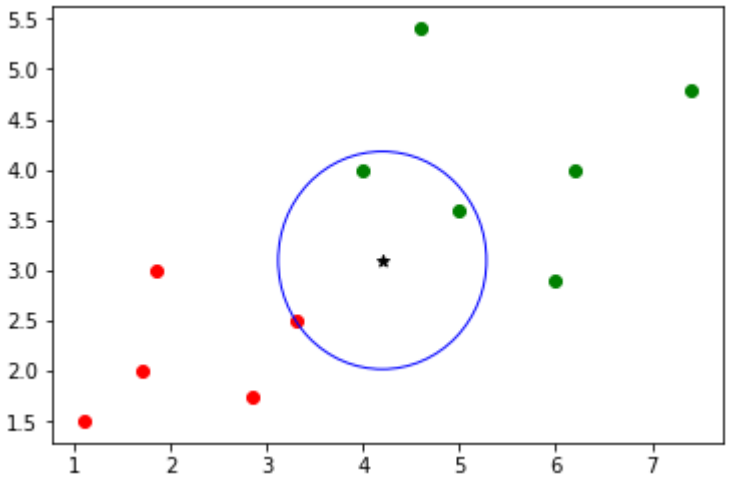
> " […] estima la distribución de probabilidad condicional de Y dado X y que clasifica a la observación a la clase cuya probabilidad estimada sea mayor. […] Dado un valor entero positivo K y una observación de prueba x_0, el clasificador *KNN* identifica primero los K puntos en los datos de entrenamiento más próximos a x_0, representados con N_0. Este entonces estima la probabilidad condicional para la clase j como la fracción de puntos en N_0 cuyos valores de respuesta equivalen a j. Finalmente, KNN aplica la regla de Bayes y clasifica la observación de prueba x_0 a la clase con mayor probabilidad " (p. 39).

$$\Pr(Y = j|X = x_0) = \frac{1}{K}\sum_{i\in N_0} I(y_i = j) \quad (23)$$

El siguiente gráfico muestra un ejemplo en el que se plasma la idea del clasificador de los k vecinos más próximos con un $K = 3$; el objetivo consiste en predecir sí la estrella negra pertenece al grupo de los puntos de color verde o a los puntos de color rojo y con $K = 3$, se buscan los tres puntos más cercanos a la estrella negra.

Se visualiza que dos de dichos puntos son verdes y uno de ellos es rojo, entonces la probabilidad de que el punto sea verde es de $\frac{2}{3}$ y de que sea rojo es de $\frac{1}{3}$. Finalmente, como el clasificador asigna la mayor probabilidad, la estrella negra se asigna a los puntos verdes.

Gráfico 4-4. Clasificación KNN con $K = 3$ para el punto marcado con una estrella.



Fuente: Elaboración propia a partir de (James, Witten, Hastie, & Tibshirani. 2013. p.40)

Retomando el ejemplo de los rangos de los profesores, se aplicará el modelo de los k vecinos más próximos con $K = 3$ para predecir el rango de un profesor según sus años transcurridos desde el PhD y su salario.

El modelo de los k vecinos más próximos para clasificación se encuentra como una clase dentro del módulo "*sklearn.neighbors"*, con el nombre "*KNeighborsClassifier"*.

En la siguiente figura se importa dicha clase desde el modulo aludido y se crea un modelo con el argumento "*n_neighbors"* con valor de 3 con el fin de especificar el número de vecindarios.

Figura 4-37. Importación de la clase *"KNeighborsClassifier"* y creación de un modelo con $K = 3$.

```
1 from sklearn.neighbors import KNeighborsClassifier
2 KVMP=KNeighborsClassifier( n_neighbors=3)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Seguido se usa el método "*fit*" para ajustar el modelo con los datos de entrenamiento.

Retomando las variables trabajadas para el modelo de análisis de discriminante lineal, la variable llamada *"X"* es la subtabla de datos que contiene los años desde el PhD y el salario, mientras que la variable llamada *"Y"* son los rangos de cada profesor. Para tal efecto, se ajusta entonces el modelo como sigue:

Figura 4-38. Ajuste de un modelo KNN con $K = 3$.

```
1 KVMP.fit(X,Y)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=3, p=2,
           weights='uniform')
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Con el modelo creado y ajustado, se puede predecir los valores de rango para los profesores según su los años transcurridos desde su PhD y su salario. Nuevamente se puede calcular la matriz de confusión. Se predicen entonces los valores para la variable *"X"* y se calcula la matriz mediante la función "*confusión_matrix"* del módulo "*metrics"*.

A pesar de que el modelo clasificador de los k vecinos más próximos es simple, provee buenos modelos de predicción cuando no se conoce la distribución condicional de Y dado X (James, Witten, Hastie, & Tibshirani, 2013).

Figura 4-39. Predicción y cálculo de matriz de confusión basado en el clasificador KNN con $K = 3$.

```
1  y_kvmp_predicho=KVMP.predict(X)
```

```
1  sklearn.metrics.confusion_matrix(Y,y_kvmp_predicho, labels=[
2      "Prof", "AsstProf", "AssocProf"])
```

```
array([[244,   9,  13],
       [  3,  55,   9],
       [ 20,   8,  36]], dtype=int64)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Existen grandes cantidades modelos de clasificación más complejos disponibles en el módulo "*sklearn"* y que se pueden aplicar para predecir variables cualitativas, como por ejemplo las máquinas de vectores de soporte. Este y otros modelos se pueden consultar a través del módulo *"sklearn"*.

4.3 Validación

Los modelos de regresión y clasificación trabajados han utilizado todos los datos disponibles como datos de entrenamiento, por lo que comparar los valores predichos con los valores de respuesta con los que se ajustó el modelo no tiene suficiente validez para evaluar un modelo.

Ahora, la evaluación del modelo debería realizarse con datos nuevos que no hayan sido utilizados para crear el modelo. Es decir, que si un modelo de regresión fue creado con las mediciones mensuales de determinada variable del año 2012, la evaluación del modelo debería realizarse contrastando el valor predicho por el modelo con las observaciones que se vayan presentado en el siguiente año.

No obstante, sí se manejaran por ejemplo mediciones anuales, no sería apropiado esperar un año para contrastar el nuevo valor con el valor predicho por el modelo. Para ello, en

lugar de utilizar todo un conjunto de datos como datos de entrenamiento, se utiliza una determinada cantidad de ellos para ajustar el modelo y se dejan los datos sobrantes para validar el modelo con el fin de evitar caer en el sobre-entrenamiento de un modelo.

Vale la pena aclarar que dicho modelo se caracteriza por predecir casi perfectamente los valores de respuesta con los que fue creado el modelo, pero con grandes errores a la hora de predecir nuevos valores, o como se definió al inicio del capítulo, tiene demasiada varianza.

Suponga que se desea evaluar el modelo de análisis de discriminante lineal trabajado en el que se predice el rango de un profesor según sus años desde el PhD y su salario, utilizando un 70% de los datos como datos de entrenamiento y un 30% como datos de validación. Para hacer este requerimiento primero se va a importar la función "*train_test_split*" presente en el *módulo "sklearn.model_selection"*, la cual permite dividir los datos en datos de entrenamiento y datos de prueba.

Figura 4-40. Importación de la función "*train_test_split".*

```
1 from sklearn.model_selection import train_test_split
```

Fuente: Elaboración propia con *Jupyter Notebook.*

La función importada devuelve las dos divisiones correspondientes a los datos de entrenamiento y de prueba respectivamente para cada entrada.

Mediante el argumento "*test_size"* se define el porcentaje de datos que serán usados como datos de prueba, y mediante el argumento "*random_state"* se crea una semilla, entendida como un número que permite generar números pseudoaleatorios que sirve para entremezclar los datos aleatoriamente. Luego se van a guardar las divisiones de datos de la variable "X" trabajada en el tema de análisis de "*discriminante lineal"* en las variables "x_entr" y "X_prueba" y los datos de la variable "Y" en las variables "y_entr" y "Y_prueba".

Figura 4-41. División de tablas en datos de entrenamiento y de prueba.

```
x_entr, X_prueba, y_entr, Y_prueba = train_test_split(
    X, Y, test_size=0.30, random_state=7)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Luego se ajusta el modelo a los datos de entrenamiento y se calcula por defecto la exactitud del modelo, desde el método *score* del modelo ajustado, teniendo en cuenta que sus argumentos serían los datos de validación "*X_prueba" y "Y_prueba".* La *exactitud* de un modelo de clasificación en "*scikit learn*" se define como:

$$exactitud(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1} 1(\hat{y}_i = y_i) \quad (24)$$

en donde $1(x)$ es la función indicatriz (Scikit-learn, 2019).

Figura 4-42. Exactitud para los datos de prueba.

```
ADL.fit(x_entr,y_entr)
ADL.score(X_prueba,Y_prueba)
```

```
0.85
```

Fuente: Elaboración propia con *Jupyter Notebook.*

De la forma anterior se obtiene una medición más correcta de la calidad del modelo. Sin embargo, se puede realizar diferentes selecciones de los datos de entrenamiento y de prueba. Por ejemplo: cambiar a un 90% de datos de entrenamiento y 10% de datos de prueba y la exactitud será diferente. Entonces, para obtener una medición más adecuada se ideó el método de remuestreo *"K-fold cross validation"* (validación cruzada de k

iteraciones) en el que se dividen los datos en K grupos y se prueba cada grupo usando como datos de entrenamiento los grupos sobrantes.

4.3.1 Validación cruzada de K iteraciones

Es una técnica que consiste en dividir los datos en K grupos del mismo tamaño, el primero de los grupos se utiliza como datos de prueba, los restantes como datos de entrenamiento y se mide la calidad un modelo determinado. Luego se utiliza el siguiente grupo como datos de prueba, mientras que con los grupos sobrantes se entrena el modelo y nuevamente se mide la calidad del modelo.

El proceso se repite hasta que se encuentren K mediciones de la calidad del modelo, con las cuales se calcula un promedio que se puede comparar con los resultados del mismo procedimiento pero aplicado con otro modelo diferente. Aquel modelo que tenga el mejor valor de calidad será el que mejor predice datos nuevos.

Ahora, en el módulo "*sklearn"* el paquete "*sklearn.model_selection"* contiene la función "*cross_val_score"*, que realiza el procedimiento de la validación cruzada de K iteraciones, utilizando como medida por defecto para predicciones cuantitativas el coeficiente R^2, o la exactitud en predicciones cualitativas.

