Unidad 2 - Manipulación de datos

Fundamentos de ciencia de datos

Definición de datos

Los datos son una porción de información de algún tema en particular que se guardan para ser utilizados en futuros análisis. Los datos pueden venir de tres formas, estructurados, no estructurados y semi estructurados. Durante este curso vamos a utilizar mayormente datos estructurados y algunos semi estructurados.

Esta sección resumen y traducción de lo presentado en el curso de "Data Science for Beginners" de Microsoft y Microsoft Learn Unit "Classify your Data". Para más información, referirse a la misma.

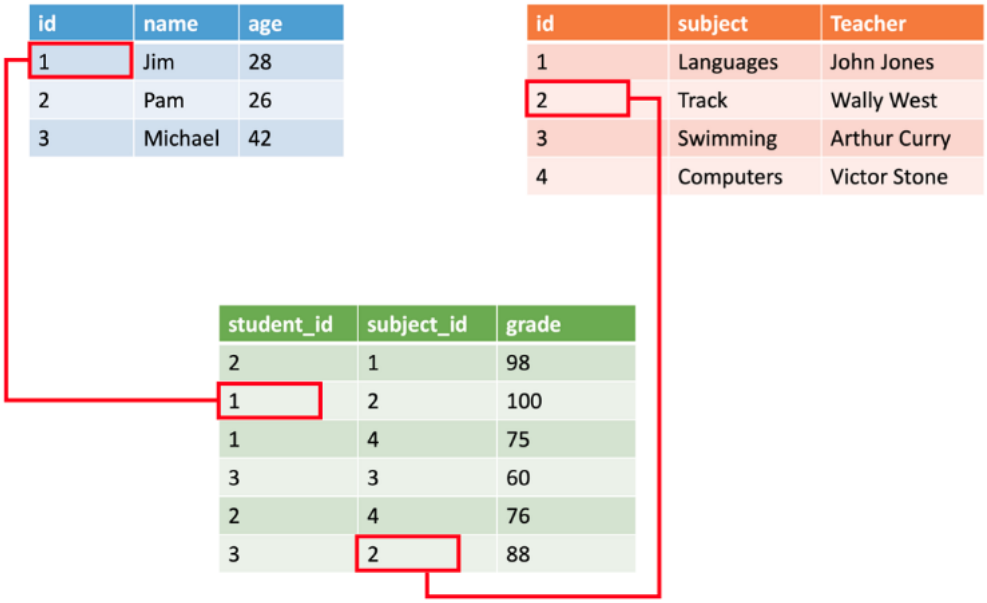
Estructurados

Los datos estructurados, también llamados relacionales, son aquellos en los que los data points comparten los mismos campos o atributos. Generalmente, están organizados en filas y columnas, donde cada fila tiene la misma cantidad de columnas. Las columnas representan valores de algún tipo y tienen un nombre que describe esos valores, mientras que las filas contienen el valor en particular.

La ventaja de los datos estructurados es que se pueden organizar de forma tal de poder relacionarse con otros datos estructurados. Sin embargo, esta estructura rígida no permite hacer cambios a la estructura de forma sencilla, y hace que la evolución de los datos sea más lenta.

En estos casos, los datos tienen la misma organización y forma, el mismo esquema de datos. Al compartir el mismo esquema los datos pueden ser consultados de forma sencilla con lenguajes como SQL (Structured Query Language).

Un ejemplo de datos estructurados puede ser la siguiente base de datos relacional con información de estudiantes:



No estructurados

Los datos no estructurados no pueden representarse con filas y columnas, no contienen un formato o siguen alguna regla. La falta de estructura hace que sea muy fácil añadir información nueva, pero también resultan más dificiles de explorar. Un ejemplo de datos no estructurados pueden ser un archivos de texto, videos, fotos, etc.

Semi-estructurados

Los datos semiestructurados son una combinación de datos estructurados y no estructurados. Generalmente, no pueden organizarse en filas y columnas pero tienen una organización que se considera estructurada. Estos datos utilizan tags para darle organización y jerarquías a los datos. Los datos semiestructurados también se conocen como datos no relacionales.

Los datos semiestructurados se construyen con lenguajes de serialización, por ejemplo: XML, JSON o YAML.

XML

Extensible Markup Language (XML) está basado en texto, lo que hace que sea fácil de leer para humanos y computadoras. El código de abajo representa el registro de una persona:

```
<Person Age="23">
    <FirstName>Quinn</FirstName>
    <LastName>Anderson</LastName>
    <Hobbies>
        <Hobby Type="Sports">Golf</Hobby>
        <Hobby Type="Leisure">Reading</Hobby>
        <Hobby Type="Leisure">Guitar</Hobby>
   </Hobbies>
</Person>
```

XML utiliza tags para darle forma a los datos. Los tags pueden ser elementos como `<First Name>` o atributos como `Age='23'` . Los elementos pueden tener elementos
hijos que permiten expresar relaciones, como `Hobby` dentro del elemento `Hobbies`.

Aplicaciones:

- El formato XML se utiliza para almacenar datos estructurados, como información de clientes, datos de sensores, registros de eventos, entre otros.
- También se utiliza mucho en el intercambiado entre diferentes aplicaciones y sistemas. Muchos servicios Web utilizan el protocolo SOAP que está basado en mensajes XML.
- El formato XML se puede combinar con otras tecnologías, como XSLT (Lenguaje de Transformación de Hojas de Estilo XML) y XPath (Lenguaje de Selección de Ruta XML), para manipular y transformar datos XML en diferentes formatos.

También se combina con XSD (XML Schema Definition), el cual es un lenguaje de esquema utilizado para describir la estructura y las restricciones de los contenidos de los documentos.

Ventajas:

- El formato XML es un lenguaje de marcado ampliamente utilizado y bien documentado, lo que lo hace fácil de implementar con una gran cantidad de librerías disponibles en diversos lenguajes de programación.
- El formato XML es flexible y extensible, lo que lo hace ideal para manejar datos estructurados y complejos.

Desventajas:

- El formato XML puede ser propenso a la redundancia de datos, lo que puede aumentar el tamaño del archivo XML.
- XML puede ser difícil de manejar para archivos grandes, ya que puede requerir más recursos de procesamiento y almacenamiento que otros formatos.
- La sintaxis XML puede tornarse compleja, sobre todo cuando se combina con XSD o XLST, lo que puede dificultar su comprensión y manipulación.

JSON

JSON (Notación de objetos de JavaScript o JavaScript Object Notation) es un formato de archivo liviano que se utiliza para almacenar e intercambiar datos y se basa en el lenguaje de programación JavaScript. Es un formato basado en texto que utiliza una estructura clave-valor para representar datos y soporta el manejo de listas y jerarquías. JSON usa llaves `{}` para indicar la estructura de los datos. Este lenguaje es más fácil de entender para los humanos. Por ejemplo:

```
{
    "firstName": "Quinn",
    "lastName": "Anderson",
    "age": "23",
    "hobbies": [
        { "type": "Sports", "value": "Golf" },
        { "type": "Leisure", "value": "Reading" },
        { "type": "Leisure", "value": "Guitar" }
    ]
}
```

Aplicación:

JSON se usa comúnmente para API web, bases de datos NoSQL y como formato de intercambio de datos entre diferentes aplicaciones. Es el formato más popular de intercambio con servicios web, habiendo desplazado al formato XML en muchos casos.

Ventajas:

- Legible por humanos y fácil de entender.
- Se puede analizar y manipular fácilmente con diferentes lenguajes de programación.
- Admite tipos de datos complejos, como matrices y objetos anidados.

Desventajas:

- Puede ser menos eficiente para almacenar y procesar grandes conjuntos de datos en comparación con otros formatos binarios.
- Soporte limitado para la compresión de datos.

YAML

YAML Ain't Markup Language (YAML) es un lenguaje de serialización más reciente. La estructura de los datos se da por la separación de las líneas y la indentación. De esta forma, se elimina la dependencia de caracteres especiales como los paréntesis, comas, corchetes y llaves.

Este lenguaje es fácil de entender para los humanos y se utiliza generalmente como archivo de configuración.

```
firstName: Quinn
lastName: Anderson
age: 23
hobbies:
    - type: Sports
      value: Golf
    - type: Leisure
      value: Reading
    - type: Leisure
      value: Guitar
```

Como pueden notar en los ejemplos de arriba, añadir nuevos elementos a este tipo de datos resulta más sencillo que en los datos estructurados ya que el nuevo elemento puede añadirse a un solo registro sin necesidad de añadirlo al resto, como debe hacerse con los datos estructurados. Por ejemplo, alguna persona puede no tener hobbies y/o tener cantidad de hermanos.

Aplicaciones:

- El formato YAML es utilizado en configuraciones de software para almacenar datos de configuración y parámetros de sistema.
- También se puede utilizar en aplicaciones de ciencia de datos para almacenar datos estructurados, como datos de sensores, registros de eventos, información de clientes, entre otros.
- El formato YAML se puede utilizar como formato de intercambio de datos entre diferentes lenguajes de programación y plataformas.

Ventajas:

- El formato YAML es fácil de leer y editar por humanos, lo que lo hace conveniente para la configuración y manipulación de datos.
- YAML es un formato de texto plano, lo que lo hace compatible con una amplia variedad de herramientas y lenguajes de programación.
- El formato YAML es fácil de usar para estructurar y organizar datos complejos, lo que lo hace ideal para aplicaciones de ciencia de datos que requieren datos estructurados.
- A diferencia de JSON, permite utilizar comentarios y resulta más compacto en tamaño.

Desventajas:

- El formato YAML no es eficiente para el almacenamiento de grandes cantidades de datos, ya que puede ser propenso a la redundancia de datos.
- YAML puede tener problemas de compatibilidad en diferentes lenguajes de programación y plataformas. Puede que un parseador determinado no soporte todas las versiones de YAML disponibles.
- La falta de estándares para el formato YAML puede generar ambigüedad en la interpretación de los datos.

Datos Tabulares

Los datos utilizados en el análisis de datos están generalmente representados en forma tabular, como una tabla, compuestos por filas y columnas. Para guardar los datos se pueden usar diferentes tipos de archivos: .csv, .json, .txt, .html, .parquet.

Archivo orientado a filas o a columnas

Antes de revisar cada tipo de archivo en particular hay que entender el concepto de archivo orientado a filas o a columnas.

En los archivos orientados a filas los datos se organizan en registros y todos los datos asociados a un registro se guardan juntos en la memoria. Al guardarse la información por registro, un mismo registro puede tener diferentes tipos de datos, lo cual complejiza las consultas al mismo. Por lo tanto, realizar consultas sobre el valor de un atributo para diferentes registros resulta ineficiente ya que se debe cargar todo el registro con datos innecesarios. Este tipo de archivo está optimizado para leer y escribir filas de forma eficiente.

En los archivos orientados a columnas, en cambio, los datos se organizan por columna/campo/variable y todos los datos de la columna se guardan juntos en la memoria. Al guardar todos los datos de una columna juntos, cuando queremos consultar los valores de una columna solo necesitamos cargar esa columna sin necesidad de leer todo el archivo, como sucede con los archivos orientados a filas. Además, al ser todos los datos de una misma columna del mismo tipo, la compresión del archivo es mejor.

Supongamos que tenemos una tabla como la que se muestra abajo:

dni	nombre	apellido	año_nacimiento
40576890	Pedro	Aguirre	1995
32492645	Julia	Martinez	1988
30298710	Camila	Suarez	1985

Si el archivo se guarda orientado a filas tendrá esta forma:

row	value
row 1	40576890
	Pedro
	Aguirre
	1995
row 2	32492645
	Julia
	Martinez
	1988
row 3	30298710
	Camila
	Suarez
	1985

Mientras que si se guarda orientado a columnas tendrá esta otra:

column	value
dni	40576890
	32492645
	40576890
nombre	Pedro
	Julia
	Camila
apellido	Aguirre
	Martinez
	Suarez
año_nacimiento	1995
	1988
	1985

El siguiente post muestra de forma clara las ventajas y desventajas de cada tipo de archivo.

