Análisis y Búsqueda de Respuestas en Datos.

Caso práctico

kkolosov (Dominio público)

El sector de la sanidad ha cambiado considerablemente en los últimos años.

Tanto grupos de investigación como hospitales siempre se han dedicado a intentar encontrar las mejores técnicas, los mejores tratamientos, los mejores medicamentos o las mejores vacunas. Todo ello haciendo uso de las mejores herramientas a su alcance, las cuales son tipos muy variados.

En la mayoría de los casos tal investigación versa sobre cosas muy pequeñas. Bacterias, virus, moléculas, ...

Pero ahora se ha dado un paso más allá, hasta llegar a lo que no tiene ni tamaño ni masa, porque es etéreo.

Algo que les permite predecir enfermedades de un modo mucho más temprano. Algo que les permite descubrir nuevos medicamentos. Algo que les permite personalizar el cuidado dado a cada paciente.

Estamos hablando, por supuesto, de los datos.

En esta unidad de trabajo vamos a tratar lo reference al análisis y la búsqueda de respuestas en datos.

Comenzamos viendo una serie de conceptos generales para después ver qué niveles existen en cuanto a analítica de datos y cuáles con las principales metodologías que se suelen emplear en minería de datos.

A continuación veremos R, un lenguaje interpretado especializado en cálculos estadísticos y en análisis de datos.

Por último conoceremos el lenguaje Python, de propósito general pero con varias librerías especializadas que le permiten trabajar muy bien analizando datos.

Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.

Aviso Legal

1.- Conceptos básicos.

Caso práctico

Según una antigua leyenda, cuando Dios creó el mundo puso sobre la tierra 3 varillas de diamante y 64 discos de oro, todos ellos de tamaño diferente y colocados en orden de diámetro decreciente sobre la primera varilla.

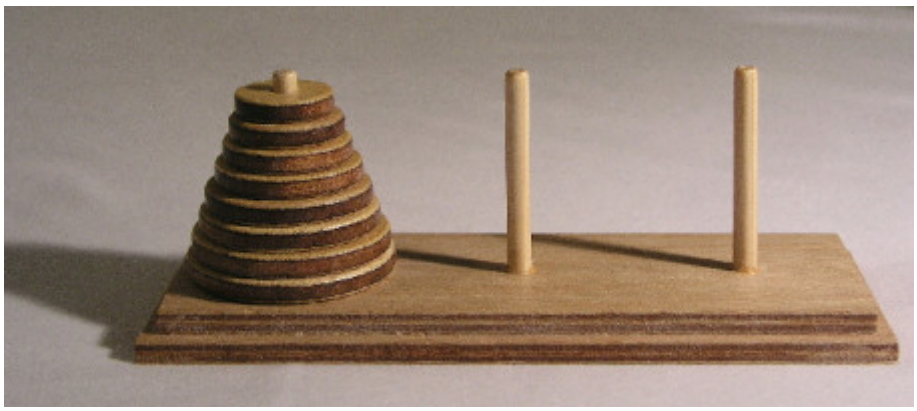
Según esa misma leyenda, Dios también creó un monasterio cuyos monjes tienen como misión el trasladar todos los discos desde la primera varilla a la terecera, con las siguientes reglas:

- Sólo pueden mover un disco cada vez.
- Un disco sólo puede quedar sobre la tierra (con ninguno debajo) o sobre otro que tenga mayor diámetro.

También cuenta la leyenda que el mundo acabará en el momento en el que los monjes terminen su misión.

Sin embargo, aunque la leyenda fuese cierta, no debemos tener ningún miedo.

¿Por qué?



En esta sección veremos algunos conceptos generales que nos pueden servir como base para comprender la complejidad a la que podemos llegar a enfrentarnos cuando analizamos y buscamos respuestas en datos.

Veremos aquí un esquema/resumen para que puedas tener una vista general:

- **Lógica algorítmica:**
  La base para comprender cómo se definen las soluciones a problemas.

- **Combinatoria y explosión combinacional:**
  Bases relacionadas con la matemática discreta para comprender lo grande que puede

llegar a ser el espacio de soluciones de un problema debido a la combinatoria.

- **Complejidad computacional:**
  El estudio de la complejidad de los problemas en función de los recursos en tiempo o memoria necesarios para resolverlos con el mejor algoritmo posible.

Para saber más

Puedes ver más información sobre el problema de las Torres de Hanói (y la complejidad de su resolución en función del número de discos de partida) en el siguiente enlace:

Torres de Hanói

1.1.- Lógica algorítmica.

Vamos a comenzar los conceptos básicos entendiendo qué es un algoritmo.

Un algoritmo es un conjunto de instrucciones que realizadas en orden permiten solucionar un problema.

Los algoritmos pueden recibir cero o más valores de entrada sobre los que trabajar, y pueden producir valores de salida o provocar algún tipo de efecto durante su ejecución.

Un algoritmo debe ser:

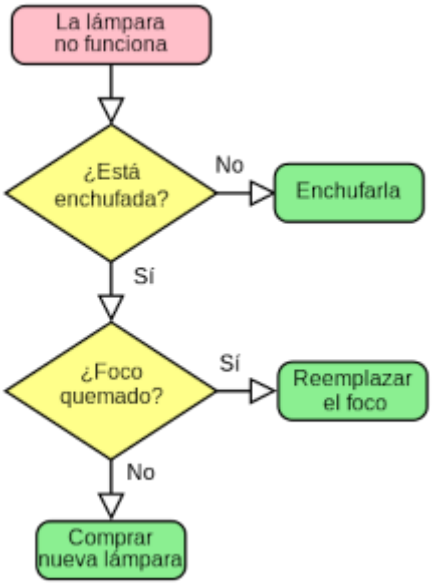
- **Preciso:** cada paso debe indicar qué hacer sin ambigüedades.
- **Finito:** con un número limitado de pasos.
- **Definido:** debe producir siempre el mismo resultado para los mismos valores de entrada.

Es posible definirlos con cierta independencia del lenguaje en el que finalmente se vayan a implementar, mediante diagramas de flujo o mediante pseudocódigo.

Mediante diagrama de flujo:

Podemos definir un algoritmo mediante un diagrama de flujo teniendo en cuenta que el diagrama debe tener un único nodo de inicio y al menos uno de final.

Diagrama de flujo



Debes conocer

En el siguiente enlace podrás ver información que debes conocer sobre los diagramas de flujo.

Diagrama de flujo

Mediante pseudocódigo:

Podemos definir un algoritmo mediante un pseudocódigo, que se escribe en un lenguaje a medio camino del lenguaje natural y del lenguaje de programación.

Pseudocódigo de un algoritmo para sumar los enteros del 1 al 100.

```
establecer resultado a 0
para i := 1 hasta 100 hacer
   establecer resultado a resultado + i
fin
```

Debes conocer

En el siguiente enlace podrás ver información que debes conocer sobre el pseudocódigo.

Pseudocódigo

Para saber más

En este enlace puedes ver más información sobre lo que es un algoritmo.

Algoritmo

Autoevaluación

¿Cuál de las siguientes afirmaciones es cierta en relación a los diagramas de flujo?

- ○ Pueden tener cualquier número de nodos de inicio.
- ○ Deben tener un único nodo de final.
- ○ Deben tener un único nodo de inicio y al menos uno de final.
- ○ No tienen nodo de inicio y deben tener uno o más nodos de final.

Incorrecto. Deben tener uno.

Incorrecto. Pueden tener uno o más nodos de final.

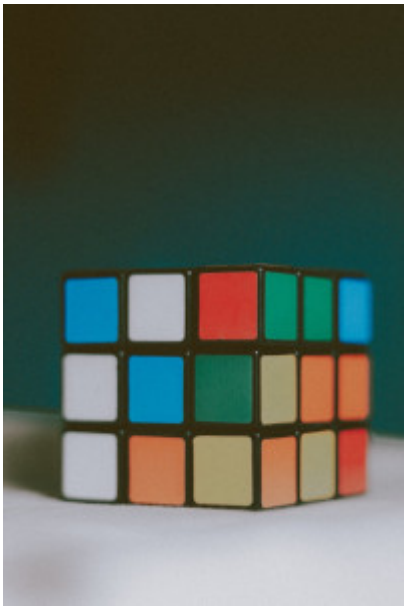
Correcto.

Incorrecto. Deben tener un nodo de inicio.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta
4. Incorrecto

1.2.- Combinatoria.



Mathias Reding (Dominio público)

Los algoritmos se utilizan para resolver problemas de todo tipo, para los cuales existe en ocasiones un amplísimo espacio de soluciones. Tal espacio de soluciones por lo general está formado por combinaciones de posibles acciones o valores.

Por esa razón, a la hora de comprender la complejidad de los problemas a los que se enfrentan los algoritmos es fundamental adquirir un conocimiento básico sobre combinatoria, y en particular saber que es posible calcular el número de combinaciones, variaciones o permutaciones posibles en base a fórmulas bien definidas.

Permutaciones:

Podemos querer disponer N elementos en un orden determinado. Eso es una permutación.

Si queremos saber de cuántos modos distintos podemos ordenar una lista, estaremos calculando entonces el número de permutaciones.

Para el ejemplo de una lista de 3 elementos, podemos escoger cualquiera de los 3 para el primer lugar, cualquiera de los 2 restantes para el segundo lugar, y finalmente ya sólo nos queda uno para el último lugar.

Por lo tanto podemos ordenarla de 3x2x1=6 formas distintas.

Variaciones:

Podemos querer disponer N elementos de entre un total de M en un orden determinado. Eso es una variación.

Si queremos saber de cuántos modos distintos podemos tomar N elementos de un conjunto de M y ordenar esos N, estaremos calculando entonces el número de variaciones.

Por ejemplo, si en una competición participan 4 personas y queremos saber de cuántas formas posibles podrían otorgarse las medallas de oro y de plata, podemos empezar calculando los posibles puestos de los cuatro como permutaciones (4x3x2x1=24). Después nos daríamos cuenta de que, si sólo nos interesan los 2 primeros puestos, por cada posible combinación de primero y segundo nos han salido dos permutaciones (una con un tercero y un cuarto y otra en el que ese cuarto está tercero y el tercero está cuarto). De modo que si dividimos entre 2 nos sale que el número de variaciones es de 24/2 = 12.

Combinaciones:

Las combinaciones son como las variaciones (tomamos N elementos de un conjunto de N) pero sin importar en qué orden.

Si queremos saber de cuántos modos distintos podemos tomar N elementos de un conjunto de M, estaremos calculando entonces el número de combinaciones.

Por ejemplo, si queremos saber de cuántas formas podemos elegir 2 países de entre 4 posibles, podemos tomar primero el país A y después el país B, o primero el país B y después el A, siendo el mismo resultado porque como hemos dicho no importa el orden. De modo que calcularíamos el número de variaciones (12, igual que en el ejemplo anterior), y después dividiríamos entre el número de formas de ordenar los 2 países que hemos sacado (2), obteniendo un resultado final de 12/2 = 6.

