Grado en Gestión de la Información y Contenidos Digitales

Procesamiento y Visualización de Datos

Tema 1. Introducción al procesamiento de datos

Bloque 1: Procesamiento de datos

UNIVERSIDAD DE MURCIA

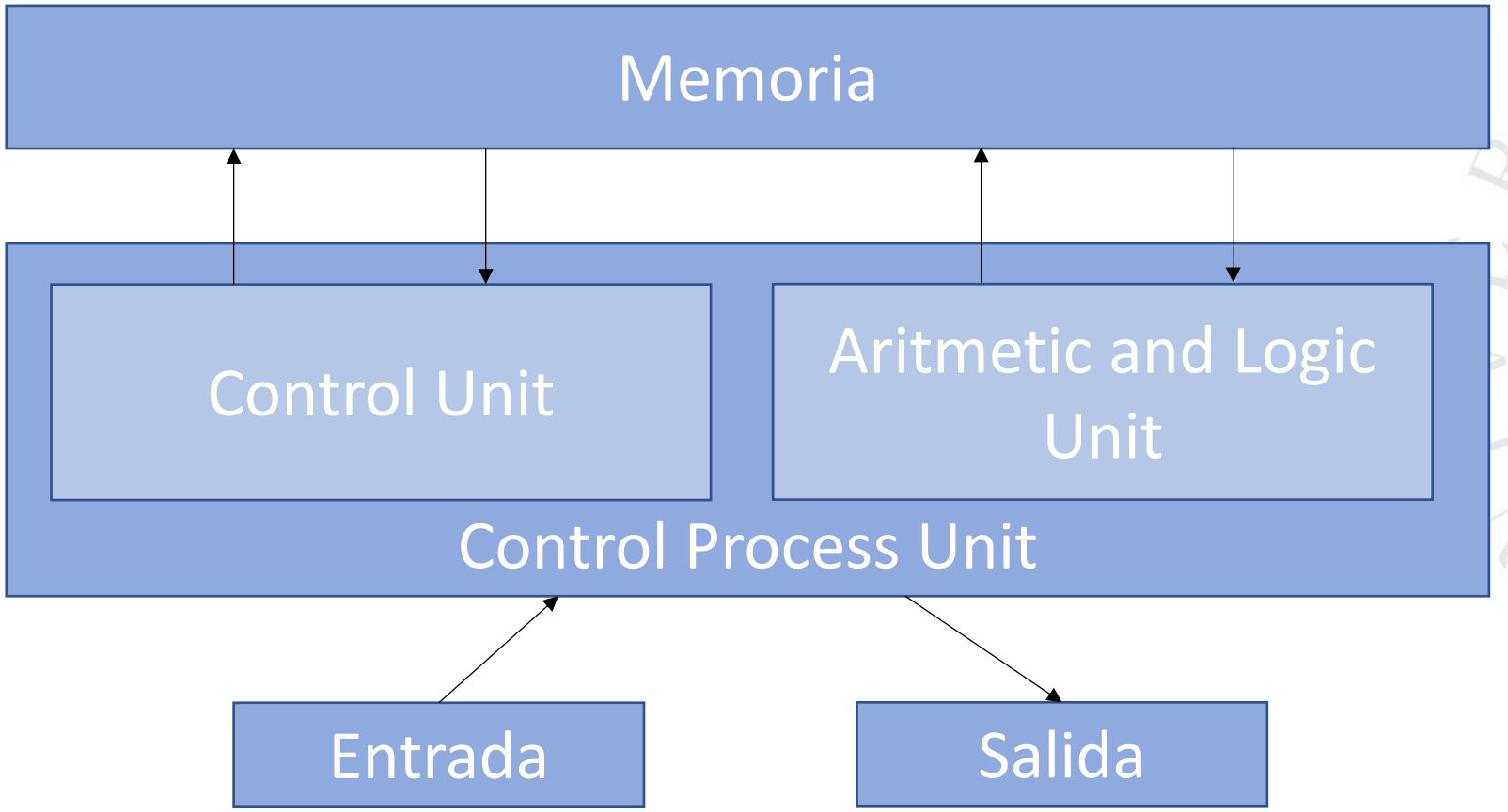
Facultad de Informática

¿Qué hace un ordenador?

- La función principal de un ordenador es ejecutar programas informáticos
- ¿Qué es un programa informático?
  - Conjunto de instrucciones ejecutadas de forma secuencial que leen datos de la memoria, opera con ellos y almacena los resultados en la memoria
- Programa más simple: Hola Mundo

```
if __name__=="__main__":
  print("hola mundo")
  # hola mundo
```

¿Qué hace un ordenador?



¿Que es un algoritmo?

- Secuencia de pasos sencillos
- Mecanismos de control de flujo que especifican cuando es ejecutado cada paso
- Una forma de poner fin al proceso

- Ejemplo: multiplicación

```
if __name__=="__main__":
    a = 4*4
    print(a)
    # 16
```

```
def multiplica(a,b):
    resultado = 0
    for i in range(b):
        resultado += a
    return resultado

if __name__=="__main__":
    a = multiplica(4,4)
    print(a)
    # 16
```

UNIVERSIDAD DE MURCIA

Lenguajes de Programación

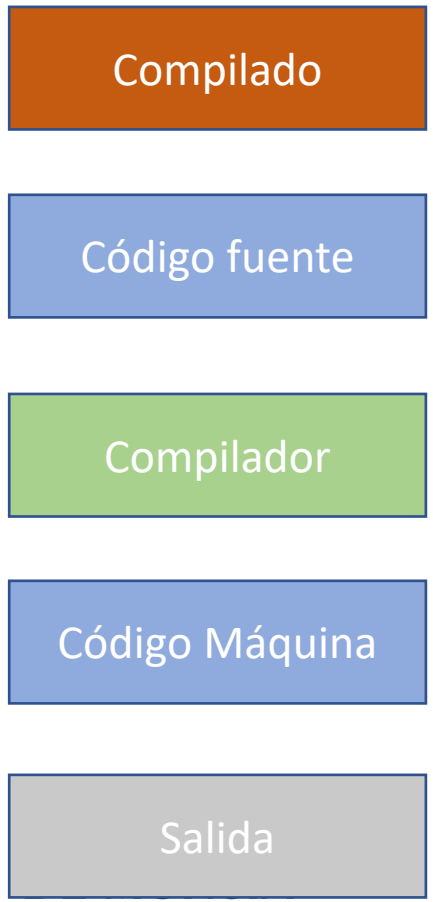
- Un programa/algoritmo se implementa con la ayuda de un lenguaje de programación.
  - C/C++ C# PHP
  - Java Python Javascript
- Los lenguajes de programación proporcionan un conjunto de operaciones primitivas
- Además, muchos de los lenguajes disponen de librerías que proporcionan funcionalidad ya programada

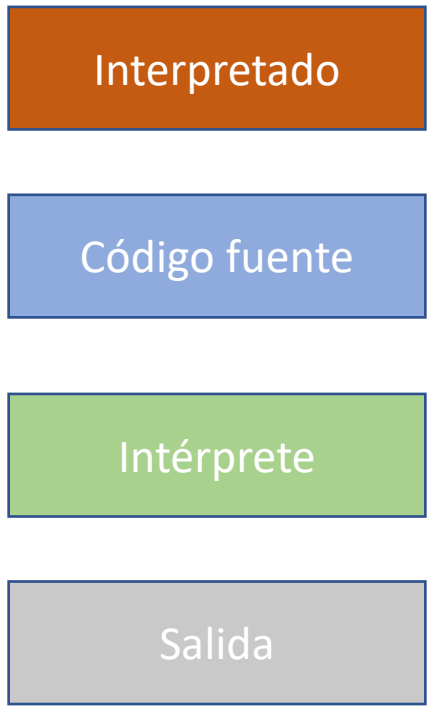
Lenguajes de Programación

- En esta asignatura usaremos el lenguaje de programación Python que tiene las siguientes características:
  - Código Libre
  - Portable entre plataformas (Windows, Linux, Mac)
  - Fácil de aprender
  - Tipado dinámico
  - Lenguaje de alto nivel
  - Lenguaje interpretado (no hay necesidad de compilar)
  - Abundantes librerías para procesamiento de datos: Numpy, Matplotlib, Pandas, Jupyter ...

Lenguajes de Programación

- Lenguaje interpretado vs Lenguaje Compilado





Procesamiento de datos

- La información se define como datos clasificados u organizados que tienen un significado útil para el usuario.
- Se considera procesamiento de datos a cualquier operación realizada sobre los datos (almacenamiento, adaptación, modificación, etc) para convertirla en información útil.

- Podemos diferenciar tres partes: entrada, procesamiento y salida.
- A esto se le conoce como ciclo del procesado de datos

¿Qué son los datos?

- Los datos son cualquier colección de cantidades, caracteres o símbolos sobre los que los ordenadores ejecutan sus operaciones.
- Los datos también son hechos contrastables de un determinado ámbito. Por ejemplo: edad y altura de una persona.
- Podemos diferenciar distintos tipos de datos
  - Numéricos
  - Cadenas
  - Imágenes
  - Audio
  - ...

Ciclo de procesamiento de datos

- **Entrada**: Es el proceso mediante el cual los datos son transformados en una forma de representación adecuada para que puedan ser usados dentro de un ordenador. Este paso se divide en: verificación, codificación y almacenamiento
  - **Verificación**: Los datos recolectados se verifican para comprobar si son correctos
  - **Codificación**: Los datos verificados son codificados a un formato que una máquina puede procesar
  - **Almacenamiento:** Los datos son almacenados en un archivo dentro de un dispositivo de almacenamiento (Disco duro, USB) y éste será la entrada para el procesamiento de datos

Ciclo de procesamiento de datos

- **Procesamiento**: Este término engloba las técnicas de manipulación de los datos tales como: clasificación, almacenamiento, cálculo y resumen
  - **Clasificación:** Los datos son clasificados en distintos grupos o subgrupos de tal forma que éstos puedan ser manejados de forma independiente
  - **Almacenamiento:** Los datos son almacenados de tal forma que puedan ser accedidos de forma rápida.
  - **Cálculo:** Las operaciones aritméticas llevadas a cabo sobre los datos para obtener el resultado deseado
  - **Resumen:** Los datos son procesados para mostrarlos de forma resumida.

Ciclo de procesamiento de datos

- **Salida**: Son los datos una vez procesados, información básicamente. En general esta información será almacenada para su acceso y uso posterior. Sobre esta salida podemos realizar las siguientes tareas
  - **Recuperación:** La información almacenada puede ser accedida en cualquier momento que sea requerido
  - **Conversión:** La salida puede ser convertida a distintos formatos
  - **Comunicación:** La salida puede ser comunicada a distintas personas y/o distintas áreas de la empresa

Tipos de procesamiento de datos

- **Procesamiento de datos manual**. Esto implica la intervención humana y por tanto este método lleva consigo una alta tasa de cometer errores.
- **Procesamiento de datos electrónico**. Este tipo de procesamiento es el llevado por un ordenador y los programas que se ejecutan sobre él.
- **Procesamiento de los datos en tiempo real**. En este tipo de procesamiento, hay un flujo continuo de entrada que debe ser procesado para ir generando una salida. Además, los datos deben ser procesados en un tiempo acotado, de lo contrario el sistema no ofrecería los requisitos de tiempo real.
- **Procesamiento por lotes**. En este caso, un conjunto de datos son procesados todos al mismo tiempo.

Ejercicios

- En relación a los lenguajes de programación comentados anteriormente
  - C/C++
  - Java
  - C#
  - Python
  - PHP
  - Javascript

1. ¿Cuáles serían lenguajes interpretados y cuales lenguajes compilados?
2. ¿Qué otros lenguajes de programación existen y en que campos se usan habitualmente?. ¿Cuáles son las características principales de cada uno de esos lenguajes?
3. En relación en tiempo de ejecución de un programa, ¿Cuál es el que otorgaría mejor rendimiento en términos generales?. Justifica la respuesta

Bibliografía

- Grus, J. (2019). *Data science from scratch: first principles with python*. O'Reilly Media, 978-1492041139.
- Osborne, J. (2012). Best Practices in Data Cleaning: A Complete Guide to Everything You Need to Do Before and After Collecting Your Data, SAGE Publications, 978-1412988018.
- Mertz, D. (2021). *Cleaning Data for Effective Data Science: Doing the other 80% of the work with Python, R, and command-line tools*. Packt Publishing Ltd.

Grado en Gestión de la Información y Contenidos Digitales

Procesamiento y Visualización de Datos

Tema 2. Conjuntos de Datos

Bloque 1: Procesamiento de datos

UNIVERSIDAD DE MURCIA

Facultad de Informática

Introducción

- Los conjuntos de datos son una serie de **datos agrupados** que **comparten** una serie de **características**
- Conjuntos de datos de: imágenes, audio, texto, datos tabulares, etc.
- En esta asignatura trabajaremos con datos tabulares
  - Podemos verlo como una tabla con datos
- Cada fila es una **muestra** y las columnas son llamadas **características** y contienen datos concretos de la muestra

Introducción

Nombre	Edad	Peso	Altura	Estudios	Salario
Juan	22	62	168	Secundaria	20.000
Angela	26	60	165	Universitarios	30.000
Antonio	23	70	174	Formación Profesional	25.000
Sofia	45	68	164	Bachiller	27.000
Eva	30	67	170	Formación Profesional	30.000
Ramón	48	70	169	Secundaria	28.000
José	65	85	178	Secundaria	26.000
María	55	74	175	Secundaria	27.000

Muestra

Característica

Introducción

- Los conjuntos de datos son **costosos** de conseguir
- Existen muchos de ellos disponibles en Internet
- Se utilizan para **obtener información** sobre un área concreta y en muchos casos para **realizar predicciones** sobre datos no vistos anteriormente
  - ¿Qué producto es el que más se vende en una tienda?
  - ¿Qué hora es la de mayor afluencia?
  - ¿Cuándo vale un piso de determinadas características en un área geográfica concreta?
  - Detección de Anomalías en entornos financieros y redes de datos

Tipos de datos

- Los conjuntos de datos están compuestos por datos. Podemos encontrar datos **estructurados** y datos **no estructurados**
- Los datos estructurados son aquellos que están perfectamente identificados

Observación	Altura (cm)	Peso (kg)
1	165	61.3
2	167	63.5
3	170	68.3

- Los datos estructurados **no tienen porque ser siempre numéricos**

Tipos de datos

- A continuación se muestra una lista no exhaustiva de los distintos tipos de datos que podemos encontrar

Nombre	Descripción	Ejemplo
Numérico	Valores enteros y decimales	Edad, peso, salario, precio de un producto
Texto	Secuencia de caracteres que forman una cadena con sentido	Tipo de casa, nombre de persona, puesto laboral, descripción de un producto
Booleano	Valores que indican Verdadero o Falso	Estado de un pulsador, desempleado,
Categóricos	Valores que determinan el tipo de un dato dentro de un conjunto disponible	Estado civil, sexo, tipo de casa, nivel de estudios

Tipos de datos

- Los datos no estructurados son aquellos que no están precisamente identificados
- Ejemplo: La persona identificada medía entre 160 y 170 cm. Con respecto a su peso, éste se encontraba entre 60 y 70 kg.
- Los humanos no tenemos problema en entender este tipo de enunciados
- Sin embargo, los programas y algoritmos si que tienen problemas con este tipo de datos
- **En esta asignatura vamos a centrarnos fundamentalmente en datos estructurados**

Tipos de conjuntos de datos

- Podemos distinguir dos grandes problemas en los que son usados los conjuntos de datos
- **Problemas de regresión**: Se utiliza el conjunto de datos para hacer una **predicción sobre un valor continuo**. Por ejemplo: valor de una casa con unas características concretas. Cada muestra del conjunto de datos está etiquetada con un valor continuo
- **Problemas de clasificación**: Se utiliza el conjunto de datos para **clasificar muestras** dependiendo de la clase a la que pertenezca. Por ejemplo: clasificar una imagen según si aparece un perro o un gato. Cada muestra del conjunto de datos pertenece a una clase distinta

Tipos de conjuntos de datos

- Los conjuntos de datos son **costosos** de conseguir
- En muchos casos se necesita un gran esfuerzos en tareas relacionadas con el **procesado**, **limpieza** y **agrupación** de datos
- La mayoría de conjuntos de datos contienen una característica especial que **etiqueta** la muestra
- Para poder etiquetar cada una de las muestras es necesario la **intervención de un experto**
- Podemos diferenciar tres tipos de conjuntos de datos: **etiquetados, no etiquetados, parcialmente etiquetados**

Tipos de conjuntos de datos

- Los conjuntos de datos **completamente etiquetados** son aquellos en los que todas las muestras tienen una **etiqueta**
- La etiqueta se utilizará para entrenar modelos de Inteligencia Artificial en problemas de clasificación o regresión
- Para desarrollar estos conjuntos de datos es necesario la **participación de un experto** y, por tanto, estos conjuntos de datos son los más costosos de conseguir

Tipos de conjuntos de datos

- Los conjuntos de datos **no etiquetados** son los más sencillos de obtener
- Al no disponer de etiqueta, **tampoco requieren de la intervención de un experto**
- Las tareas sobre estos conjuntos se basan en **obtener patrones y relaciones ocultas** en ellos u **obtener información**
- Una tarea frecuente sobre estos conjuntos de datos es realizar clustering.
  - Intentar identificar clases según la similitud entre las muestras del conjunto de datos

Tipos de conjuntos de datos

- Los conjuntos de datos **parcialmente etiquetados** están a medio camino entre los dos anteriores
- Básicamente se utilizan para tareas de **detección de anomalías** en distintos escenarios
- Un ejemplo de estos conjuntos de datos sería aquél que almacene las transacciones de tarjetas de créditos de distintos usuarios. Estas muestras **se consideran implícitamente normales**. A continuación, se utilizan técnicas de Inteligencia Artificial para **delimitar una frontera** que agrupe todas estas muestras. **Todo lo que salga de dicha frontera se considerará anómalo**

Conjuntos de datos abiertos

- Algunos gobiernos como el de Estados Unidos han dado órdenes a los organismos dependientes para gestionar los datos que generen, y más ampliamente la información, como un **recursos** desde el principio
- También aconsejan poner a **disposición pública** los datos generados siempre y cuando sea posible
- Con el fin de hacerlos **abiertos, visibles y usables**
- En base a esto, la **orden M-13-3** de Estados Unidos presenta 7 principios asociados al paradigma de datos abiertos

Conjuntos de datos

- **Público**. Las agencias deben adoptar el paradigma de datos abiertos, teniendo en cuenta las **restricciones** de **privacidad, confidencialidad y seguridad**
- **Accesible**. Los datos deben estar disponibles en **formatos abiertos y modificables** que permitan la **extracción, descarga, indexación y búsqueda** de los mismos. Los formatos deben ser aptos para que sean leídos por una máquina
- **Descritos**. Los datos abiertos deben estar descritos completamente para que los usuarios tengan suficiente información para comprender sus fortalezas y debilidades

Conjuntos de datos

- **Reusables**. Los datos abiertos deben hacerse públicos mediante **licencias abiertas** que **no restrinjan** su uso
- **Completos**. Los datos abiertos deben hacerse disponibles en **forma cruda, primaria y sin procesar**. Los datos agregados también pueden hacerse disponibles pero se debe referenciar a los datos en crudos
- **Inmediatez**. Los datos abiertos deben hacerse disponibles **tan pronto como sea necesario** para preservar el valor de los mismos
- **Gestión posterior**. Un punto de contacto debe establecerse para solucionar cualquier problema con los datos y responder a las quejas sobre el cumplimiento de estos principios

Almacenamiento de los datos

- Es frecuente encontrar los datos en bases de datos en vez de en ficheros creados para su distribución

- Un **modelo de datos** es una colección de conceptos que **describen** como se **representan y acceden** los datos
  - **Estructura**. Espacio utilizado para el almacenamiento de los datos. Hasta ahora hemos asumido tablas formadas por columnas y filas
  - **Semántica**. Se refiere a los valores, las características y las muestras almacenadas

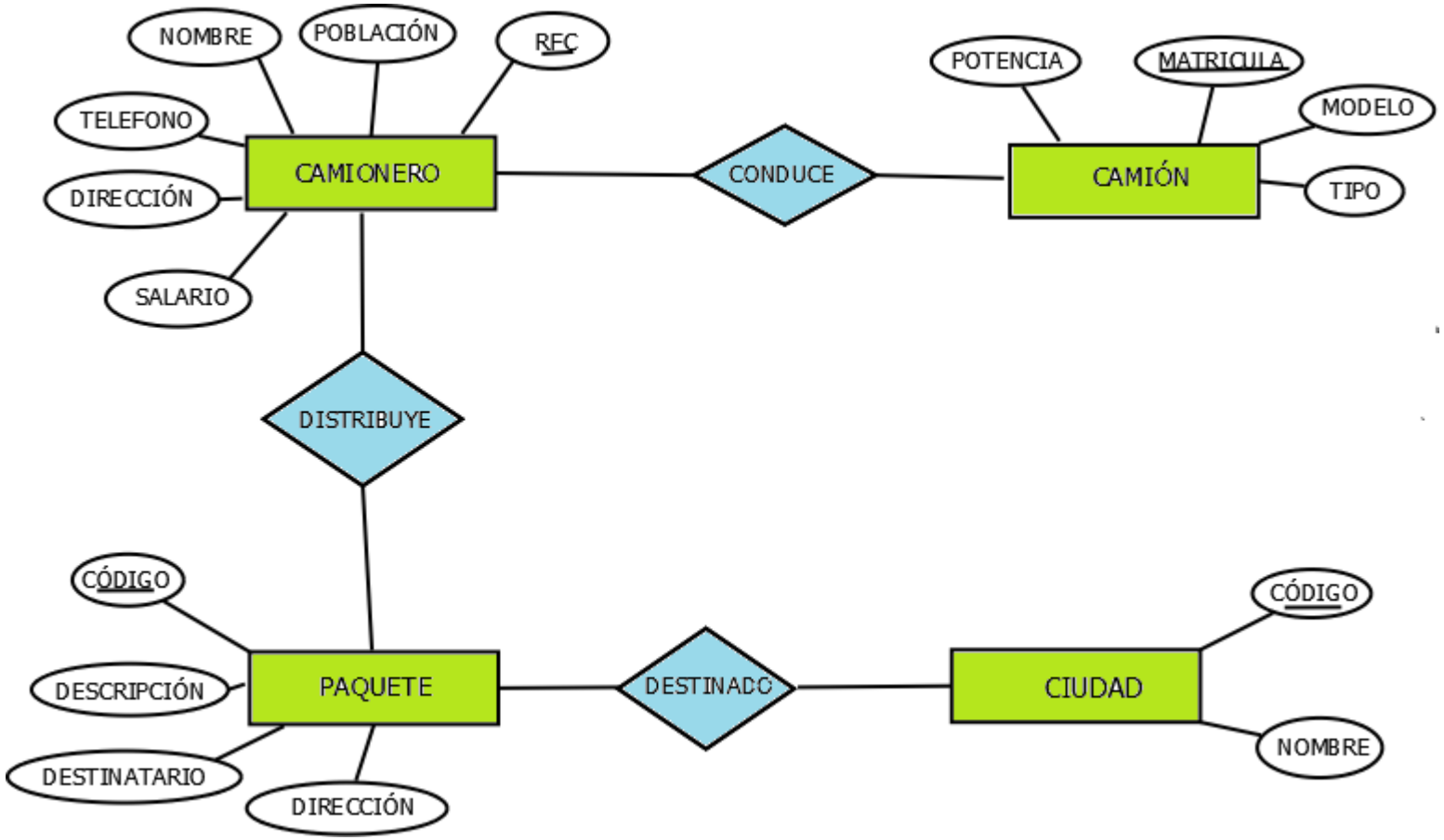
Almacenamiento de los datos

- A la hora de trabajar con bases de datos, nos referimos a la **definición de estructuras y semántica** como **modelado de datos**
  - **Modelo de datos**. Una colección de conceptos que describen como son representados y accedidos los datos
  - **Esquema**. Una descripción de una colección especifica de datos usando el modelo de datos
- Algunos ejemplos de modelos de datos son
  - Relacionales, entidad-relación, eXtensible Markup Language (XML)
  - Orientados a objetos, objeto-relacional, Resource Description Framework (RDF)
  - Protocol Buffers, JavaScript Object Notation (JSON)

Modelos Entidad-Relación

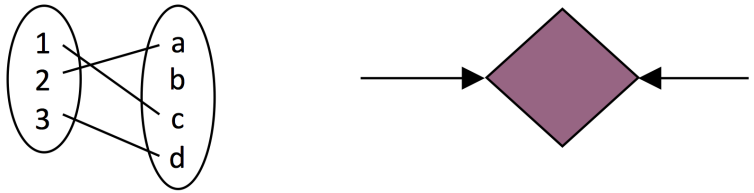
- Son ampliamente utilizados en bases de datos (ejemplo MySQL)
- Los objetos fundamentales de este modelos son las **entidades**, sus **atributos**, y las **relaciones** y sus atributos
- Las **entidades son los objetos** (muestras) representados en los conjuntos de datos/bases de datos (personas, casas, imágenes…)
- Las relaciones son simplemente las relaciones entre las distintas entidades de la base de datos

Modelos Entidad-Relación

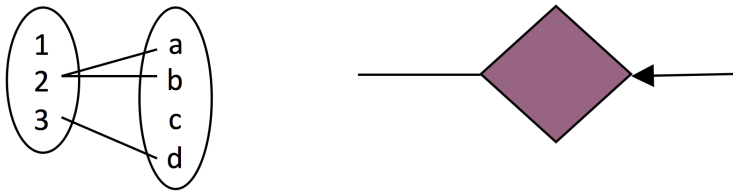


Modelos Entidad-Relación

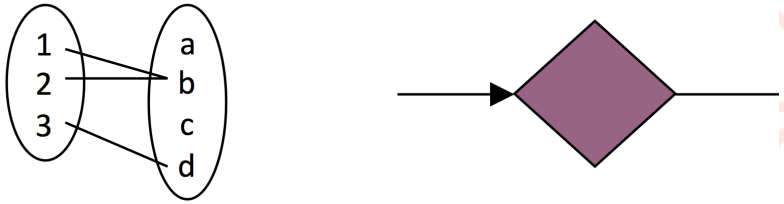
- Existen 4 tipos de relaciones
- Relación uno a uno



- Relación muchos a uno



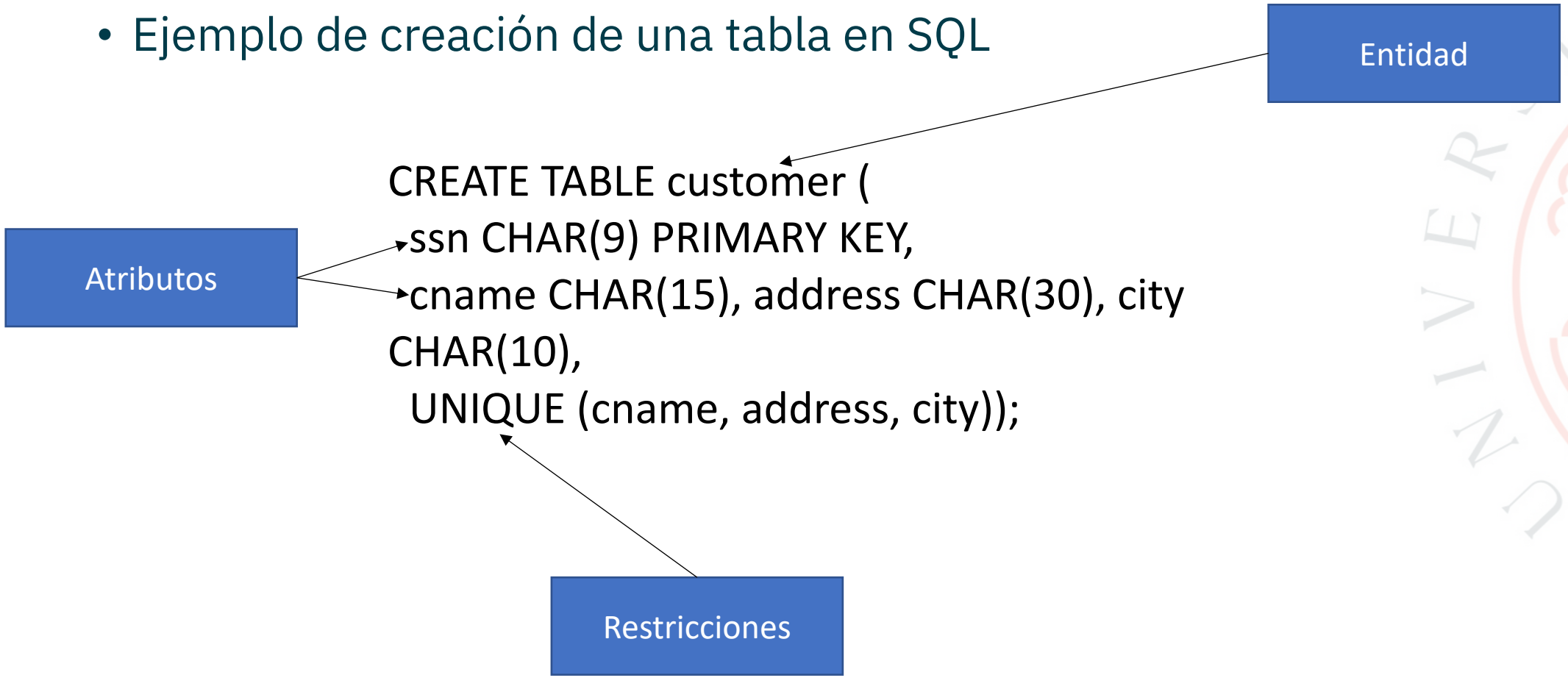
Relación muchos a uno



Relación muchos a muchos

Modelos Entidad-Relación

- Ejemplo de creación de una tabla en SQL



Almacenamiento de los datos

- Se utilizan **formatos estructurados** en forma de **texto plano**
- Estos formatos utilizan **delimitadores** o **identificadores** para separar entre los campos
- Los formatos más utilizados son aquellos que **separan** los campos por **comas** (,) o por **tabuladores**
- También hay formatos más organizados que almacenan la información mediante **etiquetas especiales** (XML) o mediante objetos de javascript (JSON)

Almacenamiento de datos

- Comma-Separated Value (CSV). Este es el formato más utilizado en procesamiento de datos
- **Cada línea representa una muestra del conjunto de datos**. Los **datos** dentro de una línea **se separan** mediante el carácter de **coma** (,)
- La **primera línea** tiene un **significado especial** y describe todos las características de las muestras

```
Year,Make,Model,Description,Price
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture Extended Edition","",4900.00
1999,Chevy,"Venture Extended Edition, Very Large","",5000.00
1996,Jeep,Grand Cherokee,"MUST SELL! air, moon roof, loaded",4799.00
```

Almacenamiento de datos

- Tab-Separated Values (TSV). Es otro formato ampliamente utilizado
- En este formato cada línea representa una muestra del conjunto de datos pero esta vez, **la separación viene dada por una tabulación**
- Como en el caso anterior, la primera línea indica el nombre de las características del conjunto de datos

Year	Make	Model	Description	Price
1997	Ford	E350	ac, abs, moon	3000.00
1999	Chevy	Venture	Extended Edition	4900.00
1999	Chevy	Venture	Extended Edition, Very Large	5000.00
1996	Jeep	Grand Cherokee	MUST SELL! air, moon roof, loaded	4799.00

Almacenamiento de datos

- Otro formato popular es eXtensible Markup Language (XML). Inicialmente fue diseñado para que sea entendible tanto por humanos como por máquinas

Almacenamiento de datos

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <name>EMPLOYEE_WEBROWSET</name>
  <metadata>
    <col name="FIRSTNME" native_type="VARCHAR(12)" nullable="false" type="12"/>
    <col name="LASTNAME" native_type="VARCHAR(15)" nullable="false" type="12"/>
    <col name="BIRTHDATE" native_type="DATE" nullable="true" type="91"/>
  </metadata>
  <data>
    <row>
      <value>CHRISTINE</value>
      <value>HAAS</value>
      <value>01011995</value>
    </row>
  </data>
</dataset>
```

Almacenamiento de datos

- El último formato que veremos es JavaScript Object Notation (JSON)
- Es un mecanismo liviano de intercambio de datos, no solo para los humanos sino también para las máquinas
- Esta formado por dos estructuras principales
  - **Una colección de pares claves-valor**. En muchos lenguajes de programación esto se podría traducir como un objeto, diccionario, estructura, array asociativo, etc
  - **Una lista ordenada de valores**. En la mayoría de lenguajes de programación esto se conoce como un array, secuencia, vector o lista

Almacenamiento de datos