CSV

Comma-Separated Values (CSV) es un tipo de archivo que sirve para guardar y transferir datos tabulares. Los diferentes registros, filas, se separan entre sí mediante saltos de líneas, mientras que los atributos/variables, columnas, se separan usando la coma (también se pueden usar otros símbolos como el punto y coma o el tab). Hoy en día es uno de los formatos más utilizados en el análisis de datos.

```
Name, Age, Gender
John, 25, Male
Jane, 30, Female
Bob, 40, Male
```

Ejemplo de archivo CSV

Aplicación: el formato CSV se usa comúnmente para pequeños conjuntos de datos y como formato estándar para el intercambio de datos entre diferentes aplicaciones.

Ventajas:

- Casi todos los softwares que realizan tratamiento de datos pueden leerlos y escribirlos con facilidad, y además son fáciles de leer para las personas. Además, resulta sencillo generarlos desde casi cualquier lenguaje de programación.
- Se puede importar fácilmente a una amplia gama de herramientas de análisis de datos.

Desventajas:

- No es eficiente para almacenar grandes conjuntos de datos con tipos de datos complejos.
- Puede provocar la pérdida de datos si los valores contienen comas o saltos de línea.
  Soporte limitado para la codificación.
- El formato está orientado a filas y por lo tanto realizar consultas de agregación, que interesan en la ciencia de datos, tiende a ser poco eficiente.
- Los CSV no guardan información acerca del tipo de dato de su contenido puesto que todo se guarda en texto simple. Por lo tanto, el tipo de dato debe ser especificado al leer el archivo.

TXT

El tipo de archivo de texto sin formato, también conocido como formato .txt, es uno de los formatos más simples y ampliamente utilizados para el almacenamiento de datos. Este formato se utiliza con frecuencia en el campo de la ciencia de datos debido a su facilidad de uso y compatibilidad con una amplia variedad de herramientas y lenguajes de programación. Cuando se almacenan datos tabulares, los archivos TXT suelen ser muy similares a CSV.

Aplicaciones

El formato .txt se utiliza comúnmente para almacenar grandes conjuntos de datos de texto, como documentos, transcripciones, registros de chat, mensajes de correo electrónico, etc.
Se utiliza en el procesamiento de lenguaje natural (NLP) para almacenar y analizar conjuntos de texto, como textos médicos, noticias, redes sociales, entre otros.
También es utilizado para el almacenamiento y procesamiento de datos estructurados, y también en el etiquetado de conjuntos de datos para procesamiento en el ámbito del machine learning.

Ventajas:

- Los archivos .txt son simples, lo que los hace fáciles de crear y manipular con una variedad de herramientas de programación.
- El formato es legible por humanos, lo que permite una fácil inspección y edición de datos.
- Los archivos de texto sin formato son adecuados para el intercambio de datos entre sistemas, dada la simplicidad para leerlos y generarlos, aunque es necesario conocer como están estructurados los datos para poder luego procesar su información.

Desventajas:

- La falta de estructura en los archivos de texto sin formato puede dificultar su procesamiento automatizado. Esto puede hacer que el análisis de datos sea más lento y menos preciso en algunos casos y se requiera de validaciones o de la construcción de funciones especiales para interpretarlos (parseo).
- El formato .txt no es adecuado para almacenar datos complejos o información multimedia, como imágenes, audio o video.
- No es eficiente para almacenar grandes cantidades de datos con una alta densidad de información.

Apache Parquet

Parquet es un formato para el almacenamiento de datos tabulares, orientado a columnas, y está optimizado para grandes cargas de trabajo de datos. Fue desarrollado por Cloudera y Twitter en 2013 como un proyecto de código abierto. Parquet se basa en una representación de datos en columnas comprimidos, lo que lo hace muy eficiente para consultas analíticas que

involucran grandes cantidades de datos, en comparación de otros formatos como CSV. El parquet se usa a menudo en sistemas de procesamiento de Big Data basados en Hadoop como Hive, Impala y Spark.

Aplicaciones:

Parquet es un formato popular para el procesamiento de Big Data y se utiliza en una variedad de aplicaciones analíticas y de ciencia de datos. Algunos casos de uso específicos incluyen:

- Almacenamiento y procesamiento de conjuntos de datos a gran escala en sistemas basados en Hadoop como Hive e Impala, y en plataformas de almacenamiento en la nube como AWS S3.
- Análisis de datos con Spark y otros sistemas de procesamiento de Big Data.
- Aplicaciones de almacenamiento de datos e inteligencia empresarial que implican el análisis de grandes conjuntos de datos.

Ventajas:

- Compresión eficiente: Parquet es muy eficiente en lo que respecta a la compresión. Utiliza varios algoritmos de compresión como Snappy, LZO y Gzip para comprimir datos, lo que reduce los requisitos de almacenamiento y mejora el rendimiento de las consultas.
- Almacenamiento en columnas: Parquet almacena datos en columnas en lugar de filas, lo que lo hace más eficiente para consultas analíticas que generalmente implican leer solo un subconjunto de columnas de un gran conjunto de datos.
- Evolución del esquema: Parquet admite la evolución del esquema, lo que significa que puede agregar, eliminar o modificar columnas sin romper la compatibilidad con los datos existentes. Esto facilita la actualización de los modelos de datos a lo largo del tiempo.
- Soporte multiplataforma: Parquet es un proyecto de código abierto y es compatible con una variedad de sistemas de procesamiento de Big Data, incluidos Hadoop, Spark e Impala.

Desventajas:

- Rendimiento de escritura: el formato de almacenamiento en columnas de Parquet puede ser más lento que los formatos basados en filas para escrituras, especialmente cuando se agregan datos a columnas existentes.
- No apto para conjuntos de datos pequeños: Parquet está optimizado para consultas analíticas a gran escala y no es adecuado para conjuntos de datos pequeños.
- Sobrecarga de planificación de consultas: el almacenamiento en columnas requiere más sobrecarga de planificación de consultas que los formatos de almacenamiento basados en filas. Esto puede aumentar el tiempo de planificación de la consulta y hacerla más compleja

JSON

Explicados en detalle en la sección de datos semi-estructurados

HTML

El formato HTML (Lenguaje de Marcado de Hipertexto o HyperText Markup Language) es un lenguaje utilizado principalmente para crear páginas web y documentos de hipertexto. En el campo de la ciencia de datos, se puede utilizar como un formato de almacenamiento de datos para almacenar información estructurada y no estructurada.

```
<html>
  <head></head>
  <body>
    <table id="customers">
      <tbody>
        <tr>
          <th>Company</th>
          <th>Contact</th>
          <th>Country</th>
        </tr>
        <tr>
          <td>Alfreds Futterkiste</td>
          <td>Maria Anders</td>
          <td>Germany</td>
```

```
        </tr>
        <tr>
          <td>Centro comercial Moctezuma</td>
          <td>Francisco Chang</td>
          <td>Mexico</td>
        </tr>
        <tr>
          <td>Ernst Handel</td>
          <td>Roland Mendel</td>
          <td>Austria</td>
        </tr>
        <tr>
          <td>Island Trading</td>
          <td>Helen Bennett</td>
          <td>UK</td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```

Aplicaciones:

- El formato HTML se puede utilizar para almacenar datos de páginas web y documentos de hipertexto, como artículos, noticias, blogs, etc. Esto lo convierte en una opción popular para el análisis de medios sociales, análisis de sentimientos y análisis de opiniones en línea.
- El formato HTML se puede combinar con CSS (Cascading Style Sheets) y JavaScript para crear visualizaciones interactivas y aplicaciones web.
- Dado que HTML es el formato de las páginas Web, es común realizar Web Scraping, el cual es una técnica para extraer información de las mismas.

Ventajas:

- HTML es un lenguaje ampliamente utilizado y bien documentado, lo que lo hace fácil de entender y manipular.
- HTML es compatible con una amplia variedad de herramientas y lenguajes de programación, lo que lo hace una opción conveniente para la integración en flujos de trabajo de ciencia de datos.
- Los datos HTML se pueden analizar para extraer información estructurada y no estructurada.

Desventajas:

- El formato HTML puede ser complejo, lo que puede dificultar la extracción de datos específicos de páginas web grandes y complejas.
- El formato HTML puede ser susceptible a cambios en la estructura de la página, lo que puede afectar la calidad y la precisión de los datos extraídos.
- HTML no es un formato de almacenamiento de datos óptimo para grandes cantidades de datos o datos no estructurados.

Lectura de archivos con datos tabulares

Existen muchas formas de leer archivos en python, hasta el momento, en cursos anteriores vimos como leer csv usando el paquete `csv` y `pandas`

Paquete csv

El siguiente código muestra como abrir el archivo de `listings` de Airbnb, que usamos en Programación I, usando el paquete `csv`. Aquí podemos ver con claridad que el archivo csv es un archivo orientado a filas puesto que para poder acceder a los datos del mismo, csv tiene que recorrer con un `for loop` cada una de las filas del archivo. Por ejemplo, para poder calcular el precio promedio tiene que entrar a cada registro, extraer el precio (que está en la posición 9) y guardar el precio en la lista `lista_precios` . Luego de haber recorrido todos los registros, recién en ese momento, puede calcular el precio promedio como la suma de todos los precios dividida la longitud de la lista.

```
import csv
lista_precios = []
```

```
precio_por_barrio = {}

with open('listings.csv') as File:
    next(File)
    reader = csv.reader(File, delimiter=',')
    for fila in reader:
        precio = float(fila[9].replace('$', '').replace(',',''))
        lista_precios.append(precio)
precio_prom = sum(lista_precios)/len(lista_precios)
```

Paquete pandas

El paquete de `pandas` que aprendimos a usar en Programación II nos permite leer los datos de un archivo csv de forma sencilla. Si bien el archivo csv sigue siendo orientado a filas, la librería se encarga de ponerlo dentro de un objeto, DataFrame, que nos evita a nosotros tener que escribir el código necesario para extraer la información como en el caso anterior

```
import pandas as pd
data = pd.read_csv('listings.csv')
data.price.mean()
```

`pd.read_csv` también nos permite leer archivos `txt`

`Pandas` también nos permite leer archivos de excel, `xlsx` o `xls`. Si el archivo de excel tiene más de una hoja, se debe especificar con que hoja queremos trabajar.

```
import pandas as pd
data = pd.read_excel('archivo_excel.xlsx', sheet_name = 'hoja1')
```

Los archivos en formato `parquet` también pueden leerse usando la librería `pandas`. Según la documentación se realiza de la siguiente manera

```
pandas.read_parquet(path, engine='auto', columns=None, storage_options=None, use_nullable_dtypes=False, **kwargs)
```

El parámetro `engine` nos permite seleccionar la librería específica de `parquet` para leer el archivo: io.parquet.engine (’auto’), ‘pyarrow’, ‘fastparquet’.

Paquete JSON

La librería nativa `json` en Python proporciona una forma fácil de trabajar con datos JSON (Notación de Objetos JavaScript), que es un formato de intercambio de datos muy utilizado en aplicaciones web y móviles.

Se utiliza para convertir datos entre JSON y diccionarios en Python. Puede ser utilizado para convertir datos JSON a diccionarios Python y viceversa. La librería `json` también proporciona herramientas para trabajar con archivos JSON.

Ejemplo de uso para leer y escribir archivos:

Para leer un archivo JSON, primero debe abrir el archivo en modo de lectura y luego utilizar la función `load()` de la librería `json` para cargar los datos JSON del archivo en un objeto Python. Por ejemplo:

```
import json

# Abrir archivo JSON en modo lectura
with open('datos.json', 'r') as f:
    # Cargar los datos JSON del archivo en un objeto Python
    datos = json.load(f)

# Mostrar los datos cargados
print(datos)
```

Para escribir un archivo JSON, primero se debe abrir el archivo en modo de escritura y luego utilizar la función `dump()` de la librería `json`. Por ejemplo:

```
import json

# Datos a escribir en el archivo
datos = [
          {'nombre': 'Juan', 'edad': 30, 'ciudad': 'Rosario'},
          {'nombre': 'Marisa', 'edad': 50, 'ciudad': 'San Lorenzo'}
        ]

# Abrir archivo JSON en modo escritura
with open('datos.json', 'w') as f:
    # Escribir los datos Python en formato JSON en el archivo
    json.dump(datos, f)

# Leer el archivo JSON recién escrito
with open('datos.json', 'r') as f:
    datos_leidos = json.load(f)

# Mostrar los datos leídos
print(datos_leidos)
```

En este ejemplo, se ha creado un diccionario Python `datos` y se ha utilizado la función `dump()` para escribir los datos en un archivo `datos.json`. Luego, se ha utilizado la función `load()` para cargar los datos del archivo en un nuevo objeto Python `datos_leidos` y se han mostrado los datos en la consola.

También es posible leer archivos JSON utilizando la librería `pandas` . A continuación, se muestra un ejemplo de cómo leer un archivo JSON:

```
import pandas as pd

# Leer archivo JSON con Pandas
datos = pd.read_json('datos.json')

# Mostrar los datos cargados con Pandas
print(datos)
```

En este ejemplo, el archivo `datos.json` es leído utilizando la función `read_json()` de `pandas` y se almacena en un objeto Pandas DataFrame llamado `datos`. La función `read_json()` automáticamente convierte los datos JSON en un DataFrame de `pandas` . Esto permite que podamos cargar los datos en memoria, para realizar diversas operaciones y análisis de datos con facilidad, como el filtrado de datos, el cálculo de estadísticas y la visualización de gráficos.

Escritura de datos tabulares en archivos

Desde Python, es posible realizar la escritura de datos tabulares (filas/columnas) en formatos como CSV y Parquet. Por ejemplo, para escribir datos en un archivo CSV, primero debemos importar la librería `csv`. A continuación, podemos abrir el archivo en modo de escritura utilizando la función `open()`. Luego, creamos un objeto `csv.writer` que nos permitirá escribir los datos en el archivo en formato CSV. Por último, escribimos los datos en el archivo utilizando el método `writerow()` **.**

```
import csv

# Datos a escribir en el archivo CSV
datos = [
    ['Nombre', 'Edad', 'Ciudad'],    # Agregamos los nombres de las columnas en la primer file. Algunos CSV no lo utilizan
    ['Juan', 30, 'Rosario'],
    ['Ana', 25, 'Madrid'],
    ['Pedro', 40, 'Lima']
]

# Escribir los datos en un archivo CSV
with open('datos.csv', mode='w') as archivo:
    # Nota: Para sistemas Windows es conveniente usar lineterminator='\n'
    writer = csv.writer(archivo, lineterminator='\n')
    for fila in datos:
        writer.writerow(fila)
```

En este ejemplo, se han creado los datos a escribir en formato de una lista de listas. Luego, se ha utilizado la librería `csv` para escribir los datos en un archivo CSV llamado `datos.csv`.

Otra forma es también usando la librería pandas

```
import pandas as pd
datos = [
    ['Nombre', 'Edad', 'Ciudad'],    # Agregamos los nombres de las columnas en la primer file. Algunos CSV no lo utilizan
    ['Juan', 30, 'Rosario'],
    ['Ana', 25, 'Madrid'],
    ['Pedro', 40, 'Lima']
]

df = pd.DataFrame(index = [], columns = datos[0]) #creamos un DataFrame vacío con las columnas necesarias
j = 0
for i in datos[1:]:
    df.loc[j] = i #cargamos cada fila
    j+=1

df.to_csv('datos.csv', index = False) #guardamos los datos sin el índice

print(pd.read_csv('datos.csv'))
```

Ejemplo de escritura de datos en formato Parquet:

Para escribir datos en un archivo Parquet, primero debemos importar la librería `pyarrow`. A continuación, podemos convertir los datos en un objeto `pandas.DataFrame`. Luego, utilizamos el método `to_parquet()` para escribir los datos en el archivo en formato Parquet.

```
import pandas as pd
import pyarrow as pa
import pyarrow.parquet as pq

# Creamos un Dataframe de pandas a partir de un diccionario Python
# Son los datos a escribir en el archivo Parquet
datos = pd.DataFrame({
    'nombre': ['Juan', 'Ana', 'Pedro'],
    'edad': [30, 25, 40],
    'ciudad': ['Rosario', 'Madrid', 'Lima']
})

# Utilizamos pyarrow para escribir los datos en un archivo Parquet
tabla = pa.Table.from_pandas(datos)
pq.write_table(tabla, 'datos.parquet')
```

En este ejemplo, se han creado los datos en formato `pandas.DataFrame` . Luego, se ha utilizado la librería `pyarrow` para convertir los datos en un objeto `pa.Table` y utilizar el método `pq.write_table()` para escribir los datos en un archivo Parquet llamado `datos.parquet` . El archivo resultante puede ser leído por cualquier aplicación que soporte el formato Parquet.

Tipos de datos

Los archivos CSV, como se comentó, no guardan información sobre tipos de datos ya que toda la información se guarda en texto simple. Cuando leemos un archivo con el paquete `csv` todo se lee como texto, mientras que cuando lo abrimos con `pandas` la librería misma asigna un tipo de dato o bien nosotros podemos explicitarlo mediante un parámetro en el comando de lectura. Por ejemplo:

```
pd.read_csv('listings.csv', dtype = {'price': 'float'}
```

También es posible inspeccionar los tipo de datos de las columnas de un Dataframe `pandas` :

```
import pandas as pd

# Descarga el dataset "titanic.csv" y lo carga en un DataFrame
df = pd.read_csv('https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv')

# Imprime información del DataFrame
print(df.info())
```

Obtenemos el siguiente resultado:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
None

Process finished with exit code 0
```

En Pandas, las columnas que contienen valores de texto se representan como el tipo de datos `object` en lugar del tipo de datos `str`. Esto se debe a que `str` es un tipo de datos específico en Python que solo puede contener caracteres alfanuméricos y algunos caracteres especiales, mientras que `object` puede contener cualquier tipo de objeto de Python, incluyendo cadenas de texto.

Además, las columnas de texto en un DataFrame de Pandas pueden contener valores faltantes (por ejemplo NaN), y `object` es un tipo de datos compatible con valores faltantes, lo que permite que una columna de texto contenga valores faltantes sin afectar su tipo de datos.

En resumen, aunque las columnas de texto se pueden representar como `str` en Python, en Pandas se representan como `object` para permitir la inclusión de valores faltantes y cualquier otro tipo de objeto de Python.

Tipos de datos usuales

Los tipos de datos que más van a utilizar son:

- `int`: para representar valores enteros. En general el tipo de dato entero puede ser *byte*, *short*, *int* o *long* y cada uno se corresponde con el volumen de memoria que puede ocupar el dato, 8, 16, 32 o 64 bits respectivamente. A partir de python 3, todos los enteros son de formato `long`, es decir 64 bits. Sin embargo, algunas librerías tienen sus propios tipos de datos y utilizan diferentes tipos de enteros. Por ejemplo, numpy tiene el tipo de dato `np.int32` e `np.int64`. Este post de la documentación de Numpy explica, por ejemplo, el comportamiento de los enteros de Numpy en comparación de los enteros nativos de Python ante un error de overflow.
- `float`: para representar valores reales de coma flotante. Existe el `float` single precision (32 bits) y el double precision (64 bits)
- `str`: sirve para representar texto.
- `bool`: para representar valores booleanos de True/False
- NaN/None/Null

Python tiene tipos de datos nativos y luego cada librería tiene sus tipos de datos. Por ejemplo, arriba comentamos que la librería Numpy tiene los enteros `np.int32` y `np.int64`. Pandas, por su lado, tiene un tipo de dato llamado `object` el cuál generalmente aparece cuando pandas interpreta un archivo y se encuentra con una columna con texto. Como Pandas no sabe como asignar automáticamente el tipo de datos, utiliza `object` como genérico, y luego nosotros debemos indicar que se interprete a esa columna como un `str` o el tipo que corresponda.

Vamos a encontrarnos también con el tipo de columna `object` cuando tenemos datos de diferente tipo en una misma columna (mixed):

```
# Creamos un Dataframe. La columna 'rank' tiene datos mixtos
df = pd.DataFrame([('sparrow', 30.0, 2),
```

```
                   ('tiger', 90.5, '1')],
                  columns=('name', 'max_speed', 'rank'))

# Chequear el tipo de dato de la columna rank
print(df['rank'].dtype)  # rank es de tipo 'object' ya que tiene los valores '1' y 2

print(df['rank'][0], type(df['rank'][0]))
# Imprime: 2 <class 'int'>
# Para Python, el valor rank del primer registro (0) es 2 y del tipo <class 'int'>

print(df['rank'][1], type(df['rank'][1]))
# Imprime: 1 <class 'str'>
# Para Python, el valor rank del primer registro (1) es '1' y del tipo <class 'str'>
```

En muchas oportunidades vamos a intentar pasar de un tipo de dato a otro, para eso podemos usar el siguiente comando:

```
data['columna'] = data['columna'].astype('int')
```

Definir el tipo de dato para una columna es intuitivo la mayoría de las veces, incluso, pandas o algún otro paquete con el que leamos el archivo nos va a ayudar en la tarea. A continuación enumeramos algunos problemas que se presentan en la práctica basados en nuestra experiencia:

- Perder datos por no leerlos con el tipo de dato apropiado. Vamos a encontrar columnas que tienen datos con una longitud fija de dígitos numéricos, por ejemplo, una columna de 6 dígitos que indique la localización del dato, con una estructura como la siguiente `xxxxxx` . Además, esta estructura puede estar formada por dos valores diferentes, los primeros 2 dígitos indican la ciudad y los últimos 4 el distrito, `ccdddd`. Si el código de la ciudad puede comenzar con el valor 0, si leemos el dato como `int`, entonces, perderemos los primero caracteres. Por ejemplo, para un dato proveniente de la ciudad `01`, en vez de lee `013349` vamos a leer `13349`. Luego, cuando queramos recuperar la ciudad de donde proviene un dato a partir de la columna de localización extrayendo los primeros 2 dígitos, obtendremos el valor `13`, en lugar del `01` esperado. En este caso, el tipo de dato óptimo es el `str`
- Intentar convertir todos los datos de una columna a `str` cuando existen valores nulos en una columna de un pandas DataFrame. Los valores nulos pueden coexistir con los valores numéricos pero no con las cadenas. Una recomendación para estos casos es darle un trato a los valores nulos primero y luego definir el tipo de dato como `str`

Datetime

Una gran cantidad de datasets que vamos a analizar van a contener columnas con datos de fecha, hora o de fecha y hora. La forma más conveniente para manipular y utilizar estos datos es usar el tipo de dato específico para los mismos. El módulo `datetime` de python nos provee clases para manipular fechas y horas. Documentación

Los objetos de fecha y hora pueden ser categorizados como “aware” o “naive” dependiendo si incluyen o no información sobre el huso horario.

Los datos “aware”, como indica la documentación, representan un momento específico en el tiempo y pueden compararse con otros tipos de datos “aware”. Para el manejo de zonas horarias, se utiliza la librería `pytz` de Python.

Trabajar con datos “naive” es mucho más sencillo pero uno tiene que tener en cuenta las limitaciones de los mismos.

```
import datetime
import pytz

# Ejemplo de fecha "naive"
fecha_naive = datetime.datetime(2022, 3, 7, 15, 30)
print(fecha_naive) # 2022-03-07 15:30:00

# Ejemplo de fecha "aware"
zona_horaria = pytz.timezone("America/Mexico_City")  # -6 Horas de diferencia con UTC
fecha_aware = datetime.datetime(2022, 3, 7, 15, 30, tzinfo=zona_horaria)
print(fecha_aware) # 2022-03-07 15:30:00-06:00
```

Fechas en Pandas

Algo a tener en cuenta, es que la carga de fechas desde un CSV será interpretada como un string (`pandas` le asigna el tipo `object`). Por ejemplo si tenemos el siguiente dataset:

```
fecha,valor
2022-03-01,100
2022-03-02,150
2022-03-03,200
2022-03-04,250
2022-03-05,300
2022-03-06,350
```

Para un correcto tratamiento de las fechas, debemos indicar a `pandas` que una columna es de tipo datetime64, por ejemplo del siguiente modo:

```
import pandas as pd
from datetime import datetime

# Cargar los datos en un dataframe de Pandas
df = pd.read_csv('datos.csv')

# Verificar el tipo de datos de la columna "fecha"
print(df['fecha'].dtype)  # Tipo object

# Convertir la columna "fecha" a un tipo datetime64
df['fecha'] = pd.to_datetime(df['fecha'])

# Verificar el tipo de datos de la columna "fecha"
print(df['fecha'].dtype)  # Tipo datetime64[ns]
```

Este código carga los datos en un dataframe de Pandas, convierte la columna "fecha" a un tipo datetime64 con el método `pd.to_datetime()`, y verifica que el tipo de datos de la columna sea correcto utilizando la propiedad `dtype`. Al convertir la columna a `datetime64`, facilitará el tratamiento desde Python, ya que podríamos por ejemplo calcular la diferencia entre fechas de una manera más sencilla, que si la columna es considerada una cadena de texto. La salida del programa es la siguiente:

```
object
datetime64[ns]
```

`datetime64` es un tipo de datos de Pandas que representa una fecha y hora con precisión de nanosegundos. Este tipo de datos se puede usar para almacenar fechas y horas en un formato compacto y eficiente, lo que lo hace ideal para trabajar con grandes conjuntos de datos.

