**Introducción**

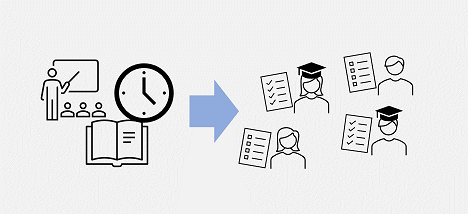
Como es lógico, el rol de un científico de datos implica principalmente la exploración y el análisis de datos. Los resultados de un análisis pueden constituir la base de un informe o de un modelo de aprendizaje automático, pero todo comienza con los datos, siendo Python el lenguaje de programación más popular entre los científicos de datos.

Tras décadas de desarrollo de código abierto, Python ofrece una amplia funcionalidad con eficaces bibliotecas estadísticas y numéricas:

* NumPy y Pandas simplifican el análisis y la manipulación de datos.
* Matplotlib proporciona visualizaciones de datos atractivas.
* Scikit-learn ofrece análisis de datos predictivo sencillo y eficaz.
* TensorFlow y PyTorch suministran funcionalidades de aprendizaje automático y aprendizaje profundo.

Normalmente, un proyecto de análisis de datos está diseñado para obtener conclusiones sobre un escenario concreto o para probar una hipótesis.

Por ejemplo, supongamos que un profesor universitario recoge datos de sus alumnos, como el número de clases a las que han asistido, las horas de estudio y la nota final obtenida en el examen de fin de curso. El profesor podría analizar los datos para determinar si existe una relación entre la cantidad de estudio que realiza un alumno y la nota final que obtiene. El profesor podría utilizar los datos para comprobar una hipótesis de que solo los alumnos que estudian un número mínimo de horas pueden esperar obtener un aprobado.



**Requisitos previos**

* Conocimientos de matemáticas básicas
* Experiencia previa con la programación en Python

**Objetivos de aprendizaje**

En este módulo, aprenderá a:

* Tareas comunes de exploración y análisis de datos.
* Uso de paquetes de Python, como NumPy, Pandas y Matplotlib, para analizar los datos

**Exploración de datos con NumPy y Pandas**

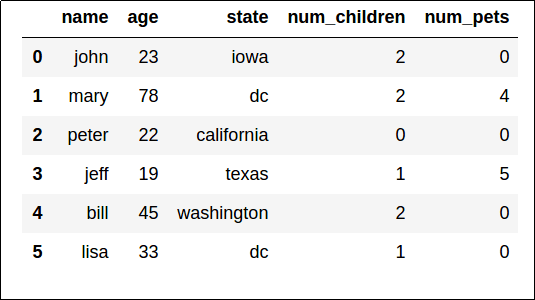
Los científicos de datos pueden usar diversas herramientas y técnicas para explorar, visualizar y manipular datos. Una de las formas más comunes en las que los científicos de datos trabajan con los datos es mediante el lenguaje de programación Python y algunos paquetes específicos para el procesamiento de datos.

**Qué es NumPy**

NumPy es una biblioteca de Python que ofrece una funcionalidad comparable a la de herramientas matemáticas como MATLAB y R. Aunque NumPy simplifica considerablemente la experiencia del usuario, también ofrece funciones matemáticas completas.

**Qué es Pandas**

Pandas es una biblioteca de Python muy conocida para el análisis y la manipulación de datos. Pandas es como el Excel de Python: proporciona una funcionalidad fácil de usar para las tablas de datos.



**Exploración de datos en un cuaderno de Jupyter Notebook**

Los cuadernos de Jupyter Notebooks son una forma conocida de ejecutar scripts básicos mediante el explorador web. Normalmente, estos cuadernos son una sola página web, dividida en secciones de texto y secciones de código que se ejecutan en el servidor en lugar de en la máquina local. Esto significa que puede empezar a trabajar rápidamente sin necesidad de instalar Python u otras herramientas.

**Prueba de hipótesis**

La exploración y el análisis de datos suele ser un proceso *iterativo* en el que el científico de datos toma una muestra de los datos y realiza las siguientes tareas para analizarlos y probar una hipótesis:

* **Limpiar los datos** para controlar errores, valores que faltan y otros problemas.
* **Aplicar técnicas estadísticas para comprender mejor los datos** y cómo se puede esperar que la muestra represente la población de datos del mundo real, teniendo en cuenta la variación aleatoria.
* **Visualizar los datos** para determinar las relaciones entre variables y, en el caso de un proyecto de aprendizaje automático, identificar las *características* que potencialmente se pueden predecir de la *etiqueta*.
* Revisión de hipótesis y repetición del proceso.

# Exploring Data with Python

A significant part of a a data scientist's role is to explore, analyze, and visualize data. There's a wide range of tools and programming languages that they can use to do this; and of the most popular approaches is to use Jupyter notebooks (like this one) and Python.

Python is a flexible programming language that is used in a wide range of scenarios; from web applications to device programming. It's extremely popular in the data science and machine learning community because of the many packages it supports for data analysis and visualization.

In this notebook, we'll explore some of these packages, and apply basic techniques to analyze data. This is not intended to be a comprehensive Python programming exercise; or even a deep dive into data analysis. Rather, it's intended as a crash course in some of the common ways in which data scientists can use Python to work with data.

**Note**: If you've never used the Jupyter Notebooks environment before, there are a few things you should be aware of:

* Notebooks are made up of cells. Some cells (like this one) contain markdown text, while others (like the one beneath this one) contain code.
* You can run each code cell by using the **► Run** button. the **► Run** button will show up when you hover over the cell.
* The output from each code cell will be displayed immediately below the cell.
* Even though the code cells can be run individually, some variables used in the code are global to the notebook. That means that you should run all of the code cells **in order**. There may be dependencies between code cells, so if you skip a cell, subsequent cells might not run correctly.

## Exploring data arrays with NumPy

Lets start by looking at some simple data.

Suppose a college takes a sample of student grades for a data science class.

Run the code in the cell below by clicking the **► Run** button to see the data.

[ ]

data·=·[50,50,47,97,49,3,53,42,26,74,82,62,37,15,70,27,36,35,48,52,63,64]

print(data)

The data has been loaded into a Python **list** structure, which is a good data type for general data manipulation, but not optimized for numeric analysis. For that, we're going to use the **NumPy** package, which includes specific data types and functions for working with Numbers in Python.

Run the cell below to load the data into a NumPy **array**.

[ ]

import numpy as np

grades = np.array(data)

print(grades)

Just in case you're wondering about the differences between a **list** and a NumPy **array**, let's compare how these data types behave when we use them in an expression that multiplies them by 2.

[ ]

print·(type(data),'x·2:',·data·\*·2)

print('---')

print·(type(grades),'x·2:',·grades·\*·2)

Note that multiplying a list by 2 creates a new list of twice the length with the original sequence of list elements repeated. Multiplying a NumPy array on the other hand performs an element-wise calculation in which the array behaves like a vector, so we end up with an array of the same size in which each element has been multiplied by 2.