```
{
 "datos": {
  "dato": [
   {
    "id_equipo": 2020018163,
    "id_tipo_equipo": "RMDT",
    "ubicacion": 362,
    "fecha_instalacion": "14/07/22",
    "fecha_consulta": "22/12/22"
   },
   {
    "id_equipo": 2020018185,
    "id_tipo_equipo": "RMDT",
    "ubicacion": 351,
    "fecha_instalacion": "20/06/22",
    "fecha_consulta": "22/12/22"
   },
   ...
   ]}
}
```

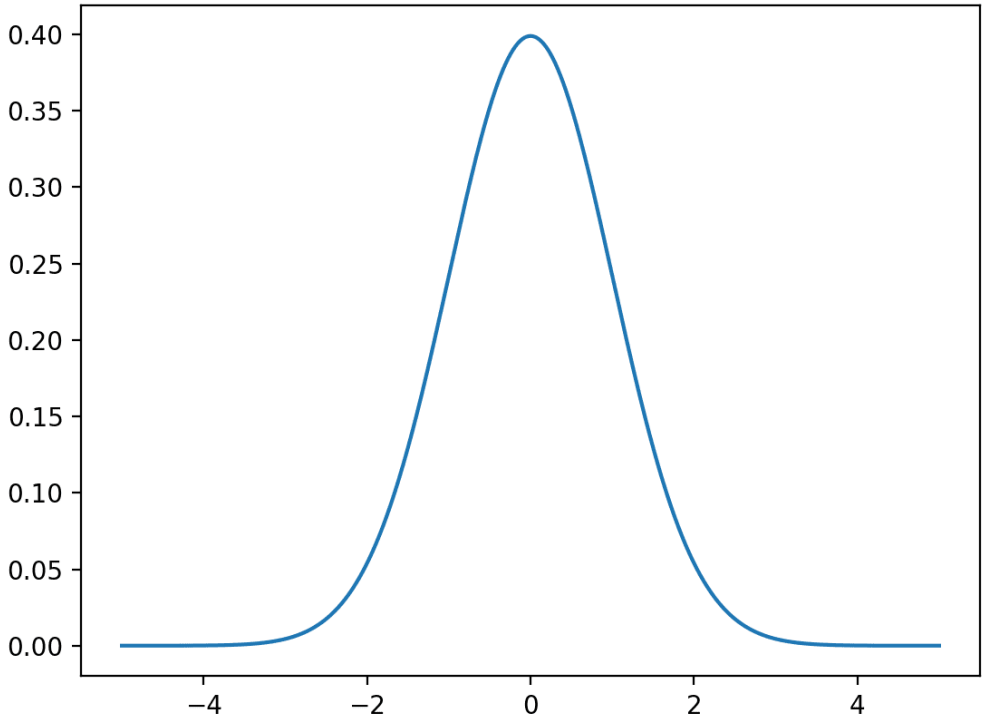
¿Dónde están los conjuntos de datos?

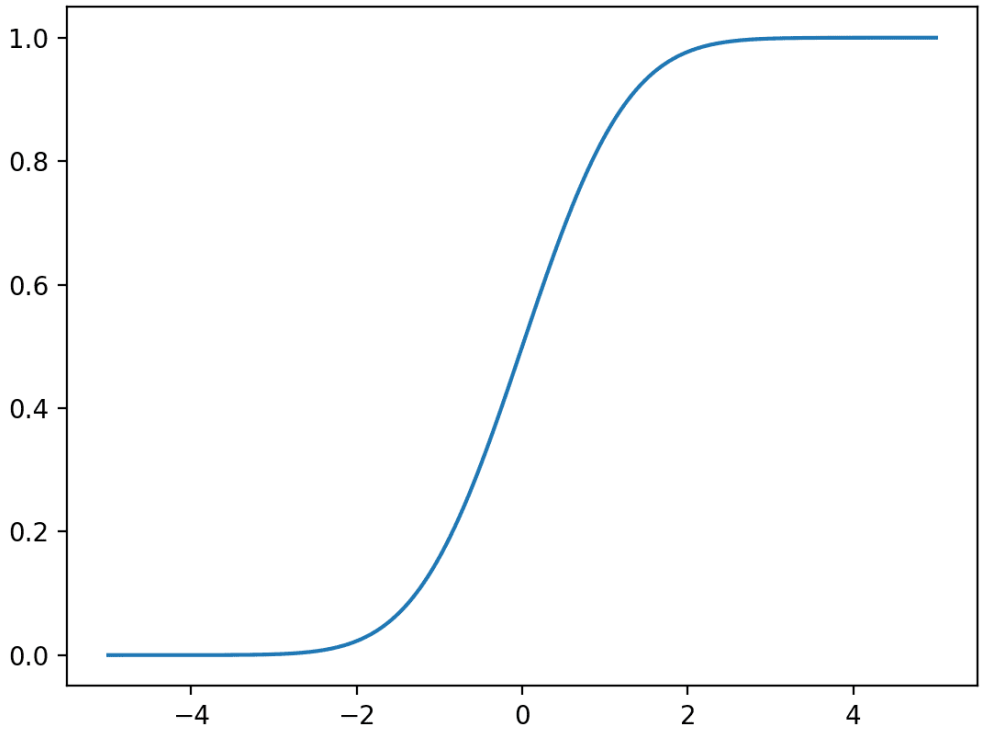
- Hay muchos sitios públicos para obtener conjuntos de datos
- https://datos.gob.es/es/catalogo
- https://github.com/awesomedata/awesome-public-datasets
- https://data.gov/
- https://www.kaggle.com/datasets
- https://datasetsearch.research.google.com/
- https://archive.ics.uci.edu/ml/datasets.php

Recordatorio sobre distribuciones

- Las distribuciones son descritas con frecuencia en términos de **funciones de densidad**
- Las funciones de densidad describen como cambia la proporción de datos o la **probabilidad** de la proporción de observaciones en el rango de la distribución
- Dos tipos importantes de distribución
  - **Funciones de densidad de probabilidad (PDF)**. Calculan la probabilidad de observar un determinado valor
  - **Funciones de densidad acumulativa (CDF)**. Calculan la probabilidad de que una observación sea igual o menor que un determinado valor

Recordatorio sobre distribuciones





Recordatorio sobre distribuciones

- Una distribución normal, como la mostrada anteriormente, se define mediante dos parámetros
  - Media
  - Desviación típica
- La ecuación matemática que define a una distribución normal es la siguiente

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

Recordatorio sobre distribuciones

- La función de densidad acumulativa es común a todas las distribuciones y se calcula de la siguiente forma

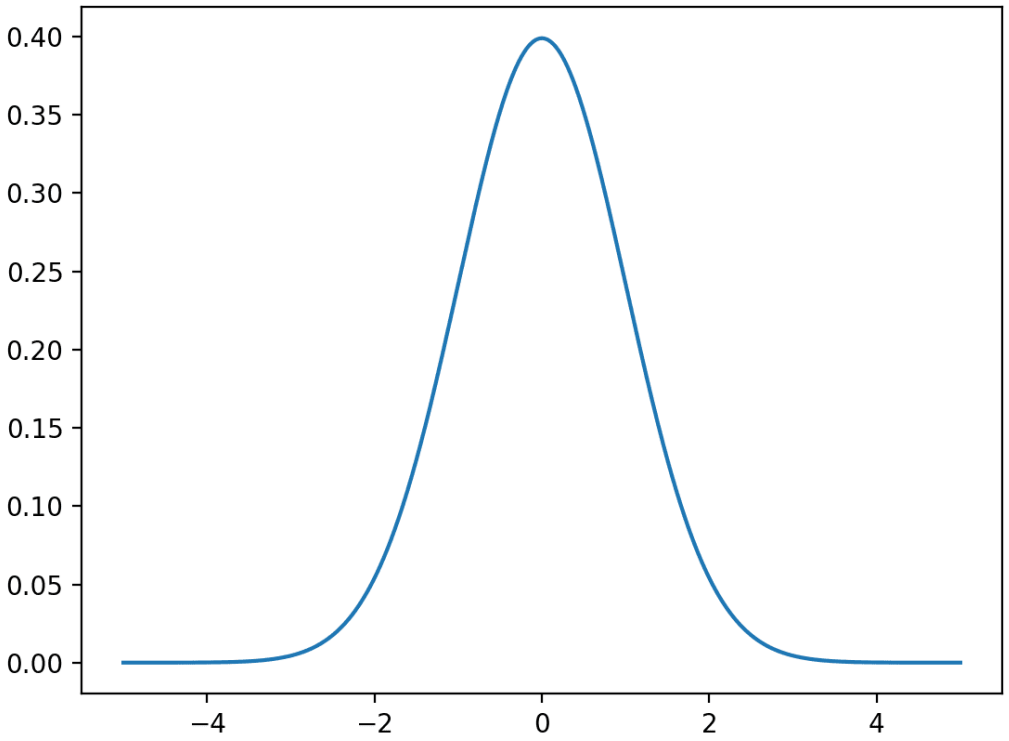
$$P(X \leq x) = \sum_{t=x_{min}}^{x} P(X = t)$$

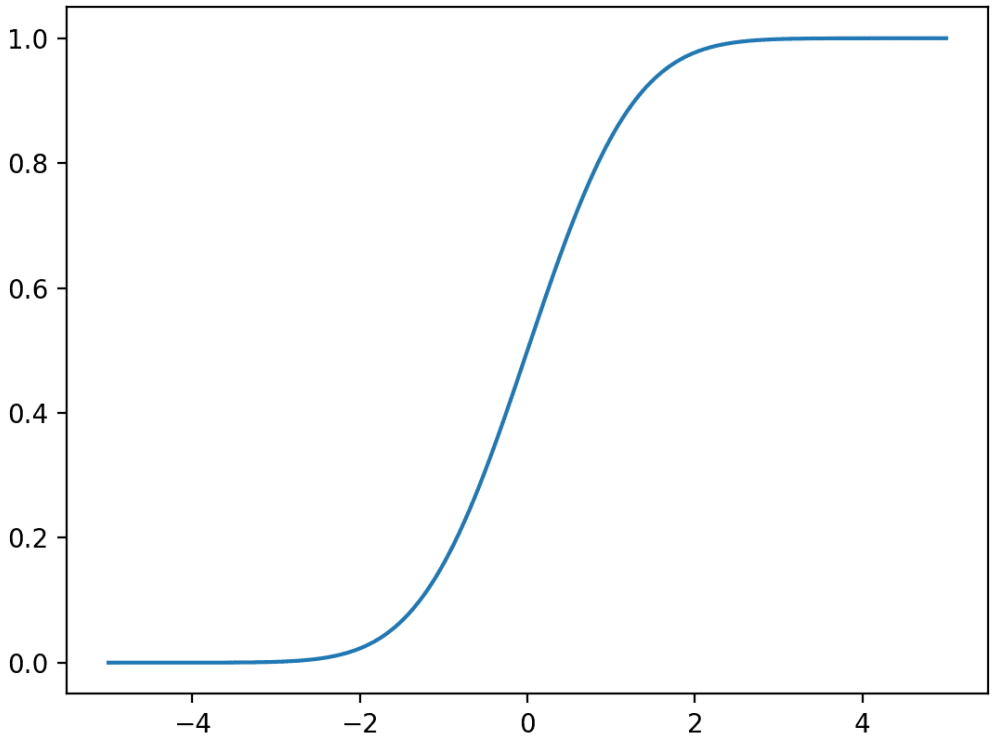
Recordatorio sobre distribuciones

- Otra distribución importante es la distribución **t-Student**
- Esta distribución surge al intentar estimar la media de una distribución normal con **muestras de distintos tamaños**. Es útil cuando se describe la incertidumbre (o el error) relacionada con la estimación de estadísticas de población extraídos de distribuciones gaussianas cuando se debe tener en cuenta el tamaño de la muestra
- La formula que describe esta distribución es la siguiente

$$f(x) = \frac{x - \mu}{\frac{\sigma}{\sqrt{n}}}$$

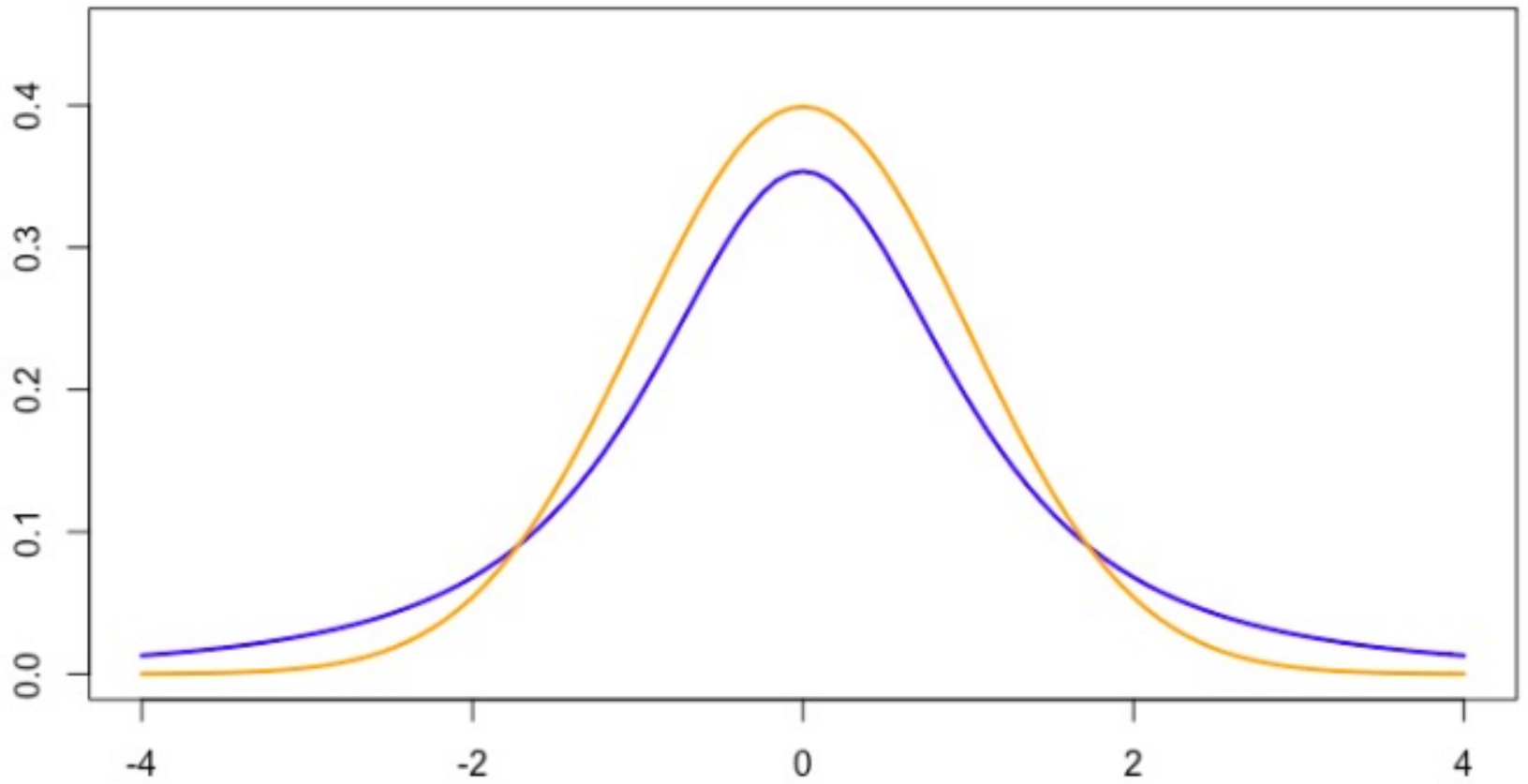
Recordatorio sobre distribuciones





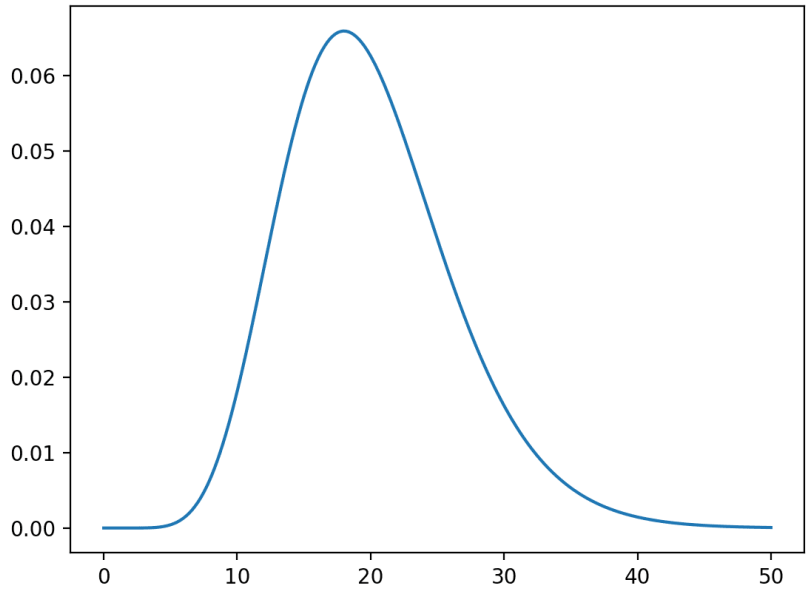
Recordatorio sobre distribuciones

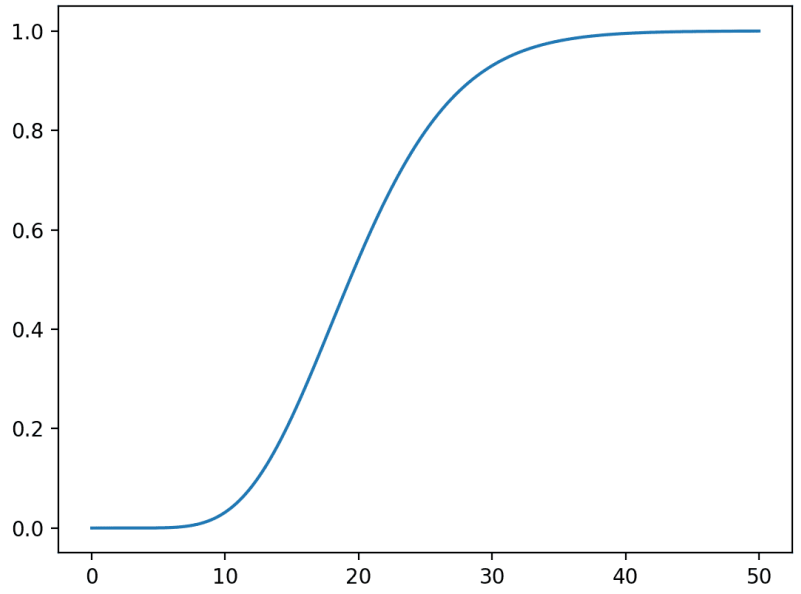
- ¿Cuál de estas distribuciones es normal y cual t-Student?



Recordatorio sobre distribuciones

- Otra distribución importante es la distribución chi-square
- Al igual que la distribución t-Student se usa en estadística cuando se quiere representar la incertidumbre derivada de una distribución normal





Recordatorio sobre distribuciones

- Las distribuciones estadísticas pueden ser divididas en diferentes conjuntos
  - **Cuartiles**. La distribución se divide en cuartos. Una distribución estadística se divide en 4 cuartiles (Q1, Q2 y Q3)
  - **Deciles**. La distribución se divide en diez partes iguales (D1, …, D9)
  - **Percentiles**. La distribución se divide en 100 partes iguales (P1, …, P99)
- Las formulas para calcular los tres cuartiles son las siguientes

$$Q1 = \frac{n+1}{4} \qquad Q2 = \frac{n+1}{2} \qquad Q3 = \frac{3(n+1)}{4}$$

Ejercicios

- La siguiente tabla contiene un conjunto de datos de aseguradores de coches ficticios junto a la puntuación recibida por los últimos 3 clientes. Si tuvieses que elegir un seguro ¿Cuál elegirías y por qué?

#	Compañía de seguro	Nota
1	Aseguradora A	4.7
2	Aseguradora A	8.3
3	Aseguradora A	9.2
4	Aseguradora B	7.4
5	Aseguradora B	6.7
6	Aseguradora B	8.9
8	Aseguradora C	3.8
9	Aseguradora C	6.3
10	Aseguradora C	8.1

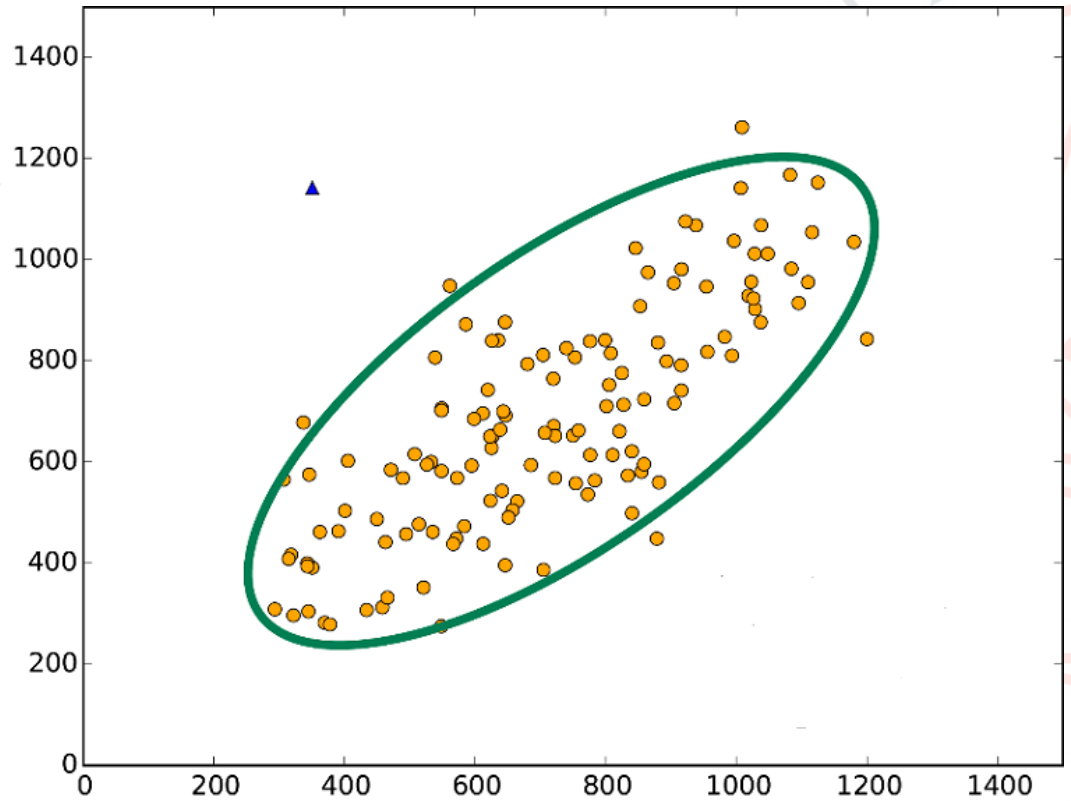
Ejercicios

- ¿De la siguiente lista que película elegirías y por qué?

Director	Película	Nota en Filmaffinity
Quentin Tarantino	Django Desencadenado	7.9
Quentin Tarantino	Los Odiosos Ocho	7.3
Quentin Tarantino	Malditos Bastardos	7.8
Cristopher Nolan	Interestelar	7.9
Cristopher Nolan	Origen	8.0
Cristopher Nolan	El Caballero Oscuro	8.1
Steven Spielberg	Atrápame Si puedes	7.3
Steven Spielberg	Salvar Al Soldado Ryan	7.8
Steven Spielberg	Parque Jurásico	7.1

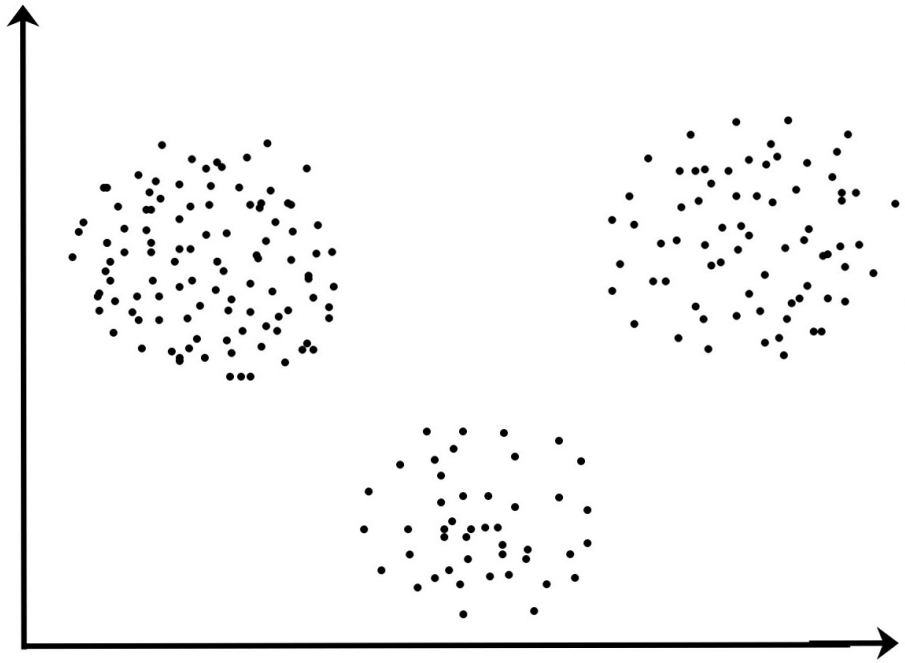
Ejercicios

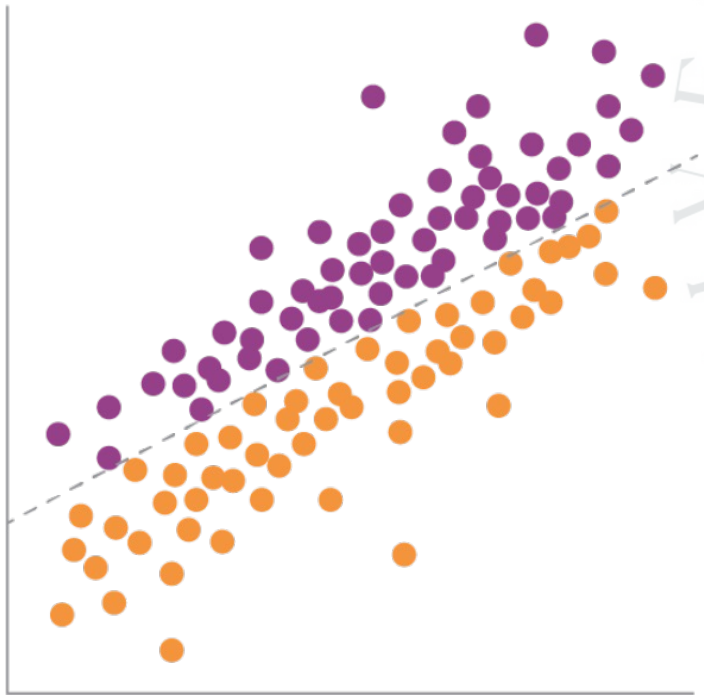
- De los tres tipos de conjuntos de datos vistos anteriormente, ¿A que tipo corresponde el conjunto de datos representado en la figura?



Ejercicios

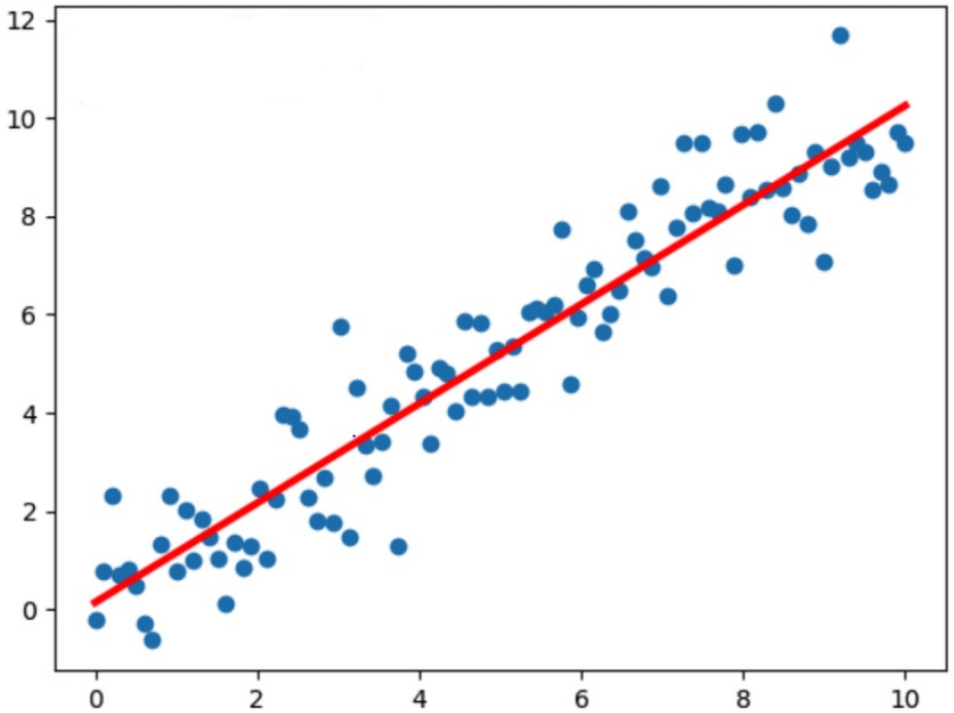
- ¿A que tipo de conjuntos de datos pertenecen las siguientes representaciones?

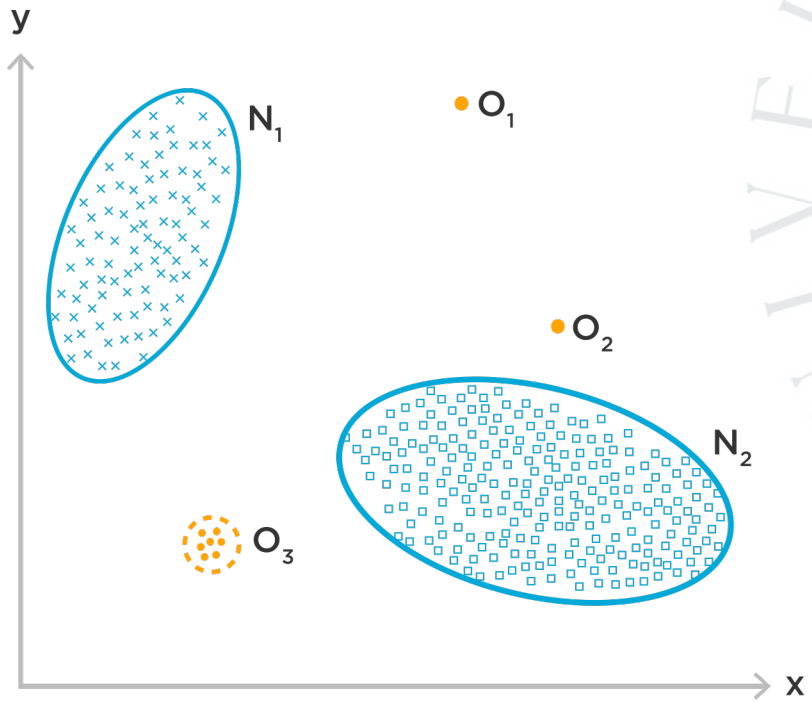




Ejercicios

- ¿A qué tipo de conjuntos de datos pertenecen las siguientes representaciones?





Bibliografía

- Grus, J. (2019). *Data science from scratch: first principles with python*. O'Reilly Media, 978-1492041139.
- Osborne, J. (2012). Best Practices in Data Cleaning: A Complete Guide to Everything You Need to Do Before and After Collecting Your Data, SAGE Publications, 978-1412988018.
- Mertz, D. (2021). *Cleaning Data for Effective Data Science: Doing the other 80% of the work with Python, R, and command-line tools*. Packt Publishing Ltd.

Grado en Gestión de la Información y Contenidos Digitales

Procesamiento y Visualización de Datos

Tema 3. Limpieza, Integración, Transformación y Reducción de Datos

Bloque 1. Procesamiento de datos

UNIVERSIDAD DE MURCIA

Facultad de Informática

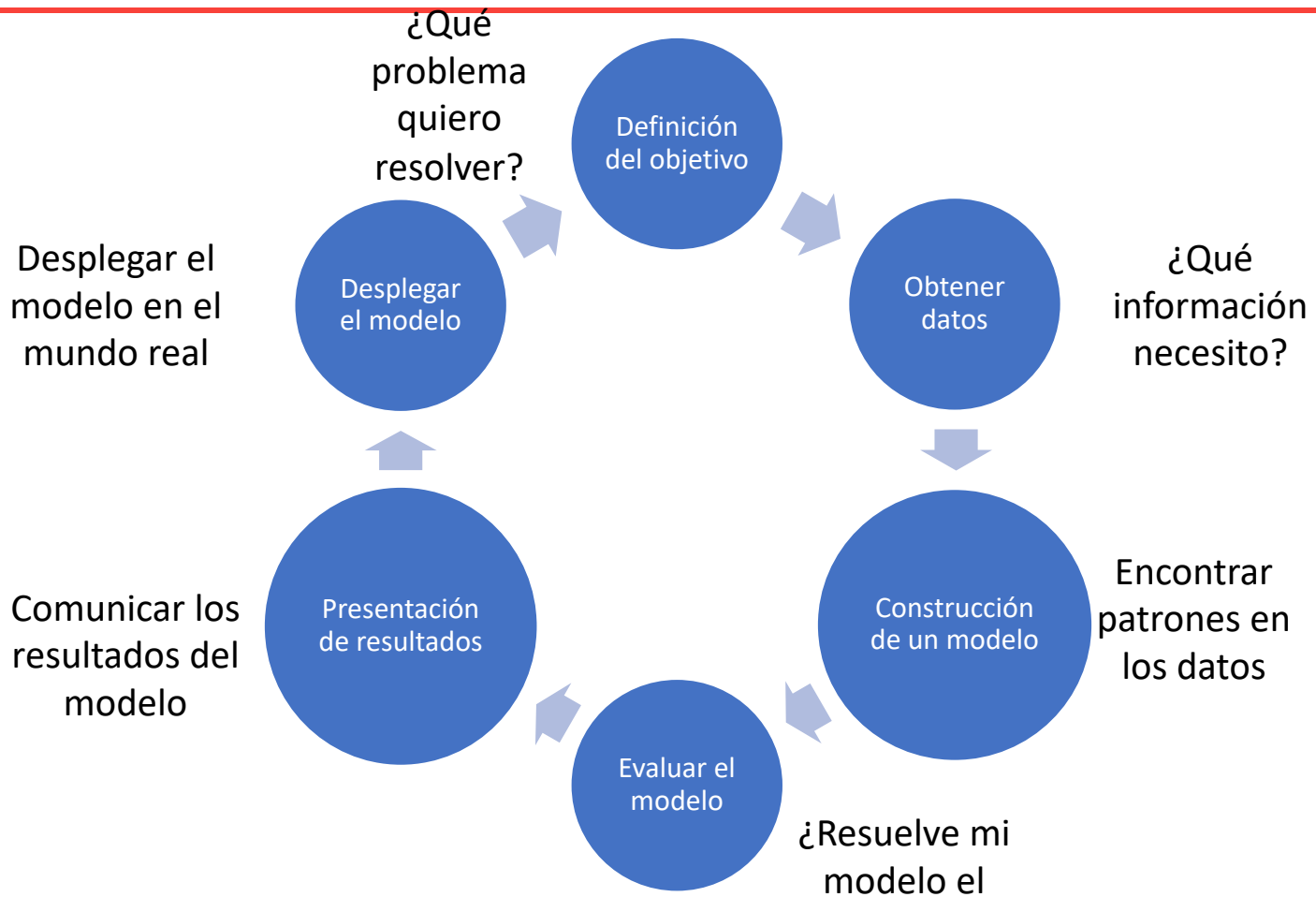
Introducción

- Con frecuencia el procesamiento de datos es una **fase previa** de un proceso mucho mayor

- Una tarea frecuente del procesado de datos es transformar los datos de tal manera que puedan emplearse para desarrollar un modelo de Inteligencia Artificial

- Este modelo será usado para **hacer predicciones** o **clasificación** en problemas del mundo real

Introducción



Introducción

- En este tema nos centraremos en como **procesar los datos** para que puedan ser presentados de forma correcta

- También se detallarán técnicas para transformar los datos de tal manera que puedan ser utilizados para desarrollar un modelo de Inteligencia Artificial

- Sin embargo, en esta asignatura **no trataremos la generación de dichos modelos**

Preprocesamiento de datos

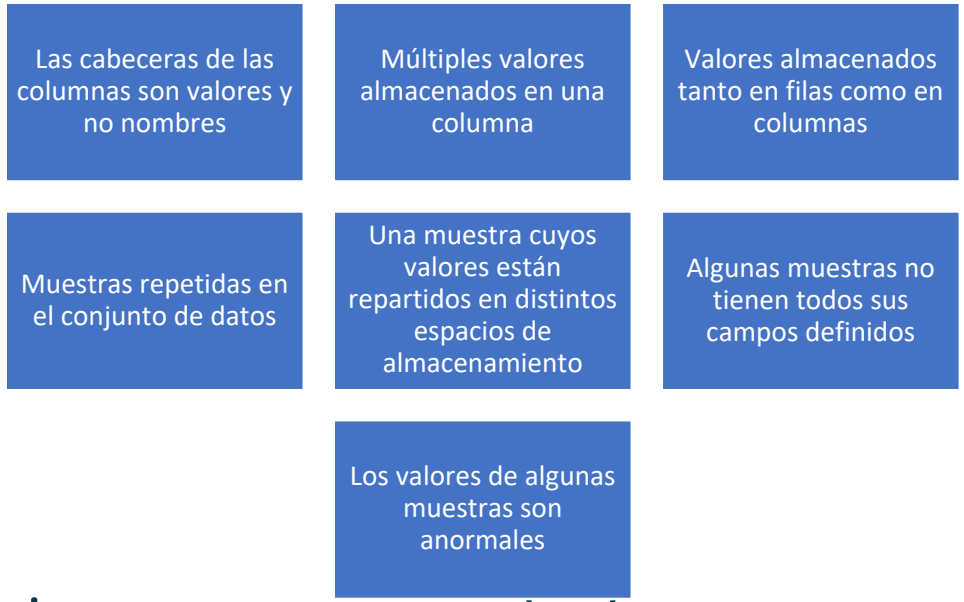
- Con frecuencia, los datos que nos encontramos en el mundo real **no están en una forma adecuada** para su procesamiento por una máquina
- Para adecuar los datos y que puedan ser usados es necesario una serie de tareas
  - Limpieza de datos
  - Integración de datos
  - Transformación de datos
  - Reducción de los datos
- No siempre es necesario aplicar todas las tareas

Limpieza de datos

- De forma coloquial podemos decir que los datos están sucios y deben ser limpiados antes de que puedan utilizarse
- Hay distintos factores por los que un conjunto de datos puede estar sucio
  - **Incompleto**. Se da cuando una o varias características de una o varias muestras no están disponibles. También se puede dar cuando la característica contiene datos agregados y no datos en crudo
  - **Ruido**. Ocurre cuando un conjunto de datos contiene muestras cuyos valores se alejan significativamente de los de otras muestras
  - **Inconsistencia**. Ocurre cuando los datos contienen discrepancias en sus características. Por ejemplo, el nivel educativo no se representa de la misma forma, o cuando el nombre contiene números o no empieza por mayúscula

Limpieza de datos

- A continuación se presentan una serie de problemas comunes que hacen que los datos estén sucios



- Vamos a ver 3 técnicas que nos ayudarán en estos problemas

Manipulación de los datos

- La primera técnica que veremos es la **manipulación de los datos**
- Esta técnicas es relativamente sencilla y se basa en manipular los datos (ya sea manual o automáticamente) para que sean entendibles por máquinas y humanos
- Imaginemos que tenemos la siguiente receta: “Agregue dos tomates cortados en cubitos, tres dientes de ajo y una pizca de sal en la mezcla”
- Tal y como está ahora mismo no resulta apto para ser procesado por una máquina

Manipulación de los datos

- Sin embargo, si lo expresamos de la siguiente forma

Ingrediente	Cantidad	Unidad/tamaño
Tomate	2	cubitos
Ajo	3	diente
Sal	1	pizca

- Ya puede ser procesado adecuadamente por una máquina
- La cuestión aquí es como convertir el texto a la tabla superior
  - No hay una solución clara y científica. Simplemente hay que hacer lo que sea necesario para realizar dicha transformación

Gestión de valores faltantes

- Algunas veces los datos están en el formato adecuado pero algunos **valores** de las características **pueden faltar**
- Imagina una base de datos de usuarios donde hay un campo para el teléfono fijo de casa y otro campo para el teléfono móvil
- Una persona que no tenga teléfono móvil dejará este campo vacío (y una persona que no tenga teléfono fijo en casa dejará el campo correspondiente vacío)
- Otras veces, estos valores que faltan se pueden deber a problemas con el proceso de recolección de datos o a un funcionamiento erróneo de un equipo

Gestión de valores faltantes

- Dependiendo de cuales sean los valores que faltan en un conjunto de datos podemos distinguir tres categorías
- Valores **faltantes completamente aleatorios** (Missing Completely At Random, MCAR). Ocurre cuando el proceso de perdida no depende de ninguna otra característica del conjunto de datos. Por ejemplo: valores que pueden faltar debido al malfuncionamiento de un dispositivo
- Valores **faltantes aleatorios** (Missing at Random, MAR). Ocurre cuando los valores que faltan dependen de alguna otra característica del conjunto de datos, pero no de la etiqueta, en el caso de que la hubiese. Es decir, los valores faltantes no se distribuyen aleatoriamente en todo el conjunto de datos pero si en un subconjunto del conjunto de datos.
- Valores **faltantes no aleatorios** (Missing Not At Random, MNAR). Si los datos faltantes no se consideran MCAR o MAR, entonces se considerarán MNAR

Gestión de valores faltantes

- Para solucionar este problema no hay una solución única y es **necesario definir una estrategia** adecuada atendiendo a las necesidades de cada caso particular

- Estrategias populares para afrontar este problema:
  - Ignorar directamente la muestra
  - Usar una constante global para indicar que el dato falta (por ejemplo: -1)
  - Generar ese valor faltante en base a los datos de las demás muestras del conjunto de datos

- Todas estás técnicas tienen sus ventajas e inconvenientes

Gestión de valores faltantes

- **Generar ese valor faltante en base a los datos de las demás muestras del conjunto de datos**
- Este conjunto de técnicas recibe el nombre de **imputación**

País	Edad	Salario
España	44	32.000
Francia	38	-
Alemania	56	56.000
Italia	26	28.000

- ¿Cómo calcular el salario faltante?
  - Establecer el salario de Francia como la media de los demás salarios. También se pueden usar otras medidas estadísticas

Gestionar valores anormales

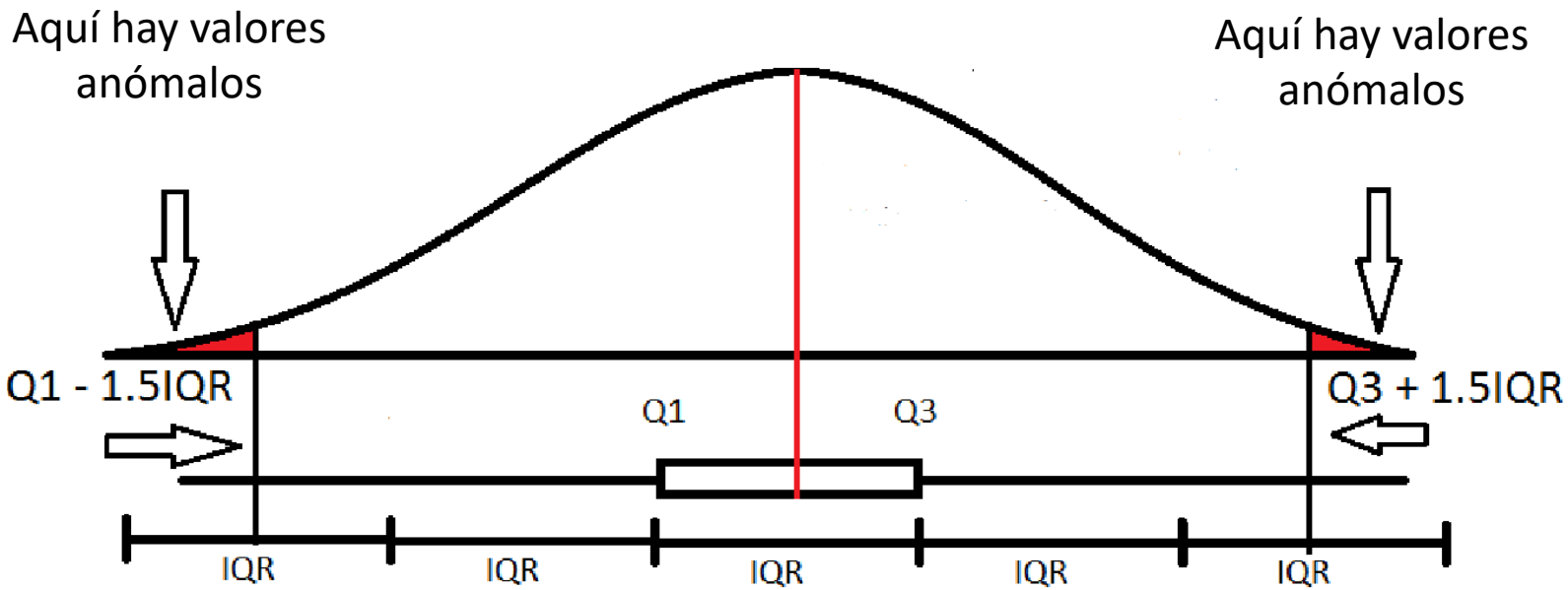
- A veces, es posible que el valor no falte, sino que el valor esté corrupto
- Esto se puede deber a muchas razones. Por ejemplo, el equipo que se encarga de recoger los datos funciona de forma errónea. Imagina un termómetro que devuelve una temperatura excesivamente alta
- Esta casuística es más problemática que cuando faltan datos
  - ¿Cómo identificar cuando un dato está corrupto?
  - ¿Cómo arreglar ese dato corrupto?

Gestionar valores anormales

- Podemos utilizar el Rango Intercuartílico (IQR) para eliminar valores anómalos de forma sencilla

$$IQR = Q3 - Q1$$

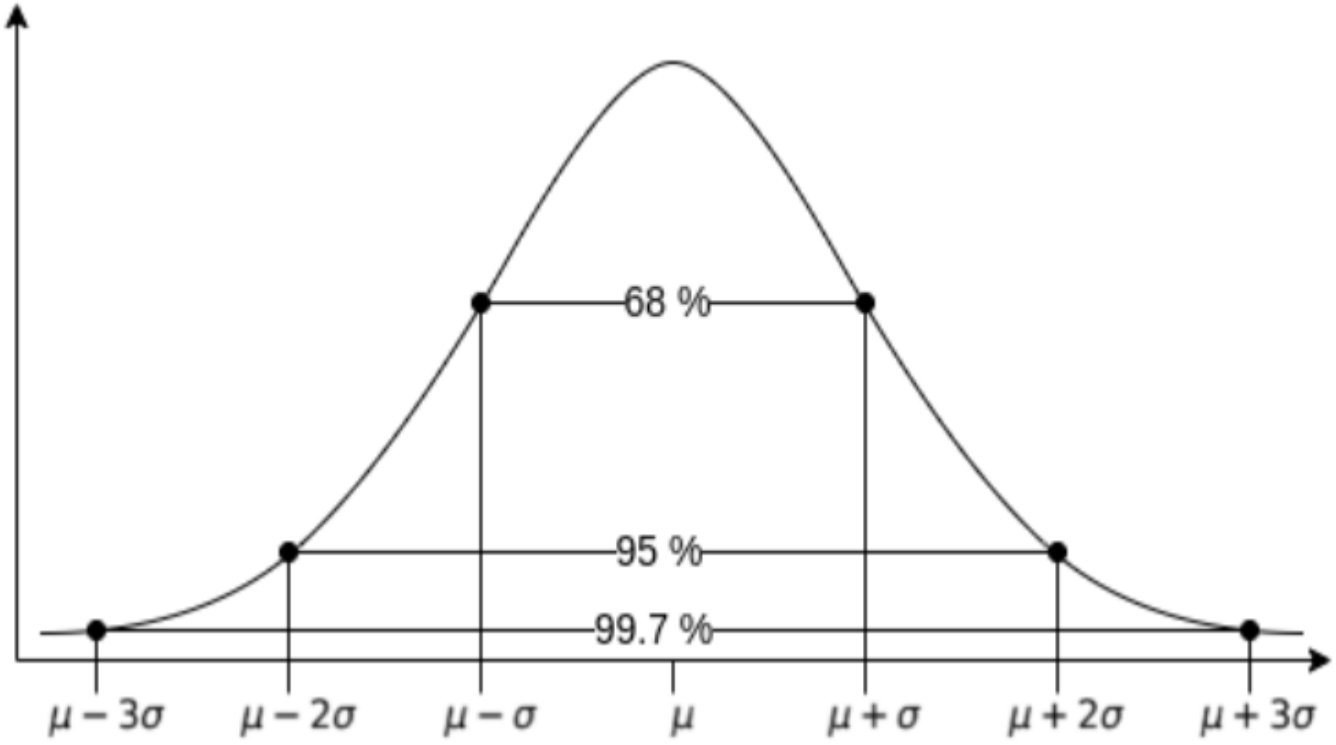
$$Umbral_i = Q1 - 1.5IQR \qquad Umbral_s = Q3 + 1.5(IQR)$$



Gestionar valores anormales

- Otra solución ampliamente aceptada es utilizar la media y la desviación típica para eliminar valores anómalos

- Todo aquello que **exceda un** determinado **umbral será considerado anómalo**
- En general valores de μ+2σ, μ+2.5σ y μ+3σ son aceptados



Integración de datos

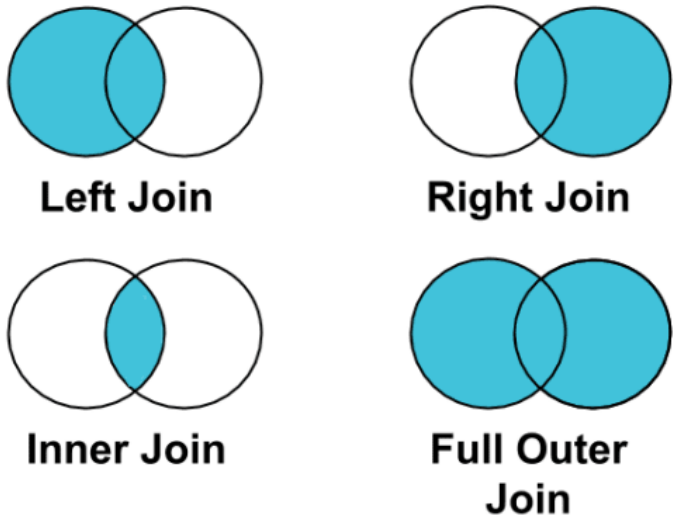
- En ocasiones es preciso **combinar datos** procedentes de distintas fuentes de para tener una visión global
- En estos casos, la estrategia a emplear seguirá un esquema parecido al siguiente
- **Paso 1**. Combinar los datos procedentes de distintas fuentes en un solo espacio de almacenamiento (fichero o base de datos)
- **Paso 2**. Participar en la integración de esquemas o combinación de metadatos de distintas fuentes

Integración de datos

- **Paso 3**. Detectar y resolver los conflictos que puedan surgir con los valores de los datos
  - Podría suceder que cada fuente de datos contenga **diferentes características y valores** para la misma entidad del mundo real
  - También podría suceder que los **valores se encuentren en distintas escalas o métricas**. Por ejemplo, medidas expresadas en el sistema británico y en el sistema métrico internacional
- **Paso 4**. Resolver la redundancia de datos una vez se hayan integrado estos. Esta redundancia es bastante común al integrar distintas fuentes
  - La misma característica tiene **distintos nombres** en los diferentes conjuntos de datos

Integración de datos

