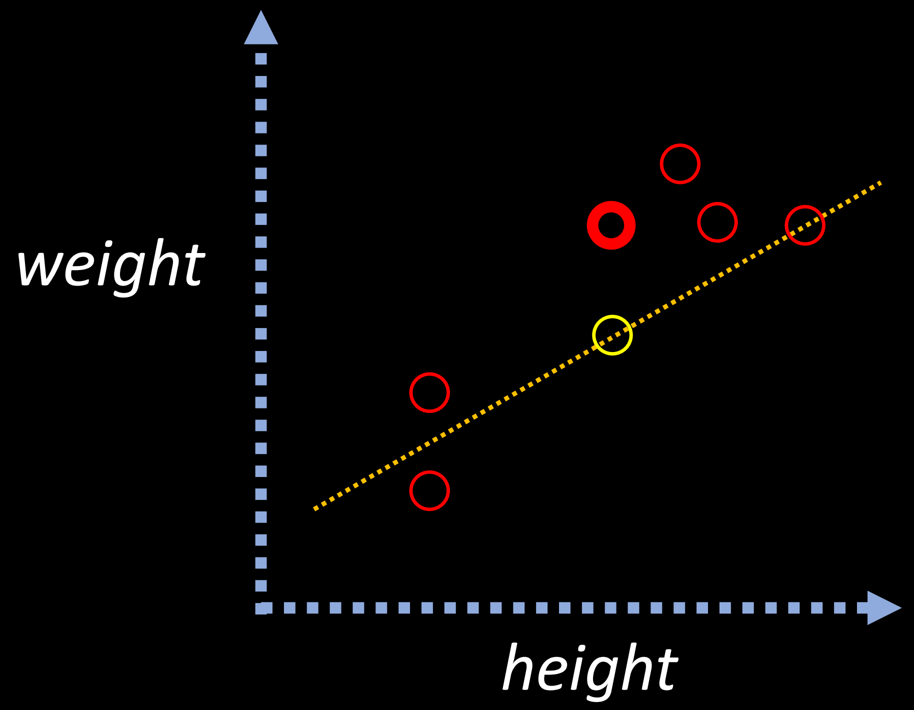
Linear & Logistic Regression

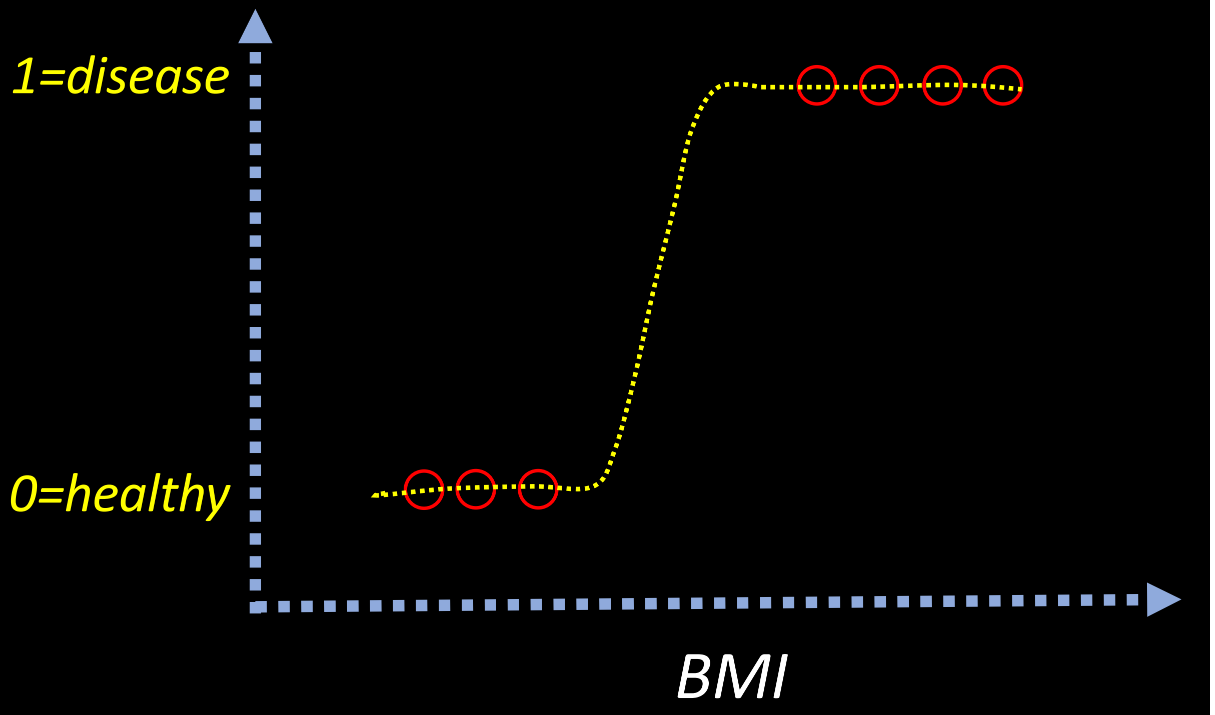
Regression vs Classification

Regression: Predicts continuous outcomes by finding the best-fit line through data points. Used for forecasting and understanding relationships between variables.

Classification: Predicts the probability of an instance belonging to a particular class.

Regression vs Classification





Regression:

Predicting drug dosage based on patient weight
Estimating bone density changes over time in osteoporosis patients
Analyzing the relationship between blood pressure and salt intake

Classification:

Predicting the likelihood of heart disease based on cholesterol levels
Determining the probability of remission in cancer patients given certain treatment
Classifying medical images as showing presence or absence of a tumor

```python
# Linear Regression: Predicting drug dosage based on patient
weight
from sklearn.linear_model import LinearRegression
import numpy as np

# Sample data: patient weights (kg) and corresponding drug
dosages (mg)

weights = np.array([[60], [70], [80], [90], [100]])
dosages = np.array([100, 115, 130, 145, 160])
```

Code Block 1

```
model = LinearRegression()

model.fit(weights, dosages)

# Predict dosage for a new patient weighing 75 kg
new_weight = np.array([[75]])
predicted_dosage = model.predict(new_weight)

print(f"Predicted dosage for 75 kg: {predicted_dosage[0]:.2f} mg")

# Output: ``Predicted dosage for 75 kg: 122.50 mg ```
```

