INTRODUCTION TO SUPERVISED LEARNING & ML ALGORITHMS: LINEAR REGRESSION

BY

SAIFUL BARI IFTU
LECTURER, DEPT. OF CSE, BRAC UNIVERSITY

CONTENTS

What Is Supervised Learning?

Types of Supervised Learning

Popular Supervised Learning Algorithms

Regression

Linear Regression

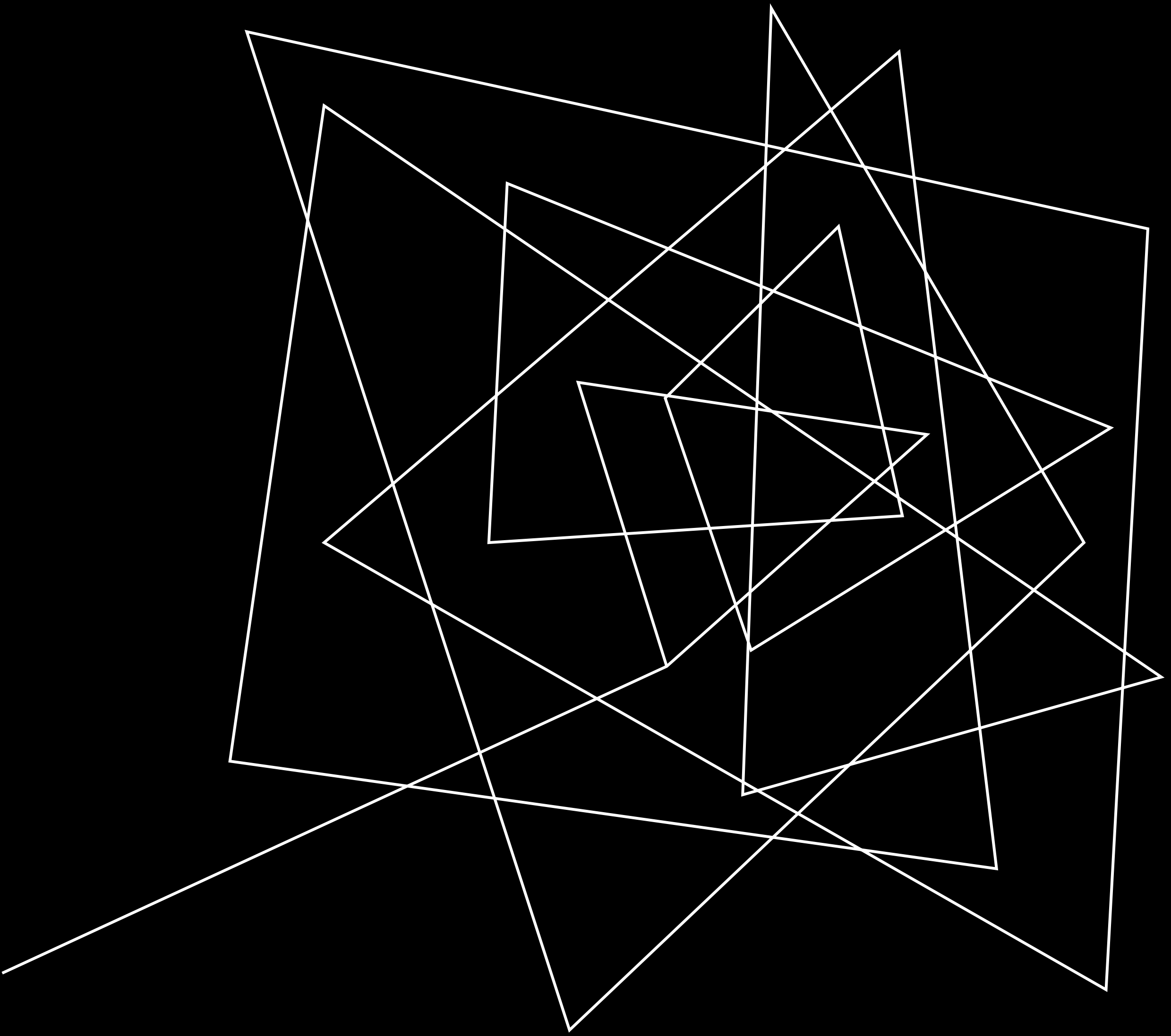
Gradient Descent & Its Variations

Vectorization

Feature Mapping

Polynomial Regression

Example Problems



WHAT IS SUPERVISED LEARNING?

WHAT IS SUPERVISED LEARNING?

Formal Definition (What?)

- Supervised Learning is a type of machine learning where the model is trained on a labeled dataset, meaning the input data is paired with the correct output. The model learns to map inputs to outputs by finding patterns in the data, which it then uses to predict outcomes for new, unseen data. The goal is to minimize the difference between the predicted output and the actual labeled output.

Alternative Definition (How?)

- Supervised learning (SL) is a paradigm in machine learning where input objects (for example, a vector of predictor variables) and a desired output value (also known as a human-labeled supervisory signal) train a model. The training data is processed, building a function that maps new data to expected output values.

CHARACTERISTICS OF SUPERVISED LEARNING

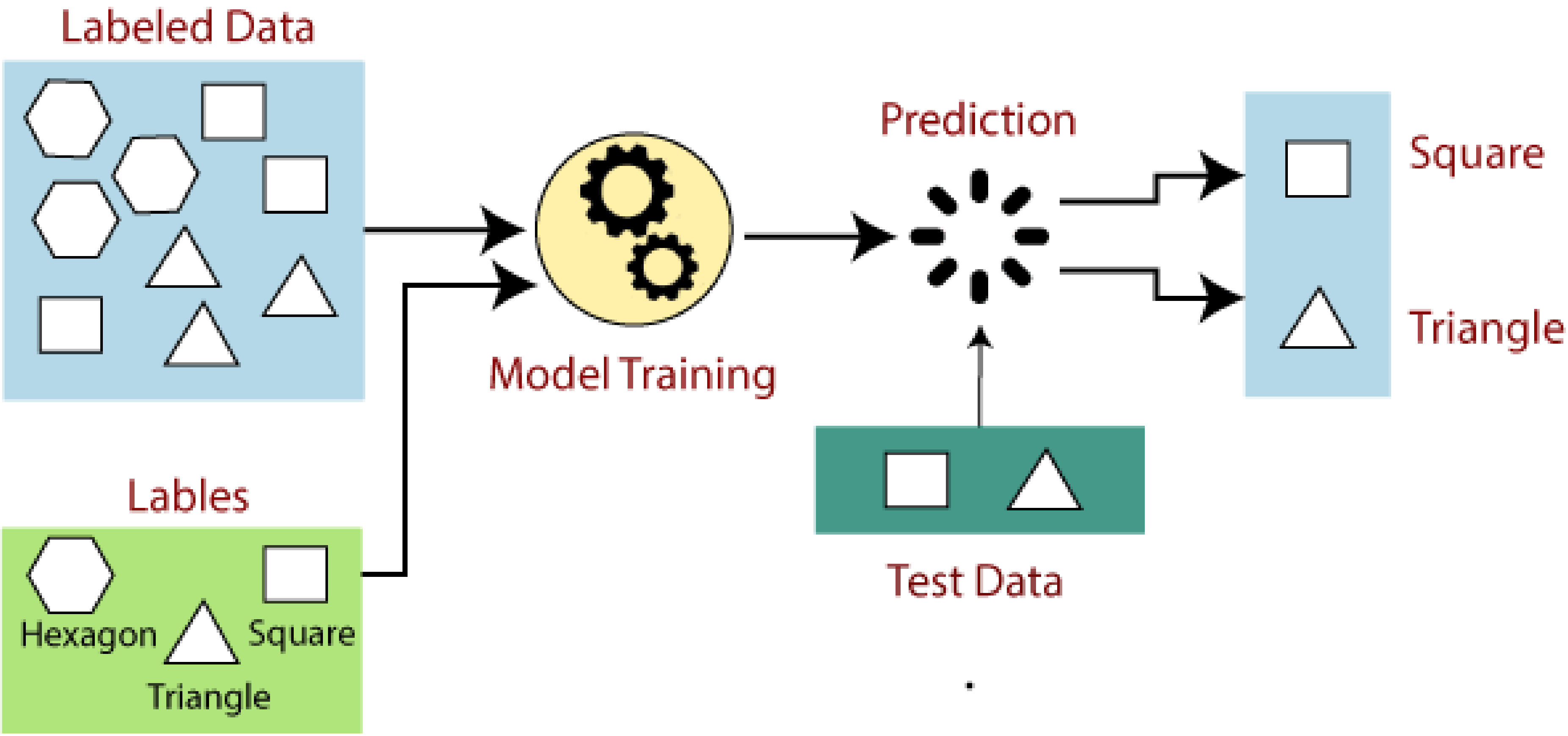
- **Labeled Data:** In supervised learning, the model is trained using a labeled dataset. Each training example consists of input features (attributes or characteristics) paired with corresponding output labels (desired outcomes).

- **Guided Learning:** Supervised learning is akin to teaching with the guidance of a teacher. The model learns to map inputs to the correct output by identifying patterns in the labeled data. The model receives feedback in the form of error correction during training, allowing it to improve its predictions over time.

- **Predictive Mapping:** The goal is to learn a mapping between input and output data. After training, the model can predict the output for new, unseen data based on the patterns learned from the training data.

- **Applications:** Supervised learning has a wide range of applications, including classification, regression, and more complex tasks like image recognition and natural language processing.

SUPERVISED LEARNING



Source: https://www.javatpoint.com/supervised-machine-learning

Supervised Learning Process

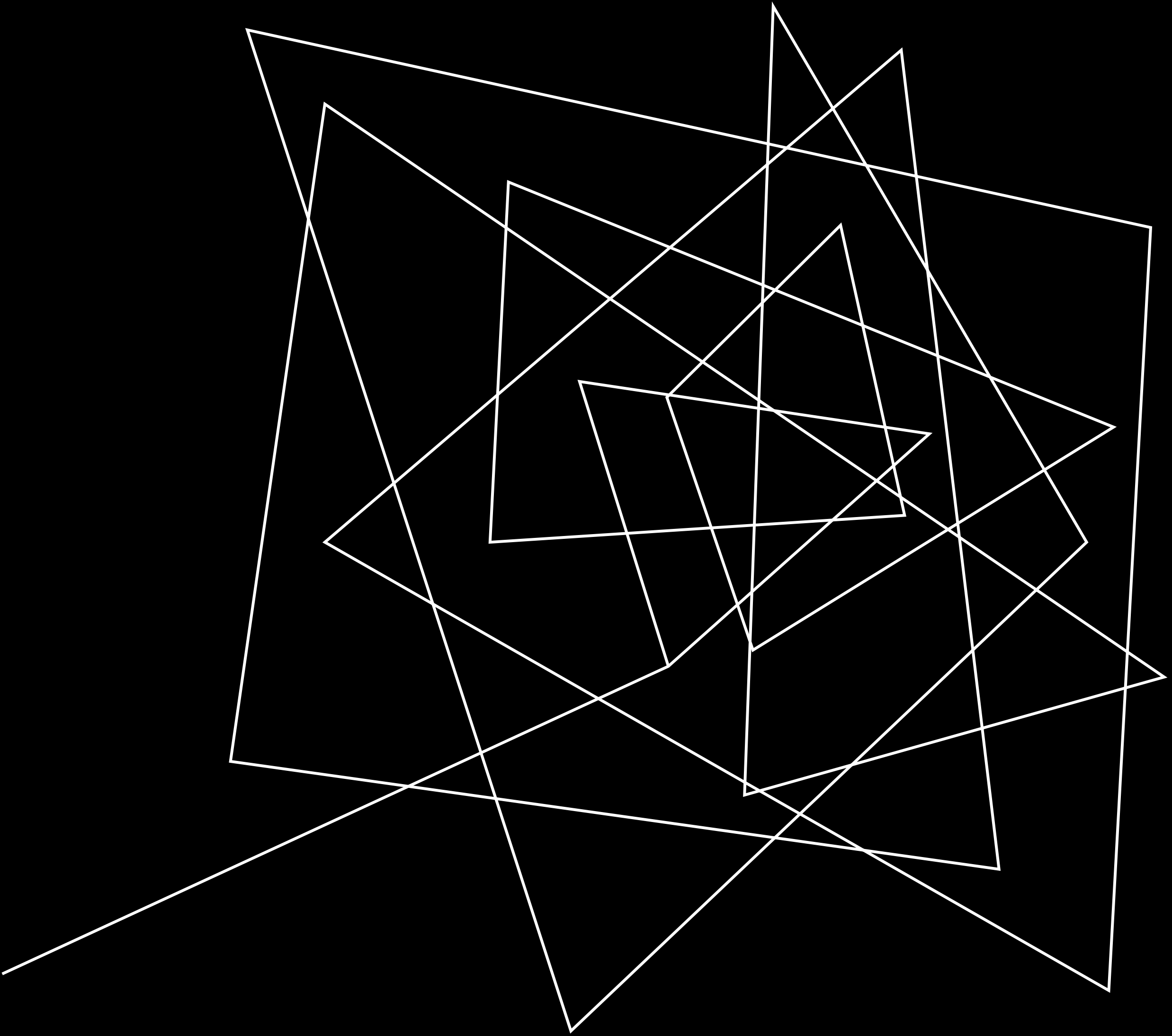
SUPERVISED LEARNING SETUP

In **Supervised Learning:**

- There is an input $\boldsymbol{x} \in \boldsymbol{x}$, typically a vector of features (or covariates).
- There is a target $\boldsymbol{t} \in \boldsymbol{\mathcal{T}}$, (also called response, outcome, output, class).
- **Objective** is to learn a function $\mathbf{f} : \boldsymbol{x} \longrightarrow \boldsymbol{\mathcal{T}}$, such that, $\mathbf{t} \approx \mathbf{y} = \mathbf{f}(\mathbf{x})$ based on some data $\boldsymbol{\mathcal{D}} = \{(\boldsymbol{x}^{(\boldsymbol{i})}, \boldsymbol{t}^{(\boldsymbol{i})}),\ \text{for } i = 1,2,\dots,\boldsymbol{N}\}$.

The General Approach:

- *Choose* a **Model** describing the relationships between variables of interest.
- *Define* a **Loss function** quantifying how bad the fit to the data is.
- *Choose* a **Regularizer** imposing some constraint/penalty on the loss function.
- Fit a model that minimizes the loss function and satisfies the constraint/penalty imposed by the Regularizer, possibly using an **Optimization Algorithm.**



TYPES OF SUPERVISED LEARNING

TYPES OF SUPERVISED LEARNING

There are **two major types** of Supervised Learning:

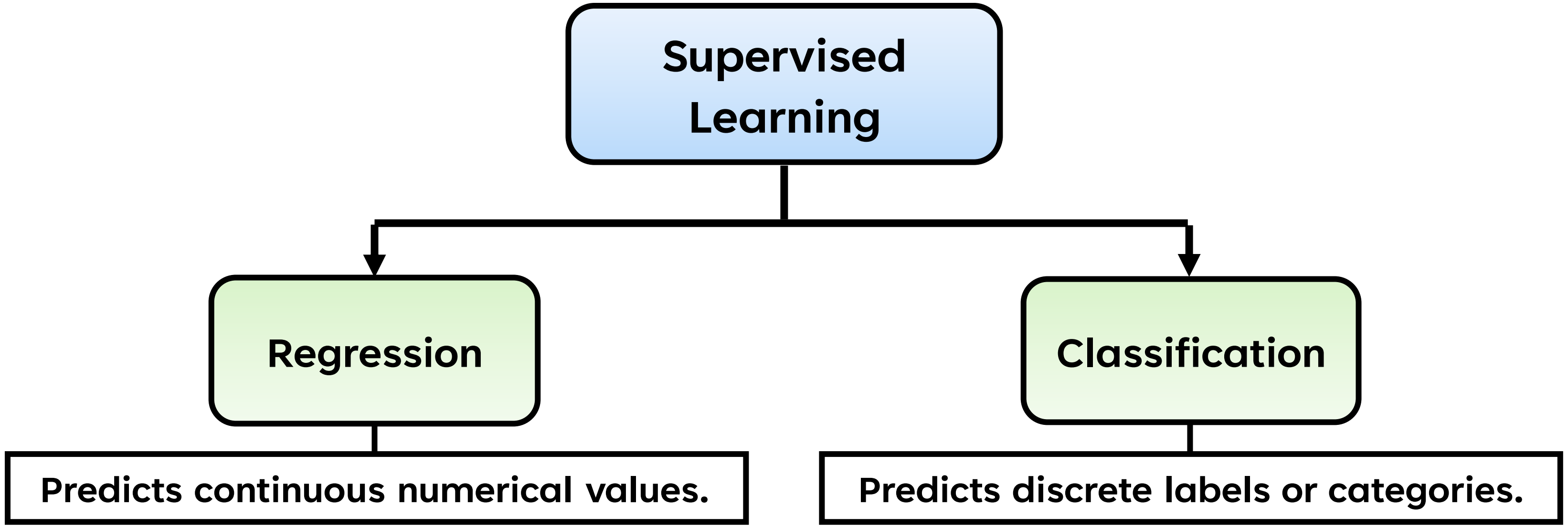
1. Regression: Regression is a type of supervised learning where the goal is to predict **a continuous output value** based on input data.

Examples: Email filtering (spam vs. not spam), disease diagnosis (cancerous vs. non-cancerous), image recognition (identifying objects like cats or dogs), etc.

2. Classification: Classification is a type of supervised learning where the goal is to categorize input data into **one of several predefined classes or labels**.

Examples: Predicting house prices, forecasting stock market trends, estimating the amount of rainfall, etc.

