Abstract geometric lines forming various polygons and shapes, primarily in the upper left and center of the slide.

# INTRODUCTION TO CLASSIFICATION, LOGISTIC REGRESSION & THE CONCEPT OF OVERFITTING

BY

SAIFUL BARI IFTU

LECTURER, DEPT. OF CSE, BRAC UNIVERSITY

# CONTENTS

Classification

Binary Classification & Probabilistic Models

The Sigmoid Function

Logistic Regression

Multi-Class Classification

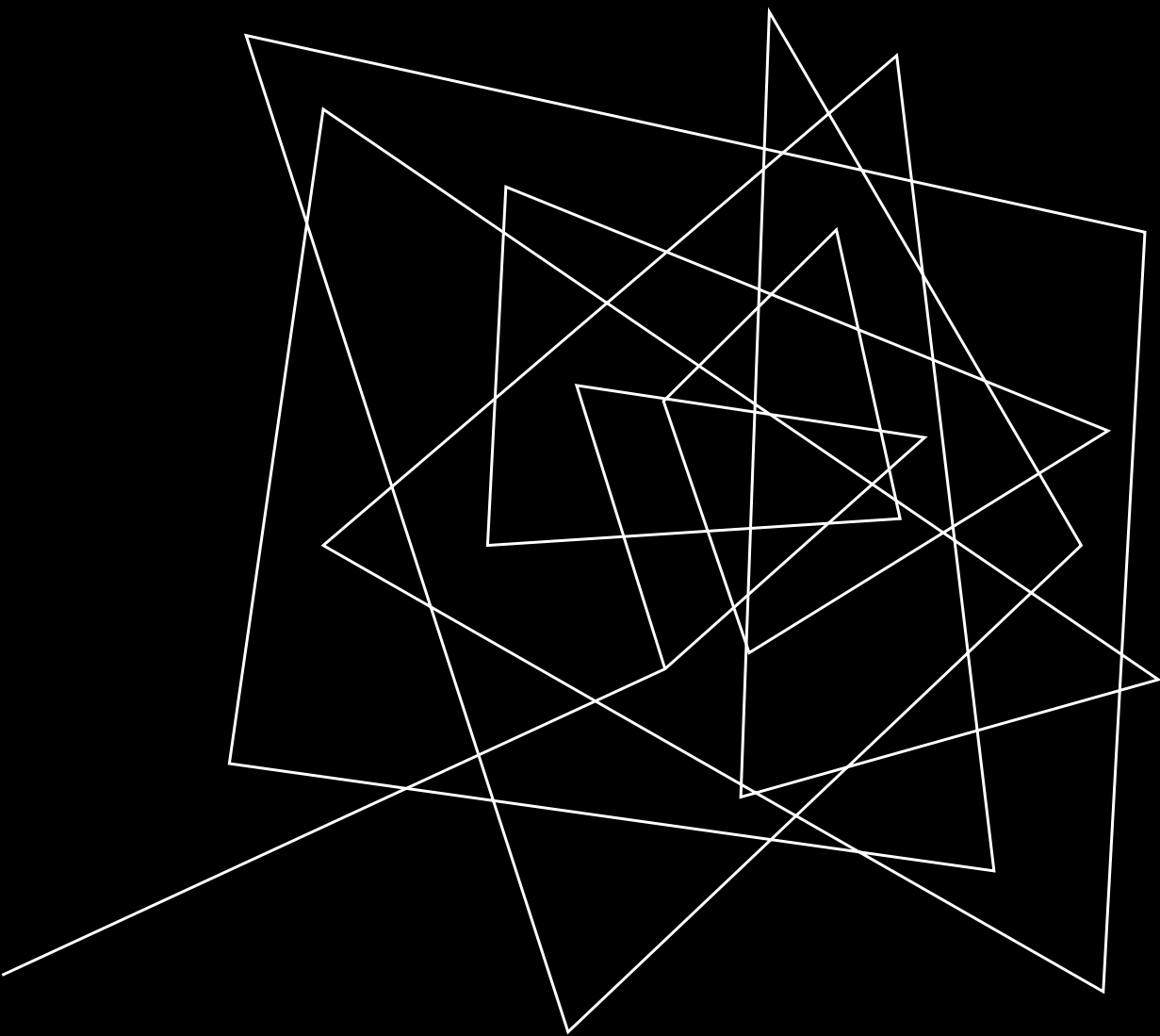
SoftMax Regression

Multi-Label Classification

Underfitting & Overfitting

Regularization

*Example Problems*



*CLASSIFICATION*

# SUPERVISED LEARNING SETUP

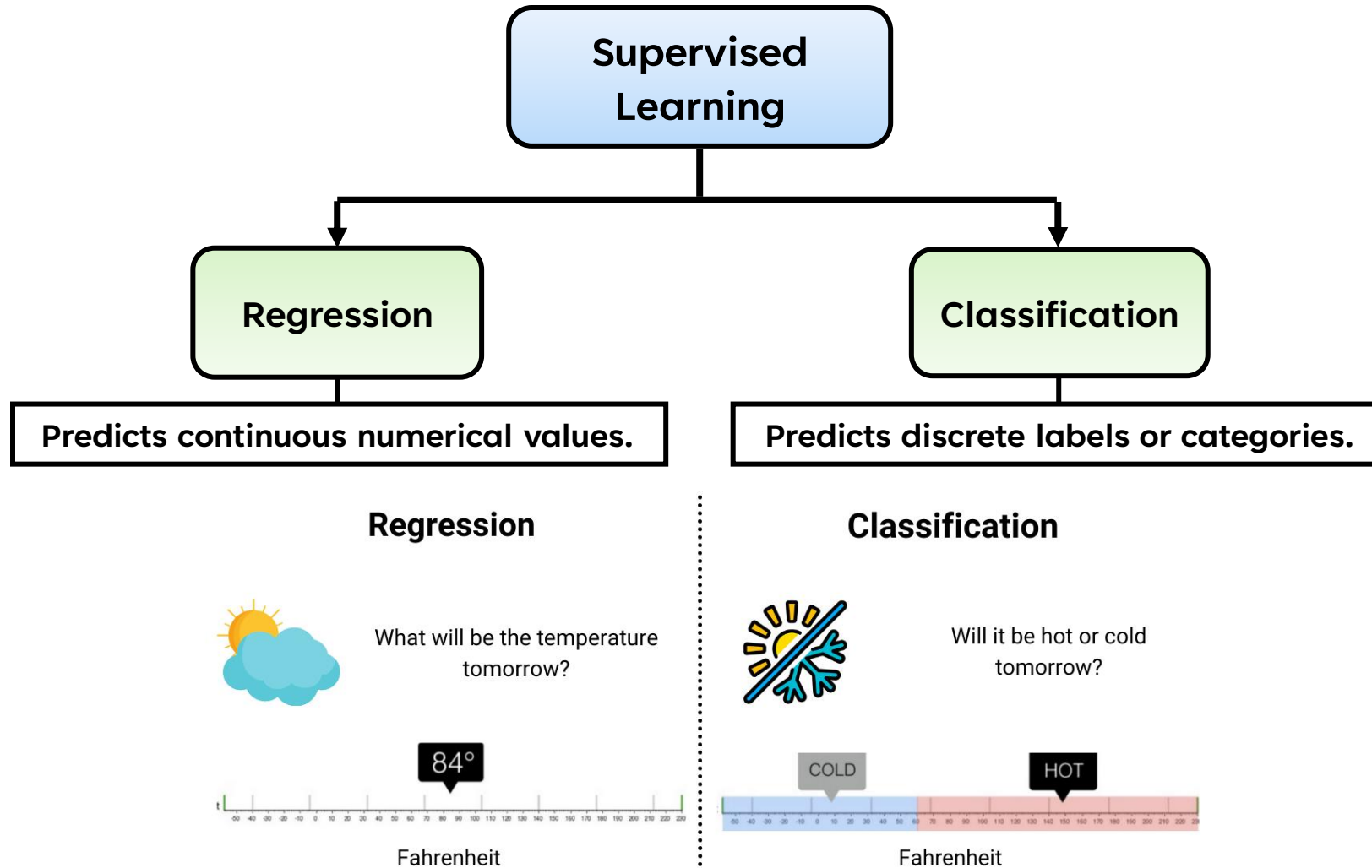
## In Supervised Learning:

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

## *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

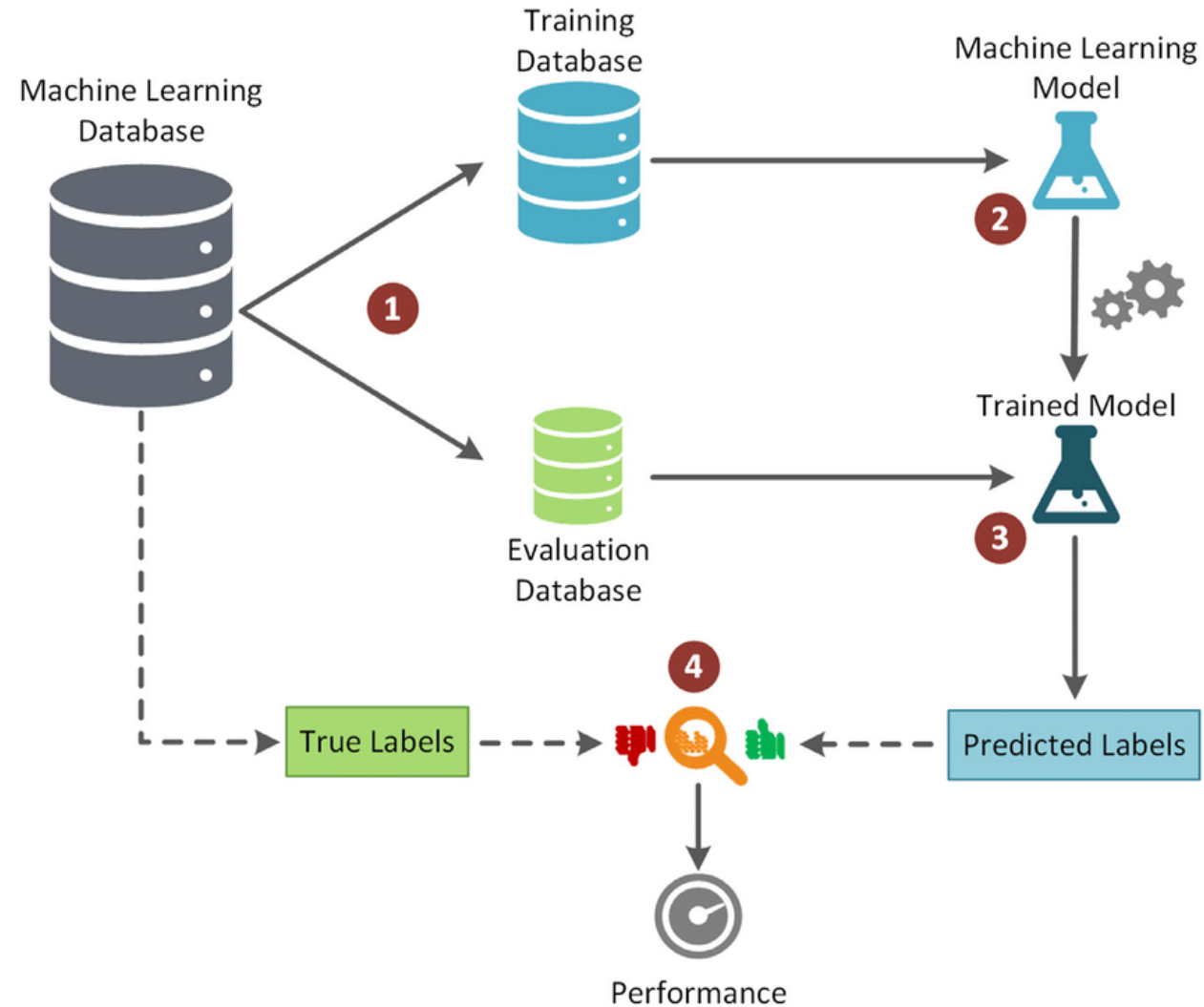


# CLASSIFICATION

**Classification** is a type of supervised machine learning where the goal is to predict the categorical label or class of a given input. In classification tasks, an algorithm learns from a labeled dataset (where each input is associated with a known class) and aims to assign the correct label to new, unseen data.

- **Use Cases:** Spam Detection (from emails), Object Recognition (from images), Disease Diagnosis (from Medical Data), etc.
- **Evaluation Metrics:** Common metrics include Accuracy, Precision, Recall, F1-Score, AUC-ROC, Log Loss and Confusion Matrix, etc.

# CLASSIFICATION WORKFLOW



Source: Tamascelli, Nicola & Paltrinieri, Nicola & Cozzani, Valerio. (2020). Predicting chattering alarms: A machine Learning approach. *Computers & Chemical Engineering*. 143. 107122. 10.1016/j.compchemeng.2020.107122.

# CLASSIFICATION (EXAMPLE)

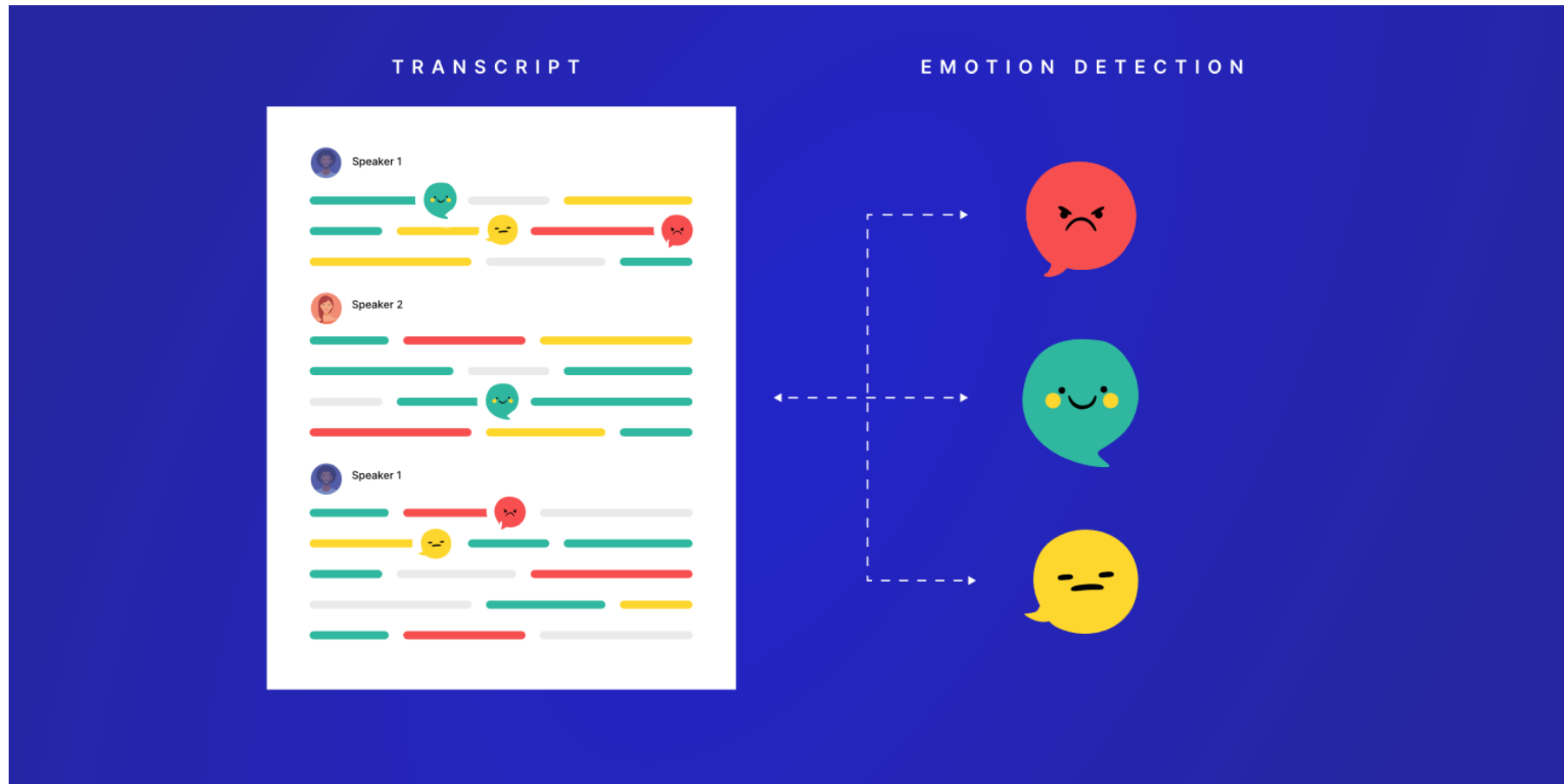


Source: <https://medium.com/@vlknerdem/full-stack-application-for-spam-detection-8ba86f10d328>

## Spam Detection & Filtering



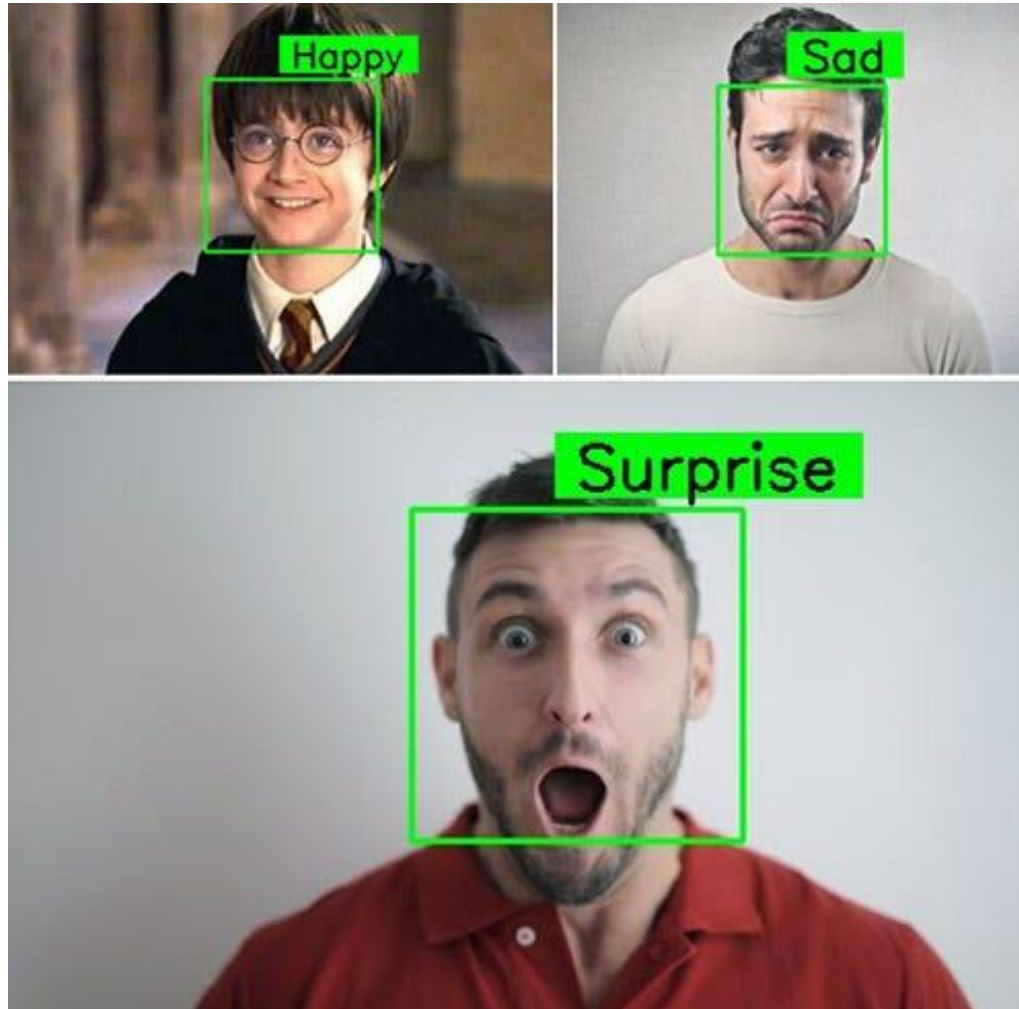
# CLASSIFICATION (EXAMPLE)