The key takeaway from this is that NumPy arrays are specifically designed to support mathematical operations on numeric data - which makes them more useful for data analysis than a generic list.

You might have spotted that the class type for the numpy array above is a **numpy.ndarray**. The **nd** indicates that this is a structure that can consists of multiple dimensions (it can have n dimensions). Our specific instance has a single dimension of student grades.

Run the cell below to view the **shape** of the array.

The shape confirms that this array has only one dimension, which contains 22 elements (there are 22 grades in the original list). You can access the individual elements in the array by their zero-based ordinal position. Let's get the first element (the one in position 0).

[ ]

grades[0]

Alright, now you know your way around a NumPy array, it's time to perform some analysis of the grades data.

You can apply aggregations across the elements in the array, so let's find the simple average grade (in other words, the mean grade value).



[ ]

grades.mean()

So the mean grade is just around 50 - more or less in the middle of the possible range from 0 to 100.

Let's add a second set of data for the same students, this time recording the typical number of hours per week they devoted to studying.

[ ]

#·Define·an·array·of·study·hours

study\_hours·=·[10.0,11.5,9.0,16.0,9.25,1.0,11.5,9.0,8.5,14.5,15.5,

···············13.75,9.0,8.0,15.5,8.0,9.0,6.0,10.0,12.0,12.5,12.0]

#·Create·a·2D·array·(an·array·of·arrays)

student\_data·=·np.array([study\_hours,·grades])

#·display·the·array

student\_data

Now the data consists of a 2-dimensional array - an array of arrays. Let's look at its shape.

[ ]

#·Show·shape·of·2D·array

student\_data.shape

The **student\_data** array contains two elements, each of which is an array containing 22 elements.

To navigate this structure, you need to specify the position of each element in the hierarchy. So to find the first value in the first array (which contains the study hours data), you can use the following code.

[ ]

#·Show·the·first·element·of·the·first·element

student\_data[0][0]

Now you have a multidimensional array containing both the student's study time and grade information, which you can use to compare data. For example, how does the mean study time compare to the mean grade?

#·Get·the·mean·value·of·each·sub-array

avg\_study·=·student\_data[0].mean()

avg\_grade·=·student\_data[1].mean()

print('Average·study·hours:·{:.2f}\nAverage·grade:·{:.2f}'.format(avg\_study,·avg\_grade))

## Exploring tabular data with Pandas

While NumPy provides a lot of the functionality you need to work with numbers, and specifically arrays of numeric values; when you start to deal with two-dimensional tables of data, the **Pandas** package offers a more convenient structure to work with - the **DataFrame**.

Run the following cell to import the Pandas library and create a DataFrame with three columns. The first column is a list of student names, and the second and third columns are the NumPy arrays containing the study time and grade data.

[ ]

import pandas as pd

df\_students = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky', 'Frederic', 'Jimmie',

                                     'Rhonda', 'Giovanni', 'Francesca', 'Rajab', 'Naiyana', 'Kian', 'Jenny',

                                     'Jakeem','Helena','Ismat','Anila','Skye','Daniel','Aisha'],

                            'StudyHours':student\_data[0],

                            'Grade':student\_data[1]})

df\_students

Note that in addition to the columns you specified, the DataFrame includes an index to unique identify each row. We could have specified the index explicitly, and assigned any kind of appropriate value (for example, an email address); but because we didn't specify an index, one has been created with a unique integer value for each row.

### Finding and filtering data in a DataFrame

You can use the DataFrame's **loc** method to retrieve data for a specific index value, like this.

[ ]

# Get the data for index value 5

df\_students.loc[5]

You can also get the data at a range of index values, like this:

[ ]

# Get the rows with index values from 0 to 5

df\_students.loc[0:5]

In addition to being able to use the **loc** method to find rows based on the index, you can use the **iloc** method to find rows based on their ordinal position in the DataFrame (regardless of the index):

[ ]

# Get data in the first five rows

df\_students.iloc[0:5]

Look carefully at the iloc[0:5] results, and compare them to the loc[0:5] results you obtained previously. Can you spot the difference?

The **loc** method returned rows with index label in the list of values from 0 to 5 - which includes 0, 1, 2, 3, 4, and 5 (six rows). However, the **iloc** method returns the rows in the positions included in the range 0 to 5, and since integer ranges don't include the upper-bound value, this includes positions 0, 1, 2, 3, and 4 (five rows).

**iloc** identifies data values in a DataFrame by position, which extends beyond rows to columns. So for example, you can use it to find the values for the columns in positions 1 and 2 in row 0, like this:

[ ]

df\_students.iloc[0,[1,2]]

Let's return to the **loc** method, and see how it works with columns. Remember that **loc** is used to locate data items based on index values rather than positions. In the absence of an explicit index column, the rows in our dataframe are indexed as integer values, but the columns are identified by name:

[ ]

df\_students.loc[0,'Grade']

Here's another useful trick. You can use the **loc** method to find indexed rows based on a filtering expression that references named columns other than the index, like this:

[ ]

df\_students.loc[df\_students['Name']=='Aisha']

Actually, you don't need to explicitly use the **loc** method to do this - you can simply apply a DataFrame filtering expression, like this:

[ ]

df\_students[df\_students['Name']=='Aisha']

And for good measure, you can achieve the same results by using the DataFrame's **query** method, like this:

[ ]

df\_students.query('Name=="Aisha"')

The three previous examples underline an occassionally confusing truth about working with Pandas. Often, there are multiple ways to achieve the same results. Another example of this is the way you refer to a DataFrame column name. You can specify the column name as a named index value (as in the df\_students['Name'] examples we've seen so far), or you can use the column as a property of the DataFrame, like this:

[ ]

df\_students[df\_students.Name == 'Aisha']

### Loading a DataFrame from a file

We constructed the DataFrame from some existing arrays. However, in many real-world scenarios, data is loaded from sources such as files. Let's replace the student grades DataFrame with the contents of a text file.

[ ]

!wget https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to-machine-learning/main/Data/ml-basics/grades.csv

df\_students = pd.read\_csv('grades.csv',delimiter=',',header='infer')

df\_students.head()

The DataFrame's **read\_csv** method is used to load data from text files. As you can see in the example code, you can specify options such as the column delimiter and which row (if any) contains column headers (in this case, the delimiter is a comma and the first row contains the column names - these are the default settings, so the parameters could have been omitted).

### Handling missing values

One of the most common issues data scientists need to deal with is incomplete or missing data. So how would we know that the DataFrame contains missing values? You can use the **isnull** method to identify which individual values are null, like this:

[ ]

df\_students.isnull()

Of course, with a larger DataFrame, it would be inefficient to review all of the rows and columns individually; so we can get the sum of missing values for each column, like this:

[ ]

df\_students.isnull().sum()

So now we know that there's one missing **StudyHours** value, and two missing **Grade** values.