TYPES OF SUPERVISED LEARNING



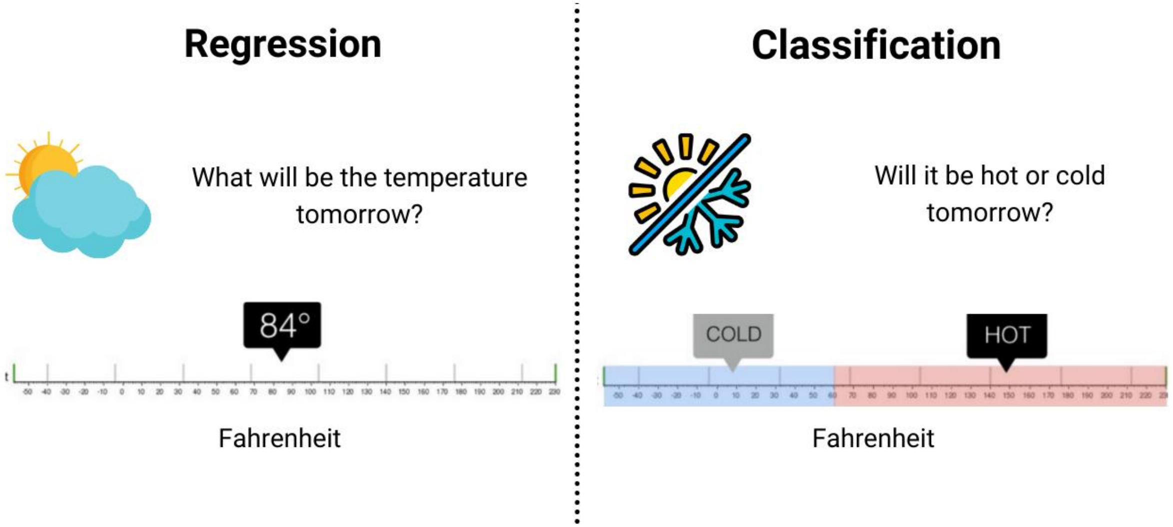
Supervised Learning

Regression

Predicts continuous numerical values.

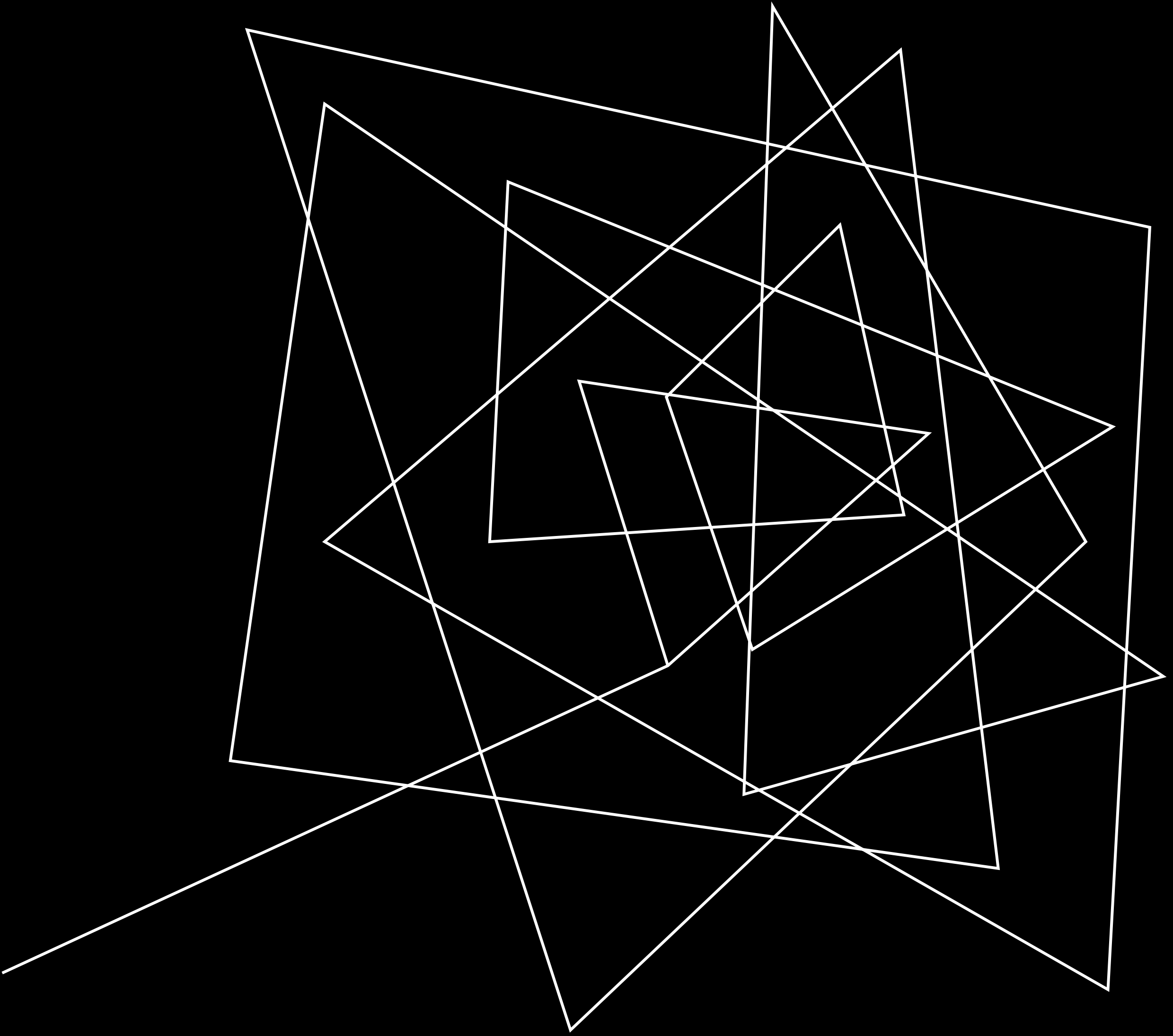
Classification

Predicts discrete labels or categories.



What will be the temperature tomorrow?

Will it be hot or cold tomorrow?



SUPERVISED LEARNING ALGORITHMS

BASIC QUESTIONS IN SUPERVISED LEARNING

Say, we have a dataset $\boldsymbol{\mathcal{D}}$ = $\{(\boldsymbol{x}^{(i)}, \boldsymbol{t}^{(i)}),\ \text{for } i = 1,2,\dots,\boldsymbol{N}\}$, where N = #samples.

The **Objective** is to learn a function $\mathbf{h} : \boldsymbol{x} \longrightarrow \mathcal{T}$, such that, $\mathbf{t} \approx \mathbf{y} = \mathbf{h}(\mathbf{x})$ based on the data $\boldsymbol{\mathcal{D}}$.

Key Questions:

- Which hypothesis space **H** to choose?
- How to measure the **degree of fit**?

Assume a relationship between the features and the labels by observing the data and its type. **Choose an appropriate algorithm** to **Model** this relationship mathematically considering the complexity associated.

- How to trade off the **degree of fit vs. complexity**?

Use simpler Model for simple tasks. If the Model is complex, use suitable **regularizing/counter overfitting techniques**.

- How do we find a good **h**?

Choose the appropriate **Cost Function** and **Optimizing Algorithm** for the task. Train & also Tune the relevant **Hyperparameters** properly.

- How do we know if a good **h** will **predict** well?

Choose a suitable **Evaluation Metric** & test the model on unseen data.

A SUPERVISED LEARNING ALGORITHM

So, Clearly, A **Model** is basically the **structure of the *assumed* relationship** between the input features and output labels. This relationship is represented mathematically (e.g. A function of the features) and is fine tuned through the training process.

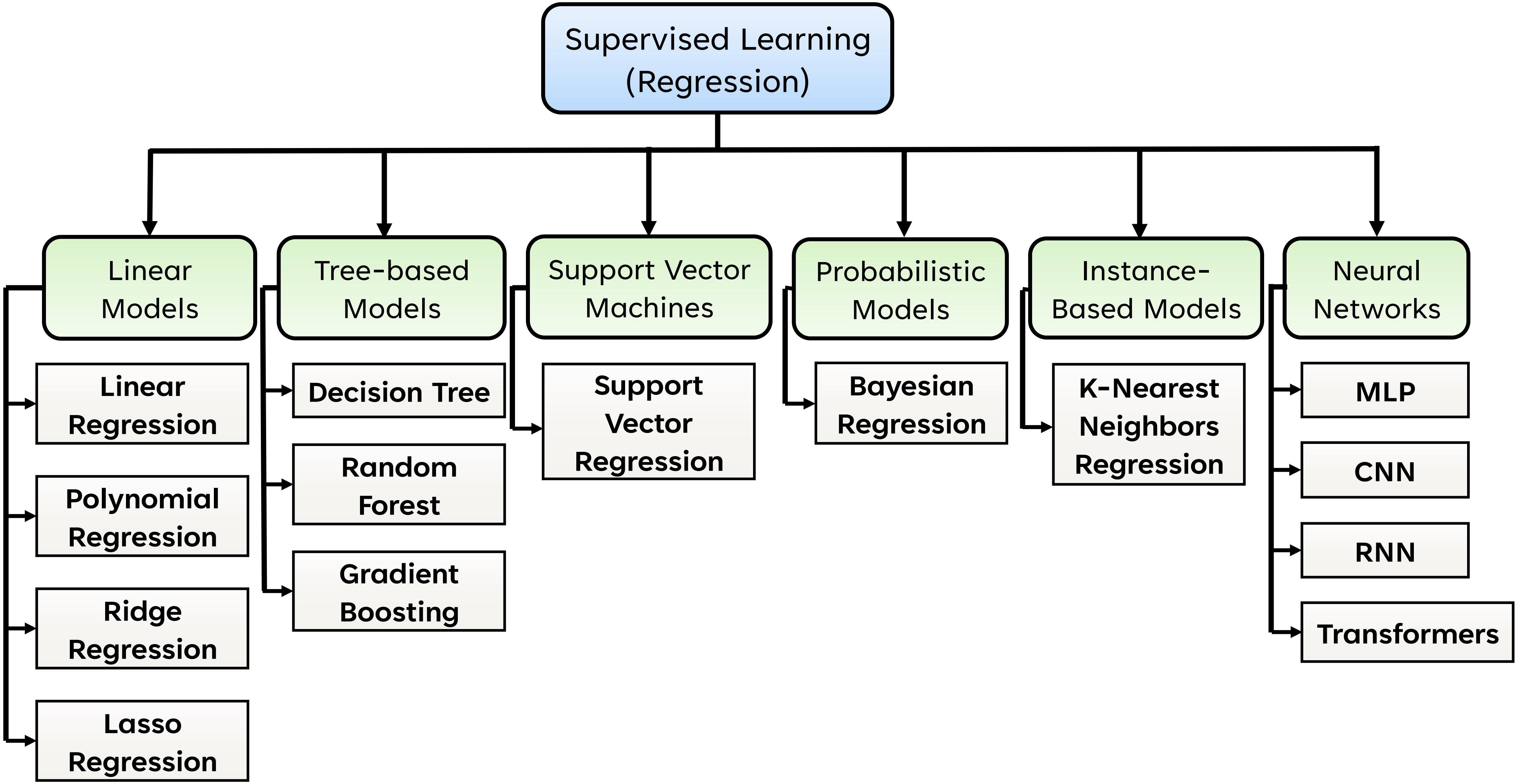
Different Algorithms assume different types of relationships between the features & the labels.

Examples:

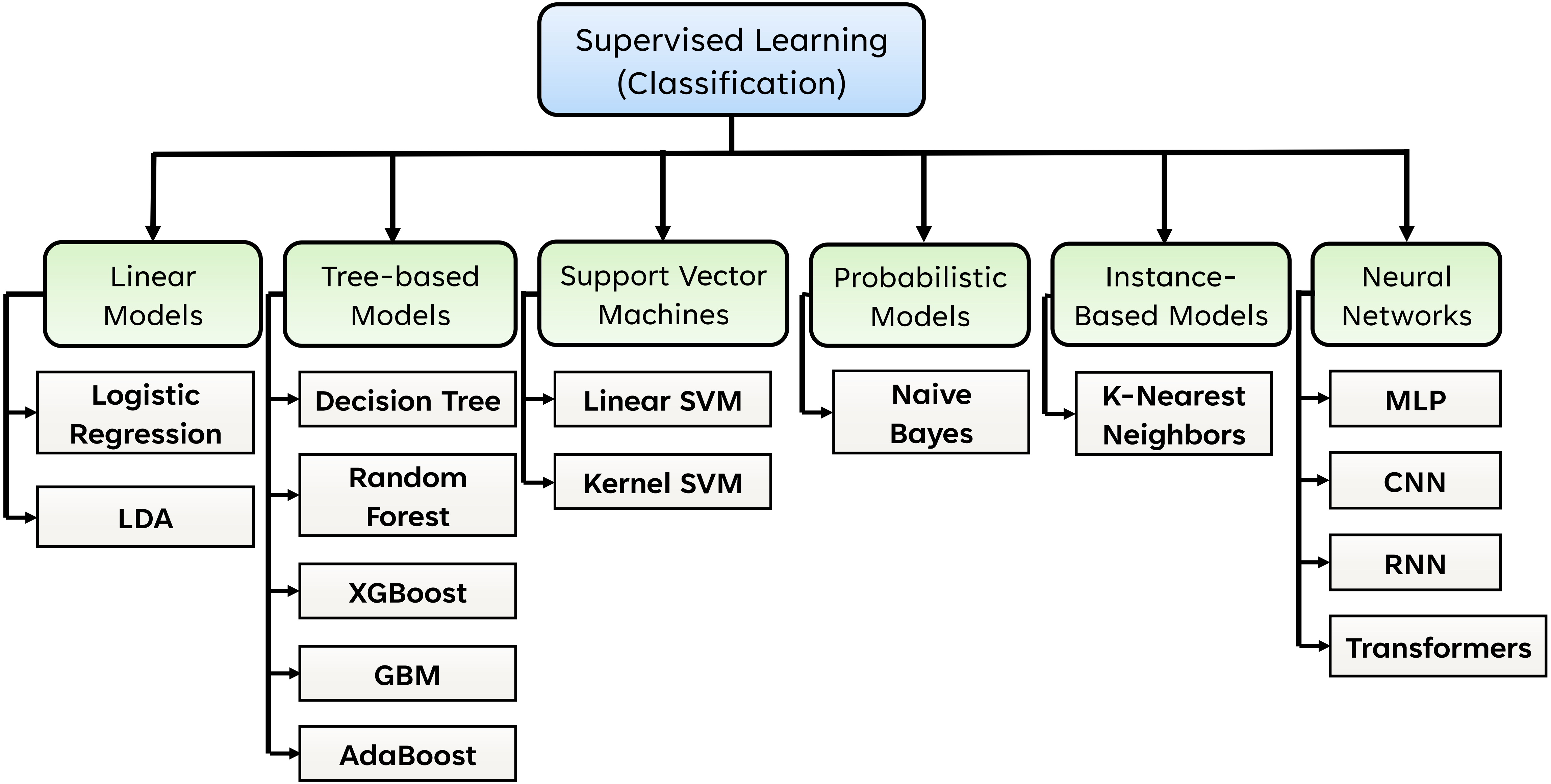
- **Linear regression:** Assumes that there is a **linear** relationship.
- **Decision Trees:** Assumption of a **Non-linear and rule-based** relationship.
- **Naive Bayes:** Assumes that **each feature independently contributes** to the output.
- **Neural Networks:** Can model very **complex, non-linear, hierarchical relationships** between features and labels.

Naturally, there are some variables, and some constants present in these models and the **objective** of the whole **Algorithm** is *to determine/tune the values of everything (bias, coefficients/weights) bar the independent variables (features) of the assumed model.*

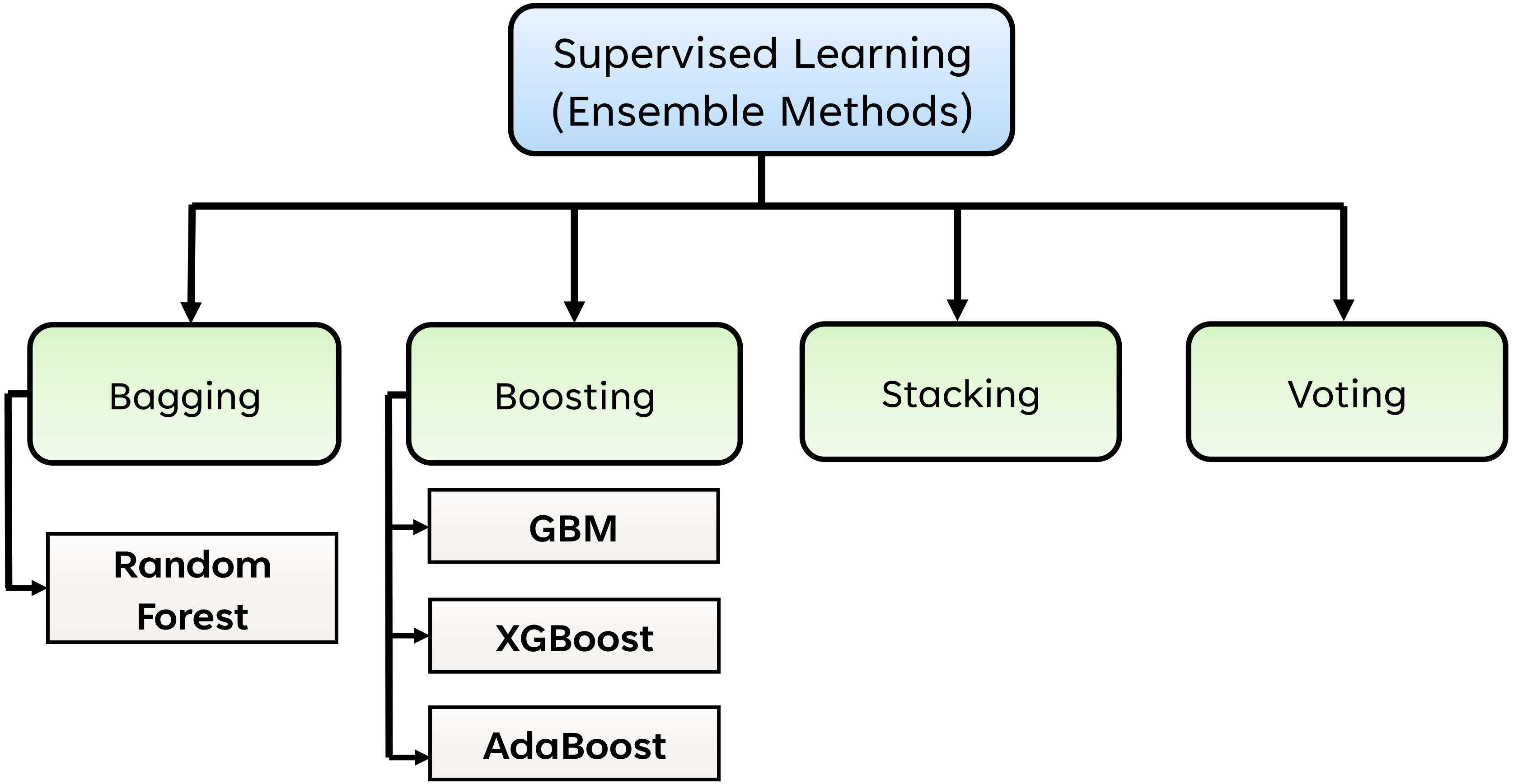
SUPERVISED LEARNING (REGRESSION)



