Module 1: Introduction to Machine Learning

Quick Reference Guide

Learning Outcomes:

1. Identify examples of the mystery box metaphor within your industry or personal experience.
2. Establish the correct ML notation and concepts.
3. Determine measures of discrete and continuous probability functions.
4. Distinguish between measures of central tendency.
5. Search for two data sources online and examine their characteristics.
6. Identify functions and codes related to DataFrames.
7. Load data into a DataFrame using pandas.
8. Analyze data using selection and statistical techniques.
9. Analyze different data visualization techniques.
10. Create histograms and data visualizations.

Introduction to Machine Learning

Defining artificial intelligence

An intelligent agent, whether human, machine or other, is one that can find efficient solutions to problems by consulting a model of the world that is learned from interactions with the environment.

The problem should be a task for which you can quantify your performance in either a discrete or continuous sense. The model, a mathematical representation of the problem, is used to estimate the quality of alternative answers.

Machine learning

Whenever the agent interacts with the environment, the environment responds with some data, which the agent uses to update and improve its model. Machine learning is the study of all the different ways in which models can be built from data. Machine learning offers algorithms for processing data that construct models of a wide variety of phenomena.

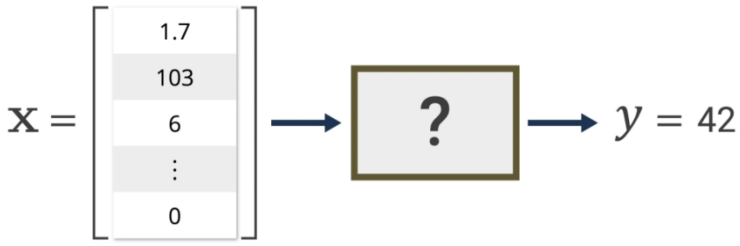
The more complex tasks require more:

- Complicated models to represent it
- Data to train the model
- Computational power to process the data

The mystery box game

Consider a game in which you are presented with a mystery box with two slots: input and output. When a list of numbers is placed into the input, a single number is output. Without understanding its inner workings, your task is to build a mathematical model that can be used to predict the output for any given input.

A bold letter $\boldsymbol{x}$ is used for the input. It is bold to indicate that it is a list of numbers. This list is also referred to as a vector, an array, or a tuple. The letter $\boldsymbol{y}$ is used for the single number, or scalar, that is output.

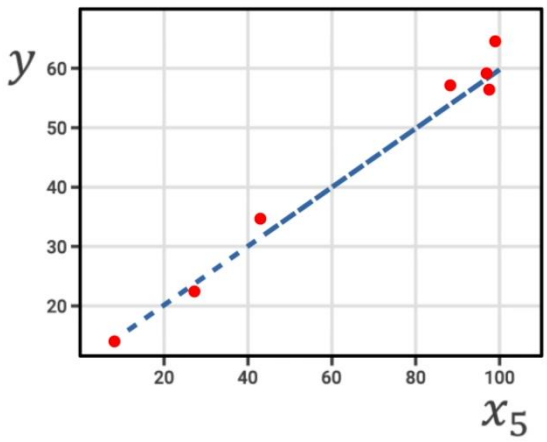


The equation for the mystery box model is $\hat{y} = f(x)$. The function, f, will be good, in so far as the predicted output $\hat{y}$ matches the actual output y.

	x1	x2	x3	x4	x5	y
0	0.247746	36.0	266.0	0.247746	88.019654	57.000823
1	0.179340	37.0	365.0	0.179340	27.211935	22.471227
2	0.956807	21.0	151.0	0.956807	97.456012	56.357366
3	0.869653	43.0	437.0	0.869653	43.203221	34.652475
4	0.825345	47.0	160.0	0.825345	98.933930	64.426163
5	0.331114	321.0	371.0	0.331114	8.257917	14.218720
6	0.765523	17.0	364.0	0.765523	96.696783	59.123769
7	0.956807	21.0	151.0	0.956807	97.456012	52.983470

To construct the model, you first need to collect some data. To do this, you input lists of numbers into the mystery box and record the resulting output in a table. In this case, eight trials were done, numbered 0 through 7. Each input vector has five entries, x_1 through x_5, and the rightmost column records the output. With this table, you can now start looking for patterns.

Suppose you suspect that the fifth input, x_5, may have a stronger influence on the output than all other inputs, x_1 through x_4. To visualize this relationship, you create a scatter plot of x_5 versus the output y.



The scatter plot confirms that the output tends to be larger when x_5 is larger. This trend suggests an approximate linear regression model: $\hat{y} = 0.5x_5 + 10$. The output is a straight line function of x_5 and ignores all other inputs. The model is not perfect and does make some errors. It is just one possibility of the many models that could be used.

Evaluating the model

A loss function, denoted with the letter L, is used to determine the accuracy of a given prediction model. It produces a score for each prediction by comparing the predicted value $\hat{y}$ to the actual value y. The score is a penalty and so the objective is to get as small a score as possible. One common example is: $L(y, \hat{y}) = (y - \hat{y})^2$.

- When using a loss function:
- Precisely predicting y yields zero penalty
- The penalty increases quadratically with an error

Complex models

Complex systems are often non-deterministic and require non-mechanistic, data-centric techniques of machine learning.