La precisión de `datetime64` se basa en el tipo de datos de arrays de NumPy, ya que Pandas se construye sobre esta biblioteca. NumPy también proporciona un tipo de datos similar llamado `datetime64`, que se utiliza para representar fechas y horas con precisión de nanosegundos.

`datetime64` es un tipo de datos numérico que se representa internamente como un número entero de 64 bits. Cada unidad de fecha y hora (año, mes, día, hora, minuto, segundo, nanosegundo) se convierte en un número entero que representa la cantidad de esa unidad desde una fecha de referencia, que es el 1 de enero de 1970. Esta fecha de referencia se utiliza como base para el cálculo de todas las demás fechas y horas.

La precisión de `datetime64` se puede controlar mediante los modificadores de unidades de tiempo. Por ejemplo, `datetime64[s]` representa una fecha y hora con precisión de segundos, `datetime64[ms]` representa una fecha y hora con precisión de milisegundos y `datetime64[us]` representa una fecha y hora con precisión de microsegundos.

Equivalencias de tipos de datos

En la siguiente tabla, se puede ver como compatibilizar los tipos de datos de las diferentes librerías:

Pandas `dtype` mapping

Pandas dtype	Python type	NumPy type	Usage
object	str or mixed	string_, unicode_, mixed types	Text or mixed numeric and non-numeric values
int64	int	int_, int8, int16, int32, int64, uint8, uint16, uint32, uint64	Integer numbers
float64	float	float_, float16, float32, float64	Floating point numbers
bool	bool	bool_	True/False values
datetime64	NA	datetime64[ns]	Date and time values
timedelta[ns]	NA	NA	Differences between two datetimes
category	NA	NA	Finite list of text values

Convenciones de nombres y buenas prácticas

Las convenciones de nombres y buenas prácticas en ciencia de datos y manipulación de datos son importantes para mantener un código organizado, fácil de entender y mantener. Además, Python cuenta con una guía de estilo para facilitar la lectura y mantenimiento del código **PEP 8 – Style Guide for Python Code (**https://peps.python.org/pep-0008/).

Algunas de las convenciones y buenas prácticas más comunes son:

- Utilizar nombres en minúscula y separados por guiones bajos: En Python se utilizan nombres bajo la convención "Snake Case", donde los nombres de las variables deben estar en minúscula y separados por guiones bajos. Por ejemplo, `edad_promedio` en lugar de `EdadPromedio` o `edadpromedio`.

```
# Convenciones de nombres inconsistentes
ListaDePalabras = ["hola", "mundo"]
DICCIONARIO_DE_DATOS = {"clave1": 1, "clave2": 2}

# Convenciones de nombres bien utilizadas
lista_numeros = [1, 2, 3, 4, 5]
lista_palabras = ["hola", "mundo"]
diccionario_datos = {"clave1": 1, "clave2": 2}
```

- Nombrar variables de manera descriptiva: Los nombres de las variables deben ser descriptivos y explicar claramente lo que representan. Los nombres deben ser cortos pero no demasiado abreviados.

```
# Mal nombre de variable. Demasiado corto
x = [1, 2, 3, 4, 5]

# Buen nombre de variable
lista_numeros = [1, 2, 3, 4, 5]

# Mal nombre de variable. Demasiado largo.
esto_es_una_lista_de_numeros_enteros = [1, 2, 3, 4, 5]
```

- Documentar el código: Es importante documentar el código para que sea fácil de entender para otras personas que puedan leerlo en el futuro. Esto incluye agregar comentarios descriptivos y explicativos en el código. En Python se utilizan los caracteres "#" para comentarios, o bien las """ para bloques de comentarios.

```
# Este es un comentario en una sola línea
nombre = "Juan"  # Este es un comentario en una línea de código
edad = 30

"""
Este es un bloque de texto de comentarios más grande.
Aquí se pueden agregar notas y explicaciones más detalladas
sobre el código.
"""
nombre = "Juan"
edad = 30
```

- El agregar de espacios en blanco se puede utilizar para mejorar la legibilidad del código. Por ejemplo, agregar líneas en blanco entre bloques de código o después de las declaraciones de funciones, o separar variables y operadores con espacios:

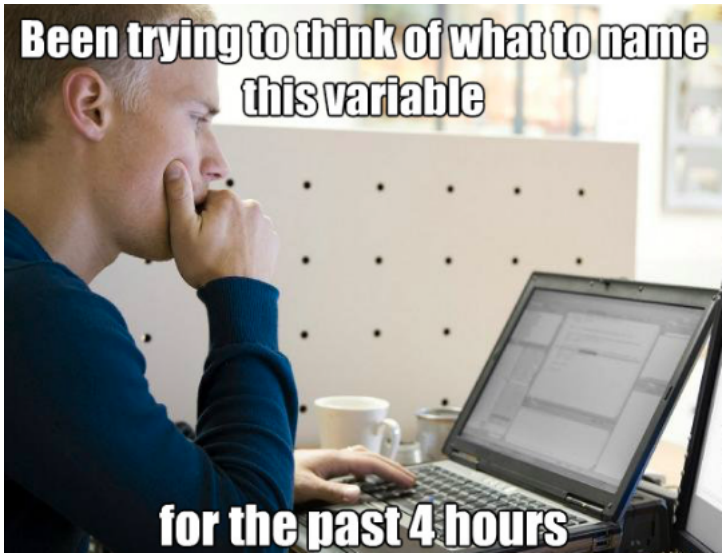
```
# Buen uso de espacios y líneas en blanco
def suma(a, b):
    resultado = a + b
    return resultado

# Mal uso de espacios y líneas en blanco
def suma(a,b):
    resultado=a+b
    return resultado

# Buen uso de líneas en blanco: Utilizar una línea en blanco después de la definición de funciones y clases
class Persona:

    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")
```

- Evitar nombres de variables reservadas: Es importante evitar utilizar nombres de variables que son reservados en el lenguaje de programación que se está utilizando. Algunas palabras reservadas en Python son: `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `False`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `None`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `True`, `try`, `while`, `with`, `yield`.

```
# Ejemplo 1: utilizar una palabra reservada como nombre de variable
def = 10  # Error de sintaxis, 'def' es una palabra reservada

# Ejemplo 2: utilizar una palabra reservada como nombre de función
def lambda():  # Error de sintaxis, 'lambda' es una palabra reservada
    pass

# Ejemplo 3: utilizar una palabra reservada como nombre de argumento en una función
def multiplicar(a, in):
    return a * in  # Error de sintaxis, 'in' es una palabra reservada
```

1. Utilizar constantes en mayúsculas: Las constantes deben estar en mayúsculas para indicar que son valores fijos que no deben cambiar. Por ejemplo, `PI` en lugar de `pi` o `Pi`.

```
# Ejemplo 1: utilizar una constante en mayúsculas
PI = 3.14159  # Se utiliza PI como una constante en mayúsculas

radio = 10
area = PI * radio**2

print("El área del círculo es:", area)

# Ejemplo 2: utilizar varias constantes en mayúsculas
DAYS_IN_A_WEEK = 7
HOURS_IN_A_DAY = 24
SECONDS_IN_A_MINUTE = 60

segundos_en_un_dia = SECONDS_IN_A_MINUTE * 60 * HOURS_IN_A_DAY
print("Segundos en un día:", segundos_en_un_dia)
```



Datos en forma larga o ancha

Una forma sencilla de entender la forma larga o ancha (long or wide) es con un ejemplo como el de abajo. En este ejemplo tenemos una encuesta de movilidad donde a cada persona se le pregunta cuál es el tiempo de viaje y costo para ir de su casa al trabajo para 4 modos de transporte diferentes, auto, moto, bus y bici, y cuál es el modo elegido por la persona. En el formato ancho podemos ver que tenemos un solo registro por persona, mientras que en el formato largo tenemos un registro por cada persona y cada modo por el cuál se le consultó.

Los datos en forma ancha son aquellos en los que la columna que identifica al dato no tiene valores repetidos, en nuestro ejemplo la `persona_id`. Mientras que en la forma larga, la columna que identifica al registro:

- Tiene valores repetidos
- Ya no puede utilizarse por si misma como identificación inequívoca del registro, sino que debe combinarse con otra columna, en este caso, el modo.

Forma ancha

persona_id	tiempo_viaje_auto	tiempo_viaje_moto	tiempo_viaje_bus	tiempo_viaje_bici	costo_auto	costo_moto	cost
1	10	8	15	20	200	100	100
2	20	15	45	50	1000	600	100

Forma larga

persona_id	modo	tiempo_viaje	costo	modo_elegido
1	auto	10	200	bici
1	moto	8	100	bici
1	bus	15	100	bici
1	bici	20	0	bici
2	auto	20	1000	auto
2	moto	15	600	auto
2	bus	45	100	auto
2	bici	50	0	auto

Ejemplo para pasar de formato a otro

Para pasar nuestra tabla de formato ancho a formato largo vamos a utilizar la operación `pd.melt` de Pandas que nos permite agrupar varias columnas en una sola.

```
import pandas as pd
import random

##### generar datos de juguete para el ejemplo
data = pd.DataFrame(index = range(100), columns=['tiempo_viaje_auto', 'tiempo_viaje_moto', 'tiempo_viaje_bus', 'tiempo_viaje_bici', 'c
```

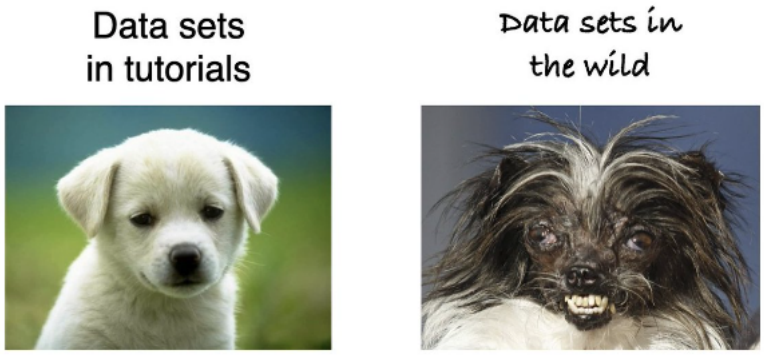
```
data['tiempo_viaje_auto'] = [random.randint(1,100) for x in range(0,100)]
data['tiempo_viaje_moto'] = [random.randint(1,100) for x in range(0,100)]
data['tiempo_viaje_bus'] = [random.randint(1,100) for x in range(0,100)]
data['tiempo_viaje_bici'] = [random.randint(1,100) for x in range(0,100)]
data['costo_auto'] = [random.randint(1,100) for x in range(0,100)]
data['costo_moto'] = [random.randint(1,100) for x in range(0,100)]
data['costo_bus'] = [random.randint(1,100) for x in range(0,100)]
data['costo_bici'] = [random.randint(1,100) for x in range(0,100)]
data['modo_elegido'] = [random.randint(1,4) for x in range(0,100)]
data.index.name = 'person_id'
data.reset_index(inplace = True)

##### generar una tabla en formato largo para el tiempo de viaje y otra para costos
tt = pd.melt(data, id_vars = ['person_id','modo_elegido'], value_vars = ['tiempo_viaje_auto','tiempo_viaje_moto','tiempo_viaje_bus','t
tc = pd.melt(data, id_vars = ['person_id'], value_vars = ['costo_auto','costo_moto','costo_bus','costo_bici']).set_index('person_id')

tt.rename(columns = {'variable': 'modo', 'value': 'tiempo_viaje'}, inplace = True)
tc.rename(columns = {'value': 'costo_viaje'}, inplace = True)
tc.drop('variable', inplace = True, axis = 1)

##### concatenar ambos dataframes en el eje 1 para obtener un solo resultado final
df = pd.concat([tt,tc], axis = 1)
df.modo.replace('tiempo_viaje_auto','auto', inplace = True)
df.modo.replace('tiempo_viaje_moto','moto', inplace = True)
df.modo.replace('tiempo_viaje_bus','bus', inplace = True)
df.modo.replace('tiempo_viaje_bici','bici', inplace = True)

person_id  modo_elegido  modo  tiempo_viaje  costo_viaje
0                     3  auto             4           54
1                     3  auto             1           56
2                     3  auto            11           83
3                     3  auto            98           81
4                     2  auto            16           88
...                 ...   ...           ...          ...
95                    1  bici            58           43
96                    4  bici            80           66
97                    2  bici            66           71
98                    3  bici            77           69
99                    4  bici            16           28

[400 rows x 4 columns]
```