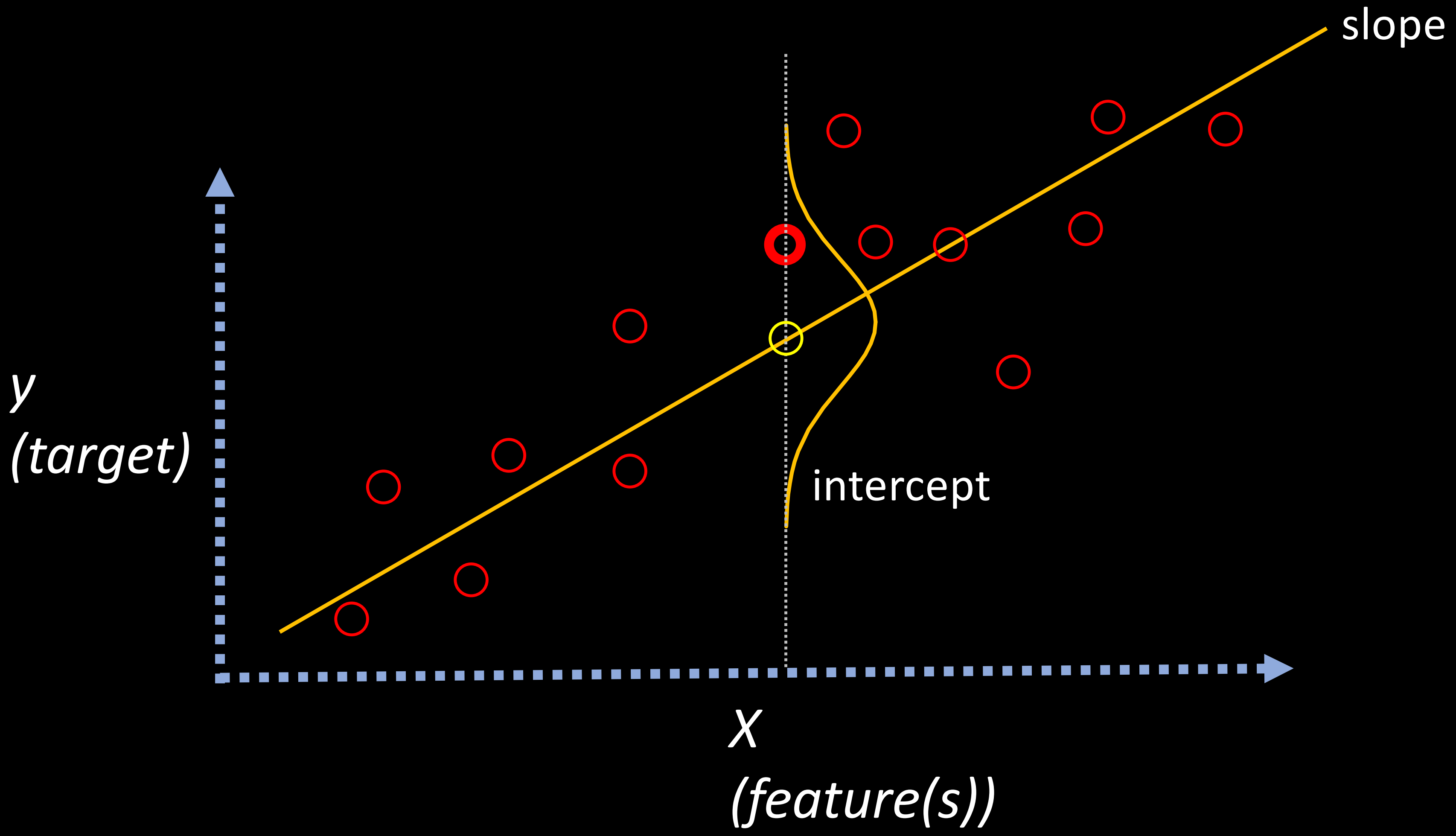
Code Block 1

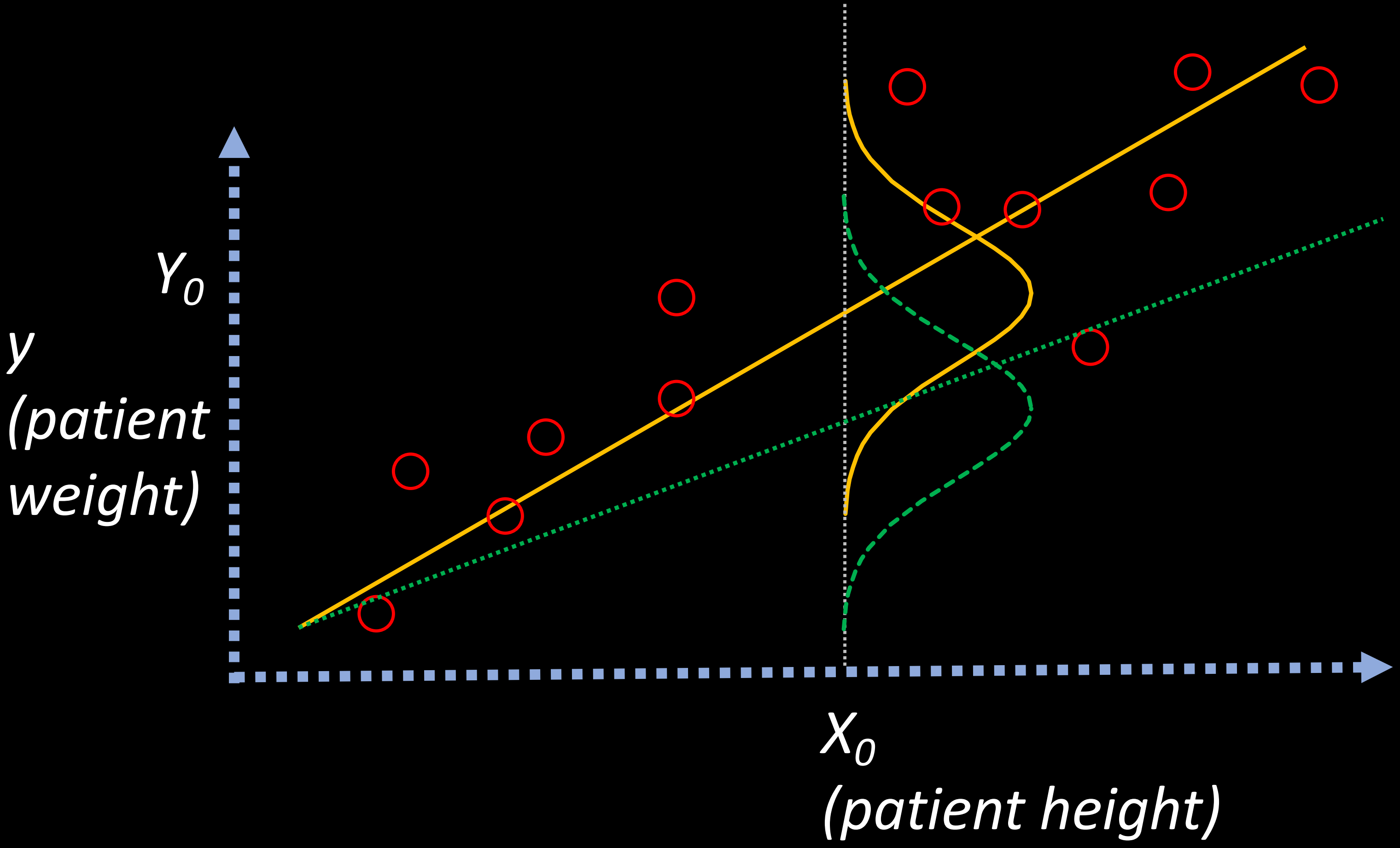
```
# Logistic Regression: Predicting presence of heart disease

from sklearn.linear_model import LogisticRegression
# Sample data: patient cholesterol levels and heart disease
presence (0: No, 1: Yes)

cholesterol = np.array([[150], [200], [250], [300], [350]])
heart_disease = np.array([0, 0, 1, 1, 1])

log_model = LogisticRegression()

log_model.fit(cholesterol, heart_disease)
```

