Ramírez - Araiza - Salazar

Fundamentos de Analítica de Datos con Python:

ETL, limpieza e ingeniería de datos

APRENDA

Fundamentos de Analítica de Datos:
ETL, limpieza e ingeniería de datos.

Este libro es una guía completa para aquellos interesados en los fundamentos teóricos de la analítica de datos y en el proceso de análisis de situaciones del mundo real. Se cubre desde la identificación de supuestos, formulación de hipótesis, diseño y clasificación de variables de trabajo, hasta la realización de ETL. Esta obra ofrece una visión detallada y práctica de cómo llevar a cabo un análisis de datos eficiente y efectivo.

En el libro, se explican técnicas formales para la identificación y documentación de diferentes fuentes de datos requeridas para satisfacer las necesidades de información específica. Se aborda la forma de recuperar datos de múltiples fuentes, hacer integración (data integration), limpieza e ingeniería de datos (feature engineering). También se cubren temas como el tratamiento de datos atípicos y ausentes, el cálculo de muestras aleatorias simples y estratificadas, así como transformaciones de datos, clásicas, tales como eliminación de duplicados, conversiones, cambios de tipo, columnas calculadas, cálculos agrupados, manipulación de cadenas y tratamiento de datos categóricos.

Con el objetivo de brindar una experiencia de aprendizaje completa, se proporcionan ejemplos en Python para cada uno de los temas abordados, usando las librerías Numpy, pandas y matplotlib. Además, se presentan casos de estudio que ilustran cómo aplicar estos conceptos en situaciones del mundo real.

En resumen, este libro es una herramienta esencial para cualquiera que busque mejorar su comprensión de la analítica de datos y adquirir habilidades prácticas para aplicar en su trabajo diario, usando Python, incluso si no son programadores. Con su enfoque detallado, accesible y práctico, es un recurso valioso para estudiantes, profesionales y entusiastas de la analítica de datos.

9 786079 597948

aprenda.mx
@aprendapracticando

Fundamentos de Analítica de Datos con Python:

ETL, limpieza e ingeniería de datos.

Felipe Ramírez
Ma. de Jesús Araiza
Francisco Salazar

Facultad de Contaduría Pública y Administración
Universidad Autónoma de Nuevo León, México.

APRENDA EDICIONES

Ramírez, J. F., Araiza, M. J., & Salazar, Á. F. (2023). Fundamentos de analítica de datos con Python: ETL, limpieza e ingeniería de datos. Monterrey, N.L.: Aprenda Ediciones.
ISBN: **978-607-95979-4-8**

Páginas: 336 Formato: 17.5 x 22.75 cm.

APRENDA PRACTICANDO
SAN FELIX 5432-D, VISTA SOL,
GUADALUPE, NUEVO LEÓN,
MÉXICO.
AÑO DE EDICIÓN: 2023
1ª EDICION.
ISBN: 978-607-95979-4-8
Release 4.0

F. Ramírez
A Viridiana,
copiloto confiable en ese
carro rojo que es la vida.

M. Araiza:
Para mis hijos Rodolfo, David y Alejandro.

El amor de mi vida no me dice "princesa",
ni "mi amor", me dice mamá.

Cada esfuerzo en mi vida es por ustedes.

F. Salazar:
A mi adorada esposa: La cómplice y compañera
insuperable que siempre me restaura sin
reservas ni condiciones.

A mi amada hija: La aventura y el proyecto más
hermoso en que me ha tocado participar.

A mis amigos: El acompañamiento y el bastión
sin los cuales la vida no sería el gozo que
tanto he disfrutado.

El contenido de este libro forma parte del curso "**Python para Data Science: ETL, limpieza e ingeniería de datos.**", disponible desde nuestras plataformas:

Aprenda.mx

AprendaStudio.com

AprendaPracticando.com

Descarga recursos, presentaciones y ejercicios, desde estos sitios.

Haz clic aquí, y recibe información adicional de este curso

Contenido

Introducción

¿Para quién es este libro?

Este libro forma parte de la colección Python para Data Science, desarrollado por Aprenda. La serie trata todos aquellos temas que van, desde lo más fundamental de la modelación y la analítica de datos, hasta temas especializados de la ciencia de datos.

Cada tomo de la serie se enfoca en un tema en particular.

Audiencia específica

Este libro está dirigido para personas con el siguiente perfil:

- Son personas no especialistas, ni en programación, ni en matemáticas, ni en estadística. Este es un curso para administradores, contadores, financieros, y todo aquél que maneje datos y necesite analizarlos.
- Tienen la necesidad de aprender los fundamentos de la analítica de datos, desde el punto de vista teórico.
- Tienen la necesidad de aprender cómo se integra información de diferentes fuentes, para integrar un solo conjunto de datos a partir del cual hacer trabajo de analítica de datos.
- Tienen la necesidad de aprender cómo desarrollar tareas de limpieza de datos, es decir, corregir en la medida de lo posible datos incorrectos, incompletos o incoherentes.

- Tienen la necesidad de aprender cómo desarrollar tareas de ingeniería de datos, es decir, seleccionar los datos que son relevantes, y excluir aquellos que no lo son.
- Tienen conocimientos básicos de programación en Python.
- Saben el conocimiento básico de cómo funcionan las libretas Jupyter, ya sea usando **Jupyter Notebook** o **Google Colab**.

¿Por qué leerlo?

Una habilidad *commodity* es aquella que todos deben saber desarrollar, y tenerla no te da ventaja competitiva profesional, pero no tenerla sí te deja en seria desventaja. Ejemplos de habilidades commodity hoy en día son hacer un documento en Word, o saber usar el correo electrónico.

Según Gartner, en el corto y mediano plazo, tener habilidades para realizar trabajos de analítica de datos van a ser un commodity. En la actualidad, los ingenieros de datos preparan las fuentes de datos, y se las dan a los analistas de datos, que preparan informes y tableros de datos, y se los dan al tomador de decisiones.

En el mediano y corto plazo, los ingenieros de datos prepararán fuentes de uso general, y los depositarán en un repositorio central; los tomadores de decisiones deberán tomar los datos desde ahí, y ser ellos mismos quienes hagan la analítica de datos que requieren para tomar sus decisiones.

La figura del analista de datos desaparece como tal, porque los tomadores de decisiones serán sus propios analistas. Esto nos deja con la necesidad de que todos los profesionales que en un momento dado requieran tomar decisiones, deben aprender a hacer analítica. El problema es que hacer analítica no es tan sencillo como aprender a usar el correo electrónico: requiere mucho más técnica.

En el mercado hay muchos libros y muchos cursos para aprender analítica de datos, ya sea con herramientas gráficas (**Power BI**, **Tableau**, **Excel**), y lenguajes de programación (**Python**, **R**). El problema con muchos de ellos es que se empeñan en complicarlo todo, explicando modelos matemáticos, estadística y modelación de datos, que mucha gente no especializada no entiende.

Este libro no es así: explica la teoría y las técnicas desde un punto de vista práctico y de utilidad. Hace énfasis en la práctica, y utiliza ejemplos sencillos que faciliten la comprensión de la técnica.

Estructura del libro

Si quieres aprender a hacer analítica de manera profesional, lo recomendable es leer todo el libro de manera secuencial.

En general, tiene dos partes:

1. **Parte teórica / formal (capítulos 1 al 5):** En esta parte se tratan conceptos y técnicas para analizar situaciones del mundo real, y traducir las observaciones en objetivos de análisis de datos, variables e hipótesis. El entregable de esta sección es haber definido las fuentes de datos que han de utilizarse, y haber identificado los datos que es necesario tener para realizar los trabajos de analítica. En esencia, esta sección trata el ***qué***.
    a. Si realizarás proyectos de analítica de datos desde cero, esta sección es indispensable para que adquieras habilidades blandas que te permitirán entregar el mayor valor que los datos pueden entregar, con conocimiento de causa.
    b. Si tu interés es programar Python con tareas relacionadas con la analítica de datos, puedes saltarte esta sección, sin problema.
        i. En los capítulos 4 y 5 se ven algunos temas de Python y la librería pandas, pero es con la finalidad de analizar la disponibilidad de datos, y tomar decisiones respecto a los objetivos de análisis y las hipótesis planteadas.

2. **Parte técnica / práctica (capítulos 6 al 13):** En esta parte se tratan a detalle técnicas de programación específica en lenguaje Python, para el desarrollo de tareas propias del modelo ETL (*Extract – Transform – Load*). Se cubren a mucho detalle las tareas de limpieza de

datos, así como de ingeniería de datos (*feature engineering*), tratamiento de datos ausentes y atípicos. El entregable de esta sección es un conjunto de datos de alta calidad, requerido para el procesamiento de datos con la herramienta de tu elección (**Excel**, **Power BI**, **Tableau**, **Python** o **R**).

  - Esta sección puede tomarse como referencia técnica para realizar tareas de limpieza de datos e ingeniería de datos, usando Python.

En el libro encontrarás:

- **Lecciones:** Explicación teórica y sintáctica de los temas de estudio para comprender la analítica de datos y las técnicas de limpieza e ingeniería de datos.
- **LABS:** Aplicación del lenguaje Python para desarrollar un trabajo de analítica completo. Los ejercicios van en secuencia, y requieren la realización de los ejercicios previos.
  - Se recomienda que realices por ti mismo los ejercicios.
  - Los recursos para el desarrollo de los ejercicios los encontrarás en **GitHub**:

https://github.com/AprendaPracticando/AnaliticaPythonR1

Cursos en línea y presenciales

La versión en videocurso de este libro está disponible en Udemy y en AprendaStudio.com.

Si deseas el contenido de este curso en modalidad Presencial o Virtual, o si deseas sesiones LIVE, Webinars y Conferencias con los autores, la información está disponible en Aprenda.mx

Estamos seguros de que este libro te será de utilidad.

Felipe Ramírez, Ma. de Jesús Araiza, Francisco Salazar

Archivos y recursos de trabajo

Formato de los archivos de trabajo

Los ejercicios de este curso son ***libretas de Jupyter***, de extensión **`.ipynb`**.

Si no sabes qué son las libretas de Jupyter, cómo se editan y cómo se ejecuta código en ellas, te recomiendo que tomes el curso Jupyter Notebook, Google Colab y Markdown para todos, disponible en Udemy en la siguiente liga:

https://www.udemy.com/course/jupyter-notebook-y-markdown-para-todos/

Plataforma de trabajo

Para editar las libretas de Jupyter y ejecutar el código que contienen, necesitas una plataforma que los soporte.

Sugerimos dos:

1. **Google Colab**, si dispones de una conexión a Internet.

   https://colab.research.google.com/

2. **Jupyter Notebook**, disponible con **Anaconda**, si no dispones de una conexión a Internet.

https://www.anaconda.com

Se recomienda ampliamente que al iniciar el curso ya hayas registrado una cuenta de **Google** en **Google Colab**, y que cuando estés estudiando el material de este libro, estés dentro de tu ambiente de **Google Colab**, listo para poner manos a la obra en los ejercicios.

Archivos de trabajo y material adicional

Los archivos complementarios a este libro se encuentran en el siguiente repositorio **GitHub**.

https://github.com/AprendaPracticando/AnaliticaPythonR1

La estructura de carpetas es la siguiente:

1. **`<main>`**: Contiene las libretas Jupyter (**`.ipynb`**) generales del curso, así como el folleto y el mapa del curso, en PDF.
2. **`data`**: Contiene los archivos de datos que se utilizan para los Demos y Ejercicios.
3. **`images`**: Contiene las imágenes de algunas porciones explicativas del contenido.
4. **`labs`**: Contiene los archivos base para la realización de los LABS contenidos en el curso.

Los archivos de datos son accesibles desde **GitHub**.

1. **`pasajeros_titanic.csv`**: está disponible mediante la liga:

https://raw.githubusercontent.com/AprendaPracticando/AnaliticaPythonR1/main/data/pasajeros_titanic.csv

2. **`clases.csv`**: Está disponible mediante la liga:

https://raw.githubusercontent.com/AprendaPracticando/AnaliticaPythonR1/main/data/clases.csv

LAB 00.01: Verificar el acceso a los datos de prueba

En este Lab se verifica la posibilidad de acceder a los datos de prueba que serán utilizados en el libro.

Lo recomendable es que hayas registrado una cuenta de **Google** para acceder a **Google Colab**, y que ingreses a **Google Colab** con tu cuenta. Esto no es requerido, pero es ampliamente recomendable.

Las tareas por realizar son:

1. Abrir una libreta de Jupyter que está en **GitHub**.
2. Abrir la libreta de Jupyter en **Google Colab**.
3. Ejecutar el código Python de la libreta Jupyter.
4. Guardar una copia tu propio **Google Colab**.

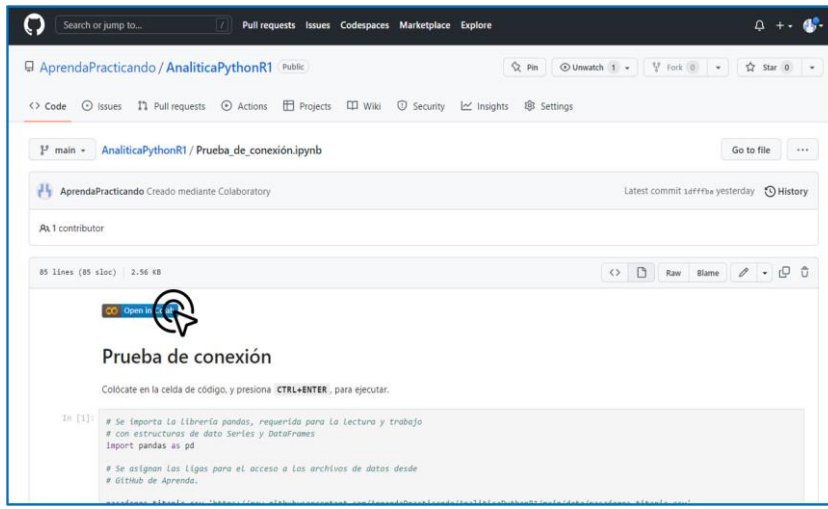
---

1. Abrir una libreta de Jupyter que está en **GitHub**.
    a. Ingresa al repositorio de GitHub donde están los archivos de trabajo del libro. La liga es esta:

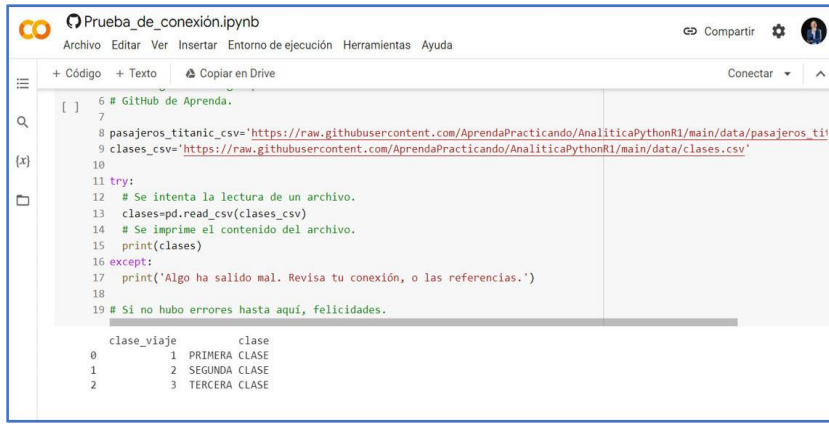
https://github.com/AprendaPracticando/AnaliticaPythonR1

2. Haz clic sobre el archivo **`prueba_conexión.ipynb`**, para abrirlo.

3. Abrir la libreta de Jupyter en **Google Colab**.
    a. Haz clic en el botón ***Open in Colab***, que aparece en la parte superior del código.
    b. Esto abrirá la libreta de Jupyter en **Google Colab**.



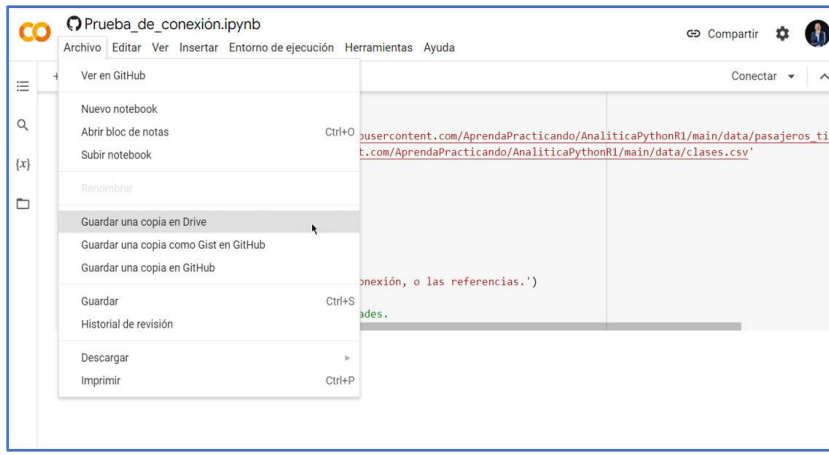
4. Ejecutar el código Python de la libreta Jupyter.
    a. Colócate en la celda de código Python de la libreta Jupyter.
    b. Presiona **CTRL+ENTER** para ejecutar el código.
    c. Si aparecen datos, ¡Listo! Todo ha transcurrido bien.



5. Guardar una copia tu propio **Google Colab**.

    a. En **Google Colab**, ve al menú ***Archivo***, y selecciona ***Guardar copia en Drive***; con esto, podrás guardar el archivo en tu ambiente de Google Colab.

    b. A partir de este momento, trabaja con tu propia libreta Jupyter, donde podrás hacer comentarios y anotaciones que encuentres pertinentes.



Esta mecánica se seguirá cada vez que utilices una libreta de Jupyter del repositorio **GitHub** donde están los archivos complementarios del libro.

Uso de los LABS

En el folder **`labs`** del repositorio **Github**, busca el archivo correspondiente al LAB que estés trabajando.

Su nombre será **`LAB_CA_SE.ext`**

Donde aparece <**CA**> vendrá el número del capítulo al que pertenece, mientras que donde aparece <**SE**>, aparecerá el número de LAB del capítulo.

En lugar de <**`.ext`**>, aparecerá la extensión del tipo de archivo que se trate: si es **`.pdf`**, quiere decir que el LAB no requiere ejecutar código; si es **`.ipynb`**, quiere decir que es una libreta Jupyter donde codificarás y ejecutarás líneas de código Python.

En este segundo caso: ***a)*** Abres la libreta Jupyter; ***b)*** Solicitas abrirlo en Colab (***Open in Colab***); ***c)*** Guarda tu propia copia de la libreta en **Colab**; ***d)*** Codifica lo que se pida en el LAB, por ti mismo; ***e)*** Comprueba que el resultado que obtienes es el mismo que se señala en el libro.

FIN DEL LAB

Capítulo 1:

Analítica de datos, datos e información

Conceptos

Dato

Definición formal de dato

Un ***dato*** es la representación simbólica, ya sea mediante números o letras, de una característica cualitativa o cuantitativa que facilita la representación abstracta de algo.

Ejemplos:

- El código de un producto.
- La fecha de una factura.
- La edad de una persona.

Los datos generalmente se almacenan en bases de datos, que los organizan de tal manera que su almacenamiento y recuperación sea eficiente.

Base de datos

Un ***base de datos*** es una "colección auto descriptiva de registros integrados" (Kroenke, 2018).

Al momento de identificar fuentes de datos, e integrarlas, deberás tener conocimiento de las bases de datos.

Metadatos

Los ***metadatos*** son datos de los datos, es decir, es información que describe los datos en cuanto a los datos, y no en cuanto a su contenido.

El metadato puede decir si el dato es numérico o textual, cuál es su longitud, si el dato tiene un dominio en particular, si tiene validaciones especiales, si puede ser nulo o no, si tiene exigencia de unicidad, etcétera.

A partir de las bases de datos, se espera poder obtener información.

Información

La ***información*** es el *a)* conjunto de datos con la *b)* cantidad y forma adecuada para *c)* aumentar el conocimiento o reducir la incertidumbre *d)* respecto a sujetos, eventos o estados, *e)* en una circunstancia o contexto particular.

Conjunto de datos

La analítica se realiza utilizando conjuntos de datos que provienen de múltiples fuentes, como pueden ser bases de datos, archivos de texto plano, hojas electrónicas, e incluso datos no estructurados, como podrían ser páginas Web, correos electrónicos, etcétera.

Principalmente los datos estarán en bases de datos de tipo SQL, es decir, basadas en el modelo relacional.

Cantidad y forma

La analítica culmina cuando los datos se representan usando la cantidad y forma adecuada.

- Por ***cantidad*** nos referimos al nivel de detalle en que se requiere ver la información.
- Por ***forma*** nos referimos a la forma de representación, principalmente tablas de datos, o gráficos.

Se puede obtener una tabla con el detalle de cientos de registros, una sola cifra, o un gráfico de barras, a partir de los mismos datos.

Aumenta el conocimiento o reduce la incertidumbre

La analítica de datos pretende aumentar el conocimiento o reducir la incertidumbre respecto a algo.

- Por ***aumentar el conocimiento*** nos referimos a que nos revela cosas que no sabemos, pero que necesitamos para la toma de decisiones.
- Por ***reducir la incertidumbre*** nos referimos a que reduce los riesgos derivados de la toma de una decisión, aumentando nuestro grado de certeza.

Sujetos, eventos y estados

La información puede ser relativa a sujetos, eventos o estados.

- Son ***sujetos*** quienes realizan cosas o a quienes les suceden las cosas. Ejemplos: empleados, productos, almacenes, colores.
  - En algunos casos, se consideran sujetos algunos atributos clasificados, siempre y cuando constituyan catálogos predefinidos de algo, en donde se tiene al menos una clave y una descripción.
  - Ejemplos: catálogo de colores, catálogo de presentaciones, catálogo de ciudades, catálogo de tamaños, etcétera.

- Son ***eventos*** aquello que realizan los sujetos, o lo que le sucede a los sujetos; los eventos existen en relación a sujetos, y generalmente contienen datos que refieren a los sujetos, y datos relativos a marcas de tiempo, lugar o secuencia.

  - Ejemplos: Entradas y salidas de almacén (relacionadas con algún producto); Operaciones bancarias (relacionadas con la cuenta de algún cliente).

- Los ***estados*** son la situación en que se encuentra algo o alguien; generalmente son características o campos de un sujeto o evento.
  - Ejemplos: Un empleado puede estar activo / inactivo; una salida de almacén puede estar autorizada / no autorizada.

Circunstancia o contexto particular

La analítica debe profundizar al caso específico, y no quedarse simplemente informando en términos generales.

La ***circunstancia o contexto*** implica los alcances geográficos, temporales, condicionales o de clasificación que delimitan a los sujetos, eventos y estados, y a su análisis.

La analítica debe proveer controles que permitan cambiar en tiempo real los parámetros del análisis, permitiendo filtros y especificaciones varias.

- No: Informe general de ventas. (Estático).
- Sí: Informe de ventas de químicos de limpieza vendidos en tiendas de abarrotes minoristas, del 01/01/2023, al 15/01/2023. (Con posibilidades de cambiar los parámetros del informe, en tiempo real).

Analítica de datos

La ***analítica de datos*** es el conjunto de tareas que permiten *a)* recolectar datos existentes, y aplicar sobre ellos *b)* técnicas estadísticas y *c)* de modelación *d)* con el fin de limpiarlos, *e)* transformarlos, *f)* y representarlos de manera abstracta y gráfica, *g)* con la intención de revelar estados y tendencias de los hechos *h)* relativos a un tema o negocio.

Recolectar datos existentes

La analítica de datos inicia recolectando datos existentes que están almacenados en alguna fuente actualmente.

El analista de datos requiere un alto conocimiento de la organización, de la ubicación de los datos, de la semántica de los datos, y su disponibilidad.

Técnicas estadísticas

El analista de datos debe conocer técnicas de estadística descriptiva, principalmente lo relacionado con ***EDA*** (***Exploratory Data Analysis***), que son un conjunto de técnicas utilizadas en el campo de la estadística y la ciencia de datos para examinar y resumir las características principales de un conjunto de datos.

El objetivo principal del EDA es ayudar a los analistas de datos, mediante el uso de técnicas de estadística descriptiva, a comprender mejor los patrones y relaciones en los datos y a identificar cualquier posible problema o anomalía en los mismos, antes de proseguir con técnicas más avanzadas. EDA se utiliza comúnmente en las primeras etapas del análisis de datos para obtener una visión general de los datos y guiar el proceso de análisis.

Entre las técnicas comunes de EDA se incluyen la visualización de datos, la exploración de la distribución de datos, la identificación de valores atípicos y la evaluación de la correlación entre diferentes variables. Al realizar un EDA efectivo, los analistas de datos pueden identificar patrones y tendencias interesantes en los datos, y utilizar esta información para tomar decisiones informadas en el análisis posterior. EDA incluso ayuda a formular los objetivos y las hipótesis de investigación.

Técnicas de modelación

El analista de datos debe tener conocimientos de modelación de bases de datos relacionales, debido a que probablemente deba integrar datos de diferentes fuentes, de una manera estructurada.

Debe saber representar una situación del mundo real en términos de tablas, campos, llaves primarias y foráneas, relaciones entre tablas, cardinalidades, etcétera.

Limpieza de datos

La analítica de datos incluye tareas de limpieza de datos, es decir, corregir datos erróneos, excluir datos incompletos, comprobación de fórmulas, eliminación de duplicados, verificación de unicidad de datos, así como el tratamiento de datos ausentes y atípicos.

El analista de datos debe conocer herramientas que permitan hacer limpieza de datos (**Excel**, **Power BI**, **Tableau**), o lenguajes de programación que lo permitan (**Python**, **R**).

Transformación de datos

La analítica de datos incluye tareas de transformación de datos, es decir, modificar datos correctos en datos equivalentes, con el fin de que sea más sencillo y efectivo su procesamiento.

Transformaciones típicas pueden ser estandarizar el nombre de tablas y campos, calcular campos a partir de los campos existentes, realizar conversiones y estandarizaciones de unidades de medición, generar campos derivados, y habilitar equivalencias categóricas.

El analista de datos debe tener sentido común, respeto a los protocolos, habilidades numéricas y pensamiento lógico y funcional.

Representación abstracta y gráfica

La analítica de datos incluye tareas de representación abstracta y gráfica:

- Por ***representación abstracta*** nos referimos a las cifras, las tablas de datos, y los KPI expresados en forma numérica.
- Por ***representación gráfica***, nos referimos a gráficos estándar o de diseño, que ilustran estados y tendencias.

El analista de datos debe saber EDA y graficación; es deseable preparación especializada en visualización y diseño de informes.

Revelación de estados y tendencias

La analítica de datos revela estados y tendencias:

1. Por ***estados*** nos referimos a que revela la situación en que se encuentra algo, en un momento determinado.
2. Por ***tendencia***, nos referimos a que revela la tendencia de los hechos en el transcurso del tiempo.

El analista de datos debe tener pertinencia operativa, es decir, conocer qué información de estados y tendencias es requerido para la toma de decisiones, para lo cual, debe tener pensamiento sistémico y orientado a procesos.

Es importante anotar que la analítica no pronostica ni estima cómo serán las cosas en el futuro, ni cómo el pasado se vería afectado con la variación de parámetros hipotéticos. Escapa de su alcance.

Enfoque a tema o negocio

El analista de datos debe tener profundo conocimiento del tema o negocio del cual tratan los datos.

Esto le permitirá hacer análisis semánticos de los datos, determinar qué es más útil, y qué herramientas de visualizaciones son las más apropiadas.

El trabajo de la analítica de datos nunca es teórico o abstracto: atiende un contexto en el cual se tiene que obtener información para tomar decisiones, comprobar hipótesis, o hacer conjeturas.

Tipos de análisis

En analítica de datos se pueden hacer dos tipos de estudio:

- ***Análisis de descubrimiento*** **(*discovery analysis*)**: Consiste en un trabajo de análisis que no persigue resolver incógnitas conocidas, y requiere una inspección general de los datos que puedan sugerir líneas de investigación posteriores. Se aplican técnicas de análisis exploratorio de datos (EDA / *Exploratory Data Analysis*), así como análisis estadísticos que pudieran sugerir patrones y anomalías.

- ***Análisis basado en objetivo*** **(*goal based analysis*)**: Consiste en un trabajo de análisis que persigue resolver incógnitas específicas y conocidas, que son descritas con claridad. El trabajo de análisis de datos es parecido a un análisis de descubrimiento, con la diferencia de que se aplican técnicas adicionales y atención especial a la solución de los problemas planteados.

Reconocer el tipo de análisis que se pretende realizar determina el proceso de investigación que se ejecutará. Las técnicas que se utilizan cuando no se sabe lo que se anda buscando (técnicas generales), son muy diferentes a cuando sí se sabe lo que se quiere buscar (técnicas particulares).

Es importante saber qué tipo de análisis se realizará, ya que la forma de abordarlos es muy diferente. Esto no se sabe sino hasta que se conoce el caso de estudio.

Técnicas para definir el tipo de objetivo

El tipo de análisis que ha de realizarse, y sus alcances, no es algo que flote en el éter y se defina por sí mismo. Es necesario que el equipo de analistas se reúna con el cliente de la información con la finalidad de definir el resultado que se persigue.

Se recomienda una o varias reuniones entre el analista y el cliente de la información, de preferencia cara a cara. Cada reunión es crucial para establecer una comprensión compartida de lo que se espera que se logre con los trabajos de analítica, y para garantizar que el equipo de analistas tenga una dirección clara para su trabajo.

Se sugieren varias técnicas de comunicación para facilitar la colaboración efectiva entre el equipo de analistas y el cliente. Algunas de estas técnicas son:

1. ***Brainstorming*****:** Los miembros del equipo de analistas y el cliente pueden generar ideas y soluciones creativas para los objetivos y elementos de trabajo de analítica a través de una lluvia de ideas estructurada.

2. ***Mapas mentales*:** Los mapas mentales pueden utilizarse para visualizar las relaciones entre los elementos de trabajo de analítica y los objetivos a largo plazo del proyecto, lo que puede ayudar a identificar oportunidades y riesgos.

3. ***Discusión en grupo*:** Las discusiones en grupo pueden ayudar a garantizar que todos los miembros del equipo de analistas y el cliente tengan una comprensión clara de los objetivos y los elementos de trabajo de analítica, y pueden permitir que los miembros del equipo de analistas hagan preguntas y aclaren cualquier ambigüedad.

4. ***Prototipos*:** Si es posible, se pueden utilizar prototipos para demostrar cómo se verá y funcionará el trabajo completado al final del análisis (puede ser un modelo de informe, o un modelo de tablero de datos). Esto puede ayudar a garantizar que todos los miembros del equipo de analistas y el cliente tengan una comprensión clara del producto final esperado y pueden permitir ajustes tempranos en el proceso de analítica si es necesario.

Analítica de datos, Big Data y Data Science

Suelen confundirse mucho a la analítica de datos con Big Data y Data Science. En realidad, son cosas muy diferentes.

Como ya vimos, la analítica de datos tiene que ver con el entendimiento de los datos usando estadística descriptiva, modelación, técnicas de análisis de datos y visualización.

Por otro lado, ***Big Data*** tiene que ver con técnicas de almacenamiento, recuperación y transporte de bases de datos SQL y NO SQL, distribuidas generalmente de forma global, estructuradas y no estructuradas, incapaces de ser manejadas de forma integrada y en una sola infraestructura de hardware habitual.

Piensa en el tipo de bases de datos que son utilizadas para almacenar toda la información de redes sociales como Facebook, Instagram o Twitter: Hay texto, hay audio, hay video, hay ligas, recordatorios, esquemas de

operación y permisos, entre muchas cosas más, y, sobre todo, hay una cantidad de información tan grande y global, que sería imposible manejarla en un servidor o conjunto de servidores de manera eficiente.

Pues bien, Big Data se encarga de estructurar el almacenamiento y la recuperación de esos datos inmanejables de otra manera. Como puedes apreciar, la analítica de datos utiliza Big Data, pero no son lo mismo. Por ejemplo, la información de Google sin duda está bajo un esquema de Big Data, por la cantidad de información y lo complejo de su distribución global; por otro lado, Google Analytics es analítica de datos: extrae la información de las bases de datos de Google, que están en un esquema de Big Data, pero una vez recuperada la información, Google Analytics se encarga del uso de técnicas de estadística descriptiva y visualización para presentar los resultados que ayuden al tomador de decisiones. Al tomador de decisiones, Big Data no le aporta valor, y no le interesa cómo funciona. Lo que le interesa es lo que le sirve, que es la analítica de datos.

Por otro lado, la ***ciencia de datos*** o ***Data Science***, tiene que ver con la generación de modelos inferenciales e interpretativos, manuales y automatizados, para la toma de decisiones y operacionalización de actividades. Incluye otras ramas especializadas, como ***Machine Learning***, ***Deep Learning***, ***Inteligencia Artificial***, entre otras cosas.

Como recuerdas, la analítica de datos trabaja con datos e información existente, lo que la limita a la estadística descriptiva; Data Science comparte con la analítica de datos el uso de base de datos, la estadística descriptiva y las técnicas de visualización, pero va mucho más allá, pues requiere conocimientos de programación, estadística inferencial (intervalos de confianza, pruebas de hipótesis, análisis de varianza, regresiones, correlaciones) métodos de clasificación (análisis discriminante, árboles de decisión, análisis de conglomerados, redes neuronales), procesamiento del lenguaje natural, entre otras cosas.

Para hacer analítica de datos, basta con que conozcas de base de datos, estadística descriptiva, y mucho conocimiento del negocio; para Data Science, ocupas eso, y mucho conocimiento de estadísticas y matemáticas adicional.

Dicho de otra manera, la analítica de datos termina donde los modelos se comienzan a convertir en inferenciales. La ciencia de datos ya incluye muchas matemáticas aplicadas.

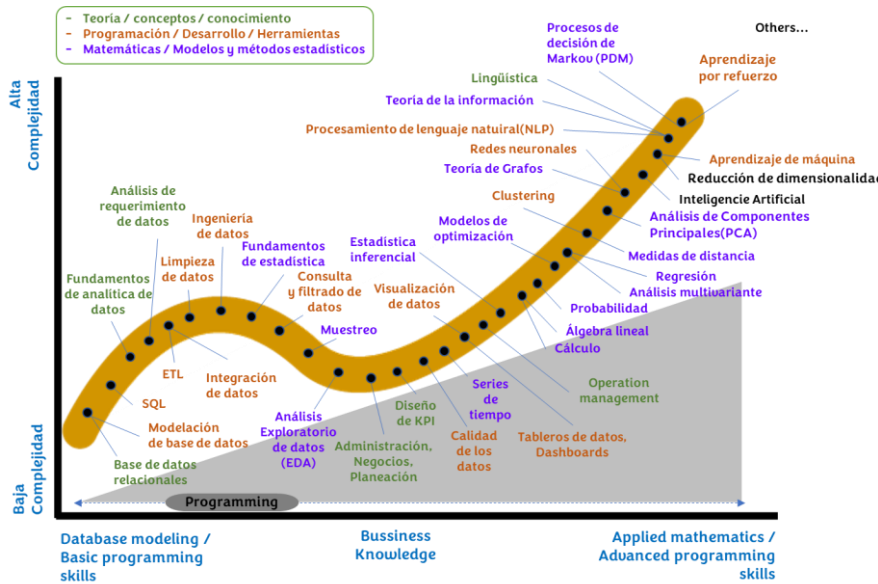


FIGURA 01.01: *Ruta de aprendizaje para la ciencia de datos.*

Este libro se mantiene en el ámbito de la analítica de datos. Big Data y Data Science quedan fuera de su alcance.

Capítulo 2:
ETL (extract, transform, load)

El modelo ***ETL*** (***Extract, Transform, Load*** o ***Extraer, Transformar , Cargar***) se refiere a un proceso utilizado para realizar tareas de integración de datos, misma que consiste en tres fases o momentos: *a)* extraer datos de diferentes fuentes, *b)* transformarlos según las necesidades específicas del usuario, y *c)* cargarlos en un sistema de destino.

Para entrar al análisis de ETL, lo primero que debemos saber es que en las organizaciones se dispone de diferentes ambientes, y que ETL es el proceso que permite llevar datos de uno a otro.

Ambientes

El ***ambiente productivo*** es el conjunto de aplicaciones y datos con los que las organizaciones operan en tiempo real, con la finalidad de registrar sus operaciones y transacciones.

Piensa en un supermercado: cuando vas a pagar productos en una caja, el sistema de *retail* que usa el cajero es un sistema en *producción*, y las bases de datos que se actualizan en cada caja, en cada operación que realiza, también es una ***base de datos en producción***.

La prioridad del ambiente productivo es *registrar* transacciones, y recuperar con rapidez los datos que se requieren para poder hacerlo. Este enfoque hace que los sistemas en producción se sean llamados también ***OLTP*** / ***on line transaction processing***.

Por otro lado, están los ***ambientes no productivos*** son el conjunto de aplicaciones y datos que replican total o parcialmente al ambiente productivo, para poder realizar otros fines distintos al registro de transacciones, como puede ser procesar datos para la generación de informes o desarrollar y modificar aplicaciones.

Hay dos ambientes no productivos que son muy usuales:

1. ***Ambiente de desarrollo*:** Es el ambiente no productivo sobre el cual los desarrolladores modifican y crean aplicaciones, afectando datos hipotéticos, o bien, un subconjunto reducido de datos extraídos de las bases de datos de producción, pertinentes para un aplicativo en particular.

2. ***Ambiente de preproducción*:** Es el ambiente no productivo en donde las aplicaciones próximas a liberarse se prueban con una copia de las bases de datos de producción, similares en volumen, para probar el desempeño con una carga de datos cercana a la real.

La idea de separar los ambientes es que no interfieran con la operación diaria de las organizaciones, y evitar riesgos de pérdida de datos. Por ejemplo, si una organización no tiene un ambiente de desarrollo, entonces los programadores deberán probar las aplicaciones que desarrollen usando los datos reales de la operación. Si por error llega a borrar o corromper información, de las ventas, de los clientes, de los movimientos contables, puede ser catastrófico.

Por otro lado, si no se tiene un ambiente de preproducción, podemos tener aplicativos que en el ambiente de desarrollo funcionan de maravilla, con pocos datos, pero al ejecutarlos con las bases de datos reales provocan problemas o colapsan.

Repositorios de datos

Almacén de datos (Data Warehouse)

Para hacer trabajos de analítica, se diseña un ***almacén de datos***, conocido también como ***data warehouse***, que es una base de datos en ambiente no productivo, diseñada específicamente para almacenar grandes cantidades de datos integrados de múltiples fuentes, y facilitar su procesamiento en tareas de cálculo y consolidación.

Este *data warehouse* es utilizado para soportar aplicaciones de análisis y consultas de datos complejas para la toma de decisiones empresariales, y pueden incluso tener una estructura bastante distinta a la que tienen las fuentes originales de las que se toman los datos.

Esto quiere decir que los datos en un *data warehouse* son organizados de forma estructurada y optimizados para la consulta y el análisis de grandes volúmenes de información, y no para el registro de transacciones. También suelen estar *preprocesados*, es decir, incluyen información redundante con resultados de cálculos frecuentes, lo que permite reducir el tiempo de procesamiento de consultas, cálculos y análisis.

Herramientas como **Power BI** o **Tableau**, e incluso ***Excel*** con **Power Query**, se enlazan al *data warehouse* para encargarse de tareas de visualización y procesamiento.

Master de datos

No siempre el destino de ETL es un *data warehousing* con una estructura definida de datos. A veces no tenemos acceso a las fuentes, ni a conocer la estructura de las bases de datos desde donde tomamos información. En algunos casos es hasta secreto industrial.

Cuando eso sucede, lo que nos es entregado es un archivo con la información que necesitamos, y nada más. Esto es común cuando la información la obtenemos de algún sistema que tiene módulos para la descarga de archivos, por ejemplo, **SAP**, o bien cuando nos conectamos a servicios de información, como podrían ser API's de fuentes de datos.

En ese caso, de lo que disponemos es de un *master de datos*.

Un ***master de datos*** es un arreglo bidimensional de datos que contiene información detallada de las tablas fuertes y débiles de un modelo de datos, anulando la necesidad de disponer de un motor que se encargue de mantener la coherencia de la base de datos mediante el uso de relaciones entre tablas y el conocimiento de llaves primarias y foráneas.

El master de datos se construye a partir de las tablas de mayor detalle dentro de un modelo.

En cierta manera, se trata de una ***desnormalización de datos***, porque en lugar de separar tablas y relacionarlas para evitar la redundancia, junta las tablas, para que no sea necesario establecer relaciones.

Analiza este modelo de datos:

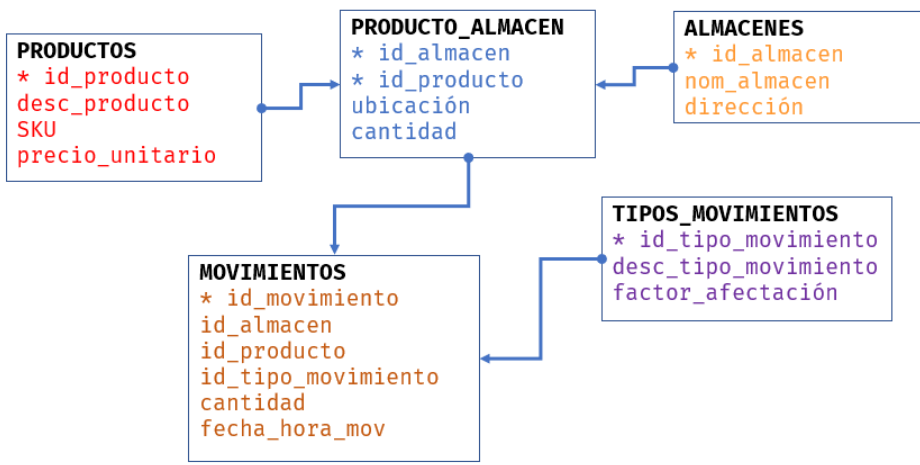


FIGURA 02.01: *Diagrama DER de un control de almacenes.*

La tabla más detallada es **`MOVIMIENTOS`**: Es tabla débil para **`PRODUCTOS_ALMACEN`** y **`TIPOS_MOVIMIENTO`**, y no es tabla fuerte de nadie.

La segunda tabla más detallada es **`PRODUCTO_ALMACEN`**, que es tabla débil para **`PRODUCTOS`** y **`ALMACENES`**, y es tabla fuerte para **`MOVIMIENTOS`**.

1. Primero, se integran los campos de la tabla **`PRODUCTO_ALMACEN`**, con los campos de sus tablas fuertes.

```
id_almacen + id_producto + ubicación + cantidad +
nom_almacen + dirección + desc_producto + SKU +
precio_unitario
```

2. Luego, se integran los campos de la tabla **`MOVIMIENTOS`**, con los campos de sus tablas fuertes. Los atributos de coincidencia persisten los de la tabla débil.

```
id_movimiento + id_almacen + id_producto +
id_tipo_movimiento +  cantidad + fecha_hora_mov +
ubicación + cantidad + nom_almacen + dirección +
desc_producto + SKU + precio_unitario +
desc_tipo_movimiento + factor_afectación
```

3. Si disponemos de este master de datos, no necesitamos conocer la estructura de la base de datos, ni las relaciones entre tablas ni la información de las llaves: todo lo que ocupamos, está ahí.

Ya sea que se utilice un data warehouse o un master de datos, al final terminas con algo parecido a un master de datos. En un data warehouse, al final, se termina haciendo consultas (*querys*) que integran la información de múltiples columnas en un solo arreglo bidimensional de datos, que es equivalente a estar trabajando con un master de datos.

Dicho de otra manera, usar un data warehouse te da la posibilidad de generar múltiples masters de datos a partir de una estructura de base de datos preparado para la analítica, pero siempre terminas trabajando con el equivalente a un master de datos.

La ruta de ETL

En un escenario real, los datos de las organizaciones están en diferentes partes: sistemas transaccionales (OLTP), archivos de Excel, archivos de texto plano, archivos serializados (JSON/XML), bases de datos, fuentes de datos en tiempo real, entre muchos otros.

El modelo ETL se encarga de tomar los datos de todas las múltiples fuentes, y depositarlos en un *data warehouse* o en un *master de datos*.

1. La extracción implica obtener los datos de una fuente determinada.

2. La transformación se refiere a la modificación de los datos para que cumplan con los requisitos del repositorio destino (*data warehouse* o *master de datos*).

3. La carga consiste en insertar los datos transformados en el repositorio destino.

El proceso ETL es importante porque ayuda a garantizar que los datos sean precisos, consistentes y estén disponibles para su uso en diferentes aplicaciones y sistemas empresariales. Además, permite a los usuarios tener una vista completa y unificada de los datos de la empresa, lo que facilita la toma de decisiones basadas en información precisa y actualizada.

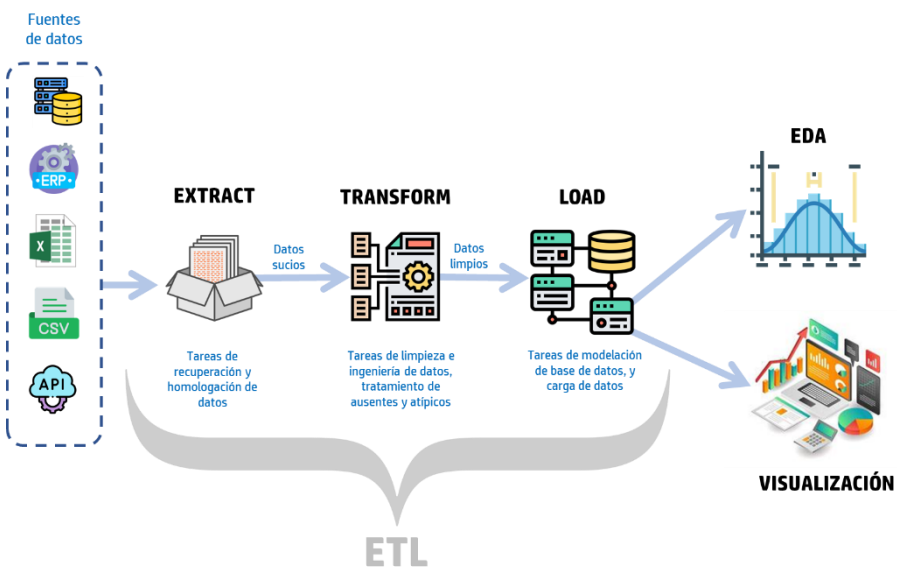


FIGURA 02.02: *Modelo ETL (Extract – Transform – Load)*

Fase de extracción (Extract)

La fase de ***extracción*** en un proceso ETL consiste en la recolección de datos de diversas fuentes de datos, como bases de datos, archivos planos, sistemas de información en línea, aplicaciones y sensores asociados con dispositivos de control numérico (CNC), entre otros. Se ocupa de la toma de datos desde diferentes fuentes, y cumplir con los requisitos para que eso suceda.

El proceso de extracción generalmente implica la selección y filtrado de los datos, la validación y verificación de estos, desde un punto de vista estructural, y la extracción de los datos desde las fuentes de origen.

Aunque muchos estudiosos hacen énfasis en las cuestiones técnicas, la verdad es que la fase de extracción tiene que ver incluso con saber *qué* es lo que se tiene que extraer.

Algunas de las tareas a realizar en la fase de extracción son:

1. ***Análisis del caso***: Consiste en analizar un caso real, entender el contexto y delimitar las necesidades de información, y la enumeración de datos requeridos.
2. ***Análisis de fuentes***: Consiste en identificar las fuentes de datos desde las cuales los datos pueden ser tomados. Aquí se verifica la disponibilidad de los datos requeridos, en sistemas, en archivos, en fuentes de terceros, así como el formato en que se encuentran.
3. ***Análisis de permisos requeridos***: Consiste en identificar el tipo de permisos, accesos y privilegios que se necesitan para acceder a los datos. Aquí se especifica el alcance de los permisos (base de datos, tabla, objeto), y los privilegios (lectura, procesamiento, y rara vez, escritura).
4. ***Gestión de permisos***: Consiste en gestionar la obtención de los datos de autenticación y autorización para poder acceder a los datos, con la periodicidad que se requiere. Se pueden gestionar permisos permanentes, por periodo (semanal, mensual, semestral), o por evento (cada que se acceda a los datos).

5. ***Automatización***: Consiste en disponer o preparar las herramientas tecnológicas para poder leer los datos de la fuente, e integrarlos de manera que puedan ser transformados, de forma periódica y automática. Aquí es donde se pueden elaborar programas en Python que realicen la tarea de extraer los datos de la manera que se necesita, con la mínima intervención humana.

La fase de extracción es un paso crítico en el proceso ETL, ya que, si los datos se extraen de manera incorrecta, o si se extraen datos que no se ocupan y se dejan fuera datos que sí se ocupaban, se pueden generar problemas en las fases posteriores de transformación y carga.

Es importante asegurarse de que los datos extraídos sean precisos, completos y coherentes con las expectativas de la organización y los requisitos del proyecto de análisis.

Fase de transformación (Transform)

La fase de ***transformación*** de ETL toma como materia prima los datos extraídos en la fase de extracción.

En esta fase se aplican una serie de transformaciones y tareas de limpieza de datos a los datos extraídos de diferentes fuentes, antes de cargarlos en el *data warehouse* o el master.

Durante esta fase, los datos extraídos se procesan para asegurar que sean coherentes y precisos, y para que se adapten al esquema de datos del sistema de destino.

Algunas de las transformaciones comunes que se aplican durante esta fase incluyen:

- ***Limpieza de datos***: Consiste en eliminar los datos duplicados, los valores nulos, las entradas no válidas; aquí se revisa la unicidad de ciertos campos y se resuelven otros problemas de calidad de datos como podrían ser cálculos erróneos o incompletos.

- ***Conversión de datos***: Consiste en convertir los formatos de datos para que sean compatibles entre sí, y con el repositorio destino. Por ejemplo, se pueden convertir las fechas a un formato común, homologar tipos de cambio o unidades de medida.

- ***Derivación de datos***: Consiste en crear nuevos datos a partir de los datos existentes. Estos cálculos pueden ser ***cálculos detallados*** (a nivel fila), como por ejemplo calcular el monto total de una operación a partir de la cantidad y el precio unitario; o pueden ser ***cálculos agregados*** (conjunto de filas), como por ejemplo calcular el costo total de la nómina por departamento, o por dirección, a partir de la plantilla de empleados.

- ***Agregación de datos***: Consiste en agrupar los datos y realizar cálculos de resumen, como sumar los ingresos por región, por mes, por la línea de producto, etcétera. Esto aplica generalmente cuando el destino es un *data warehouse*, y se tiene un esquema de tablas multidimensionales.

- ***Normalización de datos***: Consiste en reorganizar los datos para cumplir con los requisitos de esquema de datos del repositorio destino. En el caso de un *data warehouse*, generalmente implica normalizar; en el caso de un master de datos, generalmente implica desnormalizar.

- ***Enriquecimiento de datos***: Se agregan datos adicionales de fuentes externas para mejorar la calidad de los datos existentes. Aquí pueden incluirse datos de fuentes de paga, que aportan información que de otra manera sería imposible recopilar. También pueden incluirse servicios de información gratuita, como **forex**, que permite obtener los tipos de cambio actualizados al día, a nivel mundial y sin costo.

La fase de transformación en ETL es fundamental para garantizar la calidad y la coherencia de los datos antes de cargarlos en el repositorio destino, lo que permite un mejor análisis y trabajo de visualización, y con ello, de toma de decisiones.

Fase de carga (Load)

La fase de ***carga*** en ETL es el proceso en el que los datos transformados y limpios se cargan en el repositorio destino, que puede ser un *data warehouse* o un *master*.

En esta fase, se llevan a cabo las siguientes acciones.

- ***Carga de datos*:** Se cargan los datos transformados en las tablas de destino.
- ***Validación de datos*:** Se validan los datos cargados para asegurar que se carguen correctamente y se correspondan con los datos originales.
- ***Generación de registros de auditoría*:** Se generan registros de auditoría para rastrear el proceso de carga de datos y detectar cualquier problema o error.

En el caso de que el destino sea un *data warehouse* se requieren acciones adicionales, al estar involucrado un modelo de datos relacional:

- ***Mapeo de datos*:** Se mapean los datos transformados en la estructura de destino para asegurar que los datos se carguen correctamente.
- ***Creación de tablas*:** Si las tablas de destino no existen, se crean en el sistema de destino.
- ***Creación de índices y restricciones*:** Se crean índices y restricciones en las tablas de destino para optimizar la consulta de datos y asegurar la integridad de los datos.

Tanto para la fase de Extract como para Load, el conocimiento de la técnica de modelación de bases de datos relacionales es indispensable. Recomendamos que revises nuestro libro ***Modelación de Base de Datos Relacionales y su implementación en SQL Estándar (Ramírez, 2020)***. Lo puedes descargar de manera gratuita desde http://www.aprenda.mx

La fase de carga en ETL es el proceso final en el que se cargan los datos transformados en el repositorio destino. Esta fase es importante para asegurar que los datos estén disponibles y sean utilizables para la toma de decisiones y el análisis de datos.

Al concluir esta fase, termina ETL, y se comienzan los trabajos de analítica de datos, generalmente EDA y Visualización.

CAPÍTULO 3:
Análisis semántico del caso

El análisis semántico del caso implica seguir una secuencia de tareas técnicas que de manera sistemática nos permita entender el caso, y establecer un objetivo de análisis de datos, a partir de una situación del mundo real, y una necesidad especifica de información.