To see them in context, we can filter the dataframe to include only rows where any of the columns (axis 1 of the DataFrame) are null.

[ ]

df\_students[df\_students.isnull().any(axis=1)]

When the DataFrame is retrieved, the missing numeric values show up as **NaN** (not a number).

So now that we've found the null values, what can we do about them?

One common approach is to impute replacement values. For example, if the number of study hours is missing, we could just assume that the student studied for an average amount of time and replace the missing value with the mean study hours. To do this, we can use the **fillna** method, like this:

[ ]

df\_students.StudyHours = df\_students.StudyHours.fillna(df\_students.StudyHours.mean())

df\_students

Alternatively, it might be important to ensure that you only use data you know to be absolutely correct; so you can drop rows or columns that contains null values by using the **dropna** method. In this case, we'll remove rows (axis 0 of the DataFrame) where any of the columns contain null values.

[ ]

df\_students = df\_students.dropna(axis=0, how='any')

df\_students

### Explore data in the DataFrame

Now that we've cleaned up the missing values, we're ready to explore the data in the DataFrame. Let's start by comparing the mean study hours and grades.

[ ]

# Get the mean study hours using to column name as an index

mean\_study = df\_students['StudyHours'].mean()

# Get the mean grade using the column name as a property (just to make the point!)

mean\_grade = df\_students.Grade.mean()

# Print the mean study hours and mean grade

print('Average weekly study hours: {:.2f}\nAverage grade: {:.2f}'.format(mean\_study, mean\_grade))

OK, let's filter the DataFrame to find only the students who studied for more than the average amount of time.

[ ]

# Get students who studied for the mean or more hours

df\_students[df\_students.StudyHours > mean\_study]

Note that the filtered result is itself a DataFrame, so you can work with its columns just like any other DataFrame.

For example, let's find the average grade for students who undertook more than the average amount of study time.

[ ]

# What was their mean grade?

df\_students[df\_students.StudyHours > mean\_study].Grade.mean()

Let's assume that the passing grade for the course is 60.

We can use that information to add a new column to the DataFrame, indicating whether or not each student passed.

First, we'll create a Pandas **Series** containing the pass/fail indicator (True or False), and then we'll concatenate that series as a new column (axis 1) in the DataFrame.

[ ]

passes  = pd.Series(df\_students['Grade'] >= 60)

df\_students = pd.concat([df\_students, passes.rename("Pass")], axis=1)

df\_students

DataFrames are designed for tabular data, and you can use them to perform many of the kinds of data analytics operation you can do in a relational database; such as grouping and aggregating tables of data.

For example, you can use the **groupby** method to group the student data into groups based on the **Pass** column you added previously, and count the number of names in each group - in other words, you can determine how many students passed and failed.

[ ]

print(df\_students.groupby(df\_students.Pass).Name.count())

You can aggregate multiple fields in a group using any available aggregation function. For example, you can find the mean study time and grade for the groups of students who passed and failed the course.

[ ]

print(df\_students.groupby(df\_students.Pass)['StudyHours', 'Grade'].mean())

DataFrames are amazingly versatile, and make it easy to manipulate data. Many DataFrame operations return a new copy of the DataFrame; so if you want to modify a DataFrame but keep the existing variable, you need to assign the result of the operation to the existing variable. For example, the following code sorts the student data into descending order of Grade, and assigns the resulting sorted DataFrame to the original **df\_students** variable.

[ ]

# Create a DataFrame with the data sorted by Grade (descending)

df\_students = df\_students.sort\_values('Grade', ascending=False)

# Show the DataFrame

df\_students

## Summary

That's it for now!

Numpy and DataFrames are the workhorses of data science in Python. They provide us ways to load, explore, and analyze tabular data. As we will see in subsequent modules, even advanced analysis methods typically rely on Numpy and Pandas for these important roles.

In our next workbook, we'll take a look at how create graphs and explore your data in more interesting ways.

**Visualización de datos**

Los científicos de datos visualizan los datos para comprenderlos mejor. Esto puede significar examinar los datos sin procesar, las medidas de resumen, como las medias, o trazar los datos. Los gráficos son un poderoso medio de visualización de datos, ya que podemos discernir rápidamente patrones medianamente complejos sin necesidad de definir medidas matemáticas de resumen.

Representación visual de los datos

Representar los datos visualmente normalmente significa representarlos en gráficos. Esto se hace para proporcionar una evaluación cualitativa rápida de nuestros datos, que puede ser útil para entender los resultados, encontrar valores atípicos, comprender cómo se distribuyen los números, etc.

Aunque a veces sabemos de antemano qué tipo de gráfico será más útil, otras veces utilizamos los gráficos de forma exploratoria. Para entender el poder de la visualización de datos, considere los datos siguientes: la ubicación (x,y) de un coche que se conduce automáticamente. En su forma sin procesar, es difícil ver patrones reales. La media o promedio, nos dice que su trayectoria giró en torno a x=0,2 e y=0,3, y el intervalo de números parece estar entre -2 y 2 aproximadamente.

