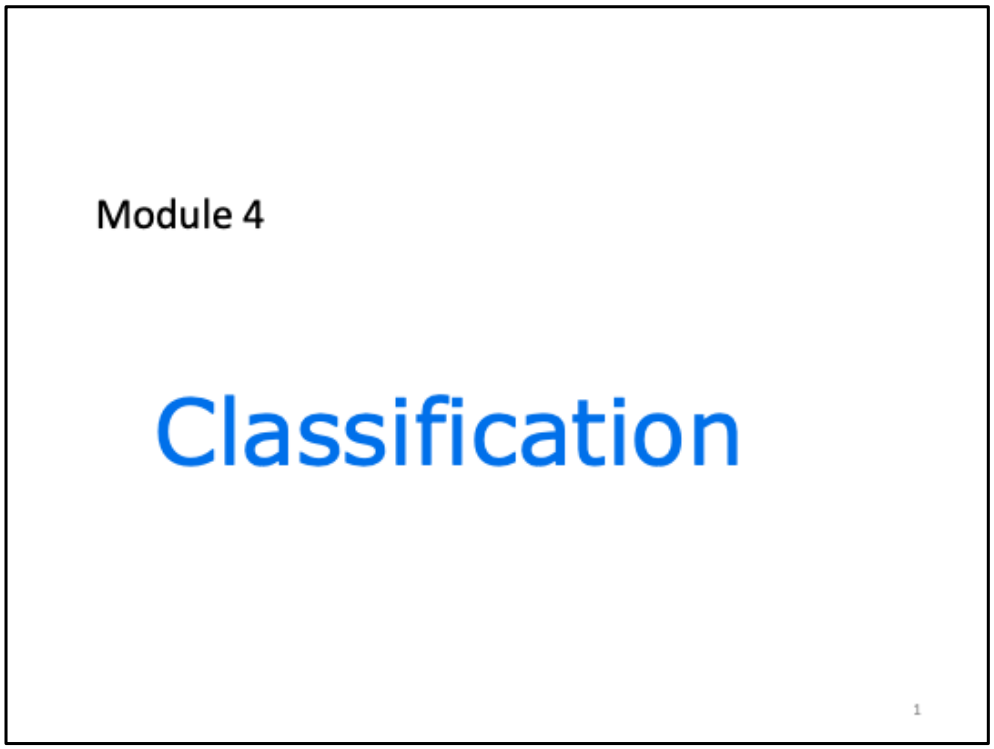


Slide Notes – Module 4: Classification

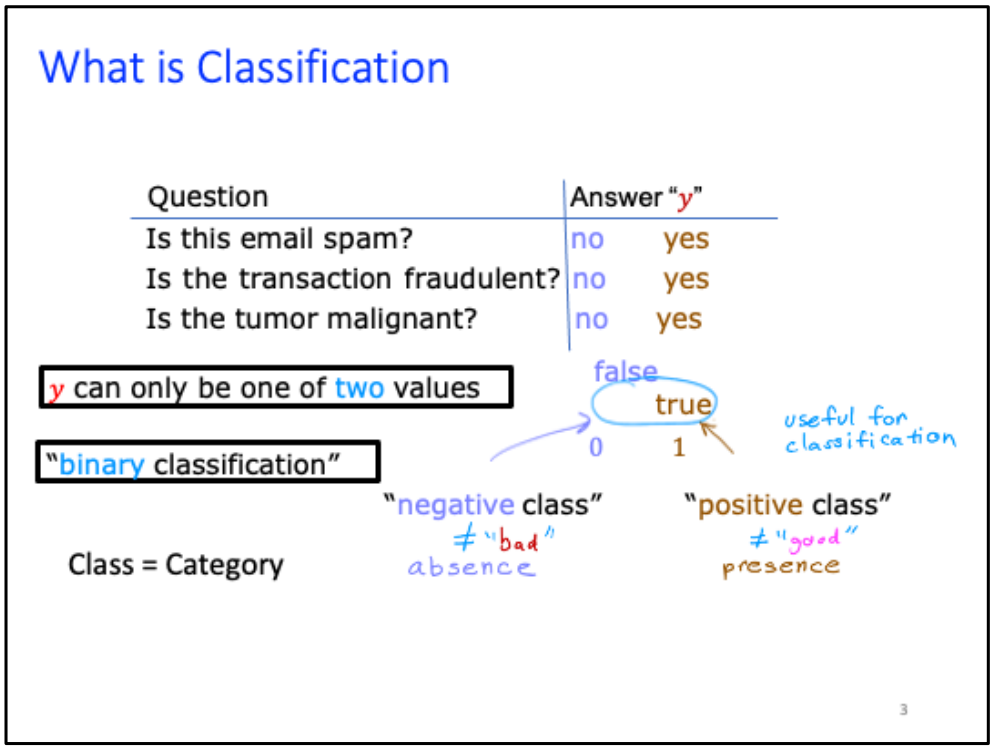
In this module, we start studying **classification**, which is one of the most important tasks in machine learning. In classification, the goal is not to predict a number, but to **assign an input to a category or class**. For example, we may want to decide whether an email is *spam or not spam*, or whether a tumor is *benign or malignant*. These outputs are usually **discrete labels**, not continuous values.

Classification is a type of **supervised learning**, which means the model learns from data that already has the correct answers (labels). During this module, we will see how classification problems are different from regression problems, what kind of outputs classification models produce, and how models learn to separate different classes. This module will build the foundation for understanding popular classification algorithms used in real-world applications.

Content

- Definition
- Logistic Regression
- Cost Function for logistic Regression
- Gradient Descent for logistic Regression
- Under fitting vs. Over fitting
- Regularization

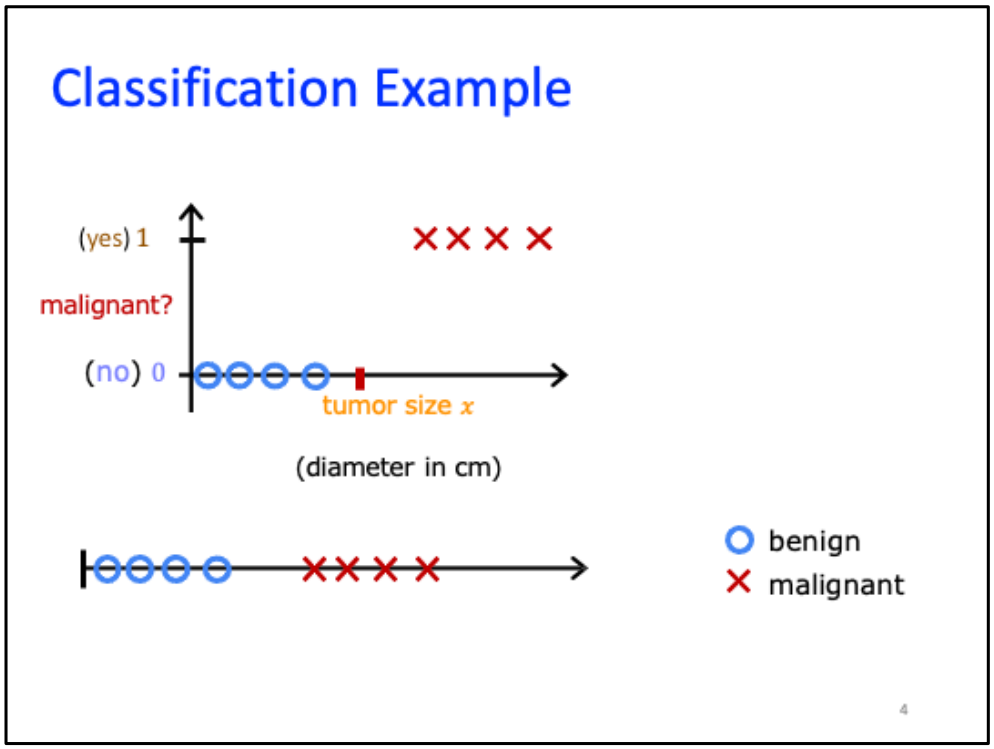
2



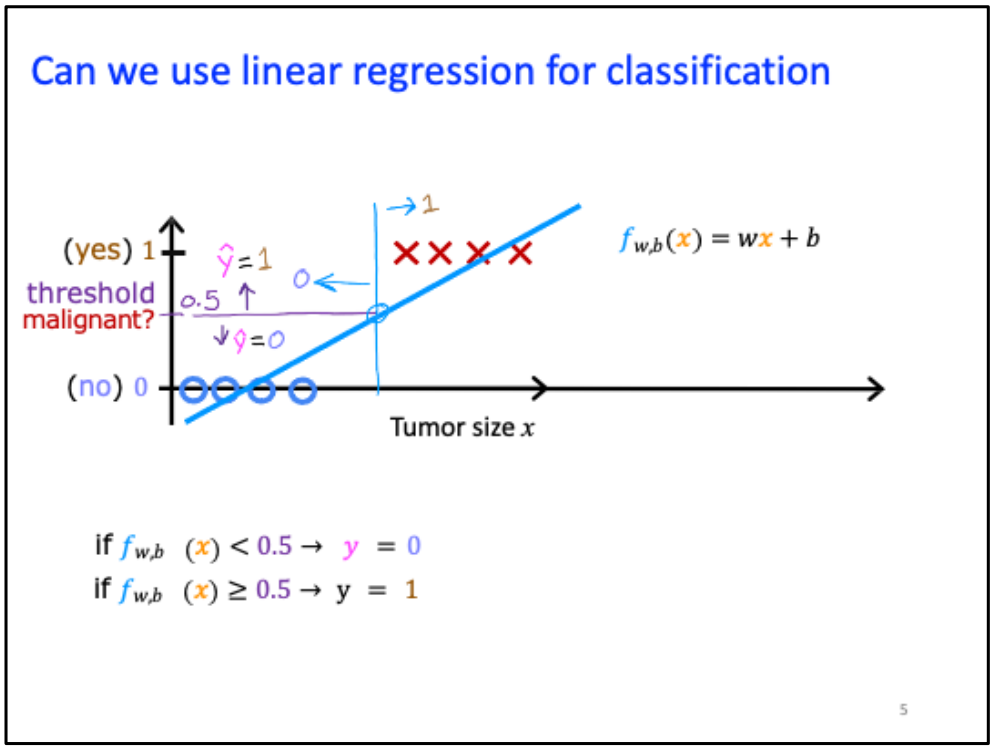
Slide Notes – What is Classification

This slide explains the basic idea of **classification**. In classification, we ask a **question with a limited number of possible answers**. For example: *Is this email spam?*, *Is the transaction fraudulent?*, or *Is the tumor malignant?* In all these cases, the answer **y** can only be **yes or no**, or equivalently **true or false**. Because the output has only two possible values, we often represent it using numbers: **0** and **1**, where 0 means *no* and 1 means *yes*.