Una secuencia que ayuda a obtener todos los entregables relacionados en la figura anterior es la siguiente:

1. ***Inmersión.*** Consiste en familiarizarse con la industria, el tema, la organización, las personas, y los conceptos claves que son del interés de quien será el cliente, entendiendo por cliente a la persona u organización que recibirá los beneficios de los trabajos de analítica. Aquí es muy recomendable usar técnicas de observación informal.
    a. ***Reunión inicial***. Consiste en celebrar una reunión inicial, en la cual se logre identificar el problema o la oportunidad de negocio que desea abordar. Esto puede incluir identificar un problema existente que necesita ser resuelto o una oportunidad que desea aprovechar.
2. ***Establecer los objetivos de análisis***: Consiste en establecer los objetivos específicos que desea lograr a través de la analítica de datos. Aquí es muy conveniente determinar qué tipo de decisiones se quieren tomar a partir de la información, y analizar qué datos se requieren para poder tomarlas; a esto se le conoce como *determinar las necesidades de información*.
3. ***Identificar posibles soluciones***: con una comprensión más profunda del problema o la oportunidad y del mercado, se genera una lista de posibles soluciones o estrategias que podrían abordar el

problema o aprovechar la oportunidad, desde el punto de vista de los datos y la información involucrada.

4. ***Elaborar el documento de entendimiento***: Consiste en elaborar un documento que, de forma narrativa, describa lo que se entendió, tanto del problema y la oportunidad, como del alcance que se espera de los trabajos de analítica en general.

Más allá de los documentos que se elaboren, debemos pasar de lo abstracto a lo técnico: pasar de a) la narrativa del caso, a la b) identificación de variables involucradas en c) objetivos de análisis establecidos para d) confirmar o rechazar hipótesis que formulamos a partir de e) supuestos informales y creencias que hay en la organización.

De un supuesto como "mejoraron las ventas con la publicidad" (suposición), pasamos a "se asume que el monto invertido en publicidad tiene una correlación positiva y significativa con el monto de las ventas, de tal manera que cada peso gastado en publicidad, genere al menos $4,850 pesos de ganancia adicional" (hipótesis demostrable, con variables específicas).

FIGURA 03.01: *Secuencia de análisis semántico. Del análisis del caso, a las variables.*

Inmersión

La ***inmersión*** (***onboarding***) es la parte del proceso en la cual entramos en contacto con el tema que se desea analizar, y con el cliente de la información.

En la inmersión del caso se identifica el sector (industrial, de servicios, financiero, etc.) y la rama de conocimiento involucrada (finanzas, logística, sistemas de información, etcétera).

Una forma de realizar la inmersión sería seguir los siguientes pasos.

1. Antes de la entrevista inicial con el cliente:
    a. Obtener información del cliente, y determinar la industria o ramo al que pertenece.
    b. Reconocer la terminología usual o general de la industria o ramo al que pertenece el cliente.
    c. Reconocer a qué se dedica el cliente, saber qué hace, qué produce, qué vende, qué valor entrega.

2. Durante la entrevista inicial:
    a. Preguntar por los problemas y oportunidades que se desean atender con los trabajos de analítica.
    b. Preguntar por los tiempos disponibles para los trabajos de analítica.
    c. Preguntar por las restricciones de herramientas, o herramientas disponibles, o herramientas de uso oficial.

3. Posterior a la entrevista inicial, como compromiso de inmersión, acotado al problema u oportunidad que se desea atender:
    a. Solicitar y estudiar el organigrama y cadena de mando, manuales de políticas, procedimientos, y procesos aplicables.
    b. Solicitar y estudiar los papeles de trabajo que se utilizan en la operación aplicable.
    c. Conocer los sistemas y la infraestructura de almacenamiento de datos que tiene el cliente, aplicable al tema.

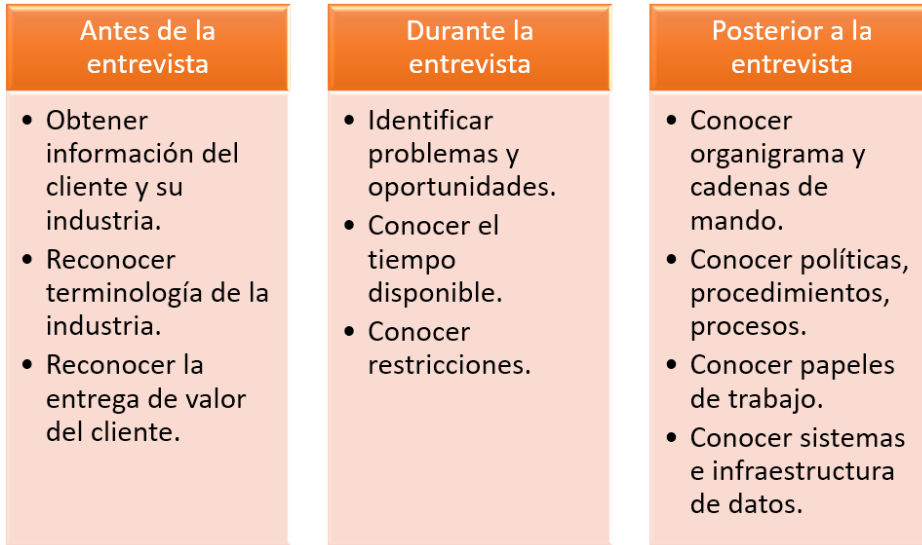


FIGURA 03.02: *Momentos y tareas básicas de inmersión.*

De lo que se trata es ponernos en una posición que nos permita conocer lo más posible el contexto en el que se desee desarrollar los trabajos de analítica, y entregar un resultado bien informado.

No toda la información mencionada estará disponible. Es sólo una referencia: entre más se conozca el terreno donde debemos trabajar, es mejor.

LAB 03.01: Analizar el contexto del caso

En este Lab se analiza el contexto del caso, tratando de entender cuál es el tema y sus alcances.

Las tareas por realizar son:

1. Entender de qué se trata el tema.
2. Suponer las probables fuentes de datos.

Entender de qué se trata el tema.

El **RMS Titanic** fue un transatlántico británico, el mayor barco de pasajeros del mundo al finalizar su construcción, que se hundió durante la noche del 14 y la madrugada del 15 de abril de 1912 durante su viaje inaugural desde Southampton a Nueva York.

En el hundimiento del Titanic murieron 1,496 personas de las 2,224 que iban a bordo, lo que convierte a esta catástrofe en uno de los mayores naufragios de la historia ocurridos en tiempos de paz.

El naufragio del Titanic conmocionó e indignó al mundo entero por el elevado número de víctimas mortales y por los errores cometidos en el accidente. Las investigaciones públicas realizadas en Reino Unido y los Estados Unidos llevaron a la implementación de importantes mejoras en la seguridad marítima y a la creación en 1914 del *Convenio Internacional para la Seguridad de la Vida Humana en el Mar* (SOLAS, por sus siglas en inglés), que todavía hoy rige la seguridad marítima.

Se ha podido documentar algo de información relacionado con la tragedia. La enciclopedia británica, una de las más completas al respecto, detalla información de 1,310 pasajeros, lo que permite hacer un poco de trabajos de analítica.

Una de las fuentes más confiables de datos respecto al tema es la Encyclopedia Britannica.

Suponer las probables fuentes de datos.

Si se deseas obtener el origen de datos, pueden obtenerse aquí: https://data.world/nrippner/titanic-disaster-dataset (Fuente: Encyclopedia Britannica, Marzo 3, 2023).

FIN DEL LAB

Observación informal

La ***observación informal*** es la parte del proceso donde el analista comienza a pronunciarse, junto con el cliente de la información, respecto a los datos. En esta fase se comienzan a revisar supuestos, desde una perspectiva no técnica.

Las personas no partimos de la nada al hacer analítica de datos: algo buscamos. Ese *algo* está representado por incógnitas que nos surgen, o detalles que nos llaman la atención y de los cuales queremos saber más.

La primera aproximación al análisis es una *observación coloquial*, entendiendo por esta, el entendimiento natural que se tiene de la situación, sin meternos en detalles técnicos. Aquí se enumeran creencias y conclusiones que en apariencia tienen sentido o lógica.

Una técnica muy útil en esta fase es la *lluvia de ideas*, es decir, dejar que los interesados e involucrados expresen sus ideas y opiniones respecto a un tema. No se debe juzgar ninguna idea o supuesto, no se debe inhibir la manifestación de ideas e inquietudes, dado que lo que se busca es conocer el contexto. Ya más adelante, las ideas que no son valiosas se irán descartando mediante el uso de la técnica y la razón.

Una ***observación coloquial***, llamado también ***supuesto***, es un juicio de valor o comentario que una persona emite sin que le conste que sea correcto o no. Dentro de los elementos que puede tener una observación coloquial, están los sujetos (quienes hacen cosas, a quienes les suceden cosas), y una afirmación respecto a las cosas como son, o acciones que pasan, o una mezcla de todo ello.

Ejemplos de supuestos son:

1. El peso tiene buen tipo de cambio gracias al ahorro de las personas.
2. Las nuevas generaciones prefieren el jabón líquido que el jabón de barra.
3. Las ventas de la línea de productos "químicos" está al alza, gracias a la pandemia.

Como ves, cada que opinamos o creemos algo, estamos ante una observación coloquial. En SCRUM, por ejemplo, se tienen técnicas para sacar a relucir las observaciones coloquiales, y dirigir los esfuerzos de desarrollo de aplicaciones.

La gente no suele emitir observaciones si se le pide que las emita. Como analista debes ser hábil para generarlas. En SCRUM, se le pide a la gente "Cuéntame cómo es un día común en tu trabajo". A la gente, más que dar información, le gusta contar historias.

De esas historias se pueden generar muchas preguntas, y muchas observaciones coloquiales.

LAB 03.02: Identificar los supuestos del caso

En este Lab se analiza el contexto del caso, y a partir de ahí, se generarán algunos supuestos o afirmaciones que deberán ser demostrada de manera técnica a través de técnicas de analítica.

Las tareas por realizar son:

1. Generar supuestos a partir del contexto.

Todos o casi todos vimos la película del TITANIC. Tomando en cuenta el entendimiento generalizado del tema que nos proporciona la película, podemos llegar a los siguientes supuestos.

1. Parecería que la gente que viajaba en primera clase tuvo más acceso a los botes salvavidas, y, por tanto, sobrevivieron.
2. Parecería que las mujeres se salvaron más que los hombres.
3. Parecería que la gente de más edad tenía más probabilidad de salvarse, así como los niños.
4. Parecería que las personas devotas y religiosas tuvieron mejores probabilidades de salvarse.
5. Parecería que las personas que viajaban con otras personas corrieron más riesgo de morir, por andar buscando a alguien más.
6. Seguramente se cumplió alguna superstición en el TITANIC, por ejemplo, que se haya quebrado un espejo, o que alguien haya tocado un cuchillo que se cayó al suelo, lo que provocó la mala suerte de la embarcación.

Estos supuestos son la materia prima para formular hipótesis.

FIN DEL LAB

Diferencia entre supuestos e hipótesis

¿Cuál es la diferencia entre los *supuestos* y las *hipótesis*? Tomemos como ejemplo el primer supuesto:

Parecería que la gente que viajaba en primera clase tuvo más acceso a los botes salvavidas, y, por tanto, sobrevivieron.

En los supuestos no se señalan *variables* (*clase*) sino *valores de variables* (*primera clase*). En esencia, el supuesto dice que ir en primera clase condujo a sobrevivir, lo que nos hace suponer que los pasajeros pudieron no ir en primera clase, y podían no haber sobrevivido.

Los supuestos nos permiten identificar variables, y construir hipótesis.

El trabajo con los supuestos es análisis puro. Formulando preguntas podemos ir determinando los datos involucrados.

1. ***...la gente que viajaba en primera clase...*** ¿Todas las personas viajaban en una clase? ¿Qué clases había?
2. ***...tuvieron acceso a botes salvavidas...*** ¿Se podría acceder o no a los botes, había alguna restricción? ¿El acceso a los botes estaba determinado por la clase?
3. ***...sobrevivieron...*** ¿No todos sobrevivieron? ¿Tener acceso a un bote salvavidas implicaba sobrevivir?

Declaración del objetivo de análisis

La ***declaración de objetivo de análisis*** ***(analysis goal statement)*** es la parte del proceso donde se redacta con todo cuidado una declaración que muestra la interacción de las variables que participan en el análisis, los fines que se persiguen, y el contexto que rodea al análisis.

Elementos de un objetivo de análisis

Un buen objetivo de análisis debe procurar los siguientes elementos:

1. ***Señalar las fuentes de datos*.** Esto es, sugerir de dónde se conseguirán los datos, a grandes rasgos. Aquí se pueden señalar fuentes, sistemas, etcétera. En caso de que no sepamos qué fuentes utilizar, no se especifica.
2. ***Señalar el tipo de técnica o análisis que se desea*.** Esto es, proporciona información del tipo de técnica estadística o metodológica que se utilizará en el análisis, en caso de conocerse.
3. ***Delimitar el alcance del estudio*.** Esto es, señalar límites y condiciones de los datos que han de incluirse en el estudio. Aquí puede tratarse de cualquier aspecto categórico o temporal, si es que aplica.
4. ***Enumerar variables y posibles relaciones entre ellas*.** Esto es, ser específicos en cuanto a las variables de interés, y las dependencias que pretendemos analizar o demostrar, sean de variación o causalidad.
    a. En ocasiones el objetivo se menciona en términos generales, y no describe variables; de todas maneras, se tendrá que traducir a variables, tarde que temprano.

Diferencias de objetivo dependiendo el tipo de análisis

Dependiendo el tipo de análisis, el objetivo puede variar.

Si el tipo de análisis es de descubrimiento, entonces el objetivo es genérico, las técnicas de estudio son genéricas, y se exploran todas las variables disponibles, en búsqueda de hallazgos o patrones.

Por ejemplo, en el caso TITANIC:

"Considerando la información histórica contenida en la enciclopedia británica, relativa a personas que iban en el TITANIC y la suerte que tuvieron, queremos hacer un análisis exploratorio histórico que nos permita saber los parámetros estadísticos, la existencia de datos atípicos y faltantes, la distribución de frecuencias, y la normalidad, considerando todas las variables disponibles en el conjunto de datos de referencia ".

Si el tipo de análisis es de descubrimiento, entonces el objetivo es específico, las técnicas de estudio son específicas, y se exploran sólo las variables requeridas por el estudio, con la finalidad de comprobar las hipótesis que se formulen.

"Considerando la información histórica contenida en la enciclopedia británica, relativa a personas que iban en el TITANIC y la suerte que tuvieron, queremos hacer un análisis exploratorio histórico que nos permita saber cómo afectaron a la sobrevivencia de las personas, aspectos como la clase de pasajero, sexo, rango de edad, religión, y si la persona viajaba sola o acompañada".

Jerarquía de objetivos

Desde luego, hay trabajos de analítica que son demasiado extensos como para describirse de manera completa en un solo objetivo, con cientos de variables. Lo que sucede en realidad es que elegimos un ***tema de interés***, alrededor del cual giran los problemas o las oportunidades.

Definido el tema, se comienzan a estructurar no uno, sino varios ***objetivos generales*** que nos permitan ir segmentando perímetros de interés. Por ejemplo, si queremos hacer un trabajo de analítica relacionado con las ventas: podemos tener objetivos generales que puedan resolverse por medio de técnicas de análisis exploratorio de datos, como, por ejemplo, sacar tablas de frecuencias respecto a las operaciones, o análisis volumétricos de unidades y montos vendidos, en general. Sería el equivalente a realizar un

trabajo de analítica de tipo *discovery*, en el cual no se trata de demostrar nada, ni hacer un juicio de valor específico.

En esta fase, se podrían generar objetivos como este:

- Utilizando la información de las operaciones de venta del último año, elaborar un análisis exploratorio de datos donde se muestre la tabla de frecuencia de número de operaciones, el análisis de datos atípicos, por sucursal.

A partir de los objetivos, se pueden generar ***objetivos particulares***, que impliquen la participación de solo algunas variables, en relación con una variable que actúe como variable dependiente, sugiriendo una causalidad o relación entre las variables.

- Utilizando la información de las operaciones de venta del último año, elaborar un análisis exploratorio de datos donde se muestre la correlación entre los montos de venta y las cantidades de venta, dependiendo del día de la semana en que ocurre la operación.

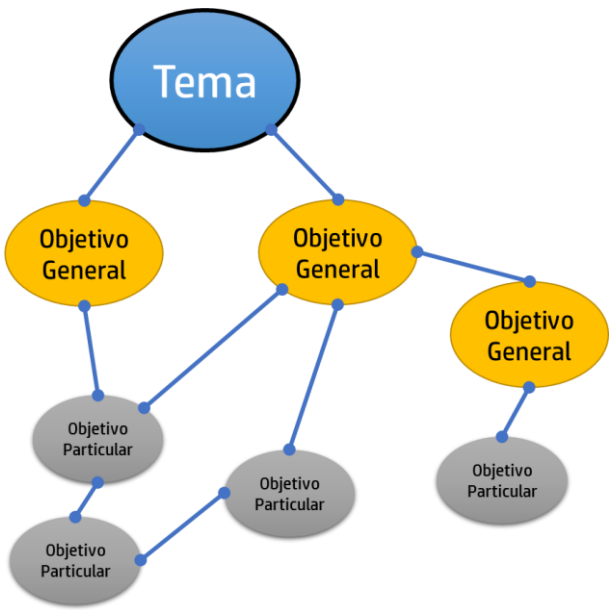


FIGURA 03.02: *Verificación de coherencia de hipótesis.*

Los objetivos más generales no concluyen nada en particular, mientras que los objetivos particulares sí lo hacen. Lo recomendable es elaborar objetivos generales que nos induzcan a elaborar objetivos particulares más pertinentes y dirigidos.

LAB 03.03: Redactar el objetivo del análisis del caso

En este Lab se analizan los supuestos coloquiales del caso, y se establece un *objetivo de análisis de datos*, que represente los alcances y límites definitivos del estudio.

Las tareas por realizar son:

5. Redactar el *objetivo de análisis de datos*.

Para el caso de estudio, podemos proponer el siguiente objetivo de análisis.

OBJETIVO DE ANÁLISIS:

"Considerando la información histórica contenida en la enciclopedia británica, relativa a personas que iban en el TITANIC y la suerte que tuvieron, queremos hacer un análisis exploratorio histórico muestral que nos permita saber cómo afectaron a la sobrevivencia de las personas, aspectos como la clase de pasajero, sexo, rango de edad, religión, y si la persona viajaba sola o acompañada".

¿Por qué esta declaratoria de objetivo es buena? Porque contiene los siguientes elementos:

1. Se hace referencia a que se trabajará con información ya existente. Eso quiere decir que no hay un flujo de información constante que se esté actualizando constantemente. Se usa lo que ya hay.
2. Se hace referencia a que es un análisis exploratorio, lo que quiere decir que se centrará en técnicas de estadística descriptiva; otras alternativas son análisis inferenciales, como por ejemplo las regresiones, análisis de clasificación y conglomerados, etcétera.

3. Se hace referencia a que es un análisis histórico, por lo cual se usa información del pasado, que puede no estar completa y es difícil de complementar.
4. Se hace referencia a que es un análisis muestral. Al no especificar qué tipo de muestra se utilizará, se asume una muestra aleatoria. Esto nos hace saber que no es un estudio censal, y que probablemente no tenemos todos los datos.
5. Se señalan una o más variables dependientes (variables que se explican en función a otras), en este caso, la sobrevivencia.
6. Se señalan una o más variables independientes (variables que explican a las variables dependientes), en este caso, clase de pasajero, sexo, rango de edad, y acompañamiento de la persona.

Para este caso de estudio, y dado que nos guiaremos por una declaratoria de objetivo de análisis, es claro que optaremos por un análisis **basado en objetivo**.

FIN DEL LAB

Desarrollo de hipótesis

El ***desarrollo de hipótesis*** es la parte del proceso donde se toman los supuestos e intereses de análisis, y se traducen a hipótesis que puedan ser trabajadas mediante técnicas de analítica.

Concepto de hipótesis

Una ***hipótesis*** es una explicación tentativa o propuesta sobre un fenómeno o problema observado, que se formula a partir de una pregunta o una observación. Es una afirmación o proposición que se basa en el conocimiento previo disponible, y que se utiliza para formular predicciones sobre lo que se espera que ocurra en un experimento o en una investigación.

Los elementos principales de una hipótesis son los siguientes:

1. ***Declaración de la relación entre dos o más variables:*** una hipótesis debe establecer una relación entre dos o más variables, es decir, una predicción sobre cómo una variable afectará a otra.

2. ***Variables independiente y dependiente:*** la hipótesis debe identificar la variable independiente (la que se cree que afecta a la otra variable) y la variable dependiente (la que se cree que es afectada por la variable independiente).

3. ***Especificidad:*** una hipótesis debe ser específica y clara, lo que significa que debe ser capaz de ser probada o refutada mediante datos empíricos.

4. ***Verificabilidad:*** una hipótesis debe ser capaz de ser sometida a prueba empírica de verificación. Esto significa que debe ser posible diseñar un experimento o estudio que pueda proporcionar datos que apoyen o refuten la hipótesis.

5. ***Rechazabilidad:*** una hipótesis debe ser *rechazable*, lo que significa que debe ser posible demostrar que es incorrecta. Si una hipótesis no es rechazable, no es una hipótesis científica válida.

6. ***Coherencia:*** una hipótesis debe ser coherente con otras teorías y observaciones relevantes en el campo. Esto aplica en el caso de investigaciones científicas.

Es importante aclarar que, en el caso de trabajos de analítica de tipo Discovery, realmente no partimos de ningún supuesto, y, por tanto, no declaramos objetivo, ni tenemos hipótesis. Lo que sucede aquí es que se inicia aplicando EDA sobre los datos, lo que nos dará pauta a generar supuestos, luego un objetivo, e hipótesis.

LAB 03.04: Redactar las hipótesis del caso

En este Lab se analiza el objetivo de análisis de datos, y a partir de ahí se generan hipótesis de investigación, que incluyan variables, relaciones entre ellas, y un supuesto a demostrar.

Las tareas por realizar son:

1. Analizar el *objetivo de análisis de datos* e identificar variables.
2. Elaborar hipótesis que incluyan variables, relaciones entre ellas, y supuestos a comprobar.

Analizar el objetivo de análisis de datos e identificar variables.

El objetivo de análisis de datos es el siguiente:

"Considerando la información histórica contenida en la enciclopedia británica, relativa a personas que iban en el TITANIC y la suerte que tuvieron, queremos hacer un análisis exploratorio histórico que nos permita saber cómo afectaron a la sobrevivencia de las personas, aspectos como la clase de pasajero, sexo, rango de edad, religión, y si la persona viajaba sola o acompañada".

Las variables identificables pueden ser estas:

[Considerando la información histórica contenida en la enciclopedia británica, relativa a personas que iban en el TITANIC y la suerte que tuvieron,] ***(fuente)*** [queremos hacer un análisis exploratorio histórico que nos permita saber] ***(tipo de estudio)*** como afectaron a la ***sobrevivencia de las personas***, aspectos como la ***clase de pasajero***, ***sexo***, ***rango de edad***, ***religión***, y si la persona viajaba sola o ***acompañada***.

Elaborar hipótesis que incluyan variables, relaciones entre ellas, y supuestos a comprobar.

De cada *supuesto* del caso, infiere las *variables*.

Trata de establecer relaciones entre variables (si es que existen), y menciona el supuesto, desde el punto de vista de las variables.

Supuesto	Hipótesis
Parecería que la gente que viajaba en primera clase tuvo más acceso a los botes salvavidas, y por tanto sobrevivieron.	**H1:** La ***clase*** en la que viajaba el pasajero, sí tuvo que ver con la probabilidad de ***sobrevivencia***. Creemos que, a mayor clase, más probabilidad de sobrevivencia.
Parecería que las mujeres se salvaron más que los hombres.	**H2:** El ***sexo*** del pasajero, sí tuvo que ver con la probabilidad de ***sobrevivencia***. Creemos que las mujeres se salvaron más que los hombres.
Parecería que la gente de más edad tenía más probabilidad de salvarse, así como los niños.	**H3:** La ***edad*** del pasajero, sí tuvo que ver con la probabilidad de ***sobrevivencia***. Creemos que los infantes y los viejos tuvieron más probabilidad de sobrevivir.
Parecería que las personas devotas y religiosas tuvieron mejores probabilidades de salvarse.	**H4:** La ***religión*** del pasajero, no tuvo que ver con la probabilidad de ***sobrevivencia***.
Supuesto	Hipótesis
Parecería que las personas que viajaban con otras personas corrieron más riesgo de morir, por andar buscando a alguien más.	**H5:** El ***número de acompañantes*** del pasajero, sí tuvo que ver con la probabilidad de ***sobrevivencia***. Creemos que las personas que viajaban solas, al no tener que preocuparse por nadie más, tuvieron más probabilidad de sobrevivir.
Es probable que alguien no cumplió con una superstición, lo que trajo mala suerte y provocó el hundimiento.	**H6:** El ***cumplimiento de supersticiones*** tuvo que ver con que ocurriera el ***hundimiento*** de la embarcación.

FIN DEL LAB

Análisis de coherencia de hipótesis

El ***análisis de coherencia de hipótesis*** es la parte del proceso donde se analiza si las hipótesis son coherentes con la declaratoria del objetivo de análisis. En caso de incongruencias, hay que modificar el objetivo de análisis, o bien, modificar o eliminar las hipótesis.

Se verifica si todas las hipótesis tienen que ver con el objetivo de investigación. Si no es así, se aumenta el objetivo, o se modifica o elimina la hipótesis.

Para hacer este análisis, los pasos sugeridos son los siguientes.

1. Identificar variables en el objetivo.
2. Identificar variables en las hipótesis.
3. Verificar correspondencia de variables.
4. Modificación o eliminación de hipótesis.
5. Modificación de objetivo.

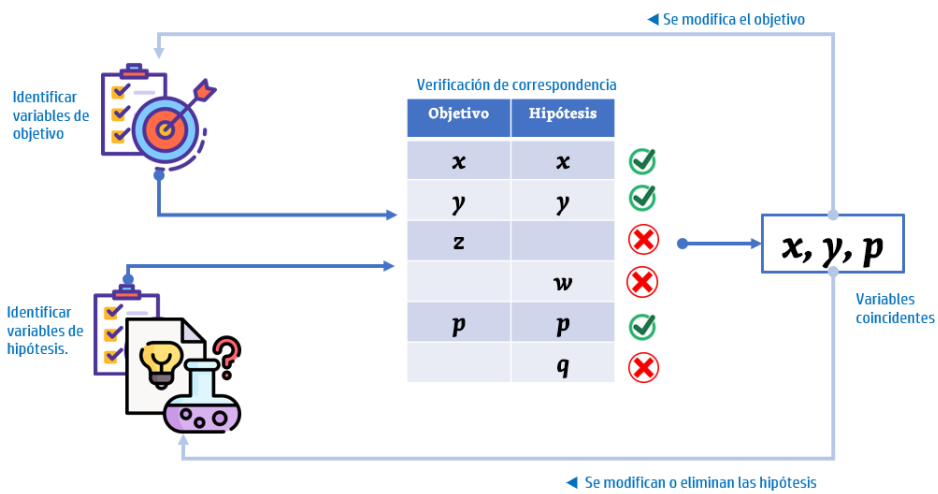


FIGURA 03.02: *Verificación de coherencia de hipótesis.*

LAB 03.05: Revisar la coherencia de las hipótesis del caso

En este Lab se analiza si las hipótesis y el objetivo de análisis de datos son coherentes. Esto es, asegurarse que no hay hipótesis que no tengan que ver con el objetivo.

Las tareas por realizar son:

1. Analizar la coherencia entre las hipótesis y el objetivo de análisis de datos.
2. Corregir diferencias o inexactitudes.

Analizar la coherencia entre las hipótesis y el objetivo de análisis de datos.

En nuestro caso, podría ser lo siguiente.

1. Respecto a la hipótesis 5:
    a. La hipótesis 5 es más amplia o detallada que el objetivo de análisis.
    b. Mientras que el objetivo el saber si la persona viajaba o no acompañada, la hipótesis profundiza en el número de acompañantes, que es más complicado de conocer.

2. Respecto a la hipótesis 6:
    a. La hipótesis 6 no es coherente. No hay registro de cumplimiento de supersticiones o no, y, además, trata del suceso en general, siendo que el resto de las hipótesis analizan rasgos de una persona, y consecuencias para una persona.
    b. Esta hipótesis sería coherente si estuviéramos comparando múltiples tragedias de hundimiento, y el hecho de que las personas en la embarcación cumplieran o no con supersticiones, y además, se documentara.

Corregir diferencias o inexactitudes.

En nuestro caso, serían las siguientes modificaciones:

1. Respecto a la hipótesis 5:
   a. Se decide modificar la hipótesis, ajustándola al objetivo, que refiere al hecho de que la persona viajara sola o acompañada, sin importar el número de acompañantes.
   b. Queda como sigue: **H5:** ~~El número de acompañantes del pasajero~~ El hecho de que la persona viajara sola o acompañada sí tuvo que ver con la probabilidad de sobrevivencia. Creemos que las personas que viajaban solas, al no tener que preocuparse por nadie más, tuvieron más probabilidad de sobrevivir.

2. Respecto a la hipótesis 6:
   a. Se elimina la hipótesis, por no ser consistente con el resto de las hipótesis.
   b. ~~**H6:** El cumplimiento de supersticiones tuvo que ver con que ocurriera el hundimiento de la embarcación.~~

FIN DEL LAB

Enumeración de fuentes de datos

La ***enumeración de fuentes*** es la parte del proceso donde se revisan los diferentes orígenes de datos en los que probablemente encontremos las variables de nuestro objetivo de análisis.

Una vez que sabemos qué variables requerimos para satisfacer nuestro objetivo de análisis, lo que sigue es revisar dónde podemos conseguir los datos que nos permitan realizar el análisis.

Si tenemos suerte, la organización tiene un departamento de TI que cuenta con ingenieros de datos que se encargan de extraer los datos de las diferentes fuentes, y nos entregan un arreglo bidimensional de datos en la forma en que necesitamos para trabajar. Esto pocas veces sucede.

Lo más común es que encontremos la información dispersa en diferentes archivos, en diferentes formatos, y con diferente nivel de actualización.

Asumiendo que tenemos varias fuentes, revisa las siguientes condiciones:

1. ***Información de la fuente.*** Hay que revisar el nombre de la fuente, la plataforma o base, el servidor en que se encuentra.
    a. Es muy común asignar a las fuentes un nombre corto, que permita referir de forma rápida toda la información específica.
    b. Aquí se proporciona información general de la fuente, que podría ser pertinente. Cada organización tendrá diferente detalle de información.

2. ***Permisos.*** Es necesario revisar si se tiene permisos o privilegios suficientes para poder tener acceso a la fuente.
    a. Si no se tienen permisos, es como si no dispusiéramos de los datos.
    b. En ocasiones es necesario seguir un protocolo de solicitud de acceso, cada vez que accedemos a los datos; en ese caso, se requiere tener perfectamente entendido el protocolo.

   c. Las cadenas de conexión siempre deben validarse, para asegurarnos que estamos leyendo datos del lugar correcto.

3. ***Frecuencia.*** Es necesario revisar la frecuencia de actualización y acceso.

   a. La ***frecuencia de actualización*** tiene que ver con qué frecuencia los datos son actualizados. Hay fuentes que se actualizan en tiempo real, es decir, de un minuto a otro pueden variar, mientras que hay otros datos que se actualizan a diario, semanalmente, mensualmente y así, en cuyo caso, los datos son los mismos mientras no haya una nueva actualización.

   b. La ***frecuencia de acceso*** es la frecuencia con la que se nos permite acceder a los datos para hacer lectura. Es común que los datos en producción, por ejemplo, se actualicen en tiempo real, mientras que solo se nos permita acceder a los datos una vez al día, en la madrugada, cuando la lectura no afecta el desempeño de las aplicaciones de la organización.

   c. De nada sirve hacer una herramienta de consulta que muestre datos en tiempo real, si las fuentes de datos se actualizan con periodicidad diaria, por ejemplo.

4. ***Sincronía.*** Es necesario que la actualización de todas las fuentes tenga cierta sincronía. Si una fuente se actualiza cada hora, y otra fuente se actualiza cada día, los datos pueden ser poco confiables, pues la fuente con mayor frecuencia puede tener datos de los cuales la fuente con menor frecuencia no se ha enterado. Es común actualizar los datos de tal forma que todos los datos tengan el mismo nivel de actualización, por ejemplo, que tanto la fuente que se actualiza cada hora, como la que se actualiza cada día, se lean de forma diaria.

La información que se incluya de cada elemento depende mucho de las organizaciones. Lo verdaderamente importante es que se tenga documentada la información, y que sea consistente.

LAB 03.06: Documentar información de la fuente

En este Lab se revisa la forma en que se tiene que documentar la información de una fuente de datos.

Las tareas por realizar son.

1. Documentar la información de una fuente de datos.

Esta tabla muestra la información de la fuente para el caso de análisis de datos del TITANIC.

Parámetro	Detalle
Nombre corto:	TITANIC
Formato:	CSV
Nombre físico de la fuente	pasajeros_titanic.csv
Ubicación	\data
Servidor	GitHub - AnaliticaPythonR1
Dirección IP	No aplica
Última actualización:	3 de marzo de 2020
Frecuencia de actualización:	Versión final. No se actualiza.
Responsable:	*Encyclopedia Britannica*
Permisos requeridos de acceso:	No aplica
Protocolo de solicitud de acceso:	No aplica
Sincroniza con otras fuentes:	No

FIN DEL LAB

Identificación de variables

La ***Identificación de variables*** es la parte del proceso donde se analiza el objetivo de análisis y las hipótesis, y se enumeran las variables que se detectan, haciendo énfasis en clasificarlas en función a si son *dependientes* o *independientes*.

Lo primero que se tiene que hacer, es analizar la redacción de la declaración de objetivo de análisis, así como las hipótesis, con la finalidad de enumerar todas las variables que se pueden identificar.

Se tienen dos tipos de variables, en cuanto a su relación: dependientes, e independientes.

1. Las ***variables dependientes*** se explican en función de las variables independientes.
2. Las ***variables independientes***, explican o permiten inferir a las variables dependientes.

Una forma fácil identificar cuáles son las variables dependientes e independientes, usando la lógica.

1. A veces, en una relación entre variables, cualquiera de las dos puede ser dependiente e independiente.

2. Imagina que se tiene información de personas, y se tienen las variables **`peso`** y **`altura`**.

3. Podríamos intentar investigar si el **`peso`** determina a la **`altura`**. Es decir, **`peso`** es independiente, y **`altura`** es dependiente. 1. Podríamos intentar investigar si la **`altura`** determina el **`peso`**. Es decir, **`altura`** es independiente, y **`peso`** es dependiente.

4. A veces, en una relación entre variables, es claro que una de las dos puede ser dependiente e independiente.

5. Imagina que se tiene información de patrones de compra, y se tienen las variables **`sexo`** y **`monto de compra mensual promedio`**.

6. Podríamos intentar investigar si el **sexo** determina el **monto de compra mensual promedio**. Es decir, **sexo** es independiente, y **monto de compra mensual promedio** es dependiente. 1. Podríamos intentar investigar si el monto **de compra mensual promedio** determina el **sexo**, lo cual es un disparate. 1. **sexo** es independiente, y **monto de compra mensual promedio** es dependiente, y esa es la única posibilidad.

LAB 03.07: Identificar variables dependientes e independientes

En este Lab se identifican las variables dependientes y dependientes que se tienen en el caso.

Las tareas por realizar son:

1. Enumerar las variables del caso.
2. Distinguir las variables dependientes e independientes.

Enumerar las variables del caso.

Las variables que se identifican en el objetivo de análisis son las siguientes:

1. sobrevivencia
2. clase de pasajero
3. sexo
4. rango de edad
5. religión
6. acompañado

Distinguir las variables dependientes e independientes.

En cuanto a su dependencia, las variables serían de esta manera.

Dependiente	Independiente
sobrevivencia	**clase de pasajero**
	sexo
	rango de edad
	religión
	acompañamiento

La mayoría de las variables anteceden al hecho de que sobrevivieran o no en el accidente del TITANIC, y no hay manera de que la sobrevivencia las determine.

Tenemos 1 variable dependiente, y 5 variables independientes. En total tenemos 6 variables.

En este momento aún no sabemos si los datos están disponibles o no en las fuentes.

FIN DEL LAB

CAPÍTULO 4:

Análisis preliminar de los datos

Análisis censal y muestral

El ***análisis volumétrico*** es la parte del proceso donde se analizan los volúmenes esperados de datos que es necesario tener para realizar un análisis de datos exitoso. En esta parte se decide realizar un análisis censal o muestral, y se definen las muestras estadísticas, en su caso.

Si tenemos datos de todos para el caso, el análisis puede ser un ***estudio censal***, y es 100% confiable. Si no se tiene el total de los datos, el análisis puede ser un ***estudio muestral***, es decir, se trabaja con una selección aleatoria de registros.

Concepto de muestra estadística

Una ***muestra estadística*** es un subconjunto seleccionado de una población más grande. En estadística, la muestra se utiliza para hacer inferencias y generalizaciones sobre la población a partir de la información contenida en la muestra.

La selección de una muestra debe ser *representativa de la población* en términos de características relevantes para el estudio, de manera que la información obtenida de la muestra pueda ser generalizada a la población de interés. Para lograr esto, se utilizan técnicas de muestreo que permiten obtener una ***muestra aleatoria simple*** o una ***muestra estratificada***, que

minimice el sesgo y la variabilidad en la estimación de las características de la población estudiada.

Una muestra aleatoria es una muestra seleccionada al azar de una población, donde cada individuo de la población tiene la misma probabilidad de ser seleccionado. En una muestra aleatoria no se consideran las características específicas de los individuos de la población, y se corre el riesgo que algunos grupos de individuos no estén representados en la muestra.

En una muestra aleatoria tomada de una población formada por 70% hombres y 30% mujeres, se corre el riesgo de que la muestra incluya un 90% hombres y 10% mujeres, lo cual no es representativo de la población.

Por otro lado, una muestra estratificada implica dividir la población en subgrupos (o estratos) basados en ciertas características comunes, como la edad, el género, la ubicación geográfica, entre otras. Conformados los estratos, se selecciona una muestra aleatoria de cada estrato.

En una muestra estratificada por sexo, tomada de una población formada por 70% hombres y 30% mujeres, debe incluir idealmente un 70% hombres y 30% mujeres, lo cual es representativo de la población.

La muestra estratificada permite asegurar que cada subgrupo esté representado en la muestra, lo que puede ser importante si los subgrupos tienen características distintas que pueden afectar los resultados.

Las muestras estadísticas se utilizan en diversos campos, como la investigación de mercados, la salud pública, la economía, la ciencia política, entre otros, y su uso permite hacer inferencias más precisas y confiables sobre la población a partir de un subconjunto de datos.

Cuando decidimos trabajar con una *muestra estadística*, requerimos el total de elementos, el nivel de confianza esperado (típicamente, 90%, 95%, 99%), y el error permitido.

Colocar nivel de confianza 100%, con error 0%, equivale a hacer un estudio censal.

Para saber qué tipo de análisis podemos realizar, es necesario ver qué datos tenemos disponibles.

Tómese en cuenta que en el Titanic viajaban 2,225 personas, por lo cual, el universo de datos de la embarcación (`N`) sería de 2,225 observaciones.

Cálculo del tamaño de la muestra estadística

El tamaño de la muestra estadística es el mínimo de observaciones que se requiere para realizar operaciones estadísticas con cierto nivel de confianza, asumiendo un margen de error.

El tamaño de la muestra es un aspecto crítico del diseño de la investigación estadística, ya que determina la precisión y la fiabilidad de las conclusiones que se puedan obtener a partir del análisis de la muestra.

Generalmente, una muestra más grande es mejor que una pequeña, pero está el tema del costo, en términos de dinero y de tiempo. Un tamaño de muestra grande, en general, proporciona una mayor precisión y confianza en los resultados obtenidos, pero también puede requerir mayores recursos y costos para la recolección de datos. Por otro lado, un tamaño de muestra pequeño puede generar conclusiones menos precisas y confiables, pero también puede ser más fácil y económico de recolectar.

Una fórmula general para calcular la muestra estadística es la siguiente:

$$n = \frac{Z^2 pqN}{NE^2 + Z^2 pq}$$

Donde:

- `Z` es el coeficiente de confianza. Lo más común es 90%=1.645, 95%=1.96, 99%=2.576
- `p` es la probabilidad de que ocurra un evento; si no se conoce, típicamente es del 50% (0.5).
- `q` es la probabilidad de que no ocurra el evento; es (`1-p`)
- `E` es el error máximo aceptado; típicamente, es el 5% (0.05)
- `N` es el tamaño de la población.

Para saber si necesitamos calcular una muestra estadística, es necesario que sepamos cuál es el universo de observaciones, el nivel de confianza que se desea, y el margen de error permitido.

Para calcular en línea el tamaño de la muestra, se puede usar un servicio de cálculo en línea. Aquí un ejemplo: https://www.corporacionaem.com/tools/calc_muestras.php

LAB 04.01: Calcular el tamaño de la muestra para el caso

En este Lab se calcula la muestra estadística para el universo de datos del TITANIC.

Las tareas por realizar son:

1. Calcular el tamaño de la muestra estadística para el caso de análisis.

En el caso de que no tuviéramos todos los datos de los pasajeros, ¿cuál sería el mínimo de observaciones que necesitamos para realizar operaciones estadísticas con un 99% de nivel de confianza y un 5% de margen de error?

El universo de datos es equivalente al número de pasajeros que viajaban en el TITANIC, en este caso 2,225.

El cálculo sería así:

```
# Declara las variables, cuidando que sean del tipo correcto.
N=2224
p=0.50
q=1-p
E=0.05
Z=2.576

# Se codifica la fórmula, para calcular el tamaño de la
# muestra (n), y muestra el resultado.
# Toma en cuenta la propiedad conmutativa 'Cuando
# multiplicamos, el orden de los factores no afecta
# al producto'.
n=int((Z**2*p*q*N)/((N*E**2)+(Z**2*p*q)))

print(f'El tamaño de la muestra es {n}')
```

```
El tamaño de la muestra es 511
```

El tamaño de la muestra es de 511 observaciones. Menos que eso, ya no permite el nivel de confianza que se espera, con el margen de error tolerado.

Desde luego, existen otras fórmulas para el cálculo de la muestra, que tienen que ver con el tipo de hipótesis que se desea demostrar o comprobar, el número de colas de la curva, etcétera.

FIN DEL LAB

Series y DataFrames

La librería pandas de Python maneja principalmente dos tipos de estructuras:

1. ***Series:*** una serie es un objeto unidimensional similar a un array o una lista que puede contener diferentes tipos de datos, como números enteros, números de punto flotante, cadenas, entre otros. Cada elemento en una serie tiene un índice que lo identifica de manera única en la serie.

2. ***DataFrame:*** un DataFrame es un objeto bidimensional similar a una tabla que contiene una colección ordenada de columnas, donde cada columna puede ser de un tipo de datos diferente (por ejemplo, una columna puede contener números enteros, mientras que otra puede contener cadenas). Un DataFrame también tiene un índice que identifica de manera única cada fila en el DataFrame.

En todos los casos, para utilizar pandas, es necesario importar la librería **`pandas`**, que con frecuencia usa el alias **`pd`**:

```
import pandas as pd
```

Tanto las **Series** como los **DataFrames** tienen múltiples maneras de crearse y poblarse de datos. Para crear Series, se puede utilizar el método **`pd.Series()`**, para crear DataFrames, podemos utilizar **`pd.DataFrame()`**.

Los DataFrame pueden crearse y poblarse a partir de diferentes fuentes.

Para crear un DataFrame desde un ***archivo .CSV***:

```
df = pd.read_csv('archivo.csv')
```

Para crear un DataFrame desde un ***archivo .TXT***:

```
df = pd.read_table('archivo.txt')
```

Para crear un DataFrame desde una ***base de datos SQL Server***:

```
import pyodbc

# Conectar a la base de datos
conn = pyodbc.connect('DRIVER={SQL Server};SERVER=server_name;
DATABASE=database_name;UID=username;PWD=password')

# Enviar una consulta SQL a la conexión, y cargar los datos
# recuperados a un DataFrame
df = pd.read_sql('SELECT * FROM tabla', conn)
```

Para crear un DataFrame desde una ***base de datos MySQL***:

```
import mysql.connector
import pandas as pd

# Conectar a la base de datos
conn = mysql.connector.connect(user='username',
password='password', host='host_name',
database='database_name')

# Enviar una consulta SQL a la conexión, y cargar los datos
# recuperados a un DataFrame
df = pd.read_sql('SELECT * FROM tabla', conn)
```

Para crear un DataFrame desde una ***base de datos SQLite***:

```
import sqlite3
import pandas as pd

# Conectar a la base de datos
conn = sqlite3.connect('database.db')

# Enviar una consulta SQL a la conexión, y cargar los datos
# recuperados a un DataFrame
df = pd.read_sql_query('SELECT * FROM tabla', conn)
```

Para crear un DataFrame desde una ***base de datos Oracle***:

```
import cx_Oracle

# Conectar a la base de datos
conn=cx_Oracle.connect('username/password@host:port/database')

# Enviar una consulta SQL a la conexión, y cargar los datos
# recuperados a un DataFrame
df = pd.read_sql('SELECT * FROM table_name', conn)
```

Toma en cuenta que, independientemente del origen, los DataFrames trabajan con un arreglo bidimensional de datos, de una o más columnas. DataFrame no trabaja a nivel modelo, no representa conjuntos de tablas, ni maneja relaciones entre tablas.

Es importante mencionar aquí que algunas funciones de pandas funcionan para las filas como para las columnas. En caso de ambigüedad, es necesario indicar el parámetro **axis**: si tiene al valor de 0, quiere decir que estamos trabajando con registros/filas, y si tiene el valor de 1, quiere decir que estamos trabajando con campos/columnas.

Listas, diccionarios y tuplas

Es importante mencionar que pandas convive mucho con otras estructuras del lenguaje Python, conocidas también como ***colecciones***: listas, diccionarios, y tuplas.

1. Las ***listas*** de Python son estructuras de datos que manejan entradas indexadas de datos, en forma de serie.

    a. Las listas se pueden reconocer porque están delimitadas por *square brackets* `[]`, y separar sus valores por comas.

2. Los ***diccionarios*** de Python son estructuras de datos que manejan entradas indexadas de datos, en forma de series de pares de valores, en donde un valor es la *llave* (*key*), que debe ser única, y el otro es el *valor* (*value*).

   a. Los diccionarios se pueden reconocer porque están delimitados por *curly brackets* `{}`, separar los pares de valores por comas, y separar la llave del valor con dos puntos.

3. Las ***tuplas*** de Python son secuencias inmutables de datos.

   a. Las tuplas se pueden reconocer porque son valores separados por comas; aunque no es necesario, suelen estar delimitados por *rounded brackets* `()`.

Listas

Este es un ejemplo de una la creación de una lista en Python. Se separan valores por comas, y se delimita con *square brackets* **`[]`**.

```
lista = [10, 20, 30, 40, 50]
```

Es importante mencionar que las listas pueden almacenar datos de diferentes tipos.

```
lista_mixta = [10, 20, 'A', True, 60]
```

Es importante mencionar que las listas pueden almacenar como elementos a otras listas.

```
lista_de_listas = [
  [10,20,30],
  ['A','B','C']
]
```

Podría decirse que una Serie de pandas es como una lista donde el contenido de los elementos es homogéneo en cuanto a su tipo. Podría decirse que un DataFrame de pandas es una lista de listas, donde la primera fila es el nombre de las columnas, y las restantes listas son filas de datos.

Diccionarios

Por otro lado, este es un ejemplo de un diccionario en Python. Se separan valores por comas, y se delimita con *curly brackets* **{}**.

```
diccionario = {
  1:'uno',
  2:'dos'
}
```

Es importante mencionar que los diccionarios pueden almacenar llaves y valores de diferentes tipos.

```
diccionario_mixto = {
  'uno':1,
  2:'dos'
```

Al igual que las listas, los diccionarios pueden almacenar a otros diccionarios como valor.

Tuplas

Por otro lado, este es un ejemplo de una tupla en Python. Se separan valores por comas, y si se delimitan, lo hacen con *rounded brackets* **()**.

```
tupla='A','B','C'

tupla=('A','B','C')
```

Iterabilidad

Una característica de todas estas estructuras es que son ***iterables***, es decir, se pueden leer de forma secuencial.

Una manera de hacerlo es usar **`for`**:

```
for elemento in colección:
  # Acciones con los elementos.
```

Creación de Serie y DataFrame a partir de listas y diccionarios

Es muy importante familiarizarse con listas y diccionarios, porque pandas hace un uso de intensivo de ellos, en forma de parámetros y funciones.

De hecho, es posible crear una Serie de pandas a partir de una lista:

```
mi_lista = [10, 20, 30, 40, 50]

mi_serie = pd.Series(mi_lista)
```

Y es posible crear un DataFrame de pandas a partir de un diccionario.

```
datos = {
     'nombre': ['Juan', 'Maria', 'Luis', 'Christy'],
     'edad': [25, 32, 18, 47],
     'ciudad': ['Bogotá', 'CDMX', 'Buenos Aires', 'Houston']
      }

df = pd.DataFrame(datos)
```

LAB 04.02: Trabajar con listas, diccionarios y tuplas

En este Lab se revisa la técnica para crear, recuperar elementos e iterar las colecciones más importantes de Python: listas, diccionarios y tuplas.

Las tareas por realizar son:

1. Codificar la creación, recuperación de valores e iteración de una lista.
2. Codificar la creación, recuperación de valores e iteración de un diccionario.
3. Codificar la creación, recuperación de valores e iteración de una tupla.

Codificar la creación, recuperación de valores e iteración de una lista.

```
# Crear una lista con 5 elementos.
lista = [10, 20, 30, 40, 50]

# Imprimir la lista creada.
print(lista)



# Imprimir el tipo de objeto y comprobar que es una lista.
print(type(lista))

# Recuperación de un elemento de la lista usando el índice
# base cero.  Recuperar el elemento con el índice 2
# (tercer valor de la lista).
print(lista[2])

# Iterar una lista usando for e imprimiendo cada elemento
# de la lista.
for e in lista:
  print(e)
```

```
[10, 20, 30, 40, 50]
<class 'list'>
30
10
20
30
40
50
```

Codificar la creación, recuperación de valores e iteración de un diccionario.

```
# Crear un diccionario.
diccionario={
  1:'uno',
  2:'dos',
  3:'tres',
  4:'cuatro',
  5:'cinco'
}

# Imprimir un diccionario.
print(diccionario)

# Imprimir el tipo de objeto y ver que es un diccionario.
print(type(diccionario))

# Recuperación de un elemento del diccionario, usando la llave.
print(diccionario[2])

print(diccionario[10]) # Provoca error: no existe la llave.

# Recuperación de un elemento del diccionario, usando
# la llave y get().
print(diccionario.get(2,'No encontrado'))
print(diccionario.get(10,'No encontrado'))

# Iteración de las llaves de un diccionario.
for k in diccionario.keys():
  print(k)
```

```
# Iteracion de los valores de un diccionario.
for v in diccionario.values():
  print(v)

# Iteración de llaves y valores de un diccionario, en
# forma de tupla.
for t in diccionario.items():
  print(t)

for k,v in diccionario.items():
  print(f'La llave {k} está asociada al valor {v}')
```

```
{1: 'uno', 2: 'dos', 3: 'tres', 4: 'cuatro', 5: 'cinco'}
<class 'dict'>
dos
dos
No encontrado
1
2
3
4
5
uno
dos
tres
cuatro
cinco
(1, 'uno')
(2, 'dos')
(3, 'tres')
(4, 'cuatro')
(5, 'cinco')
La llave 1 está asociada al valor uno
La llave 2 está asociada al valor dos
La llave 3 está asociada al valor tres
La llave 4 está asociada al valor cuatro
La llave 5 está asociada al valor cinco
```

Codificar la creación, recuperación de valores e iteración de una tupla.

```
# Crear una Tupla.
tupla=('A','B','C')

# Imprimir una tupla.
print(tupla)

# Imprimir el tipo de objeto y ver que es una tupla.
print(type(tupla))

# Recuperación de un elemento de la tupla usando el
# índice base cero. Recuperar el elemento con el índice1
# (segundo valor de la tupla).
print(tupla[1])

# Iteración de los elementos de una tupla.
for e in tupla:
  print(e)
```

```
('A', 'B', 'C')
<class 'tuple'>
B
A
B
C
```

FIN DEL LAB

Carga de datos a un DataFrame

Se pueden cargar datos desde muchas fuentes, usando pandas. Aquí exploraremos dos de las fuentes más comunes: CSV y Excel.

Cargando datos desde CSV

Los archivos ***CSV*** (***comma separated values***) son archivos textuales que almacenan información columnar, separada regularmente por comas.