Time Ubicación X Ubicación Y

0 0 2

1 1,682942 1,080605

2 1,818595 -0,83229

3 0,28224 -1,97998

4 -1,5136 -1,30729

5 -1,91785 0,567324

6 -0,55883 1,920341

7 1,313973 1,507805

12 0,00001 0,00001

13 0,840334 1,814894

14 1,981215 0,273474

15 1,300576 -1,51938

16 -0,57581 -1,91532

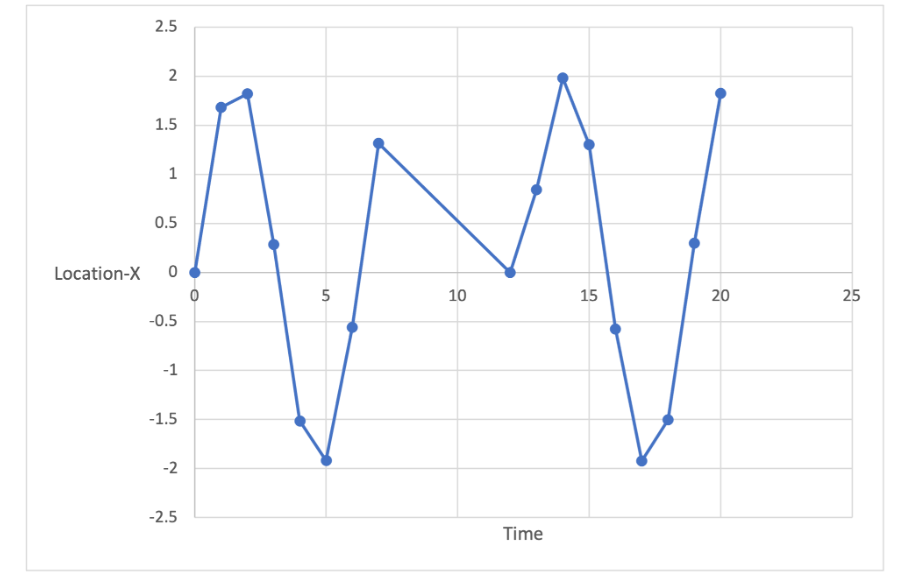
17 -1,92279 -0,55033

18 -1,50197 1,320633

19 0,299754 1,977409

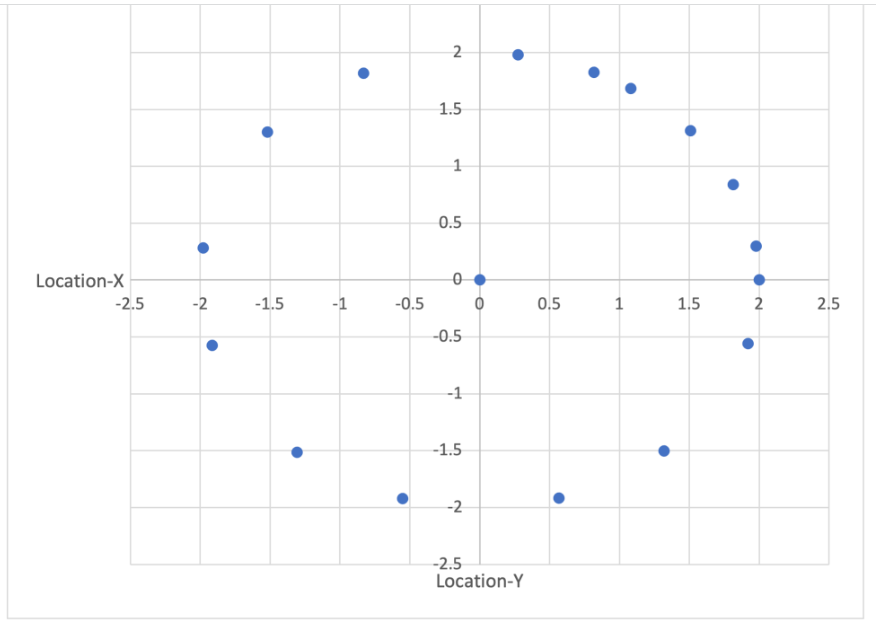
20 1,825891 0,816164

Si ahora trazamos la ubicación X a lo largo del tiempo, podemos ver que parece que tenemos algunos valores perdidos entre los tiempos 7 y 12.



Location-X coordinates plotted against time.

Si trazamos X frente a Y, terminamos con un mapa de por dónde se ha movido el coche. Es inmediatamente obvio que el coche ha estado conduciendo en círculo, pero en algún momento condujo hacia el centro de ese círculo.



Location-X and Location-Y coordinates plotted.

Los gráficos no se limitan a los diagramas de dispersión en 2D como los anteriores, sino que pueden utilizarse para explorar otros tipos de datos, como las proporciones (que se muestran a través de gráficos circulares, gráficos de barras apilados), cómo se distribuyen los datos (con histogramas, diagramas de cajas) y cómo difieren dos conjuntos de datos. A menudo, cuando intentamos comprender datos o resultados sin procesar, podemos experimentar con diferentes tipos de gráficos hasta dar con uno que explique los datos de forma visualmente intuitiva.

**Ejercicio: Visualización de datos con Matplotlib**

Para completar este módulo, se necesita un espacio aislado.

Un espacio aislado le da acceso a recursos gratuitos. La suscripción personal no se le cobrará. El espacio aislado solo se puede usar para realizar los cursos de Microsoft Learn. Está prohibido el uso con cualquier otro fin y puede dar lugar a la pérdida permanente del acceso al espacio aislado.

Microsoft proporciona esta experiencia de laboratorio y contenido relacionado con fines educativos. Toda la información proporcionada es propiedad de Microsoft y está pensada únicamente para aprender a usar los productos y servicios que cubre este módulo de Microsoft Learn.

Las sugerencias de enlaces de teclado de la barra de herramientas están ahora ocultas

Edición

Exploring data with Python - visualize data

In this notebook, we'll apply basic techniques to analyze data with basic statistics and visualise using graphs.

Loading our data

Before we begin, lets load the same data about study hours that we analysed in the previous notebook. We will also recalculate who passed in the same way as last time Run the code in the cell below by clicking the ► Run button to see the data.

import pandas as pd

# Load data from a text file

!wget https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to-machine-learning/main/Data/ml-basics/grades.csv

df\_students = pd.read\_csv('grades.csv',delimiter=',',header='infer')

# Remove any rows with missing data

df\_students = df\_students.dropna(axis=0, how='any')

# Calculate who passed, assuming '60' is the grade needed to pass

passes = pd.Series(df\_students['Grade'] >= 60)

# Save who passed to the Pandas dataframe

df\_students = pd.concat([df\_students, passes.rename("Pass")], axis=1)

# Print the result out into this notebook

df\_students

Visualizing data with Matplotlib

DataFrames provide a great way to explore and analyze tabular data, but sometimes a picture is worth a thousand rows and columns. The Matplotlib library provides the foundation for plotting data visualizations that can greatly enhance your ability to analyze the data.

Let's start with a simple bar chart that shows the grade of each student.

# Ensure plots are displayed inline in the notebook

%matplotlib inline

from matplotlib import pyplot as plt

# Create a bar plot of name vs grade

plt.bar(x=df\_students.Name, height=df\_students.Grade)

# Display the plot

plt.show()

Well, that worked; but the chart could use some improvements to make it clearer what we're looking at.

Note that you used the pyplot class from Matplotlib to plot the chart. This class provides a whole bunch of ways to improve the visual elements of the plot. For example, the following code:

Specifies the color of the bar chart.

Adds a title to the chart (so we know what it represents)

Adds labels to the X and Y (so we know which axis shows which data)

Adds a grid (to make it easier to determine the values for the bars)

Rotates the X markers (so we can read them)

# Create a bar plot of name vs grade

plt.bar(x=df\_students.Name, height=df\_students.Grade, color='orange')

# Customize the chart

plt.title('Student Grades')

plt.xlabel('Student')

plt.ylabel('Grade')

plt.grid(color='#95a5a6', linestyle='--', linewidth=2, axis='y', alpha=0.7)

plt.xticks(rotation=90)

# Display the plot

plt.show()

A plot is technically contained with a Figure. In the previous examples, the figure was created implicitly for you; but you can create it explicitly. For example, the following code creates a figure with a specific size.

# Create a Figure

fig = plt.figure(figsize=(8,3))

# Create a bar plot of name vs grade

plt.bar(x=df\_students.Name, height=df\_students.Grade, color='orange')

# Customize the chart

plt.title('Student Grades')

plt.xlabel('Student')

plt.ylabel('Grade')

plt.grid(color='#95a5a6', linestyle='--', linewidth=2, axis='y', alpha=0.7)

plt.xticks(rotation=90)

# Show the figure

plt.show()

A figure can contain multiple subplots, each on its own axis.