Como hemos dicho, existen unas fórmulas bien definidas para calcular números combinatorios (el número de variaciones, permutaciones o combinaciones según el tamaño del conjunto de partida y cuántos elementos se tomen, ya sea con o sin repetición).

Para saber más

Puedes encontrar más información sobre combinatoria y el cálculo de números combinatorios en el siguiente enlace:

Combinatoria

Si el concepto de combinatoria es importante a la hora de trabajar con algorismos, más aún lo es el de **explosión combinatoria**.

Decimos que para un determinado problema aparece una explosión combinatoria cuando el número de posibles soluciones crece muy rápido a medida que aumentamos determinados valores de configuración del propio problema.

Por ejemplo, si el problema consiste en comprobar qué permutaciones de N elementos cumplen determinada condición, podemos comprobar que el tamaño del espacio de soluciones es igual al factorial de N. Por lo tanto, rápidamente nos empezaremos a encontrar tamaños extremadamente grandes incluso para valores de N relativamente pequeños:

N	N!
1	1
2	2
3	6
4	24

5	120
6	720
7	5.049
8	40.320
9	362.880
10	3.628.800
100	$\sim 9{,}33262154 \times 10^{157}$

Para saber más

Puedes encontrar más información acerca del concepto de explosión combinatoria en el siguiente enlace:

Explosión combinatoria

Autoevaluación

Si queremos saber de cuántos modos distintos podemos tomar N elementos de un conjunto de M y ordenar esos N, estaremos calculando entonces el número de permutaciones.

○ Verdadero ○ Falso

Falso

Ese es el número de variaciones.

Si queremos saber de cuántos modos distintos podemos tomar N elementos de un conjunto de M, estaremos calculando entonces el número de variaciones.

○ Verdadero ○ Falso

Falso

Ese es el número de combinaciones.

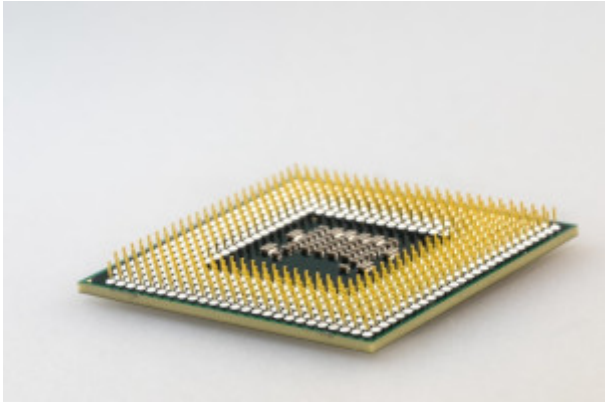
Si queremos saber de cuántos modos distintos podemos ordenar una lista, estaremos calculando entonces el número de permutaciones.

○ Verdadero ○ Falso

Verdadero

Ese es el número de permutaciones.

1.3.- Complejidad computacional.



Pixabay (CC0)

Como hemos visto, usamos algoritmos para resolver problemas, los cuales en ocasiones presentan una explosión combinacional.

Esa combinatoria produce una determinada complejidad computacional, que también es importante conocer y estudiar.

En primer lugar, hemos de tener en cuenta que cuando se estudia la complejidad computacional de un problema realmente existen dos tipos.

- **Complejidad en tiempo de ejecución:** cuánto tardaremos en resolverlo.
- **Complejidad en memoria:** cuánta memoria necesitaremos para resolverlo.

Sin embargo por lo general cuando hablamos de complejidad computacional si no se menciona nada relacionado con la memoria entonces podemos considerar que se trata de complejidad en tiempo de ejecución. Esto se debe a que son mucho más comunes los problemas cuya complejidad se dispara en tiempo de ejecución que aquellos que se disparan en memoria.

Enfocándonos ya únicamente en la complejidad computacional en tiempo, la mayor distinción que se realiza es si el problema puede resolverse en un tiempo polinómico (en función del tamaño de la entrada) o por el contrario es necesario emplear un tiempo exponencial.

También hay que tener en cuenta que los problemas se resuelven con algoritmos, para los cuales también podemos realizar un estudio sobre el tiempo y la memoria que emplean para resolver el problema para el cual se han diseñados, lo cual entra dentro del campo del análisis de algoritmos.

Por lo tanto, la complejidad de un problema viene indicada por el tiempo y/o memoria que necesite emplear el mejor algoritmo posible. En este sentido, la complejidad computacional se ocupa de estudiar la complejidad del problema como tal (no en base a algoritmos concretos).

Con ello, un problema será considerado *tratable* si se puede encontrar un algoritmo capaz de resolverlo en tiempo polinomial, e *intratable* si no existe tal algoritmo y es necesario emplear un tiempo exponencial.

Algoritmo de tiempo polinomial:

Es aquel que en el peor de los casos permite solucionar cierto problema en un tiempo determinado por un polinómio en función del tamaño de la entrada.

Por ejemplo, un algoritmo que para ordenar una lista de N elementos pueda llegar a necesitar $3N^3+N+8$ segundos en el peor caso, sería polinomial.

Evidentemente preferiríamos otro que tarde $7N^2$ segundos porque para valores grandes de N emplea menor tiempo.

A efectos de complejidad ambos se consideran polinomiales, pero el primero sería $O(N^3)$ -orden de N al cubo- y el segundo $O(N^2)$ -orden de N al cuadrado-.

Algoritmo de tiempo exponencial:

Es aquel que en el peor de los casos permite solucionar cierto problema en un tiempo determinado por una función exponencial con el tamaño de la entrada.

Por ejemplo, un algoritmo que para ordenar una lista de N elementos pueda llegar a necesitar 2^N segundos en el peor caso, sería exponencial.

Es sencillo comprobar que un algoritmo de tiempo polinomial siempre termina siendo más rápido que uno de tiempo exponencial si el tamaño de la entrada es lo suficientemente grande.

Valor de N	N^5	2^N
5	3.125	32
10	100.000	1.024
15	759.375	32.768
20	3.200.000	1.048.576
25	9.765.625	33.554.432
30	24.300.000	1.073.741.824
35	52.521.875	34.359.738.368
40	102.400.000	1.099.511.627.776

Para saber más

Puedes ver más información sobre complejidad computacional en el siguiente enlace:

Teoría de la complejidad computacional

Puedes ver más información sobre análisis de algoritmos en el siguiente enlace:

Análisis de algoritmos

Autoevaluación

¿Cuál de las siguientes afirmaciones es correcta respecto del la complejidad de algoritmos?

- ○ Un algoritmo de tiempo polinomial siempre emplea menos tiempo en resolver un problema que uno exponencial.
- ○ Un algoritmo de tiempo exponencial siempre emplea menos tiempo en resolver un problema que uno polinomial.
- ○ Un algoritmo de tiempo exponencial siempre termina siendo más rápido que uno de tiempo polinomial si el tamaño de la entrada es lo suficientemente grande.
- ○ Un algoritmo de tiempo polinomial siempre termina siendo más rápido que uno de tiempo exponencial si el tamaño de la entrada es lo suficientemente grande.

Incorrecto. Puede depender del tamaño de la entrada.

Incorrecto. Puede depender del tamaño de la entrada.

Incorrecto. Es de tiempo exponencial es peor que el polinomial si el tamaño de la entrada es muy grande.

Correcto.

Solución

1. Incorrecto
2. Incorrecto
3. Incorrecto
4. Opción correcta

2.- Analítica y búsqueda de respuestas.

Caso práctico

cdu445 (Dominio público)

Sandra y **Fernando** van de camino a una nueva reunión con los directivos de la empresa que los ha contratado como científicos de datos.

Los modelos que llevan varios días preparando han fracasado estrepitosamente en la fase de evaluación, por lo que evidentemente no serán desplegados.

A buen seguro hay algo en los datos que no están comprendiendo, y por eso necesitan entrevistarse con los directivos de nuevo, con la intención de conseguir una mejor comprensión sobre el negocio.

Pero esta vez es diferente a la primera, ya que durante los últimos meses han estado trabajando en la preparación de los datos, en la generación de modelos y en la evaluación de éstos. De modo que ahora tienen una serie de dudas que no tenían en la primera reunión y para las cuales les harán unas preguntas que serán clave para comenzar una nueva iteración en el proceso.

En esta sección tendremos un primer contacto con la analítica de datos, incluyendo sus distintos niveles, y estudiaremos las principales metodologías que se emplean en el campo de la Minería de Datos.

Veremos aquí un esquema/resumen para que puedas tener una vista general:

- **Niveles de Analítica de Datos:**
  - **Análisis Descriptivo:** ¿qué ha ocurrido?
  - **Análisis Diagnótico:** ¿por qué ha ocurrido?
  - **Análisis Predictivo:** ¿qué va a ocurrir?
  - **Análisis Prescriptivo:** ¿qué hacer para que algo ocurra?

- **Principales metodologías en Minería de Datos:**
  - **SEMMA:** Sample, Explore, Modify, Model, Assess.
  - **CRISP-DM:** Cross-industry standard process for data mining.

2.1.- Análisis de Datos y Analítica de Datos.

Timur Saglambilek (Dominio público)

Una vez que somos capaces de adquirir, integrar y gestionar grandes cantidades de datos, el siguiente paso es tratar de obtener algún tipo de valor de tal información.

Llegados a este punto, es importante diferenciar entre lo que es Analítica de Datos y el Análisis de Datos, ya que se trata de conceptos muy íntimamente relacionados pero no idénticos ni intercambiables.

Análisis de Datos:

El análisis de datos consiste en encontrar hechos, relaciones, patrones, revelaciones y/o tendencias, por lo general con la intención de poder apoyar en la mejor medida posible la toma de decisiones.

Por ejemplo, una compañía puede querer analizar las ventas de guantes de lana para determinar cómo están relacionadas con la temperatura ambiente que hace cada día y de ese modo tomar las mejores decisiones posibles a la hora de proveerse de guantes en función de la predicción del tiempo.

Analítica de Datos:

La analítica de datos es un concepto más amplio, ya que incluye dentro al propio análisis de datos así como al resto de actividades alrededor del ciclo de vida del dato (muchas de las cuales ya hemos visto).

Algunas de esas actividades, entre otras, son:

- Obtención/recolección de los datos desde diversas fuentes.
- Limpieza.
- Integración (desde las diversas fuentes a su representación unificada).
- Gobierno de Datos, mediante el cual se gestiona:
  - Disponibilidad (datos disponibles cuando van a usarse).
  - Usabilidad (datos válidos para lo que se quiere hacer con ellos).
  - Integridad (datos correctos).
  - Seguridad (de modo que no puedan ser accedidos ni desde fuera de la organización ni por quienes no tienen los apropiados permisos dentro de la propia organización).
- Análisis de Datos (ya mencionado).

