ML - Machine Learning

Learning from data

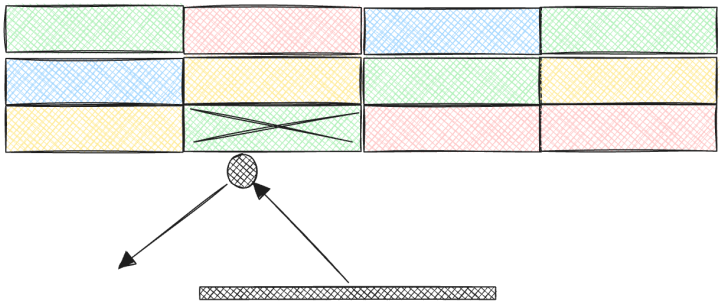
Riccardo Berta

2026.02.25

Contents

1 Learning from data

Machine Learning (ML) is a **programming paradigm** where, instead of **explicitly coding a solution**, we enable a computer **to learn how to solve a problem**. This approach is particularly valuable for tackling challenges that are difficult or even impossible to address with traditional programming alone. Consider a scenario where a solution must be expressed by defining explicit rules that govern a program's behavior. For instance, take the bricks game: the ball follows a path that changes whenever it hits a brick. As programmers, we would need to define the behavior in detail. How does the ball bounce? Which brick should disappear after a hit? What happens when the ball hits a wall or misses the paddle? Every aspect of the game must be meticulously designed, hardcoded, and tested in advance:

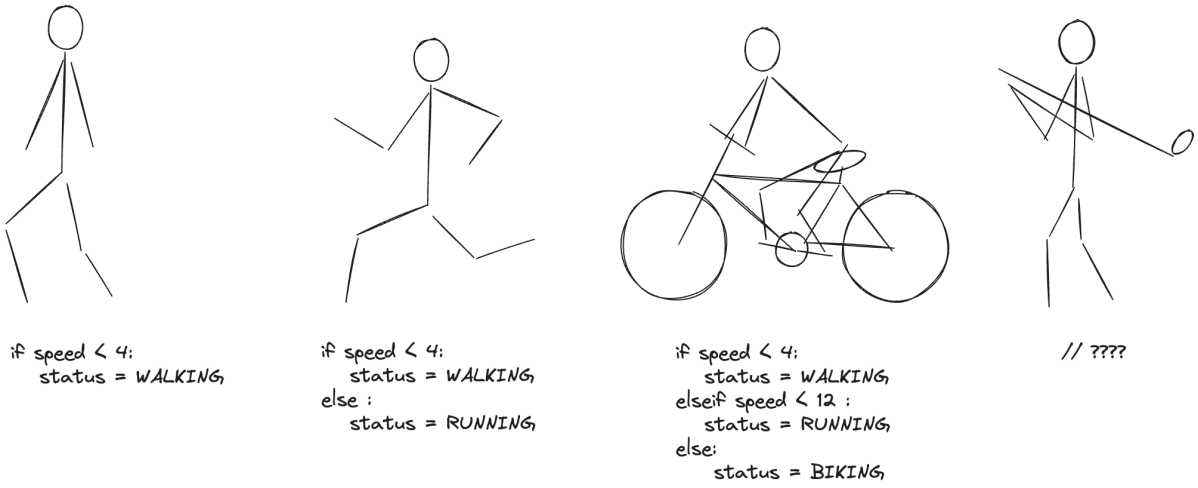


```
if ball.collide(brick) :
    removeBrick();
    ball.dy = -ball.dy;
```

In traditional software, **the programmer designs an algorithm that processes an input, applies predefined rules, and produces an output**. The inner workings of the algorithm are meticulously crafted and explicitly implemented in the code by the programmer. Using data (such as the positions of the ball and bricks in the example) the program determines outcomes: is the game over? should the score increase? where should the ball be positioned next? **Every decision is dictated by the rules hardcoded into the program**:

This approach works well for many problems and is incredibly powerful, but it has its limitations. As the complexity of a scenario increases, the amount of code the programmer needs to write and manage grows significantly. In many cases, **the programmer faces scenarios that are simply too complex or impractical to fully define and implement through traditional coding alone**.

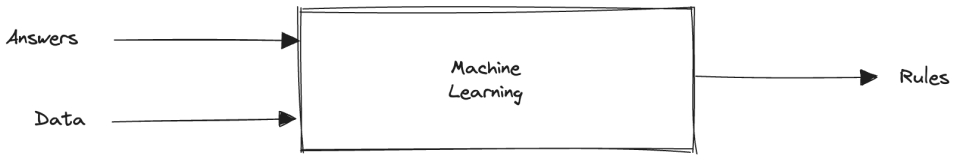
Consider the challenge of **activity detection**, a task commonly handled by smartwatches and smartphones today. The goal is to identify what activity a person is performing, such as walking, running, biking, or golfing. Using a phone's sensors, we can measure the user's speed and apply simple rules to infer the activity. For instance, if the speed is below a certain threshold, the user is likely walking; if it's above a higher threshold, they might be running. We could set another threshold for biking. But what about golfing? What kind of rule could we write to accurately determine that the user is playing golf? Defining such rules becomes increasingly difficult as the activities grow more nuanced or diverse.



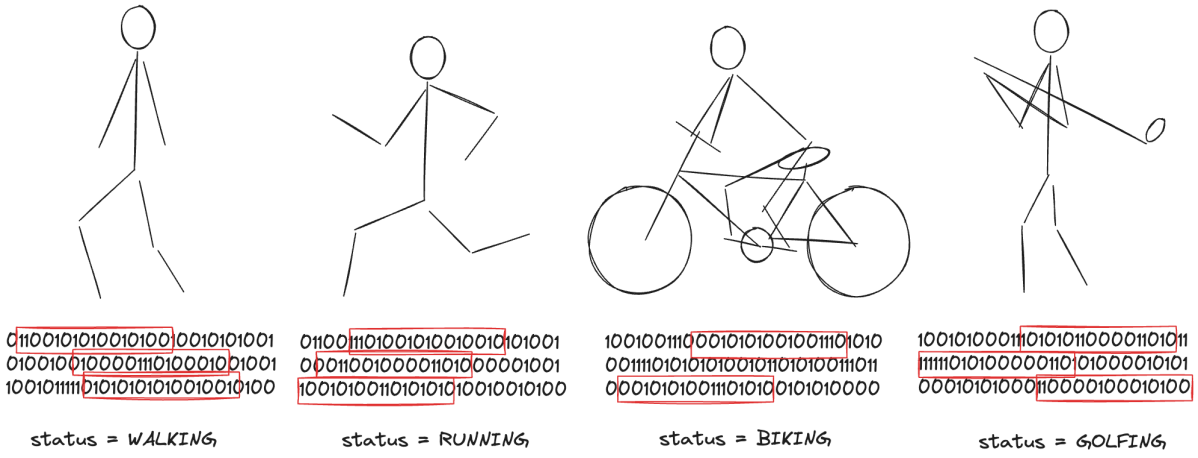
Rules are a little bit naive. We can't just go by speed. We might run downhill faster than the bike uphill, for example. Our rules don't really work that well.

Rules can be overly simplistic and fail to capture the complexity of real-world scenarios. For example, a person might run downhill faster than someone biking uphill, making speed alone an unreliable indicator. These kinds of **edge cases** highlight **the limitations of rule-based approaches**, they often fall short in handling the nuances and variability of real-life activities.

Machine learning offers a solution to this problem. Instead of manually figuring out the rules that process data to produce an answer, what if we flipped the approach? What if **we provided the computer with both the data and the corresponding answers, and let it figure out the rules that connect them?** This way, the computer **learns to generalize patterns and relationships**, removing the need for us to explicitly define every rule.

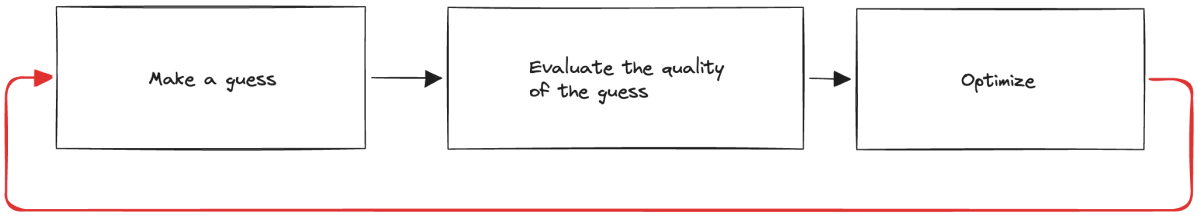


In the activity detection scenario, we could collect a large amount of sensor data and label it with the corresponding activity the user is performing. By analyzing this labeled data, the computer can learn to identify patterns and relationships, allowing it to determine the rules that distinguish activities like walking, running, biking, or even golfing. This eliminates the need to manually define rules, as the computer discovers them directly from the data.

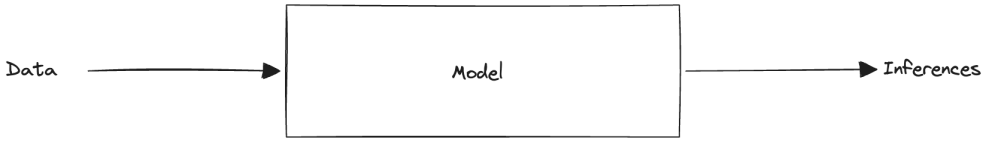


First, the computer will **make a guess** about to the relationship between the data and its labels, then it **measures how good or how bad that guess is** (**accuracy**), then it use this measure **to figure out a new better guess**, based on what it already know. If we repeat the process, each subsequent guess gets better than the previous one. It's **learning from experience** what the best guess might be:

First, the computer starts by making **an initial guess about the relationship between the data and its labels**. It then **evaluates the quality of that guess** by measuring how close it is to the correct answer. Using this feedback, it **adjusts its approach to make a better guess**, building on what it has already learned. By repeating this process iteratively, **each guess improves upon the last**, allowing the computer **to learn from experience** and gradually converge on the best possible solution:



This process **builds a model** that can then be used **to label new, unseen data**, a process known as **inference**:



A model is essentially a **mathematical representation of the relationships between different variables**. For example, if you're raising funds for your social networking site, you might create

a **business model** that takes inputs like "number of users," "ad revenue per user" and "number of employees" to predict the annual profit over the next few years. Similarly, a **cookbook recipe** is a model that connects inputs like "number of diners" and "level of hunger" to the required quantities of ingredients. Or consider an **electronic circuit design**, where the output voltage is modeled based on inputs such as the supply voltage, resistance, capacitance, and current flow. These models are **built using different methods**: the business model often relies on **simple mathematical relationships**, like profit equals revenue minus expenses, and revenue is units sold times average price. The recipe model may be based on **trial and error**, developed through experimentation in the kitchen. Meanwhile, the circuit model is derived from **physical laws**, such as Ohm's law and Kirchhoff's circuit laws, which describe the behavior of electrical components and their interactions. **Each model serves as a framework for making informed predictions based on relevant inputs**.

We'll use the term **machine learning** to describe the process of creating and using models that are learned from data. The primary goal is typically to use existing data to develop models capable of predicting outcomes for new, unseen data. A widely recognized definition, provided by Tom Mitchell in Machine Learning, 1998, states:

> A computer program is said to learn from experience (E) with respect to some task (T) and some performance measure (P), if its performance on T, as measured by P, improves with experience E.

For example, in a spam classification problem, the task (T) is to predict whether an email is spam or not spam. The experience (E) consists of labeled data, where emails are marked as spam or not spam by users. The performance measure (P) is the proportion of emails correctly classified by the model. As the program is exposed to more labeled data, its ability to correctly classify emails improves, demonstrating learning from experience.

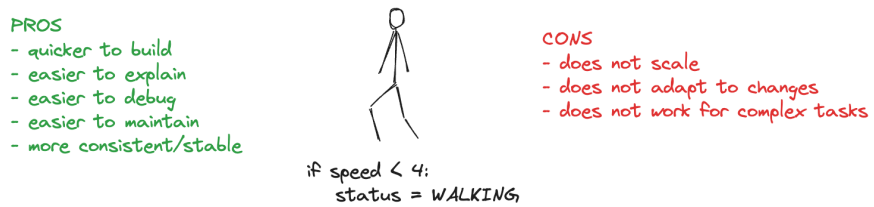
In summary, creating a machine learning program involves **feeding data** (such as dashboard video and sensor readings from a car) along with **target outputs** (like throttle levels and steering angles) into **a specialized algorithm**. Through a process called **training**, the algorithm **learns from the data, discovers the underlying rules and builds a model** (a program capable of driving the car based on sensor inputs). This model can then process new data to make predictions, a process referred to as **inference**. This approach allows us to create systems that make predictions from complex data **without needing to fully understand all the complexities ourselves**. In essence, traditional programming involves automating tasks by writing explicit programs. In machine learning, the computer generates a program by learning patterns from the data.

We can categorize machine learning appoached as **supervised** (in which there is a set of data labeled with the correct answers to learn from), **unsupervised** (in which there are no such

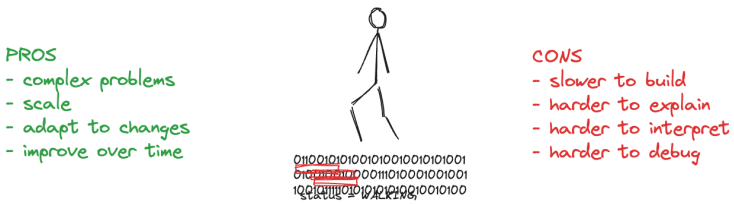
labels), **semisupervised** (in which only some of the data are labeled), and **reinforcement** (in which, after making a series of predictions, the model gets a signal indicating how well it did).

Question	**Supervised**	**Unsupervised** | **R	**Reinforcement** |
Target	f(x)=y	x in X | p(s	p(s)=a |
Target (*rephrased*)	Predict outputs given inputs	Discover structure in data | Fin	Find an optimal behavior |
Data	{(x,y)} supervisor's labels	{x} unlabelled data | {(s	{(s,a,r,s)} experience samples |
Output	Classifier or regressor	Clusters or dimension reduction | Pol	Policies, value functions |
Key algorithms	Neural networks, SVMs, etc.	k-means, PCA, etc. | Q-l	Q-learning, Policy Gradients, etc. |

As a designer, it's important to remember that **machine learning is not always** the best solution. In some cases, traditional programming can be just as effective, if not better. Traditional programming excels in scenarios where data volumes are low, and the rules are straightforward. It is faster to develop, easier to explain, debug, and maintain, and typically delivers more consistent performance. However, traditional programming has its limitations. It does not scale well, cannot adapt or update automatically, and struggles with complex tasks that require an overwhelming number of rules or where the rules are too difficult for domain experts to define. In such cases, machine learning can provide a more efficient and scalable alternative.



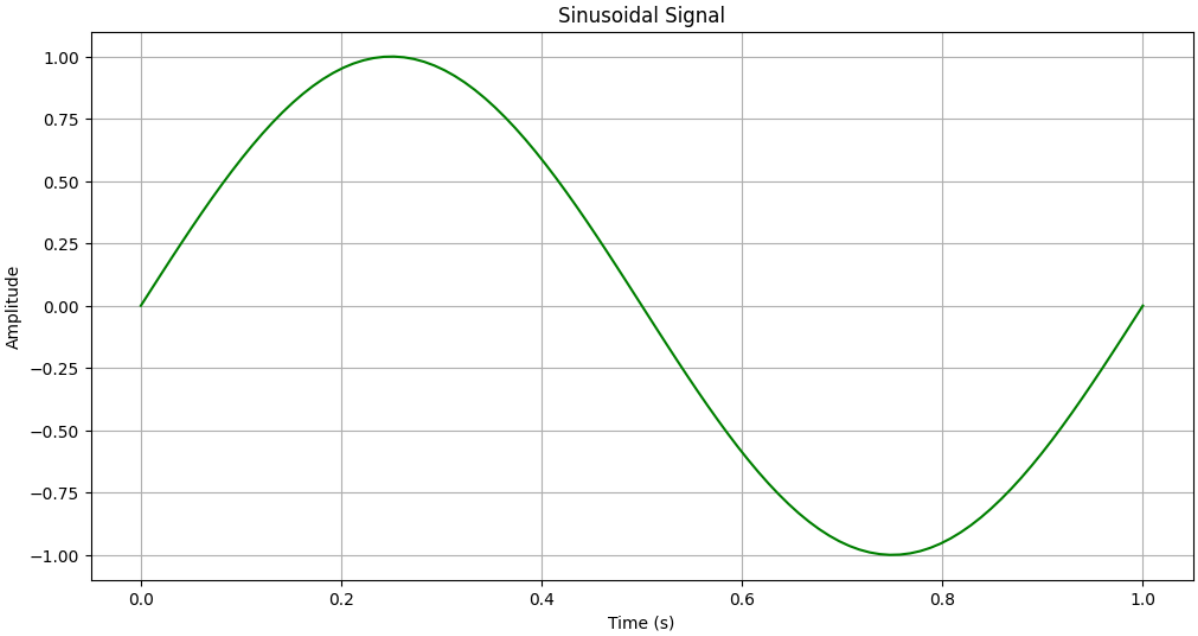
In contrast, machine learning provides several key advantages over traditional programming. It is well-suited for solving complex problems, can scale efficiently, adapt and update automatically, deliver personalized user experiences, and improve over time as it learns from new data. However, machine learning also has its downsides. It is slower to develop, requiring significant amounts of data collection and time to train the model. Additionally, machine learning models are often difficult to interpret and explain, a challenge known as the **black box problem**, which also makes them harder to debug and troubleshoot.