Suponga entonces, que se desea aplicar esta técnica utilizando como modelo el análisis de discriminante lineal para la predicción del rango de un profesor según los años transcurridos desde el PhD y su salario. Primero se debe importar la función "*cross_val_score"*:

Figura 4-43. Importación de la función *"cross_val_score".*

```
1 from sklearn.model_selection import cross_val_score
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Esta función toma como primer argumento la clase correspondiente al modelo que se desea evaluar. Los siguientes dos argumentos son el conjunto de variables predictoras, la variable de respuesta y finalmente el argumento *"cv",* el cual permite especificar el número de grupos o una forma específica de división de los datos proveniente; por ejemplo: las funciones "*KFold"* o "*StratifiedKFold"* contenidas en el paquete "*model_selection"*, con los cuales se puede entremezclar los datos antes de aplicar la validación cruzada de k iteraciones.

Usualmente se maneja $K = 10$. Entre más grupos existan, mayor será el gasto computacional empleado. Sin embargo, el resultado será más preciso. La aplicación de la función "*cross_val_score"* con 10 grupos, aplicando el modelo de análisis de discriminante lineal se muestra en la siguiente figura:

Figura 4-44. Exactitudes obtenidas aplicando la técnica de la validación cruzada con 10 iteraciones.

```
1 cross_val_score(ADL,X,Y,cv=10)
```

```
array([0.90243902, 0.90243902, 0.92682927, 0.90243902, 0.9       ,
       0.875     , 0.82051282, 0.81578947, 0.89473684, 0.92105263])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

El promedio de las 10 exactitudes daría una medida para comparar el modelo con otros. Por ejemplo, suponga que se va a aplicar la técnica de validación cruzada con 10 iteraciones con el modelo de los k vecinos más próximos.

Ahora, sí se comparan las medias de las exactitudes de ambos modelos, el valor de la exactitud del modelo cuyo valor sea más cercano a 1 será el modelo más apropiado a la hora de decidir con cuál hacer predicciones. Se calcula entonces los promedios de las exactitudes para cada modelo:

Figura 4-45. Exactitudes con base en una validación cruzada de 10 iteraciones para dos modelos.

```
1 np.mean(cross_val_score(ADL,X,Y,cv=10))
0.8861238109344656
1 np.mean(cross_val_score(KVMP,X,Y,cv=10))
0.7324115401073039
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Estos resultados sugieren que el modelo de análisis de discriminante lineal es mejor prediciendo nuevos valores, comparado con el modelo de los k vecinos más próximos. Esta técnica es una de las muchas que existen para validar los modelos, y suele emplearse en otros modelos para mejorar la calidad de los mismos.

Además del coeficiente de determinación y la exactitud, existen otros indicadores que se emplean para medir la calidad de los modelos y se encuentran disponibles como argumento de la función "*cross_val_score*". Otros indicadores se encuentran en el módulo "*metrics*" de "*sklearn*".

4.3.2 Indicadores para regresión

Además del coeficiente de determinación R^2, también se encuentran el *error absoluto medio* y el *error cuadrático medio*, definidos por las siguientes ecuaciones:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \quad (25)$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \quad (26)$$

Cada indicador debe ser importado desde el módulo "*metrics"* como se muestra continuación:

Figura 4-46. Importación de las funciones de error absoluto medio y error cuadrático medio del módulo "*metrics*"

```
1 from sklearn.metrics import mean_absolute_error
2 from sklearn.metrics import mean_squared_error
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Estas funciones trabajan con dos argumentos que son el conjunto de las variables verdaderas y el conjunto de las variables predichas. Suponiendo el modelo de regresión lineal múltiple trabajado en el capítulo, se pueden aplicar dichas funciones como sigue:

Figura 4-47. Uso de las funciones de error absoluto medio y error cuadrático medio del módulo "*metrics*"

```
1 mean_absolute_error(regr_LM.predict(sub_casas),casas.price)
```

```
170279.69156846486
```

```
1 mean_squared_error(regr_LM.predict(sub_casas),casas.price)
```

```
66455447525.5654
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Por medio de estas funciones se verifica qué tan grande es el error promedio de las predicciones de los modelos de regresión. Del ejemplo anterior se evidencia la gran imprecisión en la predicción de los precios de las casas. Un modelo con valores pequeños de estos indicadores sería el más apropiado.

4.3.3 Indicadores para clasificación

Aparte de la exactitud y de las matrices de confusión, también se encuentran otros indicadores para medir la calidad de los modelos de clasificación. Algunos de ellos solo funcionan para clasificaciones binarias, mientras que otros funcionan para modelos con múltiples clases.

Para modelos de clasificación binaria como el modelo de regresión logística, existe una herramienta muy empleada para evaluar la calidad de este tipo de modelos, conocida como la *curva ROC*.

La curva ROC significa curva de característica operativa del recepto, la cual tiene en cuenta los conceptos de *sensibilidad* y *especificidad* , a medida que se cambia el límite para la probabilidad o discriminador, es decir el valor que permite decidir cómo clasificar una variable.

La sensibilidad o tasa verdadera positiva es el porcentaje de instancias positivas del total de instancias positivas reales, es decir que se expresa con la fórmula:

$$Sensibilidad = \frac{VP}{VP+FN} \tag{27}$$

Donde VP son valores predichos como positivos que realmente son positivos y FN son los valores predichos como negativos que son realmente positivos (Nighania, 2018).

La especificidad o tasa verdadera negativa es el porcentaje de instancias negativas del total de instancias negativas reales y se expresa con la fórmula:

$$Especificidad = \frac{VN}{VN+FP} \tag{28}$$

En donde TN son los valores predichos como negativos cuando realmente son negativos y FP son los valores predichos como positivos que realmente son negativos (Nighania, 2018).

La curva ROC se obtiene al graficar $1 - Especificidad$ contra la sensibilidad para todos los límites posibles del clasificador. Para graficar esta función en *Python*, se utiliza la función "*roc_curve*" presente en el módulo "*sklearn.metrics*", la cual tiene como argumentos la variable cualitativa binaria y la probabilidad de las observaciónes para cada clase del modelo, obtenida a través del método "*predict_proba*" de los modelos.

La función "*roc_curve*" devuelve tres arreglos que son la tasa de falsos positivos, equivalente a $1 - Especificidad$, la tasa verdadera positiva o sensibilidad y los limites del clasificador. Con los dos primeros arreglos es posible graficar la curva ROC.

Volviendo al modelo de regresión logística simple, en donde se busca predecir el sexo de un profesor a partir de los años transcurridos desde su PhD, suponga que se desea graficar la curva ROC para dicho modelo de regresión logística. La obtención de las tasas se muestra en la siguiente figura:

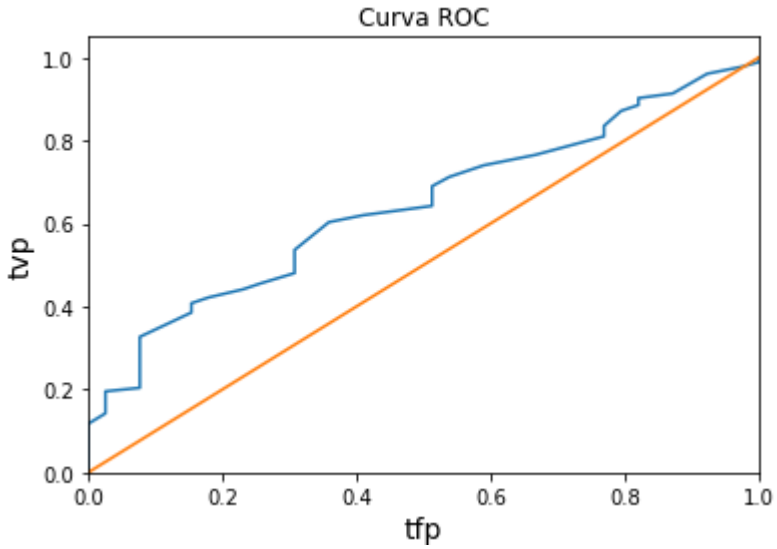
Figura 4-48. Obtención de las tasas de falsos positivos y verdaderos positivos a través de la función "*roc_curve*".

```
1 from sklearn.metrics import roc_curve
2 tfp,tvp,limites=roc_curve(salaries_clas.sex,
3                           regr_log_sklearn.predict_proba(X)[:,1])
```

Fuente: Elaboración propia con *Jupyter Notebook.*

A partir de los arreglos "*tfp*" y "*tvp*" obtenidos en la figura se puede graficar la curva ROC. La curva ROC para el modelo de regresión logística descrito es la siguiente:

Gráfico 4-5. Curva ROC para el modelo de regresión logística propuesto.



Fuente: Elaboración propia con *Jupyter Notebook.*

Cuanto más cerca se encuentre la curva del borde superior izquierdo, mejor es el clasificador. Si se comparan las curvas ROC de varios modelos se puede encontrar gráficamente el mejor. Esto es más fácil de deducir por medio del área bajo la curva de la curva ROC, denominada *AUC*, disponible a través de la función "*roc_auc_score"*, del módulo "*sklearn.metrics"*, que toma los mismos argumentos que la función "*roc_curve*":

Figura 4-49. Obtención de la AUC para el modelo de regresión logística propuesto.