Algunos usos comunes de la Analítica de Datos son los siguientes:

- En el mundo de los negocios, para disminuir los costes operacionales y facilitar la toma estratégica de decisiones.
- En el mundo científico, para entender por qué ocurre un determinado fenómeno y así mejorar el modo de abordarlo.
- En el mundo de los servicios, para disminuir costes y mejorar la calidad de los mismos.

Autoevaluación

La Analítica de Datos está contenida dentro del Análisis de Datos.

○ Verdadero ○ Falso

Falso

Es al revés.

La limpieza de datos es una de las actividades de la Analítica de Datos.

○ Verdadero ○ Falso

Verdadero

Es una de sus actividades.

El Gobierno de Datos es una de las actividades del Análisis de Datos.

○ Verdadero ○ Falso

Falso

Es una actividad de la Analítica de Datos.

2.2.- Niveles de Analítica de Datos.

Pixabay (CC0)

Dentro de la Analítica de Datos existen 4 categorías o niveles bien diferenciados:

Análisis Descriptivo:

¿Qué ha ocurrido?

El Análisis Descriptivo intenta describir lo que ha ocurrido.

Por lo general produce como resultado reportes o cuadros de mando estáticos obtenidos mediante consultas a almacenes de datos operacionales (tipo OLTP), como los CRM o ERP empresariales.

Intenta responder a preguntas del tipo de:

- ¿Cuál fue el beneficio mensual de la compañía durante los últimos 12 meses?
- ¿Qué tendencia tiene el número de llamadas de queja que estamos recibiendo en el soporte telefónico a clientes?
- ¿Qué línea de producto está produciendo los mejores resultados?

Análisis Diagnóstico:

¿Por qué ha ocurrido?

El Análisis Diagnóstico intenta determinar la causa de un fenómeno que ha ocurrido o está ocurriendo, detectando qué información resulta estar relacionada con el mismo.

Por lo general implica obtener información de diversas fuentes y almacenarla en estructuras específicas (tipo OLAP) que facilitan su análisis. Los resultados de tal análisis se muestran mediante herramientas interactivas de visualización que permiten a los usuarios identificar tendencias y patrones.

Intenta responder a preguntas del tipo de:

- ¿Por qué tenemos más quejas desde el Norte de España que desde el Sur?
- ¿Por qué estamos vendiendo menos neveras este año?
- ¿Por qué este mes hemos batido el récord de bajas de clientes?

Como puedes imaginar, el Análisis Diagnóstico tiene mayor complejidad que el Análisis Descriptivo, pero gracias a él podemos obtener un mayor valor de los datos.

Análisis Predictivo:

¿Qué ocurrirá?

El Análisis Predictivo intenta predecir qué ocurrirá en un futuro, gracias a la generación de modelos predictivos fruto de procesos de tipo Machine Learning, con la intención de ser capaces de identificar tanto riesgos como oportunidades.

Intenta responder a preguntas del tipo de:

- ¿Qué probabilidad hay de que un potencial cliente responda positivamente a esta oferta?
- ¿Qué probabilidad hay de que este tratamiento cure una determinada enfermedad en determinado rango de edades?
- ¿Si el cliente se ha interesado en este producto, con qué probabilidad le interesará este otro?

Como puedes imaginar, el Análisis Predictivo tiene mayor complejidad que el Análisis Diagnóstico, pero gracias a él podemos obtener un mayor valor de los datos.

Análisis Prescriptivo:

¿Qué hacer para que ocurra?

El Análisis Prescriptivo se apoya en los resultados que es capaz de producir el Análisis Predictivo, probando automáticamente diversas posibles acciones alternativas para así ser capaz de prescribir la mejor acción a tomar. No sólo se enfoca en la acción prescrita sino que también trata de dar información al usuario acerca de la razón por la que es la mejor.

Intenta responder a preguntas del tipo de:

- ¿Cuál de estos productos funcionará mejor si lo vendemos en Canadá?
- ¿Cuál es el mejor mes para lanzar nuestro nuevo servicio?

Como puedes imaginar, el Análisis Prescriptivo tiene mayor complejidad que el Análisis Predictivo, pero gracias a él podemos obtener un mayor valor de los datos.

2.3.- Principales métodologías en Minería de Datos.

En esta sección conoceremos las metodologías más empleadas en el campo de la Minería de Datos.

Como ya sabrás por los contenidos de anteriores unidades de trabajo, la Minería de Datos es una rama de la Inteligencia Artificial que busca obtener valor del dato fundamentalmente mediante la gereración de modelos predictivos. Por lo tanto se emplea para realizar Análisis Predictivo, llegando sus resultados a emplearse también posteriormente si realizamos Análisis Prescriptivo.

Tom Fisk (Dominio público)

Veremos aquí un esquema/resumen para que puedas tener una vista general:

- **SEMMA** (Sample, Explore, Modify, Model, Assess):
  Una lista de pasos secuenciales desarrollada por SAS Institute que se emplean en muchas ocasiones a modo de metodología para la minería de datos.

- **CRISP-DM** (Cross-industry standard process for data mining):
  Una metodología ampliamente empleada para el proceso de minería de datos que goza de una cierta oficialidad.

2.3.1.- SEMMA.

SEMMA es el acrónimo empleado para una lista de de pasos secuenciales desarrollada por *SAS Institute*, uno de los principales fabricantes de software para inteligencia empresarial.

SAS Institute (Dominio público)

Los pasos de *SEMMA* y sus tareas relacionadas son:

- **(S)ample:**
  Seleccionar los datos para el modelado mediante muestreo (*data sampling*). El conjunto de datos debe quedar con tamaño como para contener suficiente información, pero tan pequeño como para poder usarse de modo eficiente.

- **(E)xplore:**
  Ayudarse de la visualización de datos para entender los datos y encontrar relaciones entre las variables así como anormalidades.

- **(M)odify:**
  Seleccionar, crear y transformar variables como preparación para el modelado de datos.

- **(M)odel:**
  Aplicación de diversas técnicas de minería de datos para producir modelos.

- **(A)ssess:**
  Evaluación de los modelos para comprobar su utilidad práctica.

Los 5 pasos de *SEMMA* se emplean en muchas ocasiones a modo de metodología para la minería de datos. Sin embargo, voces críticas indican que para la fase *Sample* es necesario un conocimiento profundo no sólo de los datos sino de los aspectos de negocio aplicables, lo cual no aparece contemplado como una de las tareas dentro esa fase.

En todo caso, sus creadores en *SAS Instutute* avisan de que realmente no idearon los pasos como una metodología para minería de datos, sino que simplemente constituyen la organización lógica del conjunto de herramientas funcionales que emplea uno de sus productos para minería de datos, llamado *SAS Enterprise Miner*.

Para saber más

Puedes ver más información sobre *SEMMA* en el siguiente enlace:

SEMMA

2.3.2.- CRISP-DM.

CRISP-DM (o *Cross-industry standard process for data mining*) es una metodología ampliamente empleada para el proceso de minería de datos que goza de una cierta oficialidad al provenir de un proyecto de la Unión Europea bajo la iniciativa ESPRIT.

El *CRISP-DM* divide el proceso de minería de datos en 6 fases:

- Comprensión del negocio.
- Comprensión de los datos.
- Preparación de los datos.
- Modelado.
- Evaluación.
- Despliegue.

La clave de la metodología *CRISP-DM* es que no se trata de una secuencia sino de un proceso cíclico, ya que propone continuar dando vueltas por las fases (aplicando las lecciones aprendidas durante cada iteración en el ciclo para la siguiente).

Además de su naturaleza cíclica existen diversas transiciones posibles entre las fases que contempla, como se puede ver en el diagrama que resume la metodología.

Diagrama del proceso CRISP-DM

Para saber más

Puedes ver más información sobre *CRISP-DM* en el siguiente enlace:

Cross Industry Standard Process for Data Mining

3.- R para analizar datos.

Caso práctico

Hadley Wickham and others at RStudio (CC BY-SA)

Los hermanos López (**Mateo**, **Lucía** y **Elena**) están inmersos en el proceso de crear su nueva *start-up*, una empresa dedicada al análisis de datos provenientes de tests ópticos.

Lucía ha trabajado durante gran parte de su vida laboral con **MatLab**, un conocido software capaz de realizar todo tipo de operaciones sobre conjuntos de datos en varias dimensiones, así como de mostrar visualizaciones sobre los mismos, entre otras muchas funcionalidades.

MatLab es ideal pero tiene el problema de ser propietario y los 3 hermanos consideran que adquirir licencias afectaría a su exiguo presupuesto, teniendo en cuenta que también tienen que alquilar una oficina, comprar equipos e invertir en promoción.

Afortunadamente su amigo Francisco les acaba de recomendar que utilicen **R**, un lenguaje con el que van a poder hacer lo que necesitan y que al ser de código abierto pueden utlizar sin necesidad de pagar por licencias.

Al menos a buen seguro les valdrá para lanzar el negocio, y más adelante ya verán si les merece la pena comprar licencias de MatLab o pueden ahorrárselo.

R es un lenguaje de programación interpretado de código abierto creado específicamente para facilitar el análisis de datos. Su nombre proviene de que está basado en un lenguaje preexistente llamado S.

Nota: El software gracias al cual podemos ejecutar programas escritos en tal lenguaje también se llama R.

Algunas de sus peculiaridades son las siguientes:

- Es **interpretado**: no es necesario compilar los programas sino que son interpretados (por otro programa llamado intérprete).
- Es de **alto nivel**: no se tiene acceso a bajo nivel a la máquina en la que se ejecuta (gestión de la memoria, etc), sino que todo está encapsulado para que el programador pueda concentrarse en el análisis de datos.

- Es un lenguaje **imperativo** (con ejecución de código línea a línea) pero soporta programación orientada a objetos (para analizar datos usaremos clases constantemente) y en cierto modo también permite programación funcional (podemos crear funciones, llamarlas de forma recursiva y usarlas como si fuesen variables).

R puede obtenerse en su web oficial (https://www.r-project.org/) de modo que podemos trabajar con R de forma totalmente interactiva mediante su intérprete o podemos ejecutar programas que hayamos escrito con cualquier editor de texto plano.

Además, podemos instalar uno de los muchos IDE que están disponibles, siendo RStudio uno de los más utilizados. RStudio puede descargarse desde su web oficial https://www.rstudio.com/.

Si el alumno no desea o no puede instalar R en su ordenador, puede emplear uno de los varios intérpretes de R disponibles online. Te recomendamos los 2 siguientes, ya que permiten crear gráficos para visualización.

- myCompiler - Online R Compiler
- rdrr.io

Para saber más

Puedes ver más información sobre el lenguaje de programación R en la propia página oficial de R:

Página oficial de R

También puedes consultar un resumen sobre R en el siguiente enlace:

R (lenguaje de programación)

3.1.- Primeros pasos.

En este curso no vamos a pretender que el alumno se convierta en un experto en R, sino que conozca algunas bases que le permitan entender qué características tiene un lenguaje especialmente creado para análisis de datos.