1.1 A gentle introduction

Imagine we have a temporal signal, such as a voltage, current, or any other measurable quantity, denoted as $y(t)$. While the exact shape of the signal is unknown, we are provided with a dataset containing N observations of the signal. This dataset is referred to as the **training set**. Our objective is to **predict the value of new, unseen input**. The ability to make accurate predictions for previously unseen inputs is the primary goal of machine learning and is referred to as **generalization**. To better understand this concept, let's assume that the underlying signal follows a sinusoidal pattern:

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

# parameters for the sinusoidal signal
amplitude = 1        # amplitude of the sinusoid
frequency = 1        # frequency in Hz
sampling_rate = 100  # samples per second
duration = 1         # duration in seconds

# generate a time vector
t = np.linspace(0, duration, int(sampling_rate * duration))

# generate the sinusoidal signal
sinusoid = amplitude * np.sin(2 * np.pi * frequency * t)

# plot the sinusoidal signal
plt.figure(figsize=(12, 6))
plt.plot(t, sinusoid, color="green")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Sinusoidal Signal")
plt.grid(True)
plt.show()
```



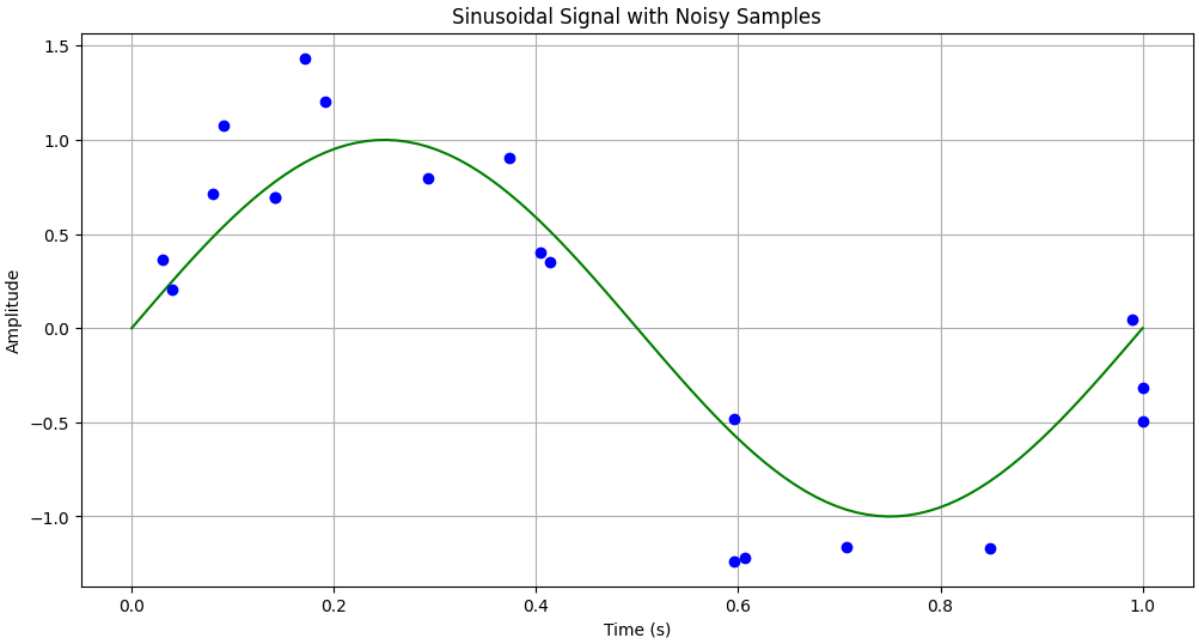
We can create a **synthetic dataset** by leveraging our knowledge of the true process that generates the data. Specifically, we can use random sampling to select N data points from the signal:

```
# number of sample to generata
N = 20

# create some samples
sample_indices = np.sort(np.random.choice(len(t), size=N))
x = t[sample_indices]
```

To generate the corresponding target values, we compute the values of the sinusoidal function for each sample and then we **add a small amount of random noise** drawn from a Gaussian distribution. This approach reflects an important characteristic of many real-world datasets: **while they exhibit an underlying regularity that we aim to learn, individual observations are often distorted by random noise**. This noise may stem from **inherently stochastic processes**, but more commonly arises from **unobserved sources of variability** that influence the system:

```
# set the random seed for reproducibility
np.random.seed(42)

# standard deviation of the noise
noise_std = 0.35
```

```python
# generate target values with noise
y = sinusoid[sample_indices] + np.random.normal(0, noise_std, size=N)

# plot the noisy samples
plt.figure(figsize=(12, 6))
plt.plot(t, sinusoid, color="green")
plt.scatter(x, y, color="blue", zorder=5)
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Sinusoidal Signal with Noisy Samples")
plt.grid(True)
plt.show()
```



In this example, we know the true process that generated the data. However, in real-world applications, our goal is to uncover the underlying patterns in the data using only a finite training set. While knowing the true process helps illustrate key machine learning concepts, our task is to use the training set to predict the target variable for new input values. This involves implicitly attempting to **discover the underlying function** that relates inputs to outputs.This is a challenging problem, as it requires generalizing from a limited dataset to an entire function. Complicating matters further, the observed data is corrupted by noise. **Probability theory** offers a framework for precisely and quantitatively representing this uncertainty, while **decision theory** enables us to leverage this probabilistic representation to make predictions that are optimal based on specific criteria. At its core, machine learning involves learning these probabilistic

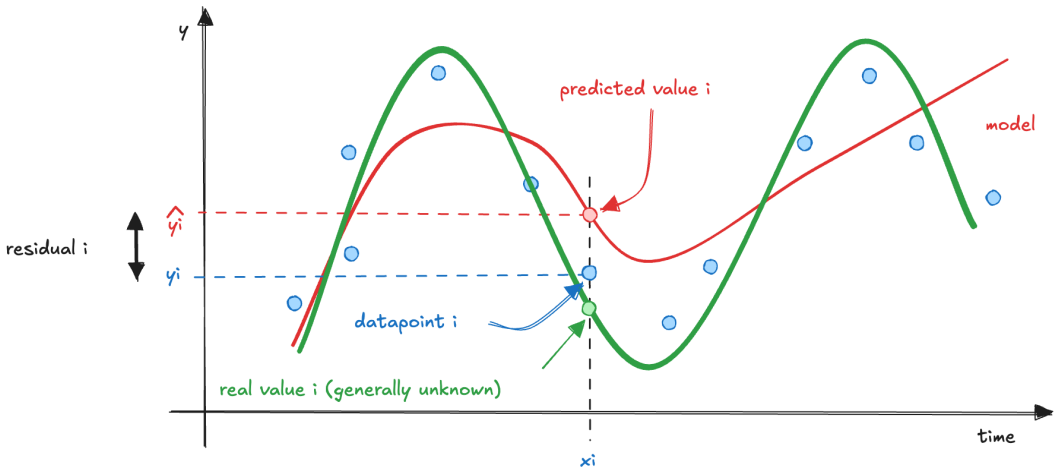
relationships from data. To keep things simple we select **a polynomial model** of the form:

$$\hat{y}(x, w) = w_0 + w_1 x + w_2 x^2 + ... + w_m x^m = \sum_{j=0}^{m} w_j x^j$$

Here, m represents the order of the polynomial model, and w is the vector of polynomial coefficients that need to be determined to adapt the model to the problem. The values of these coefficients are obtained by fitting the polynomial to the training data. This is achieved by **minimizing an error function** that quantifies the discrepancy between the model's predictions (for a given w) and the actual training data points. A commonly used error function is the **sum of squared residuals**, where the residual for each data point is the difference between the model's prediction and the corresponding target value. This approach seeks to minimize the total squared difference across all data points, ensuring the model aligns as closely as possible with the observed data:

$$E(w) = \sum_{n=1}^{N} (\hat{y}(x_n, w) - y_n)^2$$

It is important to note that the error function is always a nonnegative quantity, becoming zero only when the model perfectly predicts every training data point. A geometrical interpretation of the sum-of-squares error function is as follows:

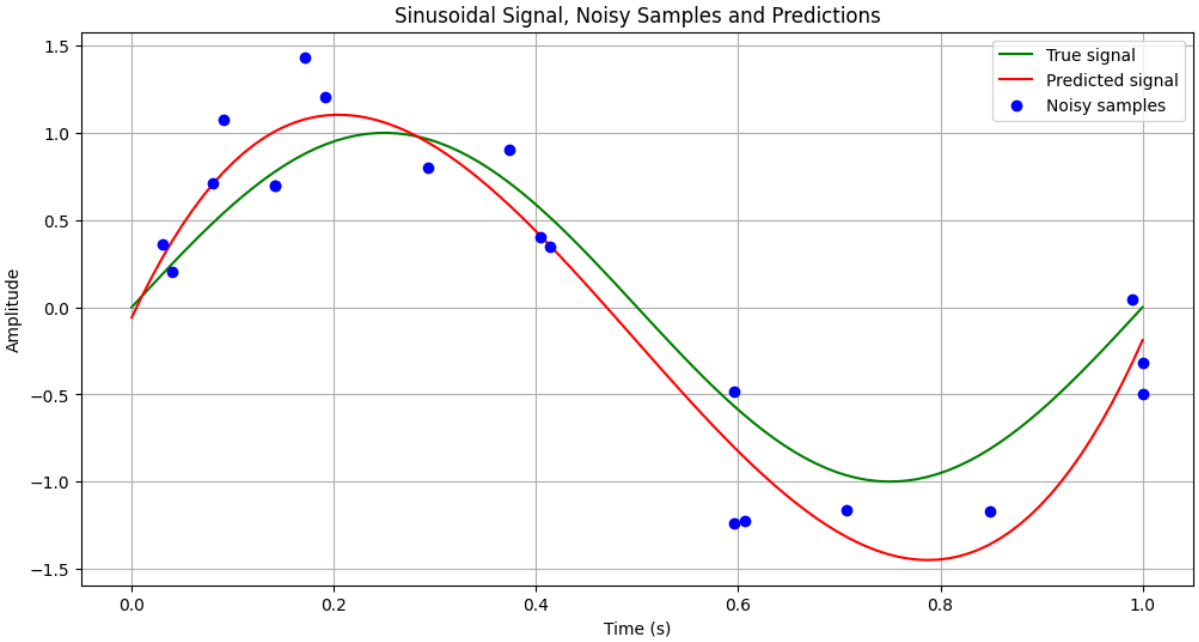


We can solve the fitting problem by selecting a parameter vector that minimizes the error, making it as small as possible.

$$w^* = \arg\min_w E(w)$$

In this specific model, since the error function is quadratic with respect to the coefficients, its derivatives with respect to these coefficients will be linear in the elements of the vector. As a result, the minimization of the error function has a unique solution, which can be obtained in closed form.

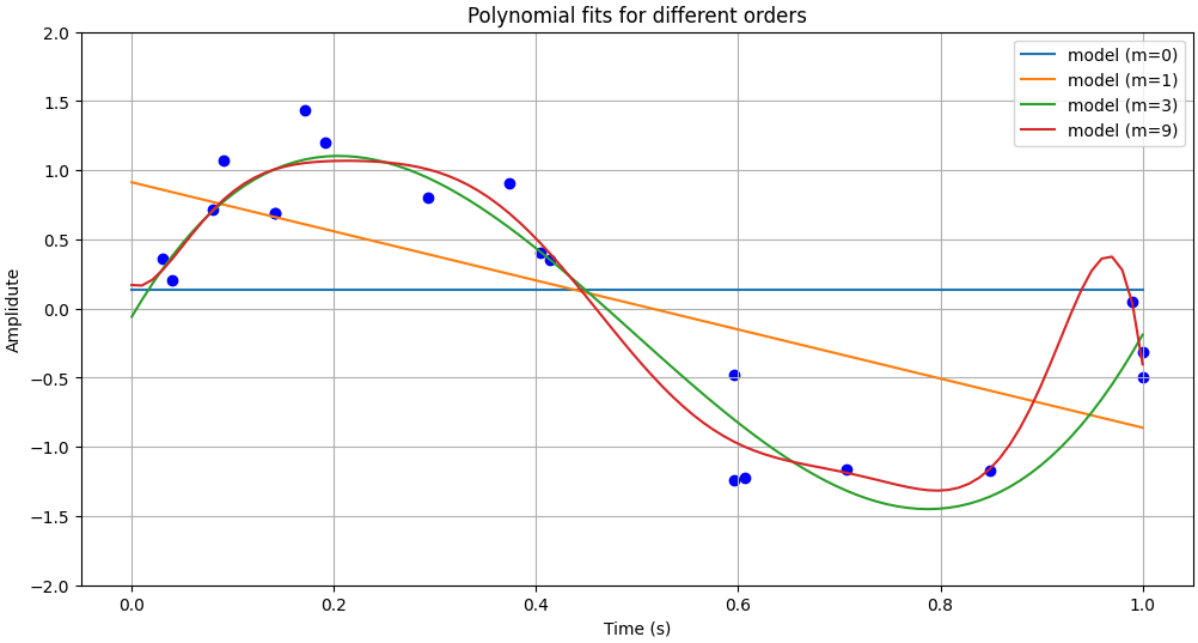
```python
# degree of polynomial to fit
m = 3

# fit the polynomial using NumPy's polyfit function
w_star = np.polyfit(x, y, m)

# crete the model
model = np.poly1d(w_star)

# make predictions
predictions = model(t)

# plot the model
plt.figure(figsize=(12, 6))
plt.plot(t, sinusoid, color="green")
plt.plot(t, predictions, color="red")
plt.scatter(x, y, color="blue", zorder=5)
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Sinusoidal Signal, Noisy Samples and Predictions")
plt.legend(["True signal", "Predicted signal", "Noisy samples"])
plt.grid(True)
plt.show()
```



The next challenge is selecting the order m of the polynomial, which is an example of an im-

portant concept known as **model selection**. We can experiment with different values of m to determine the best choice:

```
# define the polynomial orders to test
orders = [0, 1, 3, 9]

# create the plot
plt.figure(figsize=(12, 6))

# plot the dataset
plt.scatter(x, y, color='blue')

# fit and plot each polynomial model
for m in orders:
    w = np.polyfit(x, y, m)
    model = np.poly1d(w)
    plt.plot(t, model(t), label=f'model (m={m})')

# add labels and legend
plt.title('Polynomial fits for different orders')
plt.xlabel('Time (s)')
plt.ylabel('Amplidute')
plt.ylim(-2, 2)
plt.legend()
plt.grid(True)
plt.show()
```



Notice that the constant and first-order polynomials provide poor fits to the data, resulting in inaccurate representations of the underlying signal. The mid-order model offers the best fit, capturing the overall trend of the data. As we increase the polynomial order, the fit becomes perfect, as the polynomial passes through each data point exactly. However, this high-order polynomial exhibits erratic oscillations, significantly deviating from the true signal. This phenomenon, where the model fits the training data too closely at the expense of generalization to new data, is known as **overfitting**. Our goal is to achieve good generalization, meaning the ability to make accurate predictions on new, unseen data. To gain insight into how the generalization performance depends on the polynomial order, we can introduce a **test set**, a separate set of data that is **not used during training**:

```
# testset size
N_test = 15

# generate synthetic samples for the testset
sample_indices_test = np.sort(np.random.choice(len(t), size=N_test))
x_test = t[sample_indices_test]
y_test = sinusoid[sample_indices_test] + np.random.normal(0, noise_std, size=N_test)
```

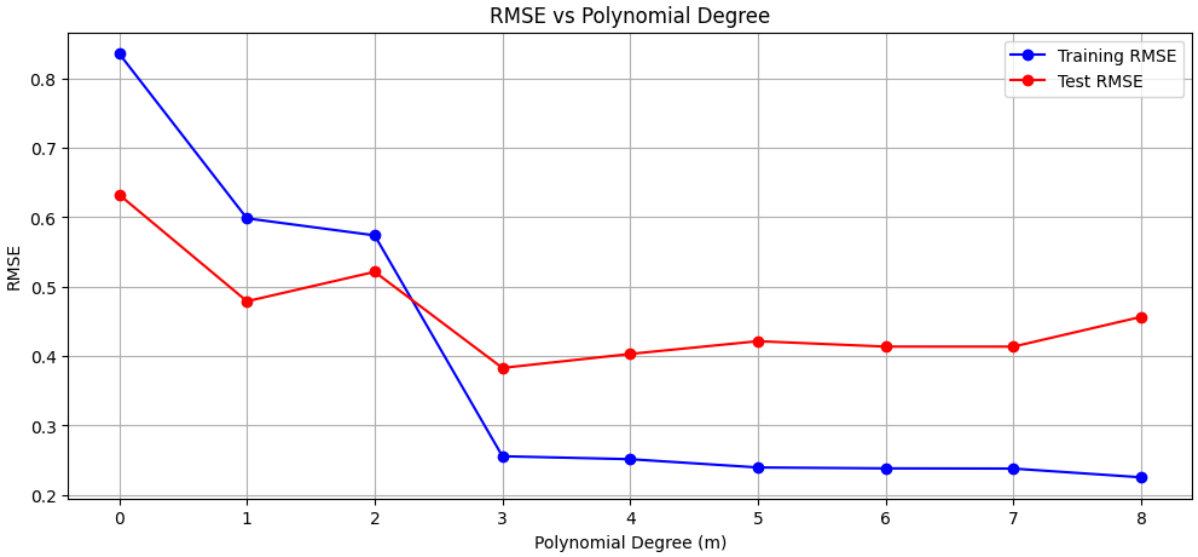
For each polynomial degree m, we can compute the error for both the training set and the test set. Instead of using the sum of squared residuals, it is often more useful to evaluate the error using the **root-mean-square (RMS) error**, which is defined as:

$$E_{\text{RMS}} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(\hat{y}(x_n, w) - y_n^2)}$$

```
# function to calculate RMSE
def rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred)**2))
```

The division by N normalizes the error, allowing us to compare datasets of different sizes on an equal basis. The square root ensures that the error is measured on the same scale as the target variable, preserving the units of the original data.

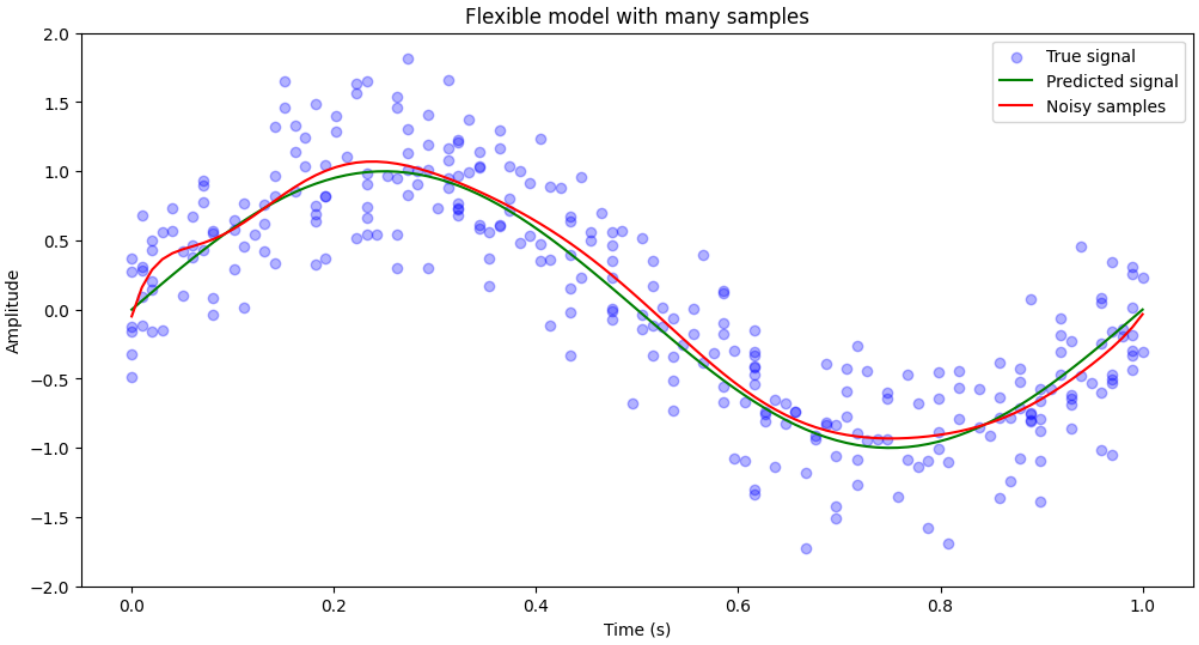
```
# create lists to store RMSE values for training and test sets
train_rmse = []
test_rmse = []