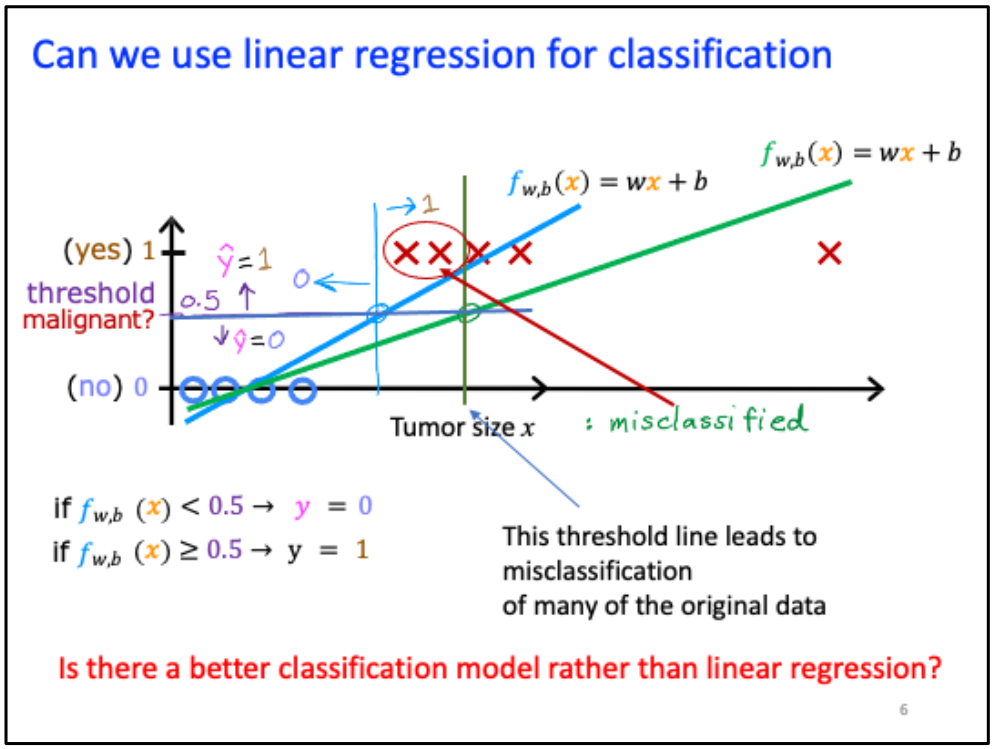
This type of problem is called **binary classification**, because there are only **two classes**. The class with value **0** is usually called the **negative class**, which means the absence of something (for example, not spam, not fraud, not cancer). The class with value **1** is called the **positive class**, which means the presence of something (spam, fraud, cancer). Here, the words *positive* and *negative* do **not** mean good or bad; they are just technical terms used in classification.



Slide Notes – Classification Example
This slide shows a simple **classification example** using medical data. The input feature **x** is the **tumor size** (diameter in centimeters), and the output **y** answers the question: *Is the tumor malignant?* The output is binary: **0 means no (benign)** and **1 means yes (malignant)**. Each point on the graph represents one patient. Blue circles correspond to benign tumors, while red crosses correspond to malignant tumors.
From the figure, we can see a clear pattern: **small tumor sizes are usually benign**, and **larger tumor sizes are usually malignant**. The goal of a classification model is to learn this pattern from the data and use it to classify new, unseen tumors. In practice, the model tries to find a **decision boundary** (a cutoff value of tumor size) that separates benign cases from malignant ones as accurately as possible.



Slide Notes – Can we use Linear Regression for Classification?
This slide explores the idea of using **linear regression** for a **classification** problem. Here, we use a linear model
(f_{w,b}(x) = wx + b)
to produce a numeric output. Since classification needs a **binary output** (0 or 1), we introduce a **threshold**, usually set to **0.5**. If the model output is **less than 0.5**, we predict **y = 0** (benign). If the output is **greater than or equal to 0.5**, we predict **y = 1** (malignant).
Although this approach may seem reasonable, it has important limitations. Linear regression is designed to predict **continuous values**, not probabilities or classes. As a result, the model can produce values **below 0 or above 1**, which do not make sense for classification. This problem motivates the need for **better models designed specifically for classification**, such as logistic regression, which we will study next.

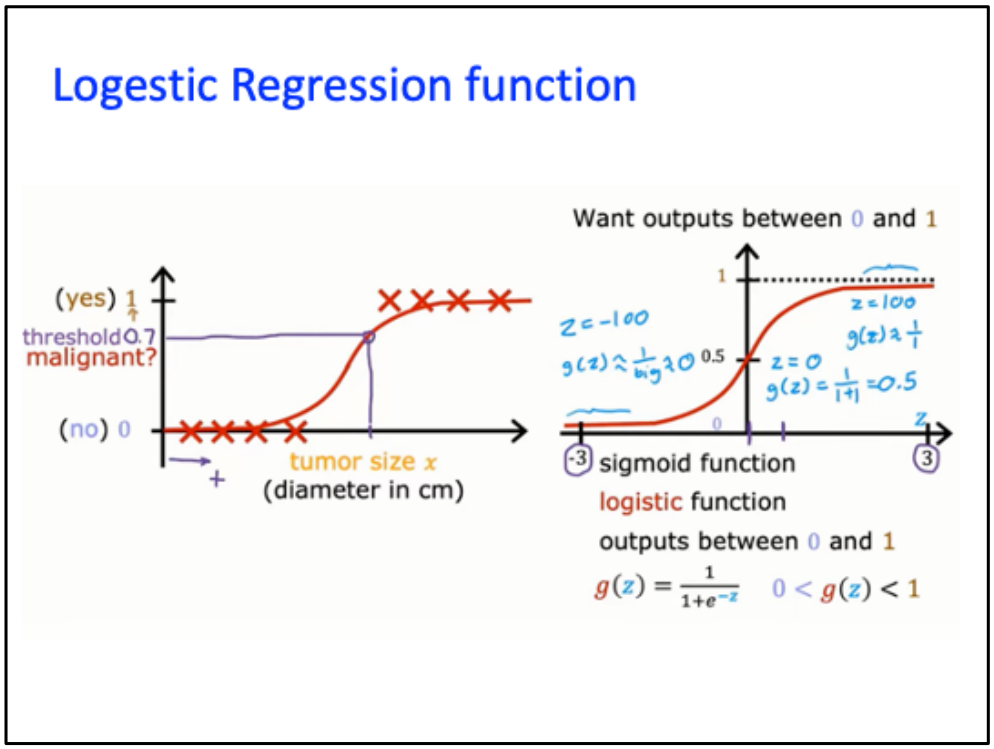


Slide Notes – Limitations of Linear Regression for Classification (Extension)

This slide extends the previous example and shows **why linear regression is not a good choice for classification**. Different straight lines (different values of (w) and (b)) can be fitted to the same data, but depending on the line, the **threshold at 0.5** may lead to **many misclassified points**. In the figure, some malignant tumors are predicted as benign, and some benign tumors are predicted as malignant. These errors happen because linear regression is very sensitive to how the line is fitted.

The key problem is that linear regression does not naturally model **class boundaries**. Its goal is to minimize squared error, not to separate classes correctly. As a result, even small changes in the data can shift the line and cause poor classification performance. This slide motivates an important question: **Is there a better model for classification than linear regression?** The answer is yes, and this leads directly to **logistic regression**, which is designed to produce outputs between 0 and 1 and to handle classification problems more reliably.

