

# Artificial Intelligence and Machine Learning

T5 Bootcamp by SDAIA

# Agenda

1. Recap: Introduction to Artificial Intelligence and Machine Learning
2. Supervised Learning and Unsupervised Learning
3. Regression:
  - Types: Univariate, Multivariate, Linear, Non-Linear, Polynomial
  - Cost functions
  - Gradient Descent (SGD, Mini batch)
  - Overfitting & Regularization (Lasso, Ridge, Elastic)
  - Decision Trees
  - SVR
  - Performance Metrics: (MSE, RMSE, MAE, R2)
  - Feature Engineering & Feature Selection
  - Data Split and Cross Validation
  - Hyperparameter tuning (Grid Search)
  - Standardization & Normalization



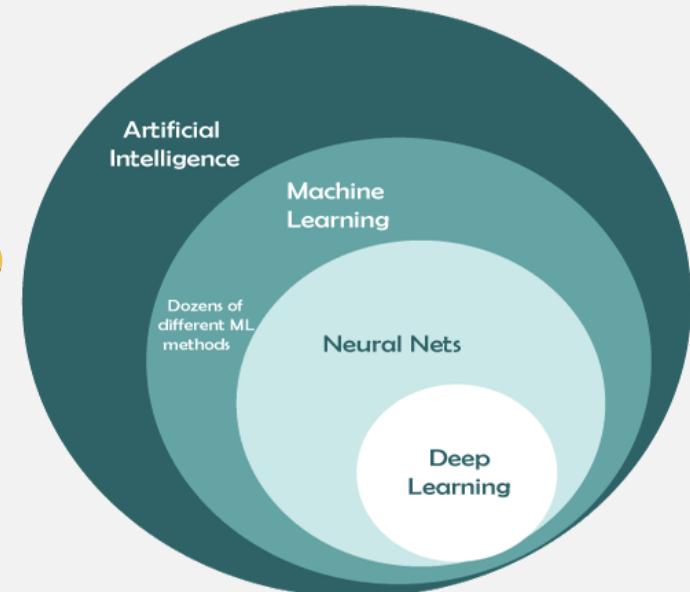
# **Introduction to AI & Machine Learning**



# Artificial Intelligence (AI)

AI, or Artificial Intelligence, is a branch of computer science that focuses on creating systems capable of performing tasks that typically require human intelligence.

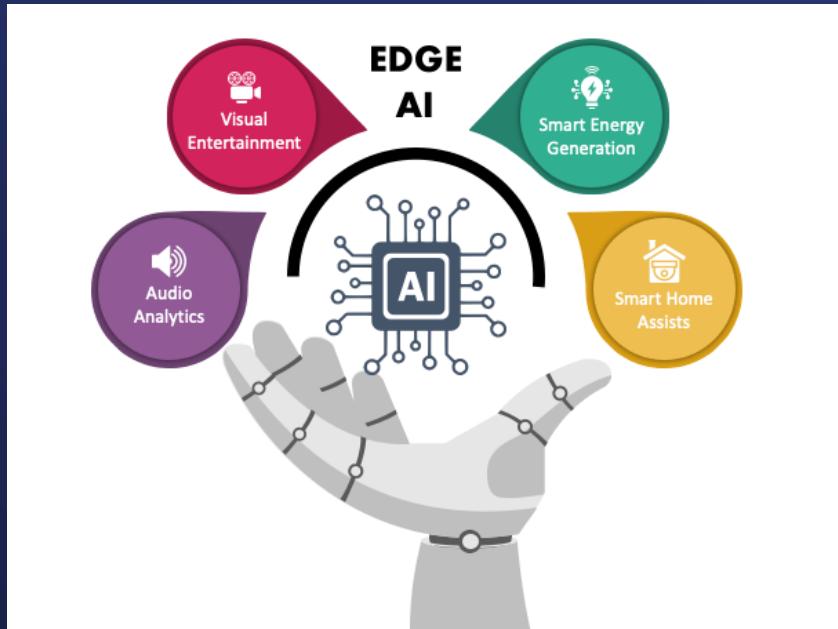
- These tasks include reasoning, problem-solving, learning, perception, language understanding, and decision-making.
- AI algorithms aim to mimic human cognitive functions and behavior, enabling machines to perform tasks autonomously and adaptively.



# Artificial Intelligence (AI)

Famous Applications of AI:

- Natural Language Processing (**NLP**): Chatbots and Language Translation language to another with high accuracy, such as Google Translate and Microsoft Translator.
- **Computer Vision**: Object Detection and Facial Recognition
- **Recommendation Systems**: Personalized Recommendations and individual interests
- **Autonomous Vehicles**: Self-driving Cars
- **Healthcare**: Disease Diagnosis





# What is Machine Learning?

What do you imagine when you read “Machine Learning”? Is it in the future?  
Decades (OCR, Spam filter, Uber, Google, Facebook (Meta))

What is Machine Learning?

*“Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed”. –Arthur Samuel, 1959*

*“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” – Tom Mitchell, 1997*

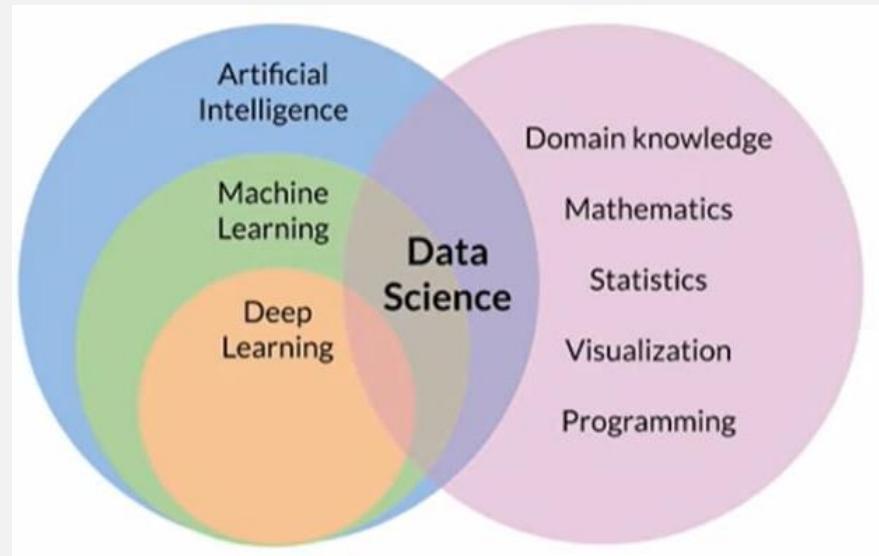




# What is data science?

Data science is the field of study that combines domain expertise, programming skills, and knowledge of mathematics and statistics to extract meaningful insights from data.

- Data science practitioners apply machine learning algorithms to numbers, text, images, video, audio, and more to produce artificial intelligence (AI) systems to perform tasks that ordinarily require human intelligence. In turn, these systems generate insights which analysts and business users can translate into tangible business value

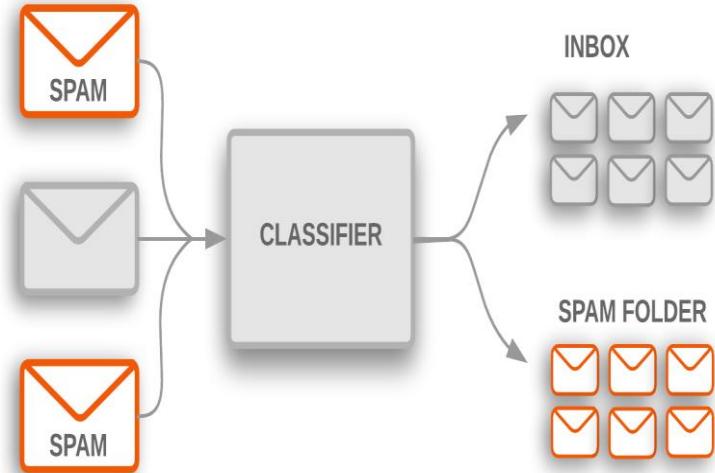




## Spam Filter Example

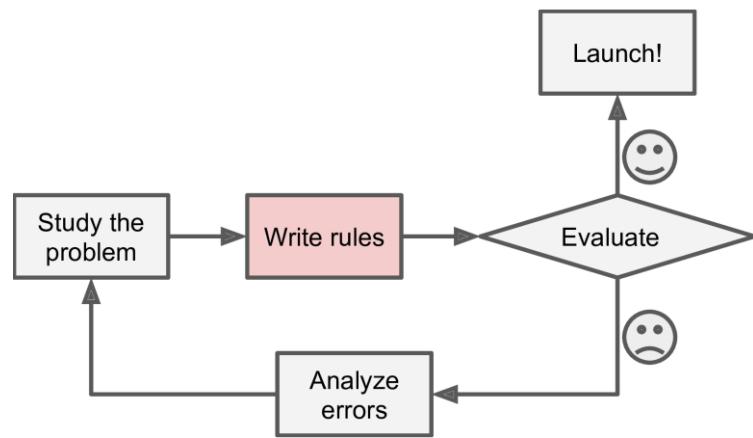
Spam filter is a Machine Learning program. It flags spam or not given examples of spam emails (*e.g., flagged by users*) and examples of regular (*nonspam, also called "ham"*) emails.

- **Training set:** email instances (samples) with answers (or labels).
- **Task T** is to flag spam for new emails
- The **experience E** is the training data
- The **performance measure P** needs to be defined: the ratio of correctly classified emails. (accuracy).

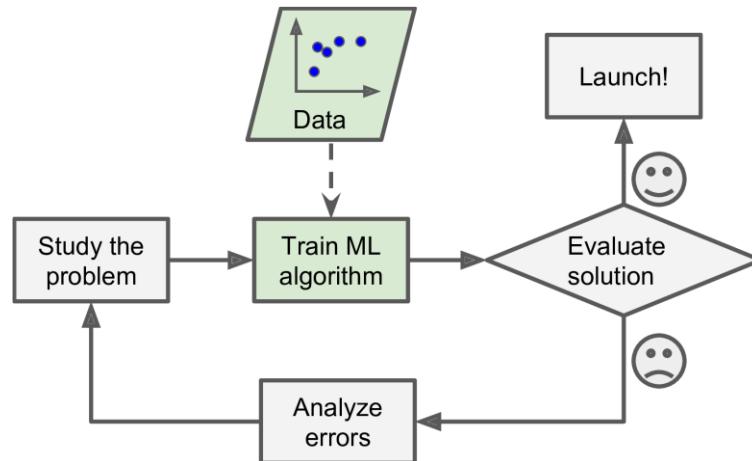




# Traditional Approach vs ML Approach



Traditional Approach



ML Approach





# Why use ML?

## Machine Learning is great for:

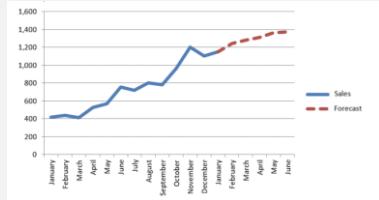
- Problems for which existing solutions require a lot of hand-tuning or long lists of rules
- Complex problems (*no good solution at all using a traditional approach*)
- Fluctuating environments (*adapt to new data*)
- Getting insights about complex problems and large amounts of data.





# ML Applications Examples:

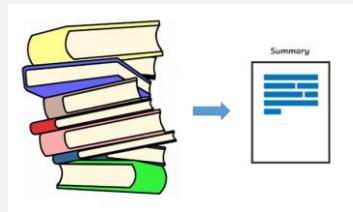
Forecasting next year revenues based on performance metrics



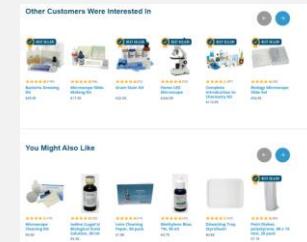
Making your app react to voice commands



Summarizing long documents automatically



Recommending a product to a client based on past purchases





# ML Applications Examples:

Automatic classification of news articles



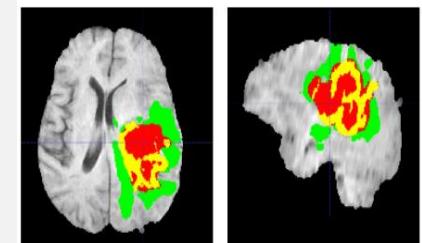
Automatically flagging offensive comments

- Today at 10:33 PM  
**We're reviewing your comment**  
We'll send an update once we've reviewed this again.
  
- Today at 10:32 PM  
**Your comment didn't follow our Community Standards on harassment and bullying**  
Only you and the group admins can see this post.

Analyzing images of products on a production line to automatically classify them



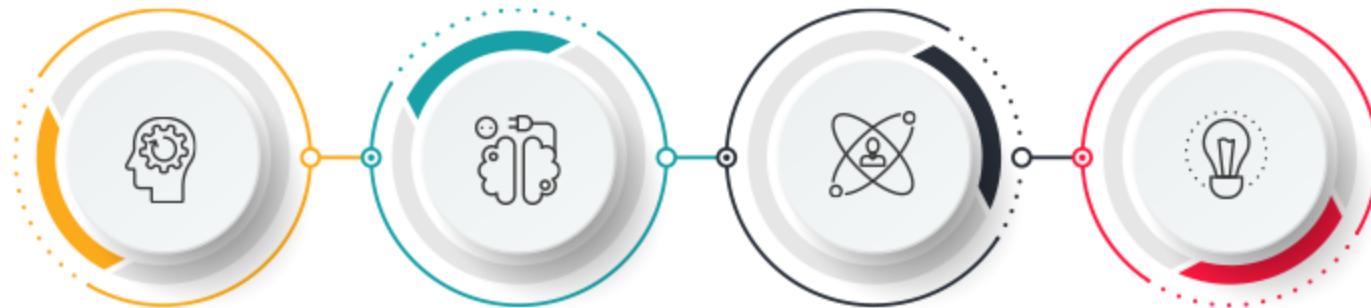
Detecting tumors in brain scans





# Machine Learning Systems Categorizations

- Was it trained with human supervision? (supervised, unsupervised, semi-supervised, and Reinforcement Learning)
- Can it learn incrementally on the fly? (online versus batch learning)
- How does it compare new data? Compare to known data or detect patterns to build predictive models, (instance-based versus model-based learning)



Supervised  
Machine Learning

Unsupervised  
Machine Learning

Semi-Supervised  
Learning

Reinforcement  
Learning



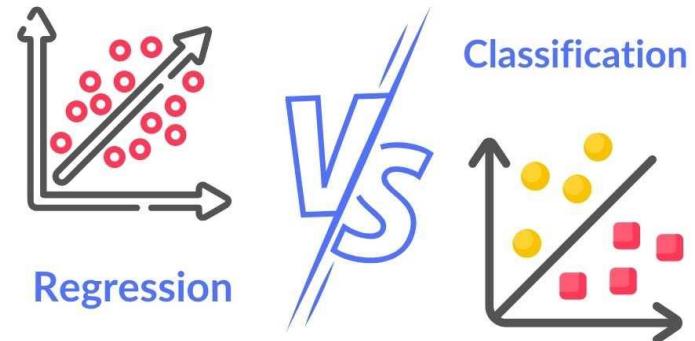


# Supervised Learning

## Supervised Learning:

When the training data you feed to the algorithm includes the desired solutions, called labels (**Labeled Data**)

- Tasks : **Classification, Regression**
- Algorithms: K-Nearest Neighbors, Linear Regression, Logistic Regression, Support Vector Machines (SVMs), Decision Trees and Random Forests, Neural Networks



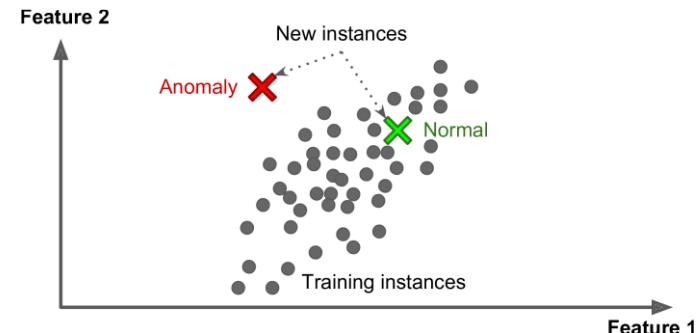
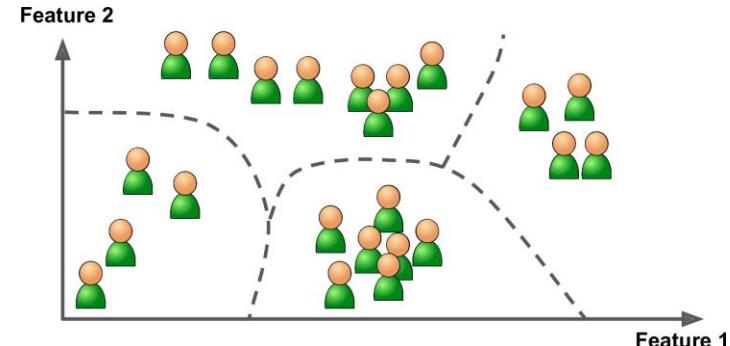


# Unsupervised Learning

## Unsupervised Learning:

The training data is **unlabeled**

- Task: **Clustering** (*K-Means, DBSCAN, Hierarchical Cluster Analysis (HCA)*)
- Task: **Anomaly detection and novelty detection**
- Task: **Visualization and dimensionality reduction** (*Principal Component Analysis (PCA), t-distributed Stochastic Neighbor Embedding (t-SNE)*)
- Task: **Association rule learning** (*Apriori & Eclat*)



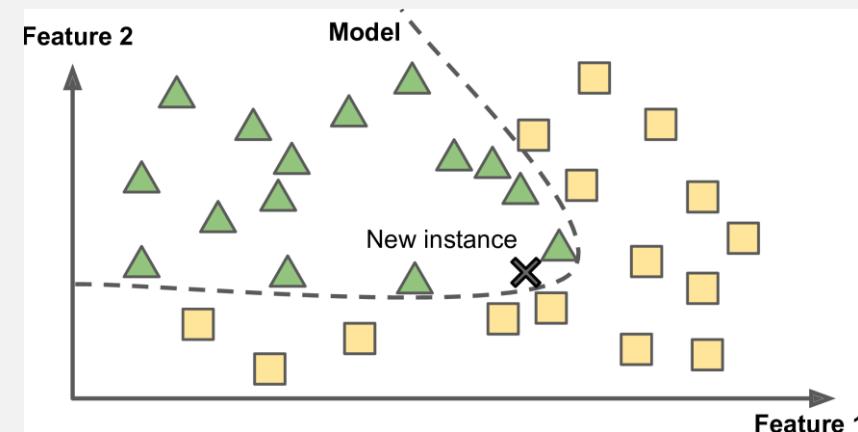


# Instance-Based Vs. Model-Based Learning

- **Instance-Based Learning:** involves memorizing examples and then applying that knowledge to new cases by comparing them to the memorized examples or a subset of them, using a similarity measure.