In a non-deterministic system:

- The same input gets different outputs
- There is no hope of achieving zero penalty
- A lower bound on the score accounts for all natural internal uncertainty
- The tools of probability and statistics are needed to build a model

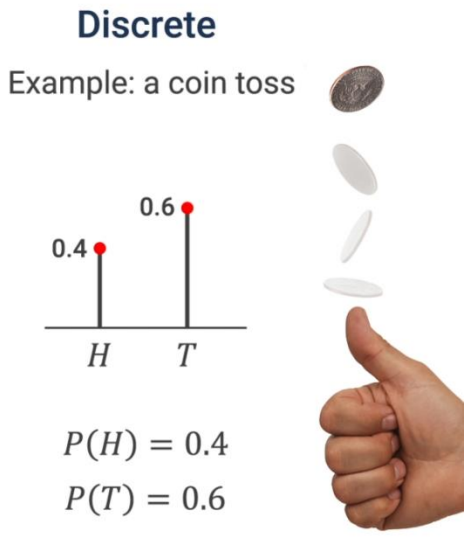
Distribution and Random Variables

Consider a system with no inputs, which outputs a different number each time it is sampled. Using the first 20 samples that are output, how should you model the fact that the system is non-deterministic? That is, that it produces different outputs for the same inputs or even for no inputs at all.

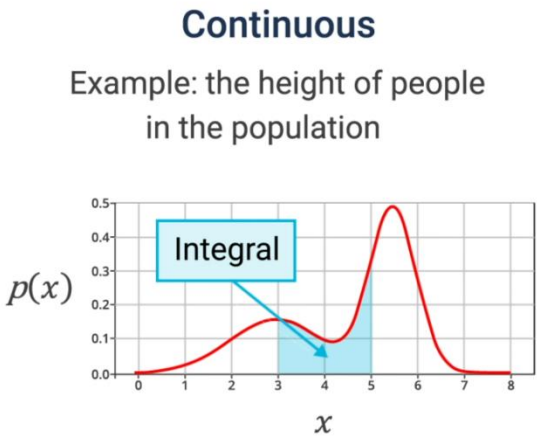
	y
0	0.050461
1	0.237693
2	0.198162
	⋮
17	0.416287
18	0.442066
19	0.903472

This is where probability theory comes in. Your model of the system will consist of a **probability density function (PDF)**, also known as a distribution.

Your goal will be to infer something about the distribution from the observed data. To do this, you use a PDF to compute the probability of every possible value of the output. There are two types of distributions, depending on whether the output is **discrete** or **continuous**.



The toss of a coin is an example of a system with a discrete valued output. There are only two possibilities: heads or tails. For discrete distributions, you read the probability of each value directly. For example, here the probability of heads (H) is 0.4 and the probability of tails (T) is 0.6.



The height of people in a population is an example of a continuous valued distribution. The output number can be any real number, such as 160cm, 161cm, or anything in between. For example, 160.3cm.

However, it does not make sense to ask for the probability that a person is 160cm tall, since no-one is exactly 160.0000cm tall. Instead, ask for the probability of a range of heights, such as from 160 to 165 cm. The probability of heights from a to b corresponds to the area under the curve between the two values. This is referred to as the **integral** of that function.

$$P([a,b] = \int_a^b p\,(x)dx$$

You denote the probability of something with a capital P, and the probability density function with a lowercase p. The probability of heights falling in the range a to b equals the integral of the probability density function from a to b.

Variables, such as Y, whose values change each time you sample them, are known as **random variables**. It is customary to denote random variables with capital letters. You use the capital letter Y for the generic output of the system, and lowercase y for samples of the output.

$Y \sim Pr$

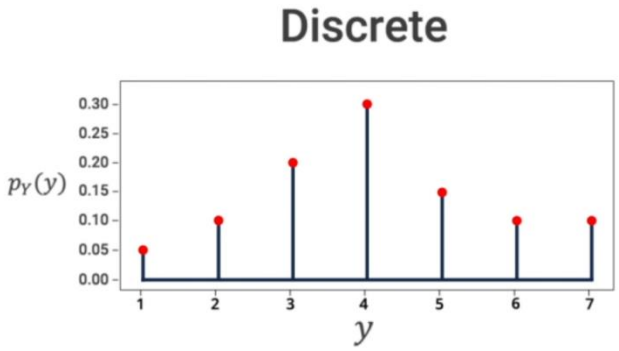
The tilde symbol means that a random variable is distributed according to, or sampled from, a particular distribution. So here you are saying that the random variable Y is distributed according to the distribution Pr.

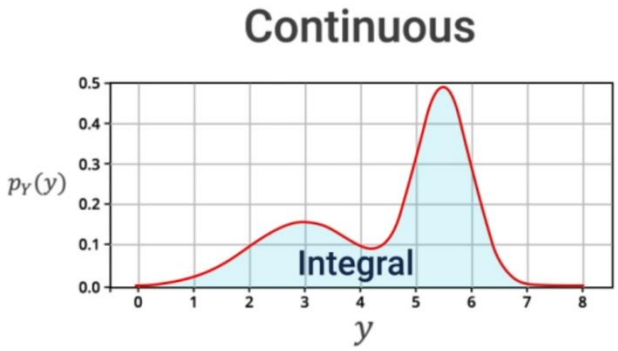
$[y]_{10} = [y_1, y_{2,} \cdots, y_9, y_{10}]$

Square brackets surrounding a lowercase letter denote a data set of size n. In this case, n equals ten data points sampled from the random variable Y.

Expected Value and Variance

The output of a system is represented as a random variable, which is a variable whose value is sampled from a PDF or distribution.





1. $p_Y(y) \geq 0$>

1. $p_Y(y) \geq 0$

2. $\sum_{y=-\infty}^{\infty} p_Y(y)=1$

2. $\int_{y=-\infty}^{\infty} p_Y(y)=1$

A function must satisfy two criteria to qualify as a distribution:

1. **It can never be negative:** All its values must be greater or equal to zero. Discrete distributions can never be above one. For continuous distributions, the area under the curve between any two vertical lines can never exceed one.
2. **The sum of all values must equal one:** In the first example, the sum of all seven values is indeed one. So this function is a discrete distribution. For continuous distributions, the requirement is that the total area under the curve, or the integral, from minus infinity to infinity, equals one.

