Supervised Learning I- Regression

Zeham Management Technologies BootCamp
by SDAIA
July 30th, 2024

Agenda

Regression:

- Types: Univariate, Multivariate, Linear, Non-Linear, Polynomial
- Generalization
- Cost functions
- Gradient Descent (SGD, Mini batch)
- Overfitting & Regularization (Lasso, Ridge, Elastic)
- Performance Metrics: (MSE, RMSE, MAE, R2)
- Feature Engineering & Feature Selection
- Hyperparameter tuning (Grid Search)
- Standardization & Normalization

Generalization

Generalization

How can you tell if a machine learning model is truly learning something valuable? It’s not as straightforward as it might seem, so it's important to delve deeper. These models usually learn by calculating derivatives in relation to a loss function and then adjusting their parameters step-by-step in the correct direction. This process of statistical learning is quite different from how humans learn. Although some algorithms are inspired by the human brain, they function in a very different manner. It's important to understand this distinction because the term "learn" can misleadingly suggest that these models understand the data like humans do – but they don’t. They only learn statistical patterns, no more and no less.

Generalization

The term 'generalization' refers to the model's ability to adapt and respond appropriately to previously unseen, new data drawn from the same distribution as that used to build the model. In other words, generalization investigates a model's ability to digest new data and make accurate predictions after being trained on a training dataset.

Generalization

Extrapolation

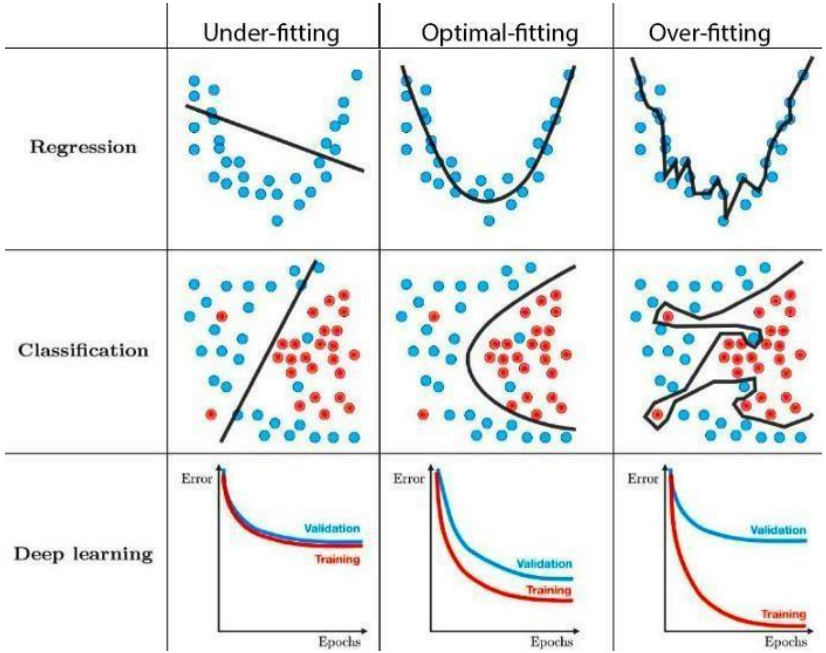
Extrapolation is the process of using a model to predict outcomes for cases where the predictor values are outside the range of the values used to train the model. This involves extending the observed trend in the known data to estimate values beyond that range. For example, if a model is built to predict blood pressure based on the weight of individuals between 150 and 200 pounds, making predictions for individuals weighing less than 150 pounds or more than 200 pounds would be extrapolation. Such predictions can be inaccurate because the model hasn't learned the relationship for those weights.

Interpolation

Interpolation involves predicting values for cases where the predictor values fall within the range of observed values used to train the model. For example, using the same blood pressure model to predict values for individuals weighing between 160 and 180 pounds is considered interpolation. Interpolation is generally reliable as it stays within the trained range.

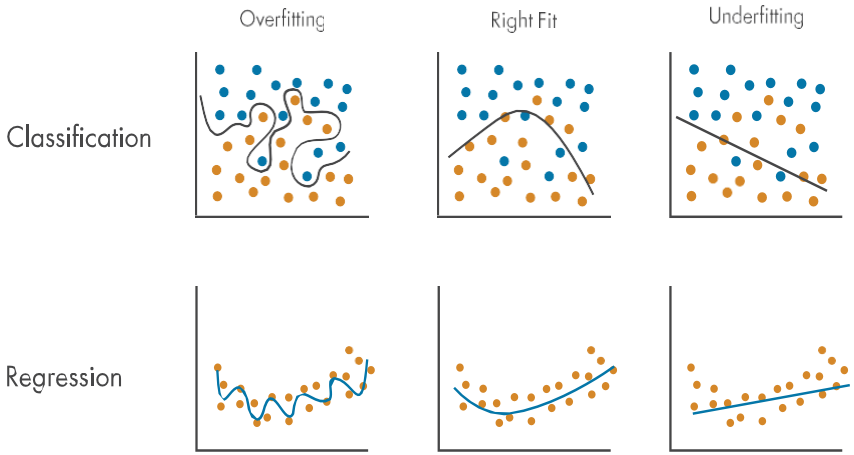
Overfitting vs Underfitting

Overfitting happens when a model becomes overly complex, essentially memorizing the training data rather than identifying useful patterns. This leads to excellent performance on the training set but poor generalization to new data. In contrast, underfitting occurs when a model is too simple to capture the data's underlying patterns, resulting in subpar performance on both the training and new data.



Preventing Overfitting

- **Simplify the Model:**
- Use fewer hidden layers and fewer units per layer to reduce complexity.
- **Early Stopping:**
- Begin with small weights and halt training before the model starts overfitting.
- **Regularization:**
- Apply techniques like L1 or L2 regularization to penalize large weights and encourage simplicity.



Preventing Overfitting

- **Add Noise:**

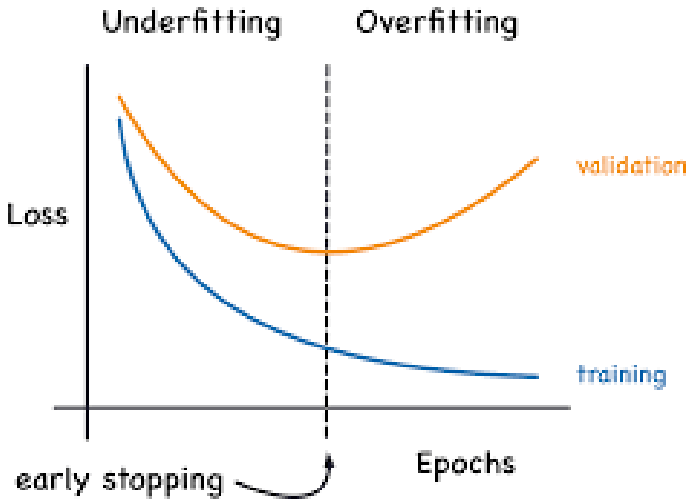
Introduce noise to the weights or activities during training to make the model more robust.

- **Gather More Data:**

Increasing the size of your training dataset can help the model generalize better.

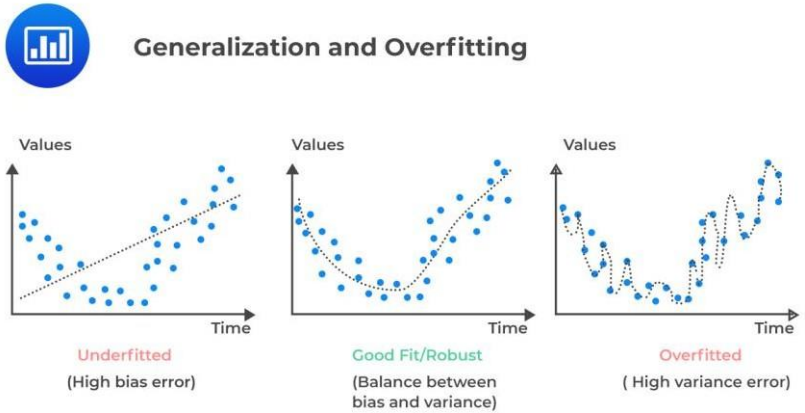
- **Cross-Validation:**

Use cross-validation to ensure your model performs well on different subsets of the data, providing a check against overfitting.



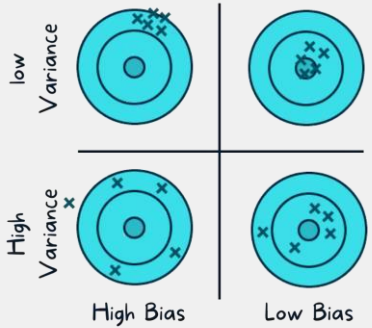
Avoiding Underfitting

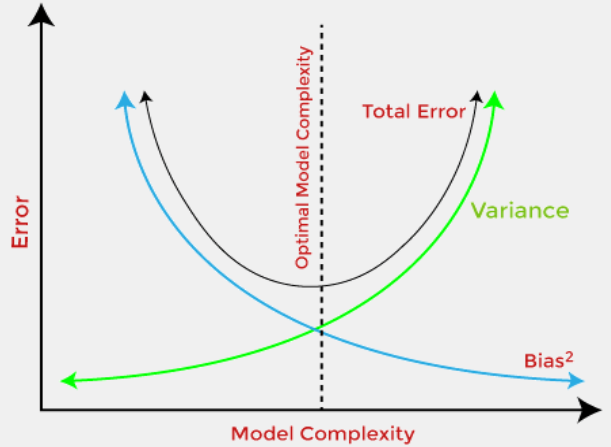
- Decrease Regularization:
- Over-regularization can cause features to become too uniform, which can prevent the model from capturing the dominant trends. To introduce more complexity and variation, reduce the amount of regularization.
- Increase Training Duration:
- Training the model for a too short period of time can lead to underfitting. To allow the model to learn more effectively, extend the training duration.
- Enhance Feature Selection:
- If the model lacks sufficient predictive features, consider adding more relevant features or increasing the importance of existing ones.



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).