Para leer datos desde un archivo CSV, se puede utilizar el método **`read_csv()`**, se pueden utilizar los siguientes parámetros:

- **`filepath_or_buffer`**: Esta es la ubicación o el nombre del archivo CSV que se va a leer. Puede ser una ruta de archivo o una URL, como, por ejemplo, una fuente de tipo ***raw*** en GitHub. No es necesario especificar el nombre del parámetro, si se coloca la ubicación o nombre del archivo en primer lugar.
- **`sep`**: Especifica el delimitador utilizado en el archivo CSV para separar los valores de las columnas. El valor predeterminado es ",".
- **`header`**: Indica si la primera fila del archivo CSV contiene encabezados de columna o no. El valor predeterminado es infer, que intenta inferir si hay encabezados de columna en función del contenido del archivo.
- **`index_col`**: Especifica la columna (o columnas) que se utilizarán como índice del DataFrame resultante.
- **`usecols`**: Se utiliza para especificar las columnas del archivo CSV que se van a incluir en el DataFrame resultante.
- **`dtype`**: Se utiliza para especificar el tipo de datos de las columnas del DataFrame resultante. Los tipos de datos deben especificarse a través de un diccionario.
- **`na_values`**: Se utiliza para especificar los valores que se consideran como valores faltantes en el archivo CSV. Por omisión, se consideran faltantes los vacíos, NaN, y None.

- **`skiprows`**: Se utiliza para saltar un número determinado de filas desde el principio del archivo CSV.

- **`nrows`**: Especifica el número máximo de filas que se leerán del archivo CSV.

Los datos de trabajo se leen del archivo CSV, y se almacenan como un arreglo bidimensional de tipo ***DataFrame***, usando la función **`read_csv()`**.

Suponiendo que tenemos un conjunto de datos almacenado en un archivo llamado **`datos.csv`**, y queremos leer el archivo y almacenar su contenido en un DataFrame llamado **`df`**, la sintaxis para hacerlo sería la siguiente:

```
import pandas as pd

df=pd.read_csv('datos.csv')
```

Cuando los datos se tienen en un repositorio en la nube, como puede ser un repositorio de **GitHub**, en lugar del nombre de archivo se pasa una URL que permite el acceso al archivo en modo *raw*.

Cuando se carga un archivo desde un archivo CSV, en ocasiones pandas interpreta los tipos de datos de las columnas con un tipo distinto al que tienen; esto también ocurre con frecuencia al existir datos vacíos o nulos.

Es posible especificar el tipo de datos de las columnas, con el apoyo de un diccionario:

```
import pandas as pd

tipos_correctos={
  'campo1':int,
  'campo2':object,
  'campo3':float
}

df=pd.read_csv('datos.csv',dtype=tipos_correctos)
```

En este caso, al leer el archivo, pandas le asignaría el tipo **`int`** a **`campo1`**, **`object`** a **`campo2`**, **`float`** a **`campo3`**.

Cargando datos desde Libros de Excel

Los archivos ***XLSX*** son Libros de Excel, que almacenan datos principalmente en dos formatos: ***Hojas electrónicas de cálculo*** (***spreadsheets***), y ***Rangos de celdas***.

Para leer datos desde un archivo XLSX, se puede utilizar el método **`read_excel()`**, con los siguientes parámetros:

- **`io`**: Especifica la ubicación o el nombre del archivo Excel que se va a leer. Puede ser una ruta de archivo o una URL. No es necesario especificar el nombre del parámetro, si se coloca la ubicación o nombre del archivo en primer lugar.
- **`sheet_name`**: Especifica el nombre o índice de la hoja del Libro de Excel que se va a leer. Si no se especifica, se leerá la primera hoja por defecto.
- **`header`**: Indica si la primera fila de la hoja de Excel contiene encabezados de columna o no. El valor predeterminado es 0, lo que significa que la primera fila es considerada como encabezado de columna.
- **`index_col`**: Especifica la columna (o columnas) que se utilizarán como índice del DataFrame resultante.
- **`usecols`**: Se utiliza para especificar las columnas del archivo Excel que se van a incluir en el DataFrame resultante.
- **`dtype`**: Se utiliza para especificar el tipo de datos de las columnas del DataFrame resultante.
- **`na_values`**: Se utiliza para especificar los valores que se consideran como valores faltantes en el archivo Excel.
- **`skiprows`**: Se utiliza para saltar un número determinado de filas desde el principio de la hoja de Excel.
- **`nrows`**: Especifica el número máximo de filas que se leerán de la hoja de Excel.

Suponiendo que tenemos un conjunto de datos almacenado en un archivo llamado **`datos.xlsx`**, y queremos leer los datos contenidos en la Hoja **`2023`**, y almacenar su contenido en un DataFrame llamado **`df`**, la sintaxis para hacerlo sería la siguiente:

```
import pandas as pd

df=pd.read_excel('datos.xlsx', sheet_name='2023')
```

Suponiendo que tenemos un conjunto de datos almacenado en un archivo llamado **`datos.xlsx`**, y queremos leer los datos contenidos en la Hoja **`2023`**, en el Rango de celdas **`E5:H450`**, y almacenar su contenido en un DataFrame llamado **`df`**, la sintaxis para hacerlo sería la siguiente:

```
import pandas as pd

df=pd.read_excel('datos.xlsx', sheet_name='2023',
      usecols='E:H', skiprows=4, nrows=446)
```

Cargando datos desde texto sin delimitadores

Los archivos ***TXT*** pueden tener sus columnas no delimitadas por ningún símbolo, sino por la posición en la que inician o terminan las columnas.

Para leer datos desde un archivo TXT por posiciones, se puede utilizar el método **`read_fwf()`** (***fixed-width formatted***), se pueden utilizar los siguientes parámetros:

- **`filepath_or_buffer`**: ruta de archivo o un objeto que contenga los datos que se van a leer. Puede no ponerse el nombre del archivo sin especificar el nombre del parámetro, si es el primer parámetro que se coloca.
- **`colspecs`**: una lista de tuplas que indica las posiciones de inicio y fin de cada columna. Si no se proporciona, se utilizará el ancho de las columnas.

- **`names`**: una lista que contiene los nombres de las columnas. Si no se proporciona, se numerarán las columnas de forma predeterminada.
- **`widths`**: una lista de enteros que indica el ancho de cada columna.
- **`header`**: entero o lista de enteros que indica qué fila o filas se utilizarán como encabezado de las columnas. Por defecto es **`0`**, que indica la primera fila. Si no hay encabezado, se puede establecer en **`None`**.
- **`index_col`**: una lista de enteros o nombres de columnas que se utilizarán como índices del DataFrame.
- **`delimiter`**: cadena de caracteres que se utilizará como delimitador de las columnas. Por defecto es un espacio en blanco.
- **`encoding`**: codificación de caracteres utilizada en el archivo. Por defecto es **`None`**, que significa que se utilizará la codificación predeterminada del sistema.
- **`skiprows`**: número o lista de números de filas que se van a omitir.
- **`skipfooter`**: número de filas que se van a omitir al final del archivo.
- **`dtype`**: diccionario de tipos de datos que se asignarán a las columnas.
- **`na_values`**: cadena de caracteres o lista de cadenas que se considerarán como valores faltantes.

Suponiendo que tenemos un conjunto de datos almacenado en un archivo llamado **`directorio.txt`**, donde cada fila contiene 3 datos: el correo, de la posición 1 a la 25, el nombre de la posición 26 a 60, y teléfono de la posición 61 al 75, donde la primera línea contiene los nombres de columna, y se desea almacenar su contenido en un DataFrame llamado **`df`**, la sintaxis para hacerlo sería la siguiente:

```
df=pd.read_fwf('directorio.txt',
                    widths=[25,60,75],
                    header='infer')
```

Forma de los datos (shape)

La ***forma de los datos*** representa el número de filas y columnas que tiene un arreglo bidimensional de datos. Analizar la forma anticipa la suficiencia de datos para realizar análisis.

Aun sin ver los datos, viendo la forma podemos darnos una idea si los datos disponibles son suficientes para el análisis que deseamos realizar. Si el número de filas no alcanza el mínimo señalado por el tamaño de la muestra, o si el número de columnas es más pequeño que el número de variables identificadas en el objetivo de análisis, podemos suponer problemas que deberemos analizar con detalle.

Hay que decirlo: existe la posibilidad que a partir de una columna se puedan calcular otras, por tanto, una falta de variables no necesariamente implica insuficiencia. También, de nada sirve que haya un número grande de columnas, si no están las que se requieren. Puede haber muchas variables, y aun así, haber insuficiencia.

La manera de consultar la forma de un DataFrame es usando la propiedad **shape**, que retorna una tupla con dos elementos; el primero de ellos corresponde al número de filas, mientras que el segundo corresponde al número de columnas.

Supongamos que tienes un DataFrame llamado **df**.

Para conocer su forma, sería de la siguiente manera:

```
df.shape
```

Si queremos imprimir por separado las filas de las columnas, sería de la siguiente manera:

```
print('Filas:', df.shape[0])
print('Columnas:', df.shape[1])
```

Consultar el elemento con el subíndice 1 de la tupla retornada por **shape** es equivalente a consultar la longitud del DataFrame, usando el método **len()**.

```
len(df)
```

LAB 04.03: Cargar datos a un DataFrame desde un archivo CSV

En este Lab se revisa el código requerido para cargar datos, desde un archivo CSV alojado en GitHub, y cargar los datos en un DataFrame de pandas.

Las tareas por realizar son:

1. Cargar datos desde un archivo CSV disponible en **GitHub**.
2. Revisar la forma de un conjunto de datos usando **shape**.
3. Analizar si la forma del DataFrame es apropiada.

Cargar datos desde un archivo CSV disponible en GitHub.

```
# Se importa la librería pandas, para el manejo de datos.
# Se le asigna el alias "pd", que es el estándar de
# codificación de la comunidad.

import pandas as pd

# Como los datos están en GitHub, en lugar del nombre de
# archivo se utiliza la URL para acceso raw a los datos,
# que es lo mismo que leer un archivo que tenemos físicamente
# en nuestro equipo.
# Se declara una variable para almacenar la URL.

pasajeros_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
pasajeros_titanic.csv'

# Se cargan los datos del archivo "pasajeros_titanic.csv" a un
# DataFrame de pandas llamado "titanic", usando la función
# pd.read_csv(), usando como nombre de archivo la variable
# que almacena la URL para acceder a los datos.

titanic = pd.read_csv(pasajeros_titanic_csv)
```

IMPORTANTE: Con pandas trabajaremos con un arreglo bidimensional de datos. En la práctica, hablar del arreglo bidimensional de datos, es lo mismo que hablar de los datos, del conjunto de datos, o del DataFrame; por otro lado, será común que hagamos referencia a las columnas del DataFrame, como columna, campo, característica, e incluso dato. A partir de este momento, nos referiremos de forma indistinta a los elementos, con cualquiera de sus sinónimos.

Revisar la forma de un conjunto de datos usando shape

```
# Muestra la forma del conjunto de datos titanic,
# usando la propiedad shape

titanic.shape
```

```
(1310, 14)
```

Consulta por separado el número de filas y el número de columnas, aprovechando que **shape** retorna una tupla con los dos valores, mismos que pueden ser consultados usando el subíndice base cero de la tupla.

```
# Imprime el número de filas del DataFrame,
# usando len()
print(len(titanic))

# Imprime el número de filas del DataFrame,
# usando el elemento 0 de shape
print(titanic.shape[0])

# Imprime el número de filas del DataFrame,
# usando el elemento 1 de shape
print(titanic.shape[1])
```

```
1310
1310
14
```

Analizar si la forma del DataFrame es apropiada

En el caso, tenemos lo siguiente:

1. En el objetivo de análisis se detectaron 6 variables: 1 variable dependiente, 5 variables independientes.
2. El universo de datos es de 2,225, y el tamaño de la muestra estadística es de 511 observaciones, para un 99% de nivel de confianza y 5% de margen de error.

Aun sin ver los datos, podemos decir que la forma mínima que debería tener el conjunto de datos debería ser de 511 filas y 6 columnas. Menos que eso, podríamos creer que los datos no son suficientes para trabajos de analítica.

Dado que tenemos 1310 observaciones y necesitamos 511, podemos decir que son suficientes en apariencia; de hecho, andamos bastante holgados de registros, y podemos darnos el lujo de eliminar datos que contengan datos faltantes o atípicos (porque el análisis no se enfoca en ello).

Dado que se requieren 6 variables y tenemos 14, podemos decir que son suficientes en apariencia.

FIN DEL LAB

CAPÍTULO 5:

Análisis de variables

Tipos de datos en pandas

Toma en cuenta que pandas admite varios tipos de datos en las columnas de un DataFrame, que incluyen:

1. **`float/float64`:** para números decimales.
2. **`int/int64`:** para números enteros.
3. **`object`:** para cadenas de texto o una mezcla de diferentes tipos de datos.
4. **`bool`:** para valores booleanos Verdadero/Falso.
5. **`datetime/datetime64`:** para valores de fecha y hora.
6. **`timedelta/timedelta[ns]`:** para valores de duración, es decir, la cantidad de tiempo que transcurre entre dos fechas.
7. **`category`:** para valores categóricos, que son datos que pertenecen a un conjunto limitado de categorías.

Semántica de los datos

La ***semántica de datos*** es la parte del proceso donde se etiende el significado de los datos, su clasificación y su interdependencia.

La *semántica* tiene que ver con el significado de las cosas. En el caso de los datos, implica saber qué tipo de dato es, cuál es su uso, y cuál es su participación dentro de un modelo de datos, en su caso.

Para un adecuado análisis semántico, lo ideal es tener a la mano:

1. Un ***diccionario de datos***, que es una plantilla columnar que enumera los datos, y sus características, como por ejemplo sus tipos de dato, su uso, su participación en un modelo de datos, su propiedad de unicidad, su capacidad para aceptar valores nulos (*nullable*).
2. Un ***diagrama de entidad relación***, que es una representación estándar del modelo relacional, donde se ilustran las tablas de datos, su relación y dependencia, su cardinalidad y opcionalidad (en caso de poseer más de una fuente dedatos).
3. La ***clasificación DTXC*** (***data taxonomic code***)de los datos, para entender qué tipo de variable son, su tipo de datos, su uso, y su relacionalidad, en un solo código.

Cuando las organizaciones no tienen esta documentación, con el paso del tiempo pueden llegar a la situación en la cual están actualizando datos que ni siquiera saben qué son y por qué se ocupan. Es común que las personas que participaron en el diseño de las bases de datos hayan salido ya de la organización, y se hayan llevado con ellos el conocimiento de la estructura.

Análisis de la estructura de datos

El ***análisis de la estructura de datos*** es la parte del proceso donde se enumeran las columnas del conjunto de datos, su tipo de datos, y se conoce las particularidades teóricas de su contenido.

En el análisis de la estructura, se enumeran las columnas disponibles en el conjunto de datos, y se recurre a los diccionarios de datos, diagramas y documentación varia, para saber lo que debería contener la fuente.

Desde el punto de vista técnico, conocer las columnas de un DataFrame es sencillo, mostrando la propiedad **`dtypes`**. Suponiendo que tuviéramos un DataFrame llamado **`df`**, sería:

```
df.dtypes
```

Toma en cuenta que no siempre el conjunto de datos contiene lo que debería contener. Por ejemplo: si tienes un catálogo de productos, la columna **`codigo_producto`** debe contener en teoría los códigos de producto, sin embargo, existe la posibilidad de que la columna esté vacía, o que tenga un mismo número repetido en todos los registros. En ese caso, la diferencia entre lo que debería ser y lo que es forma parte de la higiene de datos que debemos realizar sobre la fuente.

El análisis de la estructura nos da un punto de referencia para comparar los datos contenidos en el conjunto de datos, y poder sostener que son correctos o no.

Es importante saber qué contienen los campos, pues de otra manera no podemos saber si tenemos los datos requeridos por el análisis.

Verificación de relevancia de variables

La ***verificación de variables*** es la parte del proceso donde se revisa si el conjunto de datos contiene columnas equivalentes para cada una de las variables requeridas por el objetivo de análisis, y por las hipótesis del caso. En esta parte se identifican las columnas requeridas, requeridas indirectas, requeridas inviables y no requeridas.

Lo primero que debemos saber es qué campos tenemos, y qué tipo de datos son. También es importante saber qué datos se requieren, pero no se tienen. Existe la posibilidad de que los datos faltantes puedan generarse a partir de lo que sí se tiene.

No siempre el conjunto de datos contiene todas las variables requeridas por el objetivo de análisis o las hipótesis desarrolladas. Por ese motivo, es indispensable validar si, para cada una de las variables requeridas, se tiene una columna correspondiente en el conjunto de datos.

1. ¿Qué hacer si una variable requerida por el objetivo de análisis o las hipótesis no está presente en el conjunto de datos?

    a. En ese caso, se debe buscar en más fuentes de datos distintas, tratando de encontrar el dato que no tenemos; en caso de que sea imposible obtener los datos, es necesario hacer mención de ello en el informe de análisis, y concluir que hay cosas que no se pueden analizar por falta de datos.

    b. Una alternativa es modificar el objetivo, para que ajuste a las variables existentes.

    c. Una alternativa es modificar, o de plano, eliminar hipótesis.

2. ¿Qué hacer si una variable existe, pero no es requerida por el objetivo de análisis o las hipótesis?

    a. En ese caso, se debe considerar quitarla, a menos que sea necesaria para el cálculo o inferencia de una variable requerida.

Es importante seguir una secuencia de análisis: primero se evalúa la disponibilidad de las variables dependientes, y luego las independientes. Esto es así porque, de no existir columnas que contengan las variables dependientes, es muy probable que el análisis carezca de sentido.

Datos requeridos

Son ***datos requeridos*** aquellos que de forma evidente forman parte del objetivo de análisis, o de las hipótesis.

Datos requeridos indirectos

Son ***datos requeridos indirectos*** aquellos datos que no son requeridos, pero contribuyen a inferir o calcular datos que sí son requeridos. Por ejemplo, si en un conjunto de datos tengo una columna llamada `edad`, que es requerida, la columna `fecha_nacimiento` sería un *requerido indirecto*, porque a partir de dicho dato podemos generar la edad.

Para identificar los campos requeridos indirectos, se debe hacer una valoración de las columnas disponibles en el conjunto de datos, que podrían ser inferidos o calculados de alguna manera.

Es importante mencionar que puede presentarse el escenario donde un requerido indirecto debe inferirse a partir de otros requeridos indirectos.

Las columnas pueden eliminarse una vez que hayan sido utilizadas para inferir los datos requeridos correspondientes.

Datos requeridos inviables

Son ***datos requeridos inviables*** son aquellos datos requeridos que no hay manera de subsanar de ninguna manera. Sus características son las siguientes:

1. Cumplen con los requisitos de los datos requeridos, es decir, de forma evidente están en el objetivo de análisis o en las hipótesis.
2. No se tienen datos actualmente.
3. No hay requeridos indirectos que nos permitan calcularlos o inferirlos.

Ante este tipo de datos, las alternativas son buscar más fuentes de datos. En caso de no encontrar los datos, lo que resta es eliminar la participación del dato no viable del objetivo de análisis y de las hipótesis.

Una muy buena práctica es no aceptar objetivos de análisis sin haber evaluado la viabilidad de los datos. Cuando alguien pide un informe, el analista debe ser cauteloso, y agregar siempre la frase "si disponemos de todos los datos requeridos, podemos generar el informe", para evitar comprometerse a generar un informe para el cual no tiene los datos. A las prácticas que

realiza el analista de datos para comprometerse solamente a la generación de informes que son viables, se le llama ***control de expectativa de análisis***, en donde aterriza al cliente de la información respecto a lo que se puede y lo que no se puede obtener.

Hay que tener cuidado con los ***requeridos inviables estacionales***, que son datos inviables en un determinado contexto. Por ejemplo, puede ser que se tengan los datos de las ventas, pero de solo de un rango de fechas. Puede ser también un dato que se haya agregado a un sistema, de una fecha dada hacia adelante, o que se haya descontinuado su actualización, a partir de una fecha.

En esos supuestos, tiene que observarse los periodos en los cuáles se tienen todos los datos, pues es en ese periodo donde el análisis será confiable.

No requeridos

Son ***datos no requeridos*** aquellos datos que no son requeridos, ni requeridos indirectos.

Estas columnas pueden eliminarse del conjunto de datos, pues no participarán de ninguna manera en el análisis.

FIGURA 03.02: *Verificación de relevancia de variables.*

LAB 05.01: Análisis semántico de las variables.

En este Lab se revisan los campos requeridos por el objetivo de análisis y las hipótesis, y se identifican las variables requeridas, requeridas indirectas, y no requeridas.

Las tareas por realizar son:

1. Revisar las columnas y tipos de datos en un DataFrame.
2. Analizar el contenido teórico de las columnas.
3. Revisar la cobertura de variables dependientes e independientes.
4. Hacer la especificación de los datos que no se tienen.
5. Enumerar las variables requeridas.
6. Enumerar las variables requeridas indirectas.
7. Enumerar las variables requeridas inviables.
8. Enumerar las variables no requeridas.

Revisar las columnas y tipos de datos en un DataFrame.

```
# Datos base
import pandas as pd

pasajeros_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
pasajeros_titanic.csv'

titanic = pd.read_csv(pasajeros_titanic_csv)

# Enumerar los campos de un DataFrame, y sus tipos de datos
# usando dtypes.

titanic.dtypes
```

```
clase_viaje      float64
sobrevivencia    float64
nombre            object
sexo              object
edad             float64
parientes          int64
familiares         int64
boleto            object
tarifa           float64
cabina            object
embarque          object
bote              object
cuerpo           float64
residencias       object
dtype: object
```

Analizar el contenido teórico de las columnas

Los campos del DataFrame, en el caso del ejemplo, son los siguientes, y este es su contenido:

1. **`clase_viaje`**: Indica en qué clase viajaba el pasajero. Puede ser **`1`**, **`2`**, o **`3`**, que corresponde a **`PRIMERA CLASE`**, **`SEGUNDA CLASE`** y **`TERCERA CLASE`**, respectivamente.
2. **`sobrevivencia`**: Indica si la persona murió o vivió. Puede ser 0 o 1, que corresponde a **`MUERTO`** y **`VIVO`**, respectivamente.
3. **`nombre`**: Es el nombre del pasajero. Incluye título personal.
4. **`sexo`**: Indica el sexo de la persona. Puede ser 'hombre' o 'mujer'.
5. **`edad`**: Edad en años del pasajero.
6. **`parientes`**: Número de parientes políticos que viajaban con el pasajero.
7. **`familiares`**: Número de parientes consanguíneos que viajaban con el pasajero.
8. **`boleto`**: Folio del boleto que adquirió el pasajero.
9. **`tarifa`**: Precio que pagó por el boleto el pasajero.
10. **`cabina`**: En el caso de primera clase y segunda clase, es el número de cabina donde se hospedaba.

11. **embarque**: Identificador del puerto donde se embarcó el pasajero. Puede ser **Q**, **S** o **C**, que corresponde a **QUEENSTOWN**, **SOUTH HAMPTON** y **CHERSBOURG**, respectivamente.
12. **bote**: Folio del bote al que se subió el pasajero. Todos los que subieron a bote, se salvaron.
13. **cuerpo**: Folio del cuerpo, asignado a las personas que murieron. Todos los que tienen folio de cuerpo, murieron.
14. **residencias**: Información del lugar donde tiene su casa y reside el pasajero.

Revisar la cobertura de variables dependientes e independientes

En nuestro caso, verificamos primero que haya una variable por cada variable dependiente.

1. Variables dependientes del modelo:
    a. sobrevivencia (**sobrevivencia**)
2. Verificamos después que haya una variable por cada variable independiente.
3. Variables independientes del modelo:
    a. Clase de pasajero (**clase_viaje**)
    b. Sexo (**sexo**)
    c. Rango de edad ***No existe***
    d. Religión ***No existe***
    e. Acompañado ***No existe***

Hacer la especificación de los datos que no se tienen

De las variables enumeradas, hay algunas que no existen en el conjunto de datos, así que definimos cómo deben ser:

1. **rango_edad**: Es un categórico descriptivo, que se asigna en función al valor de la columna edad, y a la siguiente tabla.

Categoría	Rango de edad
INFANTES	[0,12)
JÓVENES	[12,21)
ADULTOS	[21,35)
MEDIANA EDAD	[35,45)
ADULTOS MAYORES	[45,INF)

1. **religión**: Es un categórico descriptivo, que indica la religión del pasajero.
2. **acompañado**: Es un categórico dicotómico, que asume el valor de **SÍ**, si la persona viajaba acompañada, y **NO**, si la persona viajaba sola.

Enumerar las variables requeridas.

En el caso, las columnas requeridas son:

1. **sobrevivencia**
2. **clase_viaje**
3. **sexo**
4. **rango_edad** (**No existe**)
5. **religión** (**No existe**)
6. **acompañado** (**No existe**)

Enumerar las variables requeridas indirectas.

En nuestro caso, estos son los requeridos indirectos:

1. **sobrevivencia**: Se puede inferir usando las columnas **bote** y **cuerpo**; si el registro tiene valor en **bote**, quiere decir que la persona

sobrevivió, así que **sobrevivenicia** es **1**; si el registro tiene valor en **cuerpo**, quiere decir que la persona no sobrevivió, así que **sobrevivencia** es **0**.

2. **clase_viaje**: Con la información que tenemos, no hay manera de inferir la clase en que viajaba el pasajero.
3. **sexo**: Se puede inferir a partir de la información contenida en **nombre**, aunque no de manera exacta. Si el nombre es femenino, podemos afirmar que **sexo** es mujer; si el nombre es masculino, podemos afirmar que el **sexo** es hombre. Si el nombre proporciona título personal, puede ayudar aún más a la tarea.
4. **rango_edad**: Se puede inferir a partir de **edad**.
5. **religión**: Con la información que tenemos no hay manera de inferir la religión, y en virtud de que el dato no existe actualmente, se descarta su uso, por inexistente.
6. **acompañado**: Se puede inferir a partir de **familiares** y **parientes**. Si sumamos el valor de ambas columnas, y la suma es mayor a cero, es que **acompañado** es **SÍ**, o de lo contrario, **NO**. Se puede crear incluso un columna que almacene la cantidad de acompañantes, que sería la suma de **familiares** y **parientes**.

Las variables ***requeridas indirectas*** para el caso de ejemplo serían:

Requerido indirecto	Columna que ayuda a inferir
bote	**sobrevivencia**
cuerpo	**sobrevivencia**
nombre	**sexo**
edad	**rango_edad**
familiares	**acompañantes**
parientes	**acompañantes**
acompañantes	**acompañado**

Enumerar las variables requeridas inviables.

En el caso de ejemplo, son requeridos inviables:

1. **religión**.

Dado que **`religion`** no se tiene y no hay manera de inferir dicho dato, se descarta, y todas las partes del análisis que involucra dicha variable.

1. El objetivo de análisis queda como sigue, eliminando el tema de la religión: Considerando la información histórica contenida en la enciclopedia británica, relativa a personas que iban en el TITANIC y la suerte que tuvieron, queremos hacer un análisis exploratorio histórico que nos permita saber cómo afectaron a la sobrevivencia de las personas, aspectos como la clase de pasajero, sexo, rango de edad, ~~religión,~~ y si la persona sola o acompañada.
2. La hipótesis 4, relacionada con la religión, se elimina: ~~**H4:** La religión del pasajero, no tuvo que ver con la probabilidad de que viviera o muriera.~~

Enumerar las variables no requeridas.

En el caso de ejemplo tenemos los siguientes campos **no requeridos**.

1. **`boleto`**
2. **`tarifa`**
3. **`cabina`**
4. **`embarque`**
5. **`residencias`**

FIN DEL LAB

Clasificación de los datos

Categorías de los datos

La ***categorización de datos*** es la parte del proceso donde se especifica el tipo de dato que *debe* tener cada columna para cumplir con su cometido de la mejor manera. En ocasiones, si no se sabe o no se conoce el uso de un dato, puede clasificarse como está (*as-is*).

La clasificación de los datos puede darse desde diferentes aspectos o categorías:

1. ***Tipo de variable*:** Puede ser variable *cualitativa* o *cuantitativa*.
    a. Las variables cualitativas pueden ser, a su vez, *ordinales* o *nominales*.
    b. Las variables cuantitativas pueden ser, en cuanto a su naturaleza, *continuas* o *discretas*.
    c. Las variables cuantitativas pueden ser, en cuanto a su escala, de *intervalo* o de *razón*.

2. ***Tipo de dato (datatype)*:** Pueden ser int, float, bool, datetime, binary, etcétera.
3. ***Uso*:** En cuanto a su uso, pueden ser de identidad, descriptivos, categóricos, de valor, temporales.
    a. A su vez, los categóricos pueden ser numéricos, codificados, descriptivos, dicotómicos y de intervalo.

4. ***Origen*:** Puede ser que los datos ya se tengan, que se calculen, que no estén disponibles.
5. ***Relacionalidad*:** En cuanto a su relacionalidad (participación en un modelo de datos relacional), pueden ser atributos primos, atributos no primos, atributos estándar, etcétera.

Data Taxonomic Code (DTXC)

El ***código DTXC*** permite clasificar, a través de un código, cada columna de un conjunto de datos.

El código DTXC se compone de varias partes:

```
<Medida>-<Tipo de dato>-<Uso>-<Origen>-<Relacionalidad>
```

El guion intermedio puede cambiarse por una diagonal (/) o diagonal invertida (\ ), en caso de requerirse.

TIPOS DE MEDIDA (*Measurement type*)

Categoría	Categoría en inglés	Código DTXC
Cuantitativo	*Quantitative*	**QT**
Cuantitativo Discreto	*Quantitative Discrete*	**QT-DIS**
Cuantitativo Continuo	*Quatitative Continuous*	**QT-CON**
Cuantitativo Discreto de Escala de Intervalo	*Quantitative Discrete, Interval Scale*	**QT-DIS-IS**
Cuantitativo Discreto de Escala de Razón	*Quantitative Discrete, Ratio Scale*	**QT-DIS-RS**
Cuantitativo Continuo de Escala de Intervalo	*Quantitative Continuous, Interval Scale*	**QT-CON-IS**
Cuantitativo Continuo de Escala de Razón	*Quantitative Continuous, Ratio Scale*	**QT-CON-RS**
Cualitativo	*Qualitative*	**QL**
Cualitativo Nominal	*Qualitative Nominal*	**QL-NM**
Cualitativo Ordinal	*Qualitative Ordinal*	**QL-OR**
Tipo de medida no definido	*Not Defined Measurement Type*	**NDMT**

TIPO DE DATO (*Datatype*)

Categoría	Categoría en inglés	Código DTXC
Numérico	*Numeric*	**NUM**
Numérico Flotante	*Numeric Float*	**NUM-FL**
Numérico Entero	*Numeric Integer*	**NUM-INT**
Alfanumérico	*Alphanumeric*	**STR**
Booleano	*Boolean*	**BL**
Marca de tiempo	*Time Stamp*	**TS**
Binario	*Binary*	**BIN**
Binario Muy Largo	*Binary - Binary Large Object*	**BIN-BLOB**
Tipo de dato no definido	*Not Defined Data Type*	**NDDT**

USO DEL DATO (*Use*)

Categoría	Categoría en inglés	Código DTXC
Identidad	*Identity*	**ID**
Identidad no requerida	*Not required identity*	**NRID**
Categórico	*Categorical*	**CAT**
Categórico Numérico	*Categorical Number*	**CAT-NM**
Categórico Codificado	*Categorical Code*	**CAT-CD**
Categórico Descriptivo	*Categorical Description*	**CAT-DES**
Categórico de intervalo	*Categorical Interval*	**CAT-IV**
Categórico Dicotómico	*Categorical Dichotomic*	**CAT-DIC**
Descriptivo	*Description*	**DS**
Valor	*Value*	**VAL**
Valor detallado	*Detail Value*	**VAL-DET**
Valor agregado	*Aggregate Value*	**VAL-AGG**
Temporalidad	*Time Measure*	**TM**
Tiempo - Fecha/Hora	*Date time*	**TM-DTM**
Tiempo - Fecha	*Date*	**TM-DATE**
Tiempo - Hora	*Just Time*	**TM-TIME**
Uso no definido	*Not Defined Use*	**NDU**

ORIGEN DE DATO (*Source*)

Categoría	Categoría en inglés	Código DTXC
Se tiene	*Already Available*	**AA**
Se calcula	*Calculated Data*	**CAL**
No disponible	*Not Available*	**NA**
Origen no definido	*Not Defined Source*	**NDS**

RELACIONALIDAD DEL DATO (*Relationality*)

Categoría	Categoría en inglés	Código DTXC
Atributo primo	*Prime Attribute*	**PK**
Atributo primo y de llave foránea	*Prime Attribute, Foreign Key Attribute*	**PK–FK**
Atributo de llave foránea	*Foreign Key Attribute*	**FK**
Atributo estándar	*Standard Attribute*	**SA**
Relacionalidad no definida	*Not Defined Relationality*	**NDR**

LAB 05.02: Clasificar los datos usando código DTXC

En este Lab se clasificarán los datos existentes en el DataFrame del caso, determinando el código DTXC que le aplican a cada campo.

Las tareas por realizar son:

1. Definir el código DTXC de cada campo en el DataFrame.

```
# Datos base
import pandas as pd

pasajeros_titanic_csv='https://raw.githubusercontent.com/Apren-
daPracticando/AnaliticaPythonR1/main/data/pasajeros_tita-
nic.csv'

titanic = pd.read_csv(pasajeros_titanic_csv)

# Se revisan las columnas del DataFrame.

titanic.dtypes
```

```
clase_viaje      float64
sobrevivencia    float64
nombre            object
sexo              object
edad             float64
parientes          int64
familiares         int64
boleto            object
tarifa           float64
cabina            object
embarque          object
bote              object
cuerpo           float64
residencias       object
dtype: object
```

En el caso de ejemplo, los datos requeridos son clasificados de la siguiente manera:

Variable	DTXC
clase_viaje	QL-OR-STR-CAT-CD-AA-NDR
sobrevivencia	QL-NM-STR-CAT-CD-AA-NDR
nombre	QL-NM-STR-DS-AA-NDR
sexo	QL-NM-STR-CAT-DES-AA-NDR
edad	QT-CON-NUM-FL-VAL-DET-AA-NDR
parientes	QT-DIS-NUM-INT-VAL-DET-AA-NDR
familiares	QT-DIS-NUM-INT-VAL-DET-AA-NDR
bote	QL-NM-STR-NRID-AA-NDR
cuerpo	QL-NM-STR-NRID-AA-NDR
rango_edad	QL-OR-STR-CAT-DES-AA-NDR
acompañado	QL-NM-STR-CAT-DES-CAL-NDR
acompañantes	QT-DIS-NUM-INT-VAL-DET-AA-NDR

Es importante mencionar que los códigos DTXC se determinan respecto a cómo debe ser el dato, y no cómo es.

FIN DEL LAB

CAPÍTULO 6:

Visualización y filtrado de datos

Ver los datos del DataFrame

Ver todos los datos del Data Frame

Se pueden ver todos los datos del conjunto, invocando simplemente el nombre del DataFrame.

Suponiendo que tenemos un DataFrame llamado **`df`**, sería:

```
df
```

También se puede imprimir el contenido del DataFrame:

```
print(df)
```

Ver los encabezados y colas de datos

Se pueden ver todos los datos iniciales o finales del conjunto, usando los métodos **`head()`** y **`tail()`**.

Si queremos mostrar los primeros 5 registros del DataFrame, usamos **`head()`**. Suponiendo que tenemos un DataFrame llamado **`df`**, sería:

```
df.head()
```

Si queremos mostrar los últimos 5 registros del DataFrame, usamos **`tail()`**

```
df.tail()
```

La cantidad de registros a mostrar se puede configurar, pero por omisión son 5 filas. Si se desea, se puede especificar cuántos registros se quieren mostrar. Por ejemplo, si se quieren mostrar las primeras 15 filas, sería así.

```
df.head(15)
```

Para mostrar las últimas 15 filas, sería así.

```
df.tail(15)
```

Filtrado de columnas

Al momento de ver los datos, pueden aplicarse filtros, ya sea a nivel columnas, o a nivel de filas.

En cuanto a las columnas, se tienen varias opciones.

- **`DataFrame`**: Muestra todas las columnas.
- **`DataFrame['columna']`** o **`DataFrame.columna`**: Muestra una sola columna.
  - Se puede usar la ***notación regular*** de referencia de columnas, o ***dot notation***.
- **`DataFrame[lista_columnas]`**: Muestra dos o más columnas.

Por eficiencia, no se muestran todos los datos del DataFrame, a menos que se configure al sistema para que lo haga así. Generalmente se truncan los datos, para economizar espacio.

Vamos a suponer que tenemos un DataFrame llamado **`df`**, con las columnas **`col1`**, **`col2`**, **`col3`**, **`col4`**, **`col5`**.

Para ver todas las columnas, no se tiene que especificar nada.

```
df
```

Para ver una sola columna, se especifica el nombre de la columna a mostrar.

```
df['col1']
```

Para referir una sola columna, también se puede utilizar *dot notation*.

```
df.col1
```

Generalmente se utiliza la referencia normal, debido a que usar *dot notation* puede ser confuso, ya que es la forma en que se refieren las propiedades y los métodos.

Para ver múltiples columnas, se proporciona la lista de columnas que se desea mostrar, en forma de lista.

```
columnas_deseadas=['col1', 'col4', 'col5']
df[columnas_deseadas]
```

O también:

```
df[['col1', 'col4', 'col5']]
```

LAB 06.01: Ver datos del DataFrame

En este Lab se revisan estrategias para visualizar datos de un DataFrame.

Las tareas por realizar son:

1. Ver todos los datos de un DataFrame.
    a. Ver las filas en los extremos de un DataFrame.
    b. Ver las primeras filas de un DataFrame.
2. Ver las últimas 13 filas de un DataFrame.
3. Ver el contenido de una columna.
    a. Ver el contenido de una columna, usando notación estándar.
    b. Ver el contenido de una columna, usando *dot notation*.
4. Ver el contenido de solo algunas columnas.

El símbolo ▸ indica que hay más columnas, que se omiten por espacio.

Ver todos los datos de un DataFrame.

```
# Datos base
import pandas as pd
pasajeros_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
pasajeros_titanic.csv'
titanic = pd.read_csv(pasajeros_titanic_csv)

# Ver todos los campos del DataFrame titanic
titanic
```

```
      clase_viaje  sobrevivencia                                 nombre  \
0             1.0            0.0                  Andrews, Mr. Thomas Jr  ▶
1             1.0            0.0  Chisholm, Mr. Roderick Robert Crispin  ▶
2             1.0            0.0                          Fry, Mr. Richard  ▶
3             1.0            0.0                   Harrison, Mr. William  ▶
4             1.0            1.0                 Ismay, Mr. Joseph Bruce  ▶
...           ...            ...                                     ...
1307          NaN            0.0                     Baumann, Mr. John D  ▶
1308          NaN            NaN             Blackwell, Mr. Stephen Weart  ▶
1309          NaN            0.0                 Baxter, Mr. Quigg Edmond  ▶
[1310 rows x 14 columns]
```

Ver las filas en los extremos de un DataFrame

1. **Ver las primeras filas de un DataFrame.**

```
# Ver las primeras 5 filas del DataFrame titanic.
titanic.head()
```

```
   clase_viaje  sobrevivencia                                 nombre    sexo ▶
0          1.0            0.0                  Andrews, Mr. Thomas Jr  hombre ▶
1          1.0            0.0  Chisholm, Mr. Roderick Robert Crispin  hombre ▶
2          1.0            0.0                          Fry, Mr. Richard  hombre ▶
3          1.0            0.0                   Harrison, Mr. William  hombre ▶
4          1.0            1.0                 Ismay, Mr. Joseph Bruce  hombre ▶
```

```
   clase_viaje  sobrevivencia                                 nombre    sexo ▶
0          1.0            0.0                  Andrews, Mr. Thomas Jr  hombre ▶
1          1.0            0.0  Chisholm, Mr. Roderick Robert Crispin  hombre ▶
2          1.0            0.0                          Fry, Mr. Richard  hombre ▶
3          1.0            0.0                   Harrison, Mr. William  hombre ▶
4          1.0            1.0                 Ismay, Mr. Joseph Bruce  hombre ▶
```

2. Ver las últimas 13 filas de un DataFrame.

```
# Ver las últimas 13 filas del DataFrame titanic.
titanic.tail(13)
```

```
      clase_viaje  sobrevivencia                          nombre    sexo ▶
1297          3.0            0.0                Sage, Miss. Ada   mujer ▶
1298          3.0            0.0       Sage, Miss. Constance Gladys   mujer ▶
1299          3.0            0.0  Sage, Miss. Dorothy Edith "Dolly"   mujer ▶
1300          3.0            0.0            Sage, Miss. Stella Anna   mujer ▶
1301          3.0            0.0          Sage, Mr. Douglas Bullen  hombre ▶
1302          3.0            0.0               Sage, Mr. Frederick  hombre ▶
1303          3.0            0.0          Sage, Mr. George John Jr  hombre ▶
1304          3.0            0.0             Sage, Mr. John George  hombre ▶
1305          3.0            0.0    Sage, Mrs. John (Annie Bullen)   mujer ▶
1306          3.0            0.0                Storey, Mr. Thomas  hombre ▶
1307          NaN            0.0                Baumann, Mr. John D  hombre ▶
1308          NaN            NaN      Blackwell, Mr. Stephen Weart  hombre ▶
1309          NaN            0.0          Baxter, Mr. Quigg Edmond  hombre ▶
```

Ver el contenido de una columna.

1. Ver el contenido de una columna, usando notación estándar.

```
# Ver el contenido de la columna nombre
# usando notación estándar (entre square brackets)
titanic['nombre']
```

```
0                     Andrews, Mr. Thomas Jr
1       Chisholm, Mr. Roderick Robert Crispin
2                           Fry, Mr. Richard
3                       Harrison, Mr. William
4                      Ismay, Mr. Joseph Bruce
                        ...
1307                     Baumann, Mr. John D
1308              Blackwell, Mr. Stephen Weart
1309                  Baxter, Mr. Quigg Edmond
Name: nombre, Length: 1310, dtype: object
```

2. **Ver el contenido de una columna, usando dot notation.**

```
# Ver el contenido de la columna nombre,
# usando dot notation.
titanic.nombre
```

```
0                      Andrews, Mr. Thomas Jr
1       Chisholm, Mr. Roderick Robert Crispin
2                           Fry, Mr. Richard
3                     Harrison, Mr. William
4                    Ismay, Mr. Joseph Bruce
                          ...
1305           Sage, Mrs. John (Annie Bullen)
1306                      Storey, Mr. Thomas
1307                    Baumann, Mr. John D
1308              Blackwell, Mr. Stephen Weart
1309                  Baxter, Mr. Quigg Edmond

Name: nombre, Length: 1310, dtype: object
```

Ver el contenido de solo algunas columnas.

```
# Ver el contenido de las columnas clase_viaje, nombre,
# y tarifa, solamente.
titanic[['clase_viaje', 'nombre', 'tarifa']]
```

```
      clase_viaje                                nombre    tarifa
0             1.0                 Andrews, Mr. Thomas Jr    0.0000
1             1.0  Chisholm, Mr. Roderick Robert Crispin    0.0000
2             1.0                       Fry, Mr. Richard    0.0000
3             1.0                  Harrison, Mr. William    0.0000
4             1.0                Ismay, Mr. Joseph Bruce    0.0000
...           ...                                    ...       ...
1305          3.0         Sage, Mrs. John (Annie Bullen)   69.5500
1306          3.0                     Storey, Mr. Thomas       NaN
1307          NaN                   Baumann, Mr. John D   25.9250
1308          NaN           Blackwell, Mr. Stephen Weart   35.5000
1309          NaN               Baxter, Mr. Quigg Edmond  247.5208
[1310 rows x 3 columns]
```

FIN DEL LAB

Técnicas de filtrado de filas

Operadores comparativos en pandas

Los más importantes ***operadores comparativos*** en pandas son estos:

Operador	Uso
`==`	Igual a.
`!=`	Diferente de.
`<`	Menor que.
`>`	Mayor que.
`<=`	Menor o igual que.
`>=`	Mayor o igual que.
`isin()`	Comprueba si los valores están presentes en una lista.
`between()`	Comprueba si los valores están dentro de un rango especificado.
`notnull()`	Comprueba si los valores no son nulos.
`isnull()`	Comprueba si los valores son nulos.

Operadores lógicos en pandas

Toma en cuenta que se puede colocar más de una condición de filtrado en pandas, siempre y cuando las condiciones involucradas se encierren entre paréntesis.

Los operadores **`and`**, **`or`** y **`not`** de Python no se utilizan, porque están diseñados para su uso con datos escalares; pandas tiene sus propios operadores, ajustados a una forma de trabajo vectorizada.

Los más importantes ***operadores lógicos*** en pandas son los siguientes:

Operador	Uso
`&`	Operador **`and`**.
`|`	Operador **`or`**.
`~`	Operador **`not`**.

Filtrando filas usando loc[]

Para filtrar las filas, se utiliza la función **`loc[]`** (*localize*).

Entre las llaves se coloca la condición o condiciones a aplicar.

Las ***condiciones*** en pandas generalmente involucran referencias a columnas, más *operadores comparativos* y *lógicos*.

Las condiciones lógicas deben encerrarse entre paréntesis, en caso de que la condición sea múltiple (varias condiciones).

`loc[]` retorna las filas que cumplan con los criterios lógicos. Es muy común que se almacene el resultado de **`loc[]`** a un nuevo DataFrame.

Vamos a suponer que tenemos un DataFrame llamado **`df`**, con las columnas **`col1`**, **`col2`**, **`col3`**, **`col4`**, **`col5`**.

Si quisiéramos crear un DataFrame llamado **`datos`**, que contenga solo las filas en donde **`col1`** sea igual a cero, sería:

```
datos=df.loc[df['col1']==0]
```

Cuando sólo se tiene una condición lógica que cumplir, se puede colocar la condición sin paréntesis. Por otro lado, si se requieren dos o más condiciones, éstas sí deberán encerrarse entre paréntesis.

Si quisiéramos crear un DataFrame llamado **`datos`**, que contenga solo las filas en donde **`col1`** sea igual a cero, y **`col2`** sea igual a **`'N'`**, sería:

```
datos=df.loc[(df['col1']==0) &
      df['col2']=='N')]
```

Si quisiéramos crear un DataFrame llamado **`datos`**, que contenga solo las filas en donde **`col2`** sea igual a **`'A'`**, **`'B'`**, o **`'C'`**, sería:

```
valores_validos=['A','B','C']
datos=df.loc[df['col2'].isin(valores_validos)]
```

Se puede mezclar el filtrado de columnas, junto con el filtrado de filas. Si quisiéramos crear un DataFrame llamado **`datos`**, que contenta solo las columnas **`col1`** y **`col1`**, y que contenga solo las filas en donde **`col1`** sea igual a cero, sería:

```
datos=df[['col1','col2']].loc[df['col1']==0]
```

Filtrando filas usando where()

La función **`where()`** al igual que **`loc[]`**, permite asignar filtros a nivel filas.

La diferencia es que en el caso de **`where()`**, cambia los valores que no cumplen una condición, con **`NaN`**. Como sabemos, **`NaN`** es el acrónimo de *Not a Number*, que es uno de los valores que toma pandas como nulos.

Mientras que **`loc[]`** solo retorna las filas que cumplen con las condiciones, **`where()`** retorna todas las filas, pero aquellas filas en donde la condición no se cumple, se coloca **`NaN`**.

LAB 06.02: Técnicas de filtrado de filas

En este Lab se revisan estrategias para filtrar filas usando **`loc[]`** y **`where()`**.

Las tareas por realizar son:

1. Filtrado usando **`loc[]`**, con una condición.
2. Filtrado usando **`loc[]`**, con varias condiciones.
3. Filtrado usando **`loc[]`**, combinado con filtrado de columnas.
4. Filtrado usando **`loc[]`**, con **`isin()`**.
    a. Valores incluidos en una lista.
    b. Valores no incluidos en una lista.
5. Filtrado usando **`where()`**.

El símbolo ▸ indica que hay más columnas, que se omiten por espacio.

```
# Datos base
import pandas as pd
pasajeros_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
pasajeros_titanic.csv'
titanic = pd.read_csv(pasajeros_titanic_csv)
```

Filtrado usando loc[], con una condición

```
# Mostrar las filas donde clase_viaje sea 1.0, usando loc[].
# Como sólo es una condición, no se requiere encerrar la
# condición entre paréntesis.
primera_clase=titanic.loc[titanic['clase_viaje']==1.0]

primera_clase
```

```
     clase_viaje  sobrevivencia  \
0            1.0            0.0
1            1.0            0.0
2            1.0            0.0
..           ...            ...
317          1.0            1.0
318          1.0            1.0
319          1.0            1.0
[320 rows x 14 columns]
```

Filtrado usando loc[], con varias condiciones

```
# Mostrar las filas donde clase_viaje sea 1.0,
# y sexo sea 'mujer', usando loc[].
# Como se usa más de una condición, éstas se
# encierran entre paréntesis.
mujeres_primera_clase=titanic.loc[
    (titanic['clase_viaje']==1.0) &
    (titanic['sexo']=='mujer')
]

# Se muestran las filas seleccionadas.
mujeres_primera_clase
```

```
     clase_viaje  sobrevivencia   ▶
9            1.0            1.0   ▶
11           1.0            1.0   ▶
12           1.0            1.0   ▶
21           1.0            1.0   ▶
28           1.0            1.0   ▶
..           ...            ...
311          1.0            1.0   ▶
312          1.0            1.0   ▶
315          1.0            1.0   ▶
317          1.0            1.0   ▶
319          1.0            1.0   ▶

[143 rows x 14 columns]
```

Filtrado usando loc[], combinado con filtrado de columnas

```
# Combinación de filtrado de columnas y filas.
# Mostrar el nombre y la edad, de las filas
# donde sexo sea mujer.
columnas_filas=titanic[['nombre',
                'edad']].loc[titanic['sexo']=='mujer']

columnas_filas
```

```
                                                nombre  edad
9     Cornell, Mrs. Robert Clifford (Malvina Helen L...  55.0
11                          Leader, Dr. Alice (Farnham)  49.0
12    Swift, Mrs. Frederick Joel (Margaret Welles Ba...  48.0
21                         Newsom, Miss. Helen Monypeny  19.0
28                             Bonnell, Miss. Elizabeth  58.0
...                                                 ...   ...
1297                                   Sage, Miss. Ada   NaN
1298                       Sage, Miss. Constance Gladys   NaN
1299                  Sage, Miss. Dorothy Edith "Dolly"   NaN
1300                            Sage, Miss. Stella Anna   NaN
1305                       Sage, Mrs. John (Annie Bullen)   NaN
[465 rows x 2 columns]
```

Filtrado usando loc[], con isin().

1. **Valores incluidos en una lista.**

```
# Uso de filtro, comparando con el contenido de una lista.
# Recuperar las filas donde las cabinas sean C32, C26 y C80.
# Mostrar sólo las columnas nombre y cabina.
cabinas_especiales=['C32','C26','C80']

especiales=titanic[['nombre','cabina']].
      loc[titanic['cabina'].isin(cabinas_especiales)]

especiales
```

```
                                       nombre cabina
263        White, Mrs. John Stuart (Ella Holmes)    C32
264                     Young, Miss. Marie Grice    C32
289                  Widener, Mr. George Dunton    C80
291  Widener, Mrs. George Dunton (Eleanor Elkins)    C80
```

2. **Valores no incluidos en una lista.**

```
# Uso de filtro, comparando con el contenido de una lista.
# Recuperar las filas donde las cabinas NO sean C32, C26 y C80.
# Mostrar sólo las columnas nombre y cabina.
cabinas_especiales=['C32','C26','C80']

no_especiales=titanic[['nombre','cabina']].
          loc[~titanic['cabina'].isin(cabinas_especiales)]

no_especiales
```

```
                                    nombre       cabina
0                    Andrews, Mr. Thomas Jr          A36
1     Chisholm, Mr. Roderick Robert Crispin          NaN
2                          Fry, Mr. Richard         B102
3                      Harrison, Mr. William          B94
4                   Ismay, Mr. Joseph Bruce  B52 B54 B56
...                                    ...          ...
1305         Sage, Mrs. John (Annie Bullen)          NaN
1306                     Storey, Mr. Thomas          NaN
1307                   Baumann, Mr. John D           NaN
1308            Blackwell, Mr. Stephen Weart            T
1309                Baxter, Mr. Quigg Edmond      B58 B60
[1306 rows x 2 columns]
```

Filtrado usando where().

```
# Se filtra el DataFrame usando where, de tal manera que
# se recuperan todas las filas; en aquellos casos donde
# sexo no sea mujer o sobrevivencia no sea 0, las columnas
# son rellenadas con NaN.
mujeres_vivas_where=titanic.where((titanic.sexo=='mujer') &
                                  (titanic.sobrevivencia==1.0))

# Se muestra el resultado. Son 1310 filas de 1310, las
# que se recuperan.
mujeres_vivas_where
```

```
      clase_viaje  sobrevivencia nombre sexo  edad  parientes  familiares  ▶
0             NaN            NaN    NaN  NaN   NaN        NaN         NaN  ▶
1             NaN            NaN    NaN  NaN   NaN        NaN         NaN  ▶
2             NaN            NaN    NaN  NaN   NaN        NaN         NaN  ▶
3             NaN            NaN    NaN  NaN   NaN        NaN         NaN  ▶
4             NaN            NaN    NaN  NaN   NaN        NaN         NaN  ▶
...           ...            ...    ...  ...   ...        ...         ...
1305          NaN            NaN    NaN  NaN   NaN        NaN         NaN  ▶
1306          NaN            NaN    NaN  NaN   NaN        NaN         NaN  ▶
1307          NaN            NaN    NaN  NaN   NaN        NaN         NaN  ▶
1308          NaN            NaN    NaN  NaN   NaN        NaN         NaN  ▶
1309          NaN            NaN    NaN  NaN   NaN        NaN         NaN  ▶
[1310 rows x 14 columns]
```

FIN DEL LAB

CAPÍTULO 7:

Limpieza de datos e ingeniería de datos (feature engineering)

Limpieza de datos (data cleaning)

La ***limpieza de datos*** (***data cleaning***) se enfoca en la limpieza y mantenimiento de la calidad de los datos, asegurando que sean precisos, consistentes y completos, para un fin específico.