```
1  sklearn.metrics.roc_auc_score(salaries_clas.sex,
2                                regr_log_sklearn.predict_proba(X)[:,1])
```

```
0.6424222890703338
```

Fuente: Elaboración propia con *Jupyter Notebook.*

El criterio de decisión es sencillo, entre más cercana a 1 sea el área bajo la curva, mejor es el modelo de clasificador en comparación.

Otras formas de evaluar modelos de clasificación se disponen a través de la función "*classification_report"* igualmente contenida en el módulo "*sklearn.metrics"*, que contiene tres principales métricas de clasificación: *precisión*, *exhaustividad* y "*f1-score"*.

La precisión mide la proporción de instancias positivas reales y se define como:

$$Precisión = \frac{VP}{VP+FP} \tag{29}$$

La exhaustividad mide la proporción de positivos reales correctamente clasificados.

$$Exhaustividad = \frac{VP}{VP+FN} \tag{30}$$

Finalmente, *"f1-score"* se interpreta como un promedio ponderado de la precisión y la exhaustividad, en donde el mejor valor es 1 y el peor es 0 (scikit-learn developers, 2019).

Suponga el modelo de análisis de discriminante lineal trabajado en clasificación. La aplicación de la función para obtener las medidas anteriores se muestra a continuación:

Figura 4-50. Obtención de indicadores para modelos de clasificación

```
1 from sklearn.metrics import classification_report
2 print(classification_report(salaries_clas["rank"],
3                            ADL.predict(X)))
4
```

	precision	recall	f1-score	support
AssocProf	0.73	0.52	0.61	64
AsstProf	0.81	0.99	0.89	67
Prof	0.94	0.96	0.95	266
avg / total	0.89	0.89	0.88	397

Fuente: Elaboración propia con *Jupyter Notebook.*

Esta y muchas otras funciones se pueden encontrar a través del módulo "*sklearn.metrics"*, con el fin de medir la calidad de los modelos creados.

4.4 Árboles de decisión

Los árboles de decisión son herramientas de aprendizaje supervisado aplicables a regresión y clasificación que consisten en segmentar el espacio de las variables predictoras en una cantidad específica de regiones y se forman a través de reglas de decisión inferidas a partir de los datos (James, Witten, Hastie, & Tibshirani, 2013).

Esta es una técnica que se puede interpretar sencillamente a través de diagramas de árbol, los cuales se estructuran mediante nodos, ramas y hojas. Un nodo corresponde a un punto donde se divide determinada variable, de acuerdo a un criterio especifico, cada división son ramas del árbol y las hojas son los resultados, que corresponderían a los valores predichos de teniendo en cuenta las condiciones dadas en cada nodo.

No obstante, los árboles de decisión no son tan efectivos como otros modelos de aprendizaje supervisado y suelen ser sobre-entrenados, sin embargo, son la base para otros modelos más precisos como el de los "*bosques aleatorios"*. Primero se trabajará un ejemplo para regresión y posteriormente otro para clasificación.

4.4.1 Arboles de decisión para regresión

Los árboles de regresión emplean la *división binaria recursiva*, con la cual se divide una variable predictora en solamente dos regiones. Las variables que se dividen se escogen de acuerdo a la que presente menor suma de los residuos cuadrados, es decir, según la fórmula:

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2 \qquad (31)$$

En donde J es la cantidad de regiones, aclarando que las regiones no se solapan, R_j son las regiones como tal y $\hat{y}_{R_j}$ es el valor promedio de respuesta de la variable de entrenamiento en la región j. (James, Witten, Hastie, & Tibshirani, 2013).

Sin embargo, el modelo incluido en "*sklearn"* utiliza el error cuadrático medio, definido por la ecuación 25.

Las divisiones podrían realizarse hasta que haya muy pocas observaciones en las hojas, corriendo el riesgo de caer en el sobre entrenamiento del modelo. De este modo, la reducción de regiones evitaría demasiado sobre entrenamiento, por lo que limitar las divisiones resulta necesario.

En el módulo *"sklearn.tree"*, se encuentran las clases que elaboran los modelos de árboles de decisión para regresión y clasificación. La clase correspondiente al modelo de regresión es "*DecisionTreeRegressor"*, la cual solo acepta como argumentos variables numéricas y que debe ser importada como sigue:

Figura 4-51. Importación de la clase "*DecisionTreeRegressor".*

```
1 from sklearn.tree import DecisionTreeRegressor
```

Fuente: Elaboración propia con *Jupyter Notebook.*

En caso de que se desee crear un modelo con variables predictoras cualitativas se deben utilizar variables ficticias binarias. De esta clase es fundamental conocer los argumentos que permiten acortar el árbol de decisión, lo que comúnmente se conoce como "podar", con el fin de evitar el sobre entrenamiento y facilitar la comprensión del mismo.

Por ejemplo, el argumento "*max_depth"* limita la profundidad o los niveles del árbol. Por ejemplo: Una clase creada con dicho argumento con valor de 2, tendrá 4 hojas, si se crea con valor de 3, se tendrán 8 hojas y así sucesivamente.

Ahora, si no se especifica este valor, por defecto, el árbol se expande hasta que las hojas sean puras, es decir que las observaciones en dicha hoja tienen el mismo valor, luego no tiene sentido dividir aún más las observaciones.

De este modo, el argumento "*min_samples_split"* permite especificar el número de observaciones necesarias para dividir un nodo. El argumento "*max_leaf_nodes"* limita la

cantidad de hojas del árbol. Con estos y otros argumentos que dispone la clase, se limita el crecimiento del árbol y se obtiene un árbol más fácil de interpretar.

Suponga entonces que se desea calcular un árbol de regresión con un máximo de dos niveles para el peso en gramos de los corazones de gatos domésticos a partir del peso total en kilogramos, con base en el conjunto de datos "*cats"*, contenido en los conjuntos de datos de "*R"*, dentro el paquete "*MASS"* y que contiene dichas medidas anatómicas de gatos domésticos usados en experimentos con medicamentos. Se debe crear entonces una clase que tenga como argumento "*max_depth"* con valor de 2:

Figura 4-52. Creación de una clase "*DecisionTreeRegressor"* con dos niveles.

```
1 arbol_regr=DecisionTreeRegressor(max_depth=2)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Finalmente se ajusta el modelo a los datos. Es necesario transformar los datos de la variable predictora en un arreglo de dos dimensiones, igual a como se trabajó en el modelo de regresión lineal simple:

Figura 4-53. Ajuste de un modelo de árbol de regresión.

```
1 arbol_regr.fit(peso_cuerpo,cats.Hwt)
```

```
DecisionTreeRegressor(criterion='mse', max_depth=2, max_features=None,
           max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           presort=False, random_state=None, splitter='best')
```

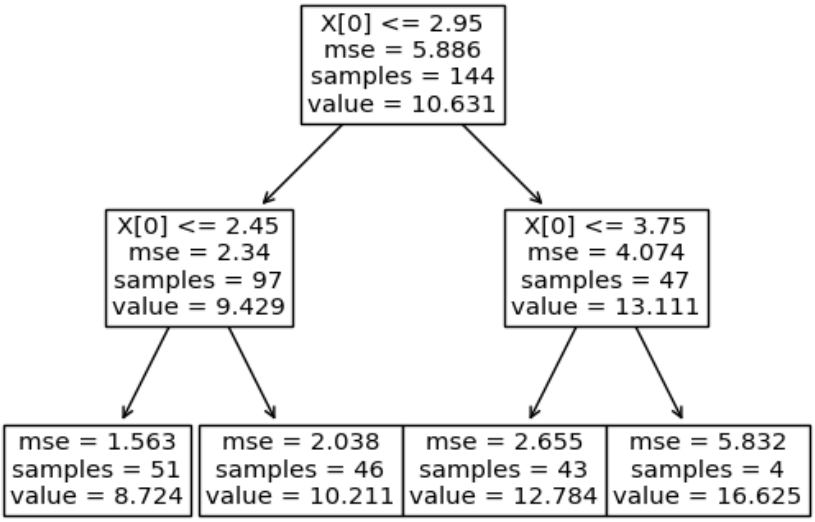
Fuente: Elaboración propia con *Jupyter Notebook.*

Muchos argumentos han quedado por defecto. La explicación de todos los argumentos del modelo se puede encontrar en la documentación de "*scikit-learn"*.

Los árboles de decisión pueden graficarse utilizando la función "*plot_tree"* del módulo "*tree"* colocando como argumento el modelo ajustado. Otra forma es utilizando funciones del módulo "*graphviz"*.

El árbol de decisión para el ejemplo planteado es el siguiente:

Figura 4-54. Ejemplo de árbol de regresión con dos niveles.



Fuente: Elaboración propia con *Jupyter Notebook.*

Las reglas de decisión se encuentran como un método de la clase creada, denominado "*tree_.threshold"* y los valores pronosticados para cada región en el método "*tree_.value"*.

En la Figura anterior, $X[0]$ corresponde a la primera variable del modelo, en este caso única. La división arroja dos ramas, la izquierda es cuando la condición se cumple, mientras que la rama derecha es cuando no se cumple. El valor es la cantidad promedio de las observaciones que pertenecen a la región delimitada por la condición. Así, para un gato que presente un peso de 3.2Kg, se calcula que su corazón pesará 12.784g.

Los árboles de decisión se emplean también para clasificación, pero en estos se emplean otros criterios para realizar la división, ya que no se puede aplicar el error cuadrático medio en variables cualitativas.

4.4.2 Árboles de decisión para clasificación

Los árboles de clasificación emplean principalmente como criterio de separación el "*índice de Gini"*, el cual es una medida de impureza. Es decir, que puede explicar qué tan buena es la división de una variable bajo cierto parámetro. Entre más pequeño sea este índice, mejor se están separando los datos, por lo que siempre se escoge la variable cuyo índice de Gini sea el menor y también se evalúa si a medida que se extiende el árbol, los índices de Gini disminuyen, ya que de lo contrario, se prefiere dejar los nodos como hojas. (Starmer, 2018)

Según (Bhavesh, 2018), el índice de Gini para un conjunto de datos D que contiene n clases se calcula como:

$$Gini(D) = 1 - \sum_{j=1}^{n} p_j^2 \quad (32)$$

Donde p_j es la frecuencia relativa de la clase j en D. Si un conjunto de datos D se divide en dos subconjuntos de datos D_1, D_2, el índice de Gini se define entonces como:

$$Gini_A(D) = \frac{|D_1|}{|D|} Gini(D_1) + \frac{|D_2|}{|D|} Gini(D_2) \quad (33)$$

Esto debido a que hay variables con distinto número de observaciones y por ende es necesaria una ponderación. Retome el conjunto de datos "*salaries"* trabajado con el modelo de clasificación de análisis de discriminante lineal. Se desea igualmente predecir el rango de un profesor dados los años desde el PhD y el salario, estos contenidos en las variables "X" y "Y" declaradas anteriormente. El árbol de clasificación en "*sklearn.tree"* se encuentra como la clase "*DecisionTreeClassifier"* que debe ser importada.