- Combinar datos de distintas fuentes es complicado
  - ¿Qué ocurre si las filas no tienen ninguna característica en común?
  - ¿Qué ocurre si en ambas fuentes de datos se definen características repetidas?
- Existen distintas estrategias de unión de datos



Integración de datos

- Left Join

C1	C2
A	1
B	2
C	3

+

C1	C3
A	T
B	F
D	T

=

C1	C2	C3
A	1	T
B	2	F
C	3	

- Right Join

C1	C2
A	1
B	2
C	3

+

C1	C3
A	T
B	F
D	T

=

C1	C3	C2
A	T	1
B	F	2
D	T	

Integración de datos

- Inner Join

C1	C2
A	1
B	2
C	3

+

C1	C3
A	T
B	F
D	T

=

C1	C2	C3
A	1	T
B	2	F

- Full Outer Join

C1	C2
A	1
B	2
C	3

+

C1	C3
A	T
B	F
D	T

=

C1	C2	C3
A	1	T
B	2	F
C	3	
D		T

Transformación de datos

- Los modelos predictivos que se van a generar posteriormente, esperan que los datos se encuentren en una formato adecuado para que puedan ser procesados
- Un **modelo no es capaz** de **procesar características categóricas** (ej: Hombre, Mujer)
- Además, muchos de los modelos predictivos requieren que los valores de las características se encuentren en un **rango común**
  - Un conjunto de datos que contenga información sobre las distancias entre el planeta tierra y el resto de estrellas conocidas. La estrella más cercana a la tierra es Alfa Centauri a unos 4.3 años luz de la tierra mientras que la más lejana es Eärendel a unos 12.900.000.000 años luz

Transformación de datos

- Para poder trabajar con características categóricas es necesario que éstas sean convertidas a alguna representación numérica
- Label Encoding o One Hot Encoding

País	Edad	Salario
España	44	32.000
Francia	38	38.000
Alemania	56	56.000
Italia	26	28.000

Transformación de datos

- La técnica Label Encoding consiste simplemente en otorgar una etiqueta numérica a cada valor categórico
- El ejemplo anterior quedaría de la siguiente forma

País	Edad	Salario
1	44	32.000
2	38	38.000
3	56	56.000
4	26	28.000

Transformación de datos

- La técnica One Hot Encoding consiste en codificar cada valor categórico como una nueva característica booleana
- El ejemplo anterior quedaría de la siguiente forma

País_España	País_Francia	País_Alemania	País_Italia	Edad	Salario
1	0	0	0	44	32.000
0	1	0	0	38	38.000
0	0	1	0	56	56.000
0	0	0	1	26	28.000

Transformación de datos

- Sin embargo, ambos enfoques tienen sus ventajas y desventajas
- Cuando se usa Label Encoding se está **presuponiendo un orden** en los datos. En el ejemplo anterior los datos se han codificado de forma que España < Francia < Alemania < Italia
- Cuando se usa One Hot Encoding podemos caer en el problema de la **multicolinearidad**. Para solucionarlo debemos eliminar una de las columnas transformadas a One Hot Encoding

Transformación de datos

- Deberemos usar One Hot Encoding cuando
  - La característica categórica **no tenga implícito un orden**. En el ejemplo anterior, la característica País no lleva implícito un orden
- Deberemos usar Label Encoding cuando
  - La característica categórica **lleve implícito un orden o jerarquía**. Por ejemplo, las posiciones de los trabajadores dentro de una empresa

Transformación de datos

- Los datos deben ser transformados para que conserven la misma escala
- Dos técnicas básicas que dependen de la distribución de los datos del conjunto de datos
- Cuando tenemos datos que siguen una distribución normal deberemos normalizar los datos siguiendo esta distribución

$$X' = \frac{X - \mu}{\sigma}$$

Transformación de datos

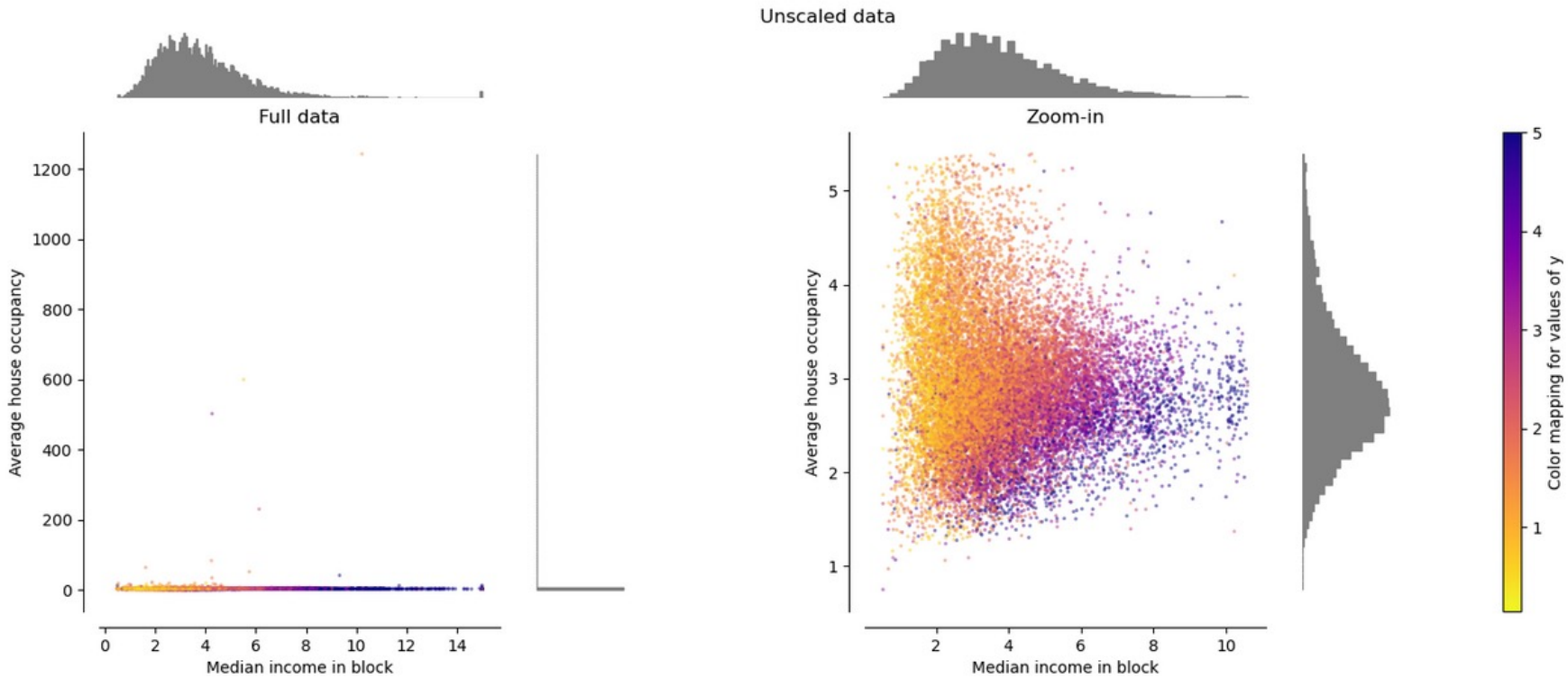
- Si la distribución de los datos almacenados en el conjunto de datos **no sigue una distribución normal** se puede usar una normalización min-max

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

- Existen otras técnicas más complejas de normalización que no serán estudiadas en esta asignatura: Normalización robusta, MaxAbs, basada en potencias, basada en cuartiles

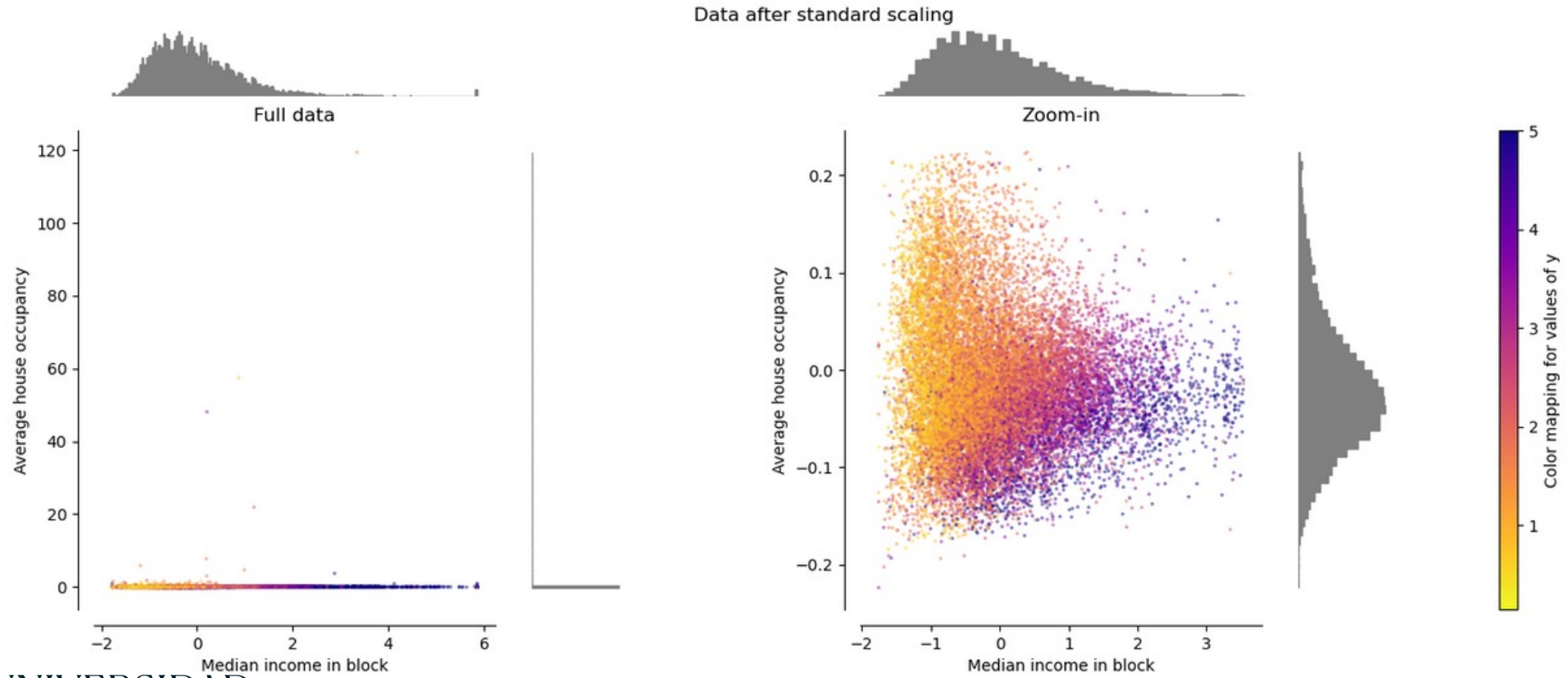
Transformación de datos

- Datos originales



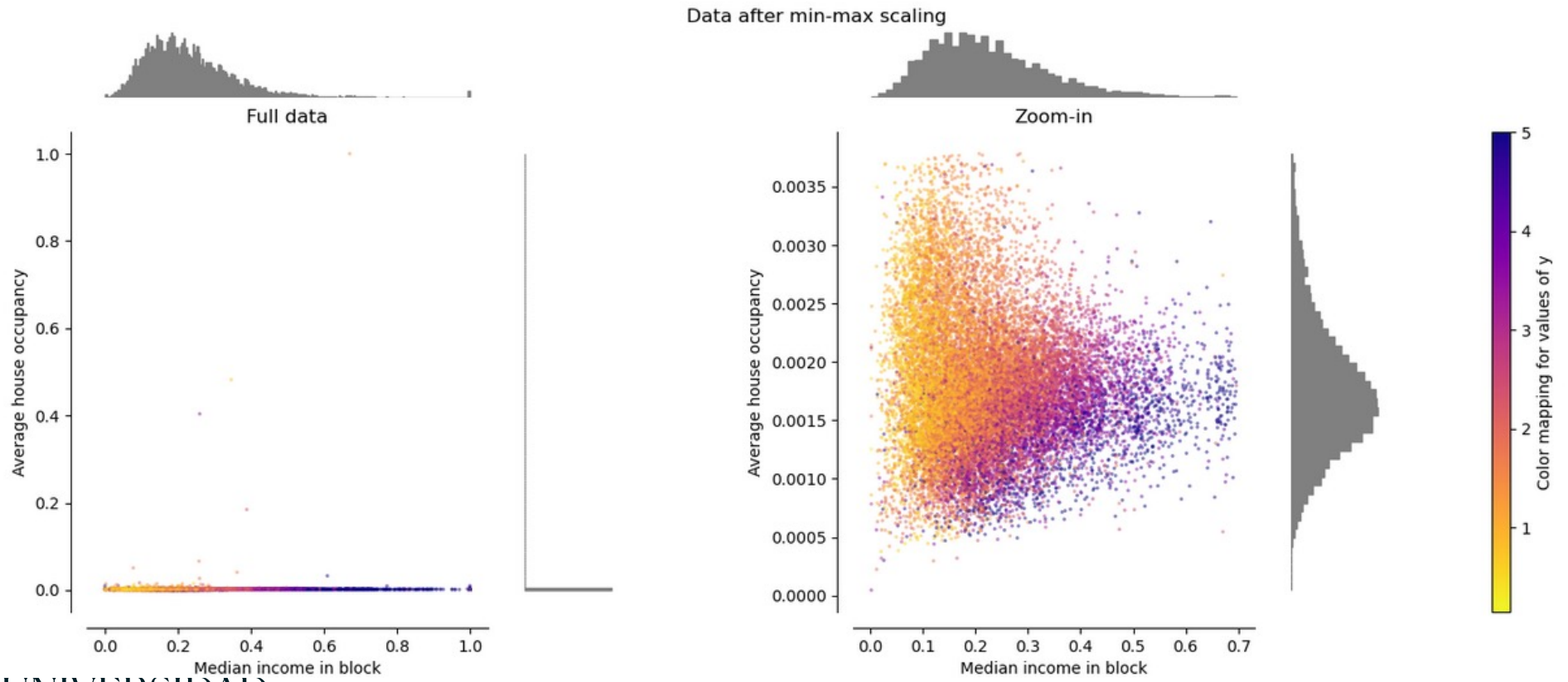
Transformación de datos

- Normalización Estándar



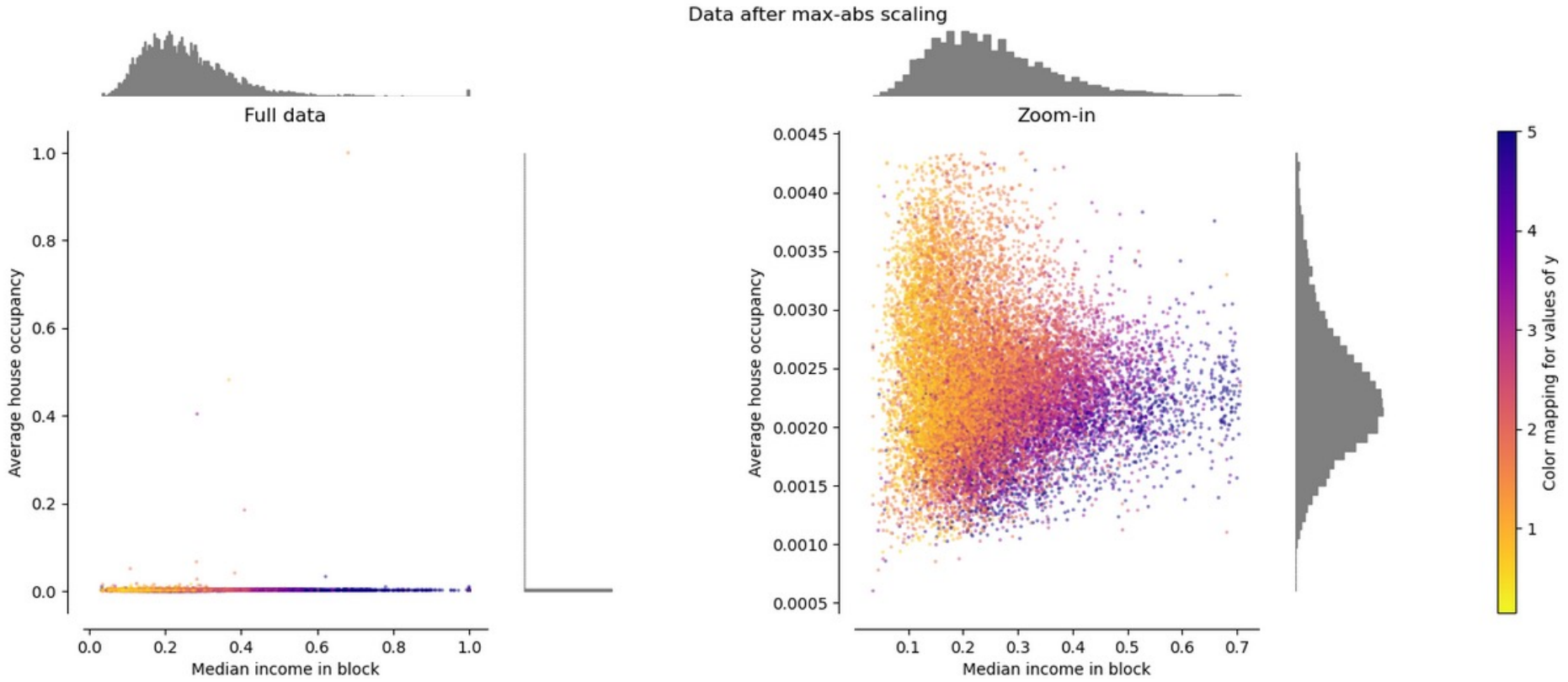
Transformación de datos

- Normalización Min-Max



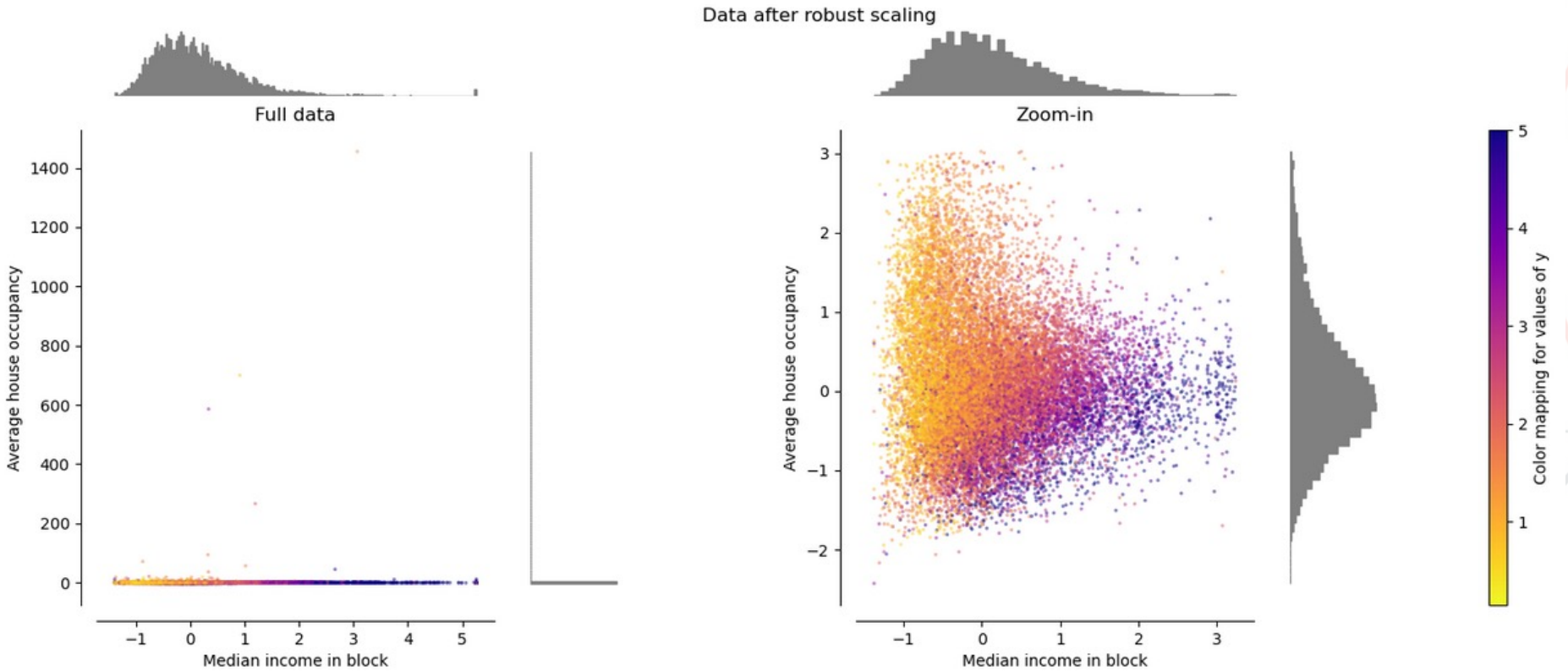
Transformación de datos

- Normalización MaxAbs



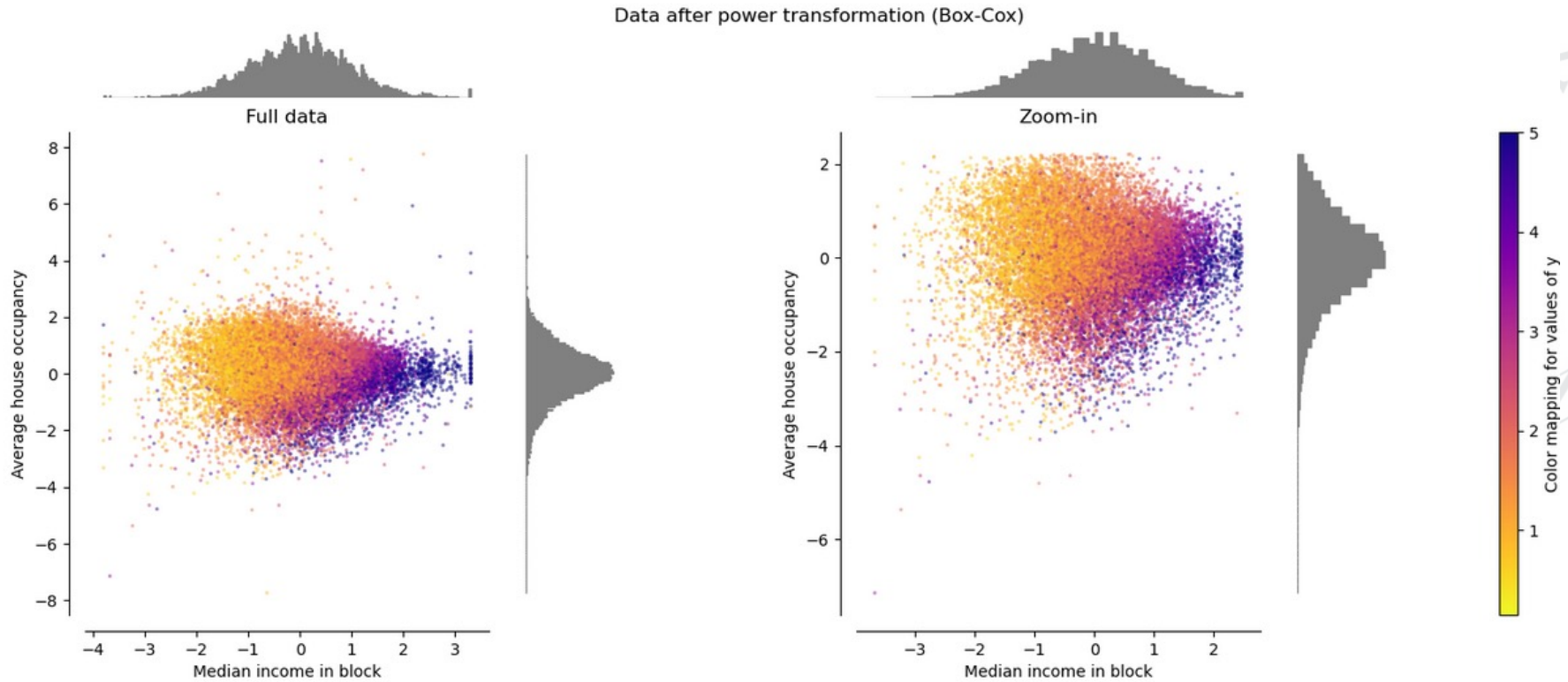
Transformación de datos

- Normalización Robusta



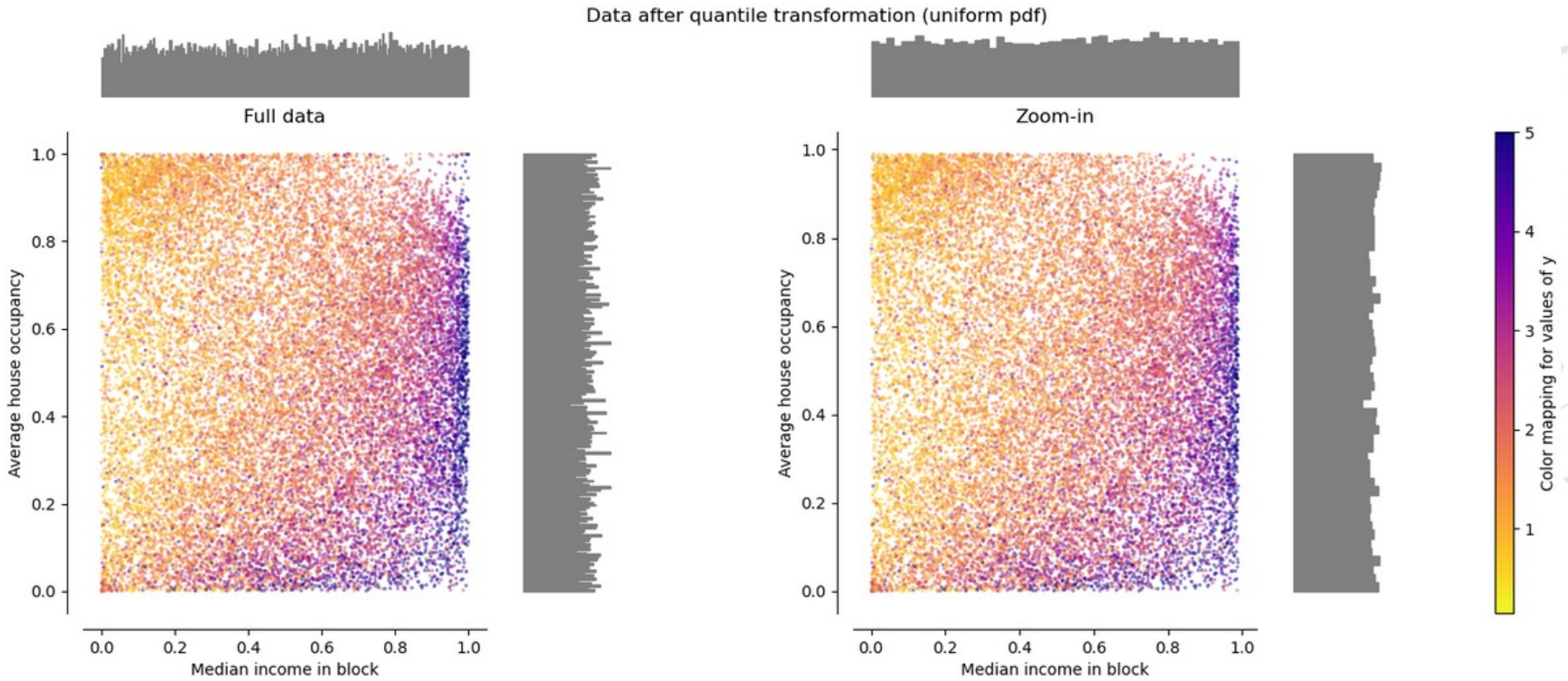
Transformación de datos

- Normalización basada en potencias



Transformación de datos

- Normalización basada en cuartiles



Reducción de la dimensionalidad

- Con frecuencia trabajamos con conjuntos de datos con una **alta dimensionalidad**
  - Tienen una gran cantidad de características
- El proceso de reducir la dimensionalidad consiste en generar una versión simplificada del conjunto de datos que ofrezca la misma información que el conjunto de datos original
- Además, reducir la dimensionalidad también nos permite poder **generar gráficos entendibles**.
  - ¿Como generamos gráficos con 4 o más dimensiones?.
  - Para poder generar un gráfico resumen de un conjunto de datos es necesario reducir la dimensionalidad como mínimo a 3

Reducción de la dimensionalidad

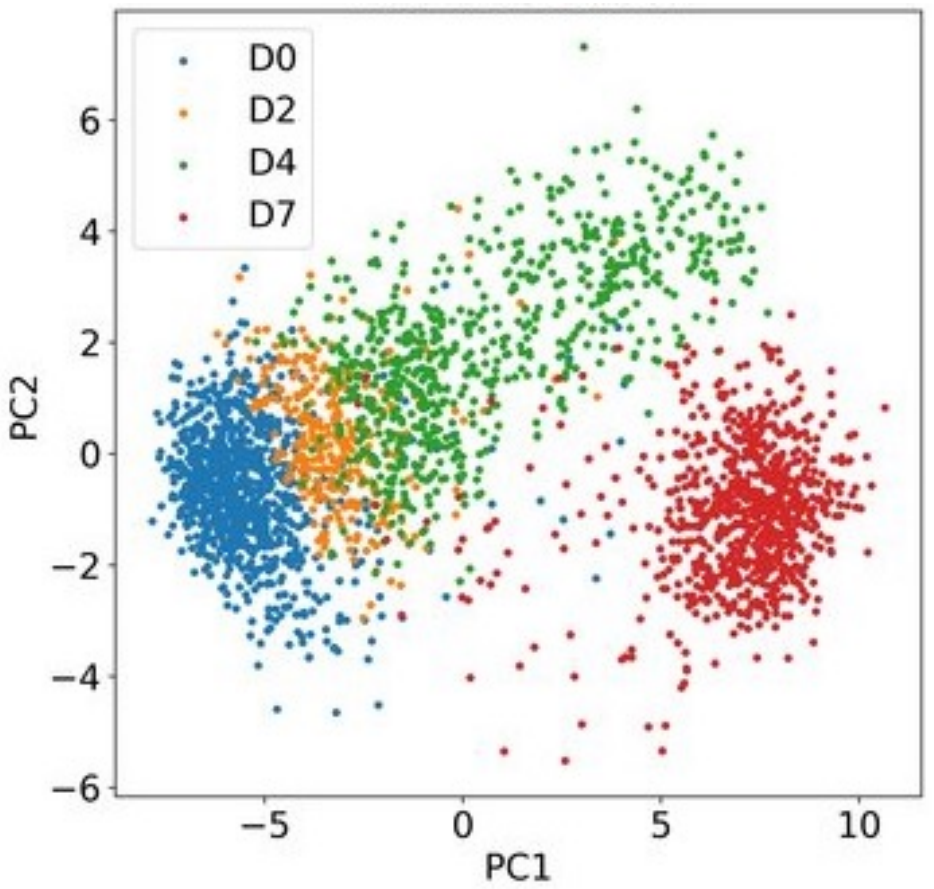
- En este tema vamos a ver dos técnicas que nos van a servir para reducir el número de características en un conjunto de datos
- **Principal Component Analysis** o PCA es un método basado en algebra lineal para reducir la dimensionalidad de un conjunto de datos
- Tras aplicar PCA obtendremos **nuevas características** que reciben el nombre de **componentes principales**
- El objetivo principal de PCA es conservar la **máxima información** posible con la **mínima cantidad de componentes principales**

Reducción de la dimensionalidad

- **El número de componentes principales dependerá de un parámetro** que debemos facilitar. Este parámetro se puede especificar de dos formas
  - Directamente como el número de componentes principales que queremos
  - Como la cantidad de varianza que queremos que sea explicada por los componentes principales
- Una componente principal es una **combinación lineal** de las características originales de tal forma que **la primera componente explica máxima cantidad de varianza** en el conjunto de datos
- La segunda componente principal intentará explicar la varianza restante mientras que se mantiene no correlacionada con la primera componente principal.

Reducción de la dimensionalidad

- Ejemplo de PCA



Reducción de la dimensionalidad

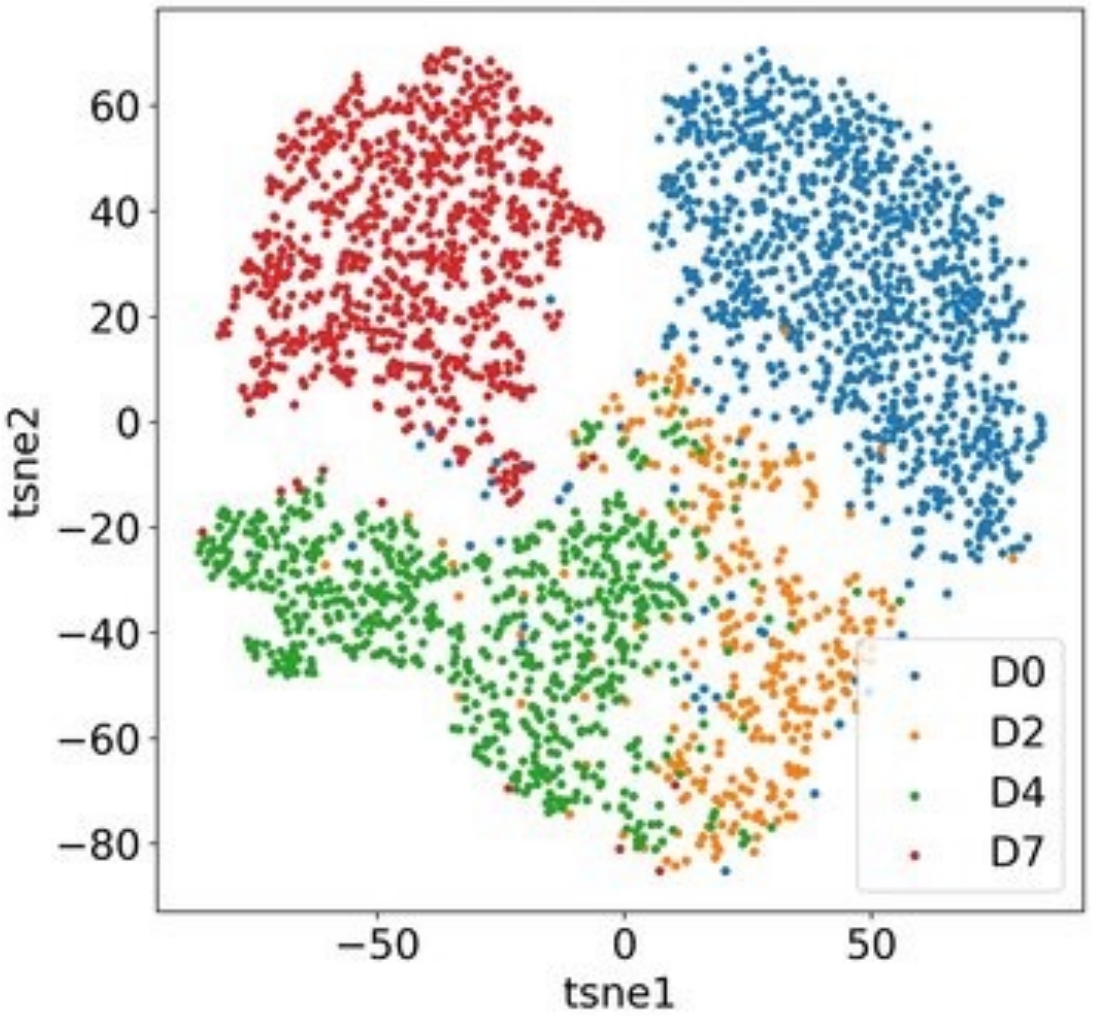
- Otra técnica ampliamente utilizada para reducir la dimensionalidad es **t-distributed stochastic neighbor embedding** o T-SNE
- Esta técnica únicamente tiene fines de representación gráfica. Es decir, no se puede utilizar para extraer nuevas características del conjunto de datos
- A diferencia de PCA, T-SNE es capaz de representar tanto **relaciones lineales como no lineales**
- T-SNE utiliza dos estrategias combinadas para lograr reducir la dimensionalidad de un conjunto de datos

Reducción de la dimensionalidad

- **Enfoque local**. T-SNE **mapea puntos cercanos** en la dimensión original (alta dimensión) **a puntos que se encuentra también cerca** en dimensiones inferiores
- **Enfoque global**. Intenta **preservar la geometría en todas las escalas**. Esto quiere decir que los puntos cercanos entre si se mapearan en puntos cercanos entre si, pero manteniendo los puntos lejanos en la dimensión original en puntos lejanos en dimensiones inferiores
- Si la dimensionalidad del conjunto de datos es muy elevada (superior a 50 características), se aconseja utilizar previamente la técnica PCA, ya que **T-SNE es una técnica muy costosa**

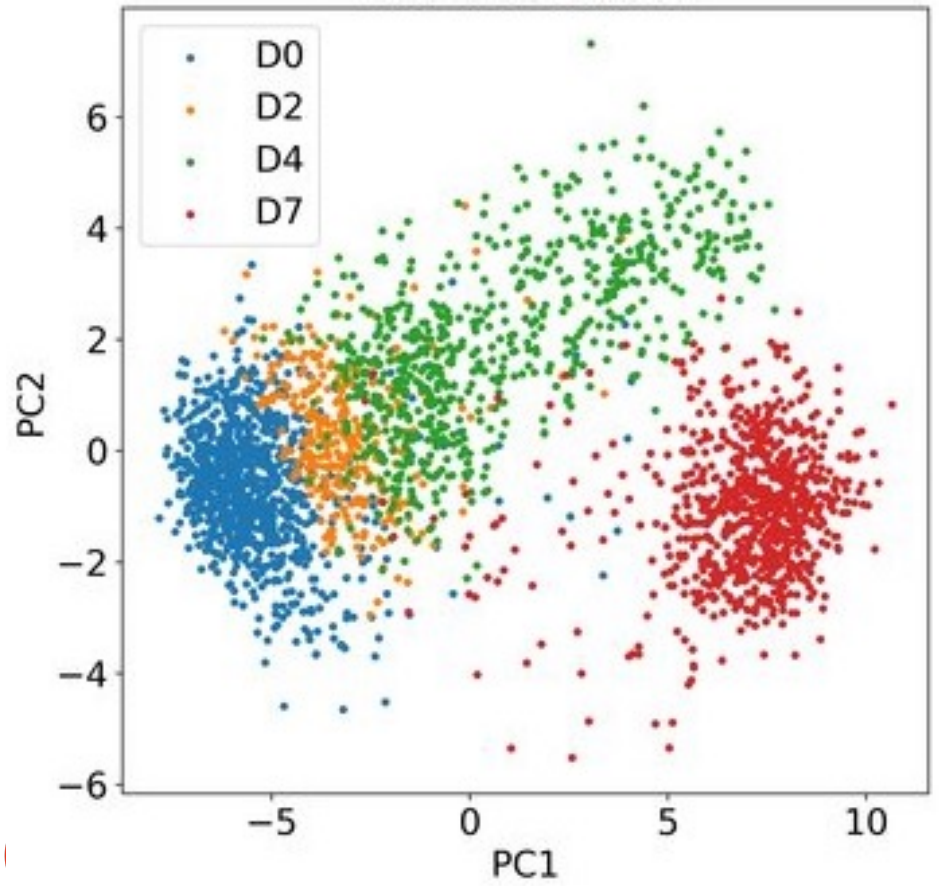
Reducción de la dimensionalidad

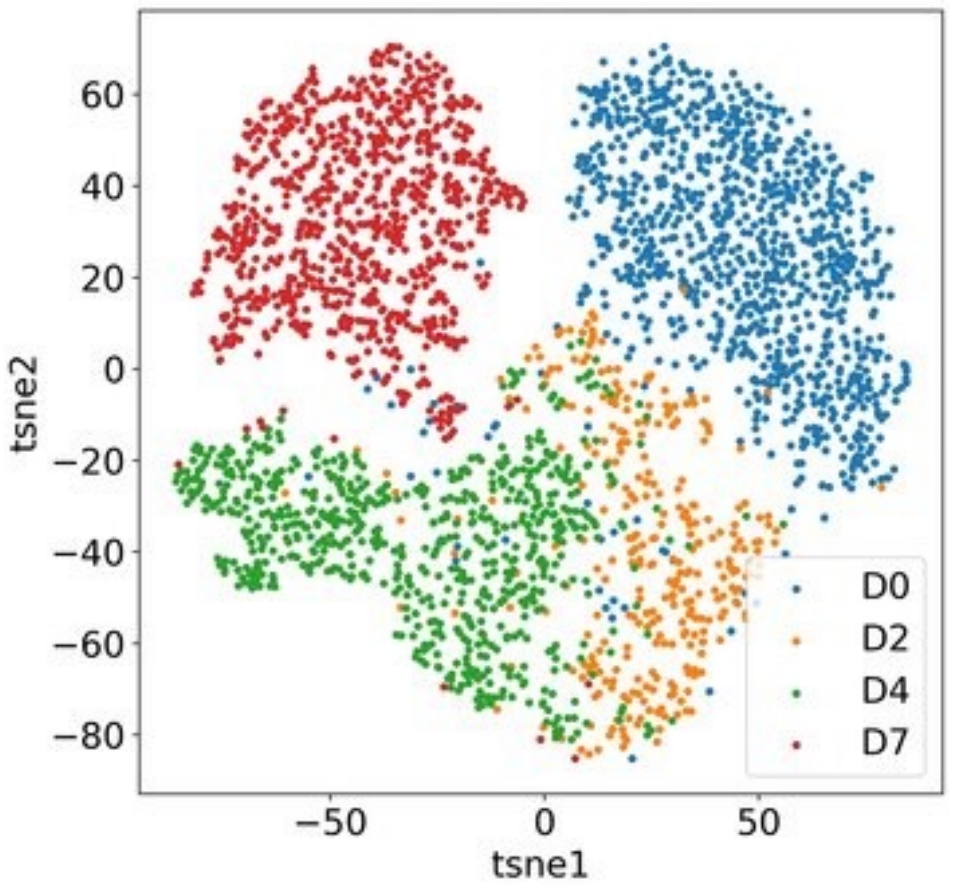
- Ejemplo de T-SNE



Reducción de la dimensionalidad

- Comparativa entre PCA y T-SNE





Discretización de los datos

- Con frecuencia trabajamos con datos que han sido recolectados de **procesos continuos**.
  - Temperatura, humedad, velocidad del viento, luminosidad…
- Muchos otros datos que no son generados de forma automática también son continuos
  - Altura y peso de una persona, nota de un examen, precio de una casa, tamaño de un área geográfica
- Los valores continuos pueden representar una cantidad infinita de valores
- A veces es necesario trabajar con **datos más manejables** y, por tanto, son necesarias técnicas de discretización de los datos

Discretización de los datos

- El objetivo de la discretización es **transformar características continuas en discretas**
- Hay razones de peso por las que en ocasiones es necesario discretizar los datos
  - En ocasiones es **más fácil entender** los valores continuos que **son divididos en una serie de grupos o categorías**
  - Las características **discretas son más fáciles de correlacionar** entre si y, por tanto, es más sencillo identificar los efectos que pueden producir
  - Algunos **modelos** pueden **no** ser **compatibles** completamente con características continuas

Discretización de los datos

- En esta asignatura nos centraremos en dos técnicas de discretización de datos
  - Basada en igual anchura
  - Basada en igual frecuencia

- Aunque existen técnicas más avanzadas como K-means (algoritmo de Machine Learning no supervisado)
  - Se genera un número determinado grupos y mediante medidas de similitud los valores de las características se van concentrando en cada uno de los grupos

Discretización de los datos

- La discretización basada en igual anchura consiste en separar todos los valores posibles en N compartimentos
- **Cada uno de estos compartimentos tiene que tener la misma anchura**

$$anchura = \frac{max - min}{N}$$

- Ejemplo: 5, 6, 8, 12, 11, 15, 4, 7, 10, 13

$$anchura = \frac{15 - 4}{3} = 3.66$$

Discretización de los datos

- El primer compartimento será desde [min, min+width]. En el ejemplo anterior el resultado sería:
  - 4, 5, 6, 7
- El segundo compartimento será desde (min+width, min+2*width]. El resultado del ejemplo anterior sería:
  - 8, 10, 11
- Finalmente, el tercer compartimento será (min+2*width, max]. El resultado del ejemplo anterior sería:
  - 12, 13, 15

Discretización de los datos

- La discretización basada en igual anchura **no mejora la dispersión de los datos**
- **Tampoco gestiona correctamente los valores anómalos**. Ni gestiona de forma correcta los valores de las características que son dispersos. ¿Por qué?
- Puede ser combinada junto a esquemas de codificación de características categóricas
- En general, esta técnica es **adecuada para propósitos de representación gráfica**

Discretización de los datos

- La discretización basada en igual frecuencia consiste en separar todos los posibles valores en N compartimentos (igual que la técnica anterior).
- Sin embargo, aquí se asegura que cada compartimento tenga el mismo número de elementos
- Ejemplo. Queremos generar 4 contenedores a partir de los siguientes datos: 5, 6, 8, 12, 11, 15, 4, 7, 10, 13

$$\text{Número elementos por contenedor} = \frac{M}{N} = \frac{10}{4} = 2.5$$

Discretización de los datos

- La técnica de discretización basada en igual frecuencia **presenta ventajas** sobre la que está basada en igual anchura
- **Puede mejorar la dispersión de los datos**
- **Consigue gestionar de forma efectiva los valores anómalos**
- Puede ser utilizada en combinación con técnicas de codificación categórica

Ejercicios

- Considerando el siguiente conjunto de datos

Nombre	Grupo Sanguíneo	Edad	Peso	Ritmo cardíaco	Presión Sistólica	Presión Diastólica
Eduardo	B+	40	70	74	129	83
Ana	0+	35	69	60	133	86
Alejandro	AB+	37	74	68	125	82
Álvaro	0+	24	70	50	110	70
Aitana	A+	48	72	68	127	82
María	A+	53	67	87	130	84
Sofia	B-	67	65	110	155	100
Antonio	A+	25	74	68	126	89
Fernando	AB+	38	75	77	131	90
Laura	0+	21	70	69	127	87

Ejercicios

1. Codifica la columna Grupo Sanguíneo siguiendo la técnica más adecuada de codificación de valores categóricos
2. Determina los valores anómalos para cada una de las columnas numéricas
3. Codifica cada una de las variables numéricas mediante un normalizador estándar y un normalizador MinMax
4. Aplica alguna técnica de imputación a las características que se muestra resaltadas en amarillo
5. Discretiza la columna de edad siguiendo las 2 técnicas vistas en clase

Ejercicios

- Dadas las siguientes tablas de datos

Nombre	Edad	Ritmo cardíaco
Eduardo	40	74
Ana	35	60
Aitana	48	68
Alejandro	37	68

Nombre	Presión Sistólica	Presión Diastólica
Laura	127	87
Eduardo	129	83
Aitana	127	82
Alejandro	125	82

6. Realiza la unión de dichas tablas usando los mecanismos de Right Join, Left Join, Inner Join y Full Outer Join

Bibliografía

- Grus, J. (2019). *Data science from scratch: first principles with python*. O'Reilly Media, 978-1492041139.
- Osborne, J. (2012). Best Practices in Data Cleaning: A Complete Guide to Everything You Need to Do Before and After Collecting Your Data, SAGE Publications, 978-1412988018.
- Mertz, D. (2021). *Cleaning Data for Effective Data Science: Doing the other 80% of the work with Python, R, and command-line tools*. Packt Publishing Ltd.

Grado en Gestión de la Información y Contenidos Digitales

Procesamiento y Visualización de Datos

Tema 4. Análisis Exploratorio de Datos

Bloque 1. Procesamiento de datos

UNIVERSIDAD DE MURCIA

Facultad de Informática

Introducción

- El Análisis Exploratorio de Datos (EDA, Exploratory Data Analysis) nos ayuda a **comprender mejor los datos** con los que estamos trabajando
