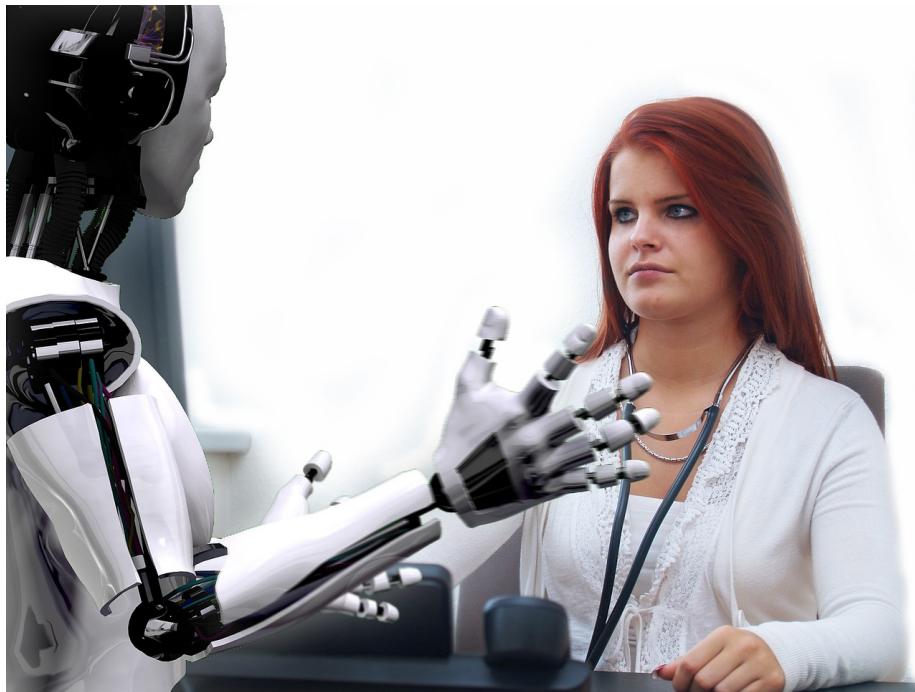


Esta sección tiene como objetivo que tengas una visión general acerca del tema central de este libro: La extracción de conocimiento a partir de grandes cantidades de datos. Como tal vez te imagines esto no es una tarea sencilla ya que involucra distintas áreas de investigación científica y un buen número de tecnologías. No te preocupes, vamos a empezar por revisar los conceptos básicos procurando que tengas una buena idea de donde y cuando aplicar estas técnicas. Algunas de las preguntas que podrás responder al final del capítulo son:

- ¿Qué buscamos en los datos?.
 - ¿Cuál es proceso para el descubrimiento de conocimiento en bases de datos?.
 - ¿Qué tecnologías intervienen en este proceso?.
 - ¿Qué técnicas de minería de datos aprenderás a utilizar?.
 - ¿Dónde podemos aplicar la minería de datos?.
-
- ¿Que retos y limitaciones existen actualmente en el área?.

Introducción



Cuando visitamos al médico, confiamos en que el haya adquirido un amplio **conocimiento** a base de interactuar con su entorno y su capacidad de procesar la información que lo rodea. Utilizando este conocimiento él podrá ser capaz de darnos un diagnóstico acertado. Si en lugar de asistir con un humano eligiéramos visitar a un robot-médico, más vale que cuente con una buena

inteligencia artificial. Según Rusell y Norvig el término inteligencia artificial se aplica cuando una máquina imita las funciones cognitivas que nosotros los humanos asociamos con otras mentes humanas, como por ejemplo: *aprender* y **resolver problemas** [ref]. Precisamente, una de las primeras aplicaciones de la inteligencia artificial han sido los Sistemas Expertos (García-Sánchez et al. 2013), los cuales pueden emular la toma de decisiones de un humano experto, tal como un médico. Un exponente importante de este tipo de sistemas fue Mycin el cual se basaba en un sencillo motor de inferencia, que manejaba una base de conocimiento de aproximadamente unas 500 reglas. Estas reglas representaban el conocimiento de los médicos. Una desventaja de este tipo de sistemas era que se requería de mucho esfuerzo y recursos para extraer el conocimiento de los expertos. Se debía entrevistarlos utilizando técnicas de elicitación del conocimiento para después representar dicho conocimiento de alguna manera para después almacenarlo en una **base de conocimiento**. Digamos que se realizaba una extracción manual del conocimiento a partir de las ideas de los expertos. Cualquier desarrollador que ha tenido que extraer los requerimientos de un usuario te dirá que fue una tarea cercana a lo imposible. No es descabellado preguntarnos entonces: ¿el conocimiento “humano” podrá ser extraído de otras fuentes?, ¿y si contamos con una base de datos de miles de diagnósticos realizados anteriormente?, ¿podremos extraer de esta base de datos un nuevo conocimiento?. Este es el tipo de preguntas son las que se trata de contestar el área de investigación en inteligencia artificial, pero como veremos más adelante, en la búsqueda de respuestas se involucran muchas otras áreas de investigación.

Descubrimiento de conocimiento en bases de datos (KDD)

Al proceso de extraer conocimiento útil a partir de datos se le denomina *Descubrimiento de Conocimiento en Bases de Datos* o *KDD* ya que es muy común utilizar las siglas en inglés de “Knowledge Discovery from Databases”. Este proceso puede hacerse manualmente, expertos en algún dominio pueden consultar y analizar bases de datos para descubrir patrones que les ayuden a tomar decisiones. Por ejemplo, en la película “The Big Short” podemos ver al inversionista Michael Burry analizando grandes cantidades de datos, para después predecir el eminente desplome de la burbuja inmobiliaria. Como podemos imaginar, este proceso no es trivial, no es simplemente hacer una consulta en una base de datos o en un motor de búsqueda. Es necesario encontrar patrones que involucren diferentes variables y relaciones no lineales entre ellas. El KDD sigue evolucionando involucrando cada vez más áreas de investigación: aprendizaje automático, reconocimiento de patrones, computación inteligente, estadística, procesamiento de lenguaje natural, visualización, ingeniería de software entre otras.

La Minería de Datos de hecho es uno de los componentes del proceso KDD. Veamos el proceso del KDD, según el esquema de Brachman y Anand:

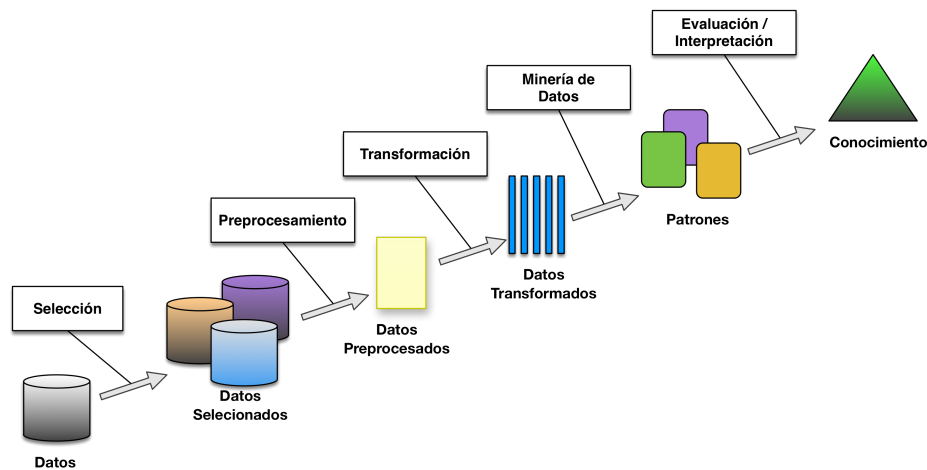


Figure 1: El proceso del KDD según Brachman y Anand

1. Como primer paso debemos *identificar el objetivo* del proceso de KDD. Por ejemplo, un proveedor de telefonía móvil podría estar interesado en identificar a aquellos clientes que ya no renovarían su contrato y se irán con la competencia. A esto se le llama la tasa de cancelación de clientes (en inglés churn rate o attrition rate) y es crucial para estimar el desempeño de la empresa.
2. El siguiente paso es *seleccionar y recolectar* aquellos datos necesarios para el proceso. Para nuestro ejemplo, podríamos requerir el historial de pago de los clientes, datos sobre quejas y llamadas que han hecho a soporte, servicios adicionales que se han contratado o cancelado, etc. Esta información puede estar distribuida en diferentes bases de datos. También se puede incluir información que se colecte por medio de sensores o sistemas externos, por ejemplo lecturas del GPS, caídas de la conexión o el número de aplicaciones que el cliente se ha instalado.
3. Es necesario *preprocesar* los datos para limpiarlos de datos erróneos, datos faltantes, cambios de formato, inconsistencias, etc. Este proceso complicado y depende bastante de los requerimientos de la empresa.
4. Dependiendo de los objetivos se deben *transformar* los datos para simplificar su procesamiento. Por ejemplo un documento de texto es necesario transformarlo en vector representativo. Una imagen puede ser transformada en una representación simplificada pero conservando sus características esenciales. Muchas veces también es necesario eliminar campos que no aportan mucho a la tarea de KDD. Siguiendo nuestro ejemplo, después de un análisis podríamos darnos cuenta que el número de teléfono en sí no es un dato importante para distinguir el comportamiento de los clientes, y que al contrario la marca y modelo de sus móviles son muy importantes.

5. Este paso consiste en *seleccionar la tarea de minería de datos* adecuado de acuerdo con la meta establecida para el proceso de KDD. Por ejemplo, clasificación, regresión, agrupamiento, etc.
6. Se hace un *análisis exploratorio*, podemos experimentar con diferentes algoritmos de minería de datos o de aprendizaje automático. Al seleccionar los algoritmos debemos considerar los tipos de datos que tenemos. Por ejemplo algunos algoritmos no son apropiados para datos categóricos (modelo de celular). También debemos ajustar los parámetros para mejorar el desempeño y comparar el rendimiento entre los algoritmos.
7. En este paso se realiza la *minería de datos* a partir de el o los algoritmos seleccionados en el paso anterior.

Este paso nos arroja los *patrones ocultos* que describen a los datos. Siguiendo nuestro ejemplo, el resultado podría ser un conjunto de reglas nos que pueden servir para decidir si un cliente cancelará su suscripción. Una de las regla podría ser:

```
SI el cliente tiene un promedio mayor a 7 días de retraso
AND su promedio de llamadas al mes es menor que 10
ENTONCES:
    el cliente cancelara el servicio
```

Los patrones son entonces Modelos que se ajustan a los datos. Los modelos no siempre están expresados en un lenguaje que los humanos podamos entender. Por ejemplo el resultado de un algoritmo de agrupamiento podrían ser simplemente grupos de clientes, que después deberíamos de interpretar.

En el texto utilizaremos el término modelo en lugar de patrones, ya que es más usual actualmente. Algo muy importante es que los modelos deben de representar también a datos nuevos. Digamos nuevos clientes que se suscriban el mes siguiente.

8. En este paso nos toca *interpretar y evaluar* los patrones que encontramos en el paso anterior. Visualizar los patrones y modelos extraídos. Decidir si se ha encontrado algún conocimiento útil.
9. El último paso es hacer algo con el conocimiento adquirido. Se puede utilizar directamente para tomar decisiones, incorporarlo como parte de un sistema o simplemente reportar los resultados a los interesados.

Algunas definiciones y Tecnologías Complementarias



Minería de datos El

libro se enfocará principalmente en el componente de Minería de Datos, que como vimos es un paso importante en el proceso de KDD. Pero también te haz dado cuenta que es muy importante hacer bien los pasos anteriores. De hecho en la mayoría de los proyectos los otros pasos requieren de mayor trabajo. La minería de datos entonces podemos definirla como: El proceso de búsqueda de patrones aplicando distintos algoritmos a grandes cantidades de datos. Algunas veces se utiliza el término Minería de Datos para referirse a todo el proceso de KDD.

Aprendizaje automático (Machine Learning)

El aprendizaje automático es el área de las ciencias computacionales encargada de estudiar y desarrollar los algoritmos capaces de aprender y hacer predicciones a partir de los datos. Podríamos decir que en el proceso de vamos a aplicar algoritmos de aprendizaje automático, por lo que debemos conocer muy bien sus fundamentos y limitaciones.

Ciencia de datos

Como hemos visto el proceso de KDD busca extraer conocimiento partir de los datos. ¿Y si vamos más allá?. La ciencia de datos es similar al KDD pero el conocimiento que se busca es el conocimiento científico. Es hacer ciencia a partir de los datos. La ciencia apoyada en datos (en inglés data-driven science) al igual que el KDD es una campo interdisciplinario que incluye métodos científicos, procesos, y sistemas para extraer conocimiento o entendimiento a partir de los datos.

Computación inteligente

La expresión computación inteligente (CI de computational intelligence) se asocia a la habilidad de un sistema de cómputo de aprender a realizar una tarea a partir de datos o la observación. Una distinción importante es que los métodos empleados son inspirados en la naturaleza. La CI busca resolver problemas complejos del mundo-real para los cuales el modelado tradicional o matemático son insuficientes. Los métodos más representativos son: redes neuronales artificiales, lógica difusa y computación evolutiva.

Big Data

Big Data se refiere al caso de sistemas que cuentan con cantidades enormes de datos. Normalmente los datos se almacenan en un servidor central. En el caso de Big Data los datos son tantos que deben almacenarse en muchos servidores. También el tipo de dato podría ser muy grande, por ejemplo un registro de un experimento determinado podría medir un terabyte. En estos casos las técnicas o algoritmos tradicionales resultan inadecuados. El usar Big Data no implica Minería de Datos, muchos sistemas solamente requieren procesar los datos, por ejemplo para realizar una consulta o algún cálculo. Claro que también hay procesos de Minería de Datos que se realizan utilizando Big Data, esto requiere de tecnología y algoritmos especializados.

Cloud Computing

¿Quién no ha escuchado esto de la nube?. El Cloud Computing es el nuevo modelo de desarrollo de sistemas. Básicamente pagamos por el poder computacional más servicios adicionales como bases de datos, algoritmos en línea y software pagando solo lo que ocupemos. Empresas con mucha experiencia en minería de datos e inmensos recursos computacionales están brindando el servicio de *On-Demand Machine Learning*. Google con ml-engine, Amazon ML, IBM con Watson, Azure de Microsoft entre muchos otros nos permiten hacer todo el proceso de KDD en sus servidores siguiendo el modelo de Cloud Computing. En lugar de nosotros implementar los algoritmos, instalar los lenguajes, procesar los datos, mantener todo actualizado, etc. podemos contratar el servicio de Almacenamiento, Procesamiento y Machine Learning. Nuestra labor se volverá entonces elegir los algoritmos adecuados, preparar los datos y otras tareas más bien de diseño.

Vamos a implementar nuestros sistemas pensando siempre en la nube como plataforma de ejecución. Esto nos permitirá desplegar elementos de nuestro sistema en la nube o utilizar diferentes servicios bajo demanda. Por último, tal vez después de que diseñes un algoritmo que funcione muy bien, podrías ponerlo a funcionar en la nube para que lo utilicemos tus clientes.

Algunas Aplicaciones

Modelado de sistemas no lineales

En general se utiliza la Minería de Datos para modelar sistemas no lineales. En caso de que no estés familiarizado veamos un ejemplo. Ana entra a trabajar a las 7:00 am. El trabajo de Ana está a 10 km de su casa. Ella normalmente sale de su casa a las 6:20 y llega a las 6:40, hace 20 minutos. En ocasiones sale a las 6:10 y claro llega a las 6:30. Hasta aquí todo va bien, los tiempos siguen un comportamiento lineal y constante. Incluso podemos calcular que si antes deja a sus hijos en la escuela que está a 5 km, llegará en 10 minutos. El problema viene cuando Ana sale de casa a las 6:25 en ese caso hace 25 minutos y si sale a las 6:30 hace 40 minutos y llega tarde a su trabajo. ¿Por que el sistema no sigue un comportamiento lineal?. El problema es que el tiempo que hace a su trabajo no solo depende de la velocidad promedio y la distancia. El tiempo depende del tráfico, la ruta que tome, si se pone un policía a dirigir el tráfico, si hay alguna manifestación, si las primarias están de vacaciones, etc. El problema se vuelve no lineal por que está ubicado en el mundo real, y si queremos hacer un calculo más exacto tendríamos que considerar muchas variables, incluso podríamos hacer un simulador. Cuando salimos a nuestro trabajo muchos hacemos un cálculo mental del tiempo que haremos por eso los optimistas llegamos tarde. Pero, ¿en que basamos nuestro cálculo si el sistema es no lineal y por lo tanto muy complejo?. Así es, en nuestra experiencia, es decir en extraer de nuestra mente el tiempo que hemos hecho los días anteriores, de nuestro sentido común “hoy es viernes, habrá mucho tráfico”. La minería de datos puede ayudarnos a modelar este tipo de sistemas, sin que tengamos nosotros que establecer las relaciones no lineales entre muchas variables.

El Ruido

El ruido es una complicación común en sistemas del mundo-real. Por ejemplo, cuando se le pone un sensor a un paciente, en ocasiones las lecturas se ven afectadas por los movimientos del paciente y registran valores erróneos o muy altos o bajos en comparación con el valor real. Descubrir patrones en presencia de ruido es bastante difícil, por lo que se deben elegir técnicas de modelado que toleren el ruido.

Internet de las Cosas

El concepto de Internet de las Cosas (IoT) se refiere básicamente a que todas las cosas están conectadas al Internet y pueden recopilar e intercambiar datos. Crear modelos de sistemas no lineales con muchísimas variables es muy complicado. La tecnología de IoT se puede aplicar en sistemas de ciudades inteligentes, para agilizar el tráfico en tiempo real o abrir el paso a los bomberos cuando se dirigen a atender una emergencia. Estos sistemas deben considerar las lecturas

en tiempo real de muchísimos sensores y determinar como se afectan entre sí, para determinar si hay alguna emergencia grave, predecir el tráfico. Utilizando técnicas de aprendizaje automático es posible modelar este tipo de sistemas.



Figure 2: Internet de las Cosas

Reconocimiento de patrones en multimedia

Al momento de subir a Facebook la tradicional *selfie* con los amigos, el sistema nos sugiere etiquetar las caras de ellos con su nombre de usuario correcto. Es decir, el sistema reconoció al usuario a partir de la imagen una imagen de rostro. Esto es parecido a lo que hicimos anteriormente con las flores Iris. Pero en lugar de 4 variables de entrada tenemos un arreglo de los cientos de píxeles que hacen la imagen y en lugar de tres flores, tenemos cientos de amigos. En el caso de las flores recordamos que ya estaban etiquetadas. En el caso de Facebook, la extracción del modelo requiere que ya haya ejemplos de los rostros asociados a los usuarios. Nosotros mismos agregamos ejemplos clasificados cada que manualmente etiquetamos un rostro. Las mismas técnicas pueden utilizarse para reconocer gestos, la voz, huellas digitales, firmas, letra escrita, copias de videos y muchos otros patrones.

Negocios

Se utiliza el KDD principalmente para el mercadeo dirigido, para calcular el riesgo de alguna operación, predecir cambios en los mercados, identificar tendencias a partir de las publicaciones en redes sociales, recomendar productos o personalizar servicios.

Minería de Textos

Resumen de correos, noticias. Análisis de Sentimientos.



Figure 3: Reconocimiento de patrones en multimedia

Ciencia

POR HACER

Tareas de la minería de datos

Clasificación

Un problema muy común es el de asignar un objeto a una clase o categoría. Veamos varios ejemplos. Los correos electrónicos que recibimos podemos clasificarlos en dos clases: *correo válido* o *correo no deseado*. Actualmente en Google Mail hay otras categorías: *social*, *promociones*, *Foros*. La noticia de un periódico: *deportes*, *nacional*, *internacional*. Un usuario visitando una tienda en-línea: *compra*, *no-compra*. Una imagen de dígitos escritos a mano: *0,1,2,3,4,5,6,7,8,9*. La imagen de una galaxia: *suave,disco,estrella*. La imagen de una persona: *ana*, *tom*, *joe*, *sue*, etc. El problema de Clasificación es el de identificar a que clase pertenece un objeto, basándose en un conjunto de objetos en los cuales la clase o categoría ya se conoce. Por ejemplo para el caso del la clasificación de correos electrónicos, primero debemos contar con un buen número de correos que han sido clasificados previamente por humanos. A partir de estos datos un algoritmo de clasificación es capaz de aprender un modelo que podemos utilizar después para clasificar nuevos correos que nos lleguen.

La tarea entonces es predecir una clase o etiqueta a partir de un conjunto de características.

Regresión

La Regresión es la misma tarea de la clasificación pero en lugar de asignarle una clase a cada instancia le asignamos un valor continuo. Por ejemplo, las instancias podrían ser casas y la variable objetivo podría ser el costo de venta en el mercado. De nuevo para hacer la predicción nos basamos en un conjunto de casas que además de tener valores en todas las características cuentan también con el precio de venta.

Objetivo del Aprendizaje Supervisado

El objetivo de la Clasificación y la Regresión es el de minimizar el error de predicción. El error es la diferencia que hay entre el valor o categoría reales y la predicción. Este es un tema muy importante que veremos en la sección de [Evaluación de Modelos]. Las dos tareas entonces son ejemplos de aprendizaje supervisado ya que requieren de datos de entrenamiento para extraer de ahí el modelo o la predicción directamente.

Agrupamiento

El agrupamiento es un ejemplo de Aprendizaje no supervisado en el cual no se requiere de contar con datos previamente clasificados. El objetivo del análisis de Agrupamiento o *Clustering* es el de encontrar grupos de objetos los cuales estén estrechamente relacionados o sean similares. El objetivo en este caso es tratar de que la distancia promedio entre los miembros del mismo grupo sea menor que la distancia promedio entre objetos de distintos grupos. Por ejemplo, un sistema de recuperación de información puede regresar los documentos de una búsqueda con mayor eficiencia si los documentos similares se guardan juntos. También podemos encontrar a clientes similares a los cuales les podríamos ofrecer ciertos servicios.

Análisis de asociación

En ocasiones podemos descubrir patrones interesantes los cuales asocian hechos que con frecuencia suceden al mismo tiempo o en cierto orden. En México por ejemplo podríamos encontrar que si un consumidor compra cebollas y tomate al mismo tiempo, es muy probable que también compre carne. Es muy probable que las tortillas las compre siempre. Normalmente los patrones de asociación se expresan como reglas de asociación. Para el ejemplo anterior la regla sería:

{cebollas, tomate} -> {carne}

Este tipo de análisis se utiliza para hacer toma de decisiones de marketing, ubicación de artículos en aparadores, ofrecer agregar algún producto al carrito de compra, entre otros.

Detección de anomalías

El objetivo de esta tarea es encontrar objetos que sean muy diferentes al resto. A este tipo de objetos se les conoce como anomalías o outliers. El reto es no etiquetar como anomalías a objetos que no lo son y viceversa. Este tipo de análisis nos permite detectar fraudes con las tarjetas de crédito, robo de identidad, ciertas enfermedades, problemas en ecosistemas entre otras aplicaciones.

Optimización

La optimización no es en si misma una tarea de la minería de datos, pero la mayoría de las técnicas que se utilizan para realizar las tareas anteriormente utilizan algoritmos de optimización. Un algoritmo de optimización podemos verlo como un algoritmo de búsqueda. Tratamos de encontrar una combinación de objetos que nos resuelva un problema de manera óptima. En este libro nos enfocaremos en algoritmos de optimización inspirados en la naturaleza.

¿Buscar o Inferir Patrones?

Hay casos en los que un algoritmo de optimización puede utilizarse para hacer minería de datos. El problema de extracción de conocimiento podemos verlo como la búsqueda del modelo óptimo de para los datos en cuestión. Por ejemplo debemos buscar el conjunto de reglas que mejor clasifique ciertos correos electrónicos. Es importante recordar que la extracción de conocimiento no siempre se basa en inferencia estadística.