Code Block 1

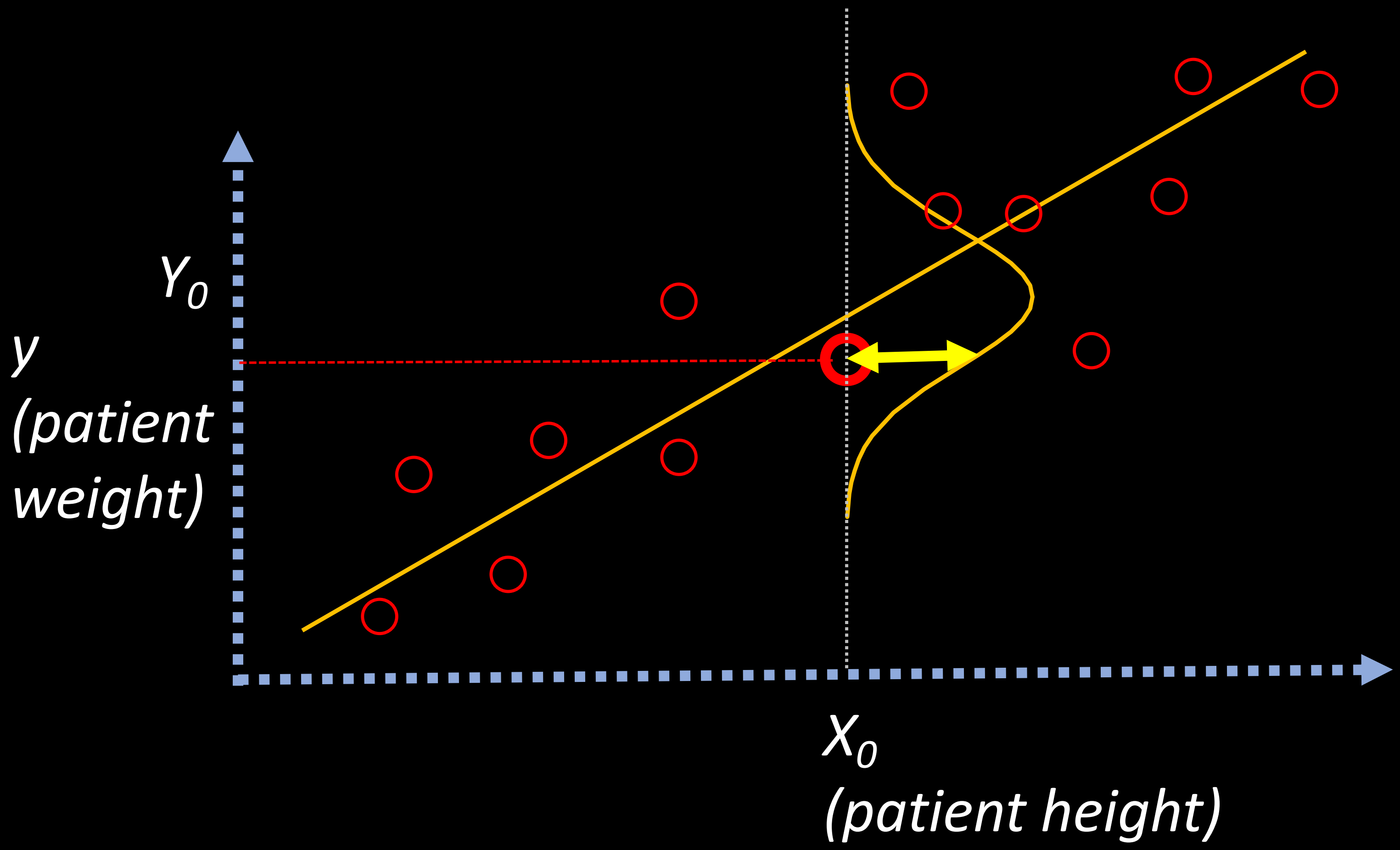
```
# Predict heart disease for a new patient with cholesterol level
275
new_cholesterol = np.array([[275]])
predicted_prob =
log_model.predict_proba(new_cholesterol)[0][1]

print(f"Probability of heart disease for 275 cholesterol:
{predicted_prob:.2f}")

# Output: ```Probability of heart disease for 275 cholesterol: 0.73
```
```

**Code Block 1**
```

Linear Regression: Predicts continuous outcomes

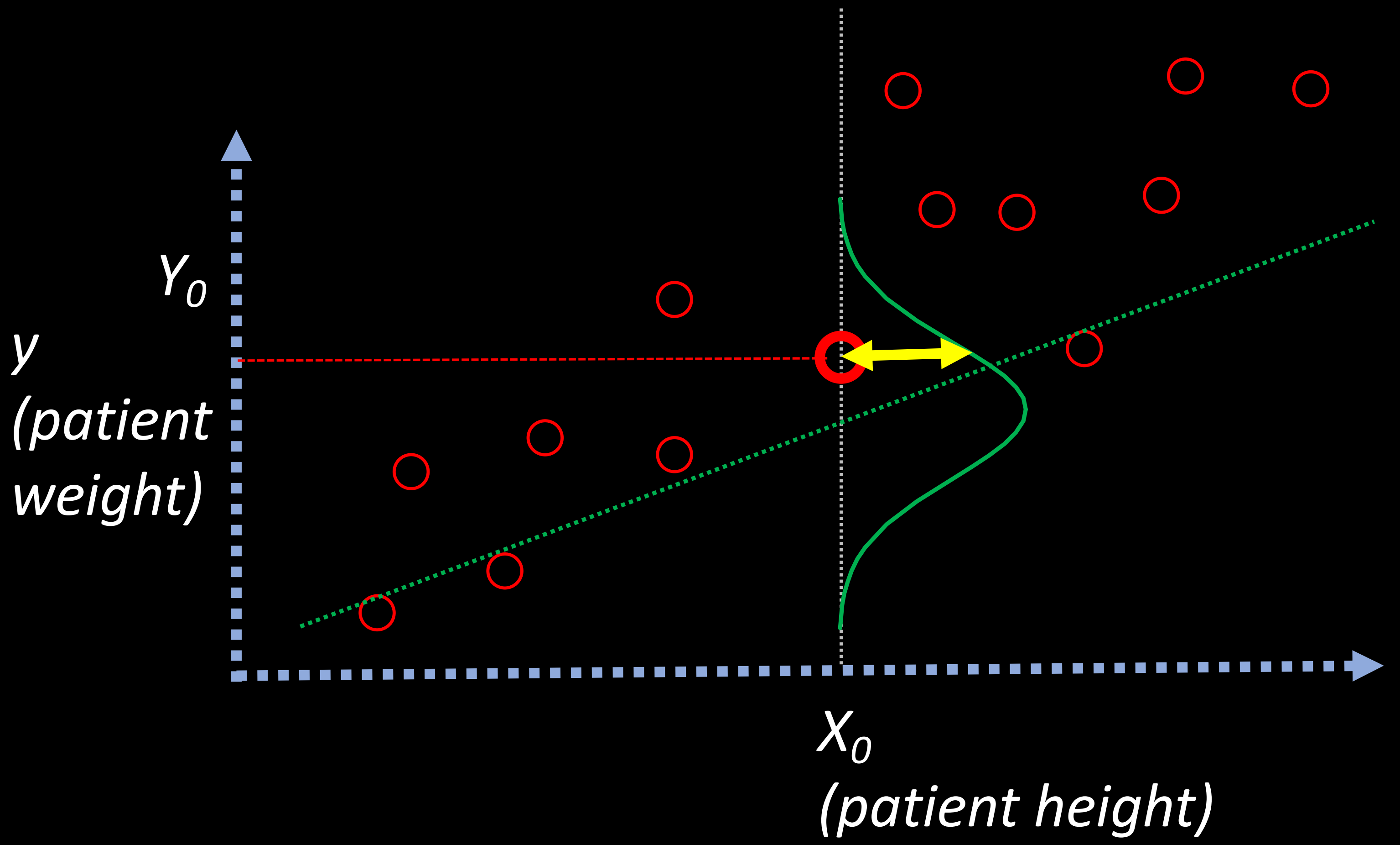




Which model
Is more probable?



Picking the more probable model



Picking the
more probable model

Maximum Likelihood Estimation (MLE)

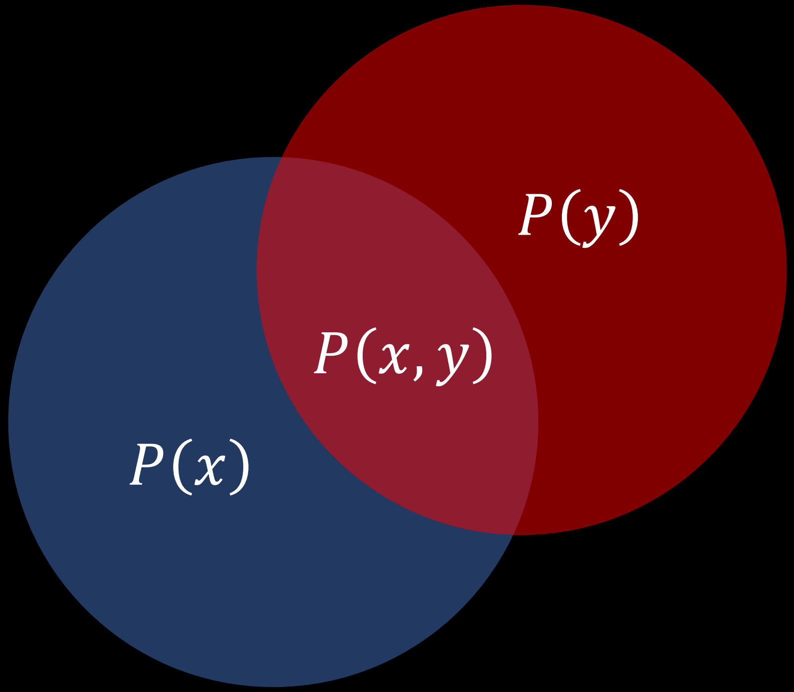
$$\theta_{ML} = argmax_{\theta} P(y|X;\theta) = argmin_{\theta}(-P(y|X;\theta))$$

MLE is a method for estimating the parameters of a statistical model by maximizing the likelihood function.

