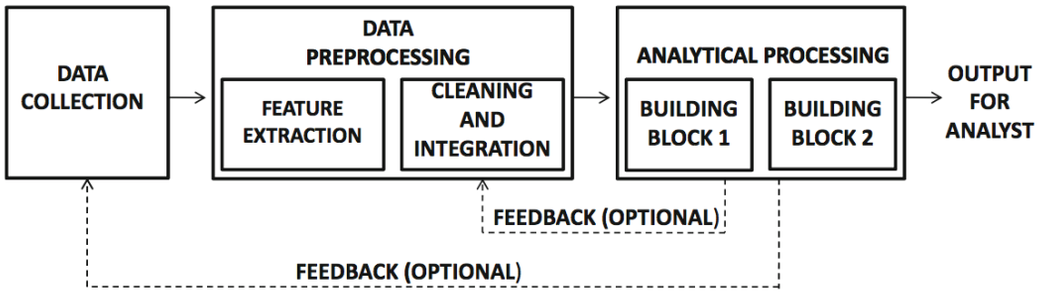
Chapter 5: Fundamentals of Machine Learning

In this Chapter, we will give an overview of data, artificial intelligence, machine learning and deep learning. We will then cover the fundamentals of machine learning, in particularly, those aspects related to supervised learning.

5.1 Data

Data is the raw material that powers our intelligent machines, without which nothing would be possible. Data often comes in large data sets. Data mining is the process of discovering **patterns** in large data sets involving methods at the intersection of machine learning and database systems. The workflow of a typical data mining application contains the following phases:



1. Data collection: Data collection may require the use of specialized hardware such as a sensor network, manual labor such as the collection of user surveys, or software tools such as a Web document crawling engine to collect documents.

   While this stage is highly application-specific and often outside the realm of the data mining analyst, it is critically important because good choices at this stage may significantly impact the data mining process. After the collection phase, the data are often stored in a database, or, more generally, a data warehouse for processing.

2. Data Preprocessing (feature extraction and data cleaning): When the data are collected, they are often not in a form that is suitable for processing. For example, the data may be encoded in complex logs or free-form documents. In many cases, different types of data may be arbitrarily mixed together in a free-form document. To make the data suitable for processing, it is essential to transform them into a format that is friendly to data analytics algorithms.

   It is crucial to extract relevant features (also known as variables, dimensions or attributes; more on

this in the sequel) for the analytics process. The feature extraction phase is often performed in parallel with data cleaning, where missing and erroneous parts of the data are either estimated or corrected. In many cases, the data may be extracted from multiple sources and need to be integrated into a unified format for processing. The final result of this procedure is a nicely structured data set, which can be effectively used by a computer program. After the feature extraction phase, the data may again be stored in a database for processing.

3. Analytical Processing (algorithms): The final part of the mining process is to design effective analytical methods from the processed data. Suitable algorithms are shortlisted. Following which, parameters of the algorithms have to be tuned to optimize results. Note that the analytical block in the workflow shows multiple building blocks representing the design of the solution to a particular application.

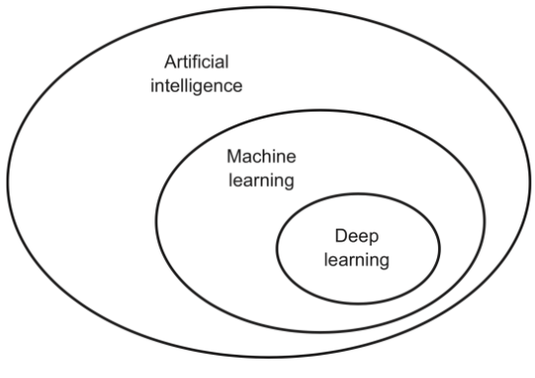
5.2 Artificial Intelligence, Machine Learning and Deep Learning

In the past few years, artificial intelligence (AI) has been a subject of intense media hype. Machine learning, deep learning, and AI come up in countless articles, often outside of technology-minded publications.

We're promised a future of intelligent chatbots, self-driving cars, and virtual assistants—a future sometimes painted in a grim light and other times as utopian, where human jobs will be scarce and most economic activity will be handled by robots or AI agents.

For a future or current practitioner of data science and machine learning, it's important to be able to recognize the signal in the noise so that you can tell world-changing developments from overhyped press releases. This section provides essential context around artificial intelligence, machine learning, and deep learning.

A Venn diagram showing how deep learning is a kind of machine learning, which is used for many but not all approaches to AI.



5.2.1 Artificial Intelligence

Artificial intelligence was born in the 1950s, when a handful of pioneers from the nascent field of computer science started asking whether computers could be made to "think"—a question whose ramifications we're still exploring today[1]. A concise definition of the field would be as follows: **the effort to automate intellectual tasks normally performed by humans**. As such, AI is a general field that encompasses machine learning and deep learning, but that also includes many more approaches that don't involve any learning.

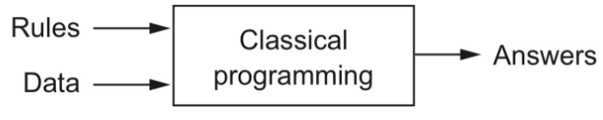
Early chess programs, for instance, only involved hardcoded rules crafted by programmers, and didn't qualify as machine learning. For a fairly long time, many experts believed that human-level artificial intelligence could be achieved by having programmers handcraft a sufficiently large set of explicit rules for manipulating knowledge. This approach is known as symbolic AI, and it was the dominant paradigm in AI from the 1950s to the late 1980s. It reached its peak popularity during the expert systems boom of the 1980s.

Although symbolic AI proved suitable to solve well-defined, logical problems, such as playing chess, it turned out to be intractable to figure out explicit rules for solving more complex, fuzzy problems, such as image classification, speech recognition, and language translation. A new approach arose to take symbolic AI's place: machine learning.

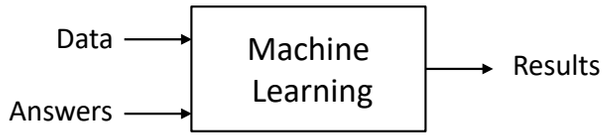
5.2.2 Machine Learning

Machine learning arises from this question: rather than programmers crafting data-processing rules by hand, could a computer automatically learn these rules by looking at data?

Machine learning opens the door to a new programming paradigm. In classical programming, the paradigm of symbolic AI, humans input rules (a program) and data to be processed according to these rules, and outcome answers (see figure below). With machine learning, humans input data as well as the answers expected from the data, and outcome the rules. These rules can then be applied to new data to produce original answers.



[1] See John McCarthy (2007), "What is Artificial Intelligence?", http://www-formal.stanford.edu/jmc/whatisai.

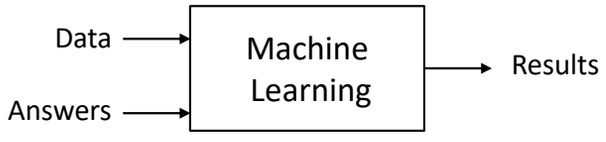


A machine-learning system is **trained** rather than explicitly programmed. It's presented with many examples relevant to a task, and it finds statistical structure in these examples that eventually allows the system to come up with rules for automating the task. For instance, if you wished to automate the task of tagging your vacation pictures, you could present a machine-learning system with many examples of pictures already tagged by humans, and the system would learn statistical rules for associating specific pictures to specific tags.

Although machine learning only started to flourish in the 1990s, it has quickly become the most popular and most successful subfield of AI, a trend driven by the availability of faster hardware and larger datasets.

Remark: Machine learning is tightly related to mathematical statistics, but it differs from statistics in several important ways. Unlike statistics, machine learning tends to deal with large, complex datasets (such as a dataset of millions of images, each consisting of tens of thousands of pixels) for which classical statistical analysis such as Bayesian analysis would be impractical. As a result, machine learning, and especially deep learning, exhibits comparatively little mathematical theory—maybe too little—and is engineering oriented. It's a hands-on discipline in which ideas are proven empirically more often than theoretically.