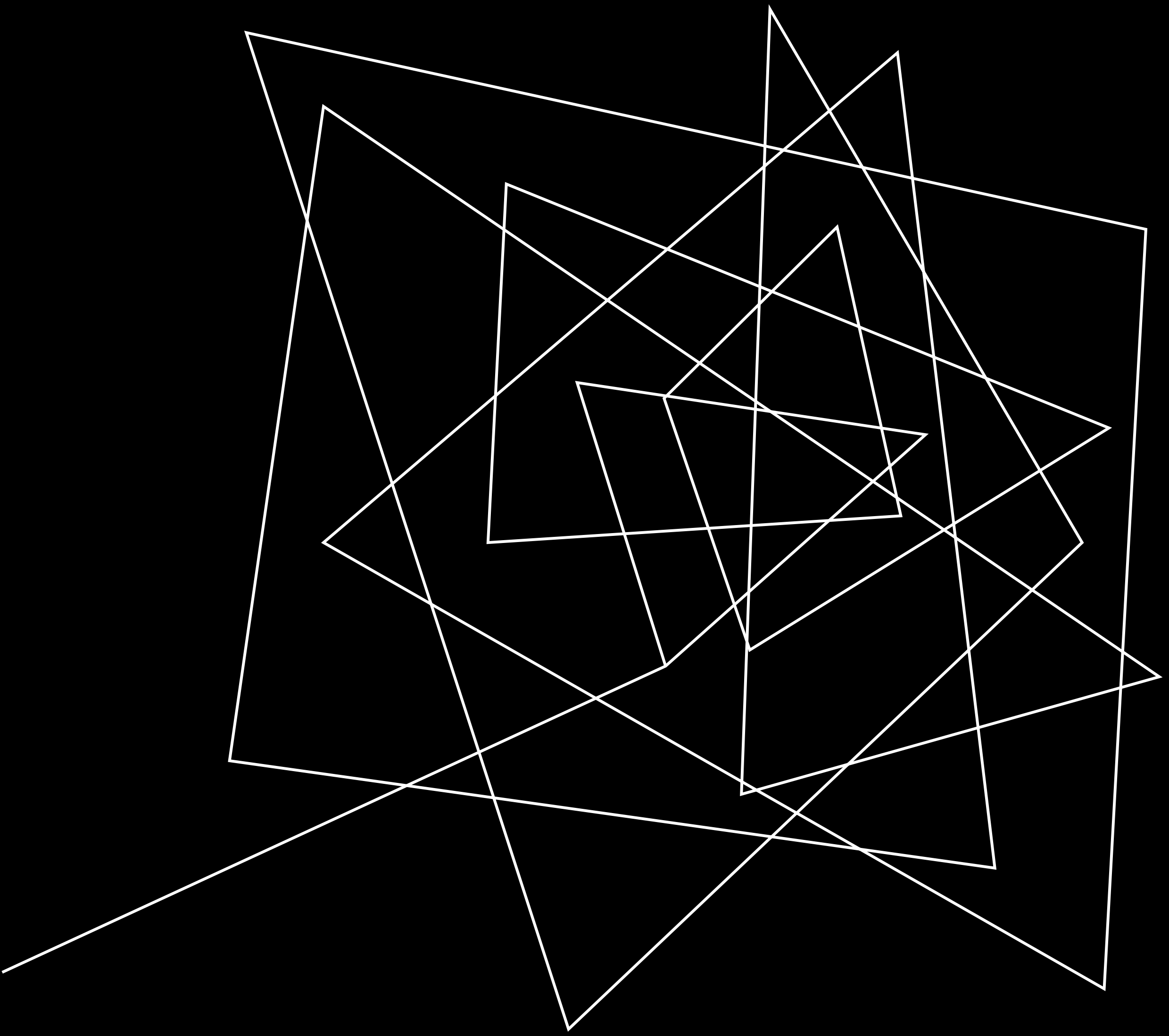
SUPERVISED LEARNING (CLASSIFICATION)



SUPERVISED LEARNING (ENSEMBLE METHODS)



Ensemble: Ensemble methods are techniques in machine learning that combine multiple models (often referred to as "weak learners") to create a stronger, more robust model. The idea is that by aggregating the predictions of several models, the ensemble can achieve better performance.



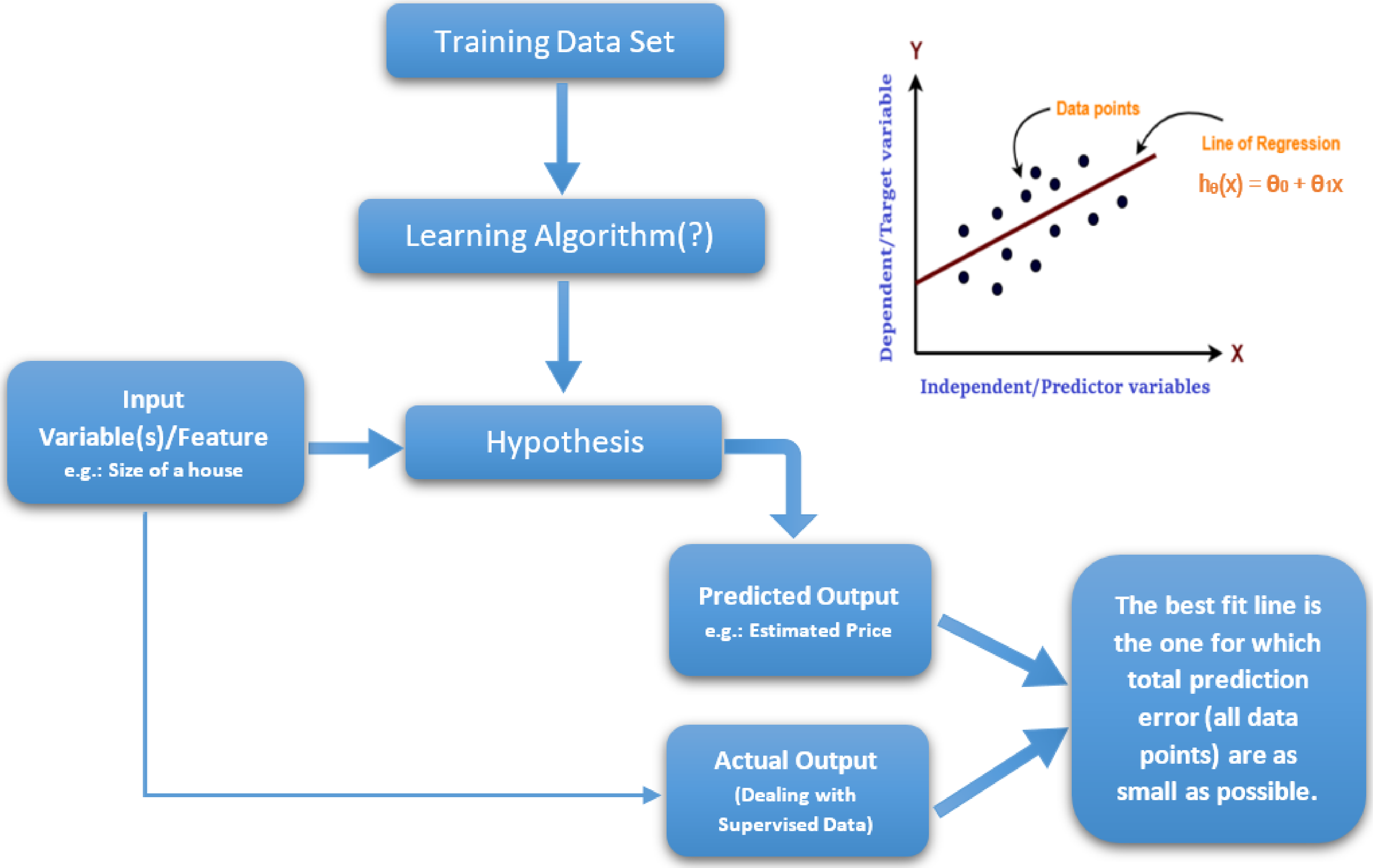
REGRESSION

REGRESSION

Regression is a type of supervised learning where we aim to **predict continuous numerical values** based on input features. It establishes a functional relationship between independent variables (features) and a dependent variable (target), allowing us to estimate or predict numeric outcomes.

- **Use Cases:** Financial Forecasting, Weather Prediction, Patient Monitoring etc.
- **Evaluation Metrics:** Common metrics include Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R-squared (coefficient of determination).

REGRESSION WORKFLOW



Training Data Set

Learning Algorithm(?)

Input Variable(s)/Feature
e.g.: Size of a house

Hypothesis

Predicted Output
e.g.: Estimated Price

Actual Output
(Dealing with Supervised Data)

The best fit line is the one for which total prediction error (all data points) are as small as possible.

Y
Dependent/Target variable
Data points
Line of Regression
$h_\theta(x) = \theta_0 + \theta_1 x$
X
Independent/Predictor variables

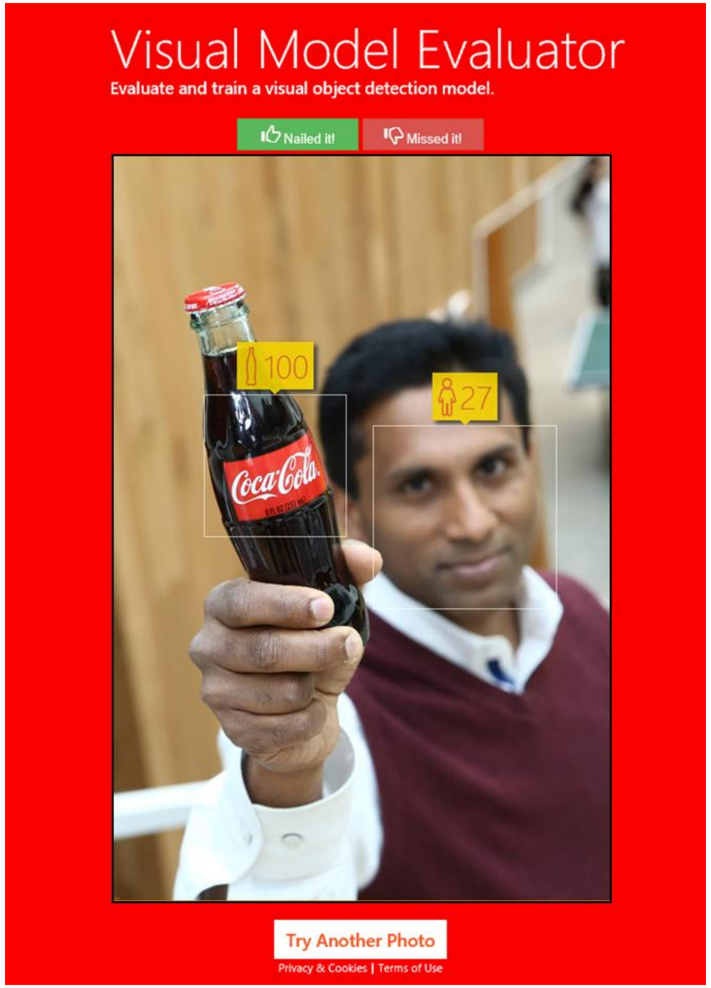
Source: https://medium.com/analytics-vidhya/linear-regression-machine-learning-algorithm-detailed-view-657beac3d2f1

REGRESSION (EXAMPLE)

Source: https://blog.paperspace.com/forecasting-stock-price-prediction-using-deep-learning/

Stock Price Prediction from Historical Data

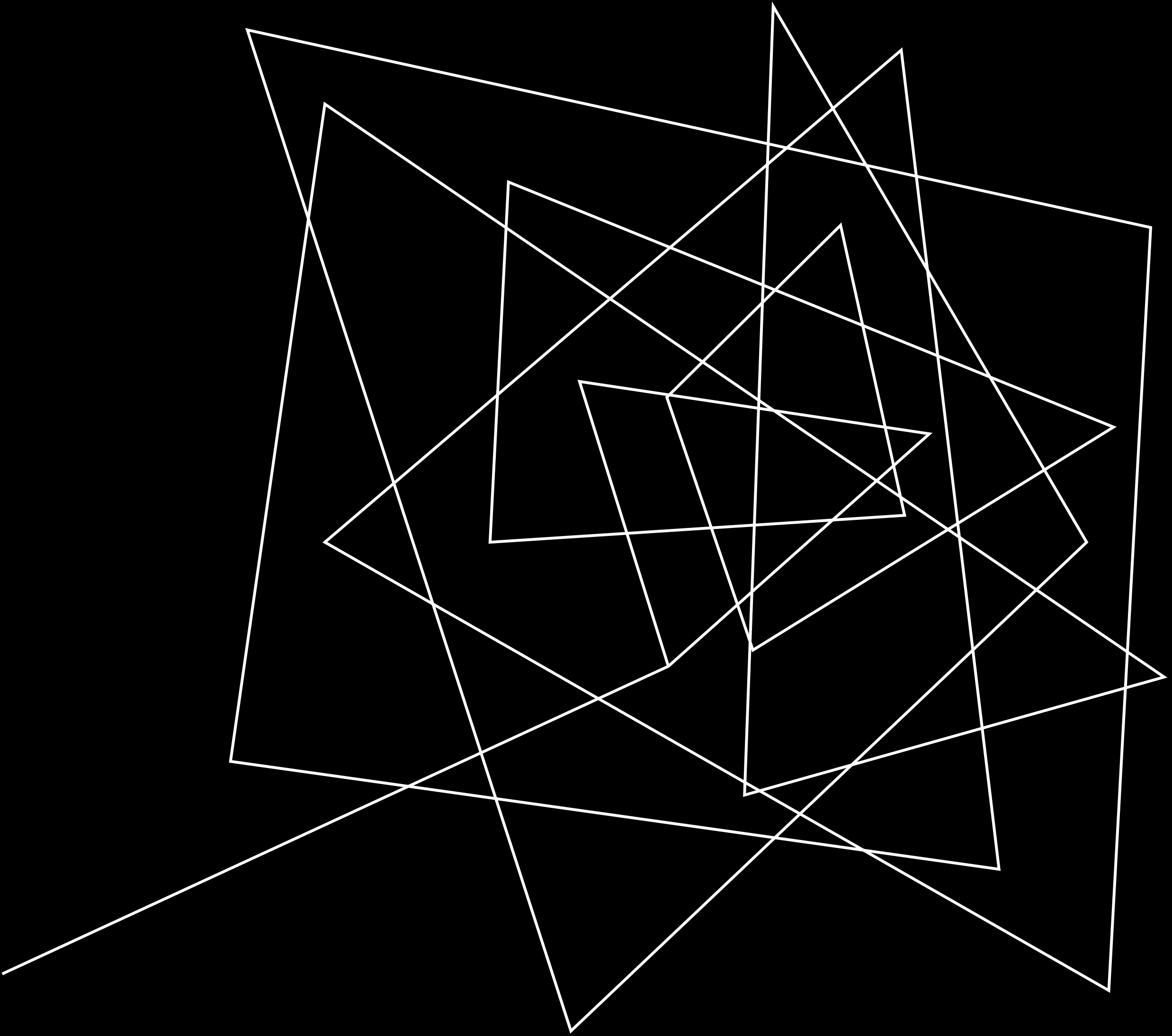
REGRESSION (EXAMPLE)



Visual Model Evaluator

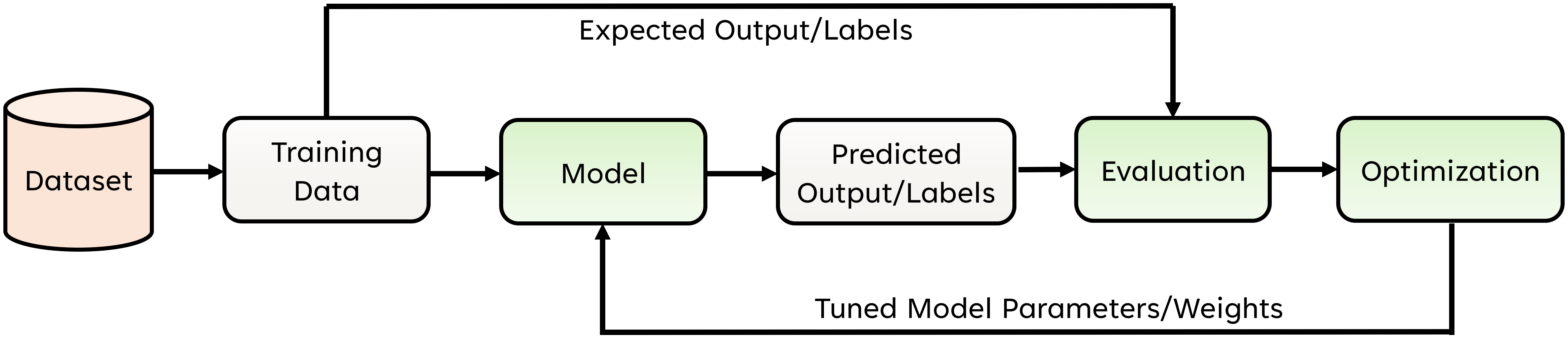
Evaluate and train a visual object detection model.