Los Retos

Tan, Steinbach y Kumar mencionan los principales retos que han motivado el desarrollo de la minería de datos:

- **Escalabilidad.** Actualmente contamos con inmensas cantidades de datos. No es raro utilizar conjuntos de datos que rondan en los terabytes o incluso petabytes. Los algoritmos, estructuras y bases de datos deben ser capaces de escalar a los niveles del *Big Data*. Esto sigue siendo un reto, que involucra otras áreas como el computo distribuido y paralelo, cómputo en la nube. Además se deben de utilizar técnicas de muestreo.
- **Alta Dimensión.** Hace algunos años era inconcebible tener objetos con miles de características. Ahora este tipo de instancias son comunes. Si un objeto tuviera solo dos o tres características lo podríamos representar

en 2D o en 3D un eje por cada una. Pero cuando un objeto tiene muchas características y sobrepasa por mucho el 3D decimos que se encuentra en un espacio de alta dimensión. Podríamos imaginarnos que el espacio donde se encuentra ubicado el objeto es cada vez mayor por cada nueva dimensión que agreguemos y comparativamente el número de objetos que cabrían en ese espacio sería muchísimo mayor. A este fenómeno se le conoce como la maldición de la dimensión pues el volumen del espacio aumenta exponencialmente haciendo que los datos disponibles se vuelvan dispersos. El reto es que algunos algoritmos de análisis de datos no dan buenos resultados al trabajar en altas dimensiones.

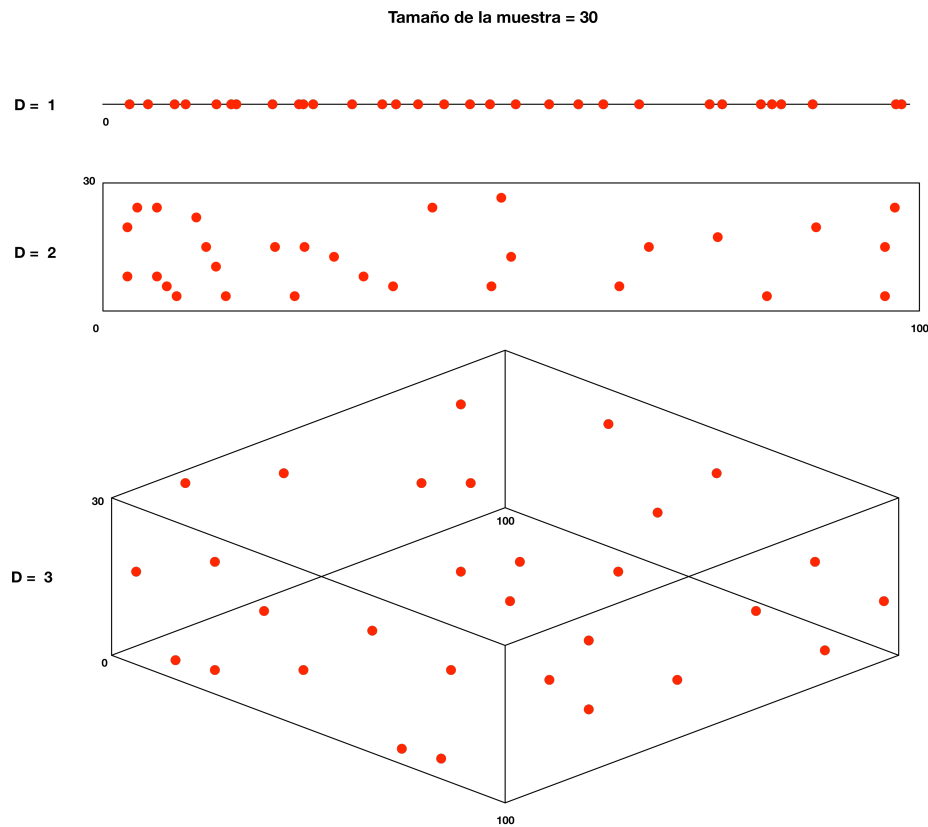


Figure 4: El tamaño del espacio aumenta exponencialmente haciendo que los datos disponibles se vuelvan dispersos

- **Datos Heterogéneos y Complejos.** Los datos que tradicionalmente se han empleado en análisis de datos son los categóricos o continuos. En la actualidad contamos con datos muy diversos como documentos, páginas web, videos, lecturas de sensores, datos con georreferenciación, series de tiempo, grafos, secuencias DNA entre muchos otros. Los algoritmos deben

considerar las relaciones y propiedades intrínsecas a los datos, relaciones temporales, propiedades de los grafos, etc.

Resumen

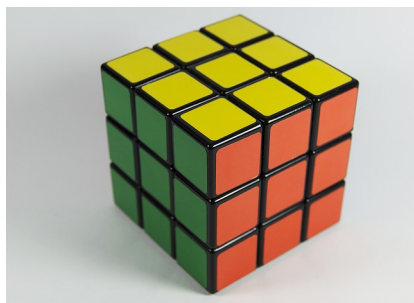
Ejercicios

Lectura adicional

Visión General de KDD

KD Nuggets Es un sitio con mucha información relacionada con minería de datos y KDD.

Bibliografía



Numpy

Computo Numérico en Python

El lenguaje Fortran con sus más de 50 años sigue siendo muy importante en las áreas de computo científico y de alto rendimiento. Fortran se considera la *referencia* cuando se habla de bibliotecas para cómputo numérico, bibliotecas como LAPACK (Linear Algebra Package) o BLAS que sirven para realizar (entre otras cosas) operaciones con vectores y matrices, tienen más de 25 años de desarrollo. Fuera del computo científico el lenguaje no es tan popular lo que ha provocado que se desarrollen herramientas e incluso lenguajes de programación para realizar operaciones numéricas utilizando sus bibliotecas. Ejemplos de herramientas de código cerrado y propietario incluyen a MATLAB, Maple y Mathematica. El problema que tienen estas plataformas es que crean una dependencia del proveedor y además representan un alto costo a los usuarios. El problema mayor que tienen estas herramientas es que van en contra del movimiento de ciencia abierta. Existen opciones de código libre que incluso son compatibles con los productos anteriores, como GNU Octave y SageMath. En el caso de lenguajes de programación libres, dos exponentes importantes son Julia para

el cómputo numérico y R para el computo estadístico. En el caso de Python contamos con la biblioteca NumPy como componente básico para el computo científico numérico. Numpy entre otras cosas nos brinda:

- Una estructura para arreglos de N-dimensiones.
- Herramientas para integrar de manera transparente código de Fortran y C//C++
- Soporte para álgebra lineal, transformadas de Fourier y números aleatorios.
- Integración con sistemas de bases de datos.
- Operaciones entre arreglos de distintas dimensiones (Broadcasting).

El código de NumPy tiene una licencia de código abierto BSD.

NumPy se utiliza en conjunto con otras bibliotecas y herramientas como matplotlib, SciPy, pandas, Jupyter, IPython, scikit-learn, theano, orange para complementar una plataforma para el análisis de datos bastante decente. Una desventaja de tener tantos proyectos por separado es que podemos sufrir al momento de integrarlo, para evitarnos la pena se han desarrollado distribuciones que integran todo esto y más, por ejemplo sage, Canopy y Anaconda Distribution.

Esta sección tipo taller, tiene como objetivo que conozcas y apliques los arreglos multidimensionales que ofrece la biblioteca NumPy. Estos conocimientos los utilizaremos en las secciones posteriores y de seguro lo agregarás a tu caja de herramientas.

NumPy: array

Mientras que en Python tenemos colecciones de objetos tipo secuencia como las listas, éstas no tienen una estructura adecuada que nos permita realizar operaciones generales sobre ellas. Como ejemplo tenemos la siguiente lista de listas:

```
>>> lista_objetos = [[1,2,3],  
                    [2,2],  
                    ['Hola', 11],  
                    [2]]
```

Esta lista tiene dos problemas: las listas son de distintos tamaños y tienen objetos de distintos tipos. Por ejemplo, no podemos sumar todos los números en la primera posición o en la tercera. En el primer caso no podemos sumar la cadena 'Hola' y en el segundo hay dos listas que no tienen un elemento en la tercera posición.

La estructura básica de NumPy es un arreglo homogéneo multidimensional. Dicho de otra manera, un arreglo es una lista que debe seguir ciertas reglas. Primero, es un arreglo homogéneo, es decir todos los elementos son del mismo

tipo, casi siempre numérico. Segundo, todas las listas contenidas tienen el mismo tamaño. Vamos a crear una lista que siga estas reglas:

```
>>> lista_nums = [[1, 2, 3],
                  [2, 2, 4],
                  [2, 3, 11],
                  [2, 1, 4]]
```

Esta lista es homogénea, ya que tiene puros enteros. Digamos que es rectangular ya que todas las listas interiores tienen el mismo número de elementos. Aunque no podemos decir todavía que es un arreglo homogéneo de dos dimensiones, vamos a crear un arreglo de NumPy a partir de nuestra lista:

```
>>> import numpy as np
>>> arreglo = np.array(lista_nums)
>>> arreglo
array([[ 1,  2,  3],
       [ 2,  2,  4],
       [ 2,  3, 11],
       [ 2,  1,  4]])
```

Ahora sí. Una dimensión de este arreglo son los renglones y otra dimensión son las columnas. ¿Y su tipo de dato?:

```
>>> arreglo.dtype
dtype('int64')
```

¿Que pasaría si queremos hacer un arreglo con la anterior lista de objetos?. Vamos a intentarlo:

```
>>> arreglo_objs = np.array(lista_objetos)
>>> arreglo_objs
array([[1, 2, 3], [2, 2], ['Hola', 11], [2]], dtype=object)
```

Como vemos no se crea un arreglo de dos dimensiones, se crea un arreglo de una sola dimensión con objetos de un mismo tipo: **object**. Esto por supuesto no es muy útil. Nos damos cuenta que *arreglo_objs* es un arreglo de dimensión uno, homogéneo pero no de una manera muy útil ya que no podemos hacer muchas operaciones matemáticas con objetos tipo **object**.

Mejor vamos a crear un arreglo *Dimensión-1* que contenga solamente enteros:

```
>>> arreglo_enteros = np.array([1, 2, 45])
>>> arreglo_enteros
array([ 1,  2, 45])
```

Vamos a comparar el atributo dimensión de los arreglos:

```
>>> arreglo.ndim
2
>>> arreglo_enteros.ndim
1
```

El número de dimensiones se le conoce en NumPy como el **rank** (rango) del arreglo. Un atributo importante es la *forma* (**shape**) del arreglo:

```
>>> arreglo.shape
(4, 3)
```

El primer valor de la tupla nos dice el número de renglones y el segundo el número de columnas. Para acordarnos de este orden podemos imaginar que los arreglos o matrices son como los cines. Para encontrar nuestro asiento primero debemos ubicar la fila (renglón) y después el número de asiento. Veamos el caso del arreglo de una dimensión:

```
>>> arreglo_enteros.shape
(3,)
```

En este caso podemos ver que se trata de un cine con un solo asiento por fila. El tamaño es el número de elementos, que como vemos es el equivalente del producto de los elementos de la tupla *shape*.

```
>>> arreglo.size # 4*3
12
```

También podemos especificar el tipo de dato al crear el arreglo. Por ejemplo:

```
>>> arreglo_complejos = np.array( [ [3 , 2], [3, 4] ], dtype=complex )
>>> arreglo_complejos
array([[ 3.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

En ocasiones queremos crear arreglos con datos iniciales, por ejemplo ceros, unos o valores arbitrarios:

```
>>> np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> np.ones( (2,3,4))
array([[[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]])
>>> np.empty( (6,))
array([ -2.68156159e+154, -2.68156159e+154,  2.23182566e-314,
        -2.68156159e+154,  6.94181405e-310,  6.15378780e-313])
```


Selección y cortes en arreglos multidimensionales

Vamos a considerar la siguiente lista de calificaciones donde cada alumno tiene tres calificaciones que van de 0 a 100:

id	nombre	tarea	examen	proyecto
1	Juan Pérez	86.2	90.0	95.0
2	Ana Rivas	100.0	95.0	95.0
3	María Sánchez	76.5	100.0	85.0
4	Luis Duarte	89.0	90.0	95.0

Vamos a capturar las calificaciones en un arreglo de NumPy:

```
>>> lista_nums = [[86.2, 90.0, 95.0], [100.0, 95.0, 95.0],
                  [76.5, 100.0, 85.0], [89.0, 90.0, 95.0]]
>>> cal = np.array(lista_nums)
>>> cal
array([[ 86.2,  90. ,  95. ],
       [100. ,  95. ,  95. ],
       [ 76.5, 100. ,  85. ],
       [ 89. ,  90. ,  95. ]])
```

Los arreglos al igual que otros objetos tipo secuencia tienen índices que inician en cero. El arreglo *cal* tiene dos índices ya que tiene dos dimensiones o ejes (*axes*) en terminología NumPy. Para leer una posición enviamos una tupla con los índices, dos en este caso (*renglón, columna*):

```
>>> cal[1,0]
100.0
>>> cal[(1,0)] # Podemos enviar un tupla
100.0
```

Veamos otros ejemplos:

```
>>> cal[:,0] # Todas las calificaciones de la tarea
array([ 86.2, 100. ,  76.5,  89. ])

>>> cal[:,0:2] # Tareas y exámenes
array([[ 86.2,  90. ],
       [100. ,  95. ],
       [ 76.5, 100. ],
       [ 89. ,  90. ]])

>>> cal[1:3,: ] # Calificaciones de Ana y María
array([[100. ,  95. ,  95. ],
       [ 76.5, 100. ,  85. ]])
```

Cuando enviamos una tupla con menos índices de los que tiene el arreglo, se considera queremos regresar todos los elementos de los ejes restantes:

```
>>> cal[0] # Equivalente a cal[0,:]
array([ 86.2,  90. ,  95. ])
```

Otra manera de indicar lo mismo es utilizando tres puntos (...).

```
>>> cal[0, ...]
array([ 86.2,  90. ,  95. ])
```

```
>>> cal[...]
array([[ 86.2,  90. ,  95. ],
       [100. ,  95. ,  95. ],
       [ 76.5, 100. ,  85. ],
       [ 89. ,  90. ,  95. ]])
```

```
>>> cal[...,2] # Tercera columna
array([ 95.,  95.,  85.,  95.]
```

Operaciones Básicas

Cuando utilizamos operaciones aritméticas sobre arreglos, la operación se realiza para cada elemento y se regresa un nuevo arreglo con el resultado. Vamos a suponer que debido al buen desempeño de todos los alumnos se subirá un punto a todas las calificaciones:

```
>>> cal + 1
array([[ 87.2,  91. ,  96. ],
       [101. ,  96. ,  96. ],
       [ 77.5, 101. ,  86. ],
       [ 90. ,  91. ,  96. ]])
```

También lo podríamos hacer solo para el primer alumno:

```
>>> cal[0] + 1
array([ 87.2,  91. ,  96. ])
```

En este caso no se ha modificado todavía el arreglo *cal*, solo estamos regresando una vista.

Las operaciones también se pueden hacer entre arreglos. Por ejemplo, vamos a suponer que tenemos un arreglo con los puntos extra que han obtenido los alumnos:

```
>>> puntos = [[0, 1, 0], [1, 0, 0], [0, 0, 0], [1, 2, 4]]
>>> extras = np.array(puntos)
>>> extras
array([[0, 1, 0],
       [1, 0, 0],
```

```

        [0, 0, 0],
        [1, 2, 4]])

>>> cal
array([[ 86.2,  90. ,  95. ],
       [100. ,  95. ,  95. ],
       [ 76.5, 100. ,  85. ],
       [ 89. ,  90. ,  95. ]])

>>> cal + extras
array([[ 86.2,  91. ,  95. ],
       [101. ,  95. ,  95. ],
       [ 76.5, 100. ,  85. ],
       [ 90. ,  92. ,  99. ]])

```

En este caso tenemos una calificación errónea ya que no se puede tener 101 de calificación.

Vamos ahora a suponer que deseamos ponderar cada una de las calificaciones, por ejemplo:

tarea	examen	proyecto
.30	.30	.40

NumPy permite realizar operaciones entre arreglos con distinta forma (shape) mediante un mecanismo que llaman *Broadcast*. En este caso se hace una búsqueda de una operación que tenga sentido. Por ejemplo:

```

>>> pesos = np.array([.30, .30, .40])
>>> cal * pesos
array([[ 25.86,  27. ,  38. ],
       [ 30. ,  28.5,  38. ],
       [ 22.95,  30. ,  34. ],
       [ 26.7 ,  27. ,  38. ]])

```

Implícitamente se completa la matriz de pesos para que tenga la misma forma que **cal** y después se hace la multiplicación como siempre, posición por posición.

Para saber la calificación final de cada alumno debemos sumar sus calificaciones ponderadas. Esto lo hacemos aplicando la función `suma` a los elementos del eje correspondiente (`axis=1`). Veamos:

```

>>> cal_ponderada = cal * pesos
>>> cal_ponderada.sum(axis=1)
array([ 90.86,  96.5 ,  86.95,  91.7 ])

```

El parámetro `axis=1` indica que se suman los renglones, si hubiéramos pasado `axis=0` se sumarían las columnas.

Para finalizar esta sección vamos a resolver un problema. ¿Como podríamos crear la una matriz de pesos que tenga la misma forma que *cal*?. Digamos algo así:

```
>>> pesos
array([[ 0.3,  0.3,  0.4],
       [ 0.3,  0.3,  0.4],
       [ 0.3,  0.3,  0.4],
       [ 0.3,  0.3,  0.4]])
```

Para resolver el problema debemos aprovechar las propiedades de los arreglos. Primero vamos a generar un arreglo con la forma de *cal* pero que contenga solo unos:

```
>>> unos = np.ones((4,3))
>>> unos
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Después simplemente multiplicamos este arreglo por el arreglo *pesos*:

```
>>> pesos = np.array([.30, .30, .40])
>>> unos * pesos
array([[ 0.3,  0.3,  0.4],
       [ 0.3,  0.3,  0.4],
       [ 0.3,  0.3,  0.4],
       [ 0.3,  0.3,  0.4]])
```

Para lograr el mismo objetivo, ¿Podríamos utilizar un arreglo con ceros *np.zeros((4,3))*?

Es importante recordar que estas operaciones no pueden realizarse en las listas convencionales de Python, por ejemplo:

```
>>> lista = [2,3,4]
>>> lista * 5
[2, 3, 4, 2, 3, 4, 2, 3, 4, 2, 3, 4, 2, 3, 4]

>>> np.array([2,3,4]) * 5
array([10, 15, 20])
```

Selección con arreglos de booleanos

Podemos enviar un arreglo de booleanos para seleccionar elementos de un arreglo. Por ejemplo, a las calificaciones iguales a 95 le vamos a sumar 5 punto:

Primero vamos a crear nuestro filtro:

```
>>> cal == 95
array([[False, False,  True],
       [False,  True,  True],
       [False, False, False],
       [False, False,  True]], dtype=bool)
>>> filtro = cal == 95
```

Si te fijas cada posición que cumple con la condición de `cal == 95` tiene `True`. Ahora vamos a modificar a aquellos elementos que cumplan con la condición:

```
>>> cal[filtro]
array([ 95.,  95.,  95.,  95.]) # Solo los valores que tienen True
>>> cal[filtro] += 5 # Le sumamos 5 al valor actual
>>> cal
array([[ 86.2,  90. , 100. ],
       [100. , 100. , 100. ],
       [ 76.5, 100. ,  85. ],
       [ 89. ,  90. , 100. ]])
>>>
```

Podemos utilizar esta técnica también para filtrar columnas por ejemplo:

```
>>> cal[:, np.array([False, True, True])]
array([[ 90., 100.],
       [100., 100.],
       [100.,  85.],
       [ 90., 100.]])
```

Como utilizamos un operador de asignación efectivamente estamos modificando al arreglo `cal`.

Iteración

En ocasiones necesitamos recorrer el arreglo haciendo alguna operación sobre los datos. ¡Cuidado! debemos tratar de utilizar lo menos posible el ciclo `for` para realizar operaciones con el arreglo. Por un lado debemos acostumbrarnos a un estilo de programación más funcional o estilo python (*pythonic*). Piensa siempre primero en como hacerlo utilizando comprensión de listas u operaciones a nivel de arreglo como vimos anteriormente.

Como última opción, cuando utilizamos ciclos en arreglos estos se hacen a partir del primer eje:

```
>>> for row in cal:
...     print(row)
...
[ 86.2  90.  100.]
[100. 100. 100.]
```

```
[ 76.5 100.  85. ]
[ 89.  90. 100.]
```

Podemos leer todos los elementos del arreglo secuencialmente con el atributo *flat* (plano):

```
>>> for item in cal.flat:
...     print (item)
...
86.2
90.0
100.0
100.0
100.0
100.0
76.5
100.0
85.0
89.0
90.0
100.0
```

Generación de arreglos con números secuenciales

En Python utilizamos la función *range()* para generar listas de números con secuencias de enteros. De manera similar en NumPy utilizamos *arange()* para generar arreglos de NumPy con valores secuenciales.

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> np.arange(0, 10, .33)
array([ 0.   ,  0.33,  0.66,  0.99,  1.32,  1.65,  1.98,  2.31,  2.64,
        2.97,  3.3 ,  3.63,  3.96,  4.29,  4.62,  4.95,  5.28,  5.61,
        5.94,  6.27,  6.6 ,  6.93,  7.26,  7.59,  7.92,  8.25,  8.58,
        8.91,  9.24,  9.57,  9.9 ])
```

En el segundo caso utilizamos un flotante para decirle a la función que nos genere números del 0 al 10, avanzando con pasos de 0.33. Un problema que podemos tener en este caso es que no sabemos cuantos elementos se van a generar y que tanto se acercará al valor final (en este caso 10). Para evitar esto, tenemos la función *linspace()* a la cual le decimos, dame 31 valores entre 0 y 10 inclusive:

```
>>> np.linspace(0, 10, 31)
array([ 0.   ,  0.33333333,  0.66666667,  1.   ,
        1.33333333,  1.66666667,  2.   ,  2.33333333,
        2.66666667,  3.   ,  3.33333333,  3.66666667,
        4.   ,  4.33333333,  4.66666667,  5.   ,
```

```

5.33333333, 5.66666667, 6.          , 6.33333333,
6.66666667, 7.          , 7.33333333, 7.66666667,
8.          , 8.33333333, 8.66666667, 9.          ,
9.33333333, 9.66666667, 10.         ])
```

De esta manera sabemos exactamente cuantos valores tendremos. Esta función se utiliza mucho para graficar funciones ya que nos genera los valores de x con los que vamos a evaluar la función.

Un ejemplo clásico es el siguiente:

```

>>> import matplotlib.pyplot as plt
>>> from numpy import pi
>>> x = np.linspace( 0, 2*pi, 100 ) # 100 números entre 0 y 2pi
>>> f = np.sin(x) # Creamos un arreglo con los resultados de sin(x)
>>> plt.plot(x,f) # Graficamos
>>> plt.show()
```

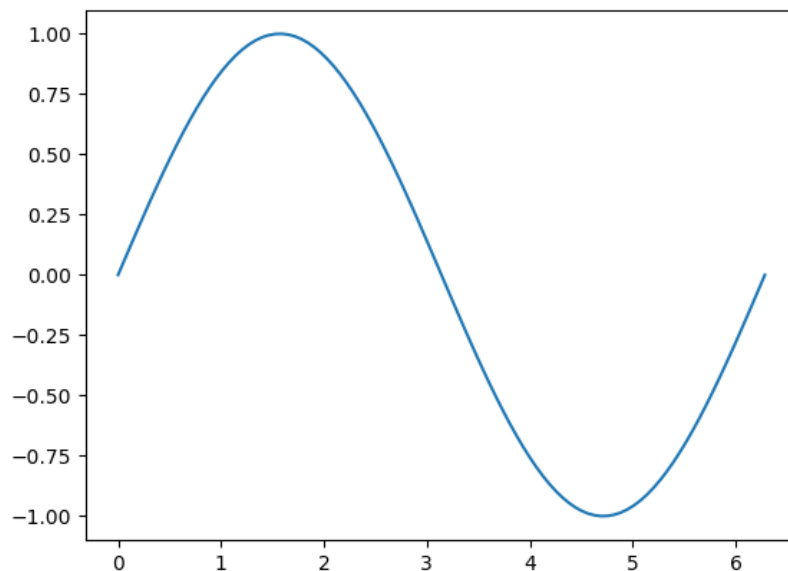


Figure 5: sin

Las funciones `linspace()` y `range()` generan arreglos de Dimensión-1, no te preocupes, podemos crear arreglos de mayor rango con la función de `reshape()`:

```

>>> np.linspace(0, 10,30).reshape(6,5)
array([[ 0.          ,  0.34482759,  0.68965517,  1.03448276,
        1.37931034],
```

```
[ 1.72413793, 2.06896552, 2.4137931 , 2.75862069,
 3.10344828],
[ 3.44827586, 3.79310345, 4.13793103, 4.48275862,
 4.82758621],
[ 5.17241379, 5.51724138, 5.86206897, 6.20689655,
 6.55172414],
[ 6.89655172, 7.24137931, 7.5862069 , 7.93103448,
 8.27586207],
[ 8.62068966, 8.96551724, 9.31034483, 9.65517241, 10.    ]])
```

Ejercicios

Lectura adicional

Bibliografía

Preprocesamiento

En esta etapa vamos a procesar los objetos crudos a representaciones favorables para los algoritmos que utilizaremos más adelante. También es conveniente el reducir en lo posible la cantidad de datos y los atributos de los objetos para reducir el costo computacional. Para lograr lo anterior, en la etapa de preprocesamiento realizar algunas de estas tareas:

- Estandarización
- Escalado
- Normalización
- Agregación
- Codificación de atributos categóricos
- Muestreo
- Reducción de la dimensión
- Creación de características
- Transformación de atributos

Estandarización

Muchos algoritmos de aprendizaje automático requieren que los datos estén normalizados para funcionar correctamente. Vamos a entender la razón por medio de un ejemplo. Vamos a observar de nuevo los datos de auto-mpg:

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> auto_mpg = pd.read_csv('datos-ejemplo/auto-mpg.data', sep='\s+',
```



```

header=None, na_values='?', names=['mpg','cylinders','displacement','horsepower',
'weight','acceleration','model_year','origin','car_name'],
dtype={'mpg':'f4', 'cylinders':'i4',
'displacement':'f4','horsepower':'f4','weight':'f4',
'acceleration':'f4','model_year':'i4','origin':'category',
'car_name':'category'})

>>> auto_mpg["origin"].cat.categories = ["USA", "Japan", "Germany"]
>>> auto_mpg.tail(3)
   mpg  cylinders  displacement  horsepower  weight  acceleration  \
395  32.0         4          135.0         84.0   2295.0          11.6
396  28.0         4          120.0         79.0   2625.0          18.6
397  31.0         4          119.0         82.0   2720.0          19.4

   model_year origin  car_name
395         82    USA  dodge rampage
396         82    USA   ford ranger
397         82    USA   chevy s-10

```

Nos vamos a concentrar en dos atributos: *weight* y *mpg*, vamos a graficar a los objetos:

```

>>> auto_mpg.plot.scatter(x='weight', y='mpg');
>>> plt.show()

```

Nos damos cuenta al observar la gráfica (o la tabla) que los valores numéricos son muy distintos, una diferencia de 20 unidades en el caso de mpg es mucha, mientras que en el peso no es así.