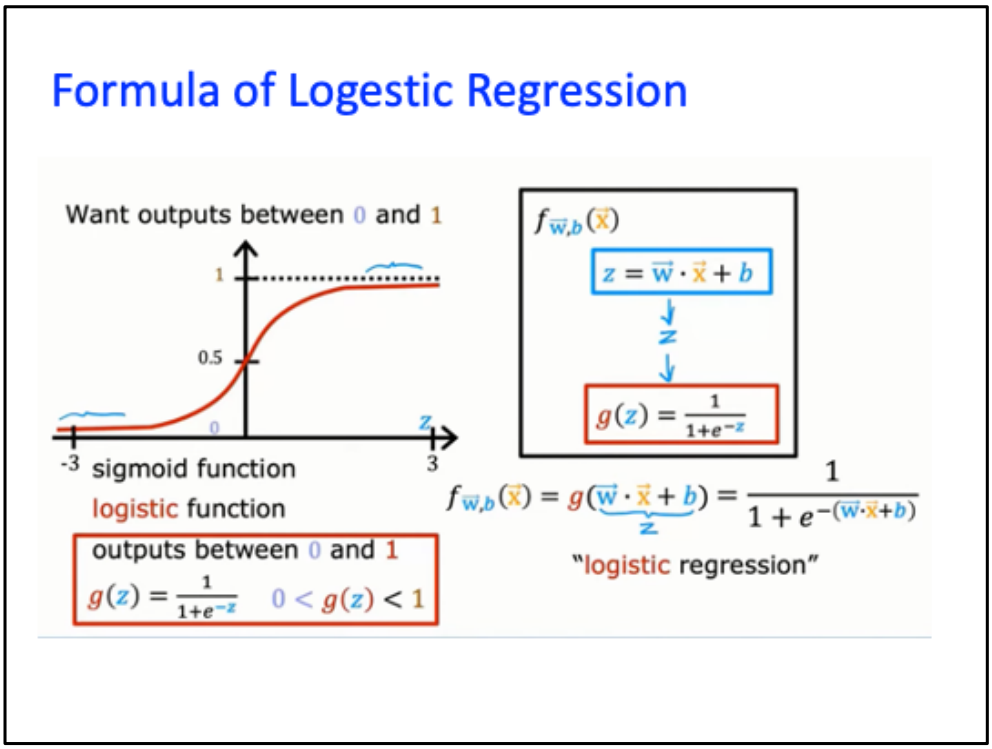
Slide Notes – Introduction to Logistic Regression
Logistic regression is a **machine learning algorithm used for classification**, especially **binary classification** problems. Unlike linear regression, which predicts continuous values, logistic regression predicts the **probability** that an input belongs to a certain class, usually class **1 (positive class)**. The output is always between **0 and 1**, which makes it well suited for answering questions like *yes or no*, *true or false*, or *spam or not spam*.



Slide Notes – Logistic Regression Function (Comparing the Two Graphs)

This slide compares **two related graphs** to explain why logistic regression works well for classification. In the **left graph**, we plot the predicted output versus **tumor size**. Instead of a straight line, the prediction follows an **S-shaped curve**. For small tumor sizes, the output is close to **0**, meaning *benign*. For large tumor sizes, the output is close to **1**, meaning *malignant*. Around the middle region, the model is uncertain, which is realistic in medical decisions.

In the **right graph**, we see the **sigmoid (logistic) function** itself. This function takes any input value (z) (which can be very large or very small) and maps it smoothly to a value **between 0 and 1**. When (z = 0), the output is **0.5**. Large positive values of (z) give outputs close to 1, and large negative values give outputs close to 0. The left graph applies this sigmoid function to the linear model output, which is why logistic regression produces meaningful probabilities and avoids the problems we saw with linear regression.



Slide Notes – Formula of Logistic Regression

This slide explains the formula of logistic regression step by step. First, we compute a **linear combination of the input features**, similar to linear regression:

z = w · x + b

Here, **w** represents the weights, **x** represents the input features, and **b** is the bias term. The value **z** can be any real number (positive or negative), but by itself it cannot be used directly for classification.

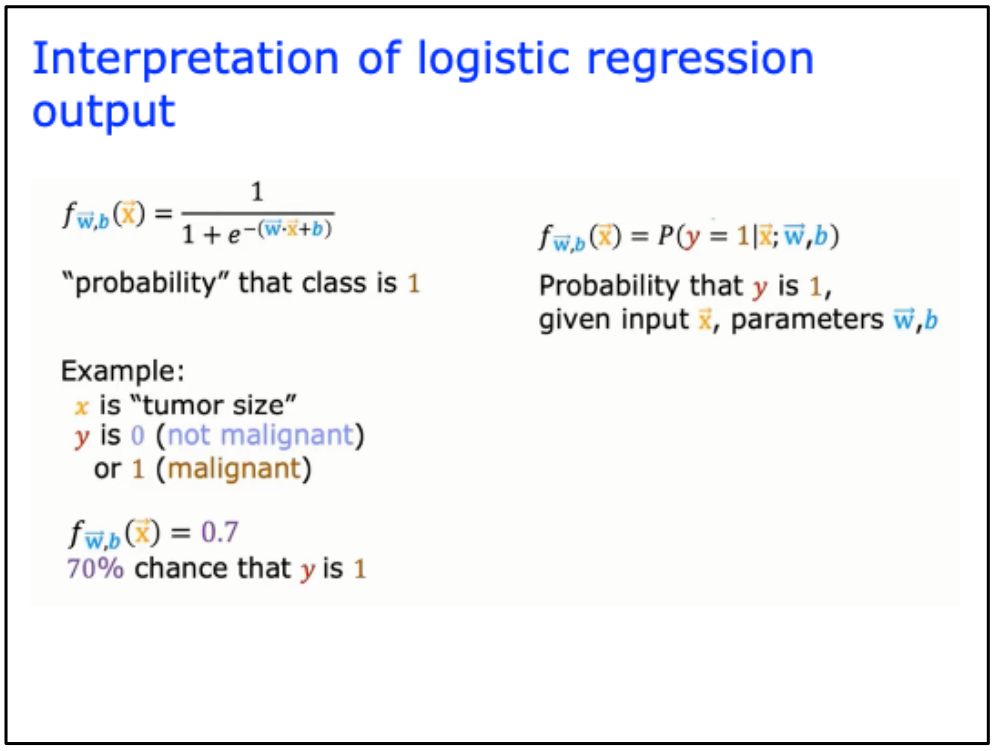
Next, we pass **z** through the **sigmoid (logistic) function**:

g(z) = 1 / (1 + e^(−z))

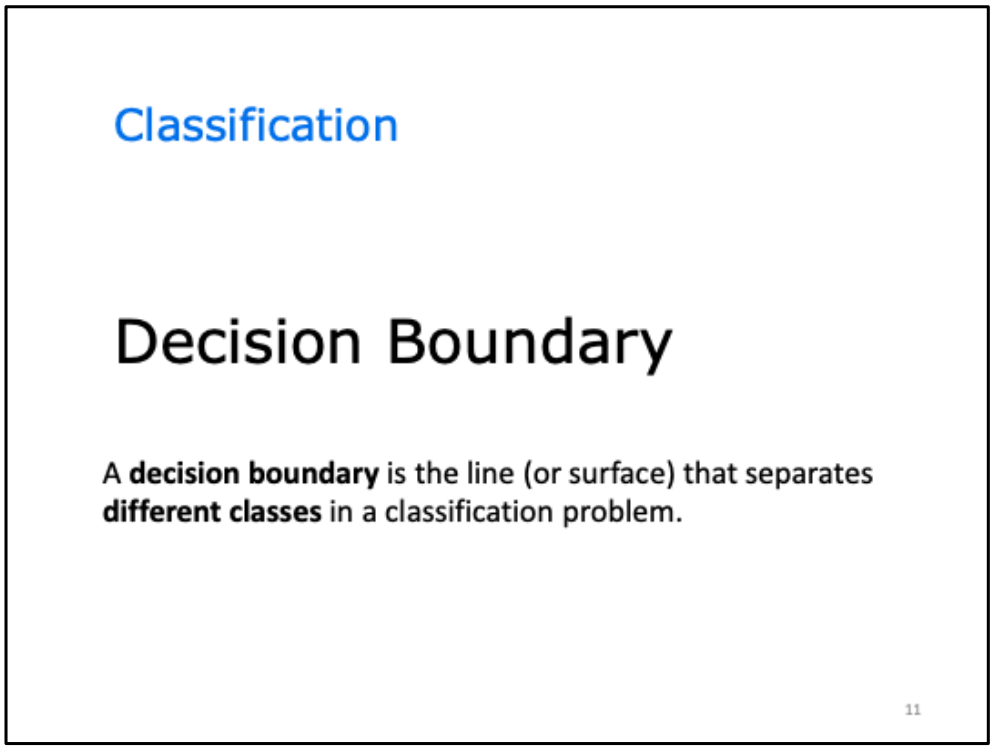
This function maps any real value of **z** into a number strictly between **0 and 1**. The final logistic regression model is:

f_w,b(x) = g(w · x + b) = 1 / (1 + e^(−(w · x + b)))

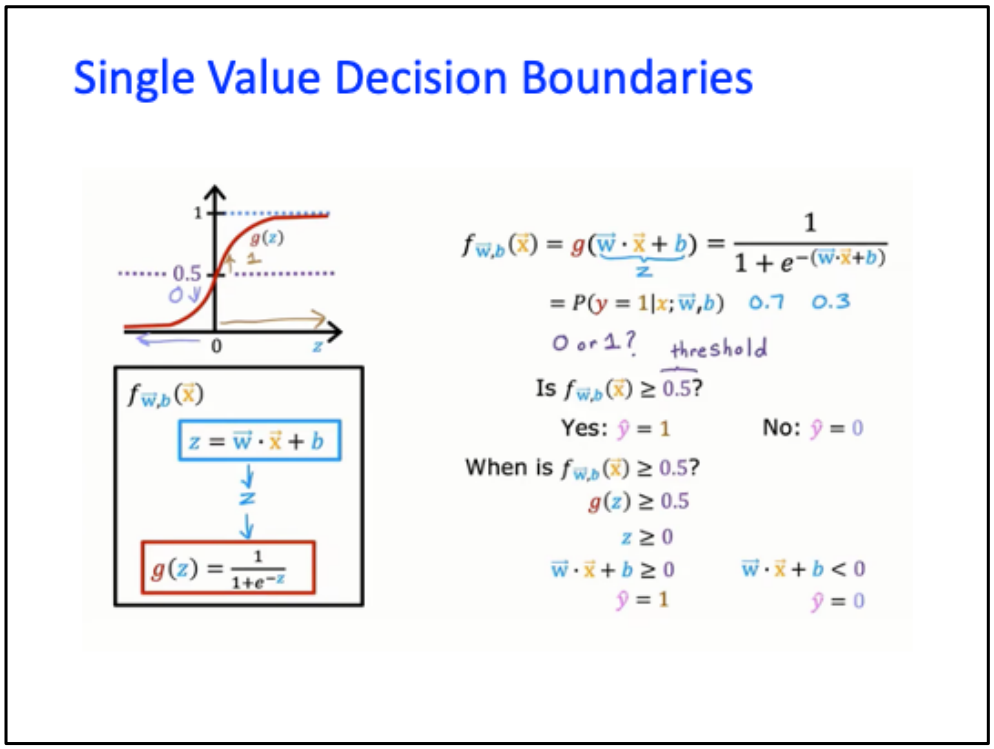
Because the output is always between 0 and 1, it can be interpreted as a **probability**, which makes logistic regression suitable for classification problems.