Por ello, vamos a limitarnos a dar unos primeros pasos muy guiados que puedes ir ejecutando en tu intérprete de R (ya sea uno instalado o un intérprete en línea entre los que te hemos recomendado).

Puedes probar a hacer ejecutar código con tus propias modificaciones para entender mejor qué es lo que está ocurriendo y cómo funciona el lenguaje.

Hola mundo:

```
sum(1:3)
```

```
[1] 6
```

Para entender este ejemplo en primer lugar debemos saber que R puede trabajar tanto con valores individuales como con **vectores** (que son conjuntos ordenados de valores).

El operador ":" crea una secuencia de números (en este caso entre 1 y 3), con lo cual tenemos un vector.

La función *sum* suma los elementos del vector que recibe como parámetro, y por ello el resultado es 6.

Obtener ayuda:

R integra un mecanismo para solicitar ayuda acerca de funciones y construcciones del lenguaje.

```
?sum          #abre la página de ayuda de la función sum
?"+"          #abre la página de ayuda de la operación de suma
?"if"         #abre la página de ayuda acerca de la construcción del lenguaje "if"
??plotting    #busca en la ayuda temas que contengan la palabra "plotting"
```

Asignación de variables:

```
x <- 3
y = 4
x+y
```

```
[1] 7
```

Podemos asignar valores a variables variables usando = o <-, aunque por razones históricas se prefiere <-.

Mostrar resultados:

```
x <- 3 + 4
```

Cuando hacemos una asignación por defecto no vemos el valor que recibe al la variable.

```
(x <- 3 + 4)

[1] 7
```

Sin embargo, podemos englobar en paréntesis para solicitarle a R que nos muestre el valor asignado.

Operaciones con vectores:

```
1:3 + 4:6

[1] 5 7 9
```

Aplicamos la operación de suma sobre dos vectores, y el resultado es otro vector cuyas posiciones se obtienen de sumar los valores de los vectores de entrada en esa misma posición.

```
c(1,3,5)

[1] 1 3 5
```

La función especial *c* crea un vector con los parámetros recibidos.

```
c(2,3,4) -1

[1] 1 2 3
```

Si realizamos una operación con un vector y un número, el resultado es el vector resultante de aplicar la operación en cada una de las posiciones.

```
c(2, 5, 4-2, 7) == 3
```

```
[1] FALSE FALSE FALSE FALSE
```

Usamos el operador == para comprobar igualdad (< para "menor que", > para "mayor que", <= para "menor o igual que", >= para "mayor o igual que").

La operación se realiza sobre todos los elementos del vector.

```
x <- c(1,2,3)
y = c(5,6,7)
x*y
```

```
[1]  5 12 21
```

Por supuesto, también podemos asignar vectores a variables.

```
length(1:8)
```

```
[1] 8
```

Podemos obtener la longitud de un vector.

```
a <- c("blanco","negro","azul")
length(a)
nchar(a)
```

```
[1] 3
[1] 6 5 4
```

Si el vector contiene cadenas de caracteres, su longitud es el número de cadenas. Si queremos conocer la longitud de cada cadena usaremos la función *nchar*.

```
a <- c(manzana=4,naranja=3,mandarina=5)
a
```

```
  manzana   naranja mandarina
        4         3         5
```

```
x <- c(1,2,3)
names(x) <- c("uno","dos","tres")
x
```

```
 uno  dos tres
   1    2    3
```

Podemos asignar nombres a los elementos de un vector.

```
x <- 2:7
x
x[c(1,3,5)]
```

```
[1] 2 3 4 5 6 7
[1] 2 4 6
```

Podemos acceder a un vector mediante sus posiciones (índices).

Ten en cuenta que R el índice del primer elemento es el 1 (en contraposición a muchos lenguajes en los que es el 0).

```
x <- 2:7
x
x[c(-1,-3,-5)]
```

```
[1] 2 3 4 5 6 7
[1] 3 5 7
```

Podemos acceder a un vector excluyendo ciertas posiciones.

```
x <- 1:6
x[c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE)]
```

```
[1] 1 3 4 6
```

Podemos acceder a un vector mediante un vector de valores booleanos que nos indica con TRUE en las posiciones que queremos y FALSE en las que no.

Números especiales:

```
c(-Inf+1,Inf-1,Inf-Inf,NA*5)
```

```
[1] -Inf  Inf  NaN   NA
```

R soporta los siguientes números especiales:

- Inf: infinito.
- -Inf: menos infinito.
- NaN: *not-a-number*, indicando que una operación no tiene sentido o no puedo ejecutarse correctamente.
- NA: *not available*, representando un valor desconocido (lo cual es muy común en análisis de datos).

Autoevaluación

¿Podemos en R operar un vector con un número?

○ Verdadero ○ Falso

Verdadero

Sí, y se aplicará la operación con el número a todos los elementos del vector.

¿Usaremos el operador <= para asignar valores a una variable?

○ Verdadero ○ Falso

Falso

El operador significa "menor o igual que". Para asignar usaremos <- o =.

¿Podemos asignar nombres a los elementos de un vector?

○ Verdadero ○ Falso

Verdadero

Cierto.

3.2.- Arrays y Matrices.

Hasta ahora hemos visto que R trabaja sin problemas con vectores, que son objetos unidimensionales.

Sin embargo, también podemos trabajar con conjuntos de datos multidimensionales mediante **arrays**. Hay que tener en cuenta que los arrays son rectangulares (es decir que todas las filas deben tener la misma longitud, al igual que todas las columnas y todas las demás posibles dimensiones).

```
(array_3d <- array(
  1:24,
  dim = c(2, 3, 4),
  dimnames = list(
    c("izquierda", "derecha"),
    c("primero", "segundo", "tercero"),
    c("uno", "dos", "tres", "cuatro")
  )
))
```

```
, , uno

          primero segundo tercero
izquierda       1       3       5
derecha         2       4       6

, , dos

          primero segundo tercero
izquierda       7       9      11
derecha         8      10      12

, , tres

          primero segundo tercero
izquierda      13      15      17
derecha        14      16      18

, , cuatro

          primero segundo tercero
izquierda      19      21      23
derecha        20      22      24
```

Podemos crear un array de cualquier número de dimensiones mediante la función *array*, indicándole un vector con todos los valores, otro con la longitud en cada dimensión (lo cual implica cuántas dimensiones queremos) y una lista de vectores con los nombres para las distintas dimensiones.

```
dim(array_3d)
length(dim(array_3d))
```

```
[1] 2 3 4
[1] 3
```

Si tenemos un array podemos saber cuántas dimensiones tiene y la longitud de cada una con la función *dim*.

```
array_3d[,1:2,c(2,3)]
```

```
, , dos

          primero segundo
izquierda       7       9
derecha         8      10

, , tres

          primero segundo
izquierda      13      15
derecha        14      16
```

Podemos acceder al array utilizando corchetes "[]" de un modo similar a como lo hacemos con los vectores, pero usando comas para separar las distintas dimensiones (si en una posición no ponemos nada -vacío- obtendremos su contenido completo).

Las **matrices** son un caso especial de arrays limitado a 2 dimensiones.

```
(matriz <- matrix(
  1:6,
  nrow = 3, #ncol = 2 hace lo mismo
  dimnames = list(
    c("uno", "dos", "tres"),
    c("arriba", "abajo")
  )
))
```

```
     arriba abajo
uno       1     4
dos       2     5
tres      3     6
```

Podemos crear una matriz con la función *matrix*, pasándole como argumentos un vector con los números, el número de filas (o de columnas) y una lista de vectores con los nombres para las distintas dimensiones.

```
matriz[,2]
```

```
 uno  dos tres
   4    5    6
```

El acceso a las posiciones de la matriz es igual que en el caso de los arrays, teniendo en cuenta que sólo tienen 2 dimensiones.

```
matriz2 <- matrix(
  3:8,
  nrow = 3, #ncol = 2 hace lo mismo
  dimnames = list(
    c("uno", "dos", "tres"),
    c("arriba", "abajo")
  )
)

matriz + matriz2
```

```
     arriba abajo
uno       4    10
dos       6    12
tres      8    14
```

Podemos realizar operaciones de álgebra de matrices.

Autoevaluación

¿Cuál de las siguientes afirmaciones es correcta respecto de arrays y matrices en R?

- ○ Son de dimensiones equivalentes.
- ○ Las matrices son bidimensioneles mientras que los arrays pueden tener hasta 3 dimensiones.
- ○ Las matrices son bidimensioneles mientras que los arrays pueden tener cualquier número de dimensiones.
- ○ Los arrays son bidimensioneles mientras que las matrices pueden tener cualquier número de dimensiones.

Incorrecto. Las matrices son un caso concreto de arrays con una limitación en cuanto a dimensiones.

Incorrecto. Los arrays pueden tener más de 3 dimensiones.

Correcto.

Incorrecto. Ni lo uno ni lo otro.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta
4. Incorrecto

3.3.- Listas y Dataframes.

Vectores, arrays y matrices tienen el inconveniente de que sólo permiten contener elementos de un único típo (por ejemplo todos numéricos o todos cadenas de caracteres).

Una **lista** es algo equivalente a un vector en el que cada elemento puede ser de un tipo distinto.

```
(lista <- list(
  c(1, 2, 3),
  33,
  matrix(c(6, 7, 14, 1), nrow = 2),
  mean
))
```

```
[[1]]
[1] 1 2 3

[[2]]
[1] 33

[[3]]
     [,1] [,2]
[1,]    6   14
[2,]    7    1

[[4]]
function (x, ...)
UseMethod("mean")
```

Podemos crear una lista mediante la función *list*, pasándole como argumentos los distintos valores que queramos incluír en ella.

En este ejemplo hemos creado la lista con un vector, un valor numérico, una matriz y una función (recordemos que R permite usar funciones como si fuesen variables).

```
names(lista) <- c("vector", "numero", "matriz", "funcion mean")

lista
```

```
$vector
[1] 1 2 3

$numero
[1] 33

$matriz
     [,1] [,2]
[1,]    6   14
[2,]    7    1
```

```
$`funcion mean`
function (x, ...)
UseMethod("mean")
```

De igual modo que con los vectores, podemos asignar nombres a sus elementos.

```
lista[2:3]

$numero
[1] 33

$matriz
     [,1] [,2]
[1,]    6   14
[2,]    7    1
```

Podemos acceder a la lista igual que hacíamos con los vectores.

Los ***dataframes*** son el equivalente a matrices (por lo tanto bidimensionales) que pueden almacenar distintos tipos de datos, siendo por lo tanto similares al contenido de una hoja de cálculo.