If the function $p_Y(y)$ qualifies as a distribution, then you can think of the random variable Y as sampled or distributed according to p_Y.

The random variable Y has two important properties: expected value and variance.

Expected value

The expected value is denoted as $E[Y]$, or μ_Y. It is also referred to as the expectation or mean of Y or p_Y.

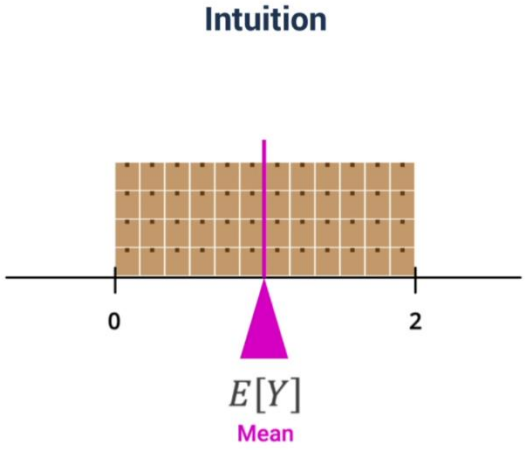
For discrete distributions, expectation is defined as the sum of the probabilities p_Y multiplied by the values Y.

$$E[Y] = \sum_{y=-\infty}^{\infty} y\, p_Y(y)$$

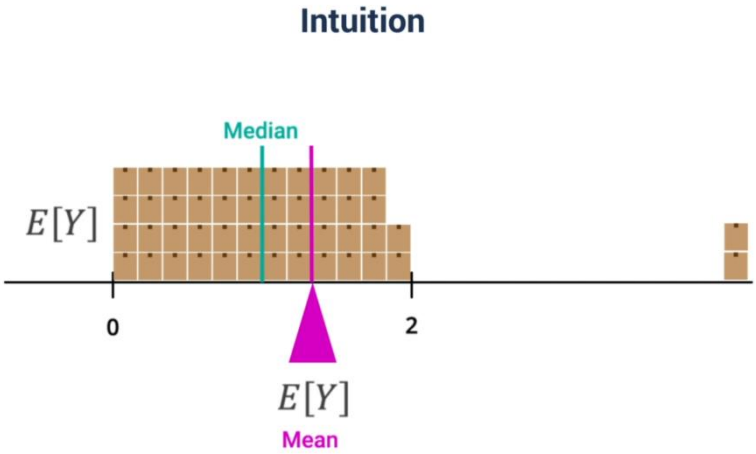
For continuous distributions, expectation is defined as the integral of the probabilities p_Y multiplied by the values Y.

$$E[Y] = \int_{-\infty}^{\infty} y\, p_Y(y) dy$$

The expectation of a distribution can be understood as its center of gravity. If you build the shape of the distribution by stacking boxes on top of each other, you can balance the distribution on a fulcrum placed at the mean.



By contrast, the **median** is the point that separates the boxes into two equal parts, with the same number of boxes on either side. If two boxes are moved much further to the right, then the balance point, or mean, shifts slightly to the right. However, the median remains at 1. The mean is therefore more sensitive to outliers than the median.



Computing the expected value

Consider an example of computing the expected value for a continuous distribution. The random variable can take any value between zero and two, all with equal probability.

This is the formula for this distribution, and it equals one half between zero and two, and zero everywhere else. You can verify that the area under the curve is one.

$$p_Y(y) = \begin{cases} 0.5 & y \in [0.2] \\ 0 & otherwise \end{cases}$$

To compute the expected value, you apply the expectation formula:

$$E[Y] = \int_0^2 y\, p_Y(y) dy$$

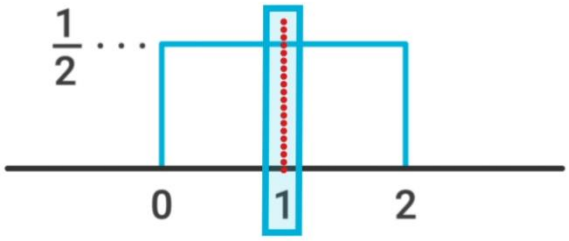
$$= \int_0^2 y\ 0.5\ dy$$

$$= 0.5\ \ y\text{^}2/2|_0^2$$

$$= 0.5\ \frac{4}{2}$$

$$= 1$$

So it has been determined that the expected value of this distribution is one, which is the midpoint between zero and two. This is reasonable since the distribution is symmetric and so its balance point should be at the center.



Variance

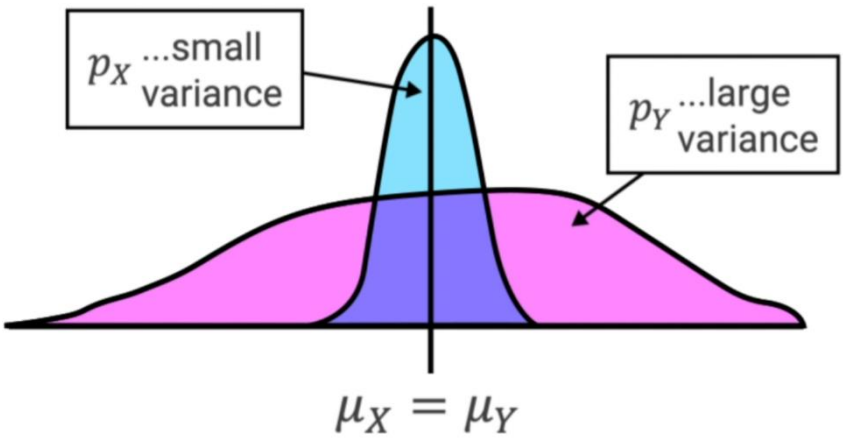
Variance is a measure of a distribution's width and is represented as

$Var[Y], or\ \sigma_Y^2$.