Figura 4-55. Importación de la clase "*DecisionTreeClassifier"*.

```
1 from sklearn.tree import DecisionTreeClassifier
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Luego se procede a crear el modelo con la clase "*DecisionTreeClassifier"*. Esta clase tiene numerosos argumentos. El principal es el "*criterion"* correspondiente al criterio de división, que por defecto utiliza el índice de Gini. También existe otro criterio similar usado, conocido como entropía, el cual se especifica cómo "*entropy".* Igualmente comprende los argumentos trabajados en la clase "*DecisionTreeRegressor"*, con el fin de podar el árbol de decisión, tales como *"max_depth", "min_samples_leaf", "max_leaf_nodes".*

Otro argumento relevante es *"min_impurity_decrease"*, el cual permite dividir nodos si estos presentan un índice de Gini menor comparado con los nodos anteriores, ya que de lo contrario se deja el nodo como hoja. Suponga que se especifica un máximo de 2 niveles para el árbol de clasificación.

Figura 4-56. Creación de una clase *"DecisionTreeClassifier"* con 2 niveles

```
1 arbol_clas=DecisionTreeClassifier(max_depth=2)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

El modelo por defecto empleará como criterio el índice de Gini y finalmente se ajusta el modelo mediante la función *"fit"*.

Figura 4-57. Ajuste de un modelo de árbol de clasificación con máximo 2 niveles

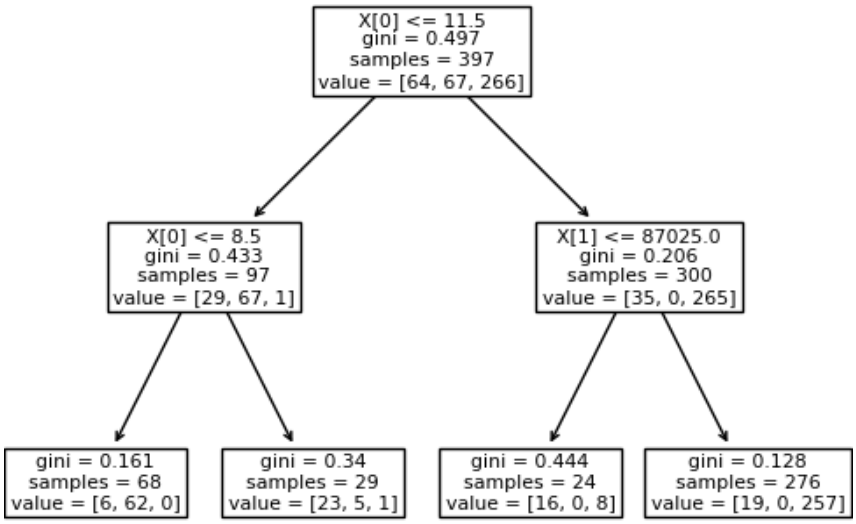
```
1 arbol_clas.fit(X,Y)
```

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=2,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

Fuente: Elaboración propia con *Jupyter Notebook.*

El árbol resultante, graficado mediante la función "*plot_tree"* del módulo "*tree"* se muestra a continuación:

Figura 4-58. Ejemplo de árbol de clasificación con dos niveles



Fuente: Elaboración propia con *Jupyter Notebook.*

Del árbol de decisión se tiene que ha empleado las variables de la sub tabla empleada para entrenar el modelo. En dicha sub tabla se tiene que la primera columna son los años desde el PhD y la segunda el salario, correspondientes a $X[0]$, $X[1]$ en el árbol de decisión.

Ahora, el valor que separa las regiones es aquel que produce el menor índice de Gini ponderado. El índice de Gini para cada nodo se muestra en cada forma y los valores son las cantidades de observaciones que se encuentran de acuerdo a los parámetros que dividen el árbol, siguiendo el orden dado en el atributo "*classes_"* del modelo. Del mismo modo, otros resultados del modelo se encuentran como atributos.

El atributo "*tree_.value"* muestra en arreglos la cantidad de observaciones por cada categoría en cada nodo, el atributo *"tree_.threshold*" muestra los parámetros que hacen la separación binaria; el atributo *"tree_.impurity"* muestra un arreglo con las impurezas en cada nodo según el criterio escogido, entre otros atributos. Con el método "*predict"* se puede predecir valores. La predicción es entonces la categoría que presente más observaciones en la región dada.

4.5 Agrupamiento

El "*clustering"* (agrupamiento) es un método de aprendizaje no supervisado que se caracteriza porque no se conocen variables de salida para las variables predictoras.

El objetivo del agrupamiento, según James, Witten, Hastie, & Tibshirani (2013) es "[…] determinar sobre la base de $x_1, \dots, x_n$, si las observaciones caen en grupos relativamente distintos " (p. 27) .

El agrupamiento consiste pues en separar en grupos, a las observaciones que seas similares entre sí, de modo que las observaciones de un grupo, son diferentes a las observaciones de los demás grupos (James, Witten, Hastie, & Tibshirani, 2013). Este método es aplicado en segmentación de mercados para identificar grupos de clientes y posibles compradores de un producto. Dos métodos elementales de agrupamiento son el agrupamiento K medias y el agrupamiento jerárquico.

4.5.1 Agrupamiento K-medias

"K-means clustering" (Agrupamiento K-medias) es un algoritmo que asigna cada observación a "K" cantidad de clústeres definidos que no se solapan, buscando un óptimo

local al reducir una variación interna del clúster, medida usualmente trabajada como el cuadrado de la distancia euclidiana.

El algoritmo inicia colocando aleatoriamente “K” puntos en el espacio, cada uno correspondiente a un clúster; luego asigna cada observación al clúster más cercano según el cuadrado de la distancia euclidiana y seguido, calcula el centroide para cada clúster formado.

Si al realizar nuevamente la asignación de las observaciones teniendo en cuenta los centroides calculados, se encuentra que alguna observación se asigna a un clúster diferente, se calculan de nuevo los centroides y se realizan de nuevo asignaciones. Es importante tener en cuenta que el algoritmo se detiene hasta que no haya cambios en la asignación de las observaciones.

Este algoritmo debe realizarse varias veces, ya que depende de la cantidad y la ubicación inicial de los clústeres, con el fin de comparar varias soluciones óptimas locales y escoger aquella que presente la menor suma de los cuadrados de las distancias euclidianas.

La distancia euclidiana para un espacio *n* dimensional para los puntos $p = (p_1, p_2, \dots, p_n)$, $q = (q_1, q_2, \dots, q_n)$ se define como:

$$d(P, Q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \quad (34)$$

Este algoritmo se encuentra disponible en el módulo “*sklearn.cluster”*, que contiene la clase “*KMeans cluster”*. Primero se importa el módulo mencionado:

Figura 4-59. Importación del módulo *“sklearn.cluster”*.

```
1 import sklearn.cluster
```

Fuente: Elaboración propia con *Jupyter Notebook*.

Se debe crear una clase "*KMeans"* especificando con el argumento "*n_clusters"* la cantidad de clusters que se formarán. El argumento "*init"* permite escoger entre tres formas de colocar los centros de los clústeres, 'k-means++', 'random' o un arreglo. 'k-means++' es el método por defecto que provee mejores puntos iniciales minimizando el promedio de la distancia cuadrada entre las observaciones del mismo clúster (Arthur & Vassilvitskii, 2006), mientras que la opción *"random"* escoge observaciones de forma aleatoria como los centros de los clústeres.

Suponga un conjunto de datos conformado por dos variables a=(1,1.5,2.1,2.5,3.3,3.6,3.8,4.1,5,5.4,5.9,7.2), b=(5,6,4.2,5.4,4.6,5.6,3.6,2,3,2.1,4,2.3), para los que se desea agrupar dos clústeres. La creación de la clase para el agrupamiento de "K" medias se muestra como sigue:

Figura 4-60. Creación de una clase "*KMeans"* especificando dos *clústeres* y una semilla especifica.

```
KM_cluster=sklearn.cluster.KMeans(n_clusters=2,
                                  random_state=0)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Luego se utiliza el método "*fit"* para ajustar una tabla de datos creada a partir de los puntos *a* y *b* propuestos anteriormente. Puesto que se trata de un modelo de aprendizaje no supervisado, solo existe como variable de entrada los datos de la tabla de datos propuesta, no se coloca ninguna variable de respuesta.

Figura 4-61. Ajuste del modelo de agrupamiento de K medias a la tabla de datos propuesta.

```
1 KM_cluster.fit(tabla_cluster)
```

```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=2, n_init=10, n_jobs=1, precompute_distances='auto',
    random_state=0, tol=0.0001, verbose=0)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Los resultados del modelo se encuentran ahora en los atributos de la clase creada. Los más importantes son: "*cluster_centers_"*, el cual devuelve un arreglo con los centroides de los clústeres resultantes; "*labels_*" devuelve etiquetados en números, a qué clúster pertenece cada observación, "*inertia_"* devuelve la inercia, definida como la suma de las distancias cuadradas de todas las observaciones a los centroides del clúster al que pertenezcan.

Figura 4-62. Atributos principales de la clase "*KMeans"* para un modelo ajustado.

```
1 KM_cluster.cluster_centers_
```

```
array([[5.23333333, 2.83333333],
       [2.33333333, 5.13333333]])
```

```
1 KM_cluster.labels_
```

```
array([1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0])
```

```
1 KM_cluster.inertia_
```