Source: <https://exemplary.ai/blog/sentiment-analysis>

**Sentiment Analysis**

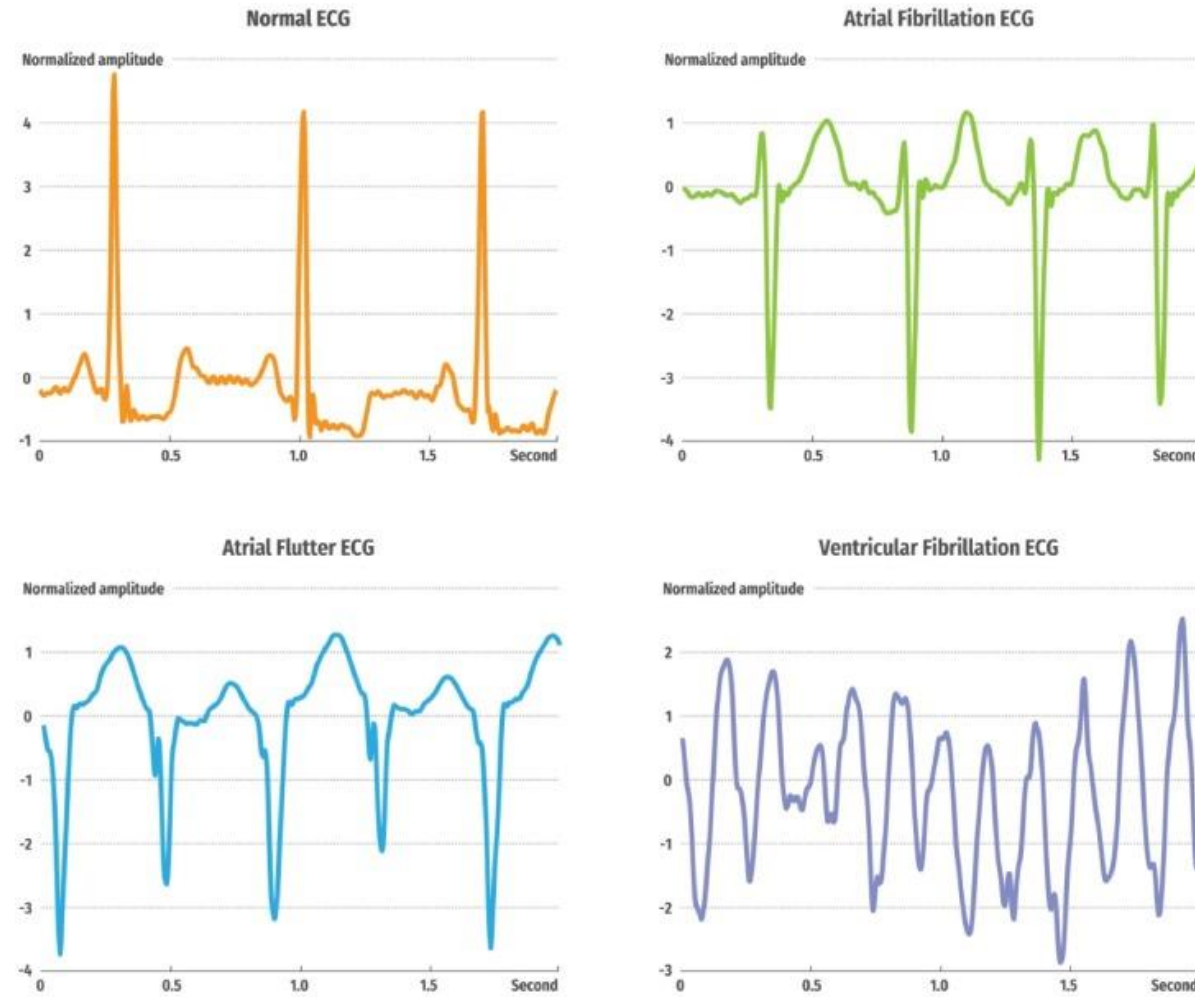
# CLASSIFICATION (EXAMPLE)



Source: <https://medium.com/@dana.fatadilla123/real-time-face-expression-recognition-b2214b6cdfb9>

## Facial Expression Recognition

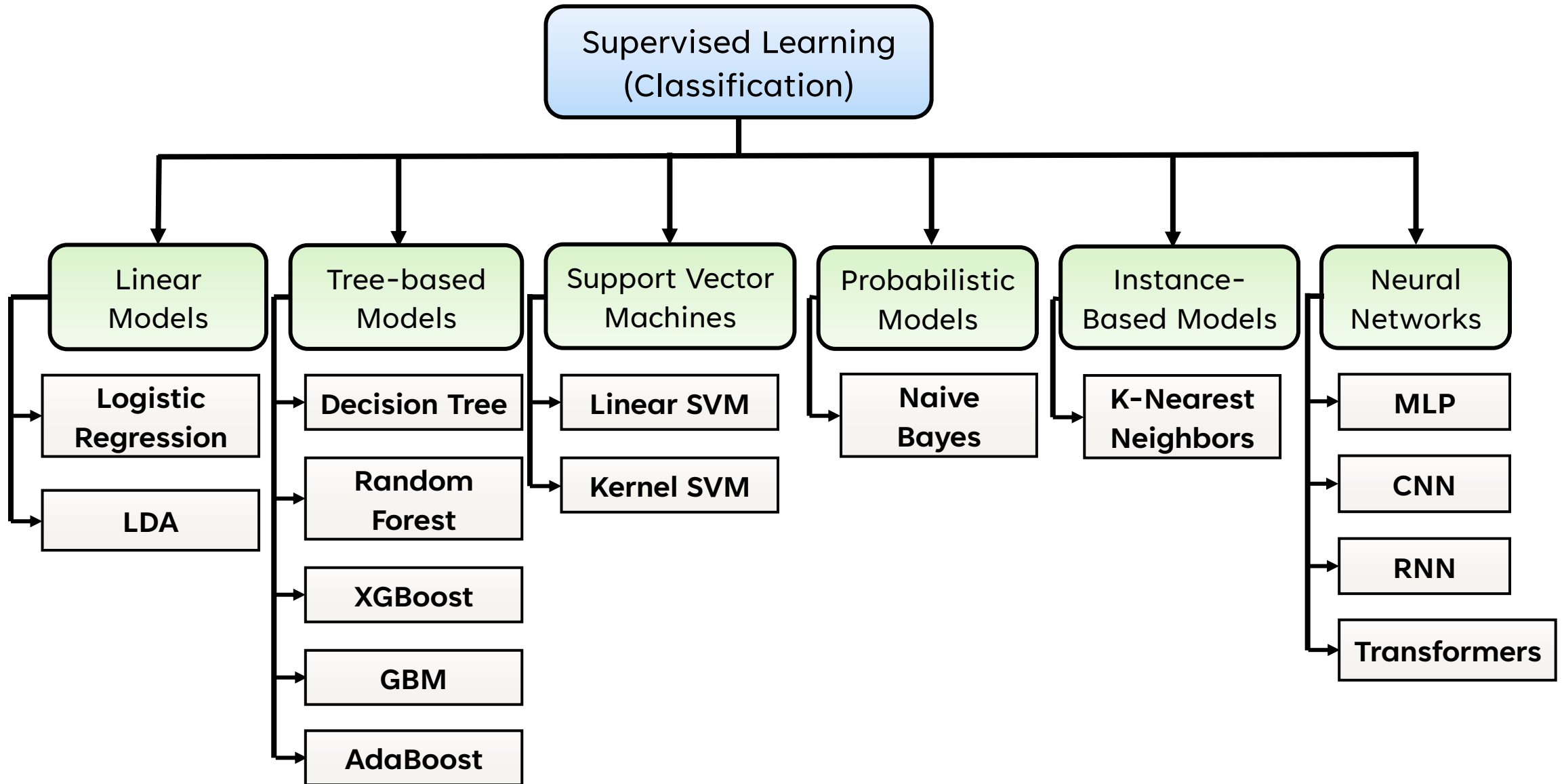
# CLASSIFICATION (EXAMPLE)

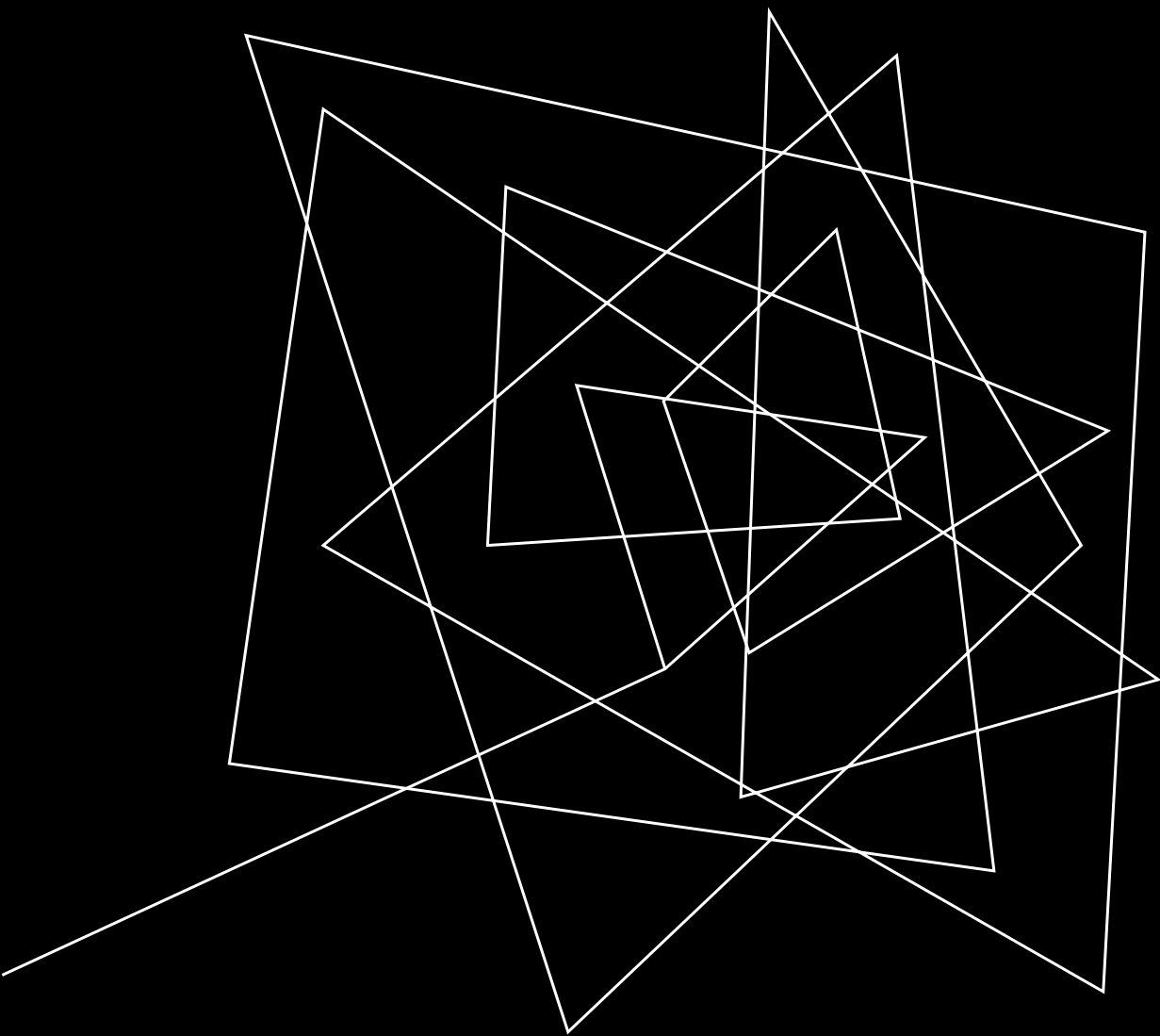


Source: <https://www.sciencedirect.com/science/article/pii/S0020025517306539>

## Arrhythmia Detection

# CLASSIFICATION ALGORITHMS





## BINARY CLASSIFICATION & PROBABILISTIC MODELS

# BINARY CLASSIFICATION

**Binary Classification** is a type of classification task where the goal is to categorize input data into one of two distinct classes. Binary classification algorithms are used to train a model that predicts one of two possible labels for a single class. Essentially, predicting **true** or **false**, **positive** or **negative**, etc. In most real scenarios, the data observations used to train & validate the model consist of multiple feature (**x**) values and a target (**t**) value that is either **1** or **0**.

## Decision Boundary

In Binary Classification, the output can be one of two categories (e.g., spam vs. not spam, disease vs. no disease). The model learns a boundary that separates the two classes based on the input features.

Most binary classification algorithms output a **probability score** for each class rather than directly assigning a label. For example, in logistic regression, the algorithm outputs a probability value **between 0 and 1**. *This value represents the likelihood of the input belonging to the positive class (e.g., class 1).* The final classification decision is made by setting a **threshold** (typically **0.5**). **If the predicted probability is above this threshold, the input is classified as class 1; otherwise, it's classified as class 0.** For example, If a model predicts a **0.7** probability that an email is spam, and the threshold is **0.5**, the email would be classified as "spam."

# UNDERSTANDING BINARY CLASSIFICATION

To understand how **Binary Classification** works, let's look at a simplified example that uses a single feature ( $x$ ) to predict whether the label  $y$  is 1 or 0. In this example, we'll use the blood glucose level of a patient to predict whether or not the patient has diabetes. Here's the data with which we'll train the model:

Blood Glucose	Diabetic? (Y)
112	1
70	0
103	1
83	0
64	0
58	0
87	0
89	1
100	1
84	1

# UNDERSTANDING BINARY CLASSIFICATION

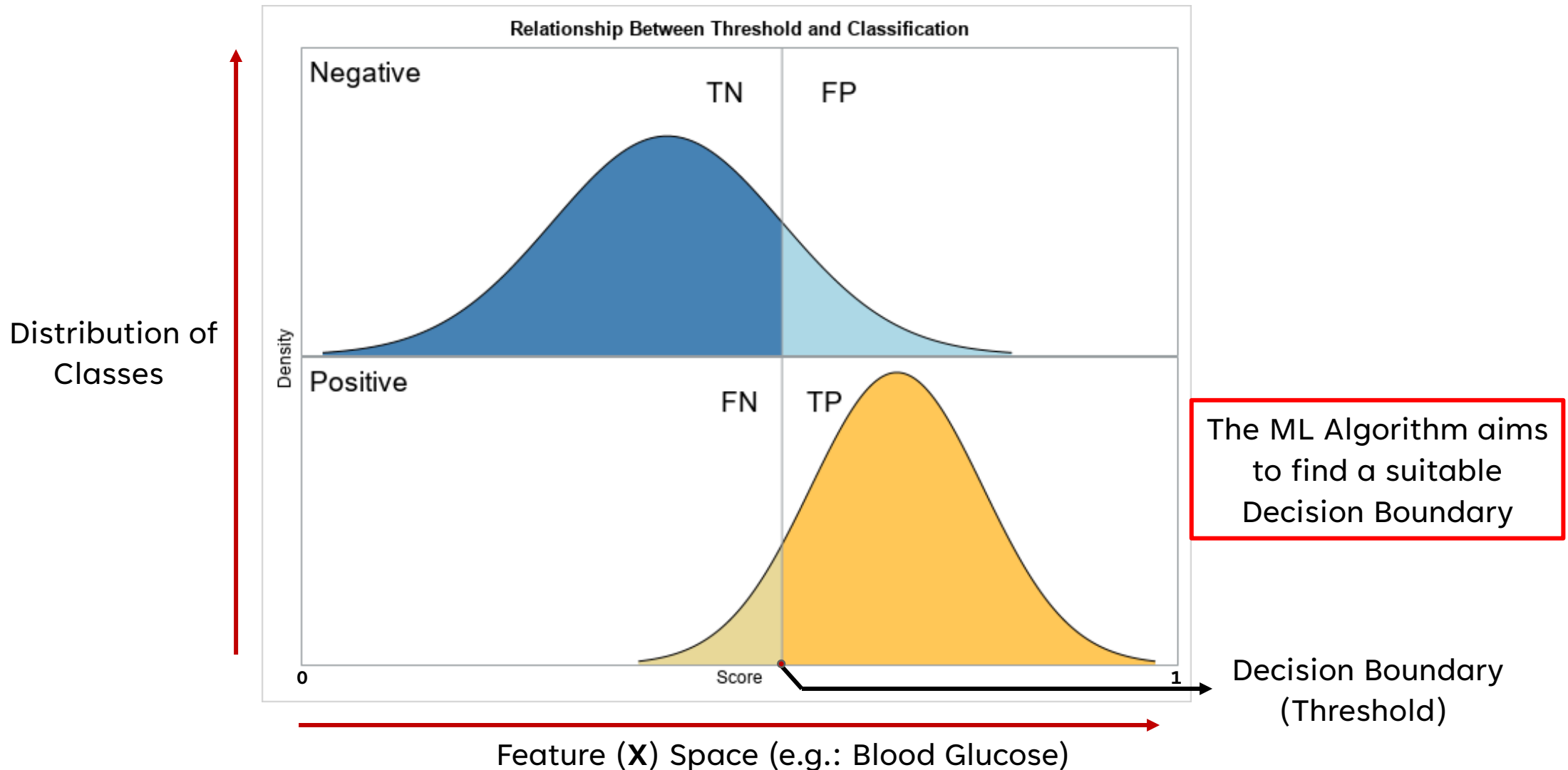
Let us sort the dataset according to Blood Glucose level to understand how the classification is done :

Blood Glucose	Diabetic? (Y)	
58	0	Almost surely Non-Diabetic. Expected <b>Output Probability ~ 0</b>
64	0	
70	0	
83	0	Unsure between Diabetic & Non-Diabetic. Expected <b>Output Probability</b> is somewhere in the middle.
84	1	
87	0	
89	1	
100	1	Almost surely Diabetic. Expected <b>Output Probability ~ 1</b>
103	1	
112	1	

Clearly, there is some **Threshold** at around **80-90** Blood Glucose level above which the patient is diagnosed to be **Diabetic**. The Binary Classification is supposed to find this threshold. This example can be extended to multiple features with a multi-dimensional **Decision Boundary**.



# UNDERSTANDING BINARY CLASSIFICATION



# BINARY LINEAR CLASSIFICATION

Let us attempt **Binary Classification** in a familiar approach, the **Linear Regression**. However, there are some issues with this algorithm when it comes to classification tasks.

## The Issue of Continuous & Unbounded Output

Linear regression predicts a continuous value, while classification problems require discrete classes (e.g., 0 or 1 in binary classification). In binary classification tasks, the labels are 0 and 1. We often need the output to be **bounded between 0 and 1** as probabilities. Linear regression doesn't provide this, as its output can be any real number from  $-\infty$  to  $+\infty$ .

However, some of these issues can be addressed by adding a thresholding step after the output of linear regression. So, the model may look like:

### Step 1: Linear Regression

$$z = \theta^T \mathbf{x} = \mathbf{w}^T \mathbf{x} + b$$

### Step 2: Thresholding

$$y = \begin{cases} 1 & \text{if } z \geq r \\ 0 & \text{if } z < r \end{cases}$$

Here, **r** is the  
threshold

# BINARY LINEAR CLASSIFICATION

Let's simplify the model a bit.

## Eliminating the Threshold

$$\mathbf{w}^T \mathbf{x} + b \geq r \implies \mathbf{w}^T \mathbf{x} + (b - r) \geq 0 \implies \mathbf{w}^T \mathbf{x} + w_0 \geq 0$$

## Eliminating the Bias

$$\mathbf{w}^T \mathbf{x} + w_0 \cdot 1 \geq 0 \implies \mathbf{w}^T \mathbf{x} \geq 0 \implies \boldsymbol{\theta}^T \mathbf{x} \geq 0$$

A dummy feature  $\mathbf{x}_0$  was added which always takes the value **1**. The weight  $w_0 = (b - r)$  is equivalent to a bias (same as linear regression).

## Simplified Model

For received input,  $\mathbf{x} \in \mathbb{R}^{D+1}$  with  $\mathbf{x}_0 = 1$ :

$$\mathbf{z} = \boldsymbol{\theta}^T \mathbf{x}$$
$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

# HOW TO CALCULATE THE COST?

The Model may look nice. But now comes the next step. How to calculate the loss for each prediction? And how to optimize the model? Even thresholding doesn't solve all the problems...

- ❑ Linear regression minimizes the **MSE** or **MAE**, which are suitable for regression tasks but not ideal for classification. MSE and MAE are regression-based loss functions that aim to minimize **the difference between the predicted value and the true label**. In classification, the true labels are typically binary (0 or 1). However, **linear regression can produce values outside the range [0, 1]**, such as negative values or values greater than 1, which are meaningless for classification, where we expect probabilities.
- ❑ MSE and MAE loss functions assume that the relationship between input features and output is continuous and linear, which is rarely the case in classification. When applied to classification, minimizing MSE or MAE might drive predictions closer to 0 or 1, but it can lead to inefficient or misleading gradient updates, particularly for misclassified data points.
- ❑ This approach leads to the optimization being influenced a great deal by points that are already classified correctly and don't really need to influence the adjustment of the linear function. They suppress the importance of the points that are wrongly classified and that should impact the optimization at large.

# COST OF BEING TOO RIGHT?

An example where the loss becomes too large due to being "too right" when using Mean Squared Error (MSE) in linear regression for classification is **when the model predicts a value that is much larger than 1 or much smaller than 0**. Even though the prediction will be accurate, the loss may become very large and affect the optimization process negatively.