Cuando se habla de que los datos deben ser precisos, consistentes y completos, se hace referencia a ciertas características deseables de los datos para asegurar que sean confiables y adecuados para su uso en análisis y modelado.

1. ***Datos precisos*:** Los datos precisos son aquellos que se acercan lo más posible a la realidad o a la verdad. Es decir, los datos no deben contener errores o inexactitudes que puedan distorsionar los resultados de análisis y modelado. Algunas tareas que pueden estar incluidas aquí son:

    a. Verificar que no haya errores en los cálculos.

    b. Verificar que no haya errores en la entrada o lectura de datos.

   c. Verificar que no haya registros duplicados cuando no debe haberlos.

   d. Verificar que los campos que no admitan repetidos, no los tengan.

2. ***Datos consistentes*:** Los datos consistentes son aquellos que se presentan de manera uniforme y coherente en toda la base de datos. Es decir, los datos no deben tener contradicciones o discrepancias que puedan confundir o distorsionar los resultados de análisis y modelado. Por ejemplo, si estamos analizando los datos de clientes de una empresa, es importante que la información de cada cliente se presente de manera consistente y uniforme en toda la base de datos. Algunas tareas que se pueden incluir aquí son.

   a. Verificar que los códigos y sus significados no admitan más de una interpretación.

   b. Verificar un uso estándar de unidades de medida y unidades monetarias.

   c. Verificar que los datos sigan un mismo patrón (mismo uso de mayúsculas, mismo orden de elementos de fecha —dma, amd, mda— etcétera).

3. ***Datos completos*:** Los datos completos son aquellos que no tienen valores faltantes o incompletos. Es decir, todos los campos de datos deben tener información para que se pueda hacer un análisis adecuado. Algunas tareas que se pueden incluir aquí son.

   a. Eliminar filas vacías.

   b. Hacer un adecuado tratamiento de datos ausentes.

El objetivo principal de la limpieza de datos es garantizar que los datos sean confiables y estén listos para su uso en análisis y modelado.

En ocasiones se confunde con el concepto ***higiene de datos*** (***data higiene***). Son muy parecidos, con la diferencia de que la higiene de datos es un proceso continuado y preventivo, es decir, es una labor de mantenimiento y aseguramiento de la calidad de los datos en todo momento. La higiene de datos puede estar presente en las validaciones que hace un sistema, la verificación de ortografía, y todo aquello que asegure que los datos son correctos.

La limpieza de datos sucede cuando los datos ya están almacenados, y requieren correcciones; se realiza generalmente cuando se hace una lectura de datos, como parte del modelo *ETL* (*Extract / Transform / Load*), y suelen ser trabajos de una sola vez.

Ingeniería de datos (feature engineering)

Por otro lado, la ***ingeniería de datos*** (***feature engineering***) se enfoca en la creación y transformación de características (variables) a partir de los datos existentes con el objetivo de mejorar el desempeño de los modelos de analítica.

Las técnicas de ingeniería de datos pueden incluir algunas de las siguientes tareas.

1. ***Normalización de datos*****.** Consiste en preparar los orígenes de datos para que puedan relacionarse adecuadamente, evitando redundancia, inconsistencia o falta de integridad (desde el punto de vista de modelación de bases de datos). Algunas tareas que se pueden incluir aquí son.

    a. Estandarización de nombres de columnas.

    b. Estandarización de códigos para que persista un código, en caso de existir códigos equivalentes o jerarquizados.

    c. Eliminación de redundancias de datos entre orígenes.

    d. Establecimiento de llaves primarias y foráneas que permitan *JOINS*.

e. Verificación de integridad referencial entre las fuentes.

f. Verificación de consistencia entre las fuentes.

2. ***Codificación y descripción de variables categóricas*.** Consiste en crear catálogos de características, y clasificar las observaciones para fines de segmentación. Algunas tareas que se pueden incluir aquí son.

   a. Crear categóricos (numéricos, codificados, descriptivos, dicotómicos).

   b. Crear categóricos de intervalo, a partir de otros datos.

   c. Generar categóricos descriptivos equivalentes, para los otros categóricos que requieran interpretación.

3. ***Creación de características derivadas*.** Creación de nuevas columnas a partir de los datos existentes. También se admite la modificación de las características existentes, a fin de que se ajusten mejor a los trabajos de analítica. Algunas de las tareas que se pueden incluir aquí son.

   a. Generar identificadores ficticios para los conjuntos de datos que no tengan uno.

   b. Generar nuevas columnas, o modificar las existentes, por conversión de tipos.

   c. Generar nuevas columnas, por cálculo a partir de otras columnas.

   d. Generar nuevas columnas, o modificar las existentes, por aplicación de transformaciones (uso de mayúsculas, cambio de orden de elementos de fecha, etcétera).

   e. Generar nuevas columnas por manipulación de cadenas (concatenar, separar, extraer).

4. ***Selección de características y filas relevantes*.** Seleccionar sólo las columnas y registros que son pertinentes para el análisis. Algunas tareas que se pueden incluir aquí son.

    a. Mantener únicamente las columnas requeridas o requeridas indirectas.

    b. Aplicar el filtrado de filas, manteniendo solo las filas que tienen sentido para el análisis.

    c. Hacer adecuado tratamiento de datos atípicos.

El objetivo principal de la ingeniería de datos es aumentar la precisión y el rendimiento de los modelos de analítica, dejando sólo lo que se necesita, en el formato que se necesita.

En resumen, la higiene de datos se enfoca en la limpieza y mantenimiento de la calidad de los datos, mientras que la ingeniería de datos se enfoca en la creación y transformación de características para mejorar el desempeño de los modelos de analítica.

Ambas son importantes para garantizar que los datos sean confiables y estén listos para su uso en análisis y modelado. En teoría, se realizan primero las tareas de higiene, y posteriormente las de ingeniería de datos, aunque llega un momento en que pueden traslaparse, ya que al hacer ingeniería de datos puede surgir una nueva necesidad de hacer higiene de datos.

Como te podrás dar cuenta, algunas tareas se repiten en la higiene de datos y en la ingeniería de datos, pero no te debes confundir: en la higiene de datos, las tareas se aplican para corregir errores, mientras que en la ingeniería de datos las tareas se aplican para mejorar la pertinencia de los datos y el rendimiento del modelo, y no porque los datos estén mal.

Tareas generales con DataFrame

Estas son algunas de las tareas generales que se pueden realizar con DataFrames.

1. ***Eliminar columnas no requeridas*.** Se eliminan las columnas que han sido identificadas como no requeridas.
2. ***Eliminación de filas duplicadas*.** Se eliminan registros duplicados, en caso de que no deban existir.
3. ***Eliminación de filas vacías*.** Se eliminan filas vacías o parcialmente vacías, en caso de que no deban existir.
4. ***Crear un identificador ficticio***. Se crea una nueva columna que es el resultado de crear una secuencia de números sin repetidos, que actúe como llave única de los registros de un conjunto de datos.
5. ***Verificar unicidad de columnas*.** Se verifica que no existan datos repetidos en columnas con unicidad.
6. ***Reordenar las columnas de un DataFrame***. Se reordena el orden en que aparecen las columnas en un DataFrame.
7. ***Cambio de nombres de columna***. Se modifican los nombres de campo o columna, para que sean más descriptivos o manejables.
8. ***Escribir un CSV a partir de un DataFrame***. Se puede guardar el estado en que se encuentra un DataFrame, a un archivo CSV.

Eliminar columnas no requeridas

En esta acción se eliminan las columnas que no aportan nada al trabajo de analítica.

Eliminar usando del

Los datos no requeridos, pueden eliminarse del conjunto de datos usando el estatuto **`del`**.

Vamos a suponer que tenemos un DataFrame llamado **`df`**, con las columnas **`col1`**, **`col2`**, **`col3`**, **`col4`**, **`col5`**.

Para eliminar la **`col1`**, sería:

```
del df['col1']
```

Una de las desventajas de **`del`** es que solo permite eliminar una columna a la vez.

Eliminar usando drop()

Para eliminar una columna usando **`drop()`**.

Para eliminar la columna **`col1`** del DataFrame **`df`**, sería:

```
df.drop(columns=['col1'], inplace=True)
```

Una de las ventajas de **`drop()`** es que se pueden eliminar varias columnas a la vez, colocando más valores en el parámetro **`columns`**.

Para eliminar la columna **`col1`** y **`col4`**, del DataFrame **`df`**, sería:

```
df.drop(columns=['col1','col4'], inplace=True)
```

Pandas no es un manejador de base de datos, y por esa razón, al integrar datos, puedes terminar con dos columnas con el mismo nombre. En ese caso, el uso de `del` puede resultar ambiguo, y para eliminar una columna puede necesitarse usar el número de columna, y no el nombre de la misma, porque es ambiguo.

Para eliminar una columna en pandas usando el número de columna (o el nombre de columna), puede utilizar el método **`drop()`** de pandas junto con el parámetro **`axis=1`**.

Por ejemplo, para eliminar la columna que tenga el índice 3 del DataFrame **`df`**, sería:

```
df.drop(df.columns[3], axis=1, inplace=True)
```

Evitar la necesidad de eliminar

Si de antemano se sabe que hay algunas columnas del origen de datos que no serán utilizadas, en lugar de eliminarlas, podemos simplemente no leerlas. Esto se especifica usando la propiedad **`usecols`**, que nos permite especificar qué columnas son las que queremos recuperar.

Si tenemos un archivo CSV que contiene las siguientes columnas, **`col1`**, **`col2`**, **`col3`**, **`col4`** y **`col5`**, y queremos leer solo las columnas **`col1`**, **`col3`** y **`col4`**, para cargarlas en un DataFrame llamado **`df`**, podríamos utilizar el siguiente código para cargar solo las columnas de nuestro interés:

```
df=pd.read_csv('datos.csv', usecols=['col1', 'col3', 'col4'])
```

Cargar así los datos puede ser una alternativa a cargar todos los datos, y luego tener que eliminar **`col2`** y **`col5`**, que no requerimos.

Eliminación de filas duplicadas

Unicidad

La propiedad de ***unicidad*** es aquella que indica que en una columna no deben existir valores repetidos.

Las columnas de identidad tienen esta propiedad: por ejemplo, en un registro de empleados, no puede haber dos empleados con el mismo número de empleado, y, por tanto, la columna `numero_empledo` debe tener unicidad.

Eliminación de duplicados con drop_duplicates()

Hay algunos conjuntos de datos que, por su naturaleza, no permiten que dos filas o registros sean completamente iguales.

En aquellos casos donde hay una llave primaria, o que alguna otra columna tenga la propiedad de unicidad, no puede haber filas duplicadas (enteramente iguales).

Si no es normal que haya dos filas enteramente iguales, es necesario quitar duplicados usando la función **`drop_duplicates()`**.

Vamos a suponer que tenemos un DataFrame llamado **`df`**, con las columnas `col1`, `col2`, `col3`, `col4`, `col5`.

Si queremos que se eliminen duplicados en el caso de que todas las columnas sean iguales, y que se guarde el resultado en un DataFrame llamado **`sin_duplicados`**, se utiliza la siguiente instrucción:

```
sin_duplicados=df.drop_duplicates()
```

En este caso, como en muchas otras funciones de pandas, la instrucción retorna un DataFrame con el resultado de haber aplicado la instrucción.

Uso de inplace

En ocasiones, lo que queremos es que el resultado se almacene en el mismo DataFrame tomado como base del proceso, En ese caso, es necesario agregar el parámetro **`inplace=True`**.

```
df.drop_duplicates(inplace=True)
```

Entonces, tenemos dos opciones para actualizar un DataFrame.

```
df=df.drop_duplicates()
df.drop_duplicates(inplace=True)
```

¿Qué es lo recomendable, usar o no **`inplace`**? Tanto el uso del parámetro **`inplace=True`** como la asignación del resultado a la misma variable que almacena el DataFrame original hacen lo mismo, por lo que podemos decir que, al producir el mismo resultado, son igualmente *efectivos*.

El problema es *cómo* lo hacen, y es ahí donde las estrategias no son igualmente *eficientes*.

La diferencia principal entre los dos métodos es el comportamiento de cómo se modifican los datos. Con el parámetro **`inplace=True`**, los datos se modifican directamente en el DataFrame original sin crear una copia de este. Por lo tanto, el método **`inplace=True`** es más eficiente en términos de memoria, ya que no se crea una copia adicional del DataFrame.

Si se asigna el resultado del método a una nueva variable que tiene el mismo nombre que el DataFrame original, se crea una copia del DataFrame original que contiene los datos modificados, en memoria. Este método puede ser menos eficiente en términos de memoria, especialmente si se trabaja con grandes conjuntos de datos.

En resumen, si se tiene la opción, usa **`inplace=True`**, en lugar de la asignación sobre la misma variable (que, en realidad, internamente, es otra variable).

También se puede especificar a **`drop()`** qué columnas deben ser consideradas para concluir que las filas son duplicadas, usando el parámetro **`subset`**.

Si queremos que se eliminen duplicados del DataFrame **`df`**, considerando como duplicadas las filas que compartan solamente el mismo valor en **`col1`**, **`col2`**, **`col5`**, se utiliza la siguiente instrucción:

```
df.drop_duplicates(subset=['col1',
          'col2','col5'], inplace=True)
```

Eliminación de filas vacías

En ocasiones, al momento de realizar integraciones de datos, algunos sistemas manejadores de bases de datos envían filas vacías, adicionales a los datos que requerimos.

En otras ocasiones, hay filas que tienen una cantidad importante de datos vacíos, que las hacen inservibles para los propósitos de los trabajos de analítica.

El método **`dropna()`** de pandas se utiliza para eliminar las filas de un DataFrame que contienen valores faltantes (**`NaN`**).

El método **`dropna()`** admite los siguientes parámetros importantes:

1. **`axis`**: Especifica el eje en el que se desea eliminar los valores faltantes. Si **`axis=0`**, significa que las filas con valores faltantes se eliminarán; si **`axis=1`**, significa se eliminarán las columnas con valores faltantes.
2. **`how`**: Especifica cómo se determina si una fila o columna contiene valores faltantes. Las opciones son:
    a. **`any`**: Elimina las filas o columnas que contienen al menos un valor faltante.
    b. **`all`**: Elimina las filas o columnas que contienen todos los valores faltantes.
    c. **`thresh`**: Especifica el número mínimo de valores no faltantes que se deben tener en una fila o columna antes de que no se elimine. Por defecto es **`thresh=None`**, lo que significa que

todas las filas o columnas con valores faltantes se terminan eliminando.

Vamos a suponer que tenemos un DataFrame llamado **`df`**, con las columnas **`col1`**, **`col2`**, **`col3`**, **`col4`**, **`col5`**.

Si no deseas tener aquellas filas con todos los campos vacíos, sería:

```
df.dropna(how='all', inplace=True)
```

Si no deseas tener aquellas filas con algún campo vacío, sería:

```
df.dropna(how='any', inplace=True)
```

Si no deseas tener aquellas filas donde los campos **`col1`**, **`col2`** y **`col5`** estén vacíos, se usa el parámetro **`subset`**, sería:

```
df.dropna(subset=['col1',
          'col2','col5'], inplace=True)
```

Si no deseas tener aquellas filas con dos o más campos vacíos, sería:

```
df.dropna(thresh=df.shape[1]-3, inplace=True)
```

Recuerda que a **`thresh`** se le especifica el mínimo número de campos con datos, por lo cual, si el DataFrame tiene 5 campos, y el número máximo de datos vacíos es 3, entonces, el mínimo número de campos con datos es 5-3, es decir, 2. Por eso recuperamos el número de columnas usando **`shape[1]`**, y le restamos los datos que pueden estar vacíos.

Creación de un identificador ficticio

La ***creación de un identificador ficticio*** es la parte del proceso donde podemos crearle una llave primaria o identificador a una tabla, en caso de que no tenga una.

En analítica es muy importante que las tablas contengan una llave primaria o columna de identidad, que cumpla con los siguientes criterios: que nunca esté vacío o nulo; que no repita valores (propiedad de *unicidad*).

Algunas tablas tienen un campo de identidad natural, por ejemplo, en un registro de empleados no puede haber un empleado sin número de empleado, o dos empleados con el mismo número de empleado; en ese caso la columna `numero_empledo` actúa como columna de identidad o llave primaria (***primary key***).

El problema es que existen tablas que no cuentan con una o varias columnas que puedan actuar como identidad o llave primaria. Piensa en los datos de una encuesta donde los encuestados responden sí o no a tres preguntas; sí hay miles de registros, es casi seguro que habrá registros con los mismos valores y no se podrá tomar ninguna combinación de columnas como identidad.

En ese caso, debemos crear un ***identificador ficticio***, lo más común, es crear una serie de números enteros, que puede iniciar de 0, de 1, o del número que queramos.

Una forma rápida es utilizar la función **`range()`**, que genera una serie numérica que perfectamente puede actuar como identificador.

Vamos a suponer que tenemos un DataFrame llamado **`df`**, con las columnas `col1`, `col2`, `col3`, `col4`, `col5`, que tiene 500 filas.

Supongamos que queremos generar una columna llamada `id`, y queremos que inicie en 100.

```
inicio=100
fin=len(df)+inicial
df['id']=pd.Series(range(inicio,fin))
```

Verificación de unicidad en columnas

La ***verificación de unicidad de columna*** es la parte del proceso donde se identifica si un campo tiene valores duplicados o no.

Para verificar si un campo tienen unicidad, podemos apoyarnos en el método **`duplicated()`** y **`any()`**. La función **`duplicated()`** genera una serie con valores booleanos, indicando si el valor está duplicado o no. La función **`any()`** permite conocer si al menos uno de los elementos de la serie es **`True`**.

Vamos a suponer que tenemos un DataFrame llamado **`df`**, con las columnas **`col1`**, **`col2`**, **`col3`**, **`col4`**, **`col5`**.

Supongamos también que la columna **`col1`** es de identificación, por lo cual no tiene valores repetidos; supongamos que la columna **`col2`** es categórica, por lo cual lo normal es que tenga valores repetidos; supongamos que la columna **`col3`** es identificador no requerido, por lo que hay varias filas vacías, pero las filas que tienen valor no deben repetir valores.

Para validar si **`col1`** tiene repetidos, sería como sigue:

```
hay_duplicados = df['col1'].duplicated().any()
```

El valor será **`True`** o **`False`**, dependiendo si hay duplicados o no.

1. Si se aplica usando **`col1`**, será **`False`**, porque no tiene duplicados.
2. Si se aplica usando **`col2`**, será **`True`**, porque no tiene duplicados.
3. Si se aplica usando **`col3`**, será **`True`**, porque tiene duplicados, dado que hay vacíos.
4. Realmente debe ser **`False`**, pues los valores son únicos, si no hubiera vacíos.
5. Se deben quitar primero los vacíos, y luego evaluar la unicidad.

```
sin_duplicados=df['col3'][df['col3'].notnull()]
hay_duplicados = df['col3'].duplicated().any()
```

Escribir un CSV a partir de un DataFrame

Cuando se concluyen los trabajos de higiene e ingeniería de datos, es muy común que queramos almacenar el resultado de los cambios y transformaciones en un archivo de datos nuevo, a partir del cual se puedan iniciar trabajos de analítica.

Para guardar un DataFrame en un archivo CSV, se utiliza la función **`to_csv()`**.

Algunos parámetros interesantes de este método son:

1. **`path_or_buf`**: Especifica la ruta del archivo donde se va a guardar el DataFrame.

2. Puede ser un archivo o un objeto de archivo (como un objeto **`StringIO`**).

3. Si se omite este parámetro, el método devuelve el contenido del archivo como una cadena.

4. **`sep`**: Especifica el delimitador a utilizar en el archivo CSV. Por defecto es una coma.

5. **`header`**: Especifica si se debe incluir la fila de encabezado en el archivo CSV. Por defecto es **`True`**.

6. **`index`**: Especifica si se debe incluir la columna de índice en el archivo CSV. Por defecto es **`True`**.

7. **`mode`**: Especifica el modo de apertura del archivo CSV. Los valores posibles son: **`'w'`** (escritura), **`'a'`** (agregar), **`'x'`** (crear y escribir) y **`'wb'`** (escritura en modo binario). Por omisión es **`'w'`**, lo que implica que remplazará el archivo, si es que ya existe.

8. **`encoding`**: Especifica la codificación a utilizar en el archivo CSV. Por defecto es **`'utf-8'`**.

9. **`line_terminator`**: Especifica el separador de línea a utilizar en el archivo CSV. Por defecto es '**`\n`**'.

Supongamos que tenemos un DataFrame llamado **df**, y queremos guardar su estructura y sus datos en un archivo llamado **archivo.csv**, será así.

```
df.to_csv('archivo.csv')
```

Con esta sintaxis, se almacena también como columna el índice del DataFrame.

Si no deseamos que se incluya el índice en el archivo a crear, es necesario usar el parámetro **index**.

```
df.to_csv('archivo.csv', index=False)
```

Por la estructura misma de los archivos CSV, solo se pueden guardar datos con una misma estructura. Esto quiere decir que, si quiero integrar datos de más de una fuente, deben tener la misma estructura, porque al final, terminará siento un único conjunto bidimensional de datos.

Por ejemplo, si tenemos dos archivos CSV, uno llamado **primer_semestre.csv** y **segundo_semestre.csv**, y que deben integrarse en un solo archivo llamado **anual.csv**.

Los requisitos serían:

1. Los archivos **primer_semestre.csv** y **segundo_semestre.csv** deben tener la misma estructura de columnas.
2. El primer archivo agregado se pasa con encabezado.
3. El segundo archivo agregado se pasa sin encabezados, y en modo agregar.

El código quedaría así:

```
archivo1=pd.read_csv('primer_semestre.csv')
archivo2=pd.read_csv('segundo_semestre.csv')

archivo1.to_csv('datos.csv', index=False)
archivo2.to_csv('datos.csv', index=False,
      mode='a', header=False)
```

Escribir un Archivo de Excel a partir de un DataFrame

Un formato muy utilizado para el transporte y procesamiento de datos son los Libros de Excel.

Para guardar un DataFrame en un archivo XLSX, se utiliza la función **`to_excel()`**.

Algunos parámetros interesantes de este método son:

1. **`path_or_buf`**: Especifica la ruta del archivo donde se va a guardar el DataFrame.

2. Puede ser un archivo o un objeto de archivo (como un objeto **`StringIO`**).

3. Si se omite este parámetro, el método devuelve el contenido del archivo como una cadena.

4. **`sep`**: Especifica el delimitador a utilizar en el archivo CSV. Por defecto es una coma.

5. **`header`**: Especifica si se debe incluir la fila de encabezado en el archivo CSV. Por defecto es **`True`**.

6. **`index`**: Especifica si se debe incluir la columna de índice en el archivo CSV. Por defecto es **`True`**.

7. **`sheet_name`**: Es el nombre de la hoja de Excel en la que se guardarán los datos. Si no se proporciona este parámetro, se usará el nombre predeterminado "**`Sheet1`**".

8. **`startrow`**: Especifica la fila de la hoja de Excel en la que se iniciará la escritura. El valor predeterminado es 0.

9. **`startcol`**: Especifica la columna de la hoja de Excel en la que se iniciará la escritura. El valor predeterminado es 0.

10. **`float_format`**: Especifica el formato que se usará para escribir números de punto flotante en el archivo de Excel. El valor predeterminado es **`None`**, lo que significa que se utilizará el formato predeterminado de Excel.

Supongamos que tenemos un DataFrame llamado **`df`**, y queremos guardar su estructura y sus datos en un archivo llamado **`datos.xlsx`**, será así.

```
df.to_excel('archivo.xlsx', sheet_name='Hoja 1'))
```

Con esta sintaxis, se almacena también como columna el índice del Data-Frame.

Si no deseamos que se incluya el índice en el archivo a generar, es necesario usar el parámetro **`index`**.

```
df.to_excel('archivo.xlsx',
    index=False, sheet_name=' Hoja 1')
```

Por la estructura misma de los archivos XLSX, se pueden guardar múltiples DataFrames y estructuras de datos.

Por ejemplo, si tenemos dos DataFrames, uno llamado **empleado** y otro llamado **almacenes**, podemos almacenarlos en un mismo libro de Excel llamado **datos.xlsx**, de la siguiente forma:

```
with pd.ExcelWriter('datos.xlsx') as flujo_datos:
    empleado.to_excel(flujo_datos, sheet_name='Hoja1')
    datos2.to_excel(flujo_datos, sheet_name='Hoja2')
```

LAB 07.01: Tareas generales con DataFrames

En este Lab se revisan estrategias para realizar tareas generales sobre filas y columnas en un DataFrame.

Las tareas por realizar son:

1. Eliminar columnas no requeridas usando **`drop()`**.
2. Eliminar filas duplicadas usando **`drop_duplicates()`**.
3. Eliminar filas con datos vacíos usando **`dropna()`**.
   a. Eliminar las filas con todas las columnas vacías.
   b. Eliminar las filas con 3 o más columnas vacías.
4. Crear un indicador ficticio.
5. Verificar unicidad de columnas usando **`duplicated()`**.
   a. Verificar la unicidad de un campo de identidad.
   b. Verificar la unicidad de un campo categórico.
   c. Verificar la unicidad de un campo con vacíos.

```
# Datos base
import pandas as pd
pasajeros_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
pasajeros_titanic.csv'
titanic = pd.read_csv(pasajeros_titanic_csv)
```

Eliminar columnas no requeridas usando `drop()`

```
# Se eliminan los campos que no son requeridos del
# DataFrame titanic
titanic.drop(columns=['boleto','tarifa',
            'cabina','embarque','residencias'],
            inplace=True)

titanic.dtypes
```

```
clase_viaje      float64
sobrevivencia    float64
nombre            object
sexo              object
edad             float64
parientes          int64
familiares         int64
bote              object
cuerpo           float64
dtype: object
```

Eliminar filas duplicadas usando drop_duplicates()

```
# Eliminar duplicados en titanic, donde todas las columnas
# sean exactamente iguales en todos sus valores, guardando
# el resultado en el mismo DataFrame
titanic.drop_duplicates(inplace=True)
```

Eliminar filas con datos vacíos usando dropna()

```
# Se revisa cuántos datos hay.
print(f'Se tienen {titanic.shape[0]:,} filas.')

# Elimina todos los registros en donde todas las columnas
# estén vacías.
titanic.dropna(how='all', inplace=True)

# Se revisa cuántos datos hay.
print(f'Se tienen {titanic.shape[0]:,} filas.')

# Se concluye que no había filas completamente vacías.
```

```
Se tienen 1,309 filas.
Se tienen 1,309 filas.
```

```
# Elimina todos los registros con más de tres columnas vacías.
titanic.dropna(thresh=titanic.shape[1]-3, inplace=True)

# Se revisa si se eliminaron filas.
print(f'Se tienen {titanic.shape[0]:,} filas.')

# Se concluye que se eliminaron 2 filas.
```

```
Se tienen 1,307 filas.
```

Crear un indicador ficticio

```
# Generar una llave primaria (columna de identidad) para
# los datos del titanic.
# Se desea que el primer identificador sea 10000

inicial=10000
titanic['identificador']=pd.Series(range(inicial,
       len(titanic)+inicial))

# Ver los primeros registros, para verificar la creación
# del identificador.

titanic.head()

# Se debió haber agregado una columna nueva, llamada
# identificador, que es una serie numérica que inicia en 10000.
# Toma en cuenta que se ha generado como flotante, cosa que
# se tendrá que corregir más adelante.
```

```
   edad  parientes  familiares bote  cuerpo  identificador  ▶
0  39.0          0           0  NaN     NaN        10000.0  ▶
1   NaN          0           0  NaN     NaN        10001.0  ▶
2   NaN          0           0  NaN     NaN        10002.0  ▶
3  40.0          0           0  NaN   110.0        10003.0  ▶
4  49.0          0           0    C     NaN        10004.0  ▶
```

Verificar unicidad de columnas usando duplicated()

```
# Se valida si en identificador hay algún valor duplicado.
# El valor se asigna a la variable hay duplicados.
hay_duplicados = titanic['identificador'].duplicated().any()

# Se imprime el resultado. Debe ser False.
print(hay_duplicados)
```

```
False
```

```
# Se valida si en clase_viaje hay algún valor duplicado.
# El valor se asigna a la variable hay_duplicados.
hay_duplicados = titanic['clase_viaje'].duplicated().any()

# Se imprime el resultado. Debe ser True.
print(hay_duplicados)
```

```
True
```

Si queremos validar que una columna no tenga valores repetidos, sin tomar en cuenta los datos vacíos o nulos, primero se deben eliminar los datos vacíos o nulos, y luego verificar la unicidad.

En nuestro caso, el campo `cuerpo`, que indica el número de cuerpo asignado a las personas fallecidas, no debe tener repetidos, pero hay muchas filas en donde ese dato no se tiene.

```
# Se genera una Serie de pandas que contenga únicamente
# los valores de la columna cuerpo, cuando cuerpo no sea
# nulo.
valores_cuepo = titanic['cuerpo'][titanic['cuerpo'].notnull()]

# Se valida si en valores_cuerpo hay algún valor duplicado.
# El valor se asigna a la variable hay_duplicados.
hay_duplicados = valores_cuepo.duplicated().any()

# Se imprime el resultado. Debe ser False.
print(hay_duplicados)
```

```
False
```

FIN DEL LAB

Tareas generales con columnas

Reordenar las columnas de un DataFrame

Para el desempeño de pandas, el orden de las columnas no tiene ningún impacto, así que no tendría ninguna utilidad modificar el orden de las mismas.

Por otro lado, a los seres humanos el orden en que aparecen las columnas puede facilitar o complicar el entendimiento del conjunto de datos.

Las organizaciones pueden determinar un orden preferente, que por sistema nos ayude a realizar verificaciones respecto a la estructura. Supón que hay un conjunto de datos con un campo de identidad, siete categóricos, dos temporales, y cinco campos de valor. Si los datos estuvieran ordenados por el uso que tienen, seguramente sería más sencillo verificar que se tienen todos los datos, a que si están desordenados.

Para reordenar las columnas, se sustituye el DataFrame con una consulta ordenada del mismo DataFrame, con el apoyo de una lista de Python.

Supongamos que tenemos un DataFrame llamado **`df`**, que tiene los campos en este orden: **`CampoB`**, **`CampoA`**, **`CampoC`**. Para reordenar los campos, sería así:

```
campos_ordenados=['CampoA', 'CampoB', 'CampoC']

df=df[campos_ordenados]
```

Cambiar el nombre de las columnas en un DataFrame

Al reunir datos de diferentes fuentes podemos terminar con un conjunto de datos que cuente con columnas que respeten diferentes protocolos de nomenclatura. Mientras que un sistema puede llamarle al número de empleado **`employee_id`**, otro puede llamarle **`numero_empleado`**, o **`idemp`**; sobra

decir que ninguno es el correcto, pues lo correcto será que todas las columnas que refieren al mismo dato se llamen de la misma manera.

En esos casos, surge la necesidad de homologar los nombres de columna, ya sea para eliminar ambigüedades e inconsistencias, o simplemente por cuestión de facilidad y claridad de uso.

Como parte de los trabajos de ingeniería datos, podemos modificar el nombre de las columnas de un DataFrame. Para hacerlo, se puede utilizar el método **`rename()`**, en conjunto con un diccionario proporcionado al parámetro **`columns`**.

Supongamos que tenemos un DataFrame llamado **`df`**, que tiene los campos en este orden: **`CampoA`**, **`CampoB`**, **`CampoC`**, y que deben actualizar su nombre a **`correo`**, **`nombre`**, **`teléfono`**, respectivamente. Para hacer el cambio de nombres, sería así:

```
nuevos_nombres={
  'col1':'correo',
  'col2':'nombre',
  'col3':'teléfono'
}

df.rename(columns=nuevos_nombres, inplace=True)
```

LAB 07.02: Trabajo con columnas

En este Lab se revisan estrategias para filtrar filas usando **`loc[]`** y **`where()`**.

Las tareas por realizar son:

1. Eliminar columnas no requeridas usando **`drop()`**.
2. Eliminar filas duplicadas usando **`drop_duplicates()`**.
3. Eliminar filas con datos vacíos usando **`dropna()`**.
    a. Eliminar las filas con todas las columnas vacías.
    b. Eliminar las filas con 3 o más columnas vacías.
4. Crear un indicador ficticio.
5. Verificar unicidad de columnas usando **`duplicated()`**.
6. Verificar la unicidad de un campo de identidad.
    a. Verificar la unicidad de un campo categórico.
    b. Verificar la unicidad de un campo con vacíos.

```
# Datos base
import pandas as pd
pasajeros_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
pasajeros_titanic.csv'
titanic = pd.read_csv(pasajeros_titanic_csv)
titanic.drop(columns=['boleto','tarifa','cabina',
         'embarque','residencias'],inplace=True)
titanic['identificador']=pd.Series(range(10000,
         len(titanic)+10000))
```

Reordenar las columnas de un DataFrame.

En nuestro caso, tenemos actualmente las siguientes columnas:

Variable	DTXC
clase_viaje	QL-OR-STR-CAT-CD-AA-NDR
sobrevivencia	QL-NM-STR-CAT-CD-AA-NDR
nombre	QL-NM-STR-DS-AA-NDR
sexo	QL-NM-STR-CAT-DES-AA-NDR
edad	QT-CON-NUM-FL-VAL-DET-AA-NDR
parientes	QT-DIS-NUM-INT-VAL-DET-AA-NDR
familiares	QT-DIS-NUM-INT-VAL-DET-AA-NDR
rango_edad	QL-OR-STR-CAT-DES-AA-NDR
bote	QL-NM-STR-NRID-AA-NDR
cuerpo	QL-NM-STR-NRID-AA-NDR
identificador	QL-NM-STR-ID-AA-NDR

Recuerda que los códigos DTXC contienen los datos como deben ser, y no como son.

Se desean colocar los campos siguiendo este orden:

1. Primero los datos de identidad (`identificador`)
2. Luego los datos correspondientes a las variables dependientes (`sobrevivencia`).
3. Luego los datos correspondientes a las variables independientes (todos los demás campos).
4. El orden para las variables dependientes e independientes, dentro de su categoría, seguirá este orden.
5. Datos descriptivos (`nombre`)
6. Luego los temporales (**no hay**)
7. Luego los categóricos descriptivos (`sexo`)
8. Luego los categóricos numéricos (**no hay**)
9. Luego los categóricos codificados (`clase_viaje`)
10. Luego los de identidad no requerido (`bote`, `cuerpo`)
11. Luego los datos de valor (`edad`, `parientes`, `familiares`).

```
# Genera una lista que contenga los nombres de columnas
# en el orden deseado.
campos_ordenados=['identificador', 'sobrevivencia',
        'nombre', 'sexo', 'clase_viaje', 'bote', 'cuerpo',
                  'edad', 'parientes', 'familiares']

# Recupera los campos en el orden especificado en la lista, y
# guárdalos en el mismo DataFrame origen. Dicho de otra forma,
# el DataFrame es la representación ordenada de sí mismo.
titanic=titanic[campos_ordenados]

# Enumera los campos del DataFrame para comprobar el
# nuevo orden, usando dtypes.
titanic.dtypes
```

```
identificador      int64
sobrevivencia    float64
nombre            object
sexo              object
clase_viaje      float64
bote              object
cuerpo           float64
edad             float64
parientes          int64
familiares         int64
dtype: object
```

Cambiar nombre de columnas en un DataFrame.

Se desean hacer los siguientes cambios de nombre de columna:

1. Los identificadores, requeridos o no requeridos, deben iniciar con **`id_`**.
2. Los datos de valor que indiquen cantidad deben iniciar con **`cantidad_`**.
3. Los categóricos numéricos y codificados, deben iniciar con **`clave_`**, porque requieren interpretación.
4. Se deben hacer precisiones de sustantivos. Por ejemplo, usar *pasajeros* resulta inexacto, pues la *tripulación* no era pasajera, entonces, lo adecuado sería hablar de *personas*.

Los nombres por modificar son estos:

Nombre actual	Nombre nuevo
identificador	**id_persona**
sobrevivencia	**clave_sobrevivencia**
bote	**id_bote**
cuerpo	**id_cuerpo**
parientes	**cantidad_parientes**
familiares	**cantidad_familiares**

```
# Genera un diccionario que contenga la equivalencia
# las llaves son los nombres de columna actuales, y los
# valores son los nombres que queremos asignar.
# Las columnas que no sufren cambio no se ponen.
nuevos_nombres={
    'identificador':'id_persona',
    'sobrevivencia':'clave_sobrevivencia',
    'bote':'id_bote',
    'cuerpo':'id_cuerpo',
    'parientes':'cantidad_parientes',
    'familiares':'cantidad_familiares'
}

# Renombra las columnas, usando rename(), aplica los
# cambios sobre el mismo DataFrame, usando inplace.
titanic.rename(columns=nuevos_nombres,inplace=True)

# Enumera los campos del DataFrame para comprobar los nuevos
# nombres, usando dtypes.
titanic.dtypes
```

```
id_persona              int64
clave_sobrevivencia    float64
nombre                  object
sexo                    object
clase_viaje            float64
id_bote                 object
id_cuerpo              float64
edad                   float64
cantidad_parientes       int64
cantidad_familiares      int64
dtype: object
```

Escribir un CSV a partir de un DataFrame.

Se toma como base el DataFrame **`titanic`**, con las transformaciones realizadas hasta el momento, y se guarda en un archivo en formato CSV llamado **`personas_titanic.csv`**, sin incluir el índice.

```
# Guarda el contenido del DataFrame titanic en un archivo
# CSV, usando to_csv(), incluyendo el parámetro index, para
# no incluir el índice.
titanic.to_csv('personas_titanic.csv', index=False)
```

Haz clic en el ícono de carpeta que aparece en la extrema izquierda de **Google Colab**, y comprueba que el archivo que has guardado está ahí.

Integrar dos DataFrames en un Libro en Excel con dos hojas

Supongamos que queremos generar dos Libros de Excel.

1. Un Libro de Excel llamado **`datos_actualizados.xlsx`**, que contiene una hoja llamado **`datos`**, que contiene todos los datos, sin incluir el índice.

2. Un Libro de Excel llamado **`vivos_muertos.xlsx`**, que:
    a. Contiene una hoja llamado **`vivos`**, que contiene el nombre, el sexo, clave sobrevivencia y la edad de las personas que hayan sobrevivido (**`clave_sobrevivencia=1`**).
    b. Contiene una hoja llamado **`muertos`**, que contiene el nombre, el sexo, clave sobrevivencia y la edad de las personas que hayan sobrevivido (**`clave_sobrevivencia=0`**).

```
# Se genera el primer Libro de Excel, llamado
# datos_actualizados.xlsx
# No se incluye el índice.
# El nombre de la hoja será datos.
titanic.to_excel('datos_actualizados.xlsx',
      index=False, sheet_name='datos' )

# Se recuperan las columnas nombre, sexo y edad, de
# las personas que vivieron.
vivieron=titanic[['nombre','sexo',
         'clave_sobrevivencia',
         'edad']][(titanic['clave_sobrevivencia']==1.0)]

vivieron
```

```
                                                nombre    sexo  ▶
4                                Ismay, Mr. Joseph Bruce  hombre  ▶
9     Cornell, Mrs. Robert Clifford (Malvina Helen L...   mujer  ▶
10                             Omont, Mr. Alfred Fernand  hombre  ▶
11                           Leader, Dr. Alice (Farnham)   mujer  ▶
12    Swift, Mrs. Frederick Joel (Margaret Welles Ba...   mujer  ▶
...                                                 ...     ...
1288                                   Chip, Mr. Chang  hombre  ▶
1289                                   Foo, Mr. Choong  hombre  ▶
1290                                     Hee, Mr. Ling  hombre  ▶
1291                                      Lam, Mr. Ali  hombre  ▶
1293                                    Lang, Mr. Fang  hombre  ▶
[500 rows x 4 columns]
```

```
# Se recuperan las columnas nombre, sexo y edad,
# de las personas que murieron.
murieron=titanic[['nombre','sexo',
          'clave_sobrevivencia',
          'edad']][(titanic['clave_sobrevivencia']==0.0)]

murieron
```

```
                                    nombre    sexo  clave_sobrevivencia  edad
0                      Andrews, Mr. Thomas Jr  hombre                  0.0  39.0
1     Chisholm, Mr. Roderick Robert Crispin  hombre                  0.0   NaN
2                          Fry, Mr. Richard  hombre                  0.0   NaN
3                       Harrison, Mr. William  hombre                  0.0  40.0
5              Parr, Mr. William Henry Marsh  hombre                  0.0   NaN
...                                      ...     ...                  ...   ...
1304                    Sage, Mr. John George  hombre                  0.0   NaN
1305          Sage, Mrs. John (Annie Bullen)   mujer                  0.0   NaN
1306                      Storey, Mr. Thomas  hombre                  0.0  60.5
1307                    Baumann, Mr. John D  hombre                  0.0   NaN
1309                Baxter, Mr. Quigg Edmond  hombre                  0.0  24.0

[807 rows x 4 columns]
```

```
# Se genera una Hoja de Excel por cada DataFrame, y se
# almacenan en un mismo libro (vivos_muertos.xlsx).
with pd.ExcelWriter('vivos_muertos.xlsx') as flujo:
    vivieron.to_excel(flujo, sheet_name='vivos', index=False)
    murieron.to_excel(flujo, sheet_name='muertos', index=False)
```

Haz clic en el ícono de carpeta que aparece en la extrema izquierda de **Google Colab**, y comprueba que los archivos que has guardado están ahí.

FIN DEL LAB

¿Qué sigue? Transformación de datos

La ***transformación de datos*** es la parte del proceso donde podemos crear nuevas columnas o modificar las existentes, realizando operaciones o cálculos con la finalidad de dejar los datos como se requieren para el análisis, corrigiéndolos, ampliándolos, adecuando su formato, entre otras cosas.

Generalmente, estas son las transformaciones básicas que se pueden realizar. Se muestran de forma categorizada, para facilitar su comprensión.

1. ***CONVERSIONES***

    a. ***Conversión a diferente tipo de dato.*** La columna es una conversión de un *datatype* a otro. Lo más típico es:

        i. Pasar de **`int`** a **`float`**.
        ii. Pasar de **`float`** a **`int`** (puede haber pérdida de datos).
        iii. Pasar de **`int`** o **`float`**, a **`object`** (cadena).
        iv. Pasar de **`float`**, a **`object`** (cadena) sin decimales.
        v. Pasar de **`object`** a **`datetime`** (fecha).
        vi. Pasar de **`datetime`** a **`object`** (completo o por sección).

    b. ***Conversión de unidades de medida.*** La columna es el resultado de aplicar un factor de conversión o fórmula determinada, que provoca una equivalencia de unidades de medida.

    c. ***Conversión de unidades monetarias.*** La columna es el resultado de convertir una cantidad en un tipo de moneda, a otra moneda, aplicando un tipo de cambio específico.

2. ***CÁLCULOS***

    a. ***Columnas derivadas usando fórmulas.*** La columna es el resultado de una fórmula que utiliza como operandos a valores existentes en las columnas del conjunto de datos.

    b. ***Columnas derivadas usando UDF.*** La columna es el resultado de una función declarada con **`def`** que utiliza valores existentes en las columnas del conjunto de datos.

    c. ***Columnas derivadas usando lambda.*** La columna es el resultado de una función anónima **`lambda`** que utiliza valores existentes en las columnas del conjunto de datos.

3. ***MANEJO DE CADENAS***

    a. ***Columnas con diferente casing.*** La columna es una equivalencia, en donde lo que varía es el uso de mayúsculas y minúsculas. Estas son algunas variantes. 1. Cambiar a mayúsculas. 1. Cambiar a minúsculas. 1. Cambiar a mayúsculas solo las primeras letras de las palabras.

    b. ***División de cadenas.*** La columna es la división de una columna de tipo cadena, en dos o más.

    c. ***Concatenación de cadenas.*** La columna es la concatenación de dos o más columnas.

    d. ***Eliminar espacios en los extremos.*** La columna es el resultado de eliminar espacios innecesarios que se encuentren en los extremos de la cadena.

    e. ***Extracción de sub-cadena.*** La columna es una porción de una cadena.

4. ***MANEJO DE CATEGÓRICOS***

   a. ***Equivalencias categóricas.*** La columna es un categórico descriptivo, que es equivalente a un categórico codificado, numérico o dicotómico.

   b. ***Equivalencia categórica de intervalo.*** La columna es un categórico descriptivo, que se genera a partir de un campo de valor, considerando rangos de intervalo.

CAPÍTULO 8:
Conversión de datos

Las principales tareas de conversión son las siguientes:

1. ***Conversión a diferente tipo de dato.*** La columna es una conversión de un *datatype* a otro. Lo más típico es:
    a. Pasar de **`int`** a **`float`**.
    b. Pasar de **`float`** a **`int`** (puede haber pérdida de datos).
    c. Pasar de **`int`** o **`float`**, a **`object`** (cadena).
    d. Pasar de **`float`**, a **`object`** (cadena) sin decimales.
    e. Pasar de **`object`** a **`datetime`** (fecha).
    f. Pasar de **`datetime`** a **`object`** (completo o por sección).

2. ***Conversión de unidades de medida.*** La columna es el resultado de aplicar un factor de conversión o fórmula determinada, que provoca una equivalencia de unidades de medida.

3. ***Conversión de unidades monetarias.*** La columna es el resultado de convertir una cantidad en un tipo de moneda, a otra moneda, aplicando un tipo de cambio específico.

Conversión a diferente tipo de dato

Cuando hacemos la carga desde una fuente de datos hacia una Serie o DataFrame de pandas, la librería intenta inferir, a partir de los datos existentes, el tipo de dato (*datatype*) que tiene cada columna. Esto en ocasiones produce resultados inexactos, que debemos corregir.

Por ejemplo, si tenemos un número de empleado, puede ser que interprete que se trata de un dato tipo **`float`**, siendo que es un identificador que no admite operaciones aritméticas, y mucho menos, requiere decimales. En ese caso, sería necesario convertir el dato a **`int`**, y posteriormente convertirlo a cadena.

Conversiones usando astype() y to_datetime()

Para el cambio de *datatype*, las opción más común es el método **`astype()`**, que representa un dato en el tipo de dato especificado.

Supongamos que tenemos un DataFrame llamado **`df`**.

Pasar de int a float.

```
df['columna_flotantes'] = df['columna_enteros'].astype(float)
```

En este caso, se genera una nueva columna a partir de la conversión de otra.

Si lo que deseas es transformar una columna cambiando su tipo, el resultado de la conversión se asigna a la misma columna.

```
df['columna'] = df['columna'].astype(float)
```

Pasar de float a int (puede haber pérdida de datos).

```
df['columna_int'] = df['columna_float'].astype(int)
```

Toma en cuenta que en este caso la precisión decimal se pierde, y el número se redondea al entero más cercano.

Toma en cuenta también que, en conversiones a tipos numéricos, pueden encontrarse problemas con la conversión de valores vacíos o nulos. En ese caso, es necesario intentar la conversión solo con los datos que no son nulos, colocando el criterio de filtro antes de la función **`astype()`**.

```
df['columna_int'] = df['columna_float'][df['columna_float'].
      notnull()].astype(int)
```

Pasar de int o float, a object (cadena).

```
df['columna_object1'] = df['columna_int'].astype(str)
df['columna_object2'] = df['columna_float'].astype(str)
```

Pasar de float a object (cadena) sin decimales.

Para realizar esto, es necesario:

1. Convertir el dato **`float`** a **`object`**.
2. Por ejemplo, un **`10000.00`**, pasaría a ser **`'10000.00'`**.
3. Luego, se aplica la función string **`split()`**, usando el punto como separador.
4. Se genera una Serie de pandas, que usa como separador el punto.
5. La Serie queda como **`['10000','00']`**.
6. Luego, se aplica la función string **`get()`** para extraer el primer elemento de la Serie.

```
df['columna_object1'] = df['columna_float'].astype(str)
df['columna_object1'] = df['columna_object1'].
      str.split('.').str.get(0)
```

Pasar de object a datetime (fecha).

```
df['columna_fecha'] = pd.to_datetime(df['columna_object'])
```

Un valor **`object`** que podría convertirse fácilmente, sería `2023-10-15`, por ejemplo. La función **to_datetime()** intentará inferir el formato de fecha automáticamente. Si el formato de la fecha no se puede inferir automáticamente, es necesario proporcionar un formato de fecha explícito utilizando el parámetro **format**.

Estos serían unos ejemplos del uso de **`format`**:

```
# Ejemplo 1: formato YY/MM/DD
df['columna_fecha'] = pd.to_datetime(df['columna_object'],
      format='%y/%m/%d')

# Ejemplo 2: formato DD-MMM-YY
df['columna_fecha'] = pd.to_datetime(df['columna_object'],
      format='%d-%b-%y')

# Ejemplo 3: formato YYYY-MM-DD HH:MM:SS
df['columna_fecha'] = pd.to_datetime(df['columna_object'],
      format='%Y-%m-%d %H:%M:%S')
```

Pasar de datetime a object.

El procedimiento utiliza los especificadores de formato que se utilizan con **`format`**, en conjunto con la función **`strftime()`**.

```
# Ejemplo 1: formato YY/MM/DD
df['fecha_texto'] = df['columna_fecha'].
      dt.strftime('%y/%m/%d')

# Ejemplo 2: formato DD-MMM-YY
df['fecha_texto'] = df['columna_fecha'].
      dt.strftime('%d-%b-%y')
```

```
# Ejemplo 3: formato YYYY-MM-DD HH:MM:SS
df['fecha_texto'] = df['columna_fecha'].
      dt.strftime('%Y-%m-%d %H:%M:%S')

# Ejemplo 4: formato YYYY
df['año'] = df['columna_fecha'].
      dt.strftime('%Y')

# Ejemplo 5: formato YYYY-MM
df['año_mes'] = df['columna_fecha'].
      dt.strftime('%Y-%m')
```

Algunos de los códigos que se pueden usar al momento de expresar un formato, son esto:

- **%Y**: Año con siglo como número decimal completo.
- **%m**: Mes como número decimal con ceros a la izquierda.
- **%d**: Día del mes como número decimal con ceros a la izquierda.
- **%H**: Hora (formato de 24 horas) como número decimal con ceros a la izquierda.
- **%M**: Minuto como número decimal con ceros a la izquierda.
- **%S**: Segundo como número decimal con ceros a la izquierda.
- **%a**: Nombre abreviado del día de la semana.
- **%A**: Nombre completo del día de la semana.
- **%b**: Nombre abreviado del mes.
- **%B**: Nombre completo del mes.
- **%c**: Fecha y hora como cadena de texto.
- **%p**: Designación de AM/PM para la hora (solo aplicable a formato de 12 horas).
- **%x**: Fecha como cadena de texto.
- **%X**: Hora como cadena de texto.
- **%j**: Día del año como número decimal.
- **%U**: Número de semana del año (domingo como primer día de la semana).
- **%w**: Número decimal del día de la semana (0 es domingo).
- **%z**: Desplazamiento de la zona horaria en formato +HHMM o -HHMM.
- **%Z**: Nombre de la zona horaria.

Conversiones al momento de cargar datos

Resulta un poco incómodo que, al momento de cargar los datos a un DataFrame, pandas equivoque los tipos de datos y debamos escribir código para establecer el tipo de datos adecuado.

Piensa en un código numérico, por ejemplo, un número de empleado, que se almacena en una columna llamada **`núm_empleado`**, de un DataFrame llamado **`df`**, que proviene de un archivo CSV llamado **`datos.csv`**.

Es claro que se trata de un dato de identificación, de tipo cualitativo nominal, puesto que no es un número con el cual podamos hacer operaciones aritméticas de ningún tipo.

Sin embargo, cuando leamos ese dato desde un archivo CSV usando **`read_csv()`**, nos encontraremos con la novedad de que pandas lo interpreta como **`int`**, o incluso **`float`**, y tendremos que hacer las conversiones necesarias después de cargar los datos.

Una forma de evitar que pandas sugiera el tipo de dato deseado es indicándole el tipo de dato que queremos para las columnas que son ambiguas, con el apoyo de un diccionario. Se le debe proporcionar al parámetro **`dtype`**, un diccionario, que indique como llave el nombre de columna, y como valor el tipo deseado.

```
df=read_csv('datos.csv', dtype={
      ' num_empleado':str
      })
```

Los datos que soporta **`dtype`** son los siguientes:

1. **`str`** o **`object`**: para cadenas de caracteres.
2. **`int`** o **`int64`**: para enteros.
3. **`float`** o **`float64`**: para números de punto flotante.
4. **`bool`**: para valores booleanos True o False.
5. **`datetime64`**: para fechas y horas.
6. **`timedelta`**: para diferencia entre fechas y horas.