Key Concept: It finds the parameter values that make the observed data most probable.

Review probability

P(x)	Probability
$P(x, y)$	Joint Probability
$P(y\|x) = P(x, y) \;/\; P(x)$	Conditional Probability



Probability:

The likelihood of an event occurring. Example: The probability of a patient responding to a specific drug treatment.

Joint Probability:

The probability of two or more events occurring together. Example: The probability of a patient having both diabetes and hypertension.

Conditional Probability:

The probability of an event occurring, given that another event has already occurred. Example: The probability of a patient having a heart attack, given that they have high blood pressure.

Maximum Likelihood Estimation (MLE)

$\theta_{MLE} = argmax_{\theta} P(y|X;\theta))$

$P(y\|X)$	Probability of seeing y given X (true probabilistic distribution)
$P(y\|X;\theta)$	Probability of seeing y predicted X from model with parameters θ
$argmax_{\theta}$	Find the parameters θ to maximize the given probability $P(y\|X,\theta)$

Probability

$$P(A|B) \quad := \quad L(B|A)$$

Likelihood

Chance of seeing data A given fixed condition B

Chance of condition B given seeing data A

Maximum Likelihood Estimation (MLE)

$\theta_{MLE} = argmax_\theta P(y|X;\theta))$

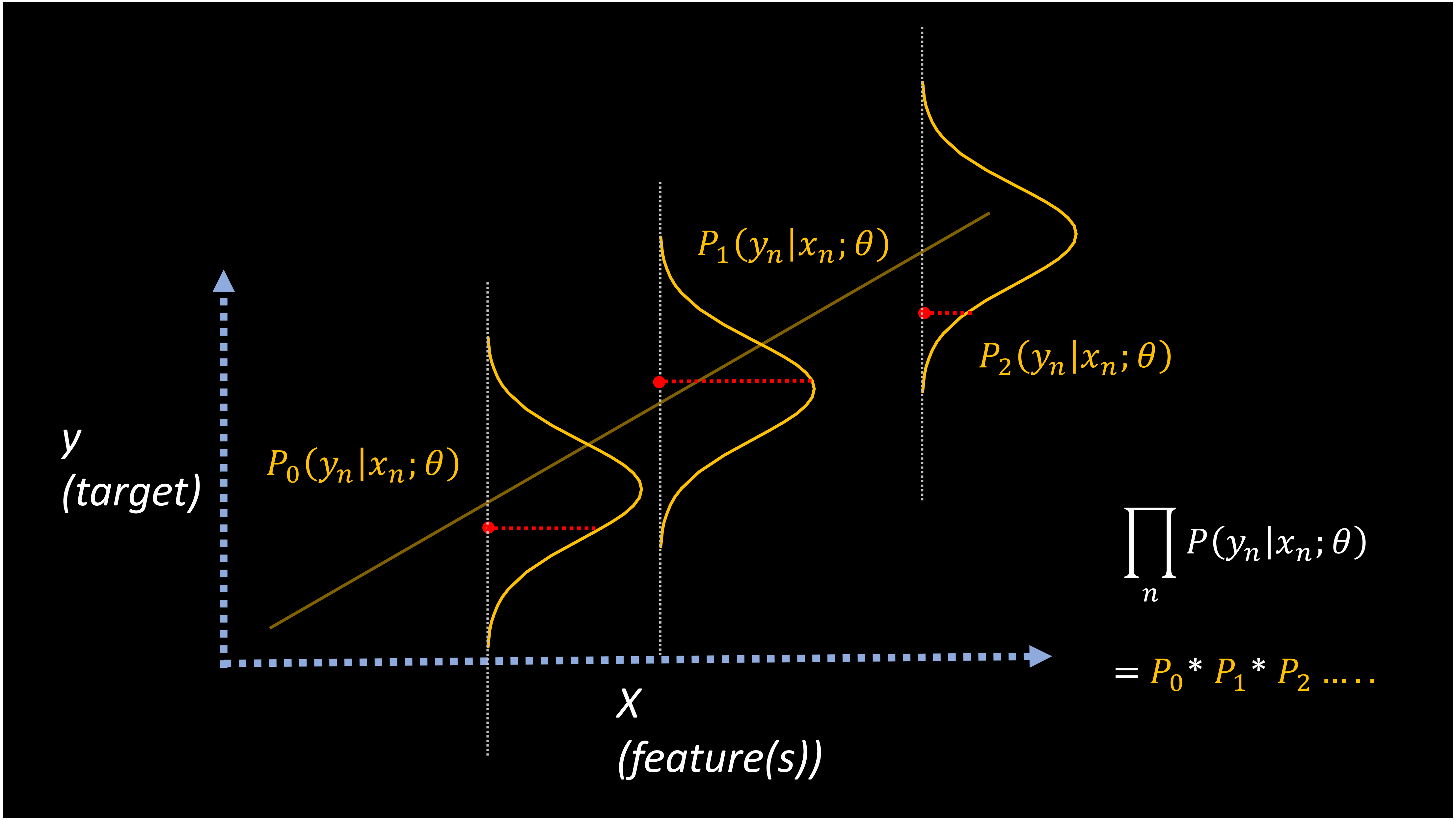
$P(y\|X)$	(true) probability of seeing stroke given blood pressure
$P(y\|X;\theta)$	Probability of seeing stroke given blood pressure from model with θ
$argmax_\theta$	Find the parameters θ to maximize the given probability $P(y\|X,\theta)$

MLE in Linear Regression

Assumption: In linear regression, we assume errors are normally distributed around the true regression line.

Likelihood Function: Measures the probability of observing our data given a particular set of regression parameters (slope and intercept).

MLE Process: Finds the regression line that maximizes this likelihood function, making our observed data most probable, equivalent to minimizing the sum of squared errors (least squares method).



$$\prod_n P(y_n|x_n;\theta)$$

$$= P_0 * P_1 * P_2 \dots\dots$$

$$P(A, B) = P(A) * P(B)$$

If A and B are independent
(often assumed that data are **independent and identically distributed (IID))**