**Example:** Suppose we have a binary classification task where the true label is **1** (positive class), but the linear regression model predicts **2** and **0.7** for two separate inputs. Clearly, if the threshold is **0.5** (or anything between 0 and 1), the first input is further from the threshold and much more likely/probable to be positive (**1**). But, according to **MSE**,

$$\text{Error}_2 = (1-2)^2 = 1$$

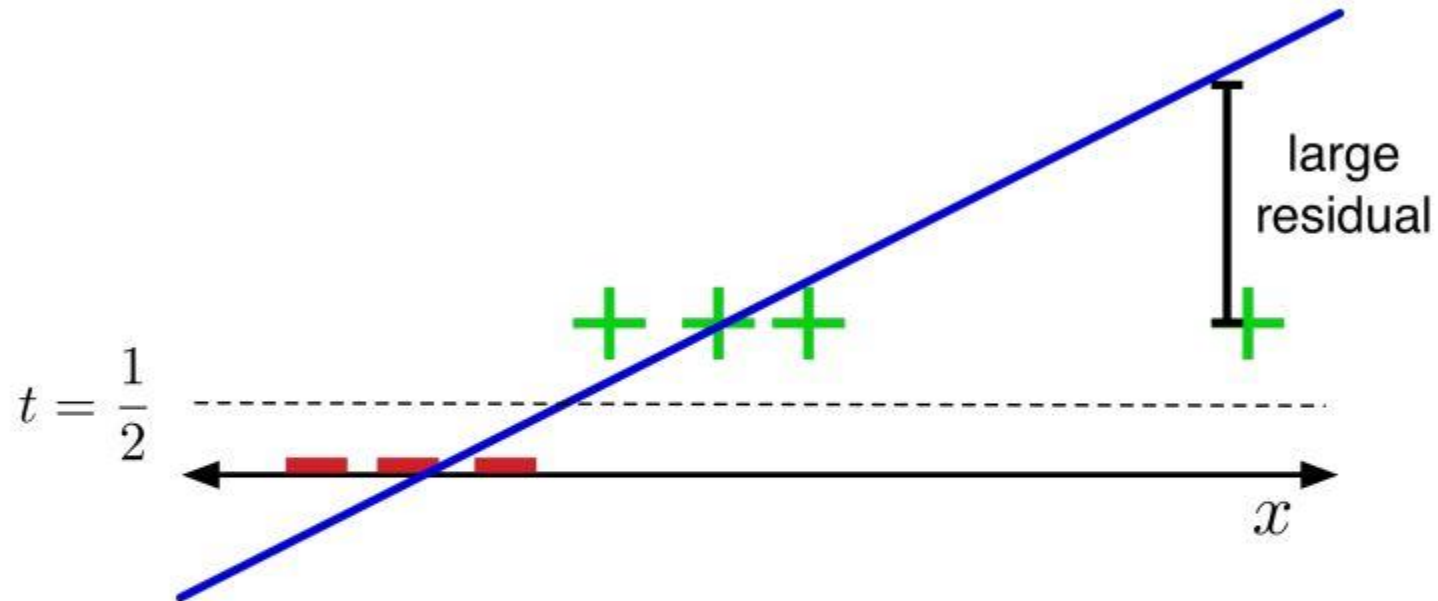
$$\text{Error}_{0.7} = (1-0.7)^2 = 0.09$$

Apparently, the corresponding errors show a different picture, penalizing the first input for being too right/confident. This exaggerated penalty results in inefficiency and severe degradation in optimization performance. Clearly, We need an alternative method. We could use the **0-1 Loss** function **based on predictions**, but it makes it **hard to Optimize (non-convex, not continuous)**.

*What about calculating the errors in Output Probabilities?*

# COST OF BEING TOO RIGHT?

The problem:



- The loss function hates when you make correct predictions with high confidence!
- If  $t = 1$ , it's more unhappy about  $z = 10$  than  $z = 0$ .

# PROBABILITY IN CLASSIFICATION

**Probability** is a measure of the likelihood that an event will occur. It ranges **from 0** (the event will not occur) **to 1** (the event will definitely occur). In machine learning, probability helps quantify the confidence of predictions made by models.

As we saw, thresholding directly predicted labels is problematic due to its unbounded nature. One intuitive solution to this is using output probabilities to measure the loss. **As probabilities are naturally bounded between 0 to 1, they present a tailor-made solution to Binary Classification problems, where also the labels happen to be 0 and 1.**

For this reason, In classification tasks, probabilities are crucial for designing and optimizing cost functions, which guide the model to make accurate predictions by minimizing errors. The most commonly used cost functions for classification are based on logarithmic transformations of probabilities to penalize incorrect or overconfident predictions.

The **Summary** is, that instead of thresholding directly predicted labels, **we can map the predicted labels to probabilities of belonging to corresponding classes**, and then compare/threshold the probabilities.

# PROBABILISTIC MODELS

The supervised models we have seen so far parameterize a function  $h_{\theta} : \mathcal{X} \longrightarrow \mathcal{Y}$ , that maps inputs  $\mathbf{x} \in \mathcal{X}$  to targets  $\mathbf{y} \in \mathcal{Y}$ . Models have parameters  $\theta \in \Theta$  living in a set  $\Theta$ .

A supervised model could also parameterize a probability distribution  $P_{\theta}(\mathbf{y} \mid \mathbf{x})$  that maps inputs  $\mathbf{x}$  to a distribution over targets  $\mathbf{y}$ . Concretely, this means *defining a formula with parameters  $\theta$  that given an  $\mathbf{x}$ ,  $\mathbf{y}$  yields a probability in  $[0, 1]$* . These kinds of models are **Probabilistic Models**.

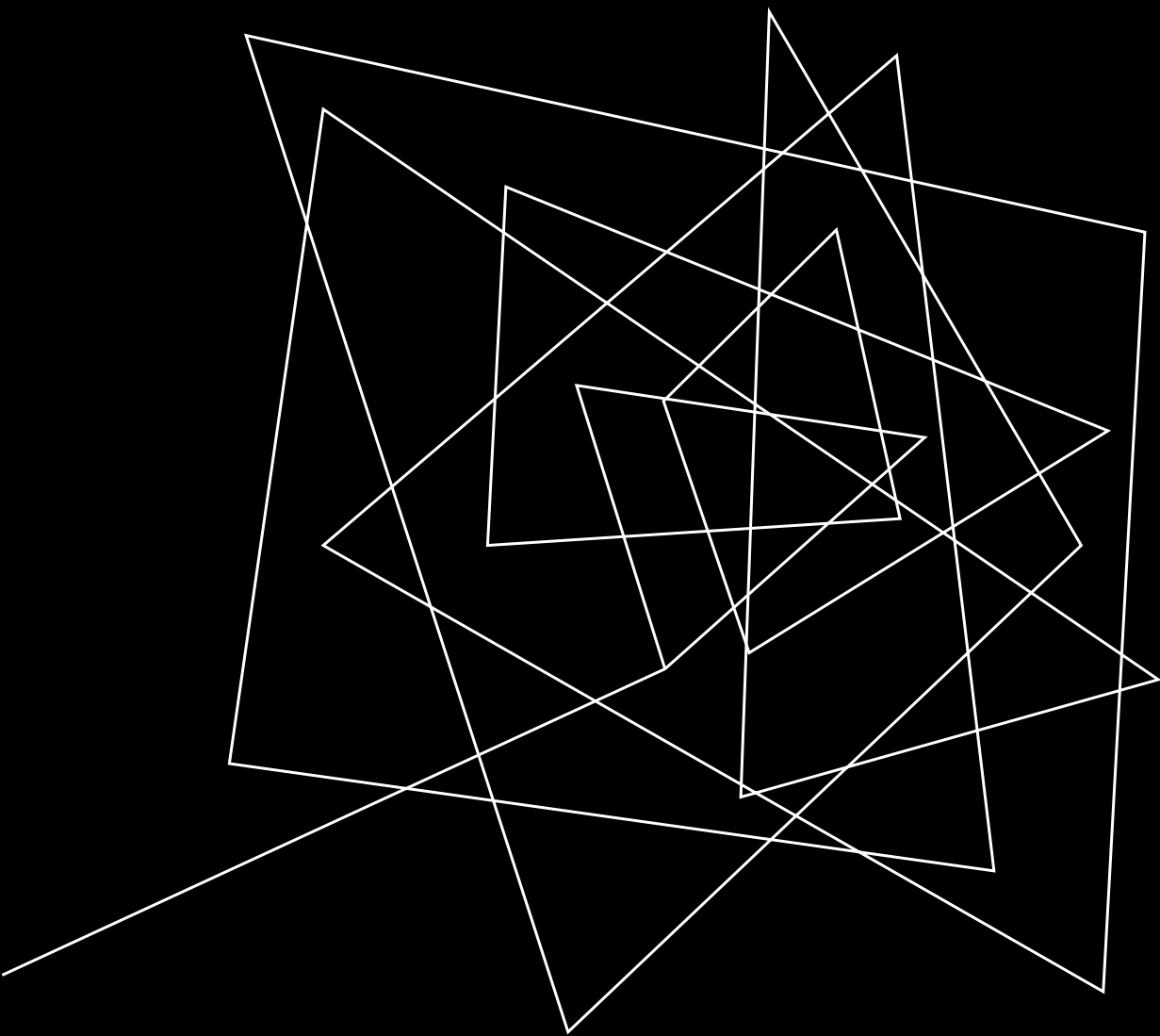
Many supervised learning models can be interpreted as defining probabilities. The logistic regression model will be our first example of a parameterization of  $P_{\theta}(\mathbf{y} \mid \mathbf{x})$ . Later, we will also see models that parameterize a joint probability  $P_{\theta}(\mathbf{x}, \mathbf{y})$  of an input-target pair  $\mathbf{x}, \mathbf{y}$ .

For example, our logistic model will define (“parameterizes”) a probability distribution as follows:

$$\begin{aligned} P_{\theta}(\mathbf{y}=1 \mid \mathbf{x}) &= \sigma(\Theta^T \mathbf{x}) \\ P_{\theta}(\mathbf{y}=0 \mid \mathbf{x}) &= 1 - \sigma(\Theta^T \mathbf{x}) \end{aligned}$$

Here, the function  $\sigma(\mathbf{x})$  is called **sigmoid function** which can map unbounded values to the range of  $[0, 1]$ , essentially representing probabilities.





# THE SIGMOID FUNCTION

# THE SIGMOID FUNCTION

As we saw, in Classification tasks, we need some mechanism to convert the linear output of a model into a probability between 0 and 1. The **Sigmoid Function** is a mathematical function that **maps any real-valued number into a value between 0 and 1**. It is commonly used in binary classification problems, particularly in logistic regression and neural networks, to map the output of a model to a probability.

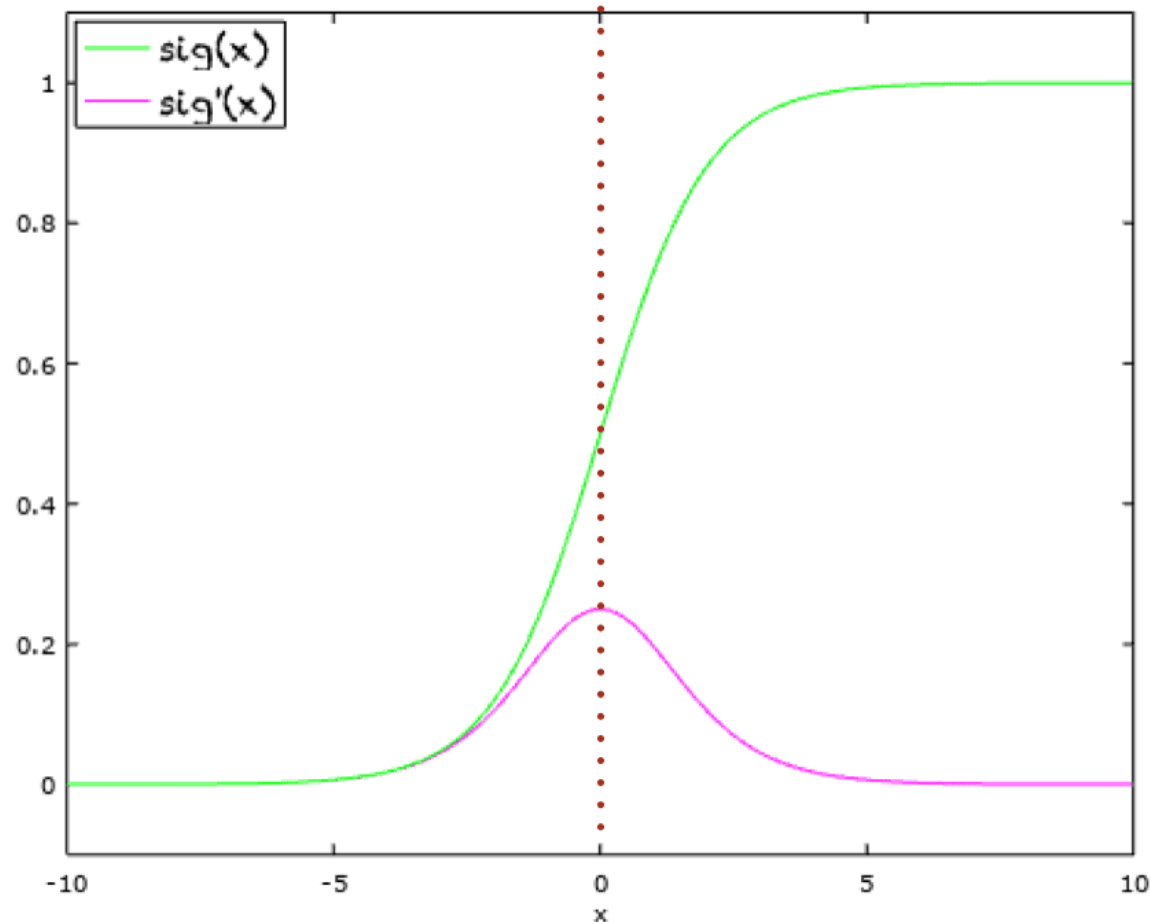
## Mathematical Representation

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{e^z}{1 + e^z}$$

- The **sigmoid function** maps any input **z** to a value between **0** and **1**. This is important because probabilities must lie within this range.
- For very large negative values of **z**, the sigmoid output approaches **0**.
- For very large positive values of **z**, the sigmoid output approaches **1**.
- When **z = 0**, the sigmoid function outputs exactly **0.5**, representing maximum uncertainty.
- In logistic regression, for instance, the model generates a linear output, **z =  $\theta^T \mathbf{x}$** . The **sigmoid function is then applied to this linear output to convert it into a probability**.

$$P(y=1|\mathbf{x}) = \sigma(\theta^T \mathbf{x}) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}$$

# THE SIGMOID FUNCTION



Plot of  $\sigma(x)$  and its derivate  $\sigma'(x)$

Source: <https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/>

Domain:  $(-\infty, +\infty)$

Range:  $(0, +1)$

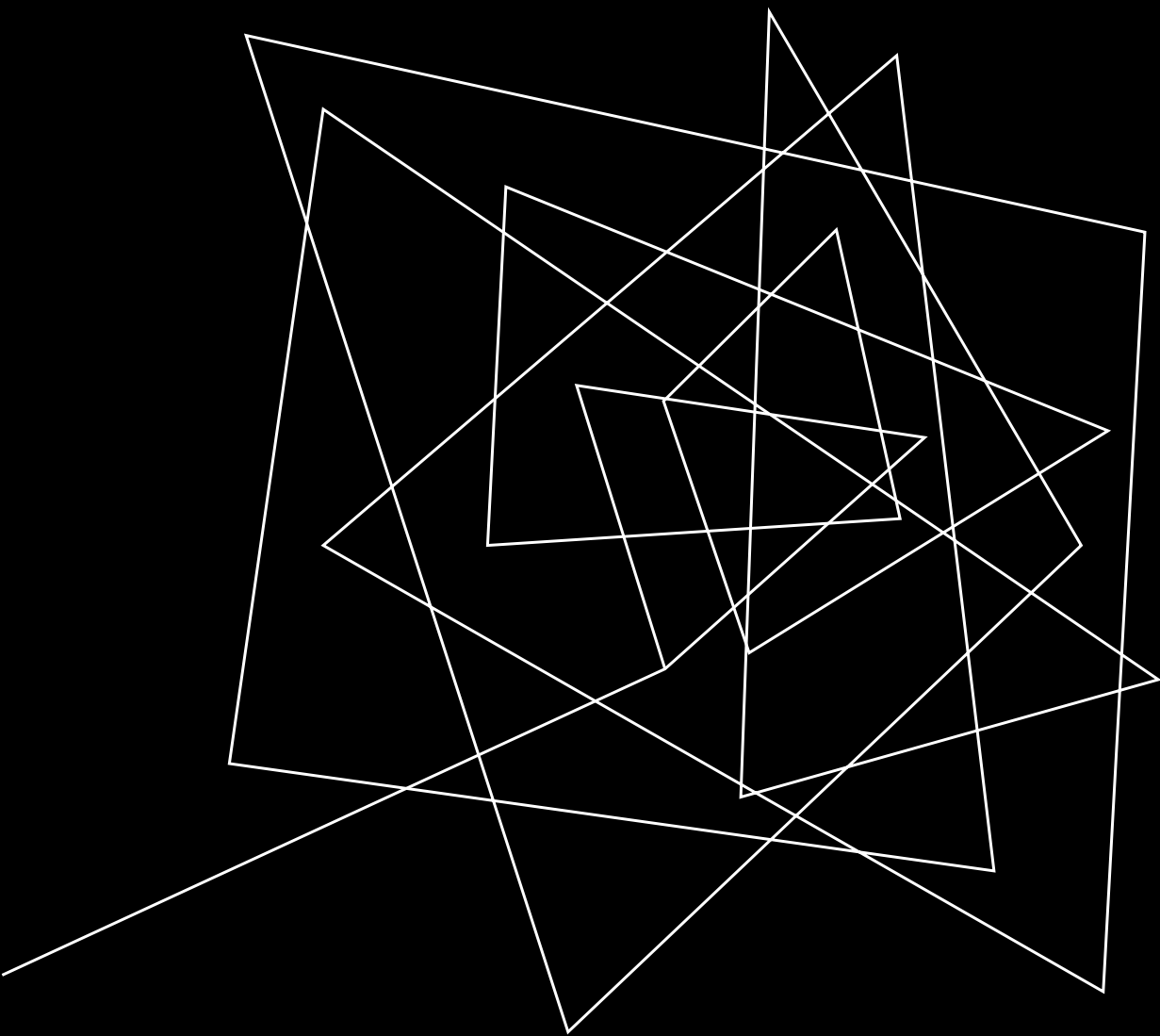
$$\sigma(0) = 0.5$$

Other properties

$$\sigma(x) = 1 - \sigma(-x)$$

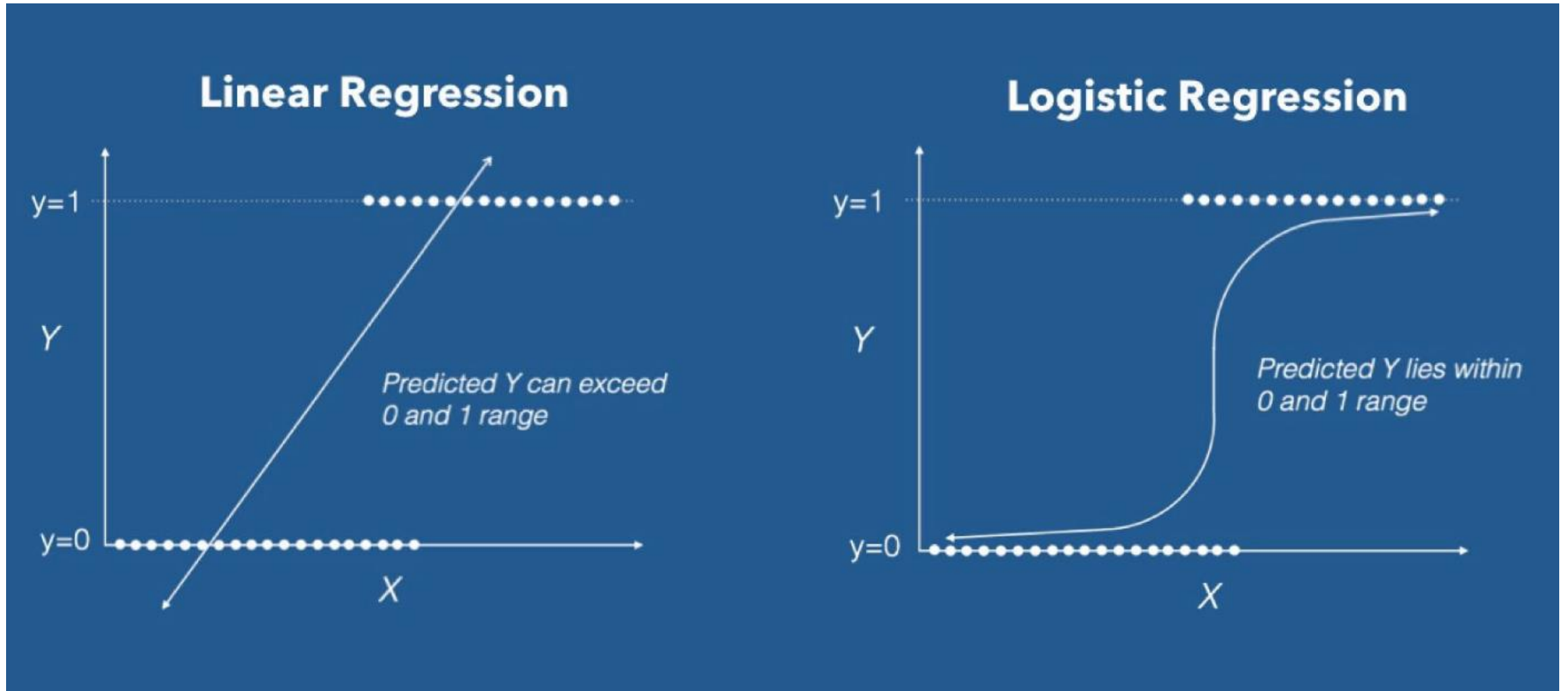
$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$