Por ello están especialmente indicados para trabajar con conjuntos de datos en su formato más común: secuencias de filas, las cuales están formadas por columnas que pueden contener distintos tipos de datos.

```
v <- 1:5
(mi_dataframe <- data.frame(
x = letters[1:5],
y = v,
z = v > 2
))

  x y     z
1 a 1 FALSE
2 b 2 FALSE
3 c 3  TRUE
4 d 4  TRUE
5 e 5  TRUE
```

Podemos crear dataframes mediante la función *data.frame*, pasándole los datos por columnas (todas de la misma longitud). Como podemos ver, cada columna puede contener distinto tipo de datos (pero toda la columna el mismo).

```
mi_dataframe[2:3,c(1,2)]

  x y
```

```
2 b 2
3 c 3
```

Podemos acceder a los dataframes de modo análogo a como lo hacemos con matrices.

```
mi_dataframe2 <- data.frame(
col4 = 6:10,
col5 = v+3,
col6 = v < 3
)

cbind(mi_dataframe, mi_dataframe2)
```

```
  x y     z col4 col5  col6
1 a 1 FALSE    6    4  TRUE
2 b 2 FALSE    7    5  TRUE
3 c 3  TRUE    8    6 FALSE
4 d 4  TRUE    9    7 FALSE
5 e 5  TRUE   10    8 FALSE
```

Podemos realizar distintas operaciones con dataframes, como por ejemplo enlazarlos por columnas con *cbind*.

```
mi_dataframe3 <- data.frame(
x = letters[6:8],
y = 6:8,
z = c(TRUE,FALSE,TRUE)
)


rbind(mi_dataframe, mi_dataframe3)
```

```
  x y     z
1 a 1 FALSE
2 b 2 FALSE
3 c 3  TRUE
4 d 4  TRUE
5 e 5  TRUE
6 f 6  TRUE
7 g 7 FALSE
8 h 8  TRUE
```

De modo similar, también podemos enlazar dos dataframes por filas con *rbind*.

Autoevaluación

¿Cuál de las siguientes afirmaciones es correcta en relación a listas y dataframes en R?

- ○ Las listas son unidimensionales y los dataframes bidimensionales.
- ○ Los dataframes son bidimensionales, con celdas de cualquier tipo sin restricción.
- ○ Las listas son unidimensionales y los dataframes pueden tener cualquier dimensión.
- ○ Los dataframes tienen dos dimensiones más que las listas.

Correcto.

Incorrecto. En un dataframe todos los valores de cada columna deben ser del mismo tipo.

Incorrecto. Los dataframes son bidimensionales.

Incorrecto. Tienen una más.

Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto
4. Incorrecto

3.4.- Análisis y Visualización.

Respecto del análisis de datos, en primer lugar hemos de tener en cuenta que R tiene una gran cantidad de conjuntos de datos dentro de paquetes a los cuales podemos acceder si están instalados.

Para acceder a ellos, podemos emplear la función *data* pasándole como argumentos el nombre del dataset y el paquete en el que éste se encuentra.

```
data("kidney", package = "survival")
head(kidney)
```

```
  id time status age sex disease frail
1  1    8      1  28   1   Other   2.3
2  1   16      1  28   1   Other   2.3
3  2   23      1  48   2      GN   1.9
4  2   13      0  48   2      GN   1.9
5  3   22      1  32   1   Other   1.2
6  3   28      1  32   1   Other   1.2
```

En este caso hemos cargado el dataset llamado *kidney* dentro del paquete *survival* y hemos mostrado las primeras filas mediante la función *head*.

```
dim(kidney)
```

```
[1] 76  7
```

Vemos que el dataset contiene 76 filas y 7 columnas.

```
time <- kidney$time
mean(time)
median(time)
var(time)
sd(time)
```

```
[1] 101.6316
[1] 39.5
[1] 17138.5
[1] 130.9141
```

R nos permite calcular diversas estadísticas utilizando funciones específicas para ello.

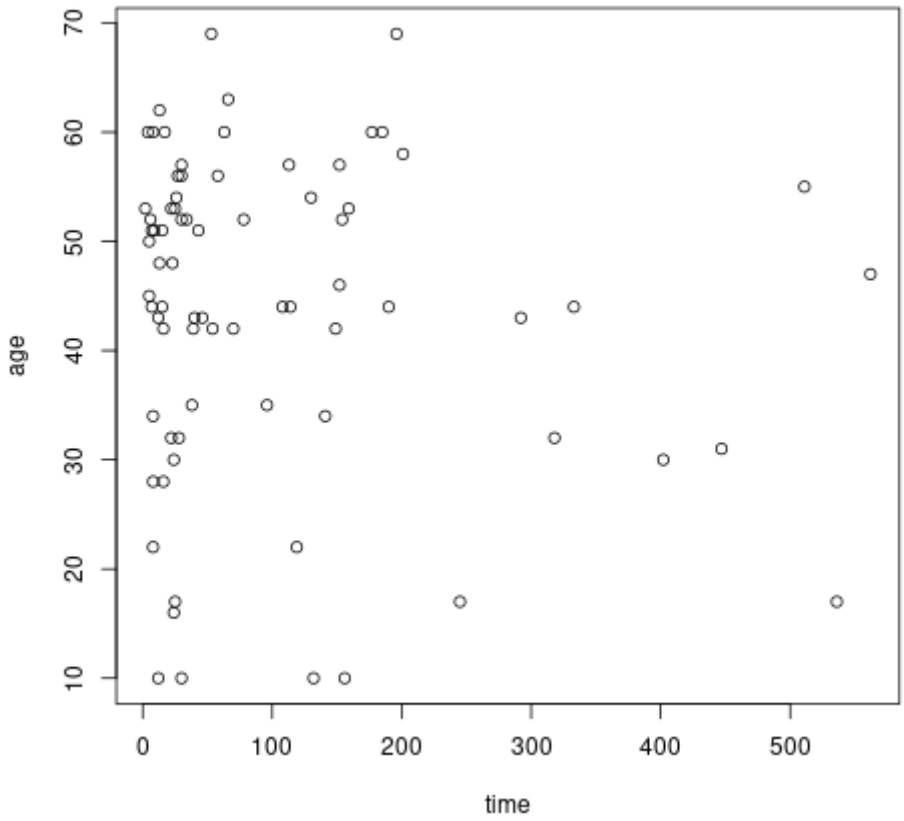
Fíjate que para este ejemplo hemos obtenido una columna del dataframe utilizando el accesor $ tras su nombre seguido del nombre de la columna.

```
summary(kidney)
```

```
       id              time           status            age              sex       
 Min.   : 1.0   Min.   :  2.0   Min.   :0.0000   Min.   :10.0   Min.   :1.000  
 1st Qu.:10.0   1st Qu.: 16.0   1st Qu.:1.0000   1st Qu.:34.0   1st Qu.:1.000  
 Median :19.5   Median : 39.5   Median :1.0000   Median :45.5   Median :2.000  
 Mean   :19.5   Mean   :101.6   Mean   :0.7632   Mean   :43.7   Mean   :1.737  
 3rd Qu.:29.0   3rd Qu.:149.8   3rd Qu.:1.0000   3rd Qu.:54.0   3rd Qu.:2.000  
 Max.   :38.0   Max.   :562.0   Max.   :1.0000   Max.   :69.0   Max.   :2.000  
  disease       frail      
 Other:26   Min.   :0.200  
 GN   :18   1st Qu.:0.600  
 AN   :24   Median :1.100  
 PKD  : 8   Mean   :1.184  
            3rd Qu.:1.500  
            Max.   :3.000  
```

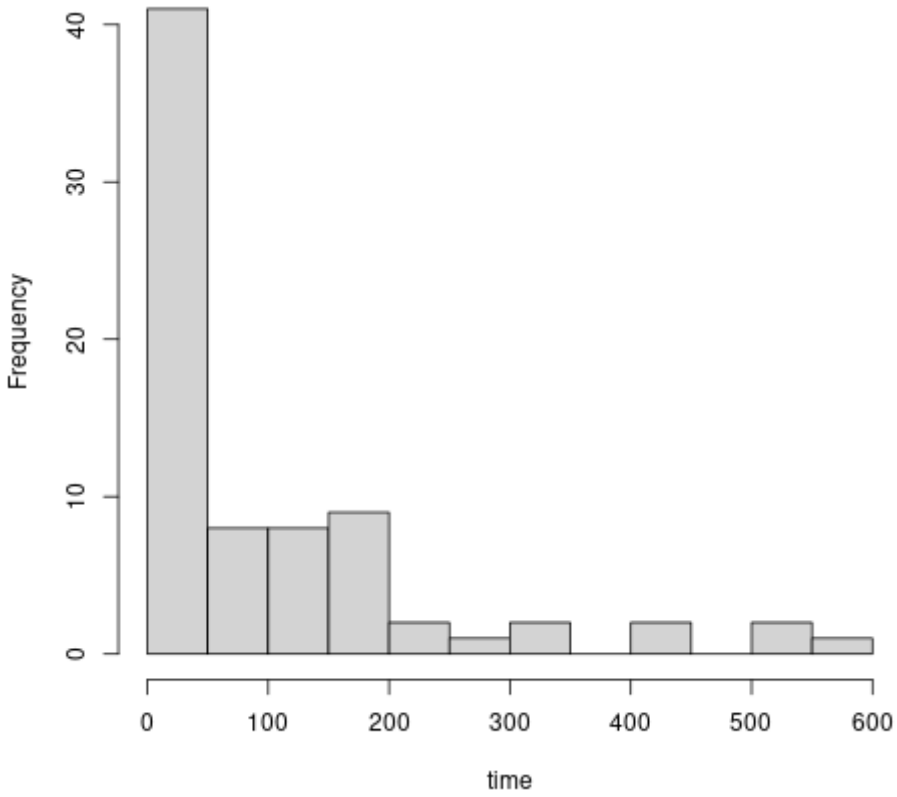
Incluso podemos utlizar la función *summary* para obtener una serie de estadísticas interesantes acerca de todas las columnas del dataframe.

```
with(kidney, plot(time, age))
```



Víctor Tomico (Dominio público)

```
with(kidney, hist(time))
```



Víctor Tomico (Dominio público)

Como podemos ver, R nos permite visualizar datos de un modo muy sencillo. Aquí mostramos gráficos de tipo *plot* (diagrama de puntos) e *hist* (histograma), pero existen muchos otros.

También hay que tener en cuenta que aunque aquí hemos usado el sistema básico de gráficos de R, hay otras dos librerías más avanzadas, llamadas *lattice* y *ggplot2*, las cuales permiten crear visualizaciones más avanzadas, detalladas y coloridas.

Para saber más

Para saber más acerca de análisis exploratorio con R, te recomendamos que visites el siguiente *notebook* en Kaggle (una web muy interesante para científicos de datos).

Exploratory Data Analysis - R

4.- Python para analizar datos.

Caso práctico

www.python.org (GNU/GPL)

En **Pérez Consultores** llevan más de 20 años ofreciendo servicios de consultoría tecnológica de amplio espectro, y en estos momentos están a punto de añadir una rama de Big Data a su oferta de servicios.

Para ello, han decidido especializar a algunos de sus empleados en analítica de datos en ambientes de Big Data, así como generar contenido dentro de la web corporativa con ejemplos de programación que les puedan servir para recibir visitas de posibles clientes y a su vez demostrar que saben hacer bien ese tipo de trabajo.