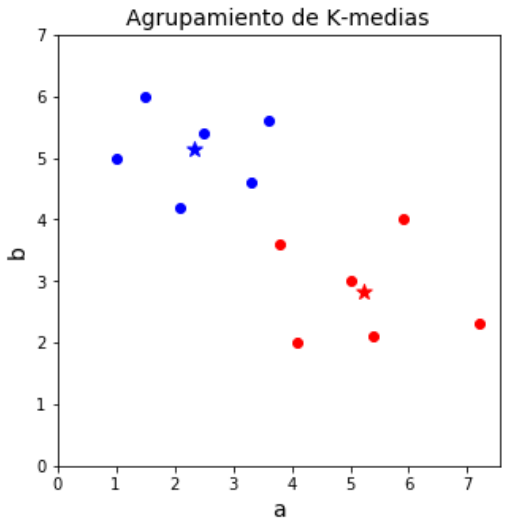
```
18.53333333333334
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Este modelo de agrupamiento tiene los métodos para predecir y para calcular métricas para agrupamientos,los cuales son "*predict"* y "*score"* respectivamente. Organizando los

resultados del agrupamiento se puede graficar para visualizar los clústeres más fácilmente como se muestra en la siguiente figura:

Gráfico 4-6. Visualización de clústeres obtenidos mediante el agrupamiento de K medias.



Fuente: Elaboración propia con *Jupyter Notebook.*

Los centroides de los clústeres se visualizan con una estrella y mediante los colores se distinguen ambos grupos. Este algoritmo requiere que se especifique la cantidad de clústeres que hay, por lo que un método conocido para encontrar una cantidad apropiada de clústeres para un conjunto de observaciones consiste en comparar la variación a medida que se incrementa la cantidad de clústeres. Cuando la cantidad de variación no se reduzca significativamente, se escoge la cantidad de clústeres trabajada en la iteración anterior.

Esta variación se puede manejar a través de la inercia encontrada anteriormente, por ejemplo, la inercia en el ejemplo anterior, cuando se trabaja con K=1 es de 62.633, con K=2, la inercia es de 18.533, mientras que con K=3 es de 13.031. Dado que la reducción de la inercia cuando K=3 no se redujo significativamente, se opta por utilizar K=2.

Según (Scikit-learn , 2019) este método de agrupamiento presenta ciertas desventajas, como por ejemplo, que a través de la inercia se asume que los clústeres son convexos e

isotrópicos, como por ejemplo clústeres alongados o con formas irregulares. Otro método básico de agrupamiento se presenta en el siguiente tema.

4.5.2 Agrupamiento aglomerativo

El agrupamiento aglomerativo es un tipo de agrupamiento jerárquico que consiste en un algoritmo basado en una afinidad y una conexión. Esta última conocida como "*linkage"*.

El algoritmo primero define a cada observación como un clúster y luego mediante una medida de afinidad, comúnmente la distancia euclidiana, se encuentran los dos clústeres más similares.

Teniendo en cuenta la distancia euclidiana como medida de afinidad, el par de observaciones que presente la menor distancia euclidiana serán agrupadas formando un nuevo clúster. Luego, dado que existen ahora clústeres con más de una observación, se establece una medida de conexión o "*linkage"* con la cual se comparan los clústeres que posean más de una observación.

Dentro de las medidas de conexión se trabajan usualmente cuatro que son: Conexión completa, conexión simple, conexión promedio y conexión centroide (James, Witten, Hastie, & Tibshirani, 2013, p. 394).

- La conexión completa o *complete "linkage"* toma como valor de afinidad el máximo valor de afinidad que haya entre todas las observaciones de un clúster con otro.

- La conexión simple, contraria a la conexión completa, utiliza el mínimo valor de afinidad que haya entre todas las observaciones de un clúster con otro.

- La conexión promedio emplea el valor promedio de afinidad que hay entre todas las observaciones de un clúster con otro.

- La conexión centroide utiliza como valor de afinidad, la afinidad entre los centroides de los clústeres en comparación.

Una vez se tengan todas las afinidades, se agrupan en un clúster aquellos clústeres que presenten mejor afinidad, en este caso la menor distancia euclidiana y se continua iterando hasta que exista un solo clúster.

El agrupamiento aglomerativo se caracteriza porque se puede representar por un diagrama conocido como dendrograma, el cual es una representación en un árbol de las uniones de los clústeres. Este diagrama se construye de abajo hacia arriba, se inicia colocando como hojas a todas las observaciones, tratadas como clústeres. Luego se unen en un nodo las observaciones con mayor afinidad a la altura definida por la medida de afinidad, sea la distancia euclidiana u otras, hasta que se complete el árbol.

El agrupamiento aglomerativo se puede ejecutar a través de la clase *"AgglomerativeClustering*" del módulo "*sklearn.cluster"*.

En esta clase, se tienen en cuenta los siguientes argumentos:

"*n_clusters"*, en el que se especifica la cantidad de clústeres que se desean obtener; "*affinity"* el cual permite escoger entre las medidas de afinidad.

Por defecto emplea la distancia euclidiana '"*euclidean"'* y finalmente la conexión o *linkage* dentro de los que se encuentra "*ward*', el cual minimiza la varianza de los clústeres que se unen; '*complete*' para conexión completa, "*average"* para conexión promedio y '*single*' para una conexión simple.

Suponga el conjunto de puntos trabajado en el agrupamiento anterior y se desea aplicar agrupamiento aglomerativo con dos clústeres y con la conexión completa. La creación de la clase sería la siguiente:

Figura 4-63. Creación de una clase "*AgglomerativeClustering"* de dos clústeres y de conexión completa.

```
Aglom_cluster=sklearn.cluster.AgglomerativeClustering(n_clusters=2,
                                                      linkage='complete')
```

Fuente: Elaboración propia con *Jupyter Notebook*.

Luego se ajusta el agrupamiento jerárquico a los datos de la tabla de datos creada con las variables a y b trabajadas en el modelo de agrupamiento anterior como sigue:

Figura 4-64. Ajuste de un modelo de agrupamiento aglomerativo.

```
Aglom_cluster.fit(tabla_cluster)
```

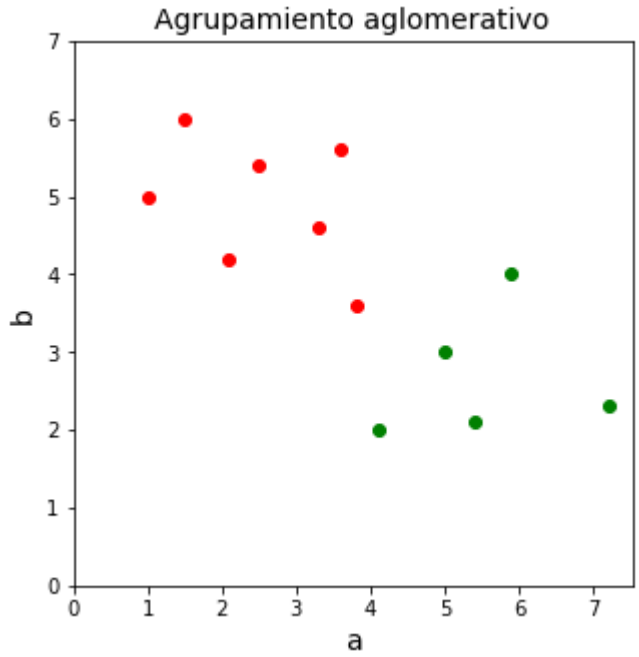
```
AgglomerativeClustering(affinity='euclidean', compute_full_tree='auto',
            connectivity=None, linkage='complete', memory=None,
            n_clusters=2, pooling_func=<function mean at 0x0397A5D0>)
```

Fuente: Elaboración propia con *Jupyter Notebook.*

Mediante el atributo "*labels_"* de la clase, se encuentran las etiquetas para cada observación.

A través de estas es posible graficar los clústeres. El siguiente gráfico muestra los clústeres obtenidos diferenciados con colores:

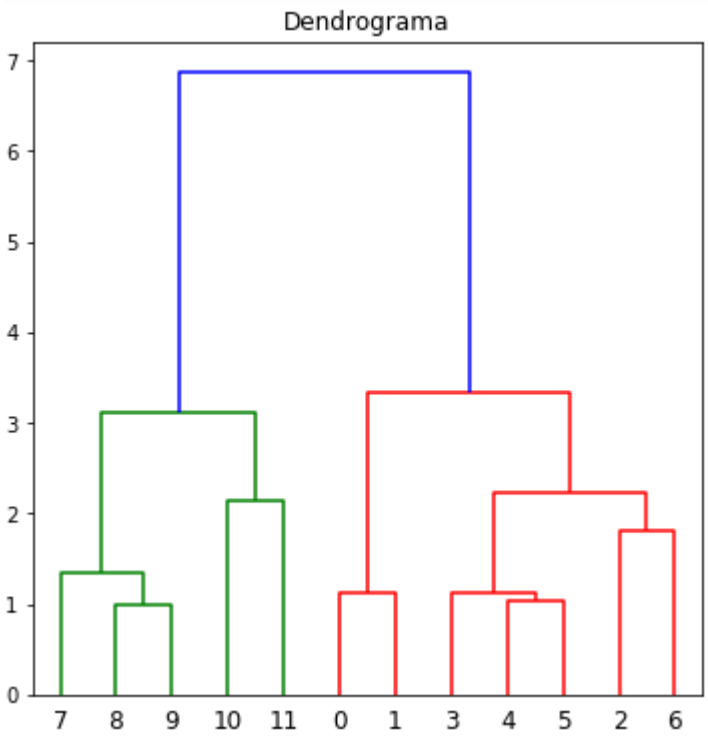
Gráfico 4-7. Clústeres obtenidos mediante agrupamiento aglomerativo.



Fuente: Elaboración propia con *Jupyter Notebook.*

Otro modulo con el que se puede elaborar el agrupamiento aglomerativo es "*scipy.cluster.hierarchy"*, el cual además facilita la creación del dendrograma a través de la funcion "*dendrogram"* aplicada a un objeto "*hierarchy.linkage"*, el cual a su vez es la función que ejecuta el modelo de agrupamiento. El dendrograma para el ejemplo anterior se muestra en la siguiente figura:

Gráfico 4-8. Dendrograma para el modelo ajustado propuesto.



Fuente: Elaboración propia con *Jupyter Notebook.*

En el dendrograma anterior se diferencian los clústeres en colores rojo y verde. Si se traza una línea horizontal a una altura determinada, se puede encontrar otra cantidad de clústeres equivalente a la cantidad de intersecciones que se den con las ramas del árbol y el conjunto de observaciones debajo de la rama serían un clúster.

El árbol parte desde las hojas o clústeres 8 y 9 (teniendo en cuenta que la numeración inicia desde 0), agrupándolos en un solo clúster con una medida de afinidad equivalente a 0.98, dada por la distancia euclidiana entre los puntos. Luego se agrupan los puntos 4 y 5 presentando una afinidad de 1.04, seguido por los puntos 0 y 1 con una afinidad de 1.12.

Se continúa con la agrupación del clúster 3 con el clúster 4 y 5 teniendo en cuenta la medida de conexión completa en donde el máximo de las distancias entre los elementos

del clúster 4 y 5 con el clúster 3 es de 1.13. Una vez se termine el algoritmo, se obtiene el dendrograma mostrado.

Tanto el agrupamiento de *"K"* medias como el agrupamiento aglomerativo basado en la distancia euclidiana deben tener en cuenta la escala en la que están expresadas las variables, ya que interfieren en el valor de la afinidad y por tanto, en todo el agrupamiento. (James, Witten, Hastie, & Tibshirani, 2013)

En el agrupamiento jerárquico también se debe escoger apropiadamente la medida de afinidad, porque diferentes métodos pueden agrupar las observaciones de distintas maneras, unas más convenientes que otras dependiendo el objetivo del analista.

4.6 Ejercicios de la unidad 4

1. Importe la tabla de datos "*Consumo_cerveja.csv"* disponible en Kaggle. Note que debe hacerse un tratamiento previo al uso de los datos. Comience eliminando los datos tipo "*NaN"*, luego utilice la función mostrada en la siguiente figura para reemplazar comas por puntos utilizando *expresiones regulares* para cada variable que lo requiera y finalmente cambie el tipo de dato para que sea de tipo numérico.

Figura 4-65. Uso de expresiones regulares para buscar y reemplazar ocurrencias de un patrón.