Ahora veamos como están distribuidos los valores utilizando un histograma:

```

>>> auto_mpg.hist(column='weight');
>>> plt.show()

>>> auto_mpg.hist(column='mpg');
>>> plt.show()

```

Si te fijas algunos valores se repiten más que otros, por ejemplo en el caso de las millas por galón vemos que un auto con 40 mpg es mucho menos probable que uno con 20 mpg. Si asumimos que los valores siguen una distribución normal podríamos calcular la probabilidad de tener 10, 30 o 40 mpg, si conocemos la media y la desviación estándar. Esto se simplifica bastante si los datos siguen una distribución normal estandarizada, es decir una distribución normal con una media de 0 y una desviación estándar de 1.

Para transformar los datos que tenemos solo debemos de utilizar la siguiente ecuación:

$$Z = (X - \mu) / \sigma$$

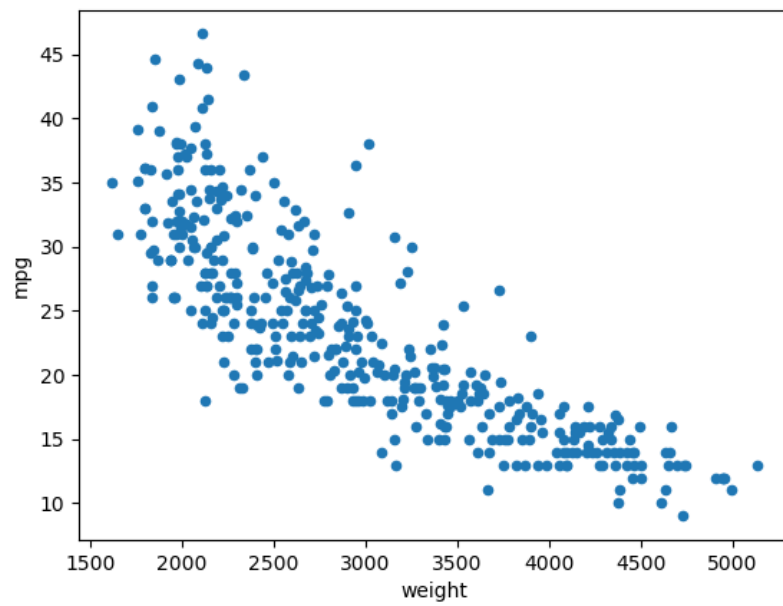


Figure 6: MPG Data

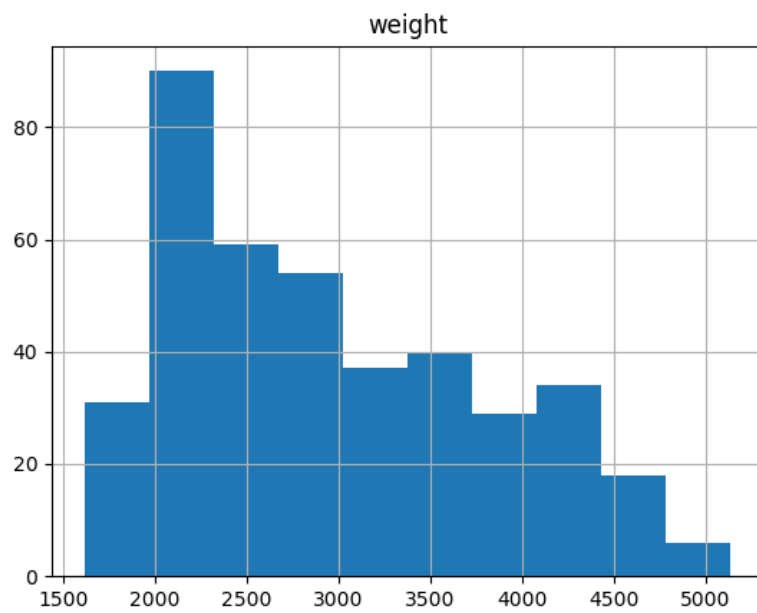


Figure 7: MPG Data

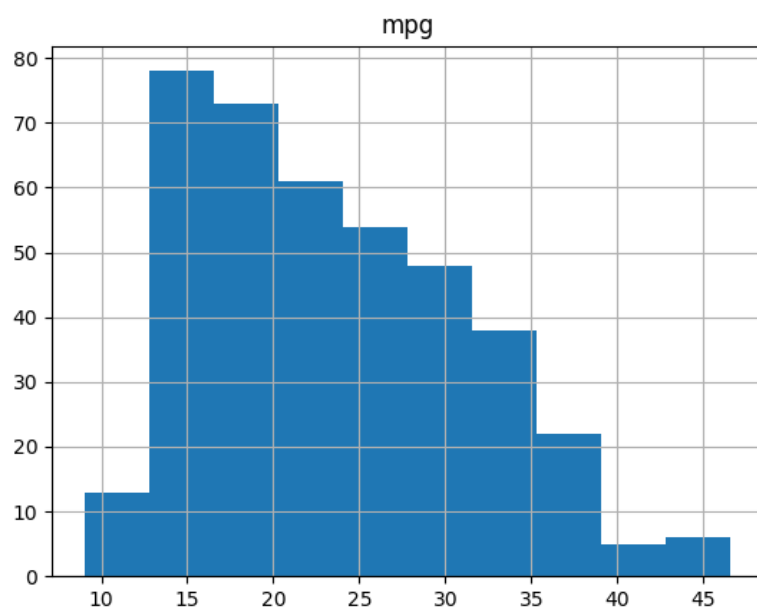


Figure 8: MPG Data

Es decir el *z-score* para un valor X es X menos la media() de X , dividido entre la desviación estándar() de X .

Podemos observar gráficamente esta distribución:

```
>>> import math
>>> import matplotlib.mlab as mlab

>>> mu = 0
>>> variance = 1
>>> sigma = math.sqrt(variance)
>>> x = np.linspace(mu - 3*sigma, mu + 3*sigma, 100)
>>> plt.plot(x, mlab.normpdf(x, mu, sigma))
>>> plt.show()
```

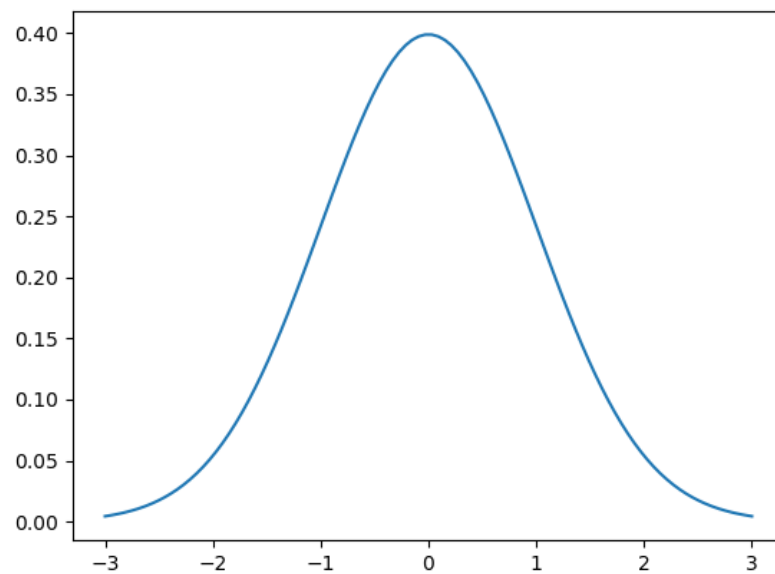


Figure 9: estandar

Podemos estandarizar los datos utilizando la biblioteca sci-kitlearn:

```
>>> from sklearn import preprocessing
>>> datos = auto_mpg.loc[:, ['mpg', 'weight']]
>>> datos_estandarizados = preprocessing.scale(datos)
>>> df = pd.DataFrame(datos_estandarizados)
>>> df.plot.scatter(x='weight', y='mpg');
>>> plt.show()
```

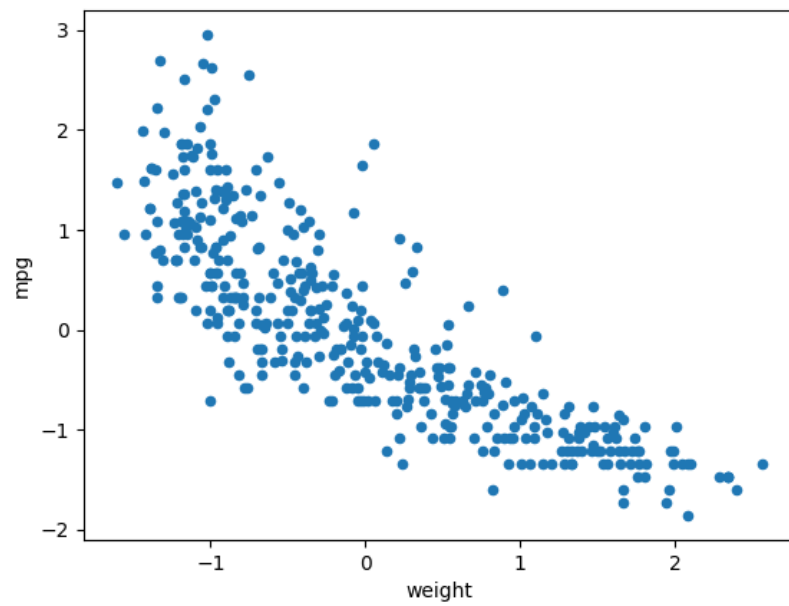


Figure 10: MPG Data

Vemos que ahora los valores numéricos no son tan distintos y en el caso del histograma de mpg:

```
>>> df.hist(column='mpg');  
>>> plt.show()
```

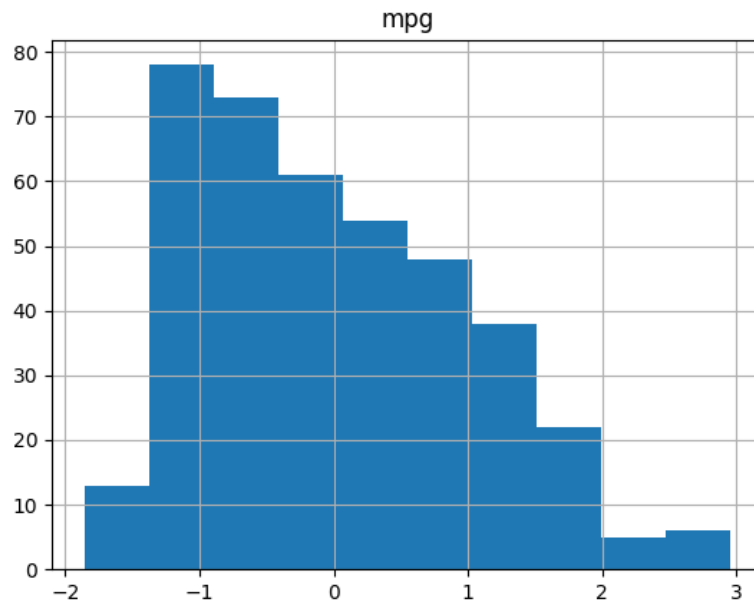


Figure 11: MPG Data

Nos damos cuenta que los datos siguen una distribución sesgada a la izquierda cuando la comparamos con la distribución normalizada anteriormente.

En muchos casos vamos a realizar este proceso en varios conjuntos de datos, por lo que el modulo *preprocessing* nos brinda la clase *StandardScaler* para calcular la media y desviación estándar en un conjunto de entrenamiento para después hacer la misma estandarización en el conjunto de datos de prueba.

Escalado a un rango

Otra transformación que podemos hacer para *weight* y *mpg*, es cambiar la escala de los valores a un rango entre cero y uno. Podemos hacerlo de la siguiente manera:

```
>>> datos = auto_mpg.loc[:, ['mpg', 'weight']]  
>>> min_max_scaler = preprocessing.MinMaxScaler()
```

```
>>> datos_minmax = min_max_scaler.fit_transform(datos)
>>> df = pd.DataFrame(datos_minmax, columns=['mpg', 'weight'] )
>>> df.plot.scatter(x='weight', y='mpg');
>>> plt.show()
```

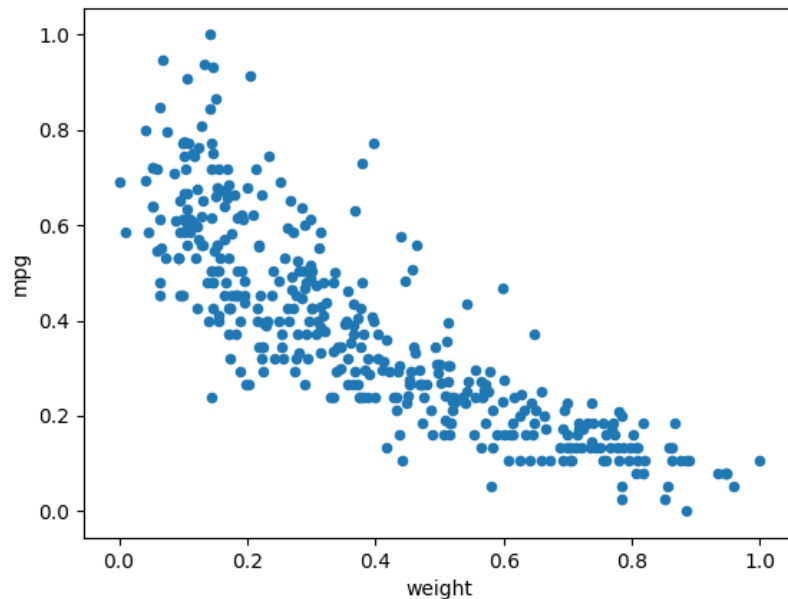


Figure 12: MPG Data

Para escalar a otros rangos enviamos una tupla en el constructor, por ejemplo:

```
>>> min = -1
>>> max = 1
>>> min_max_scaler = preprocessing.MinMaxScaler( feature_range=(min, max))
```

Escalado con valores atípicos

En caso de que los datos que estamos procesando incluyan valores atípicos, debemos utilizar una función de escalado que utilice una estadística que tolere a los *outliers*. Una estrategia es utilizar la mediana y escalar los datos de acuerdo a rangos establecidos por los percentiles. En el caso de el *RobustScaler* de scikit-learn por defecto se utiliza el rango entre el primer (percentil 25) y tercer (percentil 75) cuartiles.

```
>>> datos = auto_mpg.loc[ :, ['mpg', 'weight']]
>>> robust_scaler = preprocessing.RobustScaler()
```



```
>>> datos_robustos = robust_scaler.fit_transform(datos)
>>> df = pd.DataFrame(datos_robustos, columns=['mpg', 'weight'] )
>>> df.plot.scatter(x='weight', y='mpg');
<matplotlib.axes._subplots.AxesSubplot object at 0x11f5e6050>
>>> plt.show()
```

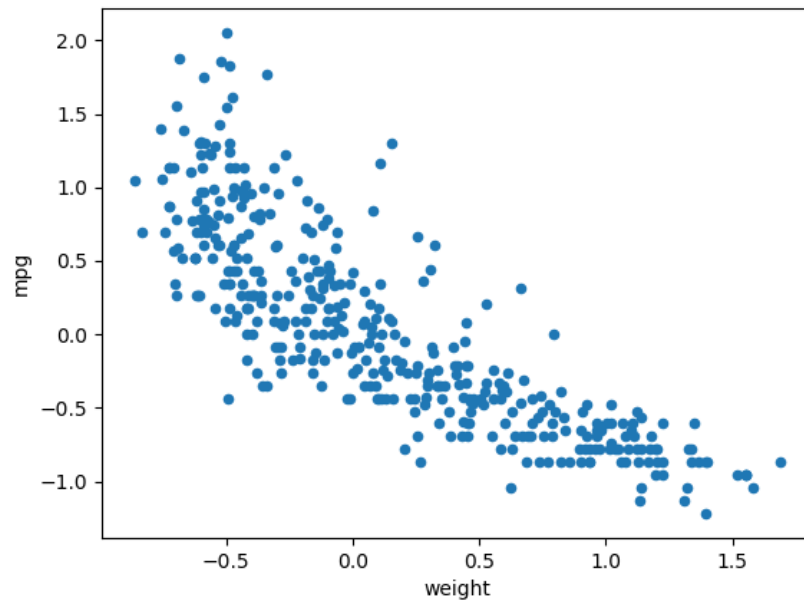


Figure 13: MPG Data

Normalización

Algunos algoritmos de clasificación de textos utilizan una representación de los documentos llamada modelo de espacio vectorial. En este modelo, cada documento es representado como un vector en el cual cada componente representa la ponderación de cierto término en el documento. Por ejemplo, veamos la representación de cuatro documentos:

	Home Run	Gol	Jugador	Penal	Tiro	Doble Play
d1	3	0	3	0	4	2
d2	0	5	7	3	2	0
d3	0	0	3	3	1	0
d4	5	0	6	0	2	3

	Home Run	Gol	Jugador	Penal	Tiro	Doble Play
d5	5	0	3	0	3	3

El Documento 1, habla de beisbol, por lo que no aparece en él, el término Gol ni Penal. Por otro lado, el Documento 2 habla de fútbol, por lo que no se menciona el término *Home Run*. Una manera de medir que tan parecidos o distintos son los documentos es simplemente calculando el producto punto entre los vectores, de tal manera que la similaridad está representada por el ángulo entre ellos.

Vamos a considerar un caso bien básico, que los documentos solo tiene dos términos: *Jugador* y *Tiro*. Al tener solo vectores con dos componentes, los podemos representar de la siguiente manera:

Los documentos menos similares son el d1 y d2, ya que el ángulo entre ellos es mayor. Los más similares son d3 y d4 ya que tienen el mismo ángulo. El cálculo de estos ángulos se simplifica si todos los vectores tienen una longitud unitaria ver. Por lo anterior es común normalizar o escalar los vectores a una longitud unitaria, dividiendo cada componente entre la longitud euclidiana del vector (a esto se le llama norma L2). Como vemos en la siguiente figura los vectores se han escalado considerando una longitud o norma euclidiana. Aunque cada documento sigue teniendo el mismo ángulo todos tienen una longitud de uno.

```
>>> from sklearn import preprocessing
>>> import numpy as np

>>> X = np.array([[ 3., 4.], [7., 2.], [3., 1.], [6., 2.], [3., 3.] ])
>>> X_normalized = preprocessing.normalize(X, norm='l2')

>>> X_normalized
array([[ 0.6          ,  0.8          ],
       [ 0.96152395,  0.27472113],
       [ 0.9486833 ,  0.31622777],
       [ 0.9486833 ,  0.31622777],
       [ 0.70710678,  0.70710678]])
```

En este ejemplo los documentos d3 y d4 tienen la misma dirección por lo que se consideran documentos iguales, lo comprobamos con el producto punto nos da 1.

```
>>> np.dot(X_normalized[2],X_normalized[3])
0.9999999999999999
```

En ocasiones también se utiliza una norma L1, también llamada distancia Man-

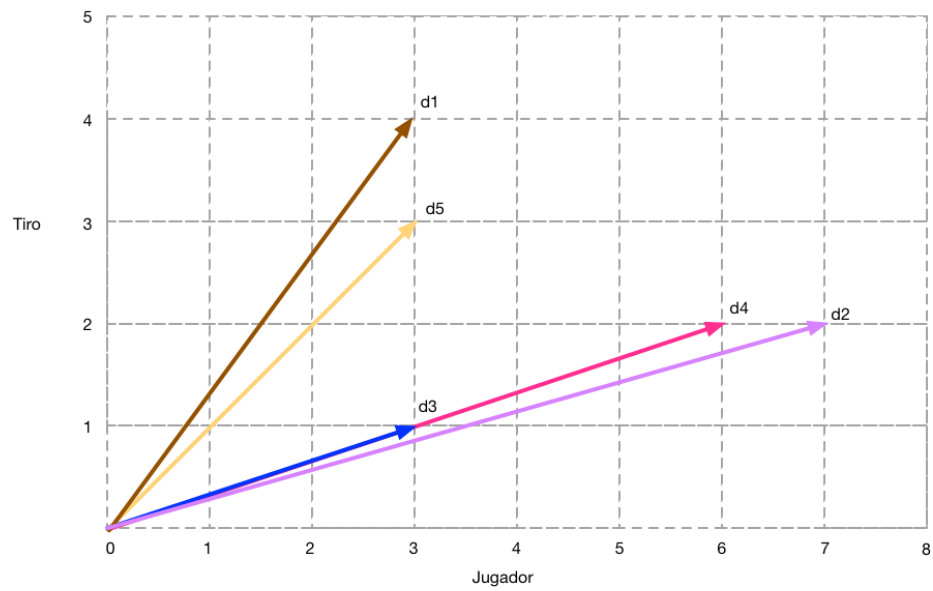
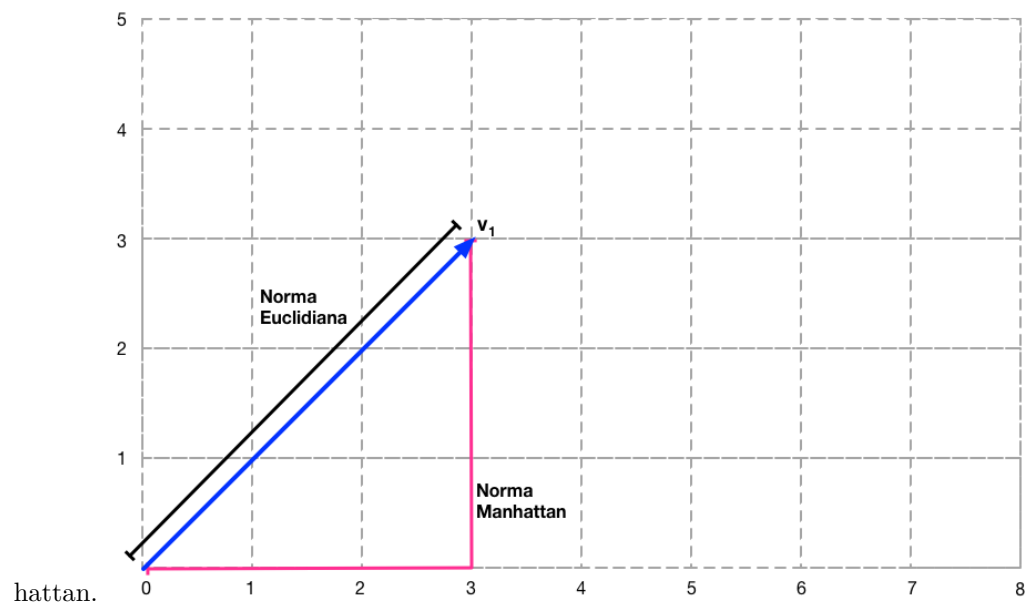