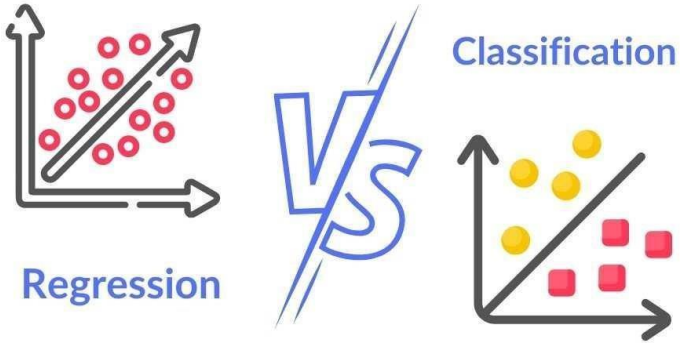
Supervised Learning 1: Regression

Supervised Learning

Supervised Learning:

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

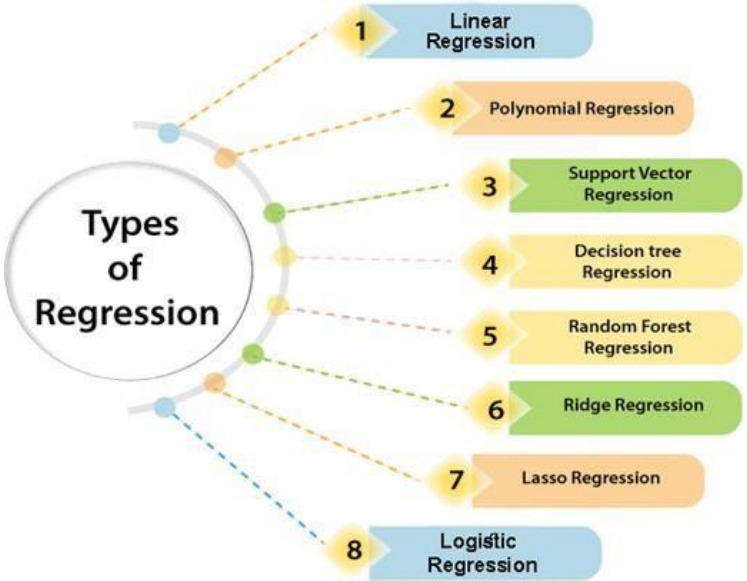
- Tasks : Classification, Regression
- Algorithms: K-Nearest Neighbors, Linear Regression, Logistic Regression, Support Vector Machines (SVMs), Decision Trees and Random Forests, Neural Networks



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).

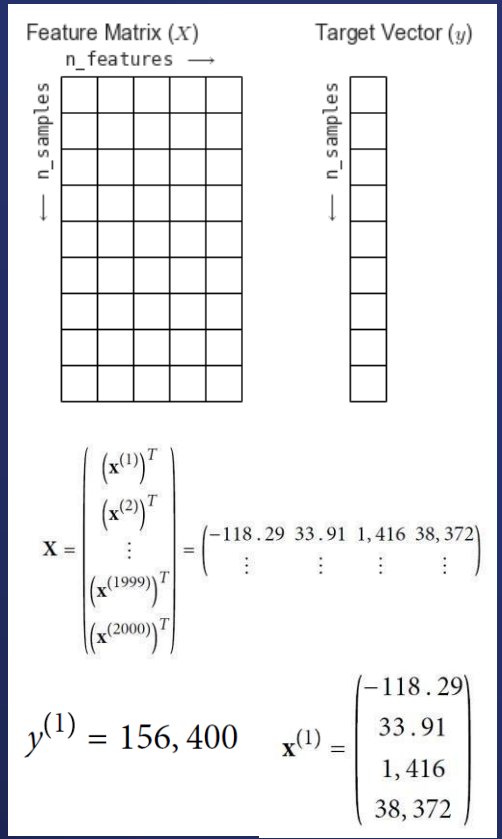
- Linear regression
- Univariate Regression
- Multiple Regression
- Multivariate Regression
- Polynomial Regression



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 ith 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



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 $y^{(i)}$ is 360,000 SR

Area (m^2)	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

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

What do you think the values for θ_0 ,θ_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"

$$h(x) = \hat{Y} = \theta_0 + \theta_1 X_1 + \theta_2 X_1 X_2 + \theta_3 X_1^2 + \theta_4 X_2^2 \ldots.$$ "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 $\|.\|_2$ or just $\|.\|$*

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

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

- **Mean Absolute Error (MAE):** Manhattan norm, L1norm noted $\|.\|1$

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

- **R-Squared:**

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

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

Example

Simple Real Estate

Area (m^2)	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

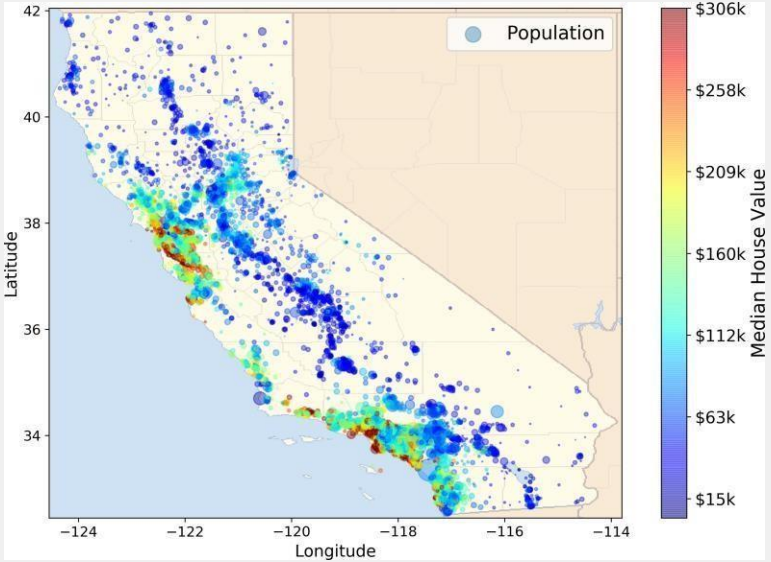
Area (m^2)	Price (y)	Mean ($\bar{y}$)	Prediction ($\hat{y}$)	RMSE	MAE	RSS	RTT	R2
100	360,000	504,000	504,000	20,736,000,000	144,000	20,736,000,000	20,736,000,000	
140	450,000	504,000	504,000	2,916,000,000	54,000	2,916,000,000	2,916,000,000	
170	570,000	504,000	504,000	4,356,000,000	66,000	4,356,000,000	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

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

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?

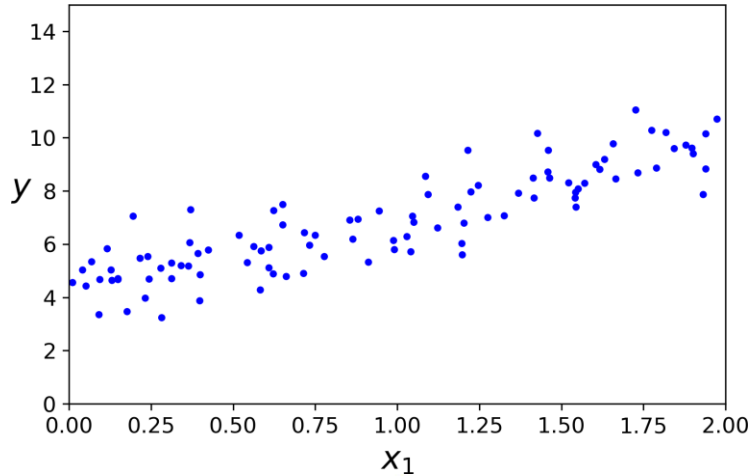
The term "train" in machine learning refers to the process of teaching a model using input-output pairs or examples, adjusting its parameters to minimize errors between its predictions and the correct outputs.

Cost function:

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

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

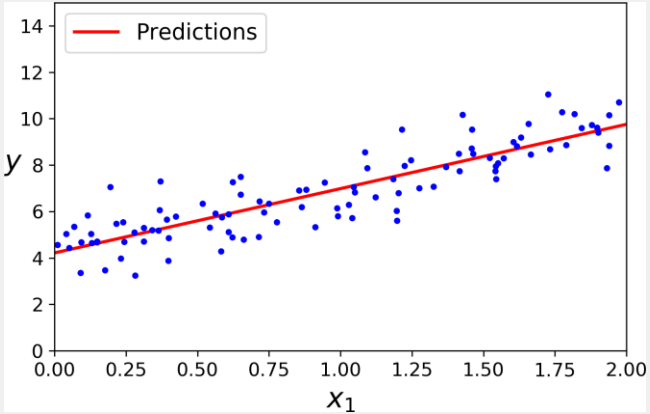
```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```



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

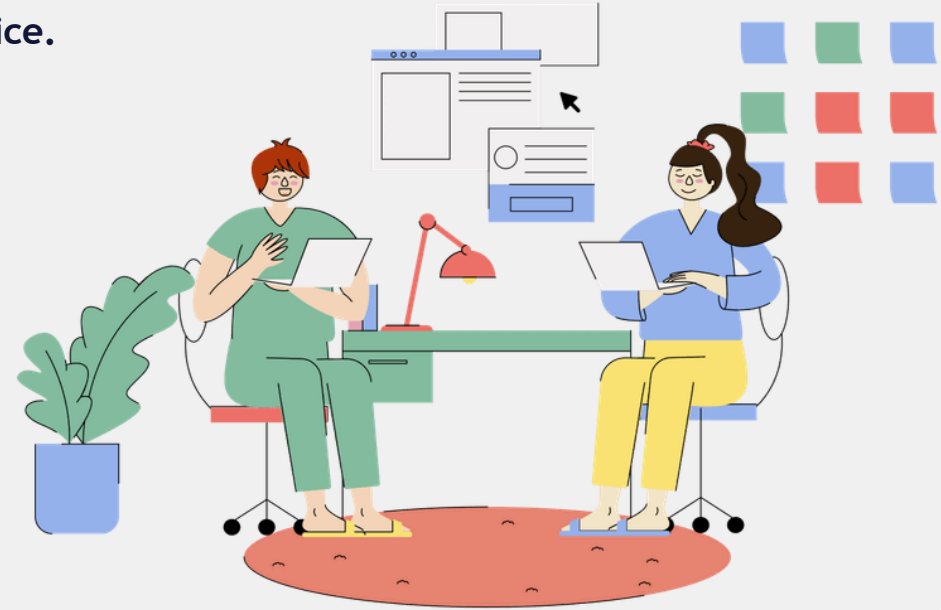
How do we train a model?

```
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.