Conversiones de unidades de medida

Para las conversiones por unidad de medida, la solución es muy simple, ya que se limita a la aplicación de un factor de conversión, y una operación aritmética.

```
# Conversión de USD a Pesos Mexicanos
tipo_cambio_USD_MXN=18.75
df['monto_MXN']=df['monto_USD']*tipo_cambio_USD_MXN

# Conversión de KG a Libras
factor_KG_LIBRA=2.20462
df['Libras']=df['KG']*factor_KG_LIBRA

# Conversión de grados centígrados a Farenheit.
df['farenheit']=(df['centigrados']*(9/5))+32
```

Conversiones de unidades monetarias

Relativo a las conversiones monetarias, podemos enfrentar el problema de estar proporcionando dinámicamente los tipos de cambio para las diferentes monedas.

Una forma de hacerlo es instalando el paquete **`forex_python`**, y recuperar las tasas de cambio de manera automática.

Para instalar paquetes en **Google Colab**, podemos hacer lo siguiente:

```
%pip install forex_python
```

Una vez instalado el paquete, *a)* Se importa la librería, *b)* Se recupera en línea el tipo de cambio deseado, y *c)* Se realiza la conversión normalmente.

Por ejemplo, para convertir a pesos mexicanos una cantidad que está en euros, podría ser de la siguiente manera.

```
from forex_python.converter import CurrencyRates

# Se genera una instancia de CurrencyRates
c = CurrencyRates()

# Obtener la tasa de cambio de EUR a MXN usando Forex_python
tipo_cambio = c.get_rate('EUR', 'MXN')

# Conversión
df['cantidad_MXN'] = df['cantidad_EUR'] * tipo_cambio
```

Si tuviéramos un DataFrame en donde tuviéramos una columna llamada **`moneda_origen`**, que contiene el código de moneda de una cantidad; y tuviéramos otra columna llamada **`cantidad`**, que tuviera la cantidad de moneda que tenemos; si quisiéramos una columna llamada **`equivalente_euros`**, que contenga la transformación de la cantidad a euros, podría ser así.

```
# Creamos una instancia de CurrencyRates para realizar
# la conversión
c = CurrencyRates()

# Definimos la moneda a la que queremos convertir las demás,
# que en este caso es euros.
moneda_destino = 'EUR'

# Creamos una función que aplica la conversión a cada fila.
def convertir_moneda(fila):
    tasa = c.get_rate(fila['moneda_origen'], moneda_destino)
    return fila['cantidad'] * tasa

# Aplicamos la función a cada fila y almacenamos los resultados
# en una nueva columna
df['equivalente_euros'] = df.apply(convertir_moneda, axis=1)

# Imprimimos el DataFrame resultante
print(df)
```

LAB 08.01: Ejecutando conversiones de tipo y de moneda

En este Lab se realizan conversiones de tipos para que las columnas se ajusten a los tipos especificados por los códigos DTXC del caso. También se realiza un demo de conversiones de moneda, tomando tipos de cambio desde un servicio de la nube.

Las tareas por realizar son:

1. Realizar conversiones de tipo de dato.
2. Realizar conversiones de tipo de moneda.
1. instalar **`forex_python`**
2. Realizar conversiones de moneda con tipos de cambio en línea.

```
# Datos base
import pandas as pd
personas_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
personas_titanic.csv'
titanic = pd.read_csv(personas_titanic_csv)
```

Realizar conversiones de tipo de dato.

En el caso, tenemos las siguientes columnas con sus tipos:

Variable	DTXC	Tipo actual
id_persona	QL-NM-STR-ID-AA-NDR	float64
clave_sobrevivencia	QL-NM-STR-CAT-CD-AA-NDR	float64
nombre	QL-NM-STR-DS-AA-NDR	object
clave_sexo	QL-NM-STR-CAT-DES-AA-NDR	object
clase_viaje	QL-OR-STR-CAT-CD-AA-NDR	float64
id_bote	QL-NM-STR-NRID-AA-NDR	object
id_cuerpo	QL-NM-STR-NRID-AA-NDR	float64
edad	QT-CON-NUM-FL-VAL-DET-AA-NDR	float64
cantidad_parientes	QT-DIS-NUM-INT-VAL-DET-AA-NDR	int64
cantidad_familiares	QT-DIS-NUM-INT-VAL-DET-AA-NDR	int64

Para que los campos correspondan a los tipos requeridos por los códigos DTXC, se requieren hacer las siguientes modificaciones.

1. **id_persona** es un identificador, cualitativo nominal.
    a. Pasar de **float** a **object** sin decimales.
2. **clave_sobrevivencia** es un categórico codificado, cualitativo nominal.
    a. Pasar de **float** a **object** sin decimales.
3. **clase_viaje** es un categórico codificado, y es cualitativo ordinal.
    a. Pasar de **float** a **object** sin decimales.
4. **id_cuerpo** es un identificador no requerido, cualitativo nominal.
    a. Pasar de **float** a **object** sin decimales.

```
# Transformación de id_persona
titanic['id_persona']=titanic['id_persona'].astype('str')
titanic['id_persona']=titanic['id_persona'].
      str.split('.').str.get(0)

# Transformación de clave_sobrevivencia
titanic['clave_sobrevivencia']=titanic['clave_sobrevivencia'].
      astype('str')
titanic['clave_sobrevivencia']=titanic['clave_sobrevivencia'].
      str.split('.').str.get(0)

# Transformación de clase_viaje
titanic['clase_viaje']=titanic['clase_viaje'].astype('str')
titanic['clase_viaje']=titanic['clase_viaje'].str.split('.').
      str.get(0)

# Transformación de id_cuerpo
titanic['id_cuerpo']=titanic['id_cuerpo'].astype('str')
titanic['id_cuerpo']=titanic['id_cuerpo'].str.split('.').
      str.get(0)

# Ver los tipos transformados
titanic.dtypes
```

```
id_persona              object
clave_sobrevivencia     object
nombre                  object
sexo                    object
clase_viaje             object
id_bote                 object
id_cuerpo               object
edad                   float64
cantidad_parientes       int64
cantidad_familiares      int64
dtype: object
```

Realizar conversiones de moneda con tipos de cambio en línea

Se instala el paquete **Forex_python**.

```
%pip install forex_python
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheel
s/public/simple/
Requirement already satisfied: forex_python in /usr/local/lib/python3.9/dist-packa
ges (1.8)
Requirement already satisfied: simplejson in /usr/local/lib/python3.9/dist-package
s (from forex_python) (3.18.4)
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages
(from forex_python) (2.27.1)
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3
.9/dist-packages (from requests->forex_python) (2.0.12)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist
-packages (from requests->forex_python) (2022.12.7)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/d
ist-packages (from requests->forex_python) (1.26.15)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packa
ges (from requests->forex_python) (3.4)
```

Instalado el paquete, se puede usar en el programa.

```
# Se importa la librería para recuperar tipos de cambio.
from forex_python.converter import CurrencyRates

# Creamos el DataFrame de ejemplo.
df = pd.DataFrame({'moneda_origen': ['USD','EUR','GBP','JPY'],
                   'cantidad': [100, 200, 300, 400]})
# Ver los datos.
df
```

```
  moneda_origen  cantidad
0           USD       100
1           EUR       200
2           GBP       300
3           JPY       400
```

```
# Creamos una instancia de CurrencyRates para realizar
# la conversión.
c = CurrencyRates()

# Definimos la moneda a la que queremos convertir las demás,
# que en este caso es euros.
moneda_destino = 'EUR'

# Creamos una función que aplica la conversión a
# cada fila.
def convertir_moneda(fila):
    tasa = c.get_rate(fila['moneda_origen'], moneda_destino)
    return fila['cantidad'] * tasa

# Aplicamos la función a cada fila y almacenamos los resultados
# en una nueva columna
df['equivalente_euros'] = df.apply(convertir_moneda, axis=1)

# Imprimimos el DataFrame resultante
print(df)
```

```
  moneda_origen  cantidad  equivalente_euros
0           USD       100          92.721372
1           EUR       200         200.000000
2           GBP       300         341.199886
3           JPY       400           2.794662
```

FIN DEL LAB

CAPÍTULO 9:

Columnas derivadas o calculadas

Las principales tareas de cálculo son las siguientes:

1. ***Columnas derivadas usando fórmulas.*** La columna es el resultado de una fórmula que utiliza como operandos a valores existentes en las columnas del conjunto de datos.
2. ***Columnas derivadas usando UDF.*** La columna es el resultado de una función declarada con **`def`** que utiliza valores existentes en las columnas del conjunto de datos.
3. ***Columnas derivadas usando lambda.*** La columna es el resultado de una función anónima **`lambda`** que utiliza valores existentes en las columnas del conjunto de datos.

Columnas derivadas usando fórmulas

Las ***columnas derivadas*** son aquellas que se generan a partir de otras columnas, aplicando fórmulas o funciones.

Supón que tienes un DataFrame llamado **`df`** con dos columnas, **`A`** y **`B`**, y ambas contienen números.

```
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [10, 20, 30]
```

```
    })

df
```

```
   A   B
0  1  10
1  2  20
2  3  30
```

Puedes generar una nueva columna a partir de una fórmula simple.

Cuando hacemos referencia a una columna de un DataFrame, ya sea en modo regular o usando *dot notation*, y le asignamos valores, lo que hace pandas es crear una nueva columna si es que no existe en el DataFrame, y remplazarla, si es que existe.

Imagina que quieres agregar una nueva columna llamada **`con_formula`** que contenga la multiplicación de **`A`** y **`B`**, usando fórmula.

```
df['con_formula']=(df['A']*df['B'])

df
```

```
   A   B  con_formula
0  1  10           10
1  2  20           40
2  3  30           90
```

Como la columna **`con_formula`** no existía, la creo, asignándole el valor resultante del cálculo que se indicó.

Con esta estrategia, basta hacer referencia a las columnas, como expresiones en la fórmula, para crear una nueva columna.

Columnas derivadas usando UDF (user defined functions)

En ocasiones los cálculos no son tan simples, y pueden llegar a requerir más de una línea para resolver el cálculo. En esas situaciones, debemos tener la posibilidad de generar una función que podamos invocar para producir un valor.

Puedes generar una nueva columna a partir de una ***función definida por el usuario*** (***user defined function*** / ***udf***), que son las típicas funciones que utilizamos usando **`def`**.

Imagina que quieres agregar una nueva columna llamada **`con_udf`** que contenga la multiplicación de **`A`** y **`B`**, usando UDF.

```
# Se declara la función.
def multiplicar(fila):
  return fila['A'] * fila['B']

# Se ejecuta la función, aplicando el proceso codificado a
# cada una de las filas del DataFrame.
df['con_udf']=df.apply(multiplicar,axis=1)

# Ver resultado.
df
```

```
   A   B  con_formula  con_udf
0  1  10           10       10
1  2  20           40       40
2  3  30           90       90
```

Con esta estrategia:

1. Se define la función usando **`def`**.
2. Se pasa un argumento (**`fila`**), que representa una fila del DataFrame, con todas sus columnas.
3. Con las columnas se pueden hacer cálculos dentro de la función, refiriéndolos a través del nombre del argumento.
4. Se utiliza el método **`apply()`**:

5. El primer parámetro especifica el nombre de la función a ejecutar (**`multiplicar`**).
6. El segundo (**`axis=1`**) se usa para indicar que se aplicará la función a cada fila del DataFrame.

También se puede utilizar la función **`map()`**, cuando se desea aplicar un cálculo a nivel columna.

```
# Definimos una función que calcula el cuadrado de un número.
def cuadrado(fila):
    return fila**2

# Aplicamos la función cuadrado a cada elemento de
# la columna 'numeros'
df['con_map'] = df['B'].map(cuadrado)

# Ver resultado.
df
```

```
   A   B  con_formula  con_udf  con_map
0  1  10           10       10      100
1  2  20           40       40      400
2  3  30           90       90      900
```

Columnas derivadas usando lamda

Las funciones lambda nos permiten declarar una ***función anónima***, conocida también como ***lamda***, en donde no requerimos declarar una función externa que deba ser llamada al momento de crear la nueva columna.

Imagina que quieres agregar una nueva columna llamada **con_lambda** que contenga la multiplicación de **A** y **B**, usando UDF.

```
# Se define una función lamda.
df['con_lambda']=df.apply(lambda fila:
      fila['A']*fila['B'],
      axis=1)

# Ver el resultado.
df
```

```
   A   B  con_formula  con_udf  con_map  con_lambda
0  1  10           10       10      100          10
1  2  20           40       40      400          40
2  3  30           90       90      900          90
```

Con esta estrategia:

1. Se utiliza el método **apply()** sobre el DataFrame.
2. Se especifica que se trata de una función **lambda**.
3. Se declara un argumento (**fila**), que representa las columnas del DataFrame.
4. Se escribe dos puntos, y se codifica el cálculo, haciendo referencia a las columnas a través del argumento **fila**.
5. El segundo argumento (**axis=1**) se usa para indicar que se aplicará la función a cada fila del DataFrame.

Es importante mencionar que lambda puede usarse para funciones y cálculos sencillos, en cuyo caso es más eficiente que las otras dos alternativas.

LAB 09.01: Cálculos con columnas

En este Lab se desea generar una nueva columna llamada **`acompañantes`**, que sea la suma de **`cantidad_familiares`** y **`cantidad_parientes`**, usando todas las variantes.

Las tareas por realizar son:

1. Generar una nueva columna usando fórmulas.
2. Generar una nueva columna usando UDF's.
   1. Pasando todas las columnas.
   2. Pasando sólo las columnas requeridas.

```
# Datos base
import pandas as pd
personas_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
personas_titanic_v2.csv'
titanic = pd.read_csv(personas_titanic_csv)
```

Generar una nueva columna usando fórmulas

```
# La tarea, en su forma más sencilla (fórmula simple)
titanic['acompañantes']=(titanic['cantidad_familiares']+
                         titanic['cantidad_parientes'])

# Se comprueba la generación de la suma y la creación
# de la nueva columna. Se genera un DataFrame que contenga
# las filas donde acompañantes sea mayor a cero.
pasajeros_acompañados=titanic.loc[titanic['acompañantes']>0]

# Se muestran los resultados.
pasajeros_acompañados
```

```
      acompañantes  ▶
9                2  ▶
21               2  ▶
33               2  ▶
51               1  ▶
67               1  ▶
...            ...
1302            10  ▶
1303            10  ▶
1304            10  ▶
1305            10  ▶
1309             1  ▶

[519 rows x 11 columns]
```

Generar una nueva columna usando UDF's

1. **Pasando todas las columnas.**

```
# Se declara la función externa.
def calcula_acompañantes(fila):
  return fila['cantidad_familiares']+fila['cantidad_parientes']

# Se invoca la función.
titanic['acompañantes_1']=titanic.apply(calcula_acompañantes,
      axis=1)

# Filtrar para ver solo personas con acompañantes.
pasajeros_acompañados=titanic.loc[titanic['acompañantes']>0]

# Ver el resultado
pasajeros_acompañados
```

```
      acompañantes  acompañantes_1  ▶
9                2               2  ▶
21               2               2  ▶
33               2               2  ▶
51               1               1  ▶
67               1               1  ▶
…
[519 rows x 12 columns]
```

2. **Pasando sólo las columnas requeridas.**

```
# Se declara la función externa.
def calcula_acompañantes(fila):
  return fila['cantidad_familiares']+fila['cantidad_parientes']

# Se invoca la función, pasando sólo algunas columnas.
titanic['acompañantes_2']=titanic[['cantidad_familiares',
     'cantidad_parientes']].apply(calcula_acompañantes,axis=1)

# Filtrar para ver solo personas con acompañantes.
pasajeros_acompañados=titanic.loc[titanic['acompañantes']>0]

# Ver el resultado
pasajeros_acompañados
```

```
      acompañantes  acompañantes_1  acompañantes_2  ▶
9                2               2               2  ▶
21               2               2               2  ▶
33               2               2               2  ▶
51               1               1               1  ▶
67               1               1               1  ▶
...            ...             ...             ...
1302            10              10              10  ▶
1303            10              10              10  ▶
1304            10              10              10  ▶
1305            10              10              10  ▶
1309             1               1               1  ▶
[519 rows x 13 columns]
```

Generar una nueva columna usando funciones anónimas (lamda)

```
# Se aplica la función lambda usando apply()
titanic['acompañantes_3']=titanic.apply(lambda x:
    x['cantidad_familiares'] +
    x['cantidad_parientes'], axis=1)

# Filtrar para ver solo personas con acompañantes.
pasajeros_acompañados=titanic.loc[titanic['acompañantes']>0]

# Ver el resultado
pasajeros_acompañados
```

```
      acompañantes  acompañantes_1  acompañantes_2  acompañantes_3  ▶
9                2               2               2               2  ▶
21               2               2               2               2  ▶
33               2               2               2               2  ▶
51               1               1               1               1  ▶
67               1               1               1               1  ▶
...            ...             ...             ...             ...
1302            10              10              10              10  ▶
1303            10              10              10              10  ▶
1304            10              10              10              10  ▶
1305            10              10              10              10  ▶
1309             1               1               1               1  ▶
[519 rows x 14 columns]
```

```
# Se eliminan los campos que no son requridos del DataFrame
# titanic
titanic.drop(columns=['acompañantes_1','acompañantes_2',
            'acompañantes_3'],
            inplace=True)
```

FIN DEL LAB

CAPÍTULO 10:
Transformación de cadenas

Las principales tareas de manejo de cadenas son:

1. ***Columnas con diferente casing.*** La columna es una equivalencia, en donde lo que varía es el uso de mayúsculas y minúsculas. Estas son algunas variantes. 1. Cambiar a mayúsculas. 1. Cambiar a minúsculas. 1. Cambiar a mayúsculas solo las primeras letras de las palabras.
2. ***División de cadenas.*** La columna es la división de una columna de tipo cadena, en dos o más.
3. ***Eliminar espacios en los extremos.*** La columna es el resultado de eliminar espacios innecesarios que se encuentren en los extremos de la cadena.
4. ***Concatenación de cadenas.*** La columna es la concatenación de dos o más columnas.
5. ***Extracción de sub-cadena.*** La columna es una porción de una cadena.

Columnas con diferente casing

El ***casing*** refiere al uso de mayúsculas y minúsculas para representar los datos.

Hay datos que usualmente van totalmente en minúsculas (correos electrónicos); hay otros datos que van generalmente todo en mayúsculas (categóricos descriptivos), y otros que van con la primera letra de cada palabra en mayúsculas y el resto en minúsculas (descripciones).

Para transformar el *casing* para columnas de tipo **`object`** (**`str`**), las funciones string a usar son las siguientes:

1. **`upper()`**: Pasa todo a mayúsculas.
2. **`lower()`**: Pasa todo a minúsculas.
3. **`title()`**: Pone la primera letra de cada palabra en mayúsculas, y el resto en minúsculas.

Es importante recordar que todas las funciones string requieren anteponer el prefijo **`str`** al ser invocadas.

Si tenemos un DataFrame llamado **`df`**, que tiene una columna llamada **`descripción`**, podríamos hacer lo siguiente:

```
# Cambiar descripción a mayúsculas
df['descripción']=df['descripción'].str.upper()

# Cambiar descripción a minúsculas
df['descripción']=df['descripción'].str.lower()

# Cambiar descripción a mayúscula la primera, y el resto
# minúsculas
df['descripción']=df['descripción'].str.upper()
```

División de cadenas

La derivación de columnas por ***división de cadenas*** consiste en partir de alguna manera un campo existente de tipo cadena, en otros campos.

Para la separación de cadenas ocupamos de un símbolo que actúe como separador, y hacer uso de la función string **`split()`**.

Imagina que tienes un DataFrame llamado **`df`**, que tiene un campo llamado **`origen`**, que almacena el nombre de la ciudad, estado / provincia donde se encuentra la ciudad, y el país, que corresponde al origen de una mercancía. Supón además que cada elemento de la ubicación está separado por el símbolo punto y coma (**`;`**).

Un ejemplo de ese dato sería como sigue: `Monterrey;Nuevo León;México`

La forma de generar campos llamados **`ciudad`**, **`estado_provincia`** y **`país`**, para almacenar los datos correspondientes, sería de la siguiente forma.

```
df['ciudad']=df['origen'].str.split(';').str.get(0)
df['estado_provincia']=df['origen'].str.split(';').str.get(1)
df['país']=df['origen'].str.split(';').str.get(2)
```

1. Lo primero que se genera es una Serie pandas que contiene todas las porciones de la cadena que están delimitados por punto y coma, usando la función string **`split()`**.

2. Al haber dos puntos y coma en el texto, se generan tres porciones. Como las Series pandas tienen un índice base cero, lo que está antes del primer punto y coma es el elemento 0, lo que está después del primer punto y coma, pero antes del segundo es el elemento 1, y lo que está después del segundo punto y coma es el elemento 2.

3. Posteriormente, se recupera la porción específica, usando la función **`get()`**, a la cual le proporcionamos el subíndice deseado.

4. Es posible usar **`str[0]`** en lugar de **`str.get(0)`**, con una importante diferencia: Si usamos el subíndice directamente, y no existe el elemento, el programa generará un error; si se usa **`get()`**, la falta de elemento retornara **`NaN`**, pero no generará error.

Otra forma de hacer el trabajo en una sola línea sería:

```
df[['ciudad','estado_provincia','país']]=df['origen'].
      str.split(';', expand=True)
```

El parámetro **`expand=True`** permite que la función regrese el valor de una columna para cada elemento de la división.

Es posible que estas sintaxis anteriores envíen advertencias en algunos casos, debido a que se está trabajando con sub-cadenas de las columnas; esto sucede particularmente cuando la transformación se asigna a la misma columna tomada como base de la transformación. En general, estas advertencias no implican problemas o errores.

Una alternativa para evitar esto sería utilizar **`loc[]`**, manejando *slices* abiertos (que no especifican inicio ni fin).

```
df.loc[:,'ciudad']=df['origen'].str.split(';').
      str.get(0).str.strip()
df.loc[:,'ciudad']=df['origen'].str.split(';').
      str.get(1).str.strip()
df.loc[:,'ciudad']=df['origen'].str.split(';').
      str.get(2).str.strip()
```

Puedes indicarle a Python que ignore las advertencias temporalmente, importando la librería **`warnings`** y usando el método **`filterwarnings()`**:

```
import warnings
warnings.filterwarnings("ignore")
```

Para habilitarlos de nuevo, puedes usar el método **`resetwarnings()`**:

```
import warnings
warnings.resetwarnings()
```

Eliminar espacios en los extremos

Es muy común que, al importar datos de diferentes fuentes, en algunos casos se agreguen espacios en blanco al inicio o al final de las cadenas, que son innecesarios y están de más.

La derivación de columnas por ***eliminación de espacios en los extremos*** consiste en eliminar espacios en blanco al inicio o al final de una columna de tipo cadena.

Para realizar esta operación se hace uso de la función string **`strip()`**.

Imagina que tienes un DataFrame llamado **`df`**, que contiene una columna llamada **`descripción`** que fue recuperada desde un sistema que con frecuencia agrega espacios innecesarios en los extremos de las cadenas.

Para eliminar los probables espacios en los extremos, podemos hacer lo siguiente:

```
df['descripción']=df['descripción'].str.strip()
```

Concatenación de cadenas

La derivación de columnas por ***concatenación de cadenas*** consiste en concatenar dos o más columnas de tipo cadena, para formar otra nueva.

Para realizar esta operación se hace uso del operador **`+`** (concatenación, cuando lo que se une son cadenas).

Imagina que tienes un DataFrame llamado **`df`**, que contiene una columna llamada **`nombre`** y otra llamada **`apellidos`**, y que juntas pueden formar otro campo llamado **`nombre_completo`**.

Concatenar las columnas para formar otra nueva, sería así:

```
df['nombre_completo']=df['nombre']+' '+df['apellidos']
```

Extracción de subcadenas

La derivación de columnas por ***extracción de subcadenas*** consiste en extraer una porción de una cadena.

Para realizar este tipo de operaciones, se utiliza mucho los ***slicers*** de Python, que es una forma de referir posiciones en una cadena.

Partamos de la idea que las cadenas son vistas por Python como series de caracteres, a los cuales le corresponde un subíndice a cada uno. A continuación, tenemos una cadena, y los subíndices que le corresponde a cada letra.

```
A P R E N D A
0 1 2 3 4 5 6
```

Se le conoce como *slicer* es un mecanismo para acceder a una porción de una serie, basada en la siguiente notación:

```
[inicio:fin]
```

Donde **`inicio`** es el primer símbolo de la porción, y **`fin`** es un símbolo después del fin de la porción, es decir, no se incluye.

También se pueden incluir referencias negativas, en cuyo caso, se inicia de atrás para adelante, sin ser base cero, sino base 1.

Si asignamos la palabra **APRENDA** a la variable **x**, podemos tener los siguientes resultados.

Slicer	Resultado
`x[2]`	R
`x[2,5]`	REN
`x[0:3]`	APR
`x[:3]`	APR
`x[3:]`	ENDA
`x[-1]`	A
`x[-3:-1]`	ND

Por ejemplo, si tienes un DataFrame llamado **df**, que contiene una columna llamada **código_producto**, que es de tipo **object** (cadena). El contenido de la columna es un código donde las primeras 3 posiciones son la línea de producto, un guion, y 6 dígitos que representan un consecutivo. Por ejemplo:

```
ELE-738829
```

Para generar una columna llamada **línea**, que contenga los primeros 3 caracteres de del código, sería así.

```
df['linea']=df['código_producto'][0:4]
```

Aquí tienes algunos ejemplos y comprobación del uso de *slicers*.

```
x='APRENDA'

for letra in x:
  print(letra)

print(x[2])
print(x[2:5])
print(x[0:3])
print(x[:3])
print(x[3:])
print(x[-1])
print(x[-3:-1])
```

```
A
P
R
E
N
D
A
R
```

```
REN
APR
APR
ENDA
A
ND
```

Los *slicers* son muy útiles cuando se trabaja con datos compuestos.

Los ***datos compuestos*** son aquellos que contienen porciones con significado. Por ejemplo, hay códigos de productos cuyos primeros 3 posiciones indican la línea de producto, las siguientes 5 posiciones son dígitos y corresponden al consecutivo, y la última posición indica si el origen es nacional (`'N'`), o importado (`'I'`), por ejemplo:

```
T U B 0 0 8 2 9 N
0 1 2 3 4 5 6 7 8
```

Imagina que tienes una columna llamada `id_producto` en un DataFrame llamado **`df`**, con las características descritas.

Para generar una columna con cada una de las porciones significativas, sería de la siguiente manera.

```
df['linea']=df['id_producto'].str[0:3]
df['consecutivo']=df['id_producto'].str[3:8]
df['origen']=df['id_producto'].str[-1]
```

Otra alternativa es usar el método **`slice()`**

```
df['linea']=df['id_producto'].str.slice(0,3)
df['consecutivo']=df['id_producto'].str.slice(3,8)
df['origen']=df['id_producto'].str.slice(-1)
```

LAB 10.01: Cálculos con columnas

En este Lab se desea generar nuevas columnas, o transformar las existentes, aplicando técnicas de cálculo de valores a partir de los datos existentes en el DataFrame.

Las tareas por realizar son:

1. Eliminar las advertencias de Python.
2. Modificar el *casing* de las columnas, usando **`lower()`**, **`upper()`**, y **`title()`**.
3. Derivar columnas por separación, usando **`split()`**.
    a. Alternativa usando **`split()`**.
    b. Alternativa usando **`split()`** con **`extend`**.
    c. Alternativa usando **`loc[]`**.
4. Extraer el título de la persona.
5. Eliminar los espacios en los extremos, usando **`strip()`**.
6. Derivar columnas por concatenación.
7. Derivar columnas por extracción.

Habilita las advertencias de Python

```
# Datos base
import pandas as pd
personas_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
personas_titanic_v3.csv'
titanic = pd.read_csv(personas_titanic_csv)
```

Eliminar las advertencias de Python

```
# Se inhabilitan temporalmente las advertencias de Python.
# Se establece un filtro de advertencias, para que las ignore.
import warnings
warnings.filterwarnings("ignore")
```

Modificar el casing de las columnas, usando lower(), upper(), y title()

Modifica el casing de las siguientes columnas:

1. **sexo**: Pasarlo todo a mayúsculas.
2. **nombre**: Pasarlo a primera letra de cada palabra en mayúscula y el resto a minúscula (ya está así, pero no en todos los casos).

```
# Se revisa el estado actual de los datos, en cuanto a casing.
titanic[['sexo','nombre']]
```

```
        sexo                                 nombre
0     hombre                 Andrews, Mr. Thomas Jr
1     hombre  Chisholm, Mr. Roderick Robert Crispin
2     hombre                      Fry, Mr. Richard
3     hombre                  Harrison, Mr. William
4     hombre               Ismay, Mr. Joseph Bruce
...      ...                                   ...
1305   mujer        Sage, Mrs. John (Annie Bullen)
1306  hombre                   Storey, Mr. Thomas
1307  hombre                  Baumann, Mr. John D
1308  hombre           Blackwell, Mr. Stephen Weart
1309  hombre               Baxter, Mr. Quigg Edmond
[1310 rows x 2 columns]
```

```
# Se pasa sexo a mayúsculas.
titanic['sexo']=titanic['sexo'].str.upper()

# Se pasa nombre a mayúscula la primera letra de cada palabra.
titanic['nombre']=titanic['nombre'].str.title()

# Se revisa el estado actual de los datos, en cuanto a casing.
titanic[['sexo','nombre']]
```

```
        sexo                                  nombre
0     HOMBRE                 Andrews, Mr. Thomas Jr
1     HOMBRE  Chisholm, Mr. Roderick Robert Crispin
2     HOMBRE                       Fry, Mr. Richard
3     HOMBRE                 Harrison, Mr. William
4     HOMBRE                Ismay, Mr. Joseph Bruce
...      ...                                   ...
1305   MUJER         Sage, Mrs. John (Annie Bullen)
1306  HOMBRE                     Storey, Mr. Thomas
1307  HOMBRE                   Baumann, Mr. John D
1308  HOMBRE            Blackwell, Mr. Stephen Weart
1309  HOMBRE                Baxter, Mr. Quigg Edmond
[1310 rows x 2 columns]
```

Derivar columnas por separación, usando split()

Realiza las siguientes tareas:

1. Genera un DataFrame de trabajo llamado **cadenas**, que contenga solo los campos **id_persona**, **nombre**, y **sexo** del DataFrame **titanic**.
2. A partir de **nombre**, usando como separador la coma, genera dos columnas: **apellidos**, con la primera parte de la cadena, y **primer_nombre**, con la segunda parte de la cadena. Utiliza varias estrategias.
3. A partir de **primer_nombre**, extrae el título de la persona. Utiliza la estrategia que quieras.

```
# Se genera un nuevo DataFrame de trabajo llamado cadenas
cadenas=titanic[['id_persona','nombre','sexo']]

cadenas
```

```
      id_persona                                  nombre    sexo
0          10000                 Andrews, Mr. Thomas Jr  HOMBRE
1          10001  Chisholm, Mr. Roderick Robert Crispin  HOMBRE
2          10002                       Fry, Mr. Richard  HOMBRE...          ...
...      ...
1307       11307                   Baumann, Mr. John D  HOMBRE
1308       11308            Blackwell, Mr. Stephen Weart  HOMBRE
1309       11309                Baxter, Mr. Quigg Edmond  HOMBRE
[1310 rows x 3 columns]
```

1. **Alternativa usando `split()`.**

```
# Se derivan nuevas columnas por separación, usando split,
# extrayendo elemento por elemento.
# La primera parte de la cadena (índice 0) formará la
# columna apellidos.
# La segunda parte de la cadena (índice 1) formará la
# columna primer_nombre.
cadenas['apellidos']=cadenas['nombre'].
     str.split(',').str.get(0)
cadenas['primer_nombre']=cadenas['nombre'].
      str.split(',').str.get(1)

# Se muestra el resultado
cadenas
```

```
      id_persona                                 nombre    sexo  apellidos  \
0          10000                Andrews, Mr. Thomas Jr  HOMBRE    Andrews
1          10001  Chisholm, Mr. Roderick Robert Crispin  HOMBRE   Chisholm
2          10002                      Fry, Mr. Richard  HOMBRE        Fry
3          10003                Harrison, Mr. William  HOMBRE   Harrison
4          10004               Ismay, Mr. Joseph Bruce  HOMBRE      Ismay
...          ...                                  ...     ...        ...
1305       11305         Sage, Mrs. John (Annie Bullen)   MUJER       Sage
1306       11306                    Storey, Mr. Thomas  HOMBRE     Storey
1307       11307                  Baumann, Mr. John D   HOMBRE    Baumann
1308       11308           Blackwell, Mr. Stephen Weart  HOMBRE  Blackwell
1309       11309               Baxter, Mr. Quigg Edmond  HOMBRE     Baxter

                    primer_nombre
0                    Mr. Thomas Jr
1      Mr. Roderick Robert Crispin
2                      Mr. Richard
3                      Mr. William
4                 Mr. Joseph Bruce
...                            ...
1305      Mrs. John (Annie Bullen)
1306                     Mr. Thomas
1307                     Mr. John D
1308              Mr. Stephen Weart
1309               Mr. Quigg Edmond
[1310 rows x 5 columns]
```

2. **Alternativa usando `split()` con `extend`.**

```
# Otra alternativa, en una sola línea, usando expand=True.
# Se coloca una lista de columnas a crear, mismas que se
# poblarán con las correspondientes porciones de la cadena,
# en el orden en que aparecen.
cadenas[['apellidos','primer_nombre']]=cadenas['nombre'].
      str.split(',', expand=True)

# Se muestra el resultado
cadenas
```

```
      id_persona                                    nombre    sexo  apellidos  \
0          10000                   Andrews, Mr. Thomas Jr  HOMBRE    Andrews
1          10001  Chisholm, Mr. Roderick Robert Crispin  HOMBRE   Chisholm
2          10002                         Fry, Mr. Richard  HOMBRE        Fry
3          10003                  Harrison, Mr. William  HOMBRE   Harrison
4          10004                Ismay, Mr. Joseph Bruce  HOMBRE      Ismay
...          ...                                    ...     ...        ...
1305       11305         Sage, Mrs. John (Annie Bullen)   MUJER       Sage
1306       11306                   Storey, Mr. Thomas  HOMBRE     Storey
1307       11307                  Baumann, Mr. John D  HOMBRE    Baumann
1308       11308           Blackwell, Mr. Stephen Weart  HOMBRE  Blackwell
1309       11309               Baxter, Mr. Quigg Edmond  HOMBRE     Baxter

                    primer_nombre
0                    Mr. Thomas Jr
1      Mr. Roderick Robert Crispin
2                      Mr. Richard
3                      Mr. William
4                Mr. Joseph Bruce
...                           ...
1305      Mrs. John (Annie Bullen)
1306                     Mr. Thomas
1307                     Mr. John D
1308             Mr. Stephen Weart
1309              Mr. Quigg Edmond
[1310 rows x 5 columns]
```

3. Alternativa usando `loc[]`.

```
# Otra alternativa, usando loc[]
cadenas.loc[:,'apellidos']=cadenas['nombre'].
      str.split(',').str.get(0)
cadenas.loc[:,'primer_nombre']=cadenas['nombre'].
      str.split(',').str.get(1)

# Se muestra el resultado
cadenas
```

```
      id_persona                                 nombre    sexo  apellidos  \
0          10000                 Andrews, Mr. Thomas Jr  HOMBRE    Andrews
1          10001  Chisholm, Mr. Roderick Robert Crispin  HOMBRE   Chisholm
2          10002                        Fry, Mr. Richard  HOMBRE        Fry
3          10003                  Harrison, Mr. William  HOMBRE   Harrison
4          10004                Ismay, Mr. Joseph Bruce  HOMBRE      Ismay
...          ...                                    ...     ...        ...
1305       11305         Sage, Mrs. John (Annie Bullen)   MUJER       Sage
1306       11306                   Storey, Mr. Thomas  HOMBRE     Storey
1307       11307                  Baumann, Mr. John D  HOMBRE    Baumann
1308       11308            Blackwell, Mr. Stephen Weart  HOMBRE  Blackwell
1309       11309                Baxter, Mr. Quigg Edmond  HOMBRE     Baxter

                     primer_nombre
0                    Mr. Thomas Jr
1      Mr. Roderick Robert Crispin
2                      Mr. Richard
3                      Mr. William
4                Mr. Joseph Bruce
...                           ...
1305      Mrs. John (Annie Bullen)
1306                    Mr. Thomas
1307                    Mr. John D
1308             Mr. Stephen Weart
1309               Mr. Quigg Edmond

[1310 rows x 5 columns]
```

Extraer el título de la persona

```
# Para obtener el título de la persona (Miss, Mr, Mrs,
# Dr, etc), y colocarlo en una columna llamada titulo.
# Se divide a partir del punto, y se extrae la primera porción.
cadenas['titulo']=cadenas['primer_nombre'].
      str.split('.').str.get(0)

# Se muestra el resultado
cadenas
```

```
      id_persona                                  nombre    sexo  apellidos  \
0          10000                   Andrews, Mr. Thomas Jr  HOMBRE    Andrews
1          10001  Chisholm, Mr. Roderick Robert Crispin  HOMBRE   Chisholm
2          10002                            Fry, Mr. Richard  HOMBRE        Fry
3          10003                     Harrison, Mr. William  HOMBRE   Harrison
4          10004                  Ismay, Mr. Joseph Bruce  HOMBRE      Ismay
...          ...                                     ...     ...        ...
1305       11305          Sage, Mrs. John (Annie Bullen)   MUJER       Sage
1306       11306                       Storey, Mr. Thomas  HOMBRE     Storey
1307       11307                    Baumann, Mr. John D  HOMBRE    Baumann
1308       11308             Blackwell, Mr. Stephen Weart  HOMBRE  Blackwell
1309       11309                 Baxter, Mr. Quigg Edmond  HOMBRE     Baxter

                     primer_nombre titulo
0                      Mr. Thomas Jr     Mr
1       Mr. Roderick Robert Crispin      Mr
2                        Mr. Richard     Mr
3                        Mr. William     Mr
4                  Mr. Joseph Bruce      Mr
...                              ...    ...
1305       Mrs. John (Annie Bullen)    Mrs
1306                      Mr. Thomas     Mr
1307                      Mr. John D     Mr
1308               Mr. Stephen Weart     Mr
1309                Mr. Quigg Edmond     Mr

[1310 rows x 6 columns]
```

Eliminar los espacios en los extremos, usando strip()

Realiza las siguientes tareas:

1. Elimina espacios en los extremos de las columnas **apellidos** y **primer_nombre**.
2. Utiliza el DataFrame **cadenas** y la función **strip()**.

```
# Se eliminan los espacios en los extremos de las
# columnas indicadas.
cadenas['apellidos']=cadenas['apellidos'].str.strip()
cadenas['primer_nombre']=cadenas['primer_nombre'].str.strip()

# Se muestra el resultado.
cadenas
```

```
      id_persona                                  nombre    sexo  apellidos  \
0          10000                 Andrews, Mr. Thomas Jr  HOMBRE    Andrews
1          10001  Chisholm, Mr. Roderick Robert Crispin  HOMBRE   Chisholm
2          10002                         Fry, Mr. Richard  HOMBRE        Fry
3          10003                  Harrison, Mr. William  HOMBRE   Harrison
4          10004                Ismay, Mr. Joseph Bruce  HOMBRE      Ismay
...          ...                                    ...     ...        ...
1305       11305         Sage, Mrs. John (Annie Bullen)   MUJER       Sage
1306       11306                  Storey, Mr. Thomas  HOMBRE     Storey
1307       11307                 Baumann, Mr. John D  HOMBRE    Baumann
1308       11308           Blackwell, Mr. Stephen Weart  HOMBRE  Blackwell
1309       11309               Baxter, Mr. Quigg Edmond  HOMBRE     Baxter

                    primer_nombre titulo
0                   Mr. Thomas Jr     Mr
1     Mr. Roderick Robert Crispin     Mr
2                     Mr. Richard     Mr
3                     Mr. William     Mr
4                Mr. Joseph Bruce     Mr
...                           ...    ...
1305     Mrs. John (Annie Bullen)    Mrs
1306                   Mr. Thomas     Mr
1307                   Mr. John D     Mr
1308            Mr. Stephen Weart     Mr
1309              Mr. Quigg Edmond     Mr

[1310 rows x 6 columns]
```

Derivar columnas por concatenación

Realiza las siguientes tareas:

1. Concatena los campos **`primer_nombre`** y **`apellidos`**, en una nueva columna llamada **`nombre_completo`**.
2. Ten cuidado de agregar un espacio intermedio entre ambas columnas, para que el contenido no quede junto.
3. Utiliza el DataFrame **`cadenas`**.

```
# Se concatenan las columnas deseadas (pueden generarse
# advertencias).
cadenas['nombre_completo']=cadenas['primer_nombre']+
      ' '+cadenas['apellidos']

# Se muestra el resultado.
cadenas
```

```
      id_persona                                  nombre    sexo  apellidos  \
0          10000                   Andrews, Mr. Thomas Jr  HOMBRE    Andrews
1          10001  Chisholm, Mr. Roderick Robert Crispin  HOMBRE   Chisholm
2          10002                         Fry, Mr. Richard  HOMBRE        Fry
3          10003                    Harrison, Mr. William  HOMBRE   Harrison
4          10004                  Ismay, Mr. Joseph Bruce  HOMBRE      Ismay
...          ...                                     ...     ...        ...
1305       11305          Sage, Mrs. John (Annie Bullen)   MUJER       Sage
1306       11306                      Storey, Mr. Thomas  HOMBRE     Storey
1307       11307                    Baumann, Mr. John D  HOMBRE    Baumann
1308       11308             Blackwell, Mr. Stephen Weart  HOMBRE  Blackwell
1309       11309                 Baxter, Mr. Quigg Edmond  HOMBRE     Baxter

                    primer_nombre titulo                       nombre_completo
0                    Mr. Thomas Jr     Mr                  Mr. Thomas Jr Andrews
1     Mr. Roderick Robert Crispin      Mr  Mr. Roderick Robert Crispin Chisholm
2                      Mr. Richard     Mr                        Mr. Richard Fry
3                      Mr. William     Mr                   Mr. William Harrison
4                Mr. Joseph Bruce      Mr                Mr. Joseph Bruce Ismay
...                             ...    ...                                   ...
1305     Mrs. John (Annie Bullen)    Mrs         Mrs. John (Annie Bullen) Sage
1306                    Mr. Thomas     Mr                      Mr. Thomas Storey
1307                    Mr. John D     Mr                    Mr. John D Baumann
1308             Mr. Stephen Weart     Mr            Mr. Stephen Weart Blackwell
1309              Mr. Quigg Edmond     Mr                Mr. Quigg Edmond Baxter

[1310 rows x 7 columns]
```

Derivar columnas por extracción

Realiza las siguientes tareas:

1. Genera una nueva columna llamada **`clave_sexo`**, que contenga la primera letra de la columna **`sexo`**.
2. Trabaja con el DataFrame **`cadenas`**.
3. Utiliza *slices* para hacer la extracción de la subcadena requerida.

```
# Se extrae el primer símbolo de la cadena, usando slicers
# y se genera una nueva columna (puede hacer advertencias).
cadenas['clave_sexo']=cadenas['sexo'].str[0]

# Se ven los resultados
cadenas
```

```
      id_persona                                  nombre    sexo  apellidos  \
0          10000                   Andrews, Mr. Thomas Jr  HOMBRE    Andrews
1          10001  Chisholm, Mr. Roderick Robert Crispin  HOMBRE   Chisholm
2          10002                         Fry, Mr. Richard  HOMBRE        Fry
3          10003                   Harrison, Mr. William  HOMBRE   Harrison
4          10004                Ismay, Mr. Joseph Bruce  HOMBRE      Ismay
...          ...                                     ...     ...        ...
1305       11305          Sage, Mrs. John (Annie Bullen)   MUJER       Sage
1306       11306                    Storey, Mr. Thomas  HOMBRE     Storey
1307       11307                   Baumann, Mr. John D  HOMBRE    Baumann
1308       11308            Blackwell, Mr. Stephen Weart  HOMBRE  Blackwell
1309       11309                Baxter, Mr. Quigg Edmond  HOMBRE     Baxter


                        nombre_completo clave_sexo
0                   Mr. Thomas Jr Andrews          H
1     Mr. Roderick Robert Crispin Chisholm          H
2                          Mr. Richard Fry          H
3                    Mr. William Harrison          H
4                   Mr. Joseph Bruce Ismay          H
...                                    ...        ...
1305         Mrs. John (Annie Bullen) Sage          M
1306                    Mr. Thomas Storey          H
1307                   Mr. John D Baumann          H
1308           Mr. Stephen Weart Blackwell          H
1309               Mr. Quigg Edmond Baxter          H

[1310 rows x 8 columns]
```

Habilita las advertencias de Python

```
# Se habilitan de nuevo las advertencias de Python
warnings.resetwarnings()
```

FIN DEL LAB

CAPÍTULO 11:

Tratamiento de categóricos e integración de datos

Las principales tareas de manejo de categóricos son las siguientes:

1. ***Equivalencias categóricas.*** La columna es un categórico descriptivo, que es equivalente a un categórico codificado, numérico o dicotómico.
2. ***Equivalencia categórica de intervalo.*** La columna es un categórico descriptivo, que se genera a partir de un campo de valor, considerando rangos de intervalo.

Equivalencias categóricas

Los datos ***categóricos*** son esenciales para los trabajos de analítica de datos, porque son la manera en que se representan la mayoría de las categorizaciones y segmentaciones.

El registro de los empleados de una compañía puede tener muchos categóricos, por ejemplo, departamento, dirección, puesto, género, y así. Posteriormente, en los trabajos de analítica podemos usar esos categóricos para filtrados y consolidaciones de datos.

Los datos categóricos tienen dos características primordiales:

1. Están definidos previamente en un catálogo de categorías. No es algo que se vaya incrementando y formando sobre la marcha.
2. Admiten valores repetidos en lo conjuntos que los utilizan.

Tipos de categóricos

Existen diferentes tipos de categóricos:

1. ***Categóricos descriptivos:*** Son los categóricos que no requieren interpretación. Por ejemplo, si tenemos un categórico llamado `ciudad`, el categórico descriptivo sería **NEW YORK**, **TOKIO**, **MADRID**, **CIUDAD DE MÉXICO**, **BOGOTÁ**.
    a. Los valores son totalmente descriptivos, y no requieren interpretación adicional para saber a qué ciudad se refieren, fuera de toda duda.

2. ***Categóricos numéricos:*** Son los categóricos representados por un número que requieren interpretación.
    a. Por ejemplo, las ciudades que mencionamos anteriormente pueden ser **1**, **2**, **3**, **4**, **5**, y para saber cuál es cual, se requiere una interpretación.

3. ***Categóricos codificados:*** Son los categóricos representados por un código que requiere interpretación.
    a. Por ejemplo, las ciudades que mencionamos anteriormente pueden ser **NY**, **TK**, **MD**, **MX**, **BG**.

4. ***Categóricos dicotómicos:*** Son los categóricos representados por un par de valores mutuamente excluyentes.
    a. Pueden ser descriptivos, numéricos o codificados; su característica es que se trata de dos valores mutuamente excluyentes.

5. ***Categóricos de intervalo:*** Son los categóricos que se asignan en función a una columna numérica que representa una escala que se divide en múltiples rangos de intervalos, o clases.

a. Dependiendo del campo numérico, se asigna un categórico, que puede ser de cualquier tipo (descriptivo, numérico, codificado o dicotómico).

Lo usual es disponer de categóricos numéricos, codificados, dicotómicos y de intervalo en la fase de integración de datos, pero antes de los trabajos de analítica que tienen que ver con la visualización, es común traducir o generar una equivalencia a categórico descriptivo.

Generando equivalencia usando map()

Se puede generar una equivalencia categórica con la ayuda de un diccionario, y **`map()`**.

Supongamos que tenemos las siguientes equivalencias para los valores **`NEW YORK`**, **`TOKIO`**, **`MADRID`**, **`CIUDAD DE MÉXICO`**, **`BOGOTÁ`**.

```
ubicaciones=pd.DataFrame({
    'numerico':[1,2,3,4,5],
    'codificado':['NY','TK','MD','MX','BG'],
    'habla_español':[0,0,1,1,1]
})

ubicaciones
```

```
   numerico codificado  habla_español
0         1         NY              0
1         2         TK              0
2         3         MD              1
3         4         MX              1
4         5         BG              1
```

Para hacer la equivalencia usando **`map()`**:

1. Se define en un diccionario las equivalencias.
2. Se aplica la función **`map()`** a la columna tomada como base.
3. Se pasa como parámetro el diccionario con las equivalencias.

```
equivalencias={
    1:'NEW YORK',
    2:'TOKIO',
    3:'MADRID',
    4:'CIUDAD DE MÉXICO',
    5:'BOGOTÁ'
}

ubicaciones['descriptivo_1']=ubicaciones['numerico'].
      map(equivalencias)

ubicaciones
```

	numerico	codificado	habla_español	descriptivo_1
0	1	NY	0	NEW YORK
1	2	TK	0	TOKIO
2	3	MD	1	MADRID
3	4	MX	1	CIUDAD DE MÉXICO
4	5	BG	1	BOGOTÁ

Generando equivalencia usando merge()

Se puede generar una equivalencia categórica uniendo los datos de dos DataFrame, haciendo un *Join* usando **`merge()`**.

Esta técnica es más común cuando estamos integrando datos desde diferentes fuentes. Imagina que tienes un conjunto de datos en un DataFrame, y tienes un catálogo con los categóricos descriptivos en otro DataFrame.

Ambos DataFrames deben tener una columna en común (***columna de coincidencia***), que les permite establecer una relación entre ellos.

En uno de los DataFrames, la columna de coincidencia es de identidad (***llave***), y tiene unicidad, mientras que en el otro DataFrame, es un categórico que generalmente admite repetidos.

En bases de datos relacionales, estaríamos hablando de la tabla fuerte y la tabla débil, y trataríamos de hacer un *inner join*. El concepto es el mismo.

Cuando usamos bases de datos, la relación entre tablas permite que la tabla fuerte sea capaz de trasladarle datos a la tabla débil, de manera inequívoca, a lo que se conoce como ***propagación de datos***. Al momento de usar

merge(), también se hace una propagación de datos, desde el DataFrame fuerte hacia el DataFrame débil.

Al aplicar **merge()** las columnas del DataFrame fuerte se agregan al DataFrame débil, que es donde se aplica la función.

```
debil=pd.DataFrame({
    'preferencia':[1,2,3,4,5,6,7,8,9],
    'clave_ciudad':['NY','TK','MD','NY','MX','BG',
          'TK','BR','TK']
     })

fuerte=pd.DataFrame({
    'clave_ciudad':['NY','TK','MD','MX','BG','BA'],
    'nombre_ciudad':['NEW YORK','TOKIO','MADRID',
          'CIUDAD DE MÉXICO', 'BOGOTÁ','BUENOS AIRES']
     })

print(debil)

print(fuerte)
```

```
   preferencia clave_ciudad
0            1           NY
1            2           TK
2            3           MD
3            4           NY
4            5           MX
5            6           BG
6            7           TK
7            8           BR
8            9           TK

  clave_ciudad     nombre_ciudad
0           NY          NEW YORK
1           TK             TOKIO
2           MD            MADRID
3           MX  CIUDAD DE MÉXICO
4           BG            BOGOTÁ
5           BA      BUENOS AIRES
```

En este ejemplo tenemos dos DataFrames.

Podemos apreciar que el campo de coincidencia es **clave_ciudad**. En **debil**, la columna es categórica, mientras que en **fuerte** es de identidad. Toma nota que en ambos DataFrames hay valores que no tienen coincidencia en el otro DataFrame.

```
# Inner join. Se le agrega (merge) al DataFrame debil
# los datos de la tabla fuerte, tomando como campo de
# coincidencia clave_ciudad

debil=debil.merge(fuerte,on='clave_ciudad',how='inner')

débil
```

```
   preferencia clave_ciudad     nombre_ciudad
0            1           NY          NEW YORK
1            4           NY          NEW YORK
2            2           TK             TOKIO
3            7           TK             TOKIO
4            9           TK             TOKIO
5            3           MD            MADRID
6            5           MX  CIUDAD DE MÉXICO
7            6           BG            BOGOTÁ
```

Toma en cuenta que en el DataFrame débil hay una fila cuyo valor en **`clave_ciudad`** es **`BR`**, que no está en el DataFrame fuerte, por lo que no se incluye en el resultado final.

En ese sentido, **`merge()`** se comporta como un *inner join* en la terminología de bases de datos relacionales. Si se desea hacer un *outer join* o *full join*, se tiene que usar el parámetro **`how=outer`** y **`how=full.`**

```
# Outer Join. Se agregan todos los registros ambos DataFrames
# y en caso de no haber coincidencias, rellena
# con NaN. Se usa how con valor 'outer', o 'left'.

debil=debil.merge(fuerte,on='clave_ciudad',how='outer')

débil
```

```
   preferencia clave_ciudad   nombre_ciudad_x   nombre_ciudad_y
0          1.0           NY          NEW YORK          NEW YORK
1          4.0           NY          NEW YORK          NEW YORK
2          2.0           TK             TOKIO             TOKIO
3          7.0           TK             TOKIO             TOKIO
4          9.0           TK             TOKIO             TOKIO
5          3.0           MD            MADRID            MADRID
6          5.0           MX  CIUDAD DE MÉXICO  CIUDAD DE MÉXICO
7          6.0           BG            BOGOTÁ            BOGOTÁ
8          NaN           BA               NaN      BUENOS AIRES
```

```
# Outer Join. Se agregan todos los registros del DataFrame
# fuerte y en caso de no haber coincidencias en la tabla
# débil, rellena con NaN. Se usa how con valor 'right'.

debil=debil.merge(fuerte,on='clave_ciudad',how='right')

débil
```

```
   preferencia clave_ciudad   nombre_ciudad_x   nombre_ciudad_y  \
0          1.0           NY          NEW YORK          NEW YORK
1          4.0           NY          NEW YORK          NEW YORK
2          2.0           TK             TOKIO             TOKIO
3          7.0           TK             TOKIO             TOKIO
4          9.0           TK             TOKIO             TOKIO
5          3.0           MD            MADRID            MADRID
6          5.0           MX  CIUDAD DE MÉXICO  CIUDAD DE MÉXICO
7          6.0           BG            BOGOTÁ            BOGOTÁ
8          NaN           BA               NaN      BUENOS AIRES

      nombre_ciudad
0          NEW YORK
1          NEW YORK
2             TOKIO
3             TOKIO
4             TOKIO
5            MADRID
6  CIUDAD DE MÉXICO
7            BOGOTÁ
8      BUENOS AIRES
```

Habrá casos en donde la correspondencia entre conjuntos de datos se presente usando llaves compuestas. Una ***llave compuesta*** es aquella que se conforma por dos o más campos.