Figure 14: Vectores



```

hattan.
>>> X_manhattan = preprocessing.normalize(X, norm='l2')
>>> X_manhattan
array([[ 0.42857143,  0.57142857],
       [ 0.77777778,  0.22222222],

```

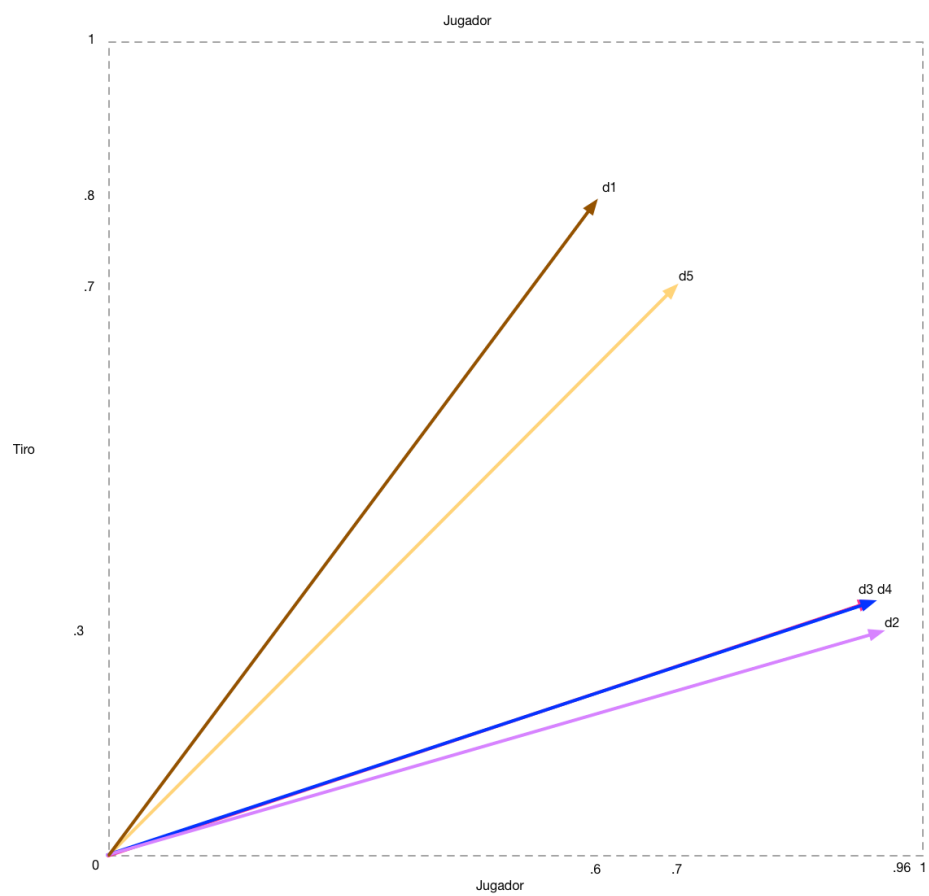


Figure 15: Norma

```
[ 0.75      , 0.25      ],
[ 0.75      , 0.25      ],
[ 0.5       , 0.5       ]])
```

Datos Agregados

Imagina que eres el ejecutivo de una cadena de librerías y tienes que tomar decisiones a cerca de los productos que se deben de promover en la siguiente temporada. Para apoyarte en la toma de decisiones utilizas un sistema OLAP el cual te permite ver los datos de las ventas con distintos nivel de detalle. Puedes ir desde el nivel de granularidad más bajo:

Producto	Cantidad	Tienda	Ciudad	Pais	Fecha	Mes	Año
El Principito	3	123	LA	USA	10/02/16	FEB	2016
El Principito	0	156	LA	USA	10/02/16	FEB	2016
El Hobbit	5	123	LA	USA	10/02/16	FEB	2016
El Hobbit	1	156	LA	USA	10/02/16	FEB	2016

Lo que vemos son las ventas diarias de cada libro en cada tienda. Como la información es muy detallada prefieres verla a otro nivel de detalle, por ejemplo:

Producto	Cantidad	Ciudad	Pais	Mes	Año
El Principito	10	LA	USA	ENE	2016
El Principito	30	LA	USA	FEB	2016
El Hobbit	30	LA	USA	ENE	2016
El Hobbit	30	LA	USA	FEB	2016

Esta vista te presenta los datos agregados por ciudad y mes. Agregar los datos consiste en hacer grupos que coincidan en determinado atributo y realizar alguna operación sobre el grupo, sumarlos en este caso. Podemos agregar todavía más:

Producto	Cantidad	Pais	Año
El Principito	29,203	México	2016
El Principito	29,788	USA	2016
El Hobbit	18,987	México	2017
El Hobbit	36,699	USA	2017

Dependiendo del tipo de análisis que estés haciendo será el tipo de agrupación que necesites. Como agregar reduce la cantidad de datos, tenemos las siguientes ventajas: * Si solo almacenamos los datos agregados, ahorramos espacio de

almacenamiento. * Se reduce el número de atributos. * Los datos agrupados tienen menor variabilidad.

Los datos agregados tienen mucha importancia en los sistemas de toma de decisiones, almacenes de datos y el concepto de cubos de datos. Para conocer más acerca de estos temas relacionados puedes revisar:

OLAP [Almacenes de Datos][Cubos de Datos] [Esquema Estrella]

- Codificación de atributos categóricos

Muchos algoritmos de aprendizaje automático solo pueden recibir como entrada vectores de características con valores continuos. ¿Qué hacemos con aquellos objetos que tienen atributos con valores categóricos?. Por ejemplo, una persona puede tener el atributo genero [“masculino”, “femenino”] o estado civil [“soltero”, “en una relación”, “casado”, “es complicado”, “divorciado”, “viudo”]. Una manera de común de solucionar este problema es simplemente asignarle un número entero a cada opción. Así, los valores de genero [“masculino”, “femenino”] los cambiamos a [0, 1] y las opciones de estado civil que acabo de decir, quedaría como [0, 1, 2, 3, 4, 5]. El problema que tiene esta solución es que se asume un orden, pero estos atributos no son ordinales; lo cual puede ser un problema en algunos casos.

Una manera sencilla de codificar los valores categóricos es utilizando una representación binaria. En esta representación se agrega un atributo por cada valor posible, así el atributo genero quedaría de la siguiente manera:

Original:

Nombre	Genero
Ana	Femenino
Tom	Masculino
Sue	Femenino
Jim	Masculino

Representación Binaria:

Nombre	Femenino	Masculino
Ana	1	0
Tom	0	1
Sue	1	0
Jim	0	1

Esta representación convierte cada atributo categórico con n valores posibles a n atributos binarios. Indicamos el valor categórico activando el atributo de la opción correspondiente; todos los otros atributos se desactivan. Para desactivar

o activar los valores se utiliza el cero y uno respectivamente. De esta manera el atributo estado civil visto anteriormente quedaría:

Nombre	Soltero	En una relación	Casado	Es complicado	Divorciado	Viudo
Ana	1	0	0	0	0	0
Tom	0	0	0	0	1	0
Sue	0	0	1	0	0	0
Jim	1	0	0	0	0	0

Vamos a practicar codificando el atributo origin de nuestro dataset *auto_mpg*. Recordemos que el atributo origin puede tener los valores categóricos: [0, 1, 2] que corresponden a ["USA", "Japan", "Germany"]. Vamos a utilizar el codificador *OneHotEncoder* de la biblioteca *sklearn*. Esta implementación solo puede codificar números enteros y no acepta valores nulos. Basándonos en el código visto anteriormente, primero importamos el módulo *preprocessing*. Después creamos un objeto codificador (enc) con el constructor *OneHotEncoder()*. Vamos a construir el codificador con el método *fit()*, a cual le enviamos como parámetro la columna *origin*. Ya una vez creado el codificador, podemos continuar pidiéndole que transforme la columna original. La transformación la guardamos temporalmente en *a*. La transformación es una arreglo escaso (sparse array), por consiguiente, si lo queremos desplegar debemos convertirlo a un array. En el ejercicio imprimimos el primer renglón, como vemos tiene tres columnas y la primera es 1, es decir: "USA".

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> from sklearn import preprocessing

>>> auto_mpg = pd.read_csv('auto-mpg.data', sep='\s+',
    header=None, na_values='?', names=['mpg', 'cylinders', 'displacement', 'horsepower',
    'weight', 'acceleration', 'model_year', 'origin', 'car_name'],
    dtype={'mpg': 'f4', 'cylinders': 'i4',
    'displacement': 'f4', 'horsepower': 'f4', 'weight': 'f4',
    'acceleration': 'f4', 'model_year': 'i4', 'origin': 'category',
    'car_name': 'category'})
>>> auto_mpg["origin"].cat.categories = ["USA", "Japan", "Germany"]

>>> enc = preprocessing.OneHotEncoder()
>>> enc.fit(auto_mpg[['origin']])
>>> a = enc.transform(auto_mpg[['origin']])
>>> a.toarray()[0]
>>> array([ 1.,  0.,  0.])
```

Muestreo

En estadística el muestreo es una herramienta de la investigación científica, que tiene como objetivo determinar la selección de una muestra de objetos de una población, que nos permita realizar inferencias sobre dicha población. Trabajamos con muestras, por que puede ser muy difícil o costoso analizar a todos los objetos de una población. Si la muestra es representativa, podemos reducir los costos llegando a las mismas conclusiones. En ocasiones simplemente no tenemos datos suficientes por la dificultad que representa obtenerlos. Cuando hablamos de minería de datos, parecería que tenemos el problema contrario, partimos de datos que se han recolectado en ocasiones masivamente, considerando objetos con un gran número de atributos. Debido a la gran cantidad de datos disponible, es necesario utilizar muestras para reducir el costo computacional del análisis. También se utilizan distintas técnicas de muestreo con el objetivo de evaluar los modelos o comparar distintos algoritmos de aprendizaje automático. Las técnicas de muestreo utilizadas en minería de datos se basan en las técnicas estándar de estadística. Algunas de las técnicas más utilizadas son el muestreo aleatorio simple o estratificado, con o sin reemplazo. Cuando se busca estimar el desempeño de los modelos, se utilizan técnicas de partición de datos, como leave-one-out y validación cruzada de k-particiones (k-fold cross validation). Veamos ejemplos de cada tipo.

Muestreo aleatorio simple

Este tipo de muestreo puede aplicarse cuando podemos calcular la probabilidad de selección de cada uno de los objetos. En el caso de minería de datos, nosotros tenemos mucho control sobre la selección de objetos que hacemos en datos electrónicos disponibles. Pero debemos tener cuidado, la validez de nuestras inferencias podrían no ser representativas de los objetos del mundo real; debido a un sesgo de origen, causado desde la misma recolección de los datos. El muestreo entonces es probabilístico, debido a que cualquiera de los objetos tiene una probabilidad de ser seleccionado. Los tipos de muestreo aleatorio simple son:

- Sin reemplazo: Cada objeto seleccionado se extrae de la población, por lo que ya no podrá ser seleccionado otra vez. Debemos considerar que cuando extraemos los objetos, el tamaño de la población y por lo tanto la probabilidad que tienen los objetos de ser seleccionados ira cambiando.
- Con reemplazo: Los objetos seleccionados se reinsertan a la población, un objeto puede ser seleccionado más de una vez y el tamaño de la población permanece constante.

Muestreo sistemático

En esta técnica se van removiendo objetos de una lista circular ordenada, se elige un punto de inicio al azar y utilizando intervalo correspondiente para el tamaño de la muestra, vamos dando saltos seleccionando a los demás objetos.

Muestreo estratificado

En ocasiones tenemos objetos que comparten cierta característica que nos permite agruparlos, pero se tiene el problema que los objetos de ese grupo son muy pocos comparados con la población en general. Por ejemplo los deportistas de Polo comparados con los de Fútbol. Si buscamos que todos los grupos estén representados en la muestra aleatoria, podemos hacer un muestreo estratificado. Para esto asignamos primero una cuota para cada grupo, por ejemplo 100 futbolistas y 2 jugadores de polo. Dentro de cada grupo o estrato utilizamos una técnica de muestreo sistemático.

Muestro por conglomerados (cluster sampling)

En ocasiones existen agrupamientos (o clusters) de objetos homogéneos en una población. Por ejemplo, grupos de estudiantes de alguna universidad con sedes en distintos estados; en general los grupos tienen características similares. Sin embargo, los elementos que constituyen cada agrupamiento (los estudiantes) deben ser diferentes entre sí; estudiantes de distintos semestres, carreras, calificaciones, etc. En estos casos se hace una selección aleatoria de los grupos. Por ejemplo, se eligen al azar un número de universidades, después en las universidades seleccionadas una muestra aleatoria de alumnos contestarán una encuesta. Esto puede ahorrar dinero a los investigadores, pues si hubieran elegido alumnos al azar deberían aplicar la encuesta en muchas universidades.

Muestreo aleatorio simple en Python

```
>>> import random
>>> pop = ["juan", "luis", "paco", "pedro", "jorge", "victor", "ana", "ada", "megan", "emma"]
>>> random.sample(pop, 3)
['juan', 'luis', 'emma']
>>> random.sample(range(100000000), 3)
[67307206, 72755273, 50751539]
>>>
```

En esta sección vamos a hacer un ejercicio práctico del proceso de KDD. Utilizaremos Python y algunas de las bibliotecas que utilizaremos a lo largo del libro. Al terminar el ejercicio tendrás una idea del tipo de programación que se requiere para el análisis de datos en Python.

Ejercicio Práctico ¿Setosa, Virginica o Versicolor?

Nota:

Para realizar los ejercicios debes tener previamente instalada la distribución Anaconda con python 2.7. En caso de que requieras ayuda en la instalación o alguna de las instrucciones, visita antes la sección [Python para Análisis de Datos].

Como primer ejercicio vamos a seguir el proceso de KDD para extraer conocimiento a partir del famoso conjunto de datos de flores Iris. El conjunto de datos incluso cuenta con su propia entrada en wikipedia. El conjunto de datos fue introducido por Ronald Fisher en un artículo en 1936. Contiene 50 muestras de tres especies de la flor Iris (Iris setosa, Iris virginica e Iris versicolor). Fisher midió cuatro características de cada muestra: el largo y ancho del sépalo y el largo y ancho del pétalo, en centímetros. Basado en la combinación de estos cuatro rasgos, Fisher se desarrolló un modelo discriminante lineal para distinguir entre una especie y otra. Como ejemplo veamos un fragmento que incluye varios registros de cada flor:

iris.data

```
sepal_length,sepal_width,petal_length,petal_width,species
5.1,3.5,1.4,0.2,setosa
4.9,3.0,1.4,0.2,setosa
4.7,3.2,1.3,0.2,setosa
4.6,3.1,1.5,0.2,setosa
7.0,3.2,4.7,1.4,versicolor
6.4,3.2,4.5,1.5,versicolor
6.9,3.1,4.9,1.5,versicolor
5.5,2.3,4.0,1.3,versicolor
6.5,2.8,4.6,1.5,versicolor
6.3,3.3,6.0,2.5,virginica
5.8,2.7,5.1,1.9,virginica
7.1,3.0,5.9,2.1,virginica
6.3,2.9,5.6,1.8,virginica
6.5,3.0,5.8,2.2,virginica
```

Para compartir los conjuntos de datos normalmente se utilizan archivos en formato texto. En este caso el *dataset* se encuentra en formato CSV (comma-separated values) con el cual es muy fácil representar datos tabulares. Los archivos CSV se pueden abrir e importar sin ningún problema a hojas de cálculo y sistemas de bases de datos. Como podemos ver en el ejemplo los registros están separados por saltos de línea, y en la primera se indica el nombre de la columna. En este caso cada registro cuenta con cinco campos. Los primeros cuatro son las lecturas correspondientes en centímetros y el último es muy importante ya que es la etiqueta o clase. En este caso en particular la etiqueta

indica el tipo de flor.

Paso 1: identificar el objetivo

El objetivo del proceso de KDD será encontrar algunos patrones que nos permitan clasificar las flores, o algún otro conocimiento nuevo.

Paso 2: seleccionar y recolectar

En este caso lo que debemos hacer es simplemente bajarnos el archivo *iris.csv* de alguna parte. Esto lo podemos hacer manualmente o directamente con Python.

Recolectar archivo iris.data

Esto lo vamos a hacer de manera interactiva desde el interprete. Para ejecutar el interprete simplemente escribimos python en la línea de comandos.

El interprete nos da la bienvenida:

```
Python 2.7.13 |Anaconda custom (x86_64)| (default, Dec 20 2016, 23:05:08)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>>
```

El archivo lo podemos bajar del Machine Learning Repository de la UCI (University of California Irvine). La URL directa es la siguiente: <https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data>. Me ha tocado algunas veces que no esta disponible el servidor, si esto sucede descárgalo de algún otro lado.

Para bajarlo utilizaremos la biblioteca de python requests. Esta biblioteca ya viene instalada en Anaconda. Solamente tenemos que importarla:

```
>>> import requests
```

Hacemos la petición *get* al archivo (esto puede tardar unos segundos):

```
>>> r = requests.get('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data')
```

Imprimimos el texto para ver que se haya bajado bien. Se debería ver esto:

```
>>> r.text
```

u'5.1,3.5,1.4,0.2,Iris-setosa\n4.9,3.0,1.4,0.2,Iris-setosa\n4.7,3.2,1.3,0.2,Iris-setosa\n4.6

¡Muy bien!, a lo que sigue.

Paso 3: Preprocesar los datos.

Una cadena enorme contiene todos los datos. Esto no nos sirven de mucho. El objetivo de este paso será procesar la cadena para tener como resultado un arreglo multidimensional de NumPy [QuickStart] (<https://docs.scipy.org/doc/numpy-dev/user/quickstart.html>). Este tipo de arreglos nos permite leer secciones a lo largo y ancho lo cual será muy útil en nuestro análisis. Más adelante veremos otras bibliotecas que simplificarán aun más este proceso como la biblioteca de análisis de datos Pandas.

Empezaremos por crear una lista a partir de una separación de la cadena. Esto lo hacemos con el método `split()` separando la cadena por los saltos de línea (`'\n'`):

```
>>> renglones_iris = r.text.split('\n')
```

Revisemos un poco el contenido de la lista:

```
>>> renglones iris[:5]
```

```
[u'5.1,3.5,1.4,0.2,Iris-setosa', u'4.9,3.0,1.4,0.2,Iris-setosa', u'4.7,3.2,1.3,0.2,Iris-setosa',
```

Podemos ver un detalle importante al final de la lista: los últimos dos renglones están vacíos:

```
>>> renglones_iris[-4:]
```

```
[u'6.2,3.4,5.4,2.3,Iris-virginica', u'5.9,3.0,5.1,1.8,Iris-virginica', u'', u'']
```

Observamos también que cada uno de los elementos de la lista a su vez está separado por comas y no necesitamos el último dato ya que queremos solo las medidas en centímetros. Lo que queremos entonces es crear una nueva lista a partir de `renglones_iris` que tenga como elementos a su vez listas con las medidas. Para esto utilizaremos comprensión de listas. Primero vamos a hacer algunas pruebas:

```
>>> [ renglon.split(',')[:-1] for renglon in renglones_iris[:-2]]
```

[[u'5.1', u'3.5', u'1.4', u'0.2'], [u'4.9', u'3.0', u'1.4', u'0.2'], [u'4.7', u'3.2', u'1.3

Lo que estamos haciendo es lo siguiente: a partir de la lista `renglones_iris` menos sus últimos dos elementos (`renglones_iris[:-2]`) vamos a generar una lista nueva procesando cada uno de sus elementos. Cada una de las cadenas en `renglones_iris` la vamos a separar por las comas y nos quedaremos con los primeros cuatro elementos (`renglon.split(',')[:-1]`). Ya casi tenemos lo que necesitamos. Solamente faltaría convertir cada elemento a tipo flotante ya que como vemos son todas cadenas. Esto lo podemos hacer utilizando la función `map()` la cual aplica una función a cada elemento de la lista regresando una nueva. En este caso queremos utilizar la función `float()`. Veamos:

```
>>> [map(float, renglon.split(',')[:-1]) for renglon in renglones_iris[:-2]]
```

`[[5.1, 3.5, 1.4, 0.2], [4.9, 3.0, 1.4, 0.2], [4.7, 3.2, 1.3, 0.2], [4.6, 3.1, 1.5, 0.2], [5.`

Ya que obtuvimos el código paso a paso, vamos a utilizarlo para crear el nuevo arreglo:

```
>>> import numpy
>>> iris_data = [map(float, renglon.split(',')[::-1]) for renglon in renglones_iris[:-2]]
>>> iris = numpy.array(iris_data)
```

Listo ya tenemos nuestros datos en una estructura adecuada:

```
>>> iris[:10]
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  1.4,  0.2],
       [ 5.4,  3.9,  1.7,  0.4],
       [ 4.6,  3.4,  1.4,  0.3],
       [ 5. ,  3.4,  1.5,  0.2],
       [ 4.4,  2.9,  1.4,  0.2],
       [ 4.9,  3.1,  1.5,  0.1]])
```

Paso 4: Transformar los datos

En este caso sencillo no será necesario transformar los datos.

Paso 5: Seleccionar la tarea de minería de datos

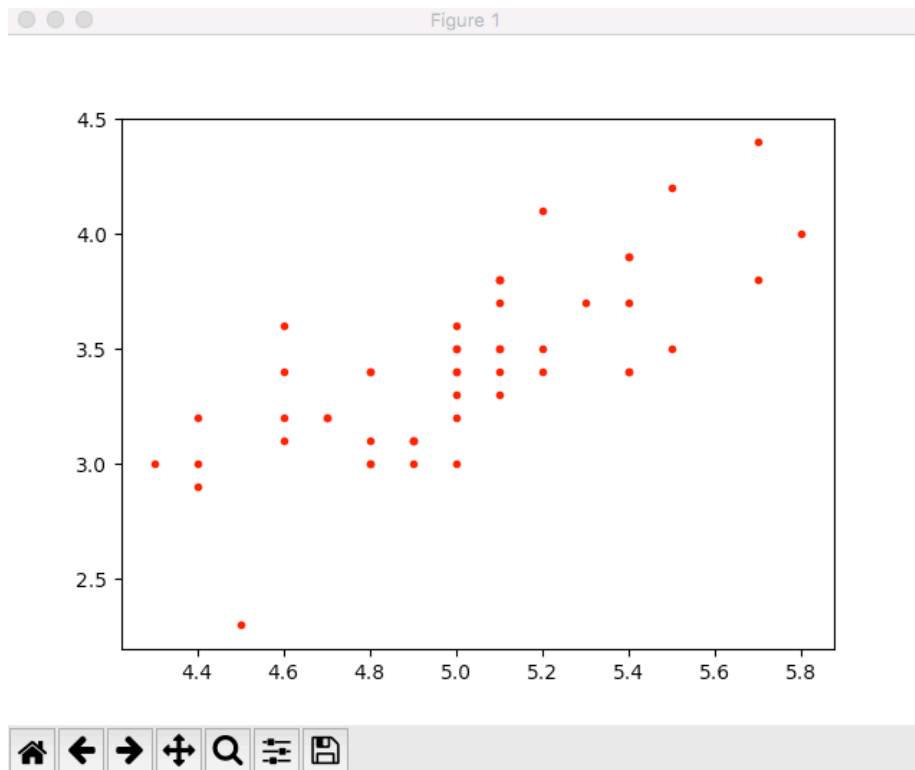
La tarea que vamos a realizar será *Clasificación*.

Paso 6: Análisis exploratorio

La minería de datos en este primer ejercicio la vamos a realizar manualmente, para esto vamos a explorar los datos visualmente. Para ello utilizaremos la popular biblioteca matplotlib. Primero vamos a seleccionar las dos primeras características: ancho y largo del sépalo para ver si hay algún patrón útil. Importamos matplotlib y hacemos el plot de la primera flor. Los primeros cincuenta datos son de Iris Setosa:

```
>>> import matplotlib.pyplot as plt
>>> x = iris[:50,0]
>>> y = iris[:50,1]
>>> plt.plot(x, y, 'r.')
>>> plt.show()
```

Al ejecutar la instrucción `plt.show()` se debería mostrar lo siguiente:

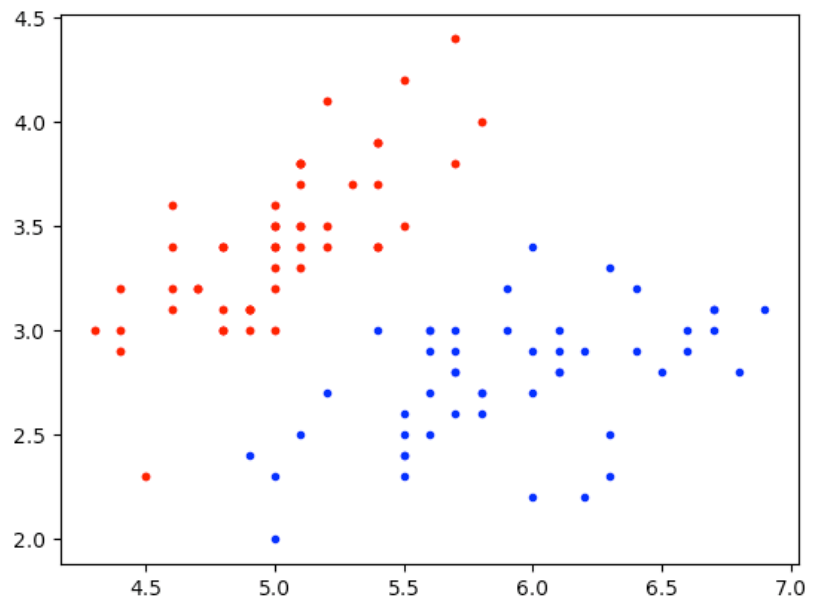


En el eje **x** tenemos el ancho del sépalo Y en el eje **y** el largo, ambos en cm. El parámetro 'r.' indica que las flores Setosa se representarán por puntos rojos ('r.' red dots).

Ahora graficaremos al mismo tiempo las flores Setosa y Versicolor:

```
>>> plt.plot( iris[:50,0], iris[:50,1], 'r.') # Setosa
[<matplotlib.lines.Line2D object at 0x1066ed610>]
>>> plt.plot( iris[51:100,0], iris[51:100,1], 'b.')
[<matplotlib.lines.Line2D object at 0x113731710>] # Virginica
>>> plt.show()
```

Descubrimos algo, es posible separar linealmente o clasificar ambas flores uti-



lizando estas dos características:

```
>>> plt.plot( iris[51:100,0], iris[51:100,1], 'b.')
[<matplotlib.lines.Line2D object at 0x114cae950>]
>>> plt.plot( iris[101:,0], iris[101:,1], 'g.')
[<matplotlib.lines.Line2D object at 0x114cbc110>]
>>> plt.show()
```

Veremos si corremos con igual suerte al agregar la Versicolor:

```
[<matplotlib.lines.Line2D object at 0x114cae850>]
>>> plt.plot( iris[51:100,0], iris[51:100,1], 'b.')
[<matplotlib.lines.Line2D object at 0x114cae950>]
>>> plt.plot( iris[101:,0], iris[101:,1], 'g.')
[<matplotlib.lines.Line2D object at 0x114cbc110>]
>>> plt.show()
```

Por lo menos al considerar estas dos características vemos que es difícil distinguir entre las flores Virginica y Versicolor.

Paso 7: Minería de Datos

Este paso lo vamos a hacer manualmente por lo pronto. ¿Como podríamos especificar el modelo?. Una manera muy sencilla sería la siguiente:

Utilizando una recta para separar a las flores Setosa del resto. Podríamos además utilizar varias rectas o incluso funciones no lineales. Estas ideas las

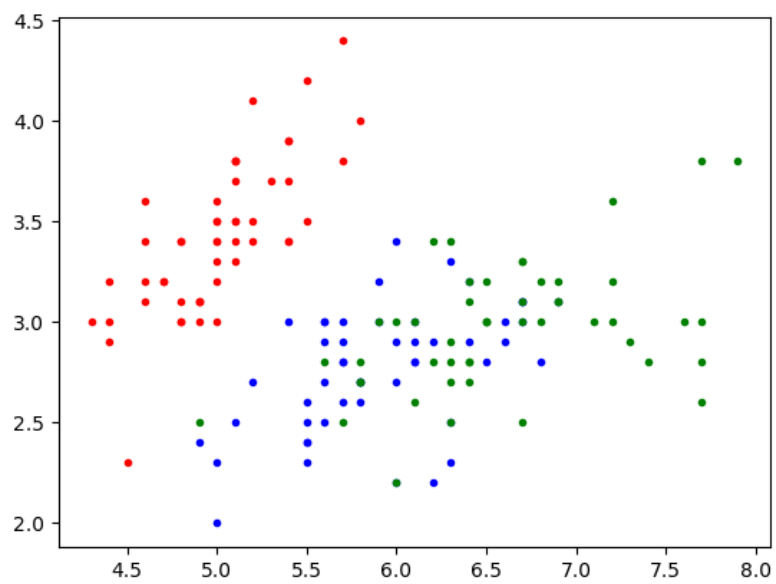


Figure 16: Setosa, Virginica, Versicolor

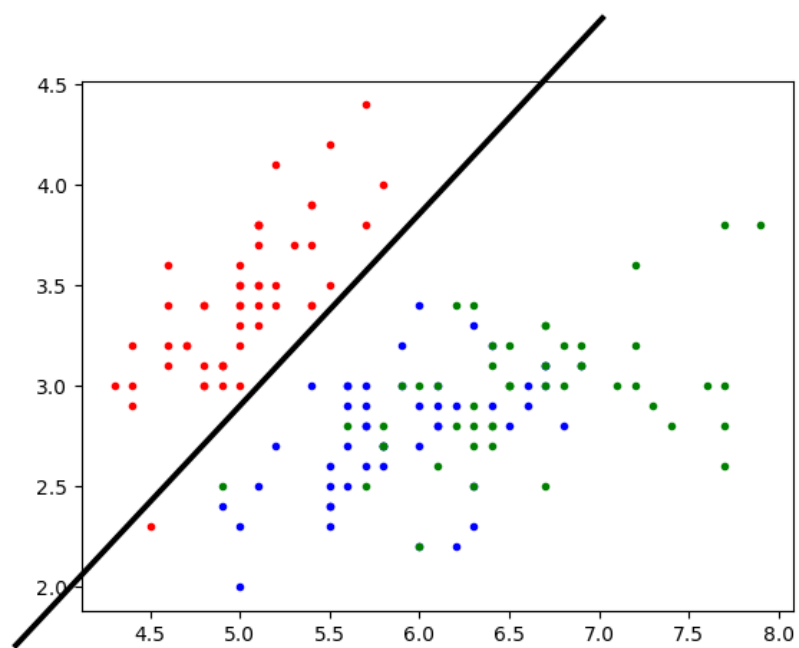


Figure 17: Setosa, Virginica, Versicolor

podremos llevar acabo “manualmente” ya que solo estamos considerando dos características. Esto se puede tornar más difícil al considerar las otras dos medidas pues estaríamos trabajando en dimensión cuatro.

Otra modelo podría ser expresado en forma de reglas:

R1:

SI sepal_length < 5.9 AND sepal_width > 2.9

ENTONCES:

Setosa

R2:

SI sepal_length < 4.7 AND sepal_width <= 2.9

ENTONCES:

Setosa

Gráficamente sería algo como:

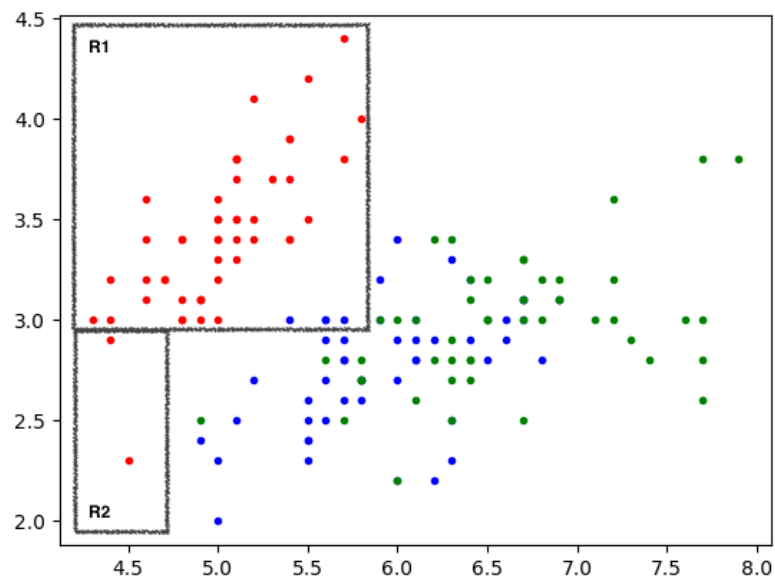
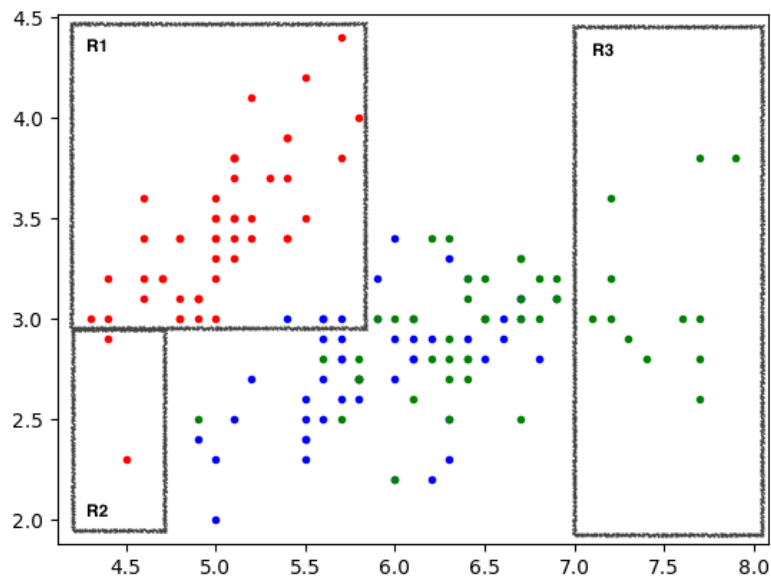


Figure 18: Setosa, Virginica, Versicolor

Ya que estamos en esto podemos proponer un nuevo modelo simplemente agre-



gando otra regla:

Podríamos seguir buscando, por ejemplo cambiando un poco los antecedentes de las reglas, por ejemplo de `sepal_length < 5.9` a `sepal_length < 6.0`. Nos empezamos a dar cuenta que no es fácil hacer esto manualmente. Mejor hagamos programas que hagan este trabajo de generar modelos automáticamente. Es decir, algoritmos de aprendizaje automático.

Paso 8. Interpretar y evaluar

Esto lo haremos más adelante, pero vale la pena pensar un poco al respecto. ¿Cual es mejor modelo?, ¿El mejor es suficientemente bueno?, ¿Existe un modelo óptimo?. ¿Que pasará cuando agreguemos nuevas flores al conjunto de datos?. También debemos pensar si pudimos extraer algún conocimiento nuevo. ¿Es útil?.

Paso 9. Aplicar el conocimiento adquirido

De este pequeño ejercicio podríamos reportar que las flores Setosa son fáciles de identificar.

Este ejercicio es básico y además no hicimos todas las consideraciones. Más adelante veremos otros detalles que no hemos considerado y por supuesto ya no haremos la minería de datos manualmente. El ejercicio también ha servido para darnos idea de como trabajaremos con Python y sus bibliotecas para este

tipo de tareas. Seguro te diste cuenta que no hemos utilizado ciclos, enviamos funciones como parámetros y trabajamos de forma interactiva para llegar a la solución.

Atención estudiantes:

Si te fijas el proceso de KDD también puede ser utilizado como guía para hacer proyectos de investigación en el área. El proceso es el mismo pero cambian los algoritmos, aplicaciones y tipos de datos.

Calidad de los Datos

Uno de los principios más antiguos de la informática nos dice “entra basura, sale basura” y se refiere al hecho de que si un programa recibe como entrada datos incorrectos o sin sentido en la salida tendremos lo mismo. Este principio también se aplica a la ciencia de datos pues el éxito del proceso depende de la validez de los datos que utilizamos. El asegurar la calidad de los datos que recolectamos es uno de los mayores retos de la minería de datos, por esto es muy importante realizar un proceso previo de verificación y *limpieza de datos*. Para darnos una idea de la complejidad de estas tareas, vamos a considerar la recolección de datos para el desarrollo de un Observatorio de Lesiones. El objetivo de estos sistemas según la secretaría de salud es “... generar datos y evidencia científica para la prevención de lesiones ocasionadas por accidentes viales”. Es indispensable para este tipo de sistemas contar con información confiable sobre los accidentes de tráfico, pero como veremos esto no es nada fácil. Para empezar, atender un accidente involucra varias organizaciones, cruz roja, paramédicos, policía, hospitales, call-centers de emergencia, rescatistas, agencias de seguros entre otras. Por otro lado, se involucran transeúntes, vehículos, pasajeros, edificios, reporteros etc. Para obtener información sobre lo sucedido normalmente se debe entrevistar a los testigos e involucrados. Desde aquí empieza el problema, ya que cada persona tiene su versión de los hechos, puede cambiarla o estar incapacitada para darla. Cada dependencia tiene la urgencia de recolectar solo la información necesaria para cumplir con su propósito y cada una tiene una versión parcial de los hechos. El accidente puede relacionarse con otros eventos posteriores, por ejemplo una persona puede sufrir secuelas del accidente mucho tiempo después. Entonces, la tarea de integrar los datos requiere la cooperación entre distintas dependencias lo cual nos da un primer golpe de realidad: **En muchas ocasiones no esta en nuestras manos la calidad de los datos, ni tendremos control sobre la fuente de los mismos.** Por lo anterior la minería de datos se enfoca solamente en dos tareas básicas [ref Tan, Kumar]: (1) la detección y corrección de problemas en los datos y (2) la utilización de algoritmos que puedan tolerar datos de mala calidad. Dicho de otra manera, no es nuestro objetivo ni la gestión ni el control de la calidad de los datos, ya que esto involucraría intervenir en los mismos sistemas de recolección.

Decimos que un producto o servicio es de buena calidad si satisface nuestras necesidades. En el caso de los datos, debemos asegurarnos de que sirvan para nuestros propósitos. Por ejemplo, una tarea de nuestro observatorio podría ser el optimizar la ubicación de estaciones de rescate. Entre los datos requeridos para este objetivo se debe incluir la ubicación exacta de los accidentes. Aquellos registros que solo incluyan la zona o la calle del accidente no tendrían calidad para esta tarea, pero podrían ser de utilidad o tener buena calidad para otros propósitos; a continuación revisaremos algunos problemas que afectan la calidad de los datos.

Fuentes de datos poco confiables

Muchas de las colecciones de datos disponibles, no se recolectaron originalmente para realizar una investigación científica, por lo que varían mucho en su autenticidad y calidad. Jianzheng Liu et al.(2015) identificaron tres diferencias entre la recolección de datos realizada por una empresa comercial y una institución de investigación científica:

1. Las empresas no adoptan procedimientos científicos de recolección de datos, como muestreos aleatorios o procedimientos para evitar el sesgo. Las empresas recolectan información con el objetivo de producir ganancias no para hacer ciencia. Normalmente el muestreo de datos se realiza sobre la población objetivo de las empresas y no sobre el público en general.
2. Los métodos de recolección de datos no son públicos y pueden cambiar sin aviso alguno.
3. Las plataformas comerciales no se pueden hacer responsables de la autenticidad ni la validez de los datos que coleccionan. Por ejemplo, al utilizar datos de una red social como Twitter debemos considerar a las cuentas de *Bots*, usuarios pagados para producir contenido, anuncios y publicaciones tipo *Spam*.

Como podemos ver, la calidad de los datos no solo se refiere a la medición o la captura en sí, también debemos tener en cuenta el procedimiento utilizado en la recolección.

Errores de medición y recolección de datos

Llamamos errores de medición a aquellos problemas que suceden en el proceso de medición o captura. En el caso de los atributos continuos, llamaremos error a la diferencia numérica entre el valor real y el medido. Un error de medición se puede dar también cuando un sensor no tiene la precisión adecuada e incluso cuando la medición se trunca al almacenarse en una variable que no tiene la escala o precisión adecuada.

Un error de recolección de datos se refiere a problemas de captura al nivel de objetos o atributos. Por ejemplo, cuando omitimos el valor de un atributo o capturamos un objeto en una categoría equivocada.

Valores desconocidos

En muchas ocasiones el valor de algunos de los atributos no se conoce. Por ejemplo, en el caso de un accidente pudo no haberse recolectado la la edad del conductor, el número de pasajeros o la información meteorológica. Normalmente se indica la ausencia de valor utilizando alguna palabra reservada o etiqueta como NULL, None o Nil. Necesitamos determinar como vamos a tratar los casos en los que desconocemos el valor. Algunas de las alternativas son:

- *Estimar el valor* Esto puede ser peligroso ya que el valor podría ni siquiera estar en el conjunto de datos.
- *Eliminar el objeto* Algunos algoritmos no pueden procesar objetos con valores desconocidos o simplemente determinamos que el objeto en esta condición no aporta información. El objeto se puede eliminar por completo o lo podemos omitir en ciertos casos.
- *Utilizar un valor por defecto*
- *Agregar objetos con todos los valores posibles* Incluso se pueden ponderar los valores de acuerdo a su probabilidad de aparecer.

Valores atípicos



Figure 19: Mercedes

Los atributos de los objetos en ocasiones tienen valores muy distintos o alejados numéricamente del resto de los datos. A estos valores les llamamos atípicos o es común utilizar su nombre en inglés *outliers*. Estos valores en ocasiones se deben a errores en la lectura o en los procedimientos y deben eliminarse, pero también pueden suceder por casualidad en cualquier distribución de probabilidad.

Los conjuntos de datos que incluyen valores atípicos, pueden engañarnos al momento de describir los datos estadísticamente. Por ejemplo, si medimos los caballos de fuerza promedio de 8 autos en un estacionamiento, la mayoría tendría

entre 50 y 170 hp, pero si de suerte hay un *Mercedes F1 W06 Hybrid* con 950 hp, este sería un claro *outlier*.

Vamos a tomar como ejemplo una muestra de auto-mpg.data:

hp	nombre
91.00	“audi 100ls”
115.0	“saab 99le”
53.00	“honda civic”
180.0	“pontiac grand prix lj”
140.0	“dodge diplomat”
125.0	“cadillac eldorado”
950.0	“Mercedes F1 W06 Hybrid”

En este caso la mediana sería 125 y la media 236. Debido al valor atípico, la mediana refleja mejor la potencia de un auto si lo seleccionamos al azar de un estacionamiento. Los valores atípicos pueden alertarnos de objetos que pertenecen a otra población, en este caso nuestro Mercedes es propio de una escudería de Formula 1.

Una de las tareas de limpieza es el identificar objetos atípicos para removerlos, pero hay casos en los que el encontrar a estos objetos es el objetivo principal de todo el proceso. Un objeto *outlier* podría ser una transacción fraudulenta o una persona de interés.

El tema de los valores atípicos lo seguiremos durante todo el proceso ya que incluso el desempeño de un algoritmo puede ser atípico. Podríamos estar muy felices por un experimento que nos dio muy buenos resultados, solo para darnos cuenta al hacer más pruebas que el desempeño esperado es mucho menor.

Datos con Ruido

En algunas publicaciones llaman *ruido* a las imperfecciones en los datos. Tal como sucede cuando hay interferencia en la señal de radio y escuchamos la música con ruido. El ruido en los datos son precisamente los errores de medición, recolección, valores atípicos e inconsistencias en los datos. En ocasiones es necesario perturbar o agregar ruido a los datos ya sea para conservar la privacidad de la fuente o para probar que tan robusto es un algoritmo. Para agregar ruido a un conjunto de datos normalmente se agregan componentes aleatorios. Por ejemplo, se pueden agregar nuevos objetos con atributos aleatorios

Visualización

Para darnos una mejor idea de la naturaleza de los fenómenos capturados por los datos debemos explorarlos. En esta sección vamos a conocer algunas de las técnicas más populares para explorar los datos.

Como primera herramienta tenemos a la estadística descriptiva, la cual nos permite organizar, presentar y describir a los conjuntos de datos. También contamos con herramientas de apoyo como las gráficas, tablas e incluso animaciones las cuales complementan a las medidas numéricas. Un herramienta muy importante es la visualización. Esta técnica de análisis nos permite convertir los datos a formatos visuales, para resaltar las características o relaciones que existen entre los distintos objetos y atributos.

Un ejemplo muy importante que resalta la necesidad de visualizar los datos es el ejercicio realizado por X. El ejercicio nos presenta varios conjuntos de datos lo cuales tienen las mismas medidas estadísticas pero con muy distintas gráficas. Para mayor información visita la página del proyecto.

Antes de elegir las herramientas de análisis que vamos a utilizar o incluso el tipo de preprocesamiento que haremos, es recomendable que hagamos una exploración preliminar de los datos. No hay que olvidar que nosotros como humanos somos excelentes para reconocer patrones, así que antes de utilizar toda la maquinaria de análisis vale la pena que echemos un vistazo. Podemos encontrar patrones los cuales nos ayuden a seleccionar las herramientas para el análisis posterior.

En esta sección nos enfocaremos en los siguientes temas:

- Parámetros estadísticos
- Visualización
- Herramientas OLAP (Online Analytical Processing)

Parámetros estadísticos

Cuando analizamos una gran cantidad de objetos, por ejemplo los alumnos de una escuela o la población de un país, muchas veces utilizamos números que nos sirven para resumir alguna característica de dichos objetos. Por ejemplo, si comparamos las edades de ciertos habitantes de dos países podríamos hablar de la edad promedio de uno y otro país. En este caso decimos que la edad es una variable estadística y la media aritmética o promedio de edad es un parámetro estadístico. Podemos decir que el parámetro o número se obtiene realizando un cálculo sobre el conjunto de datos. Como hemos visto anteriormente el tipo de cálculo que podemos hacer sobre las variables estadísticas dependerá del tipo de medición. Recordemos que hay operaciones que pueden hacerse con valores continuos pero no con valores categóricos.

Medidas de posición.

Estos parámetros tienen cierta posición dentro del rango de valores que encontramos en el conjunto de datos.

Media

La media o media aritmética es uno de los parámetros estadísticos más utilizados y se calcula muy fácilmente, simplemente se suman todos los valores y esta suma se divide entre el número de ellos.

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

Cuando los datos son muy variados o existen valores atípicos la media realmente no nos dice mucho acerca de los datos. Por ejemplo si un grupo de personas tuviera las siguientes edades: [10, 10, 90, 10, 10] la media aritmética se calcularía $(10+10+90+10+10)/5$ lo que nos arroja 26 años. En cambio para las edades: [12, 11, 9, 10, 10] la media de 10.4 años es mucho más representativa. La mediana es una mejor opción cuando tenemos casos atípicos.

Mediana

Para calcular la mediana primero ordenamos a los valores de menor a mayor y el resultado es el valor que está a la mitad. En caso de que el número de elementos sea impar el valor en el centro es el resultado en otro caso es el promedio de los dos valores centrales. Por ejemplo, para las edades: [10, 10, 90, 10, 10], primero ordenamos: [10, 10, 10, 10, 90] y como el número de elementos es impar (5) el valor de 10 es la mediana: [10, 10, 10, 10, 90]. En caso de que sea un número de elementos par: [10, 12, 24, 90] regresamos el promedio de los valores (12, 24), en este caso la mediana es 18. El cálculo de la media lo podemos expresar de la siguiente manera:

Sea n el número de elementos ordenados en orden creciente: $x_1, x_2, x_3, \dots, x_n$

Si n es par: el valor x_p en la posición $(n+1)/2$.

Si n es impar: la media aritmética de los valores que ocupan la posiciones centrales en las posiciones: $n/2$ y $(n/2) + 1$.

Moda

En el caso de variables cualitativas no podemos calcular los parámetros anteriores, podemos sin embargo calcular la frecuencia con que se repiten los valores. La moda es simplemente es el valor con mayor frecuencia en una distribución de datos.

Medidas de dispersión.

Estas medidas nos dan un resumen que tan distintos son de los valores.

Varianza

Si queremos ver que tanto varían los valores de una variable en un conjunto de datos podemos a considerar a la media aritmética como el punto de referencia. Si tomo un valor y calculo la diferencia con respecto a la media veo que tan diferente es. Por ejemplo, vamos a suponer las edades [12, 11, 9, 10, 10], sabemos que la media es 10.4 y ahora vamos a tomar dos valores por ejemplo 9 y 12, y sus diferencias con respecto a la media son: $9 - 10.4$ (-1.4) y $12 - 10.4$ (1.6). Podemos calcular entonces la media de las diferencias elevadas al cuadrado para que no haya problemas por las restas que nos arrojen valores negativos. Básicamente así se calcula varianza para un conjunto de datos:

$$\sigma_n^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$$

Donde: x_i es el dato en la posición i de una lista.

\bar{x} es la media.

n es el número de datos.

La unidad de medida del resultado es al cuadrado de la unidad de medida de la variable. Si calculamos la raíz cuadrada de la varianza obtenemos la desviación estándar. Como ventaja de la desviación estándar queda expresada en las mismas unidades que la variable en cuestión. Al igual que la media, estas medidas de dispersión también se ven afectadas por la presencia de valores atípicos (outliers).

Desviación Absoluta Promedio

En lugar de elevar al cuadrado la diferencia de los valores con respecto a la media se calcula el valor absoluto de la diferencia:

$$\frac{1}{n} \sum_{i=1}^n |x_i - m(X)|$$

Desviación Absoluta Media

La desviación absoluta media o MAD (del inglés Median Absolute Deviation) es una medida de dispersión robusta, ya que es más resistente a los valores atípicos. En lugar de utilizar las medias como en los casos anteriores se utiliza la mediana de las diferencias absolutas contra la mediana:

$$MAD(x) = \text{mediana} (|x_i - \text{mediana}(x)|)$$

Correlación

NetworkX

Por hacer

Los datos

FLIGHT	DESTINATION	STATUS
104	BRISBANE	ON TIME
108	BRISBANE	ON TIME
112	BRISBANE	ON TIME
116	BRISBANE	ON TIME
120	BRISBANE	ON TIME
124	BRISBANE	ON TIME
128	BRISBANE	ON TIME
132	BRISBANE	ON TIME
136	BRISBANE	ON TIME
140	BRISBANE	ON TIME
144	BRISBANE	ON TIME
148	BRISBANE	ON TIME
152	BRISBANE	ON TIME
156	BRISBANE	ON TIME
160	BRISBANE	ON TIME
164	BRISBANE	ON TIME
168	BRISBANE	ON TIME
172	BRISBANE	ON TIME
176	BRISBANE	ON TIME
180	BRISBANE	ON TIME
184	BRISBANE	ON TIME
188	BRISBANE	ON TIME
192	BRISBANE	ON TIME
196	BRISBANE	ON TIME
200	BRISBANE	ON TIME
204	BRISBANE	ON TIME
208	BRISBANE	ON TIME
212	BRISBANE	ON TIME
216	BRISBANE	ON TIME
220	BRISBANE	ON TIME
224	BRISBANE	ON TIME
228	BRISBANE	ON TIME
232	BRISBANE	ON TIME
236	BRISBANE	ON TIME
240	BRISBANE	ON TIME
244	BRISBANE	ON TIME
248	BRISBANE	ON TIME
252	BRISBANE	ON TIME
256	BRISBANE	ON TIME
260	BRISBANE	ON TIME
264	BRISBANE	ON TIME
268	BRISBANE	ON TIME
272	BRISBANE	ON TIME
276	BRISBANE	ON TIME
280	BRISBANE	ON TIME
284	BRISBANE	ON TIME
288	BRISBANE	ON TIME
292	BRISBANE	ON TIME
296	BRISBANE	ON TIME
300	BRISBANE	ON TIME
304	BRISBANE	ON TIME
308	BRISBANE	ON TIME
312	BRISBANE	ON TIME
316	BRISBANE	ON TIME
320	BRISBANE	ON TIME
324	BRISBANE	ON TIME
328	BRISBANE	ON TIME
332	BRISBANE	ON TIME
336	BRISBANE	ON TIME
340	BRISBANE	ON TIME
344	BRISBANE	ON TIME
348	BRISBANE	ON TIME
352	BRISBANE	ON TIME
356	BRISBANE	ON TIME
360	BRISBANE	ON TIME
364	BRISBANE	ON TIME
368	BRISBANE	ON TIME
372	BRISBANE	ON TIME
376	BRISBANE	ON TIME
380	BRISBANE	ON TIME
384	BRISBANE	ON TIME
388	BRISBANE	ON TIME
392	BRISBANE	ON TIME
396	BRISBANE	ON TIME
400	BRISBANE	ON TIME
404	BRISBANE	ON TIME
408	BRISBANE	ON TIME
412	BRISBANE	ON TIME
416	BRISBANE	ON TIME
420	BRISBANE	ON TIME
424	BRISBANE	ON TIME
428	BRISBANE	ON TIME
432	BRISBANE	ON TIME
436	BRISBANE	ON TIME
440	BRISBANE	ON TIME
444	BRISBANE	ON TIME
448	BRISBANE	ON TIME
452	BRISBANE	ON TIME
456	BRISBANE	ON TIME
460	BRISBANE	ON TIME
464	BRISBANE	ON TIME
468	BRISBANE	ON TIME
472	BRISBANE	ON TIME
476	BRISBANE	ON TIME
480	BRISBANE	ON TIME
484	BRISBANE	ON TIME
488	BRISBANE	ON TIME
492	BRISBANE	ON TIME
496	BRISBANE	ON TIME
500	BRISBANE	ON TIME
504	BRISBANE	ON TIME
508	BRISBANE	ON TIME
512	BRISBANE	ON TIME
516	BRISBANE	ON TIME
520	BRISBANE	ON TIME
524	BRISBANE	ON TIME
528	BRISBANE	ON TIME
532	BRISBANE	ON TIME
536	BRISBANE	ON TIME
540	BRISBANE	ON TIME
544	BRISBANE	ON TIME
548	BRISBANE	ON TIME
552	BRISBANE	ON TIME
556	BRISBANE	ON TIME
560	BRISBANE	ON TIME
564	BRISBANE	ON TIME
568	BRISBANE	ON TIME
572	BRISBANE	ON TIME
576	BRISBANE	ON TIME
580	BRISBANE	ON TIME
584	BRISBANE	ON TIME
588	BRISBANE	ON TIME
592	BRISBANE	ON TIME
596	BRISBANE	ON TIME
600	BRISBANE	ON TIME
604	BRISBANE	ON TIME
608	BRISBANE	ON TIME
612	BRISBANE	ON TIME
616	BRISBANE	ON TIME
620	BRISBANE	ON TIME
624	BRISBANE	ON TIME
628	BRISBANE	ON TIME
632	BRISBANE	ON TIME
636	BRISBANE	ON TIME
640	BRISBANE	ON TIME
644	BRISBANE	ON TIME
648	BRISBANE	ON TIME
652	BRISBANE	ON TIME
656	BRISBANE	ON TIME
660	BRISBANE	ON TIME
664	BRISBANE	ON TIME
668	BRISBANE	ON TIME
672	BRISBANE	ON TIME
676	BRISBANE	ON TIME
680	BRISBANE	ON TIME
684	BRISBANE	ON TIME
688	BRISBANE	ON TIME
692	BRISBANE	ON TIME
696	BRISBANE	ON TIME
700	BRISBANE	ON TIME
704	BRISBANE	ON TIME
708	BRISBANE	ON TIME
712	BRISBANE	ON TIME
716	BRISBANE	ON TIME
720	BRISBANE	ON TIME
724	BRISBANE	ON TIME
728	BRISBANE	ON TIME
732	BRISBANE	ON TIME
736	BRISBANE	ON TIME
740	BRISBANE	ON TIME
744	BRISBANE	ON TIME
748	BRISBANE	ON TIME
752	BRISBANE	ON TIME
756	BRISBANE	ON TIME
760	BRISBANE	ON TIME
764	BRISBANE	ON TIME
768	BRISBANE	ON TIME
772	BRISBANE	ON TIME
776	BRISBANE	ON TIME
780	BRISBANE	ON TIME
784	BRISBANE	ON TIME
788	BRISBANE	ON TIME
792	BRISBANE	ON TIME
796	BRISBANE	ON TIME
800	BRISBANE	ON TIME
804	BRISBANE	ON TIME
808	BRISBANE	ON TIME
812	BRISBANE	ON TIME
816	BRISBANE	ON TIME
820	BRISBANE	ON TIME
824	BRISBANE	ON TIME
828	BRISBANE	ON TIME
832	BRISBANE	ON TIME
836	BRISBANE	ON TIME
840	BRISBANE	ON TIME
844	BRISBANE	ON TIME
848	BRISBANE	ON TIME
852	BRISBANE	ON TIME
856	BRISBANE	ON TIME
860	BRISBANE	ON TIME
864	BRISBANE	ON TIME
868	BRISBANE	ON TIME
872	BRISBANE	ON TIME
876	BRISBANE	ON TIME
880	BRISBANE	ON TIME
884	BRISBANE	ON TIME
888	BRISBANE	ON TIME
892	BRISBANE	ON TIME
896	BRISBANE	ON TIME
900	BRISBANE	ON TIME
904	BRISBANE	ON TIME
908	BRISBANE	ON TIME
912	BRISBANE	ON TIME
916	BRISBANE	ON TIME
920	BRISBANE	ON TIME
924	BRISBANE	ON TIME
928	BRISBANE	ON TIME
932	BRISBANE	ON TIME
936	BRISBANE	ON TIME
940	BRISBANE	ON TIME
944	BRISBANE	ON TIME
948	BRISBANE	ON TIME
952	BRISBANE	ON TIME
956	BRISBANE	ON TIME
960	BRISBANE	ON TIME
964	BRISBANE	ON TIME
968	BRISBANE	ON TIME
972	BRISBANE	ON TIME
976	BRISBANE	ON TIME
980	BRISBANE	ON TIME
984	BRISBANE	ON TIME
988	BRISBANE	ON TIME
992	BRISBANE	ON TIME
996	BRISBANE	ON TIME
1000	BRISBANE	ON TIME

La materia prima para realizar nuestra labor de mineros de datos es inagotable. Hablamos de los datos, ese recurso en bruto que debemos procesar para sacarle provecho. En esta sección discutiremos temas relacionados con los conjuntos de datos y el tipo de operaciones que podemos hacer con ellos. Las tareas de esta sección serán:

- Definir a los datos como objetos.
- Conocer los tipos de estructuras para conjuntos de datos:
 - Tablas
 - Matrices
 - Grafos

Datos y objetos

Como la Minería de Datos es una actividad multidisciplinar, muchas veces hay confusión o incluso diferencias ideológicas con respecto a los nombres y términos que se utilizan. Si a esto le agregamos la traducción al castellano, podríamos pasarnos mucho tiempo discutiendo sin llegar a una definición única. Por ejemplo en inglés se le llama “data” al plural de “datum” o dato, esto al principio nos puede confundir a los que hablamos castellano.

Pero no a los que sepan que *data* es el plural neutro para un sustantivo de la primera declinación en latín.

Como primer paso vamos a definir los conceptos básicos utilizando de ser posible términos que sean familiares a los profesionistas en informática.

Datos: son una representación de las observaciones recolectadas. Por ejemplo un médico puede tomar (recolectar) la temperatura corporal de varios pacientes y representar la lectura en grados centígrados.

Objetos: Normalmente representamos a cada observación como un objeto o registro. Por ejemplo una lectura de los signos vitales de un paciente se podría representar como el siguiente objeto:

Atributo	Valor
id	1
nombre	Juan Pérez
día	Lunes
estatura	175
temperatura	36.2
presión_diastólica	80
presión_sistólica	135
frecuencia_cardíaca	90

Atributos: Son las propiedades o características que tienen los objetos de cierto tipo. Todas las observaciones que se hagan del mismo tipo de objeto tendrán los mismos atributos. Para el ejemplo anterior los atributos serán {id, nombre, estatura, temperatura, presión_diastólica, presión_sistólica, frecuencia_cardíaca }.

Valores: El *estado* de un objeto corresponde a los valores que tienen sus atributos en un momento dado. En el análisis de datos nos enfocamos en datos históricos por lo que es poco común que haya cambios de estado en los objetos. Los valores a su vez debemos considerarlos como objetos. Por ejemplo, en la lectura de ejemplo el atributo *nombre* tiene como valor al objeto tipo cadena “Juan Pérez”. Para analizar las observaciones estadísticamente debemos de considerar su tipo:

- Dato *cualitativo*: Normalmente la observación es expresada en palabras. Por ejemplo, la reseña de un restaurante. Los datos cualitativos pueden ayudarnos a etiquetar a un objeto como miembro de una clase o grupo. En el caso de un objeto tipo restaurante, su atributo “tipo de cocina” nos ayuda a clasificarlo por cocina: *mexicana, japonesa, italiana*.
- Dato *cuantitativo*: La observación se realiza sobre algún atributo que tiene cierta magnitud. Se tienen dos tipos:
 - Discretos. Estos datos se expresan en números enteros y normal-

- mente se refieren a conteos. Ejemplos: *Número de empleados, visitas, número de páginas, temporada*.
- Continuos. Los datos corresponden a mediciones en números reales, como *temperatura corporal, sueldo o duración*.

Conjunto de Datos: Los conjuntos de datos (en inglés *dataset*) son colecciones de objetos que casi siempre son del mismo tipo. Es común representar a un conjunto de datos en forma de registros en una tabla. Como ejemplo aquí tenemos un conjunto de datos de lecturas de signos vitales:

id	nombre	temperatura	día	estatura
1	Juan Pérez	36.2	Lunes	175
2	Ana Rivas	136.3	Martes	125
3	María Sánchez	36.5	Jueves	190
4	Luis Duarte	39.0	Lunes	180
5	José Arias	36.5	Viernes	90
6	Margarita Lee	39.0	Jueves	173

A los conjuntos de datos cuando no se han procesado les llamamos *datos crudos* (raw data), en este caso por ejemplo se tienen datos erróneos, la temperatura del paciente con *id* = 2, no es posible que sea de 136.3°C. También podría ser necesario anonimizar los datos al tratarse de datos confidenciales.

Clasificación de Escalas de Medición

El tipo de análisis que se puedan hacer en un conjunto de datos dependerá del tipo de variable de sus atributos. Por ejemplo, en el caso del conjunto de datos de pacientes, podríamos calcular la media de la estatura, pero no tendría sentido hacer el cálculo para el atributo nombre. Incluso cuando utilizamos datos enteros la media podría no tener sentido. Por ejemplo, la media del atributo *id* no nos diría nada sobre las lecturas ya que es solo una etiqueta que resulta ser entera. Utilizaremos la clasificación de Stevens para clasificar los tipos de tipos de valores asignados a los atributos en cuatro niveles:

Nominal o Categórica

Esta es una medición cualitativa como vimos anteriormente y nos puede servir para asignar a los objetos a un grupo o categoría. Por ejemplo, el atributo *especie* de las flores Iris, puede tomar los valores: “Setosa”, “Virgínica” o “Versicolor”. Estos valores nos indican a que categoría pertenece cada flor. Una medida categórica no siempre va a ser texto. Por ejemplo, el número de serie o un número de teléfono. En algunos casos nos brindan información cualitativa sobre el objeto: *país de origen, sexo, estado civil*. * Los operadores que podemos utilizar en atributos que usan este tipo de medida son: =, ≠. * Podemos calcular la moda, entropía, prueba χ^2 . * Podemos agrupar.

Ordinal

Esta es también una medición categórica, pero existe un orden con el cual podemos ordenar a los objetos. Por ejemplo, los valores de dificultad “Fácil”, “Difícil”; talla de la ropa, el nivel máximo de estudios. Otro ejemplo son los valores de una escala de agrado: “nada”, “poco”, “regular”, “mucho”, estos valores cubren un espectro de posibilidades.

* Los operadores que podemos utilizar son: $=$, \neq , $>$, $<$. * Podemos calcular la moda, mediana, percentiles, pruebas de signos. * Podemos agrupar y ordenar.

Intervalar

Esta nos permite establecer diferencia entre los objetos, pero como no tiene un cero absoluto no se puede calcular la proporción entre dos valores. Un ejemplo clásico de este tipo de medición es la temperatura cuando la medimos en grados centígrados, podemos decir que la diferencia entre 3°C y 5°C grados son 2°C . Pero no podemos decir que 20°C es el doble de caliente que la temperatura a 20°C . * Los operadores que podemos utilizar son: $=$, \neq , $>$, $<$, $+$, $-$. * Como medida de tendencia central podemos calcular la moda, mediana y la media aritmética. Son válidas las medidas de dispersión de rango y desviación estándar. * Podemos agrupar, ordenar, calcular la diferencia.

Razón

Se cuanta con un cero absoluto, por lo que las proporciones son válidas. La mayoría de las mediciones en ingeniería o física utilizan este tipo de escalas. Por ejemplo masa, longitud, duración, carga eléctrica, corriente eléctrica, temperatura en grados Kelvin.

- Los operadores que podemos utilizar son: $=$, \neq , $>$, $<$, $+$, $-$, \times , \div .
- Como medida de tendencia central podemos calcular la moda, mediana y la media aritmética, media geométrica y media armónica. Son válidas las medidas de dispersión de rango y desviación estándar.
- Podemos agrupar, ordenar, calcular la diferencia.

Tipos de Conjuntos de Datos

Muchos que ya contamos con algo de experiencia en temas de *Bases de Datos* tenemos la idea equivocada de que la *Minería de Datos* consiste en algún tipo de extensión a los lenguajes de consulta como SQL. La verdad es que la mayoría de los algoritmos de minería de datos se realizan fuera del servidor, utilizando lenguajes de propósito general como Python o Java. Recordemos que los algoritmos han sido desarrollados en gran parte por especialistas en áreas muy distintas como álgebra lineal, inferencia estadística, inteligencia artificial y reconocimiento de patrones. En estas áreas es común utilizar vectores y matrices

como entradas a los algoritmos ya que estos nos permiten expresar operaciones de una manera muy eficiente.

Como este es un material de introductorio nos enfocaremos en tres conjuntos de datos básicos: tablas, matrices y grafos. A continuación veremos una descripción breve de cada tipo.

Tablas

Recordemos que los primeros pasos del proceso de KDD están dedicados a pasar los datos de su fuente original al formato requerido por los algoritmos de minería. El formato de entrada para la mayoría de los algoritmos es un arreglo de vectores de características el cual podemos representar como una tabla.

Cuando utilizamos la estructura de tabla, en algunas bibliotecas o software para Minería de Datos como Orange o Rapid Miner se deben especificar ciertos *metadatos*. Los *metadatos* en este caso vienen siendo datos adicionales sobre cada atributo. Por ejemplo, el tipo de atributo (categórico/nominal, continuo, discreto, tiempo, cadena) y el rol (característica, clase/etiqueta, ponderación, ignorar).

Por ejemplo, para el conjunto de datos Iris:

sepal_length	sepal_width	petal_length	petal_width	species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
7.0	3.2	4.7	1.4	versicolor
6.4	3.2	4.5	1.5	versicolor
6.3	3.3	6.0	2.5	virginica
5.8	2.7	5.1	1.9	virginica

En el software Orange para cada atributo debemos indicar los metadatos de la siguiente manera:

atributo	tipo	rol	meta
sepal_length	continuo	característica	
sepal_width	continuo	característica	
petal_length	continuo	característica	
petal_width	continuo	característica	
species	nominal	etiqueta	setosa, versicolor, virginica

En otros casos, como la biblioteca scikit-learn, no es necesario especificar metadatos, se requieren los datos de cierta manera para identificar de que tipo son. Por ejemplo, nos pueden pedir dos arreglos, uno para las características:

```
[[5.1, 3.5, 1.4, 0.2], # setosa
 [4.9, 3.0, 1.4, 0.2], # setosa
 [7.0, 3.2, 4.7, 1.4], # versicolor
 [6.4, 3.2, 4.5, 1.5], # versicolor
 [6.3, 3.3, 6.0, 2.5], # virginica
 [5.8, 2.7, 5.1, 1.9]] # virginica
```

Y otro para las etiquetas:

```
['setosa', 'setosa', 'versicolor', 'versicolor', 'virginica', 'virginica']
```

En este caso, como no especificamos que es que, debemos de separar los datos a la entrada. Unos son características y los otros la clase. Aunque son dos arreglos distintos, se trata del mismo objeto. Es por esto que deben de corresponder las posiciones en ambos arreglos.

Independientemente de la representación es muy importante conocer el tipo y rol de los atributos. Por ejemplo en este caso vemos que todas las características son continuas y solo la etiqueta es categórica. Sabemos entonces que solo podremos utilizar estos datos en algoritmos que puedan recibir valores continuos. Existen algoritmos que solo pueden trabajar con datos categóricos o aquellos en los que la etiqueta debe ser binaria. En la sección de preprocesamiento veremos varias técnicas para modificar los datos de tal manera que podamos utilizar la mayor cantidad de algoritmos posible y en algunos casos mejorar el desempeño de los compatibles.

Matrices

A diferencia de las tablas, en las matrices las columnas no representan atributos de los objetos. Más bien, las matrices representan relaciones entre objetos. Por ejemplo, en una matriz podemos representar la distancia que existe entre dos ciudades o indicar las materias que ha cursado un alumno. Veámoslo también como celdas:

	Tijuana	Ensenada	Mexicali
Tijuana	0	104.2	185.5
Ensenada	104.2	0	240.3
Mexicali	185.5	240.3	0

	Cloud Computing	Big Data	Complex Systems
Ana	1	1	1
Tom	0	1	0
Joe	0	0	0

Algunas de las matrices tienen características interesantes, por ejemplo vemos que la matriz de distancia entre ciudades es simétrica o la de los cursos tiene solo datos binarios. Veamos ejemplos de matrices que utilizamos comúnmente en minería de datos:

Documento-Término

Esta matriz describe a una colección de documentos a través de los términos que aparecen en ellos. Cada renglón representa un documento y cada columna corresponde a un término. Los términos se extraen inicialmente de todos los documentos. En cada celda se tiene algún valor representativo de la relación documento-término. Los valores tienen que ver con la frecuencia con la que aparece el término en el documento y en los demás. Una representación común es el peso Tf-idf (del inglés Term frequency – Inverse document frequency) el cual se describiremos a detalle en la sección de preprocesamiento. Para entender mejor la idea consideremos estos dos documentos:

D1 : "La vida no vale nada, no vale nada la vida"

D2 : "Toda la vida coleccionado mil amores"

Podemos representar a estos documentos con la siguiente matriz, la cual captura simplemente la frecuencia los términos en cada documento:

	vida	no	vale	toda	coleccionando	mil	amores
D1	2	2	2	0	0	0	0
D2	1	0	0	1	1	1	1

Podemos observar que el proceso de generación de la matriz también removi6 algunos términos que son demasiado comunes. Eliminamos a los términos que aparecen en casi todos los documentos por que no nos dicen nada distintivo de los documentos.

Una característica importante de la matriz documento-término es que la mayoría de los elementos tendrán el valor de cero, por eso decimos que es una matriz escasa. Las matrices escasas normalmente se implementan utilizando estructuras de datos especiales para no desperdiciar espacio y que el procesamiento sea más rápido.

Filtrado Colaborativo

Los sistemas de recomendación tienen como objetivo recomendar a un usuario diferentes artículos como películas, restaurantes y libros de manera personalizada. Una técnica muy utilizada para hacer recomendaciones es el filtrado colaborativo. El algoritmo toma como base una matriz donde se indica que calificación le han puesto los usuarios a los artículos. Para el caso de un sistema de recomendación de películas, podríamos tener la siguiente matriz:

	The Matrix	Matando Cabos	Mad Max: Fury Road
Ana	0	1	5
Tom	4	4	0
Joe	5	5	5
Tim	0	1	5

Esta de nuevo es una matriz escasa donde se indica en cada entrada la calificación o utilidad que le asignó un usuario a un artículo. Comúnmente el valor de cero indica que el usuario no le ha asignado todavía una calificación al artículo. Las calificaciones normalmente son valores positivos discretos.

Una característica interesante de esta matriz es que nos permite recomendar artículos de dos maneras. Podemos recomendar buscando usuarios que son similares o películas que son similares. Para hacer esto solo necesitamos girar la matriz. En la sección de sistemas de recomendación veremos el tema con mayor detalle.

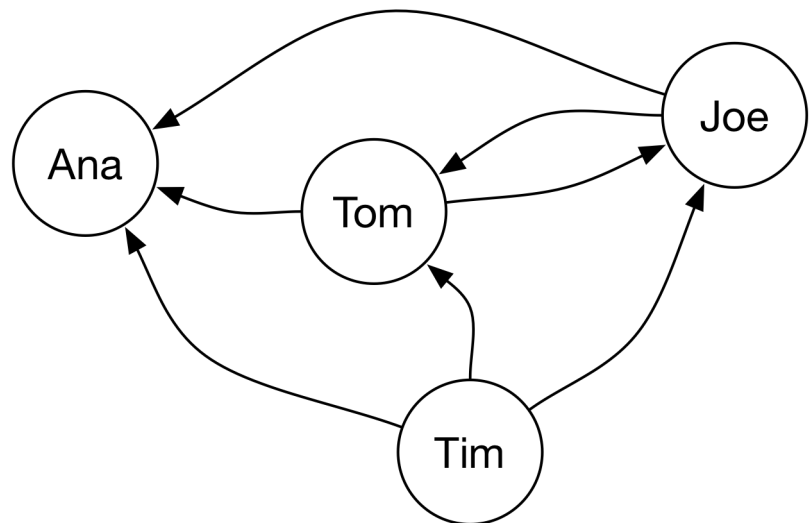
Grafos

Al igual que las matrices, los grafos nos permiten representar de una manera natural relaciones entre distintos objetos o incluso objetos y sus atributos.

Un grafo que relacione objetos del mismo tipo lo podemos representar directamente como una matriz. Veamos un ejemplo. En la red social de Twitter tenemos la relación de *sigue a* entre objetos tipo usuarios. Esta matriz no es simétrica, por ejemplo Tom sigue a Ana pero Ana no sigue a Tom.

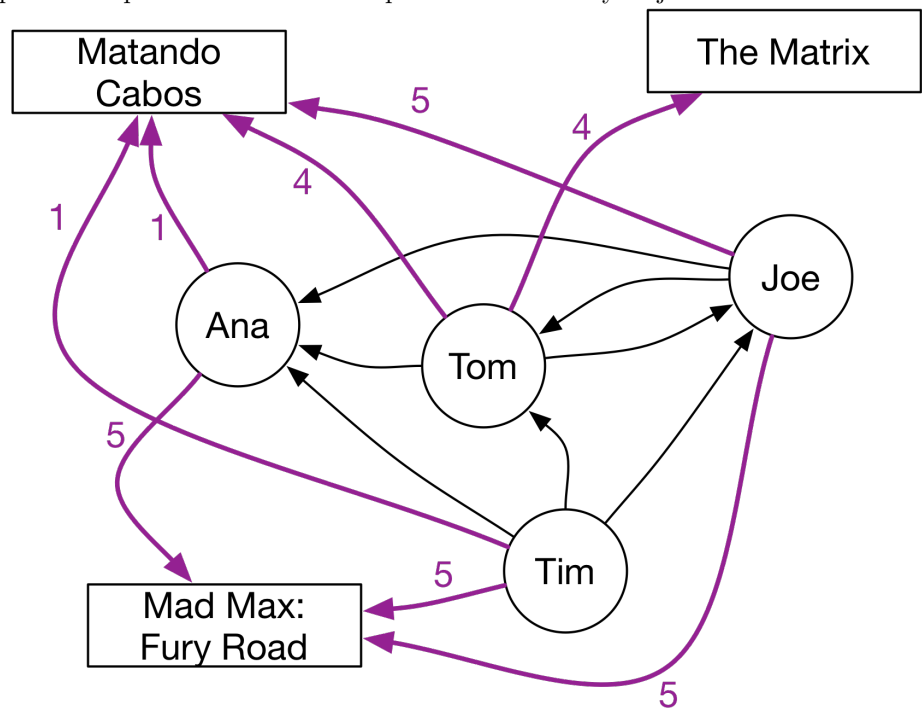
	Ana	Tom	Joe	Tim
Ana	0	0	0	0
Tom	1	0	1	0
Joe	1	1	0	0
Tim	1	1	1	0

Estas relaciones las podríamos representar con un grafo, el cual se puede repre-



sentar gráficamente como:

Un grafo nos permite representar diferentes tipos de relaciones y objetos al



mismo tiempo:

Mientras algunos algoritmos pueden trabajar directamente con grafos otros requieren su representación matricial.

A lo largo del material utilizaremos estos tres tipos de conjuntos de datos. Existen variantes de estos y algunos otros especializados. Por ejemplo, para encontrar patrones en imágenes, videos, gestos, el genoma humano, series de tiempo,

y muchas otros tipos de datos. Recuerda que este material es solo el principio de muchos otros cursos, tutoriales, videos y artículos que seguramente revisarás.

Resumen

En esta sección vimos de manera informal los tipos de datos que utilizaremos y sus propiedades. Más adelante utilizaremos las estructuras de datos vistas para resolver problemas prácticos.

Ejercicios

Falta por hacer ejercicios para reforzar los conceptos de: Observación, Dato, Objeto, Atributo, Medida, Categórico, Nominal, Razón, Ordinal, Intervalar, Conjunto de Dato, Tabla, Matriz, Grafo.

Lectura adicional

Bibliografía

Sobre la Teoría de las Escalas de Medición



Figure 21: Pandas

Pandas

El análisis de datos comúnmente se realiza sobre datos que se encuentran organizados en forma de tablas. Probablemente ya haz utilizado datos organizados de esta manera al utilizar una hoja de cálculo o tablas en servidores de bases de datos relacionales como Oracle, PostgreSQL o SQLServer. En python incluso podemos almacenar los datos de una tabla utilizando un arreglo de tuplas.

Otras opciones

¿Por que no utilizamos tablas de SQL?

Los sistemas de bases de datos tienen como objetivo almacenar grandes cantidades de datos y al mismo tiempo nos permiten realizar un gran número de transacciones de manera concurrente. Dicho de otra manera los servidores nos permiten realizar la parte operativa de una empresa, vender boletos para conciertos, inscribir alumnos, llevar el control de un almacén, procesar los pedidos en una tienda en línea, etc. Además de esto cuentan con un lenguaje de alto nivel para definir, manipular y consultar a los datos. El lenguaje estándar SQL ha sido el preferido por los desarrolladores para consultar de una manera muy flexible a los datos relacionales. El problema que tenemos en estos sistemas es que su objetivo principal no es el analizar los datos, estos sistemas nos permiten más bien implementar sistemas que necesitan manipular datos en-línea para funcionar. Por otro lado, para hacer nuestro análisis, podemos utilizar perfectamente datos que estén fuera de línea, datos históricos o incluso datos ficticios. Mucho del *overhead* de un servidor de bases de datos se tiene precisamente por que estos deben ser capaces de modificar las estructuras de las tablas, hacer modificaciones a objetos, mantener la consistencia, controlar el acceso y muchas otras operaciones. Todo en línea. Al no tener estos requisitos podemos operar con estructuras diseñadas precisamente para el análisis de datos. Dicho esto, algunos sistemas de bases de datos han agregado la funcionalidad necesaria para realizar el tipo de análisis que veremos, pero esto no es todavía el estándar. Los sistemas de bases de datos nos pueden servir para almacenar los datos que utilizaremos en el análisis y realizar algunas consultas con SQL. Podemos utilizar también la biblioteca SQLite para almacenar bitácoras, intercambiar información con otros sistemas o como parte del preprocesamiento.

¿Por que no utilizamos NoSQL?

Últimamente se han propuesto gestores de datos que aunque no siguen el modelo relacional nos permiten hacer cierto tipo de consultas de una manera mucho más eficiente. Por ejemplo, los almacenes clave-valor (key-value) pueden recuperar bastante rápido los datos por medio de su clave; pero el lenguaje de consulta, si es que lo hay, no es muy flexible. Son muy buenos para recuperar muy rápido

un documento html o un objeto con solo especificar su clave, pero no podemos hacer consultas *ad-hoc* como “recupera aquellos productos que tengan ventas superiores al promedio en las tiendas de California o Arizona, siempre que no sean del departamento de electrónica”. Estos sistemas nos pueden servir sobre todo como memoria cache o memoria compartida para realizar operaciones en paralelo.

¿Por que no utilizamos Data Warehouse?

Las técnicas de Almacenes de Datos (Data Warehouse) nos permiten realizar de una manera muy sencilla consultas *tipo cubo* sobre bases de datos relacionales. Aunque las consultas permiten a los ejecutivos tomar decisiones y tener una buena idea de lo que sucede en la empresa, su objetivo no es el de extraer patrones de los datos. Sin embargo el objetivo general y el diseño de los Almacenes de Datos tiene algunas cosas en común con el proceso de KDD. Ambos trabajan con datos históricos y requieren de un proceso previo de extracción de los datos al cual se le conoce como ETL (Extract, Transform and Load) o Extraer, Transformar y Cargar). El proceso de ETL permite a las empresas extraer datos desde diferentes fuentes, cambiar de formato y limpiarlos con el objetivo de cargarlos en el almacén de datos.

¿Por que no utilizamos Hoja de Cálculo?

De hecho las hojas de cálculo son una herramienta aceptable para realizar análisis de datos. Muchos analistas utilizan hojas de calculo como herramienta principal. No se requiere tener un amplio conocimiento de programación para sacarles provecho y pueden complementares con otras herramientas por ejemplo con sistemas de bases de datos. Los programadores normalmente preferimos tener un control total sobre el proceso de KDD y nos gusta poder integrar fácilmente nuestro código en distintas aplicaciones, por lo que preferimos la flexibilidad que nos brinda un lenguaje de propósito general.

Breve Introducción al DataFrame de Pandas

Pandas es una biblioteca código abierto en Python la cual se ha diseñado con el objetivo específico de brindarnos una estructura de tabla rápida y flexible para el análisis de datos. Según su documentación pandas nos permite operar con distintos tipos de datos:

- Tablas con atributos heterogéneos.
- Series de tiempo, ordenadas o no.
- Matrices homogéneas o heterogéneas.

Para esto en pandas se implementan dos estructuras principales: *Series* y *DataFrame*. La estructura *Series* nos permite trabajar con datos de series de tiempo especificadas como un vector. Por otro lado está *DataFrame* nos sirve para aquellos datos que tienen una estructura de tabla o matriz. En esta sección nos enfocaremos en la estructura *DataFrame* ya que nos interesa trabajar con datos organizados como tabla.

Creando un DataFrame desde un archivo de texto

Como ejemplo, vamos a utilizar una estructura tipo *DataFrame* para almacenar el conjunto de datos conocido como Auto MPG. Lo primero que debemos hacer es ver los tipos de datos de los atributos:

nombre	tipo	medida	descripción
mpg	continuo	razón	millas por galón
cylinders	discreto	ordinal	número de cilindros
displacement	continuo	razón	desplazamiento
horsepower	continuo	razón	caballos de fuerza
weight	continuo	razón	peso en libras US
acceleration	continuo	razón	aceleración
model__year	discreto	razón	año de fabricación
origin	discreto	categorico	origen
car__name	cadena	categorico	nombre único del auto

Como vemos estas observaciones tienen atributos heterogéneos ya que son de distintos tipos. El objetivo original de este conjunto de datos era el de predecir el consumo de combustible en millas por galón (mpg) utilizando los otros atributos.

Como siguiente paso vamos a descargar el conjunto de datos del repositorio de machine learning de la UC-Irvine. El archivo se llama **auto-mpg.data**. Al abrir el archivo en un editor de texto vemos que está separado por espacios y faltan algunos valores. Veamos un fragmento:

```
11.0  8  350.0    180.0    3664.    11.0  73  1  "oldsmobile omega"
20.0  6  198.0    95.00    3102.    16.5  74  1  "plymouth duster"
21.0  6  200.0     ?      2875.    17.0  74  1  "ford maverick"
```

Pandas nos brinda *IO Tools* un API de entrada/salida donde se incluyen métodos de lectura para formatos de texto, binarios y SQL. En el caso de text se incluyen lectores para formatos CSV, JSON y HTML. Vamos a intentar leer el archivo utilizando el método `read_csv()` para más detalle puedes ver la documentación. La manera en la que se lee y se hace el “parsing” a los archivos se puede configurar de una manera muy detallada, para este ejemplo solo modificaremos algunos parámetros básicos. Recordemos que en los archivos CSV, como el nombre lo dice, los valores de los atributos se separan por comas. Ya vi-

mos que en el caso del archivo **auto-mpg.data** la separación se hace por medio de espacios en blanco. Un problema adicional es que el número de espacios no siempre es el mismo. Para leer el archivo correctamente, vamos a utilizar el parámetro *sep* el cual recibe una especificación del separador que se va a utilizar, por defecto una coma ','. En nuestro caso el separador será una expresión regular la cual le diga al método que son uno o más espacios. El espacio y tabuladores se especifican en la expresión regular con la cadena '\s', mientras que el operador '+' indica que se puede repetir una o más veces. Nuestro primer intento quedaría de la siguiente manera:

```
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> import numpy as np
>>> df = pd.read_csv('datos-ejemplo/auto-mpg.data', sep='\s+')
```

Antes que nada importamos las bibliotecas que utilizaremos en estos ejercicios (1-3). Como primer parámetro enviamos la ruta al archivo *auto-mpg.data*, si ejecutamos el comando dentro del directorio del libro solamente especificamos la ruta de la siguiente manera: *'datos-ejemplo/auto-mpg.data'*. En caso de que se encuentre en otro lado simplemente cambiamos la ruta.

Nota:

En caso de que te marque algún error, tal vez debas actualizar las versiones de las bibliotecas de anaconda. Desde la línea de comando de tu sistema operativo ejecuta este comando:

```
conda update --all
```

```
>>> df
      18.0  8  307.0  130.0   3504.   12.0  70  1  \
0      15.0  8  350.0  165.0  3693.0   11.5  70  1
1      18.0  8  318.0  150.0  3436.0   11.0  70  1
2      16.0  8  304.0  150.0  3433.0   12.0  70
..
chevrolet chevelle malibu
0                buick skylark 320
1                plymouth satellite
2                  amc rebel sst
3                  ford torino
```

```
[397 rows x 9 columns]
```

El método lector espera que el primer renglón tenga el nombre de los atributos, pero nuestro archivo no los tiene. Por lo que el nombre de los atributos no tienen mucho sentido *"18.0 8 307.0 130.0 3504. 12.0 70 1 chevrolet chevelle malibu"*. Debemos indicar que nuestro archivo no incluye el renglón de encabezados con

el argumento *header=None*. En este caso, como no hemos especificado nombres aun los atributos tendrán como nombre simplemente un índice.

Vamos a corregir el problema indicando que no se tiene una línea de encabezado:

```
>>> df = pd.read_csv('datos-ejemplo/auto-mpg.data', sep='\s+', header=None)
```

```
>>> df
      0  1    2    3    4    5  6  7  \
0  18.0  8  307.0 130.0 3504.0 12.0 70  1
1  15.0  8  350.0 165.0 3693.0 11.5 70  1
..
8
0      chevrolet chevelle malibu
1      buick skylark 320
..
```

```
[398 rows x 9 columns]
```

Vamos especificando los nombres de los atributos, esto se hace agregando el parámetro **names** y enviando una lista con los nombres:

```
>>> df2 = pd.read_csv('datos-ejemplo/auto-mpg.data', sep='\s+', header=None,
names=['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration',
'model_year', 'origin', 'car_name'])
```

```
>>> df2
      mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0         8      307.0        130.0  3504.0         12.0
1  15.0         8      350.0        165.0  3693.0         11.5
2  18.0         8      318.0        150.0  3436.0         11.0
```

Mucho mejor, veamos que tipo de datos infirió el método de lectura para cada una de las columnas:

```
>>> df2.dtypes
```

```
mpg          float64
cylinders      int64
displacement  float64
horsepower    object
weight        float64
acceleration  float64
model_year    int64
origin        int64
car_name      object
dtype: object
```

Hay un problema con el atributo *horsepower* este debería de ser float64. Lo que sucede es que al leer el carácter ? el método de lectura no sabe que tipo

de dato inferir. Lo bueno es que podemos configurar como queremos que se resuelvan los casos donde tenemos datos “no disponibles” y también como se debe pasar este tipo de valores. El argumento que vamos a utilizar es *na_values*, podemos indicar una lista de valores o como en este caso simplemente el símbolo correspondiente.

```
>>> df2 = pd.read_csv('datos-ejemplo/auto-mpg.data', sep='\s+', header=None,
                      names=['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
                             'acceleration', 'model_year', 'origin', 'car_name'], na_values='?')
```

```
In [12]: df2.dtypes
```

```
Out[12]:
```

```
mpg                float64
cylinders           int64
displacement        float64
horsepower          float64
weight             float64
acceleration        float64
model_year          int64
origin             int64
car_name           object
dtype: object
```

Ahora si, cada una de las columnas corresponde al tipo de dato de los atributos de la tabla anterior. Si queremos también podemos indicar al momento de la lectura el nombre y tipo de dato de los atributos. Incluso podemos indicar si tenemos algún dato categórico.

```
>>> df2 = pd.read_csv('datos-ejemplo/auto-mpg.data', sep='\s+',
                      header=None, na_values='?', names=['mpg', 'cylinders', 'displacement', 'horsepower',
                                                         'weight', 'acceleration', 'model_year', 'origin', 'car_name'],
                      dtype={'mpg': 'f4', 'cylinders': 'i4', 'displacement': 'f4',
                             'horsepower': 'f4', 'weight': 'f4', 'acceleration': 'f4',
                             'model_year': 'i4', 'origin': 'category', 'car_name': 'category'})
```

```
>>> df2.dtypes
```

```
mpg                float32
cylinders           int32
displacement        float32
horsepower          float32
weight             float32
acceleration        float32
model_year          int32
origin             category
car_name           category
dtype: object
```

Una vez leído correctamente el *DataFrame* podemos utilizar el método **describe()** para generar un resumen estadístico descriptivo que incluye la tendencia central, dispersión y la distribución del conjunto de datos, excluyendo datos categóricos y valores no disponibles.

```
>>> df2.describe()
```

	mpg	cylinders	displacement	horsepower	weight \
count	398.000000	398.000000	398.000000	392.000000	398.000000
mean	23.514574	5.454774	193.425873	104.469391	2970.424561
std	7.815985	1.701004	104.269859	38.491138	846.841431
min	9.000000	3.000000	68.000000	46.000000	1613.000000
25%	17.500000	4.000000	104.250000	75.000000	2223.750000
50%	23.000000	4.000000	148.500000	93.500000	2803.500000
75%	29.000000	8.000000	262.000000	126.000000	3608.000000
max	46.599998	8.000000	455.000000	230.000000	5140.000000

	acceleration	model_year
count	398.000000	398.000000
mean	15.568086	76.010050
std	2.757689	3.697627
min	8.000000	70.000000
25%	13.825000	73.000000
50%	15.500000	76.000000
75%	17.175001	79.000000
max	24.799999	82.000000

Ya que hablamos de los datos categóricos, vamos especificando el nombre de cada categoría. En el archivo solo se especifican los valores 1, 2 y 3. Al ver los modelos de los autos, inferimos que deben ser “USA”, “Japan” y “Germany”.

```
>>> df2["origin"].cat.categories = ["USA", "Japan", "Germany"]
```

```
>>> df2['origin']
```

```
0      USA
1      USA
2      USA
3      USA
4      USA
...
390    Germany
391      USA
392      USA
393      USA
394      Japan
395      USA
```

```
396      USA
397      USA
```

```
Name: origin, Length: 398, dtype: category
Categories (3, object): [USA, Japan, Germany]
```

¡Muy bien!, ¿ahora que tal si hacemos una gráfica de pastel?. Para esto generamos la gráfica con el método **plot()**. Todo esto lo veremos a detalle en la sección de visualización de datos.

```
>>> df2['origin'].value_counts().plot(kind='bar')
<matplotlib.axes._subplots.AxesSubplot at 0x10d1d1cd0>
```

Una vez creada la gráfica la mostramos con el método **show()** de la biblioteca *matplotlib*.