```
1 import re
2 for i,valor in enumerate(cerveza["Temperatura Media (C)"]):
3     cerveza.loc[i,"Temperatura Media (C)"]=re.sub(r"([,])",".",
4                                                   str(valor))
```

Fuente: Elaboración propia con *Jupyter Notebook.*

2. Elabore un modelo de regresión lineal simple para predecir el consumo de cerveza diario utilizando como variable predictora la temperatura máxima en el día. ¿Cuál

es el valor predicho de litros de cerveza consumidos cuando la temperatura máxima en el día es de 30°C?, Empleando la validación cruzada de k iteraciones, ¿cuál es el valor promedio de los coeficientes R^2?

3. Emplee ahora como variables predictoras la temperatura máxima del día y si fue o no fin de semana. Emplee la validación cruzada con 10 iteraciones para definir si este modelo es mejor que el propuesto en el ejercicio 2.

   ¿Cuánto es el incremento promedio del consumo de cerveza por unidad de grado Celsius?,

   ¿Cuánto si es día fin de semana?

4. Elabore un modelo de árbol de regresión con tres niveles para predecir el consumo de cerveza a partir de la temperatura máxima del día y de si fue o no fin de semana. Empleando la validación cruzada con 10 iteraciones,

   ¿Es mejor el modelo de árboles de regresión propuesto o los modelos de regresión lineal empleados en los puntos anteriores para predecir el consumo de cerveza en un día?

5. Importe el conjunto de datos "*heart*" y elabore un modelo de regresión logística para predecir si un paciente presenta o no enfermedad del corazón a partir del tipo de dolor de pecho sufrido (cp) y el ritmo cardiaco máximo registrado.

   ¿Cuál es la probabilidad de que una persona sufra de enfermedad cardiaca si presenta un dolor de pecho tipo 0, y un ritmo cardiaco máximo de 164 latidos por minuto?

6. Importe de cualquier forma el conjunto de datos "*iris*" descrito en el capítulo 2. Elabore un modelo de análisis de discriminante lineal, un modelo de árboles de clasificación con una profundidad máxima de 3 y un modelo de k vecinos más próximos con k=4, en donde emplee los cuatro rasgos de las plantas para predecir

la especie iris. Obtenga la matriz de confusión para cada modelo y responda: ¿Cuál modelo predice correctamente más especies?

7. Emplee los dos modelos de agrupamiento vistos para agrupar clústeres con las medidas de la tabla de datos *iris*. Sabiendo la clasificación de las especies de iris, utilice una matriz de confusión para determinar cuál modelo agrupa mejor las observaciones en 3 clústeres.

5. Descripción y análisis de resultados

Como resultado se obtuvo una guía que consta de cuatro capítulos con los cuales se pretendió introducir a la enseñanza de métodos de aprendizaje automático, herramienta de la ciencia de datos, utilizando los módulos especializados para ellos disponibles en el lenguaje de programación *Python*. La guía inicia contextualizando al lector sobre la definición de ciencia de datos dada por varios autores, que caracterizan esta disciplina y la hacen diferente a otras disciplinas como la estadística.

Más adelante se define el lenguaje de programación *Python* y la distribución Anaconda que se requiere para importar fácilmente módulos estadísticos e interfaces gráficas de usuario, toda vez que la guía está elaborada desde la interfaz *Jupyter Notebook*, asequible gracias a la distribución Anaconda.

Luego de una breve introducción a la ciencia de datos, el lenguaje de programación de *Python* y la distribución Anaconda, se inician los capítulos de la guía, en donde cada capítulo lista los temas que se van a aprender.

La guía está estructurada de forma tal que brinde las bases fundamentales para posteriormente, en capítulos siguientes, se puedan ejecutar los modelos y resolver inconvenientes que se presentan a la hora de implementar los mismos.

El primer capítulo de la guía se creó con el fin de conocer las operaciones básicas y líneas de código elementales en el lenguaje de programación *Python*, adoptando los elementos básicos que se presentan a la hora de aprender cualquier lenguaje de programación. Sin embargo, como se trata de una guía enfocada en el análisis de datos, no se trabajaron elementos fundamentales de la programación como la creación y definición de clases, o el

manejo de errores a través de excepciones, el almacenamiento, lectura, actualización o eliminación de archivos; pues se sale del propósito general de la guía.

Así las cosas, a través de la enseñanza de los tipos de variables, las estructuras de datos básicas, los operadores, estructuras de control y definición de funciones se creó una base para el entendimiento de las herramientas que más se utilizan en los capítulos siguientes.

En el segundo capítulo, una vez se tiene una idea de cómo escribir líneas de código en *Python*, se introduce la estructura básica para el manejo de los datos, correspondiente a la tabla de datos o "*Dataframe"*, ya que a partir de esos objetos se pueden analizar los datos con diversas herramientas y funciones del mismo o de otros módulos.

Este capítulo se centra en explicar cómo importar la información a través de archivos de internet o presentes en el disco duro. No obstante, antes de analizar datos es necesario saber cómo obtener la información y cómo realizar ajustes en caso de que los datos provengan con errores o falte añadir variables.

Si los datos no se tienen de la forma adecuada, es imposible continuar al análisis de los mismos. Por eso la importancia de que en cada capítulo hay una parte enfocada a resolver los posibles problemas que se presenten con la información. Muchas soluciones pueden encontrarse a partir de uso de las herramientas del capítulo 1.

Nuevamente, como el propósito de la guía se enfoca más al análisis de datos, se deja de lado otro de los objetivos de la programación, que es el de minimizar el uso del código y el tiempo de ejecución.

En un principio el objetivo consiste en que el aprendiz resuelva una tarea sin importar si minimiza las líneas de código. Así mismo, la guía no está elaborada empleando el mínimo de líneas posible, ni empleando las funciones o herramientas que minimizan el tiempo de ejecución de tareas específicas.

Teniendo en cuenta que el aprendiz se ha familiarizado con las tablas de datos, se introdujo el capítulo 3, el cual trata de explicar conceptos de estadística que se emplean en modelos posteriores.

La estadística es una ciencia bastante compleja que requiere mucho tiempo de estudio y es una herramienta primordial en la ciencia de datos. El aprendizaje de la ciencia de datos debe estar acompañado del aprendizaje de la estadística, lo cual se trató en el capítulo 3.

No obstante, a pesar de que es aconsejable un buen conocimiento de la estadística, la guía no incluye tantas explicaciones y temas en esa materia, toda vez que resultaría demasiado extenso, sí se considerara explicar a detalle todos los conceptos, teoremas, axiomas, que hay detrás de la estadística.

Este capítulo también introduce al módulo que permite elaborar diferentes tipos de gráfico, necesarios para complementar los conceptos y sobretodo, ser la base para la visualización de los datos. Otro propósito del tercer capítulo consiste en la exploración de los datos, a través de medidas y parámetros estadísticos con los cuales el aprendiz crea una idea general de cómo se distribuyen los datos.

Finalmente, el ultimo capitulo presenta varios métodos de aprendizaje automático, los cuales siempre están presentes en cursos de aprendizaje de ciencia de datos, mediante dos módulos de *Python* especializados para ello, como son "*scikit-learn* y *statsmodels"*.

Existen muchos algoritmos y modelos de aprendizaje automático. En la guía se presentaron los más básicos partiendo del uso del objeto "*DataFrame"* trabajado en el capítulo 2. Ahora, dada la complejidad de algunos modelos, la guía solo presentó las ecuaciones generales del modelo sin explicar a fondo de dónde provienen, ya que puede ser enredador y sobretodo no entendible si no se tiene un amplio conocimiento en estadística.

Este capítulo presenta la división de los modelos y algoritmos entre el aprendizaje supervisado y el aprendizaje no supervisado e incorpora una herramienta para que el aprendiz pueda evaluar la calidad de los modelos con el fin de compararlos y obtener el que mejor resultados provee para un determinado conjunto de datos.

A través de los modelos se hace uso de las predicciones, tarea muy solicitada en el aprendizaje automático, empleando las funciones dispuestas en los módulos para ello.
En la medida que el aprendiz vaya haciendo uso de los modelos, le será más fácil incorporar aquellos más avanzados no presentes en la guía y disponibles a través de "*scikit-learn"*.

Los ejercicios de los módulos se plantearon de tal forma, que el aprendiz, al realizar las actividades repase lo estudiado a través del capítulo. No se incluyen problemas diferentes a lo trabajado, por lo que son estrategias para familiarizarse con el código en la medida en que la repetición de las acciones provee mejor reconocimiento de las funciones y de la sintaxis a la hora de escribir líneas de código, además de que cumplen con el propósito de predicción y comparación de modelos empleado en el aprendizaje automático.

6. Evaluación y cumplimiento de los objetivos

Retomando los objetivos de la guía y partiendo de los objetivos específicos al inicio de este proyecto:

Objetivo principal

- *Generar una guía metodológica introductoria a la ciencia de datos.*

Se ha elaborado una guía estructurada en cuatro capítulos sintetizando principales necesidades y actividades realizadas en el aprendizaje de la ciencia de datos, correspondientes a los siguientes aspectos:

El aprendizaje de un lenguaje de programación que tenga orientación o módulos relacionados con el análisis de datos, en el que se utilizó *Python*, a través del aprendizaje de variables, ciclos de control, funciones, operadores, estructuras de datos básicos y operaciones básicas. Esto corresponde al capítulo 1.

El uso y creación de una estructura básica para organizar datos, en donde se trabaja el objeto "*DataFrame"* del módulo "*pandas"* de *Python*, corrección, modificación, inserción de elementos de las tablas de datos, filtros e importación de tablas de datos, todo esto trabajado en el capítulo 2