Supongamos que tenemos un DataFrame llamado **`almacenes`**, y otro llamado **`movimientos`**, y deben relacionarse usando dos campos, llamados **`almacén`** y **`consecutivo`**, que están presentes en ambos DataFrames. Para hacer la unión de campos en un DataFrame llamado **`detalle`**, sería así.

```
detalle=movimientos.merge(almacenes,
      on=['almacén','consecutivo'])
```

LAB 11.01: Generación de categóricos descriptivos equivalentes

En este Lab se desea generar nuevas columnas, o transformar las existentes, aplicando técnicas de cálculo de valores a partir de los datos existentes en el DataFrame.

Las tareas por realizar son:

1. Generación de categóricos usando **`map()`**.
2. Generación de categóricos a partir de otros archivos, usando **`merge()`**.
3. Generación de categóricos usando UDF's.

```
# Datos base
import pandas as pd
tipos_correctos={
    'id_persona':object,
    'clave_sobrevivencia':object,
    'clase_viaje':object
}
personas_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
personas_titanic_v3.csv'
titanic = pd.read_csv(personas_titanic_csv,
      dtype=tipos_correctos)
# Al momento de la lectura especificamos que los valores
# que son categóricos, los interprete como object, y no como
# enteros o flotantes.
# Es importante hacer notar que si pandas detecta datos
# que no sabe cómo convertir, por ejemplo NaN, proporcionará
# un tipo de datos más amplio, y por eso pone columnas
# que son enteras (como clave_sobrevivencia) como flotante,
# y al convertir a object será '0.0'.

# Para eliminar los '.0' de sobra, se aplica lo siguiente.

titanic['id_persona']=titanic['id_persona'].
      str.split('.').str.get(0)
titanic['clave_sobrevivencia']=titanic['clave_sobrevivencia'].
      str.split('.').str.get(0)
titanic['clase_viaje']=titanic['clase_viaje'].
      str.split('.').str.get(0)
```

Generación de categóricos usando map()

Realiza las siguientes tareas para generar un categórico descriptivo llamado **`sobrevivencia`**.

1. Toma como base el campo **`clave_sobrevivencia`**.
2. Si **`clave_sobrevivencia`** es **`'0'`**, es **`'MURIÓ'`**; si es **`'1'`**, es **`'VIVIÓ'`**.
3. Usa **`map()`**.

```
# Se definen las equivalencias para sobrevivencia
# usando un diccionario, donde la llave es el valor
# de referencia, y el valor es el categórico descriptivo.

catalogo_sobrevivencia={
    '0':'MURIÓ',
    '1':'VIVIÓ'
}

# Se genera una nueva columna, con las equivalencias
# categóricas, usando map sobre el campo a considerar
# como base para el cálculo proporcionando como parámetro
# el diccionario que contiene las equivalencias.
titanic['sobrevivencia']=titanic['clave_sobrevivencia'].
      map(catalogo_sobrevivencia)

# Se muestra el resultado.
titanic[['nombre','clave_sobrevivencia','sobrevivencia']]
```

```
                                 nombre clave_sobrevivencia sobrevivencia
0                  Andrews, Mr. Thomas Jr                  0         MURIÓ
1     Chisholm, Mr. Roderick Robert Crispin                  0         MURIÓ
2                         Fry, Mr. Richard                  0         MURIÓ
3                    Harrison, Mr. William                  0         MURIÓ
4                  Ismay, Mr. Joseph Bruce                  1         VIVIÓ
...                                    ...                ...           ...
1305        Sage, Mrs. John (Annie Bullen)                  0         MURIÓ
1306                    Storey, Mr. Thomas                  0         MURIÓ
1307                   Baumann, Mr. John D                  0         MURIÓ
1308          Blackwell, Mr. Stephen Weart                NaN           NaN
1309              Baxter, Mr. Quigg Edmond                  0         MURIÓ

[1310 rows x 3 columns]
```

Generación de categóricos a partir de otros archivos, usando merge()

Realiza las siguientes tareas para generar un categórico descriptivo a partir de los datos de un archivo que contiene un catálogo de clases.

1. Carga en un DataFrame llamado **`catalogo_clases`** el contenido del archivo **`clases.csv`**.
2. Usa **`merge()`** para agregar los datos de **`catalogo_clases`** al DataFrame **`titanic`**.
3. Toma como campo de coincidencia **`clase_viaje`**.

```
# Se importan las clases del catálogo, desde GitHub.

catalogo_clases = pd.read_csv('https://raw.githubusercontent.
com/AprendaPracticando/AnaliticaPythonR1/main/data/clases.csv')

catalogo_clases
```

```
   clase_viaje          clase
0            1  PRIMERA CLASE
1            2  SEGUNDA CLASE
2            3  TERCERA CLASE
```

```
# Revisamos los tipos de los atributos de coincidencia.
# Vemos que no coinciden.
print(titanic.dtypes['clase_viaje'])
print(catalogo_clases.dtypes['clase_viaje'])
```

```
object
int64
```

```
# Cambiamos el tipo de dato del DataFrame de catálogo,
# para homologar tipos.
catalogo_clases['clase_viaje']=catalogo_clases['clase_viaje'].
      astype('str')

# Se hace el merge, de tipo outer join, para que no elimine
# registros en caso de que clase sea un dato ausente.
titanic=titanic.merge(catalogo_clases,on='clase_viaje',
      how='outer')

# Se muestran los resultados
titanic[['nombre','clase_viaje','clase']]
```

```
                                  nombre clase_viaje          clase
0                    Andrews, Mr. Thomas Jr           1  PRIMERA CLASE
1     Chisholm, Mr. Roderick Robert Crispin           1  PRIMERA CLASE
2                           Fry, Mr. Richard          1  PRIMERA CLASE
3                      Harrison, Mr. William          1  PRIMERA CLASE
4                    Ismay, Mr. Joseph Bruce          1  PRIMERA CLASE
...                                      ...         ...           ...
1305         Sage, Mrs. John (Annie Bullen)           3  TERCERA CLASE
1306                     Storey, Mr. Thomas           3  TERCERA CLASE
1307                   Baumann, Mr. John D          NaN            NaN
1308           Blackwell, Mr. Stephen Weart         NaN            NaN
1309               Baxter, Mr. Quigg Edmond         NaN            NaN

[1310 rows x 3 columns]
```

Generación de categóricos usando UDF's.

Realiza las siguientes tareas, para generar un categórico descriptivo llamado **`acompañada`**.

1. Toma como base el campo **`acompañantes`**.
2. Si **`acompañantes`** es mayor a cero, debe ser **`'ACOMPAÑADA'`**, y si es cero, debe ser **`'SOLA'`**.
3. Usa funciones definidas por el usuario.

```
# Se define la función que evalúa la cantidad de acompañantes
# y retorna el categórico descriptivo correspondiente.
def acompañamiento(x):
  etiqueta='NO DEFINIDO'
  if (x['acompañantes']==0):
    etiqueta='SOLA'
  if (x['acompañantes']>0):
    etiqueta='ACOMPAÑADA'
  return etiqueta

# Se aplica la función a cada filas.
titanic['acompañada']=titanic.apply(acompañamiento, axis=1)

# Se ven los resultados.
titanic[['nombre','acompañantes','acompañada']]
```

```
                                 nombre  acompañantes  acompañada
0                    Andrews, Mr. Thomas Jr             0        SOLA
1     Chisholm, Mr. Roderick Robert Crispin             0        SOLA
2                           Fry, Mr. Richard             0        SOLA
3                      Harrison, Mr. William             0        SOLA
4                    Ismay, Mr. Joseph Bruce             0        SOLA
...                                      ...           ...         ...
1305         Sage, Mrs. John (Annie Bullen)            10  ACOMPAÑADA
1306                      Storey, Mr. Thomas             0        SOLA
1307                    Baumann, Mr. John D              0        SOLA
1308           Blackwell, Mr. Stephen Weart              0        SOLA
1309               Baxter, Mr. Quigg Edmond              1  ACOMPAÑADA

[1310 rows x 3 columns]
```

FIN DEL LAB

Equivalencia de intervalos

Un ***rango de intervalo*** es un término que se utiliza en estadística para referirse a un conjunto de valores consecutivos que se agrupan en intervalos de igual ancho, dentro de una escala.

Para definir un rango de intervalo, primero se establece el ***ancho de intervalo***, que son las unidades de medida que representan el rango, y que debe ser constante en todo el conjunto de datos. Luego, se dividen los datos en intervalos de igual tamaño, donde cada intervalo abarca una cierta cantidad de valores.

Por ejemplo, si estamos trabajando con un conjunto de datos de edad de una población, podemos definir intervalos de 10 años (por ejemplo, de 0-9, 10-19, 20-29, y así).

Cuando el ancho del intervalo no es constante (por ejemplo, si la edad va de 0 a 9, y de 10 a 25, y luego de 25 a 35), ya no se les llama rango de intervalos, sino ***clases***.

Tanto los rangos de intervalo como las clases, generalmente se indican con la letra **k**.

El rango de intervalo se utiliza comúnmente para crear histogramas, que son gráficos que muestran la distribución de los datos en diferentes intervalos. Los histogramas pueden ser útiles para visualizar patrones en los datos, como si están sesgados hacia un lado, si tienen una distribución normal, etc.

La notación matemática para representar rangos de intervalo utiliza los siguientes símbolos:

- **`Square Brackets []`**: Se utilizan para indicar que se trata de un intervalo cerrado, es decir, donde los límites del rango se incluyen en el rango.
- **`Rounded Brackets ()`**: Se utilizan para indicar que se trata de un intervalo abierto, es decir, donde los límites del rango no se incluyen en el rango.

- **Coma ,**: Se utilizan para separar el límite inferior y superior del rango de intervalo.

Puede darse el caso en que sólo uno de los límites se incluya, pero el otro no. De esa forma:

- **(a,b]** Intervalo semi-abierto, o intervalo semi-cerrado por la izquierda, donde el límite inferior no se incluye, pero el límite superior sí.
- **[a,b)** Intervalo semi-abierto, o intervalo semi-cerrado por la derecha, donde el límite inferior se incluye, pero el límite superior no.

Algunos ejemplos de notación de rangos de intervalo son los siguientes.

Rango de intervalo	Alcance
[0,10]	Valores mayores o iguales a cero, y menores o iguales a diez
(0,10)	Valores mayores a cero, y menores a diez
[0,10)	Valores mayores o iguales a cero, y menores a diez
(0,10]	Valores mayores a cero, y menores o iguales a diez

Al número de rangos de intervalos, se le conoce como ***número de clases***, y suele simbolizarse con la letra **k**, así que, si una escala se divide en 5 intervalos, **k=5**

Generalmente, se debe decidir si todos los límites inferiores se incluyen, o si todos los límites superiores se incluyen, pero no se recomienda mezclar.

En aquellos casos en los que se decide no incluir el límite inferior y sí el superior, suele hacerse una excepción en el primer intervalo, para evitar que los datos iguales al valor mínimo del intervalo queden fuera.

Pandas tiene la función **cut()** que permite manejar rangos de intervalo.

Estos son los más importantes parámetros de la función:

- **`columna de datos`**: La columna de datos que se va a dividir en intervalos.
- **`bins`**: Puede ser un entero que indica la cantidad de intervalos que se quieren crear, o bien una lista que define los límites de los intervalos.
- **`right`**: Booleano que indica si los intervalos son cerrados por la derecha (**`True`** por defecto).
- **`labels`**: Una lista de etiquetas que se le pueden asignar a cada uno de los intervalos.
- **`include_lowest`**: Booleano que indica si se incluye o no el límite inferior del primer intervalo (`**`False`** por defecto).

LAB 11.02: Generación de categóricos de intervalo

En este Lab se desea generar, paso a paso, un categórico de intervalo.

Las tareas por realizar son:

1. Generar rangos de intervalos estándar, usando **`cut()`**.
2. Generar una tabla de frecuencia absoluta usando **`value_counts()`**.
3. Establecer manualmente los límites de intervalo (**`bins`**).
4. Cambiar intervalos a semi-cerrado a la derecha **`[x,y)`** (**`right`**).
5. Definir las etiquetas de intervalo (**`labels`**).
6. Generación de categórico de intervalo usando UDF's.

```
# Datos base
import pandas as pd
tipos_correctos={
    'id_persona':object,
    'clave_sobrevivencia':object,
    'clase_viaje':object
}
personas_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
personas_titanic_v4.csv'
titanic = pd.read_csv(personas_titanic_csv,
      dtype=tipos_correctos)

titanic['id_persona']=titanic['id_persona'].str.split('.').
      str.get(0)
titanic['clave_sobrevivencia']=titanic['clave_sobrevivencia'].
      str.split('.').str.get(0)
titanic['clase_viaje']=titanic['clase_viaje'].str.split('.').
      str.get(0)
```

Generar rangos de intervalos estándar, usando cut()

En el caso de ejemplo, queremos generar un categórico descriptivo equivalente al campo **`rango_edad`**, tomando como base los valores de la columna **`edad`**, de acuerdo con la siguiente tabla.

Como puede apreciarse, se trata de intervalos semi-cerrados a la derecha.

Categoría	Rango
INFANTES	[0,12)
JÓVENES	[12,21)
ADULTOS	[21,35)
MEDIANA EDAD	[35,45)
ADULTOS MAYORES	[45,∞)

Lo que se terminará haciendo es lo siguiente:

1. Evoluciona el proceso, para entender el uso de la función **`cut()`**.
2. Genera 5 (**`k=5`**) rangos de intervalo (*clases*), a partir de los valores de **`edad`**, en una nueva columna llamada **`rango_edad`**.
3. Genera una tabla de frecuencias para la columna **`rango_edad`**, usando la función **`value_counts()`**.
4. Cambiar de semi-cerrado a la derecha, a semi-cerrado a la izquierda, es decir, de **`(a,b]`** a **`[a,b)`**.
5. Establece valores específicos como límite de intervalo.
6. Establece las etiquetas correctas para las clases.

```
# Generar 5 intervalos, con un número de clases igual a 5.
k=5
titanic['rango_edad']=pd.cut(titanic['edad'],
                             bins=k)

# Ver el resultado, mostrando id_persona, edad y rango_edad
titanic[['id_persona', 'edad', 'rango_edad']]
```

```
      id_persona  edad        rango_edad
0          10000  39.0    (32.1, 48.067]
1          10001   NaN               NaN
```

```
2         10002   NaN               NaN
3         10003  40.0    (32.1, 48.067]
4         10004  49.0  (48.067, 64.033]
...         ...   ...               ...
1307      11307   NaN               NaN
1308      11308   NaN               NaN
1309      11309  24.0    (16.133, 32.1]
[1310 rows x 3 columns]
```

Lo que hizo pandas fue dividir la escala en 5 partes iguales **k=5**, determinando automáticamente el límite inferior y límite superior de cada clase.

Como puedes apreciar, pandas determinó los límites en un formato **(a,b]**, (semi-cerrado a la derecha), que quiere decir que para cada clase, están incluidos los valores mayores al límite inferior, y menores o iguales al límite superior. Pandas analiza el valor de **edad**, y establece el **rango_edad** que le corresponde.

El valor asignado es una etiqueta, pero al no tener más información de esta, se limita a colocar la notación correspondiente al rango de intervalo.

Generar una tabla de frecuencia absoluta usando value_counts()

Podemos usar el método **value_counts()**, que genera una tabla de frecuencias, y que es muy útil para analizar campos categóricos.

```
# Imprime la frecuencia absoluta de cada clase generada
titanic['rango_edad'].value_counts()
```

```
(16.133, 32.1]      525
(32.1, 48.067]      268
(0.0869, 16.133]    134
(48.067, 64.033]    106
(64.033, 80.0]       13

Name: rango_edad, dtype: int64
```

Cambiar intervalos a semi-cerrado a la derecha [a,b) (right).

Ahora, genera los rangos de intervalo, de tal manera que queden en un modo semi-cerrado a la derecha **[a,b)**, es decir, donde la clase contenga los valores mayores o iguales al valor inicial, y menores al valor final.

Muestra la tabla de frecuencias, para validar.

```
# Generar 5 intervalos, con un número de clases igual a 5,
# donde se tenga un modo [x,y), es decir, donde la clase
# contenga los valores mayores o iguales al valor inicial,
# y menores al valor final.
k=5
titanic['rango_edad']=pd.cut(titanic['edad'],
                             bins=k,
                             right=False)

# Imprime la frecuencia absoluta de cada clase generada
titanic['rango_edad'].value_counts()
```

```
[16.133, 32.1)      525
[32.1, 48.067)      268
[0.167, 16.133)     134
[48.067, 64.033)    106
[64.033, 80.08)      13

Name: rango_edad, dtype: int64
```

Establecer manualmente los límites de intervalo (bins)

Ahora, toma el control de los límites de los intervalos, para que se ajusten a las siguientes clases.

Categoría	Rango
INFANTES	[0,12)
JÓVENES	[12,21)
ADULTOS	[21,35)
MEDIANA EDAD	[35,45)
ADULTOS MAYORES	[45,∞)

```
# Determinamos el mímino y el máximo del rango.
# A máximo le sumamos 1, para que, en caso de haber
# edades iguales al máximo (cosa que va a suceder),
# no queden fuera del análisis, debido a que el modo
# que estamos usando excluye el límite superior.
mínimo=titanic['edad'].min()
máximo=titanic['edad'].max()+1
```

```
# Se colocan en una lista los valores de referencia para
# los rangos de valores.
# Ya no es necesario especificar k, porque inferirá el número
# a partir de los valores límite que pongamos en la lista,
# menos 1.
limites=[mínimo,12,21,35,45,máximo]

# Se generan de nuevo los rangos de intervalo.
titanic['rango_edad']=pd.cut(titanic['edad'],
                             bins=limites,
                             right=False)

# Imprime la frecuencia absoluta de cada clase generada
titanic['rango_edad'].value_counts()
```

```
[21.0, 35.0)     453
[45.0, 81.0)     175
[35.0, 45.0)     169
[12.0, 21.0)     158
[0.167, 12.0)     91
Name: rango_edad, dtype: int64
```

Definir las etiquetas de intervalo (labels)

Ahora, ajusta las etiquetas para las clases.

```
# Almacena las etiquetas de las clases en una lista.
etiquetas_clase=['INFANTES','JÓVENES','ADULTOS',
      'MEDIANA EDAD','ADULTOS MAYORES']

mínimo=titanic['edad'].min()
máximo=titanic['edad'].max()+1
limites=[mínimo,12,21,35,45,máximo]

# Se generan de nuevo los rangos de intervalo.
titanic['rango_edad']=pd.cut(titanic['edad'],
                             bins=limites,
                             right=False,
                             labels=etiquetas_clase)

# Imprime la frecuencia absoluta de cada clase generada
titanic['rango_edad'].value_counts()
```

```
ADULTOS            453
ADULTOS MAYORES    175
MEDIANA EDAD       169
JÓVENES            158
INFANTES            91
Name: rango_edad, dtype: int64
```

```
# Comprobamos que se ha creado una columna categórica
# descriptiva, calculada en función a la columna edad
titanic[['nombre','edad','rango_edad']]
```

```
                                  nombre  edad       rango_edad
0                    Andrews, Mr. Thomas Jr  39.0     MEDIANA EDAD
1     Chisholm, Mr. Roderick Robert Crispin   NaN              NaN
2                         Fry, Mr. Richard   NaN              NaN
3                     Harrison, Mr. William  40.0     MEDIANA EDAD
4                   Ismay, Mr. Joseph Bruce  49.0  ADULTOS MAYORES
...                                    ...   ...              ...
1305         Sage, Mrs. John (Annie Bullen)   NaN              NaN
1306                    Storey, Mr. Thomas  60.5  ADULTOS MAYORES
1307                  Baumann, Mr. John D   NaN              NaN
1308           Blackwell, Mr. Stephen Weart   NaN              NaN
1309               Baxter, Mr. Quigg Edmond  24.0          ADULTOS

[1310 rows x 3 columns]
```

Generación de categóricos usando UDF's

Otra alternativa que se tiene es codificar una función... esto es útil cuando la lógica condicional de clasificación tiene excepciones especiales que no se pueden manejar con **`cut()`**.

```
def RangoEdad(x):
  categoria=None
  if (x['edad']>=0.00 and x['edad']<12.00): categoria='INFANTES'
  if (x['edad']>=12.00 and x['edad']<21.00): categoria='JOVENES'
  if (x['edad']>=21.00 and x['edad']<35.00): categoria='ADULTOS'
  if (x['edad']>=35.00 and x['edad']<45.00): categoria='MEDIANA EDAD'
  if (x['edad']>=45.00): categoria='ADULTOS MAYORES'
  return categoria

titanic['rango_edad']=titanic.apply(RangoEdad,axis=1)

titanic[['nombre','edad','rango_edad']]
```

```
                                    nombre  edad       rango_edad
0                    Andrews, Mr. Thomas Jr  39.0     MEDIANA EDAD
1     Chisholm, Mr. Roderick Robert Crispin   NaN             None
2                           Fry, Mr. Richard   NaN             None
3                     Harrison, Mr. William  40.0     MEDIANA EDAD
4                   Ismay, Mr. Joseph Bruce  49.0  ADULTOS MAYORES
...                                    ...   ...              ...
1305         Sage, Mrs. John (Annie Bullen)   NaN             None
1306                      Storey, Mr. Thomas  60.5  ADULTOS MAYORES
1307                    Baumann, Mr. John D   NaN             None
1308           Blackwell, Mr. Stephen Weart   NaN             None
1309                Baxter, Mr. Quigg Edmond  24.0          ADULTOS

[1310 rows x 3 columns]
```

FIN DEL LAB

LAB 11.03: Integración de datos con Python y pandas

En este Lab se utilizarán todas las técnicas aprendidas para realizar una integración de datos. A partir de un conjunto de archivo dispersos, se integrará un master de datos que pueda ser utilizado para trabajos de analítica de datos.

La cámara de bienes raíces del condado King County, en Washington, tiene las operaciones de venta de los años 2022 y 2023. Se desea hacer un trabajo de analítica para comparar la competencia que tienen tres agentes inmobiliarios: GINA JEANNOT, FRANK PAINTER – NEOHOMES, y SKYLINE PROPERTIES.

Dispones del siguiente esquema de datos, que deberás integrar:

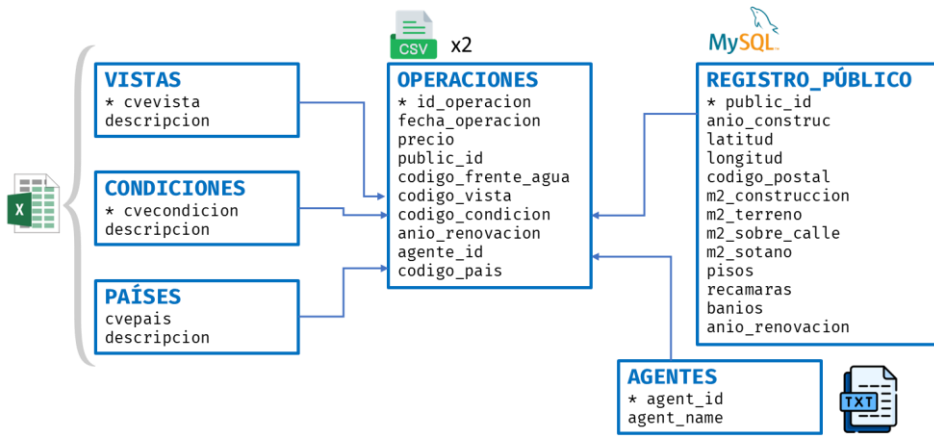


FIGURA 11.01: *Integración de master a partir de múltiples orígenes.*

Las tareas por hace son las siguientes:

1. Documentar la lista de fuentes.
2. Integrar un solo conjunto de datos a partir de dos.
3. Generar categóricos descriptivos para el año, año-mes, y mes de operación.
4. Recuperar los catálogos de diferentes fuentes.

5. Homologación de campos, tipos y nombres.
6. Integrar el master, asociando los catálogos y registro público, con operaciones.
7. Guardar el master en un archivo CSV.

Documentar la lista de fuentes

Las fuentes por utilizar en la integración son las siguientes:

Nombre corto:	**operaciones_2022**
Tipo de fuente:	Archivo CSV
Nombre de archivo:	**operaciones_2022.csv**
Permisos requeridos:	No aplica
Campos por recuperar:	**id_operacion (int)** **fecha_operacion (datetime)** **precio (float)** **public_id (int)** **codigo_frente_agua (int)** **codigo_vista (int)** **codigo_condicion (int)** **agente_id (int)** **codigo_pais (int)**
Filtros aplicables:	Solo incluir las operaciones de los agentes requeridos (**agente_id** = 2, 7, 9).
Generalidades:	Se recupera con encabezados.

Nombre corto:	**operaciones_2023**
Tipo de fuente:	Archivo CSV
Nombre de archivo:	**operaciones_2023.csv**
Permisos requeridos:	No aplica
Campos por recuperar:	**id_operacion (int)** **fecha_operacion (datetime)** **precio (float)** **public_id (int)** **codigo_frente_agua (int)** **codigo_vista (int)** **codigo_condicion (int)** **agente_id (int)** **codigo_pais (int)**
Filtros aplicables:	Solo incluir las operaciones de los agentes requeridos (**agente_id** = 2, 7, 9).
Generalidades:	Se recupera sin encabezados.

Nombre corto:	**vistas**
Tipo de fuente:	Archivo XLSX (Libro Excel)
Nombre de archivo:	**catalogos.xlsx**
Hoja electrónica:	**Categóricos**
Elemento:	**VISTAS** (Rango de celda)
Permisos requeridos:	No aplica
Campos por recuperar:	**cvevista** **descripcion**
Filtros aplicables:	Ninguno
Generalidades:	Ninguno

Nombre corto:	**condiciones**
Tipo de fuente:	Archivo XLSX (Libro Excel)
Nombre de archivo:	**catalogos.xlsx**
Hoja electrónica:	**Categóricos**
Elemento:	**CONDICIONES** (Rango de celda)
Permisos requeridos:	No aplica
Campos por recuperar:	**cvecondicion** **descripcion**
Filtros aplicables:	Ninguno
Generalidades:	Ninguno

Nombre corto:	**países**
Tipo de fuente:	Archivo XLSX (Libro Excel)
Nombre de archivo:	**catalogos.xlsx**
Hoja electrónica:	**Categóricos**
Elemento:	**A18:B28** (Rango de celda)
Permisos requeridos:	No aplica
Campos por recuperar:	**cvepais** **descripcion**
Filtros aplicables:	Ninguno
Generalidades:	Ninguno

Nombre corto:	**agentes**
Tipo de fuente:	Archivo TXT
Nombre de archivo:	**agentes_inmobiliarios.txt**
Permisos requeridos:	No aplica
Campos por recuperar:	**agent_id** **agent_name**
Filtros aplicables:	Ninguno
Generalidades:	El **agent_id** se encuentra de la columna 1 a la 10. El **agent_name** se encuentra de la columna 11 a la 40.

Nombre corto:	**registro_público**
Tipo de fuente:	Base de datos MySQL
Servidor:	**<nombre de servidor>**
Base de datos:	
Usuario:	
Password:	
Permisos requeridos:	**Basados en usuario**
Consulta:	**registro_publico**
Permisos requeridos:	No aplica
Campos por recuperar:	**public_id** **anio_construc** **latitud** **longitud** **codigo_postal** **m2_construccion** **m2_terreno** **m2_sobre_calle** **m2_sotano** **pisos** **recamaras** **banios** **anio_renovacion**
Filtros aplicables:	Ninguno
Generalidades:	Ninguno

Integrar un solo conjunto de datos a partir de dos

En algunas ocasiones se tienen que integrar datos que tienen la misma estructura, pero tienen datos diferentes por diferencia en la temporalidad, o algún filtro. En nuestro caso, tenemos dos archivos que contienen la misma estructura, pero un tiene datos de 2022, y otro de 2023, en unos DataFrame llamados **op_2022** y **op_2023**, respectivamente.

La secuencia de trabajo es la siguiente:

1. Se importan las librerías que han de ocuparse para el procesamiento de datos (**pandas**) y el trabajo con fechas (**numpy** y **datetime**).
2. Se almacena en variables la liga de acceso a datos RAW en GitHub.
3. Como no se requieren todas las columnas contenidas en los archivos, se enumeran las columnas de interés en una lista.

4. Como queremos importar los datos especificando el tipo de dato que nos interesa que tengan en nuestro DataFrame, se especifican los tipos de dato deseados, en un diccionario.
5. Se cargan los datos usando el método **`read_csv()`** de pandas.
6. Se muestra la lista de campos correspondiente a los datos recuperados.
7. Como solo se requieren datos de ciertos agentes inmobiliarios, se realiza un filtro de filas, dejado solo las operaciones de Gina Jeannot, Frank Painter, y Skyline.
8. Se normalizan los tipos de datos de los datos recuperados, con el fin de que solo se tengan como datos de valor aquellos que ameriten cálculos.
9. Se concatenan los DataFrames, para tener un solo conjunto de datos.

```
# Se importan las librerías requeridas para la
# integración.
from numpy import datetime64
import pandas as pd
import datetime as dt

# Se declaran variables con las ligas de acceso a
# los datos en GitHub.

# Operaciones 2022
url_2022='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
operaciones_2022.csv'

# Operaciones 2023
url_2023='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
operaciones_2023.csv'

# Se define una lista que enumera las columnas
# de interés.
columnas_requeridas=['id_operacion', 'fecha_operacion',
    'precio', 'public_id', 'codigo_frente_agua',
    'codigo_vista', 'codigo_condicion',
    'agente_id', 'codigo_pais'
]
```

```
# Se define un diccionario que enumera los tipos
# de datos esperados para las columnas.
tipos_requeridos={
    'id_operacion':int,
    'fecha_operacion':str,
    'precio':float,
    'public_id':int,
    'codigo_frente_agua':int,
    'codigo_vista':int,
    'codigo_condicion':int,
    'agente_id':int,
    'codigo_pais':int
}

# Se lee el CSV con los datos de 2022, recuperando
# las columnas de interés, con el tipo de dato
# deseado.
op_2022=pd.read_csv(url_2022,
                    usecols=columnas_requeridas,
                    dtype=tipos_requeridos
                    )

# Se muestran los datos y la volumetría de filas.
op_2022.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   id_operacion        21613 non-null  int64
 1   fecha_operacion     21613 non-null  object
 2   precio              21613 non-null  float64
 3   public_id           21613 non-null  int64
 4   codigo_frente_agua  21613 non-null  int64
 5   codigo_vista        21613 non-null  int64
 6   codigo_condicion    21613 non-null  int64
 7   agente_id           21613 non-null  int64
 8   codigo_pais         21613 non-null  int64
dtypes: float64(1), int64(7), object(1)
memory usage: 1.5+ MB
```

```
# Se filtran los datos, para que solamente queden
# las filas de los agentes sujetos a análisis.
op_2022=op_2022[op_2022['agente_id'].isin([2,7,9])]
```

```
# Se muestran los datos y la volumetría de filas.
op_2022.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5226 entries, 3 to 21612
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   id_operacion        5226 non-null   int64  
 1   fecha_operacion     5226 non-null   object 
 2   precio              5226 non-null   float64
 3   public_id           5226 non-null   int64  
 4   codigo_frente_agua  5226 non-null   int64  
 5   codigo_vista        5226 non-null   int64  
 6   codigo_condicion    5226 non-null   int64  
 7   agente_id           5226 non-null   int64  
 8   codigo_pais         5226 non-null   int64  
dtypes: float64(1), int64(7), object(1)
memory usage: 408.3+ KB
```

```
# Se modifican los tipos de datos de las columnas
# para mejorar su tratamiento.

# id_operacion se convierte a cadena (str / object)
op_2022['id_operacion']=op_2022['id_operacion'].astype(str)

# fecha_opercion se convierte a fecha, con formato DD/MM/AAAA.
op_2022['fecha_operacion']=pd.to_datetime(
    op_2022['fecha_operacion'], format='%d/%m/%Y')

# public_id se convierte a cadena (str / object)
op_2022['public_id']=op_2022['public_id'].astype(str)

# codigo_frente_agua se convierte a cadena (str / object)
op_2022['codigo_frente_agua']=op_2022['codigo_frente_agua'].
      astype(str)

# codigo_vista se convierte a cadena (str / object)
op_2022['codigo_vista']=op_2022['codigo_vista'].astype(str)

# codigo_condicion se convierte a cadena (str / object)
op_2022['codigo_condicion']=op_2022['codigo_condicion'].
      astype(str)

# agente_id  se convierte a cadena (str / object)
op_2022['agente_id']=op_2022['agente_id'].astype(str)

# codigo_pais se convierte a cadena (str / object)
op_2022['codigo_pais']=op_2022['codigo_pais'].astype(str)
```

```
# Se muestran los datos y la volumetría de filas.
op_2022.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5226 entries, 3 to 21612
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   id_operacion        5226 non-null   object
 1   fecha_operacion     5226 non-null   datetime64[ns]
 2   precio              5226 non-null   float64
 3   public_id           5226 non-null   object
 4   codigo_frente_agua  5226 non-null   object
 5   codigo_vista        5226 non-null   object
 6   codigo_condicion    5226 non-null   object
 7   agente_id           5226 non-null   object
 8   codigo_pais         5226 non-null   object
dtypes: datetime64[ns](1), float64(1), object(7)
memory usage: 408.3+ KB
```

```
# Se repite el proceso para los datos de 2023.
op_2023=pd.read_csv(url_2023,
                    usecols=columnas_requeridas,
                    dtype=tipos_requeridos
                    )

op_2023=op_2023[op_2023['agente_id'].isin([2,7,9])]
op_2023['id_operacion']=op_2023['id_operacion'].astype(str)
op_2023['fecha_operacion']=pd.to_datetime(
    op_2023['fecha_operacion'], format='%d/%m/%Y')
op_2023['public_id']=op_2023['public_id'].astype(str)
op_2023['codigo_frente_agua']=op_2023['codigo_frente_agua'].
      astype(str)
op_2023['codigo_vista']=op_2023['codigo_vista'].astype(str)
op_2023['codigo_condicion']=op_2023['codigo_condicion'].
      astype(str)
op_2023['agente_id']=op_2023['agente_id'].astype(str)
op_2023['codigo_pais']=op_2023['codigo_pais'].astype(str)

# Se muestran los datos y la volumetría de filas.
op_2023.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1665 entries, 2 to 6934
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   id_operacion        1665 non-null   object
 1   fecha_operacion     1665 non-null   datetime64[ns]
 2   precio              1665 non-null   float64
```

```
 3   public_id           1665 non-null   object
 4   codigo_frente_agua  1665 non-null   object
 5   codigo_vista        1665 non-null   object
 6   codigo_condicion    1665 non-null   object
 7   agente_id           1665 non-null   object
 8   codigo_pais         1665 non-null   object
dtypes: datetime64[ns](1), float64(1), object(7)
memory usage: 130.1+ KB
```

```
# Se muestra el número de filas de cada uno de
# los DataFrames creados.
print(len(op_2022))
print(len(op_2023))

# Se concatenan (unen) las filas de los dos
# DataFrames, generando un nuevo índice,
# omitiendo a los índices originales.
operaciones=pd.concat([op_2022, op_2023],
                      ignore_index=True)

# Se imprime el número de filas de las operaciones
# integradas.
print(len(operaciones))
```

```
5226
1665
6891
```

Generar categóricos descriptivos para el año, año-mes, y mes de operación

A partir de un dato fecha/hora, se generan descriptivos categóricos que serán útiles al momento de hacer analítica, con etiquetas útiles para series de tiempo. Se requiere una llamada **año**, que contendrá el año en formato cadena, **año_mes**, que muestre el año y el número de mes, en formato cadena, y **mes**, que muestre el número de mes, más el nombre completo del mes, en mayúsculas, en formato cadena.

```
# Se generan categóricos descriptivos para el
# año (AAAA), el año con el mes (AAAA-MM), y el
# mes con el nombre, en mayúsculas.
operaciones['año'] = operaciones[
    'fecha_operacion'].dt.strftime('%Y')
```

```
operaciones['año_mes'] = operaciones[
    'fecha_operacion'].dt.strftime('%Y-%m')
operaciones['mes'] = operaciones[
    'fecha_operacion'].dt.strftime('%m-%B').str.upper()

# Se imprime una tabla de frecuencia de cada
# uno de los categóricos generados.
print(operaciones['año'].value_counts(),'\n')
print(operaciones['año_mes'].value_counts(),'\n')
print(operaciones['mes'].value_counts(),'\n')
```

```
2022    3561
2023    3330
Name: año, dtype: int64

2023-04    1040
2023-03     942
2023-02     580
2022-06     554
2022-08     499
2022-07     487
2023-01     464
2022-09     450
2022-10     432
2022-05     414
2022-11     372
2022-12     335
2023-05     304
2022-04      18
Name: año_mes, dtype: int64

04-APRIL        1058
03-MARCH         942
05-MAY           718
02-FEBRUARY      580
06-JUNE          554
08-AUGUST        499
07-JULY          487
01-JANUARY       464
09-SEPTEMBER     450
10-OCTOBER       432
11-NOVEMBER      372
12-DECEMBER      335
Name: mes, dtype: int64
```

Recuperar los catálogos de diferentes fuentes

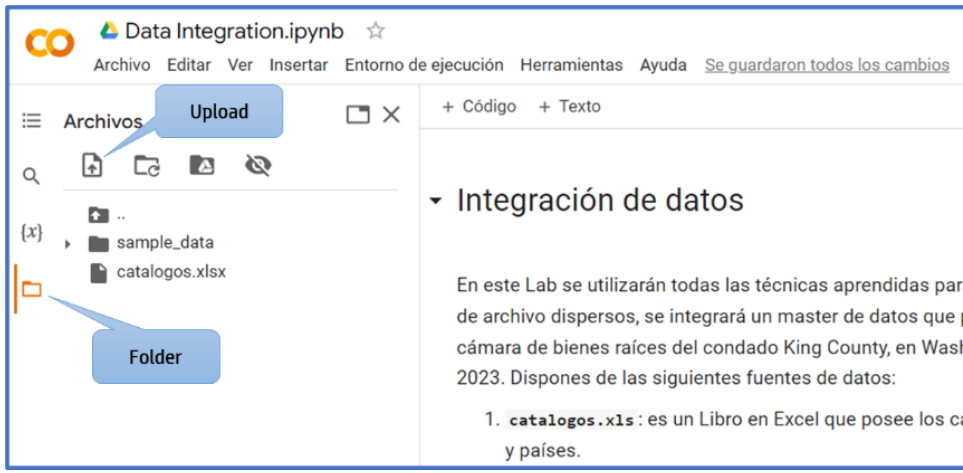
Se recopilan los categóricos descriptivos a partir de diferentes, que será útil al momento de integrar el master.

Para el caso de los catálogos de vistas, condiciones de las viviendas y países, los datos se encuentran en un Libro de Excel llamado **`catalogos.xlsx`**; los catálogos se encuentran en una misma Hoja llamada **`Categóricos`**.

El archivo catalogos.xlsx se debe descargar de la carpeta data del repositorio GitHub que acompaña al libro.

`https://github.com/AprendaPracticando/AnaliticaPythonR1/tree/main/data`

Descarga el archivo del repositorio, y cárgalo en tu ambiente de **Google Colab**. Para hacerlo, debes hacer clic en el ícono de ***Folder*** que aparece en la extrema izquierda de **Google Colab**, y luego, hacer clic en el ícono ***Upload*** (subir archivo) de **Google Colab**.



Las vistas están en el Rango de celdas **`A2:B7`**, las condiciones están en el Rango de celdas **`A10:B15`**, y los países se encuentran en el Rango de celdas **`A18:B28`**. Se leerán usando el método de pandas **`read_excel()`**, y se guarda el resultado en los DataFrame **`vistas`**, **`condiciones`** y **`países`**.

```
# Se recuperan los catálogos desde Excel

# Recuperación desde el Rango de celdas
# A2:B7, en la Hoja Catálogos, para extraer
# las vistas
vistas=pd.read_excel('catalogos.xlsx',
                     sheet_name='Categóricos',
                     usecols='A:B', skiprows=1,
                     nrows=5)

# Muestra lo recuperado.
print(vistas,'\n')

# Recuperación desde el Rango de celdas
# A10:B15, en la Hoja Catálogos, para extraer
# las condiciones.
condiciones=pd.read_excel('catalogos.xlsx',
                     sheet_name='Categóricos',
                     usecols='A:B', skiprows=9,
                     nrows=5)

# Muestra lo recuperado.
print(condiciones,'\n')

# Recuperación desde el Rango de celdas
# A18:B28, en la Hoja Catálogos, para extraer
# los países.
países=pd.read_excel('catalogos.xlsx',
                     sheet_name='Categóricos',
                     usecols='A:B', skiprows=17,
                     nrows=10)

# Muestra lo recuperado.
print(países,'\n')
```

```
   cvevista          descripcion
0         0          E-SIN VISTA
1         1     D-VISTA ESTÁNDAR
2         2      C-VISTA ABIERTA
3         3   B-VISTA PANORÁMICA
4         4  A-VISTA EXCEPCIONAL

   cvecondicion               descripcion
0             1       E-MALAS CONDICIONES
1             2   D-CONDICIONES REGULARES
2             3      C-BUENAS CONDICIONES
3             4      B-BUENAS CONDICIONES
4             5  A-EXCELENTES CONDICIONES

   cvepais     descripcion
```

```
0        1  ESTADOS UNIDOS
1        2          MEXICO
2        3     PUERTO RICO
3        4          CANADA
4        5        ALEMANIA
5        6      INGLATERRA
6        7        COLOMBIA
7        8          PANAMA
8        9           CHINA
9       10           COREA
```

El caso del catálogo de agentes es distinto. Los datos se encuentran en un archivo textual, que en la primera fila contiene el nombre de las columnas, y a partir de la segunda línea son datos.

La primera columna corresponde al id del agente, y la segunda columna es el nombre del agente. El id del agente está desde la posición 1 a la 10, mientras que el nombre del agente está de la posición 11 a la 40.

Para recuperar el contenido se utiliza el método **`read_fwf()`** de pandas, a partir de un archivo llamado agentes_inmobiliarios.txt, que se encuentra en **GitHub** y que puede ser recuperado en formato RAW, a partir de una liga de internet (*URL*).

```
# Liga para recuperar el archivo TXT en modo RAW, desde GitHub
url_txt='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
agentes_inmobiliarios.txt'

# Se recupera la información del archivo, dividiendo los campos
# terminando el primero en la columna 10, y el segundo en la
# columna 40, infiriendo si existe o no encabezados.
# Encontrará encabezados porque la primera fila contiene sólo
# etiquetas, pero de la fila 2 en adelante, se encuentran
# números en la primera columna.
# Se usa read_fwf() (fixed width formatted)
agentes=pd.read_fwf(url_txt,
                    widths=[10,40],
                    header='infer')

# Se muestra lo recuperado
print(agentes)
```

```
    agent_id               agent_name
0          1               WINDERMERE
1          2             GINA JEANNOT
2          3        MELANIE ANTONUCCI
3          4    CARRIE FOLEY- COMPASS
4          5             RACHEL ADLER
5          6       TEAMUP AT COLDWELL
6          7       SKYLINE PROPERTIES
7          8                  FLYHOMES
8          9  FRANK PAINTER - NEOHOMES
9         10           CHAMPMAN HOMES
10        11                    OWNER
```

Para recuperar los datos del registro público, se debe acceder a una base de datos MySQL. Para poder acceder a los datos, es necesario que instale el conector a base de datos MySQL de Python, ejecutando la siguiente línea.

```
!pip install mysql-connector-python
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/
colab-wheels/public/simple/
Collecting mysql-connector-python
  Downloading mysql_connector_python-8.0.32-cp39-cp39-manylinux1_x86_64.
whl  (23.5 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 23.5/23.5 MB 53.3 MB/s eta 0:00:00
Requirement already satisfied: protobuf<=3.20.3,>=3.11.0 in
/usr/local/lib/python3.9/dist-packages (from mysql-connector-python) (3.20.3)
Installing collected packages: mysql-connector-python
Successfully installed mysql-connector-python-8.0.32
```

Instalado el paquete, debes asegurarte de que dispones de una base de datos MySQL que contenga los datos. Escapa al alcance del libro decirte lo que tienes que hacer para que los datos estén disponibles. Puedes hacer lo siguiente, para disponer de la base de datos requerida:

1. Instalar tu instancia de MySQL.
2. Crear una base de datos vacía.
3. Ejecutar el siguiente estatuto SQL, para crear la tabla **`registro_publico`**:

```
CREATE TABLE registro_publico(
    public_id CHAR(8),
    anio_construc CHAR(4),
    latitud FLOAT,
    longitud FLOAT,
    codigo_postal CHAR(5),
    m2_construccion FLOAT,
    m2_terreno FLOAT,
    m2_sobre_calle FLOAT,
    m2_sotano FLOAT,
    pisos INT,
    recamaras INT,
    banios FLOAT,
    anio_renovacion CHAR(4),
    PRIMARY KEY (public_id)
)
```

4. Descargar el archivo **`registro_publico.csv`**, que está en el folder data del repositorio de datos en GitHub que acompaña al libro.
5. Importar los datos del CSV, para que se carguen en la tabla que has creado en MySQL.
6. Obtener la información de servidor, usuario y password que te permitan acceder a los datos desde tu programa.

Se debe importar el conector de datos a MySQL, establecer conexión con la base de datos proporcionando las credenciales de acceso que apliquen, y crear el DataFrame llamado registro_publico, donde se carguen los datos que se encuentren en la tabla registro_publico que se encuentra en la base de datos. Es importante aclarar que el DataFrame y la tabla no tienen por qué llamarse igual, aunque en este caso sí lo hacen.

```
import mysql.connector

# Establecer la conexión con la base de datos
mydb = mysql.connector.connect(
  host="MYSQL5043.site4now.net",
  user="9c7dd4_king",
  password="P@ssw0rd",
  database="db_9c7dd4_king"
)
```

```
# Definir la consulta SQL
query = "SELECT * FROM registro_publico;"

# Leer los datos desde MySQL y cargarlos en un DataFrame
registro_publico = pd.read_sql(query, con=mydb, index_col=None)

mydb.close()

registro_publico.drop(index=0)
```

Homologación de campos, tipos y nombres

Se homologan los campos, tipos y nombres, para poder establecer las relaciones entre datos. Este trabajo es laborioso, pero esencial para que todo funcione de maravilla.

```
# El campo de coincidencia en operaciones se llama
# codigo_vistas, mientras que en vista se llama cvevista.
# El campo de coincidencia en operaciones es de
# tipo object, y en vistas es int64.
# El campo descriptivo en vista se llama descripción
# al igual que en otros catálogos.

# Se cambian los nombres de columna de vista, a) Para
# homologación con operaciones, y b) Difereciarlo de otras
# descripciones en otros catálogos.
vistas.rename(columns={'cvevista':'codigo_vista',
                       'descripcion':'vista'}, inplace=True)

# Se cambia el tipo de dato en vista, para homologarlo en
# operaciones.
vistas['codigo_vista']=vistas['codigo_vista'].astype(str)

# Se ve el resultado
vistas.dtypes
```

```
código_vista    object
vista           object
dtype: object
```

```
# El campo de coincidencia en operaciones se llama
# codigo_condicion, mientras que en condiciones se llama
# cvecondicion.
# El campo de coincidencia en operaciones es de
# tipo object, y en condiciones es int64.
# El campo descriptivo en condiciones se llama descripción
# al igual que en otros catálogos.

# Se cambian los nombres de columna de condiciones, a) Para
# homologación con operaciones, y b) Diferenciarlo de otras
# descripciones en otros catálogos.
condiciones.rename(columns={'cvecondicion':'codigo_condicion',
            'descripcion':'condición'}, inplace=True)

# Se cambia el tipo de dato en vista, para homologarlo en
# operaciones.
condiciones['codigo_condicion']=
      condiciones['codigo_condicion'].astype(str)

# Se ve el resultado
condiciones.dtypes
```

```
código_condición    object
condición           object
dtype: object
```

```
# El campo de coincidencia en operaciones se llama
# codigo_pais, mientras que en países se llama
# cvepais.
# El campo de coincidencia en operaciones es de
# tipo object, y en países es int64.
# El campo descriptivo en países se llama descripción
# al igual que en otros catálogos.

# Se cambian los nombres de columna de países, a) Para
# homologación con operaciones, y b) Diferenciarlo de otras
# descripciones en otros catálogos.
países.rename(columns={'cvepais':'codigo_pais',
                       'descripcion':'país'}, inplace=True)

# Se cambia el tipo de dato en vista, para homologarlo en
# operaciones.
países['codigo_pais']=países['codigo_pais'].astype(str)

# Se ve el resultado
países.dtypes
```

```
código_país    object
país           object
dtype: object
```

```
# El campo de coincidencia en operaciones se llama
# agente_id, mientras que en agentes se llama
# agent_id.
# El campo de coincidencia en operaciones es de
# tipo object, y en agentes es int64.

# Se cambian los nombres de columna de países, a) Para
# homologación con operaciones.
agentes.rename(columns={'agent_id':'agente_id',
                        'agent_name':'agente'}, inplace=True)

# Se cambia el tipo de dato en vista, para homologarlo en
# operaciones.
agentes['agente_id']=agentes['agente_id'].astype(str)

# Se ve el resultado
agentes.dtypes
```

```
agente_id    object
agente       object
dtype: object
```

Integrar el master, asociando los catálogos y registro público, con operaciones

Habiendo homologado los nombres de columna y sus tipos, podemos integrar el master sin problemas. Escogemos la tabla que constituye el centro de los datos. En este caso, se trata de operaciones, porque es tabla débil en todos los casos, y todas las tablas del modelo fluyen hacia dicha tabla.

```
# Asociando Vistas
operaciones=operaciones.merge(vistas,
                  on='codigo_vista',
                  how='inner')
```

```
# Asociando Condiciones
operaciones=operaciones.merge(condiciones,
                  on='codigo_condicion',
                  how='inner')

# Asociando País
operaciones=operaciones.merge(países,
                  on='codigo_pais',
                  how='inner')

# Asociando Agentes
operaciones=operaciones.merge(agentes,
                  on='agente_id',
                  how='inner')

# Asociando Registro público
operaciones=operaciones.merge(registro_publico,
                  on='public_id',
                  how='inner')

# Ver el resultado
operaciones.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6891 entries, 0 to 6890
Data columns (total 28 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   id_operación        6891 non-null   object
 1   fecha_operación     6891 non-null   datetime64[ns]
 2   precio              6891 non-null   float64
 3   public_id           6891 non-null   object
 4   código_frente_agua  6891 non-null   object
 5   código_vista        6891 non-null   object
 6   código_condición    6891 non-null   object
 7   agente_id           6891 non-null   object
 8   código_país         6891 non-null   object
 9   año                 6891 non-null   object
 10  año_mes             6891 non-null   object
 11  mes                 6891 non-null   object
 12  vista               6891 non-null   object
 13  condición           6891 non-null   object
 14  país                6891 non-null   object
 15  agente              6891 non-null   object
 16  anio_construc       6891 non-null   object
 17  latitud             6891 non-null   float64
 18  longitud            6891 non-null   float64
 19  codigo_postal       6891 non-null   object
 20  m2_construccion     6891 non-null   float64
 21  m2_terreno          6891 non-null   float64
 22  m2_sobre_calle      6891 non-null   float64
 23  m2_sotano           6891 non-null   float64
 24  pisos               6891 non-null   int64
```

```
 25  recamaras           6891 non-null   int64
 26  banios              6891 non-null   float64
 27  anio_renovacion     6891 non-null   object
dtypes: datetime64[ns](1), float64(8), int64(2), object(17)
memory usage: 1.5+ MB
```

Guardar el master en un archivo CSV

Ya que se tiene todo el master integrado, es buena idea generar un archivo CSV a partir del mismo, con la finalidad de disponer de un único origen de datos sobre el cual trabajar, sin depender de la conectividad con las múltiples fuentes. Si no nos podemos conectar con la base de datos, o si no podemos enlazarnos a **Github**, no hay problema, porque al tener el master podemos hacer trabajos de analítica a partir del archivo generado.

```
# Se guarda el DataFrame integrado, y se guarda en un
# archivo CSV, con encabezados, sin incluir el índice
# como columna.
operaciones.to_csv('operaciones_master.csv',
                   header=True, index=False)

# Revisar en Google Colab la integración del archivo.
```

Las variantes de una integración pueden ser muchas. Algunas requieren más trabajo de homologación y limpieza, pero a grandes rasgos es lo mismo.