5.2.3 Learning Representation from Data



As we just stated that machine learning discovers rules to execute a data-processing task, given examples of what's expected. So, to do machine learning, we need three things:

- Input data points—For instance, if the task is speech recognition, these data points could be sound files of people speaking. If the task is image tagging, they could be pictures.
- Examples of the expected output—In a speech-recognition task, these could be human-generated transcripts of sound files. In an image task, expected outputs could be tags such as "dog," "cat," and so on.
- A way to measure whether the algorithm is doing a good job—This is necessary in order to

determine the distance between the algorithm's current output and its expected output. The measurement is used as a feedback signal to adjust the way the algorithm works. This adjustment step is what we call learning.

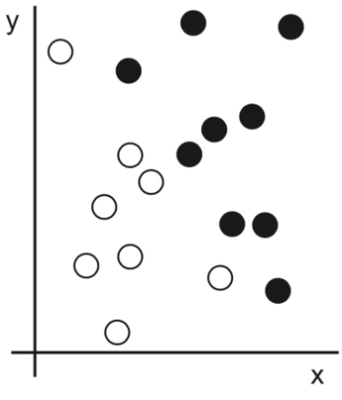
A machine-learning model transforms its input data into meaningful outputs, a process that is "learned" from exposure to known examples of inputs and outputs. Therefore, the central problem in machine learning and deep learning is to **meaningfully transform data**: in other words, to **learn useful representations of the input data** at hand—representations that get us closer to the expected output.

Before we go any further: what's a representation? At its core, it's a different way to look at data—**to represent or encode data**.

Example: A color image can be encoded in the RGB format (red-green-blue) or in the HSV format (hue-saturation-value): these are two different representations of the same data. Some tasks that may be difficult with one representation can become easy with another. For example, the task "select all red pixels in the image" is simpler in the RGB format, whereas "make the image less saturated" is simpler in the HSV format.

Machine-learning models are all about finding appropriate representations for their input data—transformations of the data that make it more amenable to the task at hand, such as a classification task.

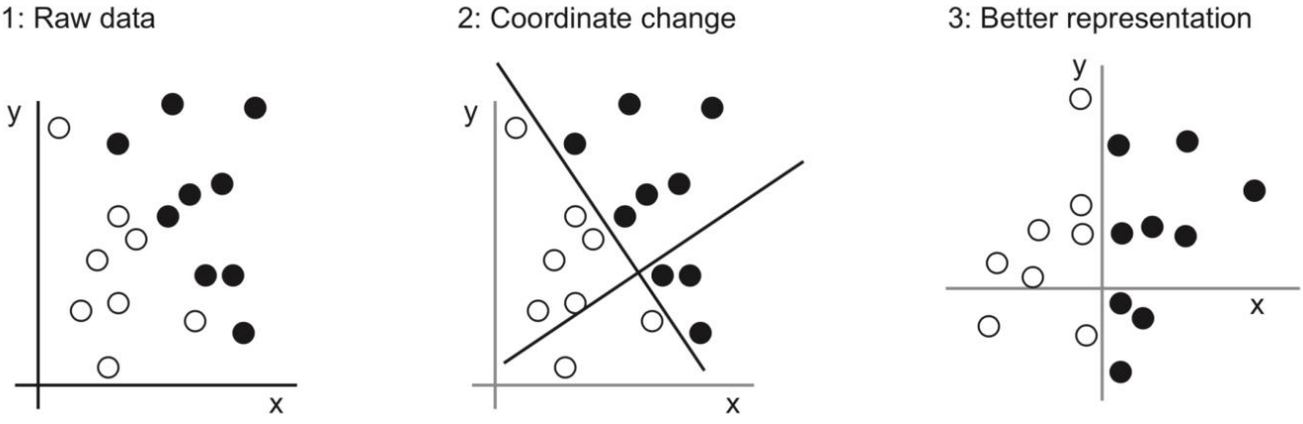
Example: Consider an x-axis, a y-axis, and some points represented by their coordinates in the (x, y) system, as shown in the figure below. As you can see, we have a few white points and a few black points. Let's say we want to develop a machine learning algorithm that can take the coordinates (x, y) of a point and output whether that point is likely to be black or to be white.



Q: What are the input, expected output and the way to measure whether the algorithm is doing a good job?

A:

What we need here is a new representation of our data that cleanly separates the white points from the black points. One transformation we could use, among many other possibilities, would be a coordinate change, as illustrated in the figure below.



In this new coordinate system, the coordinates of our points can be said to be **a new representation of our data**. And it's a good one! With this representation, the black/white classification problem can be expressed as a simple **rule**: "Black points are such that x > 0," or "White points are such that x < 0." This new representation basically solves the classification problem.

In this case, we defined the coordinate change by hand. But if instead we tried systematically searching for different possible coordinate changes, and used as feedback the percentage of points being correctly classified, then we would be doing machine learning. Learning, in the context of machine learning, describes an automatic search process for better representations.

All machine-learning algorithms consist of automatically finding such transformations that turn data into more-useful representations for a given task. These operations can be coordinate changes, as you just saw, or linear projections (which may destroy information), translations, nonlinear operations, and so on. Machine-learning algorithms aren't usually creative in finding these transformations; **they're merely searching through a predefined set of operations, called a hypothesis space**.

So that's what machine learning is, technically: **searching for useful representations of some input data, within a predefined space of possibilities, using guidance from a feedback signal.** This simple idea allows for solving a remarkably broad range of intellectual tasks, from speech recognition to autonomous car driving.

Remark: fundamentally, machine learning involves building **mathematical models** to help understand data. "Learning" enters the picture when we give these models tunable parameters that can be adapted to observed data; in this way the program can be considered to be "learning" from the data. Once these models have been **fit** to previously seen data, they can be used to **predict** and understand aspects of newly observed data.

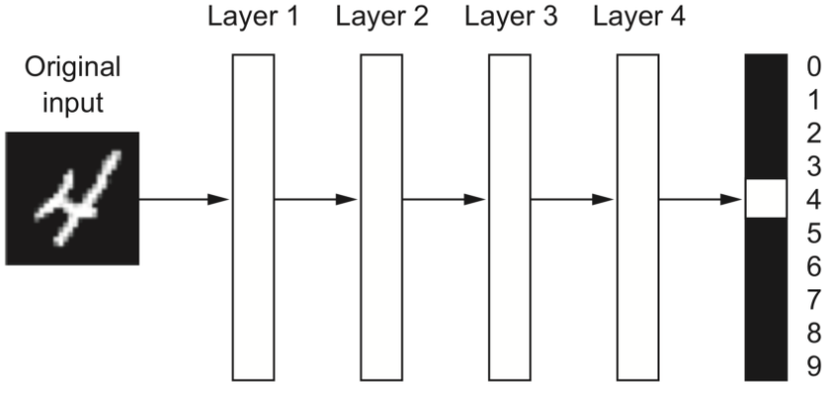
Now that you understand what we mean by learning, let's take a look at what makes deep learning special.