For example, the following code creates a figure with two subplots - one is a bar chart showing student grades, and the other is a pie chart comparing the number of passing grades to non-passing grades.

# Create a figure for 2 subplots (1 row, 2 columns)

fig, ax = plt.subplots(1, 2, figsize = (10,4))

# Create a bar plot of name vs grade on the first axis

ax[0].bar(x=df\_students.Name, height=df\_students.Grade, color='orange')

ax[0].set\_title('Grades')

ax[0].set\_xticklabels(df\_students.Name, rotation=90)

# Create a pie chart of pass counts on the second axis

pass\_counts = df\_students['Pass'].value\_counts()

ax[1].pie(pass\_counts, labels=pass\_counts)

ax[1].set\_title('Passing Grades')

ax[1].legend(pass\_counts.keys().tolist())

# Add a title to the Figure

fig.suptitle('Student Data')

# Show the figure

fig.show()

Until now, you've used methods of the Matplotlib.pyplot object to plot charts. However, Matplotlib is so foundational to graphics in Python that many packages, including Pandas, provide methods that abstract the underlying Matplotlib functions and simplify plotting. For example, the DataFrame provides its own methods for plotting data, as shown in the following example to plot a bar chart of study hours.

df\_students.plot.bar(x='Name', y='StudyHours', color='teal', figsize=(6,4))

Getting started with statistical analysis

Now that you know how to use Python to manipulate and visualize data, you can start analyzing it.

A lot of data science is rooted in statistics, so we'll explore some basic statistical techniques.

Note: This is not intended to teach you statistics - that's much too big a topic for this notebook. It will however introduce you to some statistical concepts and techniques that data scientists use as they explore data in preparation for machine learning modeling.

Descriptive statistics and data distribution

When examining a variable (for example a sample of student grades), data scientists are particularly interested in its distribution (in other words, how are all the different grade values spread across the sample). The starting point for this exploration is often to visualize the data as a histogram, and see how frequently each value for the variable occurs.

# Get the variable to examine

var\_data = df\_students['Grade']

# Create a Figure

fig = plt.figure(figsize=(10,4))

# Plot a histogram

plt.hist(var\_data)

# Add titles and labels

plt.title('Data Distribution')

plt.xlabel('Value')

plt.ylabel('Frequency')

# Show the figure

fig.show()

The histogram for grades is a symmetric shape, where the most frequently occurring grades tend to be in the middle of the range (around 50), with fewer grades at the extreme ends of the scale.

Measures of central tendency

To understand the distribution better, we can examine so-called measures of central tendency; which is a fancy way of describing statistics that represent the "middle" of the data. The goal of this is to try to find a "typical" value. Common ways to define the middle of the data include:

The mean: A simple average based on adding together all of the values in the sample set, and then dividing the total by the number of samples.

The median: The value in the middle of the range of all of the sample values.

The mode: The most commonly occuring value in the sample set\*.

Let's calculate these values, along with the minimum and maximum values for comparison, and show them on the histogram.

\*Of course, in some sample sets , there may be a tie for the most common value - in which case the dataset is described as bimodal or even multimodal.

# Get the variable to examine

var = df\_students['Grade']

# Get statistics

min\_val = var.min()

max\_val = var.max()

mean\_val = var.mean()

med\_val = var.median()

mod\_val = var.mode()[0]

…plt.title('Data Distribution')

plt.xlabel('Value')

plt.ylabel('Frequency')

# Show the figure

fig.show()

For the grade data, the mean, median, and mode all seem to be more or less in the middle of the minimum and maximum, at around 50.

Another way to visualize the distribution of a variable is to use a box plot (sometimes called a box-and-whiskers plot). Let's create one for the grade data.

# Get the variable to examine

var = df\_students['Grade']

# Create a Figure

fig = plt.figure(figsize=(10,4))

# Plot a histogram

plt.boxplot(var)

# Add titles and labels

plt.title('Data Distribution')

# Show the figure

fig.show()

The box plot shows the distribution of the grade values in a different format to the histogram. The box part of the plot shows where the inner two quartiles of the data reside - so in this case, half of the grades are between approximately 36 and 63. The whiskers extending from the box show the outer two quartiles; so the other half of the grades in this case are between 0 and 36 or 63 and 100. The line in the box indicates the median value.

For learning, it can be useful to combine histograms and box plots, with the box plot's orientation changed to align it with the histogram (in some ways, it can be helpful to think of the histogram as a "front elevation" view of the distribution, and the box plot as a "plan" view of the distribution from above.)

# Create a function that we can re-use

def show\_distribution(var\_data):

from matplotlib import pyplot as plt

# Get statistics

min\_val = var\_data.min()

max\_val = var\_data.max()

mean\_val = var\_data.mean()

med\_val = var\_data.median()

mod\_val = var\_data.mode()[0]

…

# Get the variable to examine

col = df\_students['Grade']

# Call the function

show\_distribution(col)

All of the measurements of central tendency are right in the middle of the data distribution, which is symmetric with values becoming progressively lower in both directions from the middle.

To explore this distribution in more detail, you need to understand that statistics is fundamentally about taking samples of data and using probability functions to extrapolate information about the full population of data.

What does this mean? Samples refer to the data we have on hand - such as information about these 22 students' study habits and grades. The population refers to all possible data we could collect - such as every student's grades and study habits across every educational institution throughout the history of time. Usually we're interested in the population but it's simply not practical to collect all of that data. Instead, we need to try estimate what the population is like from the small amount of data (samples) that we have.

If we have enough samples, we can calculate something called a probability density function, which estimates the distribution of grades for the full population.

The pyplot class from Matplotlib provides a helpful plot function to show this density.

def show\_density(var\_data):

from matplotlib import pyplot as plt

fig = plt.figure(figsize=(10,4))

# Plot density

var\_data.plot.density()

# Add titles and labels

plt.title('Data Density')

…col = df\_students['Grade']

show\_density(col)

As expected from the histogram of the sample, the density shows the characteristic "bell curve" of what statisticians call a normal distribution with the mean and mode at the center and symmetric tails.

Summary

Well done! There were a number of new concepts in here, so let's summarise.

Here we have:

Made graphs with matplotlib

Seen how to customise these graphs

Calculated basic statistics, such as medians

Looked at the spread of data using box plots and histograms

Learned about samples vs populations

Estimated what the population of graphse might look like from a sample of grades.

In our next notebook we will look at spotting unusual data, and finding relationships between data.

Further Reading

To learn more about the Python packages you explored in this notebook, see the following documentation:

NumPy

Pandas

Matplotlib

# Examen de datos del mundo real

Los datos presentados en el material educativo suelen ser notablemente perfectos, diseñados para mostrar a los alumnos cómo encontrar relaciones claras entre las variables. Los datos del mundo real son algo menos sencillos.