Slide Notes – Interpretation of Logistic Regression Output
This slide explains how to **interpret the output of logistic regression**.
The logistic regression model produces a value between **0 and 1**, using the formula:
f_w,b(x) = 1 / (1 + e^(−(w · x + b)))
This output is interpreted as the **probability that the output class y equals 1**. In other words:
f_w,b(x) = P(y = 1 | x, w, b)
This means: the probability that **y is 1**, given the input **x** and the model parameters **w** and **b**.
For example, if **x** represents *tumor size* and **y = 1** means *malignant* while **y = 0** means *benign*, then a model output of:
f_w,b(x) = 0.7
means there is a **70% probability that the tumor is malignant**. The final class decision is then made by comparing this probability to a threshold (commonly 0.5).



Slide Notes – Decision Boundary (Introduction)
In classification, a **decision boundary** is what the model uses to **separate different classes**. It defines which inputs are classified as one class and which are classified as another. For example, it can separate *benign* from *malignant* tumors or *spam* from *not spam* emails. Depending on the model and the number of features, a decision boundary can be a **line**, a **curve**, or a **surface**. Understanding decision boundaries helps us see **how a classifier makes decisions** and why some models perform better than others on complex data.



Slide Notes – Single Value Decision Boundaries

This slide explains how a **decision boundary** is formed in logistic regression when we have a **single output value**. Logistic regression first computes a linear value:

z = w · x + b

Then this value is passed through the sigmoid function:

g(z) = 1 / (1 + e^(−z))

The output f_w,b(x) = g(z) represents the **probability that y = 1**.

To make a final classification decision, we use a **threshold**, usually **0.5**.

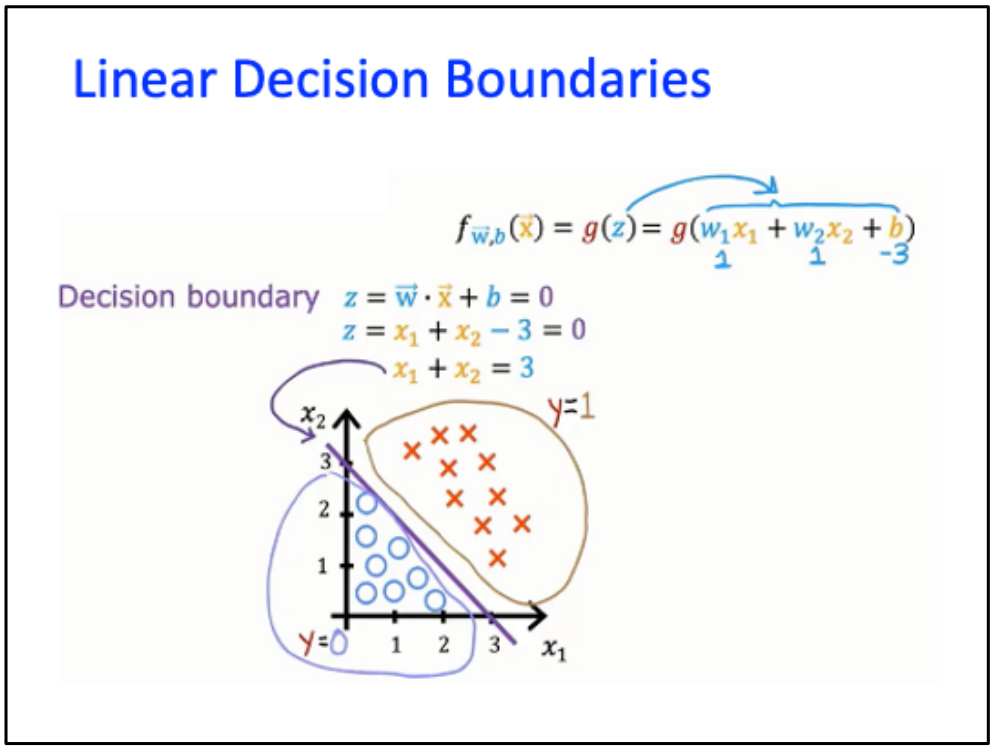
•If f_w,b(x) ≥ 0.5, we predict y = 1

•If f_w,b(x) < 0.5, we predict y = 0

Because g(z) ≥ 0.5 happens exactly when z ≥ 0, the decision boundary occurs at:

w · x + b = 0

This equation defines the **decision boundary**. On one side of the boundary the model predicts class 1, and on the other side it predicts class 0.



Slide Notes – Linear Decision Boundaries

This slide shows how logistic regression creates a **linear decision boundary** when we have **two input features**, x1 and x2. The model first computes:

z = w . x + b

z = w1 . x1 + w2 . x2 + b

The decision boundary is defined by the condition where the model is exactly unsure between the two classes, which happens when:

z = 0

The sigmoid function has an important property:

•When z = 0

•g(0) = 1 / (1 + e^0) = 1 / 2 = 0.5

So, when **z = 0**, the model predicts a probability of **0.5**, which represents **maximum uncertainty**.

That is why the **decision boundary** is defined by:

z = w . x + b = 0

So the decision boundary equation becomes:

w1 . x1 + w2 . x2 + b = 0

In the example shown on the slide, the equation is:

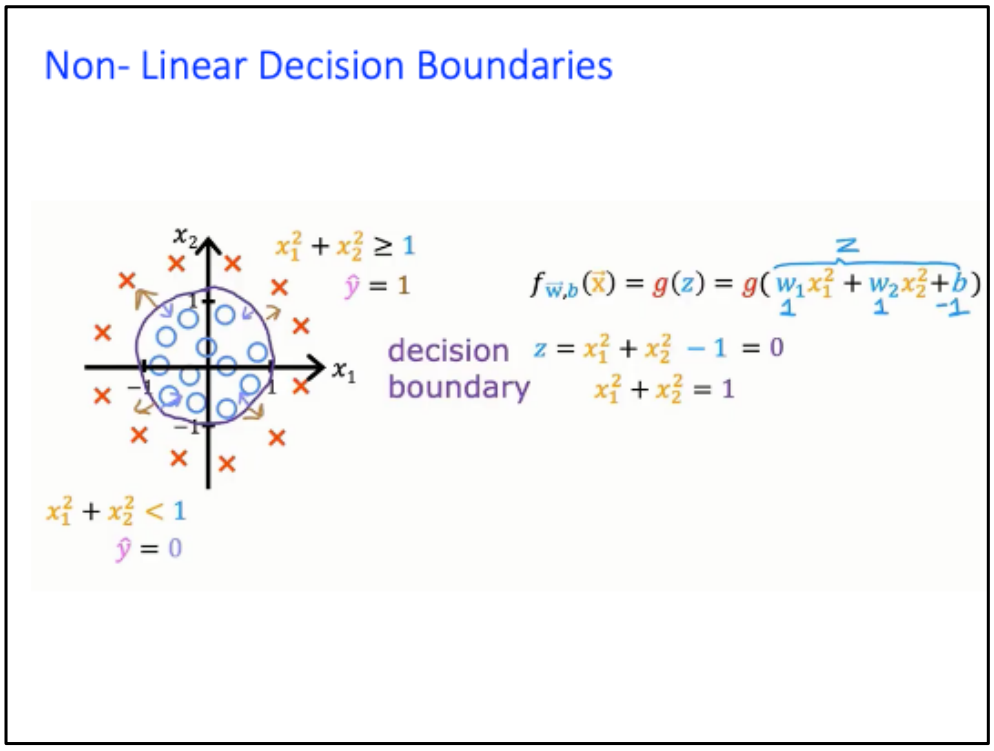
x1 + x2 − 3 = 0

or equivalently

$x1 + x2 = 3$

This equation represents a **straight line** in the (x1, x2) plane. Points on one side of the line are classified as **y = 1**, and points on the other side are classified as **y = 0**.

Because the boundary is a straight line, it is called a **linear decision boundary**.