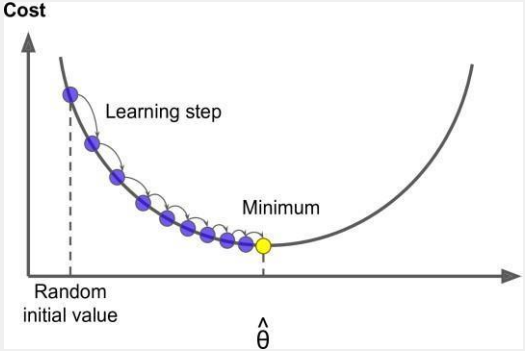


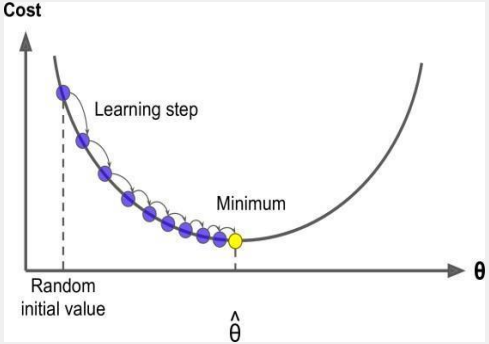
Notebook:

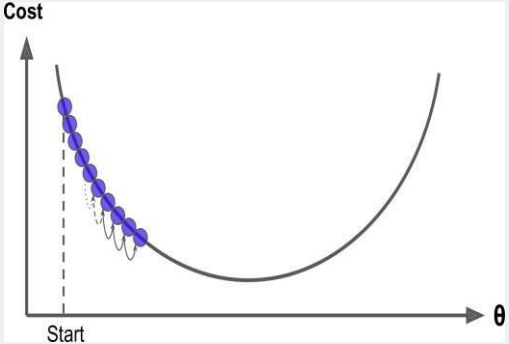
3- Supervised Learning I - Regression/
LAB/T5B3ML101N00_032024V1 - Linear_Reggression

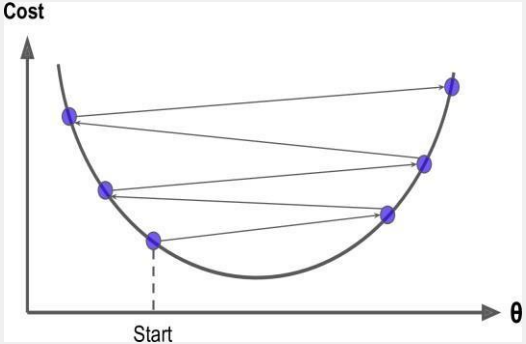
How do we train the model?

- Randomly initialized parameters (Large Loss)
- How gradient descent is doing it?
- Learning rate (α)
- Model Hyperparameter (α) vs Model Parameters (Θ)





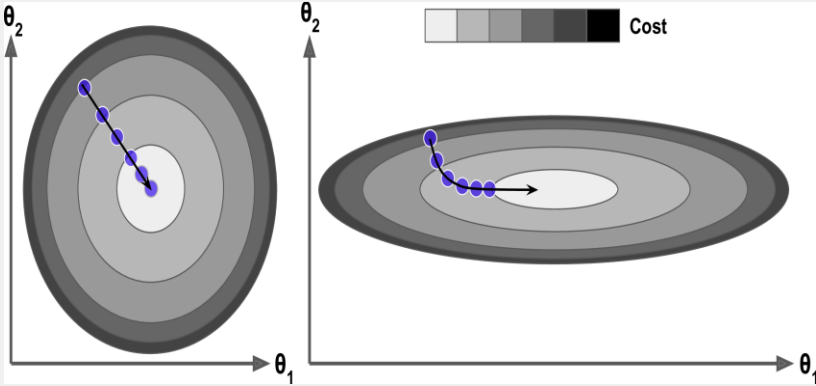




Feature Scaling (Normalization)

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.

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

Standard Scaler:

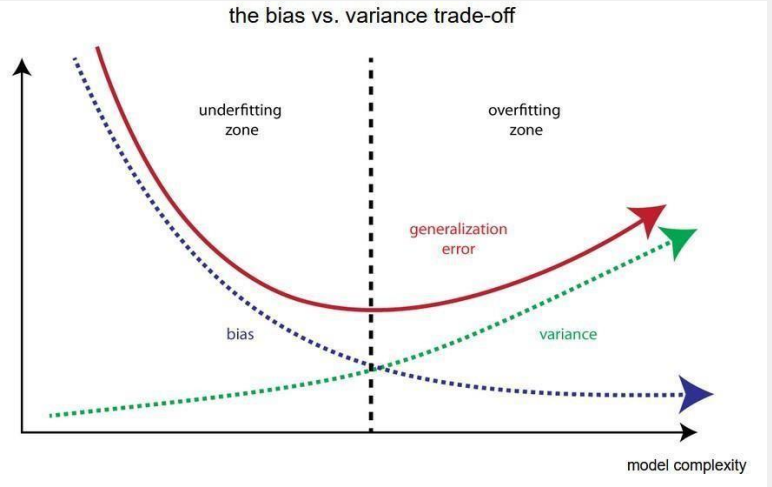
- Transforms features to have a mean of 0 and a standard deviation of 1, ensuring each feature has a similar scale.
- Robust to outliers and suitable for algorithms relying on normal distribution assumptions, like linear regression and logistic regression.

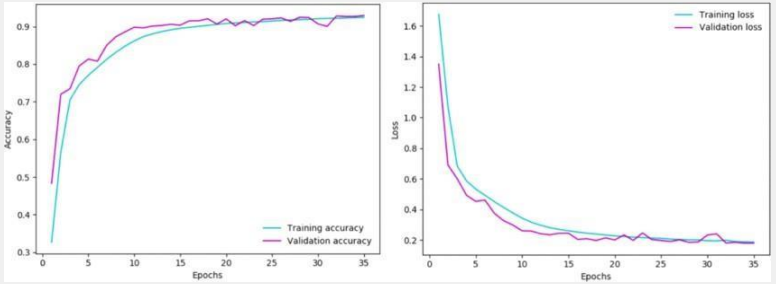
$$\frac{x_i - \text{mean}(\boldsymbol{x})}{\text{stdev}(\boldsymbol{x})}$$

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 (η): waste time
  - ✓ Too high (η): far away from the optimal solution

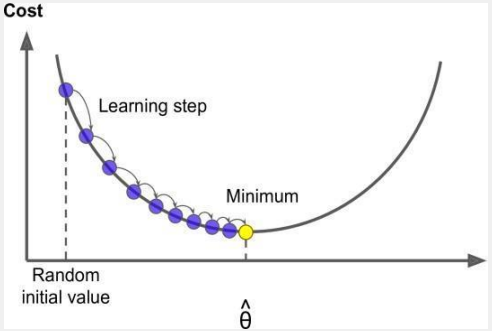




Gradient Descent

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

- Start by random values for θ (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 paramter:
  - ✓ Too low (η): waste time
  - ✓ Too high (η): 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 ϵ (tolerance).



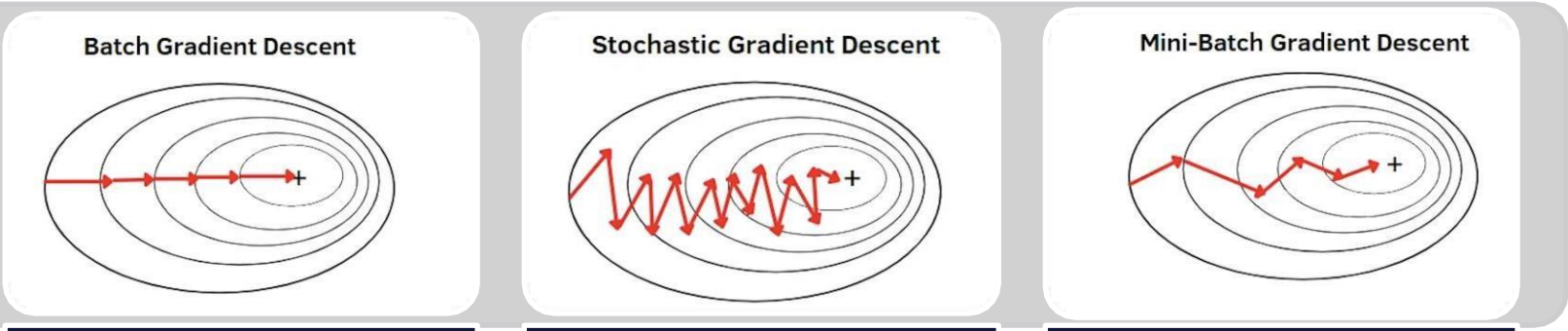
Gradient Descent

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.



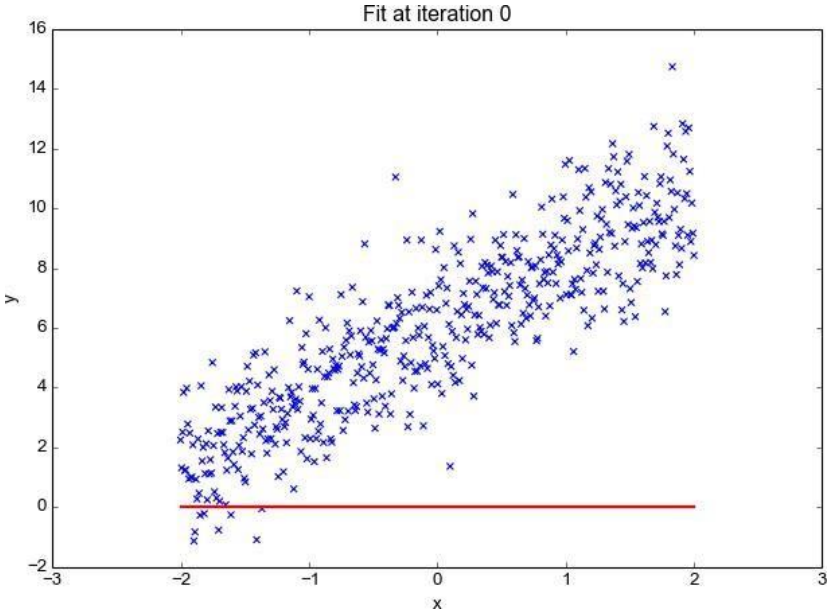
Batch gradient descent looks at the entire hill to find the steepest direction.