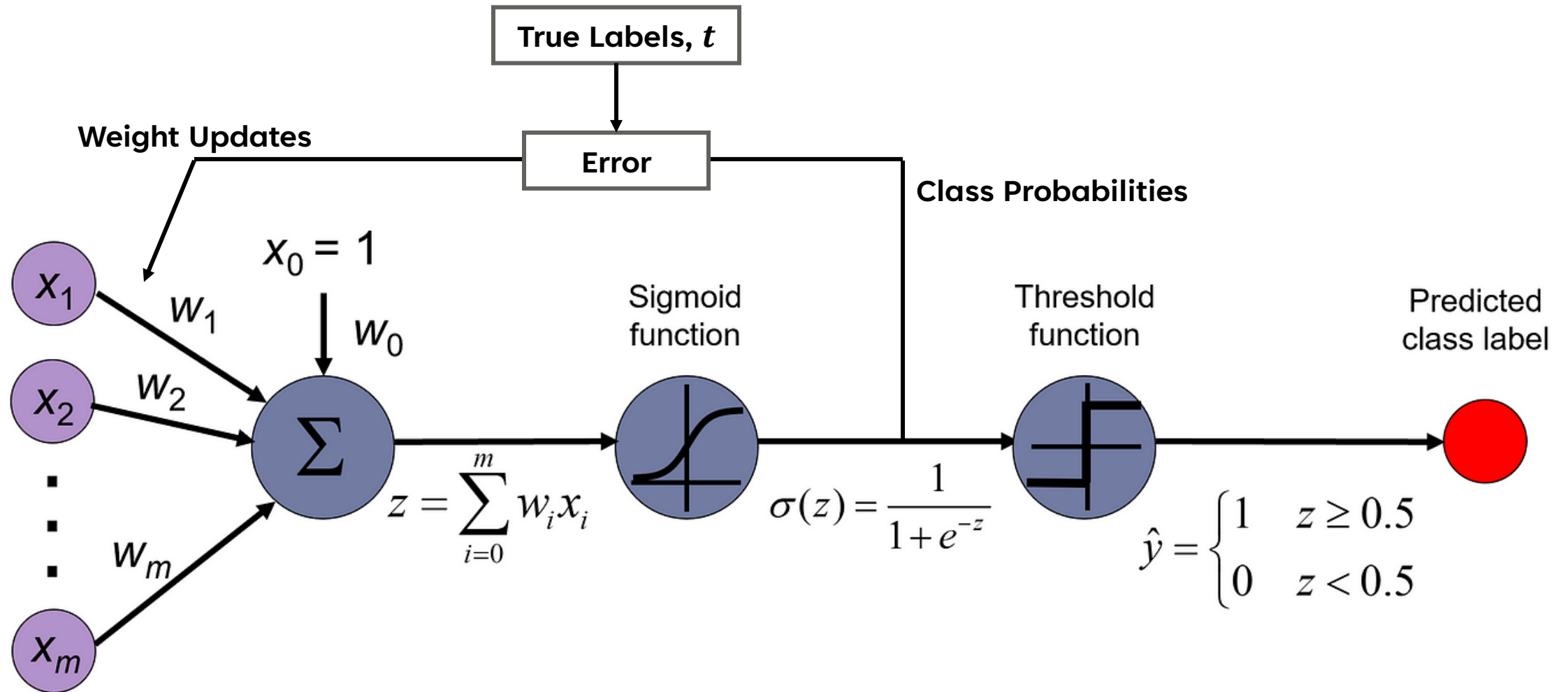
LOGISTIC REGRESSION

# LINEAR vs LOGISTIC REGRESSION



Source: <https://medium.com/@maithilijoshi6/a-comparison-between-linear-and-logistic-regression-8aea40867e2d>

# LOGISTIC REGRESSION APPROACH



Source: <https://towardsdatascience.com/mastering-logistic-regression-3e502686f0ae>

# LOGISTIC REGRESSION: MODEL

**Model:** In Binary Classification using Logistic Regression, the idea is to use a linear function of the features  $\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_D] \in \mathbb{R}^D$  to make predictions  $\mathbf{y}$  of the target class  $\mathbf{t} \in \{0, 1\}$ :

## Step 1: Linear Function

$$z = \sum_{j=1}^n w_j x_j + w_0 = \mathbf{w}^T \mathbf{x} + w_0 = \boldsymbol{\theta}^T \mathbf{x}$$

## Step 2: Sigmoid Activation

$$\text{Output Probability, } P(\mathbf{y}=1|\mathbf{x}) = \boldsymbol{\sigma}(z) = \frac{1}{1 + e^{-z}}$$

## Step 3: Thresholding & Output (*The Threshold is typically 0.5*)

$$\text{Predicted Output, } \mathbf{y} = \begin{cases} 1 & \text{if } P(\mathbf{y}=1|\mathbf{x}) \geq \textit{Threshold} \\ 0 & \text{if } P(\mathbf{y}=1|\mathbf{x}) < \textit{Threshold} \end{cases}$$

- **Objective:** To get the **Output Probability** close to the original target:  $\mathbf{P} \approx \mathbf{t}$ .

# LOGISTIC REGRESSION: MODEL (EXAMPLE)

Dataset

$x_0$	Living Area (feet <sup>2</sup> )	#Bedrooms	Price
1	2104	3	High (1)
1	1600	3	Low (0)
1	2400	3	High (1)
1	1416	2	Low (0)
1	3000	4	High (1)
1	.	.	.
1	.	.	.

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

Linear Function:

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

As we know,

$w$  and  $b$  together are the parameters,  $\theta$ , where  $b = \theta_0$ . So,  $x_0^{(i)} = 1$ , for all  $i$ . Here, Number of Features,  $n = 2$ . Hence,

$$\begin{aligned} z = f(x) &= \sum_{j=0}^n \theta_j x_j \\ &= \theta_1 x_1 + \theta_2 x_2 + \theta_0 \end{aligned}$$

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

Sigmoid Activation:

$$P(y=1|x) = \sigma(\theta_1 x_1 + \theta_2 x_2 + \theta_0) = \frac{1}{1 + e^{-z}}$$

The **Objective** is to find suitable  $\theta = [\theta_0, \theta_1, \theta_2]$  so that  $P(y=1|x) \approx t$ .



# LOGISTIC REGRESSION: COST FUNCTION

In logistic regression, The most common cost function used is the ***Cross-Entropy Loss***, also known as ***Log Loss***.

$$\text{Log Loss, } \mathcal{L}_{CE}(\theta) = - [ t \log(P) + (1 - t) \log(1 - P) ]$$

- $\mathcal{L}_{CE}(\theta)$  is the Log Loss for a single prediction.
- $P$  is the predicted probability for the positive class.
- $t$  is the actual class label.

**Why Log Loss?:** This function measures how well the model's predictions match the true class labels based on output probabilities and is minimized during training. This cost function ensures that the model penalizes wrong predictions more heavily (*multiplication with the corresponding probability and then multiplying with -1*) and encourages accurate predictions. This makes it easier for optimization algorithms, like **Gradient Descent**, to find the minimum of the cost function.

# LOGISTIC REGRESSION: COST FUNCTION

**Log Loss** can sometimes be reduced to calculate  $\log(0)$ , for very low output probabilities, **which causes numerical instabilities**. For this reason, the Loss function may be modified this way:

$$\text{Log Loss, } \mathcal{L}_{LCE}(\theta) = t \log(1 + e^{-z}) + (1 - t) \log(1 + e^z)$$

- $\mathcal{L}_{LCE}(\theta)$  is the Log Loss for a single prediction.
- $z$  is the output of the linear regression function.
- $t$  is the actual class label.

Basically, this loss function combines the activation function and the loss into a single **Logistic-Cross-Entropy** ( $\mathcal{L}_{LCE}$ ) function.

The **Cost Function** is determined by aggregating and averaging the loss for all  $n$  samples:

$$\text{Cost Function, } J(\theta) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}^i(\theta)$$

# LOGISTIC REGRESSION: OPTIMIZATION

The **Objective** of logistic regression is to find the parameters  $\theta$  that minimize the cost function  $J(\theta)$ . Using an *Iterative Algorithm*(e.g. Gradient Descent) is the way to do so. It should be noted that the derivative of the **Log Loss** function has no direct solutions. So, the go-to method of optimization is iterative algorithms like **Gradient Descent**.

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

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j} = \theta_j - \frac{\alpha}{m} \sum_{i=1}^m (\sigma(\theta^T \mathbf{x}^{(i)}) - \mathbf{t}^{(i)}) \cdot \mathbf{x}_j^{(i)}$$

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

**Convergence:** The optimization algorithm converges when further updates to the parameters  $\theta$  result in little to no change (at global minima) in the cost function  $J(\theta)$ .

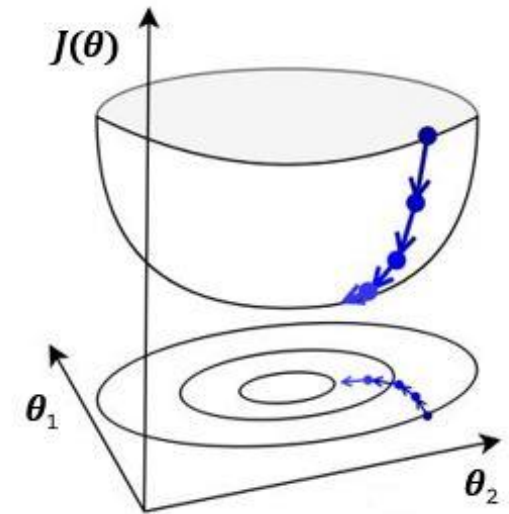
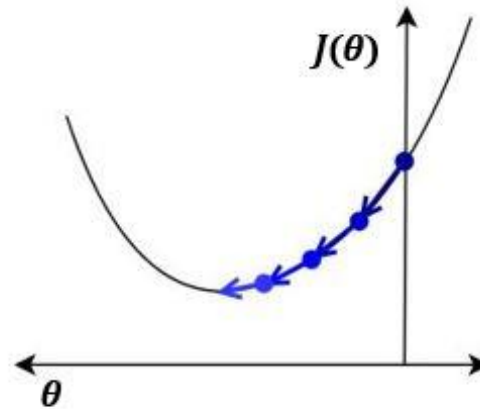
# GRADIENT DESCENT (REVIEW)

**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/](https://www.cs.toronto.edu/~rgrosse/courses/csc311_f20/)

# GRADIENT DESCENT (REVIEW)

## Observe:

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

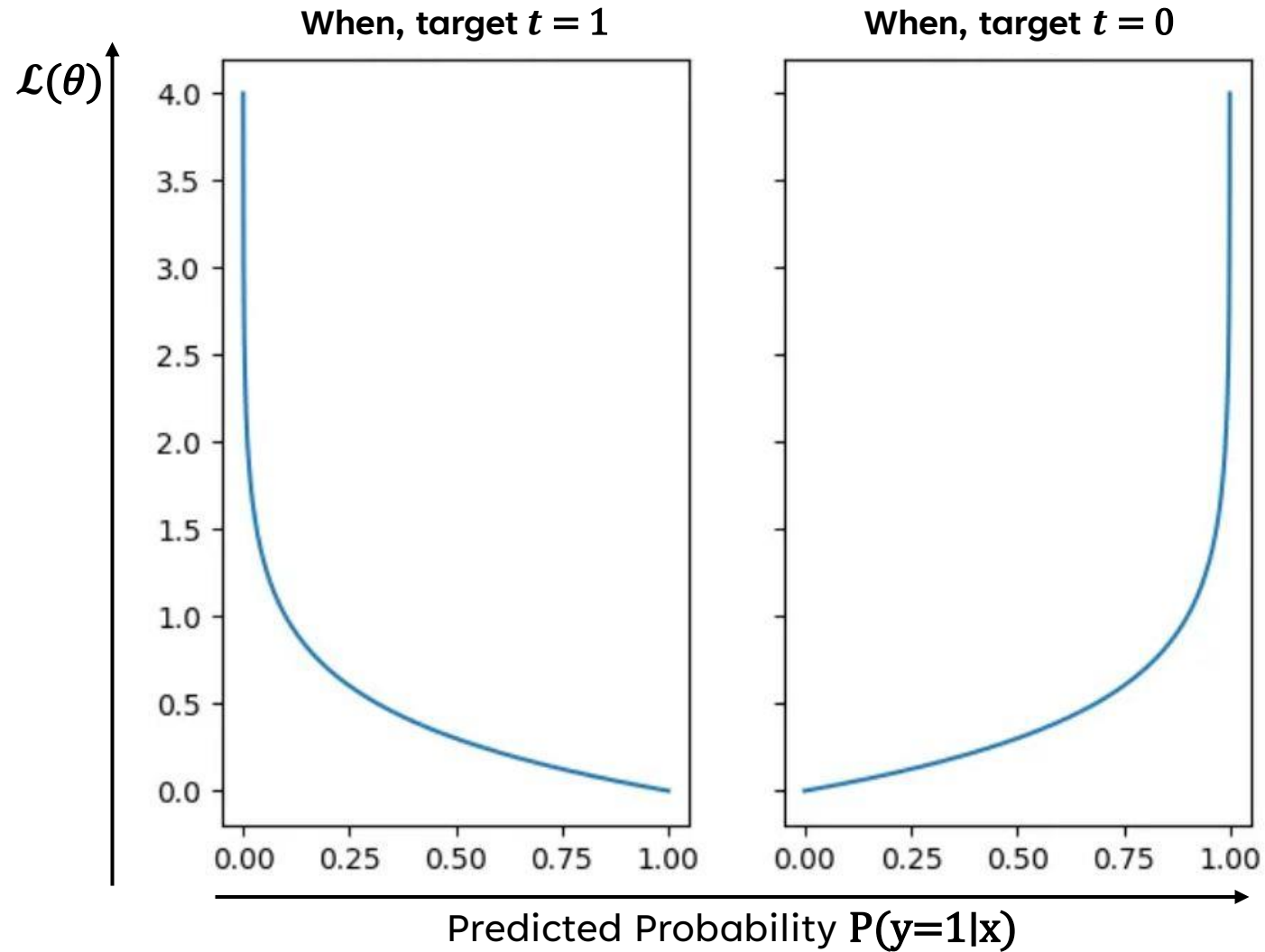
□ The following update *always decreases the cost function* for small enough  $\alpha$  (unless  $\frac{\partial J(\theta)}{\partial \theta_j} = 0$ , where it remains unchanged at the minima/maxima):

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

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

# LOG LOSS OPTIMIZATION



As we can see, **Log Loss is a convex function** in all cases. Hence, it ensures converging to the **global minima** through **Gradient Descent**.

# DECISION BOUNDARY

A **Decision Boundary** is a line (surface in higher dimensions) that **separates different classes** in a classification problem. It defines the regions in the feature space where the classifier assigns different class labels. In other words, it's the boundary that distinguishes where the decision of the classifier changes from one class to another.

Note that logistic regression finds a **linear** decision boundary. In this case, the model predicts **1** if  $P(y=1|x) \geq \textit{Threshold}$  & the model predicts **0** if  $P(y=1|x) < \textit{Threshold}$ . So, the decision boundary (separating line between the classes) here, is,

$$P(y=1|x) = \textit{Threshold}$$

For a threshold of 0.5, this relationship yields,

$$P(y=1|x) = 0.5 = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\text{So, } z = \theta^T x = 0$$

The set of  $x$  for which  $\theta^T x = 0$  represents a linear surface. So Logistic regression represents a linear classification model. The observation holds true for thresholds other than 0.5 too.

# DECISION BOUNDARY

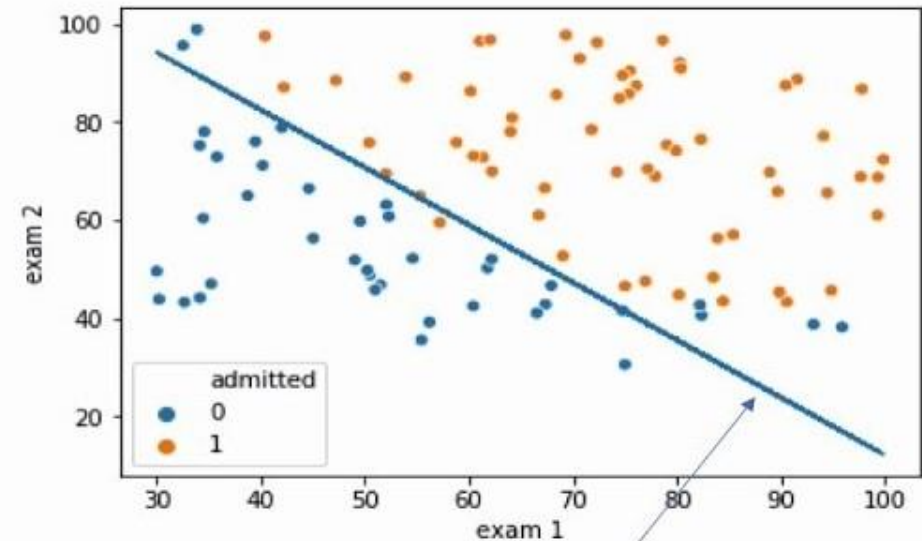
We can see the determination of the decision boundary line for a **2-dimensional feature space** in the picture below:

$$\text{act}(X_{\text{hidden}}) = \begin{cases} < 0.5 & \text{if } X_{\text{hidden}} \text{ is negative} \\ > 0.5 & \text{if } X_{\text{hidden}} \text{ is positive} \end{cases}$$

$$X_{\text{hidden}} = w_0 + w_1 X_1 + w_2 X_2$$

$$0 = w_0 + w_1 X_1 + w_2 X_2$$

$$X_2 = \frac{-(w_0 + w_1 X_1)}{w_2}$$



$$X_2 = \frac{-(w_0 + w_1 X_1)}{w_2}$$

Source: [https://youtu.be/3mLbNkt7j9g?si=Z97bD\\_IBeJohrtU7](https://youtu.be/3mLbNkt7j9g?si=Z97bD_IBeJohrtU7)



# MATHEMATICAL DERIVATIONS

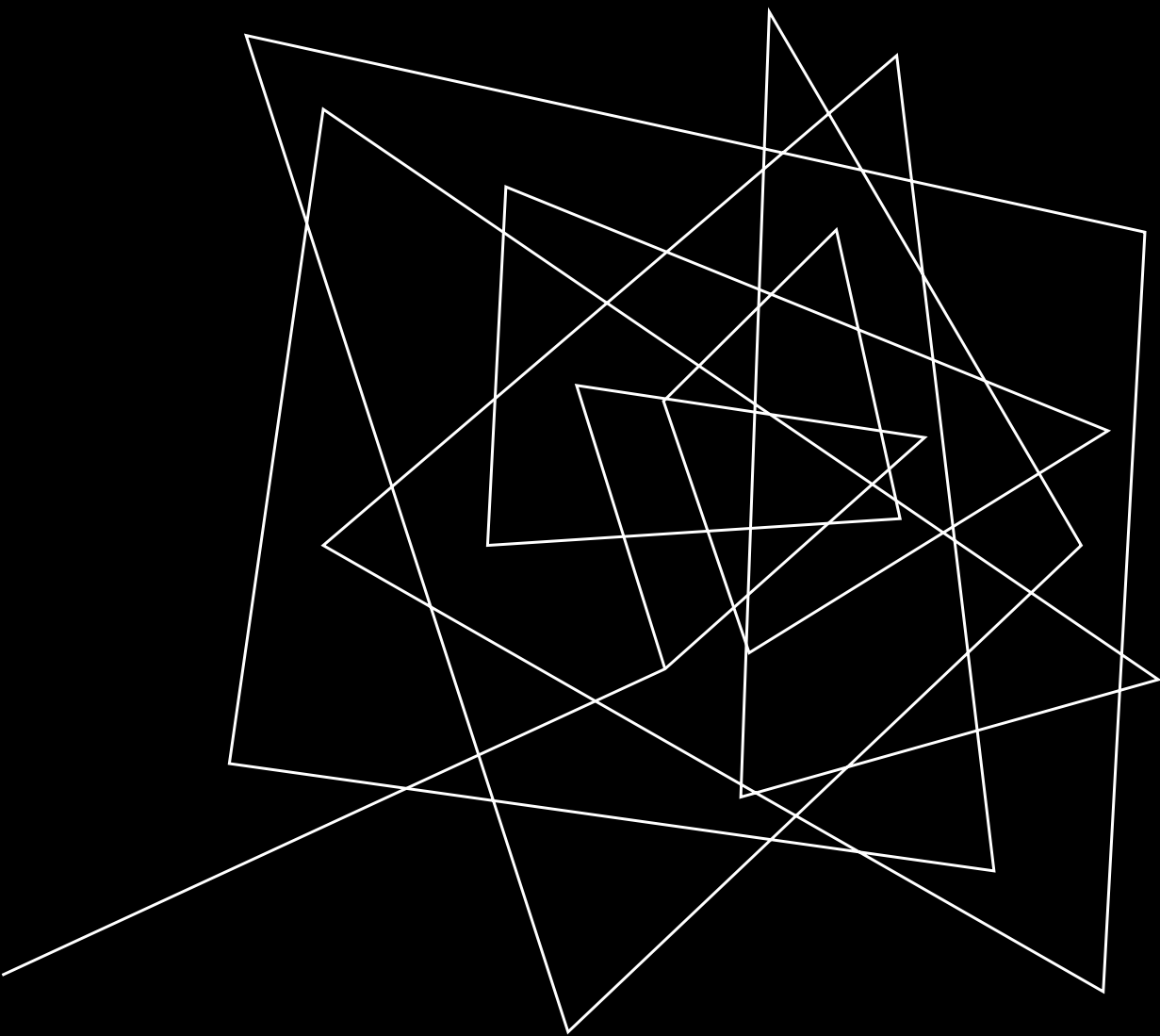
We can see that the **Gradient** of the Log Loss function takes a very simple form which **matches identically** to the gradient we observe in **linear regression**. How do we know this? Well, this is no Coincidence! Linear & Logistic Regression are both examples of **generalized linear models**, as we already saw.