El aprendizaje de conceptos básicos de la estadística, la exploración de datos, el uso de diagramas para visualizar los datos contenidos en las tablas de datos, el cálculo de medidas de tendencia central y de dispersión y la correlación con el fin de comprender la distribución y las relaciones de los datos, trabajaron en el capítulo 3.

La aplicación de modelos y algoritmos de aprendizaje automático para interpretar las relaciones entre variables predictoras y de respuesta o para predecir una variable deseada, se presentan en diversos cursos de aprendizaje de ciencia de datos, por tal razón, algunos de los más representativos se trabajan en el capítulo 4.

El propósito general consiste en que cada capítulo se trabaje con una metodología que consta de un proceso de lectura, una retroalimentación visual y una propuesta de ejercicios aplicados que permitan aplicar los conocimientos adquiridos.

Objetivos específicos

- *Definir y contextualizar la guía metodológica elaborando y desarrollando las unidades temáticas particulares en concordancia con el objetivo general.*

  La guía se ha elaborado partiendo del concepto de la ciencia de datos y de las herramientas que se utilizan para hacerla, es decir, mediante lenguajes de programación que dispongan de módulos para el análisis de datos. En este caso, se propuso el lenguaje de programación *Python*. En cada capítulo de la guía, se enuncian las temáticas a abordar, todas dirigidas a la elaboración de modelos de aprendizaje automático, con los cuales se busca el objetivo de predecir e inferir sobre un conjunto de datos, siendo estos dos últimos principales tareas en el aprendizaje automático y que hacen parte de la ciencia de datos.

- *Introducir y aplicar el lenguaje de programación Python para hacer un acercamiento a la ciencia de datos.*

  La introducción del lenguaje de programación *Python* se dio a través de una breve descripción del lenguaje y de una distribución asociada conocida como Anaconda, junto con el capítulo 1, el cual contiene numerosos ejemplos básicos para ejecutar líneas de código simples, basadas en operaciones aritméticas, operadores, tipos de variables y los más importantes, los ciclos de control y definición de funciones.

Estos temas se introducen con el fin de que el aprendiz comience a realizar sus propios ejemplos y operaciones a modo de práctica, toda vez que hacen parte de cualquier proceso de enseñanza y aprendizaje de la programación.

A pesar de ser un capítulo introductorio, tiene bastante relevancia tanto en lo referente al tratamiento futuro de los datos y al manejo de las variables, como a la aplicación de ciclos de control para recorrer elementos de una tabla de datos o diseñar sus propias funciones para que cumplan con múltiples tareas, tales como organizar datos para luego graficarlos, lo cual corresponde a procedimientos presentes cuando se trabaja la ciencia de datos.

- *Enseñar y aplicar fundamentos de estadística en la ciencia de datos.*

Los fundamentos de estadística se presentaron en el capítulo 3, una vez se trabajaron las tablas de datos. Los fundamentos estadísticos trabajados se relacionan con el análisis y la representación de los datos. Se definió la estadística descriptiva, estadística inferencial, conceptos de población, muestra, se diferenciaron algunos tipos de variables, se trabajaron gráficos para la representación de la distribución de frecuencias y medidas de tendencia central. El concepto de probabilidad se manejó como una frecuencia relativa y se enunció el teorema de Bayes, muy presente en el aprendizaje automático.

- *Elaborar videos prácticos de apoyo para el aprendizaje de la ciencia de datos.*

Se elaboraron videos que muestran la interfaz de Jupyter Notebook y muestra la ejecución de líneas de código relacionadas con cada capítulo de la guía. Los videos tienen el propósito de que el aprendiz retroalimente las funciones más importantes del capítulo y sirven como método para dinamizar el aprendizaje a través de los medios audiovisuales.

- Generar ejemplos y propuestas de ejercicios como complemento necesario para el aprendizaje.

Al final de cada capítulo se formularon ejercicios con el fin de que el aprendiz repase lo trabajado en el mismo y tenga la capacidad de construir y desarrollar sus propios modelos. El planteamiento de estos ejercicios es la segunda retroalimentación, después de los videos de apoyo, con los que el aprendiz trabaja el lenguaje de programación y se familiariza con él para resolver los inconvenientes que se le presenten.

Cabe resaltar que los ejercicios propuestos no tienen carácter investigativo, todos se pueden resolver mediante la guía de tal manera, que el aprendiz es quien explora, descubre e indaga otras formas para resolver problemas que se le presenten o se le ocurran.

7. Conclusiones

Python es un lenguaje de programación que dispone módulos para el análisis de datos y tratamiento estadístico, siendo un lenguaje de propósito general, lo que le da ventaja frente a otros lenguajes de programación como *"R"*, el cual facilita la escritura de un código legible y mantenible. No obstante no posee la extensa variedad de librerías que disponen otros lenguajes como *"R"* para el tratamiento de datos y no es preferido para visualizar datos. La alternativa a este inconveniente es la integración de lenguajes de programación, por lo cual es importante conocer múltiples lenguajes de programación en la ciencia de datos.

El científico de datos es un individuo con conocimientos multidisciplinarios que comprenden la programación, el manejo de información, la estadística y que puede aplicarlos en casi cualquier área, destacando el campo de los negocios o a nivel empresarial en donde tiene gran relevancia el ingeniero industrial para la elaboración de pronósticos de ventas, investigación de mercados, inferencia y toma de decisiones. Se ha visto entonces, la aplicabilidad de la ciencia de datos en la medicina, medio ambiente, demografía, biología y en muchos otros aspectos como el reconocimiento de figuras o patrones, entre otras.

Las clases del módulo "*sklearn"* disponen de más argumentos para especificar un modelo con criterios, limitaciones, ajustes específicos, por lo cual se debe tener precaución a la hora de ejecutar los modelos, toda vez que puede darse el caso de que un algoritmo no alcance un óptimo porque tiene por defecto insuficientes iteraciones, demasiada regularización u otros ajustes. Sin embargo, dichos ajustes permiten regular el gasto computacional, para casos en los que se tenga enormes cantidades de datos.

El módulo "*sklearn"* es una poderosa herramienta para aplicar modelos de aprendizaje automático, ya que contiene casi 100 modelos de aprendizaje supervisado, alrededor de 9 modelos de agrupamiento y trabaja también la reducción de dimensionalidad, selección de modelos y pre procesamiento de datos, entre otros.

El aprendizaje de lenguajes de programación está muy ligado al ensayo y error. La modificación del código y la comprensión del resultado debido a dicho cambio es la que genera mejor comprensión de las funciones y métodos presentes en las clases de los módulos que se encuentran en los lenguajes de programación.

Python es un lenguaje de programación que ha venido tomando relevancia en la ciencia de datos. Cabe resaltar que no existe tanta información y fuentes de este lenguaje comparado con otros lenguajes como *"R"* para realizar tareas de ciencia de datos. Sin embargo está surgiendo más como una alternativa a otros lenguajes, con ventajas claras como el ser un lenguaje de propósito general y que es más rápido en la ejecución de las tareas comparado con otros lenguajes.

De otro lado, la comprensión de los modelos de aprendizaje automático suele verse opacada ante la capacidad de las funciones y modelos de los módulos de los lenguajes de programación. Es importante no dejar de lado la teoría de los modelos, ya que el objetivo no solo radica en la aplicación de todos los modelos, sino también en comprender cómo funciona el modelo para distinguir aquellos más convenientes para un conjunto de datos dado.

La complejidad de la ciencia de datos hace imposible incluir todo lo que abarca en una sola guía. Temas como el requerimiento de información se lleva a cabo usualmente a través de otros lenguajes como *SQL*, el almacenamiento de la información mediante otras herramientas como MySQL, Oracle o SQL server y así muchas otras necesidades que se presentan cuando se trabaja ciencia de datos y que son parte esencial de dicho campo

Todo aprendiz de la ciencia de datos requiere de un amplio conocimiento en matemáticas y estadística para comprender técnicas, algoritmos, modelos computacionales, métodos empleados en la ciencia de datos, previo análisis de datos para saber si los modelos tienen validez en determinado conjunto de datos. Existen demasiados modelos y algoritmos para

el aprendizaje automático, algunos tan complejos que existen libros para cada tipo, por lo que no es posible incluir todos los modelos y algoritmos en una sola guía.

Finalmente, las funciones y modelos de los módulos de *Python* son susceptibles a cambios o modificaciones en su funcionamiento. Es importante estar al día en las actualizaciones que se hacen a los módulos, en este caso al módulo "*sklearn"* ya que pueden existir errores en las funciones o clases y estas son corregidas a través de nuevas actualizaciones. Cabe resaltar que si se encuentran errores es prudente informar a los desarrolladores sobre las fallas que presenten los modelos y algoritmos, con el fin de que puedan ser solucionadas.

8. Recomendaciones

Además del aprendizaje de los lenguajes de programación para el análisis de datos, es importante aplicar otros elementos de la programación para agregar valor al aprendizaje y crear programas que resuelvan requerimientos, como por ejemplo: Guardar información en archivos, actualizar información o eliminar la información, ya que los datos en memoria son temporales mientras el ordenador está encendido; incorporar manejo de errores en la creación de funciones, crear clases para guardar funciones y métodos nuevos, y formar módulos propios, entre otras.

Los trabajos posteriores deberían incluir la integración de lenguajes de programación, es decir el uso simultaneo de lenguajes como "*Python", "R", "Julia"*, entre otros, en el que se aplique las ventajas de cada lenguaje para lograr un desempeño optimo a la hora de ejecutar tareas de aprendizaje automático, hasta presentación de informes relacionados con ellos. Como se mencionó en las conclusiones, es recomendable aprender a hacer solicitudes con el lenguaje *SQL* para expandir el tema de importación de tablas de datos.