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 θ 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 (θ_j). For linear regression, the partial derivative with respect to (θ_j). is given by:

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

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

Update Rule:

The gradient descent update rule for each parameter (θ_j) is as follows:

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

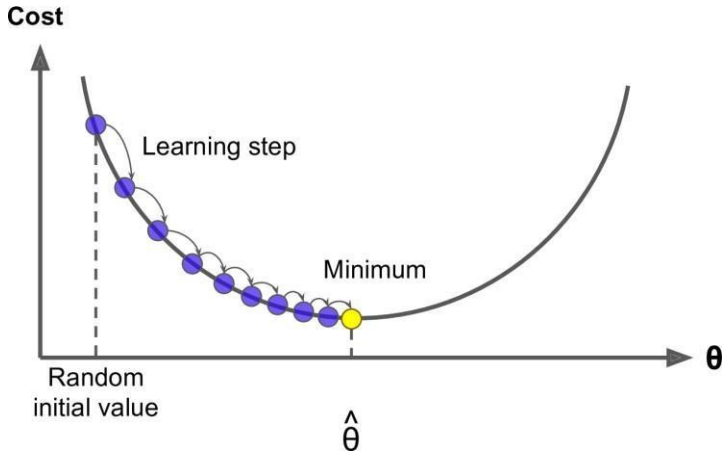
where; α is the learning rate, a hyperparameter that controls the size of the steps taken during each iteration.

Iterative Process:

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

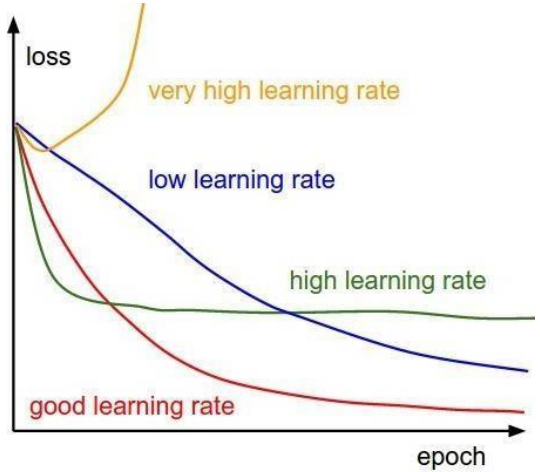
Learning Rate

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 α may cause slow convergence.
- ✓ A large learning rate α 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 $x^{(1)} = x^{(0)} - \alpha . \nabla f(x^{(0)})$, where $\nabla f(x^{(0)})$ is the gradient of f at $x^{(0)}$, we can follow these steps:

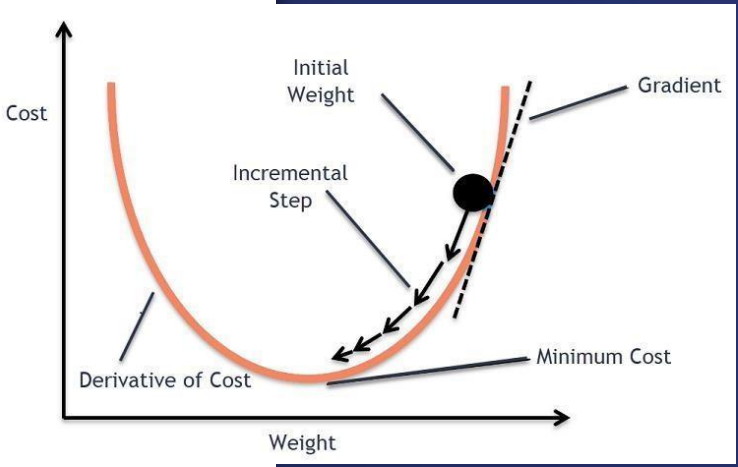
1. Define the function $f(x)$:

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

2. Initialize parameters:

$$\alpha = 0.8$$

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



Gradient Descent - Example

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

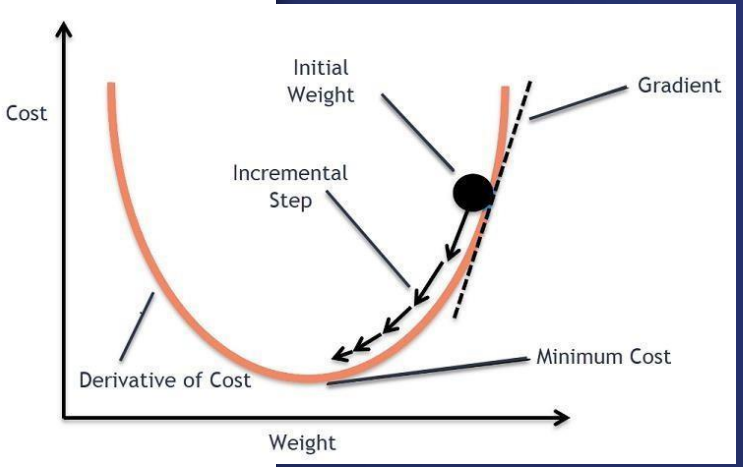
$$f'(x) = 2x$$

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

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:

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

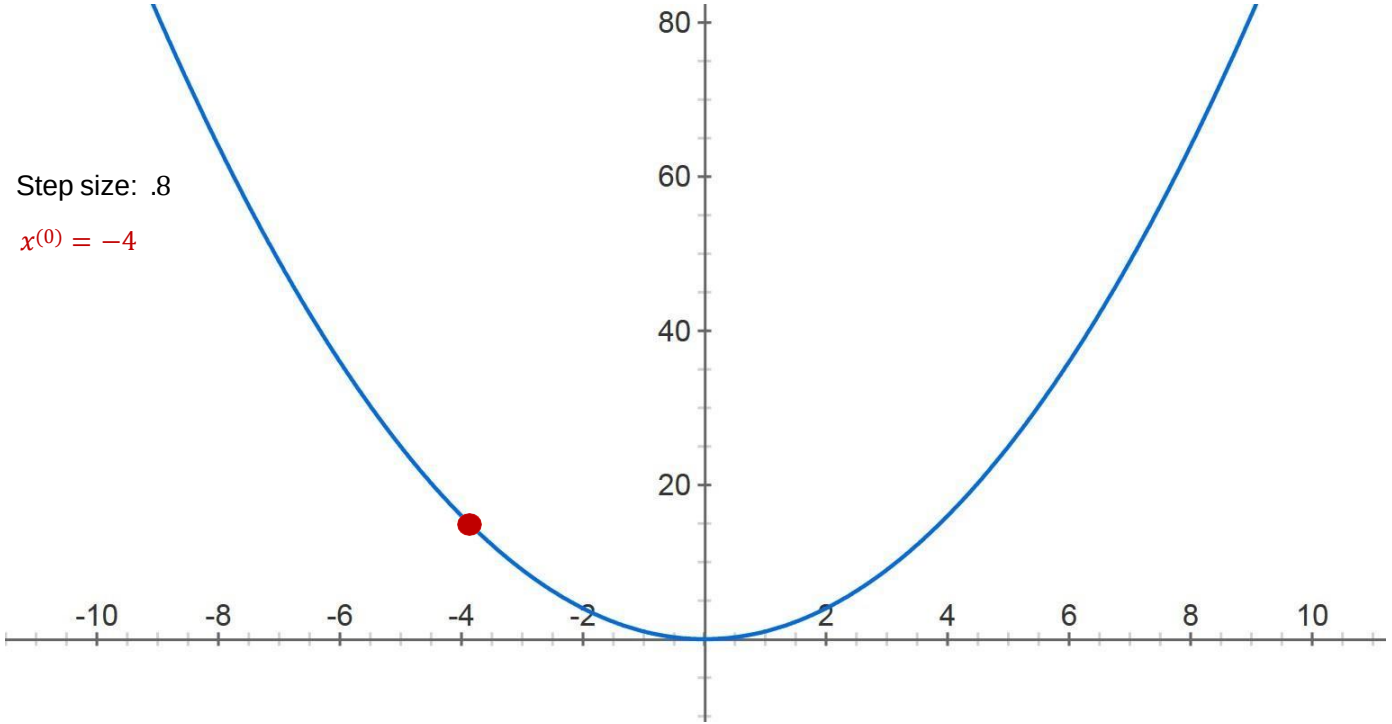
So, after one iteration of gradient descent, starting from $x^{(0)} = -4$ with a step size of 0.8, we obtain $x^{(1)} = 2.4$