Age Prediction from Facial Images



ALGORITHMS: LINEAR REGRESSION

OVERVIEW OF A GENERIC SL ALGORITHM



The Diagram above represents **Supervised Learning Workflow** in general. Any **SL Algorithm** has **three** important aspects related to three stages in this workflow. They are:

- **Model**: The structure of the assumed mathematical relationship between features and labels. ***The model is the defining characteristic of a machine learning algorithm.***
- **Cost Function:** Measures how well the model's predictions match the actual data.
- **Optimization Algorithm:** Determines how the model parameters are adjusted to minimize the cost function.

LINEAR REGRESSION

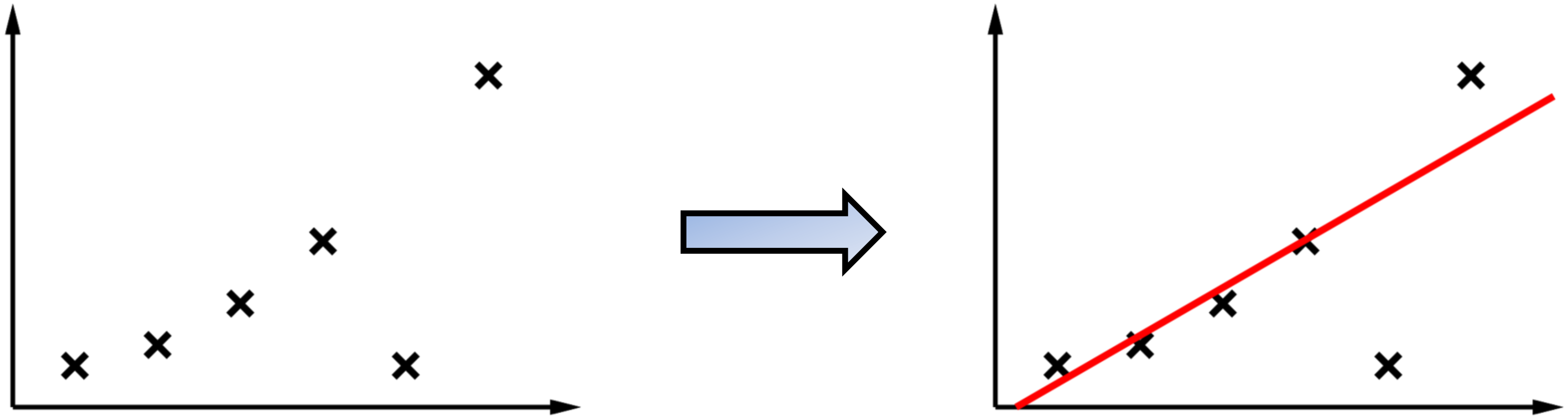
Linear regression is a fundamental statistical technique and ML algorithm used to model the relationship between a dependent variable (often called the **target** or **output**) and one or more independent variables (also known as **features** or **inputs**). The objective of linear regression is to find a linear relationship that best predicts the dependent variable based on the values of the independent variables.

The assumption of a linear relationship between the feature and the target/label is crucial in determining whether this algorithm will work well on a particular task or not.

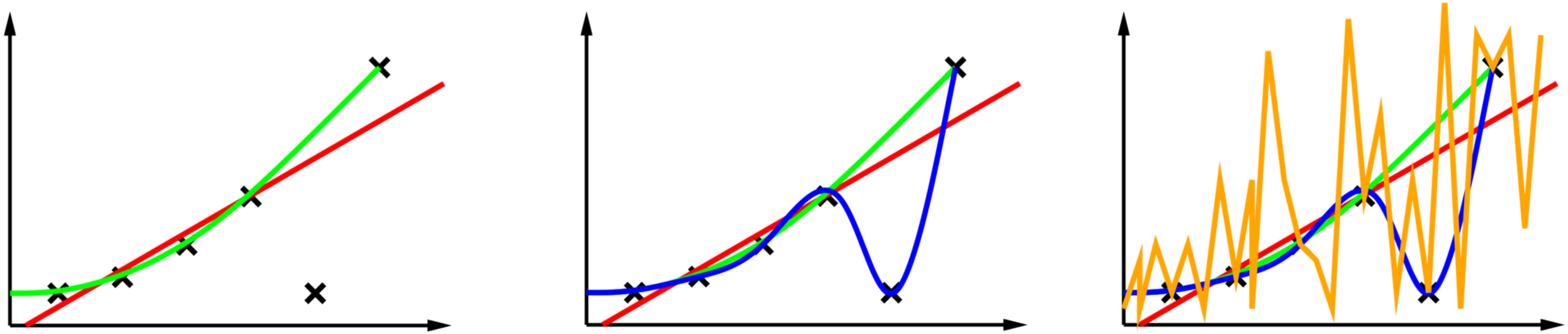
Assumptions of Linear Regression:

- **Linearity:** The *relationship* between the dependent and independent variables is *linear*.
- **Independence:** The *observations* are independent of each other.
- **Homoscedasticity:** The residuals (*errors*) have constant variance across all levels of the independent variables.
- **Normality:** The residuals of the model are normally distributed.
- **No Multicollinearity (in multiple regression):** The *independent variables* should not be highly *correlated* with each other.

ASSUMPTION OF LINEAR RELATIONSHIP

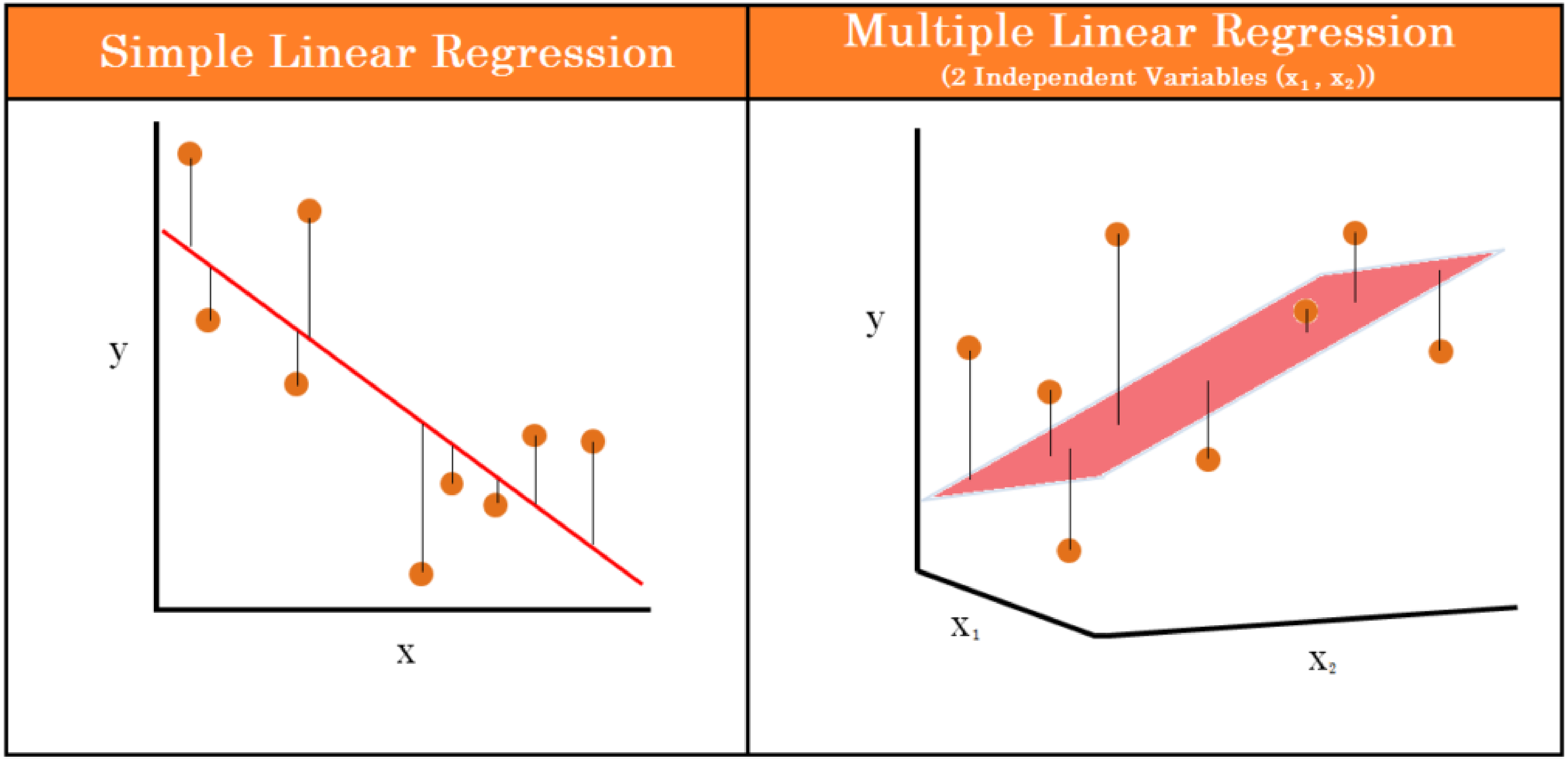


Curve Fitting Assuming Linear Relationship



Non-Linear Curve Fitting (Different Degrees)

LINEAR REGRESSION IN MULTIPLE DIMENSIONS


Simple Linear Regression
y
x
Multiple Linear Regression
(2 Independent Variables (x_1 , x_2))
y
X_1
X_2

Source: https://medium.com/@thaddeussegura/multiple-linear-regression-in-200-words-data-8bdbcef34436

LINEAR REGRESSION: MODEL

Model: In linear regression, the idea is to use a linear function of the features $\mathbf{x} = [x_1, x_2, \ldots, x_D] \in \mathbb{R}^D$ to make predictions **y** of the target value $t \in \mathbb{R}$:

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) = \sum_{j=1}^{n} w_j \mathbf{x}_j + \mathbf{b}$$

- ➢ **y** is the prediction.
- ➢ w represents the **weights**.
- ➢ **b** is the **bias** (or **intercept**).
- ➢ n is the **number of features**.
- ➢ w and **b** together are the **parameters, θ,** where **b =** $\boldsymbol{\theta}_0$ and $\mathbf{x}_0^{(i)} = 1$**,** for all **i**.

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) = \sum_{j=0}^{n} \boldsymbol{\theta}_j \mathbf{x}_j$$

In Vector form, $\mathbf{y} = \boldsymbol{\theta}^T\mathbf{x} = w^T\mathbf{x} + b$

- The **objective** is to get the **prediction** close to the original **target**: $\mathbf{y} \approx t$.

LINEAR REGRESSION: MODEL (EXAMPLE)

Dataset

X_0	Living Area (feet^2)	#Bedrooms	Price (X1000$)
1	2104	3	**400**
1	1600	3	**330**
1	2400	3	**369**
1	1416	2	**232**
1	3000	4	**516**
1	.	.	.
1	.	.	.

x_0 **Input/Features, x** **Target/Labels, *t***

General Representation:

$$\mathbf{y = f(x) = \sum_{j=1}^{n} w_j x_j + b}$$

Here,

Number of Features, n = **2.** Hence,

$$\mathbf{y^{(i)} = f(x^{(i)}) = \sum_{j=1}^{2} w_j x_j^{(i)} + b}$$

$$\mathbf{= w_1 x_1^{(i)} + w_2 x_2^{(i)} + b}$$

- **x_1 & x_2** are *Living Area* and *#Bedrooms* respectively.

As we know, $\boldsymbol{w}$ and **b** together are the **parameters, θ,** where **b = θ$_0$**. So, **$x_0^{(i)}$ = 1,** for all **i**.

$$\mathbf{y^{(i)} = f(x^{(i)}) = \sum_{j=0}^{n} \theta_j x_j^{(i)}}$$

$$\mathbf{= \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \theta_0}$$

The **Objective** is to find suitable **θ = [θ$_0$, θ$_1$, θ$_2$]** so that **y = f(x)** ≈ ***t***.

LINEAR REGRESSION: COST FUNCTION

In linear regression, The most common cost function used is the ***Mean Squared Error (MSE)***, also known as the *Squared Error Cost Function*.

$$\boldsymbol{J(\theta)} = \frac{\mathbf{1}}{\mathbf{2n}} \sum_{\mathbf{i=1}}^{\mathbf{n}} \left(\mathbf{f(x^{(i)})} - \mathbf{t^{(i)}} \right)^{2}$$

- $\boldsymbol{J(\theta)}$ is the Cost Function.
- $\mathbf{n}$ is the number of training examples.
- $\mathbf{f(x^{(i)})}$ is the predicted output value for the $\mathbf{i^{th}}$ example.
- $\mathbf{t^{(i)}}$ is the actual output value for the $\mathbf{i^{th}}$ example.
- The $\frac{\mathbf{1}}{\mathbf{2}}$ factor is just to make the calculations convenient.

Why MSE?: The squared error is chosen because it penalizes large errors more than small ones, leading to a smoother optimization surface. This makes it easier for optimization algorithms, like **Gradient Descent**, to find the minimum of the cost function.

LINEAR REGRESSION: COST FUNCTION

The **Mean Absolute Error (MAE)** is another cost function that can be used in linear regression. MAE measures the average of the absolute differences between predicted and actual values.

$$\boldsymbol{J(\theta)} = \frac{\mathbf{1}}{\mathbf{n}} \sum_{\mathbf{i=1}}^{\mathbf{n}} | \mathbf{f(x^{(i))} - t^{(i)}} |$$

Why MAE?: MAE is less sensitive to outliers compared to MSE because it does not square the errors. This means that large errors have a linear impact on MAE, rather than an exponential one. As a result, MAE can be a better choice when the data is expected to have outliers, and we don't want those outliers to disproportionately influence the model.

Limitation: While Gradient Descent can still be used to minimize MAE, the optimization process is more complex compared to MSE. The MAE cost function is not differentiable at the point where the error is zero, which can make the gradient descent process slower or less stable, hence modifications are necessary.

LINEAR REGRESSION: OPTIMIZATION

The **Objective** in linear regression is to find the parameters $\boldsymbol{\theta}$ that minimize the cost function $\boldsymbol{J(\theta)}$**.** Using an *Iterative Algorithm*(e.g. Gradient Descent) is one way to do so.

Gradient Descent: It is a very commonly used **Optimization Algorithm** for the above purpose. It iteratively updates the parameters $\boldsymbol{\theta}$ by moving in the direction that reduces the cost function the most. It uses derivatives to deduce that direction:

$$\boldsymbol{\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j} = \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} \left(f(x^{(i)}) - t^{(i)} \right) . X_j^{(i)}}$$

- $\boldsymbol{\alpha}$ is the **learning rate**, controlling the size of the step(changing rate).
- $\boldsymbol{\frac{\partial J(\theta)}{\partial \theta_j}}$ is the partial derivative of the cost function with respect to $\boldsymbol{\theta_j}$, representing the ***Gradient***.

Convergence: The optimization algorithm converges when further updates to the parameters $\boldsymbol{\theta}$ result in little to no change in the cost function $\boldsymbol{J(\theta)}$**.** At this point, the model has ideally found the **optimal parameters** that minimize the prediction error.

LINEAR REGRESSION: OPTIMIZATION

An *alternative* optimization method is using the Normal Equation**.** This method is a *Direct Solution* to the Optimization problem.

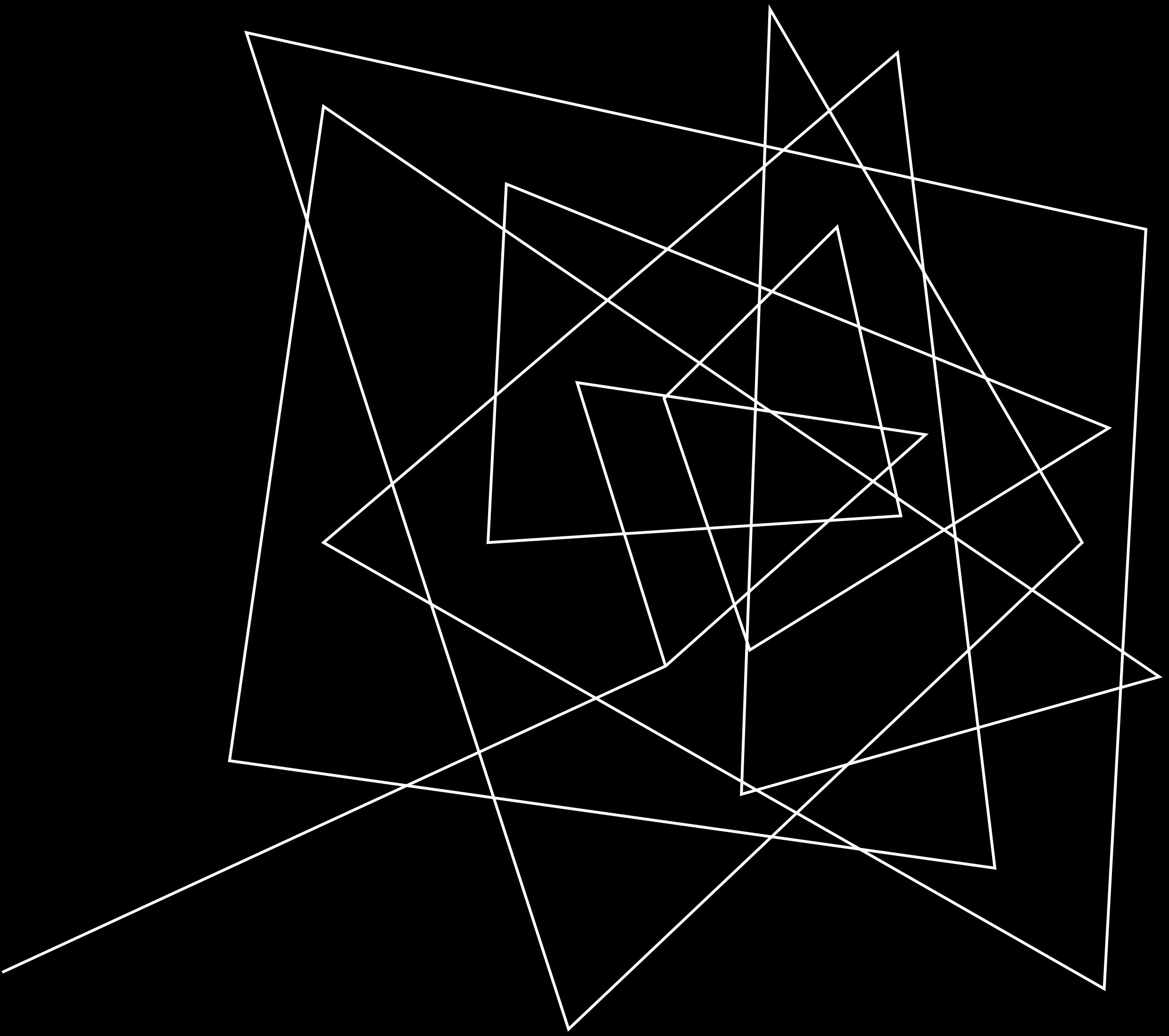
Normal Equation: The Normal Equation is a closed-form solution for finding the optimal parameters by setting the **Gradient = 0**, without needing an iterative process like Gradient Descent. The Normal Equation is given below:

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

- $\boldsymbol{X}$ is the matrix of input features.
- $\boldsymbol{y}$ is the vector of target values.
- $\boldsymbol{\theta}$ represents the vector of the parameters (weights) of the model.