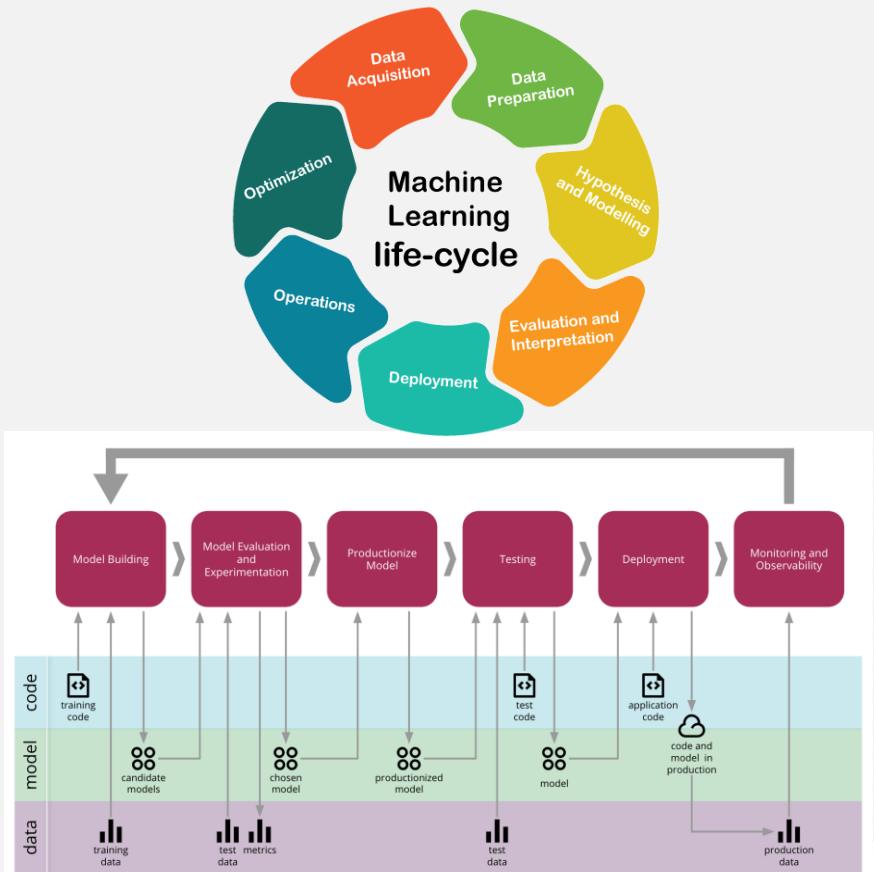
- **Model-Based Learning:** constructs a model of a set of examples, which is then utilized to make predictions..





# End-to-End ML Project

1. Look at the **big picture**.
2. **Get the data**.
3. **Discover and visualize the data to gain insights.**
4. **Prepare the data for Machine Learning algorithms.**
5. **Select a model and train it.**
6. **Fine-tune your model.**
7. **Present your solution.**
8. **Launch, monitor, and maintain your system.**





# Overfitting vs Underfitting

Avoiding **Overfitting**:

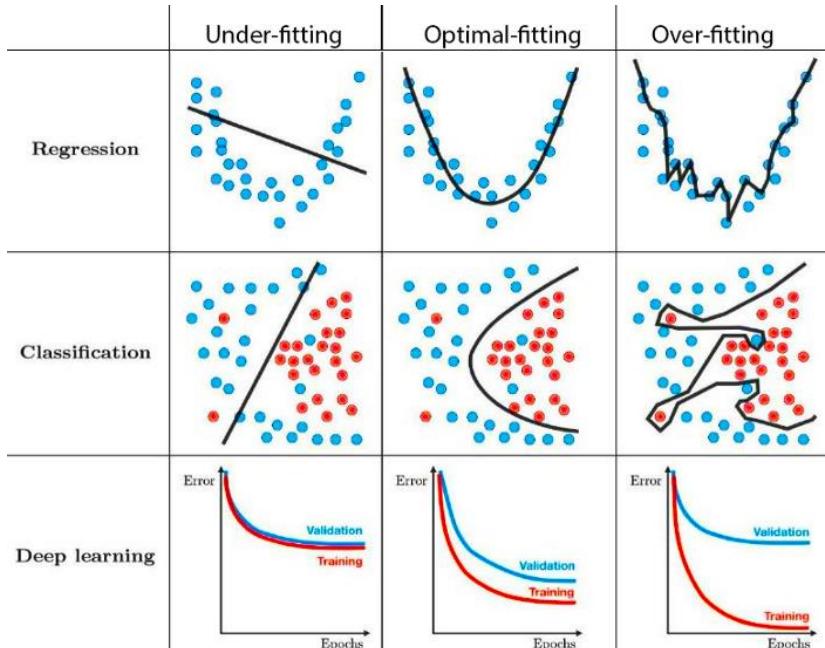
- Simplify the model
- Regularization
- More data
- Reduce the noise in training data (fix errors and remove outliers)

Avoiding **Underfitting**:

- More complex model
- Feature Engineering
- Less Regularization

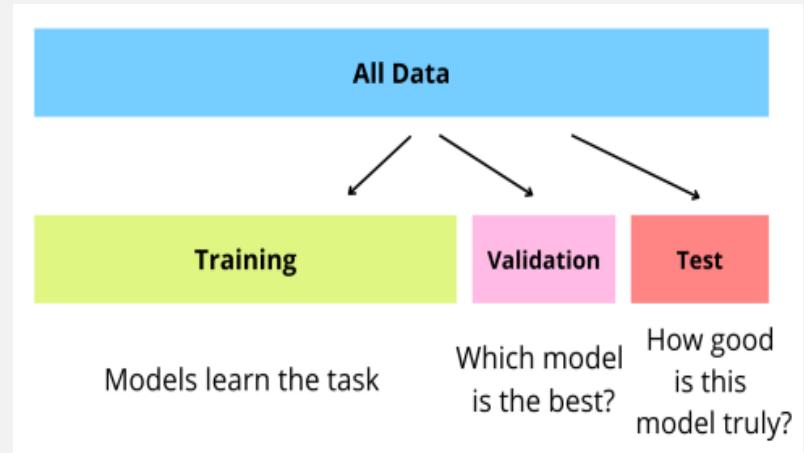
**TARGET:**

- find the right balance



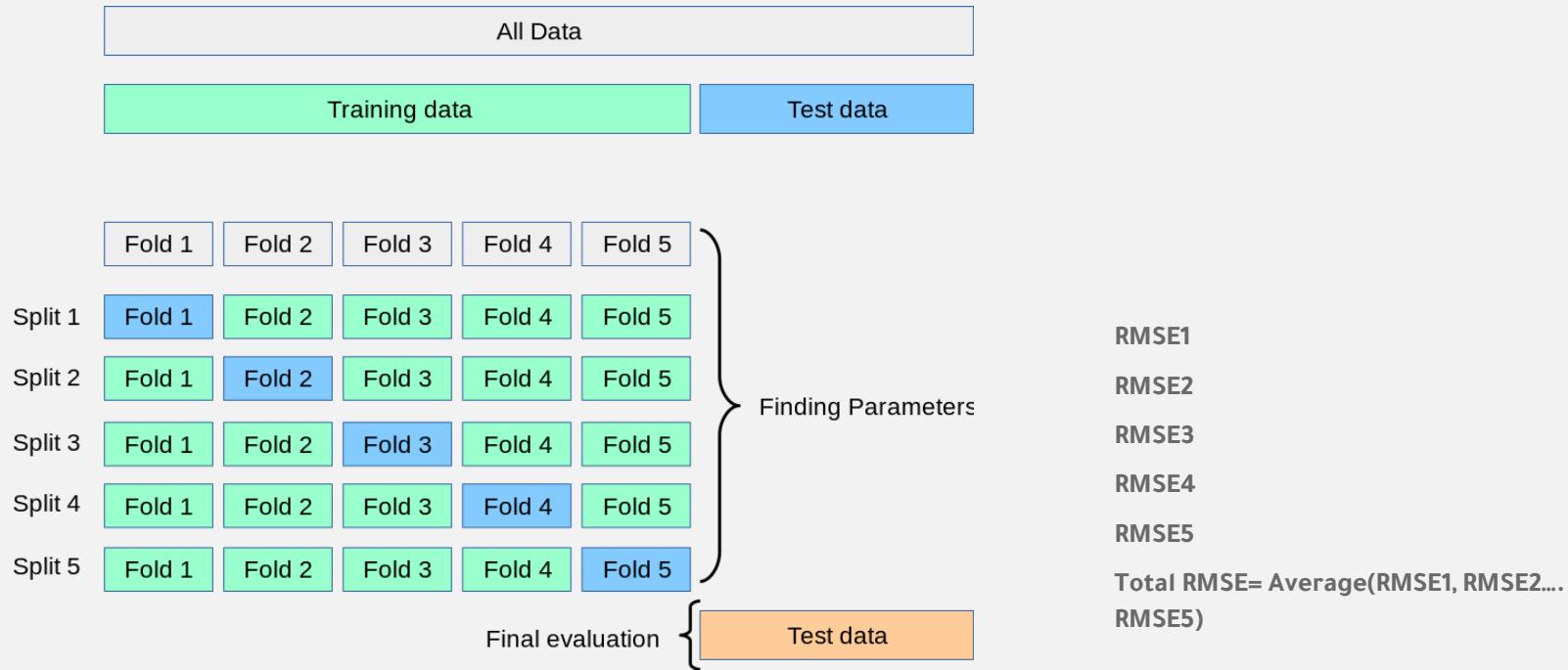
# Data Split: Training, Validation and Testing Data

- **Training Data:** Comprises input-output pairs used to teach a model, adjusting its parameters to minimize prediction errors.
- **Validation Data:** A subset for fine-tuning model hyperparameters and evaluating generalization to new data, detecting overfitting.
- **Testing Data:** A separate subset to assess final model performance on unseen real-world data, determining readiness for deployment and practical applicability.





# Cross Validation





# How Machine Learning looks like in Python?

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.linear_model
# Load the data
oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv",thousands=',',delimiter='\t',
encoding='latin1', na_values="n/a")
# Prepare the data
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]
# Visualize the data
country_stats.plot(kind='scatter', x="GDP per capita", y="Life satisfaction")
plt.show()
# Select a linear model
model = sklearn.linear_model.LinearRegression()
# Train the model
model.fit(X, y)
# Make a prediction for Cyprus
X_new = [[22587]] # Cyprus' GDP per capita
print(model.predict(X_new)) # outputs [[ 5.96242338]]
```

K-Nearest Neighbor Regression:

Replace

```
import sklearn.linear_model
```

```
model = sklearn.linear_model.LinearRegression()
```

with these two:

```
import sklearn.neighbors
```

```
model = sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)
```



---

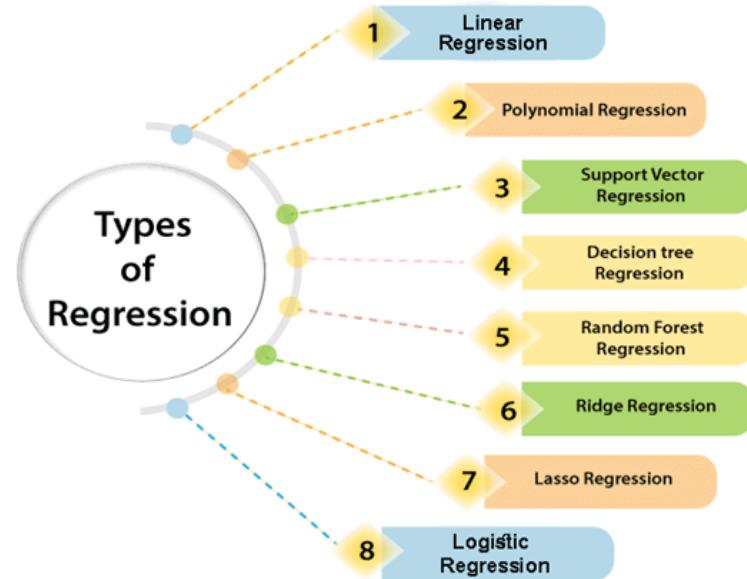
# Supervised Learning 1: Regression



# Types of Regression

**Regression:** mathematical methods that allow data scientists to predict a continuous outcome ( $y$ ) based on the value of one or more predictor variables ( $x$ ).

- Linear regression
- Univariate Regression
- Multiple Regression
- Multivariate Regression
- Polynomial Regression





# Notation in ML

- $m$  is the number of instances in the dataset
- $X(i)$  is a vector of all the feature values (excluding the label) of the  $i$ th instance
- Also called variables
- $y(i)$  is its label (target, labels or answers)
- $X$  is a matrix containing all the feature values (excluding labels)
- $Y$  is a vector containing all labels
- $h$  is your system's prediction function, also called a hypothesis, it outputs a predicted value  $\hat{y}(i) = h(x(i))$  for instance  $i$ .

Assume the first district in the dataset is located at longitude - 118.29°, latitude 33.91°, and it has 1,416 inhabitants with a median income of \$38,372, and the median house value is \$156,400

Feature Matrix ( $X$ )	Target Vector ( $y$ )
$n\_features \rightarrow$ 	$n\_samples \rightarrow$ 
$X = \begin{pmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(1999)})^T \\ (x^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$	$y^{(1)} = 156,400 \quad x^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1,416 \\ 38,372 \end{pmatrix}$



# Example

## Simple Real Estate

$m$  is the number of instances in the dataset

$X^{(i)}$  is a vector of all the feature values (excluding the label) of the  $i$ -th instance

$y^{(i)}$  is its label (the desired output value for that instance).

$X^{(1)}$  is : 100 and  $y^{(1)}$  is 360,000 SR

$$h(x) = \hat{Y} = \theta_0 + \theta_1 \times \text{Area} = \theta_0 + \theta_1 X_1$$

Area (m <sup>2</sup> )	Price (y)	Prediction ( $\hat{y}$ )
100	360,000	504,000
140	450,000	504,000
170	570,000	504,000
200	660,000	504,000
150	480,000	504,000

What do you think the values for  $\theta_0$ ,  $\theta_1$  are? What do we call them?

$h(x) = \hat{Y} = \theta_0 + \theta_1 \times \text{Area} + \theta_2 \times \text{rooms} + \theta_3 \times \text{WCs} = \theta_0 + \theta_1 X_1 + \theta_2 X_2 + \theta_3 X_3$  "Multivariate Regression"

$h(x) = \hat{Y} = \theta_0 + \theta_1 X_1 + \theta_2 X_1 X_2 + \theta_3 X_1^2 + \theta_4 X_2^2$  .... "Polynomial Regression, degree=2"





# How to measure model performance?

## Performance Measures (Error):

Measuring model performance allows us to evaluate how well a machine learning model performs on unseen data, providing insights into its effectiveness in making accurate predictions or classifications

- ❖ **Root Mean Squared Error (RMSE):** Euclidean norm,  $L_2$  norm noted  $\|\cdot\|_2$  or just  $\|\cdot\|$

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(\mathbf{x}^{(i)}) - y^{(i)})^2}$$

- ❖ **Mean Squared Error:** RMSE without the squared-root

- ❖ **Mean Absolute Error (MAE):** Manhattan norm, L1 norm noted  $\|\cdot\|_1$

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}|$$

- ❖ **R-Squared:**

$SS_{RES}$ : sum of squared errors

$SS_{TOT}$ : total sum of squared errors

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$





# Example

## Simple Real Estate

Area (m <sup>2</sup> )	Price (y)	Mean ( $\bar{y}$ )	Prediction ( $\hat{y}$ )	RMSE	MAE	RSS	RTT	R2
100	360,000	504,000	350,000	100,000,000	10,000	100,000,000	20,736,000,000	
140	450,000	504,000	470,000	400,000,000	20,000	400,000,000	2,916,000,000	
170	570,000	504,000	560,000	100,000,000	10,000	100,000,000	4,356,000,000	
200	660,000	504,000	650,000	100,000,000	10,000	100,000,000	24,336,000,000	
150	480,000	504,000	500,000	400,000,000	20,000	400,000,000	576,000,000	
			Total	14,832	14,000	1,100,000,000	52,920,000,000	0.979214

Area (m <sup>2</sup> )	Price (y)	Mean ( $\bar{y}$ )	Prediction ( $\hat{y}$ )	RMSE	MAE	RSS	RTT	R2
100	360,000	504,000	504,000	20,736,000,000	144,000	20,736,000,000	20,736,000,000	
140	450,000	504,000	504,000	2,916,000,000	54,000	2,916,000,000	2,916,000,000	
170	570,000	504,000	504,000	4,356,000,000	66,000	4,356,000,000	4,356,000,000	
200	660,000	504,000	504,000	24,336,000,000	156,000	24,336,000,000	24,336,000,000	
150	480,000	504,000	504,000	576,000,000	24,000	576,000,000	576,000,000	
			Total	102,879	88,800	52,920,000,000	52,920,000,000	0

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(\mathbf{x}^{(i)}) - y^{(i)})^2} \quad \text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}| \quad R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

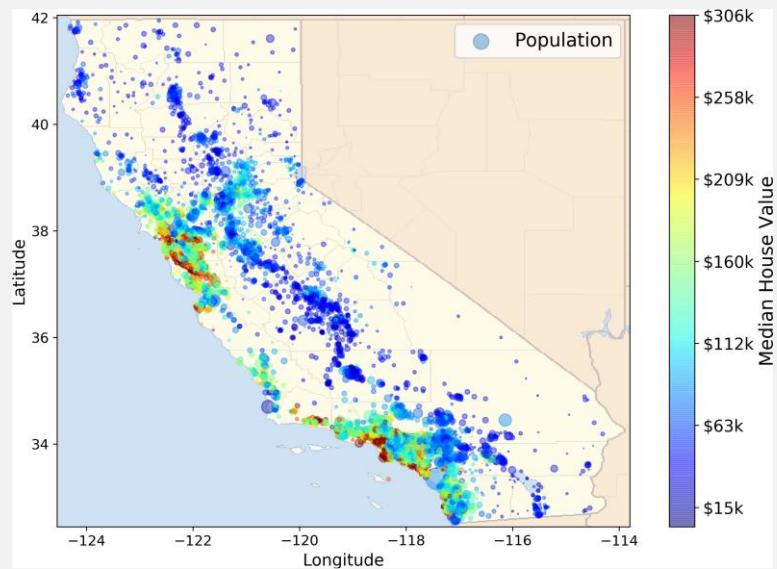




# Example

## California Housing Prices

- 1990 California census.
- Let's build a model of housing prices in the state
- Data include metrics like:
  - *Population*
  - *Median income*
  - *Median housing price for district*
- What is the predicted value?
  - (*Price* → continuous numeric → Regression)





# How do we train a model?

## The verb train (learn), What does it mean?

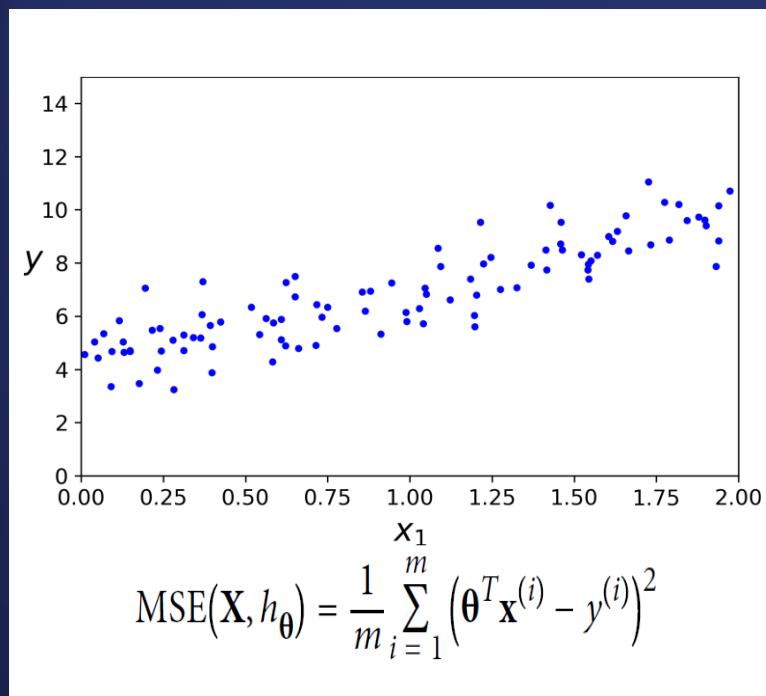
The term "train" in machine learning refers to the process of teaching a model using input-output pairs or examples, adjusting its parameters to minimize errors between its predictions and the correct outputs.

Cost function:

$$J(\theta) = \text{MSE}(\theta) = \text{Normal Equation:}$$

$$\theta = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

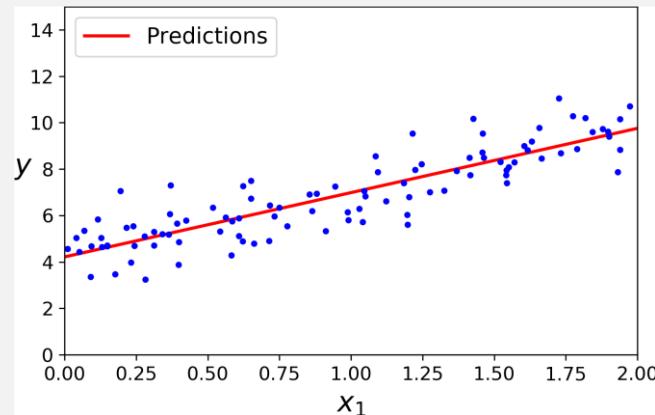
```
import numpy as np  
X = 2 * np.random.rand(100, 1)  
y = 4 + 3 * X + np.random.randn(100, 1)
```





# How do we train a model?

```
from sklearn.linear_model import LinearRegression  
lin_reg = LinearRegression()  
  
lin_reg.fit(X, y)  
  
lin_reg.intercept_, lin_reg.coef_  
(array([4.21509616]), array([[2.77011339]]))  
  