Gradient Descent - Example

$f(x) = x^2$

Step size: .8

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



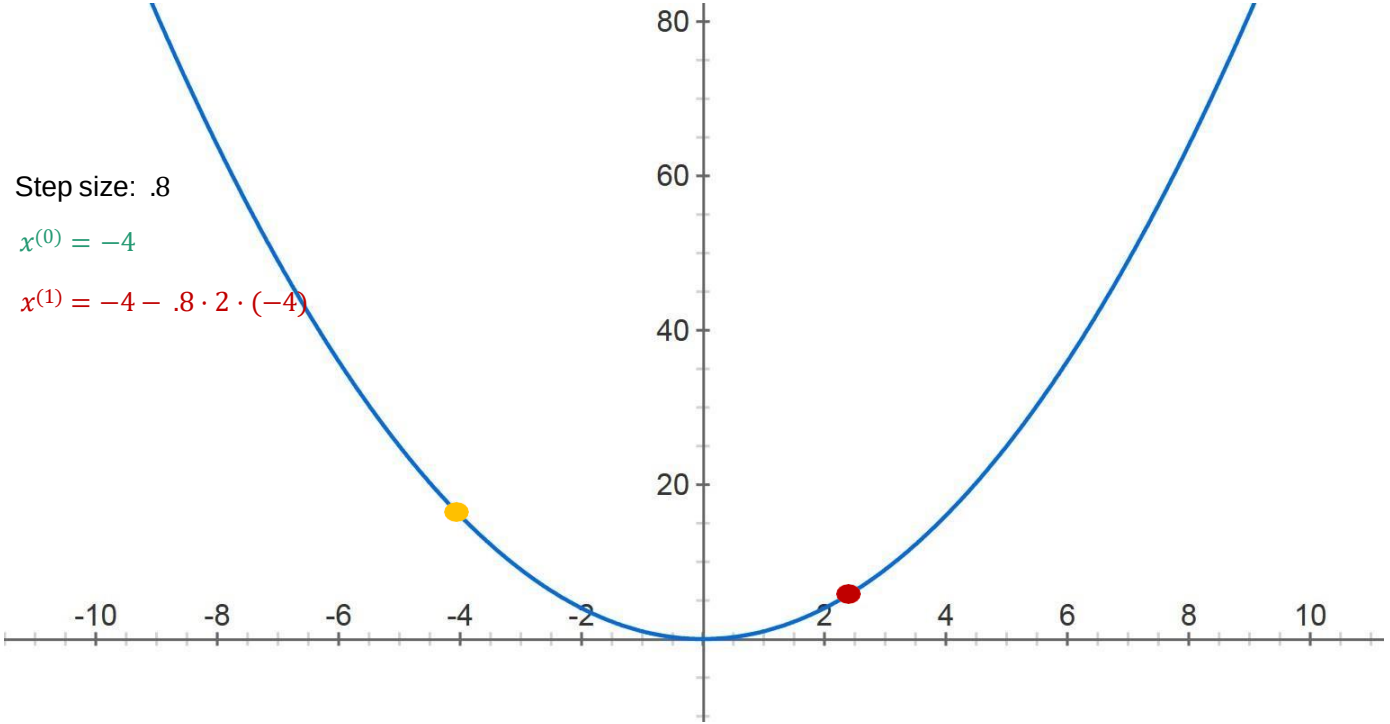
Gradient Descent - Example

$f(x) = x^2$

Step size: .8

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

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



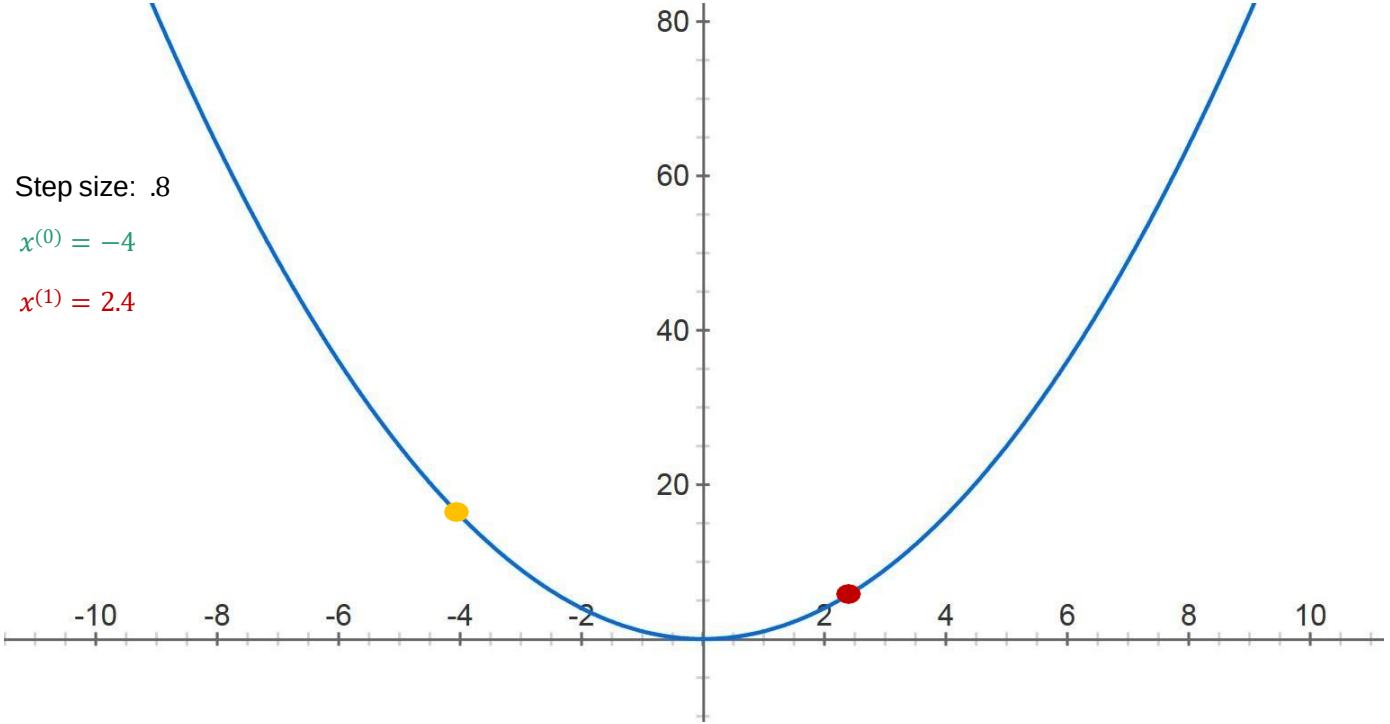
Gradient Descent - Example

$f(x) = x^2$

Step size: .8

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

$x^{(1)} = 2.4$



Gradient Descent - Example

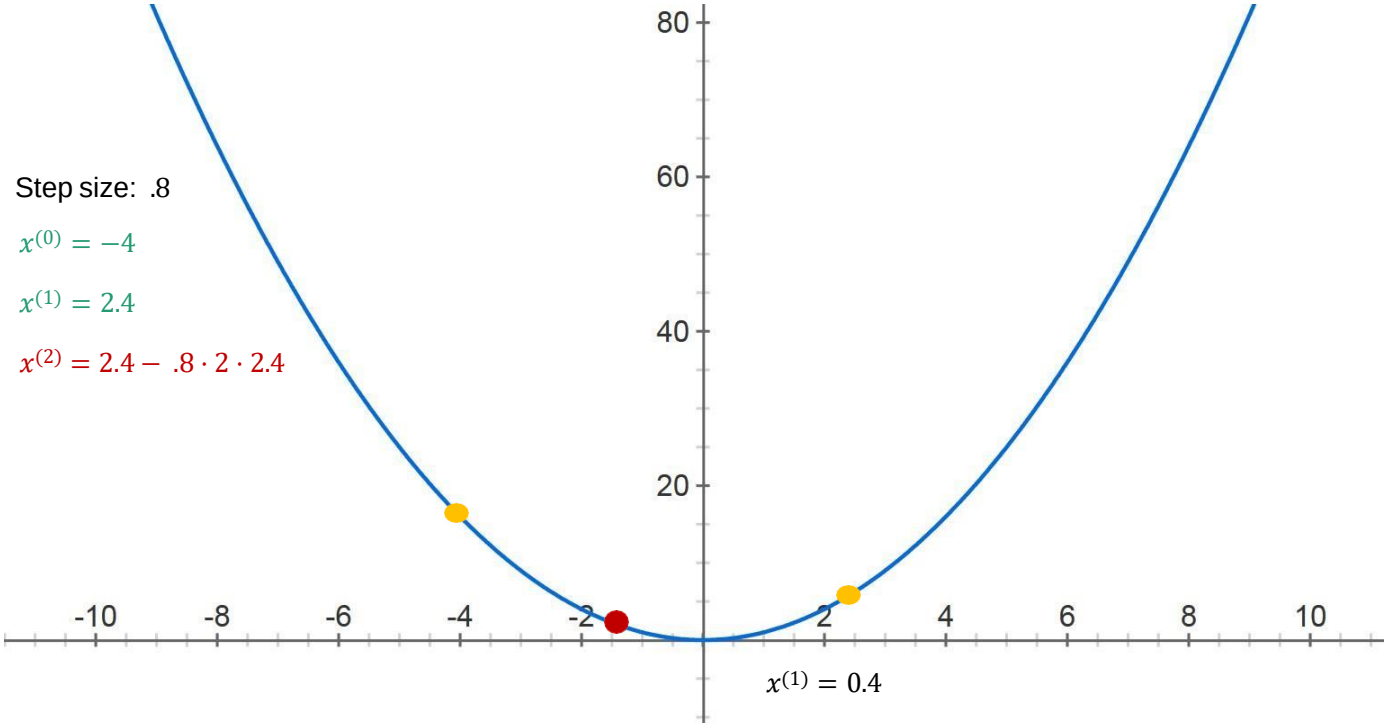
$f(x) = x^2$

Step size: .8

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

$x^{(1)} = 2.4$

$x^{(2)} = 2.4 - .8 \cdot 2 \cdot 2.4$



Gradient Descent - Example