To delve deeper into this, take a look at the mathematical derivation here:

<https://ml-explained.com/blog/logistic-regression-explained#loss-function>

Also, to see the proof of **Log Loss being a convex function**, visit:

<https://towardsai.net/p/l/proving-the-convexity-of-log-loss-for-logistic-regression>



# MULTI-CLASS CLASSIFICATION

# MULTI-CLASS CLASSIFICATION

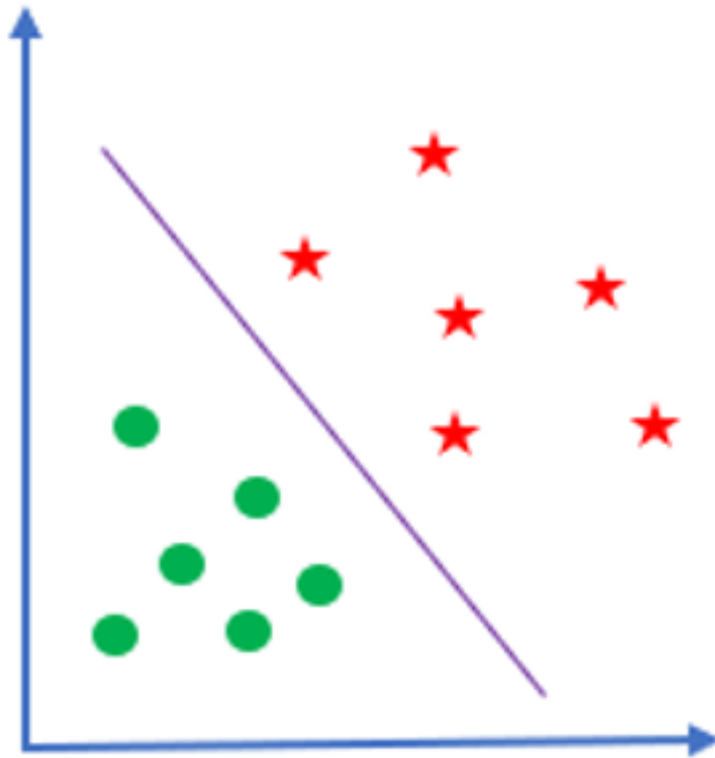
**Multi-class classification** or **Multinomial Classification** is a type of classification task where the goal is to categorize instances into one of **three or more distinct classes**. Unlike binary classification, which deals with only two possible outcomes (e.g., 0 and 1), multi-class classification involves assigning a single label from a set of more than two categories.

Logistic regression only applies to **binary** classification problems. What if we have an arbitrary number of classes **K**?

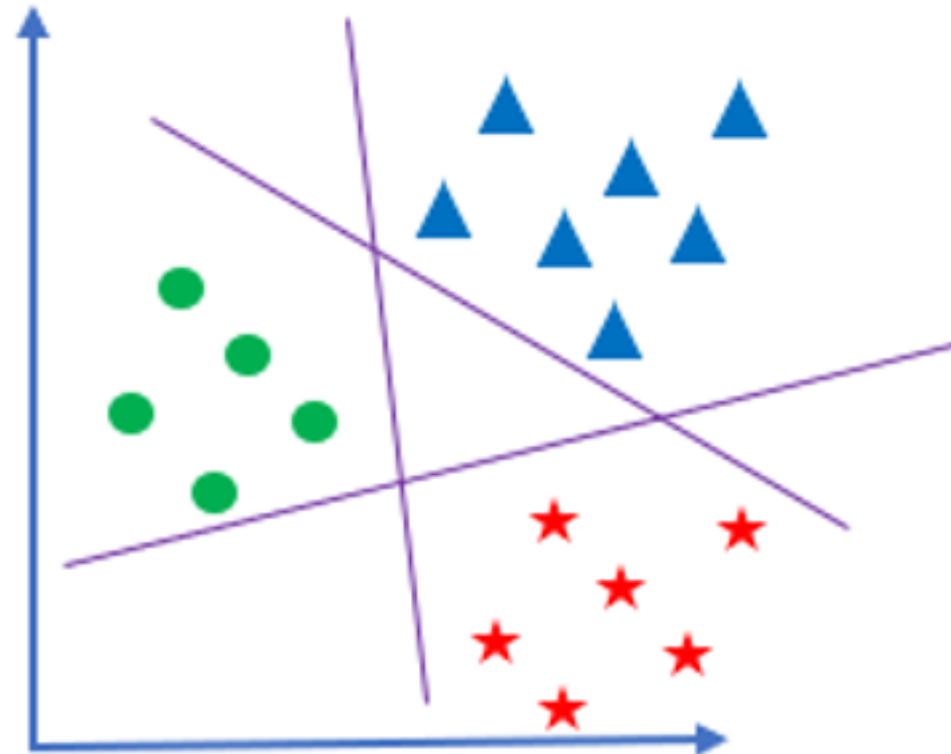
- ❑ The simplest approach that can be used with any machine learning algorithm is the **“One vs. Rest”** approach. We train one classifier for each class to distinguish that class from all the others, **extending the Binary Classification model**.
- ❑ This works but is not very elegant. Alternatively, we may fit **a probabilistic model that outputs multi-class probabilities**.

# BINARY vs MULTI-CLASS CLASSIFICATION

Binary classification



Multi-class classification



Source: <https://medium.com/@ilyurek/multi-class-classification-making-sense-of-many-possibilities-f8c14186d8da>

# ONE-vs-ONE (OvO) APPROACH

**One-vs-One (OvO)** method breaks the multi-class problem into *multiple binary classification problems between each pair of classes*.

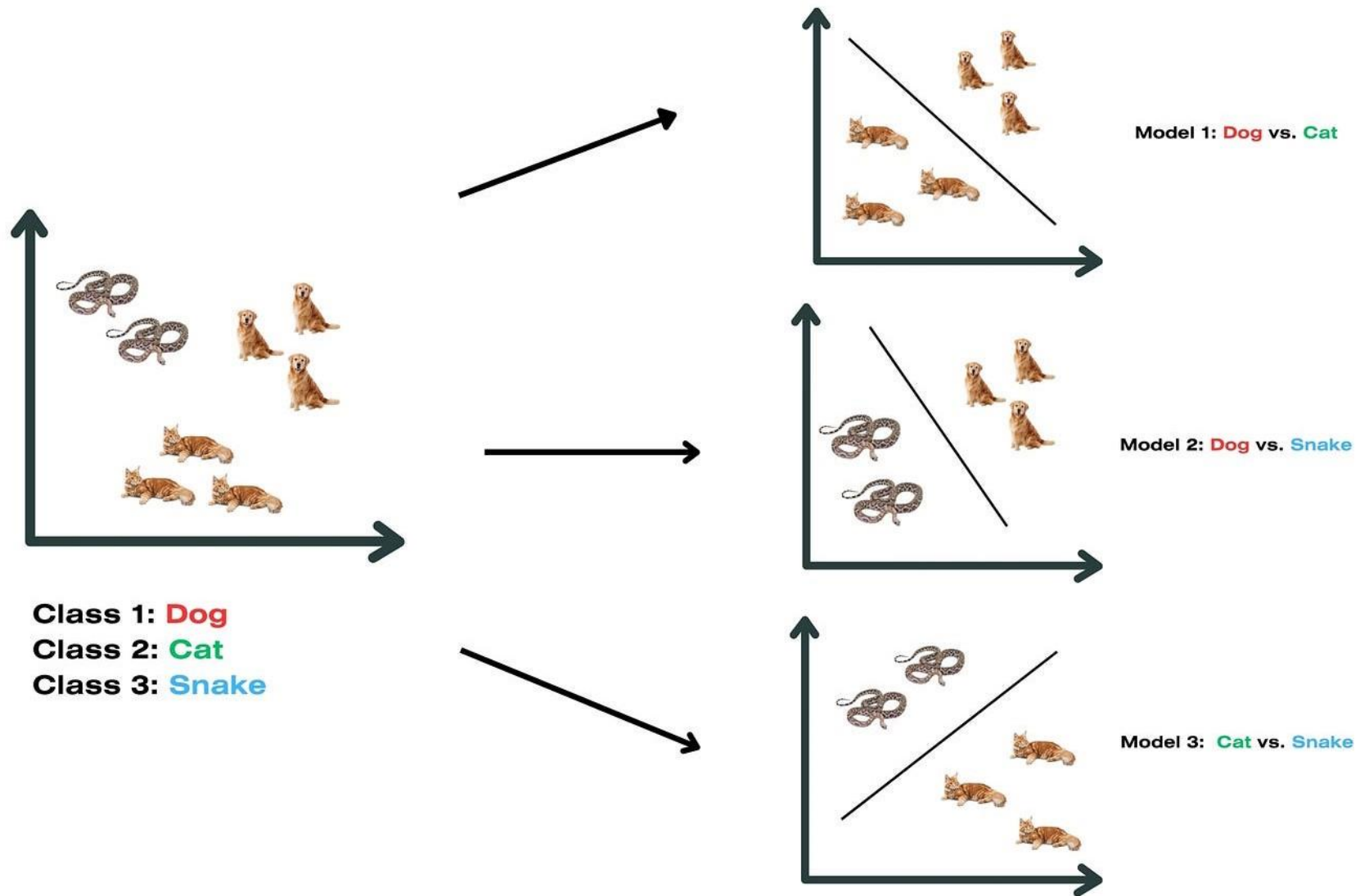
## Steps

- ❑ For each possible pair of classes, a binary classifier is trained to distinguish between those two classes only.
- ❑ In a problem with  $N$  classes,  $N(N-1)/2$  classifiers are trained, one for each pair of classes.
- ❑ When making a prediction, each classifier predicts which of the two classes the instance belongs to, and **a voting scheme** is used. The **class with the most votes is selected**.

## Issues

- ❑ Computationally expensive:  $N(N-1)/2$  classifiers are needed, which can become impractical for large  $N$ . Complexity increases abruptly as #Classes increase.

# ONE-vs-ONE (OvO) APPROACH



# ONE-vs-REST (OvR) APPROACH

**One-vs-Rest (OvR)**, also known as **One-vs-All**, involves splitting the multi-class classification problem into multiple binary classification problems, considering one of the classes as Positive (1) and the rest of them as Negative (0) at a time.

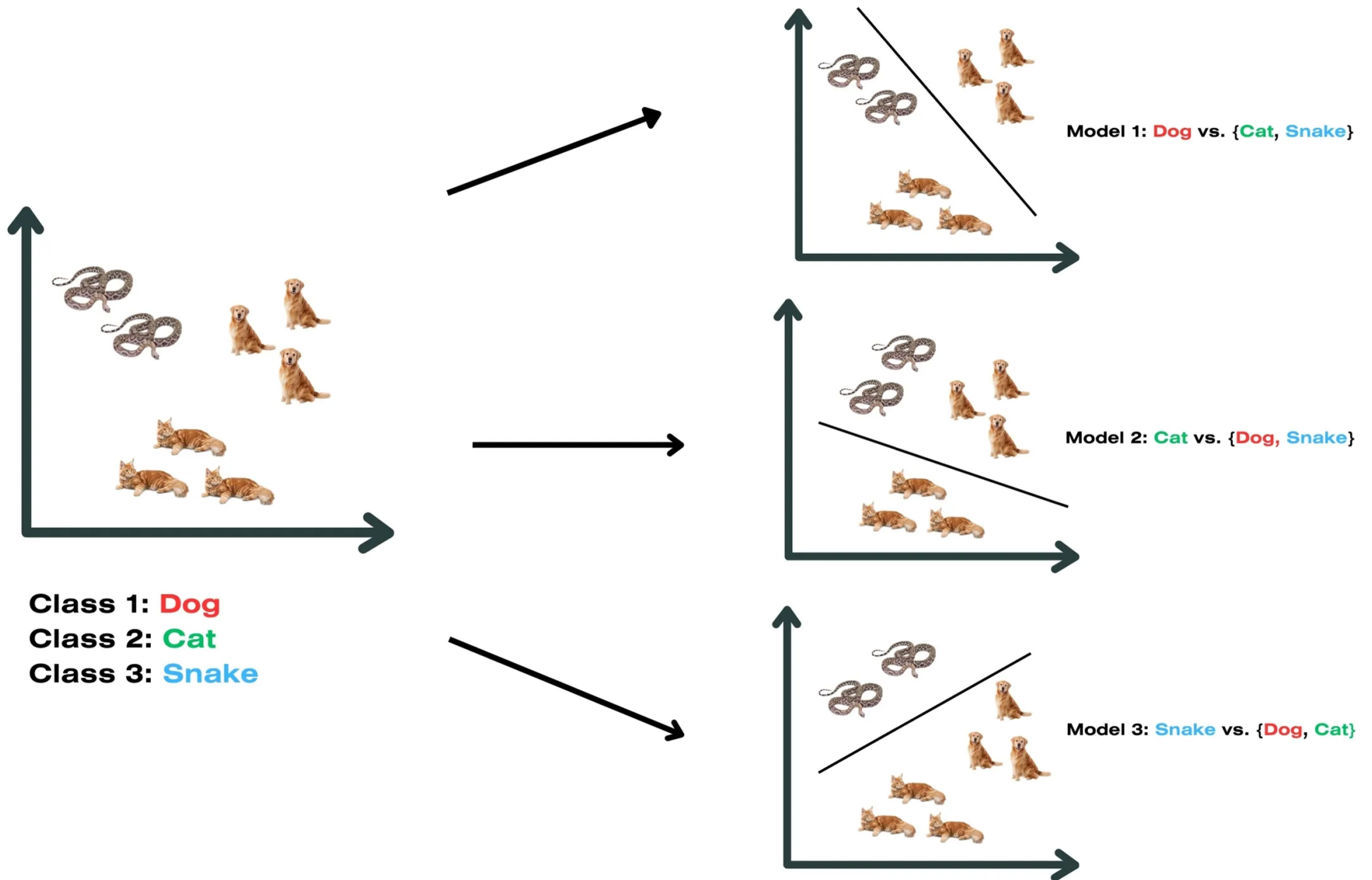
## Steps

- ❑ For each class, a separate binary classifier is trained. Each classifier is tasked with distinguishing whether an instance belongs to its corresponding class (positive) or any of the other classes (negative).
- ❑ In a problem with  $N$  classes,  $N$  binary classifiers are trained.
- ❑ When making a prediction, **each classifier outputs a score or probability**, and the class whose classifier gives the **highest score is selected**.

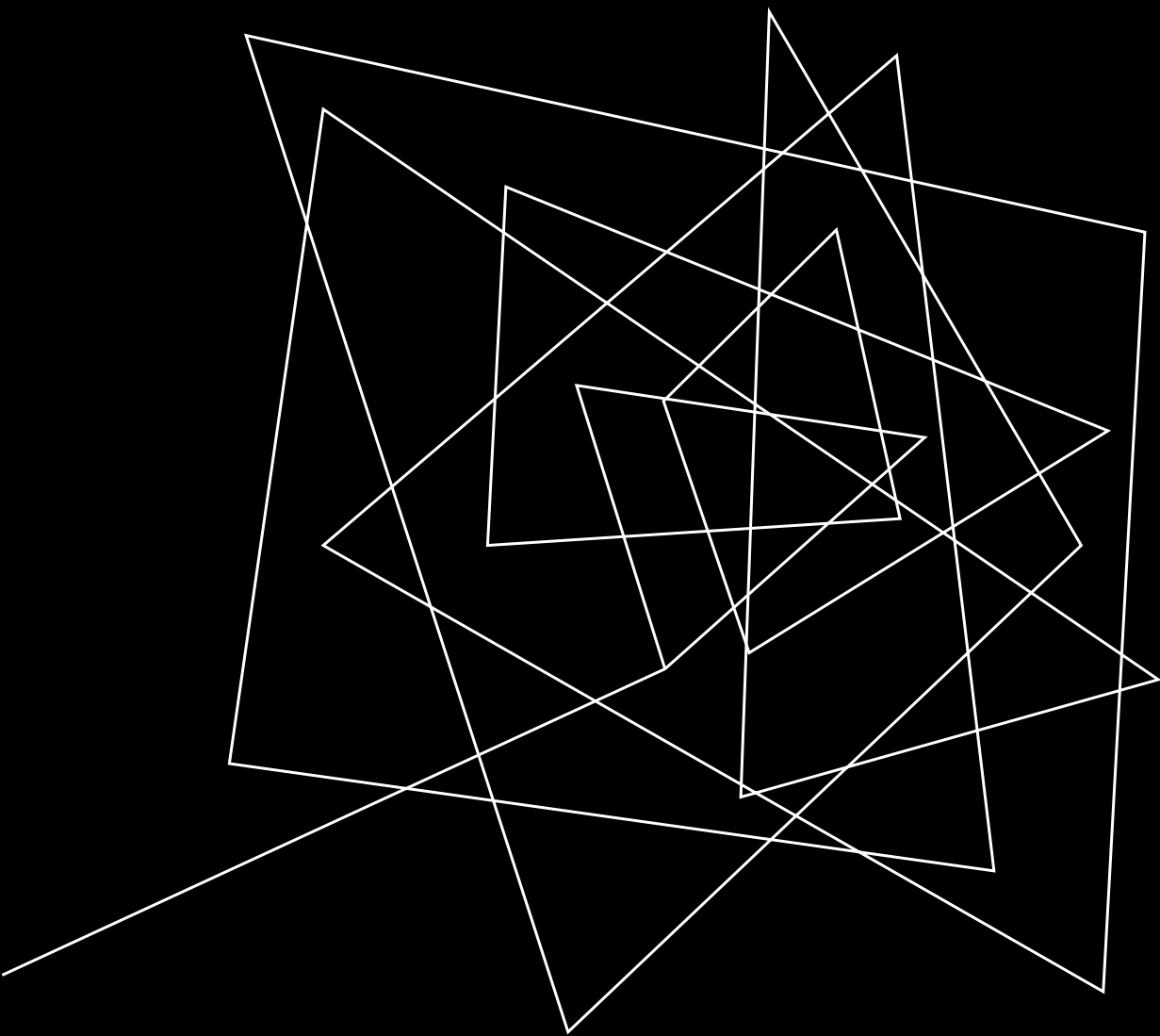
## Issues

- ❑ Requires one model to be created for each class. For example, three classes require three models. This could be an issue for large datasets, or very large numbers of classes. Also, **Optimization** becomes complicated. In short, this approach is **inefficient**.

# ONE-vs-REST (OvR) APPROACH







*SOFTMAX REGRESSION*

# THE SOFTMAX FUNCTION

The **SoftMax** function is a mathematical function commonly used in machine learning, particularly in multi-class classification problems. It converts raw model outputs (**logits**) into probabilities, ensuring that the **output values lie between 0 and 1** and that **their sum equals 1**, making them interpretable as probabilities of different classes.

## Mathematical Representation

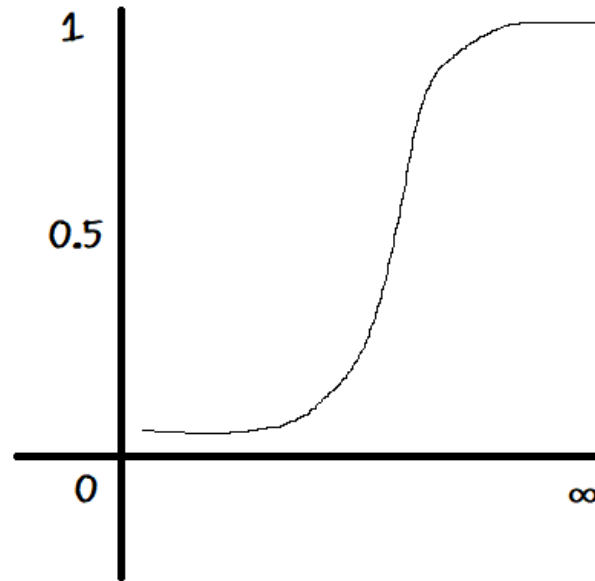
For a given vector of scores (logits)  $\mathbf{z} = [z_1, z_2, \dots, z_K]$  representing the outputs for  $K$  different classes, the SoftMax function is defined as:

$$\text{SoftMax}(\mathbf{z}) = \frac{e^z}{\sum_{j=1}^K e^z} \implies \text{SoftMax}(z_i) = \frac{\exp(z_i)}{\exp(z_1) + \exp(z_2) + \dots + \exp(z_K)}$$

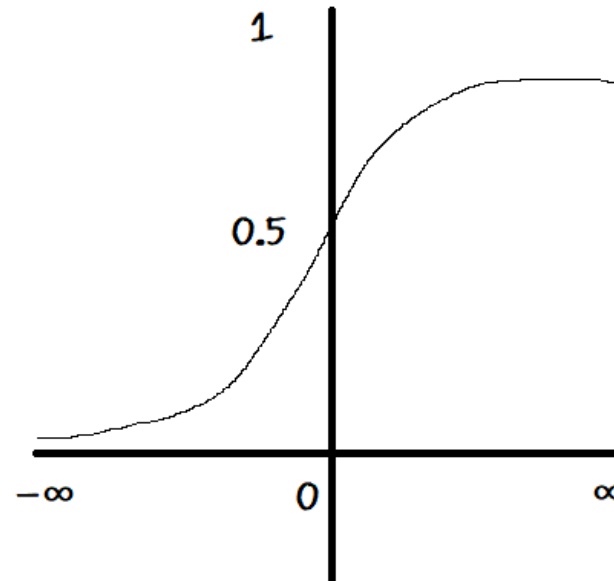
- $z_i$  is the score (logit) for class  $i$ .
- $\exp(z_i)$  is the exponential function applied to  $z_i$ , ensuring positive outputs.
- The denominator  $[\exp(z_1) + \exp(z_2) + \dots + \exp(z_K)]$  *normalizes the values* so that the outputs form a valid probability distribution.
- The sum of all outputs (for all classes) equals **1**, and each output ranges **between 0 & 1**. These two properties ensure that the outputs represent probabilities.