lin_reg.predict(X_new)  
array([[4.21509616], [9.75532293]])
```



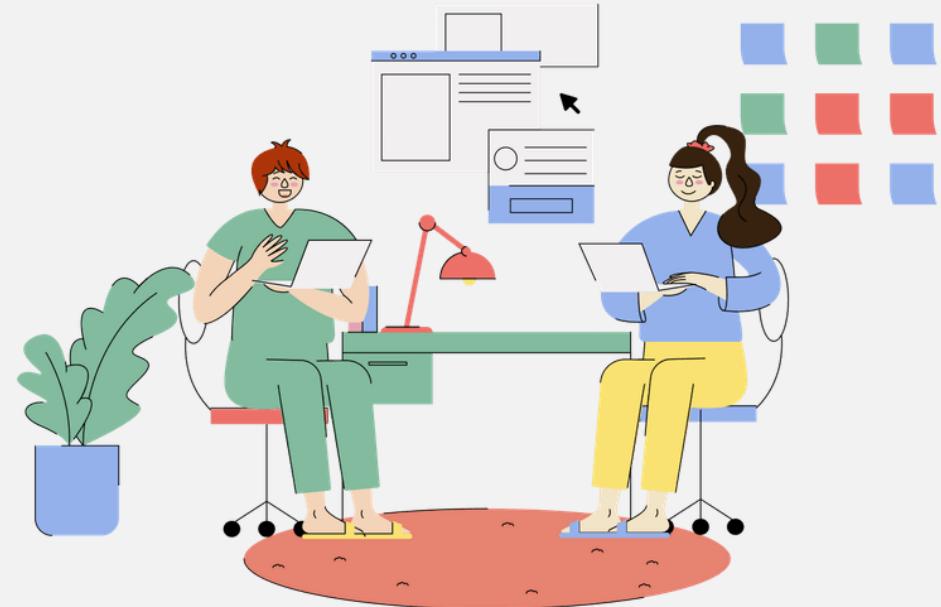


# Exercise

**Transitioning to Google-Colab for hands-on coding practice.**

Notebook:

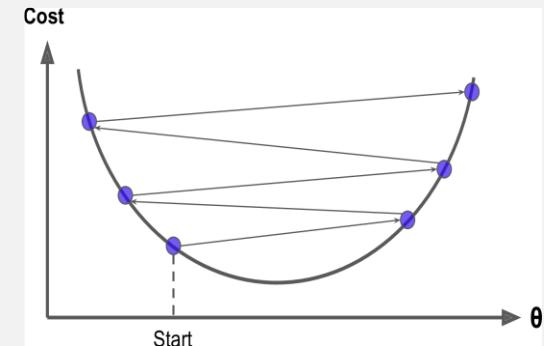
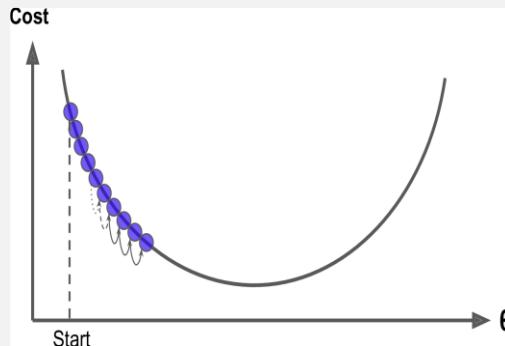
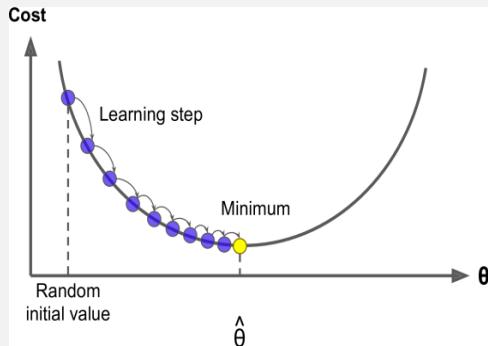
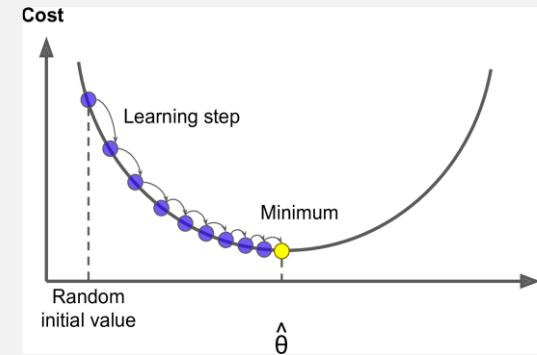
- T5B3ML101N00\_032024V1 - Linear\_Reggression





# How do we train the model?

- Randomly initialized parameters (Large Loss)
- How gradient descent is doing it?
- Learning rate ( $\alpha$ )
- Model Hyperparameter ( $\alpha$ ) vs Model Parameters ( $\Theta$ )

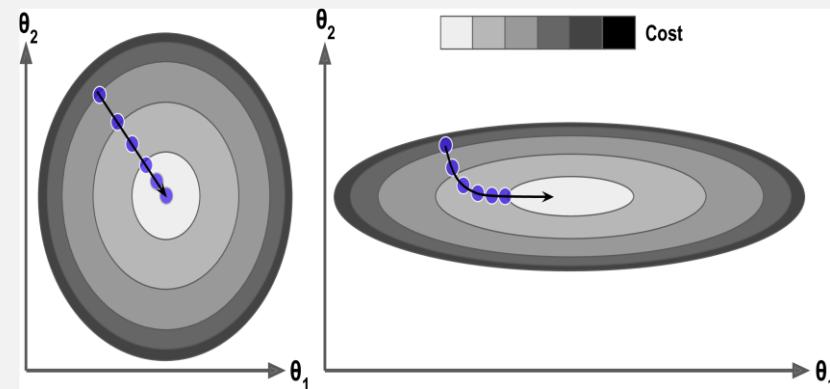




# Feature Scaling (Normalization)

## The advantages of Feature Scaling

- Enhances algorithm performance by ensuring that features contribute equally to the learning process.
- Mitigates impact of feature magnitudes on model training by preventing features with larger scales from dominating the learning process.
- Faster convergence
- Easier to interpret model coefficients and understand their relative importance
- Increase ML stability
- Facilitates regularization





# Feature Scaling (Normalization)

## Methods of Feature Scaling

### Min-Max Scaler:

- Scales features to a specified range, typically between 0 and 1, preserving the relative distances between data points.
- Suitable for algorithms sensitive to feature scaling, such as neural networks, but may be affected by outliers.

$$\frac{x_i - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})}$$

### Standard Scaler:

- Transforms features to have a mean of 0 and a standard deviation of 1, ensuring each feature has a similar scale.
- Robust to outliers and suitable for algorithms relying on normal distribution assumptions, like linear regression and logistic regression.

$$\frac{x_i - \text{mean}(\mathbf{x})}{\text{stdev}(\mathbf{x})}$$

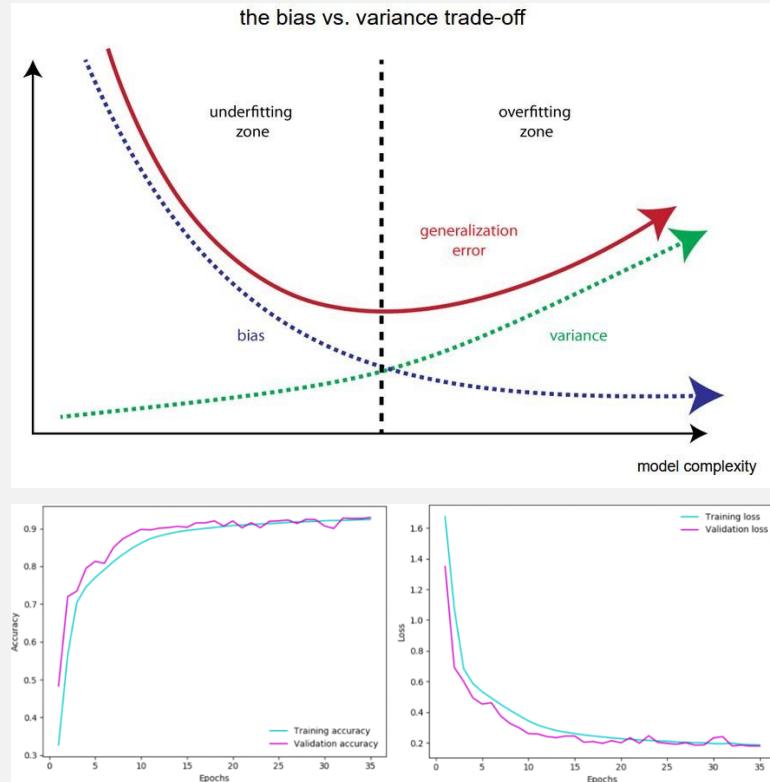




# Learning Curve

Graphical representations that illustrate the performance of a machine learning model as a function of learning step parameter.

- Depicts the relationship between training set size and model performance metrics, such as training and validation error or accuracy.
- Identifies high bias (underfitting) or high variance (overfitting).
- Learning step:
  - ✓ Too low ( $\eta$ ): waste time
  - ✓ Too high ( $\eta$ ): far away from the optimal solution

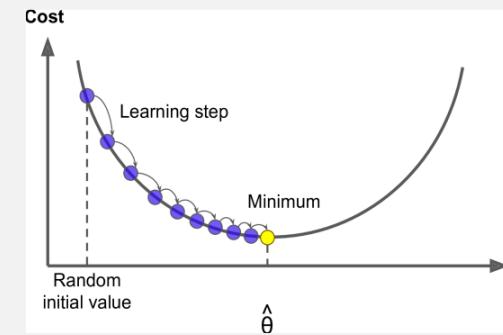




# Gradient Descent

Gradient Descent algorithm finds optimal solutions by tweaking parameters  $\theta$  iteratively to minimize a cost function  $J(\theta)$ .

- Start by random values for  $\theta$  (random initialization), and then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function until it converges to a minimum
- Learning step parameter:
  - ✓ Too low ( $\eta$ ): waste time
  - ✓ Too high ( $\eta$ ): far away from the optimal solution
- Best Practice:
  - ✓ Very large number of iterations but to interrupt the algorithm when the gradient vector becomes tiny—that is, when its norm becomes smaller than a tiny number  $\epsilon$  (tolerance).



# Gradient Descent

Imagine yourself standing at the top of a hill. Your goal is to reach the bottom. In the context of gradient descent, this is like finding the most efficient way down the hill.

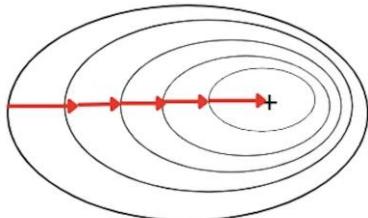
- **Gradient descent** is an optimization algorithm used in machine learning and numerical optimization. Its primary purpose is to minimize a cost function by iteratively adjusting the model parameters. The key idea is to move in the direction of steepest decrease in the cost function.



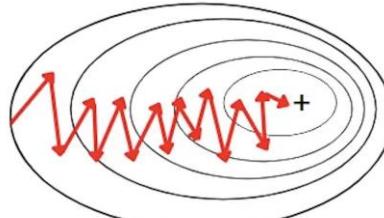
# Different types of Gradient Descent

Imagine a situation where you with group want to go down a hill together.

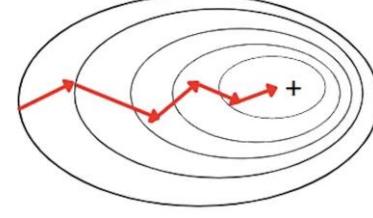
**Batch Gradient Descent**



**Stochastic Gradient Descent**



**Mini-Batch Gradient Descent**



Batch gradient descent looks at the entire hill to find the steepest direction.

Stochastic gradient descent looks at a small part of the hill and updates the position individually.

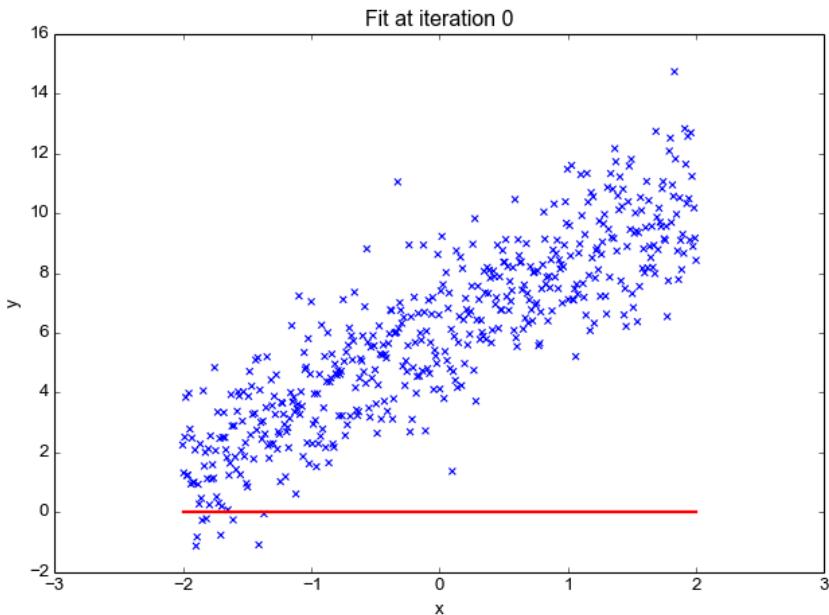
Mini-batch gradient descent divides into teams, with each team looking at a specific part of the hill and updating the position together.





# How Gradient Descent works?

- 1) Initialize the parameters of the model randomly.
- 2) Compute the gradient of the cost function with respect to each parameter. It involves making partial differentiation of cost function with respect to the parameters.
- 3) Update the parameters of the model by taking steps in the opposite direction of the model.
- 4) Repeat steps 2 and 3 iteratively to get the best parameter for the defined model