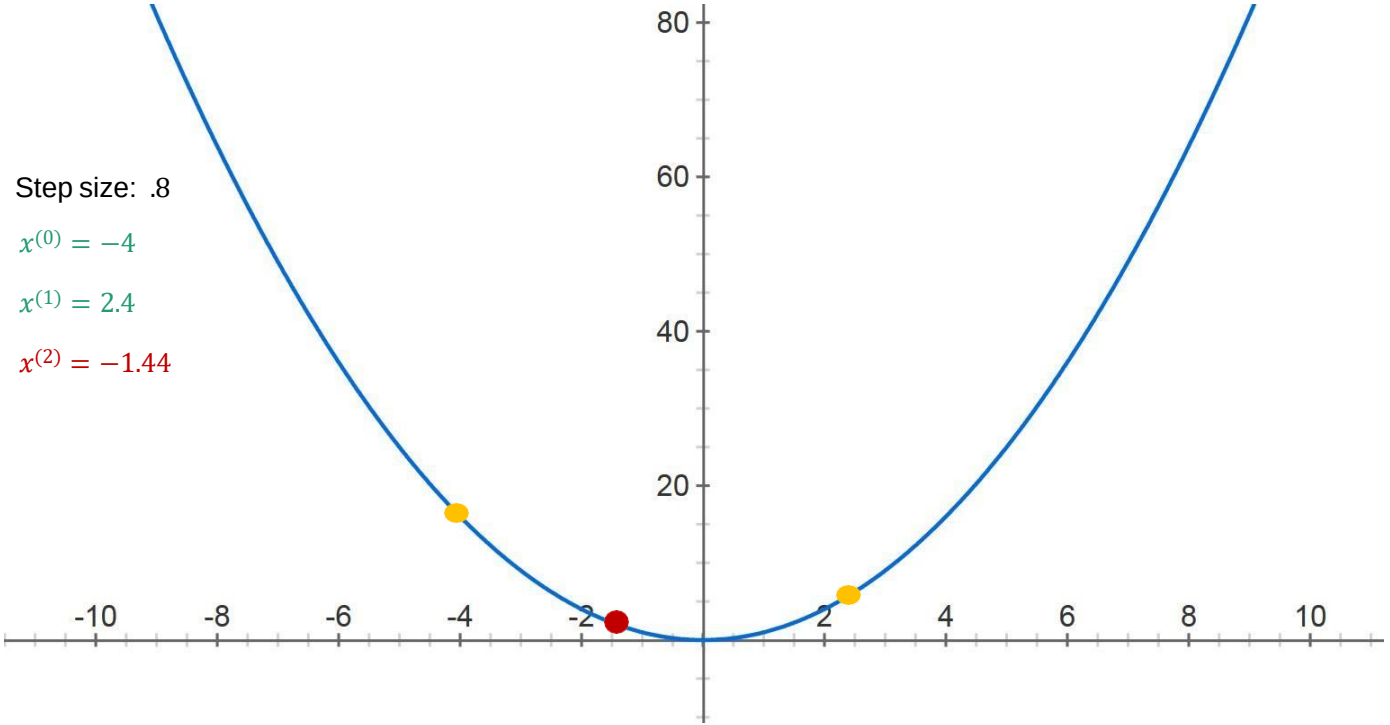
$f(x) = x^2$

Step size: .8

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

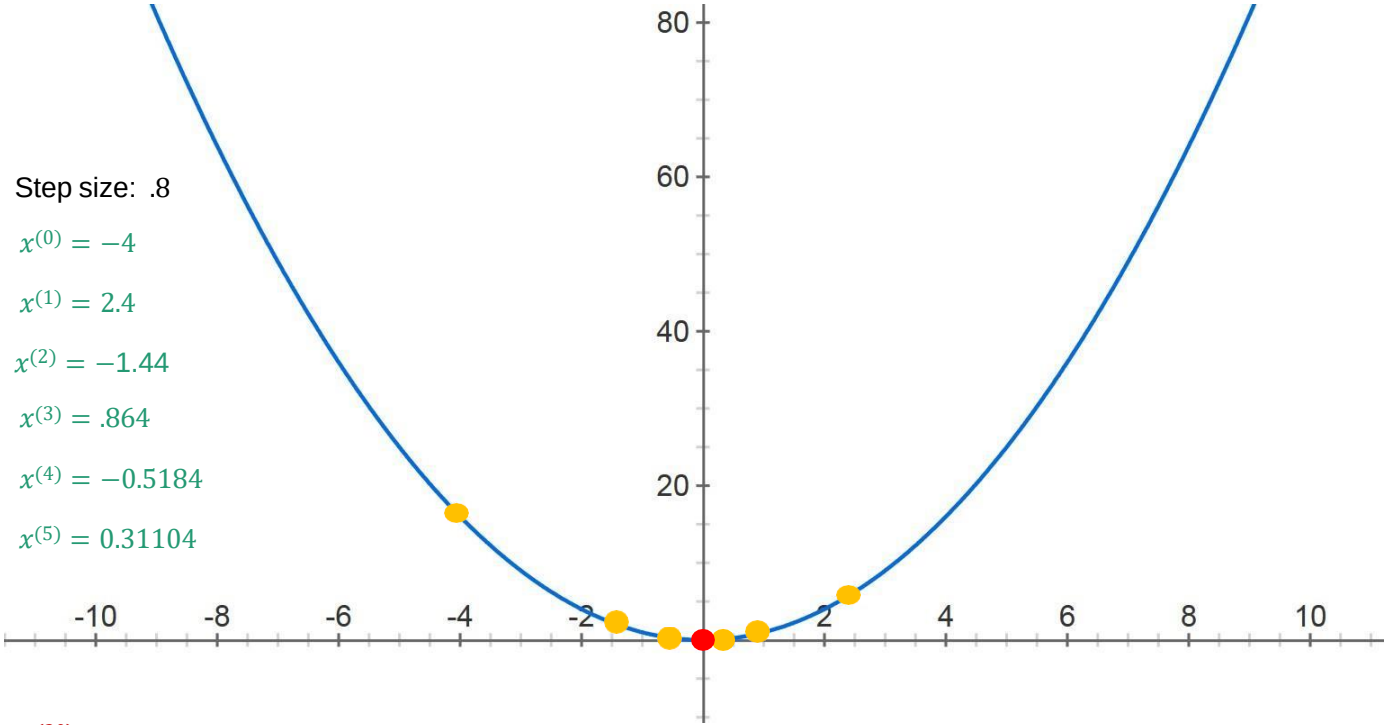
$x^{(1)} = 2.4$

$x^{(2)} = -1.44$



Gradient Descent - Example

$f(x) = x^2$



Step size: .8

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

$x^{(1)} = 2.4$

$x^{(2)} = -1.44$

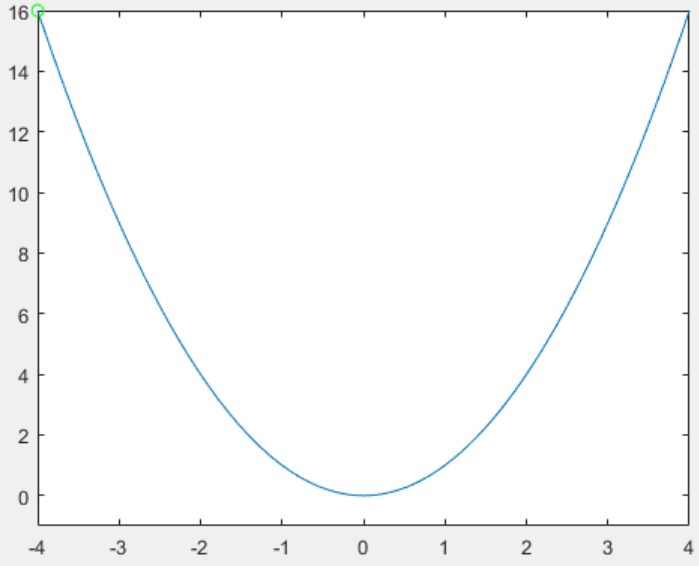
$x^{(3)} = .864$

$x^{(4)} = -0.5184$

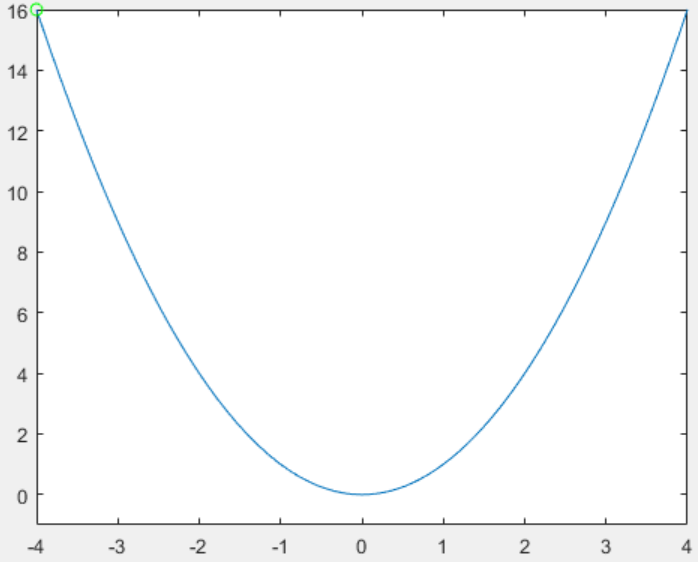
$x^{(5)} = 0.31104$

$x^{(30)} = -8.84296e - 07$

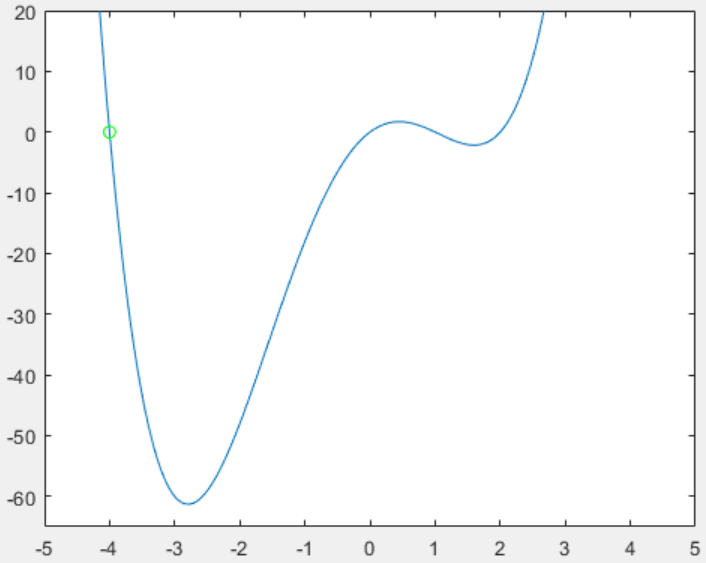
Gradient Descent – Step Size

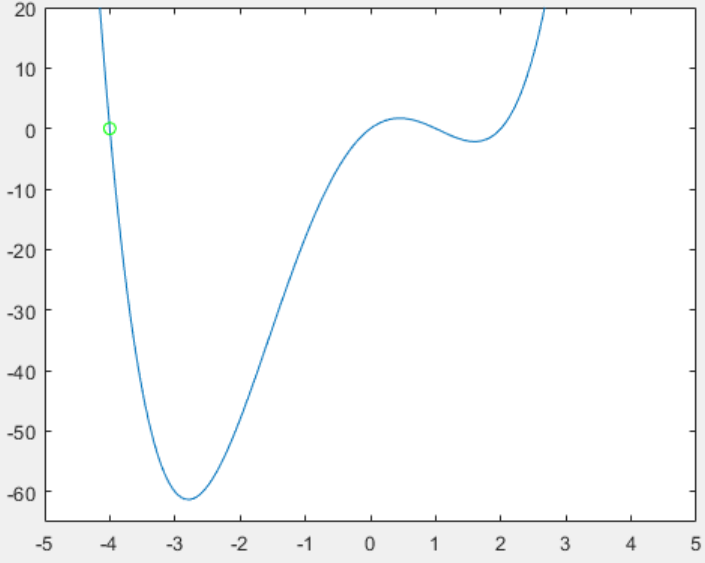


Step size: .9



Gradient Descent – Step Size





Step size matters!