- Nos ayuda a **convertir los datos en información útil que podemos presentar en forma de informes**. Por tanto, nos ayuda a presentar la información de forma más estructurada
- También ayuda para tomar decisiones a la hora de desarrollar un modelo de Inteligencia Artificial en una fase posterior
- Podemos destacar dos tipos de EDA que se complementan
  - Resúmenes estadísticos
  - Visualización de los datos

Introducción

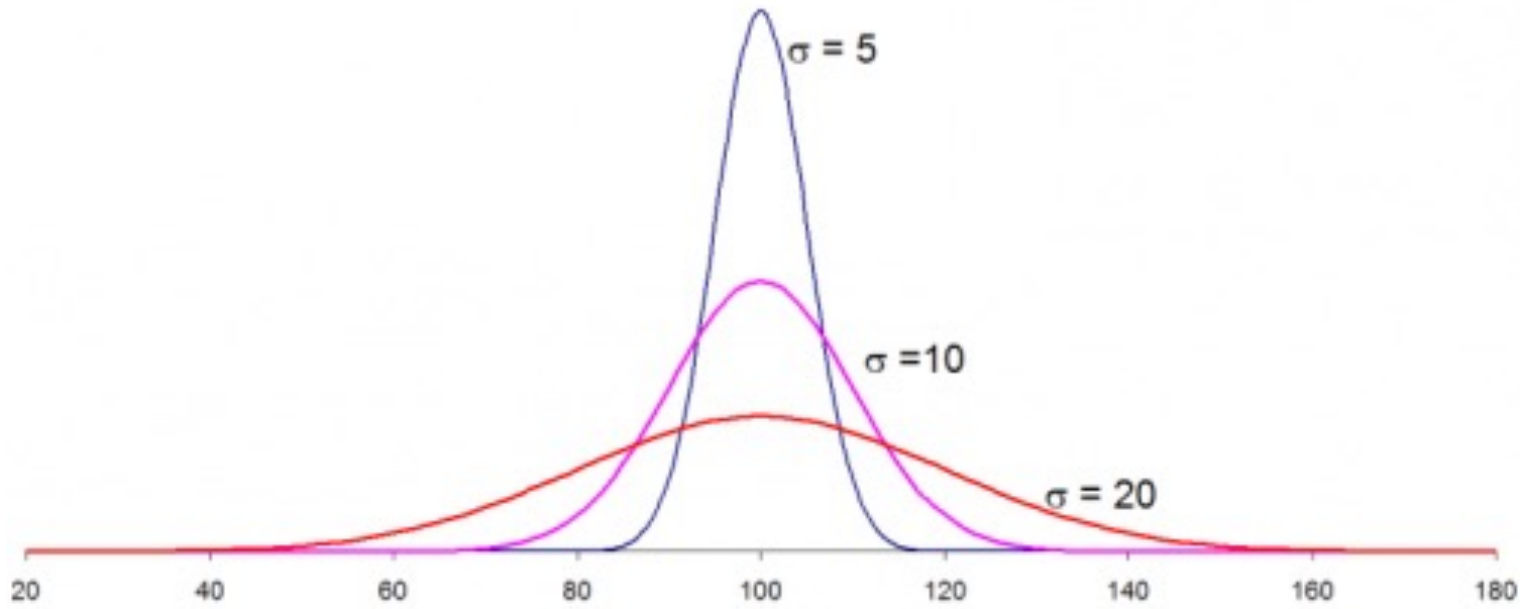
- A partir de un EDA podremos identificar una serie de propiedades sobre los datos que nos ayudarán a aplicar las técnicas que hemos visto en el tema anterior
- Durante un EDA se descubrirá que muestras tienen valores faltantes
- También identificaremos el rango de cada una de las características para poder aplicar un escalado de los datos
- Durante esta etapa también identificaremos que características pueden tener un valor anómalo y aplicar las técnicas de IQR y del umbral basado en media y desviación típica vistos en el tema anterior

Introducción

- Además, durante un EDA también **identificaremos propiedades interesantes de los datos** y otros problemas asociados
  - Tendencia central
  - Desviación de los datos
  - Sesgos
  - Valores anómalos
- También usaremos EDA para identificar **posibles relaciones entre características** (correlación)

Resúmenes estadísticos

- Recordemos una de las distribuciones de datos más importantes



Resúmenes estadísticos

- La media de un conjunto de n muestras de una variable se denota como $\bar{x}$ y es definida de la siguiente forma

$$\bar{x} = \frac{x_1 + x_2 + x_3 + \ \ldots \ + x_n}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- La media **describe como es el valor de una muestra típica** dentro de un conjunto de datos determinado. También **determina cual es el valor central** de una distribución de datos

Resúmenes estadísticos

- La mediana de una característica, ordenadas según su valor, dentro de un conjunto de n muestras se define de la siguiente forma

$$Mediana = \begin{cases} x_{\left\lfloor \frac{n}{2} \right\rfloor + 1}, & Si\ n\ es\ par \\ \dfrac{x_{\frac{n}{2}} + x_{\frac{n}{2}+1}}{2}, & Si\ n\ es\ impar \end{cases}$$

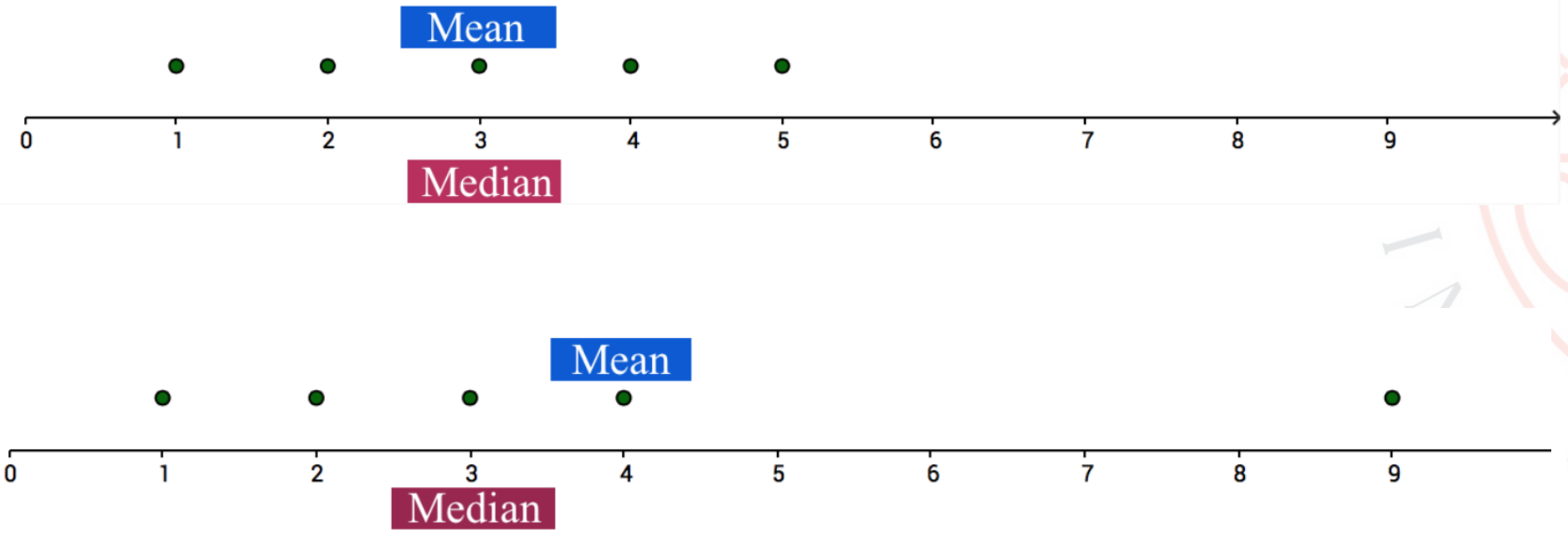
- Ejemplo. Edades: 17, 19, 21, 22, 23, 23, 23, 38

$$Mediana = \frac{22 + 23}{2} = 22.5$$

- La mediana **describe como es una muestra típica** dentro de un conjunto de datos determinado. También **determina cual es el centro de una distribución de muestras**

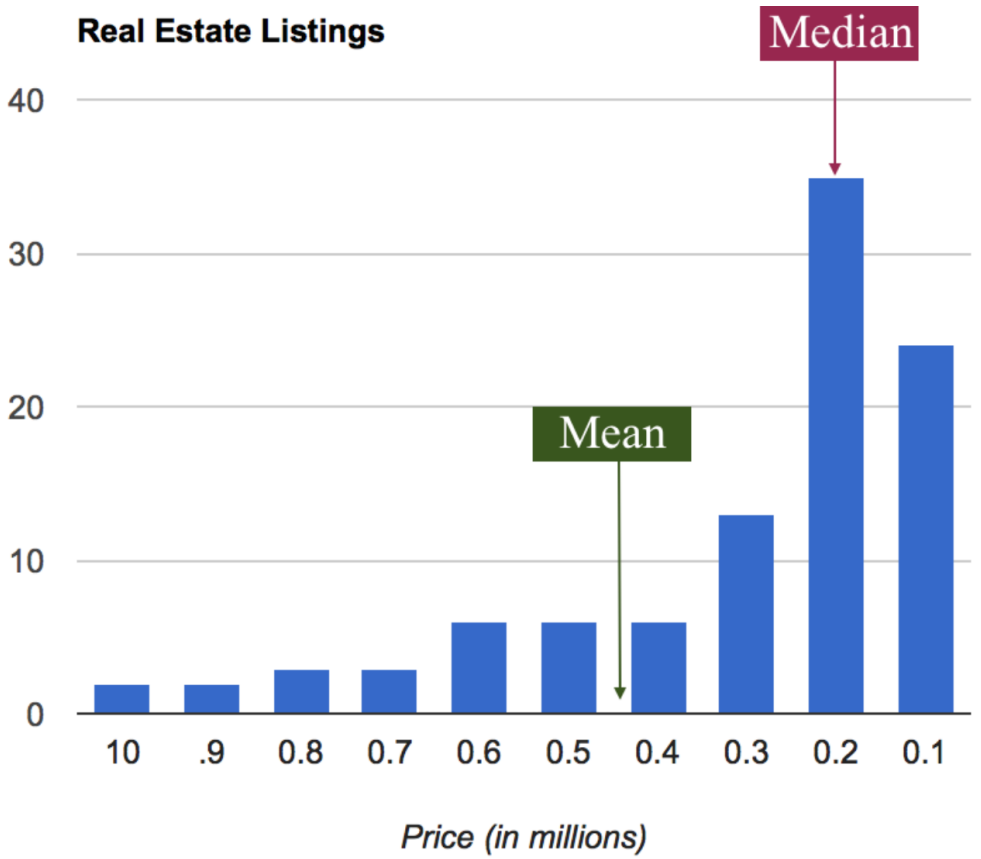
Resúmenes estadísticos

- La media es sensible a valores anómalos



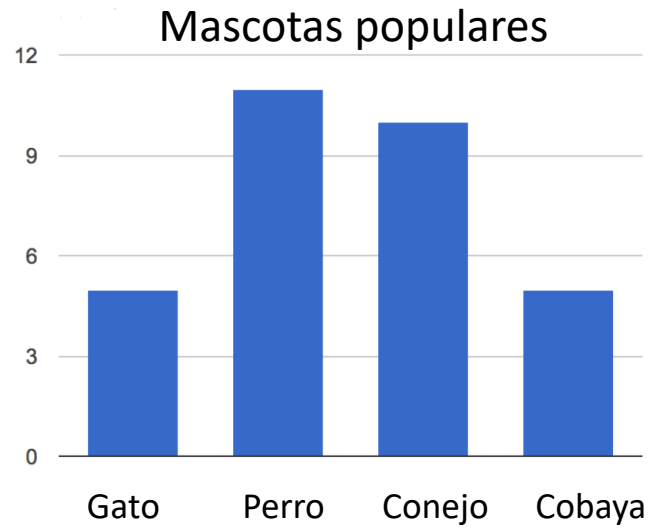
Resúmenes estadísticos

- La media también es sensible a la asimetría de los datos



Resúmenes estadísticos

- Para características con valores categóricos, la media y la mediana no tienen sentido



- En estos casos, **la moda es una medida más representativa**

Resúmenes estadísticos

- **La dispersión de las muestras mide como de bien la media o la mediana describen el conjunto de muestras**
- Una medida para cuantificar la dispersión de las muestras es el rango que se define de la siguiente forma

$$Rango = Valor\ Máximo\ - Valor\ Mínimo$$

Resúmenes estadísticos

- Otra medida útil para cuantificar la dispersión de los datos es la **varianza**
- La varianza, σ^2, indica la **diferencia media entre cada una de las muestras y la media**. Se define de la siguiente forma

$$\sigma^2 = \frac{\sum_{i=1}^{n} |x_i - \bar{x}|^2}{n-1}$$

- El término $|x_i - \bar{x}|$ cuantifica la desviación de una muestra cualquier, x_i, con respecto a la media, $\bar{x}$. El hecho de elevarlo al cuadrado hace que esta medida sea sensible a valores anómalos
- La varianza, σ^2, **no se representa en las mismas unidades** que x_i

Resúmenes estadísticos

- Otra medida de dispersión es la **desviación estándar** o **desviación típica**, σ. Se calcula como la raíz cuadrada de la varianza, σ^2

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} |x_i - \bar{x}|^2}{n-1}}$$

- En este caso, la desviación típica, σ, **si que se representa en las mismas unidades que x_i**

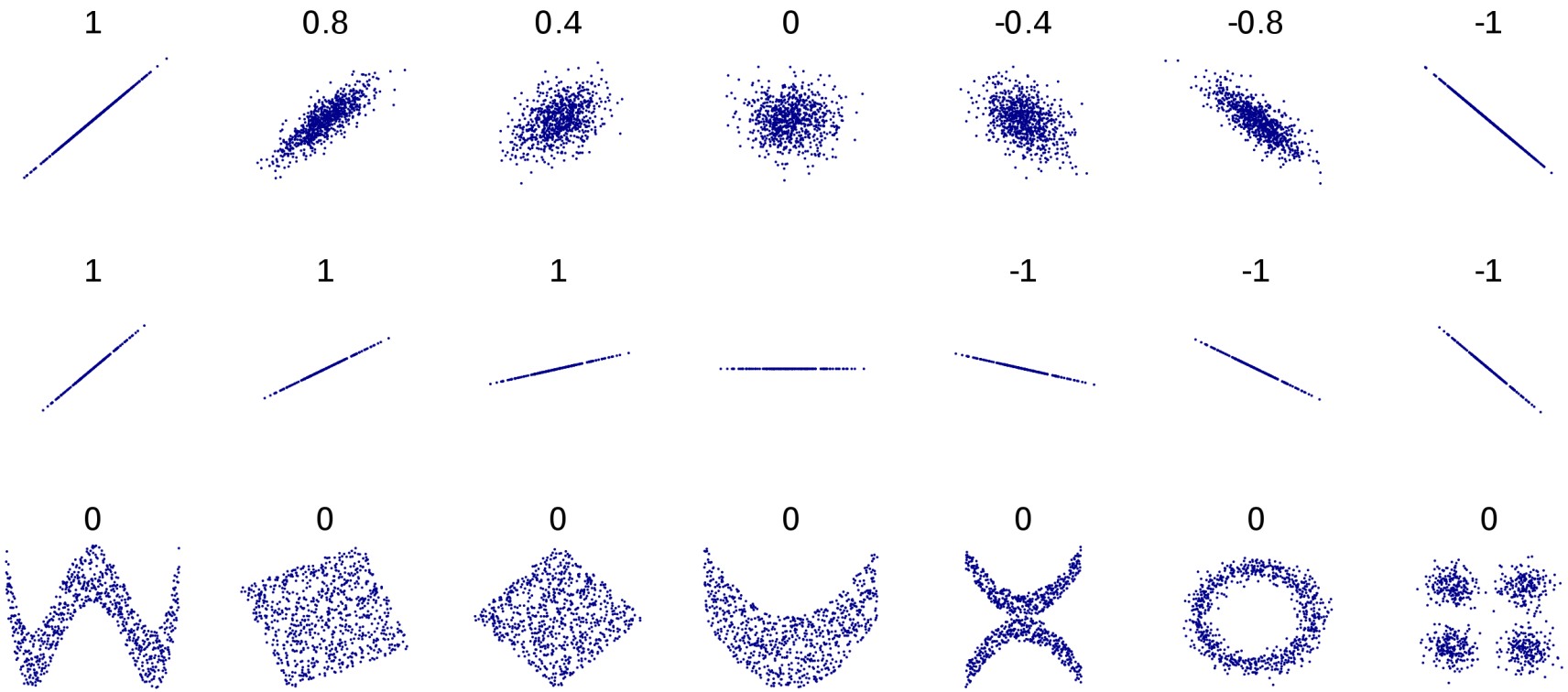
Resúmenes estadísticos

- Otra medida importante es la **relación entre cada par de características** en un conjunto de datos
- Una medida ampliamente utilizada es la **correlación de Pearson**. También conocida como los coeficientes de correlación de Pearson

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

Resúmenes estadísticos

- La correlación de Pearson solo identifica relaciones lineales



Resúmenes estadísticos

- Para solventar la limitación de la correlación de Pearson, existen otras medidas de similitud más avanzadas
- La **correlación de distancia** (dCor) mide la dependencia entre dos conjuntos de características de dimensión arbitraria y no necesariamente iguales
- La medida de **Información Mutua** (MI) mide la información o reducción de la incertidumbre de una característica determinada cuando se conoce otra característica distinta
- Los **Coeficientes de Información Máxima** (MIC) son una medida de la dependencia lineal o no lineal entre dos características cualquiera

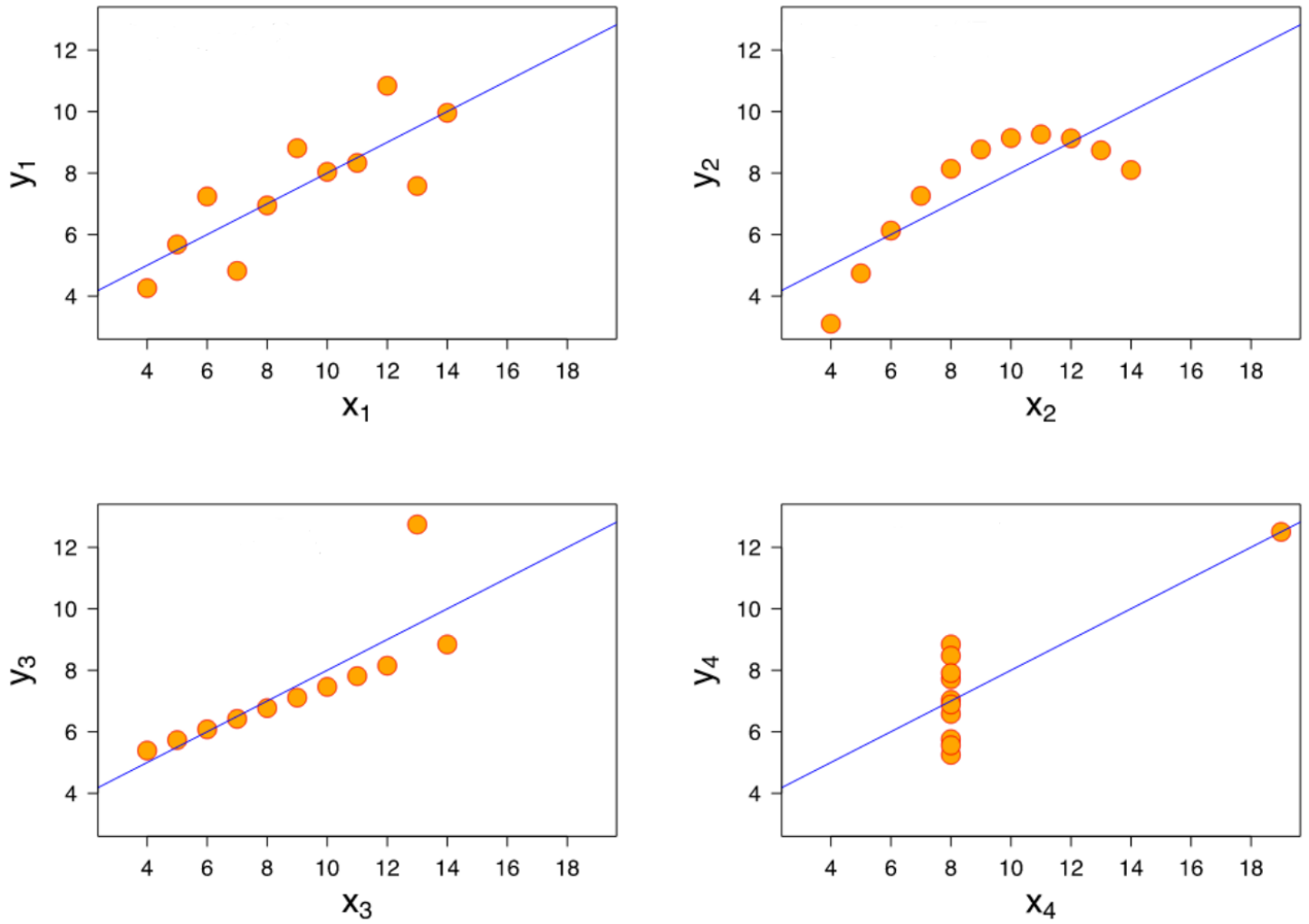
Visualización de datos

- ¿Por qué es necesaria?

	Conjunto de Datos I		Conjunto de Datos II		Conjunto de Datos III		Conjunto de Datos IV	
	x	y	x	y	x	y	x	y
	10	8.04	10	9.14	10	7.46	8	6.58
	8	6.95	8	8.14	8	6.77	8	5.76
	13	7.58	13	8.74	13	12.74	8	7.71
	9	8.81	9	8.77	9	7.11	8	8.84
	11	8.33	11	9.26	11	7.81	8	8.47
	14	9.96	14	8.1	14	8.84	8	7.04
	6	7.24	6	6.13	6	6.08	8	5.25
	4	4.26	4	3.1	4	5.39	19	12.5
	12	10.84	12	9.13	12	8.15	8	5.56
	7	4.82	7	7.26	7	6.42	8	7.91
	5	5.68	5	4.74	5	5.73	8	6.89
Suma	99.00	82.51	99.00	82.51	99.00	82.51	99.00	82.51
Media	9.00	7.50	9.00	7.50	9.00	7.50	9.00	7.50
Desviación típica	3.32	2.03	3.32	2.03	3.32	2.03	3.32	2.03

Visualización de datos

- Sin embargo …

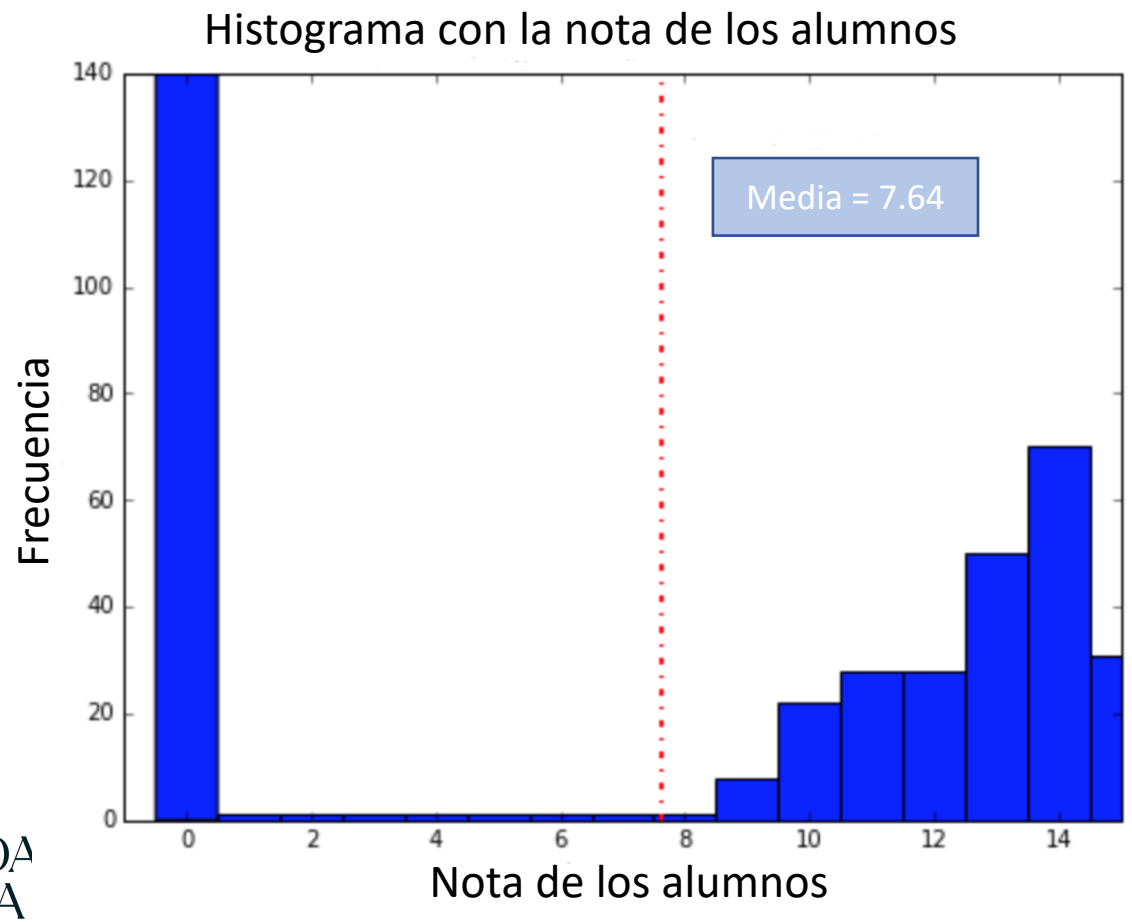


Visualización de datos

Si la media de un conjunto de exámenes es 7.64 sobre 15 ¿Qué sugiere?

Visualización de datos

- Y ahora ¿Qué sugiere?

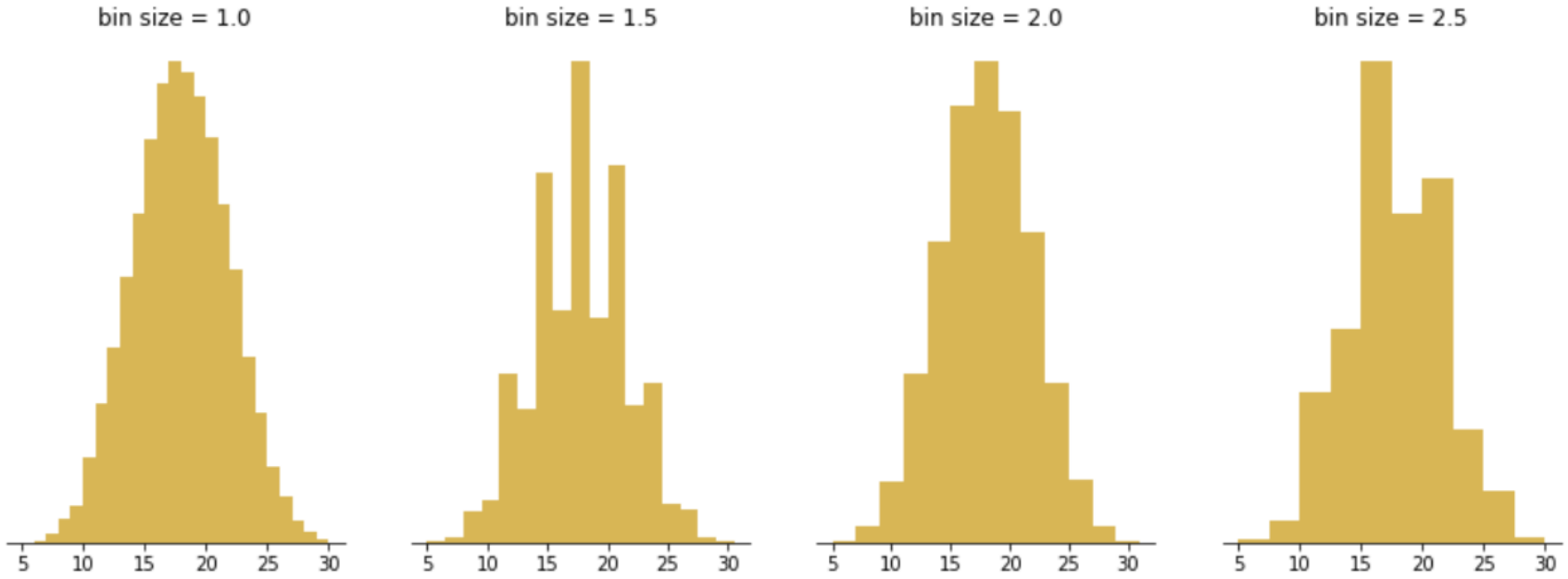


Visualización de datos

- Existen una gran cantidad de tipos de gráficos para representar una amplia variedad de datos y se pueden agrupar en los siguientes grupos
- **Distribución**. Para mostrar cómo una característica se distribuye entre unos determinados rangos
- **Relación**. Para mostrar como múltiples características se relacionan entre si
- **Composición**. Para mostrar como los valores de una característica se descompone en grupos
- **Comparación**. Para comparar los valores de distintas características

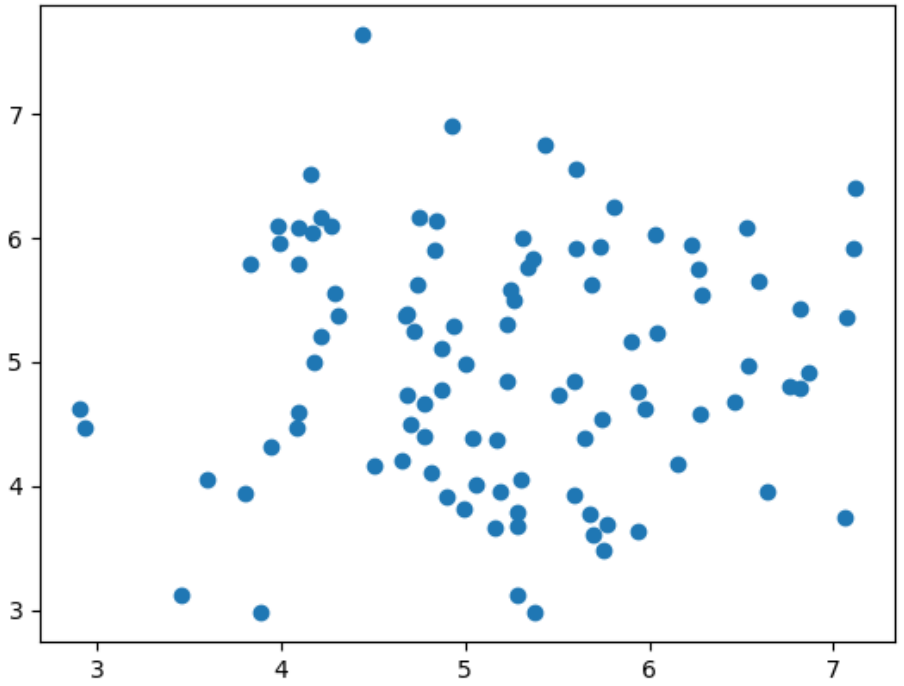
Visualización de datos

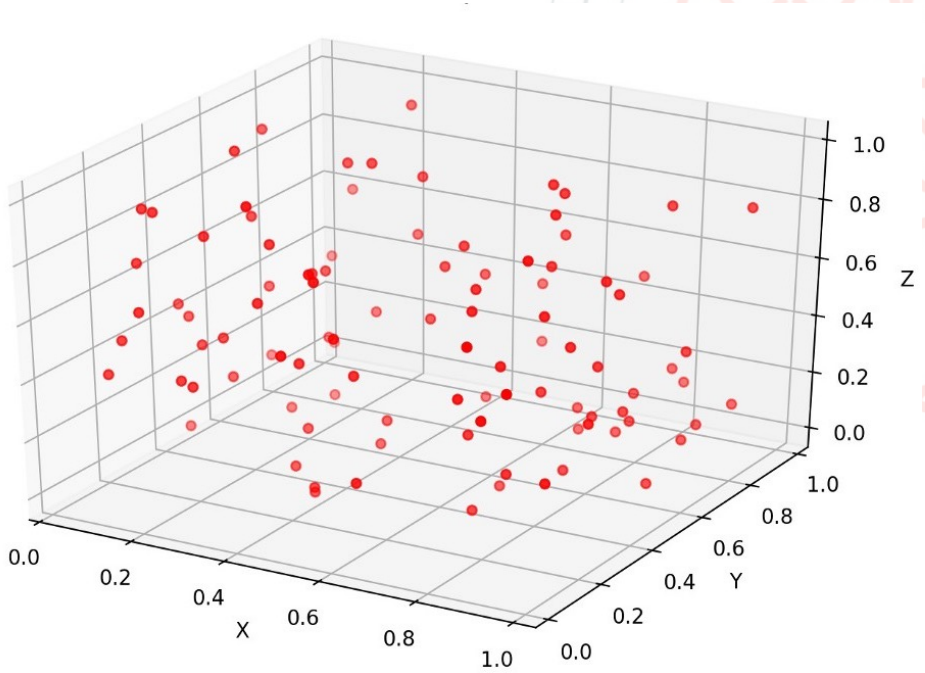
- Dentro de la categoría de distribución encontramos los histogramas
- Sirven para visualizar como los datos se distribuyen entre un rango determinado de valores



Visualización de datos

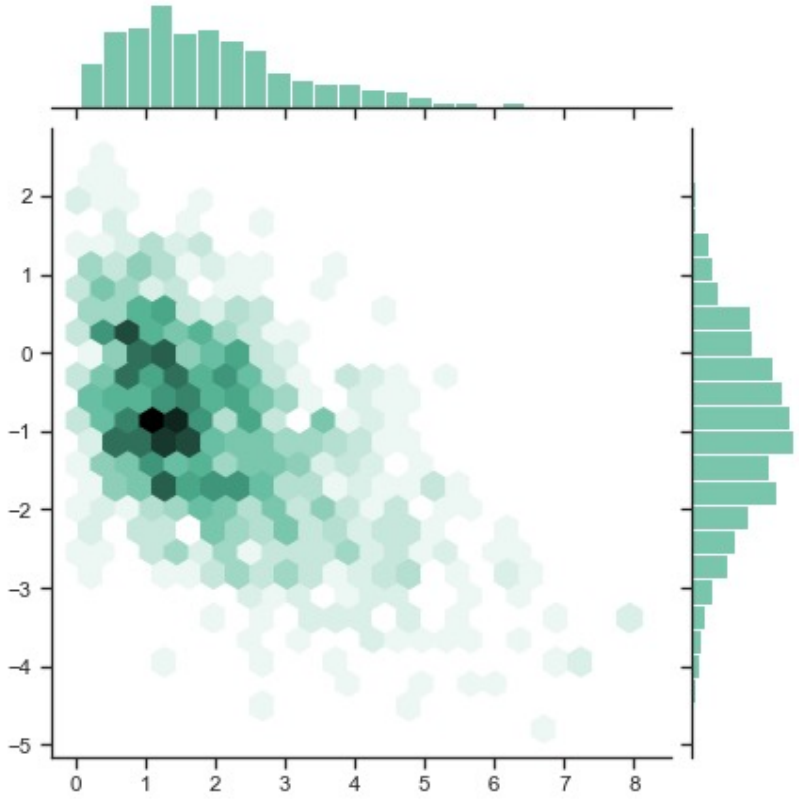
- Los gráficos de dispersión o scatter plot también nos dan una idea de la distribución de los datos en 2 o 3 dimensiones





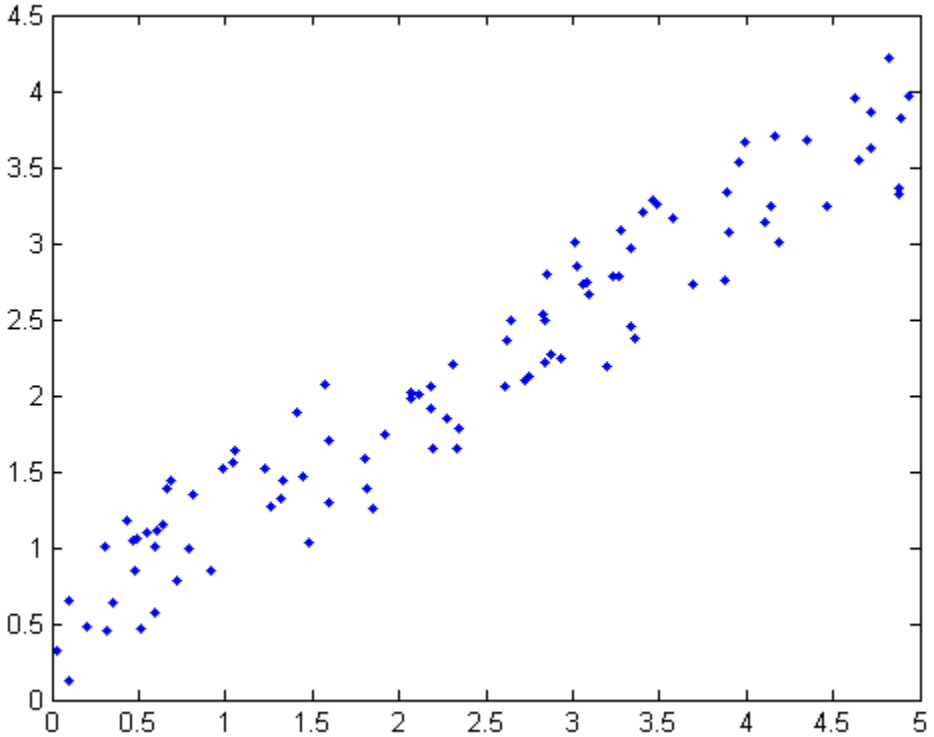
Visualización de datos

- También hay gráficos de distribución más complejos que mezclan los gráficos de dispersión con los histogramas



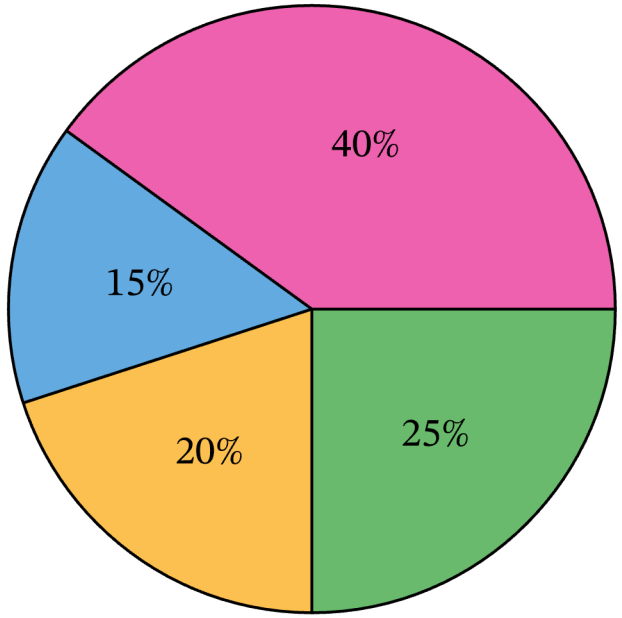
Visualización de datos

- Los gráficos de dispersión también nos ayudan a identificar relaciones entre los datos



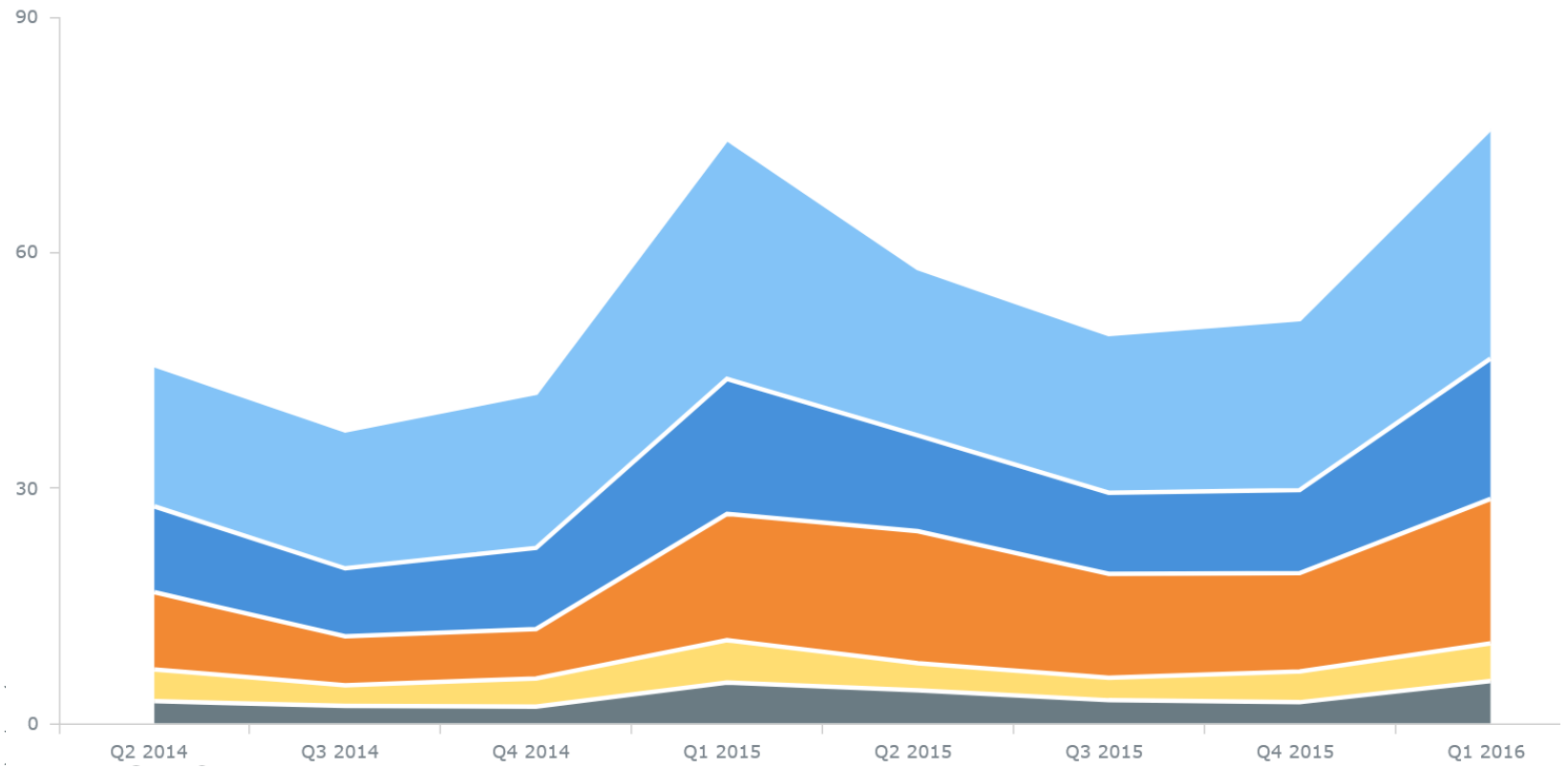
Visualización de datos

- Los gráficos de tarta nos permiten entender la composición de un todo



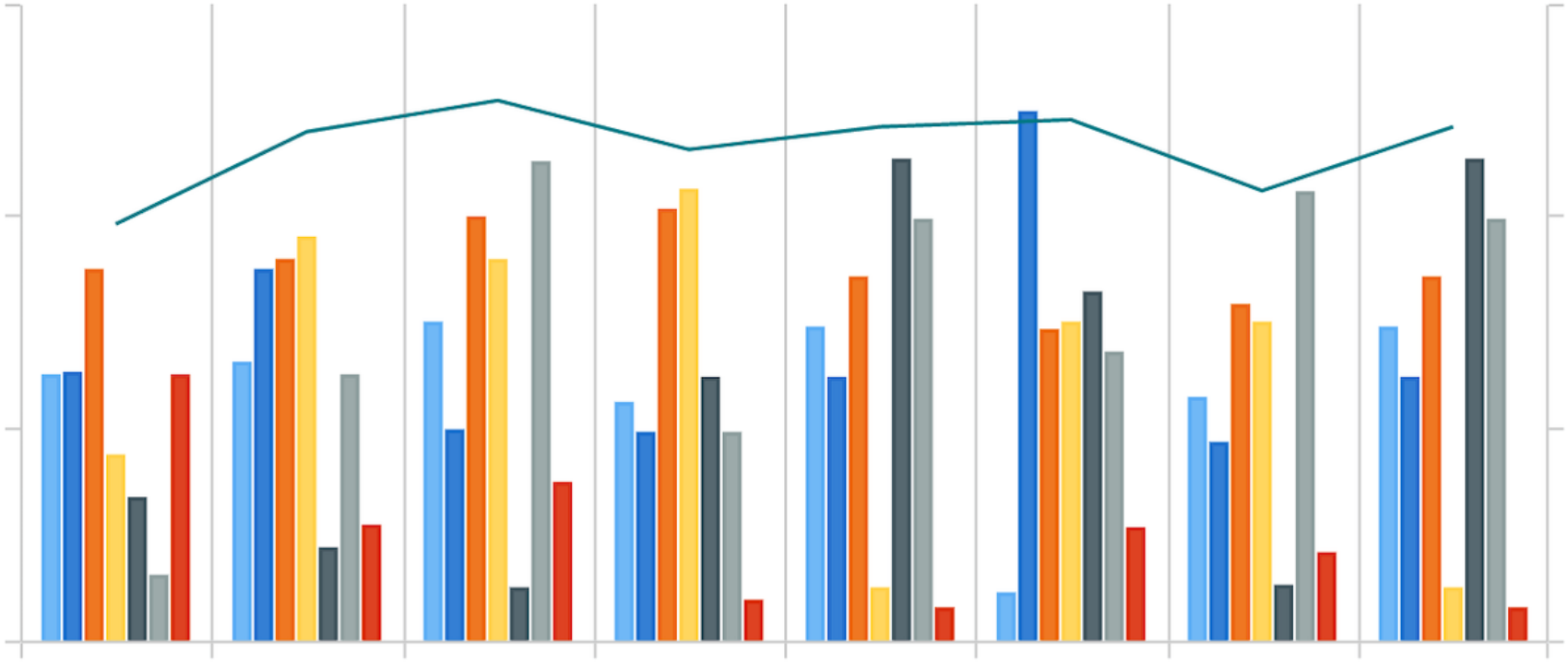
Visualización de datos

- De forma similar, los gráficos compuestos de área también nos permiten hacernos una idea de la proporción de cada elemento dentro de un todo



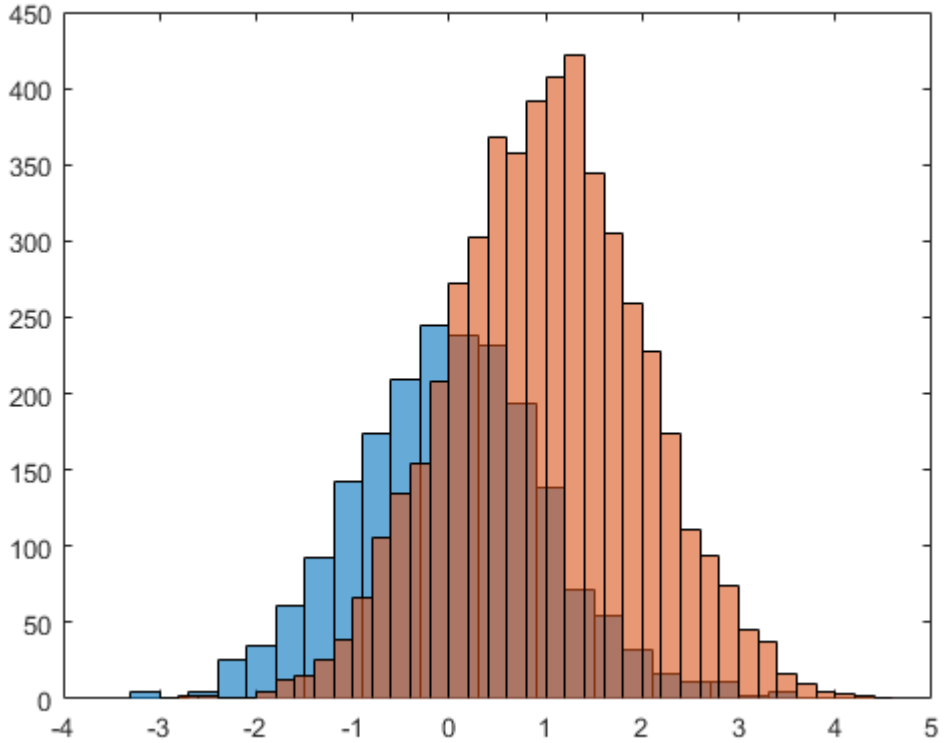
Visualización de datos

- Finalmente, los gráficos de barras nos permiten comparar los distintos valores de ciertas características



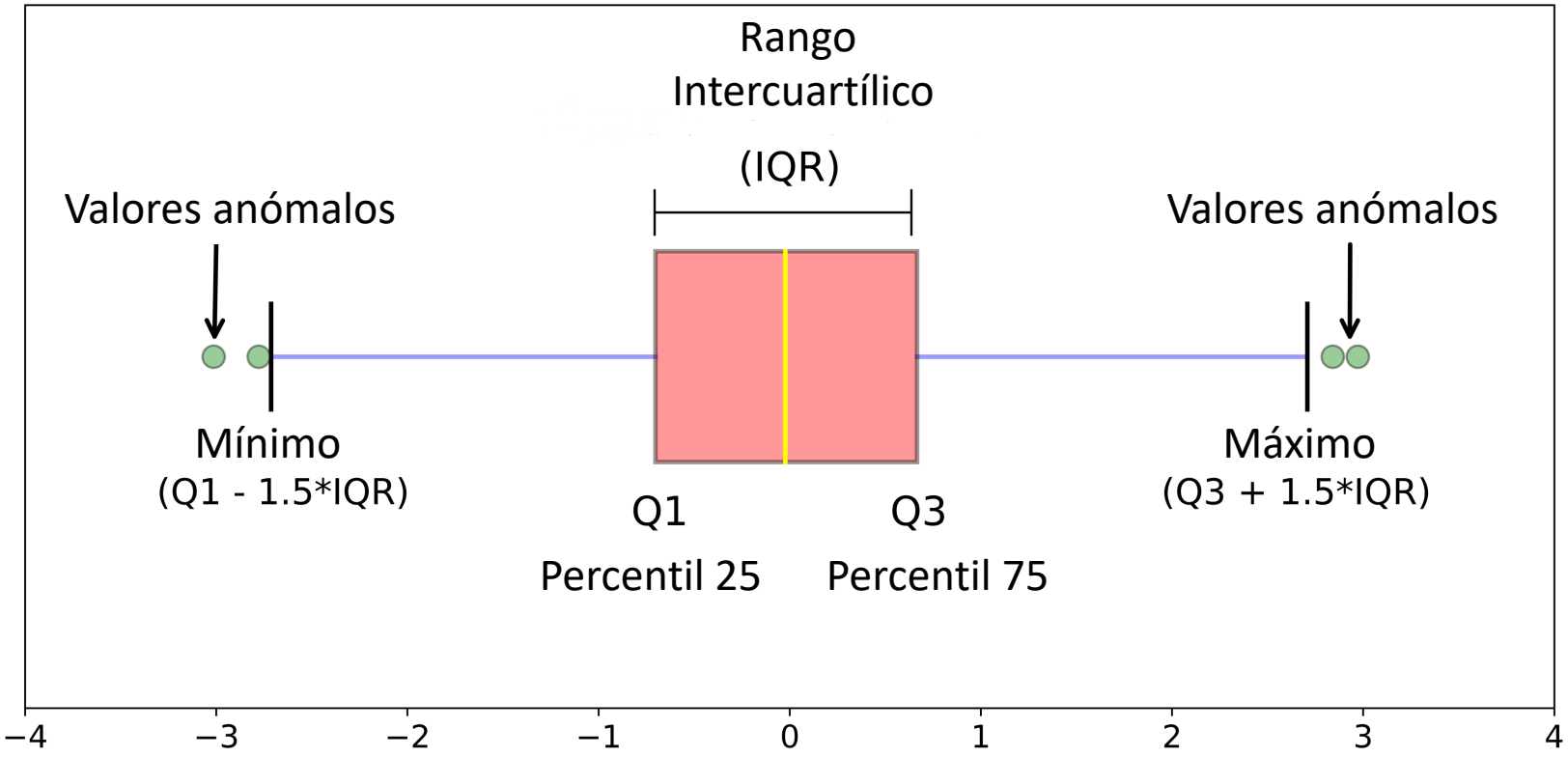
Visualización de datos

- Los histogramas múltiples también nos permiten comparar las distribuciones de distintas características



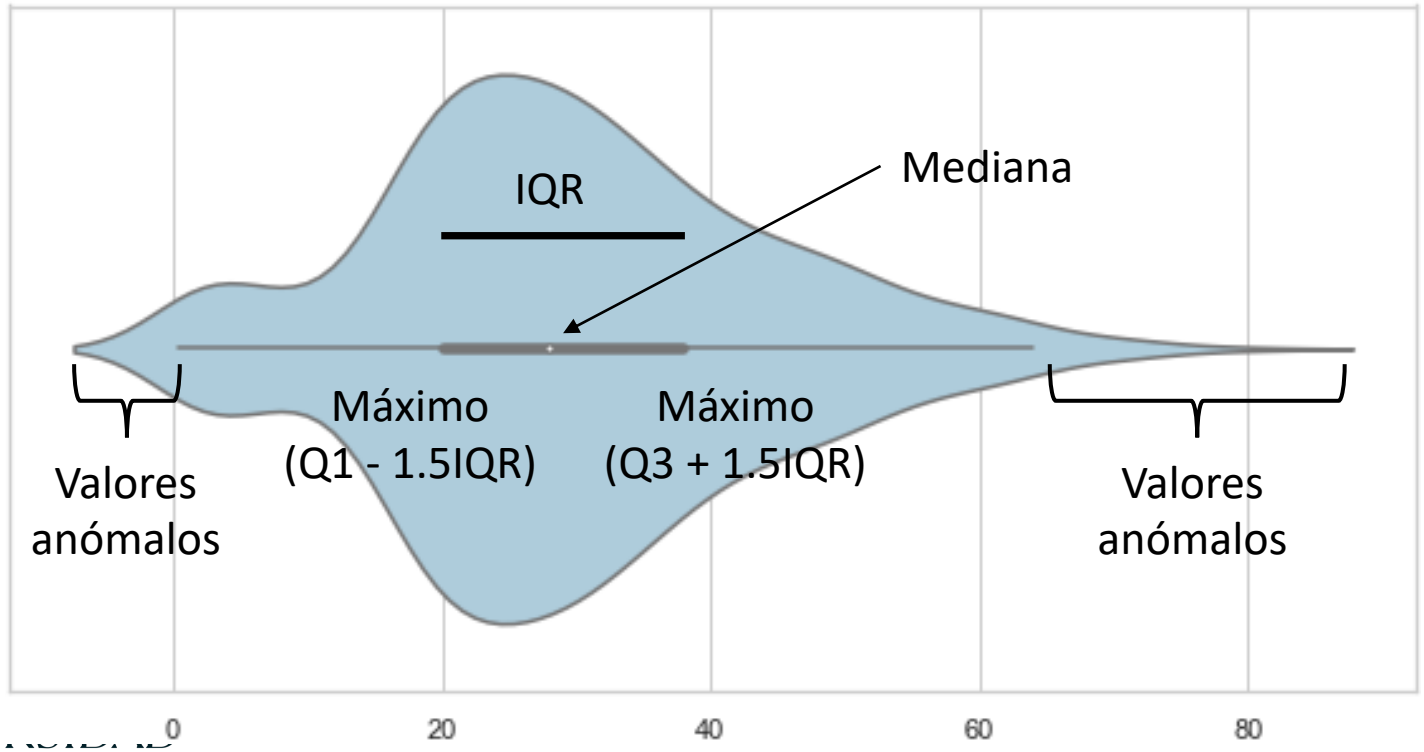
Visualización de datos

- Los gráficos de cajas dan información importante



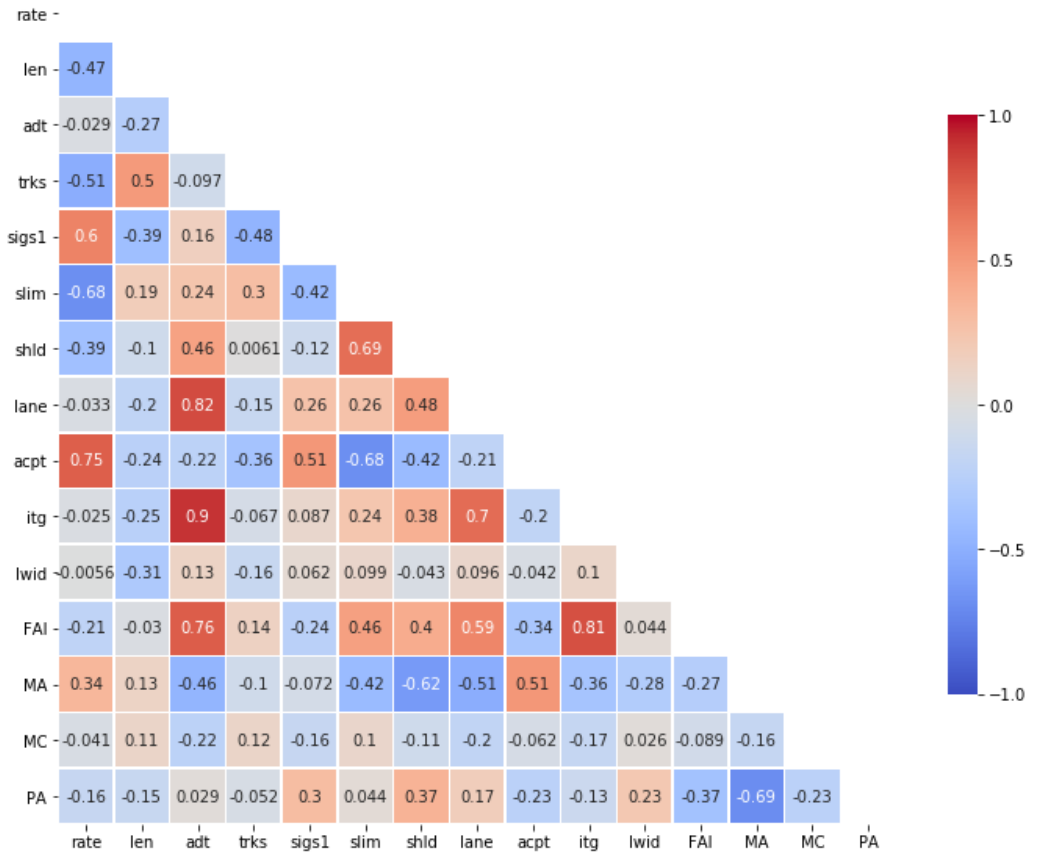
Visualización de datos

- Los gráficos de violín son una forma más avanzada de representación ya que permiten mostrar además la **distribución de los datos**



Visualización de datos

- Por último, los mapas de calor son útiles para mostrar la correlación entre cada par de características en un conjunto de datos



UNIVERSIDAD DE MURCIA

Ejercicios

1. Calcula la media de los siguientes valores
   - 12, 15, 20, 5, 7, 12, 4, 20, 13, 2, 200
2. Calcula la mediana de los siguientes valores
   - 12, 15, 20, 5, 7, 12, 4, 20, 13, 2, 200
   - 30, 40, 35, 60, 1, 200, 42, 33, 45, 37, 39
   - 20, 24, 32, 31, 31, 12, 17, 8
3. Calcula la desviación típica y la varianza de cada una de las series anteriores
4. Calcula la moda de los siguientes valores
   - 1, 3, 5, 3, 1, 2, 2, 5, 6, 2, 8, 2, 4, 1, 2, 7, 6, 5, 3, 4, 5, 5, 2

Ejercicios

5. Calcula la correlación de Pearson entre los siguientes valores

1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

6. Vuelve a calcular la correlación de Pearson usando ahora los siguientes valores

1	2	3	4	5	6	7	8	9	10
10	9	8	7	6	5	4	3	2	1

Ejercicios

7. Calcula una vez más la correlación de Pearson para los siguientes valores

7	12	5	2	3	7	15	17	6	20
3	4	3	14	16	9	8	23	4	33

8. Si queremos representar el valor de los distintos pixeles de una imagen ¿Qué tipo de gráfico usaremos?
9. Si queremos representar los valores de un sensor de temperatura a lo largo del tiempo ¿Qué tipo de gráfico sería el adecuado?

Bibliografía

- Grus, J. (2019). *Data science from scratch: first principles with python*. O'Reilly Media, 978-1492041139.
- Osborne, J. (2012). Best Practices in Data Cleaning: A Complete Guide to Everything You Need to Do Before and After Collecting Your Data, SAGE Publications, 978-1412988018.
- Mertz, D. (2021). *Cleaning Data for Effective Data Science: Doing the other 80% of the work with Python, R, and command-line tools*. Packt Publishing Ltd.

Grado en Gestión de la Información y Contenidos Digitales

Procesamiento y Visualización de Datos

Práctica 1. Notebooks

Bloque 1: Procesamiento de datos

UNIVERSIDAD DE MURCIA

Facultad de Informática

Introducción

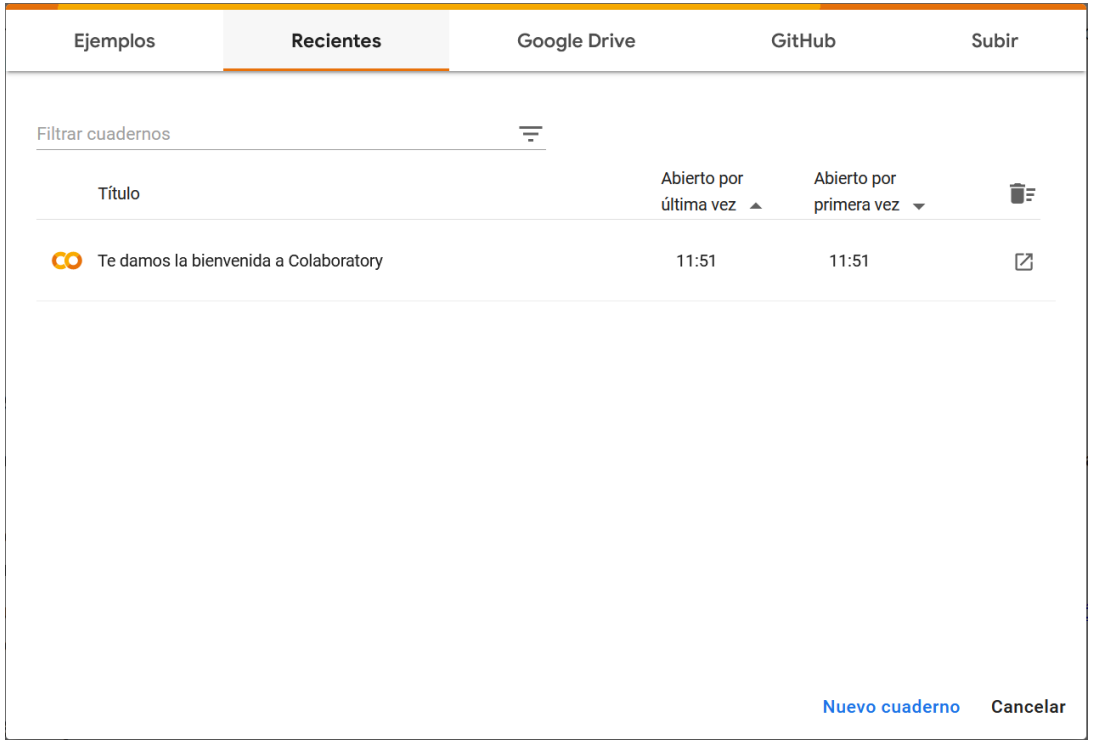
- Podemos programar en Python directamente usando editores de texto
  - Bloc de notas
  - Sublime text
- Sin embargo, existen Entornos de Desarrollo Integrados (**IDE**, en inglés) que facilitan el trabajo de programación de Python
  - Spyder
  - PyCharm
  - Visual Studio
- Estos entornos nos proporcionan funciones como autocompletado o indicación de errores

Introducción

- En esta asignatura vamos a usar **notebooks de Python**
- Los notebooks hacen referencia a **Jupyter Notebook**
  - Aplicación de código abierto basada en tecnologías web que permite crear y compartir documentos que contienen código, ecuaciones, gráficos y textos
- Los notebook de Jupyter se pueden ejecutar sobre un servidor configurado por el usuario
- Sin embargo, en esta asignatura vamos a hacer uso de la herramienta **Google Colab** para poder crear notebooks

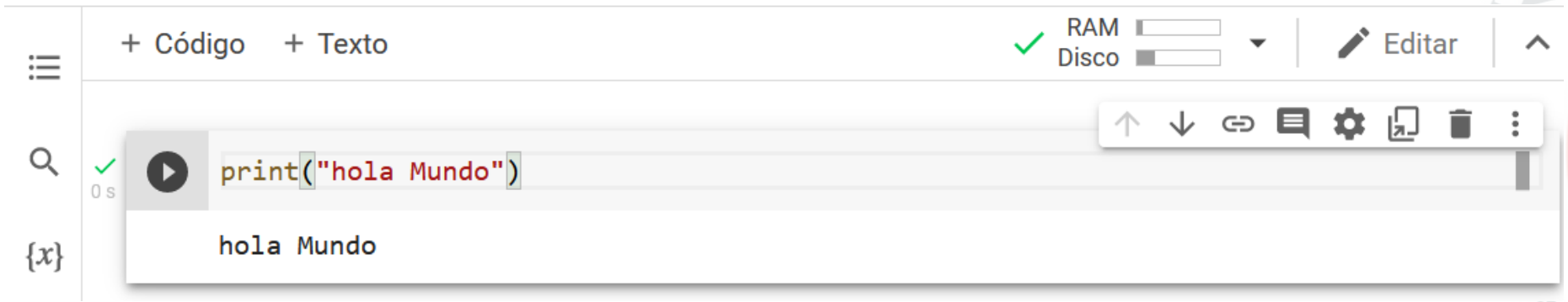
Google Colab

- El primer paso es ir a https://colab.research.google.com/?hl=es donde nos aparecerá una pantalla como la siguiente



Google Colab

- Si hacemos clic sobre el botón que pone **Nuevo Cuaderno** se nos creará un nuevo notebook listo para trabajar sobre él



- Las cajas de texto donde se permite escribir código se denominan **Celdas** y para obtener el resultado es necesario evaluarlas

Google Colab

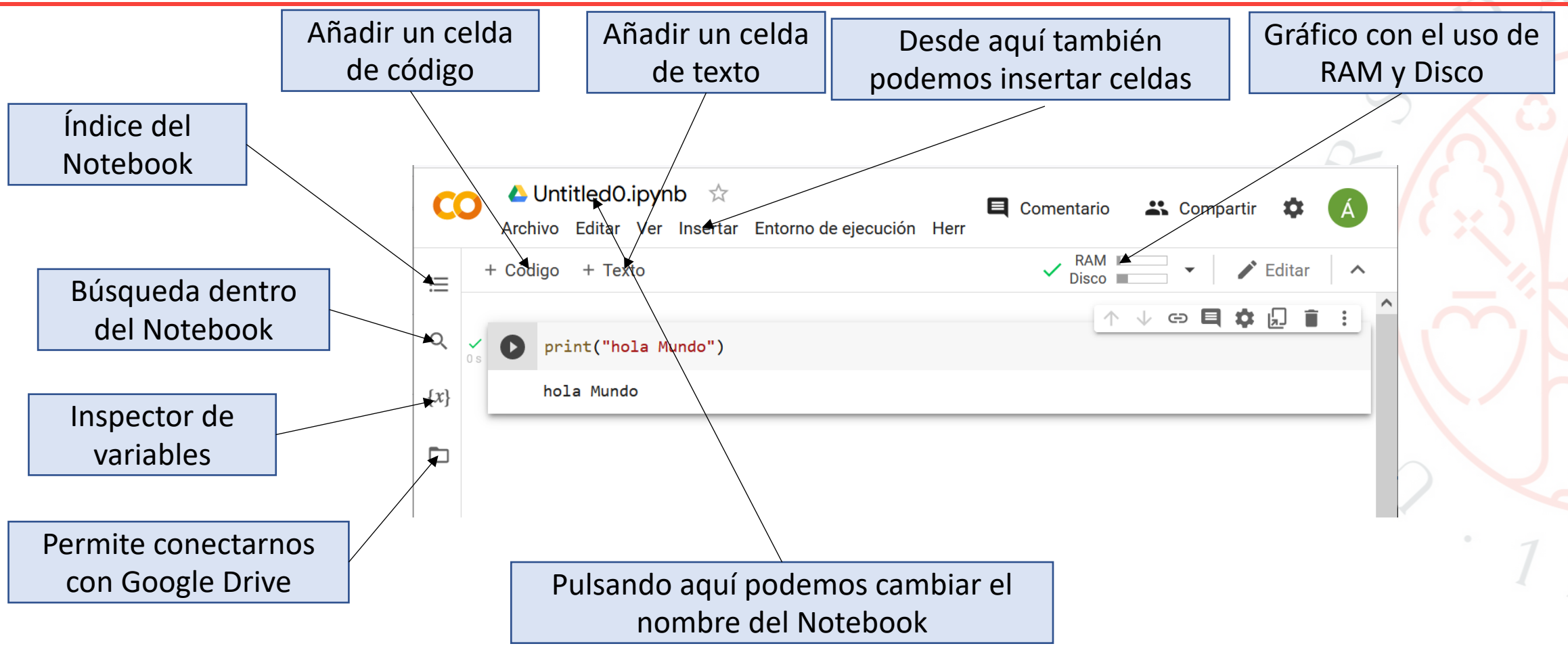
- Existen diversas formas de evaluar una celda
- Presionando **Ctrl + Enter**. Únicamente evalúa actual
- Presionando **Shift + Enter**. Evalúa la celda actual y pasa a la siguiente celda (sin evaluarla). Si nos encontramos en la última celda, creará una nueva celda
- Presionando **Alt + Enter**. Evalúa la celda actual e inserta una nueva celda a continuación
- Pulsado sobre el botón  que encontramos en cada celda

Google Colab

- Cada celda tiene una barra de herramientas donde podemos realizar de forma rápida ciertas acciones
- 

- El icono permite mover la celda actual hacia arriba
- El icono permite mover la celda actual hacia abajo
- El icono permite obtener un enlace (URL) directo a la celda actual
- El icono permite poner comentarios en la celda actual
- El icono permite acceder a la configuración del editor, donde podremos seleccionar el tamaño de letra y el ancho de la sangría entre otros
- El icono permite abrir una pestaña para la celda actual
- El icono permite eliminar la celda actual
- El icono nos permite acceder a opciones adicionales como copiar o cortar la celda

Google Colab

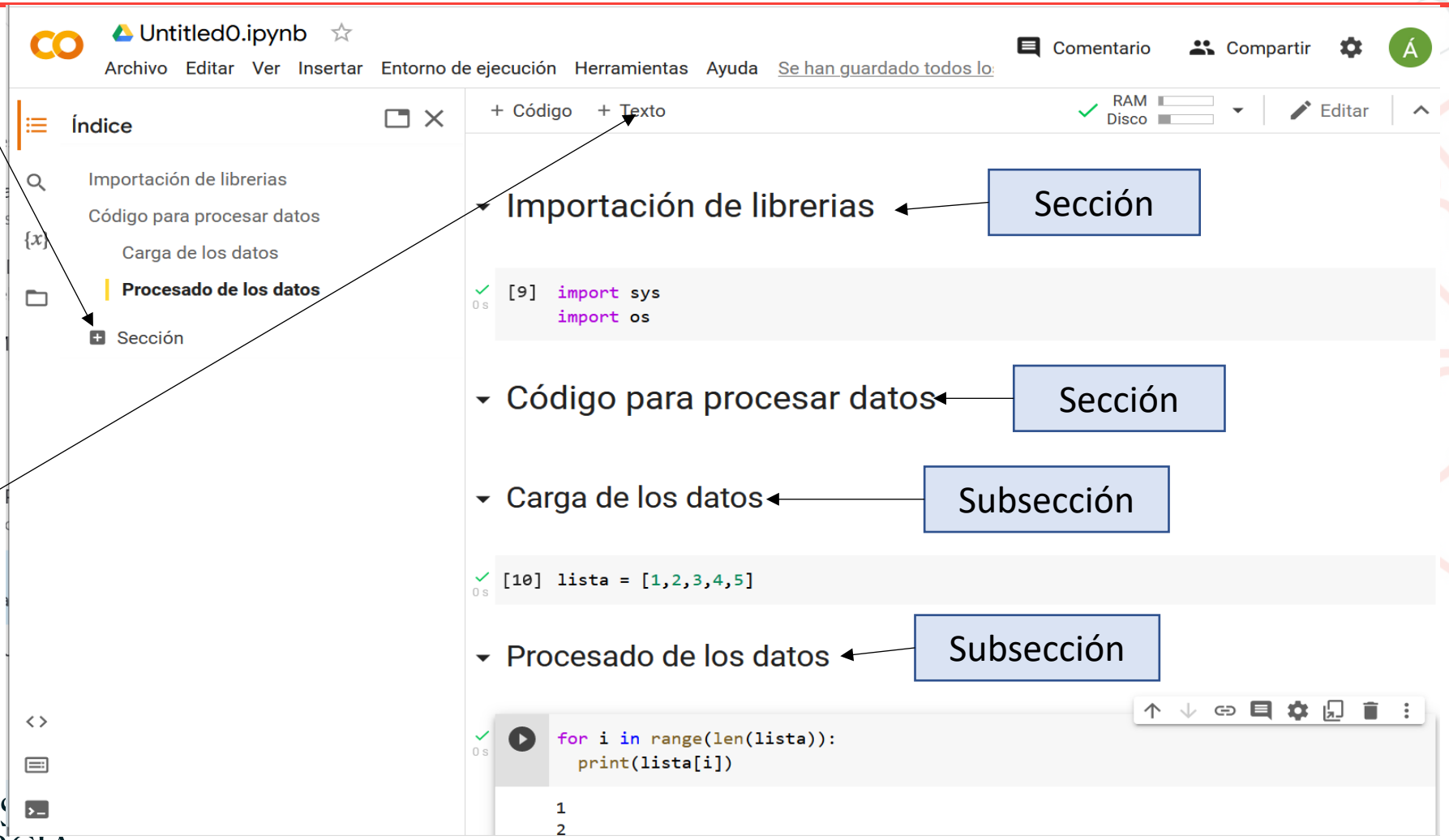


UNIVERSIDAD DE MURCIA

Google Colab

Agregar Sección

Permite agregar una sección o subsección

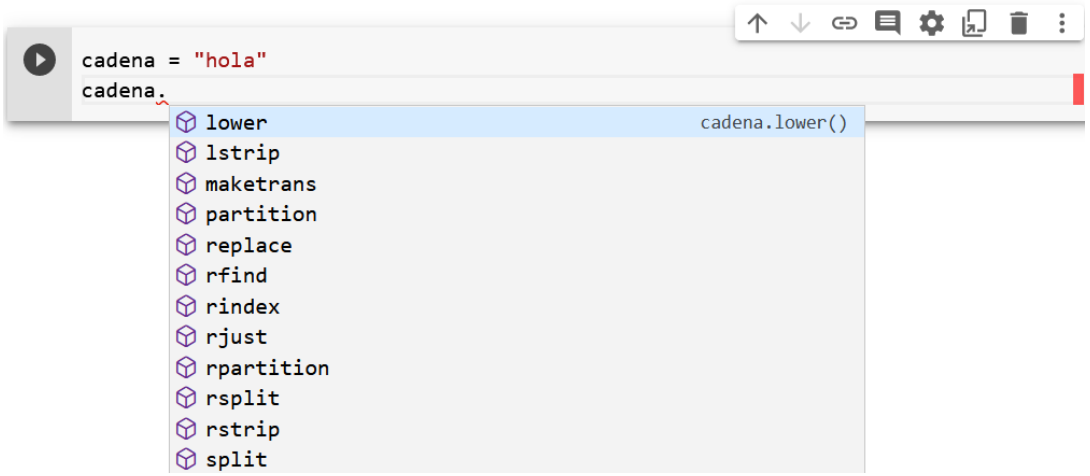


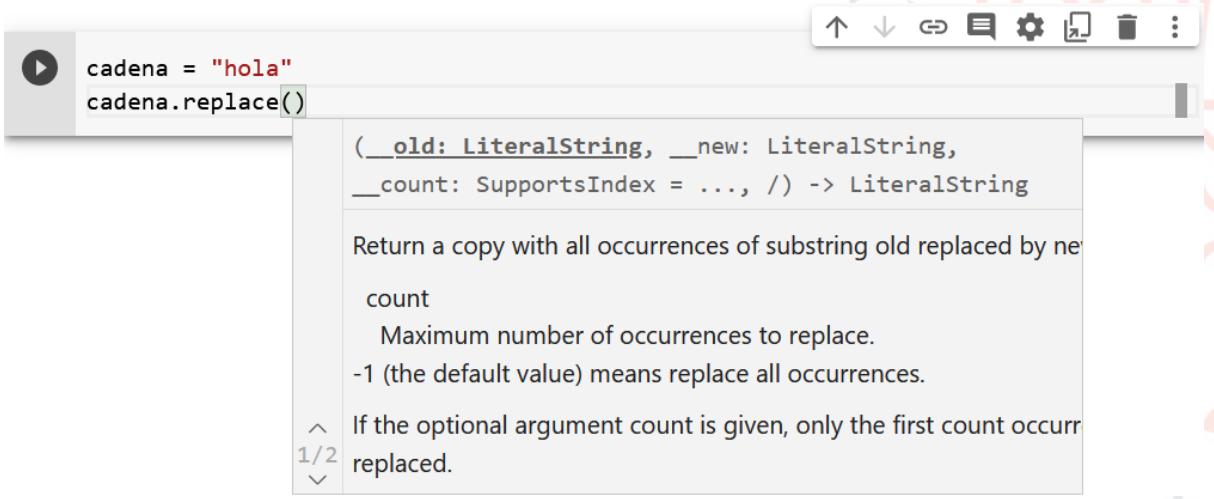
Google Colab

- Google Colab ofrece herramientas para depurar los notebooks

Google Colab

- Google Colab, como otros IDEs, nos ofrece autocompletado de funciones y también nos permite ver la documentación de las funciones





Comentarios

- Permiten hacer anotaciones en el código fuente
- El carácter **#** se utiliza para iniciar un comentario de unas sola línea