Para pasar de formato ancho a formato largo podemos usar la operación de pandas `pivot`. Continuado con nuestro ejemplo de arriba

```
df.reset_index(inplace = True)
# Pivoteamos la tabla en dos valores, el costo y el tiempo. Notar que se genera una columna multinivel
pivoteada = df.pivot(index = 'person_id', columns = 'modo', values = ['tiempo_viaje', 'costo_viaje'])
# La operación anterior nos hizo perder el modo que eligió cada persona.
df.drop_duplicates('person_id', inplace = True)
df.set_index('person_id', inplace = True)
pivoteada['modo_elegido'] = df['modo_elegido']
print(pivoteada)

                 tiempo_viaje              costo_viaje             modo_elegido
modo             auto bici bus moto       auto bici  bus moto
person_id
0                   4   66  65   77          54    8    6   80            3
1                   1   16  88   67          56   32  100   80            3
2                  11   35  50   31          83   88   48   58            3
3                  98   52  90   20          81   76   45   27            3
4                  16   97  85   61          88   21   54   18            2
...               ...  ...  ..  ...         ...  ...  ...  ...          ...
95                 27   58  86    1          46   43   51    5            1
96                 76   80  13   22          42   66   82   26            4
97                 64   66  54   28          82   71   54   37            2
98                 13   77   4   76          46   69   64   78            3
99                 12   16  89   52           6   28   40   32            4

[100 rows x 9 columns]
```

Manipulación de datos

El *Data Wrangling* por su nombre en inglés es el proceso de prepara los datos y ponerlos en el formato necesario para poder realizar un posterior análisis de los mismos. La mayoría de las veces los datos que estén usando van a requerir limpieza.



Manejo de datos faltantes

La mayoría de los datasets tienen columnas con datos faltantes a los cuales vamos a tener que darle un tratamiento para poder usar el dataset. El origen de los datos faltantes es variado:

- Mala recolección de los datos
- No es esperable que el registro tenga un valor para esa columna. Por ejemplo, si tenemos una tabla con personas que viven en un hogar, no es esperable que los niños informen cantidad de horas trabajadas en la semana.

Dependiendo del origen de los datos faltantes, el tratamiento que les demos va a ser diferente. A continuación listamos algunas de los recursos que podemos utilizar para lidiar con los datos faltantes.

Eliminar registros con datos faltantes

Esta opción es la más sencilla puesto que vamos a eliminar cualquier registro que tenga algún dato de interés faltante. Sin embargo, esta opción reduce el tamaño de nuestra muestra, por lo tanto es viable cuando el número de registros con datos faltantes no es elevado.

Supongamos que realizamos una encuesta para recolectar datos para estimar un modelo que nos informe la probabilidad de que una persona compre un celular. En la encuesta se pregunta el ingreso mensual a cada persona puesto que es un atributo que esperamos que sea una buena variable explicativa de la decisión de comprar o no un celular. Sin embargo, también sabemos que muchas personas deciden no contestar preguntas relacionadas a sus ingresos. Por lo tanto, cuando estemos estimando el modelo, vamos a descartar a todas las personas que no incluyeron su ingreso en su respuesta.

```
import pandas as pd
data = pd.read_csv('compra_celular.csv')
#eliminamos los registros con valores nulos
data = data[data.ingreso.notnull()].copy()
```

Reemplazar por valor resumen

Otra opción es reemplazar los valores faltantes por medidas de resumen. Por ejemplo, supongamos que tenemos una muestra de hogares de la ciudad de Rosario y una de las variables con datos faltantes es `precio_alquiler`. Entonces, podríamos pensar en imputar los datos que nos faltan con el precio medio de los alquileres del resto de la muestra. También, podríamos imputar los valores faltantes usando la mediana, o la moda o algún otro valor resumen que se adapte a nuestro estudio.

> 💡 Antes de realizar esta operación (al igual que la siguiente) hay que evaluar cuántos datos hay disponibles para calcular la medida de resumen. Si tenemos pocos valores para calcular le medida de resumen, esta puede no ser una opción viable

```
import pandas as pd
data = pd.read_csv('hogares.csv')
# Calculamos el precio de alquiler promedio
alquiler_medio = data[data['precio_alquiler'].notnull()].mean()
# Reemplazamos los valores nulos con el precio medio
data.precio_alquiler.fillna(alquiler_medio, inplace = True)
```

Reemplazar por valor más cercano

El caso anterior podemos mejorarlo si en lugar de imputar los valores faltantes usando una medida de resumen extraída de la muestra general, a la medida de resumen la calculamos para puntos “cercanos” al valor faltante. En nuestro ejemplo, si el valor faltante pertenece a un hogar de Barrio Alberdi, para imputar el valor nos conviene usar el `precio_alquiler` informado por otros hogares de ese mismo barrio y no de Rosario en general.

La idea de “cercanía” es un concepto muy utilizado en el análisis de datos y no solo hace referencia a una cercanía espacial o en distancia, sino que para definir la cercanía se pueden considerar diferentes aristas. También podemos pensarlo como clases/segmentos en los que dividimos el espacio. A continuación listamos algunos ejemplos de cercanía los cuales pueden considerarse juntos o separados a la hora de imputar datos:

- **Cercanía espacial**: dos puntos de medición se encuentran a una distancia X uno de otro. Si la distancia X no supera un límite establecido anteriormente, esos dos puntos pueden considerarse cercanos y usar los datos de uno para imputar datos faltantes del otro
- **Clase socioeconómica**: Dos datos pueden considerarse cercanos si pertenecen al mismo segmento socioeconómico. Los segmentos se definen de forma arbitraria (o puede usarse algún modelo) acorde al estudio que se esté realizando. Por ejemplo, un segmento podría ser personas de entre 25-35 años, que son mujeres y no tienen hijos.
- **Cercanía temporal**: Dos puntos de datos pueden considerarse cercanos si fueron recolectados o registrados en momentos similares. Por ejemplo, en un estudio sobre las temperaturas diarias de una ciudad, los datos de las temperaturas registradas en días consecutivos podrían considerarse cercanos temporalmente. Este tipo de cercanía es particularmente útil en series temporales y análisis de tendencias, donde los cambios en el tiempo pueden influir en las observaciones.

Pensar cómo resolver por cercanía, cuando en un dataset de valores de propiedades, nos falta el valor de mercado de una casa o un departamento

Estimar función y predecir valor

Otra forma de reemplazar los valores faltantes es estimar un modelo que tenga como variable dependiente a la variable con datos faltantes. Luego, podremos predecir los valores nulos utilizando el modelo estimado.

Interpolación numérica

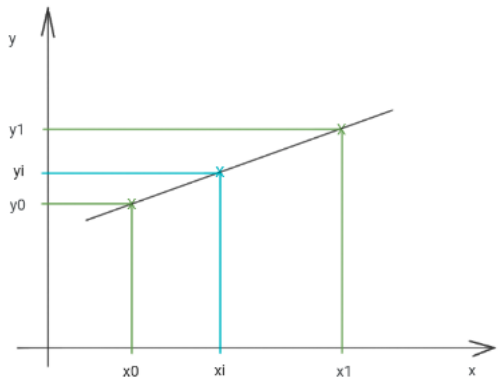
Supongamos que tenemos un conjunto de datos $(x_1, y_1), (x_2, y_2)...(x_{n+1}, y_{n+1})$ que fueron generados con una función desconocida. También supongamos que tenemos un valor de x_i y queremos saber cuál es el correspondiente valor y_i. Para obtener el resultado vamos a buscar una función/funciones que nos permitan calcular el valor deseado.

Interpolación lineal

Es la forma más sencilla de realizar una interpolación.

Teniendo dos puntos $(x_0, y_0), (x_1, y_1)$ podemos calcular una única recta que pase por los mismos. La función obtenida sirve para calcular el valor de y para cualquier valor de x perteneciente al intervalo $[x_0, x_1]$:

$$\frac{x_1 - x_0}{x_i - x_0} = \frac{y_1 - y_0}{y_i - y_0}$$

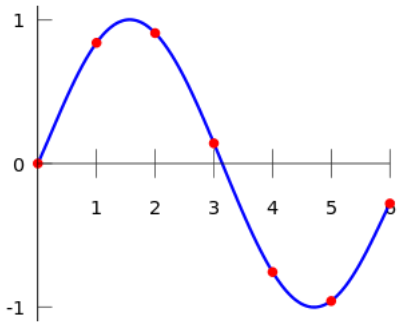


💡 Si el valor de x que queremos usar en la predicción se encuentra por fuera del rango establecido, esto deja de ser una interpolación y será una **extrapolación**.

Interpolación polinómica

Esta interpolación es global, va a buscar un polinomio que pase por todos los puntos que tenemos como dato para obtener una ecuación que estime y_i como $f(x_i)$. Dependiendo de la cantidad de puntos va a ser el grado del polinomio:

- 2 puntos: una recta, polinomio de grado 1
- 3 puntos NO alineados: una parábola, polinomio de grado 2
- 4 puntos NO alineados: un polinomio de grado 3
- n + 1 puntos NO alineados: un polinomio de grado n



💡 Existen diversos métodos de interpolación no lineales, por ejemplo: Método de Newton, de Lagrange, interpolación de spline, etc. De acuerdo al método elegido, se pueden obtener valores de y_i similares o por el contrario, bastante diferentes. Por eso es importante analizar en cada caso el método más apropiado.

Interpolación por intervalos

La interpolación que vimos arriba es global, es decir, utilizan todos los datos para generar una sola función. Existe otro tipo de interpolación llamada interpolación por intervalos en la que generaremos, como su nombre lo indica, una función para cada intervalo de los datos.

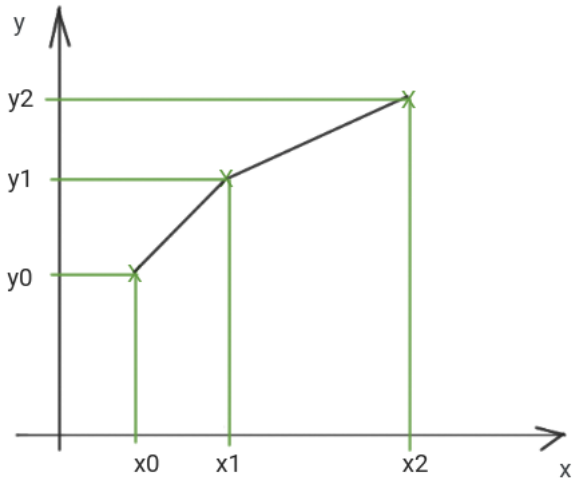
Supongamos que tenemos los datos $(x_1, y_1), (x_2, y_2)...(x_{n+1}, y_{n+1})$ vamos a generar n funciones. Estas funciones pueden ser lineales y para generarlas se sigue la misma lógica que vimos en la sección de interpolación lineal. A continuación mostramos la definición general:

$$y = f_i(x) \qquad , \qquad x_i < x < x_{i+1}$$

Por ejemplo, supongamos que tenemos 3 puntos $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ las ecuaciones para realizar la interpolación serán:

$$f_1(x) = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x_i - x_0) \qquad , \qquad x_0 < x_i < x_1$$

$$f_2(x) = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x_i - x_1) \qquad , \qquad x_1 < x_i < x_2$$



Expresiones regulares

Una expresión regular es una secuencia de caracteres que especifica un patrón de búsqueda. También son conocidas como `regex` o `RE` por su nombre en inglés de regular expressions. Según la documentación de Python un RE especifica las cadenas que coinciden con ellas (o las que no). Por ejemplo, vamos a poder saber si la sub-cadena `rr` coincide de alguna manera con la cadena `r con r guitarra, r con r barril, r con r que rápido ruedan las ruedas del ferrocarril` . La coincidencia puede ser simple, es decir si la `rr` aparece en algún lugar de la cadena, o puede ser más compleja, por ejemplo si la `rr` aparece al principio o al final de la cadena.