Debido a la complejidad de los datos del "mundo real", hay que inspeccionar los datos sin procesar para detectar problemas antes de utilizarlos.

Por ello, el procedimiento recomendado es inspeccionar los datos sin procesar y procesarlos antes de utilizarlos, lo que reduce los errores o problemas, normalmente eliminando los puntos de datos erróneos o modificando los datos para que sean más útiles.

## Problemas de los datos del mundo real

Los datos del mundo real pueden contener muchos problemas diferentes que pueden afectar a la utilidad de los datos y a nuestra interpretación de los resultados.

Es importante tener en cuenta que la mayoría de los datos del mundo real están influenciados por factores que no se registraron en ese momento. Por ejemplo, podríamos tener una tabla con los tiempos de los coches de carreras junto con los tamaños de los motores, pero otros factores que no se anotaron, como el clima, probablemente también influyeron. Si son problemáticos, la influencia de estos factores puede reducirse a menudo aumentando el tamaño del conjunto de datos.

En otras situaciones, los puntos de datos que están claramente fuera de lo esperado, también conocidos como valores atípicos, a veces se pueden quitar de forma segura de los análisis, aunque se debe tener cuidado para no quitar puntos de datos que proporcionen información real.

Otro problema común en los datos del mundo real es el sesgo. El sesgo se refiere a la tendencia a seleccionar ciertos tipos de valores con más frecuencia que otros, de forma que se falsea la población subyacente, o el "mundo real". A veces se puede identificar el sesgo explorando los datos y teniendo en cuenta los conocimientos básicos sobre la procedencia de estos.

Recuerde que los datos del mundo real siempre tendrán problemas, pero esto suele ser un problema superable. Recuerde:

* Compruebe los valores que faltan y los datos registrados de forma incorrecta.
* Considere la posibilidad de eliminar valores atípicos obvios.
* Considere qué factores del mundo real podrían afectar al análisis y vea si el tamaño del conjunto de datos es lo suficientemente grande como para controlarlos.
* Compruebe si los datos sin procesar están sesgados y considere sus opciones para corregirlos, si se encuentran.

**Ejercicio: Examen de los datos del mundo real**

Para completar este módulo, se necesita un espacio aislado.

Un espacio aislado le da acceso a recursos gratuitos. La suscripción personal no se le cobrará. El espacio aislado solo se puede usar para realizar los cursos de Microsoft Learn. Está prohibido el uso con cualquier otro fin y puede dar lugar a la pérdida permanente del acceso al espacio aislado.

Microsoft proporciona esta experiencia de laboratorio y contenido relacionado con fines educativos. Toda la información proporcionada es propiedad de Microsoft y está pensada únicamente para aprender a usar los productos y servicios que cubre este módulo de Microsoft Learn.

Las sugerencias de enlaces de teclado de la barra de herramientas están ahora ocultas

GuardarGuardar

Edición

Exploring data with Python - real world data

Last time, we looked at grades for our student data, and investigated this visually with histograms and box plots. Now we will look into more complex cases, describe the data more fully, and discuss how to make basic comparisons between data.

Real world data distributions

Last time, we looked at grades for our student data, and estimated from this sample what the full population of grades might look like. Just to refresh, lets take a look at this data again.

Run the code below to print out the data and make a histogram + boxplot that show the grades for our sample of students.

import pandas as pd

from matplotlib import pyplot as plt

# Load data from a text file

!wget https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to-machine-learning/main/Data/ml-basics/grades.csv

df\_students = pd.read\_csv('grades.csv',delimiter=',',header='infer')

# Remove any rows with missing data

df\_students = df\_students.dropna(axis=0, how='any')

…

# Show the figure

fig.show()

show\_distribution(df\_students['Grade'])

As you might recall, our data had the mean and mode at the center, with data spread symmetrically from there.

Now let's take a look at the distribution of the study hours data.

# Get the variable to examine

col = df\_students['StudyHours']

# Call the function

show\_distribution(col)

Presione Mayús + Entrar para ejecutar

The distribution of the study time data is significantly different from that of the grades.

Note that the whiskers of the box plot only begin at around 6.0, indicating that the vast majority of the first quarter of the data is above this value. The minimum is marked with an o, indicating that it is statistically an outlier - a value that lies significantly outside the range of the rest of the distribution.

Outliers can occur for many reasons. Maybe a student meant to record "10" hours of study time, but entered "1" and missed the "0". Or maybe the student was abnormally lazy when it comes to studying! Either way, it's a statistical anomaly that doesn't represent a typical student. Let's see what the distribution looks like without it.

# Get the variable to examine

# We will only get students who have studied more than one hour

col = df\_students[df\_students.StudyHours>1]['StudyHours']

# Call the function

show\_distribution(col)

For learning purposes we have just treated the value 1 is a true outlier here and excluded it. In the real world, though, it would be unusual to exclude data at the extremes without more justification when our sample size is so small. This is because the smaller our sample size, the more likely it is that our sampling is a bad representation of the whole population (here, the population means grades for all students, not just our 22). For example, if we sampled study time for another 1000 students, we might find that it's actually quite common to not study much!

When we have more data available, our sample becomes more reliable. This makes it easier to consider outliers as being values that fall below or above percentiles within which most of the data lie. For example, the following code uses the Pandas quantile function to exclude observations below the 0.01th percentile (the value above which 99% of the data reside).

# calculate the 0.01th percentile

q01 = df\_students.StudyHours.quantile(0.01)

# Get the variable to examine

col = df\_students[df\_students.StudyHours>q01]['StudyHours']

# Call the function

show\_distribution(col)

Tip: You can also eliminate outliers at the upper end of the distribution by defining a threshold at a high percentile value - for example, you could use the quantile function to find the 0.99 percentile below which 99% of the data reside.

With the outliers removed, the box plot shows all data within the four quartiles. Note that the distribution is not symmetric like it is for the grade data though - there are some students with very high study times of around 16 hours, but the bulk of the data is between 7 and 13 hours; The few extremely high values pull the mean towards the higher end of the scale.

Let's look at the density for this distribution.

def show\_density(var\_data):

fig = plt.figure(figsize=(10,4))

# Plot density

var\_data.plot.density()

# Add titles and labels

plt.title('Data Density')

# Show the mean, median, and mode

plt.axvline(x=var\_data.mean(), color = 'cyan', linestyle='dashed', linewidth = 2)

plt.axvline(x=var\_data.median(), color = 'red', linestyle='dashed', linewidth = 2)

plt.axvline(x=var\_data.mode()[0], color = 'yellow', linestyle='dashed', linewidth = 2)

# Show the figure

plt.show()

# Get the density of StudyHours

show\_density(col)

This kind of distribution is called right skewed. The mass of the data is on the left side of the distribution, creating a long tail to the right because of the values at the extreme high end; which pull the mean to the right.