5.2.4 Deep Learning

Deep learning is a specific subfield of machine learning: **a new take on learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations.**

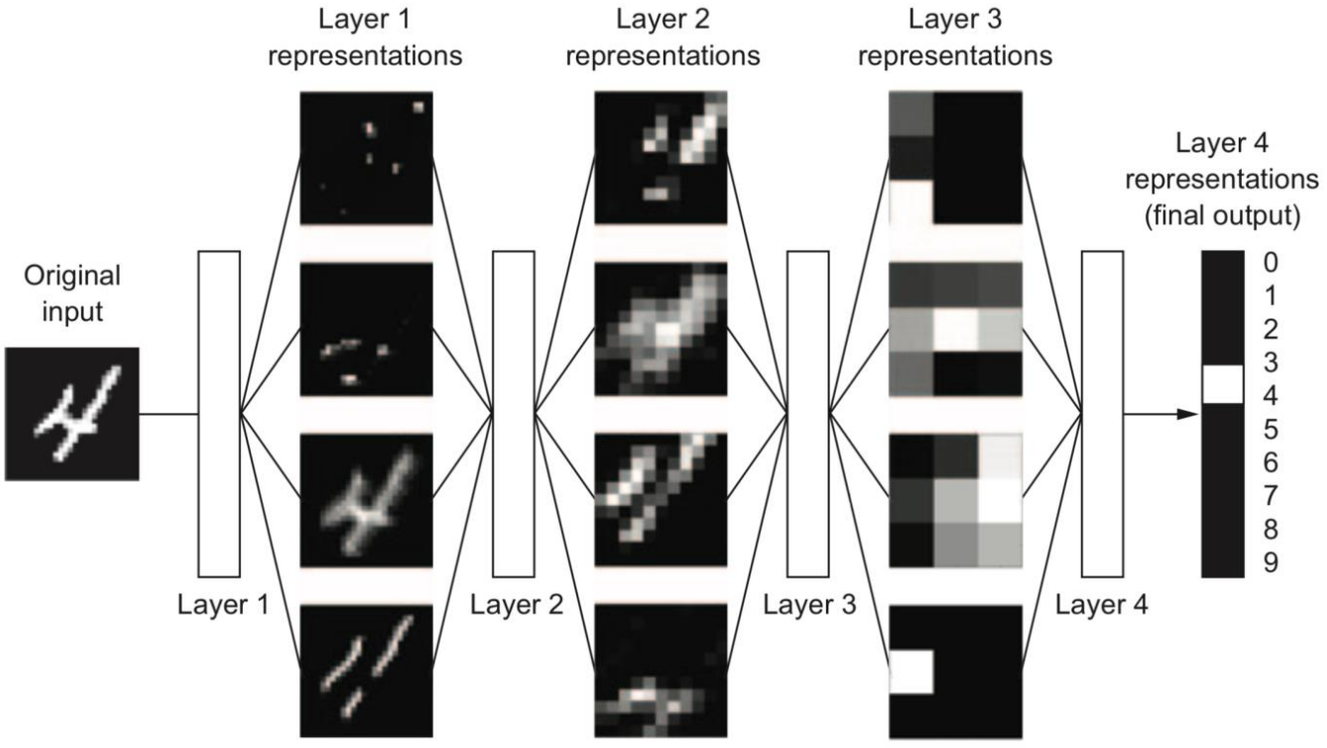
The deep in deep learning isn't a reference to any kind of deeper understanding achieved by the approach; rather, it stands for this idea of successive layers of representations. How many layers contribute to a model of the data is called the depth of the model. Modern deep learning often involves tens or even hundreds of successive layers of representations— and they're all learned automatically from exposure to training data.

Example: What do the representations learned by a deep-learning algorithm look like? Let's examine how a network several layers deep (see figure below) transforms an image of a digit in order to recognize what digit it is.



As you can see in figure below, the network transforms the digit image into representations that are increasingly different from the original image and increasingly informative about the final result. You

can think of a deep network as a multistage information-distillation operation, where information goes through successive filters and comes out increasingly purified (that is, useful with regard to some task).



So that's what deep learning is, technically: **a multistage way to learn data representations**. It's a simple idea—but, as it turns out, very simple mechanisms, sufficiently scaled, can end up looking like magic.

Remarks:

- Classical machine-learning techniques— sometimes called shallow learning—only involved transforming the input data into one or two successive representation spaces, usually via simple transformations such as high-dimensional non-linear projections (SVMs) or decision trees. But the refined representations required by complex problems generally can't be attained by such techniques. As such, humans had to go to great lengths to make the initial input data more amenable to processing by these methods: they had to manually engineer good layers of representations for their data. This is called feature engineering. Deep learning, on the other hand, completely automates this step: with deep learning, you learn all features in one pass rather than having to engineer them yourself. This has greatly simplified machine-learning workflows, often replacing sophisticated multistage pipelines with a single, simple, end-to-end deep-learning model.
- You may ask, if the crux of the issue is to have multiple successive layers of representations, could shallow methods be applied repeatedly to emulate the effects of deep learning? In practice,

there are fast-diminishing returns to successive applications of shallow-learning methods, because the optimal first representation layer in a three-layer model isn't the optimal first layer in a one-layer or two-layer model. What is transformative about deep learning is that it allows a model to learn all layers of representation **jointly**, at the same time, rather than in succession (greedily, as it's called). With joint feature learning, whenever the model adjusts one of its internal features, all other features that depend on it automatically adapt to the change, without requiring human intervention. Everything is supervised by a single feedback signal: every change in the model serves the end goal.

- These are the two essential characteristics of how deep learning learns from data: (1) the incremental, layer-by-layer way in which increasingly complex representations are developed, and (2) the fact that these intermediate incremental representations are learned jointly, each layer being updated to follow both the representational needs of the layer above and the needs of the layer below. Together, these two properties have made deep learning vastly more successful than previous approaches to machine learning.

Deep learning has achieved the following breakthroughs, all in historically difficult areas of machine learning:

- Near-human-level image classification
- Near-human-level speech recognition
- Near-human-level handwriting transcription
- Improved machine translation
- Improved text-to-speech conversion
- Digital assistants such as Google Now and Amazon Alexa
- Near-human-level autonomous driving
- Improved ad targeting, as used by Google, Baidu, and Bing
- Improved search results on the web
- Ability to answer natural-language questions
- Superhuman Go playing

We're still exploring the full extent of what deep learning can do. We've started applying it to a wide variety of problems outside of machine perception and natural-language understanding, such as formal reasoning. If successful, this may herald an age where deep learning assists humans in engineering, science, software development, and more.

5.3 The Landscape

A great way to get a sense of the current landscape of machine-learning algorithms and tools is to look at competitions on Kaggle. Due to its highly competitive environment (some contests have thousands of entrants and million- dollar prizes) and to the wide variety of machine-learning problems covered, Kaggle offers a realistic way to assess what works and what doesn't. So, what kind of algorithm is reliably winning competitions? What tools do top entrants use?

In 2016 and 2017, Kaggle was dominated by two approaches: **gradient boosting machines** and **deep learning**. Specifically, gradient boosting is used for problems where structured data is available, whereas deep learning is used for perceptual problems such as image classification. Practitioners of the former almost always use the excellent XGBoost library, which offers support for the two most popular languages of data science: Python and R. Meanwhile, most of the Kaggle entrants using deep learning use the Keras library, due to its ease of use, flexibility, and support of Python.

These are the two techniques you should be the most familiar with in order to be successful in applied machine learning today: gradient boosting machines, for shallow learning problems; and deep learning, for perceptual problems. In technical terms, this means you'll need to be familiar with XGBoost and Keras—the two libraries that currently dominate Kaggle competitions. We will cover these aspects in more details in the sequel Chapters.

5.4 Categories and Examples of Machine Learning

At the most fundamental level, machine learning can be categorized into two main types: supervised learning and unsupervised learning.

Supervised learning involves modeling the relationship between measured features of data and some label associated with the data. Once this model is determined, it can be used to apply labels to new, unknown data. This is further subdivided into **classification** tasks and **regression** tasks: in classification, the labels are discrete categories, while in regression, the labels are continuous quantities.