Variance is defined mathematically as the expected value of the square deviation of Y from its mean:

$$\sigma_Y^2 = Var[Y\} = e\{(y - \mu_Y)^2]$$

A distribution with large variance is typically wider and shorter than a distribution with small variance.



Variance gives a sense of the uncertainty in sampling a random variable. Small variance means you are more likely to sample values that are closer to the mean, whereas large variance means you are very uncertain about the values you will sample.

The units of variance are the square of the units of Y. For this reason, it is common to work with the **standard deviation** of a random variable rather than the variance. The standard deviation of Y is:

$$\sigma_Y = \sqrt{Var\{Y\}}$$

Why focus on the mean instead of the median?

The answer to this question is given by a very important property known as **the law of large numbers**.

Consider an arbitrary continuous or discrete distribution. You can compute its mean using the relevant expectation formula.

If you sample the distribution n times, you can put the results into a list,

$[y]_n$:

$$[y]_n = [y_1, y_2, \ldots, y_n]$$

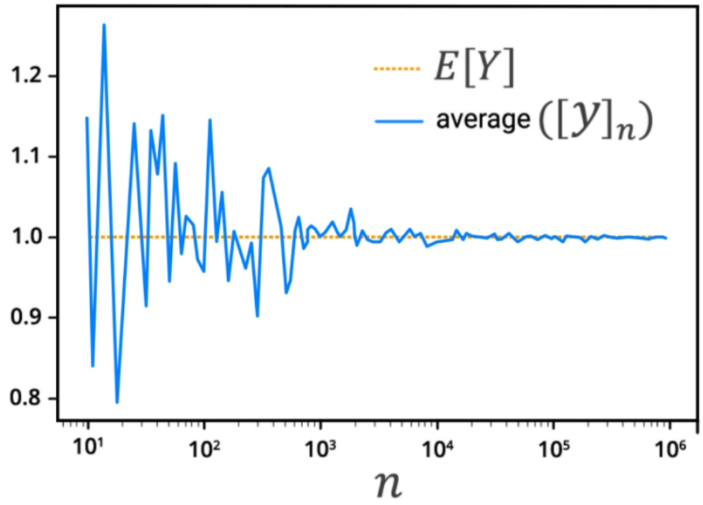
You can take the average of the list by adding up the values and dividing by n:

$$average\ ([y]_n) = \frac{1}{n}\sum_{i=1}^{n} y_i$$

The law of large numbers states that the average of n samples of a random variable converges to $E[Y]$ as n becomes larger. This can be expressed as:

$$average\ ([y]_n) \longrightarrow E[Y]\ as\ n \longrightarrow \infty$$

This plot shows the law of large numbers in action.



The blue line is the sample average for a sample size of n, plotted against n on a semi-log scale. So n varies from 10 to 1 million. The average gets closer and closer to the expected value of Y as n grows.

Introduction to pandas

The pandas package adds functionality to Python for operating on tabular data in several ways:

- **Tables** are commonly used for datasets in machine learning applications
- **Rows** represent samples from a multivariate distribution
- **Columns** represent features or single, random variables

	X1	X2	Y
0	0.5	A	0.2
1	0.9	B	0.1
2	0.1	B	-0.3

Why use pandas?

You may be familiar with various types of lists that are available in Python. There are Python native lists, which are heterogeneous, meaning they can hold different data types. For example, this is a native list that has numbers and strings: **[0.1, 0.2, 'a', 'b'].**

There are also NumPy arrays, which are homogeneous. For example, this NumPy array contains only numbers: **np.array([0.1, 0.2, 0.3, 0.4]).**

Homogeneity is efficient for numerical computation and is therefore preferred for machine learning applications.

In Python, pandas takes this one step further and offers a DataFrame type, which is used for tabular data. As an example, this DataFrame has three columns—X1, X2, and Y—and each column is homogeneous. X1 has numbers, X2 has strings, and Y has numbers.

	X1	X2	Y
0	0.5	A	0.2
1	0.9	B	0.1
2	0.1	B	-0.3

Importing pandas

Like many packages in Python, pandas has a standard alias: **pd.** To import pandas, you run the line: **import pandas as pd.**

Creating a DataFrame

The first thing to do when using pandas, is to create a DataFrame. This can be done using data stored in your computer's memory, or by importing data from an external source.

Creating a DataFrame from internal memory

If you want to create a DataFrame using data stored in computer memory, you need to load a dictionary into memory.

To load a dictionary into memory, you need to define:

- Headers for the DataFrame
- The values of the columns

You can do this by passing in a line, such as:

data = {'A': [25, 56, 93] , 'B': ['str1', 'str2', 'str3']}

In this example, if a dictionary has two strings as its keys—A and B—these serve as the headers for the DataFrame. The values in the columns—in this instance, **[25, 56, 93]** for column **A** and **['str1', 'str2', 'str3']** for column **B**—are the lists that the keys refer to.

Once you have loaded this dictionary into memory, you call the DataFrame constructor and pass in the dictionary, which you assign to some variable, such as X, using the function: **pd.DataFrame(data)**.

	A	B
0	25	str1
1	56	str2
2	93	str3

The output is the DataFrame, X, with columns A and B. Each column contains the list of values or strings assigned to it. Notice the third, untitled column on the left, which is the **index**. Every DataFrame must have an index.