Para ello deben escoger cuál será el lenguaje de programación que emplearán tanto como base de la formación de su plantilla como para los ejemplos de programación en la web.

Debe ser un lenguaje sencillo de utilizar, de uso muy extendido, y con librerías que le permitan realizar analítica de datos integrada dentro de sistemas Big Data.

La elección es importante, porque no quieren arriesgarse a perder potenciales clientes si no escogen un lenguaje de programación adecuado.

Sin embargo, esa elección tan importante realmente es muy sencilla. Tienen que usar **Python**.

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general que se ha desarrollado de un modo considerable desde sus inicios gracias a su facilidad de uso y al gran número de librerías que han ido apareciendo para todo tipo de fines.

Comparte las siguientes características con el lenguaje R que ya hemos estudiado:

- El ser interpretado.
- El ser de alto nivel.
- El ser imperativo pero además soportar programación orientada a objetos y permitir programación funcional.

Sin embargo, no ha sido específicamente creado para el análisis de datos como ocurre con R, sino que adquiere tal capacidad fundamentalmente gracias a 3 librerías:

- **NumPy:**
  Añade soporte para arrays (multidimensionales) y matrices (bidimensionales), junto a una gran cantidad de operaciones optimizadas para trabajar sobre esas estructuras de datos.

- **Pandas:**
  Escrito sobre NumPy, ofrece estructuras específicas y así como gran cantidad de funcionalidades específicas para análisis de datos sobre esas mismas estructuras de datos.

- **Matplotlib:**
  Aporta capacidades de visualización de datos.

Python puede obtenerse en su web oficial (https://www.python.org/), de modo que podemos trabajar Python de forma totalmente interactiva mediante su intérprete IPython, o podemos ejecutar programas que hayamos escrito con cualquier editor de texto plano.

También puede ser interesante explorar la posibilidad de usar *notebooks* de IPython, los cuales se emplean mediante interfaz web.

Si el alumno no desea o no puede instalar Python en su ordenador, puede emplear uno de los varios intérpretes disponibles online. Te recomendamos el siguiente, ya que permite crear gráficos para visualización.

- Intérprete de Python en trinket.io.

Para saber más

Puedes descargar Python y acceder a toda la documentación en su web oficial:

Web oficial de Python

Puedes ver más información sobre Python en el siguiente enlace:

Python en Wikipedia

Puedes ver más información sobre IPython en el siguiente enlace:

IPython en Wikipedia

4.1.- Primeros pasos.

En este curso no vamos a pretender que el alumno se convierta en un experto en Python, sino que conozca algunas bases que le permitan entender cómo podemos emplear Python junto con sus librerías NumPy, Pandas y Matplotlib para realizar análisis de datos.

Por ello, vamos a limitarnos a dar unos primeros pasos muy guiados que puedes ir ejecutando en tu intérprete de Python (ya sea uno instalado o el intérprete en línea que te hemos recomendado).

Puedes probar a hacer ejecutar código con tus propias modificaciones para entender mejor qué es lo que está ocurriendo y cómo funciona el lenguaje.

Hola mundo:

```
print("¡Hola Mundo!")
```

```
¡Hola Mundo!
```

Para nuestro primer programa hemos llamado a la función *print*, pasándole la cadena "¡Hola Mundo!" como parámetro.

Obtener ayuda:

```
help()
```

```
Welcome to Python 3.9's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.9/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
```

Para obtener ayuda llamamos a la función *help.*

Mostrar valores:

```
print(3)
```

```
3
```

Para mostrar valores usamos la función *print*, como ya hicimos en nuestro primer programa.

Asignación de variables:

```
x = 3
y = "hola"
z = 55
print(y)
```

```
hola
```

Podemos asignar valores a variables usando =.

Como podemos ver, IPython no nos escribe por pantalla resultados referentes a las asignaciones, sino que tenemos que llamar explícitamente a la función *print*.

Es importante apreciar que al ser Python un lenguaje de tipado dinámico, no es necesario declarar el tipo de las variables antes de asignarles valores, sino que adquieren el tipo de forma dinámica según el valor que reciban.

Comentarios en el código:

```
x = 3
#Esto es un comentario x=798
y = 4
print(x+y)
```

```
7
```

Podemos incluir comentarios dentro del código mediante el caracter especial #.

Aritmética:

```
x = 3
y = 6
z = x + y
print(z)
print(x-y)
print(x*y)
print(x/y)
print(y**x)   #Exponent
```

```
9
-3
18
0.5
216
```

Podemos utilizar el intérprete de Python a modo de calculadora, realizando diversas operaciones aritméticas.

Listas:

```
lista = ["uno",2, "tres", 2+2, 5, 6, 7]
print(lista)
print(lista[2])   # Acceso a posición
print(lista[-2])  # Acceso a posición desde el final (-1 es el último)
print(lista[2:4]) # Acceso a sublista

['uno', 2, 'tres', 4, 5, 6, 7]
tres
6
['tres', 4]
```

La lista es un tipo de datos integrado en Python que permite una secuencia ordenada de valores, los cuales pueden ser de distintos tipos.

```
sublista1 = [1,2,3]
sublista2 = ["uno","dos"]
lista = [33, sublista1, ["azul", [77,88,99] , "verde"], sublista2 ]
print(lista)

[33, [1, 2, 3], ['azul', [77, 88, 99], 'verde'], ['uno', 'dos']]
```

Como podemos ver, aunque las listas son secuencias unidimensionales de datos, el hecho de permitir que tales datos puedan ser de cualquier tipo (también listas) nos permite crear estructuras de distinto número de dimensiones y niveles de anidamiento.

Autoevaluación

¿Para escribir valores por pantalla con Python usaremos la función *eco*?

○ Verdadero ○ Falso

Falso

Usaremos la función *print*.

¿Podemos añadir comentarios usando el caracter especial $?

○ Verdadero ○ Falso

Falso

Usaremos el caracter especial #.

¿Las listas en Python pueden contener dentro otras listas hasta cualquier nivel de anidación?

○ Verdadero ○ Falso

Verdadero

Correcto.

4.2.- Arrays con NumPy.

Como ya hemos visto al final del apartado anterior, Python tiene integrado un tipo de datos que le permite manejar listas, las cuales son secuencias unidimensionales de valores de cualquier tipo.

Por supuesto, si creamos una lista cuyos elementos a su vez son listas de la misma longitud, podemos conseguir una estructura en la que almacenar un conjunto de datos sobre el cual realizar algún tipo de análisis. E incluso podemos ir más allá si aumentamos el nivel de anidamiento para conseguir estructuras del número de dimensiones que queramos.

Sin embargo, el hecho de que los elementos de las listas puedan ser de cualquier tipo produce una sobrecarga tanto de operaciones como de uso de memoria al tratar con ellas que las hace muy poco eficientes para tareas que necesitan ejecutarse a gran velocidad, como es el análisis de datos.

Por esa razón se creó la librería NumPy, la cual proporciona el tipo de datos *array*, que es una estructura multidimensional rectangular que soporta muy diversos típos pero fuerza a que todos sus elementos sean del mismo. Gracias a ello y a un conjunto de operaciones internas especialmente programadas para ejecutarse en modo vectorial sobre esos arrays, NumPy ofrece un alto rendimiento que no puede alcanzarse usando las listas que Python ofrece por defecto.

Importar NumPy:

```
import numpy as np
```

Necesitaremos una línea como ésta para poder usar NumPy (suponiendo que ya está instalado). Se suele utilizar *np* como alias, pero realmente podemos utilizar el que queramos.

Crear arrays a partir de listas:

```
x1 = np.array([1, 2.1, 3, 4.2, 5])
x2 = np.array([ [1,2,3], [4,5,6], [7,8,9] ])
print(x1)
print(x2)
```

```
[1.  2.1 3.  4.2 5. ]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Como podemos ver, podemos crear arrays de NumPy usando la función *array* y pasándole una lista convencional de Python.

Hay que tener en cuenta que dado que todos los elementos deben ser del mismo tipo, si hay valores decimales entonces los enteros pasan también a decimal.

También podemos ver cómo podemos crear arrays multidimensionales a partir de listas multidimensionales.

Funciones de NumPy para crear arrays:

```
x1 = np.zeros(5, dtype=float)
x2 = np.ones((2,2,2), dtype=int)
x3 = np.full((2, 4), 2.78)
x4 = np.random.random((2, 2))
print(x1)
print(x2)
print(x3)
print(x4)
```

```
[0. 0. 0. 0. 0.]
[[[1 1]
  [1 1]]

 [[1 1]
  [1 1]]]
[[2.78 2.78 2.78 2.78]
 [2.78 2.78 2.78 2.78]]
[[0.0634038  0.47718653]
 [0.56091367 0.60543974]]
```

```
[0. 0. 0. 0. 0.]
[[1 1]
 [1 1]
 [1 1]
 [1 1]]
[[2.78 2.78 2.78 2.78]
 [2.78 2.78 2.78 2.78]]
[[0.62553088 0.36445873]
 [0.34844383 0.39634507]]
```

NumPy proporciona varias funciones para crear arrays multidimensionales, algunas de las cuales aceptan un parámetro de nombre *dtype* para indicar el tipo de datos de sus valores.

Aquí hemos visto sólo algunas entre las que NumPy ofrece.

Acceso a arrays:

```
x = np.arange(8)
print(x[3])    # El elemento en la posición 3 (empezando desde 0)
print(x[:4])   # Los 4 primeros elementos
print(x[4:])   # Elementos tras el índice 4
print(x[3:6])  # Elementos intermedios
print(x[1::2]) # Elementos de 2 en 2 desde el de índice 1
```

```
3
[0 1 2 3]
[4 5 6 7]
[3 4 5]
[1 3 5 7]
```

Para acceder a arrays utilizamos una notación especial con el caracter ":", según la siguiente sintaxis (suponiendo que *x* es un array).

```
x[inicio:final:paso]
```

Esto nos proporciona los valores entre el índice *inicio* y el índice *final* (no incluído), saltando cada *paso* elementos.

Los valores por defecto si no se indica nada son:

```
start = 0
stop = tamaño de la dimensión
paso = 1
```

```
x = np.array([ [1,2,3], [4,5,6], [7,8,9] ])
print(x)
print(x[2:,:2])
print(x[::2,:])
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[7 8]]
[[1 2 3]
 [7 8 9]]
```

En el caso de arrays multidimensionales accedemos de igual modo, usando el caracter "," para separar el indexado que realizamos dentro de cada dimensión.