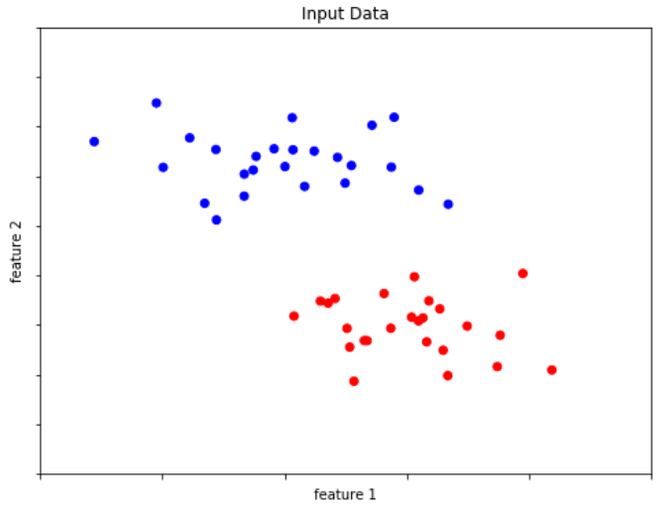
Unsupervised learning involves modeling the features of a dataset without reference to any label, and is often described as "letting the dataset speak for itself." These models include tasks such as **clustering** and **dimensionality reduction**. Clustering algorithms identify distinct groups of data, while dimensionality reduction algorithms search for more succinct representations of the data.

Supervised learning is the most successful machine learning type so far and will be the focus of our study. Let's take a look at two very simple examples, one for classification and the other for regression.

5.4.1 Classification: Predicting discrete labels

We will first take a look at a simple classification task, in which **you are given a set of labeled points and want to use these to classify some unlabeled points**.

Imagine that we have the data shown in this figure:



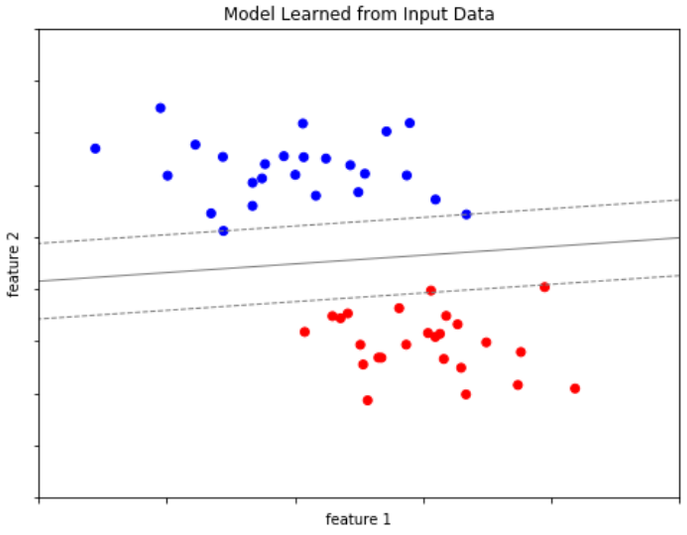
Here we have two-dimensional data: that is, we have **two features** for each point, represented by the (x, y) positions of the points on the plane. In addition, we have **one of two class labels for each point**, here represented by the colors of the points. From these features and labels, we would like to create a model that will let us decide whether a new point should be labeled "blue" or "red."

Q: Recall the aforementioned statement "machine learning involves building **mathematical models** to help understand data …". What is a proper mathematical model for such a classification task? What are the tunable parameters for the model?

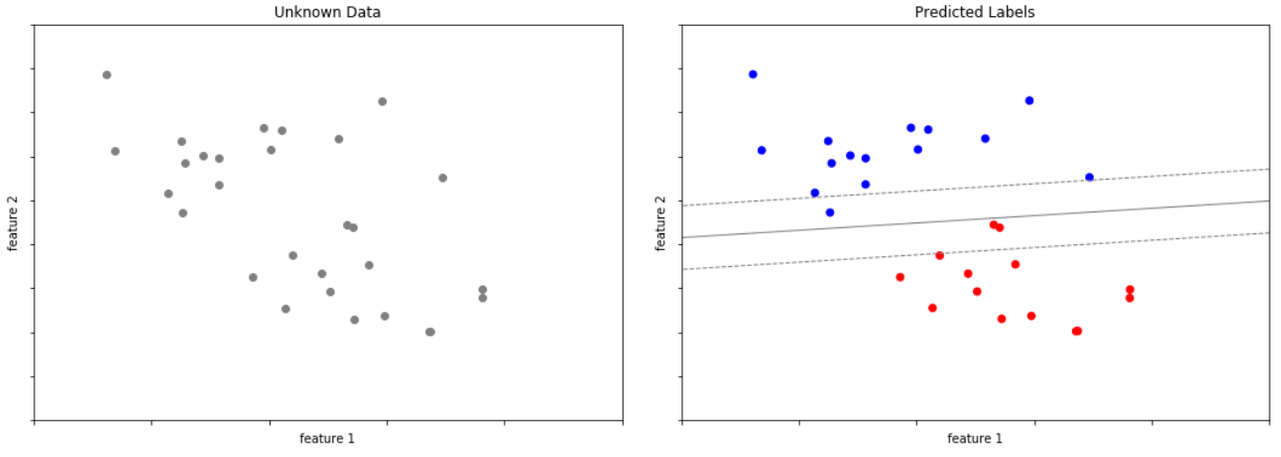
A:

There are a number of possible models for such a classification task, but here we will use an extremely simple one. We will assume that the two groups can be separated by drawing a straight line through the plane between them, such that points on each side of the line fall in the same group. Here the model is a quantitative version of the statement "a straight line separates the classes", while the model parameters are the particular numbers describing the location and orientation of that line for our data. The optimal values for these model parameters are learned from the data (this is the "learning" in machine learning), which is often called training the model.

The following figure shows a visual representation of what the trained model looks like for this data:



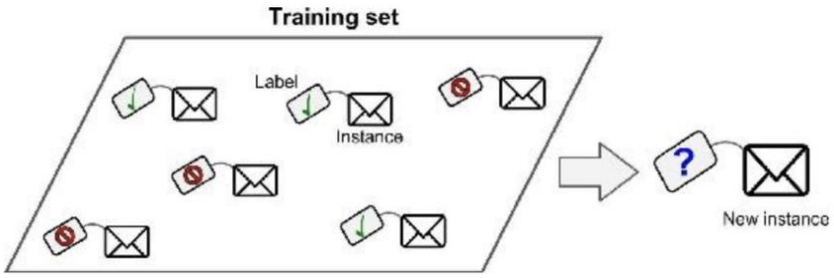
Now that this mathematical model has been **trained**, it can be **generalized** to new, unlabeled data. In other words, we can take a new set of data, draw this model line through it, and assign labels to the new points based on this model. This stage is usually called **prediction**. See the following figure:



This is the basic idea of a classification task in machine learning, where "classification" indicates that the data has discrete class labels. At first glance this may look fairly trivial: it would be relatively easy to simply look at this data and draw such a discriminatory line to accomplish this classification. A benefit of the machine learning approach, however, is that **it can generalize to much larger datasets in many more dimensions**.

(From A to A+) For those cannot wait to learn more Python for machine learning, you can find the complete Python codes for the above classification task from the course website.

Conceptual Example: Let us consider the task of **automated spam detection for email**. What are the possible **features** and **labels** we might use?



Ans

In this case, we might use the following features and labels:

- feature 1, feature 2, etc. → normalized counts of important words or phrases ("Viagra", "Nigerian prince", etc.)
- label → "spam" or "not spam"

For the training set, these labels might be determined by individual inspection of a small representative sample of emails; for the remaining emails, the label would be determined using the model. For a suitably trained classification algorithm with enough well-constructed features (typically thousands or millions of words or phrases), this type of approach can be very effective.

5.4.2 Regression: Predicting continuous labels

In contrast with the discrete labels of a classification algorithm, we will next look at a simple regression task in which the labels are **continuous quantities**.