For $y = \{y_1, y_2 \ldots y_i\}$, $X = \{x_1, x_2 \ldots x_n\}$

Features and targets for all the observations

$P(y|X;\theta)$ = Likelihood of all the observations

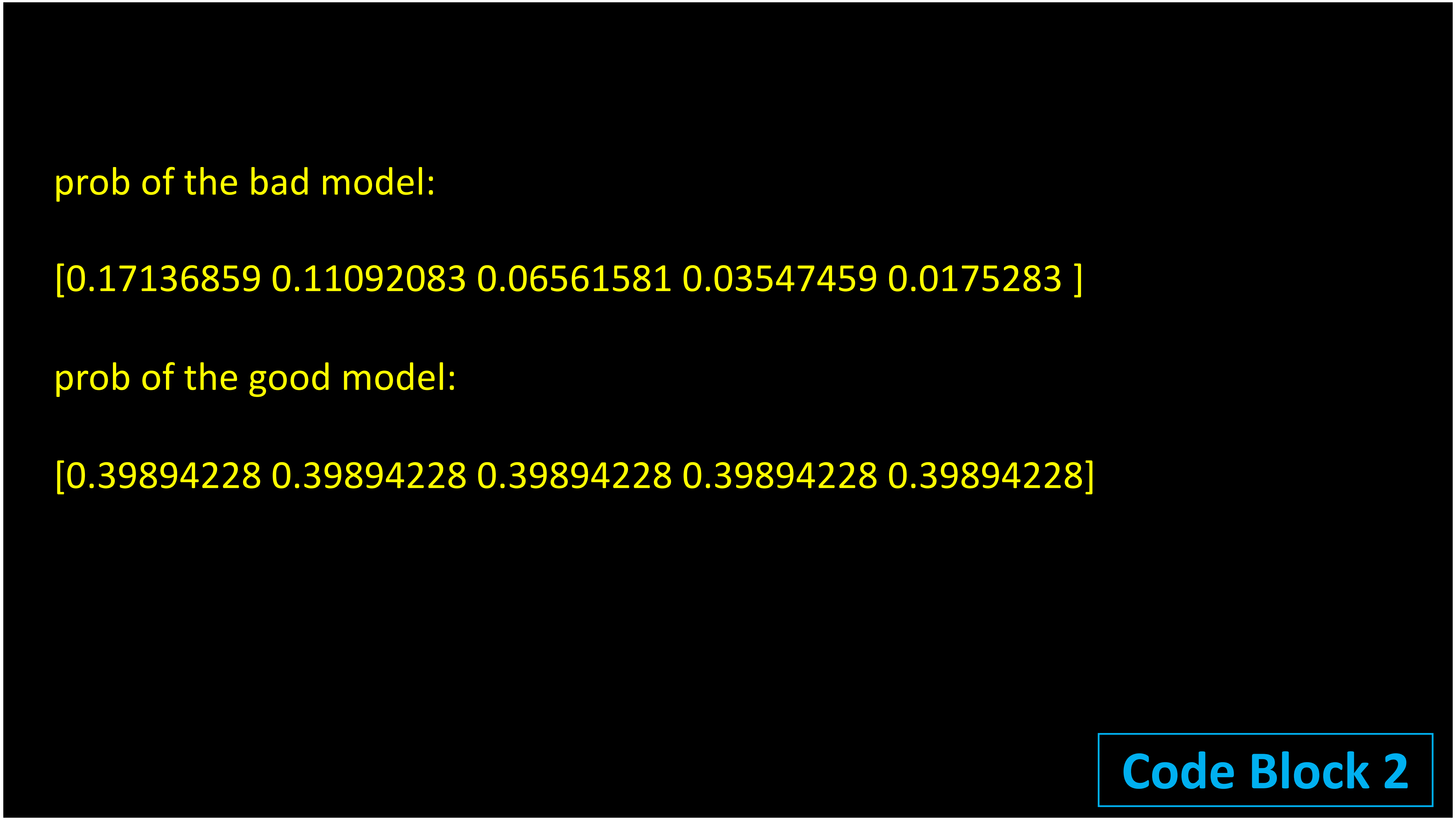
$$P(y_1|x_1;\theta) * P(y_2|x_2;\theta) \ \ldots * P(y_n|x_n;\theta) = \prod_n P(y_n|x_n;\theta)$$

```python
# Data points
weights = np.array([[60], [70], [80], [90], [100]])
dosages = np.array([100, 115, 130, 145, 160])

# Create and train the good model
model_good = LinearRegression()
model_good.fit(weights, dosages)

# Create a bad model with random coefficients
model_bad = LinearRegression()
model_bad.coef_ = np.array([1.53])
model_bad.intercept_ = np.array([9.5])
```

Code Block 2

```python
def calculate_probabilities(predictions, actual, std_dev):
        return norm.pdf(actual - predictions, loc=0, scale=std_dev)

# Predict dosages
predictions_good = model_good.predict(weights)
predictions_bad = model_bad.predict(weights)


probs_good = calculate_probabilities(predictions_good, dosages, 1)
probs_bad = calculate_probabilities(predictions_bad, dosages, 1)

print(probs_good) print(probs_bad)
```

Code Block 2



```
prob of the bad model:

[0.17136859 0.11092083 0.06561581 0.03547459 0.0175283 ]

prob of the good model:

[0.39894228 0.39894228 0.39894228 0.39894228 0.39894228]
```

Code Block 2

Joint Probability

$P(y_1|x_1;\theta) * P(y_2|x_2;\theta) \ \ldots * P(y_n|x_n;\theta) = \prod_n P(y_n|x_n;\theta)$