Advantages: No need to choose a learning rate. Direct computation leads to the optimal solution in a **single step**.

Disadvantages: Computationally expensive for large datasets because it requires computing the inverse of the matrix $\boldsymbol{X}^T\boldsymbol{X}$, which is $\boldsymbol{O(n^3)}$ in complexity. Not suitable for datasets with a large number of features (high-dimensional data).



GRADIENT DESCENT

& ITS VARIATIONS

CALCULUS REVIEW: DERIVATIVES

Definition: A derivative represents the rate at which a function changes as its input changes. Mathematically, the derivative of a function $f(x)$ with respect to x is denoted as $f'(x)$ or $\frac{df(x)}{dx}$. ***The derivative at a point gives the slope of the tangent to the function at that point.***

Example: If $f(x) = x^2$, then the derivative $f'(x) = 2x$. This means that the slope of the function x^2 at any point x is $2x$.

Significance: *In machine learning, derivatives are used to determine the direction in which a function is increasing or decreasing. This is key in optimization, where the goal is often to find the minimum (or maximum) of a cost function. By taking the derivative of the cost function,* ***direction of steepest descent can be determined to move the parameters & decrease the cost.***

CALCULUS REVIEW: PARTIAL DERIVATIVES

Definition: Partial derivatives extend the concept of a derivative to functions with multiple variables. If a function $\boldsymbol{f(x,y)}$ depends on more than one variable, the partial derivative with respect to $\boldsymbol{x}$ is denoted as $\boldsymbol{\frac{\partial f(x,y)}{\partial x}}$**,** and with respect to $\boldsymbol{y}$, it's denoted as $\boldsymbol{\frac{\partial f(x,y)}{\partial y}}$ **.** *The partial derivative with respect to one variable is found by holding the other variables constant.*

Example: If $\boldsymbol{f(x, y) = x^2 + y^2}$, then the partial derivatives are:

$$\boldsymbol{\frac{\partial f(x,y)}{\partial x} = 2x \quad \& \quad \frac{\partial f(x,y)}{\partial y} = 2y}$$

Significance: *In machine learning models with* ***multiple parameters****, partial derivatives are used to understand how the cost function changes with respect to each parameter. This information is crucial when updating parameters during optimization, as* ***each parameter needs to be adjusted in the direction that reduces the cost function, and partial derivatives give that direction.***

CALCULUS REVIEW: GRADIENTS

Definition: The **Gradient** is **a vector** that **contains all the partial derivatives** of a function with respect to its variables. If $f(x,y)$ is a function of two variables, the gradient is:

$$\nabla f(x,y) = \left(\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y} \right)$$

The Gradient vector points in the direction of the steepest Ascent (the direction in which the function increases most rapidly).

Example: If $f(x, y) = x^2 + y^2$, then the gradient is:

$$\nabla f(x,y) = (2x, 2y)$$

Significance: *In optimization, particularly in Gradient Descent,* ***the negative of the gradient points in the direction of the steepest descent*** *(the direction in which the function decreases the most). This is why the gradient is used to update the parameters in optimization algorithms. It provides the most efficient direction to reduce the cost function.*

CALCULUS: WHY IT MATTERS IN ML

- **Finding Minima/Maxima:** Optimization in machine learning often involves finding the minimum of a cost function, which represents the best set of model parameters. ***Derivatives & gradients are used to identify the direction in which the cost function decreases most rapidly***, guiding the optimization process. Remember, ***at the Minima or Maxima, the derivative(slope) of a function = 0.***

- **Efficiency:** Knowing the gradient allows for efficient updates of the parameters. Rather than trying random directions or moving blindly, the gradient gives a clear path towards improvement.

- **Convergence:** The use of derivatives and gradients ensures that optimization algorithms like Gradient Descent can converge to an optimal solution, even in complex, high-dimensional spaces.

THE OPTIMIZATION PROBLEM

The **Optimization Problem** in Linear Regression and most other ML algorithms is to find the suitable parameters **Θ = [$\Theta_{0,}$ $\Theta_{1,}$ $\Theta_{2,\ldots}$]** that minimize the cost function $\boldsymbol{J(\theta)}$**.**

There are **2** options:

1. Direct Solution: Taking derivatives of the cost function and setting them to **0**. And then finding the solutions for **Θ** using this relation.

$$\frac{\partial J(\theta)}{\partial \theta} = 0$$

However, Many times, we do not have a *direct solution*. Also, it can become extremely expensive computationally with large datasets with many features.

2. Iterative Algorithm: When we apply an update repeatedly until some criterion is met, it’s called an *Iterative Algorithm*. In this case, initializing the weights to something reasonable(e.g. all zeros) and repeatedly adjusting them in the direction of steepest descent can lead to the minimization of the cost function. This process is known as **Gradient Descent**.

MODEL INITIALIZATION

Model initialization refers to **the process of setting the initial values for the parameters** (weights and biases) of a machine learning model before training begins. *Proper initialization is crucial because it can significantly impact the speed of convergence and the likelihood of reaching a good solution during training.*

There are many Initialization techniques. Some of them are:

- **Zero Initialization:** All weights are initialized to zero. Simple but not recommended for most models, especially in neural networks, as It can make all neurons learn the same features.
- **Random Initialization:** Weights are initialized to small random values (often from a uniform or normal distribution). Common in linear models and neural networks.
- **Xavier Initialization:** Weights are initialized using a scaled uniform or normal distribution. Commonly used in deep neural networks, especially for layers with *sigmoid* or *tanh* activation functions.
- **He Initialization:** Similar to Xavier Initialization but scaled differently to suit layers with ReLU (Rectified Linear Unit) activation functions.
- **Pre-trained Initialization:** The model is initialized with weights from a pre-trained model on a related task. This is common in Transfer Learning.

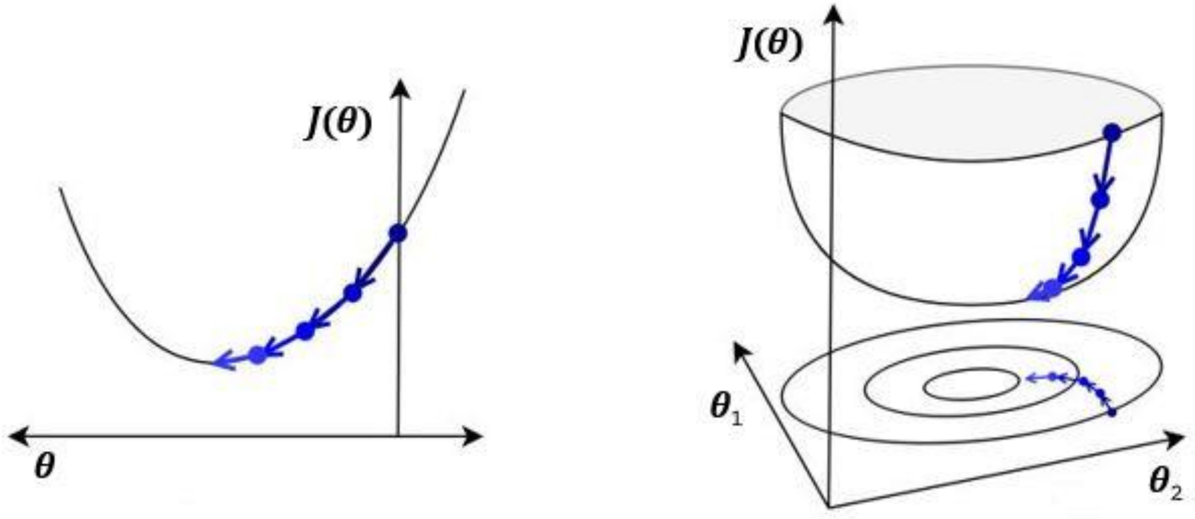
GRADIENT DESCENT

Gradient Descent is an iterative Optimization Algorithm that aims to *find the minimum of a Cost Function.* The cost function measures how well a machine learning model's predictions match the actual data. The objective is to *adjust the model parameters/weights to minimize this Cost Function.*

The Process consists of **two stages:**

- *Iteratively computing the **Gradient** to determine the direction in which the function decreases most steeply.*
- *Taking a step in that direction.*

Steps Towards the Minima (Where the value of $J(\theta)$ is minimum).



Source: https://www.cs.toronto.edu/~rgrosse/courses/csc311_f20/

GRADIENT DESCENT

Observe:

- *If* $\frac{\partial J(\theta)}{\partial \theta_j}$ ***> 0*** *(Positive slope), then increasing* $\boldsymbol{\theta_j}$ *increases* $\boldsymbol{J(\theta)}$*.*
- *If* $\frac{\partial J(\theta)}{\partial \theta_j}$ ***< 0*** *(Negative slope), then increasing* $\boldsymbol{\theta_j}$ *decreases* $\boldsymbol{J(\theta)}$*.*

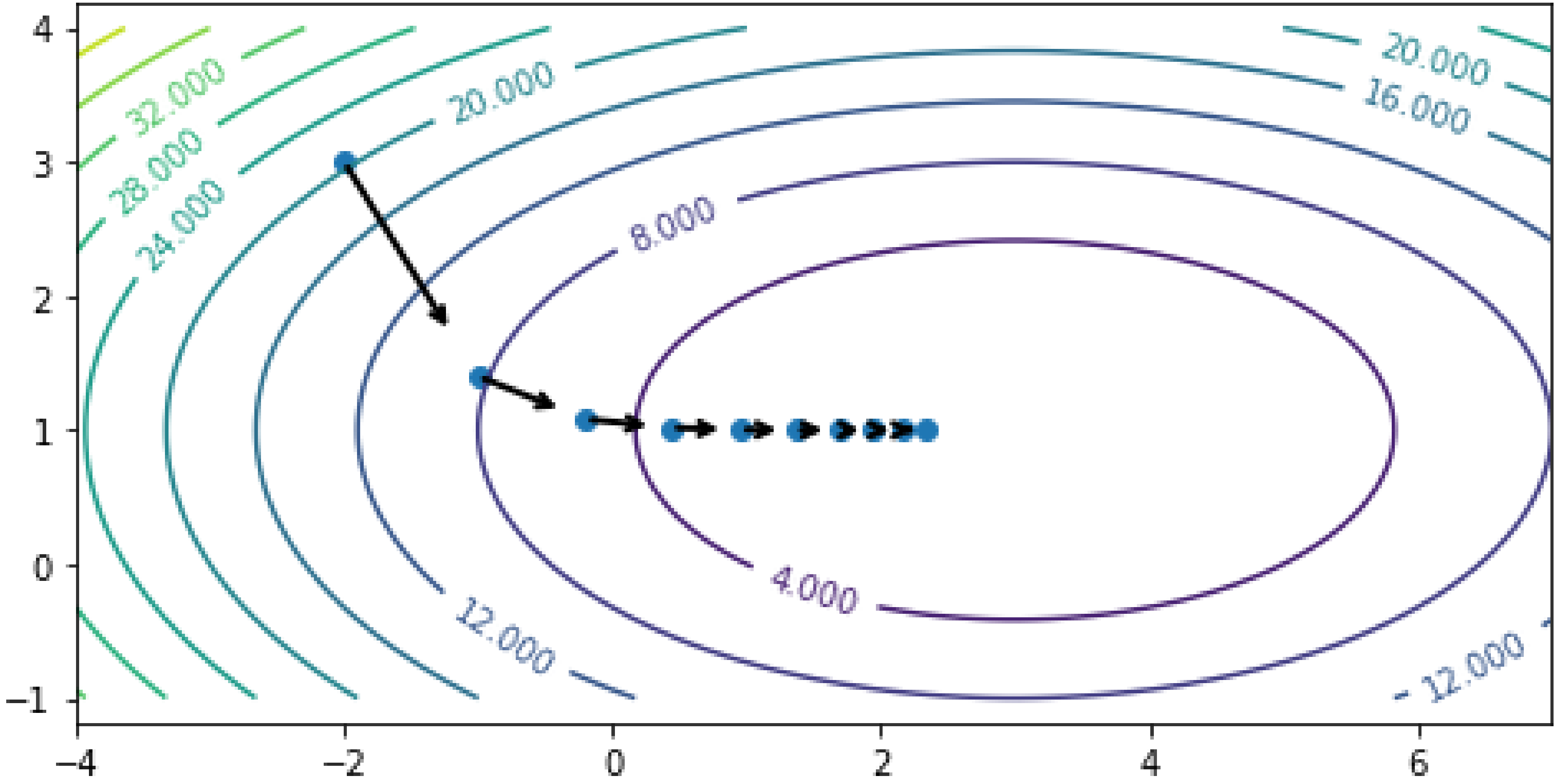
❑ The following update *always decreases the cost function* for small enough $\boldsymbol{\alpha}$ (unless $\frac{\partial J(\theta)}{\partial \theta_j}$ **= 0,** where it remains unchanged at the minima/maxima):

$$\boldsymbol{\theta_{j(new)}} \leftarrow \boldsymbol{\theta_{j(old)}} - \boldsymbol{\alpha} \frac{\partial J(\theta)}{\partial \theta_j}$$

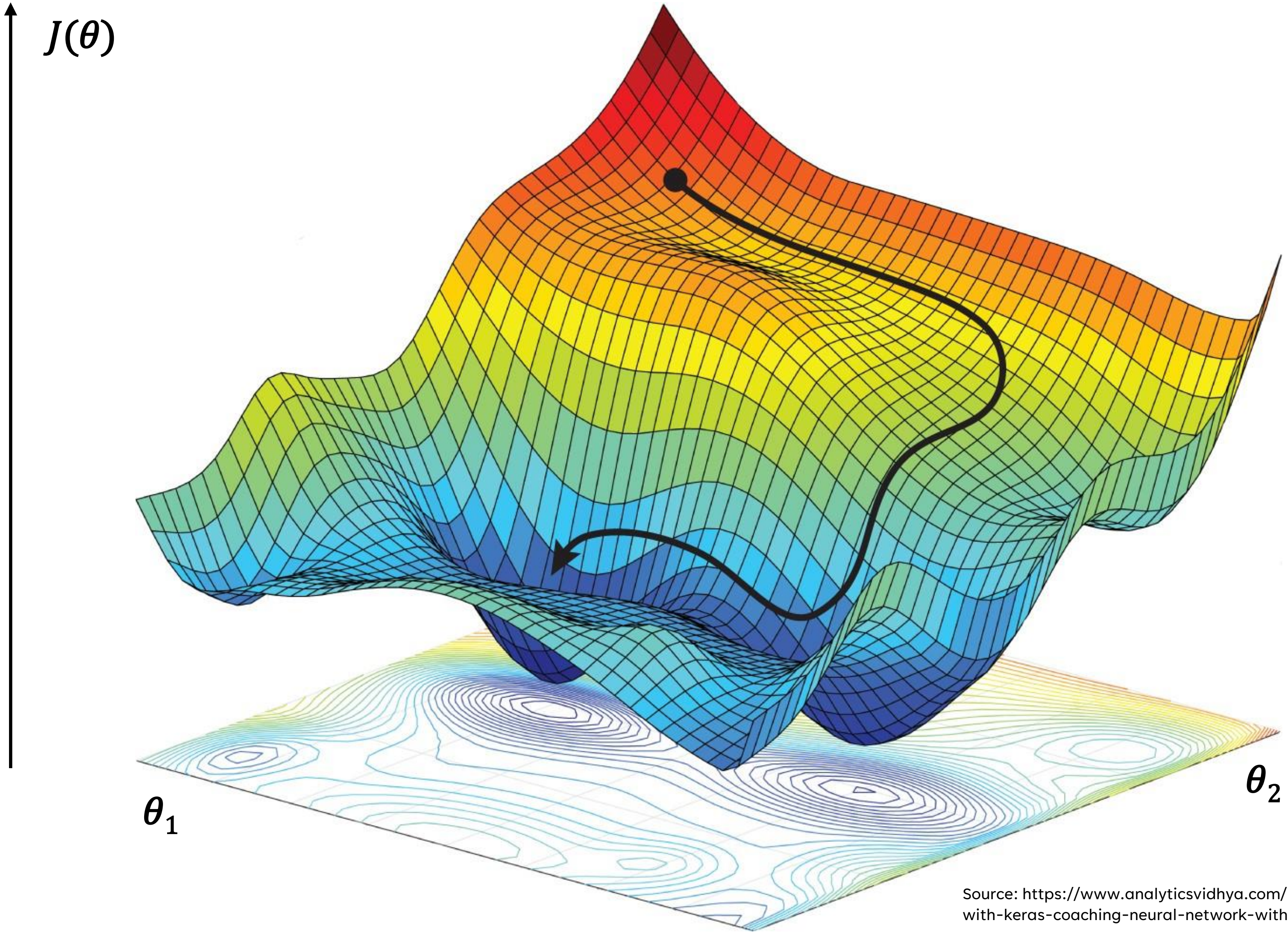
Learning Rate, $\boldsymbol{\alpha}$: It is a *hyperparameter* that controls the size of the steps taken to reach the minimum. The larger it is, the faster $\boldsymbol{\theta}$ changes. $\boldsymbol{\alpha}$ is always *greater than 0.*

- Tuning the learning rate is usually necessary, but the values are typically small, e.g., 0.01 or 0.0001.