Slide Notes – Non-Linear Decision Boundaries

This slide shows that logistic regression can also create **non-linear decision boundaries** when we use **non-linear features**. Instead of using x1 and x2 directly, we use transformed features such as $x1^2$ and $x2^2$. The model computes:

$z = w1 . x1^2 + w2 . x2^2 + b$

Then the logistic function is applied:

f_w,b(x) = g(z)

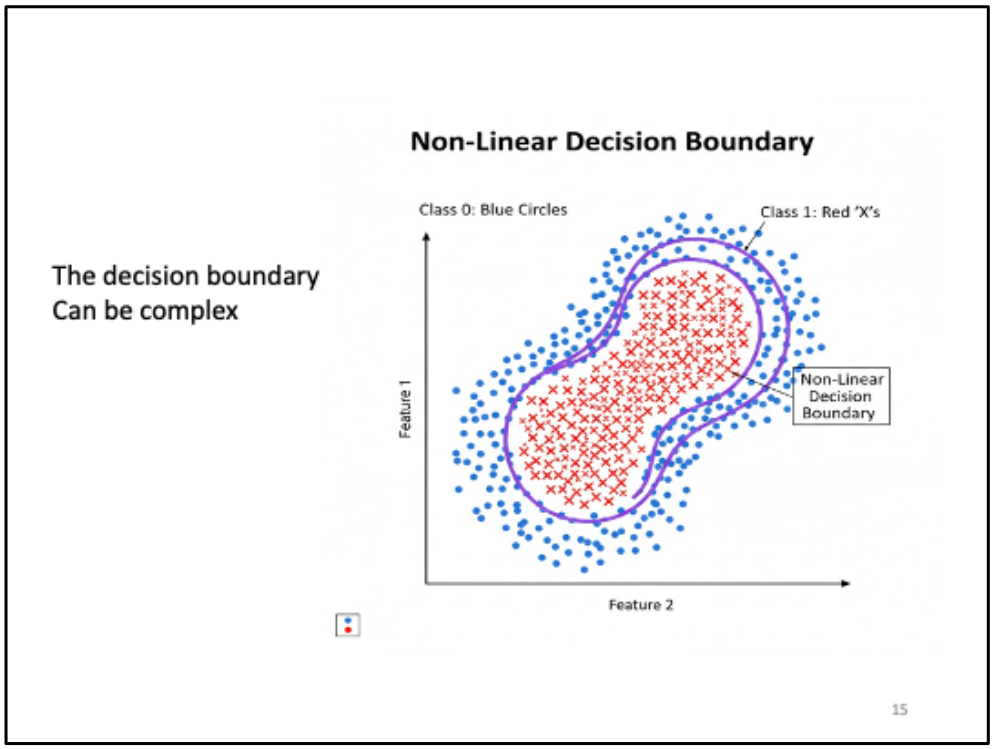
The decision boundary is again defined by z = 0. In this example:

$x1^2 + x2^2 - 1 = 0$

or equivalently

$x1^2 + x2^2 = 1$

This equation represents a **circle**. Points **inside the circle** satisfy $x1^2 + x2^2 < 1$ and are classified as y = 0, while points **outside the circle** satisfy $x1^2 + x2^2 \geq 1$ and are classified as y = 1. This shows that logistic regression can model complex, non-linear boundaries by using suitable feature transformations.

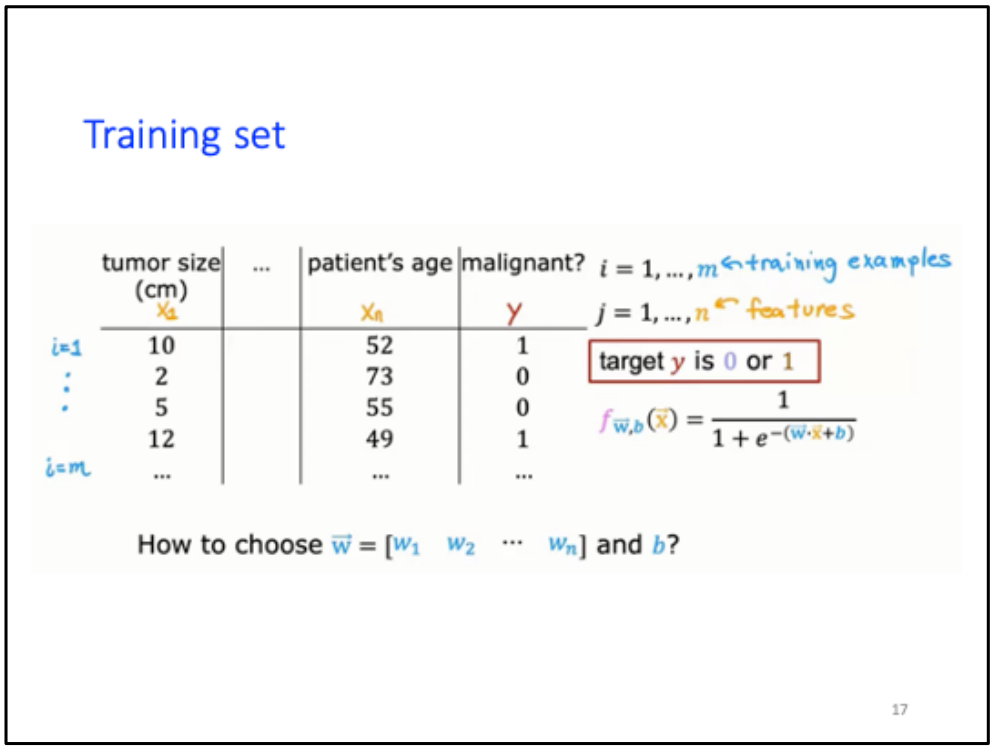


Slide Notes – Complex Non-Linear Decision Boundary

This slide shows that in real classification problems, the **decision boundary does not have to be a straight line or a simple shape**. When data points from different classes are mixed in complex ways, the model may need a **curved and irregular boundary** to separate them correctly. In the figure, blue circles represent **Class 0**, and red crosses represent **Class 1**.

A **non-linear decision boundary** allows the model to closely follow the structure of the data, separating the two classes more accurately. Such boundaries are created by using **non-linear features** or more powerful models. This example highlights why simple linear models are sometimes not enough and why more flexible decision boundaries are important in practical machine learning problems.

Slide Notes – Cost Function for Logistic Regression (Introduction)
In this section, we introduce the **cost function for logistic regression**, which tells us **how well the model is performing**. The cost function measures the difference between the **predicted probabilities** and the **true class labels** in the training data.
Choosing the right cost function is very important. The squared error cost used in linear regression does **not work well** for classification. Logistic regression uses a **different cost function** that is designed specifically for probabilities and helps the model learn parameters that produce accurate and reliable classifications.
When ready, send the **next slide** where the cost function is introduced or visualized



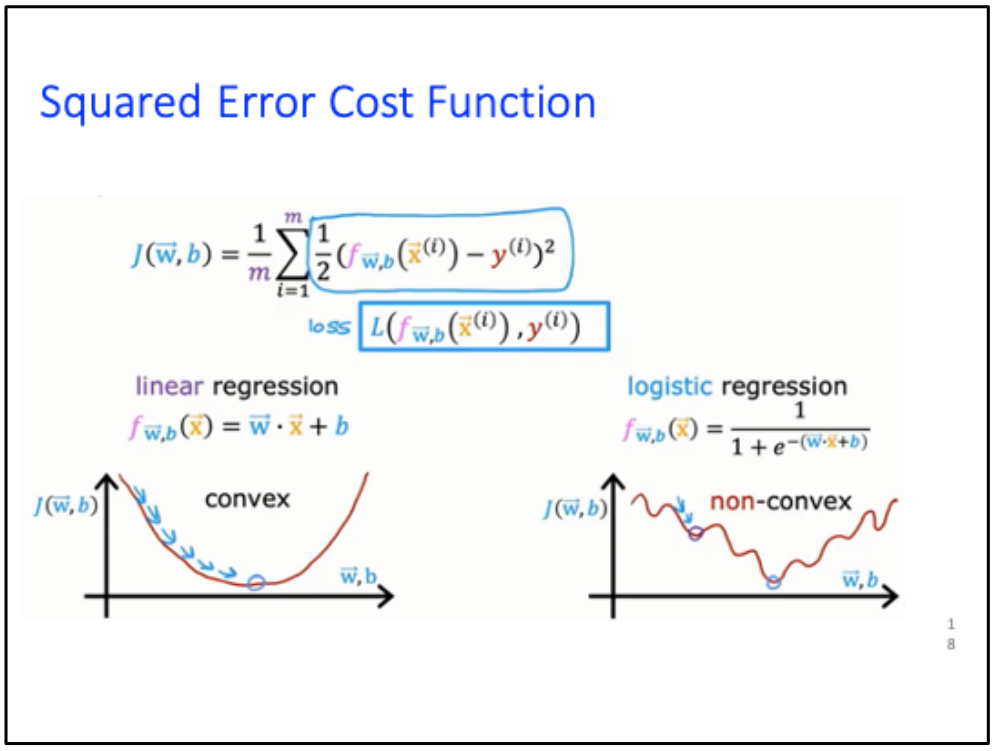
Slide Notes – Training Set

This slide describes the **training set** used for logistic regression. Each row represents **one training example**, indexed by i = 1 to m, where m is the number of training examples. Each example has several **input features** such as tumor size and patient's age, which are denoted as x1, x2, …, xn. The output **y** is the target label and can only take two values: **0 or 1**.

The logistic regression model uses the features to compute a prediction using:

f_w,b(x) = 1 / (1 + e^(−(w . x + b)))

The key learning task is to choose the parameters **w = [w1, w2, …, wn]** and **b** so that the model predicts the correct y values for the training data as accurately as possible. This is done by minimizing a cost function, which we will define and study next.