There are two ways to define the index:

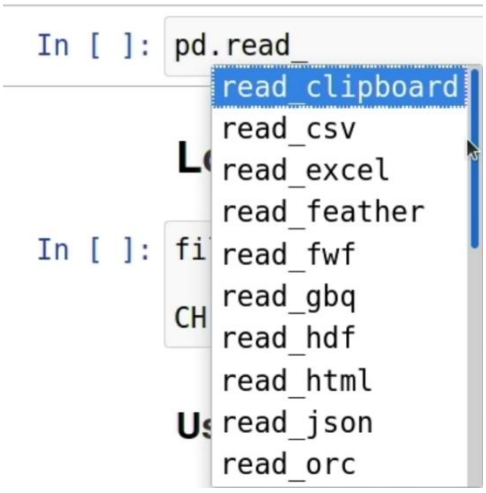
- Pass an index to the constructor using the **index** argument
- Use **set_index()** to assign a column to the index

If you do not explicitly tell pandas what to use as its index, it creates one by default using the integer IDs of the DataFrame rows, starting from zero.

Creating a DataFrame from an external source

The second way to create a DataFrame is using an external source.

To view a list of all the available read functions in pandas, you can type **pd.read_** and press the Tab key to activate the autocomplete function of Jupyter.



This lets you choose to read from .csv files, Excel, and many other data sources. In this program, you will mostly be reading from .csv files.

The functions to load a DataFrame include:

- **pd.read_csv(filename):** Loads a file from within a data folder and outputs the contents as a DataFrame
- **pd.read_csv(url):** Loads a file from an online source that you specify by URL so pandas can download it and put it into a DataFrame

DataFrame summaries

There are four methods in pandas for accessing summary information about a DataFrame:

- **info():** Gives a quick summary of the column data types
- **describe():** Provides summary statistics for numerical columns
- **head():** Shows the first five rows in a DataFrame
- **tail():** Lists the last five rows in a DataFrame

You can also specify the number of rows you would like to include when using the **head()** or **tail()** methods. For example, head(10), will output the first ten rows.

Data Selection in pandas

There are three basic ways to select data from a DataFrame:

- Bracket selection: **table[]**
- Label-based selection: **table.loc[]**
- Integer-based selection: **table.iloc[]**

Bracket selection

Bracket selection is used to select columns or rows, but not both at the same time. The syntax is the name of the DataFrame, followed by a selector in square brackets: **X[selector].**

To select a column, the selector must be a column label. So, to select the fullname column from the celebrity heights DataFrame, CH, you pass in:

CH['fullname']

To select multiple columns, you simply list each of the column labels. So, to select the fullname and lastname columns, you would pass in:

CH[['fullname' , 'lastname']]

When selecting rows, the selector must be a slice or a Boolean mask. For example, to use slice indexing to select the first four rows, you would run:

CH[:4]

And if you wanted to find celebrities whose height in meters is greater than two, you can do this using a Boolean mask:

CH['meters']>2

This returns a list of Booleans: False for celebrities shorter than two meters, and True for those taller than two meters.

You can also pass a Boolean mask into bracket selection. For instance, to return a list of all rows with a value greater than 2.2 in the meters column, you pass in:

CH[CH['meters']>2.2]

Label-based selection

Label-based selection, or loc, is a more general and powerful way of selecting from a DataFrame because it allows you to select both rows and columns. This gives you more interesting options for selecting data. The syntax is: **X.loc[rowselector, columnselector].**

The rowselector can be:

- An index, such as an integer, string, or a date

- A list of indexes
- A slice from an integer-based index
- A Boolean mask

The columnselector is optional and can be:

- A string label
- A list of string labels

The loc selection method can do almost everything bracket-based selection can. However, note that you cannot run **loc** of a slice when the index of the DataFrame is not integer based.

Using **loc**, you can:

- Select rows with a slice, for instance using **CH.loc[0:3]**
- Give it an index, such as **CH.loc[3]**
- Give it a list of indexes, for example **CH.loc[[0, 5, 7]]**
- Pass in values from the column assigned as the index, for example **CH2.loc['David Rappaport']**
- Pass in multiple strings to obtain a subselection of a table, as with **CH2.loc[['David Rappaport' , 'Warwick Davis']]**

In these examples the DataFrame, CH2, has the same celebrity heights data as CH but with the fullname column assigned as its index.

What makes **loc** especially useful, is that you can select rows and columns simultaneously.

For example, you can select the same two rows and then specify which columns to include using the column selector:

CH2.loc[['David Rappaport' , 'Warwick Davis'],['midname' , 'lastname']]

This produces a new table with only the middle name and the last name for those two rows.

As with bracket-based selection, you can do Boolean masking. For example, you can run loc on a previous Boolean mask example:

CH.loc[CH['meters']>2.2]

Alternatively, you can include only specific columns, such as midname and lastname, by passing in a column selector:

CH.loc[CH['meters']>2.2 , ['midname' , 'lastname']]

Integer-based selection

Integer-based selection, or **iloc,** is used for numerical DataFrames whose index and column headers are integers only. The syntax is: **iloc[rowselector, columnselector].** As with **loc**, the columnselector is optional.

The rowselector and columnselector can be:

- An integer or a list of integers
- A slice

For example, you can use iloc to find the first three rows and columns from the CH2 DataFrame:

CH2.iloc[:3, :3]

If you only want specific rows—such as 0, 10, and 20—you can do that using **iloc**:

CH2.iloc[[0, 10, 20] , :3]

This outputs only the specified DataFrame rows from the column you chose.

pandas Operations and Plots

Once you know how to create DataFrames and how to select data from those DataFrames, you can perform simple operations on columns of the DataFrame.