GRADIENT DESCENT



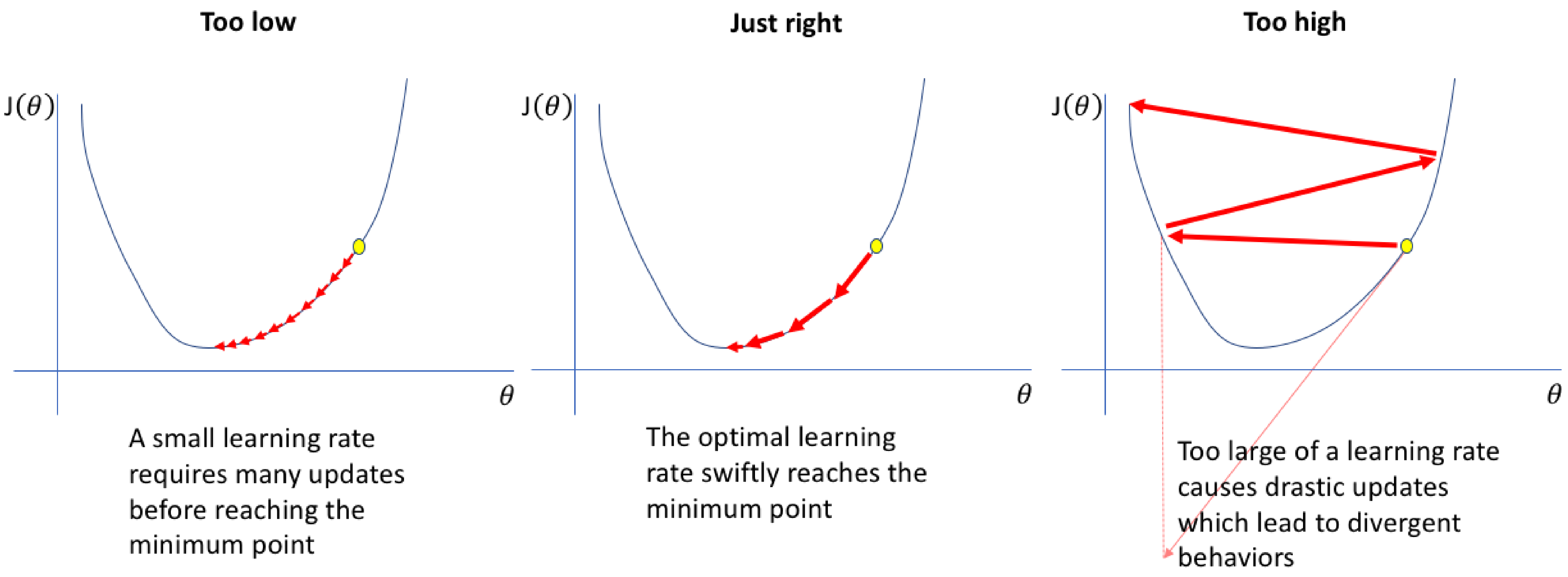
GRADIENT DESCENT (HIGHER DIMENSIONS)



Source: https://www.analyticsvidhya.com/blog/2021/10/deep-learning-with-keras-coaching-neural-network-with-keras-with-code/

GRADIENT DESCENT: LEARNING RATE

Selecting the optimal learning rate **α** is critical and often involves experimentation and fine-tuning. Here’s what happens with different learning rates:



Too low

A small learning rate requires many updates before reaching the minimum point

Just right

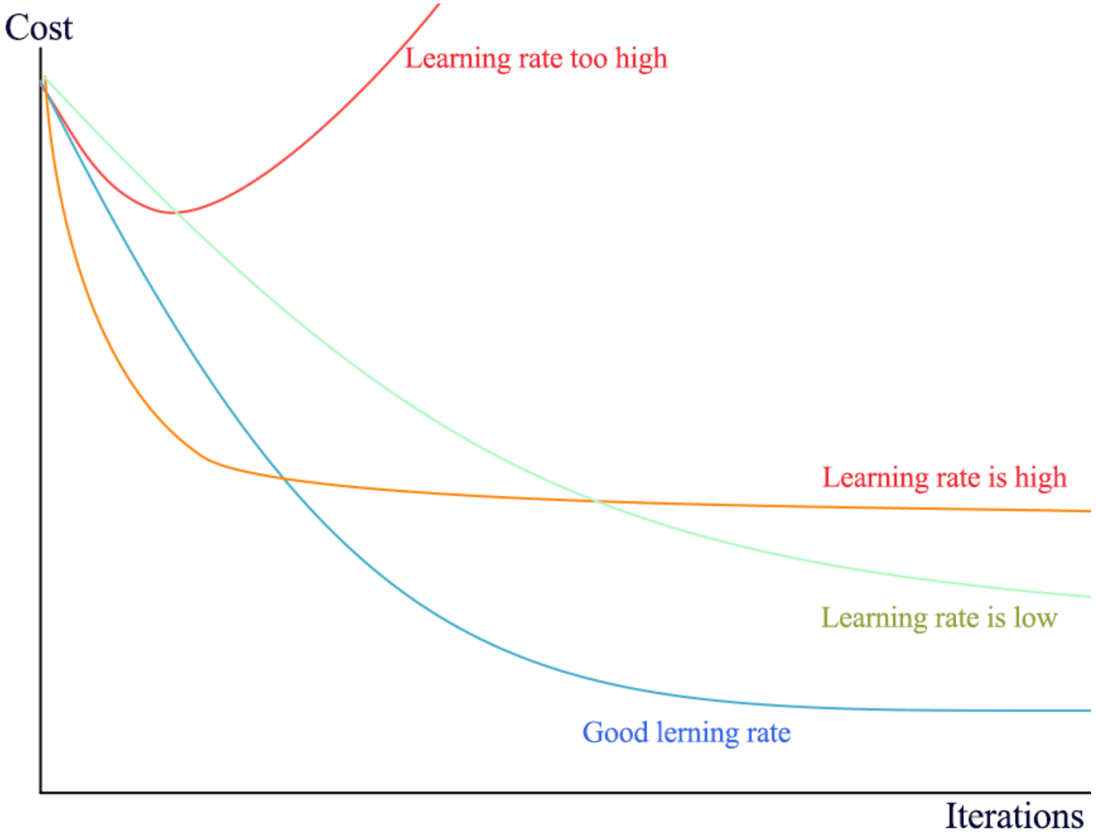
The optimal learning rate swiftly reaches the minimum point

Too high

Too large of a learning rate causes drastic updates which lead to divergent behaviors

GRADIENT DESCENT: LEARNING RATE

The value of the Learning Rate **α** affects the training process & convergence heavily, as we can see below. Techniques such as **manual tuning**, **learning rate schedules**, and **adaptive learning rates** can help in finding an effective learning rate for a given problem.



BATCH GRADIENT DESCENT

Description: In **Batch Gradient Descent**, the gradient is computed **using the entire training dataset**. The parameters are updated after summing up & averaging all the errors across the entire dataset.

Advantages

- Converges to the global minimum for *convex* cost functions.
- Stable updates since the gradient is averaged over the entire dataset.

Disadvantages

- Slow for large datasets since the gradient is computed on the entire dataset in each iteration.
- Requires significant memory to store the entire dataset in memory.

STOCHASTIC GRADIENT DESCENT (SGD)

Description: Unlike Batch Gradient Descent, which uses the entire dataset to compute the gradient, **Stochastic Gradient Descent (SGD)** updates the parameters using **only one training example at a time**. Instead of summing up the errors, it updates the parameters immediately after each example is processed.

Advantages

- Faster and more memory-efficient for large datasets.
- Can escape local minima due to its noisy updates.

Disadvantages

- Less stable; the cost function may fluctuate rather than smoothly decrease.
- Can’t take advantage of Vectorization & Matrix operations.

MINI-BATCH GRADIENT DESCENT

Description: Mini-Batch Gradient Descent is a compromise between Batch Gradient Descent and Stochastic Gradient Descent. It updates the parameters after computing the gradient **on a small subset of the data**, called a *mini-batch*.

Advantages

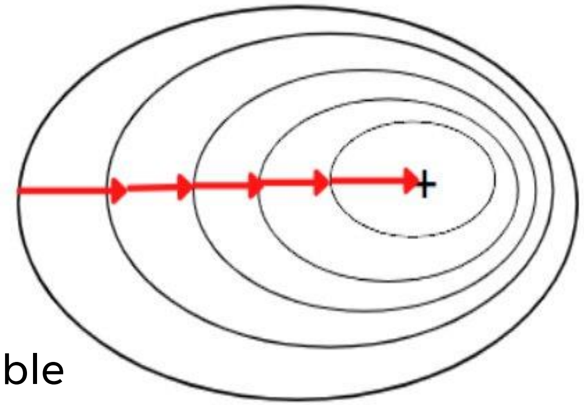
- Provides a balance between the efficiency of SGD and the stability of Batch Gradient Descent.
- Takes advantage of matrix operations, making it well-suited for parallel computation.

Disadvantages

- Requires tuning of the mini-batch size.
- Still may exhibit some noise in updates, though less than SGD.

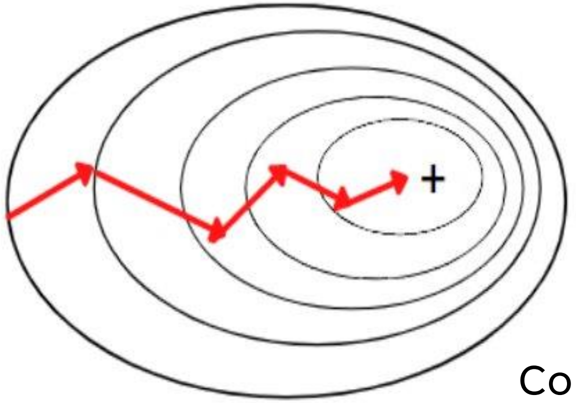
BATCH vs SGD vs MINI-BATCH

Batch Gradient Descent



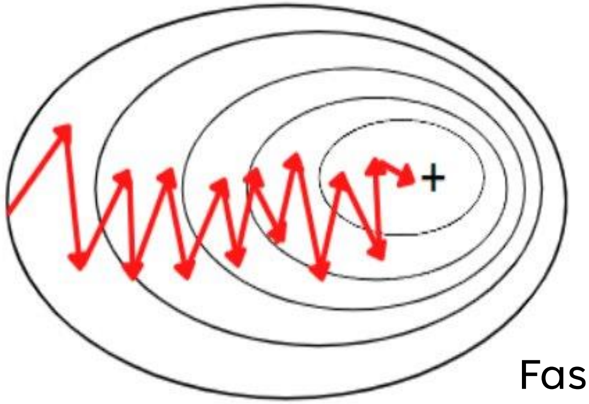
Slowest but Stable

Mini-Batch Gradient Descent



Compromise

Stochastic Gradient Descent



Fastest but Unstable

ADAM OPTIMIZER

Description: Adam (Adaptive Moment Estimation) optimizes parameters using adaptive learning rates that are computed based on the first and second moments (*Mean & Variance*) of the gradients. It combines the benefits of *AdaGrad* and *RMSProp*, making it effective for a wide range of machine learning problems.

Advantages

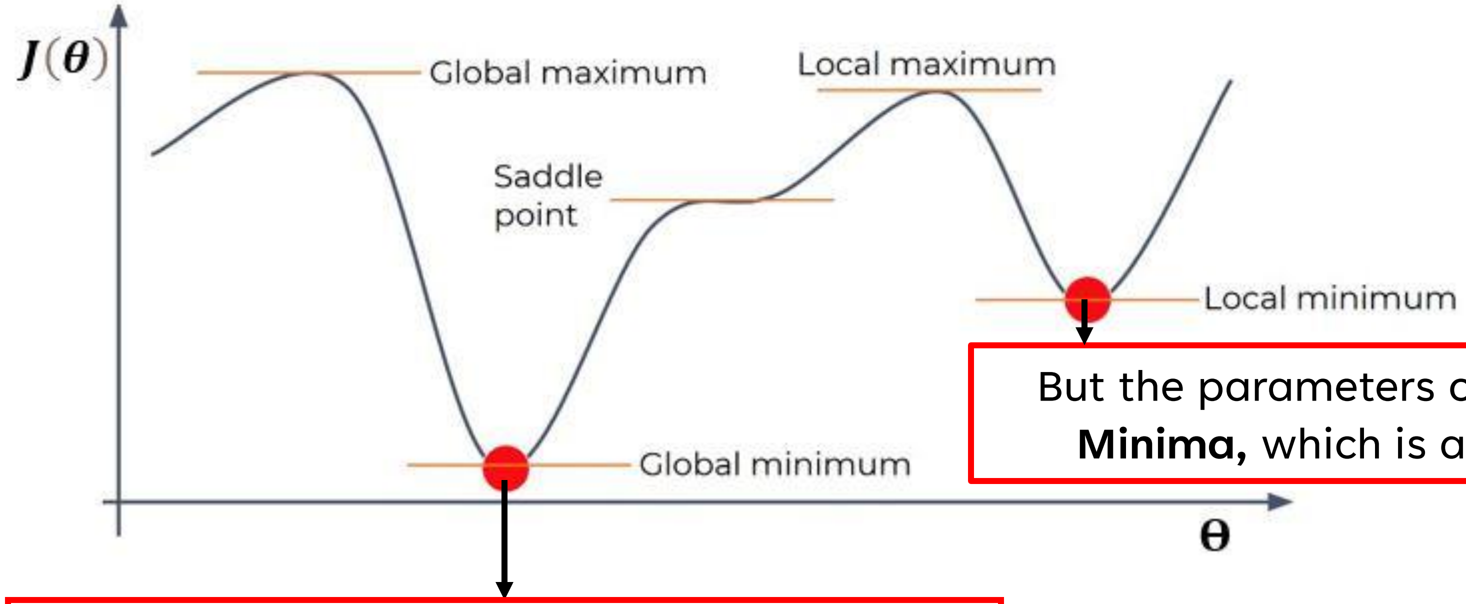
- Adaptive learning rates lead to efficient and faster convergence.
- Bias-correction improves performance in the early stages of training.
- Works well with sparse gradients and is robust to hyperparameter settings.

Disadvantages

- May not generalize as well as simpler optimizers like SGD.
- Can sometimes struggle with saddle points in the loss landscape.

LINEAR REGRESSION: CONVERGENCE

The optimization algorithm **converges** when further updates to the parameters $\boldsymbol{\theta}$ result in little to no change in the cost function $\boldsymbol{J(\theta)}$**.** At this point, the model has ideally(not always) **found the optimal parameters** that minimize the prediction error.



But the parameters can get stuck in **Local Minima,** which is also a **Critical Point**.

Ideally, the model parameters $\boldsymbol{\theta}$ will converge to **Global Minima**.

CONVEXITY

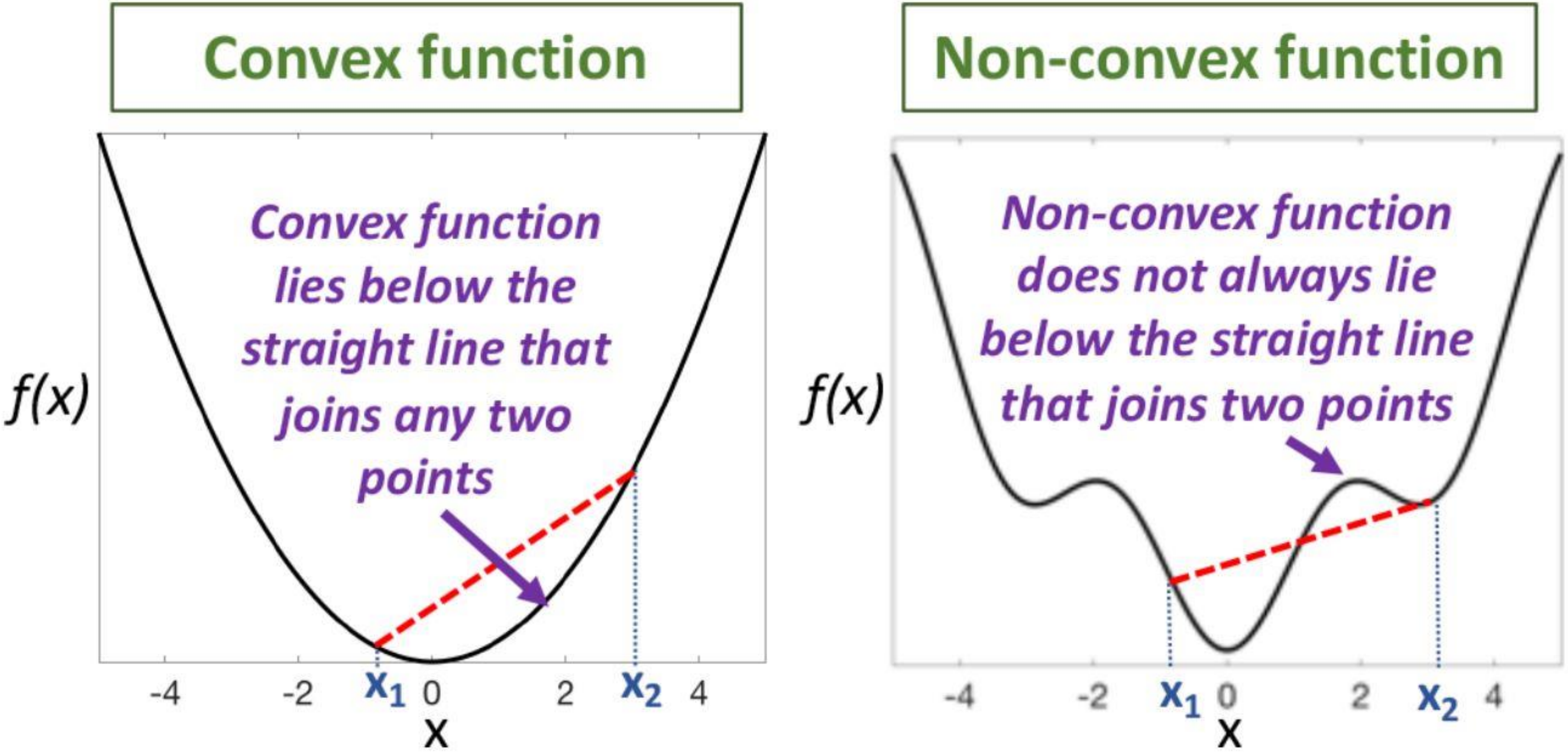
Convexity in optimization refers to the shape of the cost function. A function is convex if the line segment between any two points on its graph lies above or on the graph. **In machine learning, if the cost function is convex, it ensures that any local minimum is also a global minimum.** This property is crucial for guaranteeing that optimization algorithms like Gradient Descent converge to the global minimum. However, many real-world problems have non-convex cost functions, making global convergence challenging. Ensuring convexity can be difficult, especially in complex models like deep neural networks.