Measures of variance

So now we have a good idea where the middle of the grade and study hours data distributions are. However, there's another aspect of the distributions we should examine: how much variability is there in the data?

Typical statistics that measure variability in the data include:

Range: The difference between the maximum and minimum. There's no built-in function for this, but it's easy to calculate using the min and max functions.

Variance: The average of the squared difference from the mean. You can use the built-in var function to find this.

Standard Deviation: The square root of the variance. You can use the built-in std function to find this.

for col\_name in ['Grade','StudyHours']:

col = df\_students[col\_name]

rng = col.max() - col.min()

var = col.var()

std = col.std()

print('\n{}:\n - Range: {:.2f}\n - Variance: {:.2f}\n - Std.Dev: {:.2f}'.format(col\_name, rng, var, std))

Of these statistics, the standard deviation is generally the most useful. It provides a measure of variance in the data on the same scale as the data itself (so grade points for the Grade distribution and hours for the StudyHours distribution). The higher the standard deviation, the more variance there is when comparing values in the distribution to the distribution mean - in other words, the data is more spread out.

When working with a normal distribution, the standard deviation works with the particular characteristics of a normal distribution to provide even greater insight. Run the cell below to see the relationship between standard deviations and the data in the normal distribution.

plt.plot(x3,y3, color='orange')

plt.annotate('3 std (99.73%)', (x3[1],y3[1]))

# Show the location of the mean

plt.axvline(col.mean(), color='cyan', linestyle='dashed', linewidth=1)

plt.axis('off')

plt.show()

The horizontal lines show the percentage of data within 1, 2, and 3 standard deviations of the mean (plus or minus).

In any normal distribution:

Approximately 68.26% of values fall within one standard deviation from the mean.

Approximately 95.45% of values fall within two standard deviations from the mean.

Approximately 99.73% of values fall within three standard deviations from the mean.

So, since we know that the mean grade is 49.18, the standard deviation is 21.74, and distribution of grades is approximately normal; we can calculate that 68.26% of students should achieve a grade between 27.44 and 70.92.

The descriptive statistics we've used to understand the distribution of the student data variables are the basis of statistical analysis; and because they're such an important part of exploring your data, there's a built-in Describe method of the DataFrame object that returns the main descriptive statistics for all numeric columns.

df\_students.describe()

Comparing data

Now that you know something about the statistical distribution of the data in your dataset, you're ready to examine your data to identify any apparent relationships between variables.

First of all, let's get rid of any rows that contain outliers so that we have a sample that is representative of a typical class of students. We identified that the StudyHours column contains some outliers with extremely low values, so we'll remove those rows.

df\_sample = df\_students[df\_students['StudyHours']>1]

df\_sample

Comparing numeric and categorical variables

The data includes two numeric variables (StudyHours and Grade) and two categorical variables (Name and Pass). Let's start by comparing the numeric StudyHours column to the categorical Pass column to see if there's an apparent relationship between the number of hours studied and a passing grade.

To make this comparison, let's create box plots showing the distribution of StudyHours for each possible Pass value (true and false).

df\_sample.boxplot(column='StudyHours', by='Pass', figsize=(8,5))

Comparing the StudyHours distributions, it's immediately apparent (if not particularly surprising) that students who passed the course tended to study for more hours than students who didn't. So if you wanted to predict whether or not a student is likely to pass the course, the amount of time they spend studying may be a good predictive feature.

Comparing numeric variables

Now let's compare two numeric variables. We'll start by creating a bar chart that shows both grade and study hours.

# Create a bar plot of name vs grade and study hours

df\_sample.plot(x='Name', y=['Grade','StudyHours'], kind='bar', figsize=(8,5))

The chart shows bars for both grade and study hours for each student; but it's not easy to compare because the values are on different scales. Grades are measured in grade points, and range from 3 to 97; while study time is measured in hours and ranges from 1 to 16.

A common technique when dealing with numeric data in different scales is to normalize the data so that the values retain their proportional distribution, but are measured on the same scale. To accomplish this, we'll use a technique called MinMax scaling that distributes the values proportionally on a scale of 0 to 1. You could write the code to apply this transformation; but the Scikit-Learn library provides a scaler to do it for you.

from sklearn.preprocessing import MinMaxScaler

# Get a scaler object

scaler = MinMaxScaler()

# Create a new dataframe for the scaled values

df\_normalized = df\_sample[['Name', 'Grade', 'StudyHours']].copy()

# Normalize the numeric columns

df\_normalized[['Grade','StudyHours']] = scaler.fit\_transform(df\_normalized[['Grade','StudyHours']])

# Plot the normalized values

df\_normalized.plot(x='Name', y=['Grade','StudyHours'], kind='bar', figsize=(8,5))

With the data normalized, it's easier to see an apparent relationship between grade and study time. It's not an exact match, but it definitely seems like students with higher grades tend to have studied more.

So there seems to be a correlation between study time and grade; and in fact, there's a statistical correlation measurement we can use to quantify the relationship between these columns.

df\_normalized.Grade.corr(df\_normalized.StudyHours)

The correlation statistic is a value between -1 and 1 that indicates the strength of a relationship. Values above 0 indicate a positive correlation (high values of one variable tend to coincide with high values of the other), while values below 0 indicate a negative correlation (high values of one variable tend to coincide with low values of the other). In this case, the correlation value is close to 1; showing a strongly positive correlation between study time and grade.

Note: Data scientists often quote the maxim "correlation is not causation". In other words, as tempting as it might be, you shouldn't interpret the statistical correlation as explaining why one of the values is high. In the case of the student data, the statistics demonstrates that students with high grades tend to also have high amounts of study time; but this is not the same as proving that they achieved high grades because they studied a lot. The statistic could equally be used as evidence to support the nonsensical conclusion that the students studied a lot because their grades were going to be high.

Another way to visualise the apparent correlation between two numeric columns is to use a scatter plot.

# Create a scatter plot

df\_sample.plot.scatter(title='Study Time vs Grade', x='StudyHours', y='Grade')

Again, it looks like there's a discernible pattern in which the students who studied the most hours are also the students who got the highest grades.

We can see this more clearly by adding a regression line (or a line of best fit) to the plot that shows the general trend in the data. To do this, we'll use a statistical technique called least squares regression.

Warning - Math Ahead!Cast your mind back to when you were learning how to solve linear equations in school, and recall that the slope-intercept form of a linear equation looks like this:

Y = mx + b

In this equation, y and x are the coordinate variables, m is the slope of the line, and b is the y-intercept (where the line goes through the Y-axis).In the case of our scatter plot for our student data, we already have our values for x (StudyHours) and y (Grade), so we just need to calculate the intercept and slope of the straight line that lies closest to those points. Then we can form a linear equation that calculates a new y value on that line for each of our x (StudyHours) values - to avoid confusion, we'll call this new y value f(x) (because it's the output from a linear equation function based on x). The difference between the original y (Grade) value and the f(x) value is the error between our regression line and the actual Grade achieved by the student. Our goal is to calculate the slope and intercept for a line with the lowest overall error.Specifically, we define the overall error by taking the error for each point, squaring it, and adding all the squared errors together. The line of best fit is the line that gives us the lowest value for the sum of the squared errors - hence the name least squares regression.