To begin, you import NumPy and pandas:

import numpy as np

import pandas as pd

Then you can create (and output) a very simple table:

table = pd.DataFrame({'A' : [0.1, 0.2, 0.3] , 'B' : [10, 20, 30]})

	A	B
0	0.1	10
1	0.2	20
2	0.3	30

This table has two columns, A and B, apart from the index and three rows. All the data is numerical.

You can determine what type of column each is by using the **type()** function. For instance, to take the first column and see what type it is you

pass in: **type(table['A']).** The **output pandas.core.series.Series** indicates that it is an object of type series.

A series represents a column om a pandas DataFrame. Apart from the DataFrame, it is the other user-facing object class in pandas. There are many interesting and useful methods attached to the series object, including: **sum(), add()**, and **multiply()**. These operations are similar to those you would use in a spreadsheet application, like Excel.

Column operations

You can find the summation of a column's values. To do this, you take a column, such as A, and you run the **sum()** function on it: **table['A'] .sum().**

The output is the total – for instance, 0.6, which is the sum of the row values 0.1, 0.2, and 0.3 in column A in this example.

If you wanted to add columns A and B, you would use the **add()** function. So to add the values in column A and B for each row, and then assign those values to column C, you run this **add()** function:

table['C'] = table['A'] .add(table['B'])

The result is a column C that has the sum of columns A and B for each row.

You can also multiply columns. For instance, to multiply columns A and B and store the product values in column D, you use the **multiply()** function:

table['D'] = table['A'] .multiply(table['B'])

Plotting with pandas

The simplest way to plot data from a DataFrame is to use **DataFrame.plot().** This **plot()** function uses a **kind** parameter to specify the type of plot to produce.

First, you need to load some data into a DataFrame.

from sklearn import datasets

X = datasets.load_iris(as_frame=True)

df = X.frame

In this instance, data from the Iris scikit-learn dataset is used. Scikit-learn offers several test datasets for playing with different methods, like regression. The Iris dataset is a classical dataset used for regression.

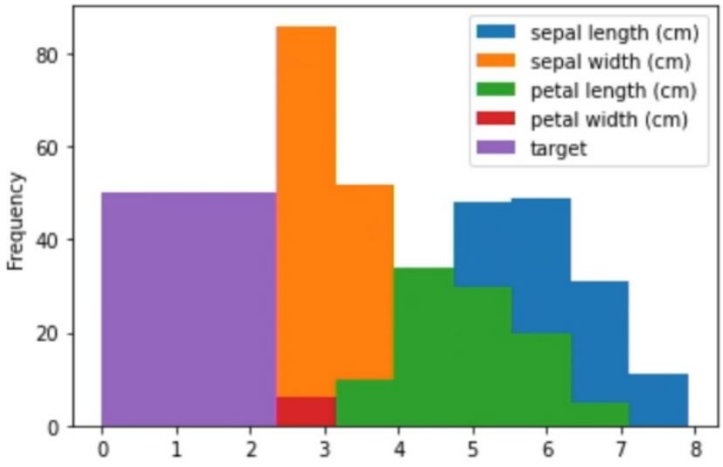
Once the dataset is in a DataFrame, you can pass in **df** to take a look at it:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

150 rows × 5 columns

In this example, the dataset contains measurements taken from 150 different Iris flowers. These include sepal length, sepal width, petal length, petal width. And on the right is the target variable, which is the Iris flower variety. This dataset is used to create regression models that will predict what type of flower corresponds to some petal measurements.

You can create plots of your data, such as a histogram, by calling the plot() method and setting the kind argument to histogram: **df.plot(kind='hist')**.

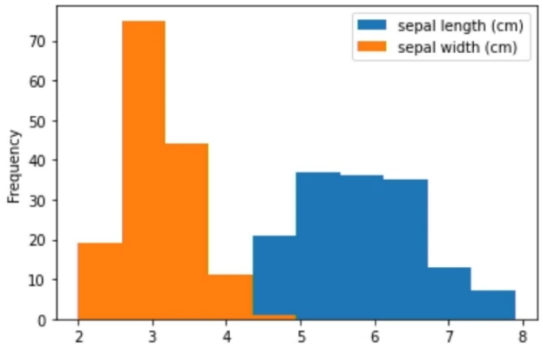


This creates a histogram of all the columns, including the target.

If you only want some of the columns, say sepal length and sepal width, you can pass these in as the y argument:

df.plot(kind='hist' , y=['sepal length cm' , 'sepal width (cm)'])

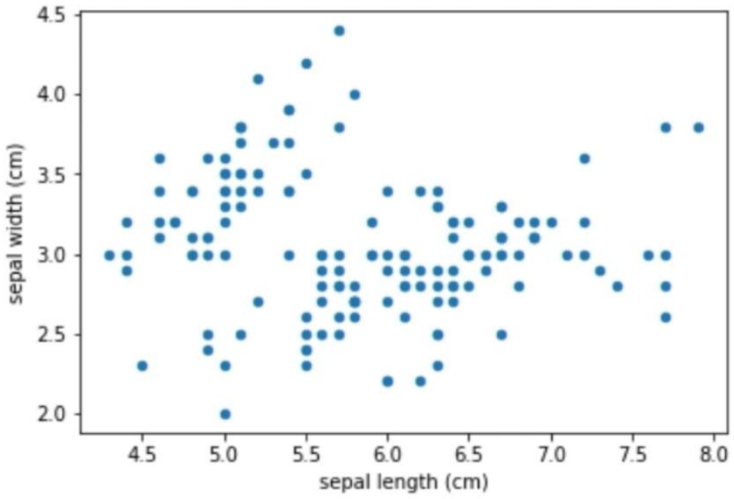
This outputs histograms only of the sepal length and sepal width.