# fit several polynomial models and calculate RMSE
m_max = 9
for m in range(m_max):

    # fit the polynomial
    w = np.polyfit(x, y, m)
    model = np.poly1d(w)

    # predict on training and test sets
    y_pred = model(x)
    y_test_pred = model(x_test)

    # calculate RMSE for both training and test sets
    train_rmse.append(rmse(y, y_pred))
    test_rmse.append(rmse(y_test, y_test_pred))

# plot RMSE for each degree
plt.figure(figsize=(12, 5))
plt.plot(range(m_max), train_rmse, label='Training RMSE', marker='o', color='blue')
plt.plot(range(m_max), test_rmse, label='Test RMSE', marker='o', color='red')
plt.xlabel('Polynomial Degree (m)')
plt.ylabel('RMSE')
plt.title('RMSE vs Polynomial Degree')
plt.legend()
plt.grid(True)
plt.show()
```



The test set error measures how well our model generalizes to new, unseen data. Notice that for small values of m, the test set error is relatively high. This is because the corresponding polynomials are **too simple and lack the flexibility to capture the oscillations in the underlying signal**. For intermediate values of m, the test set error is lower, and these models provide reasonable approximations of the generating function. However, as m increases further, the training set error decreases and eventually reaches zero, as expected. This is because higher-order polynomials have **more degrees of freedom, allowing them to perfectly fit the training data**. But, paradoxically, the test set error increases significantly, and the model begins to exhibit wild oscillations. This behavior might seem counterintuitive, given that higher-order polynomials include lower-order ones as special cases. Additionally, since the power series expansion of the underlying sinusoidal function contains terms of all orders, we might expect performance to improve monotonically as m increases. Intuitively, the issue arises because higher-degree polynomials with larger values of m become **increasingly sensitive to the random noise in the target values**. As the model gains more flexibility, **it fits not only the underlying pattern in the data but also the noise**, leading to overly complex models that fail to generalize well to new, unseen data. We can gain further insight into overfitting by examining **how the model behaves as the size of the dataset changes**:

```
# number of samples
N_big = 300

# create a big number of training samples
sample_indices_big = np.sort(np.random.choice(len(t), size=N_big))
x_big = t[sample_indices_big]
y_big = sinusoid[sample_indices_big] + np.random.normal(0, noise_std, size=N_big)
```

```
# fit an high-degree polynomial model
m_big = 12
w = np.polyfit(x_big, y_big, m_big)
model = np.poly1d(w)

# make predictions
predictions = model(t)

# plot the model predictions and the noisy samples
plt.figure(figsize=(12, 6))
plt.scatter(x_big, y_big, color="blue", alpha=0.3)
plt.plot(t, sinusoid, color="green")
plt.plot(t, predictions, color="red")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Flexible model with many samples")
plt.legend(["True signal", "Predicted signal", "Noisy samples"])
plt.ylim(-2, 2)
plt.show()
```



For a fixed model complexity, the **severity of overfitting decreases as the dataset grows larger**. In other words, with a larger dataset, we can afford to use a more complex (or more flexible) model without it overfitting. A commonly cited heuristic in classical statistics suggests that

the number of data points should be at least several times (e.g., 5 to 10 times) the number of learnable parameters in the model. However, modern machine learning has shown that it is possible to achieve excellent results even with models that have significantly more parameters than the size of the training dataset. Indeed, relying on limiting the number of parameters in a model based on the size of the training set can feel unsatisfying. A more intuitive approach would be to **select the model's complexity based on the complexity of the problem itself**. To address overfitting without restricting the number of parameters, a common strategy is **regularization**. This technique involves **adding a penalty term to the error function**, which **discourages the model's coefficients from growing too large**. By constraining the magnitude of the coefficients, the model is forced to prioritize simplicity and avoid overfitting to the noise in the training data.The simplest form of this penalty term is the sum of the squares of all the coefficients, resulting in a modified error function of the form:

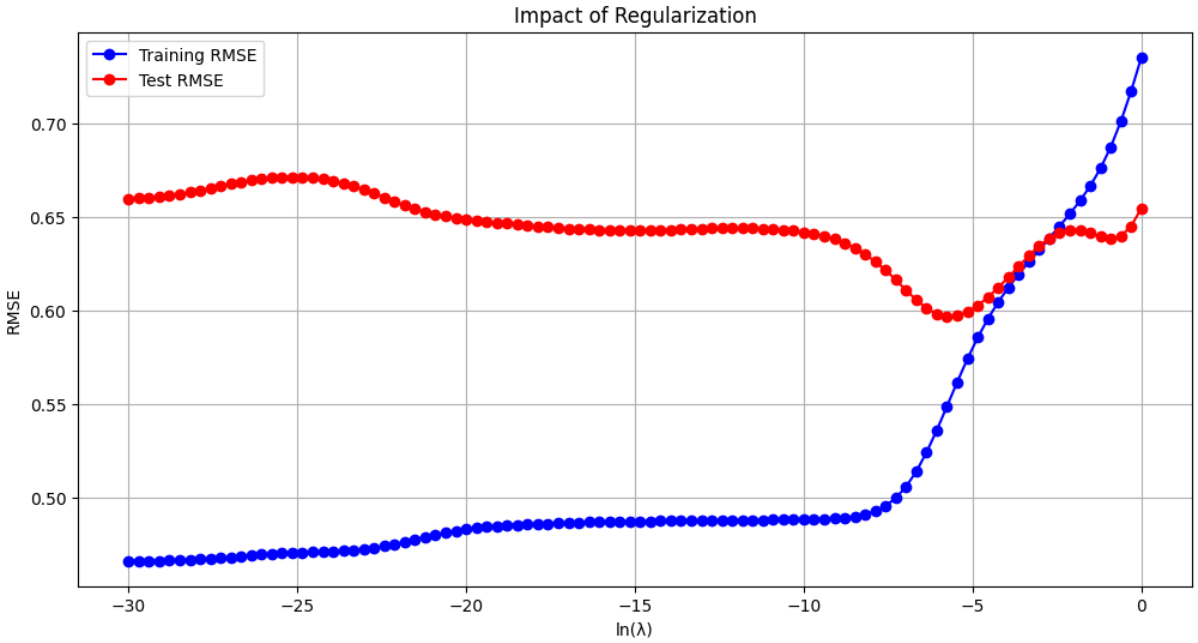
$$E(w) = \sum_{n=1}^{N} (\hat{y}(x_n, w) - y_n)^2 + \lambda \sum_{j=0}^{m} w_j^2$$

```python
# Regularized error function
def error(w, x, y, lambda_):

    # evaluate the polynomial with coefficients w
    y_hat = np.polyval(w, x)

    # compute the squared error
    mse = np.sum((y_hat - y) ** 2)

    # add the regularization term
    l2_reg = lambda_ * np.sum(w ** 2)

    return mse + l2_reg
```

The regularization coefficient determines the balance between the regularization term and the sum-of-squares error term in the objective function. This approach, known as **Ridge regularization**, is a simple yet powerful method to improve the generalization performance of machine learning models. Widely used in practice, it also serves as the foundation for more advanced regularization techniques. To observe the effect of regularization, we fit a high-degree polynomial to the data while varying the regularization parameter. By analyzing how the RMSE changes with different values of the parameter, we can understand the trade-off between model complexity and generalization. We use the logarithm of the regularization parameter to visualize these changes, as small variations in its value can result in significant shifts in the model's behavior. The l**ogarithmic scale provides better interpretability**:

```
# NumPy polyfit cannot be used to fit regularized models.
# so we will use the Ridge implementation from scikit-learn
from sklearn.linear_model import Ridge
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

# reshape x and x_test to fit scikit-learn's requirements
x = x.reshape(-1, 1)
x_test = x_test.reshape(-1, 1)

# define polynomial high degree
degree = 9

# generate ln(lambda) values
ln_lambda_min = -30
ln_lambda_max = 0
points = 100
ln_lambdas = np.linspace(ln_lambda_min, ln_lambda_max, points)
lambdas = np.exp(ln_lambdas)

# create lists to store RMSE values for training and test sets
train_rmse = []
test_rmse = []

# fit models for each lambda
for lambda_ in lambdas:

    # create a pipeline for the ridge regularization
    model = Pipeline([
        ("poly_features", PolynomialFeatures(degree=degree, include_bias=False)),
        ("ridge_regression", Ridge(alpha=lambda_))
    ])

    # fit the model on the training data
    model.fit(x, y)

    # predict on training and test sets
    y_pred = model.predict(x)
    y_test_pred = model.predict(x_test)

    # calculate RMSE for both training and test sets
    train_rmse.append(np.sqrt(rmse(y, y_pred)))
    test_rmse.append(np.sqrt(rmse(y_test, y_test_pred)))
```

```
# Plot RMSE for each regularization value
plt.figure(figsize=(12, 6))
plt.plot(ln_lambdas, train_rmse, label='Training RMSE', marker='o', color='blue')
plt.plot(ln_lambdas, test_rmse, label='Test RMSE', marker='o', color='red')
plt.xlabel('ln(λ)')
plt.ylabel('RMSE')
plt.title('Impact of Regularization')
plt.legend()
plt.grid(True)
plt.show()
```

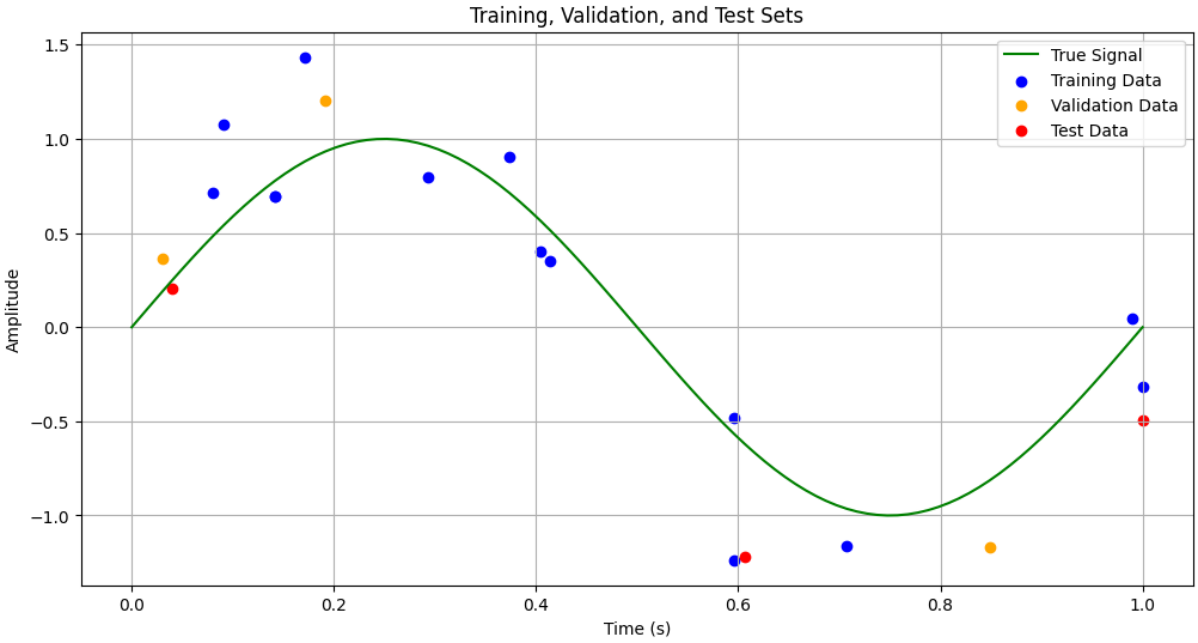


As we increase the regularization parameter, the model becomes simpler, and the RMSE on the test set decreases. This is because the penalty term discourages the model from fitting the noise in the training data, leading to better generalization performance. However, **if we set the regularization parameter too high, the model becomes too simple and fails to capture the underlying pattern in the data**. This results in a higher RMSE on the test set, as the model is unable to accurately predict the target values.

The regularization parameter is an exmaple of **hyperparameter**, whose values are fixed during the minimization of the error function to determine the model parameters. We cannot determine parameters and hypeparameters values jointly by minimizing the error function, since this will lead to a zero value for the regularization and an over-fitted model with small or zero training error Similarly, the order m of the polynomial is another hyperparameter of the model,

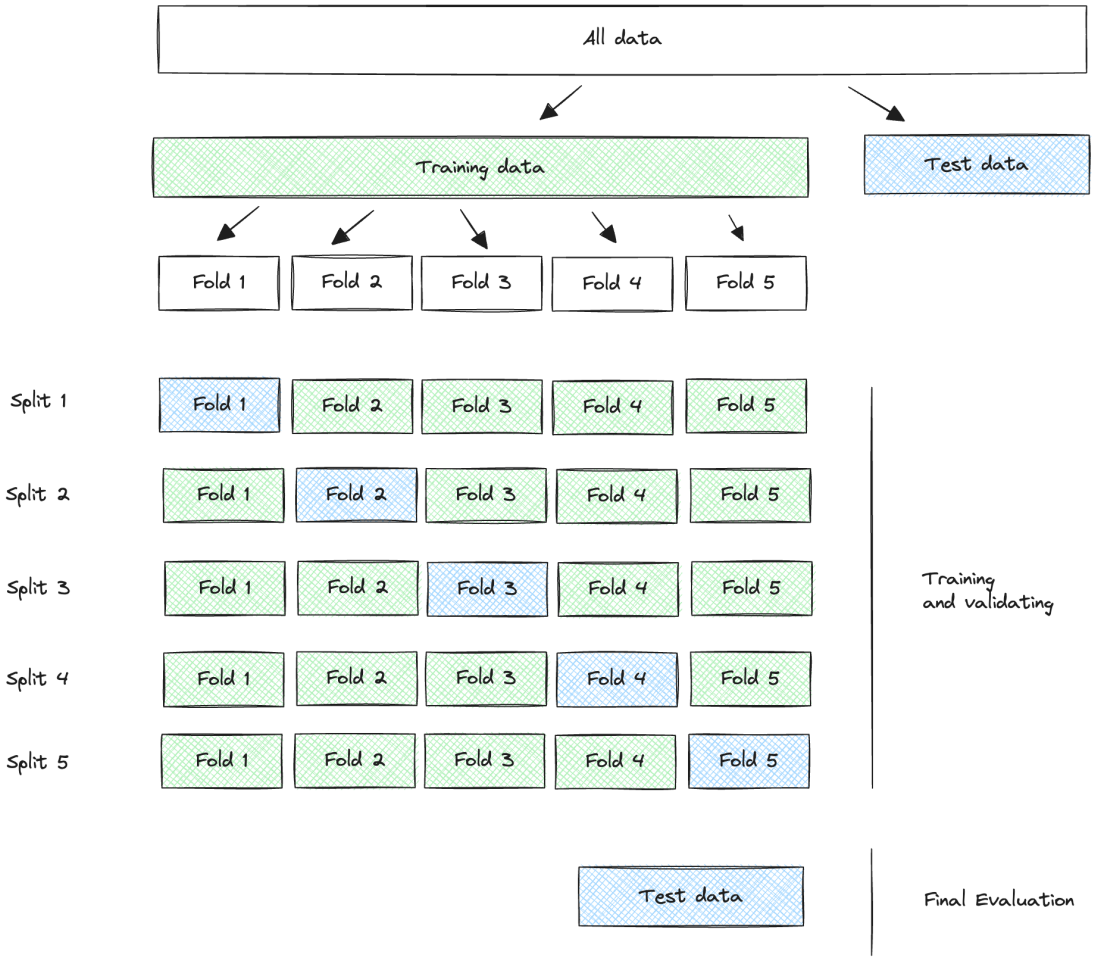
and optimizing the training set error with respect to m will lead to large value and an associated overfitting. We therefore need to find a different way to determine suitable values for hyperparameters. The results above suggest a simple way of achieving this, namely by taking the available data and partitioning it into a **training set**, used to determine the parameteres, and a separate **validation set**. We then select the hypermarameters having the lowest error on the validation set. If the model design is iterated many times using a data set of limited size, then some overfitting to the validation data can occur, and so it may be necessary to keep aside a third **test set** on which the performance of the selected model can finally be evaluated.