FIN DEL LAB

CAPÍTULO 12:

Tratamiento de datos ausentes (missing data)

Identificación de datos ausentes

Los ***datos ausentes*** o ***missing***, son datos *a)* que son requeridos, *b)* que no están disponibles, y *c)* que no son derivados o calculados.

Los ***datos derivados***, también llamados ***datos calculados***, son aquellos datos que pueden inferirse a partir de otros datos disponibles, es decir, que tienen requeridos indirectos.

Causas de datos ausentes

Los datos pueden ser requeridos por varias causas:

1. Porque son datos requeridos, al ser necesarios para el objetivo del análisis o las hipótesis.
2. Por integridad referencial.
3. Por pertenecer a un conjunto de campos que deben existir juntos.

Los datos pueden ser requeridos de forma ***condicional***, es decir, no basta con que los datos estén vacíos, sino que pueden depender de la presencia o el valor de otro campo para ser exigibles. Por ejemplo, un campo que

almacena el folio de licencia de manejar puede depender de otro campo llamado edad, porque la licencia solo pueden tramitarla los mayores de 16 años.

También pueden ser requeridos en conjunto, por ejemplo, si un producto es importado, deberá tener pedimento de importación, aduana de entrada, y país de origen, en conjunto.

Tipos de datos ausentes

Es importante identificar el tipo de dato ausente que estamos encontrando, porque cada tipo tiene su forma más recomendable de ser tratado.

Los ***datos faltantes*** (***Y***) son los datos que no tenemos, y los ***datos observados*** (***X***) son el resto de los datos que se tienen en una observación. Si en nuestro conjunto de datos tenemos las columnas `id`, `nombre`, `apellidos`, `sexo`, `religión`, `peso`, `altura` y `edad`, y si en una observación nos falta el dato `edad`, entonces `edad` será el dato faltante, y todos los demás serán datos observados.

Lo que determina al tipo de dato ausente es:

1. La aleatoriedad con la que se presentan los datos faltantes, en relación con los datos observados.
2. A su independencia, es decir, si el dato faltante debe su ausencia a sí mismo, y no a los datos observados.
3. A su dependencia, es decir, si el dato faltante debe su ausencia a los datos observados, y no a sí mismo.

Paul Rubin en su artículo "*Multiple Imputation for Nonresponse in Surveys*" publicado en 1987 en el *Journal of the American Statistical Association*, acuñó la tipología de datos ausentes que hoy en día se sigue utilizando. Según Rubin, los datos ausentes pueden ser MCAR, MAR y MNAR.

MCAR (Missing Completely at Random)

MCAR* (*Missing Completely at Random*)**: ***Datos completamente aleatorios. Se trata del tipo de datos ausentes en los cuales la probabilidad de que un valor falte es igual para todos los registros, y no depende ni de la variable faltante, ni de las variables observadas.

Imagina que aplicas una encuesta, y derivado de ella, tienes un conjunto de datos con tres columnas: `Campo 1`, `Campo 2` y `Campo 3`.

Las columnas `Campo 1` y `Campo 2` no presentan faltantes, pero `Campo 3` sí.

Tenemos la oportunidad de volver a aplicar la encuesta, y finalmente logramos recuperar todos los datos de `Campo 3`. En ese escenario, ¿cuándo podríamos decir que los datos ausentes eran de tipo MCAR?

Si los datos faltantes no tienen relación con algún dato observado, o alguna combinación de ellos, se asume que cualquier observación hubiera tenido la misma probabilidad de haber presentado faltantes.

Primer levantamiento de datos			
Campo 1	Campo 2	Campo 3	Campo 3
A	1	-	1
A	2	2	2
A	2	3	3
A	2	2	2
A	3	3	3
A	4	-	2
B	2	2	2
B	2	-	1
B	3	-	3
C	1	3	3
C	1	-	3
C	2	2	2

En este caso, no importa si `Campo 1` es **A**, **B** o **C**, o si `Campo 2` tiene algún valor específico, o si el valor es alto o bajo. Es más, los faltantes ni siquiera dependen del faltante mismo, pues faltan valores bajos, pares, nones. No hay un patrón.

Un ejemplo de datos MCAR podría ser una encuesta en la que algunos encuestados no respondieron a una pregunta particular por error o por cualquier otra razón.

Si la probabilidad de que un encuestado omita la respuesta es la misma para todos los encuestados, entonces los datos faltantes se consideran MCAR. En este caso, no hay ninguna relación entre los valores faltantes y los valores observados, lo que significa que no hay un sesgo sistemático en los datos faltantes.

Los datos ausentes MCAR ocurren cuando la probabilidad de que falte un dato es completamente aleatoria y no depende de ninguna variable medida o no medida. Estos datos faltantes son los más fáciles de detectar, ya que se puede hacer una prueba de patrón faltante para verificar si los datos faltantes están aleatorizados en todo el conjunto de datos.

Este tipo de datos ausentes es el que menos afecta a un trabajo de analítica, pues no produce sesgos, al no estar asociado realmente a nada.

MAR (Missing at Random)

MAR* (*Missing at Random*)**: ***Datos ausentes aleatorios. Se trata de datos ausentes que faltan de forma aleatoria pero cuya probabilidad de ausencia depende de otras variables observadas en el conjunto de datos, y no del faltante mismo.

Volviendo a nuestro ejemplo: si los datos faltantes presentan un patrón con respecto a un dato observado, o a una combinación de ellos, se asume que hay sesgo que afecta la probabilidad de que los datos falten.

Primer levantamiento de datos			
Campo 1	Campo 2	Campo 3	Campo 3
A	1	1	1
A	2	2	2
A	2	3	3
A	2	2	2
A	3	3	3
A	4	-	2
B	2	2	2
B	2	1	1
B	3	-	3
C	1	3	3
C	1	3	3
C	2	-	2

En este caso, el valor más alto de **`Campo 2`**, para cada categoría representada por **`Campo 1`** presenta faltantes. No importa si los faltantes son altos o bajos, pares o nones, y así.

Un ejemplo real: un estudio de Johnson y Pennell (2009) encontró que las mujeres eran más reacias que los hombres a responder preguntas sobre sus ingresos y salarios en una encuesta telefónica. En ese caso, los datos faltantes en el dato **`ingresos`** son más probables de ocurrir cuando el dato **`sexo`** sea mujer.

Si la probabilidad de que un encuestado omita una respuesta dependiendo de otra u otras respuestas que no tienen que ver con el dato faltante en sí, entonces los datos faltantes se consideran MAR. En el caso de las mujeres respondiendo la encuesta telefónica, hay relación entre los valores faltantes y los valores observados, lo que significa que puede haber un sesgo sistemático en los datos faltantes.

Estos son el tipo de datos ausentes más frecuentes.

Los datos ausentes MAR ocurren cuando la probabilidad de que falte un dato es aleatoria, pero depende de otras variables medidas. En otras palabras, la probabilidad de que falte un dato puede ser explicada por las variables conocidas en el conjunto de datos. En estos casos, se pueden usar

técnicas estadísticas como la imputación múltiple para estimar los valores faltantes.

MNAR (Missing Not at Random)

MNAR **(*Missing Not at Random*)**: ***Datos ausentes no aleatorios***. Se trata del tipo de datos ausentes en los cuales la probabilidad de que falte un valor depende de la variable que se está estudiando y del valor que falta. En otras palabras, la probabilidad de que falte un valor no es independiente de los valores que se están midiendo.

Volviendo a nuestro ejemplo: si los datos faltantes presentan un patrón con respecto al dato medido, y no a otro dato observado, o a una combinación de ellos, se asume que hay sesgo que afecta la probabilidad de que los datos falten, que tiene que ver con la variable misma que presenta los faltantes.

Primer levantamiento de datos			
Campo 1	Campo 2	Campo 3	Campo 3
A	1	1	1
A	2	-	2
A	2	3	3
A	2	2	2
A	3	3	3
A	4	-	2
B	2	2	2
B	2	1	1
B	3	3	3
C	1	3	3
C	1	3	3
C	2	-	2

En este caso, cuando el valor de `Campo 3` es 2, se incrementan las probabilidades de que presenta faltantes. No importa qué valores se tengan en `Campo 1` o `Campo 2`, o combinaciones de estos.

Un ejemplo real: un estudio publicado en la revista *Sex Roles* en 1993, encontró que las mujeres mayores eran más propensas a negarse a responder preguntas sobre su edad en una encuesta sobre hábitos de lectura, en

comparación con mujeres más jóvenes. Esto quiere decir que cuando el valor en el campo edad es mayor, es más probable que tengamos datos faltantes, que tiene que ver con lo mismo que se está preguntando, y no con otras variables observadas.

Los datos ausentes MNAR ocurren cuando la probabilidad de que falte un dato está relacionada con la variable que se está midiendo. En estos casos, la razón de los datos faltantes es desconocida y no puede ser explicada por las variables conocidas. Por lo tanto, son los más difíciles de detectar y pueden llevar a sesgos en el análisis de datos si no se abordan adecuadamente.

En este tipo de datos ausentes, los datos ausentes terminan significando algo con respecto al dato mismo.

Estrategias para tratar datos ausentes

El tratamiento de datos ausentes admite las siguientes estrategias:

- ***Usar solo observaciones completas.*** En este caso, se eliminan todas aquellas observaciones que presenten algún dato ausente.
- ***Filtrar o eliminar observaciones con faltantes específicos.*** En este caso, se eliminan todas aquellas observaciones que presenten algún dato ausente específico.
- ***Método de sustitución.*** Consiste en remplazar los datos faltantes con algún otro valor que nos permita el procesamiento, aceptando el sesgo o margen de error que eso puede provocar. Los remplazos típicamente son:
  - ***Por datos de remplazo estándar.*** Como puede ser el promedio, o un valor determinado.
  - ***Por datos condicionados.*** Son valores que se establecen dependiendo de ciertas condiciones. Por ejemplo, el ingreso mensual de una persona puede asumirse cero, siempre y cuando otros campos nos hagan saber que la persona está desempleada, o que su edad es menor a 10 años.
  - ***Por datos nulos.*** Se remplaza por valores nulos, que ofrecen una serie de funciones especiales, además de generar nulos en las operaciones. Esto es especialmente importante

cuando se tienen campos que admiten 0 como ausencia de magnitud, y donde no es lo mismo que valga cero, a que no se tenga.

- ***Modelos de máxima verosimilitud.*** Son la aplicación de algoritmos que determinan los valores más adecuados, en función a conocimiento general y estadístico, y los valores de otros campos. Por ejemplo, si se tiene la estatura y raza étnica de niños entre 5 y 10 años, podríamos determinar el peso promedio de los infantes, contrastándolo con la información de millones de niños, con cierto nivel de confianza y cierto margen de error.

Imagina que tienes un DataFrame con los siguientes campos:

- **id**: Código que identifica a la operación.
- **tipo**: Tipo de operación.
- **cantidad**: Cantidad de unidades involucradas en la operación.
- **precio**: El precio por unidad.
- **total**: Es la multiplicación de cantidad por precio.

```
# Se importa la librería pandas, para poder generar datos
# aleatorios y aplicar algunas funciones matemáticas.
import numpy as np
# Creación de un DataFrame de trabajo.
fuente={
    "id":["10","20","30","40","50","60","65",
          "70","80","90","100"],
    "cliente":["Guadalupe","Jesús","Sergio","Adrián",
               "Paola",None,None,"Alejandra","Sandra",
               "María","Josué"],
    "sexo":[None,"MASCULINO","MASCULINO","MASCULINO",
            None,"MASCULINO",None,"FEMENINO","FEMENINO",
            "FEMENINO","MASCULINO"],
    "tipo":["A",None,"A","B","B","B","C","C","C","C","C"],
    "cantidad":[1,None,3,2,1,1,1,None,None,None,None],
    "precio":[10,20,15,45,35,None,32,80,15,100,25],
    "total":[10,None,45,90,35,100,32,320,None,200,None]
}
```

```
df=pd.DataFrame(fuente)

df
```

	id	cliente	sexo	tipo	cantidad	precio	total
0	10	Guadalupe	None	A	1.0	10.0	10.0
1	20	Jesús	MASCULINO	None	NaN	20.0	NaN
2	30	Sergio	MASCULINO	A	3.0	15.0	45.0
3	40	Adrián	MASCULINO	B	2.0	45.0	90.0
4	50	Paola	None	B	1.0	35.0	35.0
5	60	None	MASCULINO	B	1.0	NaN	100.0
6	65	None	None	C	1.0	32.0	32.0
7	70	Alejandra	FEMENINO	C	NaN	80.0	320.0
8	80	Sandra	FEMENINO	C	NaN	15.0	NaN
9	90	María	FEMENINO	C	NaN	100.0	200.0
10	100	Josué	MASCULINO	C	NaN	25.0	NaN

Observa cómo se han dejado deliberadamente datos ausentes en múltiples columnas.

Revisión inicial de posibles faltantes

Secuencia sugerida:

- Aplicar la función **`info()`**
- Determinar cuántos registros debe haber (**`RangeIndex`** en encabezado).
- Analizar correspondencia entre total de registros, y no nulos (**`non-null`**).
- Si **`RangeIndex`** y **`non-null`** son iguales, quiere decir que no hay ausentes.
- Los datos ausentes son el resultado de restarle **`non-null`** a **`RangeIndex`**, para cada campo requerido.

```
# Revisión inicial de datos missing
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11 entries, 0 to 10
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   id        11 non-null     object
 1   cliente   9 non-null      object
 2   sexo      8 non-null      object
 3   tipo      10 non-null     object
 4   cantidad  6 non-null      float64
 5   precio    10 non-null     float64
 6   total     8 non-null      float64
dtypes: float64(3), object(4)
memory usage: 744.0+ bytes
```

El total de registros se encuentra en el encabezado (**`RegIndex`**), en este caso, son 11.

En nuestro modelo, tenemos los siguientes datos derivados:

- Sabemos que hay un cálculo: **`total=cantidad*precio`**.
- Por tanto, **`total=cantidad*precio`**.
- Por tanto, **`precio=total/cantidad`**.
- Por tanto, **`cantidad=total/precio`**.
- Por tanto, los faltantes de **`cantidad`**, **`precio`** y **`total`** son drivados.

Las respuestas a los datos ausentes son de diferentes tipos:

- ***Eliminación:*** Se eliminan las observaciones que contienen datos ausentes.
- ***Establecer como nulos:*** Se establecen como nulos los datos ausentes (**`NaN`**).
- ***Sustitución por omisión:*** Se sustituyen por un valor predeterminado por omisión.
- ***Cálculo basado en dependientes:*** Se utilizan los campos dependientes para realizar su cálculo, de acuerdo con las reglas o fórmulas predefinidas.

- ***Sustitución por máxima verosimilitud:*** Se utiliza información de los campos en general, y de reglas de conocimiento específico, para calcular o asignar datos estadística o probabilísticamente más coherentes o comunes.

Eliminación de ausentes

Te puedes dar el lujo de eliminar registros con ausentes, cuando dispones de registros en mayor cantidad a la requerida por la muestra estadística.

Por ejemplo: si la muestra estadística te exige 670 observaciones, y tu banco de datos es de 840 observaciones, de las cuales 50 tienen datos ausentes, te puedes dar el lujo de quitar todos los registros con ausentes, y todavía te quedarías con suficientes observaciones para trabajar.

La eliminación de filas es recomendada cuando después de la eliminación de filas, quedan suficientes como para cumplir con la muestra estadística. En caso de que no tengamos muchas filas, se debe intentar resolver el tema de los ausentes de otra manera.

```
# La función `isnull()` genera una tabla que pone en evidencia
# la existencia de nulos, señalándolos como bool. Si se muestra
# True, es que el dato es ausente o nulo.
df.isnull()
```

```
      id  cliente   sexo   tipo  cantidad  precio  total
0  False    False   True  False     False   False  False
1  False    False  False   True      True   False   True
2  False    False  False  False     False   False  False
3  False    False  False  False     False   False  False
4  False    False   True  False     False   False  False
5  False     True  False  False     False    True  False
6  False     True   True  False     False   False  False
7  False    False  False  False      True   False  False
8  False    False  False  False      True   False   True
9  False    False  False  False      True   False  False
10 False    False  False  False      True   False   True
```

```
# La forma más inflexible de manejo de ausentes, es eliminar
# toda observación que presente ausentes en cualquiera de los
# los campos requeridos.
# Nos podemos dar este lujo, si tenemos muchos más registros
# que los señalados por la muestra estadística.
sin_nulos=df.dropna()

sin_nulos
```

```
   id cliente       sexo tipo  cantidad  precio  total
2  30  Sergio  MASCULINO    A       3.0    15.0   45.0
3  40  Adrián  MASCULINO    B       2.0    45.0   90.0
```

```
# Eliminar observaciones cuando tengan ausentes considerando
# un solo campo
sin_nulos_cantidad = df.dropna(subset=['cantidad'])

sin_nulos_cantidad
```

```
   id    cliente       sexo tipo  cantidad  precio  total
0  10  Guadalupe       None    A       1.0    10.0   10.0
2  30     Sergio  MASCULINO    A       3.0    15.0   45.0
3  40     Adrián  MASCULINO    B       2.0    45.0   90.0
4  50      Paola       None    B       1.0    35.0   35.0
5  60       None  MASCULINO    B       1.0     NaN  100.0
6  65       None       None    C       1.0    32.0   32.0
```

```
# Eliminar observaciones cuando haya datos ausentes en dos
# o más campos
sin_nulos_cantidad_precio = df.dropna(subset=[
      'cantidad','precio'])

sin_nulos_cantidad_precio
```

```
   id    cliente       sexo tipo  cantidad  precio  total
0  10  Guadalupe       None    A       1.0    10.0   10.0
2  30     Sergio  MASCULINO    A       3.0    15.0   45.0
3  40     Adrián  MASCULINO    B       2.0    45.0   90.0
4  50      Paola       None    B       1.0    35.0   35.0
6  65       None       None    C       1.0    32.0   32.0
```

Sustitución por omisión

Esta estrategia consiste en sustituir los faltantes con un valor determinado.

```
# Sustituir los valores omisos en "cliente", colocando
# "No disponible".
# Esta estrategia es útil para los ausentes MCAR
# (Missing Competely At Random)
# En el DataFrame df, localiza todos los registros donde
# sea nulo "cliente", y en ese caso, remplaza la columna
# "cliente", por "No Disponible".
df.loc[pd.isnull(df.cliente), ['cliente']]="No Disponible"

# Se ve el resultado
df
```

	id	cliente	sexo	tipo	cantidad	precio	total
0	10	Guadalupe	None	A	1.0	10.0	10.0
1	20	Jesús	MASCULINO	None	NaN	20.0	NaN
2	30	Sergio	MASCULINO	A	3.0	15.0	45.0
3	40	Adrián	MASCULINO	B	2.0	45.0	90.0
4	50	Paola	None	B	1.0	35.0	35.0
5	60	No Disponible	MASCULINO	B	1.0	NaN	100.0
6	65	No Disponible	None	C	1.0	32.0	32.0
7	70	Alejandra	FEMENINO	C	NaN	80.0	320.0
8	80	Sandra	FEMENINO	C	NaN	15.0	NaN
9	90	María	FEMENINO	C	NaN	100.0	200.0
10	100	Josué	MASCULINO	C	NaN	25.0	NaN

Sustitución de un valor específico

Si queremos modificar un solo valor, en específico, podemos invocarlo conociendo su índice.

```
# En el index 4, nos damos cuenta de que "nombre" es "Paola",
# por lo que podemos asumir que "sexo" es "FEMENINO".
# Afectamos ese registro nada más.
# Esta estrategia es muy útil para los ausentes MAR
# (Missing At Random)
# Se localiza el registro con el índice 4, y se remplaza
# el valor de la columna sexo, con "FEMENINO".
df.at[4,'sexo']="FEMENINO"

df
```

	id	cliente	sexo	tipo	cantidad	precio	total
0	10	Guadalupe	None	A	1.0	10.0	10.0
1	20	Jesús	MASCULINO	None	NaN	20.0	NaN
2	30	Sergio	MASCULINO	A	3.0	15.0	45.0
3	40	Adrián	MASCULINO	B	2.0	45.0	90.0
4	50	Paola	FEMENINO	B	1.0	35.0	35.0
5	60	No Disponible	MASCULINO	B	1.0	NaN	100.0
6	65	No Disponible	None	C	1.0	32.0	32.0
7	70	Alejandra	FEMENINO	C	NaN	80.0	320.0
8	80	Sandra	FEMENINO	C	NaN	15.0	NaN
9	90	María	FEMENINO	C	NaN	100.0	200.0
10	100	Josué	MASCULINO	C	NaN	25.0	NaN

Cálculo basado en requeridos indirectos.

Se omiten ausentes que puedan resolverse usando el cálculo base para la obtención del **`total`** (**`total=cantidad*precio`**)

```
# Se remplazan valores por cálculo.
# Se localizan los registros en los cuales una columna es
# faltante, y se remplaza dicha columna, por la fórmula
# conocida para su determinación.
# Esta estrategia es muy útil para tratar los ausentes de tipo
# MNAR (Missing Not At Random)
df.loc[pd.isnull(df.total), ['total']]=df.cantidad*df.precio
df.loc[pd.isnull(df.cantidad), ['cantidad']]=df.total/df.precio
df.loc[pd.isnull(df.precio), ['precio']]=df.total/df.cantidad

df
```

	id	cliente	sexo	tipo	cantidad	precio	total
0	10	Guadalupe	None	A	1.0	10.0	10.0
1	20	Jesús	MASCULINO	None	NaN	20.0	NaN
2	30	Sergio	MASCULINO	A	3.0	15.0	45.0
3	40	Adrián	MASCULINO	B	2.0	45.0	90.0
4	50	Paola	FEMENINO	B	1.0	35.0	35.0
5	60	No Disponible	MASCULINO	B	1.0	100.0	100.0
6	65	No Disponible	None	C	1.0	32.0	32.0
7	70	Alejandra	FEMENINO	C	4.0	80.0	320.0
8	80	Sandra	FEMENINO	C	NaN	15.0	NaN
9	90	María	FEMENINO	C	2.0	100.0	200.0
10	100	Josué	MASCULINO	C	NaN	25.0	NaN

Cálculo máxima verosimilitud

Se utiliza cuando el cálculo obedece a un algoritmo más complicado que la simple comparación de valores con otras columnas dependientes, a lo que se conoce como ***cálculo de máxima verosimilitud***.

Por ejemplo, imagina que la estadística nos dice que cuando las personas del sexo masculino compran productos de tipo 'A' generalmente compran 1 unidad, y cuando las personas del sexo femenino compran productos del tipo 'C', generalmente compran 2 unidades cuando el precio cuesta 20 o menos, pero compran 1 cuando cuesta más de 20.

Este cálculo ya involucra más lógica condicional, y la mejor alternativa es el uso de una función definida por el usuario.

```
def determinacion_cantidad(x):
  nueva_cantidad=x['cantidad']
  if np.isnan(x['cantidad']):
    if (x['sexo']=='MASCULINO' and x['tipo']=='A'):
      nueva_cantidad=1.0
    if (x['sexo']=='FEMENINO' and x['tipo']=='C'):
      if (x['precio']<=20):
        nueva_cantidad=2.0
      else:
        nueva_cantidad=1.0
  return nueva_cantidad

df['cantidad']=df.apply(determinacion_cantidad,axis=1)

df
```

	id	cliente	sexo	tipo	cantidad	precio	total
0	10	Guadalupe	None	A	1.0	10.0	10.0
1	20	Jesús	MASCULINO	None	NaN	20.0	NaN
2	30	Sergio	MASCULINO	A	3.0	15.0	45.0
3	40	Adrián	MASCULINO	B	2.0	45.0	90.0
4	50	Paola	FEMENINO	B	1.0	35.0	35.0
5	60	No Disponible	MASCULINO	B	1.0	100.0	100.0
6	65	No Disponible	None	C	1.0	32.0	32.0
7	70	Alejandra	FEMENINO	C	4.0	80.0	320.0
8	80	Sandra	FEMENINO	C	2.0	15.0	NaN
9	90	María	FEMENINO	C	2.0	100.0	200.0
10	100	Josué	MASCULINO	C	NaN	25.0	NaN

```
# Se declara la función externa.
def calcula_acompañantes(x):
  return x['cantidad_familiares']+x['cantidad_parientes']

# Se invoca la función, pasando sólo algunas columnas.
titanic['acompañantes']=titanic[['cantidad_familiares',
            'cantidad_parientes']].
            apply(calcula_acompañantes,axis=1)
```

LAB 12.01: Tratamiento de datos ausentes

En este Lab se desea generar, paso a paso, un categórico de intervalo.

Las tareas por realizar son:

1. Definir la estrategia de manejo de atípicos del caso.
2. Revisión de existencia de datos ausentes.
3. Estrategia de eliminación de filas con ausentes. **`value_counts()`**.
4. Establecer manualmente los límites de intervalo (**`bins`**).
5. Cambiar intervalos a semi-cerrado a la derecha **`[x,y)`** (**`right`**).
6. Definir las etiquetas de intervalo (**`labels`**).
7. Generación de categórico de intervalo usando UDF's.

```
# Datos base
import pandas as pd
tipos_correctos={
    'id_persona':object,
    'clave_sobrevivencia':object,
    'clase_viaje':object
}
personas_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
personas_titanic_v5.csv'
titanic = pd.read_csv(personas_titanic_csv,
      dtype=tipos_correctos)

titanic['id_persona']=titanic['id_persona'].str.split('.').
      str.get(0)
titanic['clave_sobrevivencia']=titanic['clave_sobrevivencia'].
      str.split('.').str.get(0)
titanic['clase_viaje']=titanic['clase_viaje'].str.split('.').
      str.get(0)
```

Revisar la existencia de ausentes y la suficiencia de datos

Recordemos el objetivo de análisis:

OBJETIVO DE ANÁLISIS: Considerando la información histórica contenida en la enciclopedia británica, relativa a personas que iban en el TITANIC y la suerte que tuvieron, queremos hacer un análisis exploratorio histórico que nos permita saber cómo afectaron a la sobrevivencia de las personas, aspectos como la clase de pasajero, sexo, rango de edad, y si la persona sola o acompañada.

Para efecto de cubrir el objetivo de análisis y nuestras hipótesis, las columnas que permanecen como requeridas son las siguientes:

- **sobrevivencia**
- **clase**
- **sexo**
- **rango_edad**
- **acompañada**

```
# Revisamos la disponibilidad de los datos, usando info()
# Para comprobar si los campos requeridos cumplen con el mínimo
# requerido por la muestra estadística
titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1310 entries, 0 to 1309
Data columns (total 15 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   id_persona           1310 non-null   object
 1   clave_sobrevivencia  1307 non-null   object
 2   nombre               1310 non-null   object
 3   sexo                 1307 non-null   object
 4   clase_viaje          1307 non-null   object
 5   id_bote              486 non-null    object
 6   id_cuerpo            121 non-null    float64
 7   edad                 1046 non-null   float64
 8   cantidad_parientes   1310 non-null   int64
 9   cantidad_familiares  1310 non-null   int64
 10  acompañantes         1310 non-null   int64
 11  rango_edad           1046 non-null   object
 12  sobrevivencia        1307 non-null   object
 13  clase                1307 non-null   object
 14  acompañada           1310 non-null   object
dtypes: float64(2), int64(3), object(10)
memory usage: 153.6+ KB
```

Podemos concluir que todos los campos requeridos tienen más observaciones que las indicadas en el tamaño de la muestra (511), por lo cual, las operaciones realizadas sobre estos datos pueden tener validez estadística.

Campo	Requeridos	Disponibles	¿Son suficientes?
sobrevivencia	511	1307	✓
clase_viaje	511	1307	✓
sexo	511	1307	✓
rango_edad	511	1046	✓
acompañada	511	1310	✓

Eliminación de filas con datos ausentes

Podemos revisar si eliminar los registros que presenten datos ausentes nos puede servir.

```
# Por comodidad, generamos una lista con los nombres de los
# campos requeridos por el objetivo de análisis.
requeridos=['sobrevivencia','clase_viaje','sexo',
      'rango_edad','acompañada']

# Verificamos cuántas filas tenemos, sin faltantes en
# ningún campo requerido.
sin_ausentes=titanic.dropna(subset=requeridos)

print(len(sin_ausentes))
```

```
1040
```

Antes de eliminar las filas que tengan vacíos o nulos en los campos requeridos, tenemos 1,040 filas, que es muy por encima de la muestra estadística, que es 511.

Podemos afirmar que, en este caso, podemos darnos el lujo de eliminar cualquier fila que tenga un ausente, y no comprometer a la muestra estadística.

Corrección de MAR usando requeridos indirectos y UDF

Si los datos fueran escasos y cada fila fuera importante, podríamos intentar inferir los datos ausentes, a partir de los datos requeridos indirectos.

Pongamos el ejemplo de sobrevivencia: identificamos todos los datos ausentes en **`sobrevivencia`**, que como sabemos, es primordial por tratarse de la variable dependiente.

Primero, revisamos en qué filas tenemos ausentes en **`sobrevivencia`**:

```
# Identifica los ausentes del campo sobrevivencia.
titanic[['nombre','clave_sobrevivencia',
         'sobrevivencia','id_bote',
         'id_cuerpo']][(titanic['sobrevivencia'].isnull())]
```

```
                                nombre clave_sobrevivencia sobrevivencia  \
108                Artagaveytia, Mr. Ramon                NaN           NaN
272   Allison, Mr. Hudson Joshua Creighton                NaN           NaN
1308          Blackwell, Mr. Stephen Weart                NaN           NaN

     id_bote  id_cuerpo
108      NaN       22.0
272      NaN      135.0
1308     NaN        NaN
```

Como podemos ver, hay tres filas con datos ausentes, pero sabemos también que a partir de **`id_bote`** y **`id_cuerpo`** podemos inferir **`sobrevivencia`**:

1. Si **`id_bote`** no es nulo, quiere decir que la persona se subió a un bote salvavidas, y vivió.
2. Si **`id_cuerpo`** no es nulo, quiere decir que le asignaron un número de cuerpo, porque murió.

Elaboramos una función definida por el usuario para hacer el cálculo, y actualizamos **`sobrevivencia`** y **`clave_sobrevivencia`**.

```
# Función que infiere el valor de sobreviencia a partir de
# id_bote o id_cuerpo.
def corrige_sobrevivencia(fila):
  # El valor actual de sobrevivencia se asigna a una variable.
  etiqueta=fila['sobrevivencia']
  # Si id_cuerpo no es nulo, la etiqueta pasa a MURIÓ
  if (not pd.isnull(fila['id_cuerpo'])):
    etiqueta='MURIÓ'
  # Si id_bote no es nulo, la etiqueta pasa a VIVIÓ
  if (not pd.isnull(fila['id_bote'])):
    etiqueta='VIVIÓ'
  # Se retorna la etiqueta.
  return etiqueta

def corrige_clave_sobrevivencia(fila):
  # El valor actual de sobrevivencia se asigna a una variable.
  etiqueta=fila['clave_sobrevivencia']
  # Validamos que la etiqueta no esté vacía o sea nula
  if (not pd.isnull(fila['clave_sobrevivencia'])):
    if (fila['sobrevivencia']=='MURIÓ'):
      etiqueta='0'
    else:
      etiqueta='1'
  # Se retorna la etiqueta.
  return etiqueta

# Se aplica la función definida por el usuario, a cada
# fila del DataSet, actualizando sobrevivencia con el
# valor retornado.
titanic['sobrevivencia']=titanic.
      apply(corrige_sobrevivencia,axis=1)
titanic['clave_sobrevivencia']=titanic.
      apply(corrige_clave_sobrevivencia,axis=1)

# Identifica los ausentes del campo sobrevivencia.
# Sigue habiendo uno, porque no hay información ni en
# cuerpo ni en bote, y no hay otra manera de saber si
# vivió o murió.
# Muestra las columnas nombre, clave_sobrevivencia,
# sobrevivencia, id_bote y id_cuerpo, cuando sobrevivencia
# sea nulo.
titanic[['nombre','clave_sobrevivencia',
         'sobrevivencia','id_bote',
         'id_cuerpo']][(titanic['sobrevivencia'].isnull())]
```

```
# Este dato es MNAR (missing not at random),
# pues sistemáticamente, en todos los casos, si tenemos
# bote o cuerpo, podemos inferir sobrevivencia.
```

```
                          nombre clave_sobrevivencia sobrevivencia id_bote  \
1308  Blackwell, Mr. Stephen Weart                NaN           NaN     NaN

      id_cuerpo
1308        NaN
```

Corrección de MAR por asignación directa

El campo **sexo** es ***MAR*** (***missing at random***), porque podemos inferir su valor, pero no de manera sistemática en todos los casos.

Si analizamos el valor de **nombre**, tenemos amplias posibilidades de poder inferir **sexo**, pero no de manera indubitable ni en todos los casos.

Identifiquemos los ausentes en **sexo**:

```
# Identifica los ausentes del campo sexo.
titanic[['nombre','sexo']][(titanic['sexo'].isnull())]
```

```
                        nombre sexo
148     Dick, Mr. Albert Adrian  NaN
237  Douglas, Mr. Walter Donald  NaN
299                Alexis Evans  NaN
```

Las filas con el índice 148 y 237 tienen información suficiente para concluir que los pasajeros eran hombres.

La fila con el índice 299 no tiene forma de inferir el sexo de la persona, porque no incluye título, y Alexis es un nombre que aplica tanto para hombres como para mujeres.

Se actualizan los registros que se pueden corregir, de manera específica, usando **at[]**.

```
# Se le asigna el valor 'HOMBRE' a la columna
# sexo, a las filas con índice 148 y 237, usando
# at[]
titanic.at[148,'sexo']="HOMBRE"
titanic.at[237,'sexo']="HOMBRE"

# Identifica los ausentes del campo sexo.
titanic[['nombre','sexo']][(titanic['sexo'].isnull())]
```

```
           nombre sexo
299  Alexis Evans  NaN
```

FIN DEL LAB

CAPÍTULO 13:

Tratamiento de datos atípicos (Outliers)

Identificación de datos atípicos (outliers)

Datos atípicos

Los ***datos atípicos*** o ***outliers***, se presentan cuando el valor de alguna característica, sola o en combinación con otra, es claramente diferente al resto.

Por ejemplo, no es raro que una persona humana mida 160 centímetros, pues es una estatura típica, pero medir 220 centímetros de altura, es atípico, porque es muy raro que exista.

De la misma forma, puede haber combinaciones, si decimos que la estatura de 160 centímetros es típica, combinado con una característica de edad, con un valor de 10 años, resulta atípica, porque no es común que niños de 10 años midan 160 centímetros de altura.

Dependiendo del objetivo del análisis, deberá ser la rigurosidad del manejo de datos atípicos.

Los datos atípicos pueden existir por:

- Error de procedimiento al levantar datos.
- Acontecimientos extraordinarios que en realidad ocurren.
- Haber valores dentro de rango, pero combinación única con otra variable.
- Causas inexplicables.

Percentiles y cuartiles

Los ***percentiles*** son las medidas dentro de la escala, bajo las cuales se encuentra un cierto porcentaje de elementos de la muestra.

Decir percentil 5% implica identificar la medida dentro de la escala bajo la cual se encuentran el 5% de las observaciones, tomando como base un valor determinado y ordenado de menor a mayor. Que quede claro: los percentiles no implican la división de la escala. Por ejemplo, si tenemos una muestra de 200 observaciones o filas relacionadas con la calificación obtenida por los alumnos en un examen, y hay un campo que se llama `calificación`, con una escala va del 1 al 100, el percentil 5% no necesariamente es 5; puede ser 42, por ejemplo, si el 5% de las observaciones (10 alumnos) sacaron calificaciones entre 0 y 42.

Aunque los percentiles pueden calcularse para cualquier porcentaje, hay algunos de uso muy difundido, por ejemplo, los percentiles seriados de 10% en 10%, se le llaman ***deciles***, y son muy comunes en la evaluación de parámetros sociales de la población en estudios censales, como puede ser edad o ingreso.

Para el tratamiento estadístico de los datos, los percentiles más importantes son 25%, 50% y 75% nos llevan a dividir la muestra en 4 partes, a las que se llaman ***quartiles*** ***(Q1, Q2, Q3)***.

El ***rango intercuartílico*** ***(RIC)*** es el resultado de restarle a Q3 el valor de Q1.

$$RIC = Q3 - Q1$$

El rango intercuartílico es importante porque representa el rango donde se encuentran aglomerados el 50% de los datos más normales y típicos de la muestra, contiene el promedio y la mediana, y no contiene atípicos.

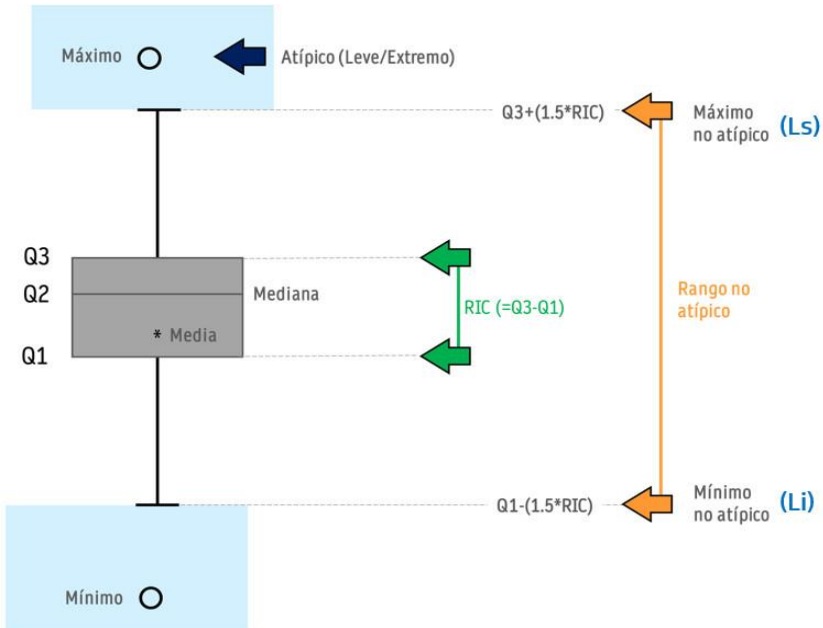


FIGURA 13.01: *Box-Plot.*

Regla de Tukey

Una de las formas más comunes de identificar a los datos atípicos, es calculando el ***mínimo no atípico*** y el ***máximo no atípico***, conocidos también como ***límite superior*** **(*Ls*)** y ***límite inferior*** **(*Li*)**, a lo que se conoce como la ***regla de Tukey***.

El límite inferior (Li) es el resultado de restarle 1.5 veces el rango intercuartílico (RIC) al primer cuartíl $(Q1)$.

$$Li = Q1 - (1.5 * RIC)$$

El límite superior (Ls) es el resultado de restarle 1.5 veces el rango intercuartílico (RIC) al tercer cuartíl ($Q3$).

$$Ls = Q3 + (1.5 * RIC)$$

Cualquier número por debajo del mínimo no atípico (Li), o por encima del máximo no atípico (Ls), es un ***dato atípico***.

En ocasiones incluso entre los datos atípicos hay niveles. Hay ocasiones en que ciertos atípicos son demasiado atípicos; en ese supuesto, se amplía el rango de discriminación. Si en lugar de multiplicar por 1.5 el rango intercuartílico se multiplica por 3.0, estaríamos hablando de que nos preparamos para encontrar un ***dato atípico extremo***.

Un buen informe de analítica de datos debe mencionar, para los datos de valor, si poseen datos atípicos y datos atípicos extremos.

Veamos un ejemplo de cálculo.

Imagina que tenemos un DataFrame con las estaturas y los pesos de un conjunto de personas, y que deseas calcular el mínimo no atípico, el máximo no atípico, ver las observaciones con atípicos y con atípicos extremos.

```
import pandas as pd

# Se genera un DataFrame con datos atípicos y atípicos
# extremos.
datos={
    'estatura_cm':[160,172,180,170,160,169,164,178,174,
         173,166,163,171,180,174,169,160,166,177,175,166,
         165,179,162,173,171,162,161,171,180],
    'peso_kg':[67.80,66.24,88.00,47.20,69.00,65.55,72.96,
         79.56,84.36,69.35,71.28,71.82,81.65,104.83,68.08,
         73.14,53.40,42.05,70.07,67.50,62.70,63.70,81.37,
         62.00,75.19,145.20,65.72,57.95,133.90,76.00]
}

df=pd.DataFrame(datos)
```

```
# Se muestra la información, la forma y el contenido
# del DataFrame.
print(df.info(),'\n')
print(df.shape,'\n')
print(df)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30 entries, 0 to 29
Data columns (total 2 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   estatura_cm  30 non-null     int64
 1   peso_kg      30 non-null     float64
dtypes: float64(1), int64(1)
memory usage: 608.0 bytes
None

(30, 2)

    estatura_cm  peso_kg
0           160    67.80
1           172    66.24
2           180    88.00
3           170    47.20
4           160    69.00
5           169    65.55
6           164    72.96
7           178    79.56
8           174    84.36
9           173    69.35
10          166    71.28
11          163    71.82
12          171    81.65
13          180   104.83
14          174    68.08
15          169    73.14
16          160    53.40
17          166    42.05
18          177    70.07
19          175    67.50
20          166    62.70
21          165    63.70
22          179    81.37
23          162    62.00
24          173    75.19
25          171   145.20
26          162    65.72
27          161    57.95
28          171   133.90
29          180    76.00
```

Para calcular los cuartiles, pandas ofrece la función **`quantile()`**, a la cual se le especifica el cuartil deseado (Q1 es 0.25, Q2 es 0.50, Q3 es 0.75).

Trabajemos con la columna **peso_kg**.

```
# Se determina el primer cuartil (Q1) y el tercer
# cuartil (Q3) para la variable.
Q1=df['peso_kg'].quantile(0.25)
Q3=df['peso_kg'].quantile(0.75)

# Se calcula el rango intercuartílico.
RIC=(Q3-Q1)

# Se calcula el mínimo no atípico y el máximo no atípico, se
# incluyen extremos.
Li=Q1-(1.5*RIC)
Ls=Q3+(1.5*RIC)
LiE=Q1-(3.0*RIC)
LsE=Q3+(3.0*RIC)

# Se muestran los resultados.
print(f"El primer cuatril (Q1) es {Q1:.2f}")
print(f"El tercer cuatril (Q1) es {Q3:.2f}")
print(f"El rango intercuartílico (RIC) es {RIC:.2f}")
print(f"El mínimo no atípico (Li) es {Li:.2f}")
print(f"El máximo no atípico (Ls) es {Ls:.2f}")
print(f"El mínimo no atípico extremo (LiE) es {LiE:.2f}")
print(f"El máximo no atípico extremo (LsE) es {LsE:.2f}")
```

```
El primer cuatril (Q1) es 65.59
El tercer cuatril (Q1) es 78.67
El rango intercuartílico (RIC) es 13.08
El mínimo no atípico (Li) es 45.98
El máximo no atípico (Ls) es 98.29
El mínimo no atípico extremo (LiE) es 26.36
El máximo no atípico extremo (LsE) es 117.90
```

Para mostrar los datos que son atípicos, basta con aplicar una condición en la cual el campo de referencia sea menor al límite inferior, o mayor al límite superior.

```
# Muestra de atípicos
pesos_atipicos=df[
    (df['peso_kg']<Li) |
    (df['peso_kg']>Ls)
    ]

# Se ve el resultado.
print(pesos_atipicos.shape,'\n')
print(pesos_atipicos)
```

```
(4, 2)

    estatura_cm  peso_kg
13          180   104.83
17          166    42.05
25          171   145.20
28          171   133.90
```

Para mostrar los datos que son atípicos extremos, basta con aplicar una condición en la cual el campo de referencia sea menor al límite inferior extremos, o mayor al límite superior extremo.

```
# Muestra de atípicos extremos
pesos_atipicos_extremos=df[
    (df['peso_kg']<LiE) |
    (df['peso_kg']>LsE)
    ]

# Se ve el resultado.
print(pesos_atipicos_extremos.shape,'\n')
print(pesos_atipicos_extremos)
```

```
(2, 2)

    estatura_cm  peso_kg
25          171    145.2
28          171    133.9
```

Finalmente, mostramos los datos de las filas con datos no atípicos respecto al cálculo de referencia.

```
# Muestra de típicos
pesos_tipicos=df[
    (df['peso_kg']>=Li) &
    (df['peso_kg']<=Ls)
    ]

# Se ve el resultado.
print(pesos_tipicos.shape,'\n')
print(pesos_tipicos)
```

```
(26, 2)

    estatura_cm  peso_kg
0           160    67.80
1           172    66.24
2           180    88.00
3           170    47.20
4           160    69.00
5           169    65.55
6           164    72.96
7           178    79.56
8           174    84.36
9           173    69.35
10          166    71.28
11          163    71.82
12          171    81.65
14          174    68.08
15          169    73.14
16          160    53.40
18          177    70.07
19          175    67.50
20          166    62.70
21          165    63.70
22          179    81.37
23          162    62.00
24          173    75.19
26          162    65.72
27          161    57.95
29          180    76.00
```

Estrategias para tratar datos atípicos

Una vez que se identifican los datos atípicos, es necesario indicar qué hacer con ellos.

En un informe serio de analítica de datos, debe indicarse en todos los casos lo que se hará con los atípicos de cada una de las columnas requeridas que sean de valor, incluso si no se piensa hacer nada, incluso si es claro que no hay atípicos. En estudios posteriores, esta información puede ser relevante para los analistas, incluso con otro conjunto de datos distinto.

Las estrategias más comunes para el tratamiento de atípicos son las siguientes.

Dejar como están

En ocasiones, el estudio se refiere precisamente en el trabajo con atípicos, por lo que se identifican y se trabaja con ellos, y no se les transforma.

Por ejemplo, para los científicos que analizan los terremotos, quizá una materia de estudio de especial interés sean los terremotos con magnitud atípica. En ese caso, se les identifica, pero no se les transforma en ningún sentido, ya que si estudio es el objetivo de análisis.

Eliminación de atípicos

La ***eliminación de atípicos*** es una estrategia común, que consiste en eliminar los valores atípicos del conjunto de datos.

Esto se sugiere cuando los atípicos deforman los resultados estadísticos, no son de particular interés, y se tienen muchos datos por encima de la muestra estadística, que quitarlos no afecta demasiado.

Esto se puede hacer utilizando la función **`drop()`** para eliminar las filas que contienen valores atípicos.

Truncamiento de atípicos

El ***truncamiento de atípicos*** es una estrategia que consiste en truncar los valores atípicos a un valor máximo o mínimo.

Esto se sugiere cuando los atípicos deforman los resultados estadísticos, y a precisión de los resultados no es tan importante.

Esto se puede hacer utilizando la función **`clip()`**.

Transformación de atípicos

Se pueden utilizar diversas técnicas de ***transformación de atípicos*** para reducir el impacto de los valores atípicos. Algunas técnicas comunes son la transformación logarítmica, la transformación de raíz cuadrada y la transformación de Box-Cox.

Esto se sugiere cuando los atípicos deforman los resultados estadísticos, y a presición de los resultados no es tan importante.

En esencia, se trata de suavisar o tender a la normalidad los datos, mediante técnicas matemáticas.

Binning de atípicos

La técnica de ***binning de atípicos*** consiste en dividir los datos en intervalos o categorías y luego asignar los valores atípicos a la categoría más cercana. Es equivalente a establecer etiquetas categóricas en función a los diferentes niveles de anormalidad de los datos.

Esto se sugiere cuando el tratamiento numérico de los atípicos no es lo más importante, sino su identificación y visibilidad.

Esto se puede hacer utilizando la función **`cut()`**.

Imputación de atípicos

La técnica de ***imputación de atípicos*** consiste en reemplazar los valores atípicos por valores más razonables o de mayor verosimilitud.

Esto se sugiere cuando los atípicos deforman los resultados estadísticos, y a precisión de los resultados no es tan importante.

Esto se puede hacer utilizando una función definida por el usuario.

LAB 13.01: Tratamiento de datos atípicos

En este Lab se desea desarrollar la estrategia para el tratamiento de datos atípicos.

Imagina que tenemos un DataFrame con las estaturas y los pesos de un conjunto de personas, y que deseas calcular el mínimo no atípico, el máximo no atípico, ver las observaciones con atípicos y con atípicos extremos. Se tienen 4 filas con valores atípicos: 2 son atípicos, y 2 atípicos extremos.

En este Lab se demostrará el uso de diversas técnicas para el manejo de datos atípicos.

Las tareas por realizar son las siguientes:

1. Crear un DataFrame con atípicos y atípicos extremos.
2. Calcular el mínimo no atípico, máximo no atípico, y RIC.
3. Eliminar datos atípicos.
4. Truncar datos atípicos.
5. Suavizar atípicos usando transformación logarítmica.
6. Verificar normalidad usando la prueba Shapiro-Wilk.
7. Verificar normalidad usando histogramas.
8. Manejar atípicos usando la técnica de binning.
9. Manejar atípicos usando técnica de imputación.

Crear un DataFrame con atípicos y atípicos extremos

Genera un DataFrame que deliberadamente tenga datos atípicos.

Se trabajará específicamente con la columna **peso_kg**.

```
import pandas as pd

# Se genera un DataFrame con datos atípicos y atípicos
# extremos.
datos={
    'estatura_cm':[160,172,180,170,160,169,164,178,174,173,
         166,163,171,180,174,169,160,166,177,175,166,165,179,
         162,173,171,162,161,171,180],
    'peso_kg':[67.80,66.24,88.00,47.20,69.00,65.55,72.96,
         79.56,84.36,69.35,71.28,71.82,81.65,104.83,68.08,
         73.14,53.40,42.05,70.07,67.50,62.70,63.70,81.37,
         62.00,75.19,145.20,65.72,57.95,133.90,76.00]
}

df=pd.DataFrame(datos)

print(df)
```

```
    estatura_cm  peso_kg
0           160    67.80
1           172    66.24
2           180    88.00
3           170    47.20
4           160    69.00
5           169    65.55
6           164    72.96
7           178    79.56
8           174    84.36
9           173    69.35
10          166    71.28
11          163    71.82
12          171    81.65
13          180   104.83
14          174    68.08
15          169    73.14
16          160    53.40
17          166    42.05
18          177    70.07
19          175    67.50
20          166    62.70
21          165    63.70
22          179    81.37
23          162    62.00
24          173    75.19
25          171   145.20
26          162    65.72
27          161    57.95
28          171   133.90
29          180    76.00
```

Calcular el mínimo no atípico, máximo no atípico, y RIC

```
# Se calculan el primer y tercer cuartil, el rango
# intercuartílico y los máximos y mínimos no atípicos,
# en su versión regular y extrema.
Q1=df['peso_kg'].quantile(0.25)
Q3=df['peso_kg'].quantile(0.75)

RIC=(Q3-Q1)

Li=Q1-(1.5*RIC)
Ls=Q3+(1.5*RIC)
LiE=Q1-(3.0*RIC)
LsE=Q3+(3.0*RIC)
```

Eliminar datos atípicos

Genera un nuevo DataFrame llamado **`sin_atípicos`**, donde apliques la estrategia de eliminación de atípicos.

El DataFrame **`sin_atípicos`** solo contendrá filas con valores típicos.

Solo habrá filas donde **`peso_kg`** sea mayor o igual al mínimo no atípico, y menor o igual al máximo no atípico.

El resultado debe contener 26 filas.

```
# Se genera un DataFrame que contenga solo las filas donde
# peso tenga valores típicos.
sin_atípicos = df[(df['peso_kg']>=Li) & (df['peso_kg']<=Ls)]

print(sin_atípicos.shape,'\n')
print(sin_atípicos)
```

```
(26, 2)

    estatura_cm  peso_kg
0           160    67.80
1           172    66.24
2           180    88.00
3           170    47.20
4           160    69.00
5           169    65.55
6           164    72.96
7           178    79.56
8           174    84.36
9           173    69.35
```

```
10          166    71.28
11          163    71.82
12          171    81.65
14          174    68.08
15          169    73.14
16          160    53.40
18          177    70.07
19          175    67.50
20          166    62.70
21          165    63.70
22          179    81.37
23          162    62.00
24          173    75.19
26          162    65.72
27          161    57.95
29          180    76.00
```

Genera un nuevo DataFrame llamado **`sin_atípicos_extremos`**, donde apliques la estrategia de eliminación de atípicos.

El DataFrame **`sin_atípicos_extremos`** solo contendrá filas con valores típicos y atípicos que no sean extremos.

Solo habrá filas donde **`peso_kg`** sea mayor o igual al mínimo no atípico extremo, y menor o igual al máximo no atípico extremo.

El resultado debe contener 28 filas.

```
# Se genera un DataFrame que contenga solo las filas donde
# peso tenga valores típicos y atípicos no extremos.
sin_atípicos_extremos = df[(df['peso_kg']>=LiE) &
      (df['peso_kg']<=LsE)]

print(sin_atípicos_extremos.shape,'\n')
print(sin_atípicos_extremos)
```

```
(28, 2)

    estatura_cm  peso_kg
0           160    67.80
1           172    66.24
2           180    88.00
3           170    47.20
4           160    69.00
5           169    65.55
6           164    72.96
7           178    79.56
8           174    84.36
9           173    69.35
10          166    71.28
11          163    71.82
```

```
12          171    81.65
13          180   104.83
14          174    68.08
15          169    73.14
16          160    53.40
17          166    42.05
18          177    70.07
19          175    67.50
20          166    62.70
21          165    63.70
22          179    81.37
23          162    62.00
24          173    75.19
26          162    65.72
27          161    57.95
29          180    76.00
```

Truncar datos atípicos

Genera una nueva columna llamada **`truncado`**, donde apliques la estrategia de truncado de atípicos.