# Gradient Descent

For simplicity, we'll consider the case of linear regression, where the goal is to find the optimal weights  $\theta$  to minimize the mean squared error cost function.

## Cost Function for Linear Regression:

The mean squared error (MSE) cost function for linear regression is given by:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

Here:

- $m$  is the number of training examples.
- $h_\theta(x^{(i)})$  is the hypothesis function, representing the predicted value for the  $i$ -th example.
- $y^{(i)}$  is the actual target value for the  $i$ -th example.





# Gradient Descent

## Gradient of the Cost Function:

The gradient ( $\nabla J(\theta)$ ) is a vector of partial derivatives, indicating the rate of change of the cost function with respect to each parameter ( $\theta_j$ ). For linear regression, the partial derivative with respect to ( $\theta_j$ ) is given by:

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

Where;  $x^{(i)}$  is the  $j$ -th feature of the  $i$ -th example.

## Update Rule:

The gradient descent update rule for each parameter ( $\theta_j$ ) is as follows:

$$\theta_j = \theta_j - \alpha \cdot \frac{\partial J}{\partial \theta_j}$$

where;  $\alpha$  is the learning rate, a hyperparameter that controls the size of the steps taken during each iteration.

## Iterative Process:

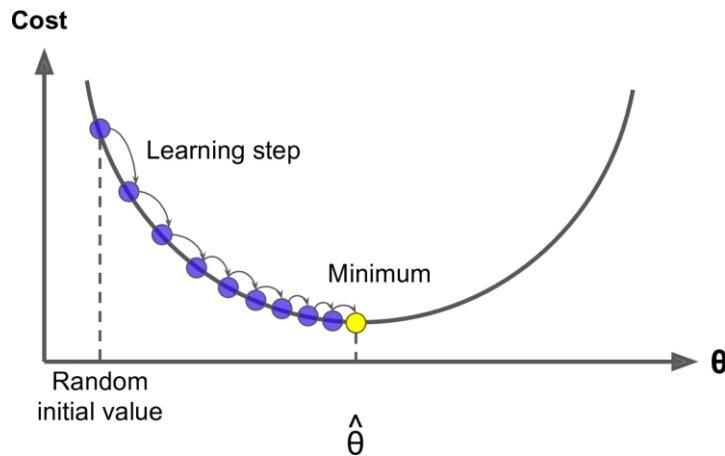
The overall iterative process of gradient descent involves repeatedly applying the update rule for each parameter until convergence.





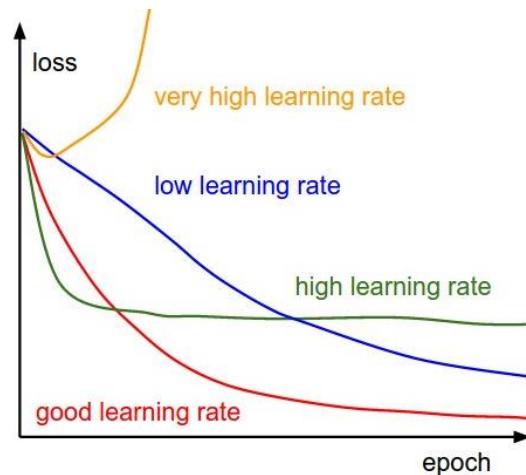
# Learning Rate

The learning rate is a hyperparameter that determines the size of the steps taken during optimization.



The choice of learning rate is critical for successful convergence:

- ✓ A small learning rate  $\alpha$  may cause slow convergence.
- ✓ A large learning rate  $\alpha$  may lead to oscillation or divergence.



# Gradient Descent - Example

To perform gradient descent on the function  $f(x) = x^2$

with a step size of 0.8, starting from  $x^{(0)} = -4$ , and using the update rule

$x^{(1)} = x^{(0)} - \alpha \cdot \nabla f(x^{(0)})$ , where  $\nabla f(x^{(0)})$  is the gradient of  $f$  at  $x^{(0)}$ ,

we can follow these steps:

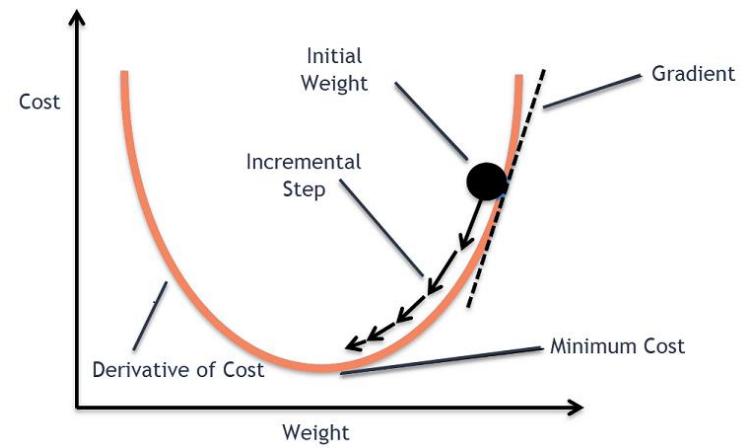
1. Define the function  $f(x)$ :

$$f(x) = x^2$$

2. Initialize parameters:

$$\alpha = 0.8$$

$$x^{(0)} = -4$$



# Gradient Descent - Example

3. Calculate the gradient  $\nabla f(x) x^2$ :

$$f'(x) = 2x$$

$$\nabla f(x^{(0)}) = 2 \cdot (-4) = -8$$

4. Update  $x$  using the gradient descent update rule:

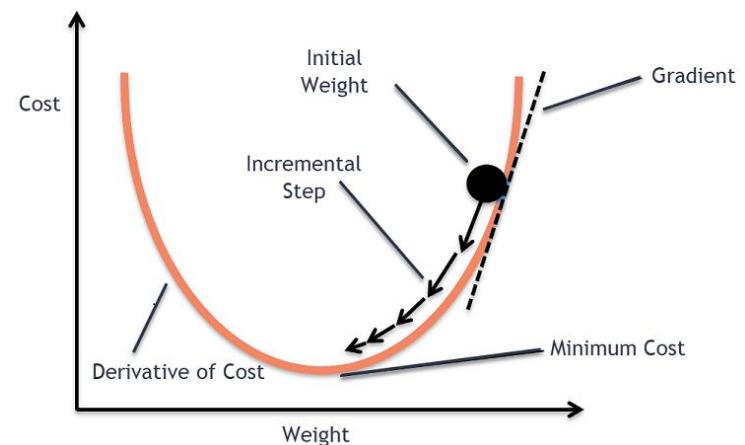
$$x^{(1)} = x^{(0)} - \alpha \nabla f(x^{(0)})$$

$$x^{(1)} = -4 - 0.8(-8)$$

Now, let's perform the calculation:

$$x^{(1)} = -4 - 0.8 \cdot (-8) = -4 + 6.4 = 2.4$$

So, after one iteration of gradient descent, starting from  $x^{(0)} = -4$  with a step size of 0.8, we obtain  $x^{(1)} = 2.4$



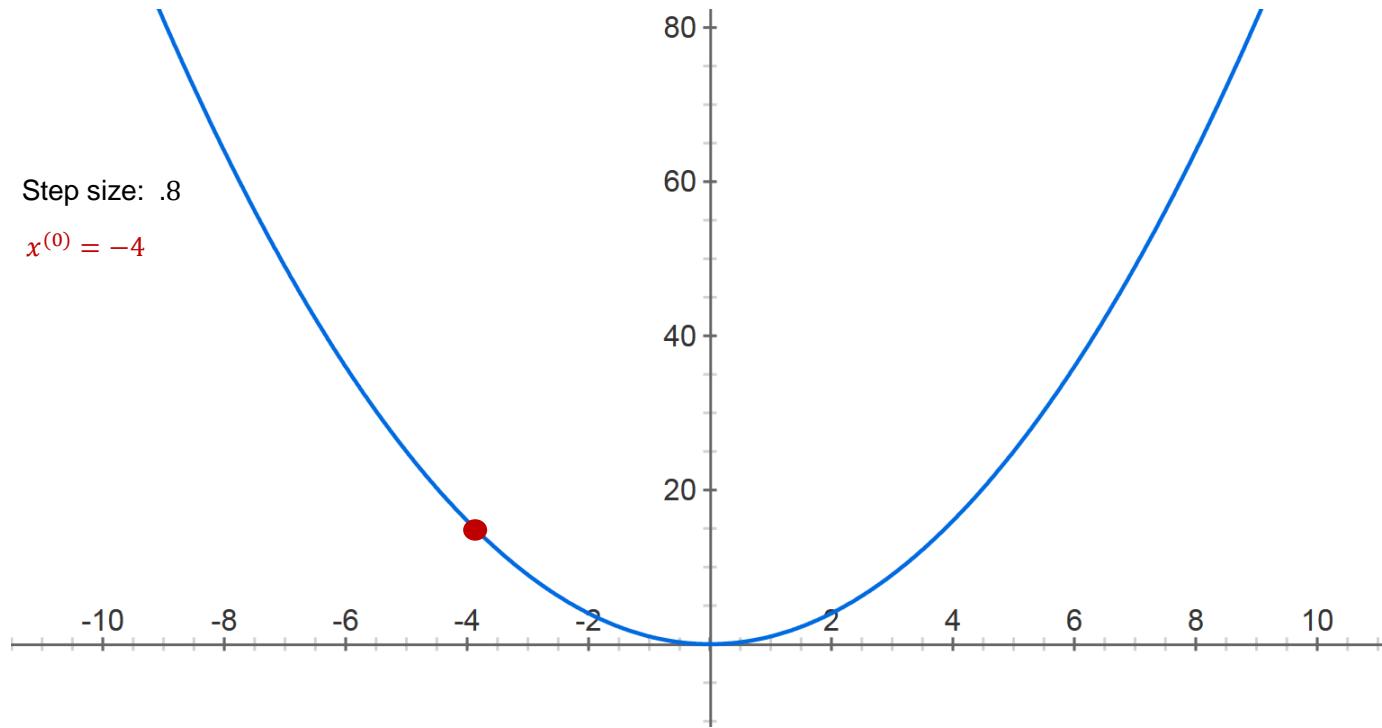


## Gradient Descent - Example

$$f(x) = x^2$$

Step size: .8

$x^{(0)} = -4$





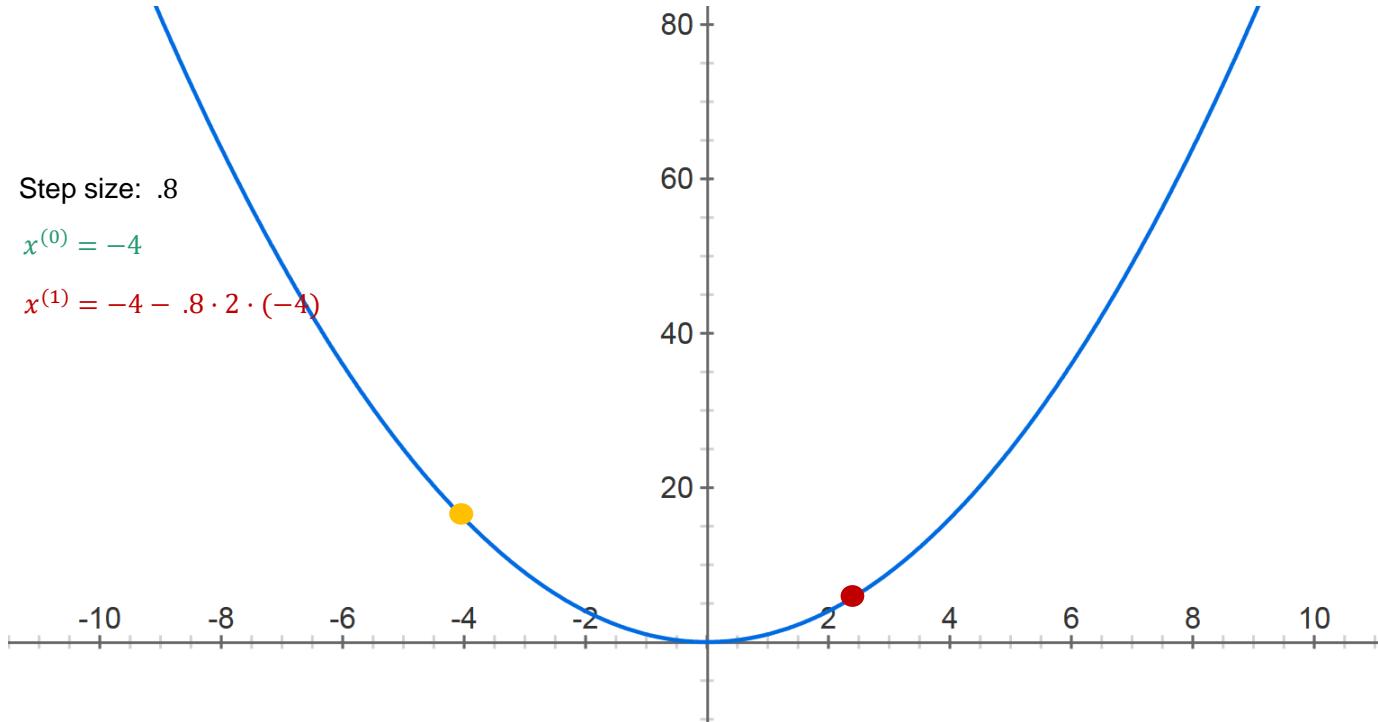
# Gradient Descent - Example

$$f(x) = x^2$$

Step size: .8

$$x^{(0)} = -4$$

$$x^{(1)} = -4 - .8 \cdot 2 \cdot (-4)$$





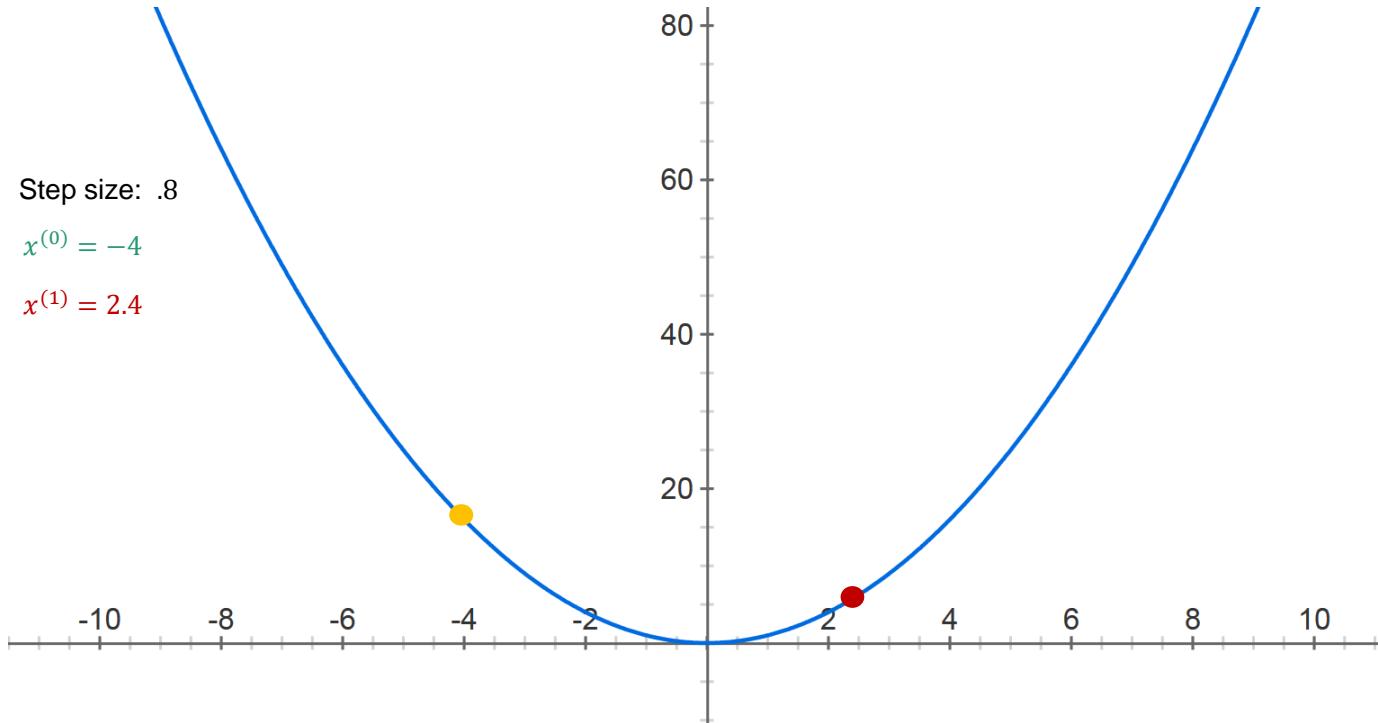
# Gradient Descent - Example

$$f(x) = x^2$$

Step size: .8

$$x^{(0)} = -4$$

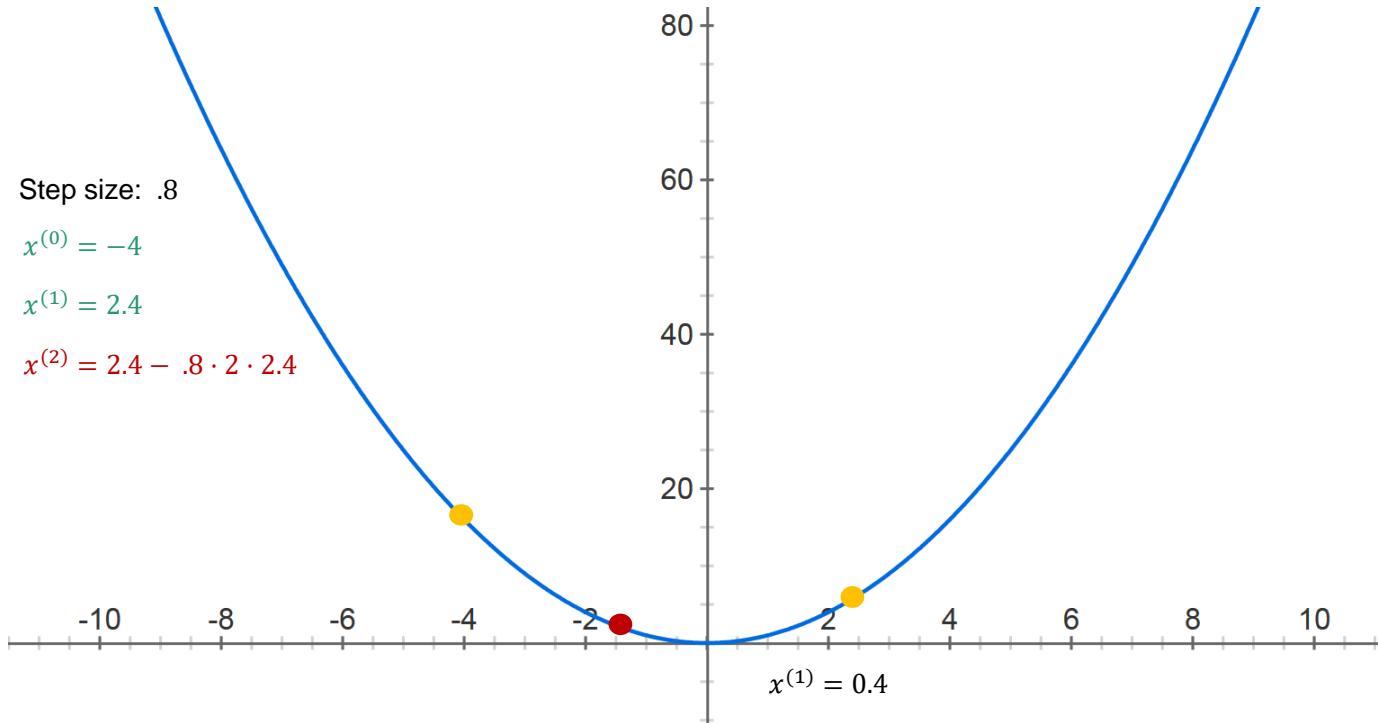
$$x^{(1)} = 2.4$$





# Gradient Descent - Example

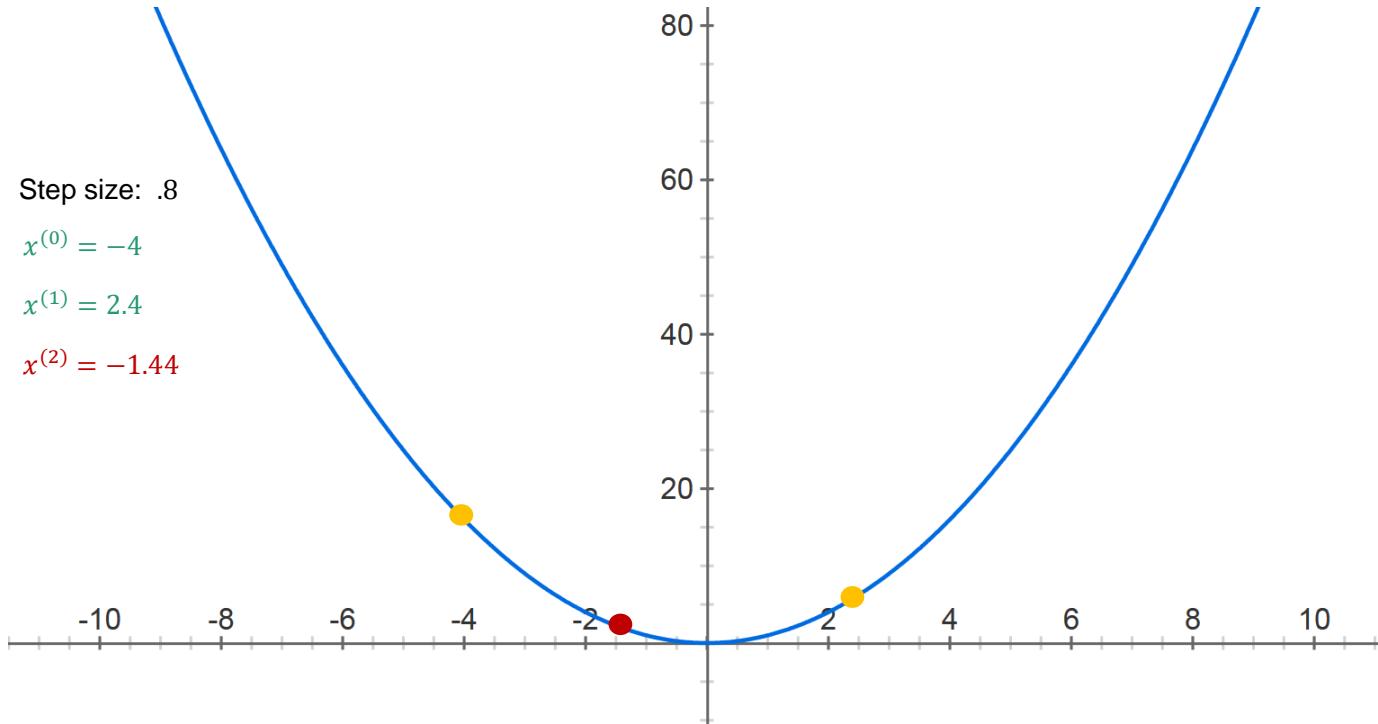
$$f(x) = x^2$$





# Gradient Descent - Example

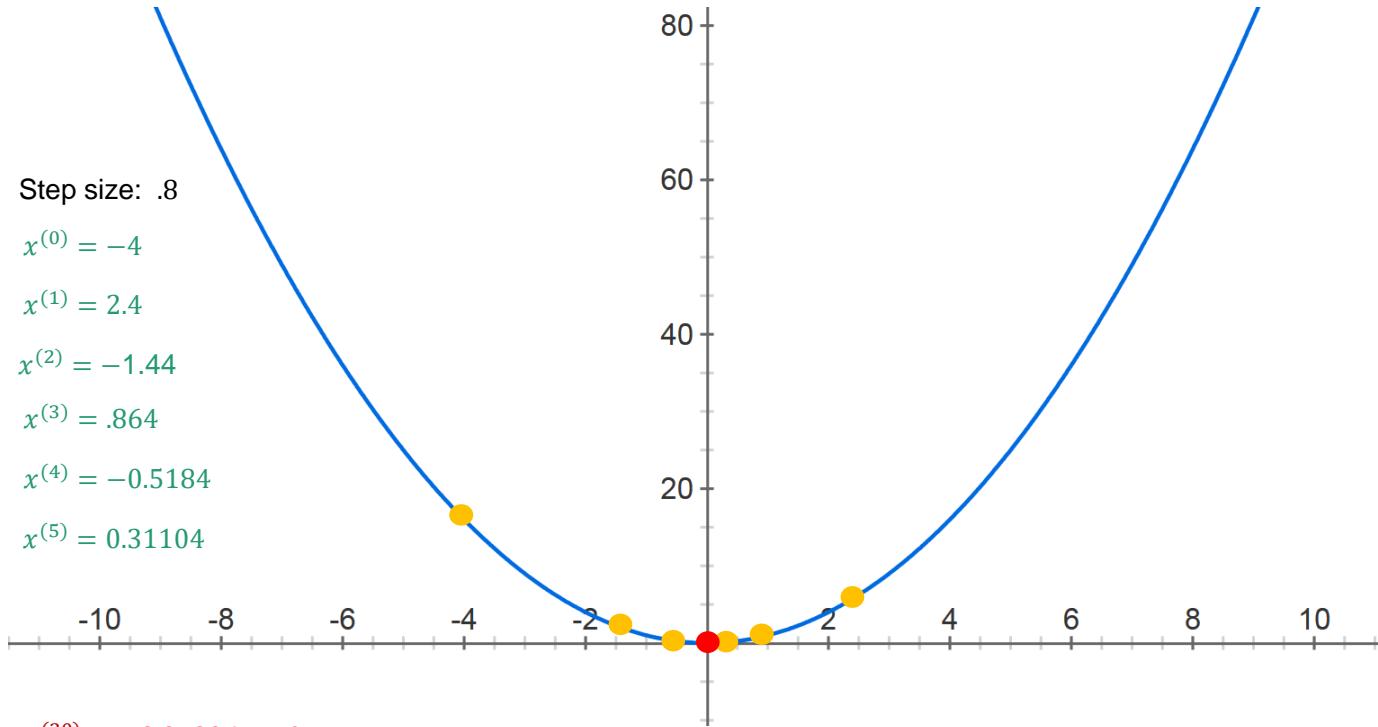
$$f(x) = x^2$$





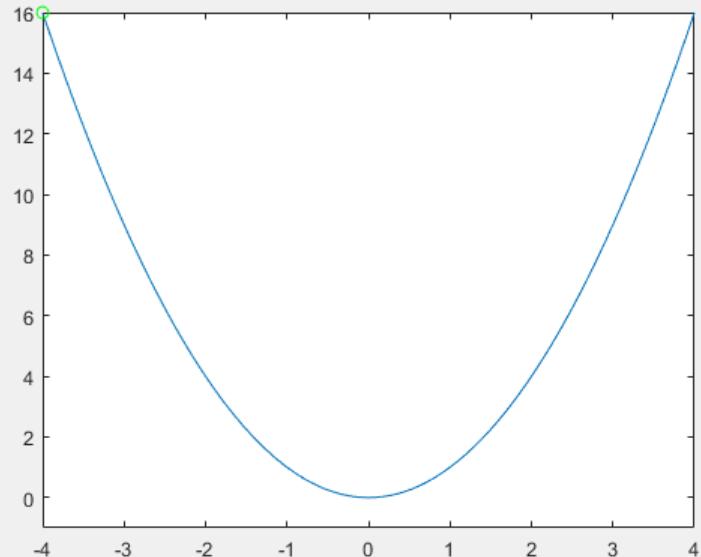
# Gradient Descent - Example

$$f(x) = x^2$$

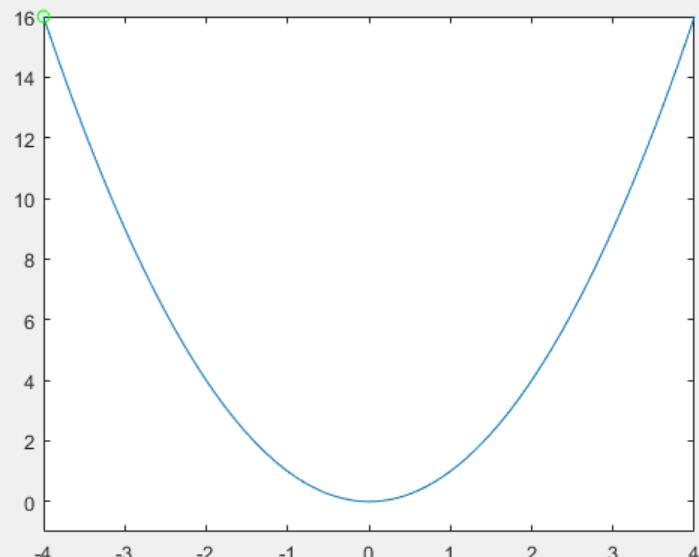




# Gradient Descent - Step Size

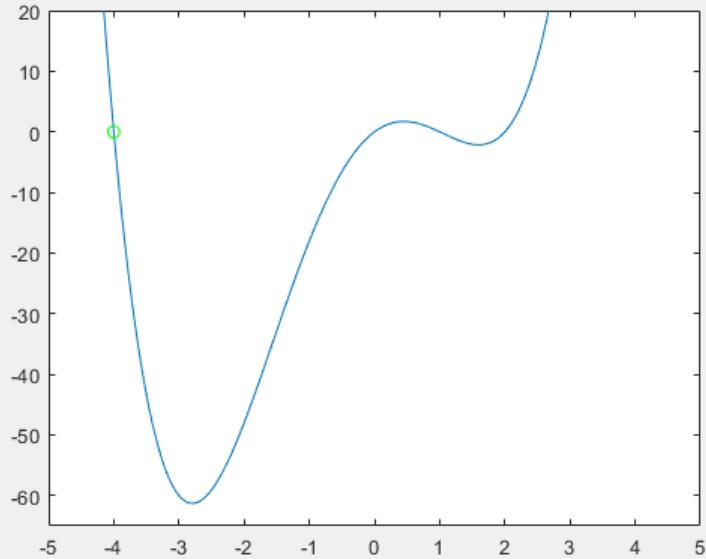
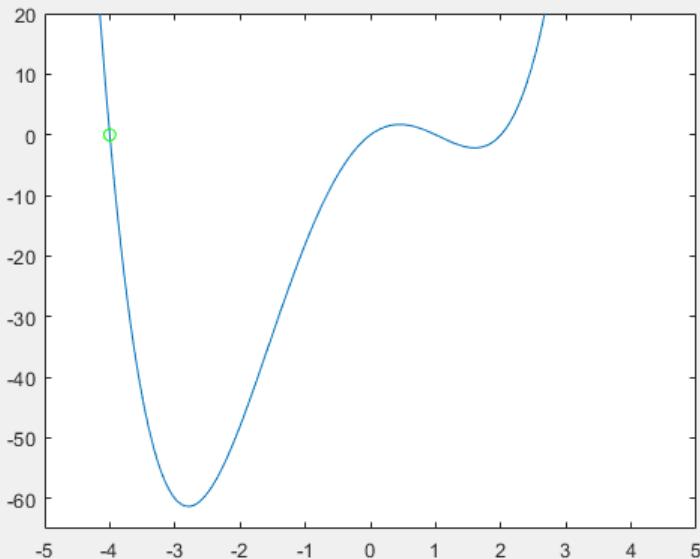


Step size: .9





# Gradient Descent - Step Size



Step size matters!





# Things to remember

- ✓ **Model representation:**

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = \theta^{\top} x$$

- ✓ **Cost function**

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

- ✓ **Gradient descent for linear regression:** *repeat until convergence*

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$



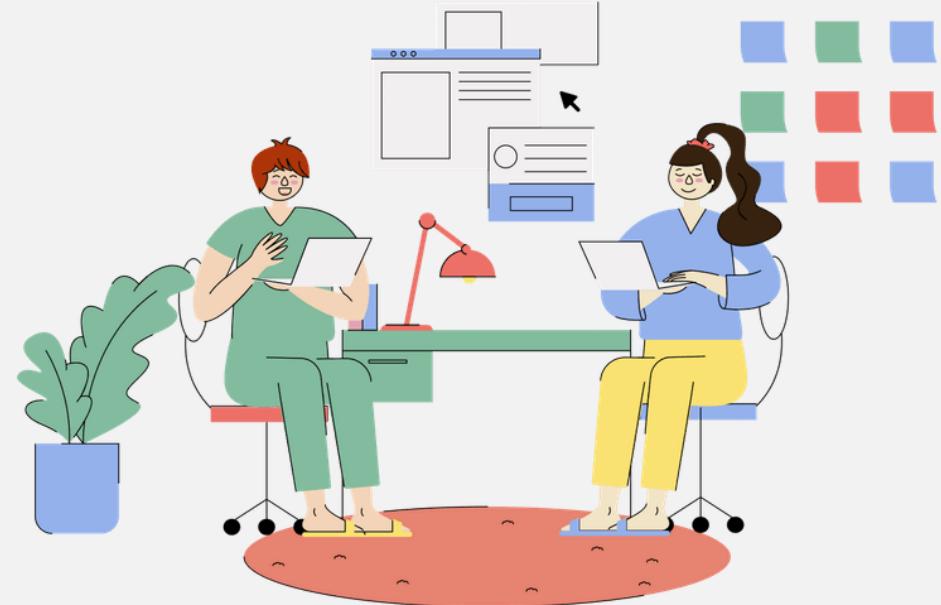


# Exercise

**Transitioning to Google-Colab for hands-on coding practice.**

Notebook:

- T5B3ML101N01\_032024V1 -  
Linear\_Reggression\_with\_Gradient\_Descent





# Batch Gradient Descent

Compute the gradient of the cost function wrt  $\theta_j$  :

- How much the cost function will change if you change  $\theta_j$  just a little bit (partial derivative, slope)

Gradient Descent Step iteratively:

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