# THE SOFTMAX FUNCTION

The **SoftMax** function itself doesn't have a single shape, as it applies to multiple inputs simultaneously and normalizes them into probabilities. However, we can visualize how the function operates by considering its behavior on a simple **two-class** or **three-class** problem. Below we compare the shapes for the two-class case of SoftMax to Sigmoid function.



Sigmoid

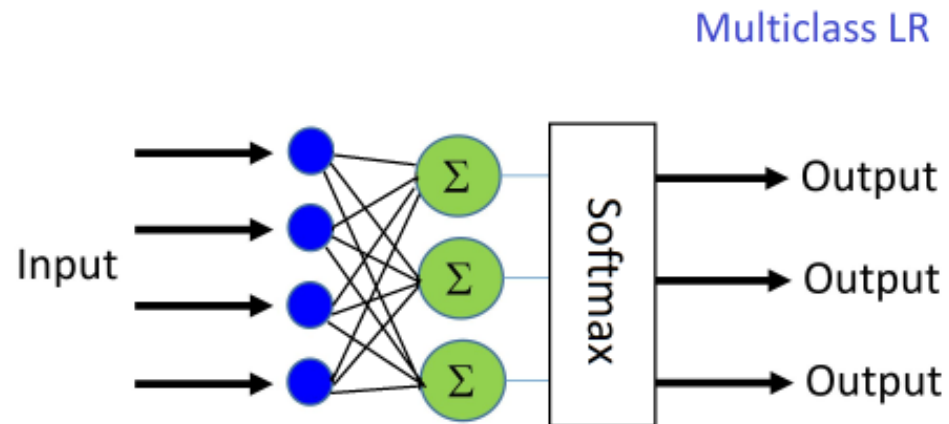
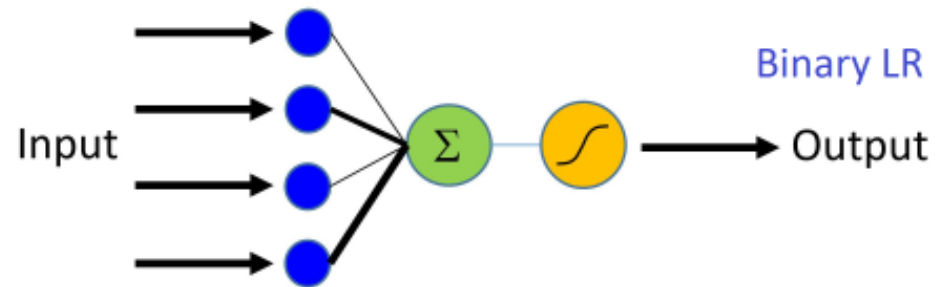


Softmax

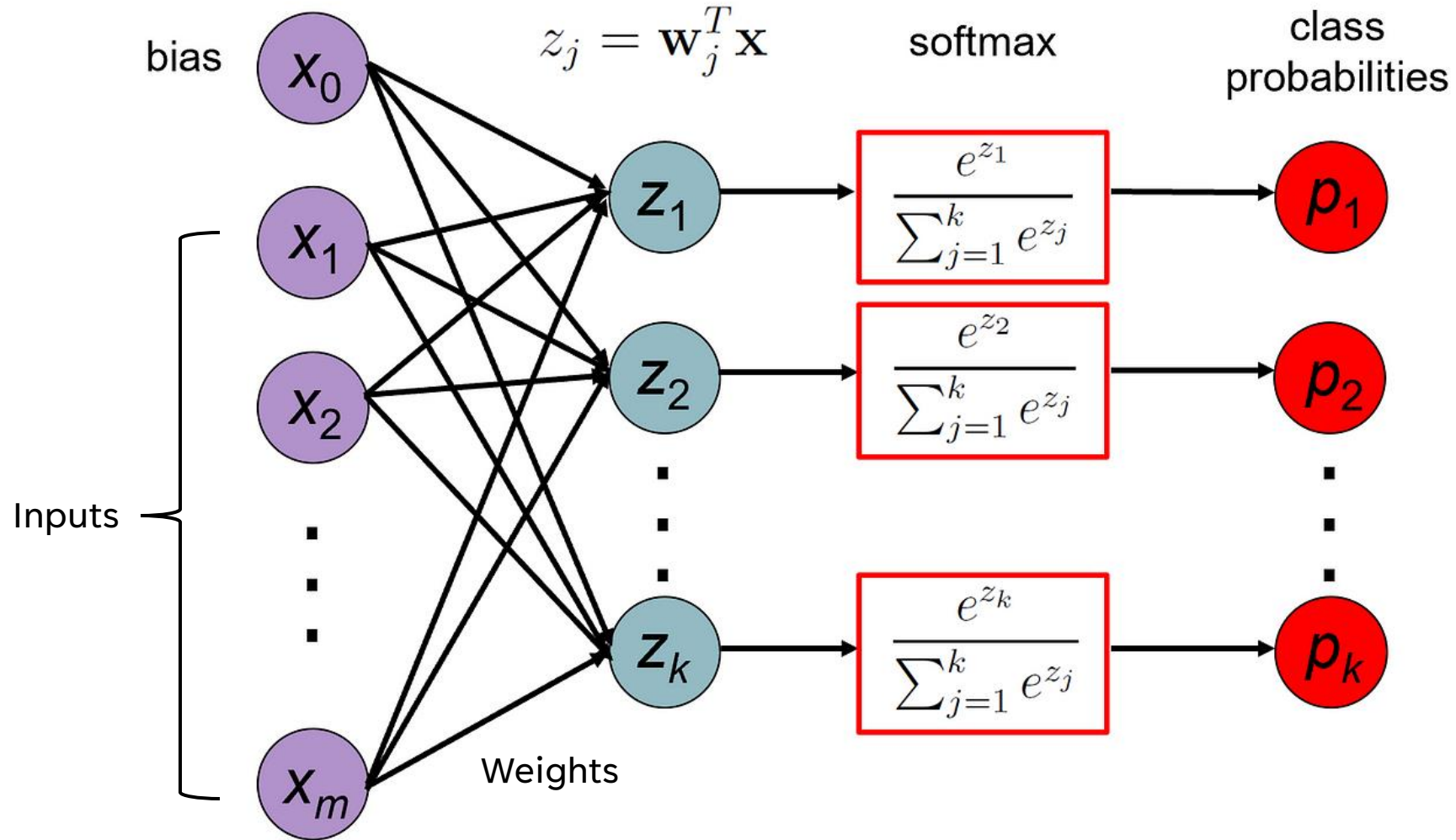
Source: <https://datascience.stackexchange.com/questions/57005/why-there-is-no-exact-picture-of-softmax-activation-function>

# SOFTMAX REGRESSION

**SoftMax Regression** (or **Multinomial Logistic Regression**) is a generalization of logistic regression to the case where we want to handle multiple classes. In binary logistic regression, we assumed that the labels were binary. So, feeding a single linear output to the sigmoid function was enough. In the case of Multi-Class LR, a linear output is needed for each of the classes, increasing the number of parameters in the process.

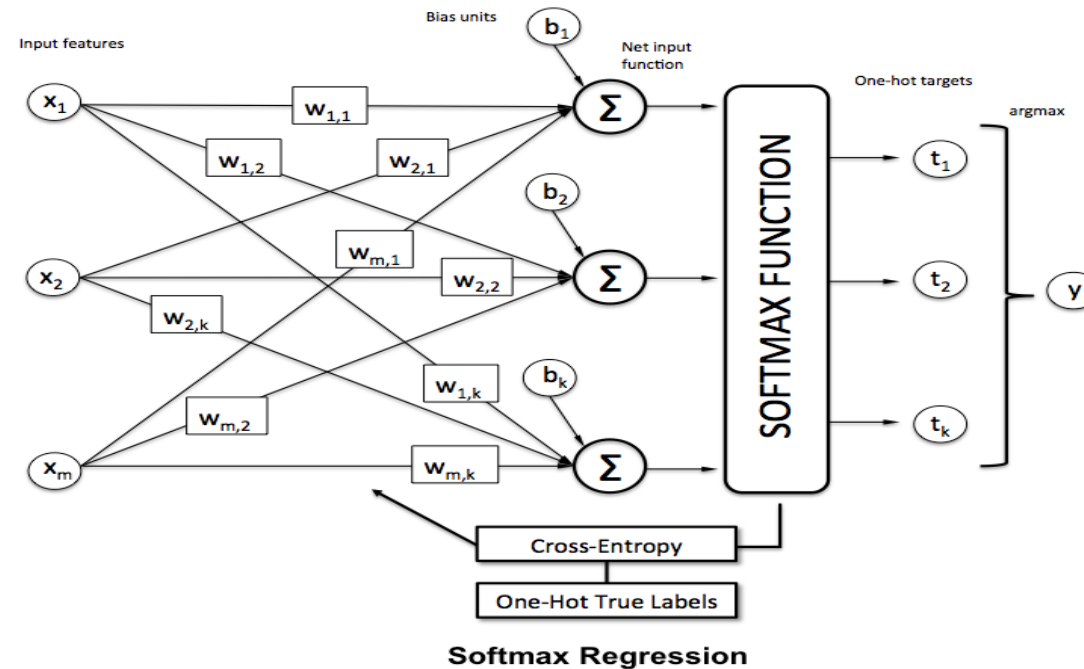
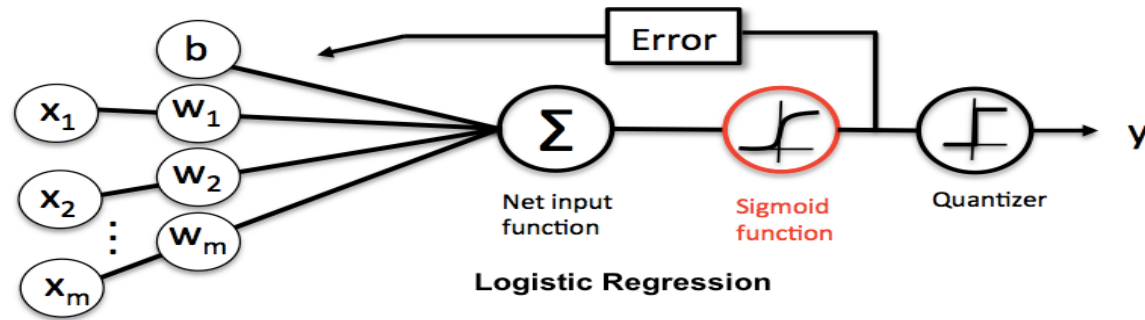


# SOFTMAX REGRESSION



Source: <https://towardsdatascience.com/deep-dive-into-softmax-regression-62deea103cb8>

# SOFTMAX VS BINARY LOGISTIC REGRESSION



Source: [https://sebastianraschka.com/faq/docs/softmax\\_regression.html](https://sebastianraschka.com/faq/docs/softmax_regression.html)

# SOFTMAX REGRESSION: MODEL

**Model:** In Multi Classification using Logistic Regression, the idea is to use a linear function of the features  $\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_D] \in \mathbb{R}^D$  to make predictions  $\mathbf{y}$  of the target class  $t \in \{0, 1, \dots, K\}$ :

## Step 1: Linear Function

$$z_i = \sum_{j=1}^n w_{ij}x_j + w_{i0} = \mathbf{w}_i^T \mathbf{x} + w_{i0} = \boldsymbol{\theta}_i^T \mathbf{x} \quad \text{where, } i \in \{0, 1, \dots, K\}$$

## Step 2: SoftMax Activation

$$\text{Class 1 Probability, } P(y=1|\mathbf{x}) = \text{SoftMax}(z_1) = \frac{\exp(z_1)}{\exp(z_1) + \exp(z_2) + \dots + \exp(z_K)}$$

$$\text{Class 2 Probability, } P(y=2|\mathbf{x}) = \text{SoftMax}(z_2) = \frac{\exp(z_2)}{\exp(z_1) + \exp(z_2) + \dots + \exp(z_K)}$$

.....

## Step 3: Prediction of Output Class (*The one with the highest probability*)

$$\text{Predicted Class, } \mathbf{y} = \operatorname{argmax} [ P(y=1|\mathbf{x}), P(y=2|\mathbf{x}), \dots, P(y=K|\mathbf{x}) ]$$

- **Objective:** To get the **Predicted Class Probability** close to the original target:  $\mathbf{P} \approx \mathbf{t}$ .

# SOFTMAX REGRESSION: MODEL (EXAMPLE)

**Dataset**

$X_0$	Living Area (feet <sup>2</sup> )	#Bedrooms	Price
1	2104	3	Medium (2)
1	1600	3	Low (1)
1	2400	3	Medium (2)
1	1416	2	Low (1)
1	3000	4	High (3)
1	.	.	.
1	.	.	.

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

As we know, Linear Function:

$$z_i = f(x) = \sum_{j=1}^n w_{ij}x_j + b_i = \sum_{j=0}^n \theta_{ij}x_j$$

Here, Number of Features,  $n = 2$ . Hence,

$$z_1 = \theta_{11}x_1 + \theta_{12}x_2 + \theta_{10}$$

$$z_2 = \theta_{21}x_1 + \theta_{22}x_2 + \theta_{20}$$

$$z_3 = \theta_{31}x_1 + \theta_{32}x_2 + \theta_{30}$$

SoftMax Activation:

$$P(y=1|x) = \frac{\exp(z_1)}{\exp(z_1) + \exp(z_2) + \dots + \exp(z_K)}$$

$$P(y=2|x) = \frac{\exp(z_2)}{\exp(z_1) + \exp(z_2) + \dots + \exp(z_K)}$$

$$P(y=3|x) = \frac{\exp(z_3)}{\exp(z_1) + \exp(z_2) + \dots + \exp(z_K)}$$

Predicted Class:

$$y = \operatorname{argmax} [ P(y=1|x), P(y=2|x), P(y=3|x) ]$$



# SOFTMAX REGRESSION: COST FUNCTION

In **SoftMax Regression** (also called multinomial logistic regression), the cost function is also based on the **Cross-Entropy** loss, which is used to measure the difference between the predicted probability distribution and the actual class labels.

$$\text{Cross-Entropy Loss, } \mathcal{L}_{CE}(\theta) = -\sum_{i=1}^K t_i \log(P_i)$$

- $\mathcal{L}_{CE}(\theta)$  is the Cross-Entropy Loss for a single prediction.
- $P_i$  is the predicted probability for the  $i^{\text{th}}$  class,  $P(y=i|x)$ .
- $t_i$  is the actual class label (**one-hot encoded**) for class  $i$ .

As we saw before, **CE Loss** function measures how well the model's predictions match the true class labels based on output probabilities and is minimized during training. Optimization algorithms, like **Gradient Descent**, can be used easily to find the minimum of the cost function. The overall Cost function takes the following shape for  $n$  training samples.

$$\text{Overall Cost, } J(\theta) = -\frac{1}{n} \sum_{j=1}^n \sum_{i=1}^K t_{ji} \log(P_{ji})$$

# SOFTMAX REGRESSION: OPTIMIZATION

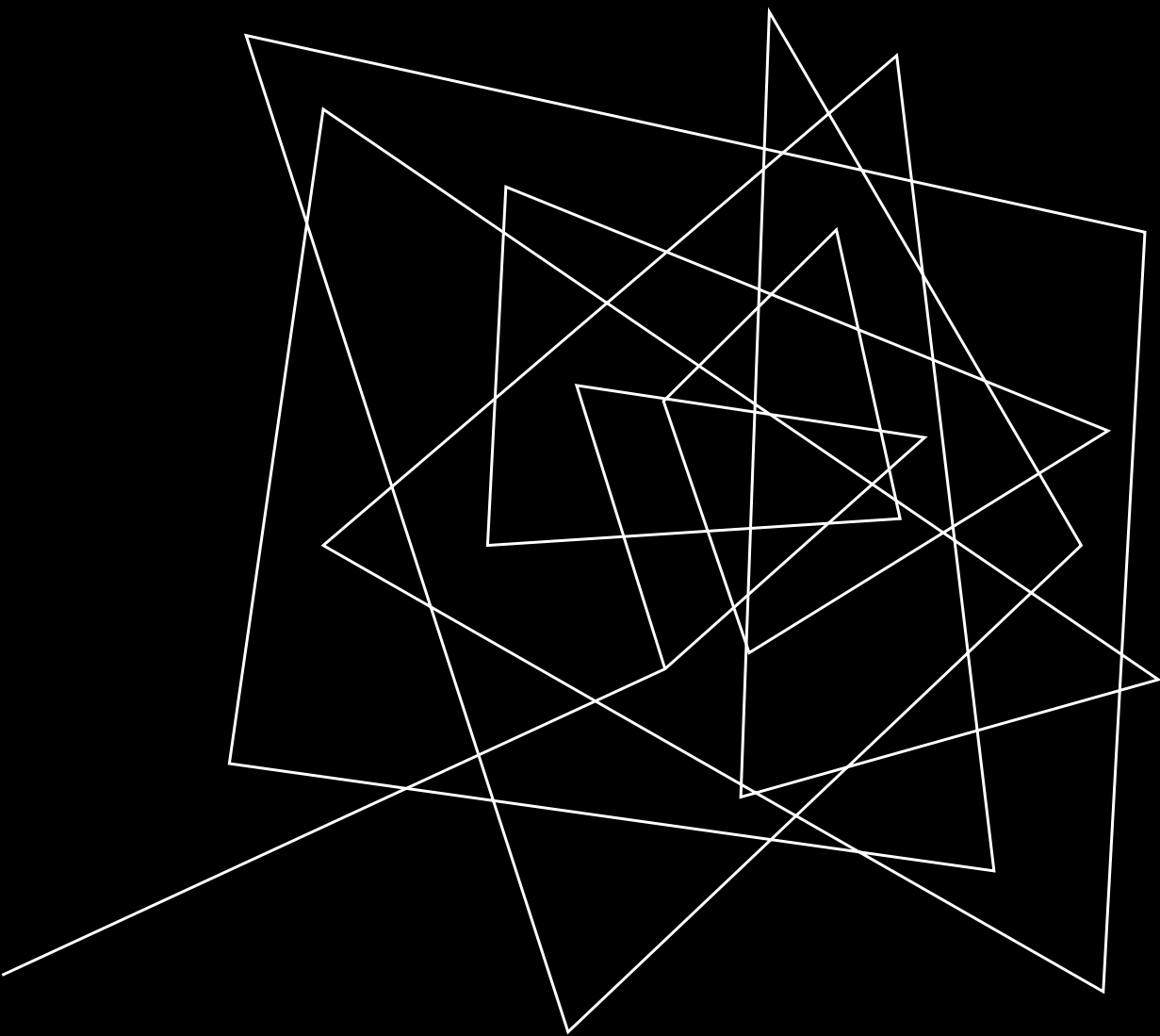
The **Objective** of SoftMax regression is also to find the parameters  $\theta$  that minimize the cost function  $J(\theta)$ . Using Gradient Descent is the way to do so. It should be noted that the derivative of the **CE Loss** function has no direct solutions, as we mentioned earlier.

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

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j} = \theta_j - \frac{\alpha}{m} \sum_{i=1}^m (\text{SoftMax}(\Theta_K^T \mathbf{x}^{(i)}) - \mathbf{t}_K^{(i)}) \cdot \mathbf{x}_j^{(i)}$$

- $\alpha$  is the **learning rate**, controlling the size of the step(changing rate).
- $\frac{\partial J(\theta)}{\partial \theta_j}$  is the partial derivative of the cost function with respect to  $\theta_j$ .
- $\text{SoftMax}(\Theta_K^T \mathbf{x}^{(i)})$  is the predicted probability of class **K** for the  $i^{\text{th}}$  sample.
- $\mathbf{t}_K^{(i)}$  is the actual output label (one hot encoded) of Class **K** for the  $i^{\text{th}}$  sample.

**Convergence:** The optimization algorithm converges when further updates to the parameters  $\theta$  result in little to no change (at global minima) in the cost function  $J(\theta)$ .