Things to remember

- ✓ **Model representation:**

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

- ✓ **Cost function**

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

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

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

Exercise

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

Notebook:

3- Supervised Learning I – Regression/
LAB/T5B3ML101N01_032024V1 -Linear_Reggression_with_Gradient_Descent

Batch Gradient Descent

Compute the gradient of the cost function wrt θj :

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

Gradient Descent Step iteratively:

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

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

```
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]])
```

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

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]])
```

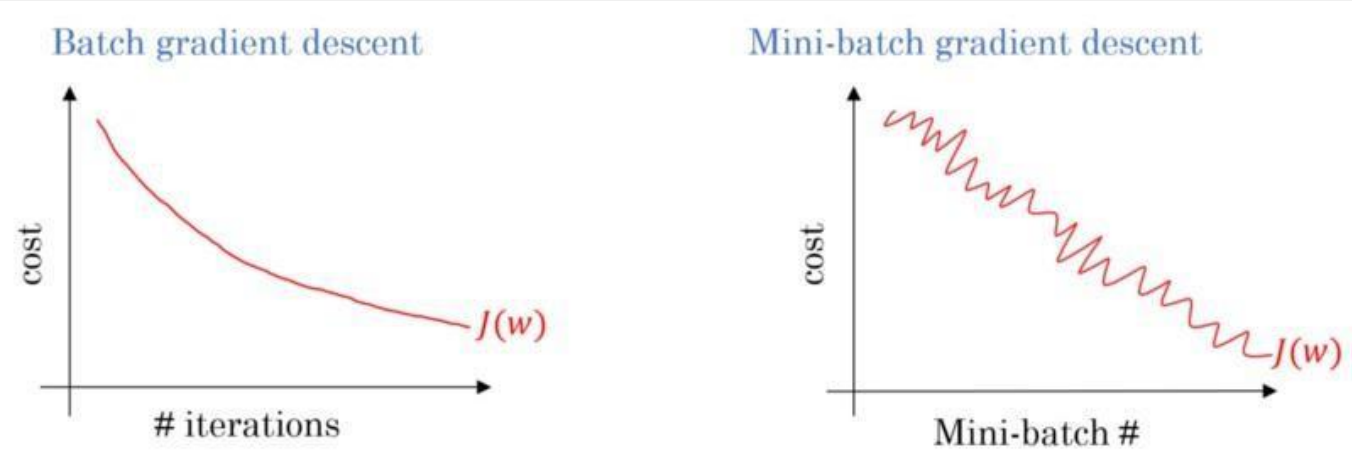
- Batch Gradient Descent is slow with big data.
- Picks a random instance at every step
- Instead of gently decreasing, it will bounce up and down (randomness).

```
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 GDover Stochastic GDis that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.

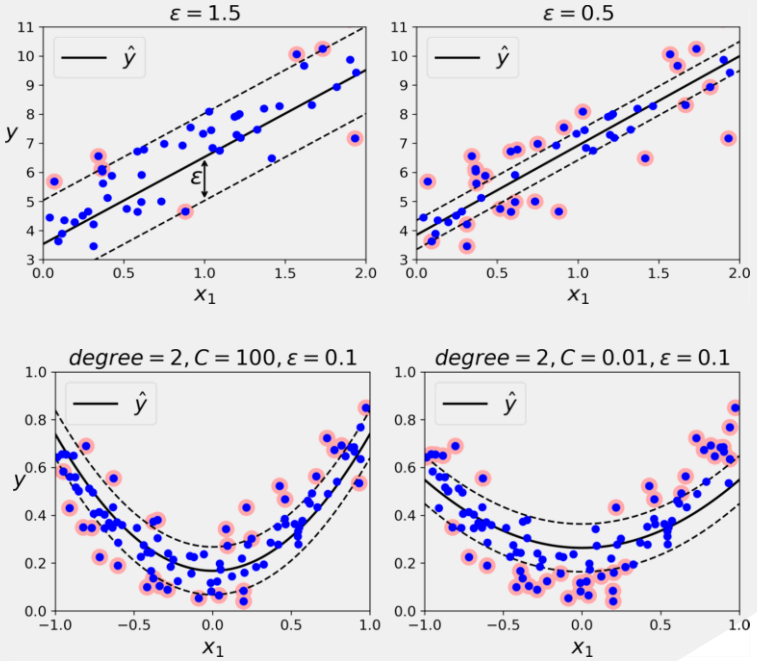


SVM Regressor

SVM algorithm is versatile (Linear and Nonlinear Regression)

- SVM Classifier: fits the largest possible street between two classes while limiting margin violations
- SVM Regressor: fit as many instances as possible *on* the street while limiting margin violations(i.e., instances *off* the street). The width of the street is controlled by a hyperparameter
- ϵ : Street width, Scale the data

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
from sklearn.svm import SVR
svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```



Note: SVM can be used as an outlier detector

Follow lab: T5B3ML101N11-0424-SVM_Regression

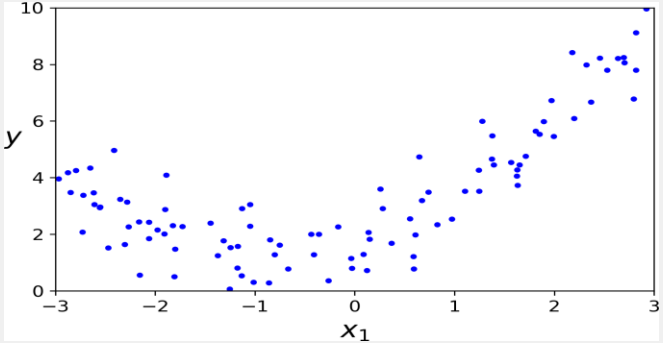
To run the code yourself and compare models

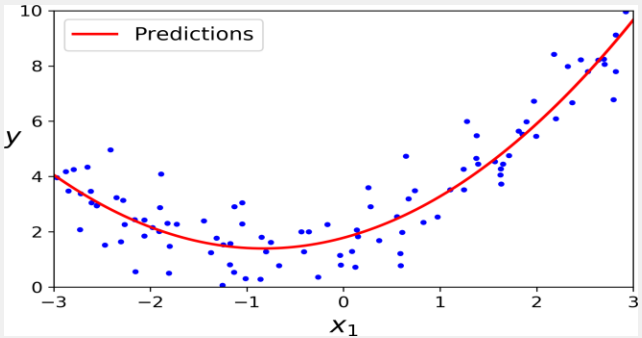
Polynomial Regression

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

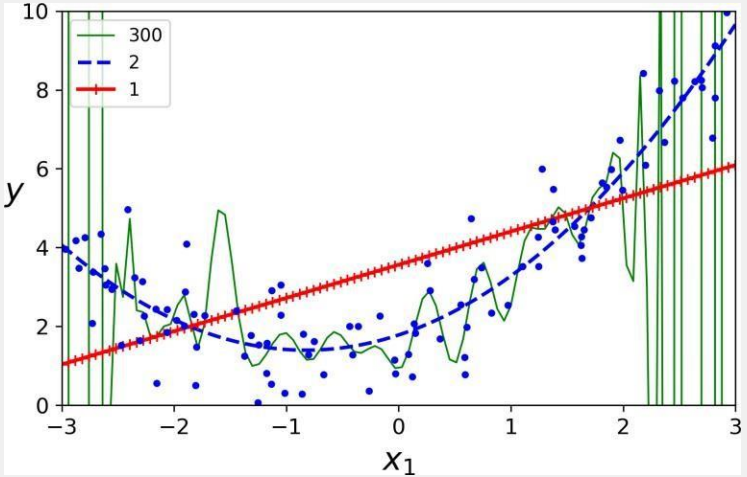
Complex data is not linear.

- 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: $\hat{Y} = 0.56X_1^2 + 0.93X_1 + 1.78$ when in fact the original

Original: $Y = 0.5X_1^2 + 1.0X_1 + 2.0 +$ Gaussian noise.

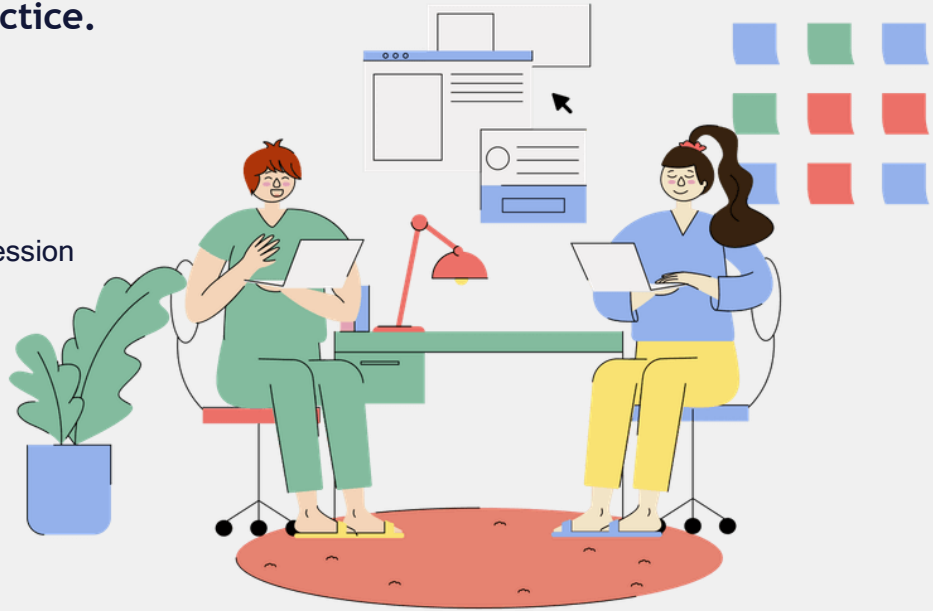


Exercise

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

Notebook:

3- Supervised Learning I - Regression/
LAB/T5B3ML101N02_032024V1 - Polynomial_Regression



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()
```

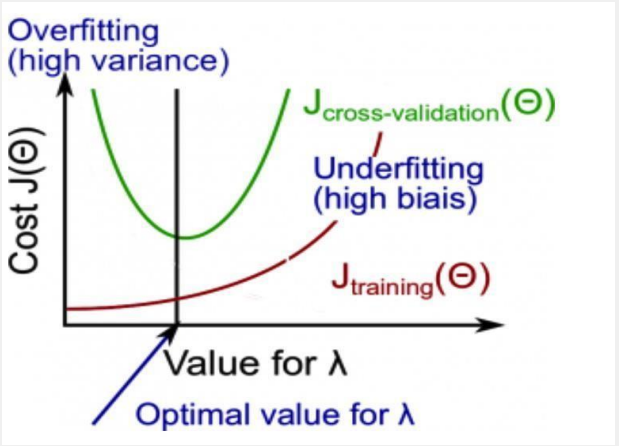
with these two:

```
import sklearn.neighbors

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


Regularized Linear Models

To regularize a polynomial model: reduce its degree or constraint the weights (the θs).