```
eta = 0.1 # learning rate
```

```
n_iterations = 1000
```

```
m = 100
```

```
theta = np.random.randn(2,1) # random initialization
```

```
for iteration in range(n_iterations):
```

```
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
```

```
    theta = theta - eta * gradients
```

```
theta
```

```
array([[4.21509616], [2.77011339]])
```

$$\nabla_{\theta} \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\theta - \mathbf{y})$$





# Stochastic (Random) Gradient Descent

```
n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters
def learning_schedule(t):
    return t0 / (t + t1)
theta = np.random.randn(2,1) # random initialization
for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
theta
array([[4.21076011], [2.74856079]])
```

- Batch Gradient Descent is slow with big data.
- Picks a random instance at every step
- Instead of gently decreasing, it will bounce up and down (randomness).

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

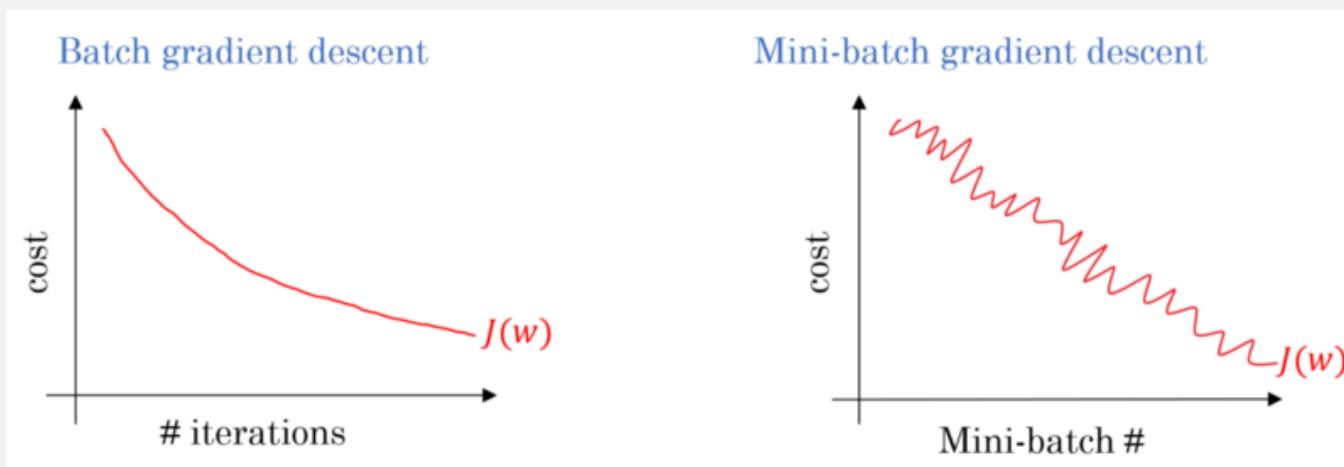




# SGD or BGD: Mini Batch Gradient Descent

Computes the gradients on small random sets of instances called minibatches.

The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.





# Polynomial Regression

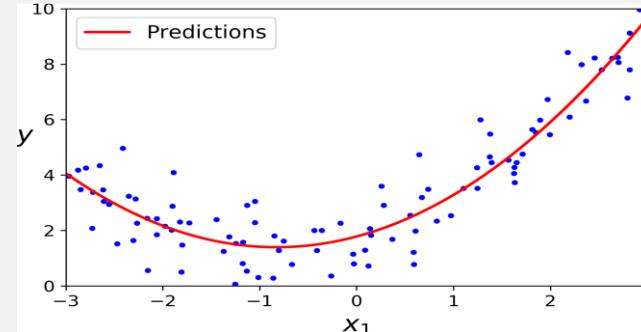
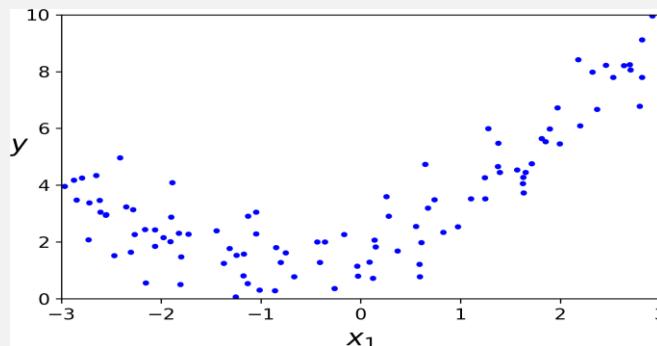
Complex data is not a straight line

- You can actually use a linear model to fit nonlinear data.
- Just add powers of each feature as new features, then train a linear model on this extended set of features

$m = 100$

$X = 6 * np.random.rand(m, 1) - 3$

$y = 0.5 * X^{**2} + X + 2 + np.random.randn(m, 1)$





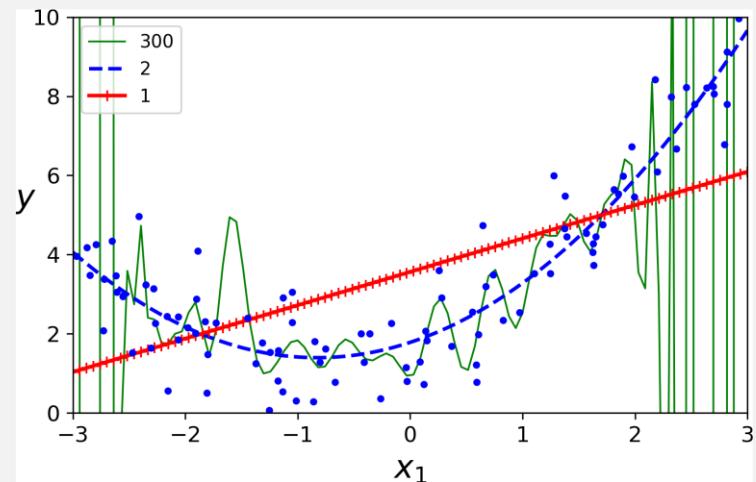
# Polynomial Regression

**Complex data is not linear.**

- We use a linear model to fit nonlinear data.

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0] #→array([-0.75275929])
X_poly[0] #→ array([-0.75275929, 0.56664654])
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

*Estimated:  $\hat{Y} = 0.56 X_1^2 + 0.93 X_1 + 1.78$  when in fact the original  
Original:  $Y = 0.5 X_1^2 + 1.0 X_1 + 2.0 + \text{Gaussian noise.}$*



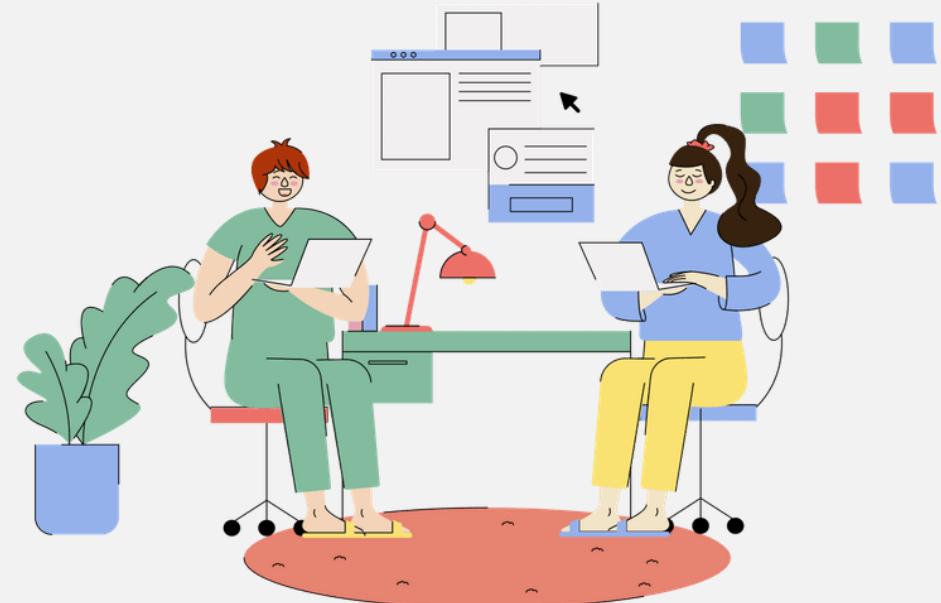


# Exercise

**Transitioning to Google-Colab for hands-on coding practice.**

Notebook:

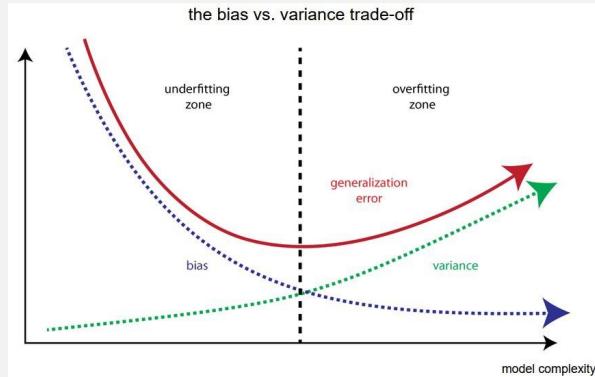
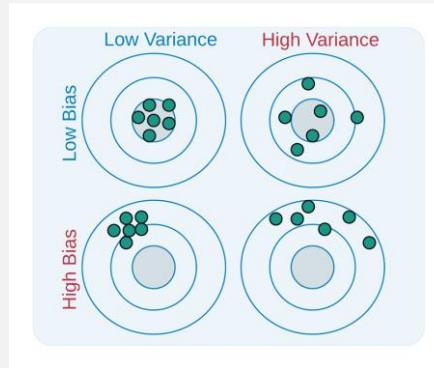
- T5B3ML101N02\_032024V1 -  
Polynomial\_Regression





# Generalization: Bias Variance Trade-off

- **Bias:** Due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data
- **Variance:** Due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance, and thus to overfit the training data.
- **Irreducible** error: Due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).



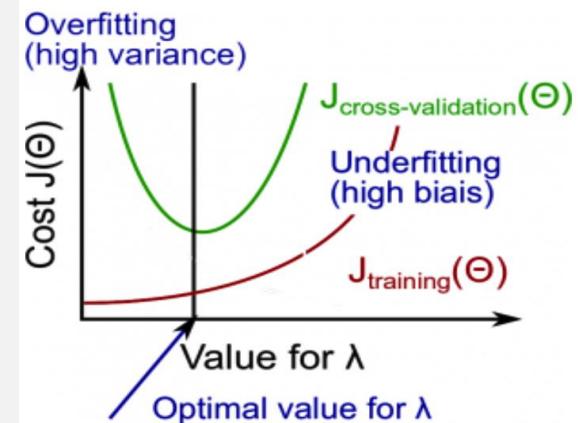


# Regularized Linear Models

To regularize a polynomial model: reduce its degree or constraint the weights (the  $\theta$ s).

- Regularized Linear Regression: Ridge, Lasso and Elastic Net
  - $\text{MSE}(\theta) + \lambda * \text{Regularized Term}$  (training only)
  - $\alpha$  is in fact  $\lambda$  (Hyperparameter: how much you want to regularize the model), 0 means no regularization.
  - Too high value  $\rightarrow$  High bias (flat line), too low  $\rightarrow$  High Variance
- Assume  $\lambda$  ( $\alpha$ ) = 1000 then  $J(\theta) = \text{MSE}(\theta) + 1000 \theta_1 + 1000 \theta_2$
- No Regularization for  $\theta_0$
- **Remember to scale the data**

$$\text{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^m (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)})^2$$





# Regularized Regression: Ridge Regression

## Using RIDGE

```
from sklearn.linear_model import Ridge  
ridge_reg = Ridge(alpha=1, solver="cholesky")  
ridge_reg.fit(X, y)  
ridge_reg.predict([[1.5]])  
array([1.55071465])
```

$$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

## Using SGDRegressor

```
sgd_reg = SGDRegressor(penalty="l2")  
sgd_reg.fit(X, y.ravel())  
sgd_reg.predict([[1.5]])  
array([1.47012588])
```





# Regularized Regression: Lasso Regression

## Using LASSO

```
from sklearn.linear_model import Lasso  
lasso_reg = Lasso(alpha=0.1)  
lasso_reg.fit(X, y)  
lasso_reg.predict([[1.5]])  
array([1.53788174])
```

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^n |\theta_i|$$

## Using SGDRegressor

```
sgd_reg = SGDRegressor(penalty="l1")  
sgd_reg.fit(X, y.ravel())  
sgd_reg.predict([[1.5]])  
array([1.47012588])
```

**Least Absolute Shrinkage and Selection Operator:** eliminate the weights of the least important features so it is considered as a feature selection





# Regularized Regression: Elastic Regression

## ELASTIC NET REGRESSOR

```
from sklearn.linear_model import ElasticNet  
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)  
elastic_net.fit(X, y)  
elastic_net.predict([[1.5]])  
array([1.54333232])
```

## Middle Ground

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^n \theta_i^2$$





# Full Comparison

## RIDGE REGRESSOR

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

## ELASTIC NET REGRESSOR

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2} \alpha \sum_{i=1}^n \theta_i^2$$

## LASSO REGRESSOR

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^n |\theta_i|$$

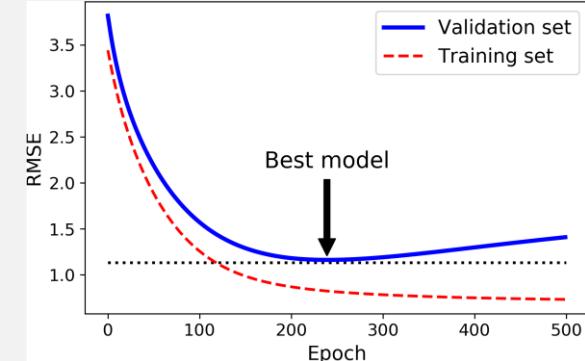


# Regularization Methos

## Early Stopping

- Stops training or learning as soon as the validation error reaches a minimum.