```
def multiplica(a,b):
  resultado = 0          # Variable para almacenar el resultado
  for i in range(b):     # Repetimos el proceso b veces
    resultado += a       # Sumamos el valor de a a resultado
  return resultado       # Devolvemos el valor de resultado
```

Variables y Tipos de datos

- La variables ocupan espacio en la memoria y almacenan valores que son necesarios para ejecutar los programas
- Los tipos básicos de las variables en Python son:
  - Enteros (**int**)
  - Decimales (**float**)
  - Booleanos (**bool**)
  - Cadenas (**strings**)
- En Python no es necesario especificar el tipo de datos de una variables puesto que el lenguaje es de tipado dinámico

Variables y Tipos de datos

```
entero = 1
print(type(entero))         # <class 'int'>
decimal = 1.4
print(type(decimal))        # <class 'float'>
cadena = "hola mundo"
print(type(cadena))         # <class 'str'>
booleano = True
print(type(booleano))       # <class 'bool'>
lista = []
print(type(lista))          # <class 'list'>
tupla = ()
print(type(tupla))          # <class 'tuple'>
diccionario = {}
print(type(diccionario))    # <class 'dict'>
```

Operadores Aritméticos

Operador	Nombre	Ejemplo
+	Suma	x + y
-	Resta	x – y
*	Multiplicación	x * y
/	División	x / y
//	División Entera	x // y
**	Exponente	x ** y
%	Módulo	x % y
=	Asignación	x = y