A continuación listamos algunos ejemplos de las RE, para una lista completa, consultar la documentación:

- Caracter: todos los caracteres, excepto los especiales según RE, coinciden con ellos mismos, como en el ejemplo de arriba.
- Secuencia de caracteres: buscar la coincidencia de una cadena dentro de otra. Por ejemplo, buscar `casa` dentro de otra cadena como `Mi casa es naranja` las `flores florecieron`. En el primer caso tenemos una coincidencia y en el segundo ninguna.
- Caracteres especiales:

Caracter	Descripción	Ejemplo
\	Para evadir a los caracteres especiales.	+ es un caracter especial, pero si deseamos buscarlo con una RE podemos hacerlo usando `\+`. Del mismo modo, si deseamos buscar a `\` debemos usar `\\`
[]	Conjunto de caracteres.	[0-9] acepta la coincidencia con cualquier caracter dentro del rango 0 a 9. [0 3 9] busca la coincidencia con 0, 3 y/o 9 [^ 0 3 9] acepta a cualquiera que NO sea 0, 3 o 9
.	Cualquier caracter excepto una nueva linea	“c.sa” acepta la coincidencia “cosa”, “casa”, etc.
^	Comienza con	“^camino” acepta cadenas que comienzan con la palabra camino
$	Termina con	“camino$” acepta cadenas que terminan con la palabra camino
*	Cero o más ocurrencias	“ca.*e” acepta la cadena “calle”
+	Una o más ocurrencias	“ca.+e” acepta la cadena “calle”
?	Cero o una ocurrencia	“ca.?e” NO acepta la cadena “calle”
{}	Solamente el número especificado de ocurrencias	“ca.{2}e” acepta la cadena ”ca.{4}” NO acepta la cadena calle
|	Una o la otra	“cuatro|4” acepta la cadena

		“cuatro” o “4”
\A	Devuelve un match si la coincidencia ocurre al principio de la cadena	“\ALa” acepta la cadena “La casa”
\b	Devuelve un match si el patrón aparece al principio o final de una palabra	“\bsa” NO acepta la palabra “cosa” ”sa\b” acepta la palabra “cosa”
\d	Devuelve un match si la cadena contiene dígitos	“\d” acepta “1”
\D	Devuelve un match si la cadena NO contiene dígitos	“\D” no acepta “1”
\w	Devuelve un match si la cadena contiene caracteres de palabras	“\w” acepta “A”
\W	Devuelve un match si la cadena NO contiene caracteres de palabras	“\W” NO acepta “A” ”\W+” acepta “!!!”

💡 Las palabras se definen como una secuencia de caracteres de palabras. Entonces `r'\bfoo\b'` coincide con `'foo'`, `'foo.'`, `'(foo)'`, `'bar foo baz'` pero no con `'foobar'` o `'foo3'`

Raw String Notation

La raw string notation, `r"texto"` permite que las expresiones regulares sean interpretadas tal cual fueron escritas. Sin esto, la barra `\` sería interpretada como un caracter especial y deberíamos agregarle otra barra para escapar del mismo.

Cómo aplicar Regex en python

Python cuenta con un paquete llamado `re` que permite usar expresiones regulares. El mismo cuenta con varias funciones y en este apunte mencionamos solo algunas, consultar la respectiva documentación para conocer más al respecto.

- re.search(pattern, string, flags=0)
  Busca la primera ocurrencia del patrón en la cadena y devuelve un objeto del tipo match.
- re.split(pattern, string, maxsplit=0, flags=0)
  Parte la cadena en cada ocurrencia del patrón. Si el patrón está entre paréntesis, también se devuelve el texto en cada ocurrencia. Si *maxsplit* se configura como un valor diferente a cero, entonces, se devuelven como máximo *maxsplits* y el resto se devuelve como una cadena.
- re.findall(pattern, string, flags=0):
  Devuelve todos las ocurrencias del patrón encontradas en la cadena como una lista o tupla.
- re.sub(pattern, repl, string, count=0, flags=0):
  Se utiliza para realizar reemplazos en una cadena. Devuelve una cadena en la cuál se reemplazo cada ocurrencia del patrón por el valor *repl*. Si no se encontró ningún valor, entonces, se devuelve la cadena original, sin modificar.

Veamos un ejemplo:

La regex `r'\D+'` va a buscar todo lo que NO sea un dígito que ocurra una o más veces. Mientras que `r'\D+'` busca dígitos que ocurran una o más veces

```
import re

>>> re.search(r'\D+', 'Se necesitan 30 azulejos para revestir 1 m2')
<re.Match object; span=(0, 13), match='Se necesitan '>

>>>re.search(r'\d+', 'Se necesitan 30 azulejos para revestir 1 m2')
<re.Match object; span=(13, 15), match='30'>

>>> re.findall(r'\D+', 'Se necesitan 30 azulejos para revestir 1 m2')
['Se necesitan ', ' azulejos para revestir ', ' m']

>>> re.findall(r'\d+', 'Se necesitan 30 azulejos para revestir 1 m2')
['30', '1', '2']
```

```
>>> re.split(r'\D+', 'Se necesitan 30 azulejos para revestir 1 m2')
['', '30', '1', '2']

>>> re.split(r'\d+', 'Se necesitan 30 azulejos para revestir 1 m2')
['Se necesitan ', ' azulejos para revestir ', ' m', '']

>>> re.sub(r'm2', 'sqm', 'Se necesitan 30 azulejos para revestir 1 m2')
'Se necesitan 30 azulejos para revestir 1 sqm'
```

Editores de RegEx online

Dado que construir expresiones regulares pueden resultar tedioso según la complejidad de la búsqueda, es posible utilizar asistentes que facilitan el testeo de manera interactiva. Existen diversos sitios online que permiten probar expresiones, por ejemplo:

- https://regexr.com/
- https://regex101.com/
- https://www.regextester.com/

Una vez contruída la expresión regular, podremos trasladarla a Python sin inconvenientes.

Otras alternativas

RegEx no es la única manera de buscar de realizar búsquedas. Existen otras librerías de Python que pueden resultar más intuitivas, por ejemplo:

- **pyparsing**: https://github.com/pyparsing/pyparsing/
- **simplematch**: https://github.com/tfeldmann/simplematch
- **kleenexp:** https://github.com/sonoflilit/kleenexp
- **parse**: https://github.com/r1chardj0n3s/parse
- **pygrok**: https://github.com/garyelephant/pygrok

La ventaja de RegEx es su popularidad y soporte en la mayoría de los lenguajes de programación de hoy en día.

Combinaciones de conjuntos de datos

Son operaciones que se realizan entre diferentes datasets para ampliar la información disponible para el análisis. Supongamos que queremos calcular el porcentaje del ingreso que una persona gastaría en promedio en transporte público por provincia. Para realizar el cálculo tenemos una tabla con una muestra de personas de toda Argentina con sus ingresos y provincia, y por otro lado, una tabla con el costo promedio del pasaje en transporte público por provincia. Si unimos la información de estas dos tablas vamos a poder realizar el cálculo deseado.

Tabla de personas

id_persona	ingreso	id_provincia
1	100.000	1
2	150.000	5
3	300.000	8

Tabla de costos

id_provincia	costo_boleto
1	100
2	90

Resultado final

id_persona	ingreso	id_provincia	costo_boleto
1	100.000	1	100
2	150.000	5	102
3	300.000	8	95

Métodos más comunes de combinación de datos

Pandas ofrece varios métodos de combinación de dataframes. Aquí están algunos de los más comunes:

1. `merge()`: El método `merge()` combina dos dataframes basados en una o varias columnas compartidas. Es similar a la operación de "JOIN" en SQL. Puede especificar el tipo de unión (por ejemplo, 'inner', 'outer', 'left' o 'right') y cómo tratar los valores perdidos.
2. `concat()`: El método `concat()` combina dos o más dataframes uno encima del otro o uno al lado del otro a lo largo de un eje determinado (fila o columna). Los dataframes deben tener la misma forma en el eje de concatenación.
3. `join()`: El método `join()` combina dos dataframes basados en el índice de las filas. Es similar a la operación de "JOIN" en SQL. Puede especificar el tipo de unión (por ejemplo, 'inner', 'outer', 'left' o 'right') y cómo tratar los valores perdidos.
4. `merge_ordered()`: El método `merge_ordered()` combina dos dataframes basados en una o varias columnas compartidas y ordena el resultado en función de esas columnas. Es útil cuando se combinan datos temporales, como series de tiempo.
5. `merge_asof()`: El método `merge_asof()` combina dos dataframes basados en una o varias columnas compartidas y las fechas/horas más cercanas. Es útil cuando se combinan datos de series de tiempo que no están perfectamente alineados.

También es posible combinar datos cuya coincidencia no es exacta (difusa). En esta sección veremos también opciones para realizar ese tipo de combinaciones o fusiones de datos.

concat()

La función `concat()` en pandas se utiliza para concatenar dos o más dataframes a lo largo de un eje específico, ya sea horizontal o verticalmente.

La sintaxis básica de la función `concat()` es la siguiente:

```
pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None)
```

Donde:

- `objs`: es una lista de objetos pandas que se desean concatenar.
- `axis`: es el eje a lo largo del cual se desea concatenar los dataframes, 0 para concatenar verticalmente y 1 para concatenar horizontalmente.
- `join`: es el tipo de unión que se desea realizar, puede ser "outer" para una unión externa o "inner" para una unión interna.
- `ignore_index`: es un valor booleano que indica si se desea ignorar los índices originales de los dataframes que se están concatenando.
- `keys`: es una lista de claves que se pueden utilizar para identificar los dataframes originales en el resultado.

Por ejemplo, si se tienen dos dataframes `df1` y `df2` con las mismas columnas y se desea concatenarlos verticalmente, se puede hacer de la siguiente manera:

```
nuevo_df = pd.concat([df1, df2], axis=0)
```

Si los dataframes tienen diferentes columnas, se pueden concatenar horizontalmente utilizando el mismo enfoque:

```
nuevo_df = pd.concat([df1, df2], axis=1)
```

También se puede utilizar la función `concat()` para concatenar más de dos dataframes a la vez. Por ejemplo, si se tienen tres dataframes `df1`, `df2` y `df3` y se desea concatenarlos verticalmente, se puede hacer de la siguiente manera:

```
nuevo_df = pd.concat([df1, df2, df3], axis=0)
```

Aquí se muestra un ejemplo completo de cómo usar la función `concat()` en pandas:

```
import pandas as pd

# Definir dos dataframes de ejemplo
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
```

```
df2 = pd.DataFrame({'A': [4, 5, 6], 'B': [7, 8, 9], 'C': [10, 11, 12]})

# Concatenar los dataframes verticalmente
nuevo_df = pd.concat([df1, df2], axis=0)

# Imprimir el nuevo dataframe
print(nuevo_df)
```

En este ejemplo, se están definiendo dos dataframes de ejemplo, `df1` y `df2`, con diferentes columnas. Luego, se está utilizando la función `concat()` para concatenarlos verticalmente utilizando el parámetro `axis=0`. El resultado de esta operación se está almacenando en un nuevo dataframe llamado `nuevo_df`.

El resultado de la ejecución del código anterior sería el siguiente:

```
   A  B    C
0  1  4  NaN
1  2  5  NaN
2  3  6  NaN
0  4  7  10.0
1  5  8  11.0
2  6  9  12.0
```

Como se puede ver, el nuevo dataframe resultante tiene todas las columnas de ambos dataframes originales, y los valores de índice se han mantenido.

También se puede utilizar la función `concat()` para concatenar los dataframes horizontalmente, como se muestra en el siguiente ejemplo:

```
import pandas as pd

# Definir dos dataframes de ejemplo
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df2 = pd.DataFrame({'C': [7, 8, 9], 'D': [10, 11, 12]})

# Concatenar los dataframes horizontalmente
nuevo_df = pd.concat([df1, df2], axis=1)

# Imprimir el nuevo dataframe
print(nuevo_df)
```

En este ejemplo, se están definiendo dos dataframes de ejemplo, `df1` y `df2`, con diferentes columnas. Luego, se está utilizando la función `concat()` para concatenarlos horizontalmente utilizando el parámetro `axis=1`. El resultado de esta operación se está almacenando en un nuevo dataframe llamado `nuevo_df`.