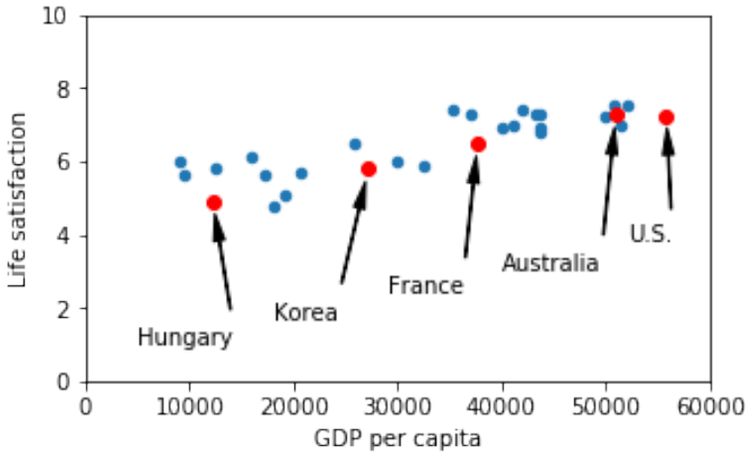
Q: Can you give some examples for regression?
A:

All these questions have a common structure: they ask for a response y that can be expressed as **x, a combination of one or more (independent) variables**. The role of regression is to build a **model** to predict the response from the variables.

Example: Suppose you want to know if “money makes people happy”, so you download the Better Life Index data from the OECD’s website (https://stats.oecd.org/index.aspx?DataSetCode=BLI) as well as stats about GDP per capita from the IMF’s website (https://www.imf.org/en/Data). Then you join the tables and sort by GDP per capita. Table below shows an excerpt of what you get.

Country	GDP per capita (USD)	Life satisfaction
Hungary	12,240	4.9
Korea	27,195	5.8
France	37,675	6.5
Australia	50,962	7.3
United States	55,805	7.2

Let's plot the data together with other countries



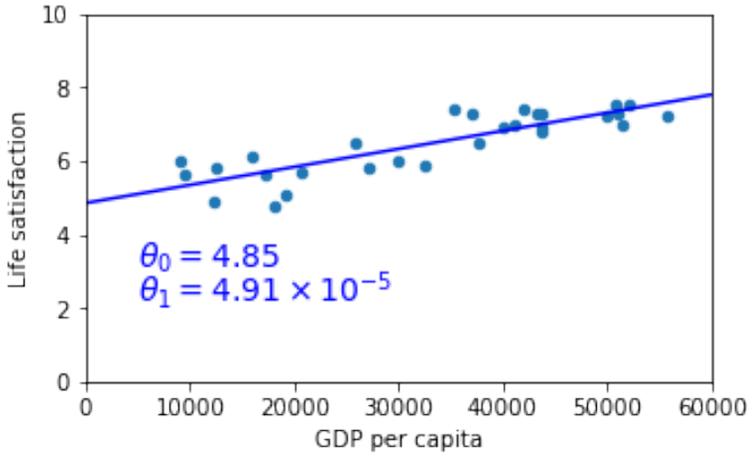
There does seem to be a trend here! Although the data is partly random, it looks like life satisfaction goes up more or less linearly as the country's GDP per capita increases. So you decide to model life satisfaction (y) as a **linear function** of GDP per capita (x). This step is called **model selection**: you selected a linear model of life satisfaction (y) with just one feature, GDP per capita (x):

$$y = \theta_0 + \theta_1 x$$

Q: How many parameters we have for the model?
A:

For linear regression problems, people typically use a loss function that measures the **distance** between the linear model's predictions and the training examples; **the objective is to minimize this distance.**

This is where the Linear Regression algorithm comes in: you feed it your training examples and it finds the parameters that make the linear model fit best to your data. This is called training the model. In our case the algorithm finds that the optimal parameter values are $\theta_0 = 4.85$ and $\theta_1 = 4.91 \times 10^{-5}$.



You are finally ready to run the model to make predictions. For example, say you want to know how happy Taiwanese are, and the OECD data does not have the answer. Fortunately, you can use your model to make a good prediction: you look up Taiwan's GDP per capita (https://www.ceicdata.com/en/indicator/taiwan/gdp-per-capita), find $24,318, and then apply your model and find that life satisfaction:

Q: So what is the life satisfaction in Taiwan?
A:

5.5 Theoretical Minimum for Machine Learning[2]

Proper mathematics and formal notations allow us to think abstract and generalize our understanding. In order to abstract the common core of the machine learning problem, we will pick one application and use it as a *metaphor* to illustrate the important mathematical concepts and notations for machine learning.

Metaphor Example: Credit Approval

[2] The phrase "theoretical minimum" is taken from a very successful book series written by Leonard Susskind, a great physicist at Stanford University. "Theoretical minimum" means just the minimum theories and equations you need to know in order to proceed to the next level.

A bank receives thousands of credit applications (e.g., credit cards, loans etc.) every day, and it wants to automate the process of evaluating them. The bank knows of no magical formula that can pinpoint when credit should be approved, but it has a lot of data. This calls for learning from data, so the bank uses historical records of previous customers to figure out a good mathematical model for credit approval.

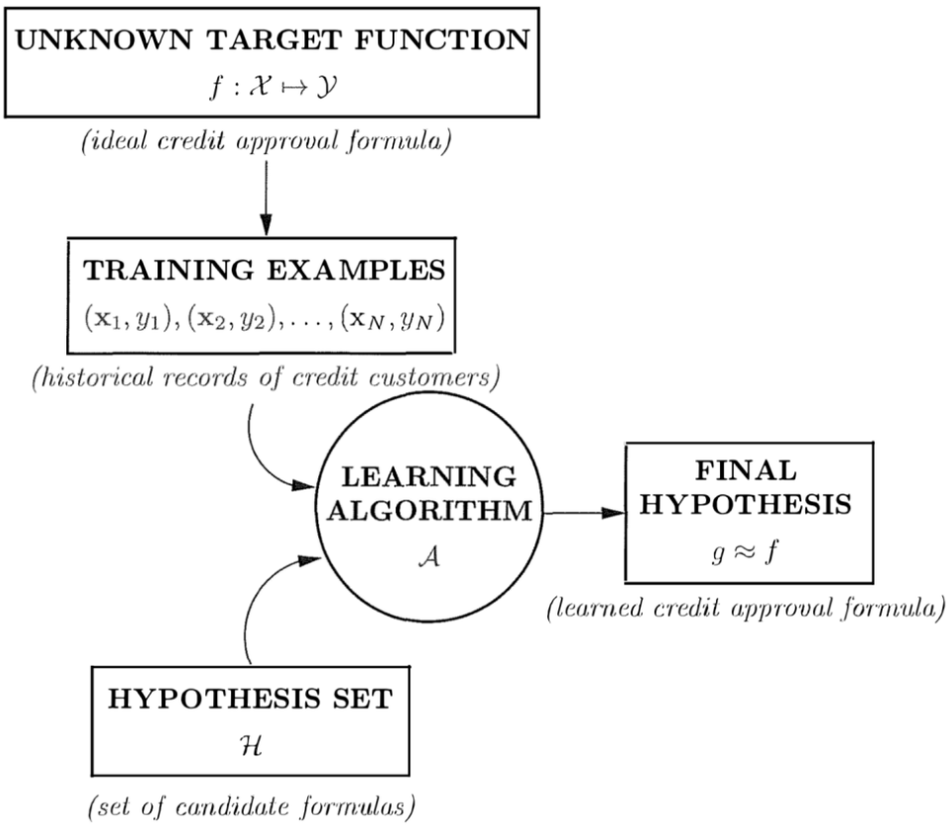
Each customer record has information related to credit. Let us assume the related customer information are annual salary, age, years in job, outstanding loans. The record also keeps track of correct credit decision. In this case, this might be whether approving credit for that customer was a good idea, i.e., did the bank make money on that customer. This data guides the construction of a successful mathematical model for credit approval that can be used on future applicants.