```
from sklearn.base import clone
# prepare the data
poly_scaler = Pipeline([
    ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
    ("std_scaler", StandardScaler())
])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)
sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True, penalty=None,
learning_rate="constant", eta0=0.0005)
minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train) # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```



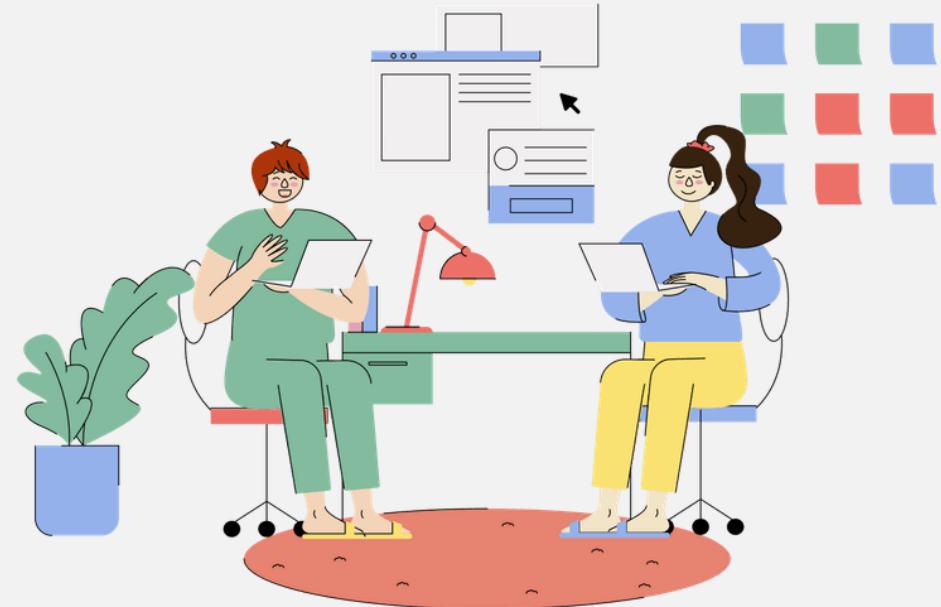


# Exercise

**Transitioning to Google-Colab for hands-on coding practice.**

Notebook:

- T5B3ML101N03\_032024V1 - Regularized\_Linear\_Models

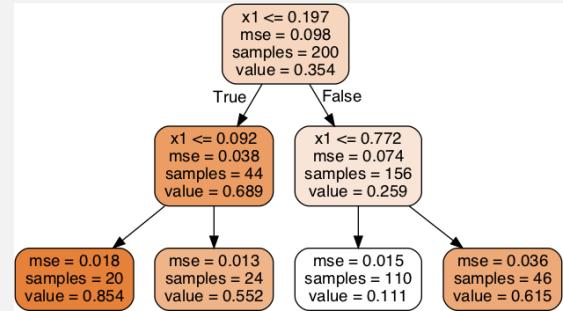
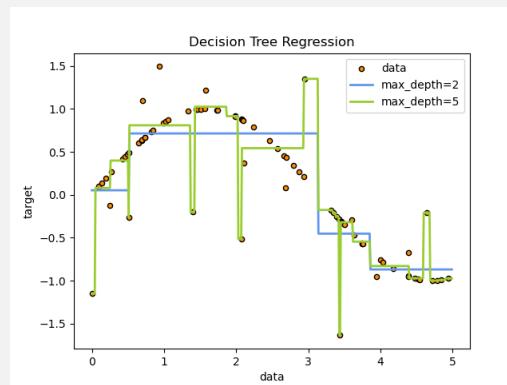
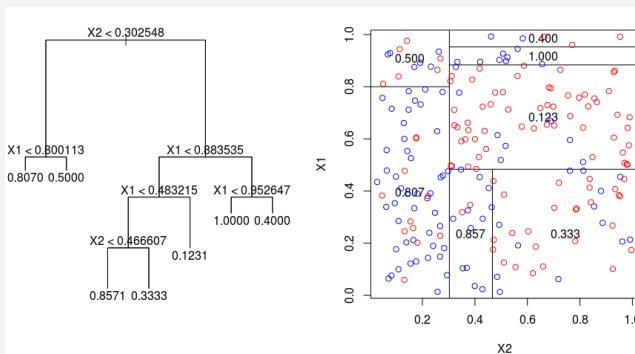




# Tree-Based Models

## Decision Tree Regression Model

```
from sklearn.tree import DecisionTreeRegressor  
  
tree_reg = DecisionTreeRegressor(max_depth=2)  
  
tree_reg.fit(X, y)
```



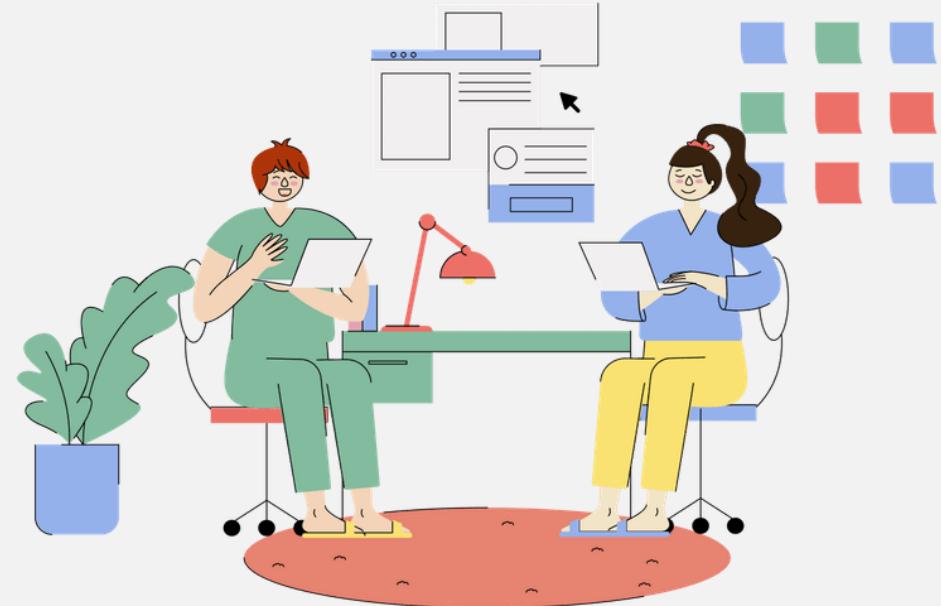


# Exercise

**Transitioning to Google-Colab for hands-on coding practice.**

Notebook:

- T5B3ML101N04\_032024V1 -  
Decision\_Trees\_Regression

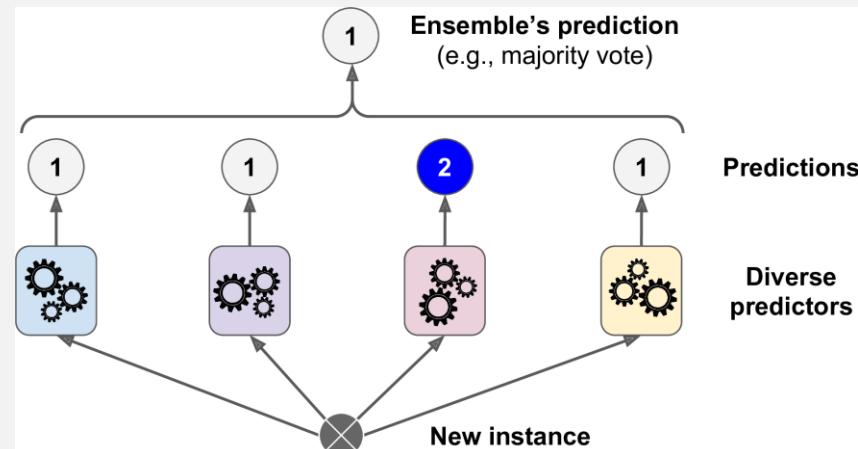
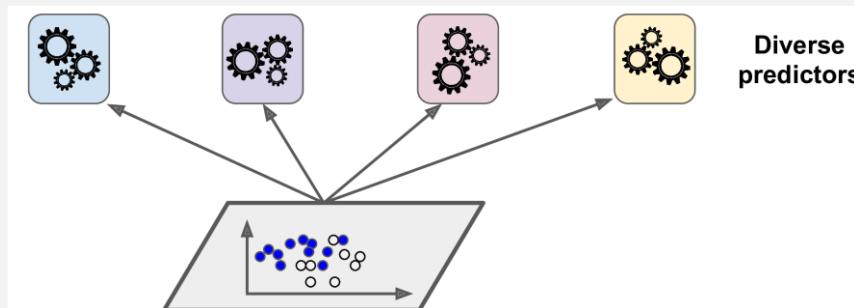




# Ensemble Methods

## Voting Regressor

- To make ensemble methods work best, the predictors should be as independent as possible. Diverse classifiers (different algorithms) make very different types of errors, improving the ensemble's accuracy.





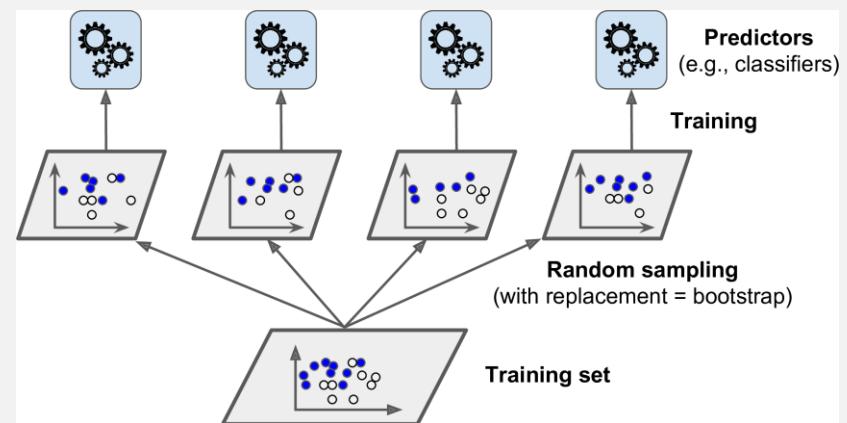
# Ensemble Methods

## Bagging and Pasting (Bootstrap Aggregation)

Diverse Classifiers approach: train the same algorithm but with different random subsets of the training set

- With Replacement vs Without Replacement
- Aggregation reduces Bias and Variance

```
from sklearn.svm import SVR
from sklearn.ensemble import BaggingRegressor
from sklearn.datasets import make_regression
regr = BaggingRegressor(base_estimator=SVR(),
...                   n_estimators=10, random_state=0,
...                   oob_score=True).fit(X, y)
#OR regre.fit(X,y)
regr.predict([[0, 0, 0, 0]])
```

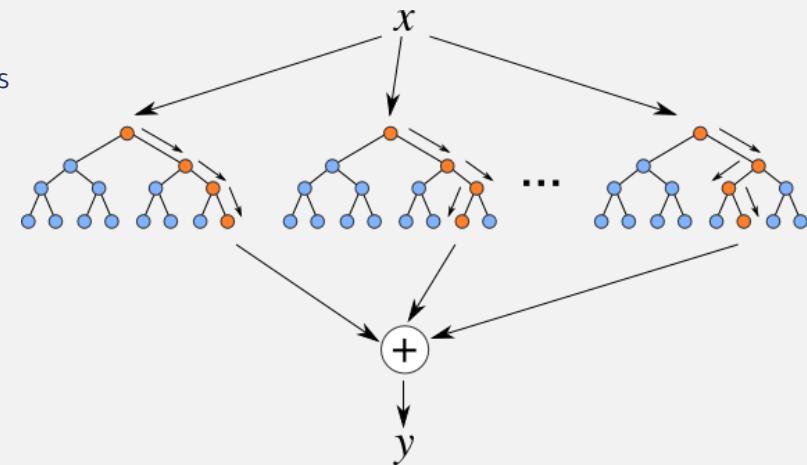




# Random Forest

- Random Forest: ensemble of Decision Trees (Bagging or Pasting, `max_samples`= training set size)
- `BaggingRegressor` → `DecisionTreeRegressor` = `RandomForestRegressor` (highly optimized)
- `RandomForestRegressor` hyperparameters: All from both algorithms

```
from sklearn.ensemble import RandomForestRegressor  
  
from sklearn.datasets import make_regression  
  
X, y = make_regression(n_features=4, n_informative=2, random_state=0,  
shuffle=False)  
  
regr = RandomForestRegressor(max_depth=2, random_state=0)  
  
regr.fit(X, y)  
  
RandomForestRegressor(...)  
  
print(regr.predict([[0, 0, 0, 0]]))  
  
>>>[-8.32987858]
```





# Boosting

- Ensembling with several weak learners into a strong learner
- Each predictor is trying to correct its predecessor
- Base classifier → increase misclassified instances  
(boosting) weights → another classifier...
- Predicted class receives the majority of weighted votes.
- Not parallelizable

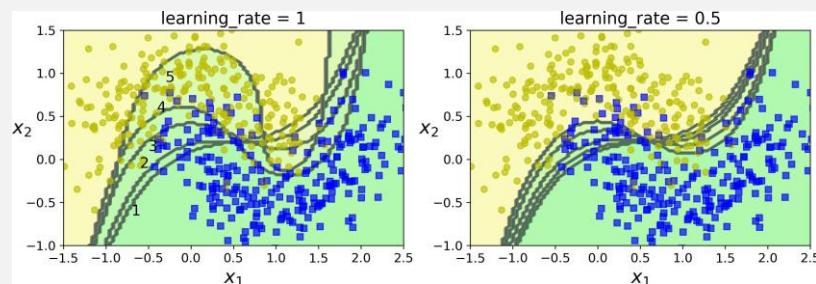
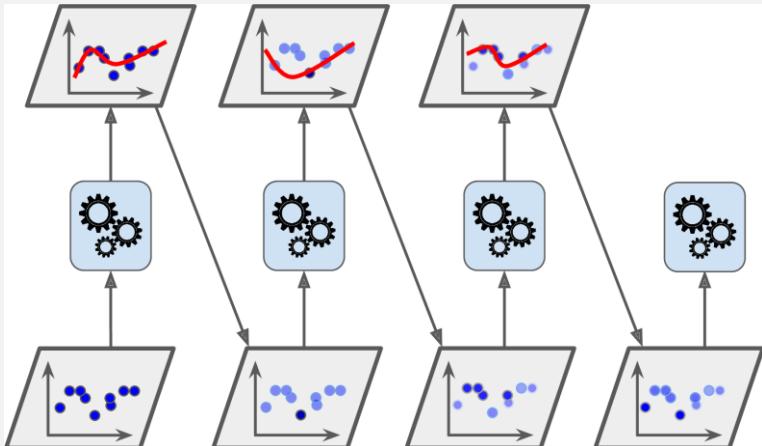
```
from sklearn.ensemble import AdaBoostRegressor #
```

AdaBoostClassifier

```
dtr = DecisionTreeRegressor(max_depth=3)
```

```
adareg = AdaBoostRegressor(base_estimator=dtr, learning_rate=1.0,  
loss='linear', n_estimators=100, random_state=None)
```

```
adareg.fit(X_train, y_train)
```





# Gradient Boosting

Boosting fits the predictor on the residual errors made by previous predictor.