```
>>> plt.show()
```

Deberíamos ver la siguiente gráfica:

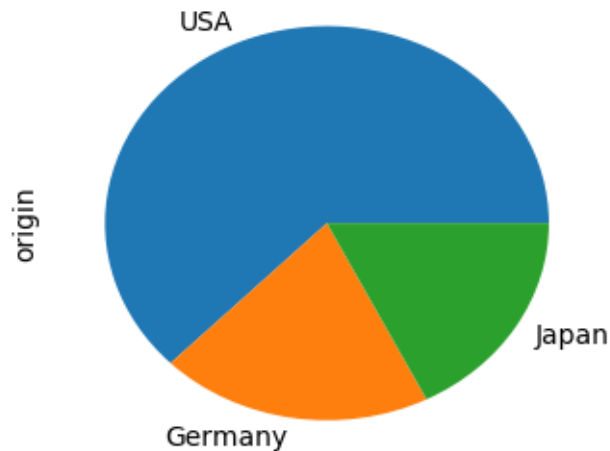


Figure 22: pie

Nota:

Debes de cerrar la ventana donde se muestra la gráfica para desbloquear el interprete y poder continuar. Puedes grabar la gráfica si gustas.

Vamos ahora a graficar tres variables para ver si vemos algo interesante:

```
>>> df2.plot.scatter(x='weight', y='mpg', c='horsepower', cmap='viridis');  
>>> plt.show()
```

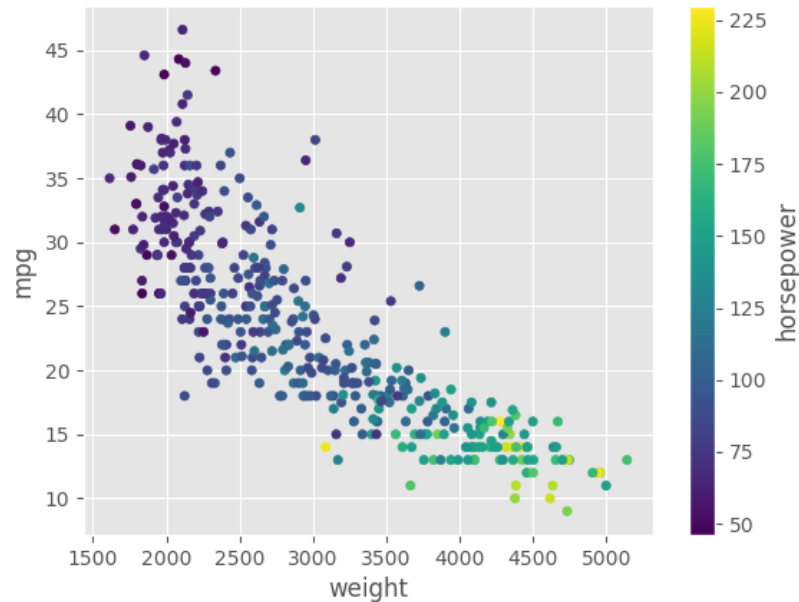


Figure 23: Plot

Intenta cambiar el mapa de colores con otras opciones Color Maps.

Si quieres intentar algunas otras gráficas adelante: <http://pandas.pydata.org/pandas-docs/stable/visualization.html#visualization>

Como hemos visto una vez que tenemos nuestros datos en la estructura *DataFrame* podemos hacer cosas interesantes muy fácilmente.

En esta sección vimos un ejemplo sencillo de como leer los datos de un archivo ya que es una tarea muy común. Ahora veremos como almacenar los datos utilizando objetos de python ya que esto nos brindará una flexibilidad adicional, por ejemplo para leer los datos desde una base de datos No-SQL o por http.

Creando un DataFrame con el constructor

El constructor de la clase *DataFrame* puede tomar los siguientes parámetros: **data**, **index**, **columns**, **dtype** y **copy**. Vemos cada uno de ellos:

data

En este parámetro enviamos los datos, como sucede muchas veces en Python, estos pueden ser de varios tipos:

lista de tuplas

La manera más sencilla de enviar datos a un *DataFrame* es utilizando una lista de tuplas:

```
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> import numpy as np
>>> datos = [('ana', 'ann@example.com', 'active'),
             ('tom', 'tommy@example.com', 'active'),
             ('joe', 'jj@example.com', 'active')]
>>> df2 = pd.DataFrame(datos)
>>> df2
```

	0	1	2
0	ana	ann@example.com	active
1	tom	tommy@example.com	active
2	joe	jj@example.com	active

```
>>>
```

La lista de tuplas puede venir de una consulta de base de datos u otro proceso.

numpy.ndarray

Los datos pueden venir también como un arreglo *n-dimensional* de la biblioteca numpy. De hecho la estructura *DataFrame* utiliza internamente este tipo de objetos. Un *ndarray* es un contenedor multidimensional de objetos del mismo tipo y tamaño. Los arreglos multidimensionales también cuentan con un objeto de definición de tipo (data-type dtype).

Como ejemplo vamos a crear un arreglo *dimensión-2*, de tamaño 2x4 el cual contiene enteros de 32 bits:

```
>>> arreglo = np.array([[1, 2, 3, 4], [9, 9, 4, 8]], dtype=np.int32)
>>> type(arreglo)
<type 'numpy.ndarray'>
>>> arreglo.shape
(2, 4)
>>> arreglo.dtype
dtype('int32')
```

Podemos revisar el tamaño y tipo de los objetos *numpy.ndarray* con los atributos *shape* y *dtype* respectivamente. Si te fijas la tupla que regresa el atributo *shape* nos dice el número de renglones primero y después el número de columnas.

Ahora vamos a crear un *DataFrame* a partir de nuestro arreglo:

```
>>> df = pd.DataFrame(arreglo)
>>> print df
   0  1  2  3
0  1  2  3  4
1  9  9  4  8
>>> print arreglo
[[1 2 3 4]
 [9 9 4 8]]
```

Aunque tienen los mismos elementos, podemos ver que el objeto *df* incluye el nombre de los atributos y renglones. En este caso por defecto los nombres son de nuevo un índice. Más adelante veremos los parámetros *index* y *columns* para cambiar los nombres. Recordemos que en el caso de conjuntos de datos tipo matriz podría no tener mucho sentido el nombre de las columnas ya que no representan atributos de un objeto.

Diccionario

Podemos utilizar un diccionario de python para enviar los datos. Podemos utilizar dos variantes:

{atributo: secuencia}

En este caso las claves son los nombres de atributos y el valor de cada clave es un vector con los datos correspondientes:

```
>>> datos = {'nombre': ['ana', 'tom', 'joe'],
             'email': ['ann@example.com', 'tommy@example.com', 'jj@example.com']}
>>> df2 = pd.DataFrame(datos)
>>> df2
             email nombre
0  ann@example.com   ana
1  tommy@example.com  tom
2   jj@example.com   joe
```

Como vemos el nombre de los renglones es el índice. En caso de que pasemos a un solo objeto en lugar de un vector, el valor de este se va a repetir en cada renglón.

```
>>> datos = {'nombre': ['ana', 'tom', 'joe'],
             'email': ['ann@example.com', 'tommy@example.com', 'jj@example.com'],
             'current_state': 'active'}
>>> df2 = pd.DataFrame(datos)
>>> df2
current_state             email nombre
0         active  ann@example.com   ana
```

```

1      active  tommy@example.com    tom
2      active    jj@example.com    joe

```

{atributo: diccionario}

Si queremos especificar el nombre de cada renglón lo podemos hacer pasando un diccionario por cada atributo, en el diccionario la clave es el nombre del renglón:

```

>>> datos = {
    'nombre': {
        'row1': 'ana',
        'row2': 'tom',
        'row3': 'joe'},
    'email': {'row1': 'ann@example.com',
              'row2': 'tommy@example.com',
              'row3': 'jj@example.com'}}
>>> df3 = pd.DataFrame(datos)
>>> print df3

```

	email	nombre
row1	ann@example.com	ana
row2	tommy@example.com	tom
row3	jj@example.com	joe

dtype

index

En este parámetro se especifica el índice o nombre de cada renglón.

columns

En este parámetro se especifica el nombre de cada atributo o columna.

Vamos a pasar ciertos valores para **index** y **columns**:

```

>>> datos = [('ana', 'ann@example.com', 'active'),
              ('tom', 'tommy@example.com', 'active'),
              ('joe', 'jj@example.com', 'active')]

>>> nombre_columna = ['nombre', 'email', 'current_status']
>>> nombre_renglon = ['r1', 'r2', 'r3']
>>> df2 = pd.DataFrame(datos, index=nombre_renglon, columns= nombre_columna)
>>> df2

```

	nombre	email	current_status
r1	ana	ann@example.com	active
r2	tom	tommy@example.com	active


```
r3      joe      jj@example.com      active
>>>
```

copy

Cuando enviamos datos utilizando el *numpy.ndarray*, realmente pasamos una referencia a *data*. Esto es más eficiente ya que no se crea una copia adicional. Si lo que queremos es que los datos se copien debemos enviar *True* al parámetro booleano *copy*.

DataFrame como matriz

¿Podemos utilizar un *DataFrame* para representar una matriz?. Intentemos representar los conjuntos de datos vistos en la sección Los Datos.

Relaciones entre objetos

	Tijuana	Ensenada	Mexicali
Tijuana	0	104.2	185.5
Ensenada	104.2	0	240.3
Mexicali	185.5	240.3	0

Para esta matriz va bien un diccionario:

```
>>> datos = {
    'Tijuana': {'Tijuana':0, 'Ensenada':104.2, 'Mexicali':185.5},
    'Ensenada': {'Tijuana':104.2, 'Ensenada':0, 'Mexicali':240.3},
    'Mexicali': {'Tijuana':185.5, 'Ensenada':240.3, 'Mexicali':0}
}
```

```
>>> distancias = pd.DataFrame(datos)
```

```
>>> distancias
```

```

      Ensenada  Mexicali  Tijuana
Ensenada      0.0      240.3    104.2
Mexicali      240.3        0.0    185.5
Tijuana       104.2      185.5        0.0
```

¿Por qué el orden de las ciudades es distinto?

¿Podremos ahorrar espacio o simplificar esta matriz? Si te fijas la matriz es simétrica, es la misma distancia de Tijuana a Ensenada que de Ensenada a

Tijuana. También todas las distancias entre la misma ciudad son obviamente cero.

¿Que tal si eliminamos los casos de la misma ciudad?

```
>>> datos = { 'Tijuana':{'Ensenada':104.2,'Mexicali':185.5},
               'Ensenada':{'Tijuana':104.2,'Mexicali':240.3},
               'Mexicali':{'Tijuana':185.5,'Ensenada':240.3}}
```

¿Que tal si eliminamos los casos repetidos?

```
>>> datos = { 'Tijuana': {'Ensenada':104.2,'Mexicali':185.5},
               'Ensenada': {'Mexicali':240.3}
               }
```

Ya que veamos operaciones con matrices, matrices escasas y la biblioteca *NumPy* vamos a regresar a estos ejemplos.

¿Y que tal una matriz para filtrado colaborativo como *DataFrame*?

	The Matrix	Matando Cabos	Mad Max: Fury Road
Ana	0	1	5
Tom	4	4	0
Joe	5	5	5
Tim	0	1	5

Haz esta matriz como ejercicio. Por lo pronto utiliza un diccionario y utiliza una matriz densa, es decir agrega todas las relaciones incluso cuando el valor es cero. Recuerda que el valor de cero en este caso significa que el usuario no ha evaluado la película.

Más adelante veremos como “girar” la matriz para tener en los renglones a las películas y en las columnas a los usuarios.

Nota:

Si te fijas, un diccionario nos puede servir perfectamente para almacenar una matriz. En algunos casos incluso será más eficiente y tenemos la ventaja de pasar el diccionario a un *DataFrame* directamente.

Operaciones básicas del *DataFrame*

Ya tenemos los datos en nuestro *DataFrame*, ¿y ahora?. En esta sección veremos como realizar operaciones básicas de consulta y manipulación de conjuntos de datos.

Vamos a trabajar con los datos de auto-mpg:

```
>>> import numpy as np
>>> import pandas as pd
>>> auto_mpg = pd.read_csv('datos-ejemplo/auto-mpg.data', sep='\s+',
    header=None, na_values='?', names=['mpg', 'cylinders', 'displacement', 'horsepower',
    'weight', 'acceleration', 'model_year', 'origin', 'car_name'],
    dtype={'mpg': 'f4', 'cylinders': 'i4',
    'displacement': 'f4', 'horsepower': 'f4', 'weight': 'f4',
    'acceleration': 'f4', 'model_year': 'i4', 'origin': 'category',
    'car_name': 'category'})
```

```
>>> auto_mpg["origin"].cat.categories = ["USA", "Japan", "Germany"]
```

Podemos ver una pequeña muestra de los **primeros** y **últimos** renglones del conjunto de datos, utilizando los métodos `head()` y `tail()`. En ambos casos el número de registros a mostrar por defecto son 5. Aunque podemos enviar como parámetro el número que deseamos ver. Veamos los tres últimos registros:

```
>>> auto_mpg.tail(3)
      mpg  cylinders  displacement  horsepower  weight  acceleration  \
395  32.0          4         135.0          84.0  2295.0          11.6
396  28.0          4         120.0          79.0  2625.0          18.6
397  31.0          4         119.0          82.0  2720.0          19.4

      model_year origin      car_name
395          82    USA  dodge rampage
396          82    USA   ford ranger
397          82    USA   chevy s-10
```

Mostrar información sobre los *índices* o nombres de renglón y las columnas:

```
>>> auto_mpg.index
RangeIndex(start=0, stop=398, step=1)
>>> auto_mpg.columns
Index([u'mpg', u'cylinders', u'displacement', u'horsepower', u'weight',
      u'acceleration', u'model_year', u'origin', u'car_name'],
      dtype='object')
```

Ver los datos almacenado internamente como *numpy.array*:

```
>>> auto_mpg.values
array([[18.0, 8, 307.0, ..., 70, 'USA', 'chevrolet chevelle malibu'],
      [15.0, 8, 350.0, ..., 70, 'USA', 'buick skylark 320'],
      [18.0, 8, 318.0, ..., 70, 'USA', 'plymouth satellite'],
      ...,
      [32.0, 4, 135.0, ..., 82, 'USA', 'dodge rampage'],
      [28.0, 4, 120.0, ..., 82, 'USA', 'ford ranger'],
      [31.0, 4, 119.0, ..., 82, 'USA', 'chevy s-10']], dtype=object)
>>>
```

Para ordenar por algún atributo se utiliza `sort_values()`, veamos los primeros registros ordenados por `car_name`:

```
>>> auto_mpg.sort_values(by='car_name').head()
      mpg  cylinders  displacement  horsepower  weight  acceleration  \
96  13.000000         8         360.0         175.0  3821.0         11.000000
9   15.000000         8         390.0         190.0  3850.0          8.500000
66  17.000000         8         304.0         150.0  3672.0         11.500000
315 24.299999         4         151.0          90.0  3003.0         20.100000
257 19.400000         6         232.0          90.0  3210.0         17.200001

      model_year  origin  car_name
96             73    USA  amc ambassador brougham
9              70    USA    amc ambassador dpl
66             72    USA    amc ambassador sst
315            80    USA      amc concord
257            78    USA      amc concord
```

Se pueden pasar varios atributos al ordenar:

```
>>> auto_mpg.sort_values(by=['origin', 'car_name']).tail()
      mpg  cylinders  displacement  horsepower  weight  acceleration  \
123 20.000000         6         156.0         122.0  2807.0         13.500000
210 19.000000         6         156.0         108.0  2930.0         15.500000
343 39.099998         4          79.0          58.0  1755.0         16.900000
348 37.700001         4          89.0          62.0  2050.0         17.299999
82  23.000000         4         120.0          97.0  2506.0         14.500000

      model_year  origin  car_name
123            73  Germany  toyota mark ii
210            76  Germany  toyota mark ii
343            81  Germany  toyota starlet
348            81  Germany  toyota tercel
82             72  Germany  toyouta corona mark ii (sw)
```

Seleccionando

Podemos utilizar tajadas (*slicing*) sobre los renglones:

```
>>> auto_mpg[2:5]
      mpg  cylinders  displacement  horsepower  weight  acceleration  \
2   18.0         8         318.0         150.0  3436.0          11.0
3   16.0         8         304.0         150.0  3433.0          12.0
4   17.0         8         302.0         140.0  3449.0          10.5
```

También podemos utilizar el nombre de la columna como un atributo del objeto `DataFrame`.

```
>>> auto_mpg.car_name[:3]
0    chevrolet chevelle malibu
1          buick skylark 320
2    plymouth satellite
Name: car_name, dtype: category
```

Podemos pasar también una lista de atributos:

```
>>> auto_mpg[['car_name', 'origin', 'model_year' ] ].head()
      car_name origin  model_year
0  chevrolet chevelle malibu    USA         70
1      buick skylark 320    USA         70
2    plymouth satellite    USA         70
3      amc rebel sst    USA         70
4      ford torino    USA         70
```

Se recomienda utilizar el atributo *loc* para hacer slicing indexado por las etiquetas. Veamos un ejemplo:

```
>>> auto_mpg.loc[3:5, 'mpg': 'weight']
      mpg  cylinders  displacement  horsepower  weight
3  16.0           8         304.0        150.0  3433.0
4  17.0           8         302.0        140.0  3449.0
5  15.0           8         429.0        198.0  4341.0
```

Es importante ver que la tajada difiere de los cortes de Python pues el resultado abarca ambas etiquetas.

También se puede pasar una lista de las etiquetas que requerimos:

```
>>> auto_mpg.loc[[3,2,5], 'mpg': 'weight']
      mpg  cylinders  displacement  horsepower  weight
3  16.0           8         304.0        150.0  3433.0
2  18.0           8         318.0        150.0  3436.0
5  15.0           8         429.0        198.0  4341.0
```

Si enviamos una lista binaria, se hace un *match* con solo las posiciones que tienen valor verdadero:

```
>>> auto_mpg.loc[[0,12,167,234], [True,True,False,False,False,True,False,True,True]].head()
      mpg  cylinders  acceleration  origin  car_name
0   18.0           8         12.0    USA  chevrolet chevelle malibu
12  15.0           8          9.5    USA    chevrolet monte carlo
167 29.0           4         16.0  Germany    toyota corolla
234 24.5           4         16.0    USA  pontiac sunbird coupe
```

Para cortes basados en la posición se utiliza *iloc*:

```
>>> auto_mpg.iloc[0:2,0:2]
      mpg  cylinders
0   18.0           8
1   15.0           8
```

En este caso los cortes funcionan igual que en Python.

Un aspecto poderoso de estas estructuras es que podemos realizar operaciones tipo *map()* de una manera muy sencilla. Por ejemplo, para calcular el doble de la columna *mpg* y regresar una nueva lista se haría lo siguiente:

```
>>> auto_mpg.mpg.head() * 2
0    36.0
1    30.0
2    36.0
3    32.0
4    34.0
```

Esta operación no se podría hacer en datos categóricos:

```
>>> auto_mpg.origin.head() * 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/mariosky/anaconda/lib/python2.7/site-packages/pandas/core/ops.py", line 721,
    result = wrap_results(safe_na_op(lvalues, rvalues))
  File "/Users/mariosky/anaconda/lib/python2.7/site-packages/pandas/core/ops.py", line 682,
    return na_op(lvalues, rvalues)
  File "/Users/mariosky/anaconda/lib/python2.7/site-packages/pandas/core/ops.py", line 672,
    op=str_rep))
TypeError: Categorical cannot perform the operation *
```

En los datos categóricos podemos utilizar métodos de cadenas mediante el atributo *str*:

```
>>> auto_mpg.car_name.str.upper().iloc[:5]
0    CHEVROLET CHEVELLE MALIBU
1           BUICK SKYLARK 320
2    PLYMOUTH SATELLITE
3           AMC REBEL SST
4           FORD TORINO
Name: car_name, dtype: object
```

Vamos a utilizar varios conceptos vistos hasta este momento para resolver un problema. Vamos a comprar un auto, pero queremos que tenga un rendimiento superior a 40 mpg.

Como primer paso vamos a generar una lista binaria con aquellos autos que cumplen con la condición:

```
>>> auto_mpg.mpg > 40
0    False
1    False
...
393  False
394    True
395  False
```

```

396     False
397     False
Name: mpg, Length: 398, dtype: bool

```

Ahora vamos a realizar un corte a partir de la lista binaria como lo habíamos hecho antes y vamos a mostrar solo cuatro atributos:

```

>>> auto_mpg[auto_mpg.mpg > 40 ].loc[ :, ['mpg','model_year','origin','car_name']]
      mpg  model_year  origin  car_name
244  43.099998      78   Japan  volkswagen rabbit custom diesel
309  41.500000      80   Japan                vw rabbit
322  46.599998      80  Germany                mazda glc
324  40.799999      80  Germany                datsun 210
325  44.299999      80   Japan  vw rabbit c (diesel)
326  43.400002      80   Japan                vw dasher (diesel)
329  44.599998      80  Germany                honda civic 1500 gl
330  40.900002      80   Japan                renault lecar deluxe
394  44.000000      82   Japan                vw pickup

```

Podemos ver que solo autos de Japón y Alemania cumplen con la condición.

Si queremos filtrar por datos categóricos podemos utilizar comprensión de listas.

Por ejemplo, para ver solo aquellos autos de Japón:

```

>>> auto_mpg[[ auto in ['Japan']  for auto in auto_mpg.origin]].loc[19:55,['mpg','model_year','car_name']]
      mpg  model_year  origin  car_name
19  26.0          70   Japan  volkswagen 1131 deluxe sedan
20  25.0          70   Japan                peugeot 504
21  24.0          70   Japan                audi 100 ls
22  25.0          70   Japan                saab 99e
23  26.0          70   Japan                bmw 2002
50  28.0          71   Japan                opel 1900
51  30.0          71   Japan                peugeot 304
52  30.0          71   Japan                fiat 124b
55  27.0          71   Japan                volkswagen model 111

```

Un detalle de este ejemplo es que al utilizar el corte por etiqueta *loc* los valores que se ponen en la especificación del rango *19:55* corresponden a la etiqueta no a la posición. Si hubiéramos puesto *0:5* para obtener los primeros 5 valores realmente nos regresaría un DataFrame vacío. Para utilizar ese estilo debemos de utilizar el atributo *iloc*.

Concatenar

Primeramente vamos a extraer a los autos de Japón y Alemania:

```

>>> japon = auto_mpg[[ auto in ['Japan']  for auto in auto_mpg.origin]]
>>> alemania = auto_mpg[[ auto in ['Germany']  for auto in auto_mpg.origin]]

```

Ahora vamos a crear un nuevo *DataFrame* con ambos conjuntos de datos:

```
>>> non_usa = pd.concat([japon, alemania])
>>> non_usa.loc[:, ['mpg', 'model_year', 'origin', 'car_name']]
```

	mpg	model_year	origin	car_name
19	26.000000	70	Japan	volkswagen 1131 deluxe sedan
20	25.000000	70	Japan	peugeot 504
21	24.000000	70	Japan	audi 100 ls
22	25.000000	70	Japan	saab 99e
23	26.000000	70	Japan	bmw 2002
50	28.000000	71	Japan	opel 1900
51	30.000000	71	Japan	peugeot 304
52	30.000000	71	Japan	fiat 124b
...				
380	36.000000	82	Germany	nissan stanza xe
381	36.000000	82	Germany	honda accord
382	34.000000	82	Germany	toyota corolla
383	38.000000	82	Germany	honda civic
384	32.000000	82	Germany	honda civic (auto)
385	38.000000	82	Germany	datsum 310 gx
390	32.000000	82	Germany	toyota celica gt

```
[149 rows x 4 columns]
```

Join estilo SQL

Aunque es preferible hacer estas operaciones en el sistema de base de datos, es posible hacer *Joins* entre estructuras *DataFrame*. Como ejemplo vamos a crear dos *DataFrames* para empleados y sus puestos.

```
>>> empleado = {'nombre': ['ana', 'tom', 'joe'],
                'email': ['ann@example.com', 'tommy@example.com',
                          'jj@example.com'],
                'puesto': [1, 1, 2] }
```

```
>>> emplado_df = pd.DataFrame(empleado)
>>> emplado_df
```

	departamento	email	nombre
0	1	ann@example.com	ana
1	1	tommy@example.com	tom
2	2	jj@example.com	joe

```
>>> puesto = {'pid': [1, 2, 3],
```



```

        'puesto':['DBA','ML','DQ'] }
>>> puesto_df = pd.DataFrame(puesto)
>>> puesto_df
   pid puesto
0    1    DBA
1    2    ML
2    3    DQ

```

Ahora vamos a hacer un *Join* para desplegar los datos del empleado y su departamento:

```

>>> pd.merge(empleado_df, puesto_df, left_on='puesto', right_on='pid')

   email nombre  puesto_x  pid puesto_y
0  ann@example.com   ana      1      1    DBA
1  tommy@example.com   tom      1      1    DBA
2   jj@example.com   joe      2      2    ML

```

Para otros tipos de Join la documentación de pandas es una buena referencia.

Agrupando

```

>>> auto_mpg.groupby('origin').count().loc[:, 'mpg']
origin
USA      249
Japan     70
Germany   79

```

Resumen

Pandas es una biblioteca para el análisis de datos en Python, que tiene como principales estructuras a *Series* y *DataFrame*. La estructura *Series* nos permite procesar datos en dimensión-1 para series de tiempo y vectores. Para los conjuntos de datos en forma tabular y de matriz de n-dimensiones se utiliza el *DataFrame*. Debes de saber como crear objetos tipo *DataFrame* ya sea leyendo de un archivo de texto o desde código en Python. También debes conocer como crear matrices para distintas aplicaciones.

Ejercicios

iris.data

1. Lee desde archivo el conjunto de datos iris.data utiliza el método `read_csv()`.
2. Nombra a los atributos como se especifica en la información del conjunto de datos: “Attribute Information:

3. sepal length in cm
4. sepal width in cm
5. petal length in cm
6. petal width in cm
7. class: – Iris Setosa – Iris Versicolour – Iris Virginica “
8. Haz una gráfica con los datos como se hizo en el ejercicio de introducción utilizando el método **plot()**.
9. Imprime cual es el ancho y largo promedio del sépalo y pétalo.

Lectura adicional

Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython por William McKinney.

Bibliografía

Documentación de la biblioteca Pandas

García-Sánchez, Pablo, Jesús González, Pedro A Castillo, Maribel García Arenas, and JJ Merelo-Guervós. 2013. “Service Oriented Evolutionary Algorithms.” *Soft Computing* 17 (6). Springer: 1059–75.