El resultado de la ejecución del código anterior sería el siguiente:

```
   A  B  C   D
0  1  4  7  10
1  2  5  8  11
2  3  6  9  12
```

Como se puede ver, el nuevo dataframe resultante tiene todas las filas de ambos dataframes originales, y las columnas se han combinado en función de sus nombres.

merge()

Nos permite unir dos o más datasets que comparten una o más dimensiones. Por ejemplo, supongamos que como resultado de una encuesta de hogares obtenemos una tabla del hogar con las variables `id_hogar` y `barrio` y una tabla de personas con las variables `id_persona`, `motivo_viaje`, `género` e `id_hogar`, como se muestra abajo.

Tabla de personas

id_persona	motivo_viaje	género	id_hogar
3449	trabajo	femenino	450956
3450	no_trabajo	masculino	450956

Tabla de hogares

id_hogar	barrio

450956	Centro
450957	Lourdes

Supongamos que queremos calcular la cantidad de viajes de trabajo por barrio. Para eso, necesitamos saber a qué barrio pertenece cada persona y podemos averiguarlo usando el `id_hogar` que se encuentra en ambas tablas. Por ejemplo, la persona 3449 pertenece al hogar 450956 el cuál pertenece al barrio Centro. De esta manera la tabla personas pasa a tener la siguiente forma:

id_persona	motivo_viaje	género	id_hogar	barrio
3449	trabajo	femenino	450956	Centro
3450	no_trabajo	masculino	450956	Centro

Pandas tiene la función `merge` que nos permite realizar estas operaciones de forma muy similar a como se realizan en SQL, pueden consultar la documentación completa aquí. A continuación mostramos un ejemplo:

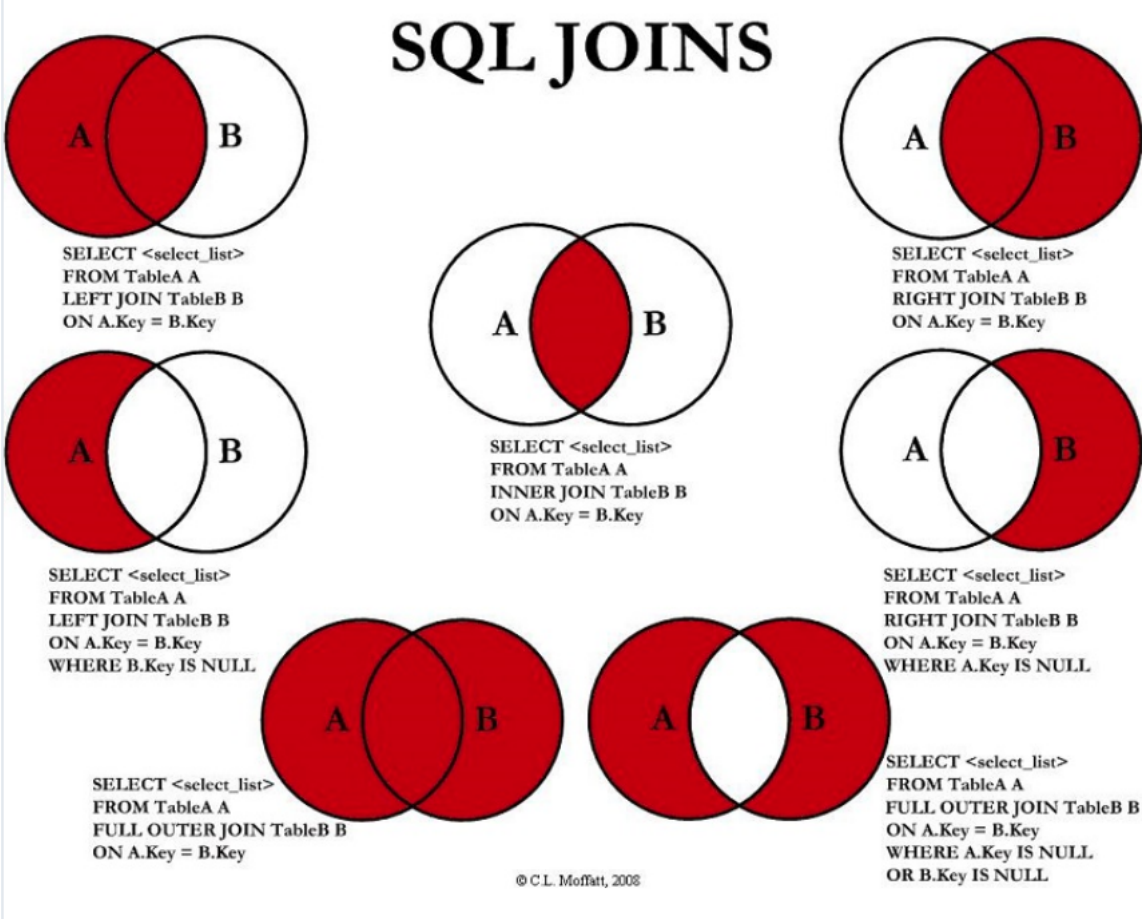
```
import pandas as pd

personas = pd.read_csv("datos_personas.csv")
hogares = pd.read_csv("datos_hogares.csv")

resultado = pd.merge(personas, hogares, left_on = 'id_hogar', right_on = 'id_hogar', how = 'left')

id_persona    motivo_viaje   género     id_hogar  barrio
3449          trabajo        femenino   450956    Centro
3450          no_trabajo     masculino  450956    Centro
```

En nuestro ejemplo se dio el caso de que fue una unión de muchos a uno; la tabla de personas tiene muchas filas con la misma provincia, mientras que la tabla de provincias, como es de esperarse, solo tiene una fila por cada provincia.

El parámetro `how` nos va a indicar cómo queremos que se realice la unión y definirlo resulta de gran importancia y depende del análisis que se esté realizando. Las posibilidades son:

- left: usa solamente las keys del DataFrame izquierdo
- right: usa solamente las keys del DataFrame derecho
- outer: usa la unión de las keys de ambos DataFrames
- inner: usa la intersección de las keys de ambos DataFrames
- cross: calcula el producto cartesiano de ambos DataFrames

A la hora de definir `how` pueden consultar el siguiente gráfico para saber qué es lo que necesitan.



join()

La función `join()` en Pandas se utiliza para combinar dos o más dataframes *en función de sus índices o columnas*. La sintaxis básica de la función `join()` es la siguiente:

```
pd.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False)
```

Donde:

- `other`: es el dataframe con el que se desea combinar.
- `on`: es el nombre de la columna (o índice) en la que se desea realizar la unión.
- `how`: es el tipo de unión que se desea realizar, puede ser "left", "right", "outer" o "inner".
- `lsuffix`: sufijo a agregar a los nombres de las columnas del dataframe original (izquierdo) en caso de conflictos.
- `rsuffix`: sufijo a agregar a los nombres de las columnas del dataframe que se está uniendo (derecho) en caso de conflictos.
- `sort`: es un valor booleano que indica si se deben ordenar los índices antes de realizar la unión.

A continuación se muestra un ejemplo de cómo utilizar la función `join()` en Pandas:

```
import pandas as pd

# Definir dos dataframes de ejemplo
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['a', 'b', 'c'])
df2 = pd.DataFrame({'C': [7, 8, 9], 'D': [10, 11, 12]}, index=['a', 'b', 'c'])

# Unir los dataframes utilizando la función join()
nuevo_df = df1.join(df2)
```

```
# Imprimir el nuevo dataframe
print(nuevo_df)
```

En este ejemplo, se están definiendo dos dataframes de ejemplo (`df1` y `df2`) con el mismo índice. Luego, se está utilizando la función `join()` para combinar los dos dataframes en función de sus índices comunes. El resultado de esta operación se está almacenando en un nuevo dataframe llamado `nuevo_df`.

El resultado de la ejecución del código anterior sería el siguiente:

```
   A  B  C   D
a  1  4  7  10
b  2  5  8  11
c  3  6  9  12
```

Como se puede ver, el nuevo dataframe resultante tiene todas las columnas de ambos dataframes originales, y las filas se han combinado en función de sus índices comunes.

También se puede utilizar la función `join()` para combinar dataframes en función de columnas específicas. En este caso, se especifica la columna en la que se desea realizar la unión utilizando el parámetro `on`. Por ejemplo:

```
import pandas as pd

# Definir dos dataframes de ejemplo
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})
df2 = pd.DataFrame({'D': [4, 5, 6], 'C': [7, 8, 9], 'E': [10, 11, 12]})

# Unir los dataframes utilizando la función join()
nuevo_df = df1.join(df2.set_index('C'), on='C')

# Imprimir el nuevo dataframe
print(nuevo_df)
```

En este ejemplo, se están definiendo dos dataframes de ejemplo (`df1` y `df2`) con una columna en común (`C`). Luego, se está utilizando la función `join()` para combinar los dos dataframes en función de esa columna. El resultado de esta operación se está almacenando en un nuevo dataframe llamado `nuevo_df`.

El resultado de la ejecución del código anterior sería el siguiente:

```
   A  B  C  D  E
0  1  4  7  4  10
1  2  5  8  5  11
2  3  6  9  6  12
```

Como se puede ver, el nuevo dataframe resultante tiene todas las columnas de ambos dataframes originales, y las filas se han combinado en función de sus valores comunes en la columna `C`.

Uniones cruzadas (Cross Joins)

El método `merge()` también permite realizar una unión cruzada (cross join), que devuelve todas las combinaciones posibles entre los registros de dos dataframes, independientemente de si los valores coinciden o no. Para realizar una unión cruzada, se puede establecer el parámetro `how` en "cross".

A continuación se muestra un ejemplo de cómo realizar una unión cruzada entre dos dataframes utilizando el método `merge()` con el parámetro `how='cross'`:

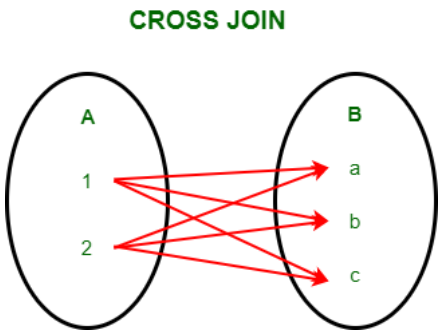
```
import pandas as pd

# Definir dos dataframes de ejemplo
df1 = pd.DataFrame({'A': [1, 2]})
df2 = pd.DataFrame({'B': ['a', 'b', 'c']})

# Realizar una unión cruzada entre los dataframes
nuevo_df = pd.merge(df1, df2, how='cross')

# Imprimir el nuevo dataframe resultante
print(nuevo_df)
```

En este ejemplo, se están definiendo dos dataframes de ejemplo (`df1` y `df2`) con una sola columna cada uno. Luego, se está utilizando el método `merge()` para realizar una unión cruzada entre los dos dataframes. El resultado se está almacenando en un nuevo dataframe llamado `nuevo_df`.

El resultado de la ejecución del código anterior sería el siguiente:

```
   A  B
0  1  a
1  1  b
2  1  c
3  2  a
4  2  b
5  2  c
```

Como se puede ver, el nuevo dataframe resultante tiene todas las combinaciones posibles entre los valores de ambas tablas, sin importar si los valores coinciden o no. En este caso, el dataframe resultante tiene una columna `A` y una columna `B`, y tiene todas las combinaciones posibles entre los valores de las dos tablas.



Es importante tener en cuenta que realizar una unión cruzada puede generar un dataframe muy grande si los dataframes originales son grandes. Por lo tanto, se debe tener cuidado al utilizar esta técnica y asegurarse de que sea realmente necesaria para el análisis que se está realizando.

Supongamos que se tiene una tienda en línea que vende productos electrónicos y se quiere realizar un análisis para conocer todas las posibles combinaciones de productos que se pueden vender juntos en un paquete promocional. Para ello, se tiene un DataFrame con una lista de productos electrónicos y otro DataFrame con una lista de accesorios.