Slide Notes – Squared Error Cost

This slide shows the **squared error cost function** and explains why it works well for **linear regression** but not for **logistic regression**. The squared error cost is defined as:

J(w, b) = (1 / m) * Σ (1 / 2) * (f_w,b(x(i)) − y(i))²

For linear regression, where

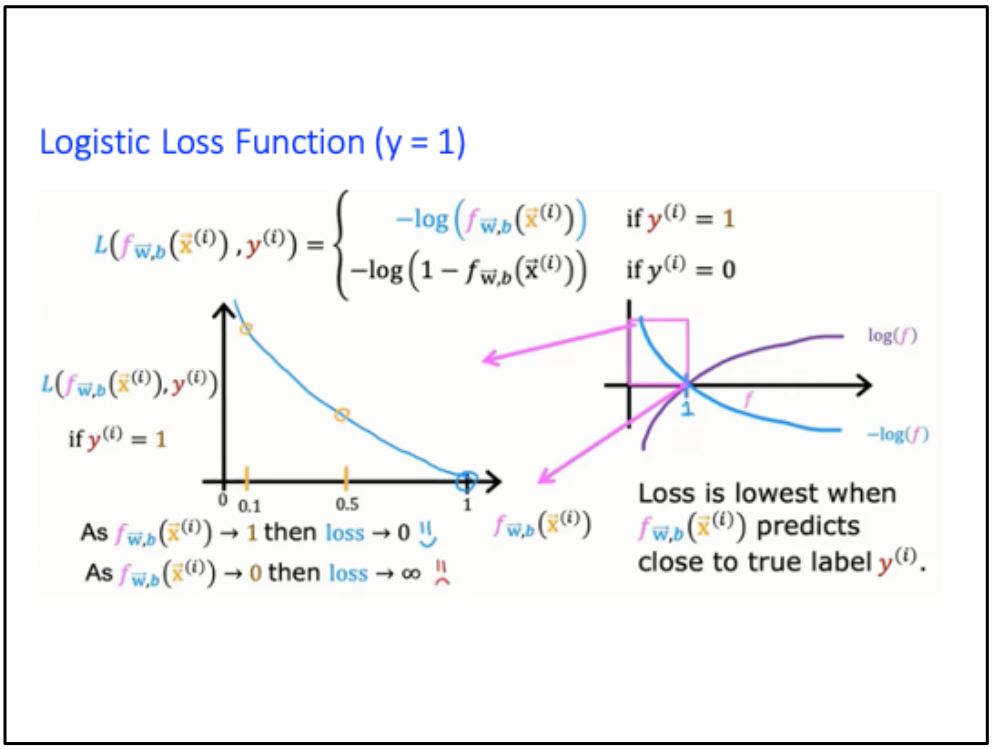
f_w,b(x) = w . x + b,

this cost function produces a **convex shape**. A convex cost function has a single global minimum, which makes optimization using gradient descent reliable and efficient.

However, when we use the **same squared error cost** with logistic regression, where

f_w,b(x) = 1 / (1 + e^(−(w . x + b))),

the cost function becomes **non-convex**. This means it can have many local minima, making optimization difficult and unreliable. For this reason, squared error cost is **not suitable for logistic regression**, and a different cost function is required.



Slide Notes – Logistic Loss Function

This slide introduces the **logistic loss function**, which is used to train logistic regression models. The loss is defined **per training example** and depends on the true label y(i):

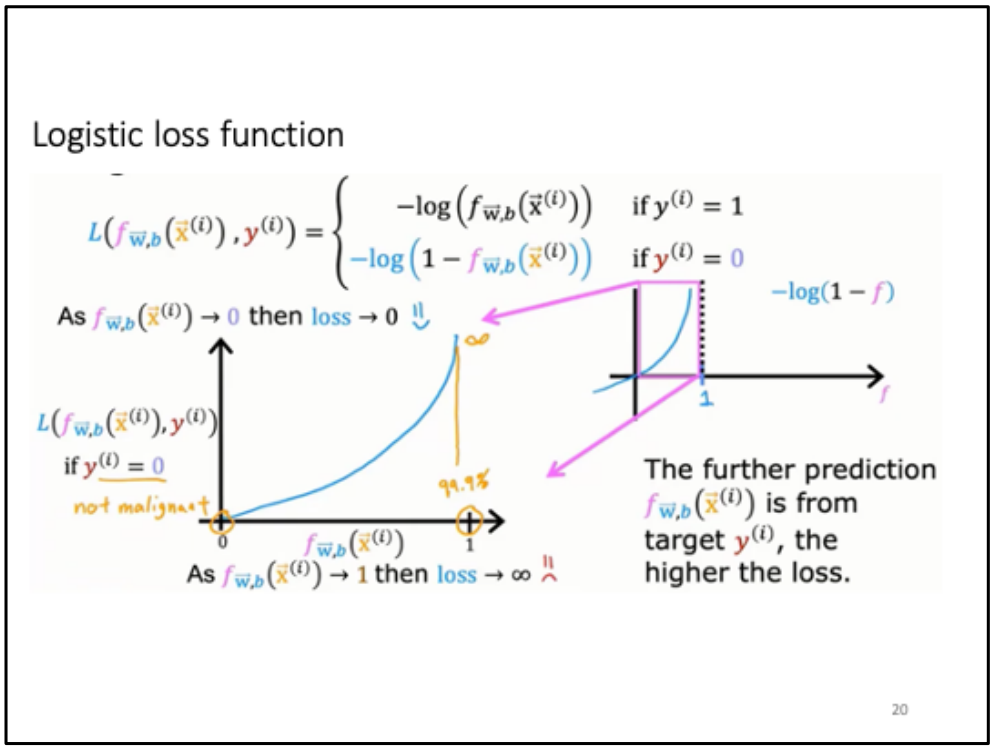
If y(i) = 1:

L(f_w,b(x(i)), y(i)) = −log(f_w,b(x(i)))

If y(i) = 0:

L(f_w,b(x(i)), y(i)) = −log(1 − f_w,b(x(i)))

The logistic loss becomes **very large when the model is confident but wrong**. For example, predicting f_w,b(x(i)) close to 1 when the true label is y(i) = 0 (or close to 0 when y(i) = 1) results in a very high loss. When the model predicts values close to the true label, the loss is close to zero. This behavior forces the model to avoid overconfident incorrect predictions and encourages accurate probability estimates.



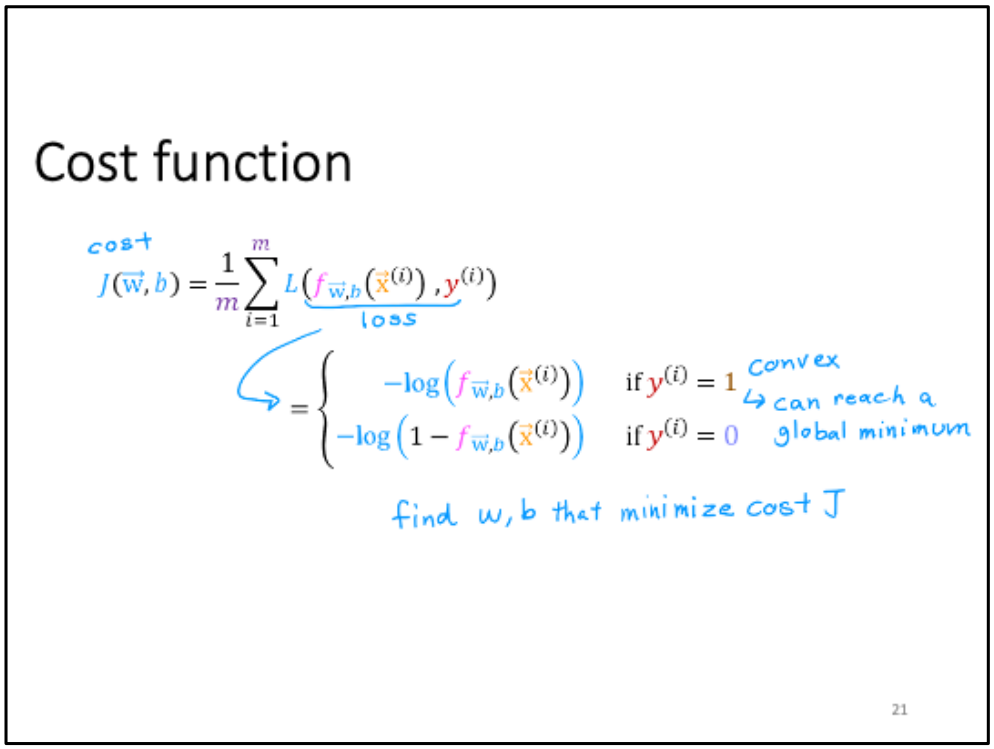
Slide Notes – Logistic Loss Function (y = 0 case)

This slide explains the **logistic loss function when the true label is y(i) = 0** (for example, a benign tumor). In this case, the loss is defined as:

L(f_w,b(x(i)), y(i)) = −log(1 − f_w,b(x(i)))

If the model predicts f_w,b(x(i)) close to **0**, the prediction is correct and the loss is close to **0**. However, if the model predicts a value close to **1**, the prediction is **confident but wrong**, and the loss increases rapidly and approaches **infinity**.

This behavior shows that the logistic loss strongly penalizes predictions that are far from the true label. The **further the predicted probability is from the correct class**, the higher the loss. This encourages the model to produce probabilities that are well aligned with the true labels and discourages overconfident mistakes.