Intuitively: A Convex function is bowl-shaped.

How to Ensure Convexity:

- ***Simple Models:*** *Linear models like linear regression are inherently convex, making them easier to optimize.*
- ***Regularization:*** *Adding regularization terms can help maintain convexity.*
- ***Convex Loss Functions:*** *Choose loss functions known to be convex, such as Mean Squared Error (MSE) or logistic loss for binary classification.*

CONVEXITY



Convex function

Convex function lies below the straight line that joins any two points

Non-convex function

Non-convex function does not always lie below the straight line that joins two points

Source: https://qiml.radiology.wisc.edu/wp-content/uploads/sites/760/2022/12/notes_023_Convex.pdf

VECTORIZATION

Vectorization in machine learning refers to the process of converting operations that would typically be performed in *loops* into *vector or matrix operations*. This is done to take advantage of efficient numerical computation libraries (e.g., NumPy) that are optimized for handling such operations in a single step, leveraging low-level optimizations and **parallelism**.

Advantages:

- **Speed**: Vectorized operations are much faster because they minimize the overhead of looping and make use of highly optimized linear algebra libraries.
- **Simplicity**: Code becomes cleaner and more concise, reducing the likelihood of errors.
- **Parallelism**: Utilizes the underlying hardware better, often leveraging CPU or GPU parallelism.

Disadvantages:

- **Memory Usage:** Vectorization can increase memory usage because entire datasets need to be loaded into memory at once.
- **Complexity in Implementation:** Some complex operations may be less intuitive to vectorize, requiring more advanced knowledge of linear algebra.

VECTORIZATION (EXAMPLE)

Notation-wise, Cost Function, $J(\theta) = \frac{1}{2n}\sum_{i=1}^{n}(\mathrm{y}^{(i)} - \mathrm{t}^{(i)})^2$ gets messy if we expand $\mathbf{y}^{(i)}$:

$$J(\theta) = \frac{1}{2n}\sum_{i=1}^{n}((\sum_{j=1}^{D}\boldsymbol{w}_{j}x_{j}^{(i)} + b) - t^{(i)})^2$$

The Python code equivalent is to compute the prediction using a *for loop*:

```
# Non-vectorized prediction and cost calculation
for i in range(n):
    y_pred[i] = np.dot(X[i], w) + b  # Compute prediction for each example

# Calculate Mean Squared Error (MSE)
cost = 0
for i in range(n):
    cost += (y_pred[i] - y_true[i]) ** 2
cost = cost / (2 * n)
```

VECTORIZATION (EXAMPLE)

Excessive super/sub scripts are hard to work with, and Python loops are slow, so we **Vectorize** algorithms by expressing them in terms of **vectors** and **matrices**.

$$\boldsymbol{w} = [\, \boldsymbol{w}_1, \boldsymbol{w}_2, \dots , \boldsymbol{w}_{\mathrm{D}} \,]^{\mathrm{T}} \qquad \mathbf{x} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_D^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_D^{(2)} \\ \dots & \dots & \dots & \dots \\ x_1^{(n)} & x_2^{(n)} & \dots & x_D^{(n)} \end{bmatrix}^{\mathrm{T}}$$

$$\boldsymbol{J} = \boldsymbol{w}^{\mathrm{T}}\mathbf{x} + \mathrm{b}$$

In this case, Python code implementation is simpler and executes much faster:

```
# Vectorized prediction
y_pred = np.dot(X, w) + b  # Compute predictions for all examples at once

# Vectorized Mean Squared Error (MSE)
cost = (1 / (2 * n)) * np.sum((y_pred - y_true) ** 2)
```

VECTORIZATION (EXAMPLE)

Getting rid of the **bias** term and simply adding another column with constant 1s make the implementation even simpler and more concise. It exploits **Vectorization** even better.

$$\boldsymbol{\theta} = [\,\theta_0, \theta_1, \dots, \theta_D\,]^T \qquad \mathbf{x} = \begin{bmatrix} 1 & x_1^{(1)} & \dots & x_D^{(1)} \\ 1 & x_1^{(2)} & \dots & x_D^{(2)} \\ \dots & \dots & \dots & \dots \\ 1 & x_1^{(n)} & \dots & x_D^{(n)} \end{bmatrix}^T$$

$$J = \boldsymbol{\theta}^T \mathbf{x}$$

In this case, Python code implementation is even simpler:

```
# Vectorized prediction
y_pred = np.dot(X, theta)  # Compute predictions for all examples at once

# Vectorized Mean Squared Error (MSE)
cost = (1 / (2 * n)) * np.sum((y_pred - y_true) ** 2)
```

EXAMPLE PROBLEMS

EXAMPLE 1: SINGLE FEATURE

Let's solve a simple linear regression problem analytically using Gradient Descent with just **2 iterations**. We'll predict house prices based on the area (single feature).

Problem Setup:

- Feature (**X**): Area of the house in square feet.
- Target (**t**): Price of the house in $1000s.

Living Area (feet^2)	Price (X1000$)
1000	**300**
1500	**450**
2000	**500**

$$\underbrace{}_{X} \quad \underbrace{}_{t}$$

Step 1: Model/Algorithm Selection

Let us choose **Linear Regression** for this problem. So, for **1** feature,

$$y^{(i)} = f(X^{(i)}) = \sum_{j=0}^{1} \theta_j X_j^{(i)}$$

$$y^{(i)} = \theta_1 X_1^{(i)} + \theta_0$$

Step 2: Cost Function Selection

Let us choose **MSE** for this problem. So,

$$J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (f(X^{(i)}) - t^{(i)})^2 = \frac{1}{6} \sum_{i=1}^{3} ((\theta_1 X_1^{(i)} + \theta_0) - t^{(i)})^2$$

Step 3: Model Initialization

- Initial weight, $\theta_0 = 0, \theta_1 = 0$
- Learning Rate, $\alpha = 0.0000001$

EXAMPLE 1: SINGLE FEATURE

Living Area (feet^2)	Price (X1000$)
1000	300
1500	450
2000	500

X t

Step 4: Optimization (Iterative Training)

Let us use **Gradient Descent** to optimize in this case.

$$\theta_{0(new)} = \theta_{0(old)} - \alpha \frac{\partial J(\theta)}{\partial \theta_0}$$

$$\theta_{1(new)} = \theta_{1(old)} - \alpha \frac{\partial J(\theta)}{\partial \theta_1}$$

Note that, here,

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{3}\sum_{i=1}^{3} ((\Theta_1 X_1^{(i)} + \Theta_0) - t^{(i)})$$

$$\frac{\partial J(\theta)}{\partial \theta_1} = \frac{1}{3}\sum_{i=1}^{3} ((\Theta_1 X_1^{(i)} + \Theta_0) - t^{(i)}) . X_1^{(i)}$$

Iteration 1:

Step 4.1: *Compute Predictions using Current Parameter (**θ**) value*

$$y_{pred} = 0 \times X + 0 = 0$$

EXAMPLE 1: SINGLE FEATURE

Living Area (feet^2)	Price (X1000$)
1000	300
1500	450
2000	500

X

t

Step 4: Optimization (Iterative Training)

Step 4.2: *Compute gradients:*

$$\text{Gradient for } \boldsymbol{\theta_0} = \frac{1}{3}((0-300)+(0-450)+(0-500))$$

$$= \frac{1}{3} \times -1250$$

$$= \mathbf{-416.33}$$

$$\text{Gradient for } \boldsymbol{\theta_1} = \frac{1}{3}((0-300)\times 1000+(0-450)\times 1500+(0-500)\times 2000)$$

$$= \frac{1}{3}(-300000-675000-1000000)$$

$$= \frac{1}{3} \times -1975000$$

$$= \mathbf{-658333.33}$$

EXAMPLE 1: SINGLE FEATURE

Living Area (feet^2)	Price (X1000$)
1000	300
1500	450
2000	500

$$\underbrace{\text{Living Area}}_{X} \quad \underbrace{\text{Price}}_{t}$$

Step 4: Optimization (Iterative Training)

Step 4.3: *Update parameters*

Using the update rule for **Gradient Descent:**

$$\boldsymbol{\theta}_1 = 0 - (0.0000001) \times -658333.33 = \mathbf{0.0658333}$$

$$\boldsymbol{\theta}_0 = 0 - (0.0000001) \times -416.67 = \mathbf{0.00004167}$$

Iteration 2:

Step 4.1: *Compute Predictions using Current Parameter (*$\boldsymbol{\theta}$*) value*

$$\mathbf{y_{pred} = 0.0658333 \times X + 0.00004167}$$

*For each **X**:*

For 1000 sq ft: $y_{pred} = 0.0658333 \times 1000 + 0.00004167 \approx \mathbf{65.8333}$

For 1500 sq ft: $y_{pred} = 0.0658333 \times 1500 + 0.00004167 \approx \mathbf{98.75}$

For 2000 sq ft: $y_{pred} = 0.0658333 \times 2000 + 0.00004167 \approx \mathbf{131.67}$

EXAMPLE 1: SINGLE FEATURE

Living Area (feet^2)	Price (X1000$)
1000	300
1500	450
2000	500

X

t

Step 4: Optimization (Iterative Training)

Step 4.2: *Compute gradients:*

Gradient for $\boldsymbol{\theta_0} = \frac{1}{3}((65.8333-300)+(98.67-450)+(131.67-500))$

$= \frac{1}{3} \times -1417.7533$

$\mathbf{=-472.584}$

Gradient for $\boldsymbol{\theta_1}$

$= \frac{1}{3}((65.8333-300)\times 1000+(98.67-450)\times 1500+(131.67-500)\times 2000)$

$= \frac{1}{3} \times -1389166.7$

$\mathbf{=-463055.56}$

EXAMPLE 1: SINGLE FEATURE

Living Area (feet²)	Price (X1000$)
1000	**300**
1500	**450**
2000	**500**

X **t**

Step 4: Optimization (Iterative Training)

Step 4.3: *Update parameters*

Using the update rule for **Gradient Descent:**

$$\boldsymbol{\theta}_1 = 0.0658333 - (0.0000001) \times -463055.56 = \mathbf{0.1121389}$$

$$\boldsymbol{\theta}_0 = 0.00004167 - (0.0000001) \times -472.584 = \mathbf{0.000088}$$

After 2 Iterations, the optimized parameters: $\boldsymbol{\theta_1} = \mathbf{0.1121389}$, $\boldsymbol{\theta_0} = \mathbf{0.000088}$

$$\mathbf{y_{pred} = 0.1121389 \times X + 0.000088}$$

*Predictions for each **X**:*

For 1000 sq ft: $y_{pred} = 0.1121389 \times 1000 + 0.000088 \approx \mathbf{112.14}$

For 1500 sq ft: $y_{pred} = 0.1121389 \times 1500 + 0.000088 \approx \mathbf{168.21}$

For 2000 sq ft: $y_{pred} = 0.1121389 \times 2000 + 0.000088 \approx \mathbf{224.28}$

EXAMPLE 1: SINGLE FEATURE

Living Area (feet^2)	Price (X1000$)
1000	300
1500	450
2000	500

$$\underbrace{}_{X} \quad \underbrace{}_{t}$$

Step 5: Model Evaluation

As we can see, the model significantly underestimates the house prices, predicting much lower values than the actual prices for all three areas. However, compared to the initial predictions, the predictions after 2 iterations were much closer to the original values. It shows that the model was being optimized properly. The predictions will improve significantly with more iterations, which would drive the weights closer to their optimal values, resulting in predictions that are much closer to the actual prices.

It's important to have an **Evaluation Metric** to quantify the performance of the Model. For Linear Regression, **Mean Squared Error (MSE)** is a common evaluation metric. In this case,

$$\textbf{MSE} = \frac{1}{6}\sum_{i=1}^{3}((y^{(i)} - t^{(i)})^2 \qquad \textit{[Discussed before as the cost function]}$$

$$= \frac{1}{6}((112.14-300)^2 + (168.21-450)^2 + (224.28-500)^2)$$

$$= \frac{1}{6}(35283.8596+79390.8041+76031.3184)$$

$$\approx \mathbf{31784.5}$$

This high value of MSE shows that that the model is still far from accurately predicting the house prices.

EXAMPLE 1: SINGLE FEATURE

Living Area (feet^2)	Price (X1000$)
1000	**300**
1500	**450**
2000	**500**

X **t**

Step 6: Model Tuning

After Evaluating the model before **Deployment**, if the performance is not up to the mark, it must go through the **Hyperparameter Tuning** process to enhance the performance. *Which Hyperparameters would you like to adjust in this case?*

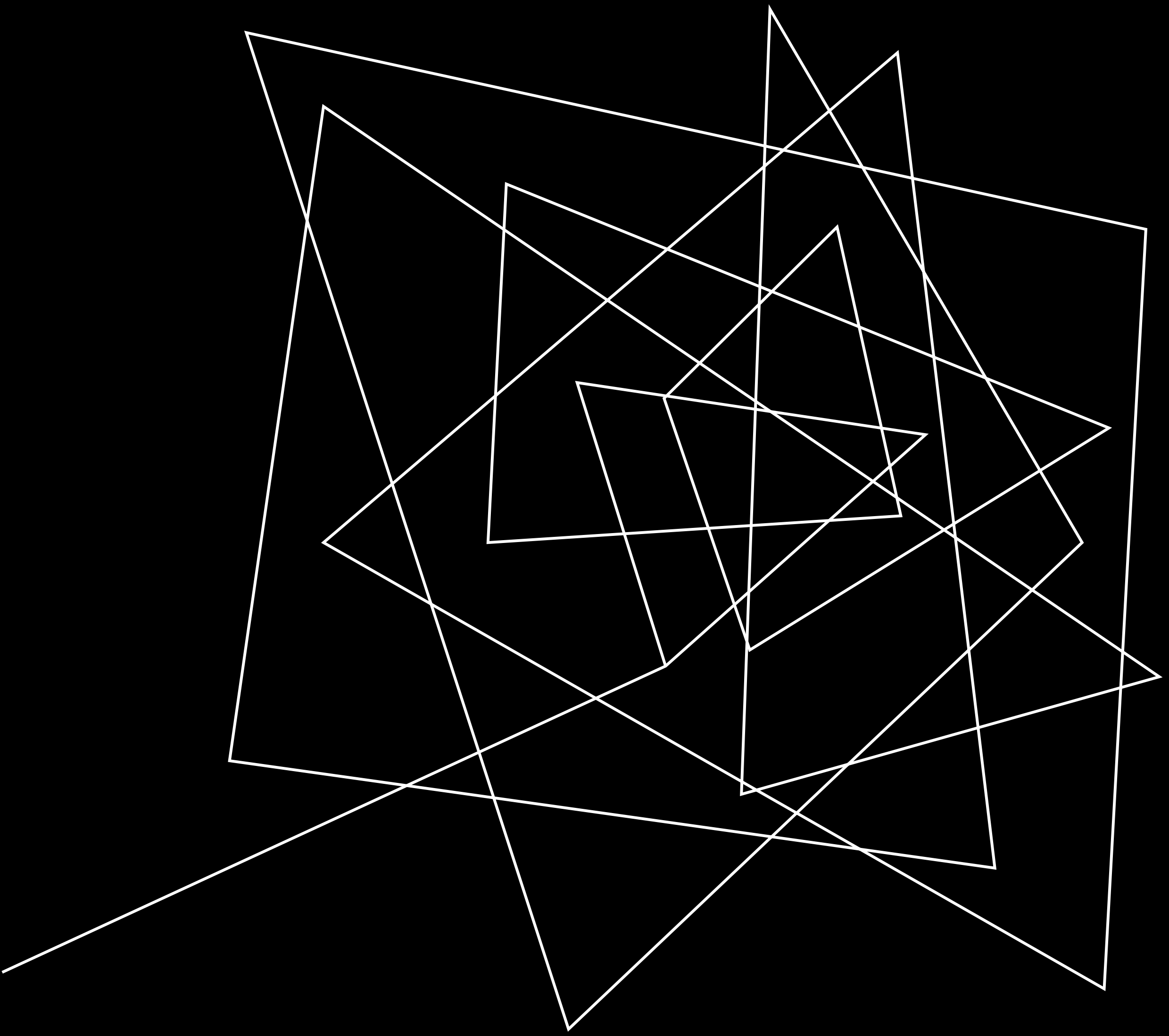
- Learning Rate, $\boldsymbol{\alpha}$
- Number of Iterations
- Initialization Method
- Optimization Algorithm *(SGD, maybe?)*

EXAMPLE 2: MULTIPLE FEATURES

Living Area (feet^2)	#Bedrooms	Price (X1000$)
2104	3	**400**
1600	3	**330**
2400	3	**369**
1416	2	**232**
3000	4	**516**

❑ **Find the optimal parameters** for a linear regression model to **predict house prices** based on the living area & the number of Bedrooms (multiple features) using the above dataset.