```
x = np.array([ [1,2,3], [4,5,6], [7,8,9] ])
y = x[1:3,1:3]
print(x)
print(y)
y[1,1] = 99
print(y)
print(x)
z = x[:2,:2].copy()
z[0,0] = 88
print(z)
print(x)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
[[5 6]
 [8 9]]
[[ 5  6]
 [ 8 99]]
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8 99]]
[[88  2]
 [ 4  5]]
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8 99]]
```

Cuando obtenemos una porción de un array mediante indexación, lo que tenemos realmente es una *vista* en lugar de una copia de los datos. Por lo tanto, si hacemos una modificación en un determinado valor de la vista, también estaremos modificando el array original.

Para evitar este efecto podemos realizar copias de los arrays mediante la función *copy*.

Operaciones con arrays:

```
x1 = np.array([1,2,3,4,5,6])
x2 = np.array([2,3,4,5,6,7])
print(x1+3)
print(x1*3)
print(x1/2)
print(x1+x2)
print(x1/x2)
print(np.exp2(x1))
print(np.power(3,x1))
print(np.log2(x1))
print(np.log10(x1))
print("Suma:  ", np.sum(x1))
print("Min:   ", np.min(x1))
print("Max:   ", np.max(x1))
print("Media: ", np.mean(x1))
print("Std:   ", np.std(x1))
print("Var:   ", np.var(x1))
```

```
[4 5 6 7 8 9]
[ 3  6  9 12 15 18]
[0.5 1.  1.5 2.  2.5 3. ]
[ 3  5  7  9 11 13]
[0.5        0.66666667 0.75       0.8        0.83333333 0.85714286]
[ 2.  4.  8. 16. 32. 64.]
[  3   9  27  81 243 729]
[0.         1.         1.5849625  2.         2.32192809 2.5849625 ]
[0.         0.30103    0.47712125 0.60205999 0.69897    0.77815125]
Suma:   21
Min:    1
Max:    6
Media:  3.5
```

```
Std:    1.707825127659933
Var:    2.9166666666666665
```

Python incluye una gran cantidad de funciones matemáticas aplicables sobre arrays.

```
x1 = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(x1+3)
print(x1*3)
print(x1/2)
print(np.exp2(x1))
print(np.power(3,x1))
print(np.log2(x1))
print(np.log10(x1))
print("Suma:  ", np.sum(x1))
print("Min:   ", np.min(x1))
print("Max:   ", np.max(x1))
print("Media: ", np.mean(x1))
print("Std:   ", np.std(x1))
print("Var:   ", np.var(x1))
```

```
[[ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[[ 3  6  9]
 [12 15 18]
 [21 24 27]]
[[0.5 1.  1.5]
 [2.  2.5 3. ]
 [3.5 4.  4.5]]
[[  2.   4.   8.]
 [ 16.  32.  64.]
 [128. 256. 512.]]
[[    3     9    27]
 [   81   243   729]
 [ 2187  6561 19683]]
[[0.         1.         1.5849625 ]
 [2.         2.32192809 2.5849625 ]
 [2.80735492 3.         3.169925  ]]
[[0.         0.30103    0.47712125]
 [0.60205999 0.69897    0.77815125]
 [0.84509804 0.90308999 0.95424251]]
Suma:   45
Min:    1
Max:    9
Media:  5.0
Std:    2.581988897471611
Var:    6.666666666666667
```

Esas funciones también pueden aplicarse sobre arrays multidimensionales.

Reflexiona

Fíjate en que en en R el índice de la primera posición es el 1 mientras que en Python (como en la mayoría de lenguajes) es el 0.

Fíjate también en que cuando en R usamos la notación *inicio:fin*, la posición *fin* está incluida y en Python no.

Para saber más

Puedes ver información adicional sobre NumPy en los siguientes enlaces:

Web oficial de NumPy

NumPy en Wikipedia

Autoevaluación

¿Con NumPy podemos crear arrays a partir de listas?

○ Verdadero ○ Falso

Verdadero

Correcto.

¿Para acceder a un array de NumPy usaremos la sintaxis x[inicio:paso:final], suponiendo que x es un array?

○ Verdadero ○ Falso

Falso

Usaremos la sintaxis x[inicio:final:paso]

¿Si x es un array bidimensional de NumPy, para tomar el valor de columna de índice 0 y fila de índice 0 accederemos mediante x[0][0]?

○ Verdadero ○ Falso

Falso

Accederemos mediante x[0,0].

4.3.- Análisis con Pandas.

Pandas es un librería de Python específicamente creada para manipulación y análisis de datos. Ha sido construida sobre las funcionalidades que ya ofrece NumPy, añadiendo un tipo de datos de vital importancia a la hora de analizar datos, llamado *DataFrame*.

Un DataFrame es el equivalente a un array bidimensional al que podemos asociar etiquetas tanto para columnas como para filas y que permite trabajar con datos desconocidos.

Del mismo modo que ocurría con NumPy y sus arrays, la gran aportación de Pandas respecto de sus DataFrames es la implementación de una serie de operaciones altamente optimizadas sobre ellos.

Además del tipo DataFrame, Pandas también incluye otra estructura de datos llamada *Series*, junto con sus correspondientes operaciones. El uso de Series queda fuera del alcance de este curso, pero lo mencionamos aquí para que el alumno conozca su existencia y pueda investigar por su cuenta si se siente interesado.

Veremos a continuación cómo emplear DataFrames de Pandas:

Importar Pandas:

```
import pandas as pd
```

Necesitaremos una línea como ésta para poder usar Pandas (suponiendo que ya está instalado). Se suele utilizar *pd* como alias, pero realmente podemos utilizar el que queramos.

Crear DataFrames:

```
lista = [['Manzana',7],['Pera',9],['Mandarina',11]]
df = pd.DataFrame(lista,columns=['Fruta','Numero'])
print(df)
```

```
       Fruta  Numero
0    Manzana       7
1       Pera       9
2  Mandarina      11
```

Creamos un DataFrame mediante la función *DataFrame*. Hemos escogido crear nuestro DataFrame a partir de listas de Python, aunque existen otros modos que el alumno interesado puede investigar.

Seleccionar una columna:

```
lista = [['Manzana',7],['Pera',9],['Mandarina',11]]
df = pd.DataFrame(lista,columns=['Fruta','Numero'])
print(df['Fruta'])
```

```
0      Manzana
1         Pera
2    Mandarina
Name: Fruta, dtype: object
```

Podemos seleccionar una columna accediendo a ella mediante su nombre.

Añadir y eliminar columnas:

```
lista = [['Manzana',7],['Pera',9],['Mandarina',11]]
df = pd.DataFrame(lista,columns=['Fruta','Numero'])
col = [3,5,7]
print(df)
df['OtroNum'] = col
print(df)
del df['Numero']
print(df)
```

```
       Fruta  Numero
0    Manzana       7
1       Pera       9
2  Mandarina      11
       Fruta  Numero  OtroNum
0    Manzana       7        3
1       Pera       9        5
2  Mandarina      11        7
       Fruta  OtroNum
0    Manzana        3
1       Pera        5
2  Mandarina        7
```

Podemos añadir columnas accediendo al array mediante el nombre que tendrá la nueva columna, o eliminar columnas utilizando *del*.

Tomar filas:

```
lista = [['Manzana',7],['Pera',9],['Mandarina',11],['Mango',2]]
df = pd.DataFrame(lista,columns=['Fruta','Numero'])
print(df)
print(df[1:3])
```

```
       Fruta  Numero
0    Manzana       7
```

```
1       Pera       9
2  Mandarina      11
3      Mango       2
       Fruta  Numero
1       Pera       9
2  Mandarina      11
```

Podemos tomar filas indexando mediante la notación *inicio:fin:paso* del mismo modo que habíamos con los arrays de NumPy.

Añadir y eliminar filas:

```
lista = [['Manzana',7],['Pera',9]]
lista2 = [['Mandarina',11],['Mango',2]]
df = pd.DataFrame(lista,columns=['Fruta','Numero'])
df2 = pd.DataFrame(lista2,columns=['Fruta','Numero'])
df = df.append(df2)
print(df)
df = df.drop(1)
print(df)
```

```
       Fruta  Numero
0    Manzana       7
1       Pera       9
0  Mandarina      11
1      Mango       2
       Fruta  Numero
0    Manzana       7
0  Mandarina      11
```

Podemos añadir las filas de un DataFrame a otro mediante *append* y eliminarlas con *drop* indicando el valor del índice a eliminar.

Fíjate que en este ejemplo el índice 1 estaba repetido, por lo que se han eliminado dos filas.

```
lista = [['Manzana',7],['Pera',9]]
lista2 = [['Mandarina',11],['Mango',2]]
df = pd.DataFrame(lista,columns=['Fruta','Numero'],index=['uno','dos'])
df2 = pd.DataFrame(lista2,columns=['Fruta','Numero'],index=['tres','cuatro'])
df = df.append(df2)
print(df)
df = df.drop('dos')
print(df)
```

```
            Fruta  Numero
uno       Manzana       7
dos          Pera       9
tres    Mandarina      11
cuatro      Mango       2
```

```
           Fruta  Numero
uno       Manzana       7
tres    Mandarina      11
cuatro      Mango       2
```

Por supuesto, podemos imponer nuestros propios índices a las filas de los DataFrame, de modo que esos borrados sean únicos.

Obtener una descripcion de los datos dentro de un DataFrame:

```
lista = [[1,2,3],[4,5,6],[7,8,9]]
df = pd.DataFrame(lista)
print(df)
print(df.describe())
```