$\prod_n P(y_n|x_n;\theta)$ $(0.1 \ * 0.1 \ * 0.1 \ * 0.1 \ldots)$ 0.000000001

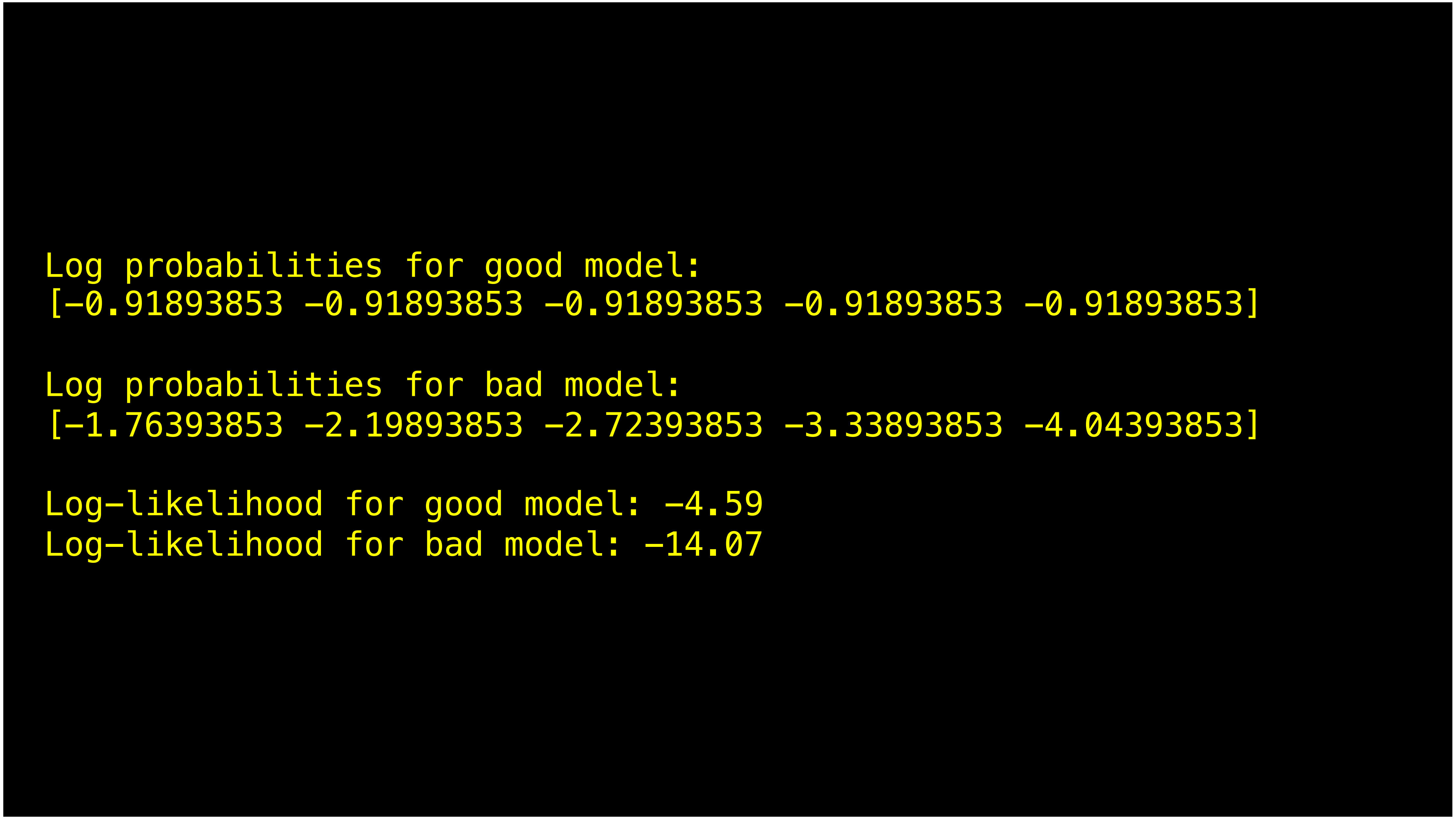
VS

$\sum_i log P(y_n|x_n;\theta)$ (-1 -1 -1 -1 -1.....) -9

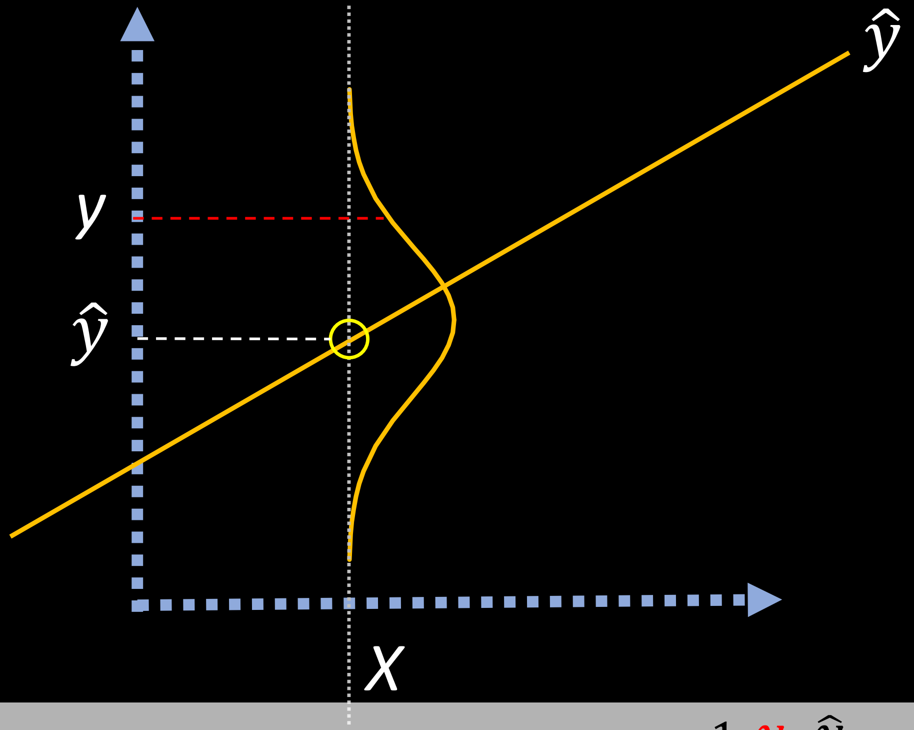
Log Probability

```python
log_probs_good = np.log(probs_good)
log_probs_bad = np.log(probs_bad)
print("\nLog probabilities for good model:")
print(log_probs_good)print(log_probs_bad)


# Calculate sum of log probabilities (log-likelihood)
log_likelihood_good = np.sum(log_probs_good)
log_likelihood_bad = np.sum(log_probs_bad)
print(f"\nLog-likelihood for good model: {log_likelihood_good:.2f}")
print(f"Log-likelihood for bad model: {log_likelihood_bad:.2f}")
```

Code Block 3



```
Log probabilities for good model:
[-0.91893853 -0.91893853 -0.91893853 -0.91893853 -0.91893853]

Log probabilities for bad model:
[-1.76393853 -2.19893853 -2.72393853 -3.33893853 -4.04393853]

Log-likelihood for good model: -4.59
Log-likelihood for bad model: -14.07
```



Picking the most probable parameters
(w_i and b)
for the linear regression mode, or

the model which has the most
LIKELIHOOD based on the present
observation.

aka Maximum Likelihood Estimation (MLE)

$$P(y|x) = N(\hat{y}, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y-\hat{y}}{\sigma}\right)^2} \qquad \sigma\ (variance)$$

$$\hat{y} \ = wx + b$$

or = $w_1x_1 + w_2x_2 + \cdots + w_nx_n + b = \Sigma_i w_i x_i$+ b for multi-variables

$$\sum_n logP(y_n|x_n;\theta)$$

$$=\mathrm{n}\sum \log(\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(y_n-wx_n-b)2}{2\sigma^2}\right))$$ Linear Regression picked from Normal Distribution

$$= \text{-n} * log\frac{1}{\sqrt{2\pi\sigma^2}} - \sum_i \frac{(y_n-wx_n-b)2}{2\sigma^2}$$

Assuming constant variance

$$argmax(\sum_i logP(y_n|x_n;\theta)) = argmin(\sum_n (y_n - wx_n - b)^2)$$

$$= argmin(\sum_n (y_n - \hat{y})^2)$$

Mean Square Error (MSE)

Linear Regression is optimized by minimize mean square error (MSE)

Linear Regression and Maximum Likelihood Estimation (MLE)

Keys:

- Likelihood: Probability of observing the data given the model parameters.
- Log-Likelihood: Sum of log probabilities, used for numerical stability and easier optimization.
- Assumptions: Errors are normally distributed around the true regression line.

MLE Process:

-Define the likelihood function based on the probability distribution of errors.

-Find the parameters that maximize the log-likelihood.

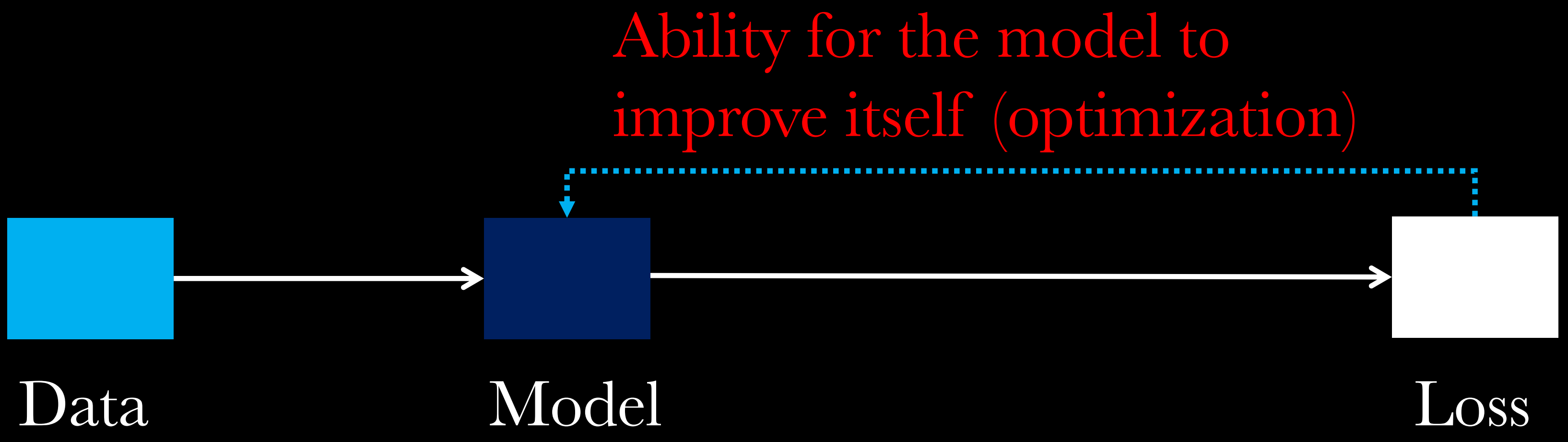
Advantages of MLE:

- Provides a principled way to estimate model parameters.
- Allows for comparison between different models using log-likelihoods

Regression by just called a library (Scipy)

```
from sklearn.linear_model import Regression

Regr = Regression()

Regr.fit(x_train, y_train)
```

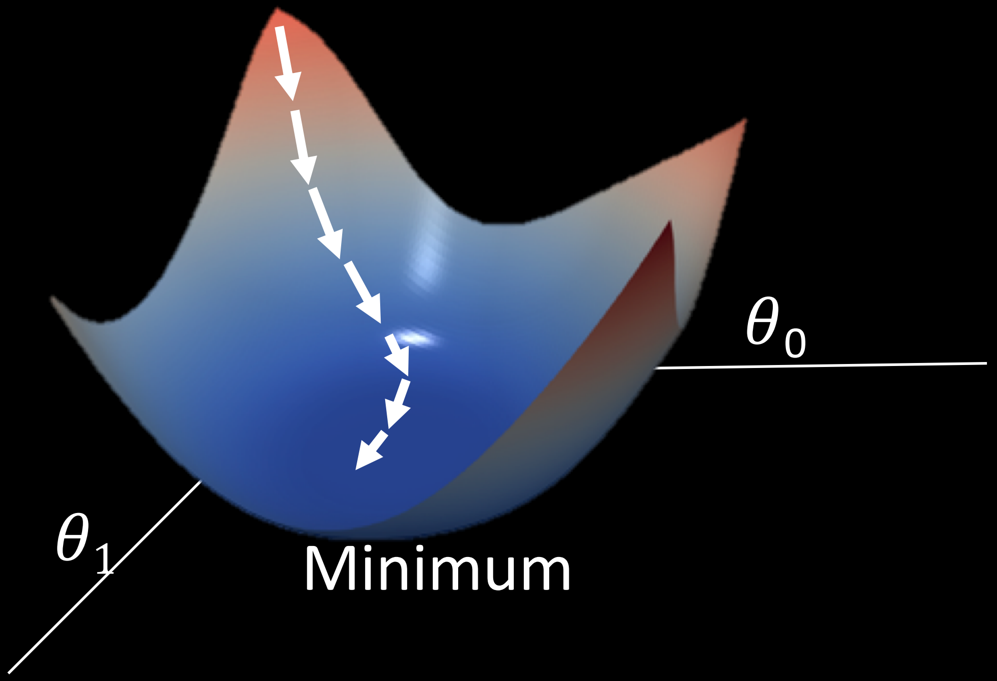
Behind the Scene Regr.fit(x_train, y_train)