Let us give names and symbols to the main components of this learning problem. There is the input **x** (customer information that is used to make a credit decision), the unknown target function $f: \mathcal{X} \rightarrow \mathcal{Y}$ (ideal formula for credit approval), where $\mathcal{X}$ is the input space (**set** of all possible inputs **x**), and $\mathcal{Y}$ is the output space (**set** of all possible outputs, in this case just a yes/no decision). There is a data set $\mathcal{D}$ of input-output examples $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)$ where $y_n = f(\mathbf{x}_n)$ for $n = 1, \cdots, N$ (inputs corresponding to previous customers and the correct credit decision).

Q: What is the dimension of input space $\mathcal{X}$ for this credit approval example?
A:

Finally, there is the **learning algorithm** that uses the data set $\mathcal{D}$ to pick a formula $g: \mathcal{X} \rightarrow \mathcal{Y}$ that approximates f. The algorithm chooses g from a set of candidate formulas under consideration, which we call the hypothesis set $\mathcal{H}$. For instance, $\mathcal{H}$ could be the set of all linear formulas from which the algorithm would choose the best linear fit to the data.

When a new customer applies for credit, the bank will base its decision on g (the hypothesis that the learning algorithm produced), not on f (the ideal target function which remains unknown). The decision will be good only to the extent that g faithfully replicates f. To achieve that, the algorithm chooses g that best matches f on the **training** examples of previous customers, with the hope that it will continue to match f on new customers. Figure below illustrates the basic setup of the learning problem.



Remark: We should always remember that **the aim of machine learning is rarely to replicate the training data but the prediction for new cases**. That is we would like to be able to generate the right output for an input instance outside the training set, one for which the correct output is not given in the training set. How well a model trained on the training set predicts the right output for new instances is called **generalization**.

Recap and Generalization

Let us recapitulate and generalize what we have learned. Consider a data set $\mathcal{D}$ with *N* input-output data points:

$$\mathcal{D} = \{\mathbf{x}_n, y_n\}_{n=1}^{N}$$

where $\mathbf{x}_n$ is the input with arbitrary d dimensions and y_n is the desirable output. The data is **independent and identically distributed** (i.i.d). The i.i.d means that the ordering is not important and all instances are drawn from the same distribution.

Q: what is y_n for the binary classification?
A:

Q: what is y_n for regression learning?
A:

The aim is to build a good and useful approximation to $\mathbf{x}_n$ using the model $g(\cdot)$. In doing this, there are three decisions we must make:

1. **Model** we use in learning, denoted as:

   $g(\mathbf{x}|\theta)$

   where $g(\cdot)$ is the model, $\mathbf{x}$ is the input, and θ are the parameters.

   $g(\cdot)$ defines the hypothesis class $\mathcal{H}$, and a particular set of values of θ **instantiates one hypothesis** $h \in \mathcal{H}$. For example, in linear regression, the model is the linear function of the input whose slope and intercept are the parameters learned from the data.

   The model or $\mathcal{H}$, is fixed by the machine learning system designer based on his or her knowledge of the application and the hypothesis h is chosen **(parameters are tuned)** by a learning algorithm using the training set, sampled from the i.i.d.

2. **Loss function**, $L(\cdot)$, to compute the difference between the desired output, y_n, and our approximation to it, $g(\mathbf{x}_n|\theta)$, given the current values of the parameters, θ. The approximation error, or loss, is the sum of losses over the individual instances:

   $$E(\theta|\mathcal{D}) = \sum_{n=1}^{N} L\big(y_n, g(\mathbf{x}_n|\theta)\big)$$

   In binary classification where outputs are 0/1, $L(\cdot)$ checks for equality or not; in regression, because the output is a numeric value, we have ordering information for distance and one possibility is to use the square of the difference.

3. **Optimization procedure** to find θ^* that minimizes the total error.

   $\theta^* = \arg\min_\theta E(\theta|\mathcal{D})$

   where $\arg\min$ returns **the argument that minimizes**. In polynomial regression, we can solve analytically for the optimum, but this is not always the case. With other models and error functions, the complexity of the optimization problem becomes important.

For this setting to work well, the following conditions should be satisfied:

1. The hypothesis class of $g(\cdot)$ should be large enough, that is, have enough capacity, to include the unknown function that generated the data that is represented in $\mathcal{X}$ in a **noisy** form.
2. There should be enough training data to allow us to pinpoint the correct (or a good enough) hypothesis from the hypothesis class.
3. We should have a good optimization method that finds the correct hypothesis given the training data.

Different machine learning algorithms differ either in the models they assume (their hypothesis class), the loss measures they employ, or the optimization procedure they use. We will see many examples in the coming Chapters.

5.6 Introducing Scikit-Learn

There are several Python libraries which provide solid implementations of a range of machine learning algorithms. One of the best known is `Scikit-Learn`, a package that provides efficient versions of a large number of common algorithms. `Scikit-Learn` is characterized by a clean, uniform, and streamlined API, as well as by very useful and complete online documentation. **A benefit of this uniformity is that once you understand the basic use and syntax of `Scikit-Learn` for one type of model, switching to a new model or algorithm is very straightforward.**

This section provides an overview of the `Scikit-Learn` API; a solid understanding of these API elements will form the foundation for understanding the deeper practical discussion of machine learning algorithms and approaches in the following chapters.

We will start by covering data representation in `Scikit-Learn`, and followed by covering the Estimator (algorithm) API.

5.6.1 Data Representation in Scikit-Learn

Machine learning is about creating models from data: for that reason, we'll start by discussing how data can be represented in order to be understood by the computer. The best way to think about data within `Scikit-Learn` is in terms of **tables of data**.