```
   0  1  2
0  1  2  3
1  4  5  6
2  7  8  9
         0    1    2
count  3.0  3.0  3.0
mean   4.0  5.0  6.0
std    3.0  3.0  3.0
min    1.0  2.0  3.0
25%    2.5  3.5  4.5
50%    4.0  5.0  6.0
75%    5.5  6.5  7.5
max    7.0  8.0  9.0
```

Podemos ver una descripción de los datos que hay dentro de un DataFrame mediante la función *describe*.

Para saber más

Por supuesto, con Pandas y sus DataFrame podemos hacer muchas más cosas que las que vemos aquí.

Puedes ver mucho más en este *notebook* de Kaggle en el que se hace exploración de datos sobre un *dataset* de precios de casas en función de ciertos atributos.

Comprehensive data exploration with Python (en inglés)

Aquí puedes ver una descripción de los datos que se emplean en la competición de Kaggle en la cual se basa el anterior *notebook* (tendrás que hacer clic en el cuadro amarillo con flecha hacia abajo para expandir la zona

que queda sombreada y así poder ver la descripción de las columnas del *dataset*)

Data Description: House Prices - Advanced Regression Techniques (en inglés)

Autoevaluación

¿Cuáles de las siguientes afirmaciones son ciertas en relación a los DataFrames de Pandas?

- [ ] Podemos añadir columnas con *add* y eliminarlas con *del*.
- [ ] No podemos añadir columnas una vez creado el DataFrame.
- [ ] Podemos añadir columnas accediendo al array mediante el nombre que tendrá la nueva columna, o eliminar columnas utilizando *del*.
- [ ] Podemos añadir columnas pero no eliminarlas.

Mostrar retroalimentación

Solución

1. Incorrecto
2. Incorrecto
3. Correcto
4. Incorrecto

Para saber más

Puedes ver información adicional sobre Pandas en los siguientes enlaces:

Web oficial de Pandas

Pandas (software) en Wikipedia

4.4.- Visualización con Matplotlib.

Si has explorado los enlaces que aparecen en el primer "Para saber más" del apartado anterior, ya habrás visto la librería Matplotlib en funcionamiento, ya que la persona que creó el notebook la va utilizando para mostrar el aspecto que tienen los datos a medida que va realizando su análisis exploratorio de los mismos.

Matplotlib es la librería comúnmente utilizada en Python para generar todo tipo de gráficos. Existe desde antes de la llegada de Pandas, por lo que se diseñó para trabajar con listas convencionales de Python así como con los arrays de NumPy. Sin embargo, a día de hoy también funciona sobre DataFrames y Series de Pandas.

Importar MatplotLib:

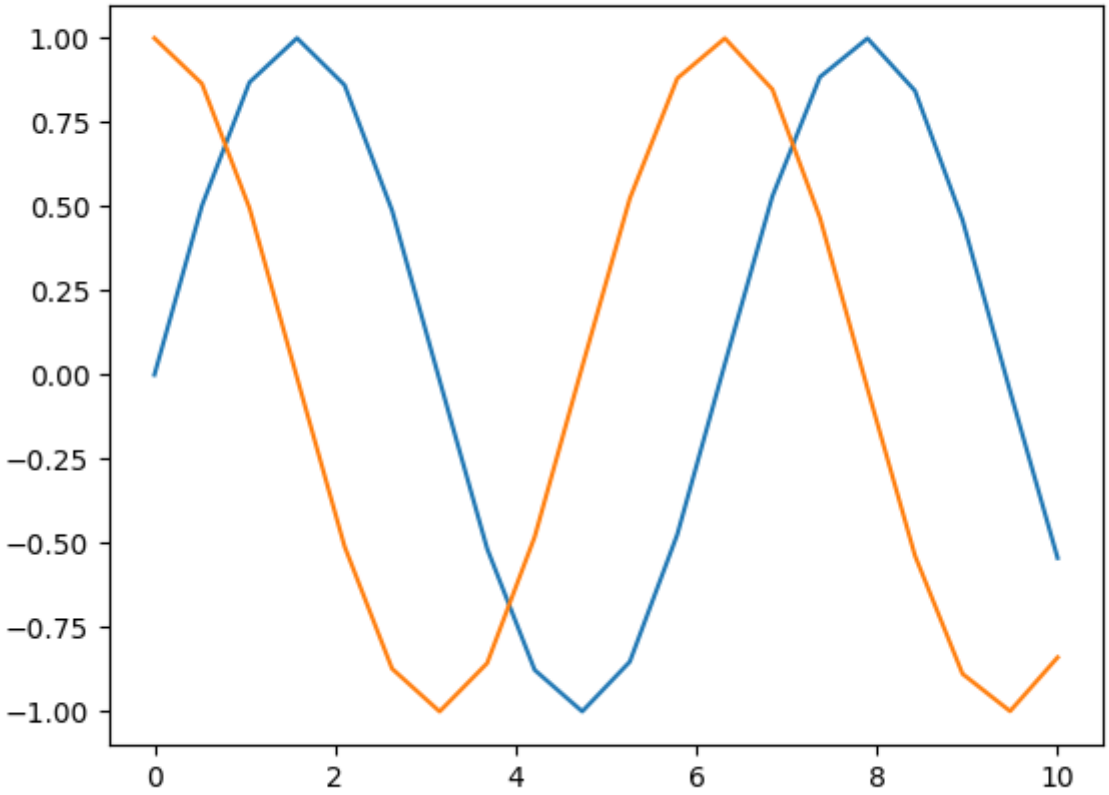
```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Necesitaremos un par de líneas éstas para poder usar tanto *matplotlib* como *matplotlib.pyplot* (suponiendo que ya están instalados). Se suele utilizar *mpl* y *plt* como alias, pero realmente podemos utilizar los que queramos.

Creación básica de gráficos:

```
import numpy as np
x = np.linspace(0, 10, 20)
print(x)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.show()
```

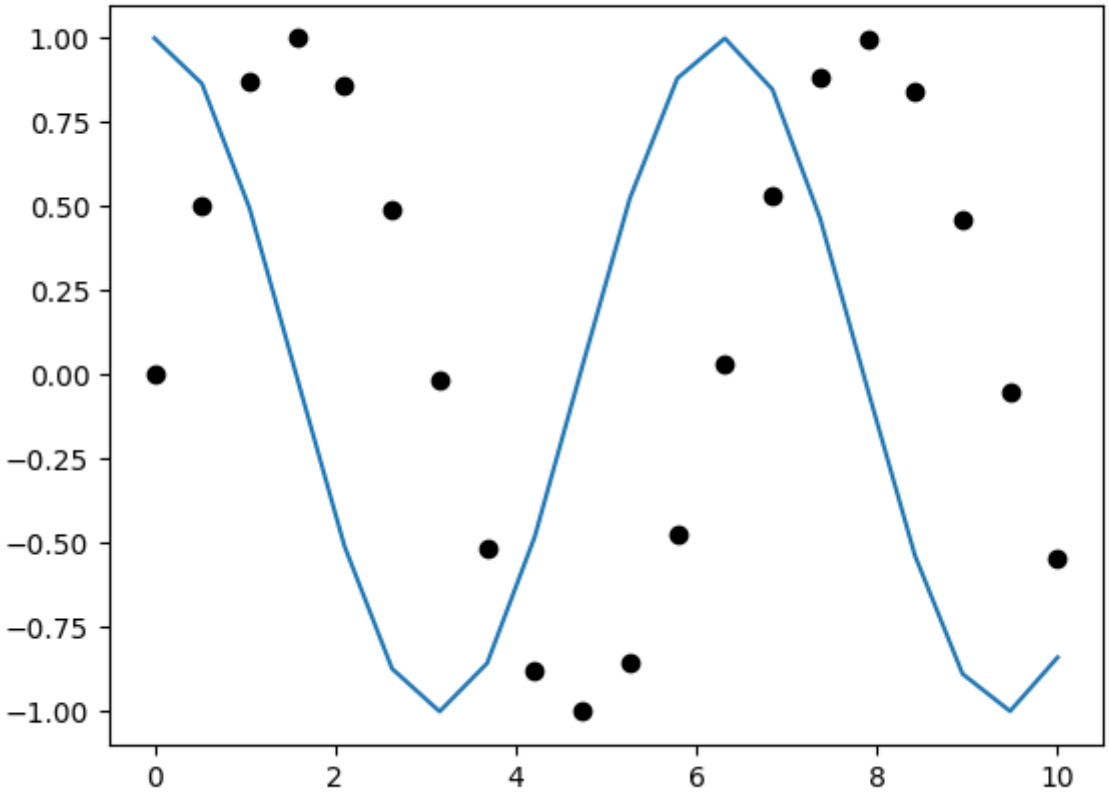
```
[ 0.          0.52631579  1.05263158  1.57894737  2.10526316  2.63157895
  3.15789474  3.68421053  4.21052632  4.73684211  5.26315789  5.78947368
  6.31578947  6.84210526  7.36842105  7.89473684  8.42105263  8.94736842
  9.47368421 10.        ]
```



Víctor Tomico (Dominio público)

En este ejemplo hemos creado una sencilla gráfica aplicando directamente Matplotlib sobre un array de NumPy mediante la función *plot*.

```
import numpy as np
x = np.linspace(0, 10, 20)
plt.plot(x, np.sin(x), 'o', color="black" )
plt.plot(x, np.cos(x))
plt.show()
```



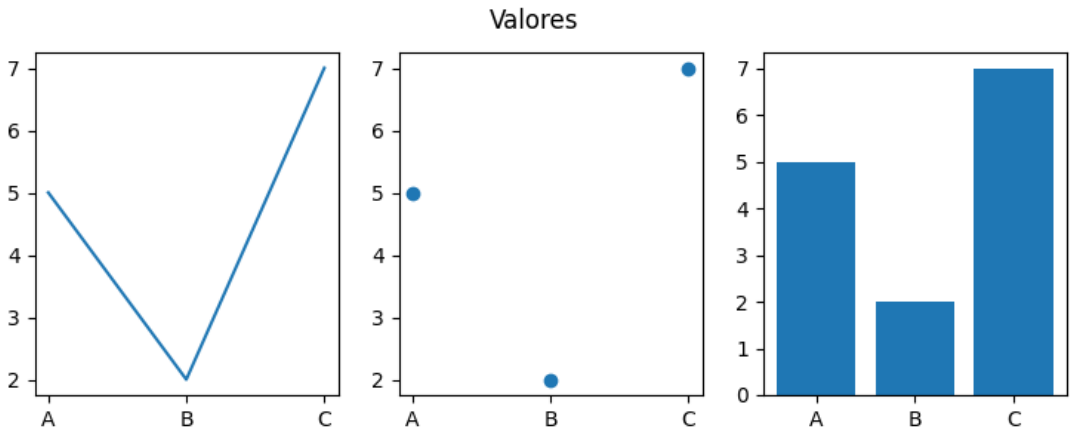
Víctor Tomico (Dominio público)

Por defecto la función *plot* genera líneas, pero podemos dibujar puntos pasándole el parámetro 'o'.

```
names = ['A', 'B', 'C']
values = [5, 2, 7]

plt.figure(figsize=(9, 3))

plt.subplot(131)
plt.plot(names, values)
plt.subplot(132)
plt.scatter(names, values)
plt.subplot(133)
plt.bar(names, values)
plt.suptitle('Valores')
plt.show()
```



Víctor Tomico (Dominio público)

También podemos crear gráficos de barras e incluso crear un gráfico con varios subgráficos. En este caso hemos creado un gráfico de línea con la función *plot*, un gráfico de puntos con la función *scatter* y otro de barras con la función *bar*.

Lo cierto es que con *matplotlib.pyplot* podemos crear gráficos muy elaborados. Por ello, es interesante que veas el tutorial acerca de Pyplot que te indicamos en la sección "Para saber más".

Para saber más

Puedes ver un tutorial sobre Pyplot en el siguiente enlace, dentro de la página oficial de Matplotlib.

Pyplot tutorial (en inglés)

Puedes ver más información sobre Matplotlib en su página oficial:

Página oficial de Matplotlib

También puedes ver más información sobre Matplotlib en el siguiente enlace:

Matplotlib en Wikipedia

Autoevaluación

¿Cuáles de las siguientes afirmaciones son correctas respecto de Matplotlib?

- [ ] Sólo puede emplearse con arrays de NumPy.

- [ ] Puede emplearse con DataFrames y Series de Pandas.

- [ ] Puede crear diversos tipos de gráficos, entre ellos los de línea, de puntos o de barra.

- [ ] Sólo puede dibujar un gráfico a la vez.

Mostrar retroalimentación

Solución

1. Incorrecto
2. Correcto
3. Correcto
4. Incorrecto