```
a = 2 + 2                    # 4
b = 3.2 * 4                  # 12.8
c = 10 / 3                   # 3.33
d = 10 // 3                  # 3
e = 10 % 3                   # 1
f = 2 ** 4                   # 16
g = "hola mundo" * 2         # hola mundohola mundo
```

Operadores de Asignación

Operador	Nombre	Ejemplo
+=	Suma	x += y
-=	Resta	x –= y
*=	Multiplicación	x *= y
/=	División	x /= y
//=	División Entera	x //= y
**=	Exponente	x **= y
%=	Módulo	x %= y

```
a = 2; b = 3.2; c= 10; d = 10
e = 10; f = 2; g = "hola mundo"
a +=2              # 4
b *= 4             # 12.8
c /= 3             # 3.33
d //= 3            # 3
e %= 3             # 1
f **= 4            # 16
g *= 2             # hola mundohola mundo
```

Operadores de comparación

Operador	Nombre	Ejemplo
==	Igual	x == y
!=	No igual	x != y
>	Mayor que	x > y
<	Menor que	x < y
>=	Mayor o igual que	x >= y
<=	Menor o igual que	x <= y

```
a = 1
a == 1          # True
a != 2          # True
a > 1           # False
a >= 1          # True
a < 1           # True
a <= 1          # True
```

Operadores Lógicos

Operador	Nombre	Ejemplo
and	Operador lógico y	x and y
or	Operador Lógico o	x or y
not	Operador lógico not	x not y

```
a = 1
b = 1
a == 1 and b == 2    # True
a == 1 and b == 3    # False
a > 3 or b <= 3      # True
a < 0 or b == 2      # False
a > 0 and b != 4     # True
```

Control de flujo

- Se utilizan para conducir el flujo del programa
- La estructura más básica de control de flujo es la estructura **if**. Que frecuentemente va acompañada de una estructura **elif** y/o **else**

```
if a == 1:
  print("1")
elif a == 2:
  print("2")
elif a == 3:
  print("3")
else:
  print("Otro número")
```

```
if a == 1:
  print("1")
else:
  print("Otro número")
```

Control de flujo

- Los operadores de comparación y lógicos que hemos visto anteriormente se pueden combinar con las estructuras if/elif/else
- Además, todo valor distinto de 0 se evalúa como verdadero y todo valor igual a 0 se evalúa como falso

```
a = 1
b = -2
c = 0

if a == 1 and b == -2 and c:
  print("hola")          # Esto no se imprimirá
if a or b or c:
  print("hola")          # Esto si se imprimirá
```

Bucles for

- El operador **for** nos permite reproducir una serie de acciones un determinado número de veces

```
lista = [1,2,3,4,5]
for i in range(len(lista)):
  print(lista[i])
```

```
lista = [1,2,3,4,5]
for numero in lista:
  print(numero)
```

- Ambas formas son válidas pero la forma de la derecha es más **pythonic**

Bucles while

- Igual que el bucle **for**, el bucle **while** repite una serie de acciones un determinado número de veces
- El bucle terminará cuando se cumpla la condición de la cláusula **while**

```
i = 0
while i < 10:
  print(i)
  i += 1
```

Interrupción de bucles

- Los bucles **for** y **while** pueden ser interrumpidos por dos sentencias
- La primera de ella es **break**. Cuando se utiliza break, el flujo del programa continua al acabar el bucle

```
i = 0
while i < 10:
  print(i)
  if i == 5:
    break
  i += 1
cadena = "hola mundo" # Al llegar al break, el flujo del
print(cadena + i)     # programa continua en esta línea
```

Interrupción de bucles

- La segunda forma de interrumpir la ejecución de un bucle es mediante la sentencia **continue**
- Cuando se encuentre esta sentencia el bucle pasa a ejecutar la siguiente iteración sin necesidad de haber ejecutado todas las sentencias de la iteración anterior

```
i = 0
while i < 10:
  print(i)                              # 0 1 2 3 4 6 7 8 9
  if i == 5:
    continue
  i += 1
```

Funciones

- Una función permite organizar el código y que de esta forma quede más legible el programa
- Las funciones se definen con la palabra reservada **def**, un **nombre** y una **lista de parámetros** que acepta la función

```
def multiplica(a,b):
  resultado = 0
  for i in range(b):
    resultado += a
  return resultado

if __name__=="__main__":
  a = multiplica(4,4)
  print(a)
  # 16
```

Identación y espaciado

- Python usa la **identación** para agrupar sentencias
  - Cada bloque de código se identa con la misma cantidad de espacios
  - Bloques de código: cuerpo de una instrucción condiciones (if/elif/else), cuerpo de un bucle (while, for), cuerpo de una función, etc...
  - Ejemplo de identación errónea

```
i = 0
while i < 10:
  print(i)
    if i == 5:   # Esta línea está mal identada
    break
  i += 1
```

Estructuras de Datos

- A parte de los tipos de datos sencillos vistos anteriormente, Python dispone de estructuras de datos más complejas
  - Listas
  - Tuplas
  - Cadenas
  - Conjuntos
  - Diccionarios
- Todos estos tipos de datos vienen integradas en la **librería estándar** de Python y no es necesario utilizar ninguna librería de terceros

Listas

- Sirven para almacenar datos de forma secuencial
- Los datos de una lista se pueden modificar. **Las listas son mutables**
- Se dispone de funciones para
  - Añadir elementos a la lista (append)
  - Añadir los elementos de otra lista a la lista actual (extend)
  - Insertar un elemento en una posición determinada (insert)
  - Ordenar los elementos de una lista (sort)
  - Eliminar todos los elementos de una lista (clear)
  - Eliminar el primer elemento con el valor indicado (remove)
  - Eliminar el elemento de una posición determinada (pop)
  - Obtener el índice del elemento con el valor determinado (index)
  - Obtener la longitud de la lista (len)
  - Contar el número de veces que aparece un valor determinado (count)

Listas

```
lista = [1,1,1,2,3,4,5]
lista.append(8)                    # [1,1,1,2,3,4,5,8]
lista.extend([-100,-101,-102])     # [1,1,1,2,3,4,5,8,-100,-101,-102]
lista.insert(2,300)                # [1,1,300,1,2,3,4,5,8,-100,-101,-102]
lista.sort()                       # [-102,-101,-100,1,1,1,2,3,4,5,8,300]
lista.remove(300)                  # [-102,-101,-100,1,1,1,2,3,4,5,8]
lista.pop(0)                       # [-101,-100,1,1,1,2,3,4,5,8]
lista.index(3)                     # 6
lista.count(1)                     # 3
len(lista)                         # 10
lista.clear()                      # []
```

Listas

- Los elementos de una lista pueden ser accedidos mediante índices enteros
  - El **primer elemento** de una lista es el **0** y no el 1
- Se puede acceder a valores de una lista utilizando **índices negativos**
- Se puede obtener **rodajas o slices** de una lista utilizando un formato de indexado como el siguiente: [inicio:final:paso]
  - Esto nos permite también ordenar de forma inversa una lista, mediante el indexado [::-1]
- Se permite sumar dos listas. El resultado es una nueva lista que contiene los elementos de ambas listas
- Las listas pueden contener elementos no homogéneos

Listas

```
lista = [1,2,3,4,5,6,7]
print(lista[1])          # 2
print(lista[-2])         # 6
print(lista[1:3])        # [2,3]
print(lista[::-1])       # [7,6,5,4,3,2,1]
print(lista[::2])        # [1,3,5,7]

l1 = [1,2]
l2 = [3,4]
l3 = l1 + l2             # [1,2,3,4]
l3.append("hola")        # [1,2,3,4,"hola"]
```

Listas

- Las listas se pueden iterar mediante bucles **for** y bucle **while**. Aunque el uso de bucles **for es más phytonic**

```
lista = [1,2,3,4,5,6,7]

i = 0
while i < len(lista):
  print(lista[i])
  i += 1
```

```
lista = [1,2,3,4,5,6,7]

for i in range(len(lista)):
  print(lista[i])
```

```
lista = [1,2,3,4,5,6,7]

for elemento in lista:
  print(elemento)
```

- Si queremos obtener el índice en la forma más pythonic podemos usar la función **enumerate ( )**

```
lista = [1,2,3,4,5,6,7]

for i,elemento in enumerate(lista):
  print(i,elemento)
```

Listas

- Se puede utilizar el operador **in** para comprobar la pertenencia de un elemento a la lista

```
lista = [1,2,3,4,5,6,7]
if 1 in lista:
  print("El número 1 está en la lista")
lista.append("hola")
if "hola" in lista and 8 in lista:
  print("La cadena 'hola' y el número 8 están en la lista")
```

- Se pueden crear **listas por comprensión**

```
lista = [1,2,3,4,5,6,7]
doble = [i*2 for i in lista]                    # [2,4,6,8,10,12,14]
dobles_pares = [i*2 for i in lista if i%2 ==0]  # [4,8,12]
```

Tuplas

- Las tuplas son parecidas a las listas, pero son **tipos de datos inmutables.** Es decir, las tupas no se pueden modificar
- Tanto las listas como tuplas soportan el **operador de unpacking**

```
tupla1 = (1, 0.3, -12, "hola")
tupla2 = (4.5, -5, 6, "mundo")

tupla1[1] = 2          # Error!
print(tupla1[1])       # 0.3

a,b,c,d = tupla2
print(a,b,c,d)         # 4.5, -5, 6, "mundo"
```

Cadenas

- Las cadenas nos permiten usar texto en nuestros programas
- Comparten muchas de las funciones con las listas
  - Indexado mediante números enteros positivos y negativos
  - Uso de **rodajas o slices** mediante el formato [inicio:fin]
  - Cálculo de la longitud de una cadena con la función **len( )**
- Las cadenas permiten el uso del **operador +** para concatenar dos o más cadenas.
- Python posee un **formateo de cadenas** extremadamente potente que permite enlazar variables a posiciones determinadas dentro de una cadena

Cadenas

- Las cadenas pueden ser tratadas como listas. Además, Python ofrece un formateo muy potente

```
cadena = "hola mundo"

print(cadena)                                   # hola mundo
print(cadena[0])                                # h
print(cadena[-1])                               # o
print(cadena[0:4])                              # hola

cadena += "!!!"                                 # hola mundo!!!
print(cadena)

print("{} {}".format("hola", "mundo"))          # hola mundo
print("{b} {a}".format(
    a="variable1", b="variable2"))              # variable2 variable1
```

Cadenas

- Python proporciona muchas funciones integradas en la librería estándar para el manejo de cadenas

capitalize()	find()	isdigit()	rjust()	partition()
casefold()	rfind()	isidentifier()	ljust()	rpartition()
center()	format_map()	islower()	lower()	split()
count()	index()	isnumeric()	upper()	rsplit()
encode()	rindex()	isprintable()	strip()	splitlines()
startswith()	isalnum()	isespace()	rstrip()	swapcase()
endswith()	isalpha()	istitle()	lstrip()	translate()
expandstabs()	isascii()	isupper()	maketrans()	title()
format()	isdecima()	join()	replace()	zfill()

Diccionarios

- Los diccionarios son asociaciones de un conjunto de claves a un conjunto de valores
- Los diccionarios también reciben el nombre de **arrays asociativos**
- Los diccionarios se componen de dos elementos
  - **Clave**: mediante el cual se referencia una posición del diccionario
  - **Valor**: valor de la clave

$$K = \{ keys \}, V = \{ values \}. D : K \rightarrow V : K \rightarrow v_k \in V$$

Diccionarios

- Ejemplos de construcción de diccionarios

```
diccionario1 = {}
diccionario2 = dict()
diccionario3 = {"año": 2000, "mes": "enero"}

diccionario1["clave"] = "valor"
diccionario1["altura"] = 1.75
diccionario1["edad"] = 24

diccionario2["precio"] = 5.23
diccionario2["coordenadas"] = [3,4]

print(diccionario1)
# {'clave': 'valor', 'altura': 1.75, 'edad': 24}
print(diccionario2)
# {'precio': 5.23, 'coordenadas': [3, 4]}
print(diccionario3)
# {'año': 2000, 'mes': 'enero'}
```

Diccionarios

- Acceso, borrado y sobreescritura de claves

```
mes = diccionario3["mes"]
dia = diccionario3["dia"]
# KeyError: 'dia'

diccionario3["mes"] = "diciembre"
diccionario3["dia"] = 4
del diccionario3["año"]

print(diccionario3)
# {'mes': 'diciembre', 'dia': 4}
```

Diccionarios

- También pueden usarse bucles **for** y clausulas **in** para iterar sobre ellos o comprobar la existencia de claves

```
for key in diccionario1:
  print(key, diccionario1[key])
# clave valor
# altura 1.75
# edad 24
for key,value in diccionario1.items():
  print(key, value)
# clave valor
# altura 1.75
# edad 24
if "clave" not in diccionario3:
  print("No existe la clave")
# No existe la clave
```

Conjuntos

- Un conjunto (set) es una colección de elementos sin repetición ni orden

```
conjunto = {1,"a", "b", 3, 5, 1}
print (conjunto)
# {1, 3, 5, 'b', 'a'}
```

- También se pueden usar los bucles **for** junto con la cláusula **in** para recorrerlos

```
for elemento in conjunto:
    print(elemento)
print("a" in conjunto)
# True
print("c" in conjunto)
# False
```

Mutable e inmutable

- Algunos tipos no pueden cambiar su contenido (como las strings) mientras que otros sí que puede (ejemplo: listas)

```
x = "hola"
y = x
print(x)
# hola
y += " mundo"
print(x)
# hola

x = [1, 2, 3]
y = x
print(x)
# [1, 2, 3]
y += [3, 2, 1]
print(x)
# [1, 2, 3, 3, 2, 1]
```

UNIVERSIDAD DE MURCIA

Excepciones

- A veces, en los programas pueden ocurrir excepciones, que si no son capturadas correctamente acabarán arrojando un error

```
x = 3 / 0
# ZeroDivisionError: division by zero
try:
x = 3 / 0
except:
print("División por 0")
# División por 0
try:
f = open("hola.txt")
except IOError as e:
print("IOError: {0}, {1}".format(e.errno,
e.strerror))
# IOError: 2, No such file or directory
```

Cláusula import

- La cláusula **import** usa para cargar un módulo (librería, u otro módulo fuente)

```
import math
print(math.pi)
# 3.141592653589793
```

```
import math as m
print(m.pi)
# 3.141592653589793
```

```
from math import pi
print(pi)
# 3.141592653589793
```

```
from math import *
print(pi)
# 3.141592653589793
```

Ejercicios

1. Define una lista numérica y calcula las siguientes métricas
    - Suma de todos los elementos
    - La media de los números contenidos en la lista
    - El número más pequeño
    - El número más grande
    - El índice del número más pequeño
    - El índice del número más grande

Ejercicios

2. Escribe una función que acepte **dos parámetros**. El primer parámetro es una **cadena** cualquiera. El segundo parámetro es un **carácter**. La función debe devolver una nueva cadena creada a partir de la cadena del primer parámetro, pero eliminando todas las ocurrencias del carácter pasado en el segundo parámetro
3. Escribe una función que reciba **dos parámetros**. El primer parámetro es una **cadena** cualquiera. El segundo parámetro **indica cuantas veces se van a repetir los 2 últimos caracteres** de la cadena del primer parámetro. La función devuelve los 2 últimos caracteres de la primera cadena repetidos tantas veces como se especifique en el segundo parámetro

Ejercicios

4. Escribe una función que recibe una cadena y devuelve el número de repeticiones de cada uno de los caracteres que contiene la cadena
5. Escribe una función que dada una cadena devuelva una lista con las palabras de la lista
6. Escribe una función que dada una cadena, devuelva el número total de vocales y que vocales son
7. Escribe una función que dada una cadena devuelva la palabra de menor y mayor longitud
8. Escribe una función que, dada una cadena, devuelva si esta es un palíndromo. Un palíndromo es una palabra que es igual si se leer de izquierda a derecha que si se lee de derecha a izquierda

Ejercicios

9. Escribe una función que, dada una lista de cadenas, devuelva una nueva lista con aquellas cadenas cuya longitud sea par
10. Escribe una función que dada una cadena con 1 o más palabras cambie la primera letra de cada una de las palabras a mayúscula
11. Escribe una función que dada una cadena que contiene una dirección de correo electrónico, devuelva únicamente el nombre
12. Escribe una función que dados 3 números determine cual es el mayor y el menor de ellos.

Grado en Gestión de la Información y Contenidos Digitales

Procesamiento y Visualización de Datos

Práctica 2. Pandas

Bloque 1: Procesamiento de datos

UNIVERSIDAD DE MURCIA

Facultad de Informática

Introducción

- **Pandas**, una librería para manipulación y análisis de datos. Es una herramienta open source y muy flexible
- Aunque, como veremos más adelante, debido a la flexibilidad que ofrece, su rendimiento puede ser una limitación en algunas ocasiones
- El núcleo de Pandas son las estructuras **Series** y **DataFrame**. Este último consiste en un array multidimensional con filas y columnas etiquetadas con la posibilidad de almacenar tipos heterogéneos y datos faltantes.

Introducción

- Para usar la librería Pandas en nuestros cuadernos basta con ejecutar el **import** correspondiente

```
import pandas
pandas.__version__
```

- Es frecuente encontrar la librería pandas como referenciada como **pd**. Durante este curso nosotros también seguiremos esta nomenclatura

```
import pandas as pd
pd.__version__
```

El objeto Series

- Los objetos de Pandas son básicamente arrays, los cuales disponen de columnas y filas **etiquetadas** en vez de acceder a ellos únicamente de forma numérica
- Un objeto **Series** en Pandas se puede considerar como un array unidimensional y puede ser creado mediante una lista o mediante un diccionario de Python
- A diferencia de un array convencional, **las series pueden tener un índice explicito**
- Utilizando el atributo **values** de un objeto serie podemos acceder al array de numpy

El objeto Series

```
serie = pd.Series([1,2,3,4,5])
print(serie)
#0    1
#1    2
#2    3
#3    4
#4    5
#dtype: int64
print(serie[0])
# 1
print(serie[0:2])
# 0    1
# 1    2
# dtype: int64
print(serie.values)
# array([1, 2, 3, 4, 5])
```

```
serie = pd.Series([1,2,3,4,5], index=["a",
"b", "c", "d", "e"])
print(serie)
#a    1
#b    2
#c    3
#d    4
#e    5
#dtype: int64
print(serie["a"])
# 1
print(serie["a":"c"])
# a    1
# b    2
# c    3
# dtype: int64
print(serie.values)
# array([1, 2, 3, 4, 5])
```

El objeto Series

- Se puede acceder también a los índices mediante el atributo index

```
print(serie.index)   # RangeIndex(start=0, stop=5, step=1)
print(serie1.index) # Index(['a', 'b', 'c', 'd', 'e'], dtype=
'object')
```

- Una serie también se puede construir a partir de un diccionario de Python

```
poblacion_dict = {'California': 38332521,'Texas': 26448193,
                  'New York': 19651127,'Florida': 19552860,
                  'Illinois': 12882135}
poblacion = pd.Series(poblacion_dict)
print(poblacion)
# California    38332521
# Texas         26448193
# New York      19651127
# Florida       19552860
# Illinois      12882135
# dtype: int64
```

El objeto Series

- En definitiva, el objeto **Series** admite una lista unidimensional de datos (data) y de forma opcional un array que contenga los índices (index)

```
serie = pd.Series(data, index = index)
```

- Distintas formas de crear series en Pandas

```
serie = pd.Series(5, index=[100, 200, 300])
print(serie)
# 100    5
# 200    5
# 300    5
# dtype: int64
```

```
serie = pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
print(serie)
# 3    c
# 2    a
# dtype: object
```

El objeto DataFrame

- El objeto **DataFrame** se puede ver como un array multidimensional de numpy o como un diccionario de Python
- A diferencia del objeto **Series**, el objeto **DataFrame** permite almacenar arrays multidimensionales con **índices y nombres de columnas personalizados**
- El objeto **DataFrame** también puede ser visto como una agrupación de objetos **Series** de Pandas

El objeto DataFrame

```
poblacion_dict = {'California': 38332521,'Texas': 26448193,
                  'New York': 19651127,'Florida': 19552860,
                  'Illinois': 12882135}
poblacion = pd.Series(poblacion_dict)
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
df = pd.DataFrame({'population': poblacion,
                   'area': area})
print(df)
#             population    area
# California    38332521  423967
# Texas         26448193  695662
# New York      19651127  141297
# Florida       19552860  170312
# Illinois      12882135  149995
# dtype: int64
```

El objeto DataFrame

- Al igual que con las series, en un **DataFrame** podemos acceder a los índices del mismo
- También podemos acceder a las columnas

```
print(estados.index)
# Index(['California', 'Texas', 'New York', 'Flo
rida', 'Illinois'], dtype='object')
print(estados.columns)
# Index(['population', 'area'], dtype='object')
```

El objeto DataFrame

- También podemos crear un **DataFrame** a partir de un array bidimensional de numpy

```
import numpy as np
df = pd.DataFrame(np.random.rand(3, 2),
                  columns=['foo', 'bar'],
                  index=['a', 'b', 'c'])
print(df)
#          foo       bar
# a  0.521215  0.147606
# b  0.051292  0.056368
# c  0.852218  0.711915
```

Selección de datos en Series

- La selección y modificación de datos se hace de forma similar a como se hace en las estructuras de Python

```
serie = pd.Series([1,2,3,4,5])
serie1 = pd.Series([1,2,3,4,5], index=
["a", "b", "c", "d", "e"])
print(serie[0])
# 1
print(serie1["a"])
# 1
print("a" in serie1)
# True
print("a" in serie)
# False
print(1 in serie)
# True
```

```
print(serie.keys())
# RangeIndex(start=0, stop=5, step=1)
print(serie1.keys())
# Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
serie[0] = 3
serie1["a"]=4
print(list(serie.items()))
# [(0, 3), (1, 2), (2, 3), (3, 4), (4, 5)]
print(list(serie1.items()))
# [('a', 4), ('b', 2), ('c', 3), ('d', 4), ('e', 5)]
```

Selección de datos en Series

- **Problema**: cuando se indexa con un escalar, Pandas usará los índices explícitos. Pero cuando se indexa mediante un **slice**, Pandas usará los índices implícitos

```
serie = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
print(serie)
# 1    a
# 3    b
# 5    c
# dtype: object
print(serie[1])
# a
print(serie[1:3])
# 3    b
# 5    c
# dtype: object
```

Selección de datos en Series

- **Solución**: Usar los atributos **iloc** y **loc**

```
print(serie.loc[1])
# a
print(serie.loc[1:3])
# 1    a
# 3    b
# dtype: object
print(serie.iloc[1])
# b
print(serie.iloc[1:3])
# 3    b
# 5    c
# dtype: object
```

Selección de datos en DataFrames

- Todo lo que hemos visto previamente funciona de forma similar para los objetos **DataFrame**

```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
poblacion = pd.Series({'California': 38332521, 'Texas': 26448193,
                  'New York': 19651127, 'Florida': 19552860,
                  'Illinois': 12882135})
df = pd.DataFrame({'area':area, 'poblacion':poblacion})
print(df)
#                area  poblacion
# California   423967   38332521
# Texas        695662   26448193
# New York     141297   19651127
# Florida      170312   19552860
# Illinois     149995   12882135
```

Selección de datos en DataFrames

```
print(df["area"])
# California     423967
# Texas          695662
# New York       141297
# Florida        170312
# Illinois       149995
# Name: area, dtype: int6
```

```
print(df.area)
# California     423967
# Texas          695662
# New York       141297
# Florida        170312
# Illinois       149995
# Name: area, dtype: int6
```

```
df['densidad'] = df['poblacion'] / df['area']
print(df)
#                area  poblacion    densidad
# California   423967    38332521   90.413926
# Texas        695662    26448193   38.018740
# New York     141297    19651127  139.076746
# Florida      170312    19552860  114.806121
# Illinois     149995    12882135   85.883763
```

Selección de datos en DataFrames

```
print(df.iloc[:3, :2])
#                area  poblacion
# California   423967    38332521
# Texas        695662    26448193
# New York     141297    19651127
print(df.loc[:'Illinois', :'poblacion'])
#                area  poblacion
# California   423967    38332521
# Texas        695662    26448193
# New York     141297    19651127
# Florida      170312    19552860
# Illinois     149995    12882135
```

```
print(df.loc[df.densidad > 100, ['poblacion', 'densidad']])
#            poblacion    densidad
# New York    19651127  139.076746
# Florida     19552860  114.806121
```

Operaciones sobre Series y DataFrame

- Tanto las Series como los DataFrame de Pandas ofrecen la posibilidad de utilizar una gran variedad de funciones

```
serie = pd.Series(np.random.randint(0, 10, 4))
print(serie)
# 0    6
# 1    9
# 2    2
# 3    2
# dtype: int64
print(np.exp(serie))
# 0     403.428793
# 1    8103.083928
# 2       7.389056
# 3       7.389056
# dtype: float64
```

Operaciones sobre Series y DataFrame

```
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
print(A + B)
# 0    NaN
# 1    5.0
# 2    9.0
# 3    NaN
# dtype: float64
print(A.add(B, fill_value=0))
# 0    2.0
# 1    5.0
# 2    9.0
# 3    5.0
# dtype: float64
```