- Regularized Linear Regression: Ridge, Lasso and Elastic Net
  - MSE(θ) + λ *__Regularized Term__ (training only)
  - α is in fact λ (Hyperparameter: how much you want to regularize the model), 0 means no regularization.
  - Too high value ➔ High bias (flat line), too low ➔ High Variance
- Assume λ (α) =1000 then J(θ) = MSE(θ) + 1000 θ_1 + 1000 θ_2
- No Regularization for θ_0
- ***Remember to scale the data***



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

Regularized Regression: Ridge Regression

- **Ridge Regression:** involves adding a penalty that is equal to the square of the magnitude of coefficients. This approach helps reduce variance and is particularly useful for handling multicollinearity without eliminating features.
- **When to Use:** Ridge Regression is recommended when dealing with numerous correlated features and the goal is to retain all of them in the model.

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]])
```

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

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

- **Lasso Regression:** Lasso (Least Absolute Shrinkage and Selection Operator) Regression adds a penalty equal to the absolute value of the magnitude of coefficients. This encourages sparsity and is useful for feature selection and building simpler models.
- **When to use:** Use Lasso Regression when you suspect that only a few features are important, or when you need a more interpretable model with fewer non-zero coefficients.

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])
```

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \Sigma_{i=1}^{n} \left|\theta_i\right|$$

Using SGDRegressor

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

Least **A**bsolute **S**hrinkage and **S**election **O**perator: eliminate the weights of the least important features so it is considered as a feature selection

Regularized Regression: Elastic Regression

- **Elastic Net Regression**: Integrates the penalties of both Lasso and Ridge, providing a balance between sparsity and regularization, ideal for high-dimensional datasets with correlated features.
- **When to Use**: Best suited for situations with numerous correlated features, offering the advantages of both Lasso and Ridge, and particularly effective in high-dimensional scenarios with potentially irrelevant features.

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

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + r\alpha\Sigma_{i=1}^{n}\left|\theta_i\right| + \frac{1-r}{2}\alpha\Sigma_{i=1}^{n}\theta_i^2$$

Ful Comparison

RIDGE REGRESSOR

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

ELASTIC NET REGRESSOR

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + r\alpha\Sigma_{i=1}^{n}\left|\theta_i\right| + \frac{1-r}{2}\alpha\Sigma_{i=1}^{n}\theta_i^2$$

LASSO REGRESSOR

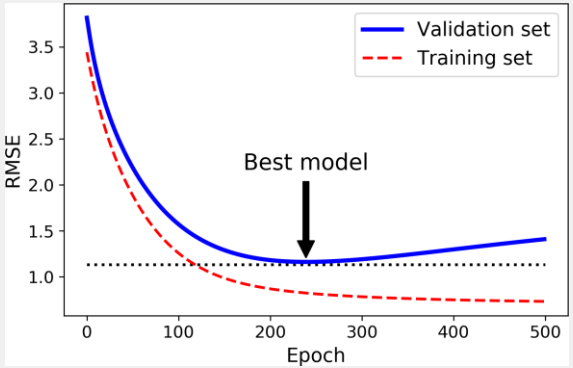
$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha\Sigma_{i=1}^{n}\left|\theta_i\right|$$

Regularization Methods

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

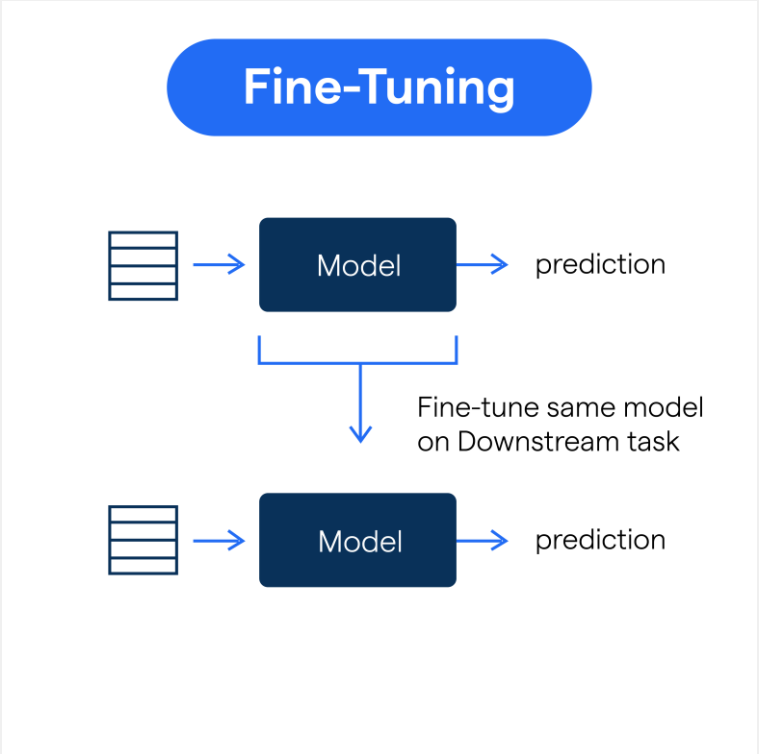
Transitioning to Google-Colab for hands-on coding practice.

Notebook:

3- Supervised Learning I - Regression/
LAB/T5B3ML101N03_032024V1 -Regularized_Linear_Models

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 (λ)	Co-efficient and slope in Regression (β)
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:

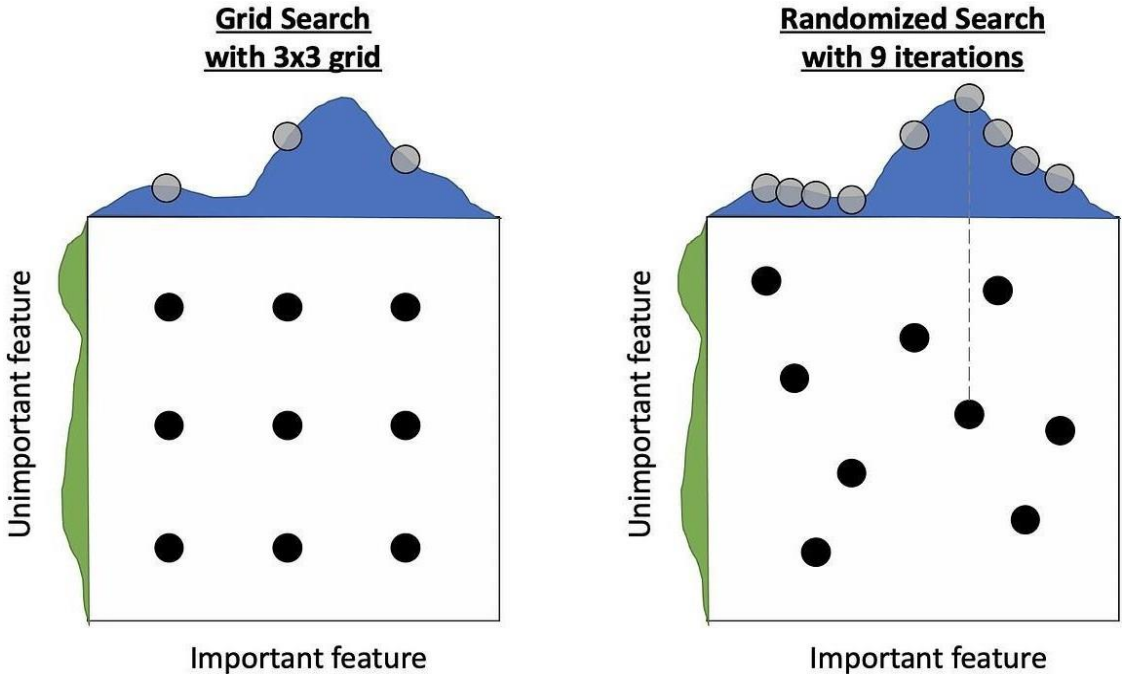
Grid search

- 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

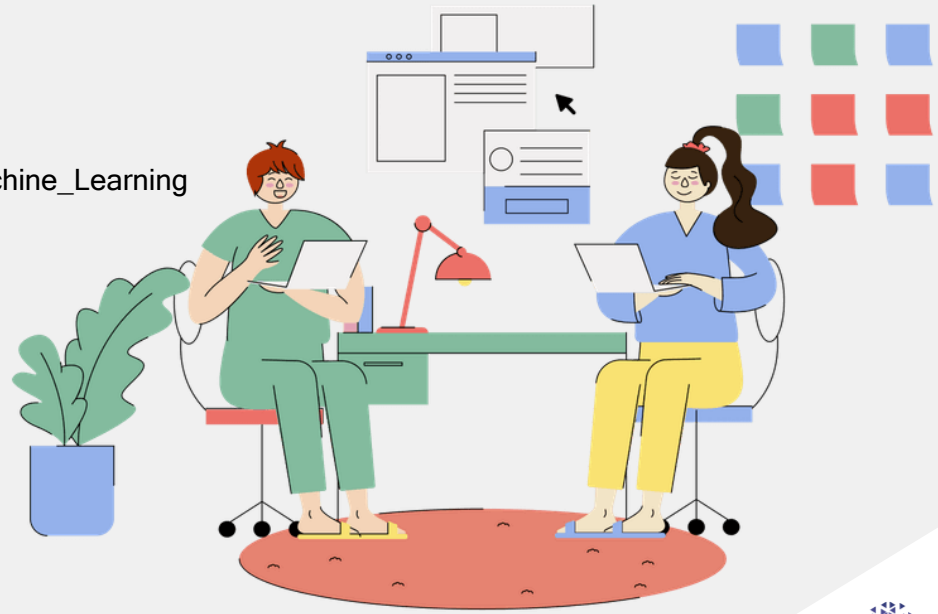
```
# Create the RandomizedSearchCV object
random_search = RandomizedSearchCV(estimator=rf,
param_distributions=hyperparameters,  n_iter=10, cv=5, n_jobs=-1, verbose=1, random_state=42)

# 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:

3- Supervised Learning I - Regression/
LAB/T5B3ML101N07_032024V1- End_to_End_Machine_Learning



Thank You

SDAIA
الهيئة السعودية للبيانات والذكاء الاصطناعي
Saudi Data & AI Authority