❑ Use that **Model** to **Determine the Price** of a house that features **4 Bedrooms** and **2200 feet^2 of Living Area.**



FEATURE MAPPING & POLYNOMIAL REGRESSION

LIMITATIONS OF LINEAR REGRESSION

- **Linearity Assumption:** Assumes a linear relationship between features and the target, which may not capture complex patterns.
- **Outlier Sensitivity:** Highly sensitive to outliers, which can distort predictions.
- **Independence Assumption:** Assumes observations are independent; violations can lead to unreliable estimates.
- **Multicollinearity:** High correlation between features can make the model unstable and hard to interpret.
- **No Feature Selection:** Doesn't inherently select relevant features, risking overfitting and complexity.

Some of the limitations of Linear Regression can be overcome with **Feature Mapping.** Let us discuss that in the next slides...

FEATURE MAPPING

Feature mapping is a powerful technique in machine learning that involves **transforming the original features into a new set of features**, often to enable the model to **capture more complex patterns in the data**. While it is commonly associated with transforming features into higher-dimensional spaces, feature mapping encompasses a broader range of transformations aimed at enhancing model performance.

Why Transform Features?

By creating new features from the original ones, a model that originally assumed linearity can now learn more complex patterns. It can be very useful **to capture non-linear relationships between the features and the target variable** while keeping the basic algorithm simple.

Example: For an **original feature x**, feature mapping might involve creating **new features** like **x^2, x^3, sin(x), e^x etc.**, turning a linear model into one that can fit a curve or oscillations.

KEY ASPECTS OF FEATURE MAPPING

- **Higher-Dimensional Spaces (Polynomial Features):** The most common form of feature mapping involves creating polynomial features. For example, given a feature $\boldsymbol{x}$, we can create $\boldsymbol{x^2}$, $\boldsymbol{x^3}$, and so on. *This transforms the data into a higher-dimensional space, allowing a linear model to capture non-linear relationships.* **Polynomial Regression** exploits this factor, where these higher-dimensional features allow the model to fit curves rather than just straight lines.

- **Non-Linear Transformations:** Beyond polynomials, feature mapping can involve applying non-linear functions to features, such as logarithms, exponentials, or trigonometric functions. For instance, transforming a feature $\boldsymbol{x}$ into **log($\boldsymbol{x}$), sin($\boldsymbol{x}$),** or $\boldsymbol{e^x}$. This is very useful in situations where the relationship between the features and the target is inherently non-linear, but not necessarily polynomial.

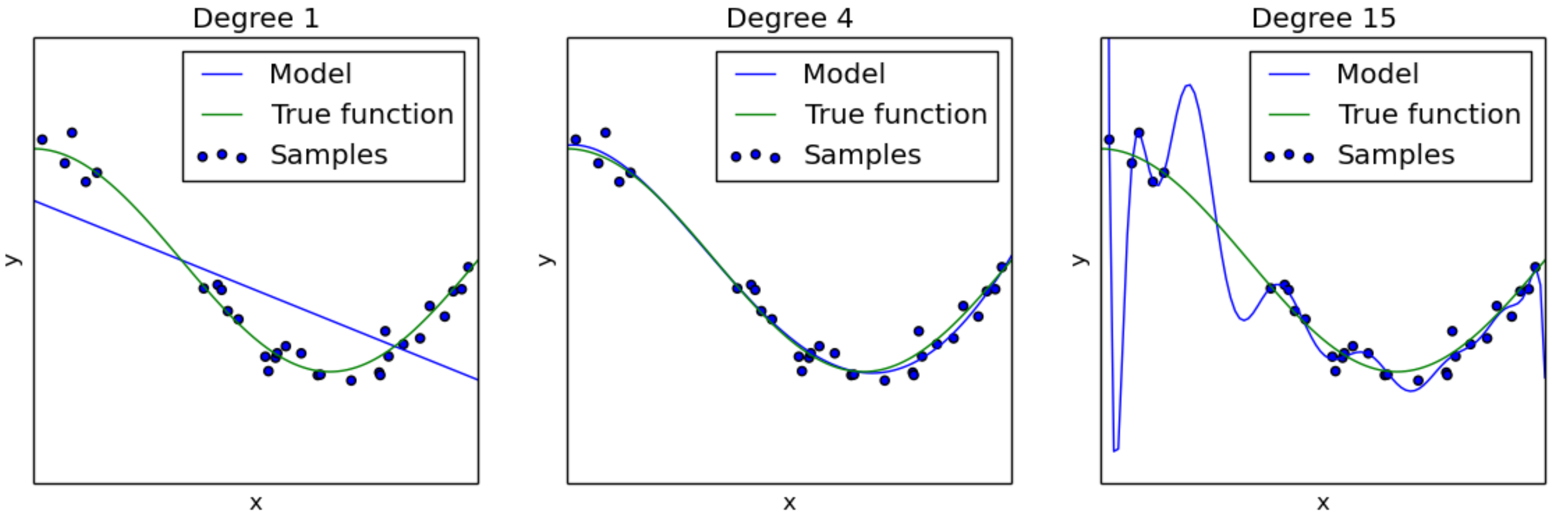
- **Interaction Features:** Creating interaction terms involves combining two or more features together to capture their combined effect on the target variable. For example, for two features $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$, a new feature $\boldsymbol{x_1 \times x_2}$ might be added. These features effectively exploit higher dimensions.

KEY ASPECTS OF FEATURE MAPPING

- **Kernel Methods:** In algorithms like **Support Vector Machines (SVMs)**, feature mapping is achieved through kernel functions. These functions implicitly map the data into a higher-dimensional space without explicitly computing the coordinates in that space, allowing for non-linear decision boundaries. *Non-linear SVMs use kernel methods to separate data points that are not linearly separable in the original feature space.*
- **Encoding Categorical Variables:** Feature mapping can also involve encoding categorical variables into numerical features, such as one-hot encoding or target encoding. These mappings transform categories into a form that machine learning models can understand.
- **Dimensionality Reduction:** Feature mapping can also involve reducing the dimensionality of data while preserving as much information as possible. Techniques like **Principal Component Analysis (PCA)** transform the original features into a new set of uncorrelated features (principal components) in a lower-dimensional space.

CHALLENGES OF FEATURE MAPPING

- **Overfitting:** Creating too many features, especially high-degree polynomial features, can lead to overfitting, where the model becomes too complex and captures noise rather than just the underlying pattern.
- **Increased Complexity:** More features mean a more complex model, which can be harder to interpret and may require more computational resources.
- **Feature Selection:** Not all generated features will be useful. Careful feature selection or regularization may be needed to prevent the model from becoming overwhelmed by irrelevant features.



POLYNOMIAL REGRESSION

Polynomial regression is a specific application of **Feature Mapping** where the input features are transformed into polynomial features. *It extends **Linear Regression** by adding polynomial terms (like x^2, x^3) to the model.* These models can capture non-linear relationships while maintaining the simplicity of linear regression settings.

Model

For a single feature x, a Polynomial Regression model might look like:

$$\mathbf{y = \theta_0 + \theta_1 x^1 + \theta_2 x^2 + \ldots + \theta_M x^M}$$

*Where, **M** = Degree of the Polynomial*

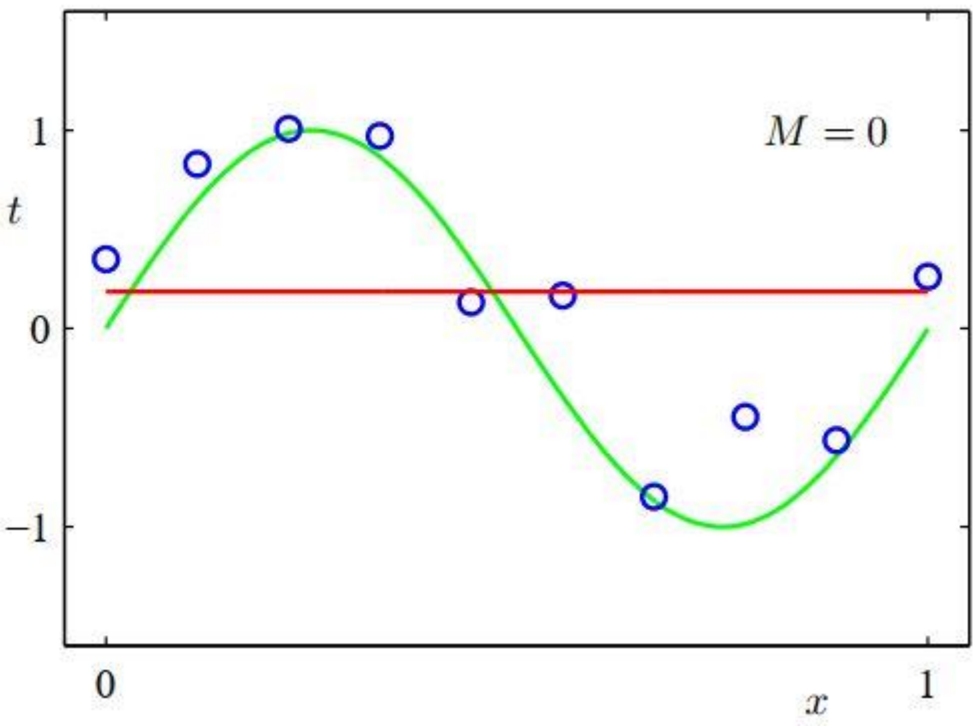
Now, for a Polynomial Regression model of degree **M = 2** with 2 features x_1 and x_2:

$$\mathbf{y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1{}^2 + \theta_4 x_2{}^2 + \theta_5 x_1 x_2}$$

***Note that,** the interaction term $\mathbf{x_1x_2}$ can also be included in the model of degree = 2 since $\mathbf{x_1x_2}$ is a degree 2 term.*

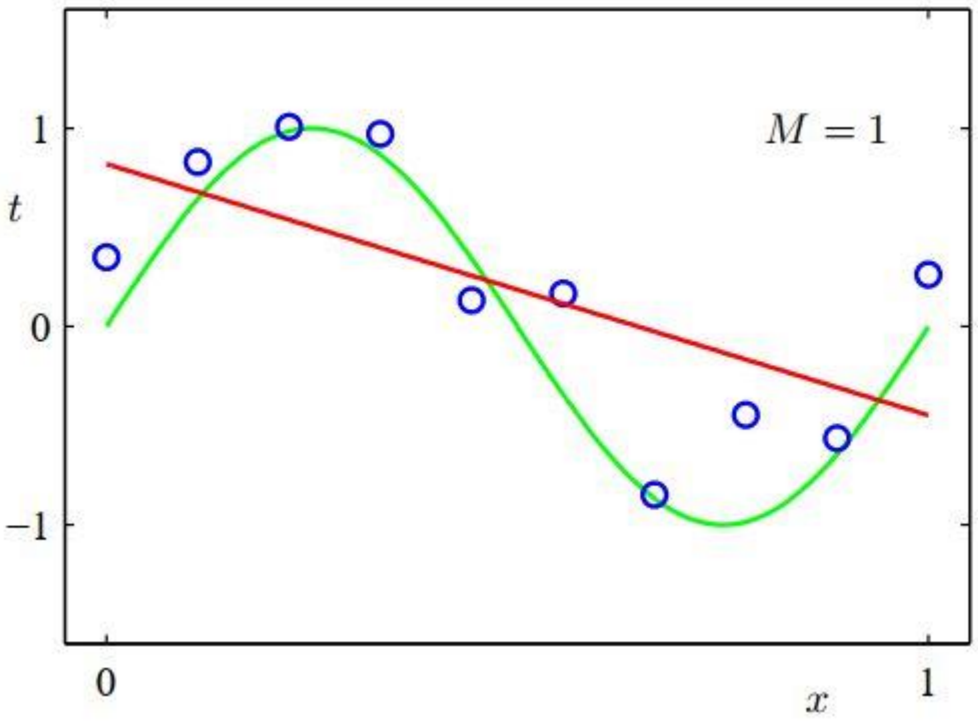
POLYNOMIAL REGRESSION

$y = \theta_0$



Just a Constant

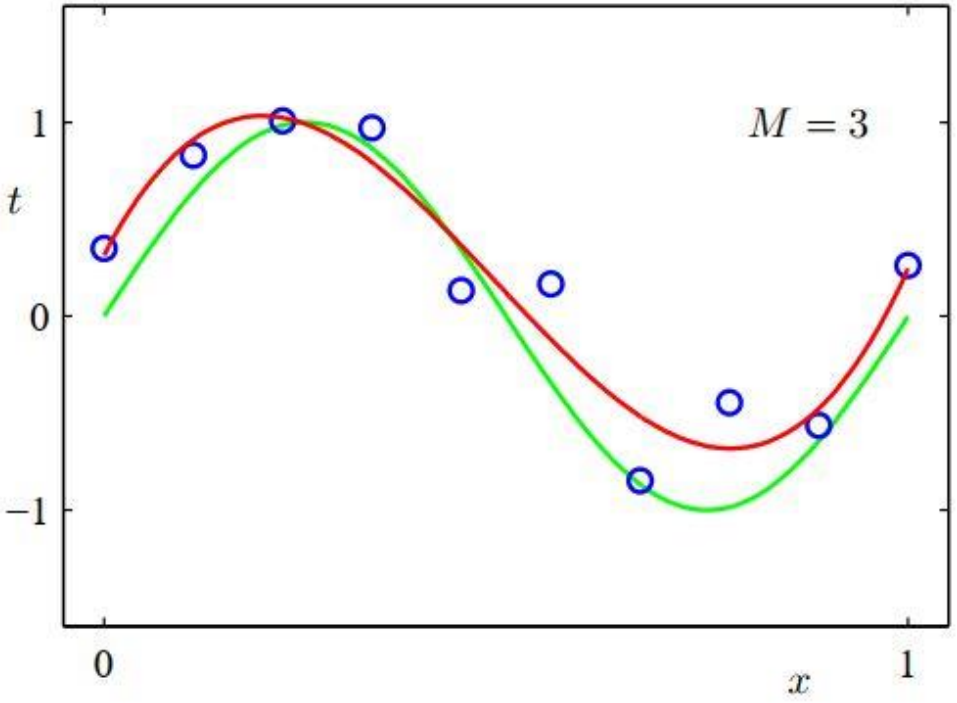
$y = \theta_0 + \theta_1 x^1$



Model Too Simple (Underfitting)

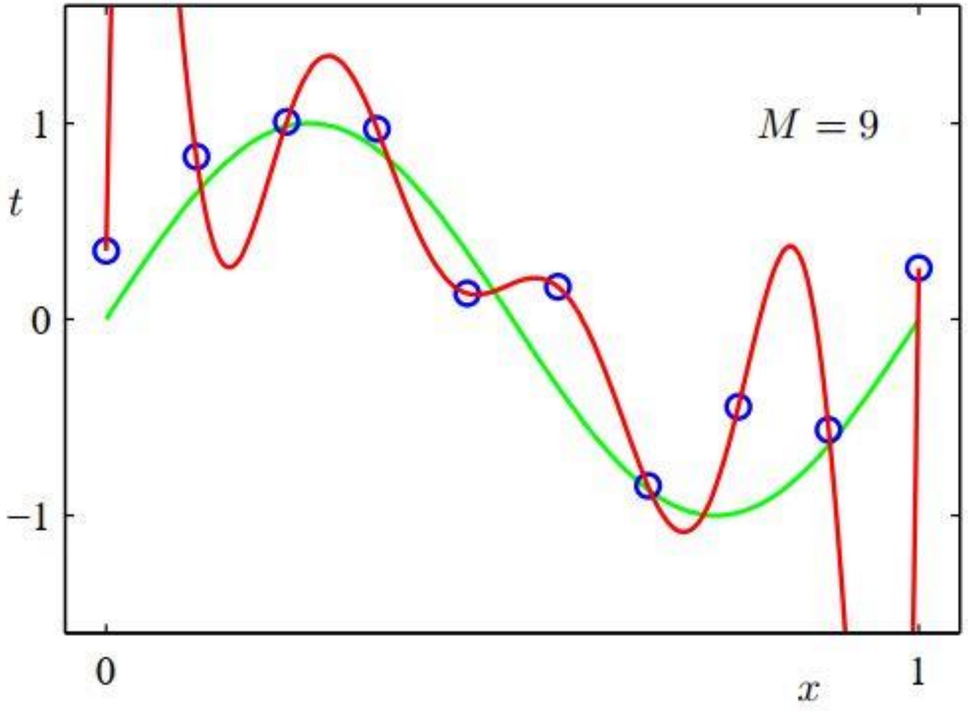
POLYNOMIAL REGRESSION

$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$



Good Fit

$y = \theta_0 + \theta_1 x^1 + \theta_2 x^2 + \ldots + \theta_9 x^9$



Model Too Complex (Overfitting)

IS TOO GOOD A FIT A GOOD THING?

- Clearly, when a model becomes too complex, it tries to fit each of its Training Data.
- In the process, it learns to captures noise present in that data and fits them too.
- But is it a good thing?
- **How well will that model perform when it sees data it hasn't trained on?**
- If it doesn't perform well, what to do? **How to evaluate the model that performs well with its training data but fails in the Real-world scenario?**

Let us find the answers in the next lecture…

REFERENCES

1. University of Toronto - CSC411/2515: Machine Learning and Data Mining (https://www.cs.toronto.edu/~rgrosse/courses/csc311_f20/)
2. Stanford cs231n Course Notes (https://cs231n.github.io/)
3. CMU's Introduction to Machine Learning (10-601) Lectures (https://www.cs.cmu.edu/%7Etom/10701_sp11/lectures.shtml)
4. MIT OpenCourseWare: 6.867 Machine Learning (https://people.csail.mit.edu/dsontag/courses/ml16/)
5. Applied ML course at Cornell and Cornell Tech (https://github.com/kuleshov/cornell-cs5785-2020-applied-ml/)
6. Pattern Recognition and Machine Learning, Christopher Bishop

THANK YOU