```
A = pd.DataFrame(np.random.randint(0, 20, (2, 2)),
                 columns=list('AB'))
print(A)
#     A   B
# 0  11   8
# 1  13  17
B = pd.DataFrame(np.random.randint(0, 10, (3, 3)),
                 columns=list('BAC'))
print(B)
#    B  A  C
# 0  0  8  8
# 1  5  0  1
# 2  2  1  2
print(A + B)
#       A     B   C
# 0  19.0   8.0 NaN
# 1  13.0  22.0 NaN
# 2   NaN   NaN NaN
print(A.add(B, fill_value=0))
#       A     B    C
# 0  19.0   8.0  8.0
# 1  13.0  22.0  1.0
# 2   1.0   2.0  2.0
```

UNIVERSIDAD DE MURCIA

Operaciones sobre Series y DataFrame

- Todas las operaciones aritméticas tienen su contrapartida en funciones de Pandas

Operación	Función de Pandas
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Operaciones sobre Series y DataFrame

- Pandas también permite operaciones entre DataFrames y Series

```
df = pd.DataFrame(np.random.randint(10, size=(3, 4)),
                  columns=list('QRST'))
print(df - df.iloc[0])
#    Q  R  S  T
# 0  0  0  0  0
# 1  3  2 -4  2
# 2  4  5  1  1
print(df.subtract(df['R'], axis=0))
#    Q  R  S  T
# 0 -2  0 -2 -2
# 1  7  0  1  2
# 2 -4  0 -1 -1
```

Tratando con valores faltantes

- Cuando nos enfrentamos a problemas del mundo real, encontraremos muchos conjuntos de datos donde **faltan datos**
- Hay dos estrategias principales a la hora de indicar la falta de datos
  - Mediante un array independiente donde en cada posición nos indica si el dato falta o no
  - Indicándolo en los propios datos mediante alguna convención, como por ejemplo el uso de Not A Number (**NaN**)
- La librería Pandas indica los datos faltante mediante esta segunda estrategia mediante **NaN** y **None**

Tratando con valores faltantes

- **None** no se puede usar en cualquier numpy/Pandas array. Solamente en aquellos cuyo **dtype** sea object
- Inconveniente las operaciones aritméticas y de agregación sobre arrays con dtype object son **lentas**

```
for dtype in ['object', 'int']:
    print("dtype =", dtype)
    %timeit np.arange(1E6, dtype=dtype).sum()
# dtype = object
# 79.1 ms ± 1.42 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

# dtype = int
# 3.23 ms ± 101 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Tratando con valores faltantes

- Además, las operaciones de agregación sobre arrays con valores faltantes suelen resultar en **errores**

```
x = np.array([None, 2, None, 1])
print(x)
# array([None, 2, None, 1]
print(x.sum())
# TypeError: unsupported operand type(s)
# for +: 'NoneType' and 'int'
```

Tratando con valores faltantes

- **NaN** es un valor **float especial** y, por tanto, los arrays que contienen este valor pueden realizar operaciones con mayor velocidad
- **Problema**: un valor NaN no puede realizar ninguna operación con otro dato
- **Solución**: funciones especiales de numpy para tratar con estos valores

```
x = np.array([1, np.nan, 3, 4])
print(x.dtype)
# float64
print(x.sum())
# nan
print(np.nansum(x))
# 8
```

Tratando con valores faltantes

- **NaN** solo es válido para datos tipo **float**, por eso si creamos un array de tipo entero y le asignamos un valor **None** a cualquiera de sus valores, automáticamente se convertirá a tipo float

```
x = pd.Series(range(2), dtype=int)
print(x)
# 0    0
# 1    1
# dtype: int64
x[0] = None
print(x)
# 0    NaN
# 1    1.0
# dtype: float64
```

Tatando con valores faltantes

- En Pandas hay diversas funciones para detectar valores faltantes, eliminarlos o rellenarlos con otros datos

```
x = pd.Series([1, np.nan, 'hello', None])
print(x.isnull())
# 0    False
# 1     True
# 2    False
# 3     True
# dtype: bool
print(x[x.notnull()])
# 0        1
# 2    hello
# dtype: object
```

```
print(x.dropna())
# 0        1
# 2    hello
# dtype: object
```

Tatando con valores faltantes

- Uso de **how** y **thresh** para controlar la eliminación de valores NaN

```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
print(df.dropna())
#      0    1  2
# 1  2.0  3.0  5
print(df.dropna(axis='columns'))
#    2
# 0  2
# 1  5
# 2  6
df[3] = np.nan
print(df)
print(df.dropna(axis='columns', how='all'
))
#      0    1  2
# 0  1.0  NaN  2
# 1  2.0  3.0  5
# 2  NaN  4.0  6
```

```
print(df.dropna(axis='rows', thresh=3))
#      0    1  2   3
# 1  2.0  3.0  5 NaN
```

Tratando con valores faltantes

```
x = pd.Series([1, np.nan, 2, None, 3],
              index=list('abcde'))
print(x)
# a    1.0
# b    NaN
# c    2.0
# d    NaN
# e    3.0
dtype: float64
```

```
print(x.fillna(0))
# a    1.0
# b    0.0
# c    2.0
# d    0.0
# e    3.0
dtype: float64
```

```
print(x.ffill())
# a    1.0
# b    1.0
# c    2.0
# d    2.0
# e    3.0
# dtype: float64
```

```
print(x.bfill())
# a    1.0
# b    2.0
# c    2.0
# d    3.0
# e    3.0
# dtype: float64
```

UNIVERSIDAD DE MURCIA

Concatenación de Series y DataFrame

- La función **concat( )** de Pandas se define de la siguiente forma

```
pd.concat(objs, axis=0, join='outer', join_axes=None,
          ignore_index=False, keys=None, levels=None,
          names=None, verify_integrity=False, copy=True)
```

- Se puede usar de forma sencilla

```
serie1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
serie2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
print(pd.concat([serie1, serie2]))
# 1    A
# 2    B
# 3    C
# 4    D
# 5    E
# 6    F
# dtype: object
```

UNIVERSIDAD DE MURCIA

Concatenación de Series y DataFrame

- **concat( )** se puede aplicar sobre **DataFrame** y admite el argumento **axis**. Además, **concat( )** preserva los índices

```
df1 = pd.DataFrame([[1,1],[2,2]],
                   columns=list('AB'))
df2 = pd.DataFrame([[3,3],[4,4]],
                   columns=list('AB'))
print(df1)
#    A  B
# 0  1  1
# 1  2  2
print(df2)
#    A  B
# 0  3  3
# 1  4  4
```

```
print(pd.concat((df1,df2)))
#    A  B
# 0  1  1
# 1  2  2
# 0  3  3
# 1  4  4
print(pd.concat((df1,df2), axis=1))
#    A  B  A  B
# 0  1  1  3  3
# 1  2  2  4  4
```

Concatenación de Series y DataFrame

- Por defecto, **concat( )** concatena los **DataFrame** en el eje 0.
- Los ejes también se pueden referenciar como **index** y como **columns**

```
print(pd.concat((df1,df2),
                axis='index'))
#    A  B  A  B
# 0  1  1  3  3
# 1  2  2  4  4
```

```
print(pd.concat((df1,df2),
                axis="columns"))
#    A  B  A  B
# 0  1  1  3  3
# 1  2  2  4  4
```

Concatenación de Series y DataFrame

- Las **funciones de numpy** de concatenación **se pueden usar** directamente sobre los DataFrame y las Series

```
print(np.concatenate((df1, df2)))
# array([[1, 1],
#        [2, 2],
#        [3, 3],
#        [4, 4]])
```

Concatenación de Series y DataFrame

- La función **concat( )** de Pandas preserva los índices, aunque haya duplicados

```
df1 = pd.DataFrame([[1,1],[2,2]],
                   columns=list('AB'))
df2 = pd.DataFrame([[3,3],[4,4]],
                   columns=list('AB'))
df = pd.concat((df1, df2))
print(df)
#    A  B
# 0  1  1
# 1  2  2
# 0  3  3
# 1  4  4
print(df.loc[0])
#    A  B
# 0  1  1
# 0  3  3
```

Concatenación de Series y DataFrame

- Podemos evitar concatenar 2 DataFrame en el caso de que tengan índices duplicados

```
try:
    pd.concat([df1, df2], verify_integrity=True)
except ValueError as e:
    print("ValueError:", e)
# ValueError: Indexes have overlapping values: Int64Index([0, 1], dtype='int64')
```

- También podemos ignorar los índices

```
print(pd.concat((df1, df2),
                ignore_index = True))
#    A  B
# 0  1  1
# 1  2  2
# 2  3  3
# 3  4  4
```

Concatenación de Series y DataFrame

- Hasta ahora hemos concatenado DataFrame que tienen las mismas columnas

```
df1 = pd.DataFrame([[1,1],[2,2]],
                   columns=list('AB'))
df2 = pd.DataFrame([[3,3],[4,4]],
                   columns=list('CD'))
df = pd.concat((df1, df2))
print(df)
#       A    B    C    D
# 0   1.0  1.0  NaN  NaN
# 1   2.0  2.0  NaN  NaN
# 0   NaN  NaN  3.0  3.0
# 1   NaN  NaN  4.0  4.0
```

Concatenación de Series y DataFrame

- Podemos usar el argumento **join** para controlar la situación anterior. Por defecto, el valor es **outer** pero podemos especificar **inner**

```
df1 = pd.DataFrame([[1,1],[2,2]],
                   columns=list('AB'))
df2 = pd.DataFrame([[3,3],[4,4]],
                   columns=list('AC'))
df = pd.concat((df1, df2), join="inner")
print(df)
#    A
# 0  1
# 1  2
# 0  3
# 1  4
```

Concatenación de Series y DataFrame

- Para concatenar Series y DataFrame podemos usar la función **append( )** en vez de **concatenate( )**

```
df1 = pd.DataFrame([[1,1],[2,2]],
                   columns=list('AB'))
df2 = pd.DataFrame([[3,3],[4,4]],
                   columns=list('AB'))
df = pd.concat((df1, df2))
print(df)
#    A  B
# 0  1  1
# 1  2  2
# 0  3  3
# 1  4  4
```

```
print(df1.append(df2))
#    A  B
# 0  1  1
# 1  2  2
# 0  3  3
# 1  4  4
```

Integración de DataFrame

- Hasta ahora hemos concatenado DataFrame y Series que son independientes entre si
- A veces queremos crear un DataFrame con datos proveniente de dos fuentes distintas

```
df1 = pd.DataFrame({'empleado': ['Fernando', 'Antonio', 'Sofia', 'Alicia'],
                    'grupo': ['Contabilidad', 'Ingeniería', 'Ingeniería', 'Recursos Humanos']})
df2 = pd.DataFrame({'empleado': ['Sofia', 'Fernando', 'Antonio', 'Alicia'],
                    'fecha_contratacion': [2004, 2008, 2012, 2014]})
print(df1)
#    empleado             grupo
# 0  Fernando      Contabilidad
# 1   Antonio        Ingeniería
# 2     Sofia        Ingeniería
# 3    Alicia  Recursos Humanos
print(df2)
#    empleado  fecha_contratacion
# 0     Sofia                2004
# 1  Fernando                2008
# 2   Antonio                2012
# 3    Alicia                2014
```

Integración de DataFrame

- Podemos fusionar DataFrame con la función **merge( )** siempre y cuando tengan una columna en común

```
df3 = pd.merge(df1, df2)
print(df3)
#     empleado             grupo  fecha_contratacion
# 0   Fernando      Contabilidad                2008
# 1    Antonio       Ingeniería                2012
# 2      Sofia       Ingeniería                2004
# 3     Alicia  Recursos Humanos                2014
```

```
df4 = pd.DataFrame({'grupo': ['Contabilidad', 'Ingeniería', 'Recursos Humanos'],
                    'supervisor': ['Carlos', 'Juan', 'Rocio']})
print(pd.merge(df3, df4))
#     empleado             grupo  fecha_contratacion supervisor
# 0   Fernando      Contabilidad                2008     Carlos
# 1    Antonio       Ingeniería                2012       Juan
# 2      Sofia       Ingeniería                2004       Juan
# 3     Alicia  Recursos Humanos                2014      Rocio
```

Integración de DataFrame

```
df5 = pd.DataFrame({'grupo': ['Contabilidad', 'Contabilidad',
                              'Ingeniería', 'Ingeniería',
                              'Recursos Humanos', 'Recursos Humanos'],
                    'habilidades': ['matemáticas', 'hojas de álculo', 'codificación',
                                    'linux', 'hojas de cálculo', 'organización']})
print(pd.merge(df1, df5))
#     empleado              grupo           habilidades
# 0   Fernando       Contabilidad         matemáticas
# 1   Fernando       Contabilidad    hojas de álculo
# 2    Antonio         Ingeniería       codificación
# 3    Antonio         Ingeniería              linux
# 4      Sofia         Ingeniería       codificación
# 5      Sofia         Ingeniería              linux
# 6     Alicia  Recursos Humanos   hojas de cálculo
# 7     Alicia  Recursos Humanos       organización
```

Integración de DataFrame

- Podemos usar el argumento **on** (y **left_on** y **right_on**) para controlar sobre que columna vamos a realizar el merge

```
print(pd.merge(df1, df2, on='empleado'))
#     empleado              grupo  fecha_contratacion
# 0  Fernando      Contabilidad                2008
# 1   Antonio        Ingeniería                2012
# 2     Sofia        Ingeniería                2004
# 3    Alicia  Recursos Humanos                2014
```

```
df3 = pd.DataFrame({'nombre': ['Fernando', 'Antonio', 'Sofia', 'Alicia'],
                    'salario': [22000, 23000, 30000, 25000]})
print(pd.merge(df1, df3, left_on="empleado", right_on="nombre"))
#     empleado              grupo    nombre  salario
# 0  Fernando      Contabilidad  Fernando    22000
# 1   Antonio        Ingeniería   Antonio    23000
# 2     Sofia        Ingeniería     Sofia    30000
# 3    Alicia  Recursos Humanos    Alicia    25000
```

Integración de DataFrame

- Estos argumentos nos permiten fusionar DataFrame sobre distintas columnas
- Sin embargo nos aparecerá una columna con datos duplicados que podremos eliminar con la función **drop( )**

```
print(pd.merge(df1, df3, left_on="empleado",
                right_on="nombre").drop('nombre', axis=1))
#     empleado              grupo  salario
# 0   Fernando       Contabilidad    22000
# 1    Antonio         Ingeniería    23000
# 2      Sofia         Ingeniería    30000
# 3     Alicia  Recursos Humanos     25000
```

Integración de DataFrame

- ¿Qué ocurre cuando no tenemos datos tan homogéneos?

```
df6 = pd.DataFrame({'nombre': ['Pedro', 'Pablo', 'María'],
                    'comida': ['pescado', 'arroz', 'pan']},
                   columns=['nombre', 'comida'])
df7 = pd.DataFrame({'nombre': ['María', 'José'],
                    'bebida': ['vino', 'cerveza']},
                   columns=['nombre', 'bebida'])
print(df6)
#    nombre    comida
# 0   Pedro   pescado
# 1   Pablo     arroz
# 2   María       pan
print(df7)
#    nombre    bebida
# 0   María      vino
# 1    José   cerveza
```

Integración de DataFrame

- Podemos utilizar el parámetro **how** para indicar como se va a realizar la fusión. Posibles valores: **inner**, **outer**, **left** y **right**

```
print(pd.merge(df6, df7))
#    nombre comida bebida
# 0  María     pan    vino
```

```
print(pd.merge(df6, df7, how='inner'))
#    nombre comida bebida
# 0  María     pan    vino
```

```
print(pd.merge(df6, df7, how='outer'))
#    nombre    comida    bebida
# 0   Pedro  pescado       NaN
# 1   Pablo    arraz       NaN
# 2   María      pan      vino
# 3    José      NaN   cerveza
```

```
print(pd.merge(df6, df7, how='left'))
#    nombre    comida bebida
# 0   Pedro  pescado    NaN
# 1   Pablo    arraz    NaN
# 2   María      pan   vino
```

```
print(pd.merge(df6, df7, how='right'))
#    nombre comida    bebida
# 0  María     pan      vino
# 1   José     NaN   cerveza
```

UNIVERSIDAD DE MURCIA

Funciones de agregación y agrupamiento

- Las funciones de agregación son una pieza fundamental a la hora de analizar datos
- Nos permiten obtener una vista global de los datos
- Pandas soporta las operaciones típicas de agregación

Función	Descripción
count()	Número total de elementos
first(), last()	Primer y último elemento
mean(), median()	Media y mediana
min(), max()	Mínimo y máximo
std(), var()	Desviación estándar y varianza
sum(), prod()	Suma y producto de todos los elementos
Mad()	Desviación media

Funciones de agregación y agrupamiento

- A partir de aquí vamos a usar el conjunto de datos de planetas
- Las funciones **info( )** y describe nos ofrecen información de los datos

```
import seaborn as sns
planetas = sns.load_dataset('planets')
print(planetas.shape)
# (1035, 6)
print(planetas.describe())
#              number  orbital_period           mass     distance         year
# count  1035.000000      992.000000  513.000000   808.000000  1035.000000
# mean      1.785507     2002.917596    2.638161   264.069282  2009.070531
# std       1.240976    26014.728304    3.818617   733.116493     3.972567
# min       1.000000        0.090706    0.003600     1.350000  1989.000000
# 25%       1.000000        5.442540    0.229000    32.560000  2007.000000
# 50%       1.000000       39.979500    1.260000    55.250000  2010.000000
# 75%       2.000000      526.005000    3.040000   178.500000  2012.000000
# max       7.000000   730000.000000   25.000000  8500.000000  2014.000000
```

Funciones de agregación y agrupamiento

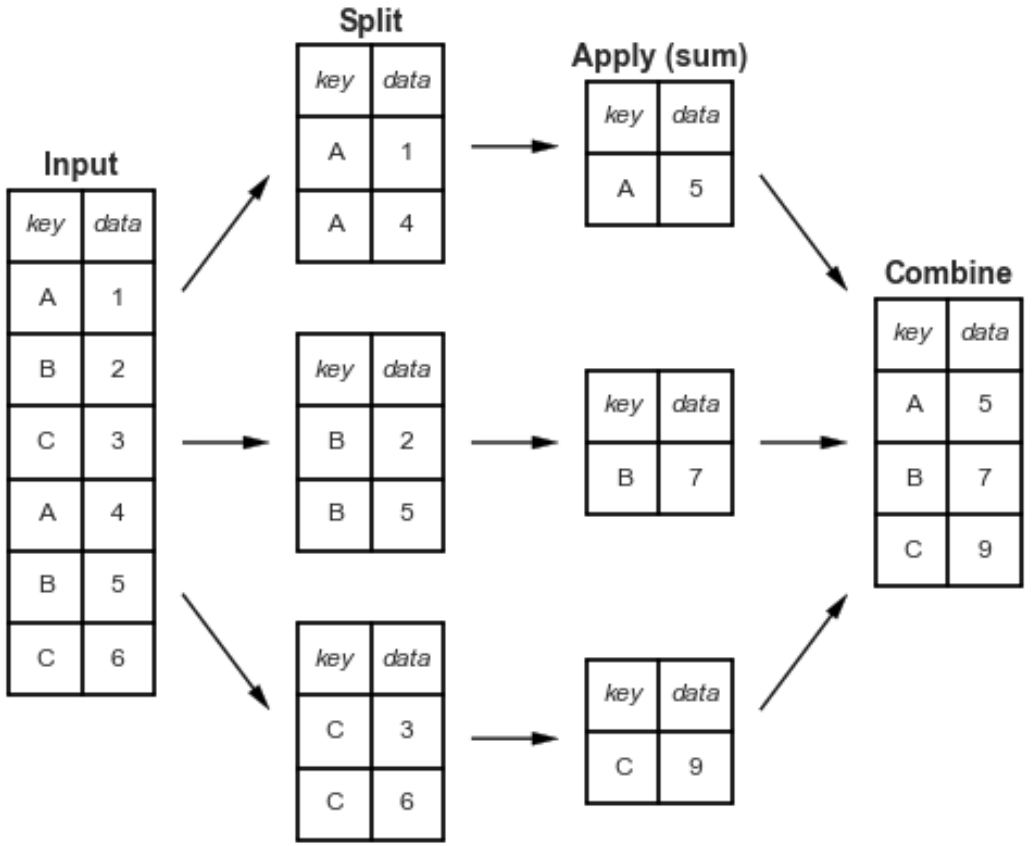
```
planetas.info()
# <class 'pandas.core.frame.DataFrame'>
# RangeIndex: 1035 entries, 0 to 1034
# Data columns (total 6 columns):
#  #   Column           Non-Null Count  Dtype
# ---  ------           --------------  -----
#  0   method           1035 non-null   object
#  1   number           1035 non-null   int64
#  2   orbital_period   992 non-null    float64
#  3   mass             513 non-null    float64
#  4   distance         808 non-null    float64
#  5   year             1035 non-null   int64
# dtypes: float64(3), int64(2), object(1)
# memory usage: 48.6+ KB
```

Funciones de agregación y agrupamiento

```
print(planetas.mean())
# number              1.785507
# orbital_period    2002.917596
# mass                2.638161
# distance          264.069282
# year              2009.070531
# dtype: float64
print(planetas[(~planetas["number"].isnull()) & (planetas["number"]==1)].mean())
# number              1.000000
# orbital_period    2641.632828
# mass                3.329634
# distance          291.743606
# year              2008.433613
# dtype: float64
# ERROR
print(planetas[~planetas["mass"].isnull()]["mass"].mean())
# 2.638160584795321 6
```

Funciones de agregación y agrupamiento

- Las funciones de agrupación nos permiten obtener un primer vistazo de los datos
- Sin embargo, a veces queremos agrupar según una determinada columna
- **Solución**: usar la función **groupby( )** que sigue un patrón **split, apply y combine**



Funciones de agregación y agrupamiento

```
print(planetas.groupby('method'))
# <pandas.core.groupby.generic.DataFrameGroupBy object at 0x
7f6b05f8d090>
planets.groupby('method')['orbital_period'].median()
# method
# Astrometry                         631.180000
# Eclipse Timing Variations         4343.500000
# Imaging                          27500.000000
# Microlensing                      3300.000000
# Orbital Brightness Modulation        0.342887
# Pulsar Timing                       66.541900
# Pulsation Timing Variations       1170.000000
# Radial Velocity                    360.200000
# Transit                              5.714932
# Transit Timing Variations           57.011000
# Name: orbital_period, dtype: float64
```

Funciones de agregación y agrupamiento

- Con **groupby( )** podemos aplicar varias funciones de agrupamiento a la vez

```
import numpy as np
print(planetas.groupby('number')["mass"].aggregate(['min', np.median, max]))
#               min  median      max
# number
# 1         0.00360  1.7900  25.000
# 2         0.00870  0.9900  21.420
# 3         0.00755  0.0580   8.060
# 4         0.00600  0.6876   4.132
# 5         0.16500  0.4860   3.530
# 6         0.00800  0.0194   0.205
# 7             NaN     NaN     NaN
```

Funciones de agregación y agrupamiento

- También podemos especificar mediante un diccionario que función aplicar a cada columna

```
print(planetas.groupby('number')[["mass","distance"]].aggregate({'mass': 'min',
                                'distance': 'max'}))
#              mass  distance
# number
# 1         0.00360   8500.00
# 2         0.00870   4080.00
# 3         0.00755   1400.00
# 4         0.00600     39.94
# 5         0.16500    368.00
# 6         0.00800    613.00
# 7             NaN    780.00
```

Funciones de agregación y agrupamiento

- También podemos filtrar datos con **filter( )**

```
def filter_func(x):
    return x['distance'].max() > 4000.0
print(planetas.groupby('number')[["number","mass","distance"]].
      filter(filter_func))
#       number   mass  distance
# 0          1   7.10     77.40
# 1          1   2.21     56.95
# 2          1   2.60     19.84
# 3          1  19.40    110.62
# 4          1  10.50    119.47
# ...      ...    ...       ...
# 1030       1    NaN    172.00
# 1031       1    NaN    148.00
# 1032       1    NaN    174.00
# 1033       1    NaN    293.00
# 1034       1    NaN    260.00

# [854 rows x 3 columns]
print(planetas.groupby('number')[["number","mass","distance"]].
      filter(filter_func)["number"].value_counts())
# 1    595
# 2    259
# Name: number, dtype: int64
```

Funciones de agregación y agrupamiento

- Y transformarlos mediante **apply( )**

```
def transformacion(x):
    # x is a DataFrame of group values
    x['dato1'] = x["distance"] / len(x['distance'])
    return x
print(planetas.groupby('number')[["number","mass","distance"]].
      apply(transformacion))
#       number   mass  distance     dato1
# 0          1   7.10     77.40  0.130084
# 1          1   2.21     56.95  0.095714
# 2          1   2.60     19.84  0.033345
# 3          1  19.40    110.62  0.185916
# 4          1  10.50    119.47  0.200790
# ...      ...    ...       ...       ...
# 1030       1    NaN    172.00  0.289076
# 1031       1    NaN    148.00  0.248739
# 1032       1    NaN    174.00  0.292437
# 1033       1    NaN    293.00  0.492437
# 1034       1    NaN    260.00  0.436975
#
# [1035 rows x 4 columns]
```

Funciones de agregación y agrupamiento

- Podemos usar la función **pivot_table( )** para hacer agrupaciones más complejas
- En estos ejemplos vamos a usar el conjunto de datos del titanic

```
titanic = sns.load_dataset('titanic')
print(titanic.info())
```

```
print(titanic.pivot_table('survived', index='sex', columns='class'))
# class      First    Second     Third
# sex
# female  0.968085  0.921053  0.500000
# male    0.368852  0.157407  0.135447
```

Funciones de agregación y agrupamiento

- La función **pivot_table( )** tiene muchas opciones de uso
- Por ejemplo, podemos usar la función **pd.cut( )** para crear intervalos y pasárselos a la función **pivot_table( )**

```
age = pd.cut(titanic['age'], [0, 18, 80])
print(titanic.pivot_table('survived', ['sex', age], 'class'))
# class                First    Second     Third
# sex    age
# female (0, 18]    0.909091  1.000000  0.511628
#        (18, 80]   0.972973  0.900000  0.423729
# male   (0, 18]    0.800000  0.600000  0.215686
#        (18, 80]   0.375000  0.071429  0.133663
```

Funciones para trabajar con cadenas

- Pandas tiene funciones que pueden trabajar con cadenas

```
nombres = pd.Series(['pedro', 'Pablo',
                     None, 'MARÍA', 'jUAN'])
print(nombres)
# 0    pedro
# 1    Pablo
# 2     None
# 3    MARÍA
# 4     jUAN
# dtype: object
```

```
print(nombres.str.capitalize())
# 0    Pedro
# 1    Pablo
# 2     None
# 3    María
# 4     Juan
# dtype: object
```

Funciones para trabajar con cadenas

- Hay muchas funciones para trabajar con cadenas

len()	lower()	translate()	islower()
ljust()	upper()	startswith()	isupper()
rjust()	find()	endswith()	isnumeric()
center()	rfind()	isalnum()	isdecimal()
zfill()	index()	isalpha()	split(()
strip()	rindex()	isdigit()	rsplit()
rstrip()	capitalize()	isspace()	partition()
lstrip()	swapcase()	istitle()	rpartition()

Trabajar con ficheros externos

- Podemos leer ficheros desde pandas mediante la familia de funciones **read_***
  - read_pickle( )
  - read_table( )
  - read_csv( )
  - read_excel( )
  - read_json( )
  - read_html( )
  - read_xml( )
  - …

Trabajar con variables categóricas

- Podemos usar las funciones OneHotEncoder y LabelEncoding de la librería Scikit-Learn para codificar las variables categóricas con dichas técnicas

```
import sklearn
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
label_encoder = LabelEncoder()
df["pais"] = label_encoder.fit_transform(df["pais"])
print(df)
#    pais  poblacion
# 0     1      47.33
# 1     0      83.13
# 2     2      67.50
# 3     3      59.07
```

Trabajar con variables categóricas

```
import sklearn
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
ohe_encoder = OneHotEncoder()
ohe_df = pd.DataFrame(ohe_encoder.fit_transform(df[["pais"]]).t
oarray())
df_concat = pd.concat([df, ohe_df], axis=1)
print(df_concat)
#    pais  poblacion    0    1    2    3
# 0     1      47.33  0.0  1.0  0.0  0.0
# 1     0      83.13  1.0  0.0  0.0  0.0
# 2     2      67.50  0.0  0.0  1.0  0.0
# 3     3      59.07  0.0  0.0  0.0  1.0
```

Trabajar con ficheros externos

- Nosotros nos centraremos en el uso de la función **read_csv( )**, cuya signatura es demasiado extensa
- Se puede consultar en: https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html
- Con dicha función podemos leer tanto archivos locales que tengamos en nuestro equipo como ficheros que se encuentren en una dirección de Internet

Trabajar con ficheros externos

- A continuación, se muestra un ejemplo de carga de un conjunto de datos que contiene datos de distintas especies de Iris

```
import pandas as pd
iris = pd.read_csv(
    'https://raw.githubusercontent.com/mwaskom/seaborn-
data/master/iris.csv')
print(iris.head())
#    sepal_length  sepal_width  petal_length  petal_width species
# 0           5.1          3.5           1.4          0.2  setosa
# 1           4.9          3.0           1.4          0.2  setosa
# 2           4.7          3.2           1.3          0.2  setosa
# 3           4.6          3.1           1.5          0.2  setosa
# 4           5.0          3.6           1.4          0.2  setosa
```

Trabajar con ficheros externos

- En el siguiente código hacemos lo mismo, pero subiendo el fichero desde nuestro propio ordenador

```
import pandas as pd
import io
from google.colab import files
uploaded = files.upload()
iris = pd.read_csv(io.BytesIO(uploaded[iris.csv']))
print(iris.head())
#    sepal_length  sepal_width  petal_length  petal_width species
# 0           5.1          3.5           1.4          0.2  setosa
# 1           4.9          3.0           1.4          0.2  setosa
# 2           4.7          3.2           1.3          0.2  setosa
# 3           4.6          3.1           1.5          0.2  setosa
# 4           5.0          3.6           1.4          0.2  setosa
```

Trabajar con ficheros externos

- Una vez que acabamos de trabajar con el conjunto de datos, podemos volver a guardarlo en un fichero a disco mediate las funciones de la familia **to_***
  - to _pickle( )
  - to _table( )
  - to _csv( )
  - to _excel( )
  - to _json( )
  - to _html( )
  - to_xml( )
  - ...

Escalado de características

- El escalado de características se puede hacer de forma manual o bien utilizar librerías. En este caso haremos uso de las clases StandardScaler y MinMaxScaler de la librería Scikit-Learn
- Además, esta librería también implementa las demás clases de escalado que hemos visto en teoría
  - RobustScaler
  - PowerTransformer
  - QuantileTransformer
  - MaxAbsScaler
  - ...

Escalado de Características

```
import sklearn
from sklearn.preprocessing import StandardScaler
caracteristicas = ["sepal_length", "sepal_width", "petal_length", "pet
al_width"]
standard_scaler = StandardScaler()
standard_scaler.fit(iris[caracteristicas])
iris[caracteristicas] = standard_scaler.transform(iris[caracteristicas
])
print(iris.head())
#    sepal_length  sepal_width  petal_length  petal_width species
# 0     -0.900681     1.019004     -1.340227    -1.315444  setosa
# 1     -1.143017    -0.131979     -1.340227    -1.315444  setosa
# 2     -1.385353     0.328414     -1.397064    -1.315444  setosa
# 3     -1.506521     0.098217     -1.283389    -1.315444  setosa
# 4     -1.021849     1.249201     -1.340227    -1.315444  setosa
```

Escalado de Características

```
import sklearn
from sklearn.preprocessing import MinMaxScaler
caracteristicas = ["sepal_length", "sepal_width", "petal_length", "petal
_width"]
minmax_scaler = MinMaxScaler()
minmax_scaler.fit(iris[caracteristicas])
iris[caracteristicas] = minmax_scaler.transform(iris[caracteristicas])
print(iris.head())
#    sepal_length  sepal_width  petal_length  petal_width species
# 0      0.222222     0.625000      0.067797     0.041667  setosa
# 1      0.166667     0.416667      0.067797     0.041667  setosa
# 2      0.111111     0.500000      0.050847     0.041667  setosa
# 3      0.083333     0.458333      0.084746     0.041667  setosa
# 4      0.194444     0.666667      0.067797     0.041667  setosa
```

Tratando con valores faltantes

- Podemos trabajar directamente con valores faltantes modificando el conjunto de datos de forma manual
- Sin embargo, existen librerías que hacen la práctica totalidad del trabajo por nosotros
- En este caso volveremos a usar la librería Scikit-Learn que dispone de varias clases para trabajar con valores faltantes
  - SimpleImputer
  - KNNImputer
  - MissingIndicator

Tratando con valores faltantes

- La clase SimpleImputer nos permite rellenar los valores faltantes mediante un valor constante o mediante algún estadístico

```
import numpy as np
from sklearn.impute import SimpleImputer
imp = SimpleImputer(missing_values=np.nan, strategy='mean')
X = [[100, 2],
     [np.nan, 3],
     [2, 6]]
imp.fit(X)
print(imp.transform(X))
# [[8, 2],
#  [np.nan, 3],
#  [7, 6]]
```

Tratando con valores faltantes

- La clase KNNImputer nos permite rellenar los valores faltantes mediante utilizando el vecino más próximo

```
import numpy as np
from sklearn.impute import KNNImputer
nan = np.nan
X = [[1, 2, nan],
     [3, 4, 3],
     [nan, 6, 5],
     [8, 8, 7]]
imputer = KNNImputer(n_neighbors=2, weights="uniform")
imputer.fit_transform(X)
# array([[1. , 2. , 4. ],
#        [3. , 4. , 3. ],
#        [5.5, 6. , 5. ],
#        [8. , 8. , 7. ]])
```

Tratando con valores faltantes

- La clase MissingIndicator nos permite crear una máscara para indicar que valores falta. Podemos usar el argumento missing_values para especificar el valor que representa a los datos faltantes (por defecto es **np.nan**)

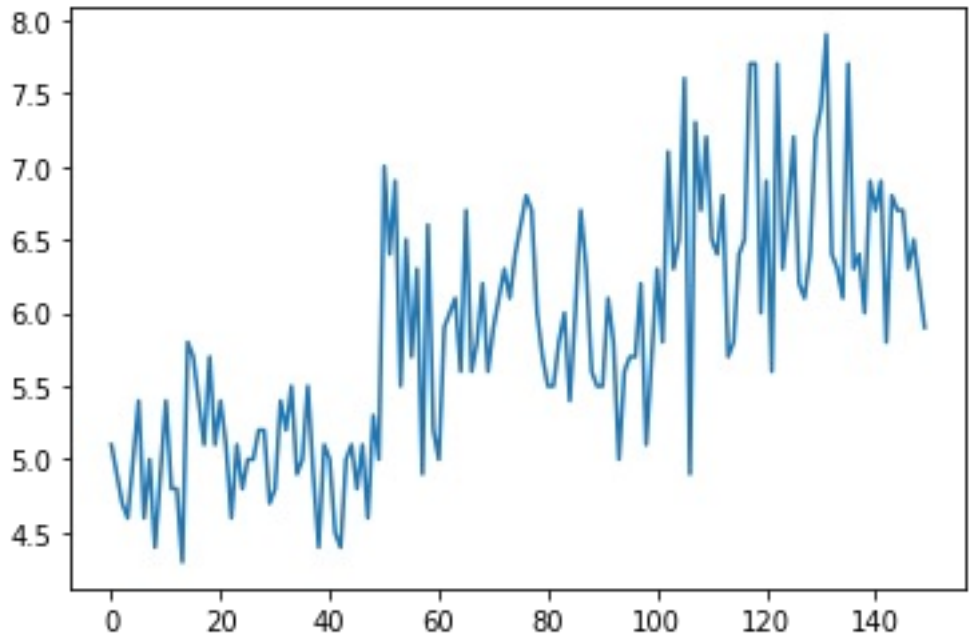
```
from sklearn.impute import MissingIndicator
X = np.array([[-1, -1, 1, 3],
              [4, -1, 0, -1],
              [8, -1, 1, 0]])
indicator = MissingIndicator(missing_values=-1)
mask_missing_values_only = indicator.fit_transform(X)
mask_missing_values_only
# array([[ True,  True, False],
#        [False,  True,  True],
#        [False,  True, False]])
```

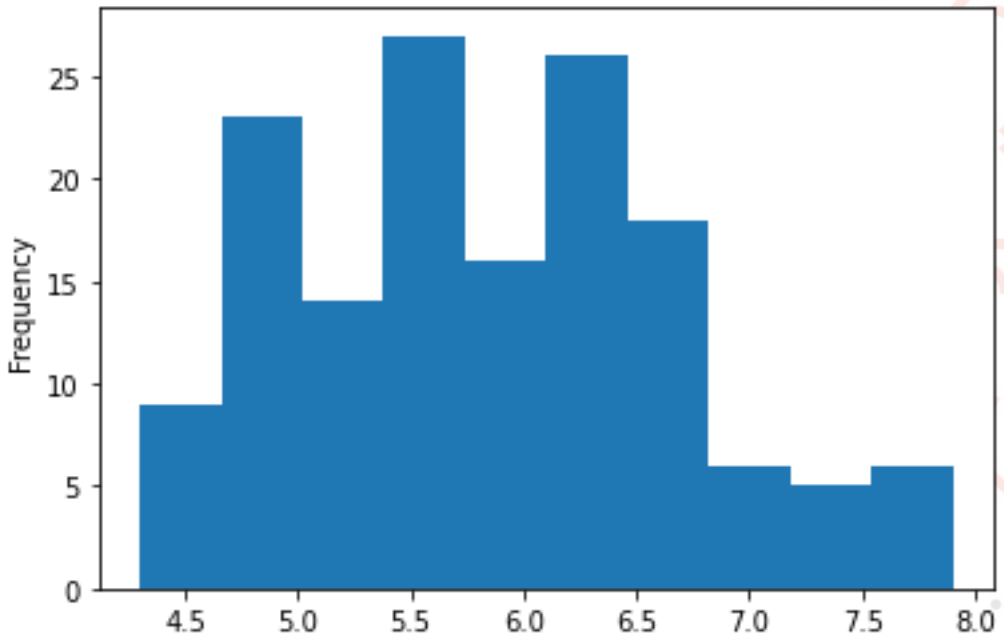
Visualización de DataFrame

- Con la función **plot( )** tanto del objeto DataFrame como Serie podemos mostrar gráficos de forma sencilla

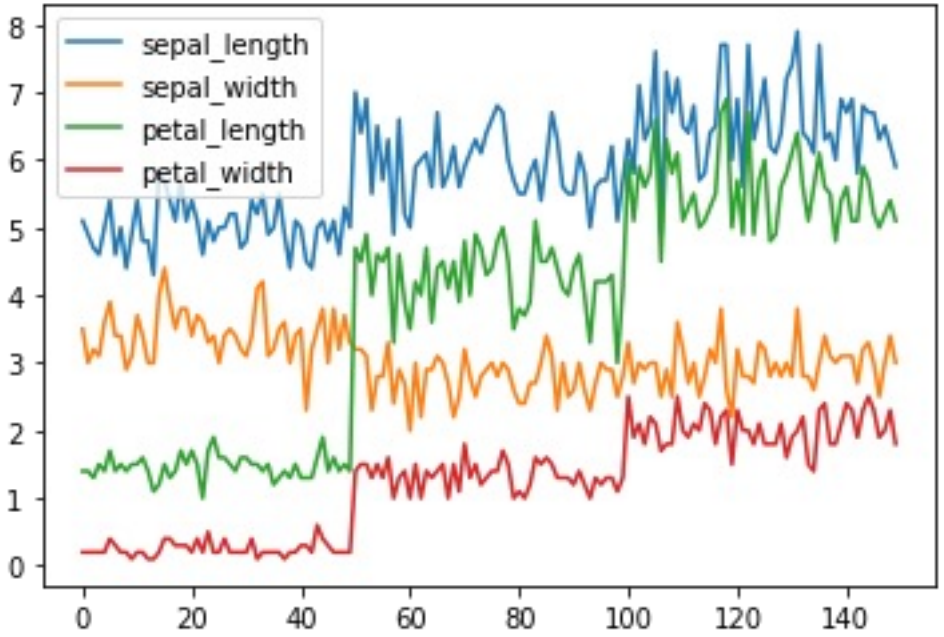
```
iris["sepal_length"].plot()
iris["sepal_length"].plot(kind="hist")
iris.plot()
iris.plot(x="sepal_length", y="sepal_width", kind="scatter")
```