Data as table

A basic table is a two-dimensional grid of data, in which the rows represent individual elements of the dataset, and the columns represent quantities related to each of these elements. For example, consider the Iris dataset as we mentioned before. We can download this dataset in the form of a Pandas `DataFrame` using the Seaborn library[3]:

```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Here each row of the data refers to a single observed flower, and the number of rows is the total number of flowers in the dataset. In general, we will refer to the rows of the matrix as *samples*, and the number of rows as `n_samples`. In the Iris set, we have `n_samples=150`.

Likewise, each column of the data refers to a particular quantitative piece of information that describes each sample. In general, we will refer to the columns of the matrix as features, and the number of columns as `n_features`.

Features matrix

This table layout makes clear that the information can be thought of as a two-dimensional numerical array or matrix, which we will call the features matrix. By convention, this features matrix is often stored in a variable named `X`. The features matrix is assumed to be two-dimensional, with shape `[n_samples, n_features]`, and is most often contained in a NumPy array or a Pandas `DataFrame`.

Target array

In addition to the feature matrix `X`, we also generally work with a label or target array, which by convention we will usually call `y`.

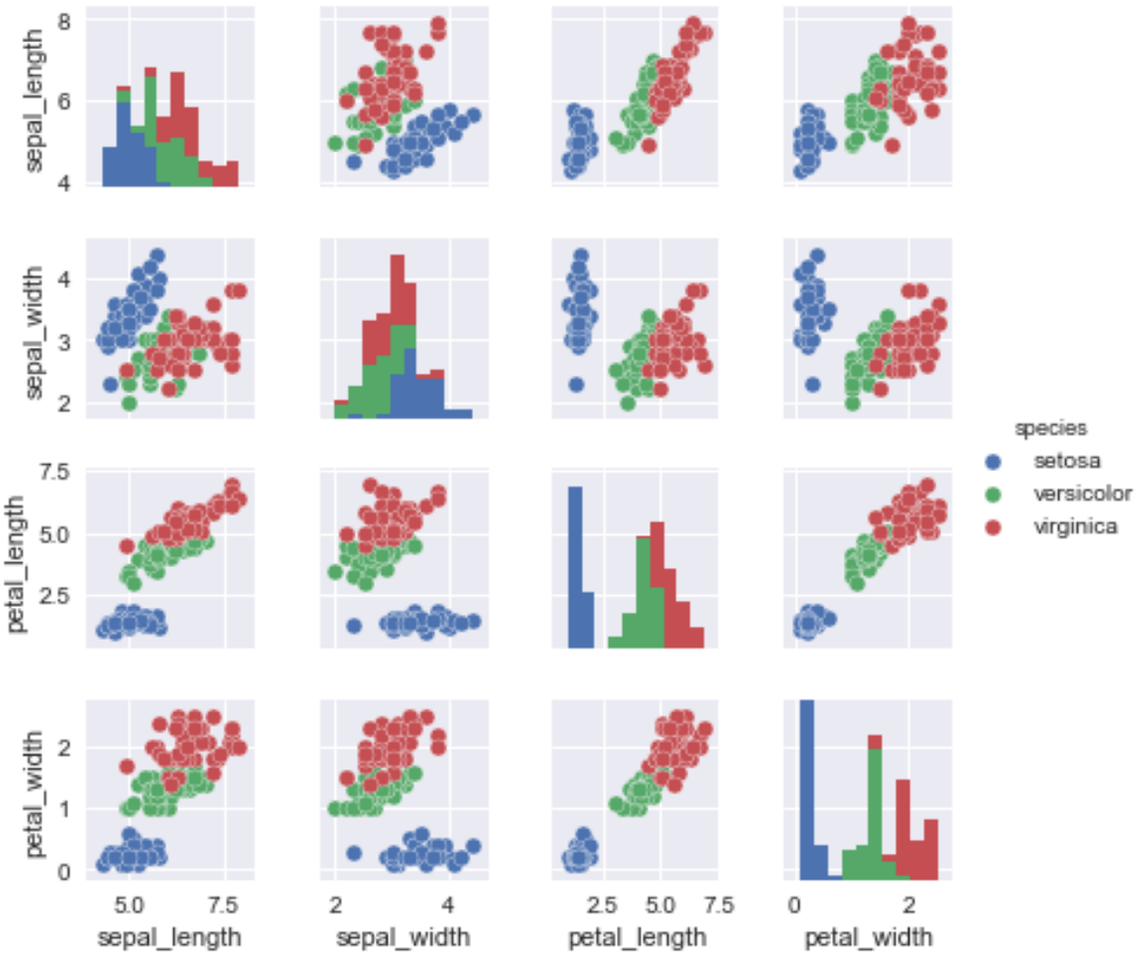
[3] Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. See https://seaborn.pydata.org

The target array is usually one dimensional, with length `n_samples`, and is generally contained in a NumPy array or Pandas `Series`. The target array may have continuous numerical values, or discrete classes/labels. While some `Scikit-Learn` estimators do handle multiple target values in the form of a two-dimensional, `[n_samples, n_targets]` target array, we will primarily be working with the common case of a one-dimensional target array.

The target array is the quantity we want to predict from the data. For example, in the preceding data we may wish to construct a model that can predict the species of flower based on the other measurements; in this case, the species column is the target array.

With this target array in mind, we can use Seaborn to conveniently visualize the data. Visualizing the multidimensional relationships among the samples is as easy as calling `sns.pairplot`:

```
%matplotlib inline
import seaborn as sns; sns.set()
sns.pairplot(iris, hue='species', size=1.5);
```



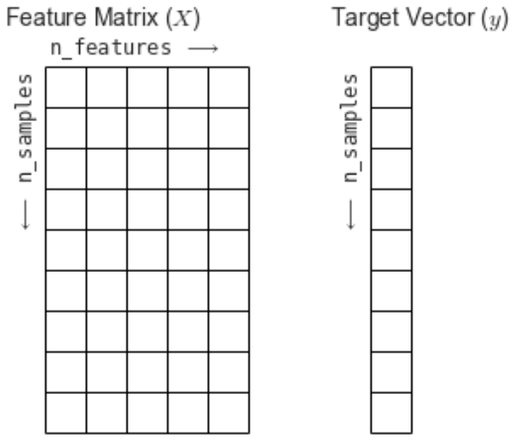
For use in `Scikit-Learn`, we will extract the features matrix and target array from the `DataFrame`, which we can do using some of the Pandas `DataFrame` operations:

```
X_iris = iris.drop('species', axis=1)
X_iris.shape
```

The parameter `axis=1` indicates to drop columns of the labels.

Q: what is the output?
A:

To summarize, the expected layout of features and target values is visualized in the following diagram:



With this data properly formatted, we can move on to consider the estimator API of `Scikit-Learn`.

5.6.2 Scikit-Learn's Estimator (Algorithm) API

The `Scikit-Learn` API is designed with the following guiding principles in mind:

- **Consistency**: All objects share a common interface drawn from a limited set of methods, with consistent documentation.

- **Inspection**: All specified parameter values are exposed as public attributes.

- **Limited object hierarchy**: Only algorithms are represented by Python classes; datasets are represented in standard formats (NumPy arrays, Pandas `DataFrames`) and parameter names use standard Python strings.

- **Composition**: Many machine learning tasks can be expressed as sequences of more fundamental algorithms, and `Scikit-Learn` makes use of this wherever possible.

- **Sensible defaults**: When models require user-specified parameters, the library defines an appropriate default value.

In practice, these principles make `Scikit-Learn` very easy to use, once the basic principles are understood. Every machine learning algorithm in `Scikit-Learn` is implemented via the Estimator API, which provides a consistent interface for a wide range of machine learning applications.

Basics of the API

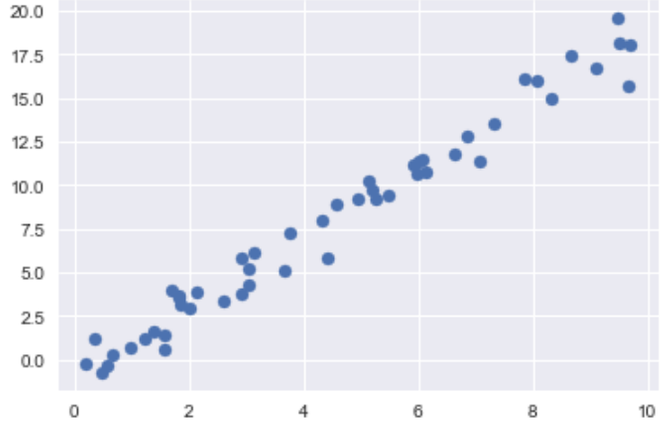
Most commonly, the steps in using the `Scikit-Learn` estimator API are as follows (we will step through a handful of detailed examples later).

1. Choose a class of model by importing the appropriate estimator class from `Scikit-Learn`.
2. Choose model hyperparameters[4] by instantiating this class with desired values.
3. Arrange data into a features matrix and target vector following the discussion above.
4. Fit the model to your data by calling the `fit()` method of the model instance.
5. Apply the Model to new data:
    - For supervised learning, often we predict labels for unknown data using the `predict()` method.
    - For unsupervised learning, we often transform or infer properties of the data using the `transform()` or `predict()` method.

We will now step through two simple examples of applying supervised learning methods.

Supervised learning example: Simple linear regression

As an example of this process, let's consider a simple linear regression—that is, the common case of fitting a line to `(x,y)` data. We will use the following simple data for our regression example:




```
import matplotlib.pyplot as plt
import numpy as np

rng = np.random.RandomState(42)
x = 10 * rng.rand(50)
y = 2 * x - 1 + rng.randn(50) #normal distribution
plt.scatter(x, y);
```

[4] In machine learning, hyperparameters refer to parameters that must be set before the model is fit to data.

With this data in place, we can use the recipe outlined earlier. Let's walk through the process:

1. Choose a class of model

   In `Scikit-Learn`, every class of model is represented by a Python class. So, for example, if we would like to compute a simple linear regression model, we can import the linear regression class:

```
from sklearn.linear_model import LinearRegression
```

2. Choose model hyperparameters

   An important point is that a class of model is not the same as an instance of a model.

   Once we have decided on our model class, there are still some options open to us. Depending on the model class we are working with, we might need to answer one or more questions like the following:

- Would we like to fit for the offset (i.e., intercept)?
- Would we like the model to be normalized?
- Would we like to preprocess our features to add model flexibility?
- What degree of regularization would we like to use in our model?
- How many model components would we like to use?

   These are examples of the important choices that must be made once the model class is selected. These choices are often represented as **hyperparameters** (the parameters that must be set before the model is fit to data). In `Scikit-Learn`, hyperparameters are chosen by passing values at model instantiation. We will explore how you can quantitatively motivate the choice of hyperparameters in sequel.

   For our linear regression example, we can instantiate the `LinearRegression` class and specify that we would like to fit the intercept using the `fit_intercept` hyperparameter:

```
model = LinearRegression(fit_intercept=True)
model
```

   Keep in mind that when the model is instantiated, the only action is the storing of these hyperparameter values. In particular, we have not yet applied the model to any data: the `Scikit-Learn` API makes very clear the distinction between **choice of model and application of model to data**.

3. Arrange data into a feature matrix and target vector

   Previously we detailed the `Scikit-Learn` data representation, which requires a two-dimensional features matrix and a one-dimensional target array. Here our target variable `y` is already in the correct form (a length-`n_samples` array), but we need to massage the data `x` to make it a matrix of size `[n_samples, n_features]`. In this case, this amounts to a simple reshaping of the one-dimensional array:

```
X = x[:, np.newaxis]
X.shape #output (50,1)
```

   The `newaxis` object is used in all slicing operations to create an axis of length one.

4. Fit the model to your data

   Now it is time to apply our model to data. This can be done with the `fit()` method of the model:

```
model.fit(X, y)
```

   This `fit()` command causes a number of model-dependent internal computations to take place, and the results of these computations are stored in model-specific attributes that the user can explore. In `Scikit-Learn`, by convention all model parameters that were learned during the `fit()` process have trailing underscores; for example in this linear model, we have the following:

```
model.coef_
model.intercept_
```

   These two parameters represent the slope and intercept of the simple linear fit to the data. Comparing to the data definition, we see that they are very close to the input slope of `2` and intercept of `-1`.

5. Predict labels for unknown data

   Once the model is trained, the main task of supervised machine learning is to evaluate it based on what it says about new data that was not part of the training set. In `Scikit-Learn`, this can be done using the `predict()` method. For the sake of this example, our "new data" will be a grid of `x` values, and we will ask what `y` values the model predicts:

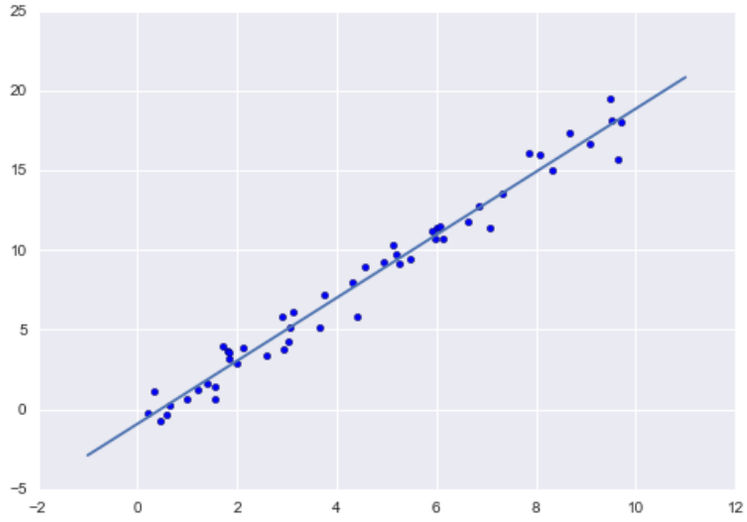
```
xfit = np.linspace(-1, 11)
```

As before, we need to reshape these `x` values into a `[n_samples, n_features]` features matrix, after which we can feed it to the model:

```
Xfit = xfit[:, np.newaxis]
yfit = model.predict(Xfit)
```

Finally, let's visualize the results by plotting first the raw data, and then this model fit:

```
plt.scatter(x, y)
plt.plot(xfit, yfit);
```



Recipe Recap and Summary: **import/instantiate/fit/predict**

The process of doing machine learning using `Scikit-Learn` consists of the following **four** steps:

1. Choose a `class` of model by `import`ing the appropriate estimator class from `Scikit-Learn`.

```
from sklearn.linear_model import LinearRegression
```

2. Choose model hyperparameters by instantiating this `class` with desired values. For example

```
model = LinearRegression(fit_intercept=True)
```

3. Fit the model to your data by calling the `fit()` method of the model instance.

```
model.fit(X, y)
```

4. Apply the Model to new data:

- For supervised learning, we often predict labels for unknown data using the `predict()` method.

```
yfit = model.predict(Xfit)
```

Supervised learning example: Simple classification

Let's take a look at another example of this import/instantiate/fit/predict **recipe**, using the Iris dataset we discussed earlier. Our question will be this: **given a model trained on a portion of the Iris data, how well can we predict the remaining labels**?

For this task, we will use a support vector machine classifier (SVC). We would like to evaluate the model on data it has not seen before.

(Why evaluate the model on data it has not seen before? Conceptual Analogy) Before the final exam, I may hand out some practice problems and solutions to the class. Although these problems are not the exact ones that will appear on the exam, studying them will help you do better. They are the 'training set' in your learning.

If my goal is to help you do better in the exam, why not give out the exam problems themselves?

Q: Why not?
A:

We will **split the data into a training set and a testing set**. This could be done by hand, but it is more convenient to use the `train_test_split` utility function from `Scikit-Learn`:

```
from sklearn.datasets import load_iris
iris = load_iris()
X_iris, y_iris = iris.data, iris.target
```

```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,
random_state=1)
```

With the data arranged, we can follow our import/instantiate/fit/predict **recipe** to predict the labels:

```
from sklearn.svm import SVC # 1. choose "Support vector classifier"
model = SVC(kernel='rbf', gamma=0.01, C=10) # 2. instantiate model
```

```
model.fit(Xtrain, ytrain)                    # 3. fit model to data
y_model = model.predict(Xtest)               # 4. predict on new data
```

Finally, we can use the `accuracy_score` utility to see the fraction of predicted labels that match their true value:

```
from sklearn.metrics import accuracy_score
accuracy_score(ytest, y_model)
```

```
0.97368421052631582
```

The `accuracy_score` function computes the accuracy. If $\hat{y}_i$ is the predicted value of the i^{th} sample and y_i is the corresponding true value, then the fraction of correct predictions over N samples is defined as:

$$\text{accuracy}(y_i, \hat{y}_i) = \frac{1}{N}\sum_{n=1}^{N} 1(\hat{y}_i = y_i)$$

The $1(\hat{y}_i = y_i)$ is an indicator function; it is 1.0 if $\hat{y}_i = y_i$ and 0.0 if $\hat{y}_i \neq y_i$.

With an accuracy topping 97%, we see that this classification algorithm is effective for this dataset.

5.6.3 Summary

In this section we have covered the essential features of the `Scikit-Learn` data representation, and the estimator API. Regardless of the type of estimator, the same **import/instantiate/fit/predict** process holds. Armed with this information about the estimator API, you can explore the `Scikit-Learn` documentation and begin trying out various models on your data. See http://scikit-learn.org/stable.