Engineer

Machine learning

building software that learns a task from examples in data

What it is

Machine learning is the practice of building software that learns a task from examples instead of from rules a programmer writes out. You give it many cases where the answer is already known, it finds a relationship between the inputs and the answers, and it uses that relationship to predict the answer for each new case. The name is older than nearly all the software that now uses it: A. L. Samuel put it in the title of a study of the game of checkers that the IBM Journal of Research and Development printed in July 1959.

The vocabulary is easiest to pin down with scikit-learn, the open source Python library whose documentation this article follows. The examples sit in a table. Each row is one sample, such as an invoice, a customer or a sensor reading, and each column is a feature, a number that describes the sample. The answer you want the software to learn is the target. In scikit-learn's terms the thing that learns is an estimator. Every estimator has a `fit` method that learns from the rows you show it. Predictive estimators use `predict` to give answers for new rows; transformers use `transform` to produce a transformed sample table.

Two families cover most of the work. In supervised learning every training row comes with its target. When the target is a category, such as paid on time or paid late, the task is classification; when it is a quantity, such as the number of days until payment, it is regression. In unsupervised learning there is no target at all, and the software looks for structure by itself, for instance by grouping customers who behave alike. The scikit-learn documentation gives each family a chapter of its own, followed by one on model selection and evaluation, which is where most of the practical difficulty lives.

What machine learning is not, at least in this sense, is a word for any software that seems clever. A rule that flags every invoice above a fixed amount was written by a person and does exactly what it says. A model that flags invoices has derived its rule from past invoices, so it can be wrong in ways nobody wrote down, and its behaviour changes whenever the data it learns from changes.

Use the concept in a decision

Put “What is machine learning?” to work in a real decision. We can help relate the concept to the system you are building or buying. An agent wired to the systems you use, tested on cases your own experts write, overseen by your people and never tied to one model.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Built a 268-check guard engine for the repo — 268 commit checks, and 146 tests of the checks themselves

Built a 268-check guard engine for the repo Get in touch →

Why it matters

Some tasks are far easier to show by example than to describe. Nobody can write a complete rule for which invoices will be paid late, but most companies hold years of invoices with the outcome recorded. Machine learning turns that record into a working rule, and it can do the same for many such questions at once, which is why it has moved from research into ordinary business software.

The price is that the data now does the job the code used to do. If the history holds mistakes, duplicates or gaps, the model learns those as well, and nothing in the code shows it. D. Sculley and nine colleagues at Google described what follows in *Hidden Technical Debt in Machine Learning Systems*, a paper in the proceedings of the 2015 NIPS conference. Their first figure draws the learning code as a small black box in the middle of a much larger system, and the caption calls the infrastructure around it vast and complex. In one passage they estimate that a mature system may end up as at most 5% machine learning code and at least 95% glue code that moves data in and out of general-purpose packages.

They also argue that a dependency on data costs more than a dependency on code. Compilers and build systems trace which code depends on which, while tools that do the same for data are far less common, so a column can change meaning without anything noticing. And they name a principle, CACE, for “Changing Anything Changes Everything”: shift the distribution of one input and the way the model uses every other input may shift with it.

So a model is cheap to make and expensive to keep. The work that decides whether it helps is mostly about data: where it comes from, whether each customer appears once or three times, whether the recorded outcome is right, and who will notice when a system upstream starts sending something different.

What it looks like in practice

Take a company that wants to know, on the day it issues an invoice, which invoices are likely to be paid late, so that someone can call those customers before the due date rather than after it. An export holds one row per past invoice, sorted by issue date, with four columns that were known on the day each invoice went out: the amount, the days of credit granted, how many invoices the customer received in the year before, and how many of those were paid late. A fifth column records what happened: 1 if the invoice was paid late, 0 if it was paid on time.

The script below was written for this article and checked line by line against the scikit-learn 1.9.1 and NumPy 2.5 documentation in October 2026. It trains a model on the older invoices and tests it on the newest quarter:

```
# invoices.csv: one row per invoice, sorted by issue date, numbers only.
# The last column is 1 if the invoice was paid late, 0 if on time.
import numpy as np
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import balanced_accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

data = np.loadtxt("invoices.csv", delimiter=",", skiprows=1)
X, y = data[:, :-1], data[:, -1].astype(int)

# Hold the newest quarter back: nothing learns from it.
X_old, X_new, y_old, y_new = train_test_split(
    X, y, test_size=0.25, shuffle=False
)

model = make_pipeline(StandardScaler(), LogisticRegression())
baseline = DummyClassifier(strategy="most_frequent")
for name, candidate in [("model", model), ("baseline", baseline)]:
    candidate.fit(X_old, y_old)
    score = balanced_accuracy_score(y_new, candidate.predict(X_new))
    print(name, round(score, 3))
```

Four choices in it matter more than the choice of algorithm.