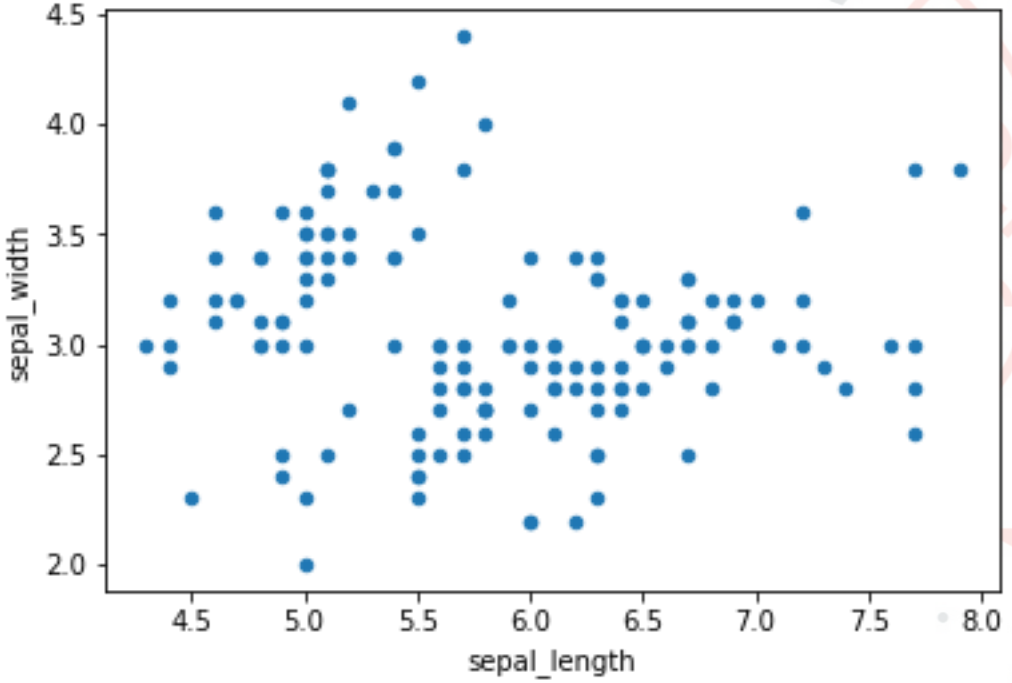
Visualización de datos





Visualización de datos





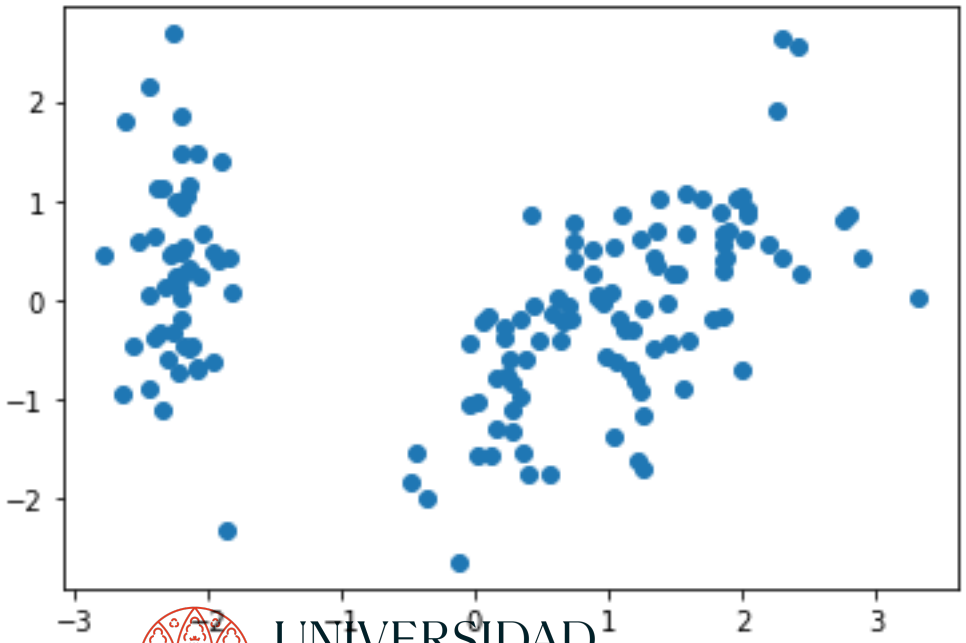
Visualización de datos

- Para realizar las técnicas de PCA y t-SNE podemos usar nuevamente la librería de Scikit-Learn

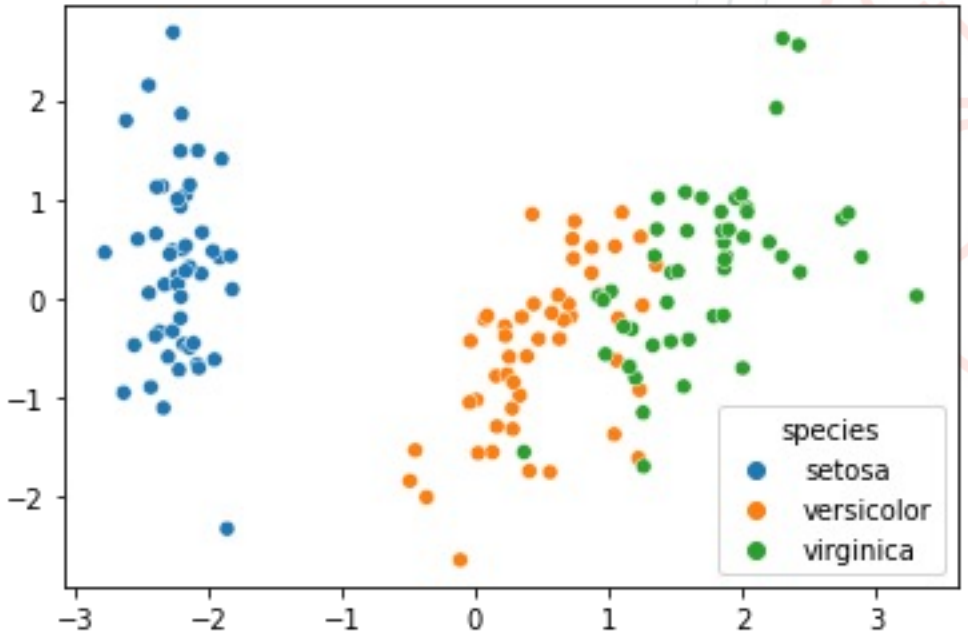
```
import sklearn
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
caracteristicas = ["sepal_length", "sepal_width", "petal_length", "
petal_width"]
pca.fit(iris[caracteristicas])
iris_transformado = pca.transform(iris[caracteristicas])
print(iris_transformado[:5])
# [[-2.26470281  0.4800266 ]
#  [-2.08096115 -0.67413356]
#  [-2.36422905 -0.34190802]
#  [-2.29938422 -0.59739451]
#  [-2.38984217  0.64683538]]
```

Visualización de datos

```
import matplotlib
import matplotlib.pyplot as plt
plt.scatter(iris_transformado[:,0],
 iris_transformado[:,1])
```




```
import seaborn as sns
sns.scatterplot(iris_transformado[:,0],
iris_transformado[:,1], iris["species"])
```

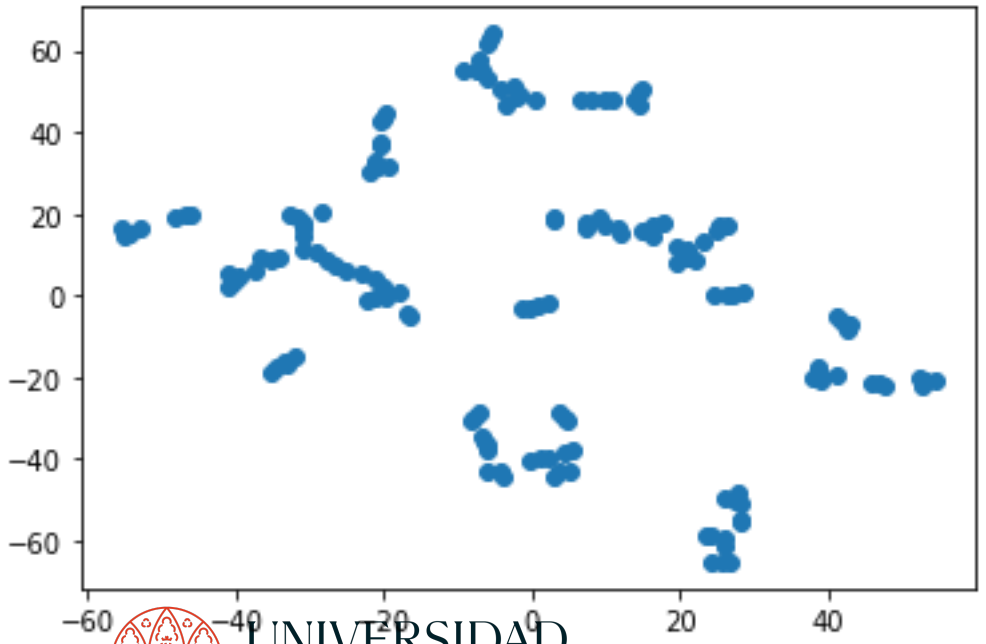


Visualización de datos

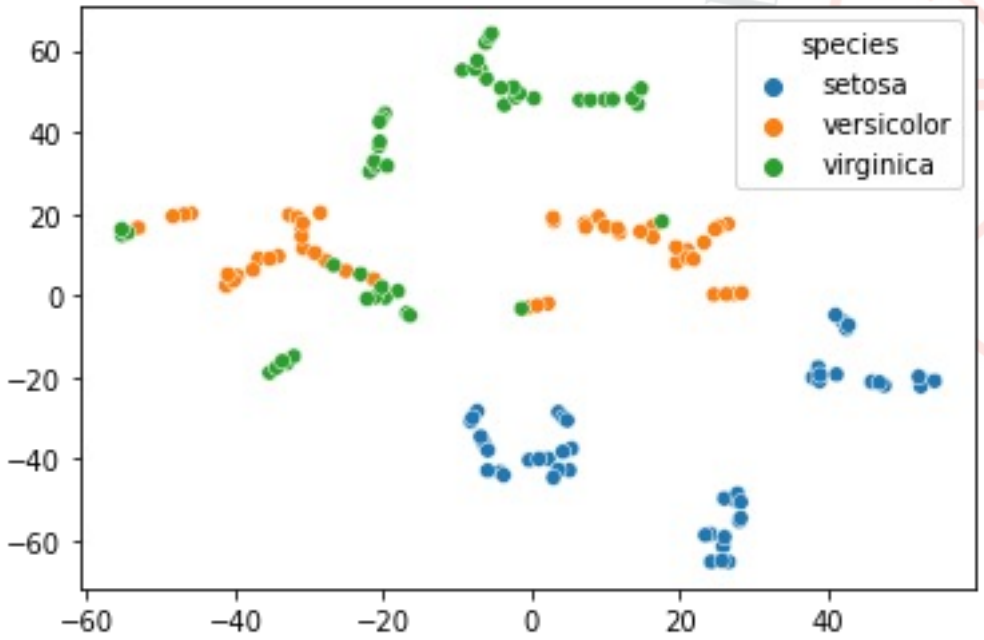
```
from sklearn.manifold import TSNE
caracteristicas = ["sepal_length", "sepal_width", "petal_length",
 "petal_width"]
tsne = TSNE(n_components=2, learning_rate='auto', init='random',
perplexity=3)
iris_transformado = tsne.fit_transform(iris[caracteristicas])
print(iris_transformado[:5])
# [[ -6.21025  -36.863026]
#  [ 53.052834 -20.904736]
#  [ 38.036842 -20.144493]
#  [ 41.222305 -19.37771 ]
#  [ -8.172599 -30.880867]]
```

Visualización de datos

```
import matplotlib
import matplotlib.pyplot as plt
plt.scatter(iris_transformado[:,0],
 iris_transformado[:,1])
```




```
import seaborn as sns
sns.scatterplot(iris_transformado[:,0],
iris_transformado[:,1], iris["species"])
```



Ejercicios

1. Crea un DataFrame con la siguiente información

Nombre	Grupo Sanguíneo	Edad	Peso	Ritmo cardíaco	Presión Sistólica	Presión Diastólica
Eduardo	B+	40	70	70	129	-
Ana	0+	35	69	60	133	86
Alejandro	AB+	37	74	68	125	82
Álvaro	0+	24	70	50	110	70
Aitana	A+	48	72	68	-	82
María	A+	53	67	87	130	84
Sofia	B-	67	65	110	155	100
Antonio	A+	25	74	-	126	89
Fernando	AB+	38	75	77	131	90
Laura	0+	21	70	69	127	87

Ejercicios

2. Crea una nueva columna en el DataFrame anterior donde aparezca el cálculo del Indice de Masa Corporal (IMC) que se calcula de la siguiente forma

$$IMV = \frac{peso}{altura^2}$$

3. Convierte la columna Grupo Sanguíneo de 2 formas distintas (sin eliminar la columna original)
   - Usando un Label Encoding
   - Usando un esquema de One Hot Encoding
4. Elimina la columna original de Grupo Sanguíneo

Ejercicios

5. Une el DataFrame anterior con el siguiente

Nombre	Ciudad	Nivel de Estudios	Azucar	Colesterol
Eduardo	Valencia	Primaria	80	156
Ana	Murcia	Primaria	82	167
Manuel	Madrid	Primaria	114	204
Aitana	Barcelona	Primaria	94	226
María	Sevilla	Universidad	130	167
Antonio	Madrid	Formación Profesional	83	190
Fernando	Barcelona	Secundaria	82	199
Angela	Murcia	Secundaria	103	192

Ejercicios

6. Calcula las medias y las desviaciones típicas de todos los valores numéricos del DataFrame
7. Crea una nueva columna en el DataFrame que se ha unido anteriormente que indique si la persona tiene colesterol alto o no
    - Valores de 125 a 200 son normales
    - Valores por encima de 200 son valores altos
8. Crea una columna que indique si la persona tiene niveles normales de azúcar, prediabetes o diabetes
    - Valores de azúcar menores de 100 son normales
    - Valores entre 100 y 125 (ambos inclusive) indican prediabetes
    - Valores superiores a 126 indican diabetes

Grado en Gestión de la Información y Contenidos Digitales

Procesamiento y Visualización de Datos

Práctica 3. Numpy

Bloque 1: Procesamiento de datos

UNIVERSIDAD DE MURCIA

Facultad de Informática

Introducción

- **Numpy** (Numerical Python) es la librería que utilizaremos en Python para poder manipular datos
- Los datos se encuentran en una gran variedad de formatos, pero en esencia todos ellos son matrices de datos
- Las imágenes se pueden ver como una matriz de ancho x alto donde cada posición indica la intensidad del pixel
- Los clip de audio pueden ser representados por un vector donde cada posición indica la intensidad en un instante t determinado
- Los textos también pueden ser representados mediante vectores y matrices mediante diversas representaciones

Introducción

- Los arrays de Numpy son el núcleo de prácticamente todo el ecosistema de Data Science y de procesamiento de datos de Python
- Los arrays de Numpy son como listas de Python pero que ofrecen formas eficientes para cargar y manejar datos
- Para saber si tenemos Numpy instalado y cuál es su versión podemos ejecutar lo siguiente

```
import numpy
print(numpy.__version__)     # 1.23.4
```

Introducción

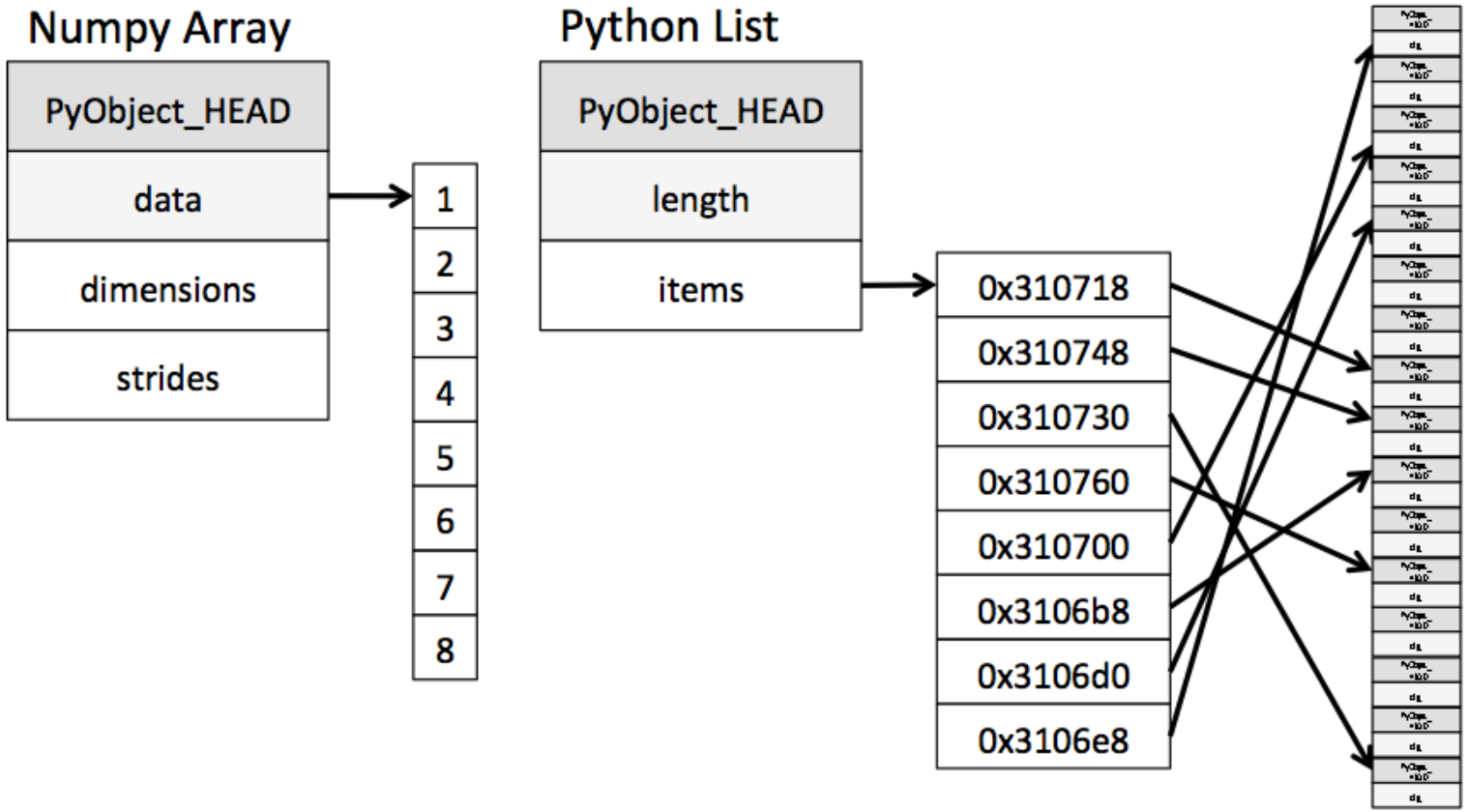
- Aunque es frecuente importar numpy de la siguiente forma

```
import numpy as np
print(np.__version__)          # 1.23.4
```

- Esta es la forma en la que durante el curso vamos a importar numpy. Nos permite llamar a las funciones de numpy utilizando el prefijo **np**, en vez del prefijo numpy

Introducción

- Diferencia entre una lista de Python y un array de Numpy



Creación de arrays

- Para crear arrays podemos utilizar la función **array( )**

```
np.array([1, 4, 2, 5, 3])          # array([1, 4, 2, 5, 3])
np.array([3.14, 4, 2, 3])          # array([ 3.14,  4.  ,  2.  ,  3.  ])
```

- Sin embargo, a diferencia de las listas de Python, los arrays de numpy **no pueden albergar distintos tipos de datos**. Podemos forzar el tipo de datos pasando el parámetro **dtype** a la función **array( )**

```
np.array([1, 2, 3, 4], dtype='float32')
# array([ 1., 2., 3., 4.], dtype=float32)
np.array([1, 2, 3, 4.6], dtype='int')
# array([1, 2, 3, 4])
```

Creación de arrays

- También podemos crear arrays multidimensionales

```
np.array([[1,2,3],[4,5,6],[7,8,9]])
# array([[1, 2, 3],
#        [4, 5, 6],
#        [7, 8, 9]])
np.array([range(i, i + 3) for i in [2, 4, 6]])
# array([[2, 3, 4],
#        [4, 5, 6],
#        [6, 7, 8]])
```

Creación de arrays

- Existen diversas funciones de numpy que permiten inicializar arrays de manera concreta
- **zeros( )**. Permite inicializar un array a valor 0
- **ones( )**. Permite inicializar un array a valor 1
- **full( )**. Permite inicializar un array a un valor determinado
- **arange( )**. Permite crear un array desde un valor inicial a un valor final con incrementos determinados
- **linspace( )**. Permite crear un array espaciado uniformemente a partir de un intervalo dado
- **eye( )**. Permite crear una matriz identidad

Creación de arrays

```
np.zeros(10, dtype=int)          # array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
np.ones((2, 2), dtype=float)     # array([[ 1.,  1. ],
                                 #        [ 1.,  1. ]])
np.full((2, 2), 1.5)             # array([[ 1.5,  1.5 ],
                                 #        [ 1.5,  1.5 ]])
np.arange(0, 10, 2)              # array([ 0,  2,  4,  6,  8])
np.linspace(0, 1, 5)             # array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
np.eye(2)                        # array([[ 1.,  0. ],
                                 #        [ 0.,  1. ]])
```

Creación de arrays

- También podemos crear arrays con valores aleatorios
- **random.random( )**. Permite crear un array de números aleatorios entre 0 y 1 siguiendo una distribución uniforme
- **random.normal( )**. Permite crear un array de numero aleatorios siguiendo una distribución normal
- **random.randint( )**. Permite crear un array de números enteros aleatorios

Creación de arrays

```
np.random.random((3, 3))
# array([[ 0.99844933,   0.52183819],
#        [ 0.08007488,  0.45429293]])
np.random.normal(0, 1, (2, 2))
# array([[ 0.29074237,  0.71914731],
#        [-0.24229766, -0.54525217]])
np.random.randint(0, 10, (2, 2))
# array([[8, 4],
#        [4, 2]])
```

Tipos de datos en numpy

Tipo de dato	Descripción
int8	Entero con signo de 8 bits o byte (-128 to 127)
int16	Entero con signo de 16 bits (-32768 to 32767)
int32	Entero con signo de 32 bits (-2147483648 to 2147483647)
int64	Entero con signo de 64 bits Integer (-9223372036854775808 to 9223372036854775807)
uint8	Entero sin signo de 8 bits (0 to 255)
uint16	Entero sin signo de 16 bits (0 to 65535)
uint32	Entero sin signo de 32 bits (0 to 4294967295)
uint64	Entero sin signo de 64 bits (0 to 18446744073709551615)
float16	Float de media precisión (1 bit de signo, 5 bits de exponente, 10 bits de mantisa)
float32	Float de simple precisión (1 bit de signo, 8 bits de exponente, 23 bits de mantisa)
float64	Float de doble precisión (1 bit de signo, 11 bits de exponente, 52 bits de mantisa)

Atributos de los arrays

- Los arrays creados en numpy tienen una serie de atributos que pueden ser consultados
- **ndim**. Devuelve el número de dimensiones del array
- **shape**. Devuelve el tamaño de cada dimensión del array
- **size**. Devuelve el tamaño total del array
- **dtype**. Devuelve el tipo de dato que contiene el array
- **itemsize**. Devuelve el tamaño que ocupa en memoria cada elemento del array (en bytes)
- **nbytes**. Devuelve el tamaño total que ocupa el array en memoria (en bytes)

Atributos de los arrays

```
x = np.random.randint(0, 10, (2, 2))
# array([[8, 4],
#        [4, 2]])
print("x ndim: ", x.ndim)
# x ndim:  2
print("x shape:", x.shape)
# x shape: (2, 2)
print("x size: ", x.size)
# x size:  4
print("dtype:", x.dtype)
# dtype: int32
print("itemsize:", x.itemsize, "bytes")
# itemsize: 4 bytes
print("nbytes:", x.nbytes, "bytes")
# nbytes: 16 bytes
```

Accediendo a elementos del array

- Los arrays de numpy se pueden acceder de la misma forma que las listas de Python

```
x = np.random.randint(0,10,4)                # array([5, 0, 6, 7])
print(x[0])                                  # 5
print(x[-1])                                 # 7
print(x[1:3])                                # array([0, 6])
print(x[2:])                                 # array([6, 7])
x = np.random.randint(0, 10, (2, 2))         # array([[7, 3],
                                             #        [0, 1]])
print(x[0,0])                                # 7
print(x[1,0])                                # 0
x[0,0] = 20
print(x[:, 0])                               # array([20  0])
print(x[0,:])                                # array([20, 3])
```

Accediendo a elementos del array

- Cuidado, cuando accedemos mediante **slices** estamos accediendo a una **view** del array y no una copia. Solución: usar la función **copy( )**

```
x = np.random.randint(0, 10, (3, 3))     # array([[2, 8, 5],
                                         #        [9, 6, 1],
                                         #        [0, 0, 4]])

new_x = x[:2, :2]
new_x[0, 0] = 15
print(new_x)                             # array([[15,  8],
                                         #        [ 9,  6]])
print(x)                                 # array([[15,  8,  5],
                                         #        [ 9,  6,  1],
                                         #        [ 0,  0,  4]])

new_x = x[:2, :2].copy()
print(new_x)                             # array( [[15,  8],
                                         #         [ 9,  6]])
new_x[0, 0] = 20
print(x)                                 # array([[15,  8,  5],
                                         #        [ 9,  6,  1],
                                         #        [ 0,  0,  4]])
```

Cambiando el shape de un array

- Podemos cambiar el shape de un array mediante la función reshape

```
x = np.arange(1, 10)
# array([1, 2, 3, 4, 5, 6, 7, 8, 9,])
x = x.reshape((3, 3))
# array([[1, 2, 3],
#        [4, 5, 6],
#        [7, 8, 9]])
```

Concatenar arrays

- Podemos usar las funciones **concatenate( )**, **hstack( )**, **vstack( )** para concatenar arrays

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])                    # array([1, 2, 3, 3, 2, 1])
x = np.array([[1, 2, 3],[4, 5, 6]])
np.concatenate([x, x])                    # Mismo resultado que usar np.vstack([x,x])
                                          # array([[1, 2, 3],
                                          #        [4, 5, 6],
                                          #        [1, 2, 3],
                                          #        [4, 5, 6]])
np.concatenate([grid, grid], axis=1)      # Mismo resultado que usar np.hstack([x,x])
                                          # array([[1, 2, 3, 1, 2, 3],
                                          #        [4, 5, 6, 4, 5, 6]])
```

Operaciones vectoriales

- Numpy permite realizar operaciones vectoriales sobre los arrays sin necesidad de usar bucles. Es recomendable usarlas ya que ofrecen mejor rendimiento

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
z = np.zeros(3)
for i in range(len(x)):
    z[i] = x[i] + y[i]          # array([4, 4, 4])
z = x + y                       # array([4, 4, 4])
```

Operaciones Vectoriales

Operador	Función	Descripción
+	np.add()	Suma de dos operandos
-	np.subtract()	Resta de dos operandos
-	np.negative()	Negación de un número
*	np.multiply()	Multiplicación de dos operandos
/	np.divide()	División entre dos operandos
//	np.floor_divide()	Divisón entera entre dos operandos
**	np.power()	Porencia entre dos operandos
%	np.mod	Modulo entre dos operandos

Operadores Vectoriales

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
z = x + y
# array([4, 4, 4]). Equivalente a  z = np.add(x,y)
z = x - y
# array([-2,  0,  2]). Equivalente a  z = np.subtract(x,y)
z = x * y
# array([3, 4, 3]). Equivalente a  z = np.mulitply(x,y)
z = x / y
# array([0.33333333, 1.  , 3. ]). Equivalente a  z = np.divide(x,y)
z = x // y
# array([0, 1, 3]). Equivalente a  z = np.floor_divide  (x,y)
z = x ** y
# array([1, 4, 3]). Equivalente a  z = np.power(x,y)
z = x % y
# array([1, 0, 0]). Equivalente a  z = np.mod(x,y)
```

Funciones de Agregación

- Permiten obtener resúmenes estadísticos de los datos con los que trabajamos

Función	Descripción
np.sum()	Suma los elementos de un array
np.mean()	Calcula la media de un array
np.std()	Calcula la desviación típica de un array
np.var()	Calcula la varianza de un array
np.min()	Devuelve el elemento mínimo de un array
np.max()	Devuelve el elemento máximo de un array
np.argmin()	Devuelve el índice del elemento mínimo de un array
np.argmax()	Devuelve el índice del elemento máximo de un array
np.median()	Calcula la mediana de un array
np.any()	Evalúa si cualquier elemento de un array es True
np.all()	Evalúa si todos los elementos de un array son True

Funciones de Agregación

```
x = np.array([1, 2, 3])
y = np.array([True, False, False])
z = np.array([True, True, True])

print(np.sum(x))            # 6
print(np.mean(x))           # 2.0
print(np.std(x))            # 0.816496580927726
print(np.var(x))            # 0.6666666666666666
print(np.min(x))            # 1
print(np.max(x))            # 3
print(np.argmin(x))         # 0
print(np.argmax(x))         # 2
print(np.median(x))         # 2.0
print(np.any(y))            # True
print(np.all(z))            # True
```

Broadcasting

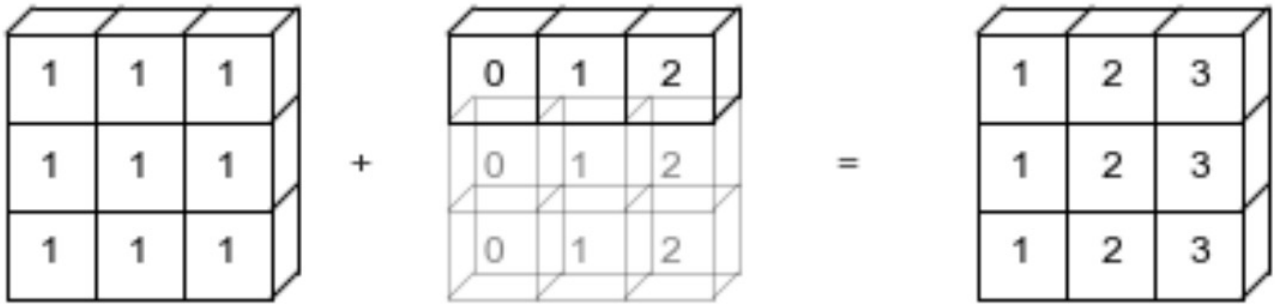
- El **broadcasting** es un conjunto de reglas para aplicar las funciones vectoriales vistas anteriormente sobre arrays **con distinto shape**
- Antes hemos visto que podemos realizar operaciones sobre 2 arrays con las mismas dimensiones
- Sin embargo, también podemos sumar un número escalar a un array

```
x = np.array([1, 2, 3])
print(x + 2)                    # array([3, 4, 5])
y = np.ones((3,3))              # array([[1, 1, 1],
                                #        [1, 1, 1],
                                #        [1, 1, 1]])
print(y + 2)                    # array([[3, 3, 3],
                                #        [3, 3, 3],
                                #        [3, 3, 3]])
```

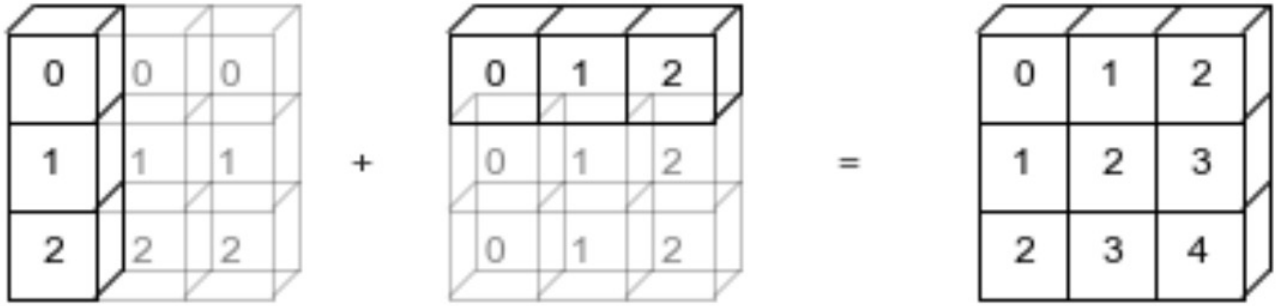
Broadcasting

`np.arange(3)+5`

`np.ones((3, 3))+np.arange(3)`



`np.arange(3).reshape((3, 1))+np.arange(3)`



Broadcasting

- Hay **tres reglas** que rigen el funcionamiento del broadcasting
- **Regla 1**: Si dos arrays difieren en el número de dimensiones, el shape de aquél con menor dimensión es modificado para añadir una nueva dimensión en su lado inicial (izquierda)
- **Regla 2**: Si el shape de los arrays no coincide en ninguna dimensión, el array con shape igual a 1 se estira para que coincida con el shape del otro array
- **Regla 3**: Si en alguna de las dimensiones, los tamaños no coinciden y ninguno es igual a 1, se lanza un error

Broadcasting

```
x = np.ones((2, 3))                 # array([[1, 1, 1],
                                    #        [1, 1, 1]])
y = np.array([1, 2, 3])
print(x.shape)                      # (2, 3)
print(y.shape)                      # (3, )
# Por la regla número 1 modificamos el shape de y
# (3, ) -> (1, 3)
# Por la regla número 2 modificamos la primera
# dimensión de y para que coincida con la primera
# dimensión de x
# (1, 3) -> (2, 3)
```

Broadcasting

```
x = np.ones((3, 2))                 # array([[1, 1, 1],
                                    #        [1, 1, 1]])
y = np.array([1, 2, 3])
print(x.shape)                      # (3, 2)
print(y.shape)                      # (3, )
# Por la regla número 1 modificamos el shape de y
# (3, ) -> (1, 3)
# Por la regla número 2 modificamos la primera
# dimensión de y para que coincida con la primera
# dimensión de x
# (1, 3) -> (3, 3)
# Vemos que el shape de x (3, 2) sigue siendo incompatible
# con el shape de y (3,3), por tanto, por la regla número 3
# se lanzaría un error:
# ValueError: operands could not be broadcast together
# with shapes (3,2) (3,)
```

Operadores de comparación sobre arrays

- Anteriormente hemos visto operaciones aritméticas sobre arrays
- También podemos realizar operaciones de comparación sobre arrays

```
x = np.array([1, 2, 3, 4, 5])
print(x < 3)                          # array([ True,  True, False, False, False])
print(x > 3)                          # array([False, False, False,  True,  True])
print(x <= 3)                         # array([ True,  True,  True, False, False])
print(x >= 3)                         # array([False, False,  True,  True,  True])
print(x != 3)                         # array([ True,  True, False,  True,  True])
print(x == 3)                         # array([False, False,  True, False, False])
x = np.random.randint(10, size=(3, 4))   # array([[2, 0, 4, 2],
                                      #        [0, 1, 4, 2],
                                      #        [6, 4, 1, 5]])
print(x < 6)                          # array([[ True,  True,  True,  True],
                                      #        [ True,  True,  True,  True],
                                      #        [False,  True,  True,  True]])
```

Operaciones de comparación sobre arrays

- Podemos usar también las funciones de numpy en vez de los operadores de comparación

Operador	Función de numpy
==	np.equal(x, y)
!=	np.not_equal(x, y)
<	np.less(x, y)
>	np.greater(x, y)
<=	np.less_equal(x, y)
>=	np.greater_equal(x, y)

Trabajando con arrays booleanos

- Anteriormente hemos visto funciones como **np.any()** o **np.all()** que evaluaban si alguno o todos los elementos de un array son True, respectivamente
- También podemos contar el número de elementos True en un array
- Y podemos realizar operaciones booleanas con los operadores & (y), | (o), ~ (negación) y ^ (o exclusivo)

```
x = np.array([1, 2, 3, 4, 5])
print(np.count_nonzero(x < 3))    # 2
print(np.sum(x < 3))              # 2
print(((x < 2) | ( x > 3)))       # array([ True, False, False,  True,  True])
print(~(x < 2))                   # array([False,  True,  True,  True,  True])
```

Trabajando con arrays booleanos

- Tambien podemos usar los arrays booleanos para indexar arrays de numpy

```
x = np.array([1, 2, 3, 4, 5])
mascara = np.array([True, False, False, False, False])
print(x[mascara])
# array([1])
print(x[x < 2])
# array([1])
print(np.mean(x[x > 2]))
# 4.0
```

Ordenando arrays

- Podemos ordenar arrays mediante la función **np.sort( )**
- Podemos ordenar los índices de un array mediante la función **np.argsort()**
- Podemos también encontrar la posición de un elemento concreto o de elementos que cumplen cierta condición con la función **np.where( )**. Se pueden utilizar operadores binarios en la condición
- En arrays con más de una dimensión, mediante el parámetro **axis** podemos elegir sobre que dimensión queremos que se apliquen las funciones anteriores

Ordenando arrays

```
x = np.array([10, 2, 1, 9, 3])
print(np.sort(x))                    # array([ 1,  2,  3,  9, 10])
print(np.argsort(x))                 # array([2, 1, 4, 3, 0])
x = np.array([[10, 3, 2], [2, 9, 4], [5, 1, 2]])
print(np.sort(x, axis=0))            # array([[ 2,  1,  2],
                                     #        [ 5,  3,  2],
                                     #        [10,  9,  4]])
print(np.sort(x, axis = 1))          # array([[ 2,  3, 10],
                                     #        [ 2,  4,  9],
                                     #        [ 1,  2,  5]])
print(np.where(x == 3))              # (array([0]), array([1]))
print(np.where(x > 4))               # (array([0, 1, 2]), array([0, 1, 0]))
```

Concatenar y dividir arrays

- Podemos usar las funciones **split( )**, **hsplit( )**, **vsplit( )** para dividir arrays

```
import numpy as np
x = np.array([[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12],
[13, 14, 15, 16]])
y = np.vsplit(x, 2) # np.split(x, 2,
axis=0)
print(y)
# [array([[1, 2, 3, 4],
# [5, 6, 7, 8]]), array([[ 9, 10, 11,
12],
# [13, 14, 15, 16]])]
```

```
import numpy as np
x = np.array([[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12],
[13, 14, 15, 16]])

y = np.hsplit(x, 2) # np.split(x, 2, axis=1)
print(y)
# [array([[ 1, 2],
# [ 5, 6],
# [ 9, 10],
# [13, 14]]), array([[ 3, 4],
# [ 7, 8],
# [11, 12],
# [15, 16]])]
```