Another type of plot you might want to create is a scatterplot. To do this, you set the **kind** argument to scatterplot and specify which variables to use for the x and y axes:

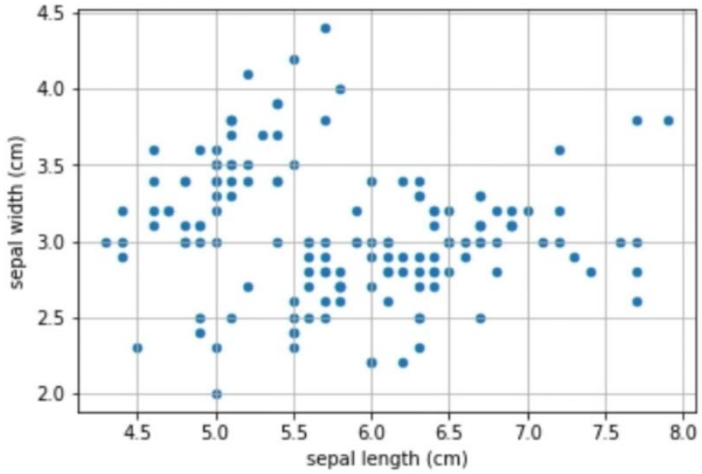
df.plot(kind='scatter' , x='sepal length (cm)' , y='sepal width (cm)')

For instance, you can choose the sepal length and sepal width columns.



There are several other arguments you can pass in to this plot function. For example, you can add a grid by passing in **"grid=True":**

df.plot(kind='scatter' , x='sepal length (cm)' , y='sepal width (cm)' , grid=True)



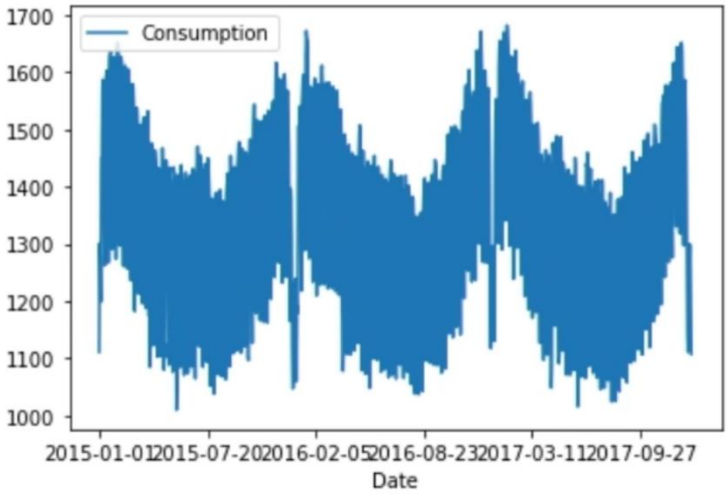
Another type of plot that you might want to make is a line plot. Line plots are common for time series data. To do this, you can load a dataset–such as one called timeseries, which is a .csv file–by passing in this line:

df = pd.read_csv('data/timeseries.csv')

In this instance, the timeseries dataset contains 1,096 measurements, taken over three years of power consumption. These include wind and solar energy production values and a column summing the two.

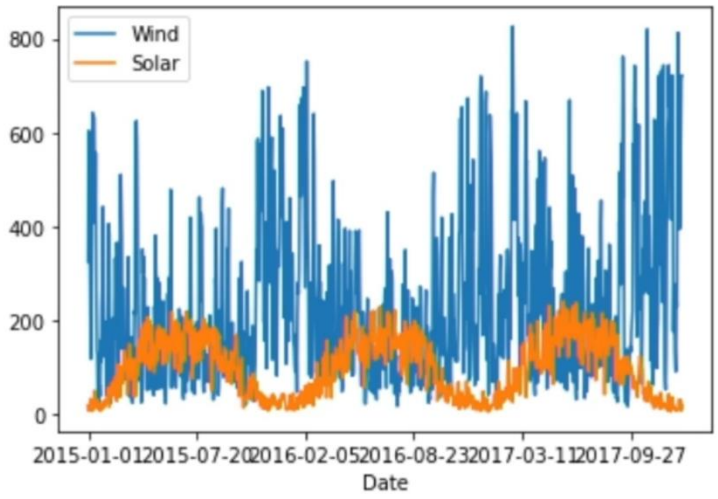
To make a line plot with Date as the x-axis label and Consumption as the values plotted along the y-axis, you pass in:

df.plot(kind='line' , x='Date' , y='Consumption')



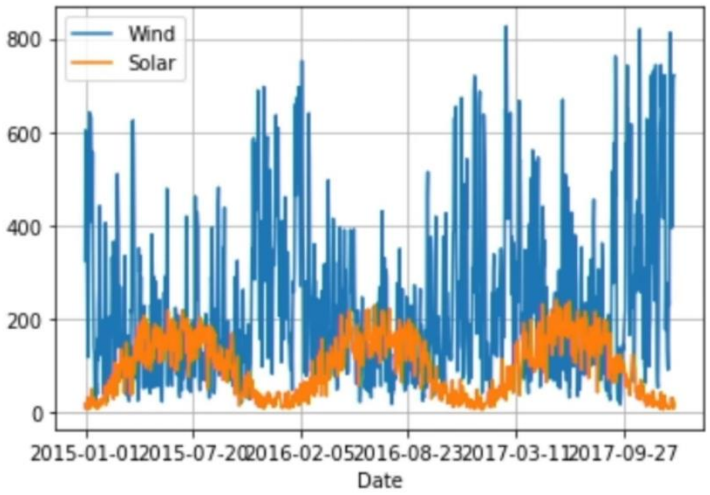
You might also want to have more than one plotted line. To do this, you can turn this into a vector by including Wind and Solar in the y argument:

df.plot(kind='line' , x='Date' , y=['Wind' , 'Solar'])