Es necesario complementar el aprendizaje con más conceptos estadísticos y especialmente en probabilidad, ya que por ejemplo, esta se presenta en todos los modelos que utilizan el teorema de Bayes, el aprendizaje de las distribuciones de probabilidad, el complemento de las pruebas de hipótesis con otros parámetros, métodos no paramétricos, pruebas de bondad de ajuste, etc.

Esta dinámica facilita la comprensión de los modelos e incluso la creación de los mismos y puede compaginar con el tema de la aleatoriedad que hace parte de la estadística y se evidencia en grandes temas como la simulación que hacen parte de la formación del ingeniero industrial.

Dada la enorme cantidad de modelos de aprendizaje automático, es importante continuar con la exploración de modelos más avanzados y precisos como por ejemplo modelos de bosques aleatorios que parten de árboles de decisión.

También se deben explorar las herramientas para reducir la dimensionalidad, con el fin de facilitar el procesamiento de los datos o mejorar la calidad de los modelos y aquellas para seleccionar la flexibilidad de los modelos con el fin de minimizar la varianza y la "*bias"*, en busca del mejor modelo posible que represente un conjunto de datos.

Existen muchos otros inconvenientes que se pueden presentar durante el manejo de tablas de datos o aquellos relacionados con los datos que contienen. Se recomienda dominar las expresiones regulares que permiten identificar rápidamente partes de texto o caracteres específicos con el fin de visualizarlos, modificarlos o reemplazarlos. Otras particularidades importantes son el manejo de fechas y horas, para las cuales existen módulos específicos y el manejo de datos incompletos.

Posterior al entendimiento del aprendizaje automático se debe continuar con modelos de aprendizaje profundo como por ejemplo las redes neuronales. El módulo de aprendizaje automático más empleado en *Python* es *"Tensorflow"*, con este se pueden crear modelos avanzados de redes neuronales para solucionar problemas de aprendizaje automático.

Índice

Anexos

Respuestas a ejercicios

Capítulo 1.

2. [47, 78, 102], [887, 945, 975]

4. [0.70017637 0.29982363]
   [0.57823129 0.42176871]

   [0.65854579 0.34145421]
   [0.65851883 0.34148117]

   [0.65853659 0.34146341]
   [0.65853659 0.34146341]

6. a. [0.02, 0.85, 0.78, 0.81, 0.94, 0.17, 0.5, 0.93, 0.46, 0.09]
   b. [0.18, 0.71, 0.62, 0.39, 0.58, 0.51, 0.22, 0.59, 0.98, 0.31]
   c. [0.79, 0.43, 0.75, 0.91, 0.99, 0.03, 0.55, 0.31, 0.19, 0.63]

Capítulo 2.

2. Si.
3. 17, 21.
4. a. 14.
   b. 10.
5. No, 49.

Capítulo 3.

1. bedrooms=33.
2. Si.
3. sqft_living (pies cuadrados de la vivienda).
4. Tendencia a disminuir.
5. Alza.
6. Si, 6.56, 2.29e-9.

Capítulo 4

2. 27.62, 0.248.
3. 0.6, 0.677, 5.185.
4. 0.525, no es mejor.
5. 0.418.
6. ADL: 147/150, AC: 146/150, KVMP: 144/150.
7. Iguales, 134/150.

Bibliografía

Ajay, O. (2018). *Python for R Users: a data science approach.* New Jersey: John Wiley & Sons Inc.

Anaconda Inc. (2019). *Anaconda Distribution*. Retrieved from The world's Most Popular Python/R Data Science Platform: https://www.anaconda.com/distribution/

Arthur, D., & Vassilvitskii, S. (2006). *k-means++: The Advantages of Careful Seeding.* Stanford. Retrieved from k-means++: The Advantages of Careful Seeding.

Beau, C. (2019, Noviembre 1). *What is a CSV File and How to Open the CSV File Format*. Retrieved from freeCodeCamp: https://www.freecodecamp.org/news/what-is-a-csv-file-and-how-to-open-the-csv-file-format/

Bhavesh, B. (2018, Febrero 17). *Gini index based Decision Tree [archivo de video]*. Retrieved from https://www.youtube.com/watch?v=2lEcfRuHFV4

Canavos, G. (1998). *Probabilidad y estadística aplicaciones y métodos.* México: McGRAW-HILL.

Cao, L. (2017). Data Science: A Comprehensive Overview. *ACM Computing Surveys, 50*(3), 43:8. doi:10.1145/3076253

Fisher, D. (1970). *CONTROL STRUCTURES FOR PROGRAMMING LANGUAGES.* Pittsburgh: Carnegie Mellon University. Retrieved from https://pdfs.semanticscholar.org/237f/33308e8e9dc794e56307649155e6aa7a5882.pdf

harlfoxem. (2016). *Kaggle*. Retrieved from House Sales in King County, USA: https://www.kaggle.com/harlfoxem/housesalesprediction

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An Introduction to Statistical Learning with Applications in R.* New York: Springer.

JSON.org. (2017). *Introducción a JSON*. Retrieved from https://www.json.org/json-es.html

Jupyter Team. (2015). *User Documentation*. Retrieved from The Jupiter Notebook: https://jupyter-notebook.readthedocs.io/en/latest/notebook.html

Jupyter Team. (2019, Julio 8). *IP[y]*. Retrieved from IPython Interactive Computing: https://nbviewer.jupyter.org/github/ipython/ipython/blob/master/examples/IPython%20Kernel/Index.ipynb

Kim, P. (2017). Machine Learning. In P. Kim, *MATLAB Deep Learning* (p. 2). Berkeley, California: Apress.

Kirkman, T. (1996). *Display of Statistical Distribution*. Retrieved from Statistics to Use: http://www.physics.csbsju.edu/stats/display.distribution.html

Learn Python Org. (2019). *Welcome*. Retrieved from Modules and packages: https://www.learnpython.org/en/Modules_and_Packages

Mindfire Solutions. (2017, Octubre 3). *Python: 7 Important Reasons Why You Should Use Python*. Retrieved from https://medium.com/@mindfiresolutions.usa

Mueller, J. (2018). *Beginning Programming with Python.* Hoboken, New Jersey: John Wiley & Sons Inc.

Nighania, K. (2018, Diciembre 30). *Various ways to evaluate a machine learning model's performance*. Retrieved from towards data science: https://towardsdatascience.com/various-ways-to-evaluate-a-machine-learning-models-performance-230449055f15

Pedregosa, e. a. (2011). *Scikit-learn*. Retrieved from Machine Learning in Python: https://scikit-learn.org/stable/index.html

Piraeus, Z. (2017, Septiembre 11). *stackoverflow*. Retrieved from How to use the pass statement?: https://stackoverflow.com/questions/13886168/how-to-use-the-pass-statement

Python Software Foundation. (2019, Junio 4). *The Python Standard Lybrary*. Retrieved from Built-in Functions: https://docs.python.org/2/library/functions.html#help

Python Software Foundation. (2019, Junio 17). *The Python tutorial*. Retrieved from https://docs.python.org/3.6/tutorial/datastructures.html

Python Software Foundation. (2019, Junio 4). *The Python Tutorial*. Retrieved from Data Structures: https://docs.python.org/2/tutorial/datastructures.html

Python Software Foundation. (2019). *What is Python? Executive Summary*. Retrieved from https://www.python.org/doc/essays/blurb/

Ripley, B. (2019). *RDocumentation*. Retrieved from MASS v7.3-47: https://www.rdocumentation.org/packages/MASS/versions/7.3-47/topics/Traffic

Rossum, G. (2009, Enero 13). *The History of Python*. Retrieved from Introduction and Overview: http://python-history.blogspot.com/2009/01/introduction-and-overview.html

Saltz, J., & Dewar, N. (2019). Data science ethical considerations: a systematic literature review and proposed proyect framework. *Ethics and Information Technology*. doi:10.1007/s10676-019-09502-5

Scikit-learn . (2019). *Clustering*. Retrieved from K-means: https://scikit-learn.org/stable/modules/clustering.html#k-means

Scikit-learn. (2019). *Metrics and scoring: quantifying the quality of predictions*. Retrieved from https://scikit-learn.org/stable/modules/model_evaluation.html#accuracy-score

scikit-learn developers. (2019). *sklearn.metrics.f1_score¶*. Retrieved from scikit-learn: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html#sklearn.metrics.f1_score

SciPy org. (2019, Enero 31). *numpy.histogram*. Retrieved from https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram.html#numpy.histogram

sentdex. (2015, julio 11). *Matplotlib Tutorial 6 - Pie Charts [archivo de video]*. Retrieved Julio 24, 2019, from https://www.youtube.com/watch?v=Oh2Dkkswy30

Starmer, J. (2018, Enero 22). *StatQuest: Decision Trees*. Retrieved from https://www.youtube.com/watch?v=7VeUPuFGJHk

Statistics How To. (2020). *Coefficient of Determination (R Squared): Definition, Calculation*. Retrieved from statisticshowto.datasciencecentral.com/probability-and-statistics/coefficient-of-determination-r-squared/

Takemura, A. (2018). A new era of statistics and data science education in Japanese universities. *Perspectives on data science for advanced statistics, 1*, 110,113. doi:10.1007/s42081-018-0005-7

The pandas proyect. (2019). *pandas 0.19.2 documentation*. Retrieved from pandas.DataFrame.sort_values: http://pandas.pydata.org/pandas-docs/version/0.19/generated/pandas.DataFrame.sort_values.html#pandas.DataFrame.sort_values

The pandas proyect. (2019). *pandas 0.24.2 documentation*. Retrieved from pandas.Dataframe.iloc: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iloc.html

The pandas proyect. (2019). *pandas 0.24.2 documentation*. Retrieved from pandas.DataFrame.drop: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html

The pandas proyect. (2019). *pandas 0.25.0 documentation*. Retrieved from pandas.set_option: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.set_option.html

The SciPy community. (2019, Enero 31). *Linear algebra (numpy.linalg)*. Retrieved from Matrix and vector products: https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

tutorialspoint. (2019). *Computer Programming Tutorial*. Retrieved from Computer Programming-Operators: https://www.tutorialspoint.com/computer_programming/computer_programming_operators

tutorialspoint. (2019). *Python 3 Basic Tutorial* . Retrieved from Python 3 - Lists: https://www.tutorialspoint.com/python3/python_lists.htm

Universidad Nacional de Colombia. (2016, Junio 29). *Bdigital REPOSITORIO INSTITUCIONAL UN*. Retrieved from Plantilla para la presentación de Tesis UN: http://www.bdigital.unal.edu.co/50813/

vincentarelbundock. (2019). *Salaries*. Retrieved from R Documentation: https://vincentarelbundock.github.io/Rdatasets/doc/carData/Salaries.html

Walpole, R., Myers, R., & Myers, S. (1998). *Probabilidad y estadística para ingenieros.* México: Pearson.

Weihs, C., & Ickstadt, K. (2018). Data science: the impact of statistics. *International Journal of Data Science and Analytics*(6), 189-194. Retrieved from https://doi-org.bdigital.udistrital.edu.co/10.1007/s41060-018-0102-5