Slide Notes – Cost Function for Logistic Regression

This slide defines the **cost function** used in logistic regression. The total cost J(w, b) is computed as the **average of the loss over all training examples**:

J(w, b) = (1 / m) * Σ L(f_w,b(x(i)), y(i))

Each individual loss L(·) is given by the logistic loss function:

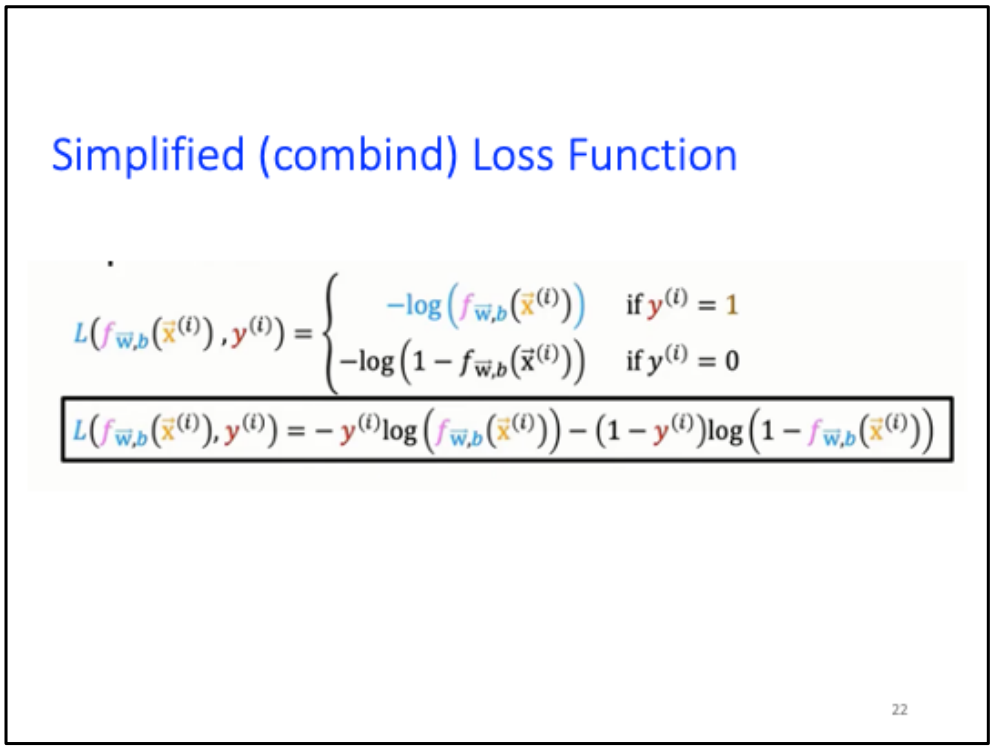
If y(i) = 1:

L = −log(f_w,b(x(i)))

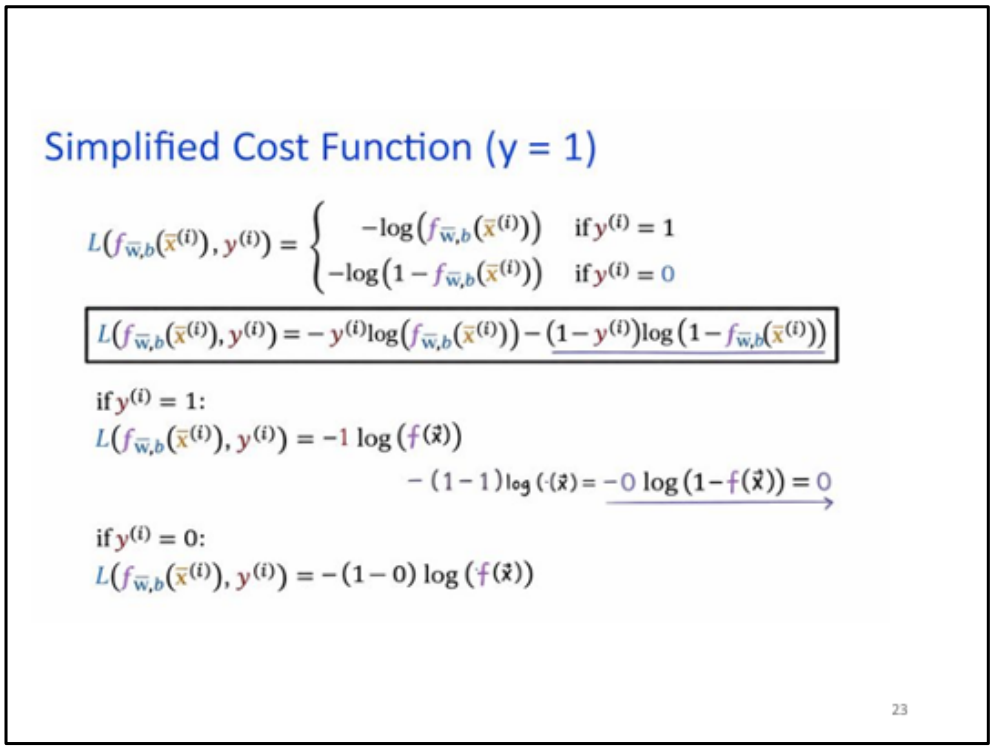
If y(i) = 0:

L = −log(1 − f_w,b(x(i)))

The goal of training is to **find the values of w and b that minimize the cost J(w, b)**. An important property of this cost function is that it is **convex**, which means it has a single global minimum. This allows gradient descent to reliably find the best parameters without getting stuck in local minima.



Slide Notes – Simplified (Combined) Loss Function
This slide shows how the two cases of the logistic loss function can be **combined into a single equation**. Previously, we defined the loss separately for y(i) = 1 and y(i) = 0. Both cases can be written compactly as:
L(f_w,b(x(i)), y(i))
= − y(i) . log(f_w,b(x(i))) − (1 − y(i)) . log(1 − f_w,b(x(i)))
This single formula works because **y(i) is either 0 or 1**. If y(i) = 1, the second term becomes zero and only −log(f_w,b(x(i))) remains. If y(i) = 0, the first term becomes zero and only −log(1 − f_w,b(x(i))) remains. This combined form is easier to implement in practice and is the standard loss function used for training logistic regression models.



Slide Notes – Simplified Cost Function (y = 1)

This slide shows how the **combined logistic loss function** simplifies when the true label is **y(i) = 1**. Recall the combined loss formula:

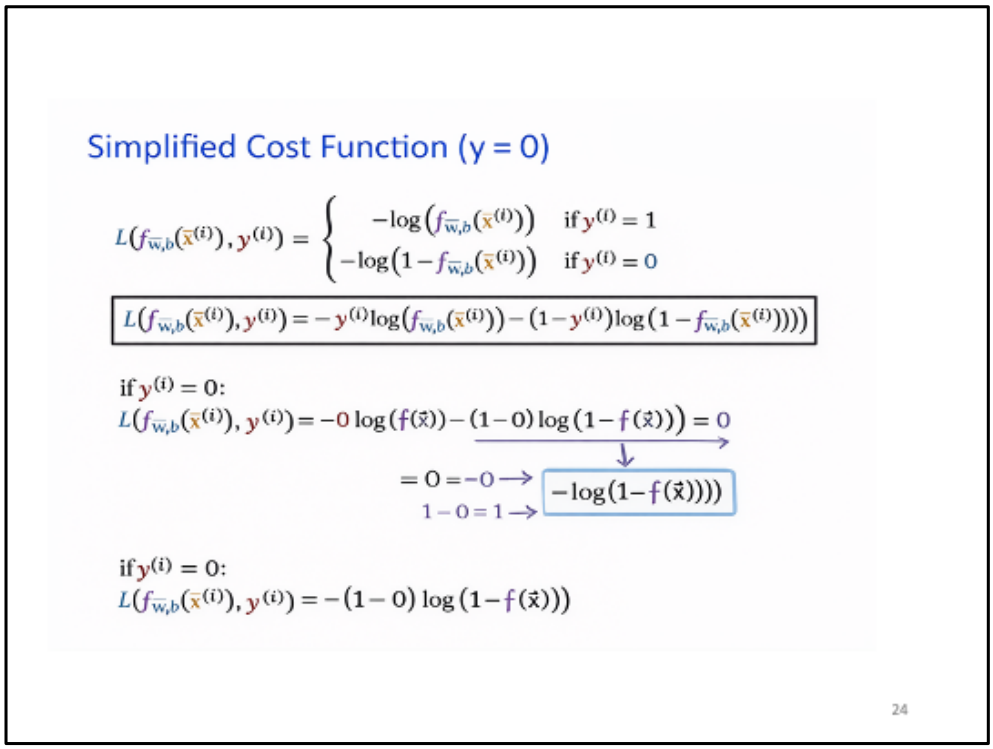
L(f_w,b(x(i)), y(i))

= − y(i) . log(f_w,b(x(i))) − (1 − y(i)) . log(1 − f_w,b(x(i)))

When **y(i) = 1**, we substitute into the formula. The second term becomes zero because (1 − y(i)) = 0, so it disappears. The loss simplifies to:

L = − log(f_w,b(x(i)))

This means that when the true label is 1, the loss depends only on how close the predicted probability **f_w,b(x(i))** is to 1. If the model predicts a value close to 1, the loss is small. If it predicts a value close to 0, the loss becomes very large. This strongly encourages the model to assign high probability to the correct positive class.



Slide Notes – Simplified Cost Function (y = 0)

This slide explains how the **combined logistic loss function** simplifies when the true label is **y(i) = 0**. We start from the combined loss formula:

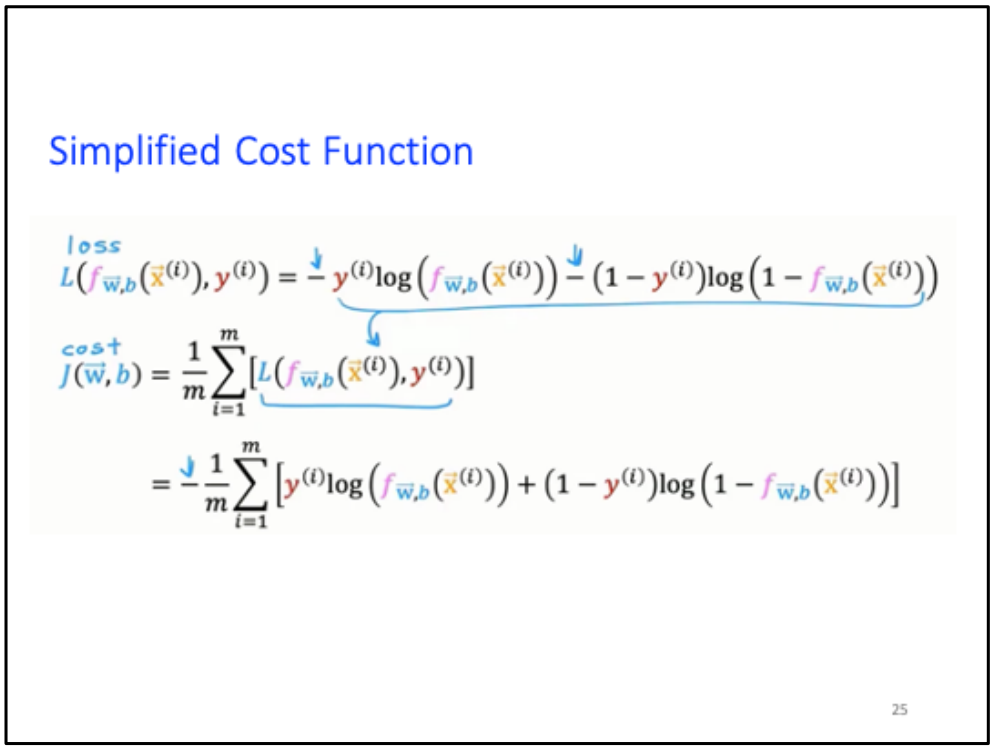
L(f_w,b(x(i)), y(i))

= − y(i) . log(f_w,b(x(i))) − (1 − y(i)) . log(1 − f_w,b(x(i)))

When **y(i) = 0**, the first term becomes zero because y(i) = 0. The loss therefore simplifies to:

L = − log(1 − f_w,b(x(i)))

This means that when the true label is 0, the loss depends on how close the predicted probability **f_w,b(x(i))** is to 0. If the model predicts a value close to 0, the loss is small. If it predicts a value close to 1, the loss becomes very large. This strongly penalizes confident but incorrect predictions for the negative class and helps the model learn accurate probabilities.