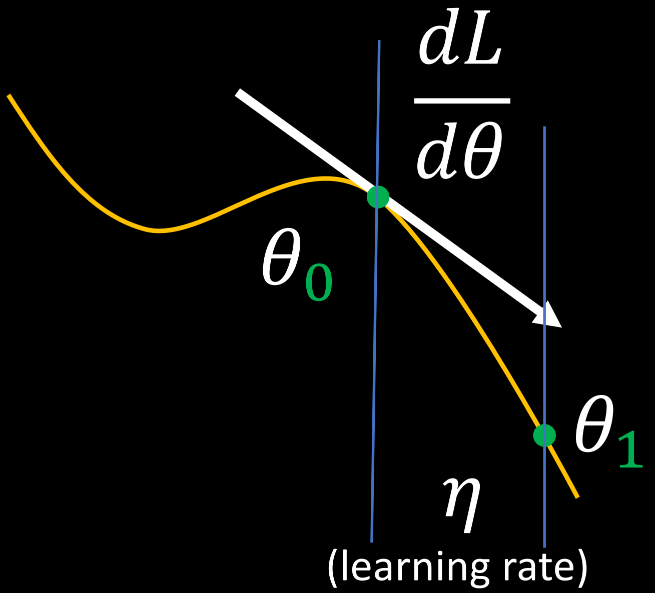
Gradient Descent

Moves in the direction of steepest descent (negative gradient) to minimize the objective function.

Steps:
1. Start with initial parameter values
2. Calculate the gradient of the objective function
3. Update parameters in the opposite direction of the gradient
4. Repeat steps 2-3 until convergence



Gradient Descent



$$\theta = argmin\left(\sum_n (y_n - \hat{y})^2\right) =$$

$$argmin\left(\sum_n (y_n - wx_n - b)^2\right)$$

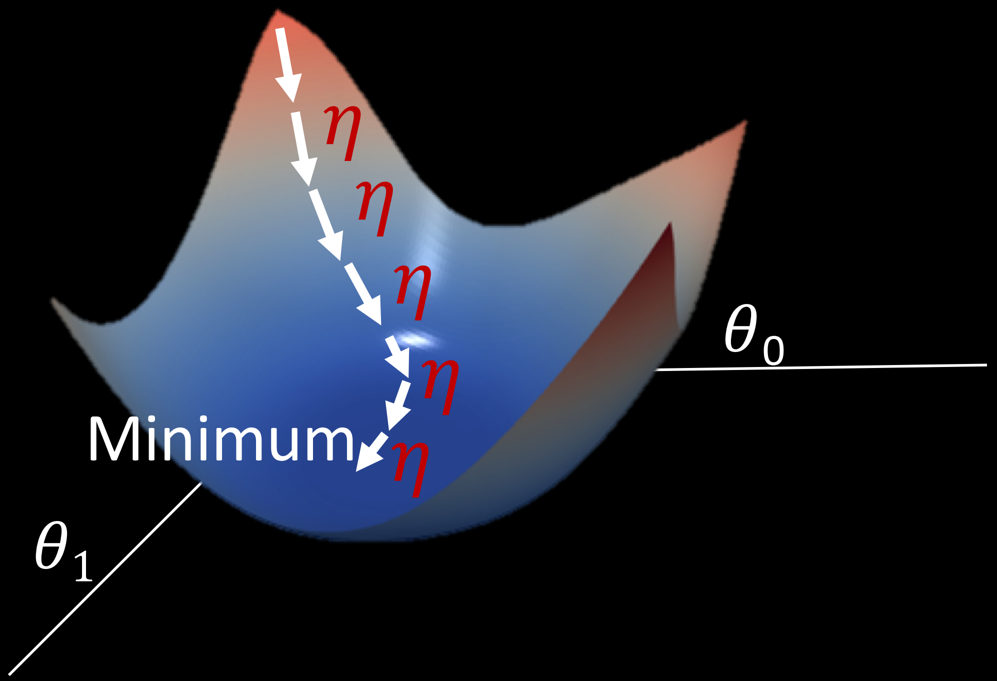
$$L = \sum_n (y_n - wx_n - b)^2$$

Loss function

$$\frac{dL}{dw} = \frac{}{dw}\sum_n (y_n - wx_n - b)^2$$

$$\frac{dL}{db} = \frac{}{db}\sum_n (y_n - wx_n - b)^2$$

Gradient of L to parameters



$$\frac{dL}{dw} = \frac{}{dw}\sum_n (y_n - wx_n - b)^2 \quad = -\sum_n 2x_n(y_n - wx_n - b)$$

$$\frac{dL}{db} = \frac{}{db}\sum_n (y_n - wx_n - b)^2 = -\sum_n 2(y_n - wx_n - b)$$

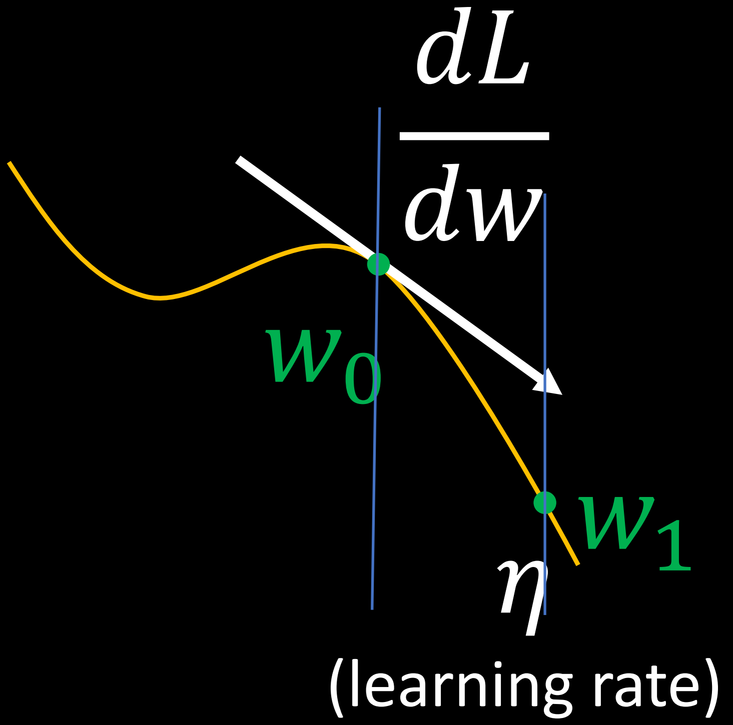
Remember chain rules?

$$F'(x) = f'(g(x))\, g'(x)$$

```python
import numpy as np

# Sample data
X = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 6, 8, 10])

# Two parameter sets
w1, b1 = 1.5, 1.0
w2, b2 = 1.5, 0.5

# Function to calculate predictions
def predict(X, w, b):
    return w * X + b

# Function to calculate Mean Squared Error
(MSE) loss
def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)


# Function to calculate gradient of loss with
respect to b
def gradient_b(y_true, y_pred):
    return -2 * np.mean(y_true - y_pred)
```

```python
# Calculate predictions, loss, and gradient
for both parameter sets
y_pred1 = predict(X, w1, b1)
loss1 = mse_loss(y, y_pred1)
grad_b1 = gradient_b(y, y_pred1)


y_pred2 = predict(X, w2, b2)
loss2 = mse_loss(y, y_pred2)
grad_b2 = gradient_b(y, y_pred2)
```

```
Parameter set 1: w = 1.5, b = 1.0
Loss: 0.7500
Gradient dL/db: -1.0000

Parameter set 2: w = 1.5, b = 0.5
Loss: 1.5000
Gradient dL/db: -2.0000
```