```
# split data into training, validation, and test sets (e.g., 70%, 15%, 15%)
train_ratio, val_ratio, test_ratio = 0.7, 0.15, 0.15
train_size = int(train_ratio * N)
val_size = int(val_ratio * N)

# shuffle indices for splitting
shuffled_indices = np.random.permutation(N)
train_indices = shuffled_indices[:train_size]
val_indices = shuffled_indices[train_size:train_size + val_size]
test_indices = shuffled_indices[train_size + val_size:]

# create the splits
x_train, y_train = x[train_indices], y[train_indices]
x_val, y_val = x[val_indices], y[val_indices]
x_test, y_test = x[test_indices], y[test_indices]

# plot the training, validation, and test sets
plt.figure(figsize=(12, 6))
plt.plot(t, sinusoid, color="green", label="True Signal")
plt.scatter(x_train, y_train, color="blue", label="Training Data")
plt.scatter(x_val, y_val, color="orange", label="Validation Data")
plt.scatter(x_test, y_test, color="red", label="Test Data")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Training, Validation, and Test Sets")
plt.grid(True)
plt.legend()
plt.show()

# print dataset sizes
print(f"Training set size: {len(x_train)}")
print(f"Validation set size: {len(x_val)}")
print(f"Test set size: {len(x_test)}")
```




```
Training set size: 14
Validation set size: 3
Test set size: 3
```

However, for some applications, **the supply of data for training and testing will be limited**. To build a good model, we should use as much of the available data as possible for training. However, if the validation set is too small, it will give a relatively noisy estimate of predictive performance. One solution to this dilemma is to use **cross-validation**:



This involves taking the available data and partitioning it into S groups of equal size (fold). Then S−1 folds are used to train a model wich is then evaluated on the remaining fold. This procedure is then repeated for all possible choices for the **held-out fold** and the performance scores from the runs are averaged. We can implement this idea in Python, first of all we create the folds:

```
# number of folds
k = 5

# manually create k fold indices
n = len(x)
indices = np.arange(n)
np.random.shuffle(indices)
folds = np.array_split(indices, k)
```

For each polynomial degree, we train the model on the full training set to record the training

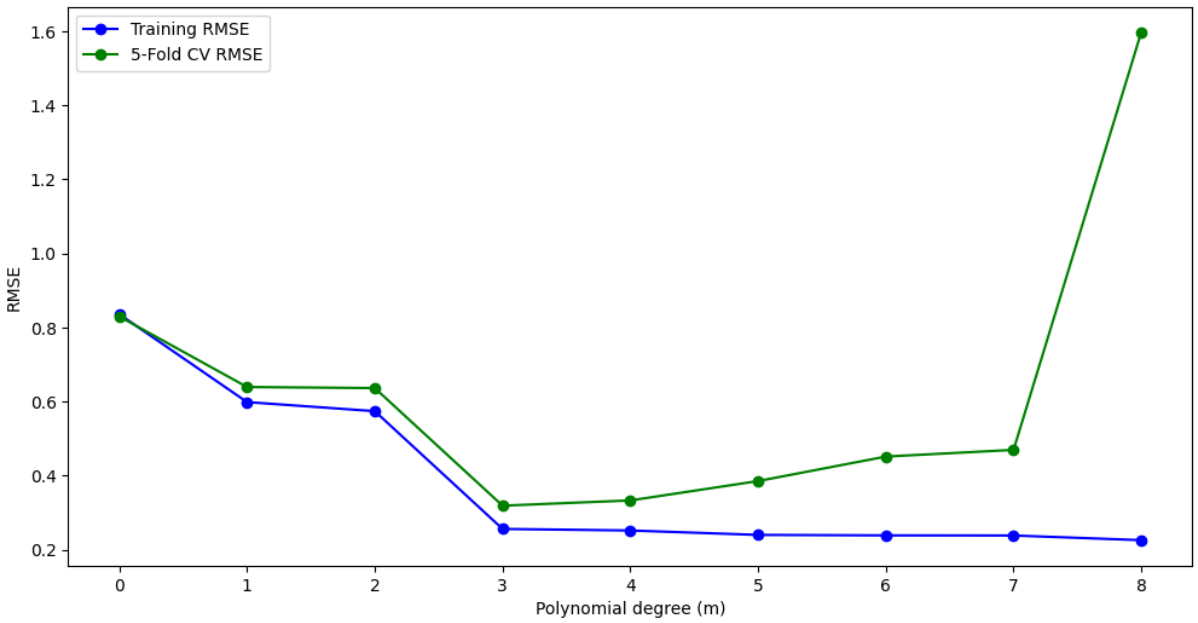
error, and then run k-fold cross-validation to obtain a more reliable estimate of the generalisation error by averaging the RMSE across all validation folds:

```
# store mean CV RMSE for each polynomial degree
train_rmse_cv = []
cv_rmse       = []

for m in range(m_max):

    # recompute train and test RMSE for this degree
    w = np.polyfit(x.ravel(), y.ravel(), m)
    model_poly = np.poly1d(w)
    train_rmse_cv.append(rmse(y.ravel(), model_poly(x.ravel())))

    # cross-validation
    fold_rmse = []
    for i in range(k):
        # validation fold is fold i, training is all others
        val_index   = folds[i]
        train_index = np.concatenate([folds[j] for j in range(k) if j ≠ i])

        x_train_cv, x_val_cv = x[train_index].ravel(), x[val_index].ravel()
        y_train_cv, y_val_cv = y[train_index].ravel(), y[val_index].ravel()

        # fit polynomial on training fold
        w_cv = np.polyfit(x_train_cv, y_train_cv, m)
        model_cv = np.poly1d(w_cv)

        # evaluate on validation fold
        fold_rmse.append(rmse(y_val_cv, model_cv(x_val_cv)))

    cv_rmse.append(np.mean(fold_rmse))
```

We then plot the training RMSE and the cross-validation RMSE as a function of the polynomial degree, to visually identify the model complexity that best balances bias and variance.

```
# plot CV RMSE alongside training and test RMSE
plt.figure(figsize=(12, 6))
plt.plot(range(m_max), train_rmse_cv, label='Training RMSE',     marker='o',
 ↪  color='blue')
plt.plot(range(m_max), cv_rmse,       label=f'{k}-Fold CV RMSE', marker='o',
 ↪  color='green')
```

```
plt.xlabel('Polynomial degree (m)')
plt.ylabel('RMSE')
plt.legend()
plt.show()
```



Finally, we select the polynomial degree that minimises the cross-validation RMSE as our best model:

```
best_m = np.argmin(cv_rmse)
print(f"Best polynomial degree selected by CV: {best_m}")
```

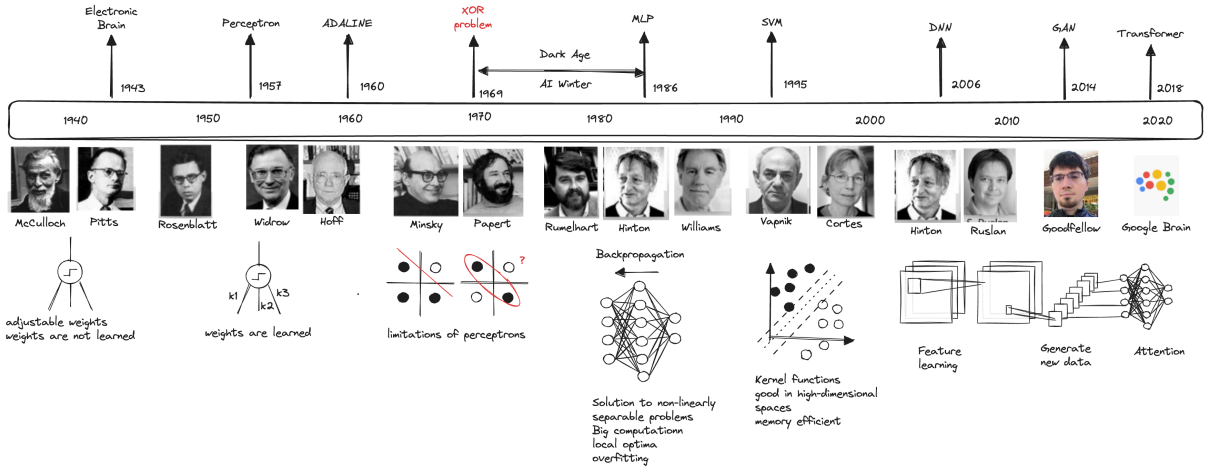
```
Best polynomial degree selected by CV: 3
```

The main drawback of cross-validation is that **the number of training runs that must be performed is increased by a factor of S**, and this can be problematic for models in which the training is itself computationally expensive. A further problem is that **we might have multiple hyperparameters** for a single model. Exploring combinations of settings for such hyperparameters could, in the worst case, require a number of training runs that is exponential in the number of hyperparameters. The state of the art in modern machine learning involves extremely large models, trained on commensurately large data sets. Consequently, there is limited scope for exploration of hyperparameter settings, and heavy reliance is placed on experience obtained with smaller models and on heuristics.

This simple example highlights several key concepts of machine learning. However, real-world applications differ significantly in several ways. **Training datasets are often many orders of magnitude larger**, and **the number of input variables can be immense**, sometimes in the millions, as in image analysis. Additionally, there may be **multiple output variables to predict**. The relationship between inputs and outputs is **modeled using complex functions** with **a large number of parameters**(e.g., neural networks), which can reach hundreds of billions of parameters. The associated **error function is highly nonlinear and cannot be minimized using a closed-form solution**. Instead, **iterative optimization techniques** are required, often necessitating specialized computational hardware and incurring significant computational costs.

1.2 A brief History of Neural Networks

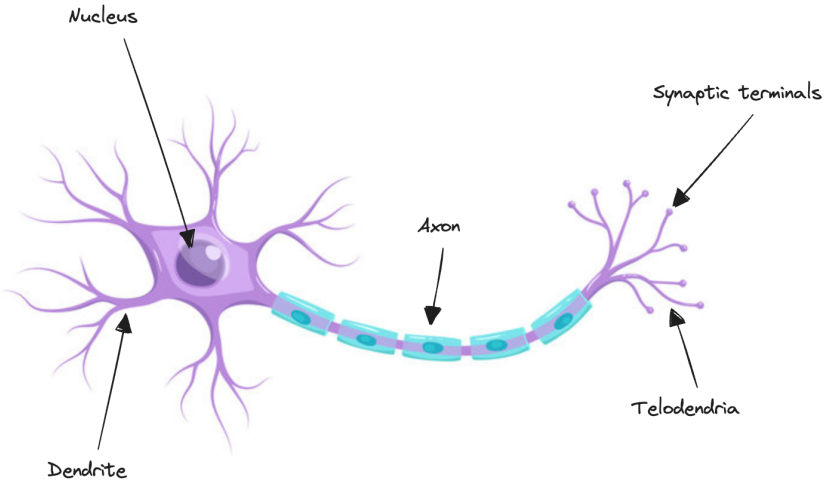
Machine learning has a long and rich history, marked by the exploration of various alternative approaches. In this discussion, we focus on the development of neural network-based methods, which form the backbone of deep learning. These methods have emerged as the most effective solution for addressing real-world machine learning challenges. The field was founded in 1956, at a workshop at Dartmouth College, where John McCarthy proposed that "every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it". Then the field has seen **two major periods of enthusiasm**, followed by **disillusionment**. The following timeline highlights key events about **neural networks**, the most popular AI technique today:



1.2.1 The first ideas: 1943-1956

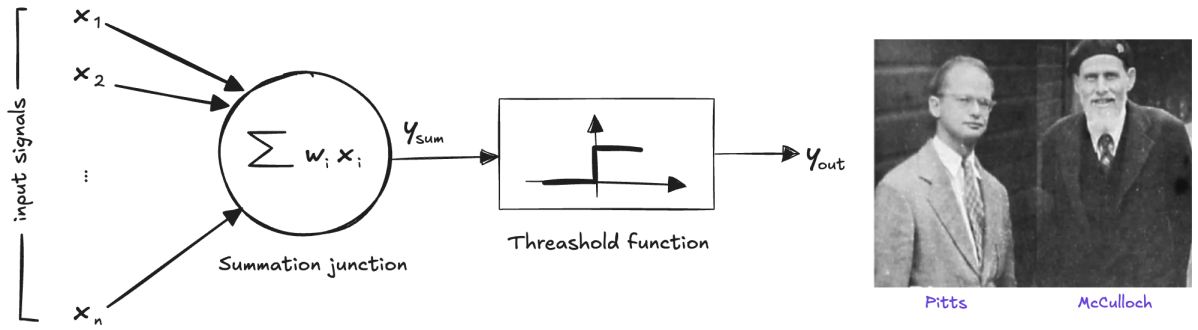
The first work recognized as AI was done by **Warren McCulloch** and **Walter Pitts** (1943). Early works were largely inspired by the theories of information processing in the brains of humans

and other mammals. The basic processing units in the brain are electrically active cells called neurons:



When a neuron "fires", it sends an electrical impulse down the axon where it reaches junctions, called synapses, which form connections with other neurons. Chemical signals called neurotransmitters are released at the synapses, and these can stimulate, or inhibit, the firing of subsequent neurons. If a particular neuron receives sufficient stimulation from the firing of other neurons then it too can be induced to fire. However, some synapses have a negative, or inhibitory, effect whereby the firing of the input neuron makes it less likely that the output neuron will fire. The extent to which one neuron can cause another to fire depends on the strength of the synapse, and it is changes in these strengths that represents a key mechanism whereby the brain can store information and learn from experience.

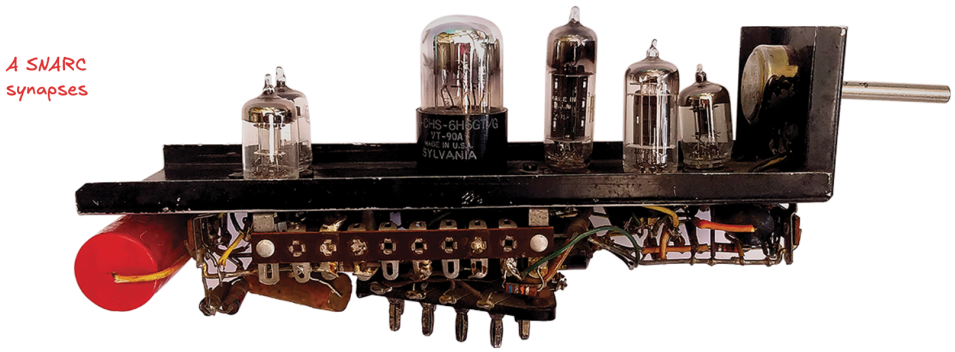
Exploiting the basic physiology and function of neurons in the brain, McCulloch and Pitts proposed a model of **artificial neuron** that captured in a very simple mathematical model, known as **thresholded logic unit** ("A Logical Calculus of the ideas Imminent in Nervous Activity Offsite Link", 1943) the way a neuron was thought to work:



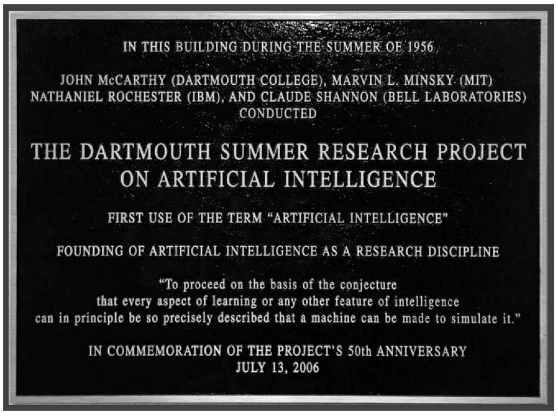
$$y_{\text{sum}} = \sum_i w_i x_i$$

$$y_{\text{out}} = f(y_{\text{sum}})$$

where vector x represent inputs corresponding to the activities of other neurons that send connections to this neuron, and vector w contains continuous variables, called **weights**, which represent the strengths of the associated synapses. The sum quantity is called the pre-activation, the nonlinear function is called the **activation function**, and the output y is called the activation. This simple mathematical formulation has formed the basis of neural network models from the 1960s up to the present day. They showed, for example, that any computable function could be computed by some network of connected neurons, and that all the logical connectives (AND, OR, NOT, etc.) could be implemented by simple network structures. In the following years, **Donald Hebb** (1949) introduced a rule for adjusting the connections between neurons, known as Hebbian learning. Meanwhile, in 1950, **Marvin Minsky** and **Dean Edmonds** constructed the first neural network computer, the Stochastic Neural Analog Reinforcement Calculator (SNARC), which utilized 3,000 vacuum tubes to simulate a network of 40 neurons:



At Princeton, Minsky explored universal computation in neural networks, though his Ph.D. committee questioned whether it should be considered mathematics. Von Neumann reportedly said, "if it isn't now, it will be someday." While there were other early AI efforts, **Alan Turing's** 1950 article "Computing Machinery and Intelligence" had the most lasting impact, introducing the Turing Test to evaluate machine intelligence. In 1955, **John McCarthy** convinced **Marvin Minsky**, **Claude Shannon**, and **Nathaniel Rochester** to organize the **landmark 1956 Dartmouth workshop**, where they proposed that "every aspect of learning or intelligence can be precisely described for a machine to simulate." This proposal was met with enthusiasm, and the field of AI was born:



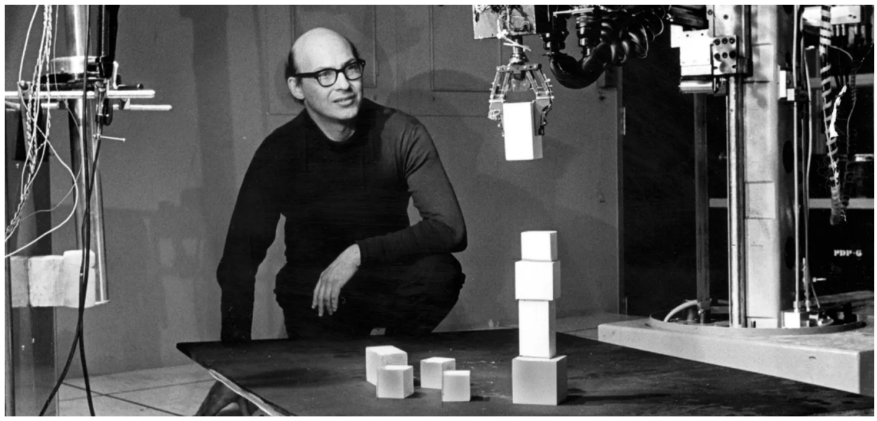
1.2.2 Early enthusiasm, great expectations (1952–1969)

In the 1950s, the intellectual establishment was skeptical that machines could perform human-like tasks, prompting researchers to demonstrate otherwise. They focused on tasks such as games, puzzles and mathematics. In 1959, Newell and Simon introduced the **General Problem Solver** (**GPS**), the first program to mimic human problem-solving, embodying the "thinking humanly" approach. The success of GPS led them to propose the **physical symbol system hypothesis** (1976), asserting that "a physical symbol system has the necessary and sufficient means for general intelligent action." In 1960, Herbert Gelernter created the **Geometry Theorem Prover**, which could prove theorems that many students of mathematics would find challenging. Of all these early works, perhaps the most influential in the long run was **Arthur Samuel's** 1956 work on **checkers**, where he used **reinforcement learning** techniques, allowing the program to learn to play better than its creator.

Arthur Samuel with the checkers machine (1959)

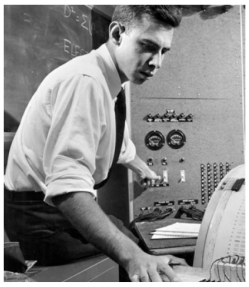
In 1958, **John McCarthy** made two significant contributions: he defined **Lisp**, a high-level programming language that became the dominant language for AI, and in his paper **Programs**

with Common Sense, he proposed systems based on knowledge and reasoning. The paper introduced the Advice Taker, a hypothetical program capable of general world knowledge and using it to derive action plans, such as driving to the airport. This marked the first step in **logical AI**. In the 1960s, several programs were developed, especially at MIT (Minsky), Stanford (McCarthy, AI lab), and Edinburgh (Minsky and McCarthy), focused on logical reasoning for tasks like question-answering, planning, geometry, integration, algebra, and other mathematical problems. These tasks were part of **microworlds**, a concept exemplified by the famous **blocks world**, where a robot hand rearranges blocks on a tabletop in specific configurations.

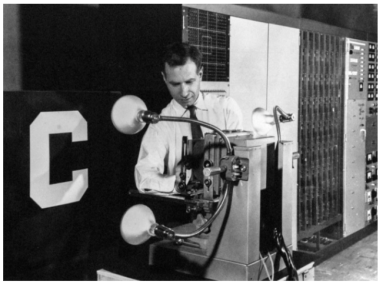


Marvin Minsky with Block Blocks Vision Robot at MIT (1968)

Early work building on the neural networks of McCulloch and Pitts also flourished. Hebb's learning methods were advanced by **the perceptron**, introduced by **Frank Rosenblatt** in his 1958 paper, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain". Rosenblatt developed a **training algorithm** for the perceptron that guaranteed the discovery of a solution if a set of weights existed to perfectly classify the training data. Beyond the algorithm, the perceptron was also implemented in hardware, most notably in the **Mark 1 perceptron**, a significant step toward building neural networks in physical systems:



Frank Rosenblatt '50, Ph.D. '56, works on the "electronic profile analyzing computer", a precursor to the perceptron

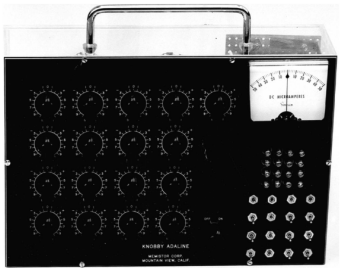


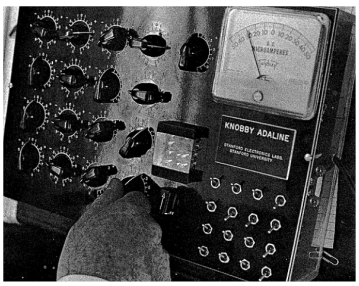
The inputs were obtained using a simple camera system in which an input scene (e.g. a printed character, was illuminated by powerful lights, and an image focused onto a 20 × 20 array of cadmium sulphide photocells, giving a primitive 400-pixel image.

The perceptron had a patch board which allowed different configurations of input features to be tried. Often these were wired up at random to demonstrate the ability of the perceptron to learn without the need for precise wiring, in contrast to a digital computer.

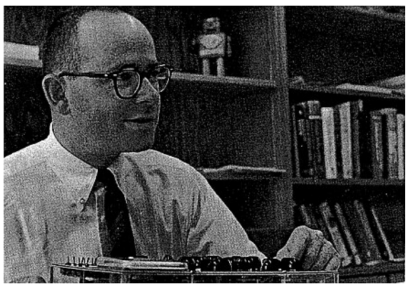
Each weight was implemented using a potentiometer, driven by an electric motor thereby allowing the value of the weight to be adjusted automatically by the learning algorithm.

The perceptron's ability to learn from data in a brain-like manner was considered remarkable. In 1959, Rosenblatt famously stated that "the perceptron is the embryo of an electronic computer that the Navy expects will be able to walk, talk, see, write, reproduce itself, and be conscious of its existence". Around the same time, **Bernard Widrow and Ted Hoff** introduced the **ADALINE (Adaptive Linear Element) system** ("Adaptive Switching Circuits"). Like the perceptron, ADALINE is a single-layer system, but it introduced the important concept of a **loss function** that operates on the continuous output of the artificial neuron before activation function, rather than relying on the discrete binary output used in the perceptron.





An ADALINE machine, with hand-adjustable weights implemented by rheostats



Bernard Widrow

1.2.3 The AI Winter (1966–1974)

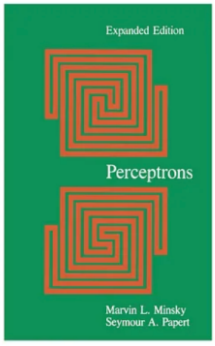
The early successes of AI research led to **overly optimistic expectations** about its progress, but by the late 1960s, the limitations of AI techniques became evident, triggering the **first AI win-**

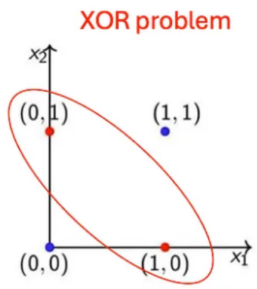
ter.

Several factors contributed to this decline. One was the **underestimation of the complexity** of many AI problems. Early problem-solving systems relied on trying different combinations of steps, which worked in microworlds with few objects and simple solutions. However, scaling up to larger problems proved more difficult, as **computational complexity theory** revealed that faster hardware and larger memories weren't enough. For example, attempts to use resolution theorem proving failed to handle problems involving more than a few dozen facts.

Another setback came with the perceptron model, which Marvin Minsky criticized in 1968. He demonstrated that a perceptron could not solve simple problems like XOR and proved it was impossible for a single-layer perceptron to do so, no matter how long it trained ("Perceptrons: An Introduction to Computational Geometry".

Minsky and Papers (1969)





Although later research disproved his conjecture about multilayer networks, this criticism dampened enthusiasm for neural networks. Additionally, the lack of effective training algorithms for multilayer networks further stalled progress. These setbacks led to a **loss of confidence** in AI, highlighted by the critical **Lighthill report** (1973), which led to a significant reduction in AI funding in the UK. The report concluded that AI had failed to achieve its goals and criticized its lack of practical applications.

1.2.4 Expert systems (1969-1986)

Early AI problem-solving methods focused on general-purpose search mechanisms, known as **weak methods**. While general, these approaches struggled to scale to complex problems. An alternative emerged in which used **domain-specific knowledge** for more efficient reasoning. The **Dendritic ALgorithm** (**DENDRAL**) **program** developed at Stanford University by **Edward Feigenbaum** exemplified this approach to infer molecular structures. This is an intracable problem, but DENDRAL used domain-specific rules to recognize common substructures in molecules, reducing the number of possible candidates significantly. This knowledge-based

approach marked a shift toward **expert systems**, which embodied expertise through specialized rules rather than general principles.

Edward Feigenbaum at Computation Center (1966)

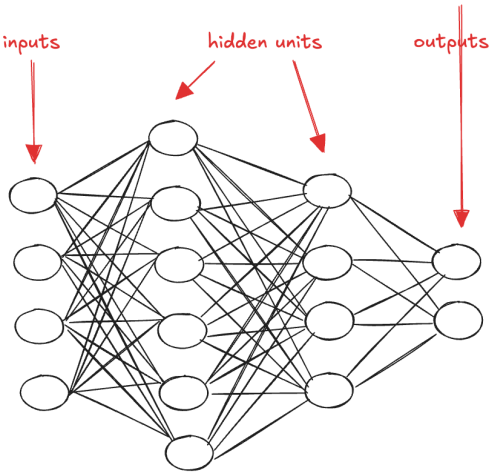
This led to the development of several expert systems, including **MYCIN** for diagnosing blood infections, which relied on approximately 450 **rules derived from expert interviews**. Expert systems gained widespread adoption and commercial success, with **R1** at Digital Equipment Corporation saving the company an estimated $40 million annually by 1986 through its use in configuring orders for new computer systems. The expanding use of AI in real-world applications drove the creation of various representation and reasoning tools, such as the popular **Prolog programming language**, based on formal logic and primarily used for symbolic reasoning.

Starting in 1981, national initiatives such as Japan's **Fifth Generation Project**, the U.S. **Microelectronics and Computer Technology Corporation** (**MCC**) and the UK's **Alvey Program** (which restored funding cut by the Lighthill Report) fueled an AI boom, growing the industry from millions in 1980 to billions by 1988. This surge led to the rise of expert systems, vision systems, robots, and specialized AI hardware and software. However, maintaining expert systems proved challenging, as their reasoning methods struggled with **uncertainty** and lacked the ability to **learn from experience**.

1.2.5 The return of neural networks (1986-1993)

By mid-80, the neural network community began to show signs of revival. The breakthrough in training networks with more than one layer of learnable parameters came through the **application of differential calculus and gradient-based optimization methods**. A crucial innovation was replacing the step function with continuous, differentiable activation functions that have a non-zero gradient, enabling smoother updates to the model's parameters. Another key ad-

vancement was the introduction of **differentiable error functions**, which measure how well a set of parameter values predicts the target outputs in the training set. These changes made it possible to compute the derivatives of the error function with respect to each parameter in the network, allowing the training process to adjust parameters efficiently. With these developments, it became feasible to train multi-layer networks, laying the foundation for the modern era of deep learning:



A neural network having three layers of parameters. Each of the hidden units and each of the output units computes a function of the form given by the single neuron which the activation function f(·) is differentiable.

Hidden units in a neural network are so named because their values are not directly observed, only input and output values are provided in the training set. Each hidden and output unit functions similarly to a neuron, processing inputs and passing information forward through the network. In 1986, **Geoffrey Hinton, David Rumelhart, and Ronald Williams** introduced **backpropagation**, a breakthrough method for training multilayer neural networks by propagating errors backward to adjust parameters efficiently. This enabled neural networks to learn **nonlinear functions**, overcoming the perceptron's key limitation.


David Rumelhart


Geoffrey Hinton


Ronald William

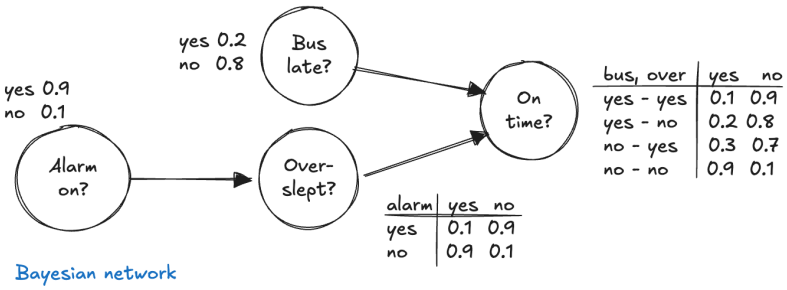
In 1989, **Kurt Hornik, Maxwell Stinchcombe, and Halbert White** proved that **multilayer feedforward networks** can approximate any continuous function with sufficient hidden units, a

result known as the **universal approximation theorem**. These **connectionist models** emerged as a challenge to symbolic AI, with Hinton famously comparing symbols to the "aether of AI"—a reference to the discredited 19th-century concept of an invisible medium for electromagnetic waves.

1.2.6 Probabilistic reasoning and neural networks decline (1993-2012)

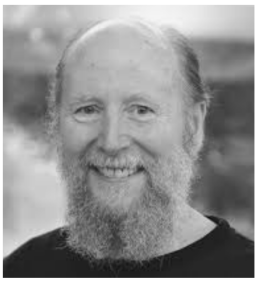
Neural networks evolved beyond neurobiological inspiration, incorporating **probability theory and statistical concepts**, leading to early successes like networks for handwritten digit recognition. However, reliance on hand-crafted feature extraction limited their scalability, prompting a shift to alternatives.Neural networks were simply **ahead of their time**, constrained by computational limits. A key milestones included **Bayesian networks** (1988), a probabilistic graphical model that represents a set of variables and their conditional dependencies using directed acyclic graphs, allowing for efficient reasoning and inference under uncertainty from **Judea Pearl**.

Judea Pearl

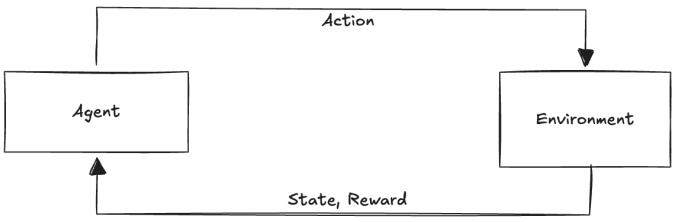


Bayesian network

We assist also the development of **Reinforcement Learning** by **Rich Sutton** (1988), which connected planning to Markov decision processes (MDPs), enabling agents to learn optimal policies through trial and error.



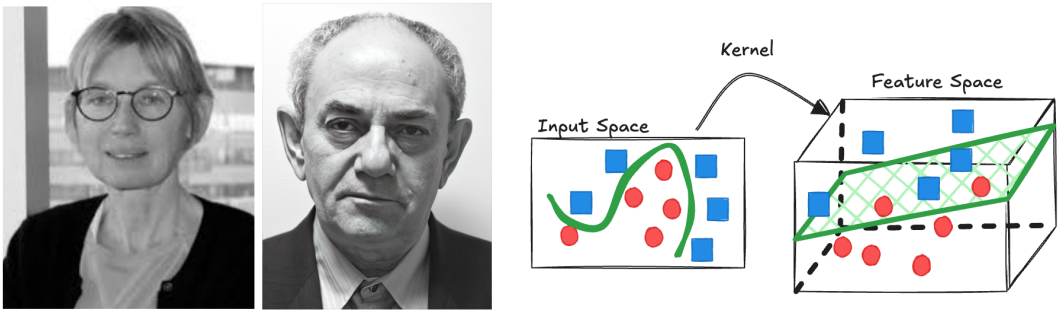
Rich Sutton



Reinforcement learning

In 1990s, an important result was the **Support Vector Machines (SVMs)**, introduced by **Corinna Cortes and Vladimir Vapnik** (1995), SVM finds the optimal hyperplane that best

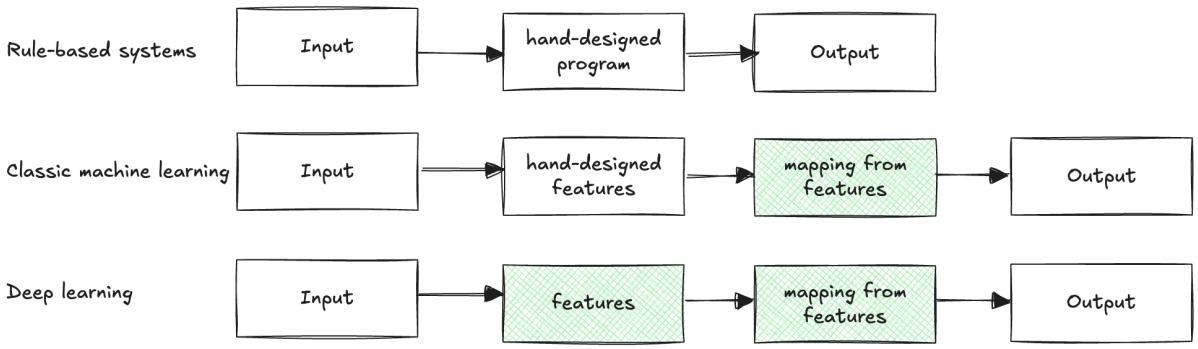
separates data into different classes, maximizing the margin between them, and uses kernel functions to transform data into higher-dimensional spaces for handling non-linearly separable problems.



This shift towards **data-driven methods** unified AI subfields, fostering breakthroughs in computer vision, robotics, and natural language processing and expanding AI's real-world impact. Early AI research isolated itself from classical computer science, believing symbolic computation rendered traditional theories obsolete. This isolation starts to be abandoned, with AI integrating machine learning with information theory, uncertain reasoning with stochastic modeling, search with optimization and control, and automated reasoning with formal methods.

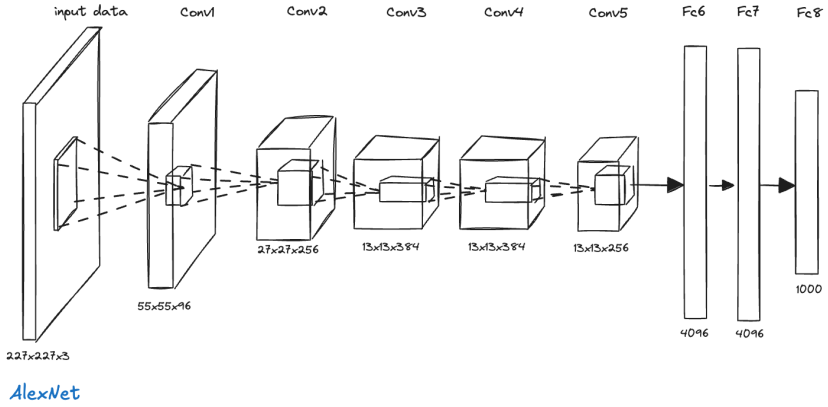
1.2.7 Big data and the deep learning revolution (2012-today)

Advances in **computing power** and the creation of the World Wide Web enabled the development of **vast datasets**, spurring the third phase of neural network evolution in the 21st century. In 2006, **Geoffrey Hinton** introduced **unsupervised pretraining** to training effectivelly deeper networks than had previously been possible, leading to a rebranding of "neural networks" as **"deep learning"**. Deep learning leverages the ability of computers to build complex concepts from simpler ones, facilitating the modeling of hierarchical representations.



Despite initial struggles, deep learning gained momentum, especially after the revolution at

the 2012 **Large Scale Visual Recognition Challenge** (**LSVRC**). This annual competition was built around **ImageNet**, a massive dataset created in 2010 containing millions of labeled images, designed to advance computer vision research. In its first two years, the top-performing models achieved error rates of 28% (2010) and 26% (2011), showing incremental progress. In 2012, **Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton** present **AlexNet** which cut the error rate nearly in half, achieving an astonishing 16%. This breakthrough was achieved by combining several innovations that would become standard in deep learning. Among these, the most critical was **the use of graphics processing units** (**GPUs**), specialized processors originally developed for rendering graphics in video games. GPUs excel at parallel computation, which aligned perfectly with the parallel nature of neural networks.



Today, performance improvements in AI often stem from scaling (training on larger datasets, using bigger models, and leveraging greater computational power). Beyond scaling, a key breakthrough was the introduction of the **Attention Mechanism**, particularly the **transformer architecture** proposed by **Vaswani et al.** in "Attention Is All You Need" famous paper. This mechanism allows models to dynamically focus on relevant input elements, effectively capturing long-range dependencies. This paradigm shift laid the foundation for **Large Language Models** (**LLMs**), which not only achieve state-of-the-art results in specialized tasks but also generalize across diverse problem domains using the same trained network. In 2024 **Geoffrey Hinton** was co-awarded the **Nobel Prize in Physics** for his contributions to the development of deep learning, marking a significant milestone in the history of AI, recognizing the transformative impact of neural networks on science and society.

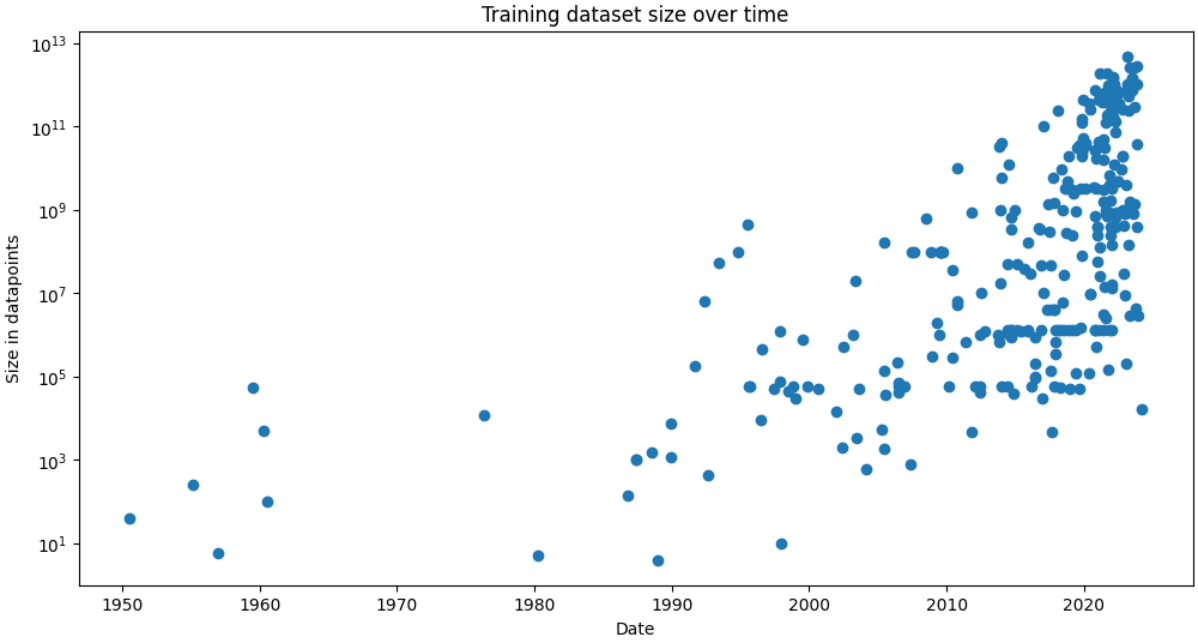
1.3 Big models and TinyML

Many of the core concepts underlying deep learning had already been established by the 1980s and 1990s, so what changed after 2010 to ignite its rapid progress? To explore this, we can refer to the **Epoch AI Dataset**, a comprehensive database of notable machine learning models and

their characteristics across domains such as language, vision, and speech. This dataset provides clear evidence of a transition after 2010. During this period, we see **the emergence of datasets that are orders of magnitude larger than those available in the preceding decade**. Machine learning can be likened to a furnace, it requires a significant amount of fuel to keep burning, and by the 2010s, we finally had that fuel: vast datasets capable of driving the performance of increasingly complex models:

```
import pandas as pd

ml_trends = pd.read_csv('data/trends/machine-learning.csv')
ml_trends.sort_values(by=['Publication date'], inplace=True)
```

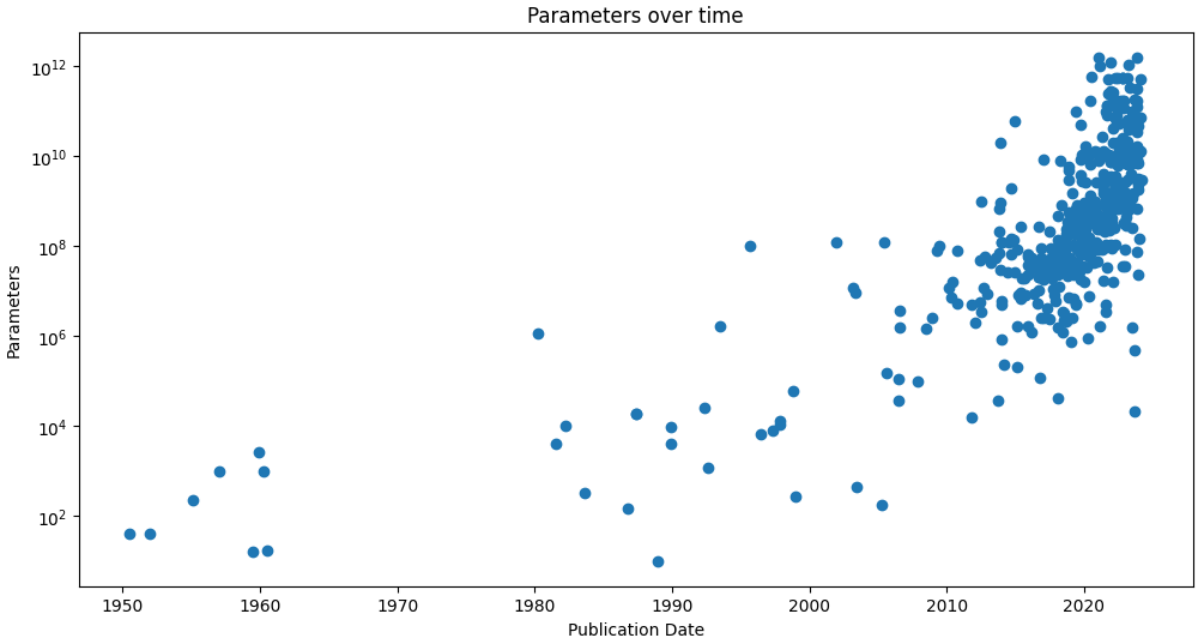
```
time = pd.to_datetime(ml_trends['Publication date'])
dataset_size = ml_trends['Training dataset size (datapoints)'];

plt.figure(figsize=(12,6))
plt.scatter(time, dataset_size)
plt.title('Training dataset size over time')
plt.yscale("log")
plt.xlabel('Date')
plt.ylabel('Size in datapoints')
plt.show()
```



As more data became available, models began to grow larger and larger, with **the number of**

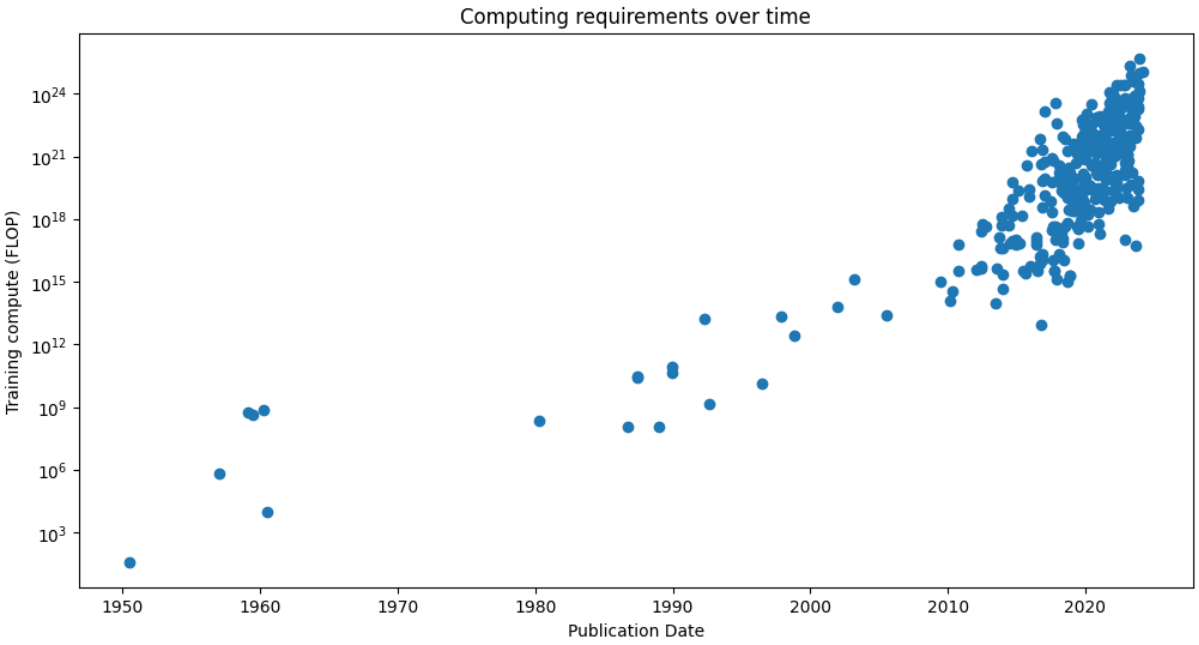
parameters increasing exponentially. Between the 1950s and 2018, model size gradually expanded by seven orders of magnitude. However, since 2018, this growth has accelerated significantly, particularly for language models, with model size increasing by an additional four orders of magnitude in just the past few years.

```
parameters = ml_trends['Parameters'];

plt.figure(figsize=(12,6))
plt.scatter(time, parameters)
plt.title('Parameters over time')
plt.yscale("log")
plt.xlabel('Publication Date')
plt.ylabel('Parameters')
plt.show()
```



To keep up with the increasing complexity of models, **the computational power required has grown at an equally remarkable pace**. Looking back at the history of machine learning, from 1960 to around 2010, there was a steady, linear growth in computational requirements. However, since 2012, when the rise of machine learning was sparked by AlexNet and the adoption of GPUs, there has been a dramatic acceleration in the computational demands necessary to support modern machine learning systems:

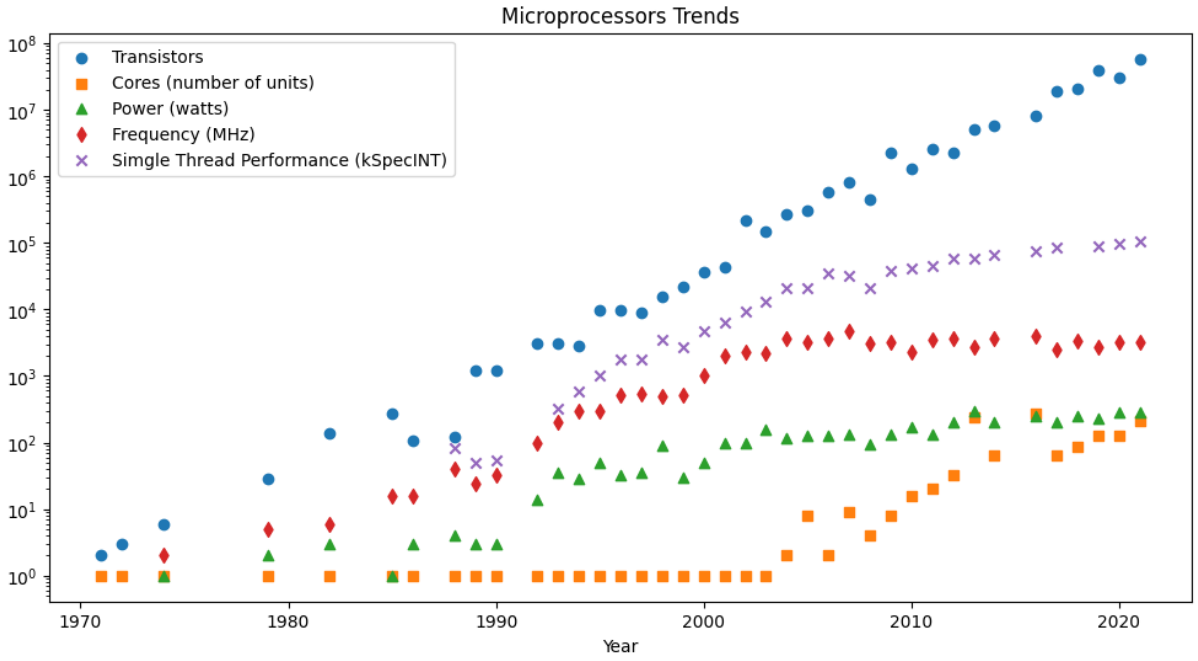
```
computing_requirements = ml_trends['Training compute (FLOP)'];

plt.figure(figsize=(12,6))
plt.scatter(time, computing_requirements)
plt.title('Computing requirements over time')
plt.yscale("log")
plt.xlabel('Publication Date')
plt.ylabel('Training compute (FLOP)')
plt.show()
```



During the same period, we also witnessed **the rise of massively parallel computing platforms like GPUs and TPUs**. It turns out that neural networks consist primarily of floating-point calculations, which can be efficiently executed in parallel. The shift from CPU-based to GPU-based training has resulted in dramatic speed-ups in model training. This transition is clearly reflected in the **"50 Years of Microprocessor Trend Data" dataset**, which tracks the evolution of various microprocessor features, such as the number of transistors, frequency, power, performance, and number of cores, over the past 50 years (from 1975 to today). Notably, since 2010, the ability to leverage parallelism has helped overcome the plateau in single-core performance, leading to significant gains in computational capacity:

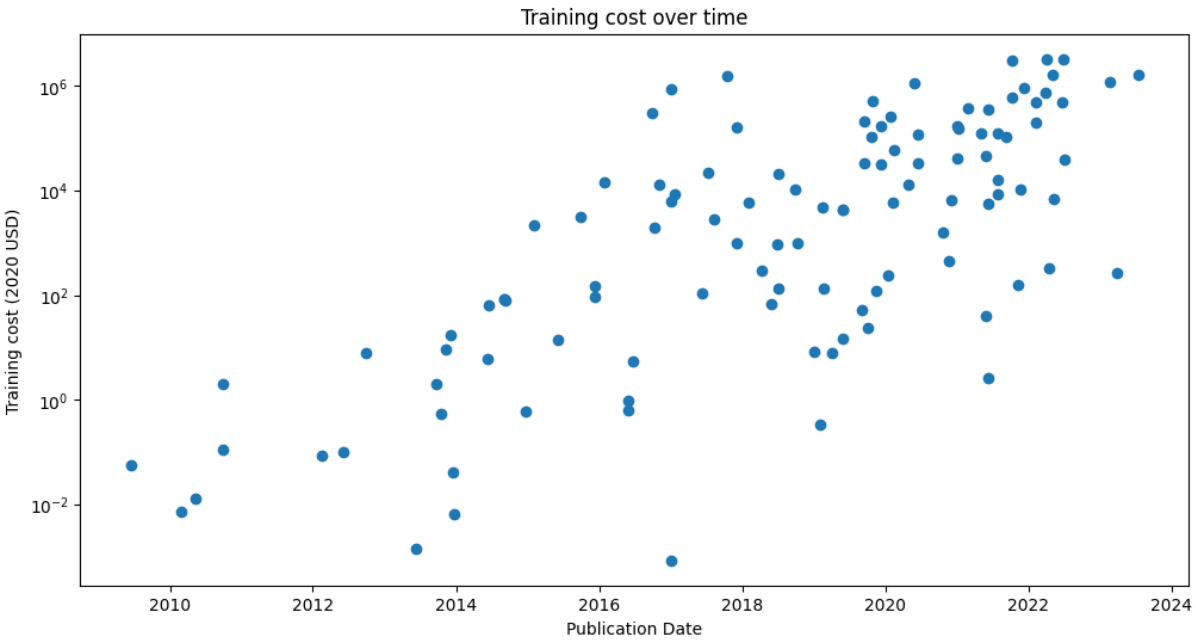
```
micro_trends = pd.read_csv('data/trends/micro.csv', sep=';')

year = micro_trends['Year']
```

```
transistors = micro_trends['Transistors (thousands)'];
cores = micro_trends['Cores (number of unit)'];
power = micro_trends['Power (watts)'];
frequency = micro_trends['Frequency (MHz)'];
performance = micro_trends['Single Thread Performance (kSpecINT)'];

plt.figure(figsize=(12,6))
plt.scatter(year, transistors, label='Transistors', marker='o')
plt.scatter(year, cores, label='Cores (number of units)', marker='s')
plt.scatter(year, power, label='Power (watts)', marker='^')
plt.scatter(year, frequency, label='Frequency (MHz)', marker='d')
plt.scatter(year, performance, label='Simgle Thread Performance (kSpecINT)',
 ↪  marker='x')
plt.title('Microprocessors Trends')
plt.yscale("log")
plt.xlabel('Year')
plt.legend()
plt.show()
```



However, **the computational cost of training large models has become increasingly prohibitive**. For instance, the estimated cost to train GPT-3 was around $4.6 million, highlighting the significant financial resources required for training state-of-the-art models:

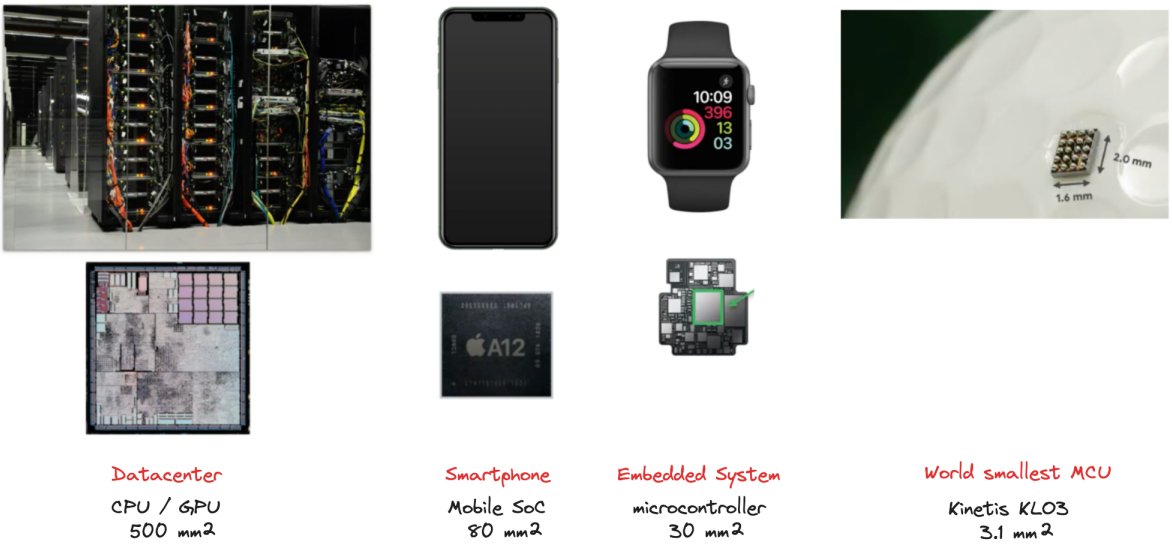
```
cost = ml_trends['Training compute cost (2020 USD)'];