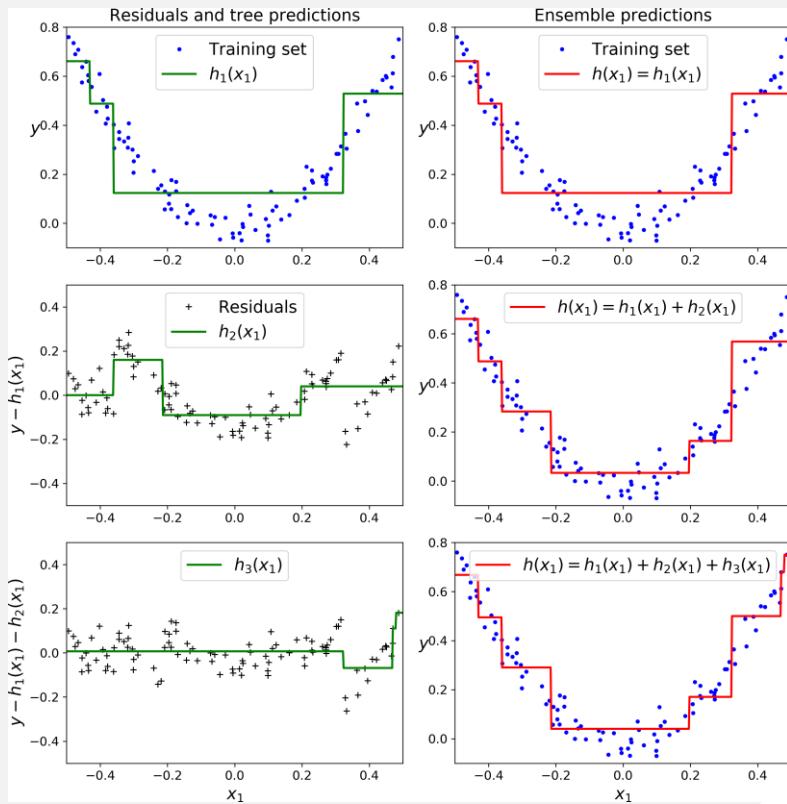
```
from sklearn.tree import DecisionTreeRegressor
tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)

y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)

y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)

y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
OR

from sklearn.ensemble import GradientBoostingRegressor
gbdt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbdt.fit(X, y)
```



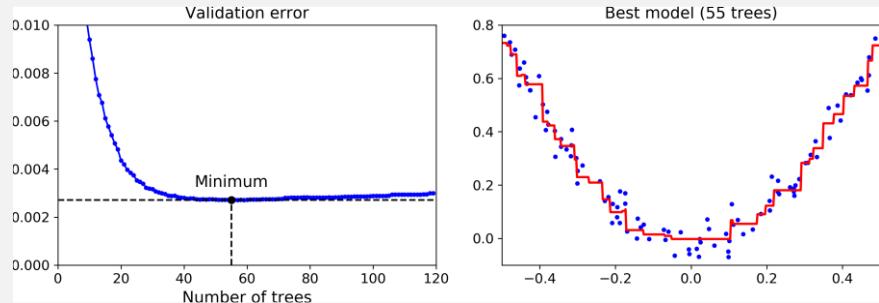
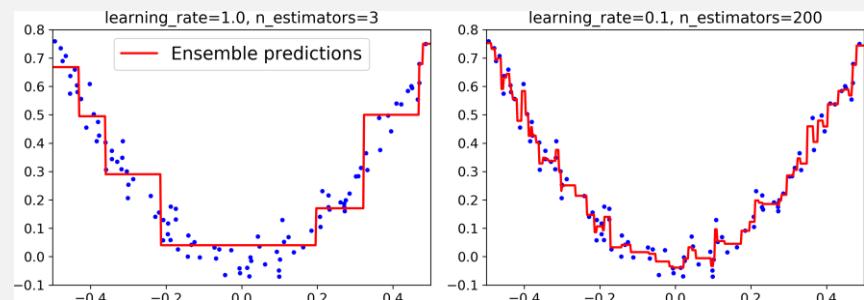


# Gradient Boosting

- Learning\_rate : scales each tree's contribution. (low (0.1), you need more trees to fit the training set, but it will generalize better. (Early Stopping (warm\_start=True))
- Extreme GradientBoosting XGBoost : optimized implementation. ML competition

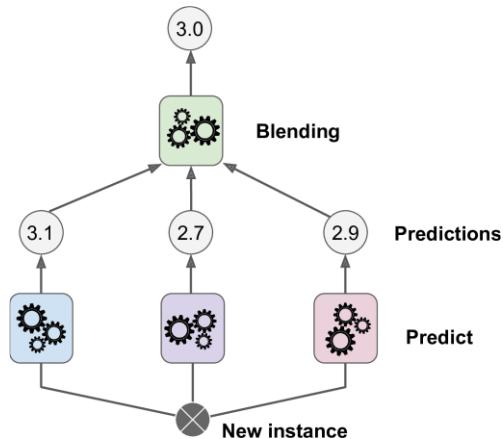
```
import xgboost
```

```
xgb_reg = xgboost.XGBRegressor()  
xgb_reg.fit(X_train, y_train)  
  
y_pred = xgb_reg.predict(X_val)  
  
xgb_reg.fit(X_train, y_train, eval_set=[(X_val, y_val)],  
early_stopping_rounds=2)  
  
y_pred = xgb_reg.predict(X_val)
```

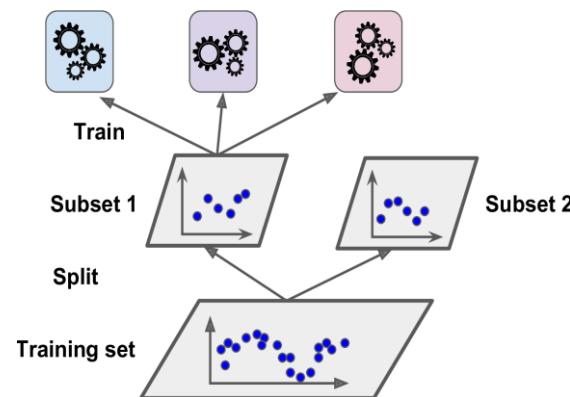




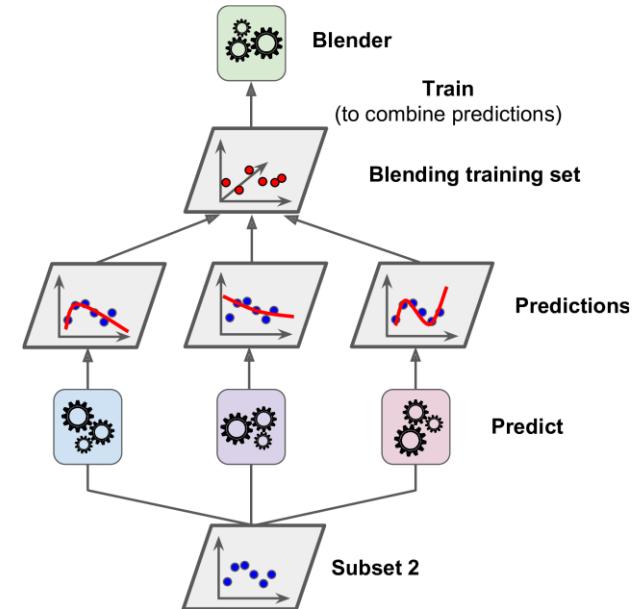
# Model Stacking



Stacked  
Generalization



Instead of using trivial function  
(hard voting: avg,...), let's train a  
model to learn this function



Blender or Meta Learner



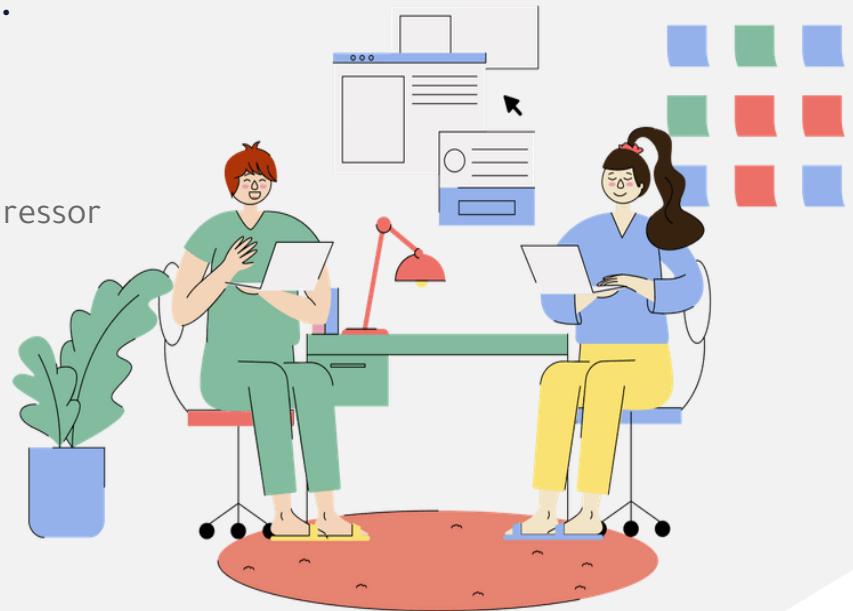


# Exercise

**Transitioning to Google-Colab for hands-on coding practice.**

Notebook:

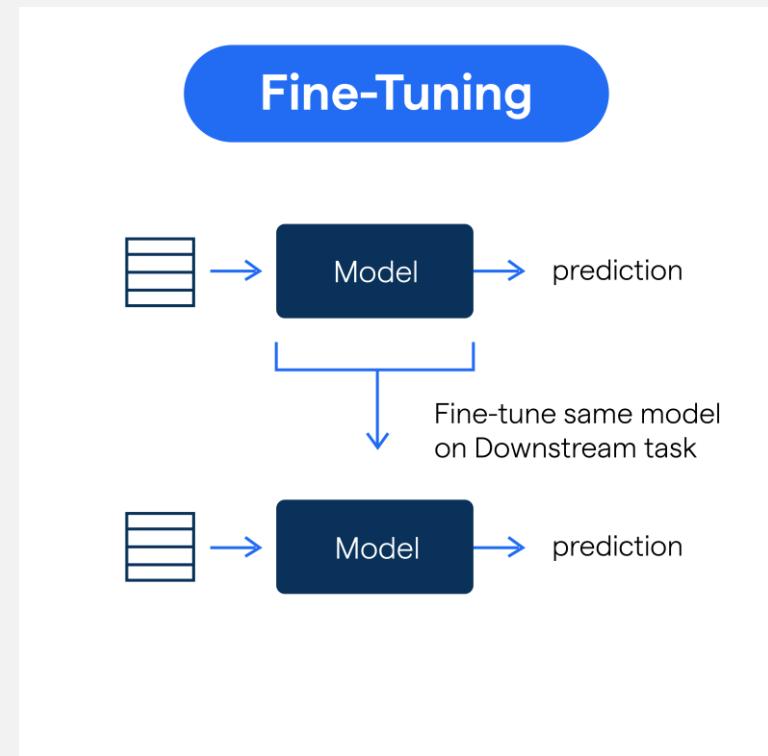
- T5B3ML101N05\_032024V1 - Ensemble\_Learning\_GB\_Regressor
- T5B3ML101N06\_032024V1 - SVM\_Regression





# Model Fine-Tuning

- Across various ML techniques, the adjustment of trainable parameters is essential for learning.
- Each ML algorithm possesses hyperparameters, which dictate how this adjustment is executed.
- The setting of these hyperparameters significantly influences the learning process and ultimately determines the accuracy of the model.
- This process of setting hyperparameters is known as **model fine-tuning** or simply model tuning.





# Model Fine-Tuning

In a machine learning model, there are 2 types of parameters:

Aspect	Model Hyperparameters	Model Parameters
Definition	Settings or configurations are set before training, controlling the learning process.	Variables learned during training, define the mapping from input to output.
Example	shrinkage factor in Ridge Regression ( $\lambda$ )	Co-efficient and slope in Regression ( $\beta$ )
Tuning	Set manually or through optimization techniques such as grid search or random search.	Automatically learned from training data through optimization algorithms like gradient descent.
Role	Influence the behavior and performance of the learning algorithm.	Determine the specific function that the model represents and its ability to make predictions.





# Model Fine-Tuning

## Hyperparameters Optimization Techniques

Common algorithms include:

### Grid Search

- You define a **parameter grid**, where we include a set of possible hyperparameter values used to build the model.
- The hyperparameters are placed into a matrix-like structure, and the model is trained on every combination of hyperparameter values.
- The model with the best performance is then selected.

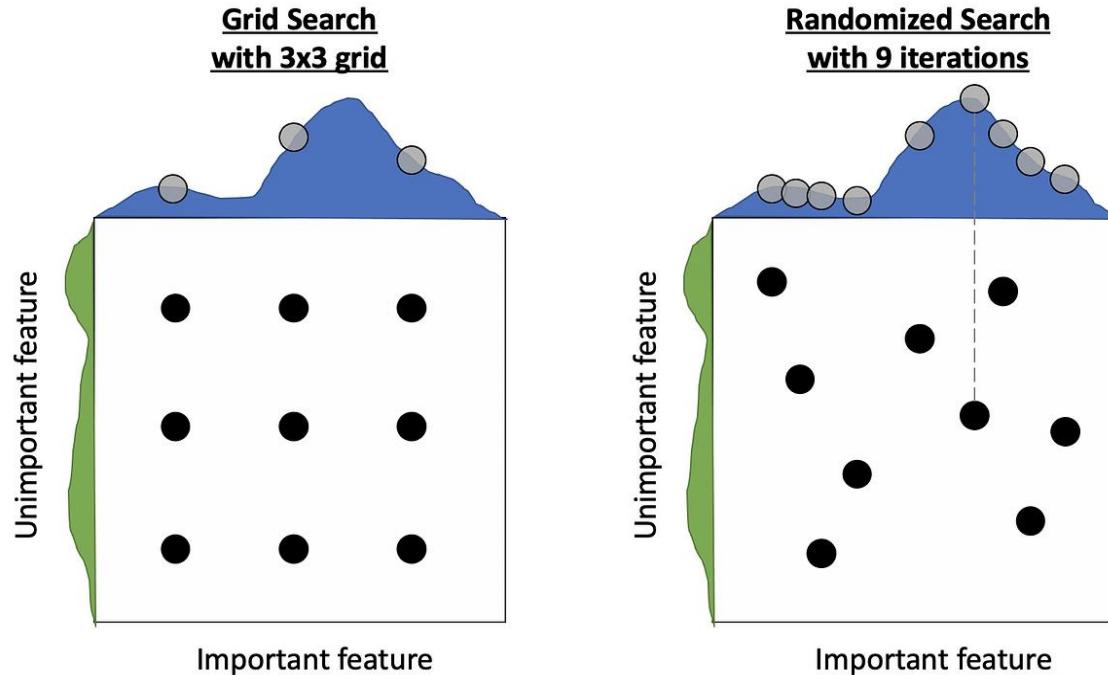
### Random Search

- Random search only selects and tests a **random** combination of hyperparameters.
- This technique randomly samples from a grid of hyperparameters instead of conducting an exhaustive search.
- We can specify the number of total runs the random search should try before returning the best model.





# Model Fine-Tuning



# Model Fine-Tuning

```
# Create a RandomForestClassifier model

rf = RandomForestRegressor()

# Define hyperparameters to be tuned

hyperparameters = { 'n_estimators': [10, 50, 100, 200], 'max_depth': [None, 10, 20, 30],
                    'min_samples_split': [2, 5, 10]}

# Create the GridSearchCV object

grid_search = GridSearchCV(estimator=rf, param_grid=hyperparameters, cv=5, n_jobs=-1,
                           verbose=1)

# Fit the model on the training set

grid_search.fit(X_train, y_train)

# Fit the model on the training set

grid_search.fit(X_train, y_train)

# Get the best hyperparameters found by GridSearchCV

best_params = grid_search.best_params_

print("Best hyperparameters found by GridSearchCV:", best_params)

# Evaluate the model on the test set

test_score = grid_search.score(X_test, y_test)

print("Test set accuracy with best hyperparameters:", test_score)
```



# Model Fine-Tuning

```
# Create the RandomizedSearchCV object

random_search = RandomizedSearchCV(estimator=rf,
param_distributions=hyperparameters, n_iter=10, cv=5, n_jobs=-1, verbose=1, random_state=42)

# Fit the model on the training set

random_search.fit(X_train, y_train)

# Get the best hyperparameters found by RandomizedSearchCV

best_params = random_search.best_params_
print("Best hyperparameters found by RandomizedSearchCV:", best_params)

# Evaluate the model on the test set

test_score = random_search.score(X_test, y_test)
print("Test set R^2 score with best hyperparameters:", test_score)
```



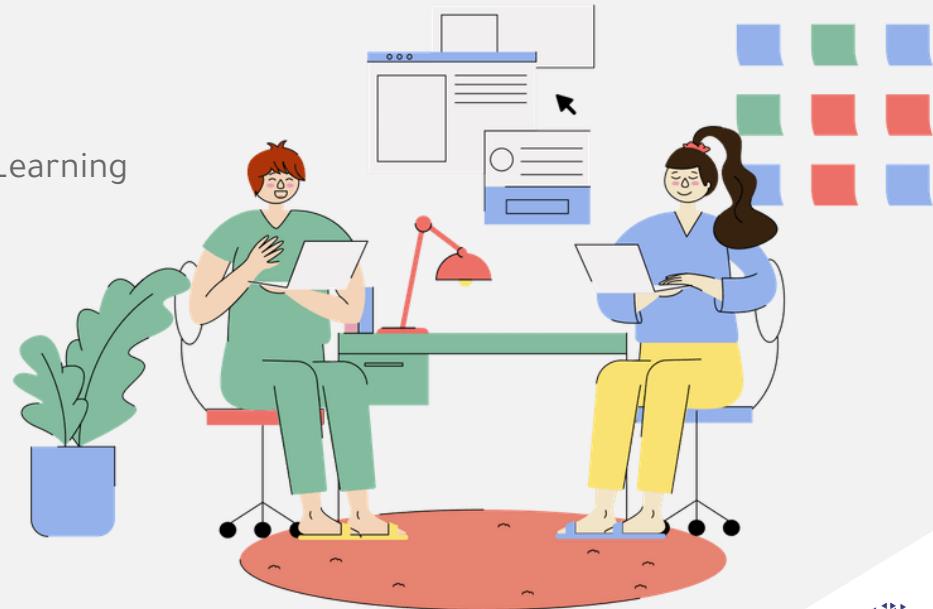


# Exercise

**Transitioning to Google-Colab for hands-on coding practice.**

Notebook:

- T5B3ML101N07\_032024V1-End\_to\_End\_Machine\_Learning



# Thank You