$$w_1 = w_0 - \frac{dL}{dw}\eta$$

$$b_1 = b_0 - \frac{dL}{db}\eta$$

$$(w_0, b_0) \longrightarrow (w_1, b_1)$$

```python
import numpy as np
X = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 6, 8, 10])

# Initial parameters
w, b = 1.5, 1.0

# Function to calculate predictions
def predict(X, w, b):
    return w * X + b

def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

def gradient_w(X, y_true, y_pred):
    return -2 * np.mean(X * (y_true -
y_pred))

def gradient_b(y_true, y_pred):
    return -2 * np.mean(y_true - y_pred)
```

```python
# Calculate initial predictions, loss, and
gradients
y_pred = predict(X, w, b)
initial_loss = mse_loss(y, y_pred)
grad_w = gradient_w(X, y, y_pred)
grad_b = gradient_b(y, y_pred)


# Perform one step of gradient descent
learning_rate = 0.01
w_new = w - learning_rate * grad_w
b_new = b - learning_rate * grad_b


# Calculate new predictions and loss
y_pred_new = predict(X, w_new, b_new)
new_loss = mse_loss(y, y_pred_new)
```

```python
print(f"Initial parameters: w = {w}, b = {b}")
print(f"Initial loss: {initial_loss:.4f}")
print(f"Gradient dL/dw: {grad_w:.4f}")
print(f"Gradient dL/db: {grad_b:.4f}")


print(f"\nAfter one gradient descent step:")
print(f"New parameters: w = {w_new:.4f}, b =
{b_new:.4f}")
print(f"New loss: {new_loss:.4f}")


print(f"\nLoss reduction: {improvement:.2f}%")
```

```
Initial parameters: w = 1.5, b = 1.0
Initial loss: 0.7500
Gradient dL/dw: -5.0000
Gradient dL/db: -1.0000

After one gradient descent step:
New parameters: w = 1.5500, b = 1.0100
New loss: 0.5206
```

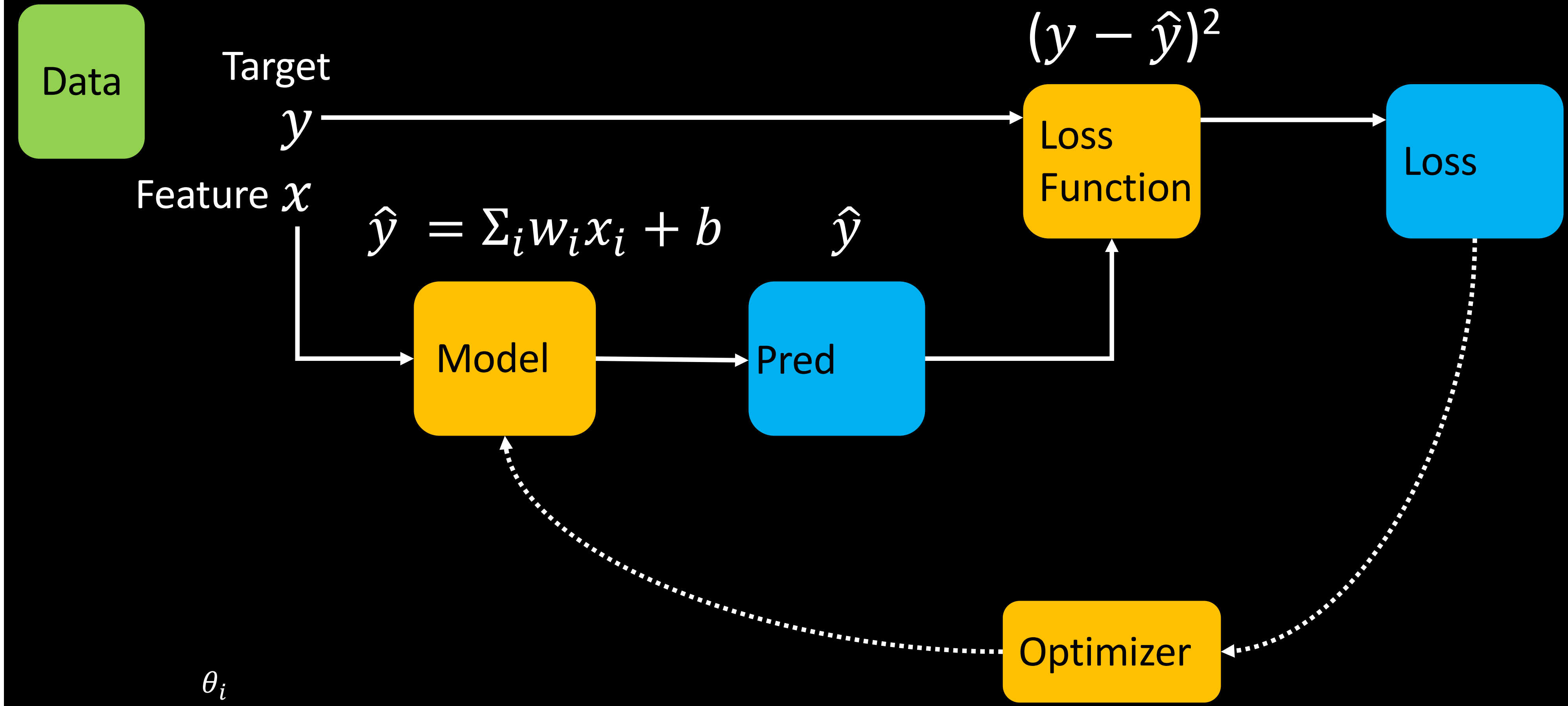
Gradient Direction:

- Gradient ∇L points towards the steepest increase in loss
- We move in the opposite direction ($-\nabla L$) to minimize loss
- For each parameter θ: $\theta_{new} = \theta_{old} - \eta * \partial L/\partial\theta$

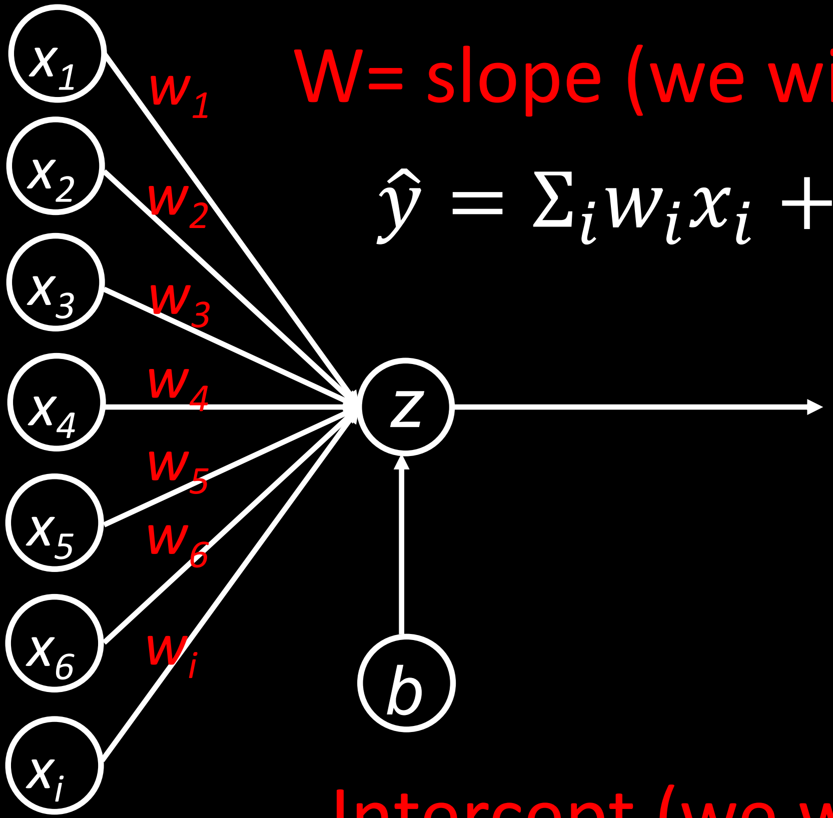
Learning Rate (η):

- Controls the step size in each iteration
- Too large: May overshoot the minimum, causing divergence
- Too small: Slow convergence, may get stuck in local minima

Flow Chart for Linear Regression



Graph of (Multi-variable)Linear Regression



W= slope (we will call it weight by some point)

$$\hat{y} = \Sigma_i w_i x_i + b$$

Outcome

Intercept (we will call it bias by some point)