Fortunately, you don't need to code the regression calculation yourself - the SciPy package includes a stats class that provides a linregress method to do the hard work for you. This returns (among other things) the coefficients you need for the slope equation - slope (m) and intercept (b) based on a given pair of variable samples you want to compare.

from scipy import stats

#

df\_regression = df\_sample[['Grade', 'StudyHours']].copy()

# Get the regression slope and intercept

m, b, r, p, se = stats.linregress(df\_regression['StudyHours'], df\_regression['Grade'])

print('slope: {:.4f}\ny-intercept: {:.4f}'.format(m,b))

print('so...\n f(x) = {:.4f}x + {:.4f}'.format(m,b))

…plt.plot(df\_regression['StudyHours'],df\_regression['fx'], color='cyan')

# Display the plot

plt.show()

Note that this time, the code plotted two distinct things - the scatter plot of the sample study hours and grades is plotted as before, and then a line of best fit based on the least squares regression coefficients is plotted.

The slope and intercept coefficients calculated for the regression line are shown above the plot.

The line is based on the f(x) values calculated for each StudyHours value. Run the following cell to see a table that includes the following values:

The StudyHours for each student.

The Grade achieved by each student.

The f(x) value calculated using the regression line coefficients.

The error between the calculated f(x) value and the actual Grade value.

Some of the errors, particularly at the extreme ends, and quite large (up to over 17.5 grade points); but in general, the line is pretty close to the actual grades.

# Show the original x,y values, the f(x) value, and the error

df\_regression[['StudyHours', 'Grade', 'fx', 'error']]

Using the regression coefficients for prediction

Now that you have the regression coefficients for the study time and grade relationship, you can use them in a function to estimate the expected grade for a given amount of study.

# Define a function based on our regression coefficients

def f(x):

m = 6.3134

b = -17.9164

return m\*x + b

study\_time = 14

# Get f(x) for study time

prediction = f(study\_time)

# Grade can't be less than 0 or more than 100

expected\_grade = max(0,min(100,prediction))

#Print the estimated grade

print ('Studying for {} hours per week may result in a grade of {:.0f}'.format(study\_time, expected\_grade))

So by applying statistics to sample data, you've determined a relationship between study time and grade; and encapsulated that relationship in a general function that can be used to predict a grade for a given amount of study time.

This technique is in fact the basic premise of machine learning. You can take a set of sample data that includes one or more features (in this case, the number of hours studied) and a known label value (in this case, the grade achieved) and use the sample data to derive a function that calculates predicted label values for any given set of features.

Summary

Here we've looked at:

What an outlier is and how to remove them

How data can be skewed

How to look at the spread of data

Basic ways to compare variables, such as grades and study time

Further Reading

To learn more about the Python packages you explored in this notebook, see the following documentation:

NumPy

Pandas

Matplotlib

**Resumen**

En este módulo, ha aprendido a usar Python para explorar, visualizar y manipular datos. La exploración de datos es la base de la ciencia de datos y un elemento clave en el análisis de datos y el aprendizaje automático.

El aprendizaje automático es un subconjunto de la ciencia de datos que se ocupa del modelado predictivo. En otras palabras, el aprendizaje automático usa datos para crear modelos predictivos, con el fin de predecir valores desconocidos. Podría utilizar el aprendizaje automático para predecir la cantidad de alimentos que debe pedir un supermercado, o para identificar plantas en fotografías.

Lo que hace el aprendizaje automático es identificar las relaciones entre los valores de datos que describen las propiedades de algo, sus características, como la altura y el color de una planta, y el valor que se quiere predecir, la etiqueta, como las especies de plantas. Estas relaciones se integran en un modelo a través de un proceso de entrenamiento.

Reto: Análisis de datos de vuelos

Si los ejercicios de este módulo le han inspirado para intentar explorar los datos por sí mismo, ¿por qué no acepta el reto de un conjunto de datos del mundo real que contenga registros de vuelos del Departamento de Transporte de Estados Unidos? Encontrará el reto en el cuaderno 01 - Flights Challenge.ipynb.

https://raw.githubusercontent.com/MicrosoftDocs/ml-basics/master/challenges/data/flights.csv

**Flights Data Exploration Challenge**

In this challenge, you'll explore a real-world dataset containing flights data from the US Department of Transportation.

Let's start by loading and viewing the data.

import pandas as pd

df\_flights = pd.read\_csv('data/flights.csv')

df\_flights.head()

The dataset contains observations of US domestic flights in 2013, and consists of the following fields:

Year: The year of the flight (all records are from 2013)

Month: The month of the flight

DayofMonth: The day of the month on which the flight departed

DayOfWeek: The day of the week on which the flight departed - from 1 (Monday) to 7 (Sunday)

Carrier: The two-letter abbreviation for the airline.

OriginAirportID: A unique numeric identifier for the departure aiport

OriginAirportName: The full name of the departure airport

OriginCity: The departure airport city

OriginState: The departure airport state

DestAirportID: A unique numeric identifier for the destination aiport

DestAirportName: The full name of the destination airport

DestCity: The destination airport city

DestState: The destination airport state

CRSDepTime: The scheduled departure time

DepDelay: The number of minutes departure was delayed (flight that left ahead of schedule have a negative value)

DelDelay15: A binary indicator that departure was delayed by more than 15 minutes (and therefore considered "late")

CRSArrTime: The scheduled arrival time

ArrDelay: The number of minutes arrival was delayed (flight that arrived ahead of schedule have a negative value)

ArrDelay15: A binary indicator that arrival was delayed by more than 15 minutes (and therefore considered "late")

Cancelled: A binary indicator that the flight was cancelled

Your challenge is to explore the flight data to analyze possible factors that affect delays in departure or arrival of a flight.

Start by cleaning the data.

Identify any null or missing data, and impute appropriate replacement values.

Identify and eliminate any outliers in the DepDelay and ArrDelay columns.

Explore the cleaned data.

View summary statistics for the numeric fields in the dataset.

Determine the distribution of the DepDelay and ArrDelay columns.

Use statistics, aggregate functions, and visualizations to answer the following questions:

What are the average (mean) departure and arrival delays?

How do the carriers compare in terms of arrival delay performance?

Is there a noticable difference in arrival delays for different days of the week?

Which departure airport has the highest average departure delay?

Do late departures tend to result in longer arrival delays than on-time departures?

Which route (from origin airport to destination airport) has the most late arrivals?

Which route has the highest average arrival delay?

Add markdown and code cells as required to create your solution.

Note: There is no single "correct" solution. A sample solution is provided in 01 - Flights Challenge.ipynb.