plt.figure(figsize=(12,6))
plt.scatter(time, cost)
plt.title('Training cost over time')
plt.yscale('log')
plt.xlabel('Publication Date')
plt.ylabel('Training cost (2020 USD)')
plt.show()
```



These high costs can only be sustained by large organizations with substantial computational resources. Companies like Google, for example, have developed custom processors specifically designed for machine learning tasks. Just as Intel and Apple produce processors for general-purpose computing, Google has created TPUs tailored to efficiently handle ML workloads. These TPUs are often packed into massive racks, which are then housed in **large data centers**. These data centers consume such vast amounts of power that they are strategically located near power sources and water supplies for cooling, as exemplified by this Google data center in the Netherlands:



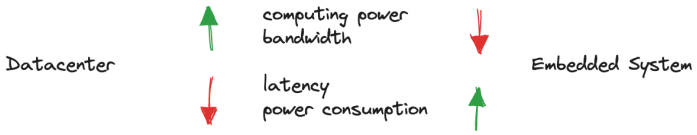
This creates **a significant barrier to entry for smaller teams or individuals without access to such vast resources**. Consequently, there is an increasing demand for more efficient algorithms and techniques that can deliver comparable performance with fewer computational resources. At the same time, there are growing efforts to democratize access to high-performance computing for developers and researchers, ensuring that these capabilities are available to a wider audience. On the other hand, there is an intriguing trend in **the rise of embedded systems equipped with small microcontrollers**. An embedded system is a compact computing device designed to operate with **extremely low power**, typically in the milliwatt range. But what defines ”big” and ”small” in this context? To better understand, let's examine the physical area occupied by processors in different systems:



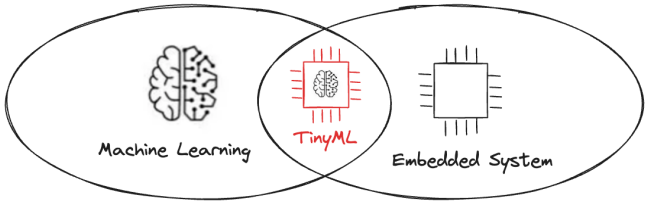
Some devices can operate for days, weeks, or even months, sometimes years, on a coin cell battery, or in some cases, without a battery at all, relying on **energy harvested from their surroundings**. The demand for microcontrollers (MCUs) is growing rapidly as we continue to develop new devices. As the volume of MCUs increases, their cost will decrease due to their widespread adoption. MCUs are designed to be affordable enough to be embedded in almost any product, from cars and coffeemakers to refrigerators, **essentially everywhere**. The holy

grail for embedded systems is **to be deployable anywhere without requiring maintenance**, such as docking or battery replacement. Devices that rely on tethered electricity face significant deployment challenges, as they are limited to locations with electrical wiring. Even in places where wiring is available, practical difficulties can arise when attempting to plug in new devices. To put things into perspective, a large GPU consumes around 300 watts, requiring a wall outlet. In contrast, the processor inside an iPhone consumes about 3 watts, 100 times smaller. When we shift to microcontrollers (MCUs), their power consumption drops to the microwatt range.

In summary, these devices are **incredibly small** (making them highly pervasive), **consume extremely low power** (enabling long-term deployment on small batteries), and are **remarkably affordable** (allowing them to be embedded virtually everywhere). The core concept of **Tiny Machine Learning (TinyML)** is **to bring machine learning models to embedded systems**, leveraging the widespread presence of microcontrollers. When comparing a large computing system, such as a data center, to a small embedded device, what are the key differences?



The goal is to shift from data centers to embedded devices, **enabling us to process data locally and in real-time** rather than sending it to the cloud for processing and then retrieving the results. This approach saves both power and time:



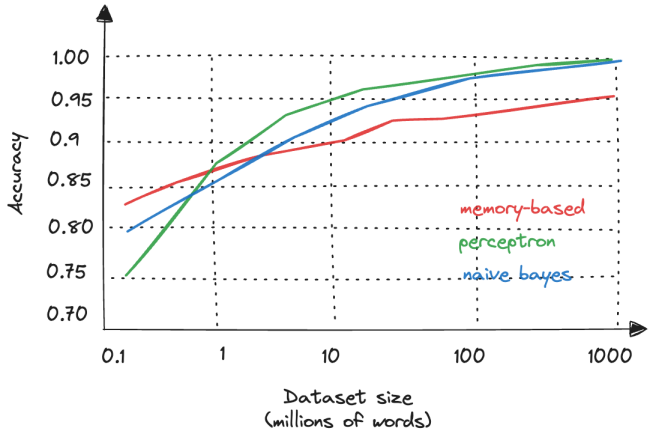
1.4 Main Challenges

Since our primary task is to select a learning algorithm and train it on data, the two potential pitfalls are a **poor algorithm** and **poor data**. Let's delve deeper into these two challenges.

1.4.1 Insufficient Quantity of Data

For a child to learn what an apple is, all it takes is to point to an apple and say "apple" (possibly repeating this procedure a few times). Now the child is able to recognize apples in all sorts of colors and shapes. Machine Learning is not quite there yet: even for very simple problems, we

typically need thousands of examples, and for complex problems (image or speech recognition) we may need millions of examples. A famous paper of 2001, M. Banko, E. Brill. "Scaling to Very Very Large Corpora for Natural Language Disambiguation", 2001 showed that very different algorithms, including simple ones, performed almost identically well on a problem of natural language processing once they were given enough data:



As the authors put it, these results suggest that we may want **to consider the trade-off between spending time and money on algorithm development versus spending it on corpus development**. The TensorFlow Datasets (TFDS) project makes it easy to load common datasets, from small ones like MNIST to huge datasets like ImageNet. The list includes image datasets, text datasets, audio, and video datasets, time series, and much more. And Know Your Data is a convenient tool to explore and understand many of the datasets provided by TFDS.

1.4.2 Nonrepresentative Dataset

In order to generalize well, it is crucial to exploit a dataset **representative of the new cases we want to generalize to**. By using a nonrepresentative dataset, we train models that are unlikely to make accurate predictions. It is crucial to use a dataset that is representative of the cases we want to generalize to. This is often harder than it sounds: if the sample is too small, we will have **sampling noise** (i.e., nonrepresentative data as a result of chance), but even very large samples can be nonrepresentative if the sampling method is flawed. This is called **sampling bias**.

1.4.3 Poor-Quality Data

Obviously, if our dataset is full of errors, outliers, and noise (e.g., due to poor-quality measurements), it will make it harder for the system to detect the underlying patterns, so our system is less likely to perform well. It is often well worth the effort to spend time **cleaning up dataset**.

As the saying goes: **garbage in, garbage out**. Our system will only be capable of learning if the dataset contains enough relevant features and not too many irrelevant ones. A critical part of the success of a project is coming up with a good set of features to train on. This process is called **feature engineering**. However, today we can often bypass this step entirely by allowing the algorithm to automatically detect good features, as we seen before introduction deep neural networks. This is called **feature learning**.

1.4.4 Generalization: overfitting and underfitting

Consider we are visiting a foreign country and the taxi driver rips we off. We might be tempted to say that all taxi drivers in that country are thieves. **Overgeneralizing** is something that humans do all too often, and unfortunately machines can fall into the same trap if we are not careful. This is called **overfitting**: it means that the model performs well on the dataset, but it does not generalize well. For example, consider to train a really flexible model (a polynomial regressor with a high degree) on the training data:

The flexible model **strongly overfits** the dataset. Even though it performs much better on the training data than the simple linear model, would we trust its predictions? Complex models can detect subtle patterns in the data, but if the dataset is noisy, or if it is too small, then the model is likely to detect patterns in the noise itself. Obviously these patterns will not generalize to new instances. Some possible solutions are: **simplify the model** by selecting one with fewer parameters by reducing the number of attributes in the dataset, or by constraining the model (called **regularization**); **gather more data**; **reduce the noise** in the dataset (e.g., fix data errors and remove outliers). For example, the linear model we defined earlier has two parameters, height and slope. This gives the learning algorithm **two degrees of freedom** to adapt the model to the dataset. If we forced the slope=0, the algorithm would have only one degree of freedom and would have a much harder time fitting the data properly: all it could do is move the line up or down to get as close as possible to the dataset instances, so it would end up around the mean. If we allow the algorithm to modify the slope but we force it to keep it small, then the learning algorithm will effectively have somewhere in between one and two degrees of freedom. It will produce a model that's simpler than one with two degrees of freedom, but more complex than one with just one. We need to find the **right balance between fitting the dataset perfectly and keeping the model simple enough to ensure that it will generalize well**. Let's try with the **Ridge Regression** algorithm, a regularized version of Linear Regression, on the partial data:

Regularization forced the model to have a smaller slope: this model does not fit the dataset as well as the first model, but it actually generalizes better to new examples that it did not see during training. The amount of regularization to apply during learning can be controlled by an **hyperparameter** (a parameter of a learning algorithm, not of the model). As such, it is not

affected by the learning algorithm itself, it must be set prior to training and remains constant during training. If we set the regularization hyperparameter to a very large value, we will get an almost flat model (a slope close to zero); the learning algorithm will almost certainly not overfit the dataset, but it will be less likely to find a good solution. Tuning hyperparameters is an important part of building a learning system.

Underfitting is the opposite: it occurs when the model is too simple to learn the underlying structure of the data. For example, the previous linear model is prone to underfit; reality is just more complex than the model, so its predictions are bound to be inaccurate, even on the examples. Here are the main options for fixing this problem: **select a more powerful model**, with more parameters; **feed better features to the learning algorithm** (feature engineering); **reduce the constraints on the model** (e.g., reduce the regularization hyperparameter).

1.4.5 Bias-Variance trade-off

The only way to know how well a model will generalize to new cases is to **actually try it out on new cases**. One way to do that is to put the model in production and monitor how well it performs. This works well, but if the model is horribly bad, the users will complain: not the best idea. A better option is to split data into two sets: the **training set** and the **test set**. We train the model using the training set, and we test it using the test set. The error rate on new cases is called the **generalization error** (how well the model will perform on instances it has never seen before) and evaluating the model on the test set provides an estimate of this error. It is common to use 80% of the data for training and hold out 20% for testing. However, this depends on the size of the dataset. In general, a large gap between training and test error means that the model is overfitting the data.

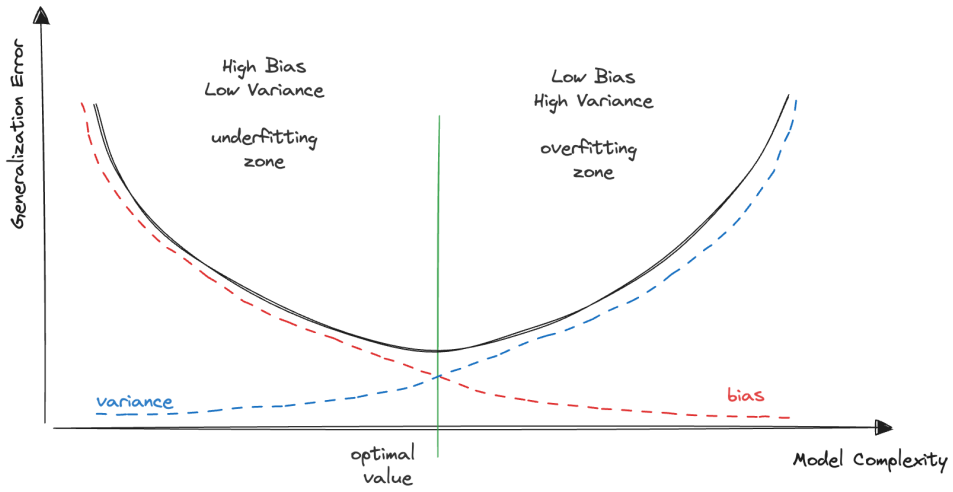
In considering the generalization error, we need to introduce two important concept: **bias** and **variance**.

Bias is a systematic error in the model predictions, that occurs **when the model makes assumptions about the relationship between the input and output variables** that do not accurately reflect the underlying reality. For example, a model that only uses a limited number of features may be biased because it does not capture all the important information in the data. Similarly, an algorithm can be inherently biased, such as a linear model, because of its underlying assumptions about the relationship between the input and output variables.

Variance is the error that is introduced by the model **s**ensitivity to small fluctuations in the training data**. It occurs when the model is too complex and fits the training data too closely, making it difficult to generalize to new, unseen data. For example, a model that uses many features or has a high degree of freedom may be more prone to overfitting, resulting in high

variance. Similarly, an algorithm that is flexible, such as a decision tree, may produce models with high variance due to its ability to fit the data very closely.

It is important to strike a **balance between bias and variance**. A model with high bias is likely to underperform, while a model with high variance is likely to overperform. Therefore, finding the right trade-off between bias and variance is crucial in ensuring high-quality models.



Suppose we are hesitating between several models, how can we decide between them? One option is to train all and compare how well they generalize using the test set. Now suppose that a polynomial model show best performance, but we want to apply some regularization to avoid overfitting. The question is, how do we choose the value of the regularization hyperparameter? One option is to train the model using different values for the hyperparameter and then select the value that produces the lowest generalization error. Finally, we launch that model into production, but unfortunately it does not perform as well as expected. What just happened? The problem is that we measured the generalization error multiple times on the test set, and we adapted the model and its hyperparameter to produce the best model for that particular set. This means that the model is unlikely to perform as well on new data!

A common solution is **hold out part of the training set to evaluate several candidate models** and select the best one. The new set is called the **validation set**. We train multiple models with various hyperparameter values on the reduced training set, then we select the model that performs best on the validation set, and we train the best model on the full training set. Lastly, we evaluate this final model on the test set to get an estimate of the generalization error. This solution usually works quite well. However, if the validation set is too small, then model evaluations will be imprecise. One way to solve this problem is to perform repeated **cross-validation**, using many small validation sets. Each model is evaluated once per validation set after it is trained on the rest of the data. By averaging out all the evaluations of a model, we get a much more accurate measure of its performance.

On of the most important rule to remember is that the validation set and the test set must be as representative as possible of the data we expect to use in production!

1.4.6 No free-lunch theorem

A model is a **simplified version** of the observations. The simplification is meant to discard the superfluous details that are unlikely to generalize to new instances. To decide what data to discard and what data to keep, we must make **assumptions**. For example, a linear model makes the assumption that the data is fundamentally linear and that the distance between the instances and the straight line is just noise, which can safely be ignored.

A famous paper D. H. Wolpert. "The Lack of A Priori Distinctions Between Learning Algorithms", Neural Comput 1996 demonstrated that if you make absolutely no assumption about the data, then there is no reason to prefer one model over any other. This is called the No Free Lunch (NFL) theorem. For some datasets the best model is a linear model, while for other datasets it is a neural network. **There is no model that is a priori guaranteed to work better** (hence the name of the theorem). The only way to know for sure which model is best is to evaluate them all. Since this is not possible, **in practice we make some reasonable assumptions about the data** and evaluate only a few reasonable models.

This results is a consecuense of what the philosofer David Hume called the **problem of induction**, a philosophical question that asks whether inductive reasoning (a form of reasoning where we draw conclusions about the world based on past observations) really leads us to true knowledge. Strangely enough, this is exactly what machine learning algorithms do. If a neural network sees 100 images of white swans, it will likely conclude that all swans are white. But what happens if the neural network sees a black swan? Now the pattern learned by the algorithm is suddenly disproved by just one counter-example.

Hume used this logic to highlight a limitation of inductive reasoning, the fact that we cannot apply a conclusion about a particular set of observations to a more general set of observations. This same idea became the inspiration for the NFL theorem for machine learning over 200 years later.

Anyway, does it mean that all algorithms are equal? No, of course not. In practice, all algorithms are not created equal. This is because the entire set of machine learning problems is a theoretical concept in the NFL theorem and it is much larger than the set of practical machine learning problems that we will actually attempt to solve. Some algorithms may generally perform better than others on certain types of problems, but every algorithm has disadvantages and advantages due to the prior assumptions that come with that algorithm.

1.5 Exercise: does money make people happy?

To answer this question, we will use a dataset containing information about how happy people are in different countries (the "Better Life Index" dataset from the Organization for Economic Co-operation and Development (OECD) website (bli)), and another dataset containing information about how wealthy people are in different countries (statistics about "GDP per capita" from the International Monetary Fund (IMF) website (gdp)). First of all, we create a folder to store the datasets:

```
!mkdir data
```

```
mkdir: data: File exists
```

Download the data:

```
import urllib.request
import zipfile

remote = 'https://bit.ly/dpg-bli'
urllib.request.urlretrieve(remote, 'data/dpg-bli.zip');

with zipfile.ZipFile('data/dpg-bli.zip', 'r') as zip_ref:
    zip_ref.extractall('data/')
```

Load and prepare "Life satisfaction" data:

```
import pandas as pd

oecd_bli = pd.read_csv('data/gdp-bli/bli.csv', thousands=',')
oecd_bli = oecd_bli[oecd_bli["INEQUALITY"]=="TOT"]
oecd_bli = oecd_bli.pivot(index="Country", columns="Indicator", values="Value")
oecd_bli["Life satisfaction"].head()
```

```
Country
Australia    7.3
Austria      6.9
Belgium      6.9
Brazil       7.0
Canada       7.3
Name: Life satisfaction, dtype: float64
```

Load and prepare "GDP per capita" data:

```
gdp_per_capita = pd.read_csv('data/gdp-bli/gdp.csv', thousands=',', delimiter='\t',
 ↪  encoding='latin1', na_values="n/a")
gdp_per_capita.rename(columns={"2015": "GDP per capita"}, inplace=True)
gdp_per_capita.set_index("Country", inplace=True)
gdp_per_capita.head(4)
```

```
                                          Subject Descriptor         Units  \
Country
Afghanistan  Gross domestic product per capita, current prices  U.S. dollars
Albania      Gross domestic product per capita, current prices  U.S. dollars
Algeria      Gross domestic product per capita, current prices  U.S. dollars
Angola       Gross domestic product per capita, current prices  U.S. dollars

             Scale                      Country/Series-specific Notes  \
Country
Afghanistan  Units  See notes for:  Gross domestic product, curren ...
Albania      Units  See notes for:  Gross domestic product, curren ...
Algeria      Units  See notes for:  Gross domestic product, curren ...
Angola       Units  See notes for:  Gross domestic product, curren ...

             GDP per capita  Estimates Start After
Country
Afghanistan         599.994                 2013.0
Albania            3995.383                 2010.0
Algeria            4318.135                 2014.0
Angola             4100.315                 2014.0
```

Merge the two tables and sort element by GDP per capita:

```
full_country_stats = pd.merge(left=oecd_bli, right=gdp_per_capita, left_index=True,
 ↪  right_index=True)
full_country_stats.sort_values(by="GDP per capita", inplace=True)
full_country_stats[['GDP per capita', 'Life satisfaction']].head(10)
```

```
                GDP per capita  Life satisfaction
Country
```

```
Brazil                 8669.998              7.0
Mexico                 9009.280              6.7
Russia                 9054.914              6.0
Turkey                 9437.372              5.6
Hungary               12239.894              4.9
Poland                12495.334              5.8
Chile                 13340.905              6.7
Slovak Republic       15991.736              6.1
Czech Republic        17256.918              6.5
Estonia               17288.083              5.6
```

Check data for Italy and USA:

```
full_country_stats[['GDP per capita', 'Life satisfaction']].loc['Italy']
```

```
GDP per capita       29866.581
Life satisfaction        6.000
Name: Italy, dtype: float64
```

```
full_country_stats[["GDP per capita", 'Life satisfaction']].loc["United States"]
```

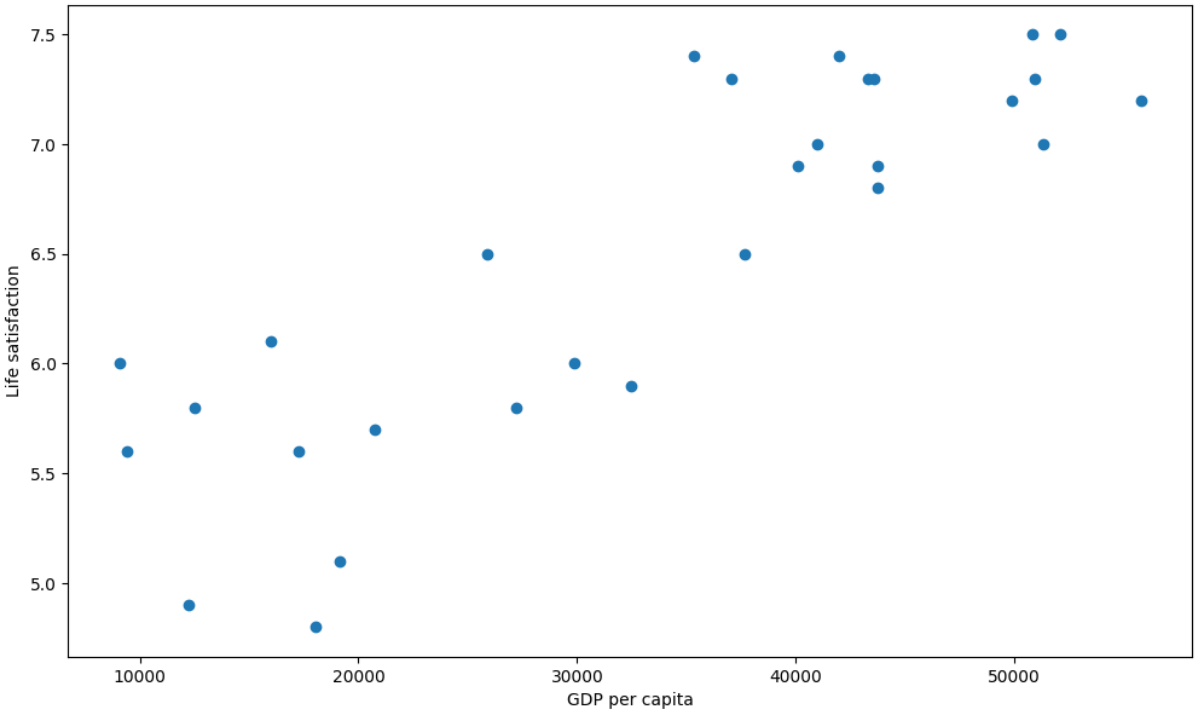
```
GDP per capita       55805.204
Life satisfaction        7.200
Name: United States, dtype: float64
```

Now we remove some data, just to make an important observation later:

```
remove_indices = [0, 1, 6, 8, 33, 34, 35]
keep_indices = list(set(range(36)) - set(remove_indices))

sample_stats = full_country_stats[["GDP per capita", 'Life
 ↪  satisfaction']].iloc[keep_indices]
missing_stats = full_country_stats[["GDP per capita", 'Life
 ↪  satisfaction']].iloc[remove_indices]
```

Visualize the data:

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

plt.figure(figsize=(12,7))
plt.scatter(sample_stats['GDP per capita'], sample_stats['Life satisfaction'])
plt.ylabel('Life satisfaction')
plt.xlabel('GDP per capita')
plt.show()
```



One way to categorize ML systems is by how they **generalize**: that is, how well they perform on new instances (instances they have never seen before). There are two main approaches to generalization: instance-based learning and model-based learning. The most trivial form of learning is simply to learn by heart, then generalizes to new cases using a similarity measure. This is called **instance-based learning**. Another way to generalize is to build a model from the available examples, then use the model to make predictions. This is called **model-based learning**.