# MULTI-LABEL CLASSIFICATION

# MULTI-LABEL CLASSIFICATION

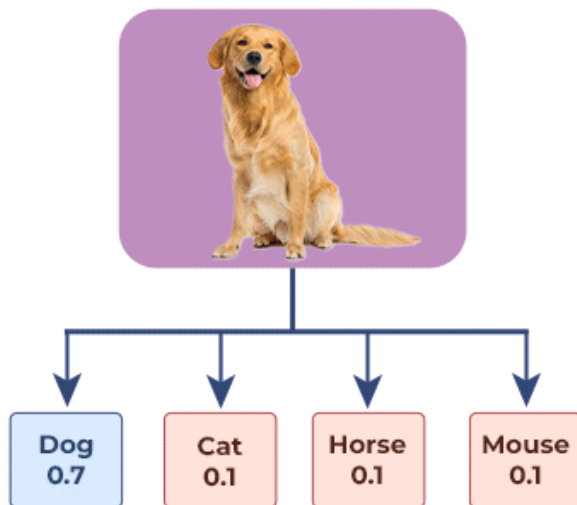
**Multi-Label Classification** involves **predicting multiple labels** for **each instance**, meaning that each input can belong to more than one class at the same time. This differs from multi-class classification, where each instance can belong to only one class out of many.

In multi-label classification, the goal is to assign a set of labels (rather than a single label) to each input. For example, in an image classification problem, a picture could be labeled as both "cat" and "dog".

In multi-label classification, **logistic regression** can be extended by **treating each label as a separate binary classification problem**. For each label, we apply logistic regression independently to predict whether or not the instance belongs to that particular class. This method is known as **Binary Relevance**. This approach is similar to One-vs-Rest (OvR) method for Multi-Class Classification apart from the final step, where **thresholding** is applied (similar to **Binary Classification** using Sigmoid activation) instead of taking the highest probability, to check whether each of the possible labels apply to that particular instance.

# MULTI-LABEL CLASSIFICATION

## Multiclass Classification

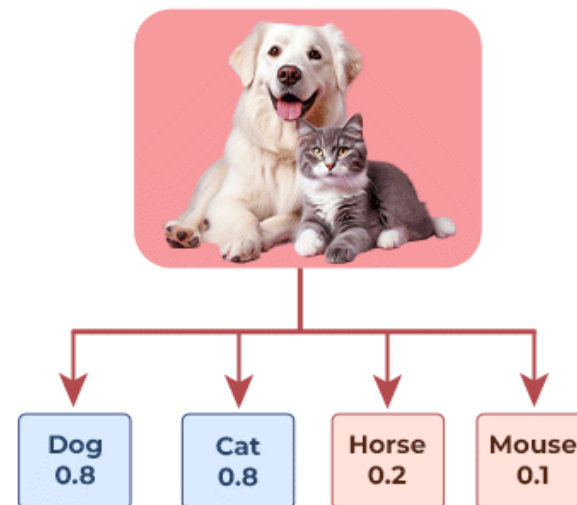


### Classes

(pick one class)

- ☒ Dog
- ☐ Cat
- ☐ Horse
- ☐ Mouse

## Multilabel Classification



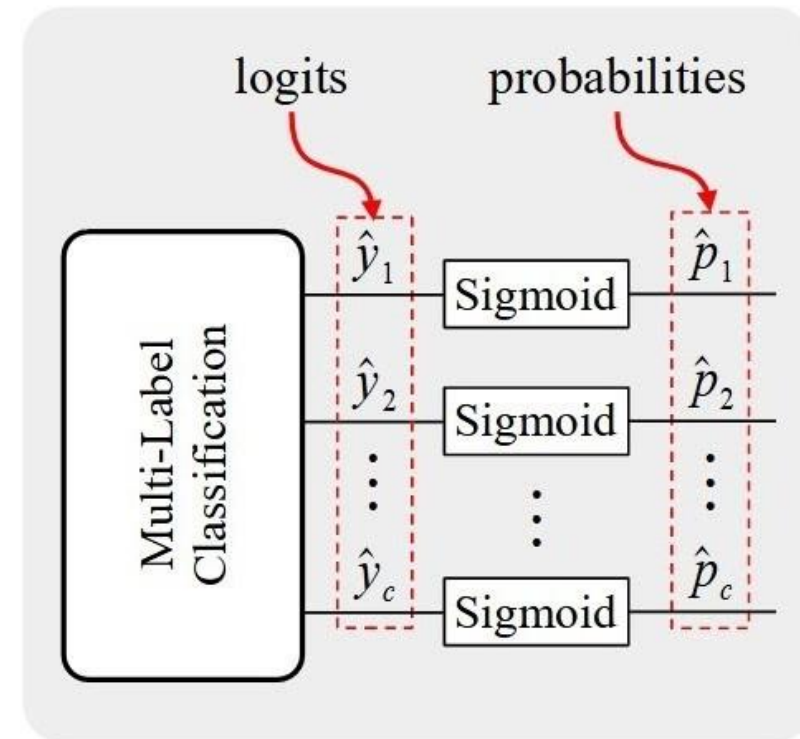
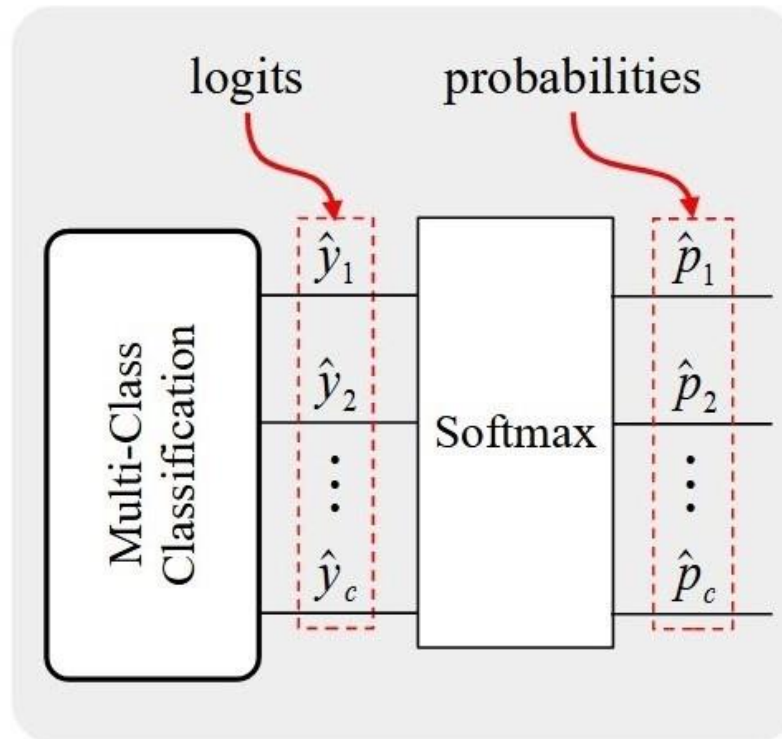
### Classes

(pick all the labels present in the image)

- ☒ Dog
- ☒ Cat
- ☐ Horse
- ☐ Mouse

# MULTI-LABEL CLASSIFICATION

Note that, for **Multi-Label classification**, the **sigmoid function** is used (not **SoftMax**) in the output layer for each class because it maps the predicted values to probabilities between 0 and 1, which can then be interpreted as **the probability of the instance belonging to each label**. This means Multi-Label Classification is essentially **K Binary Classifications**(K is the number of Classes). So, the **Model**, **Cost function**, **Optimizer** can be derived easily using **Binary LR** concepts.



Source: <https://youtu.be/Epx2V3Kd3dE?si=YhIFoiywwmVVbPKI>



EXAMPLE PROBLEMS


# EXAMPLE 1: BINARY CLASSIFICATION

Let's solve a simple linear regression problem analytically using Gradient Descent with just **2 iterations**. We'll predict house price levels based on the area (single feature). Find out the **trained parameters** after 2 iterations and also Predict the Price Level of a **1400 sq. feet** house.

## Problem Setup:

- Feature (**X**): Area of the house in square feet.
- Target (**t**): Price of the house in two categories: **High** & **Low**.

Living Area (feet <sup>2</sup> )	Price Level
1000	Low
1500	High
2000	High

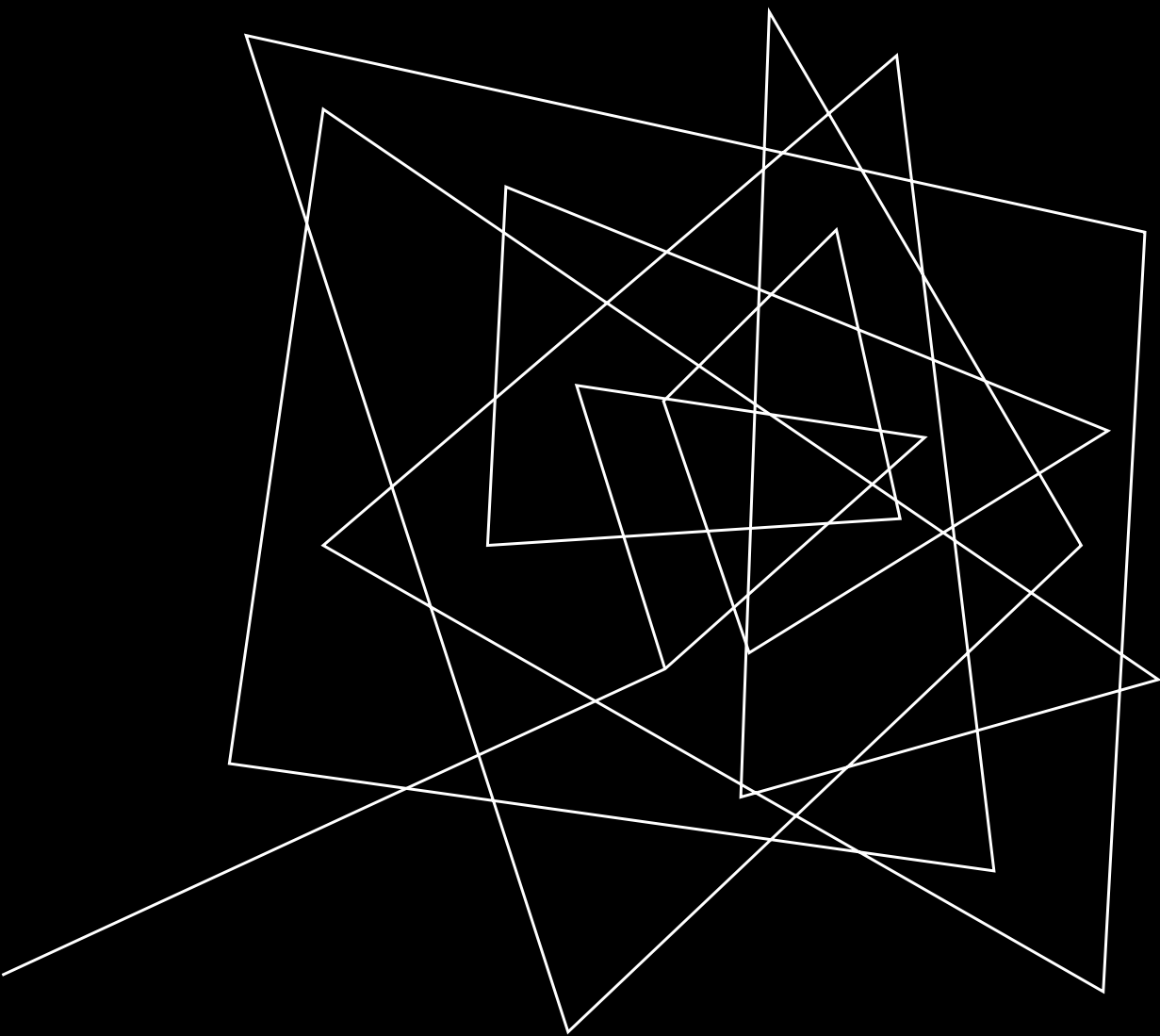




## EXAMPLE 2: MULTINOMIAL CLASSIFICATION

Living Area (feet <sup>2</sup> )	#Bedrooms	Price Level
2104	3	Medium
1600	3	Low
2400	3	Medium
1416	2	Low
3000	4	High

- ❑ Find the optimal parameters for a linear regression model to predict house price levels based on the living area & the number of Bedrooms (multiple features) using the above dataset.
- ❑ Use that Model to Determine the level of Price of a house that features 4 Bedrooms and 2200 feet<sup>2</sup> of Living Area.



OVERFITTING &  
UNDERFITTING

# GENERALIZATION: WHAT & WHY

## What is Generalization?

- **Generalization** refers to a machine learning model's *ability to perform well on new, unseen data* that was not used during the model's training process. A model that generalizes well is one that accurately predicts outcomes for data it has never encountered before, rather than just fitting the training data.

## Why is it important?

- Generalization is crucial in machine learning because it ensures that a model not only performs well on the training data but also on new, unseen data. *It is essential for creating models that are useful and reliable in real-world applications, where data can be noisy, diverse, and different from the training dataset.*

# UNDERFITTING

**Underfitting** occurs when a machine learning model fails to capture the underlying patterns in the data. It fails to learn both the training data and generalize to new, unseen data, leading to poor performance overall.

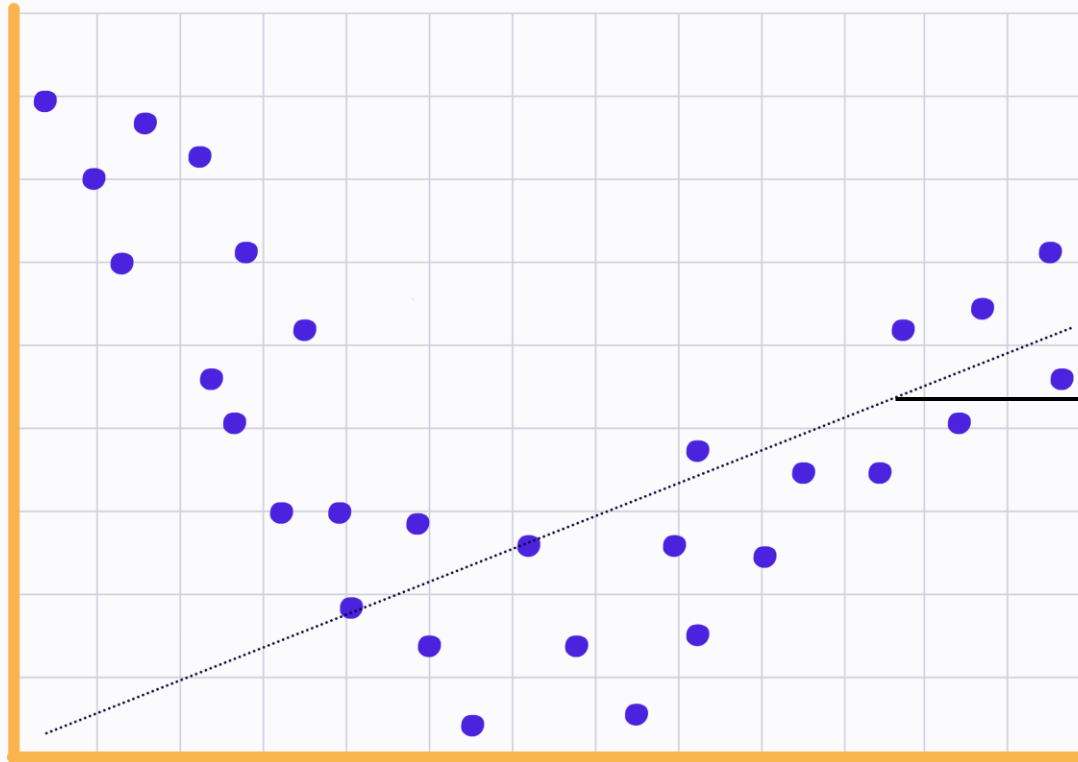
Underfitting can be linked to **High Bias & Low Variance**.

- **Bias** refers to the error introduced by approximating a real-world problem, which may be complex, with a simplified model. It represents how far off the model's predictions are from the actual values. *A model with high bias makes strong assumptions about the data, leading to an oversimplified model that cannot capture the true relationships within the data. High bias causes the model to miss important patterns, resulting in underfitting.*
- **Variance** refers to the model's sensitivity to fluctuations in the training data. It measures how much the model's predictions would change if it were trained on a different subset of the data. *When the model is too simple, making it less sensitive to the training data (low variance), it may underfit, missing important patterns. Hence, the model needs to be complex & flexible enough to learn properly from the training data.*

# THE ISSUES WITH UNDERFITTING

## Underfitting

Target,  $t$



Feature,  $x$

The overly simple model failing to capture the complex pattern of the non-linear relationship between the input & the output.

Source: <https://levity.ai/blog/overfitting-vs-underfitting-in-machine-learning>

# CAUSES OF UNDERFITTING

Common causes of **Underfitting** include:

- **Overly Simplistic Models:** Using models that are too simple for the complexity of the data, like linear regression on non-linear data.
- **Insufficient Training:** Not training the model long enough or using too few iterations in algorithms like gradient descent.
- **Inadequate Features:** Not including enough relevant features or using poorly chosen features that do not adequately represent the data.
- **Too Much Regularization:** Applying excessive regularization (e.g., too high L1 or L2 penalties), which can constrain the model too much, leading to underfitting.

# ADDRESSING UNDERFITTING

- **Increasing Model Complexity:** Using a more complex model that can capture the underlying patterns, such as adding polynomial features, switching to a more sophisticated algorithm, or increasing the number of layers/nodes in a neural network.
- **Feature Engineering:** Improving the features used in the model by adding new relevant features, removing irrelevant ones, or transforming existing features.
- **Reducing Regularization:** If the model is underfitting due to excessive regularization, reducing the regularization strength can allow the model to fit the data better.
- **More Training:** Ensuring the model is trained sufficiently by increasing the number of iterations, epochs, or using a larger dataset.

# OVERFITTING

**Overfitting** occurs when a machine learning model fits the training data *too well*. In such cases, the model *learns not only the underlying patterns in the training data but also the **noise** and **random fluctuations***. As a result, the model performs exceptionally well on the training data but poorly on new, unseen data because it **fails to Generalize**.

Overfitting can be linked to **Low Bias & High Variance**.

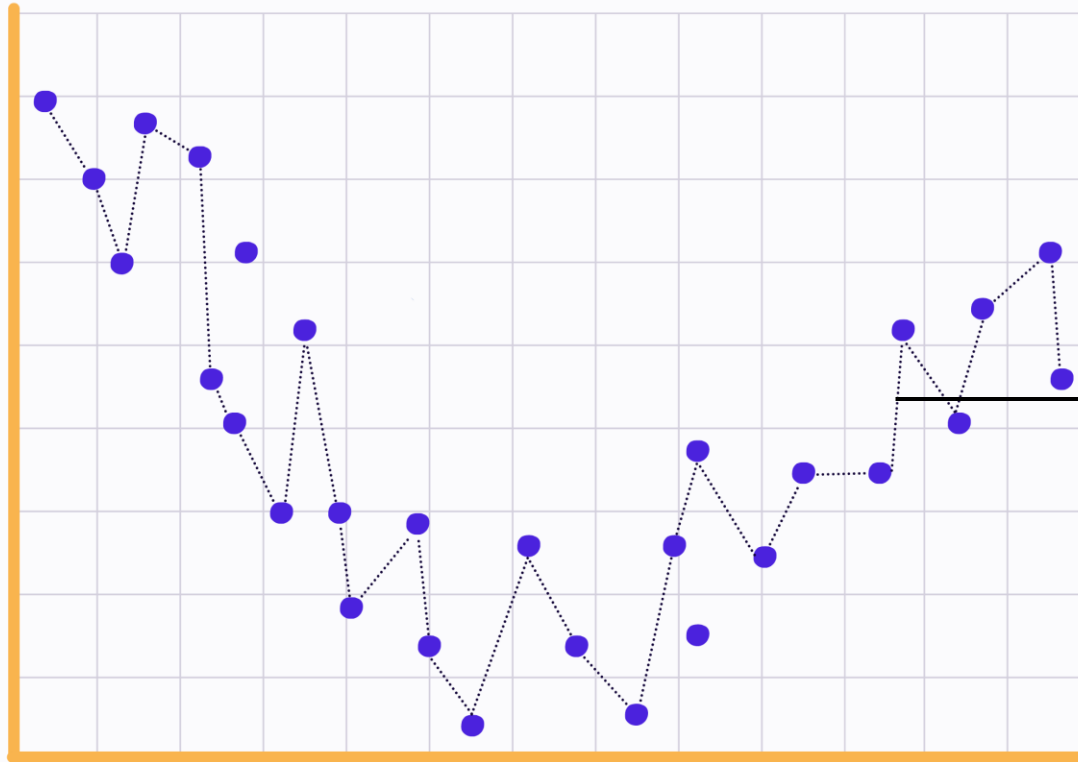
- Overfitting is directly related to **high variance** in a model. Variance refers to the model's sensitivity to small fluctuations in the training data. A model with high variance is overly complex, capturing all the nuances of the training data, including noise. *This makes the model's predictions highly sensitive to the specific training data used, leading to poor performance on new data.*
- An overfitted model typically has **low bias** because it fits the training data very closely. While low bias is generally desirable, in the case of overfitting, the model has *too little bias, meaning it tries to model every detail and noise in the data rather than just the underlying patterns.*



# THE ISSUES WITH OVERFITTING

## Overfitting

Target,  $t$



Feature,  $x$

The overly complex model tries too hard to fit all the training data, capturing noise in the process. This won't help to Generalize on unseen data.