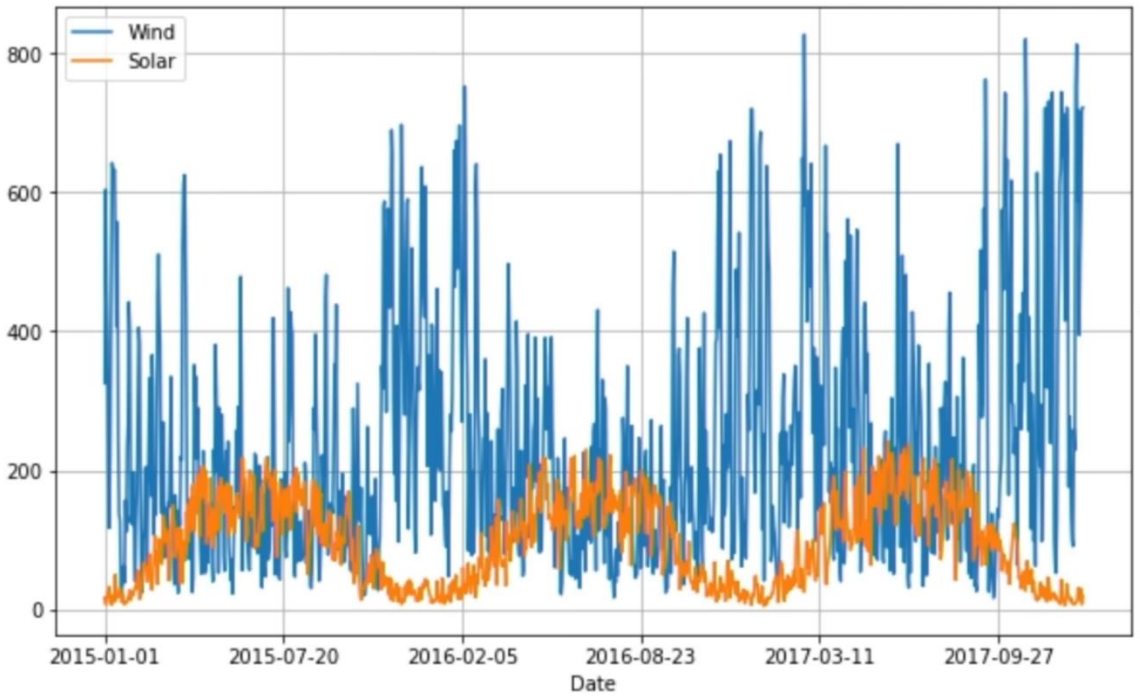
Again, you can add a grid to the plot by setting "grid=**True":**

df.plot(kind='line' , x='Date' , y=['Wind' , 'Solar'] , grid=True)



Another thing you can do is change the size of the plot. In this instance, you make it a little bit bigger by passing the **figsize** argument:

df.plot(kind='line' , x='Date' , y=['Wind' , 'Solar'] , grid=True, figsize=(10, 6))



Suppose you want to create a summary of the total consumption, wind power, and solar power over a specific period, and report it as a bar plot. To do this, you can use bracket selection to extract the Consumption, Wind, and Solar columns from the DataFrame: **df[['Consumption' , 'Wind' , 'Solar']].**

	Consumption	Wind	Solar
0	1111.33600	325.128	17.079
1	1300.88400	603.558	7.758
2	1265.27100	462.953	7.236
3	1198.85400	385.024	19.984
4	1449.86100	216.543	26.524
...	...	...	...
1091	1263.94091	394.507	16.530
1092	1299.86398	506.424	14.162
1093	1295.08753	584.277	29.854
1094	1215.44897	721.247	7.467
1095	1107.11488	721.176	19.980

1096 rows × 3 columns

You calculate the sum of those columns:

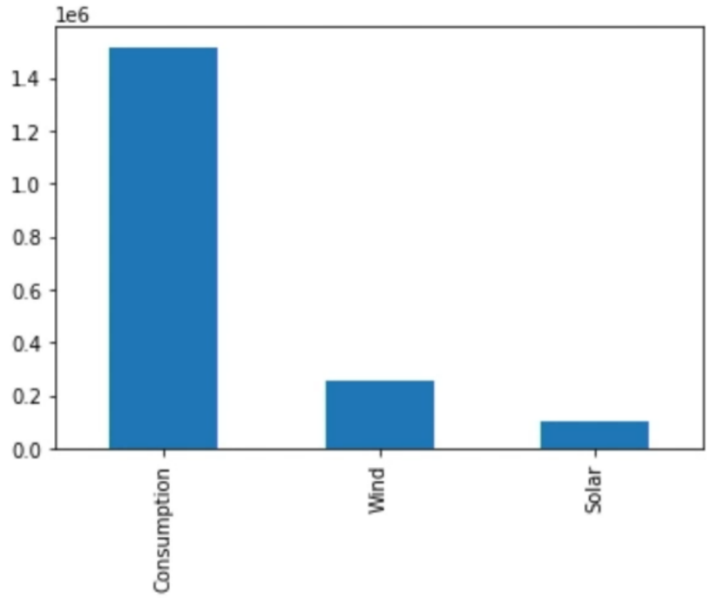
df[['Consumption' , 'Wind' , 'Solar']] .sum()

```
Out[28]: Consumption    1.515928e+06
         Wind           2.571445e+05
         Solar          1.053526e+05
         dtype: float64
```

That gives you the total consumption and production of wind and solar energy.

Next, you can make a bar plot by setting the plot **kind** to bar:

df[['Consumption' , 'Wind' , 'Solar']] .sum().plot(kind='bar')



And there you have it: a plot of the total consumption, as well as the total production of wind-, and solar energy.