In our example seems to be a trend. Although data are noisy (i.e., partly random), it looks like that life satisfaction goes up more or less linearly as the GDP per capita increases. We decide **to model** life satisfaction as a linear function of GDP per capita. This step is called **model selection**. Before we can use our model, we need to define the **parameters of the model** (the line). How can we know which parameters will make our model perform best? To answer this question,

we need to specify a **performance measure**. we can either define a **utility function** (or fitness function) that measures how good a model is, or we can define a **cost function** that measures how bad it is. For linear regression problems, typically we use a cost function that measures the distance between the linear model predictions and the dataset examples; the objective is to minimize this distance. This is where the **Linear Regression algorithm** comes in: we feed it our examples and it finds the parameters that make the linear model fit best to our data. This is called **training the model**.

```
# Prepare the data
X_sample = np.c_[sample_stats["GDP per capita"]]
y_sample = np.c_[sample_stats["Life satisfaction"]]
```

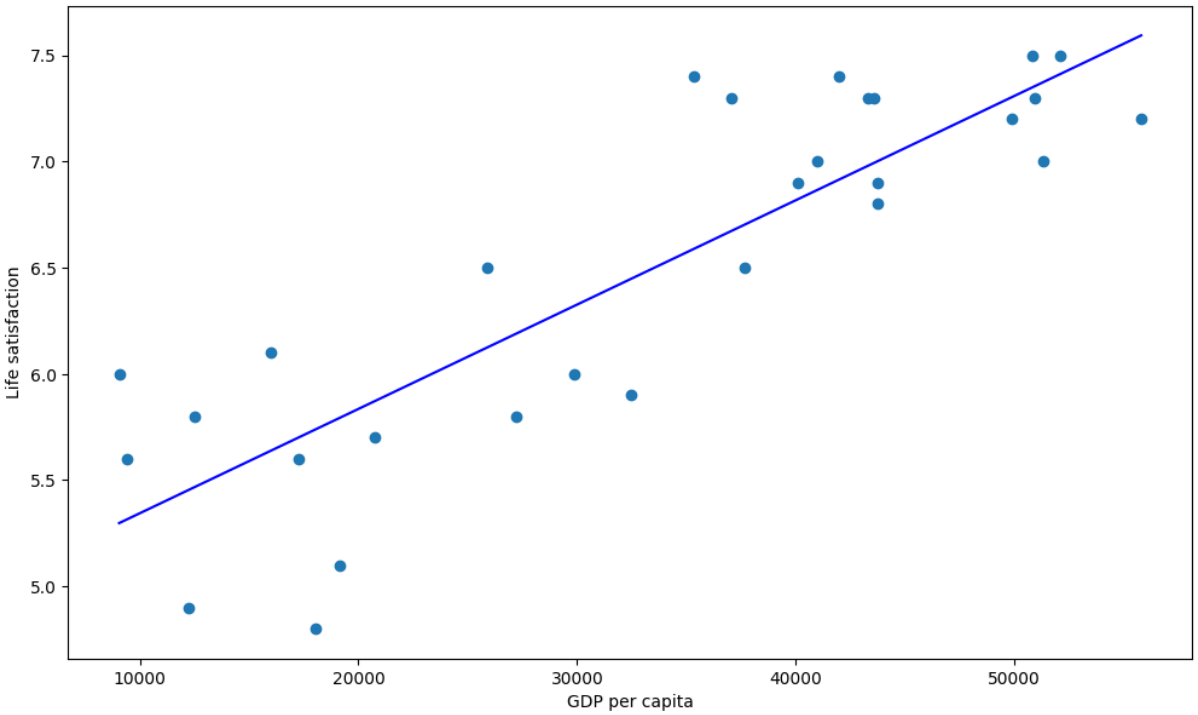
```
import sklearn.linear_model

# Select the linear model
model_linear = sklearn.linear_model.LinearRegression()
```

```
# Train the model
model_linear.fit(X_sample, y_sample)
```

```
LinearRegression()
```

```
# Make predictions
y_pred = model_linear.intercept_[0] + model_linear.coef_[0][0] * X_sample
```

```
# Visualize model and data
plt.figure(figsize=(12,7))
plt.scatter(X_sample, y_sample)
plt.plot(X_sample, y_pred, "b")
plt.ylabel('Life satisfaction')
plt.xlabel('GDP per capita')
plt.show()
```



We are finally ready to run the model to make predictions on unknown data. For example, we want to know how happy Cypriots are, and the OECD data does not have the answer. Fortunately, we can use our model to make a good prediction: we look up Cyprus GDP per capita ($22,587) and then we apply our model to find that life satisfaction prediction:

```
X_cyprus = [[22587]]
y_cyprus = model_linear.predict(X_cyprus)

print(y_cyprus)
```

```
[[5.96242338]]
```

Using an instance-based learning approach instead, we search out around the Cyprus GDP value and we look at the closest countries (e.g. Slovenia, Portugal and Spain), then averaging those values, we can get an estimation of the life satisfaction of Cyprus (5.77), which is close to the model-based prediction obtained before. This simple algorithm is called **k-Nearest Neighbors (KNN)** regression (in this example, k = 3):

```
import sklearn.neighbors
```

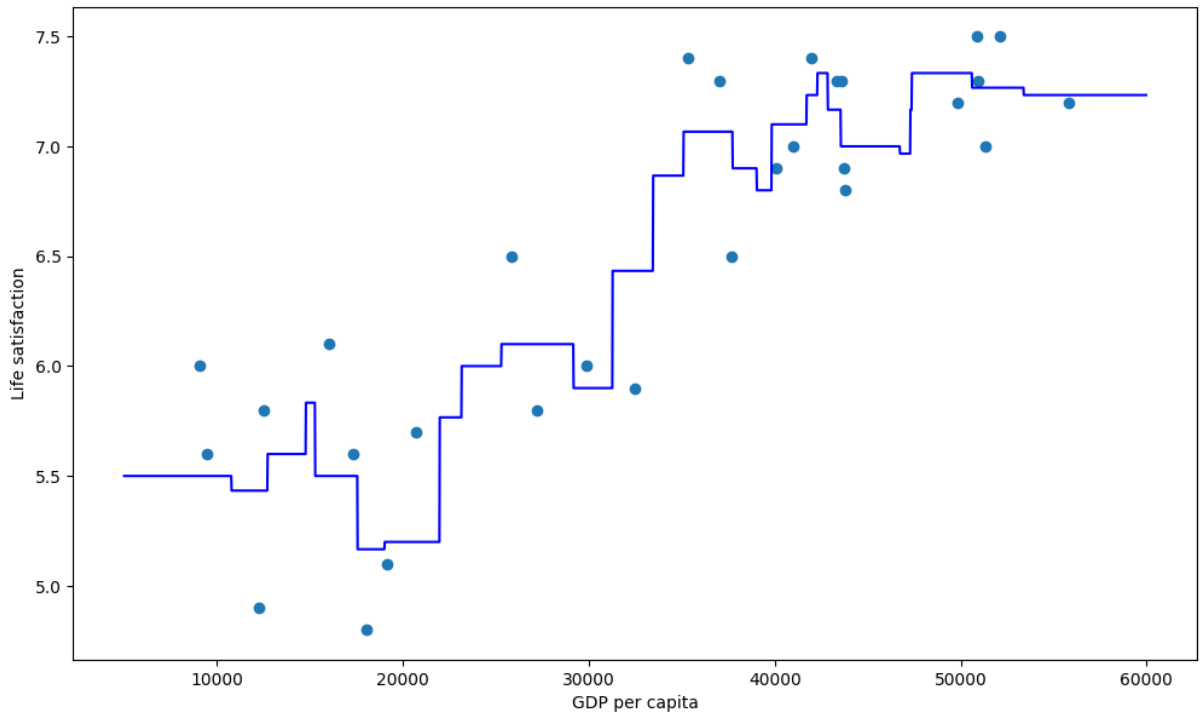
```
# Select KNN
model_knn = sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)
```

```
# Train the model
model_knn.fit(X_sample, y_sample)
```

```
KNeighborsRegressor(n_neighbors=3)
```

```
# Make predictions
values = np.c_[np.arange(5000, 60000, 20)]
y_pred_knn = model_knn.predict(values)
```

```
# Visualize model and data
plt.figure(figsize=(12,7))
plt.scatter(X_sample, y_sample)
plt.plot(values, y_pred_knn, "b")
plt.ylabel('Life satisfaction')
plt.xlabel('GDP per capita')
plt.show()
```



Make a prediction for Cyprus:

```
X_cyprus = [[22587]]
print(model_knn.predict(X_cyprus))
```

```
[[5.76666667]]
```

If all went well, our model will make good predictions. If not, we may need to use **more attributes** (employment rate, health, air pollution, etc.), get **more or better-quality datasets**, or perhaps **select a more powerful model** (e.g., a Polynomial Regression model).

For example, the set of countries we used before was not perfectly representative, because a few countries were missing. We can check the different model that we can obtain using the complete dataset:

Adding a few missing countries **significantly alter the model**, moreover it is clear that such a simple linear model is probably never going to work well. It seems that very rich countries are not happier than moderately rich countries (in fact, they seem unhappier), and conversely some poor countries seem happier than many rich countries.

```
# Use the complete dataset
X_complete = np.c_[full_country_stats["GDP per capita"]]
y_complete = np.c_[full_country_stats["Life satisfaction"]]
```

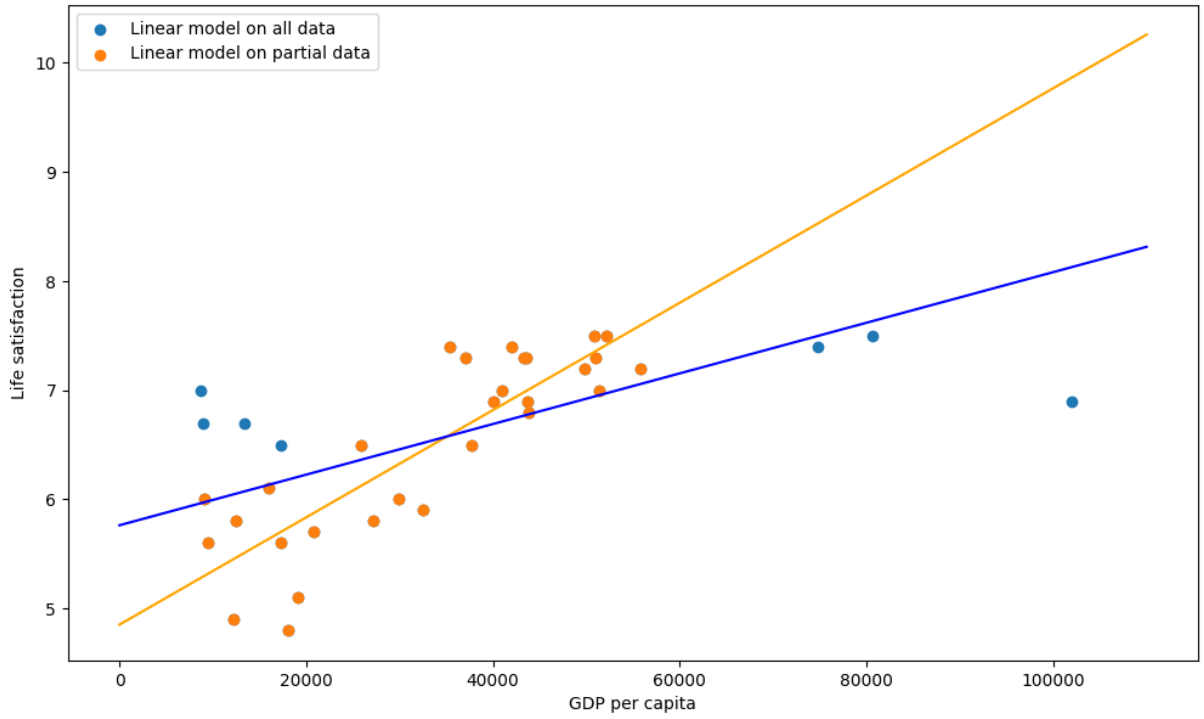
```
# Select a linear model
model_linear_complete = sklearn.linear_model.LinearRegression()
```

```
# Train the model
model_linear_complete.fit(X_complete, y_complete)
```

```
LinearRegression()
```

```
# Make predictions with both models
X = np.linspace(0, 110000, 1000)
y_pred = model_linear.intercept_[0] + model_linear.coef_[0][0] * X
y_pred_complete = model_linear_complete.intercept_[0] +
 ↪  model_linear_complete.coef_[0][0] * X
```

```
# Visualize model and data
plt.figure(figsize=(12,7))
plt.scatter(X_complete, y_complete, label="Linear model on all data")
plt.scatter(X_sample, y_sample, label="Linear model on partial data")
plt.plot(X, y_pred, "orange")
plt.plot(X, y_pred_complete, "b")
plt.ylabel('Life satisfaction')
plt.xlabel('GDP per capita')
plt.legend()
plt.show()
```



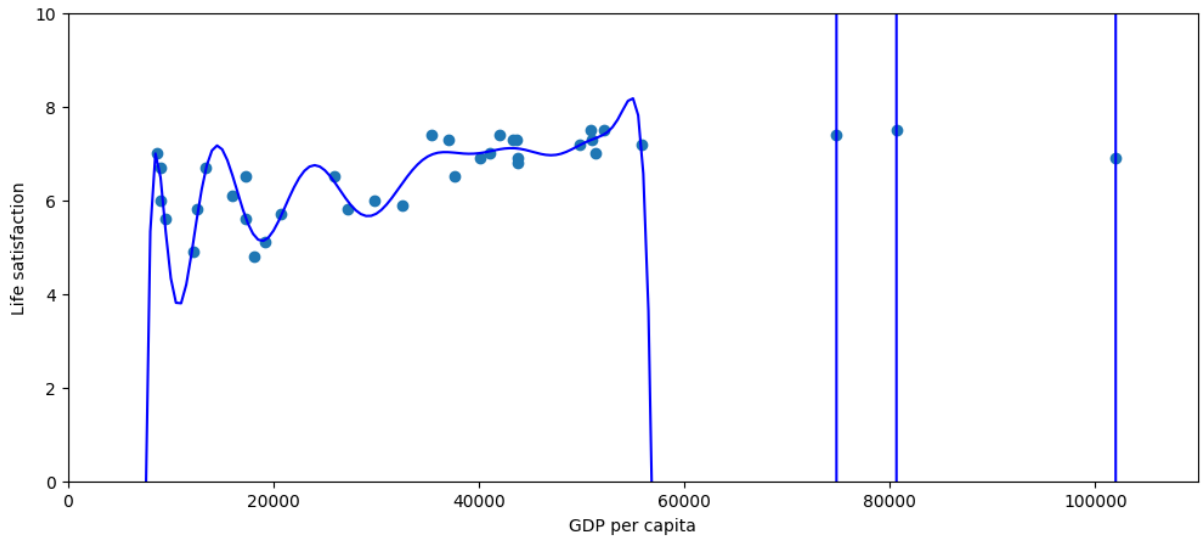

```
from sklearn import preprocessing
from sklearn import pipeline

# Select a high flexible model
poly = preprocessing.PolynomialFeatures(degree=30, include_bias=False)
scaler = preprocessing.StandardScaler()
lin_reg2 = sklearn.linear_model.LinearRegression()
pipeline_reg = pipeline.Pipeline( [('poly', poly),
                                   ('scal', scaler),
                                   ('lin', lin_reg2)])
```

```
# Train the model
pipeline_reg.fit(X_complete, y_complete)
```

```
Pipeline(steps=[('poly', PolynomialFeatures(degree=30, include_bias=False)),
                ('scal', StandardScaler()), ('lin', LinearRegression())])
```

```
# Make predictions
values = np.c_[np.arange(5000, 150000, 500)]
y_pred_flexible = pipeline_reg.predict(values)
```

```
# Visualize model and data
plt.figure(figsize=(12,5))
plt.scatter(X_complete, y_complete)
plt.plot(values, y_pred_flexible, "b")
plt.axis([0, 110000, 0, 10])
plt.ylabel('Life satisfaction')
plt.xlabel('GDP per capita')
plt.show()
```




```
# Select a regularized model
model_linear_ridge = sklearn.linear_model.Ridge(alpha=10**9.5)
```

```
# Train the model
model_linear_ridge.fit(X_sample, y_sample)
```

```
Ridge(alpha=3162277660.1683793)
```

```
# Make predictions
y_pred_regularized = model_linear_ridge.intercept_ + model_linear_ridge.coef_[0] * X
```

```
# Visualize model and data
plt.figure(figsize=(12,5))
plt.scatter(X_complete, y_complete)
plt.scatter(X_sample, y_sample)
plt.plot(X, y_pred, "orange", label="Linear model on partial data")
plt.plot(X, y_pred_complete, "b", label="Linear model on all data")
plt.plot(X, y_pred_regularized, "y", label="Regularized model on partial data")
plt.legend()
plt.ylabel('Life satisfaction')
plt.xlabel('GDP per capita')
plt.show()
```