Source: <https://levity.ai/blog/overfitting-vs-underfitting-in-machine-learning>

# CAUSES OF OVERFITTING

Common causes of **Overfitting** include:

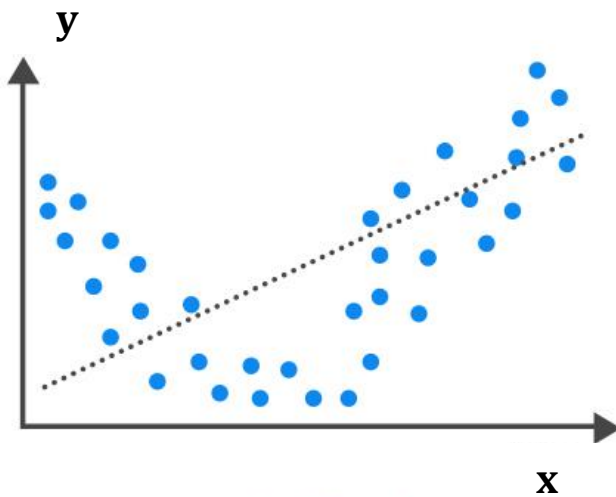
- **Excessive Model Complexity:** Using models that are too complex relative to the size and nature of the data (e.g., deep neural networks with many layers).
- **Insufficient Training Data:** Having too little data for the model to learn general patterns, causing it to fit to noise.
- **Too Many Features:** Including too many irrelevant or redundant features can lead to overfitting as the model tries to account for every feature, including those that don't contribute to the target variable.
- **Lack of Regularization:** Not applying techniques like L1, L2 regularization, or dropout can lead to overfitting, as the model is not penalized for becoming overly complex.

# ADDRESSING OVERFITTING

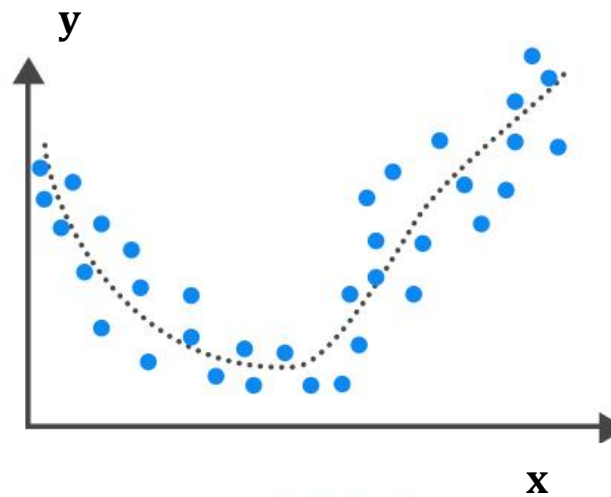
- **Simplifying the Model:** Using a less complex model that is better suited to the amount of data and the problem at hand.
- **Regularization:** Applying techniques like L1 (lasso) or L2 (ridge) regularization to penalize large coefficients and reduce model complexity.
- **Reducing the Number of Features:** Using feature selection techniques to remove irrelevant or redundant features.
- **Increasing the Amount of Data:** More training data can help the model learn general patterns and reduce the impact of noise.
- **Early Stopping:** In iterative algorithms like gradient descent, monitoring performance on **Validation Set** and stopping training once the performance starts to degrade is a common technique, as it indicates overfitting.

# BIAS vs VARIANCE TRADE-OFF

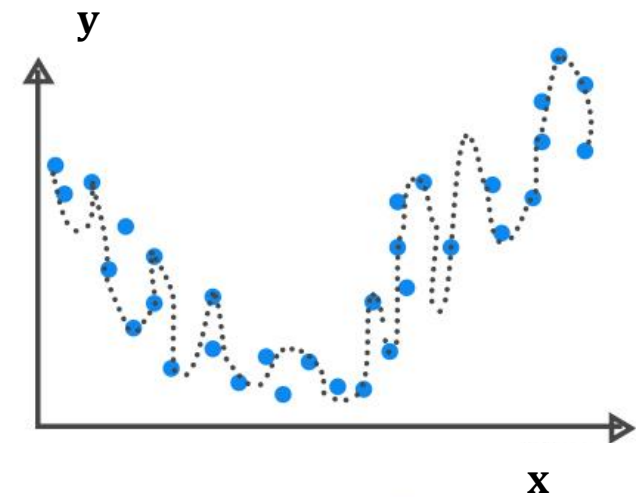
## Generalization and Overfitting



**Underfitted**  
(High bias error)



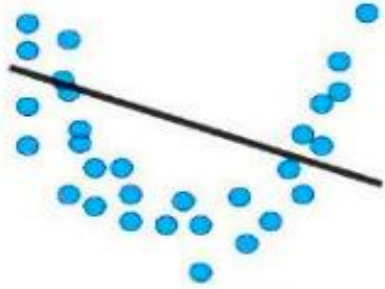
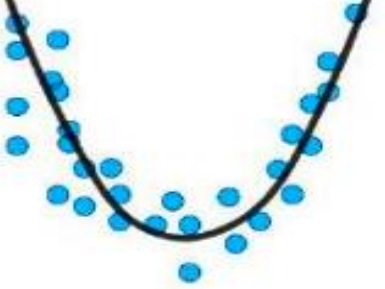
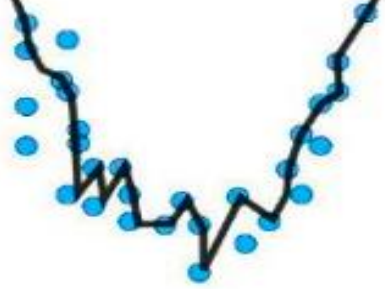
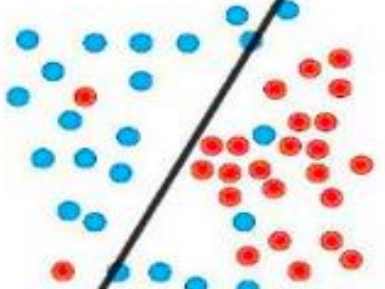
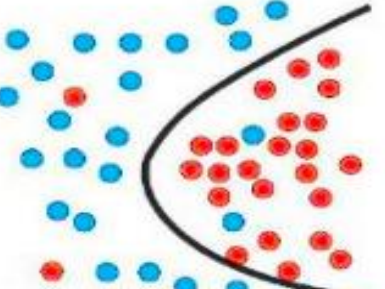
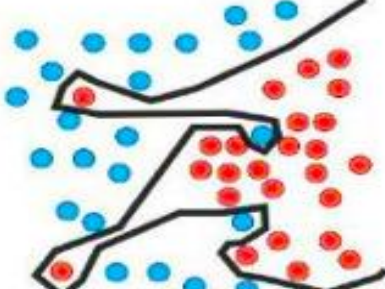


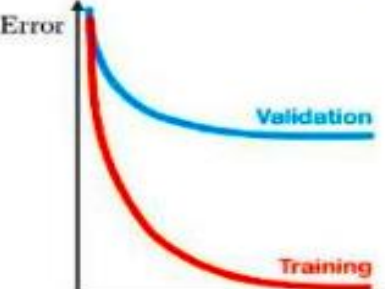
**Good Fit/Robust**  
(Balance between  
bias and variance)

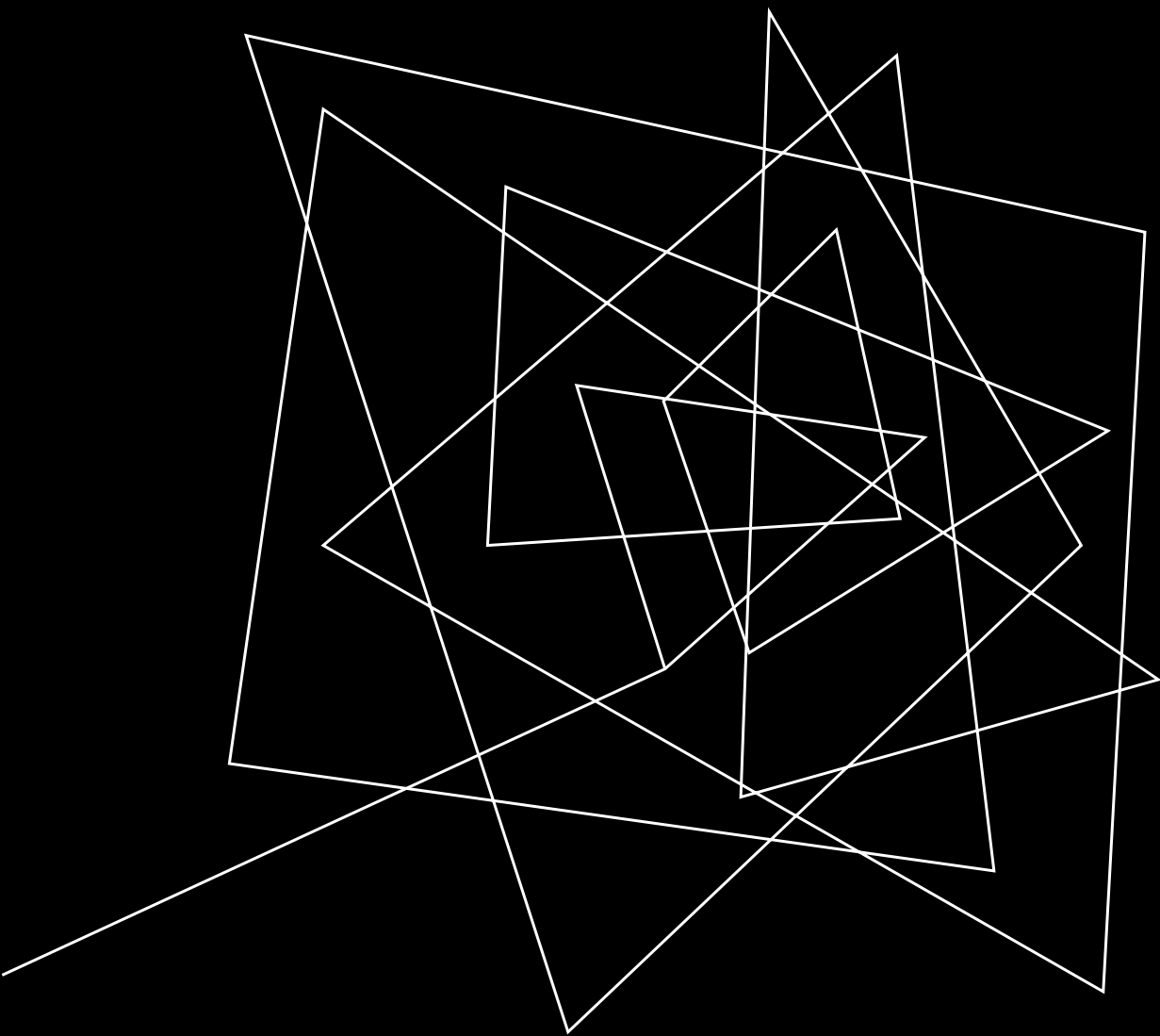


**Overfitted**  
( High variance error)

Source: <https://analystprep.com/study-notes/cfa-level-2/quantitative-method/overfitting-methods-addressing/>

# THE OPTIMAL FIT

	Under-fitting	Optimal-fitting	Over-fitting
Regression			
Classification			
Deep learning			



REGULARIZATION

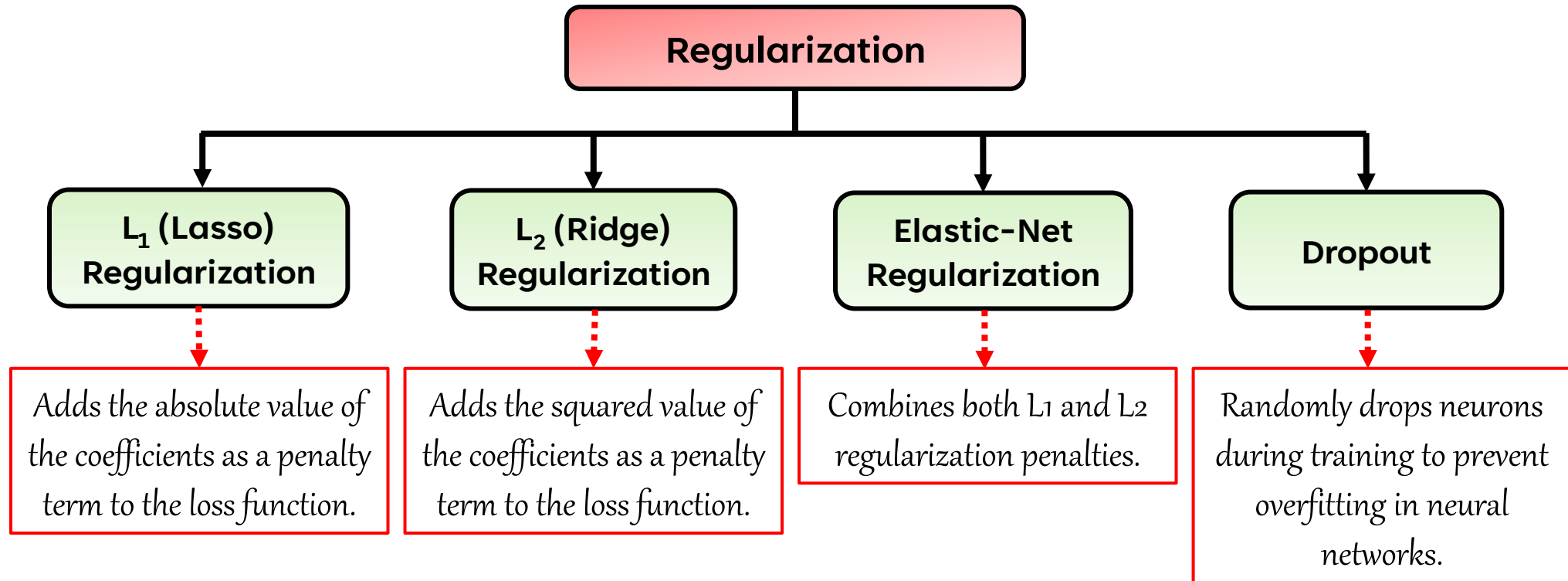
# REGULARIZATION

**Regularization** is a technique used to *prevent overfitting* by adding a penalty to the loss function of a machine learning model. The goal is to discourage the model from becoming overly complex by penalizing large weights or coefficients, thus encouraging the model to *generalize better* to unseen data.

## Why Regularization is Important

- **Preventing Overfitting:** Regularization reduces the model's tendency to overfit the training data by simplifying the model. This allows the model to perform better on new, unseen data.
- **Controlling Model Complexity:** By penalizing large weights, regularization controls the complexity of the model, ensuring that it does not try to fit every small fluctuation in the training data.
- **Improvement of Generalization:** Regularization helps the model focus on the most important features, leading to better generalization and more accurate predictions on test data.

# TYPES OF REGULARIZATION



**Think:** Can you suggest any other way to Regularize the Parameters/Prevent Overfitting?



# $L_2$ (RIDGE) REGULARIZATION

**$L_2$  Regularization** adds a penalty equal to the sum of the squared values of the model coefficients (weights) to the loss function. This encourages the model to keep the weights small, but not necessarily zero. It's also known as **Ridge Regularization**.

**Mathematical Formulation:**

$$L_2 \text{ Penalty} = \frac{1}{2} \sum_{j=0}^n \theta_j^2$$

$$\text{So, Regularized Cost Function} = \text{Original Cost Function} + \frac{\lambda}{2} \sum_{j=0}^n \theta_j^2$$

Where,  $\lambda$  is the *regularization parameter* and  $\theta_j$  are the model parameters.

**Regularization Parameter,  $\lambda$ :**  $L_2$  regularization strikes a balance between fitting the training data well and keeping the model simple. The regularization parameter  $\lambda$  controls this balance. The larger the value of  $\lambda$ , the stronger the regularization, and the simpler the model becomes.

# $L_2$ (RIDGE) REGULARIZATION

## How Overfitting is Prevented:

In overfitting, the model tries to fit the training data too closely, including noise and outliers, which can result in large weight values. Large weights make the model sensitive to small fluctuations in the data, leading to high variance and poor generalization to new data.

$L_2$  regularization discourages large weights by adding the squared value of the weights to the loss function. The model is incentivized to keep the weights small because increasing any weight would lead to a larger penalty and, consequently, a higher loss. The model now **can no longer minimize the original loss function without considering the penalty on large weights. As a result, the model opts for smaller weights, leading to a smoother decision boundary/regression line that captures the general trend of the data rather than the noise.**

This smoothing effect reduces overfitting by ensuring the model does not become too complex or sensitive to the training data, improving its generalization to new data.

# $L_1$ (LASSO) REGULARIZATION

**$L_1$  Regularization** adds a penalty equal to the sum of the absolute values of the model coefficients to the loss function. This can lead to some weights being exactly zero, effectively performing feature selection. It's also known as **Lasso Regularization**.

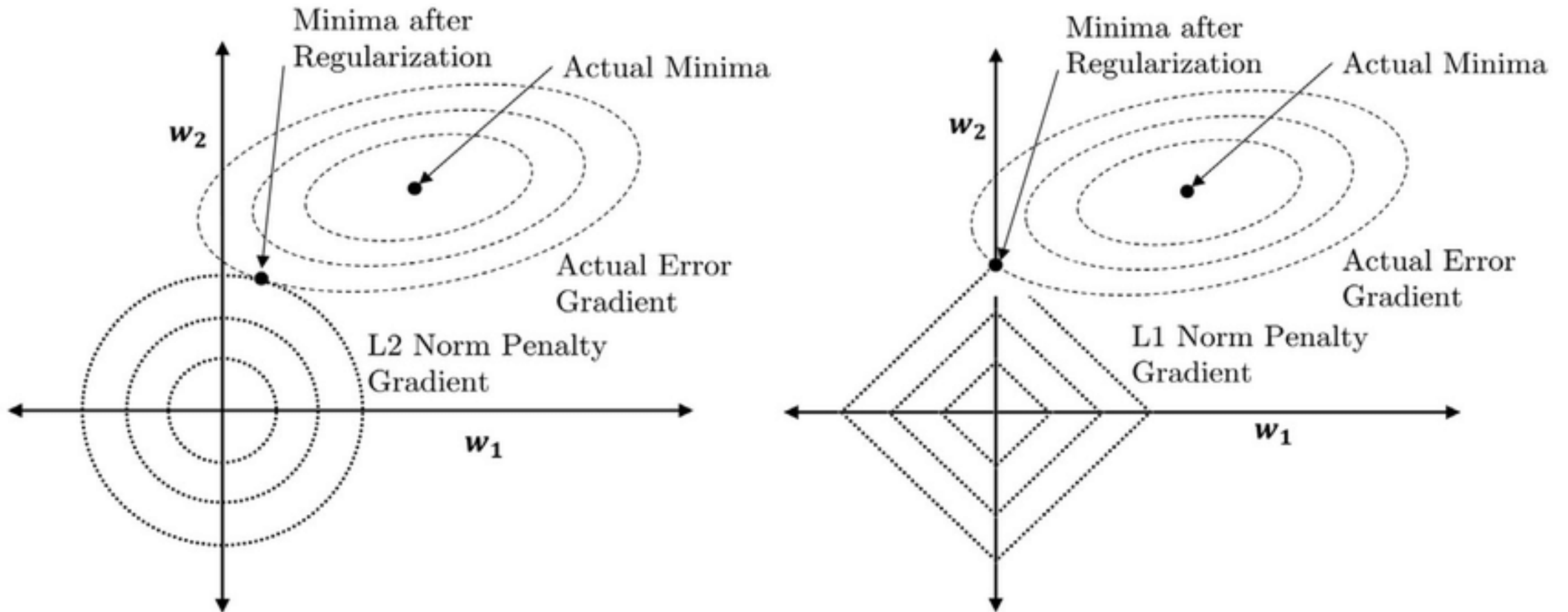
**Mathematical Formulation:**

$$L_1 \text{ Penalty} = \sum_{j=0}^n |\theta_j|$$

$$\text{So, Regularized Cost Function} = \text{Original Cost Function} + \lambda \sum_{j=0}^n |\theta_j|$$

**Feature Selection:** In  $L_1$  regularization (Lasso), the penalty term is the absolute value of the weights. This can cause some weights to shrink exactly to zero because the optimization process favors **sparsity** by minimizing the loss function, effectively performing feature selection by eliminating less important features from the model.  $L_1$  is therefore useful for feature selection, as we can drop any variables associated with coefficients that go to zero.  $L_2$ , on the other hand, is useful when we have collinear/codependent features.

# $L_2$ vs $L_1$ REGULARIZATION



Source: Santosh, Kc & Das, Nibaran & Ghosh, Swarnendu. (2022). Deep learning models. 10.1016/B978-0-12-823504-1.00013-1.

# REFERENCES

1. University of Toronto - CSC411/2515: Machine Learning and Data Mining ([https://www.cs.toronto.edu/~rgrosse/courses/csc311\\_f20/](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](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
7. <https://www.linkedin.com/pulse/binary-classification-pavan-kumar-4iqbc/>
8. <https://medium.com/@murpanironit/a-comprehensive-guide-to-multiclass-classification-in-machine-learning-c4f893e8161d>

A series of white, thin, overlapping geometric lines on a black background, forming a complex, abstract shape on the left side of the slide.

THANK YOU