Los valores mayores a 100 se truncarán con un valor de **`peso_kg`** igual a 100.

Los valores menores a 50 se truncarán con un valor de **`peso_kg`** igual a 50.

Revisa los registros donde se haya realizado truncado.

```
# Se realiza el truncado de datos. Lo que esté por abajo
# de 50 se ajusta a 50, y todo lo que esté por encima de
# 100, se ajusta a 100.
df['truncado']=df['peso_kg'].clip(lower=50,upper=100)

# Se muestran las filas que fueron truncadas.
df[['peso_kg','truncado']][(df['peso_kg'] != df['truncado'])]
```

```
    peso_kg  truncado
3     47.20      50.0
13   104.83     100.0
17    42.05      50.0
25   145.20     100.0
28   133.90     100.0
```

Suavizar atípicos usando transformación logarítmica

Genera una nueva columna llamada **`logaritmo`**, donde apliques la estrategia de ***transformación logarítmica de atípicos*** usando logaritmos.

Transforma logarítmicamente el valor de **`peso_kg`** y genera la columna **`logaritmo`** con el valor de la transformación.

Muestra los resultados de la transformación.

```
# Se importa la librería pandas, porque se requiere para
# el trabajo con logaritmos.
import numpy as np

# Se genera un cálculo logarítmico sobre la columna con
# atípicos. El valor de logaritmo es más suavizado, llevando el
valor de peso_kg más cercano a la normalidad.
df['logaritmo']=np.log(df['peso_kg'])

# Se muestran las filas que fueron truncadas.
print(df[['peso_kg','logaritmo']])
```

```
    peso_kg  logaritmo
0     67.80   4.216562
1     66.24   4.193285
2     88.00   4.477337
3     47.20   3.854394
4     69.00   4.234107
5     65.55   4.182813
6     72.96   4.289911
7     79.56   4.376511
8     84.36   4.435093
9     69.35   4.239166
10    71.28   4.266616
11    71.82   4.274163
12    81.65   4.402442
13   104.83   4.652340
14    68.08   4.220683
15    73.14   4.292375
16    53.40   3.977811
17    42.05   3.738859
18    70.07   4.249495
19    67.50   4.212128
20    62.70   4.138361
21    63.70   4.154185
22    81.37   4.399007
23    62.00   4.127134
24    75.19   4.320018
25   145.20   4.978112
…
```

Verificar normalidad usando la prueba Shapiro-Wilk

Utiliza la prueba ***Shapiro-Wilk*** para comprobar que la nueva columna ha suavizado las diferencias entre valores, y es más cercana a una distribución normal.

La prueba de normalidad estadística llamada ***Shapiro-Wilk***, que prueba la hipótesis nula de que los datos provienen de una distribución normal.

Si el valor **p** de la prueba es menor que un nivel de significancia predeterminado (como 0.05), se rechaza la hipótesis nula y se concluye que los datos no siguen una distribución normal.

```
# Se importa la librería scipy, porque se requiere para
# el trabajo con la pruea Shapiro-Wilk.
from scipy.stats import shapiro

# Realizar la prueba de Shapiro-Wilk, sobre peso_kg y logaritmo
# para comprobar que la transformación hizo tender los datos
# hacia la normalidad.
stat, p_peso = shapiro(df['peso_kg'])
stat, p_logaritmo = shapiro(df['logaritmo'])

# Imprimir el valor p y la conclusión
print(f'Valor p para peso_kg: {p_peso:.8f}')
print(f'Valor p para logaritmo: {p_logaritmo:.8f}')
```

```
Valor p para peso_kg: 0.00005744
Valor p para logaritmo: 0.01252854
```

Dados estos resultados, se puede concluir que la variable **`logaritmo`** es más cercana a una distribución normal que la variable **`peso_kg`**.

Esto se debe a que el valor **p** obtenido para la variable **`logaritmo`** es mayor que el nivel de significancia estándar de 0.05, lo que significa que no hay suficiente evidencia para rechazar la hipótesis nula de que los datos de la variable **`logaritmo`** provienen de una distribución normal.

Por otro lado, el valor **p** obtenido para la variable **`peso_kg`** es mucho menor que el nivel de significancia de 0.05, lo que sugiere que la hipótesis nula de normalidad puede ser rechazada en favor de una hipótesis alternativa.

Es importante tener en cuenta que los valores **p** no proporcionan una medida directa de qué tan cerca está una variable de una distribución normal, sino que indican la probabilidad de obtener los datos observados o datos más extremos bajo la hipótesis nula de que los datos provienen de una distribución normal. Por lo tanto, un valor de **p** menor indica que es menos probable que los datos provengan de una distribución normal.

Suavizar atípicos usando raíz cuadrada

También se puede suavizar los datos usando una ***transformación de raíz cuadrada de atípicos***.

```
# Se genera un cálculo de raíz cuadrada sobre la columna
# con atípicos. El valor de logaritmo es más suavizado,
# llevando el valor de peso_kg más cercano a la normalidad.
df['raíz']=np.sqrt(df['peso_kg'])

# Se muestran las filas que fueron truncadas.
print(df[['peso_kg','raíz']])
```

```
    peso_kg       raíz
0     67.80   8.234076
1     66.24   8.138796
2     88.00   9.380832
3     47.20   6.870226
4     69.00   8.306624
5     65.55   8.096295
6     72.96   8.541663
7     79.56   8.919641
8     84.36   9.184770
9     69.35   8.327665
10    71.28   8.442748
11    71.82   8.474668
12    81.65   9.036039
13   104.83  10.238652
14    68.08   8.251061
15    73.14   8.552193
16    53.40   7.307530
17    42.05   6.484597
18    70.07   8.370783
19    67.50   8.215838
20    62.70   7.918333
21    63.70   7.981228
22    81.37   9.020532
23    62.00   7.874008
24    75.19   8.671217
25   145.20  12.049896
…
```

```
# Realizar la prueba de Shapiro-Wilk, sobre peso_kg y raíz
# para comprobar que la transformación hizo tender los datos
# hacia la normalidad.
stat, p_raíz = shapiro(df['raíz'])

# Imprimir el valor p y la conclusión
print(f'Valor p para peso_kg: {p_peso:.8f}')
print(f'Valor p para raíz: {p_raíz:.8f}')
```

```
Valor p para peso_kg: 0.00005744
Valor p para raíz: 0.00099591
```

Qué transformación es más efectiva, ¿la logarítmica, o la raíz cuadrada?

En general, cuanto más grande sea el valor **p** obtenido de la prueba, mayor será la evidencia en contra de la hipótesis alternativa de que los datos no provienen de una distribución normal.

Es importante mencionar que Shapiro-Wilk no es una prueba que revise grados de normalidad. No podemos argumentar, usando solo Shapiro-Wilk, el grado en que es mejor **`logaritmo`** que **`raíz`**.

Lo que sí podemos decir es que los valores de **p** obtenidos para ambas variables (**`raíz`**, **`peso_kg`**) son muy pequeños, lo que sugiere que ambas muestras no provienen de una distribución normal.

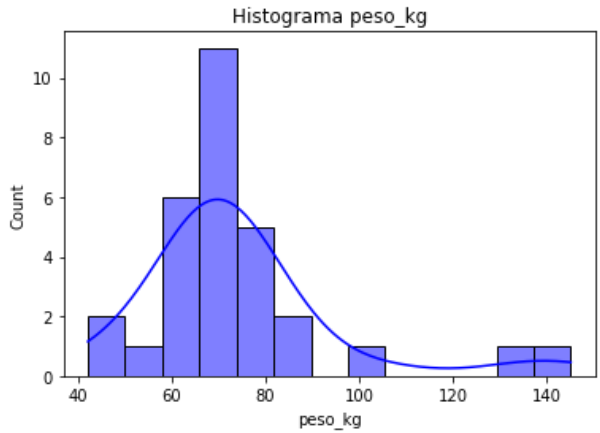
En este caso, sí podemos afirmar que **`logaritmo`** nos es más útil, pero sí los resultados hubieran sido diferentes, es decir, que **`raíz`** también fuera lo suficientemente alto como para suponer que provienen de una distribución normal, no sería claro cuál de las dos transformaciones nos es más útil.

Verificar normalidad usando histogramas

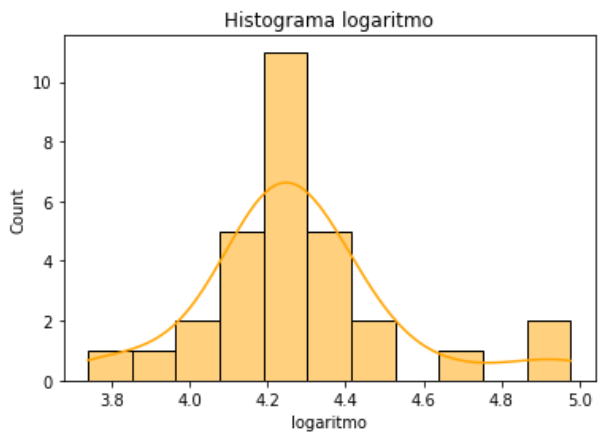
Grafica histogramas de **`peso_kg`**, **`logaritmo`** y **`raíz`**, para observar visualmente cuál tiene más a la curva normal.

Podemos generar una distribución de frecuencias, y revisar visualmente qué gráfica se ajusta más a una distribución normal. Aquí hay que ver las cimas, y lo pronunciado de estas, así como la centralidad.

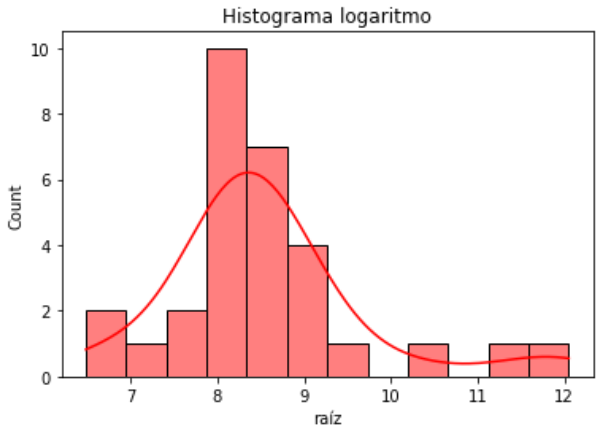
```
# Se importa la librerías par graficación.
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm

# Histograma de peso_kg
sns.histplot(df, x='peso_kg', color='blue',
      alpha=0.5, kde=True)
plt.title('Histograma peso_kg')
plt.show()
```




```
# Histogama de logaritmo
sns.histplot(df, x='logaritmo', color='orange',
      alpha=0.5, kde=True)
plt.title('Histograma logaritmo')
plt.show()
```



```
# Histogama de raíz
sns.histplot(df, x='raíz', color='red',
      alpha=0.5, kde=True)
plt.title('Histograma logaritmo')
plt.show()
```



Podemos comprobar que la columna **`logaritmo`** es la que más se ajusta a la curva normal.

Manejar atípicos usando técnica de *binning*

Genera una nueva columna llamada **`estado`**, donde apliques la estrategia de ***binning*** de atípicos usando **`cut()`**.

Establece los límites de las clases a partir de la información del mínimo, máximo, mínimo no atípico y máximo no atípico.

Muestra el resultado, y muestra una tabla de frecuencias para cada etiqueta del categórico.

```
# Determinamos el mínimo y el máximo.
mínimo=df['peso_kg'].min()
máximo=df['peso_kg'].max()

# Se establecen como límites el mínimo,
# Li, Ls, y el máximo.
limites=[mínimo,Li,Ls,máximo]
```

```
# Se asignan etiquetas para los atípicos arriba y abajo,
# y los típicos.
etiquetas=['ATIPICO BAJO','TÍPICO','ATÍPICO ALTO']

# Se generan las clases de intervalo, considerando los
# límites establecidos, con las etiquetas establecidas,
# e incluyendo el valor menor, para que no se excluye.
df['estado']=pd.cut(df['peso_kg'],
                    bins=limites,
                    labels=etiquetas,
                    include_lowest=True)

print(df[['peso_kg','estado']],'\n')
print(df['estado'].value_counts())
```

```
    peso_kg        estado
0     67.80        TÍPICO
1     66.24        TÍPICO
2     88.00        TÍPICO
3     47.20        TÍPICO
4     69.00        TÍPICO
5     65.55        TÍPICO
6     72.96        TÍPICO
7     79.56        TÍPICO
8     84.36        TÍPICO
9     69.35        TÍPICO
10    71.28        TÍPICO
11    71.82        TÍPICO
12    81.65        TÍPICO
13   104.83  ATÍPICO ALTO
14    68.08        TÍPICO
15    73.14        TÍPICO
16    53.40        TÍPICO
17    42.05  ATIPICO BAJO
18    70.07        TÍPICO
19    67.50        TÍPICO
20    62.70        TÍPICO
21    63.70        TÍPICO
22    81.37        TÍPICO
23    62.00        TÍPICO
24    75.19        TÍPICO
25   145.20  ATÍPICO ALTO
26    65.72        TÍPICO
27    57.95        TÍPICO
28   133.90  ATÍPICO ALTO
29    76.00        TÍPICO

TÍPICO          26
ATÍPICO ALTO     3
ATIPICO BAJO     1
Name: estado, dtype: int64
```

Este programa funciona si hay atípicos arriba y abajo.

Si no hay atípicos, la lógica se debe cambiar, pues el mínimo estará por encima de **`Li`**, y el máximo estaría por debajo de **`Ls`**, lo que produciría errores.

Manejar atípicos usando técnica de imputación

Genera una nueva columna llamada **`valor_imputado`**, donde apliques la estrategia de imputación de atípicos.

Genera la mediana del **`peso_kg`** sin incluir atípicos. Este valor aplicará como valor imputado.

Remplaza los valores atípicos por el valor imputado.

Muestra el resultado.

```
# Se calcula la mediana de peso_kg, sin incluir datos atípicos.
mediana=df.loc[(df['peso_kg']>=Li) &
               (df['peso_kg']<=Ls),'peso_kg'].median()

print("El valor de la mediana (valor imputado) es",
      f"{mediana:.2f}", '\n')

# Se genera una función definida por el usuario, donde se
# retorne el valor imputado, en lugar
def imputación(fila):
  valor=fila['peso_kg']
  if (fila['peso_kg']<Li or fila['peso_kg']>Ls):
    valor=mediana
  return valor

df['valor_imputado']=df.apply(imputación,axis=1)

print(df)
```

```
El valor de la mediana (valor imputado) es 69.17

    estatura_cm  peso_kg  truncado  logaritmo       raíz        estado  \
0           160    67.80     67.80   4.216562   8.234076        TÍPICO
1           172    66.24     66.24   4.193285   8.138796        TÍPICO
2           180    88.00     88.00   4.477337   9.380832        TÍPICO
3           170    47.20     50.00   3.854394   6.870226        TÍPICO
4           160    69.00     69.00   4.234107   8.306624        TÍPICO
5           169    65.55     65.55   4.182813   8.096295        TÍPICO
6           164    72.96     72.96   4.289911   8.541663        TÍPICO
7           178    79.56     79.56   4.376511   8.919641        TÍPICO
8           174    84.36     84.36   4.435093   9.184770        TÍPICO
9           173    69.35     69.35   4.239166   8.327665        TÍPICO
```

```
10          166    71.28     71.28   4.266616   8.442748        TÍPICO
11          163    71.82     71.82   4.274163   8.474668        TÍPICO
12          171    81.65     81.65   4.402442   9.036039        TÍPICO
13          180   104.83    100.00   4.652340  10.238652  ATÍPICO ALTO
14          174    68.08     68.08   4.220683   8.251061        TÍPICO
15          169    73.14     73.14   4.292375   8.552193        TÍPICO
16          160    53.40     53.40   3.977811   7.307530        TÍPICO
17          166    42.05     50.00   3.738859   6.484597  ATIPICO BAJO
18          177    70.07     70.07   4.249495   8.370783        TÍPICO
19          175    67.50     67.50   4.212128   8.215838        TÍPICO
20          166    62.70     62.70   4.138361   7.918333        TÍPICO
21          165    63.70     63.70   4.154185   7.981228        TÍPICO
22          179    81.37     81.37   4.399007   9.020532        TÍPICO
23          162    62.00     62.00   4.127134   7.874008        TÍPICO
24          173    75.19     75.19   4.320018   8.671217        TÍPICO
25          171   145.20    100.00   4.978112  12.049896  ATÍPICO ALTO
26          162    65.72     65.72   4.185403   8.106787        TÍPICO
27          161    57.95     57.95   4.059581   7.612490        TÍPICO
28          171   133.90    100.00   4.897093  11.571517  ATÍPICO ALTO
29          180    76.00     76.00   4.330733   8.717798        TÍPICO

    valor_imputado
0           67.800
1           66.240
2           88.000
3           47.200
4           69.000
5           65.550
6           72.960
7           79.560
8           84.360
9           69.350
10          71.280
11          71.820
12          81.650
13          69.175
14          68.080
15          73.140
16          53.400
17          69.175
18          70.070
19          67.500
20          62.700
21          63.700
22          81.370
23          62.000
24          75.190
25          69.175
26          65.720
27          57.950
28          69.175
29          76.000
```

FIN DEL LAB

LAB 13.02: Tratamiento de datos atípicos y ausentes para el Titanic

En este Lab se desea eliminar las filas con atípicos respecto a la edad de las personas que viajaban en el Titanic.

Las tareas por realizar son:

1. Determinar el mínimo no atípico y máximo no atípico de la columna **edad**.

```
# Datos base
import pandas as pd
tipos_correctos={
    'id_persona':object,
    'clave_sobrevivencia':object,
    'clase_viaje':object
}

personas_titanic_csv='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
personas_titanic_v5.csv'

titanic = pd.read_csv(personas_titanic_csv,
      dtype=tipos_correctos)

titanic['id_persona']=titanic['id_persona'].str.split('.').
      str.get(0)
titanic['clave_sobrevivencia']=titanic['clave_sobrevivencia'].
      str.split('.').str.get(0)
titanic['clase_viaje']=titanic['clase_viaje'].str.split('.').
      str.get(0)

print('Número de filas del conjunto de datos: ', len(titanic))
```

```
Número de filas del conjunto de datos:  1310
```

Tratar los datos ausentes por eliminación

Dado que la muestra estadística es de 511 y tenemos bastante más datos que esos, se decide ejecutar como estrategia de tratamiento de ausentes la eliminación de las filas que los contengan.

```
# Por comodidad, se enumeran en una lista los campos
# requeridos, que son a fin de cuentas los que pueden
# considerarse con datos ausentes.
requeridos=['sobrevivencia','clase_viaje','sexo',
      'rango_edad','acompañada']

# Se eliminan aqullas filas donde alguno de los campos
# requeridos tenga datos ausentes.
print('Filas antes de eliminación de ausentes:',
      len(titanic))

titanic.dropna(subset=requeridos, inplace=True)

print('Filas después de eliminación de ausentes:',
      len(titanic))
```

```
Filas antes de eliminación de ausentes: 1310
Filas después de eliminación de ausentes: 1040
```

```
# Se verifica que se tienen 1040 filas con todos los datos
# requeridos disponibles.
titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1040 entries, 0 to 1306
Data columns (total 15 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   id_persona           1040 non-null   object
 1   clave_sobrevivencia  1040 non-null   object
 2   nombre               1040 non-null   object
 3   sexo                 1040 non-null   object
 4   clase_viaje          1040 non-null   object
 5   id_bote              415 non-null    object
 6   id_cuerpo            117 non-null    float64
 7   edad                 1040 non-null   float64
 8   cantidad_parientes   1040 non-null   int64
 9   cantidad_familiares  1040 non-null   int64
 10  acompañantes         1040 non-null   int64
 11  rango_edad           1040 non-null   object
 12  sobrevivencia        1040 non-null   object
```

```
 13  clase                1040 non-null   object
 14  acompañada           1040 non-null   object
dtypes: float64(2), int64(3), object(10)
memory usage: 130.0+ KB
```

Tratar los datos atípicos por eliminación

Dado que la muestra estadística es de 511 y tenemos bastante más datos que esos, se decide ejecutar como estrategia de tratamiento de datos atípicos la eliminación de las filas que los contengan.

```
# Se calcula el mínimo no atípico (Li) y máximo no atípico (Ls)
# del campo edad.
Q1_edad=titanic['edad'].quantile(0.25)
Q3_edad=titanic['edad'].quantile(0.75)
RIC_edad=(Q3_edad-Q1_edad)
Li_edad=Q1_edad-(1.5*RIC_edad)
Ls_edad=Q3_edad+(1.5*RIC_edad)

print('El mínimo no atípico de la edad es: ', Li_edad)
print('El máximo no atípico de la edad es: ', Ls_edad)
```

```
El mínimo no atípico de la edad es:  -6.0
El máximo no atípico de la edad es:  66.0
```

```
# Eliminación de filas con atípicos.
print('Filas antes de quitar atípicos:', len(titanic))

titanic=titanic[(titanic['edad']>=Li_edad) &
      (titanic['edad']<=Ls_edad)]

print('Filas antes de quitar atípicos:', len(titanic))
```

```
Filas antes de quitar atípicos: 1040
Filas antes de quitar atípicos: 1032
```

Eliminar campos no requeridos

Ya no tiene sentido que mantengamos en el conjunto de datos a los requeridos indirectos, por lo cual, se dejan solo los campos requeridos.

```
# Se actualiza titanic, dejándolo solo con los campos
# requeridos.
titanic=titanic[requeridos]

# Se verifica el resultado, mostrando los campos que
# permanecen.
titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1032 entries, 0 to 1306
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   sobrevivencia  1032 non-null   object
 1   clase_viaje    1032 non-null   object
 2   sexo           1032 non-null   object
 3   rango_edad     1032 non-null   object
 4   acompañada     1032 non-null   object
dtypes: object(5)
memory usage: 48.4+ KB
```

```
# Se muestran los datos que nos permiten hacer trabajos
# de analítica, de acuerdo con el objetivo de análisis
# establecido.
titanic
```

```
     sobrevivencia clase_viaje    sexo       rango_edad  acompañada
0            MURIÓ           1  HOMBRE     MEDIANA EDAD        SOLA
3            MURIÓ           1  HOMBRE     MEDIANA EDAD        SOLA
4            VIVIÓ           1  HOMBRE  ADULTOS MAYORES        SOLA
6            MURIÓ           1  HOMBRE     MEDIANA EDAD        SOLA
7            MURIÓ           1  HOMBRE          ADULTOS        SOLA
...            ...         ...     ...              ...         ...
1288         VIVIÓ           3  HOMBRE          ADULTOS        SOLA
1293         VIVIÓ           3  HOMBRE          ADULTOS        SOLA
1294         MURIÓ           3  HOMBRE          ADULTOS        SOLA
1296         MURIÓ           3  HOMBRE          JOVENES  ACOMPAÑADA
1306         MURIÓ           3  HOMBRE  ADULTOS MAYORES        SOLA

[1032 rows x 5 columns]
```

Guardado de archivo final, en formato CSV y Excel

El conjunto de datos ya está listo para su procesamiento con técnicas de analítica. Se guarda en formato CSV (**`titanic_final.csv`**) y Microsoft Excel (**`titanic_final.xlsx`**).

```
# Se guarda el DataFrame en un archivo CSV, sin incluir índice.
titanic.to_csv('titanic_final.csv', index=False)

# Se guarda el DataFrame en un archivo XLSX, sin incluir
# índice, en una hoja llamada datos.
titanic.to_excel('titanic_final.xlsx',
      index=False, sheet_name='datos')
```

Revisa en **Google Colab**, haciendo clic en el ícono de carpeta que está en la lateral izquierda, si se generaron los archivos.

FIN DEL LAB

CAPÍTULO 14:

Muestreo aleatorio simple y muestreo estratificado

Que todos los datos estén limpios y buen estructurado, no significa que serán utilizados todos. Una cosa es conocer el tamaño de la muestra, otra muy distinta es elegir al azar las observaciones que consideraremos en los trabajos de analítica y la aplicación de técnicas estadísticas.

A la acción de extraer, de un conjunto de datos, el número de observaciones requerido por el tamaño de la muestra se le llama ***muestreo***.

Podemos haber determinado a la perfección el tamaño de la muestra, haber hecho un buen trabajo de limpieza e ingeniería de datos, tratamiento de ausentes y atípicos, pero si el muestreo es realizado de forma deficiente, nuestro trabajo de análisis puede resultar un fracaso.

Hay dos formas básicas de hacer muestreo: muestra aleatoria simple, y muestra estratificada.

Muestra aleatoria simple

Es importante mencionar que se recomienda obtener la muestra a partir de datos que hay han pasado por la limpieza y la ingeniería de datos, tratamiento de ausentes y atípicos. No tiene caso incluir en una muestra observaciones que en el proceso pudieran ser descartados por alguna razón.

El muestreo deberá hacerse después de la fase de *transform* de ETL, pero antes de la fase *load*.

Una ***muestra aleatoria simple*** es un conjunto de observaciones o datos seleccionados de manera aleatoria y sin sesgo de una población, donde cada observación de la población tiene la misma probabilidad de ser seleccionado para formar parte de la muestra.

Asumiendo que tenemos un DataFrame con datos limpios, generar una muestra aleatoria simple se puede lograr utilizando el método **`sample()`** de pandas, que regresa un conjunto de observaciones usando muestreo aleatorio simple.

Supongamos que tenemos un DataFrame con datos limpios, llamado **`df`**, y queremos extraer una muestra aleatoria simple, siendo el tamaño de la muestra de 450 observaciones. Sería así.

```
muestra_aleatoria_simple = df.sample(n=450)
```

El parámetro **`n`** determina el tamaño de la muestra. También puede optarse por hacer el muestreo sobre un porcentaje del número de observaciones, utilizando el parámetro **`frac`**.

Por ejemplo, si deseáramos una muestra compuesta por el 10% de las observaciones, sería así.

```
muestra_aleatoria_simple = df.sample(frac=0.1)
```

Una de las desventajas de la muestra aleatoria simple, es que la muestra no puede ser del todo representativa, porque tiene sesgos.

El ***sesgo*** se presenta cuando al seleccionar observaciones que formarán parte de una muestra, la selección no es realmente aleatoria y existe algún tipo de influencia externa que otorga sobre representación de algunos criterios, y la subrepresentación de otros.

Por ejemplo, si tenemos un universo de datos compuesta por 30% hombres y 70% mujeres, en donde ambos sexos deben estar igualmente representados en la muestra, se esperaría que la muestra incluyera 30% hombres y

70% mujeres, y eso no está garantizado por una muestra aleatoria simple. Si al final la selección de las observaciones para la muestra termina resultando en un 50% de hombres y un 50% de mujeres, los hombres estarán sobre representados, y las mujeres, subrepresentadas.

Existen varios factores que pueden causar sesgo en una muestra aleatoria simple, algunos de los cuales son:

- ***Sesgo de selección:*** puede ocurrir si los individuos se seleccionan de manera no aleatoria, como por ejemplo si se seleccionan solo los individuos más accesibles o convenientes. Esto es muy típico cuando las encuestas se realizan con conocidos, con miembros de un grupo, o personas que tienen teléfono.
- ***Sesgo de respuesta:*** puede ocurrir cuando los individuos seleccionados no responden a la encuesta o cuestionario, lo que puede llevar a que la muestra no represente adecuadamente la población. Los datos ausentes deberán considerarse, en ese caso, como una categoría adicional.
- ***Sesgo de información:*** puede ocurrir si los individuos proporcionan información inexacta, incompleta, o inducida por la pregunta, lo que puede llevar a estimaciones incorrectas.

Muestra estratificada

Una ***muestra estratificada*** es un conjunto de observaciones o datos seleccionados de manera aleatoria dentro de un estrato. Un ***estrato*** es un subgrupo homogéneo de la población que se define en función a una o varias características relevantes.

Por ejemplo, imaginemos que tenemos los datos de una encuesta en donde los participantes responden una pregunta que informa si están laborando actualmente, o no (`laborando`).

Se calcula el tamaño de la muestra, y resulta que es de 768 observaciones.

Si al final del levantamiento de datos tomamos como ***característica de estratificación*** a la variable `laborando`, y si el 38% de los encuestados respondieron que **`SÍ`**, y el 62% respondieron que **`NO`**, entonces el 38% de las 768 observaciones señaladas en el tamaño de la muestra deberán ser **`SÍ`**,

mientras que el 62% deberán ser **NO**; cada respuesta representará un estrato, que deberá estar representado en la misma proporción que presenta la población.

Este método de muestreo se utiliza cuando se desea garantizar que la muestra represente adecuadamente cada uno de los subgrupos que componen la población, especialmente cuando hay diferencias importantes entre los subgrupos y pudieran afectarse los resultados de pruebas estadísticas.

La secuencia para determinar una muestra estratificada es la siguiente:

1. Se determina el tamaño de la muestra.
2. De determina qué característica será considerada la característica de estratificación. Cada valor distinto en esta característica será un estrato.
3. A partir de la población, se determina la frecuencia relativa que le corresponde a cada estrato.
4. Multiplicando la frecuencia relativa del estrato por tamaño de la muestra, se determina la frecuencia absoluta de observaciones que deben representar al estrato, a lo que se conoce como ***tamaño de la muestra estratificado***.
5. Sobre las observaciones de cada estrato, se recupera una muestra aleatoria simple, atendiendo al tamaño de la muestra estratificado.
6. Se juntan las observaciones elegidas aleatoriamente para cada estrato, y conforman la muestra estratificada.

LAB 14.01: Muestra aleatoria simple y muestra estratificada

En este Lab se utilizarán todas las técnicas aprendidas para realizar el muestreo aleatorio simple y el muestreo estratificado, sobre los datos de la cámara de bienes raíces del condado King County, en Washington; se considerarán las operaciones de venta de los años 2022 y 2023 de los agentes inmobiliarios GINA JEANNOT, FRANK PAINTER – NEOHOMES, y SKYLINE PROPERTIES, que ya han sido limpiados y tratados en otros ejercicios. En otras palabras, ya disponemos de los datos limpios, sin atípicos y ausentes.

Las tareas por realizar son:

1. Recuperar los datos previamente limpiados y tratados.
2. Determinar el tamaño de la muestra estadística.
3. Determinar la muestra aleatoria simple.
4. Determinar si la muestra es representativa.
5. Generación de la muestra estratificada.
6. Verificar que hubo mejora en la representatividad de los estratos.

Recuperar los datos previamente limpiados y tratados

Lo primero que hacemos es recuperar los datos limpios y tratados, que se encuentran en **GitHub**.

```
import pandas as pd

# Se almacena en una variable la liga de acceso a los
# datos master que se encuentran en Github
url_master='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
operaciones_master.csv'
```

```
# Se enumeran los tipos de datos correctos para los datos.
tipos_esperados={
  'id_operacion':str,
  'fecha_operacion':object,
  'precio':float,
  'public_id':str,
  'codigo_frente_agua':str,
  'codigo_vista':str,
  'codigo_condicion':str,
  'agente_id':str,
  'codigo_pais':str,
  'año':str,
  'año_mes':str,
  'mes':str,
  'vista':str,
  'condición':str,
  'país':str,
  'agente':str,
  'anio_construc':str,
  'latitud':str,
  'longitud':str,
  'codigo_postal':str,
  'm2_construccion':float,
  'm2_terreno':float,
  'm2_sobre_calle':float,
  'm2_sotano':float,
  'pisos':float,
  'recamaras':float,
  'banios':float,
  'anio_renovacion':str
}

# Se leen los datos y se cargan en un DataFrame
operaciones_master=pd.read_csv(url_master,
      dtype=tipos_esperados)

# La columna fecha_operacion se recuperó como
# object, y se transforma a datetime
operaciones_master['fecha_operacion']=pd.to_datetime(
    operaciones_master['fecha_operacion'])

# Se muestra el resultado
operaciones_master.shape
```

```
(6891, 28)
```

Determinar el tamaño de la muestra estadística

Como podemos comprobar, tenemos 6891 filas, con 28 registros.

Suponiendo que queremos un 99% de confianza con un margen de error del 5%, el tamaño de la muestra se calcularía así.

```
# Declara las variables, cuidando que sean del tipo correcto.
N=len(operaciones_master)
p=0.50
q=1-p
E=0.05
Z=2.576

# Se codifica la fórmula, para calcular el tamaño de la
# muestra (n), y muestra el resultado.
# Toma en cuenta la propiedad conmutativa 'Cuando
# multiplicamos, el orden de los factores no afecta
# al producto'.
n=int((Z**2*p*q*N)/((N*E**2)+(Z**2*p*q)))

print(f'El tamaño de la muestra es {n}')
```

```
El tamaño de la muestra es 605
```

Determinar la muestra aleatoria simple

Entonces, el tamaño de la muestra es 605.

Para generar una muestra aleatoria simple, se utiliza el método **`sample()`**

```
# Se calcula la muestra aleatoria simple, considerando
# un tamaño de la muestra de 605.
muestra_aleatoria_simple=operaciones_master.sample(n=605)
```

Ahora, el DataFrame **`muestra_aleatoria_simple`** contiene 605 observaciones, que representan la muestra aleatoria simple de la población.

Determinar si la muestra es representativa

Vamos a suponer que deseamos que la muestra esté estratificada por el tipo de condición en que se encuentra la propiedad, es decir, por la columna **`condición`**.

Dado que la muestra estratificada pretende reproducir la representatividad que cada estrato tiene, lo primero que debemos determinar es precisamente cuál es la representatividad por reproducir.

Esto lo logramos conociendo la frecuencia relativa que tiene la característica de estratificación en la población. No hay más representatividad más precisa que esa.

```
# Se define cuál es la columna de referencia para
# la estratificación.
columna='condición'

# Se muestra la tabla de frecuencias absolutas y relativas.
original=operaciones_master[columna].value_counts()
for condicion, fi in original.items():
  hi=fi/len(operaciones_master)
  print(f"{condicion:30s} {fi:10d} {hi:0.4%}")
```

```
C-BUENAS CONDICIONES                4460 64.7221%
B-BUENAS CONDICIONES                1844 26.7595%
A-EXCELENTES CONDICIONES             508 7.3719%
D-CONDICIONES REGULARES               67 0.9723%
E-MALAS CONDICIONES                   12 0.1741%
```

La Serie pandas llamada **`original`** contiene las frecuencias absolutas para la característica de estratificación, a partir de la población.

Posteriormente, determinamos las frecuencias absolutas y relativas para la característica de estratificación, a partir de la muestra aleatoria simple que ya hemos generado.

```
muestra=muestra_aleatoria_simple[columna].value_counts()

for condicion, fi in muestra.items():
  hi=fi/len(muestra_aleatoria_simple)
  print(f"{condicion:30s} {fi:10d} {hi:0.4%}")
```

```
C-BUENAS CONDICIONES                387 63.9669%
B-BUENAS CONDICIONES                170 28.0992%
A-EXCELENTES CONDICIONES             42 6.9421%
D-CONDICIONES REGULARES               5 0.8264%
E-MALAS CONDICIONES                   1 0.1653%
```

La Serie pandas llamada **muestra** contiene ahora las frecuencias absolutas para la característica de estratificación, a partir de la muestra.

En este caso, podemos darnos cuenta de que hay discrepancias, pero no son muy significativas; desde luego, habrá casos en donde puede presentarse un mayor problema.

Generación de la muestra estratificada

Si queremos estratificar, de tal manera que la representatividad en una población y en una muestra sean lo más pequeñas posibles, podemos generar muestras aleatorias sobre los estratos, en la proporción que les corresponde a cada uno.

```
from pandas.core.internals import concat
import math

# Se especifica la característica de estratificación.
columna='condición'

# Toma en cuenta que hi, la frecuencia relativa,
# representa la proporción de cada condición
# en la población.
original=operaciones_master[columna].value_counts()
filas=len(operaciones_master)
tamaño_muestra=605

# Se genera un DataFrame con la estructura del DataFrame
# base. Se genera vacío, con la finalidad de irle agregando
# las filas producto del muestreo.
muestra_estratificada=pd.DataFrame(
      columns=operaciones_master.columns)
```

```
# Se lee de forma secuencial la tabla de frecuencias,
# para determinar cuántas observaciones se deben
# tomar de cada estrato.
for categoria, fi in original.items():
  hi=fi/filas
  # Generamos un conjunto de datos que solo contenga
  # las filas del segmento.
  segmento=operaciones_master[(
      operaciones_master[columna]==categoria)]

  # Se genera una muestra aleatoria simple, sobre
  # las filas correspondientes al estrato, usando
  # la frecuencia relativa (hi) como proporción.
  elementos_representativos=math.ceil(tamaño_muestra*hi)
  muestra=segmento.sample(n=elementos_representativos)

  # Se agregan las filas seleccionadas a la muestra
  # estratificada. Se irán acumulando las filas
  # representativas de cada estrato.
  muestra_estratificada=pd.concat(
      [muestra_estratificada,muestra],
      ignore_index=True)

  # Se muestran las filas que cumplen con el criterio,
  # las filas que constituyen la muestra del segmento,
  # las filas que se acumularán a la muestra estratificada,
  # y las filas acumuladas de la muestra estratificada.
  # Aquí se aprecia cómo va creciendo la muestra.
  print(filas, len(segmento), len(muestra),
        len(muestra_estratificada))

# Se muestra la forma final que tiene la muestra estratificada.
print(muestra_estratificada.shape)
```

```
6891 4460 392 392
6891 1844 162 554
6891 508 45 599
6891 67 6 605
6891 12 2 607
(607, 28)
```

Ahora, determinamos la tabla de frecuencia de la muestra estratificada, para ver si la representatividad mejoró.

```
# Se genera la tabla de frecuencias con la muestra
# estratificada, para comprobar que se acerca más
# a la representatividad deseada.
# Es muy poco probable que coincidan exactamente,
# debido a que las muestras aleatorias seleccionan
# diferentes filas cada vez, y el tamaño de observaciones
# a tomar por segmento se redondean.
muestra_est=muestra_estratificada[columna].value_counts()

for condicion, fi in muestra.items():
  hi=fi/len(muestra_aleatoria_simple)
  print(f"{condicion:30s} {fi:10d} {hi:0.4%}")
```

```
C-BUENAS CONDICIONES                392 64.7934%
B-BUENAS CONDICIONES                162 26.7769%
A-EXCELENTES CONDICIONES             45 7.4380%
D-CONDICIONES REGULARES               6 0.9917%
E-MALAS CONDICIONES                   2 0.3306%
```

La Serie pandas llamada **`muestra_est`** contiene ahora las frecuencias absolutas para la característica de estratificación, a partir de la muestra.

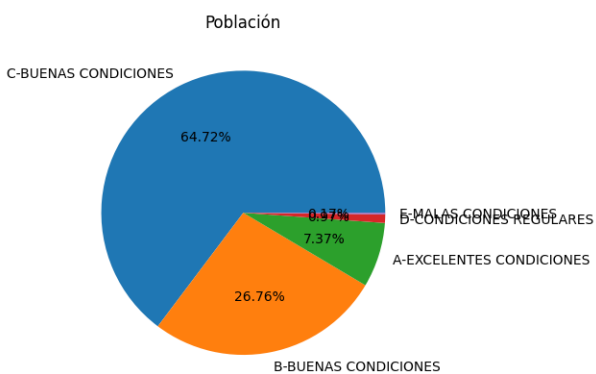
Verificar que hubo mejora en representatividad de los estratos

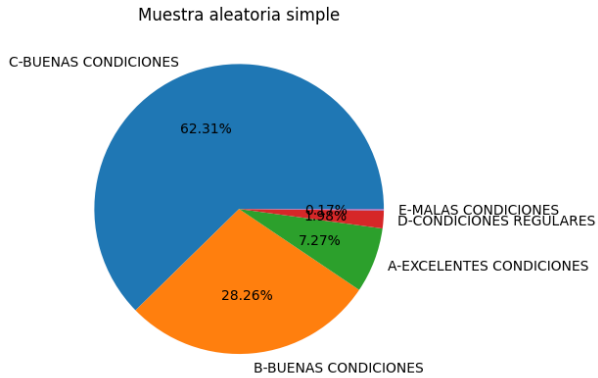
Ahora verificaremos visualmente cuál de las dos muestras, aleatoria simple, o estratificada, representa más acertadamente a los estratos en la población.

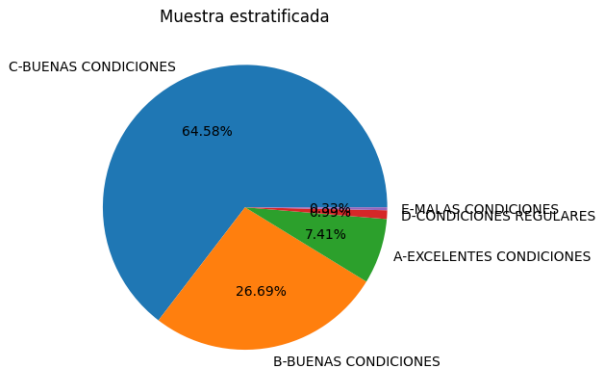
Esta es la distribución de segmentos para la población, usado como característica de estratificación a **`condición`**.

Para esta parte, usaremos la librería **`matplotlib`**. Generaremos 3 gráficas de sectores, a partir de Series pandas que contienen las tablas de frecuencias correspondiente a la característica de estratificación, para la población (**`original`**), para la muestra aleatoria simple (**`muestra`**) y para la muestra estratificada (**`muestra_est`**).

```
# Librería para poder graficar.
import matplotlib.pyplot as plt

# Creamos la gráfica de sectores, tomando la serie
# original, que contiene la tabla de frecuencias de
# la característica de estratificación, a partir de
# la población.
plt.pie(original, labels=original.index, autopct='%1.2f%%')
plt.title('Población')

# Se genera una nueva gráfica.
plt.figure()
# Creamos la gráfica de sectores, tomando la serie
# muestra, que contiene la tabla de frecuencias de
# la característica de estratificación, a partir de
# la muestra aleatoria simple.
plt.pie(muestra, labels=muestra.index, autopct='%1.2f%%')
plt.title('Muestra aleatoria simple')

# Se genera una nueva gráfica.
plt.figure()
# Creamos la gráfica de sectores, tomando la serie
# muestra, que contiene la tabla de frecuencias de
# la característica de estratificación, a partir de
# la muestra aleatoria simple.
plt.pie(muestra_est, labels=muestra_est.index,
      autopct='%1.2f%%')
plt.title('Muestra estratificada')

# Se muestran los gráficos
plt.show()
```







Como puede observarse, la muestra estratificada tiene una representatividad más cercana a la de la población, que la que presenta la muestra aleatoria simple.

FIN DEL LAB

CAPÍTULO 15:

Serialización JSON y Pickle

La ***serialización*** es el proceso de convertir un objeto en una secuencia de bytes o caracteres, para almacenarlo o transmitirlo a la memoria, a una base de datos o a un archivo. Su propósito principal es guardar el estado de un objeto para poder volver a crearlo cuando sea necesario. El proceso inverso se denomina deserialización.

Serialización JSON

JSON (***JavaScript Object Notation*** / ***Notación de Objetos de JavaScript***) es un formato ligero de intercambio de datos. Dentro de sus ventajas es que leerlo y escribirlo es simple para los humanos (*human readable*), mientras que para las máquinas es simple interpretarlo y generarlo.

Está basado en un subconjunto del Lenguaje de Programación JavaScript, Standard ECMA-262 3rd Edition. JSON es un formato de texto que es completamente independiente del lenguaje, pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos.

Para tener acceso a las herramientas para el manejo de JSON, es necesario importar el módulo **`json`**.

```
import json
```

Se tienen estas instrucciones, que nos pueden ayudar a la serialización y deserialización JSON.

Función	Acción
`json.dumps(objeto)`	Permite generar el equivalente JSON de un objeto.
`json.dump(objeto,archivo)`	Permite generar el equivalente JSON de un objeto, para su almacenamiento en un archivo.
`json.load(archivo)`	Se utiliza para leer codificación JSON desde un archivo.
`json.loads(json)`	Permite generar un objeto Python a partir de su equivalente JSON.

Estas son las conversiones que se realizarán, cuando se trabaja con equivalencias JSON y Python.

En JSON	En Python
`object`	`dict`
`array`	`list`
`array`	`tuple`
`string`	`str`
`number (int)`	`int`
`number (real)`	`float`
`true`	`True`
`false`	`False`
`null`	`None`

Para serializar a JSON, lo primero que se tiene que hacer es transformar a formato JSON aquello que se desea serializar, usando **`dumps()`**.

Posteriormente, cuando se desea recuperar el contenido de una serialización, se utiliza **`loads()`**.

Este podría ser un ejemplo.

```
# Se importa el módulo para trabajar con JSON.
import json

# Se crea un objeto de muestra, para serializar.
Original=[
    ['correo','nombre','telefono'],
    ['juan@gmail.com','Juan','8123232323'],
    ['maria@gmail.com','Maria','5545454545'],
    ['diana@homail.com','Diana','4490909090']
]

# Se revisa el tipo del objeto a serializar, y su contenido.
print(">> Tipo del objeto.\n")
print(type(Original))

print("\n>> Contenido del objeto.\n")
print(Original)

# Se hace la serialización a JSON.
print("\n>> Serialización a JSON.\n")
Original_JSON=json.dumps(Original,indent=4)
print(Original_JSON)

# Deserialización
print("\n>> Deserialización desde JSON.\n")
Nueva_Lista=json.loads(Original_JSON)
print(Nueva_Lista)
print(type(Nueva_Lista))

# Se comprueba que no hubo pérdida ni corrupción de datos
print("\n>> Comprobando igualdad de objetos.\n")
print(Original==Nueva_Lista)
```

```
>> Tipo del objeto.

<class 'list'>

>> Contenido del objeto.

[['correo', 'nombre', 'telefono'], ['juan@gmail.com', 'Juan', '8123232323'],
['maria@gmail.com', 'Maria', '5545454545'], ['diana@homail.com', 'Diana',
'4490909090']]
```

```
>> Serialización a JSON.

[
    [
        "correo",
        "nombre",
        "telefono"
    ],
    [
        "juan@gmail.com",
        "Juan",
        "8123232323"
    ],
    [
        "maria@gmail.com",
        "Maria",
        "5545454545"
    ],
    [
        "diana@homail.com",
        "Diana",
        "4490909090"
    ]
]

>> Deserialización desde JSON.

[['correo', 'nombre', 'telefono'], ['juan@gmail.com', 'Juan', '8123232323'],
['maria@gmail.com', 'Maria', '5545454545'], ['diana@homail.com', 'Diana',
'4490909090']]
<class 'list'>

>> Comprobando igualdad de objetos.

True
```

El contenido de un objeto JSON puede pasarse a un archivo JSON, y leerse desde ahí, para darle persistencia.

```
# Grabando el JSON en un archivo.
with open("archivo.json","w+") as f:
    json.dump(Original,f,indent=4)

# Leyendo datos de un archivo json
with open("archivo.json","r") as f:
    recuperados=json.load(f)

print(recuperados)
print("\n>> Comprobando igualdad de objetos.\n")
print(Original==recuperados)
```

```
[['correo', 'nombre', 'telefono'], ['juan@gmail.com', 'Juan', '8123232323'],
['maria@gmail.com', 'Maria', '5545454545'], ['diana@homail.com', 'Diana',
'4490909090']]

>> Comprobando igualdad de objetos.

True
```

Serialización usando Pickle

El módulo **`pickle`** implementa protocolos binarios para la serialización y deserialización de objetos en Python.

Se le llama *Pickling* al proceso a través del cual un objeto de Python es convertido a un flujo de bytes equivalente, y se le llama *Unpickling* a la acción inversa.

Para tener acceso a las herramientas para el manejo de Pickle, es necesario importar el módulo **`pickle`**.

```
import pickle
```

Se tienen estas instrucciones, que nos pueden ayudar a la serialización y deserialización Pickle.

Función	Acción
`pickle.dumps(objeto)`	Permite generar el equivalente pickle (binario) de un objeto.
`pickle.dump(objeto,archivo)`	Permite generar el equivalente pickle de un objeto, para su almacenamiento en un archivo.
`pickle.load(archivo)`	Se utiliza para leer codificación pickle desde un archivo.
`pickle.loads(pickle)`	Permite generar un objeto Python a partir de su equivalente pickle.

Para serializar a Pickle, lo primero que se tiene que hacer es transformar a formato pickle aquello que se desea serializar, usando **`dumps()`**. Posteriormente, cuando se desea recuperar el contenido de una serialización, se utiliza **`loads()`**.

Este podría ser un ejemplo.

```
# Se importa el módulo para trabajar con Pickle.
import pickle

# Serializa a pickle, y observa cómo pickle es un
# formato binario.
print("\n>> Serialización a Pickle.\n")
Original_pickle=pickle.dumps(Original)
print(Original_pickle)

# Deserialización pickle
print("\n>> Deserialización desde Pickle.\n")
Nueva_Lista=pickle.loads(Original_pickle)
print(Nueva_Lista)
print(type(Nueva_Lista))

# Comprobación de la igualdad del origen y destino.
print("\n>> Comprobando igualdad de objetos.\n")
print(Original==Nueva_Lista)
```

```
>> Serialización a Pickle.

b'\x80\x04\x95\xa6\x00\x00\x00\x00\x00\x00\x00]\x94(]\x94(\x8c\x06correo\x94\x8c\x
06nombre\x94\x8c\x08telefono\x94e]\x94(\x8c\x0ejuan@gmail.com\x94\x8c\x04Juan\x94\
x8c\n8123232323\x94e]\x94(\x8c\x0fmaria@gmail.com\x94\x8c\x05Maria\x94\x8c\n554545
4545\x94e]\x94(\x8c\x10diana@homail.com\x94\x8c\x05Diana\x94\x8c\n4490909090\x94ee
.'

>> Deserialización desde Pickle.

[['correo', 'nombre', 'telefono'], ['juan@gmail.com', 'Juan', '8123232323'], ['mar
ia@gmail.com', 'Maria', '5545454545'], ['diana@homail.com', 'Diana', '4490909090']
]
<class 'list'>

>> Comprobando igualdad de objetos.

True
```

El contenido de un objeto pickle puede pasarse a un archivo, y leerse desde ahí, para darle persistencia. Es importante hacer notar que tanto la escritura como la lectura, son especificando contenidos binarios (**b**).

```
# Grabando el pickle en un archivo.
with open("archivo.pickle","wb+") as f:
    pickle.dump(Original,f)

# Leyendo datos de un archivo pickle
with open("archivo.pickle","rb") as f:
    recuperados=pickle.load(f)

# Comprobando la igualdad de origen y destino.
print(recuperados)
print("\n>> Comprobando igualdad de objetos.\n")
print(Original==recuperados)
```

```
[['correo', 'nombre', 'telefono'], ['juan@gmail.com', 'Juan', '8123232323'], ['mar
ia@gmail.com', 'Maria', '5545454545'], ['diana@homail.com', 'Diana', '4490909090']
]

>> Comprobando igualdad de objetos.

True
```

LAB 15.01: Serialización de un DataFrame usando Pickle

En este Lab se utilizarán las técnicas de serialización y deserialización, para comprobar que se puede usar Pickle para el transporte de información sin pérdida o corrupción de contenidos.

Las tareas por realizar son:

1. Recuperar los datos previamente limpiados y tratados.
2. Serializar a pickle y almacenar en un archivo.
3. Leer desde un archivo pickle y deserializar.

Recuperar los datos previamente limpiados y tratados

Lo primero que hacemos es recuperar los datos limpios y tratados, que se encuentran en **GitHub**.

```
import pandas as pd

# Se almacena en una variable la liga de acceso a los
# datos master que se encuentran en Github
url_master='https://raw.githubusercontent.com/
AprendaPracticando/AnaliticaPythonR1/main/data/
operaciones_master.csv'

# Se leen los datos y se cargan en un DataFrame
origen=pd.read_csv(url_master,
      dtype=tipos_esperados)

# Se muestra el resultado
origen.shape
```

```
(6891, 28)
```

Serializar a pickle y almacenar en un archivo

Se serializa a pickle el DataFrame, y se almacena en un archivo de extensión **`.pickle`**

```
# Librería para poder usar pickle
import pickle

# Se abrirá un archivo llamado datos.pickle, en modo
# write (w) binary (b) en donde, si existe lo remplaza
# y si no existe lo crea (+). El archivo tendrá un
# apuntador llamado f.
with open("datos.pickle","wb+") as f:
    # Se serializa usando pickle el contenido del objeto
    # llamado origen, y se guarda en el archivo.
    pickle.dump(origen,f)

# El archivo ya debe existir en el ambiente.
```

Leer desde un archivo pickle y deserializar

Se lee el contenido de un archivo binario de extensión **`.pickle`**, que contiene un objeto serializado usando pickle. Se deserializa, y se obtiene el objeto original.

```
# Se abrirá un archivo llamado datos.pickle, en modo
# read (r) binary (b). El archivo tendrá un
# apuntador llamado f.
with open("datos.pickle","rb") as f:
    destino=pickle.load(f)

# Se comprueba que el nuevo objeto es un DataFrame, con
# el contenido original.
destino.shape
```

```
(6891, 28)
```

FIN DEL LAB

Índice

U

V

W

X

APRENDA EDICIONES

SAN FÉLIX 5432, DESPACHO “D”

VISTA SOL, GUADALUPE, NUEVO LEÓN, MÉXICO

FECHA DE IMPRESIÓN: 21/ABRIL/2023