The newest quarter is set aside before anything learns. With `shuffle=False`, `train_test_split` keeps the rows in order and puts the last quarter in the test set, so the model learns from older invoices and is judged on newer ones, which is how it will be used. With the default, `shuffle=True`, the split would be random, and the model would partly be tested on invoices older than some it learned from.

The scaling sits inside the pipeline. `StandardScaler` removes each feature's mean and scales it to unit variance. Because `make_pipeline` chains it to the classifier, the mean and variance come from the training rows alone, and nothing about the newest quarter seeps into the model.

The score is balanced accuracy. It averages, over the two classes, the share of each class the model got right: the late invoices it caught and the punctual ones it recognised. If one invoice in ten is paid late, plain accuracy rewards a model for answering "on time" every time. Balanced accuracy does not.

The model has to beat a baseline. `DummyClassifier` with `strategy="most_frequent"` ignores the features and always gives the commonest answer. On balanced accuracy it scores 0.5 with two classes, since it gets every punctual invoice right and every late one wrong. A model that does not clearly beat it has learned nothing useful, however good its plain accuracy looks.

None of this rescues a model from bad rows. If one customer exists as three records, the count of that customer's late invoices is split three ways, and the model learns from a history that never happened. Our page on AI Data Readiness is about that groundwork: a single record per customer, each person able to see only what they should, and any answer traceable to the data behind it. For a model, the last of those is what lets you walk back from a strange prediction to the rows that produced it.

Where it goes wrong

Information from the future gets into training. The scikit-learn guide to common pitfalls calls this data leakage: building the model with information that will not exist when it makes a prediction. In the invoice example, the tempting extra column is the number of reminders sent, which predicts lateness superbly

because reminders go out only once an invoice is overdue. The same leak happens more quietly when a scaler or an imputer is fitted on every row before the split, and the guide's rule is never to call `fit` on the test data. The fix is to write down, for each column, the moment its value becomes known, to split first, and to keep every preprocessing step inside the pipeline so that it is fitted on training rows only.

The model is judged on what it has already seen. A model that only repeated the targets it was trained on would score perfectly on its own training rows and be useless on anything new; the cross-validation guide calls this overfitting. A subtler form is to try many settings and keep whichever scores best on the held-out rows, which lets knowledge of those rows leak into the choice. The fix is to compare settings with cross-validation on the training part, where the rows are cut into several folds and each fold takes a turn as the test set, and to look at the held-out rows once, at the end.

A high accuracy that a constant could match. When one class is rare, accuracy is dominated by the common one, and the guide warns that with rare classes, as in fraud detection, a fold can end up with no example of the rare class at all. Report balanced accuracy or another per-class measure, compare with a `DummyClassifier`, and use stratified folds, which keep the class proportions roughly equal in every fold. `cross_val_score` uses them by default for a classifier when you pass it a number of folds.

A random split for data that moves over time. The cross-validation guide notes that the assumption of independent, identically distributed samples rarely holds in practice, and that data ordered in time should be tested on observations later than the ones the model learned from. A random split lets the model peek at the future and reports a score it will not reach. Split by date, as the example does, and when cross-validating use a time-aware splitter such as `TimeSeriesSplit`, whose folds the guide says should each cover the same span of time.

Preprocessing that differs between training and production. If the model learned from scaled features and the live system sends raw ones, the predictions are wrong and no error is raised. The pitfalls guide requires any transformation used in training to be applied to test data and to data in production alike. Ship the fitted pipeline as one object, rather than a model plus a note on what to do to the data first.

The world changes after launch. A model trained on last year's invoices assumes that customers still behave as they did then. Sculley and his co-authors suggest tracking prediction bias: in a system that works as intended, the share of each predicted label should usually match the share actually observed, and a gap that opens between them is often the first sign that the world has moved. Keep that comparison running, retrain on recent data, and hold a frozen, versioned copy of any input signal another team owns, so that a well-meant change upstream does not silently change your model.

Further reading

- User Guide — scikit-learn's reference for supervised and unsupervised learning and for model selection and evaluation, chapter by chapter.
- Getting Started — the estimator interface in a few short examples: fitting and predicting, transformers, pipelines and cross-validation.
- Common pitfalls and recommended practices — inconsistent preprocessing, data leakage and randomness, with the wrong way and the right way side by side in code.
- Cross-validation: evaluating estimator performance — why testing on training data misleads, and which splitting scheme suits grouped, imbalanced or time-ordered data.
- Hidden Technical Debt in Machine Learning Systems — Sculley and colleagues at Google, NIPS 2015, on what running a model in production costs beyond the model itself.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is machine learning?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/machine-learning/