Se puede utilizar un cross join para generar un DataFrame con todas las posibles combinaciones de productos y accesorios. Por ejemplo:

```
import pandas as pd

# Crear DataFrame de productos electrónicos
productos_electronicos = pd.DataFrame({'Producto': ['Laptop', 'Smartphone', 'Tablet']})

# Crear DataFrame de accesorios
accesorios = pd.DataFrame({'Accesorio': ['Cargador', 'Auriculares', 'Estuche']})

# Realizar cross join
combinaciones = pd.merge(productos_electronicos.assign(key=0), accesorios.assign(key=0), on='key').drop('key', axis=1)

# Mostrar DataFrame resultante
print(combinaciones)
```

La salida del código anterior sería:

```
     Producto    Accesorio
0      Laptop     Cargador
1      Laptop  Auriculares
2      Laptop      Estuche
3  Smartphone     Cargador
4  Smartphone  Auriculares
5  Smartphone      Estuche
6      Tablet     Cargador
7      Tablet  Auriculares
8      Tablet      Estuche
```

En este ejemplo, se ha utilizado un cross join para generar todas las posibles combinaciones de productos electrónicos y accesorios que se podrían ofrecer juntos en un paquete promocional. Esto podría ayudar a identificar combinaciones de productos y accesorios que se venden bien juntos y a diseñar paquetes promocionales efectivos para los clientes.

Enlace difuso de registros (Fuzzy joins)

Se refiere a la idea de unir dos o más registros que probablemente hacen referencia a la misma entidad y es parte del proceso de fusión de datos. Esto puede ser realizado entre dos registros de dos tablas diferentes o entre registros de una misma tabla para identificar duplicados. En la sección anterior vimos como unir dos o más datasets utilizando un atributo en común y dimos por sentado que ese atributo estaba limpio y completo para todos los datasets. Sin embargo, esto puede no ser así.

Enlace determinístico de registros

Es la forma más sencilla de enlazar registros y puede ser realizada de diferentes formas:

- Coincidencia de variables: para que dos registros sean considerados el mismo, debe haber una coincidencia entre una o más variables. En los casos de arriba, existía un identificador único entre los datasets que podía ser utilizado como la variable de enlace. Otras veces, no existe un identificador único y el mismo debe ser sustituido por una combinación de variables, por ejemplo nombre, apellido, edad, lugar de nacimiento, etc.
- Basado en reglas: Dos o más registros son considerados el mismo o no de acuerdo a un grupo de reglas. Por ejemplo, usamos el DNI para identificar registros únicos de personas pero en algunos registros este dato está en falta. En estos casos, usamos nombre, apellido, dirección, etc., para identificar los registros únicos.

Dataset A

dni	nombre	apellido	direccion	ingreso
48756980	Pedro	Lopez	Ugarte 17, Rosario, Santa Fe	120.000
45567483	Rosario	Ledesma	Rivadavia 16, Rosario, Santa Fe	200.000

Dataset B

dni	nombre	apellido	direccion	beca_transporte
48756980	Pedro	Lopez	Ugarte 17, Rosario, Santa Fe	si
	Rosario	Ledesma	Rivadavia 16, Rosario, Santa Fe	no

Enlace probabilístico de registros

(Extraído del libro “Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection”, consultar el libro para mayor información)

La idea del enlace probabilístico de registros fue introducida por Newcombe et al. en 1959. Ellos reconocen que en la ausencia de identificadores únicos de entidades los otros atributos disponibles en los datasets tienen que ser usados para matchear los registros. Como los valores de los atributos pueden ser erróneos, faltantes, o sin datos y porque la diferencia de los valores y sus distribuciones pueden diferir entre atributos, se debe asignar diferentes pesos a cada atributo cuando estos son usados para calcular la similaridad entre registros.

Primero, se calcula una vector de comparación γ entre el par-registro r conformado por s_1, s_2. γ es un vector de ceros y unos que indica si los atributos que se están comparando coinciden o no. Tiene K componentes, siendo K el número de atributos usado en la comparación. Por ejemplo:

registro	nombre	apellido	dirección	edad
1	Carlos	Gonzalez	Laprida 155	78
2	Carlos	Gonzalez	Laprida 156	78
γ	1	1	0	1

Luego, se calcula cuál es la probabilidad de que ocurra γ siendo que el par r es un match, es decir pertenece al conjunto *M*, y cuál es la probabilidad de que ocurra γ siendo que el par r es un no-match, es decir pertenece al conjunto *U*. En el ejemplo de arriba, la probabilidad de que ocurra γ siendo un match es bastante alta siendo que el único atributo en el que no hay coincidencia es en la dirección la cual puede fallar debido a los métodos de recolección.

Luego, usando las dos probabilidades calculadas arriba se obtiene el siguiente ratio:

$$R = \frac{\text{probabilidad de } \gamma \text{ dado que r pertenece M}}{\text{probabilidad de } \gamma \text{ dado que r pertenece U}}$$

Finalmente, se utiliza la siguiente regla de decisión para saber si el par de registros r es un match o no.

$$R \geq tu \implies r \rightarrow Match$$

$$t_l < R < tu \implies r \rightarrow \text{Match Potencial}$$

$$R < tu \implies r \rightarrow \text{No-match}$$

Este algoritmo puede mejorarse si en lugar de construir el vector de comparación y con coincidencias/no-coincidencias usamos una comparación aproximada de los atributos. Esto lo podemos realizar usando la similaridad de Jaro-Winkler o de Levensthein. Pueden consultar: Porter & Winkler (1997) o Winkler (1990).

Distancia de Jaro-Winkler

La similaridad de Jaro es 0 cuando no hay ninguna coincidencia entre las cadenas o está definida como la fórmula de abajo:

$$sim_j = \frac{1}{3}(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m})$$

Donde:

s_1, s_2 son la longitud de la cadena 1 y 2 que se están comparando

m es el número de caracteres que coinciden.

t es el número de caracteres que se transponen, de los que coinciden

Se considera que dos caracteres coinciden si no se encuentran más alejados que $\lfloor \frac{max(s_1,s_2)}{2} \rfloor$

La similaridad de Jaro es 0 cuando no hay coincidencia alguna o es 1 cuando la coincidencia es perfecta. Para entenderlo mejor, veamos un ejemplo:

A	R	G	E	N	T	I	N
A	R	G	E	N	T	E	N
						x	

En nuestro caso de arriba s1 y s2 son iguales a 9, m es igual a 8 y t es igual a 1. Por lo tanto:

$$sim_j = \frac{1}{3}(\frac{8}{9} + \frac{8}{9} + \frac{8-1}{8}) = 0.8843$$

Luego, Winkler propone una modificación a esta fórmula donde aumenta el rating de similaridad si las cadenas comparadas comienzan con el mismo prefijo.

$$sim_{j-w} = sim_j + lp(1 - sim_j)$$

Donde:

sim_j es la similaridad de jaro entre las cadenas s_1, s_2

l es la cantidad de caracteres que coinciden en el prefijo, hasta un máximo de 4 caracteres. En este caso la coincidencia debe ser perfecta y no admite que los caracteres estén transpuestos

p es un factor que indica cuánto se ajusta por la coincidencia en el prefijo, no puede ser mayor a 0.25. El valor estándar de p es 0.1

En nuestro ejemplo la similarida de Jaro-Winkler se calcula como:

$$sim_{j-w} = sim_j + 3 * 0.1(1 - sim_j) = 0.9555555555555556$$

💡 La distancia de Jaro-Winkler se calcula como $d_{j-w} = 1 - sim_{j-w}$

En Python, podemos utilizar la librería `jaro` para calcularla, por ejemplo:

```
# Podemos instalar la librería con: pip install jaro-winkler
# https://github.com/richmilne/JaroWinkler

import jaro

dist = jaro.jaro_winkler_metric('ARGENTINA', 'ARGENTENA')

print(dist) # Imprime 0.9555555555555556
```

Distancia de Levenshtein o distancia de edición

La distancia de edición cuenta la menor cantidad de operaciones de edición que son necesarias para convertir a una cadena en otra. La distancia de edición de **Levenshtein** es la más básica y está definida como el menor número de operaciones de inserción, borrado o sustitución en un solo caracter que son necesarias para convertir la cadena s_1 en la cadena s_2. Por ejemplo:

- casa → cosa: 1
- hola → ola: 1
- estrella → estela: 2

La similaridad de Levenshtein se puede calcular entonces con la siguiente fórmula

$$sim_{levenshtein} = 1 - \frac{dist_{levenshtein}(s_1, s_2)}{max(|s_1|, |s_2|)}$$

En Python, podemos utilizar la librería `levenshtein` para calcularla, por ejemplo:

```
# Podemos instalar la librería con: pip install levenshtein
# https://github.com/maxbachmann/python-Levenshtein

from Levenshtein import distance

dist = distance("lewenstein", "levenshtein")
print(dist) # Imprime 2
```

También la librería `thefuzz` (anteriormente `fuzzywuzzy`) permite trabajar con distancias de Levenshtein: https://github.com/seatgeek/thefuzz

Volviendo a nuestro ejemplo, si ahora al vector de comparación lo calculamos usando alguna de estas similaridades vamos a obtener lo siguiente:

registro	nombre	apellido	dirección	edad	SimSum (sin pesos)
1	Carlos	Gonzalez	Laprida 155	78	
2	Carlos	Gonzalez	Laprida 156	78	
γ	1	1	0	1	3
γ Jaro-Winkler	1	1	0.963	1	3.96
γ Levnshtein	1	1	0.909	1	3.909

Utilizando estas herramientas en Python, podríamos fusionar Dataframes cuando tenemos datos “difusos” entre si, pero que se refieren a lo mismo. Por ejemplo, utilizando la librería `thefuzz` podemos hacer algo como lo siguiente:

```
import pandas as pd
from thefuzz import fuzz  # Instalar con pip install thefuzz[speedup]

# Creamos el primer DataFrame con datos personales de alumnos
df1 = pd.DataFrame({'Nombre': ['Juan Pérez', 'María González', 'Luisa Martínez'],
                    'Edad': [21, 19, 20],
                    'Email': ['juan@gmail.com', 'maria@gmail.com', 'luisa@gmail.com']})

# Creamos el segundo DataFrame con datos de notas de evaluaciones
df2 = pd.DataFrame({'Nombre': ['Juan Perez', 'María Gonzales', 'Luisa Martínz'],
                    'Nota_Parcial': [8.5, 7.5, 9],
                    'Nota_Final': [9, 8, 9.5]})
```

```
# Realizamos un fuzzy match para unir las dos tablas por el campo "Nombre"
for index, row in df2.iterrows():
    name = row['Nombre']
    max_score = -1
    max_name = ""
    # Aquí iteramos para encontrar el mejor puntaje de similitud con el df1
    for index2, row2 in df1.iterrows():
        score = fuzz.token_set_ratio(name, row2['Nombre'])
        if score > max_score:
            max_score = score
            max_name = row2['Nombre']
    if max_score > 70:
        # Reemplazamos el nombre actual por el nombre de mejor puntaje
        df2.at[index, 'Nombre'] = max_name
    else:
        df2.at[index, 'Nombre'] = ""

# Unimos los dos DataFrames
df_final = pd.merge(df1, df2, on='Nombre')

# Mostramos el DataFrame final con los campos requeridos
print(df_final[['Nombre', 'Edad', 'Email', 'Nota_Parcial', 'Nota_Final']])
```

Y obtenemos finalmente un Dataframe como resultado de la fusión:

```
Nombre              Edad    Email             Nota_Parcial  Nota_Final
0      Juan Pérez    18   juan@gmail.com           8.5         9.0
1  María González    19  maria@gmail.com           7.5         8.0
2  Luisa Martínez    20  luisa@gmail.com           9.0         9.5
```

En este ejemplo, hemos utilizado la librería `thefuzz` para realizar el fuzzy match entre los nombres de ambos DataFrames. En este caso, hemos utilizado el algoritmo `token_set_ratio`
que compara dos strings y devuelve un score en base a la similitud entre las palabras que los conforman. Hemos establecido un umbral de similitud del 70% para considerar que dos nombres se refieren a la misma persona. Después de realizar el fuzzy match, hemos unido los dos DataFrames utilizando la función `merge` de pandas.

> 💡 Al fusionar Dataframes por nombre de personas, ciudades, etc., debe considerarse la posibilidad de que dos personas o ciudades diferentes se llamen del mismo modo. En ese caso habría que desarrollar otra estrategia para la fusión de los Dataframes, quizá utilizando más campos para evitar coincidencias incorrectas (por ejemplo, país o email).