Slide Notes – Simplified Cost Function

This slide puts everything together by showing how the **cost function** for logistic regression is built from the **loss function**. First, the loss for a single training example is written in its combined form:

L(f_w,b(x(i)), y(i))

= − y(i) . log(f_w,b(x(i))) − (1 − y(i)) . log(1 − f_w,b(x(i)))

This loss measures how wrong the prediction is for **one example**.

The **overall cost function** J(w, b) is then defined as the **average loss over all m training examples**:

J(w, b)

= (1 / m) . Σ L(f_w,b(x(i)), y(i))

Substituting the loss expression gives:

J(w, b)

= − (1 / m) . Σ [y(i) . log(f_w,b(x(i))) + (1 − y(i)) . log(1 − f_w,b(x(i)))]

This is the final cost function used to train logistic regression. The goal of learning is to find values of **w** and **b** that **minimize this cost**, meaning the model predicts probabilities that are close to the true labels across the entire training set.

1️⃣ Loss Function (Single Training Example)

$$L\big(f_{w,b}(x^{(i)}),\, y^{(i)}\big) = -y^{(i)} \log\big(f_{w,b}(x^{(i)})\big) - (1 - y^{(i)}) \log\big(1 - f_{w,b}(x^{(i)})\big)$$

What this means:

- Measures **error for one data point**
- If prediction is good → loss is small
- If prediction is bad → loss is large

2️⃣ Cost Function (All Training Examples)

$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} L\big(f_{w,b}(x^{(i)}),\, y^{(i)}\big)$$

What this means:

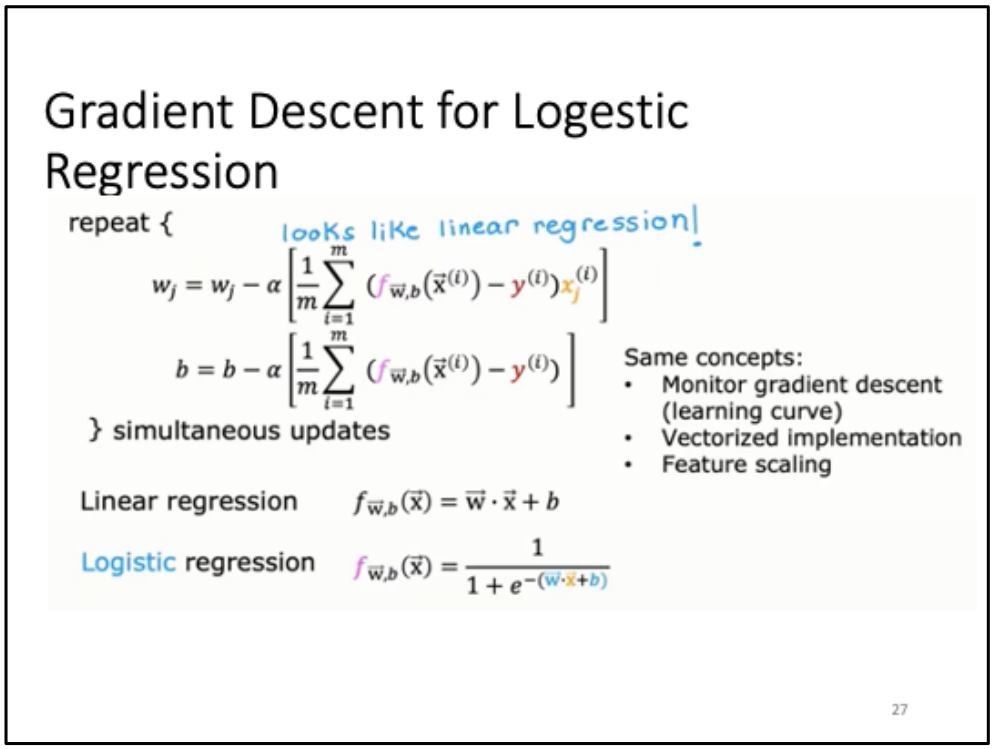
- Average loss over **all training examples**
- Tells us **how good the model is overall**

3️⃣ Expanded Cost Function

$$J(w,b) = -\frac{1}{m} \sum_{i=1}^{m} \Big[y^{(i)} \log\big(f_{w,b}(x^{(i)})\big) + (1 - y^{(i)}) \log\big(1 - f_{w,b}(x^{(i)})\big) \Big]$$

What this means:

- Combines all losses into **one formula**
- This is the function we **minimize using gradient descent**



Slide Notes – Gradient Descent for Logistic Regression

This slide shows how **gradient descent** is used to train a logistic regression model. The update rules for the parameters look very similar to those used in linear regression. For each weight w_j, the update rule is:

w_j = w_j − α . (1 / m) . Σ (f_w,b(x(i)) − y(i)) . x_j(i)

The bias term is updated as:

b = b − α . (1 / m) . Σ (f_w,b(x(i)) − y(i))

Here, α is the **learning rate**, m is the number of training examples, and f_w,b(x(i)) is the prediction of the logistic regression model.

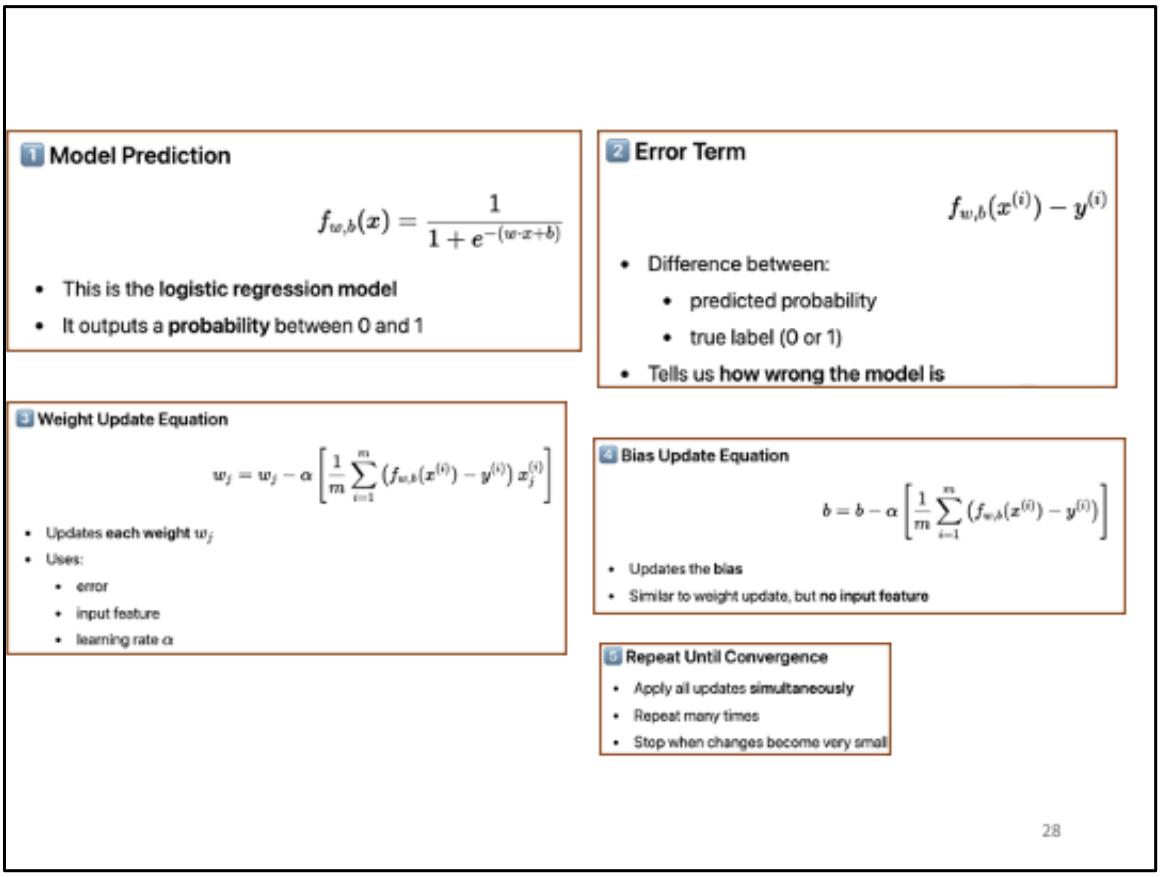
The key difference from linear regression is the **prediction function**. In linear regression,

f_w,b(x) = w . x + b,

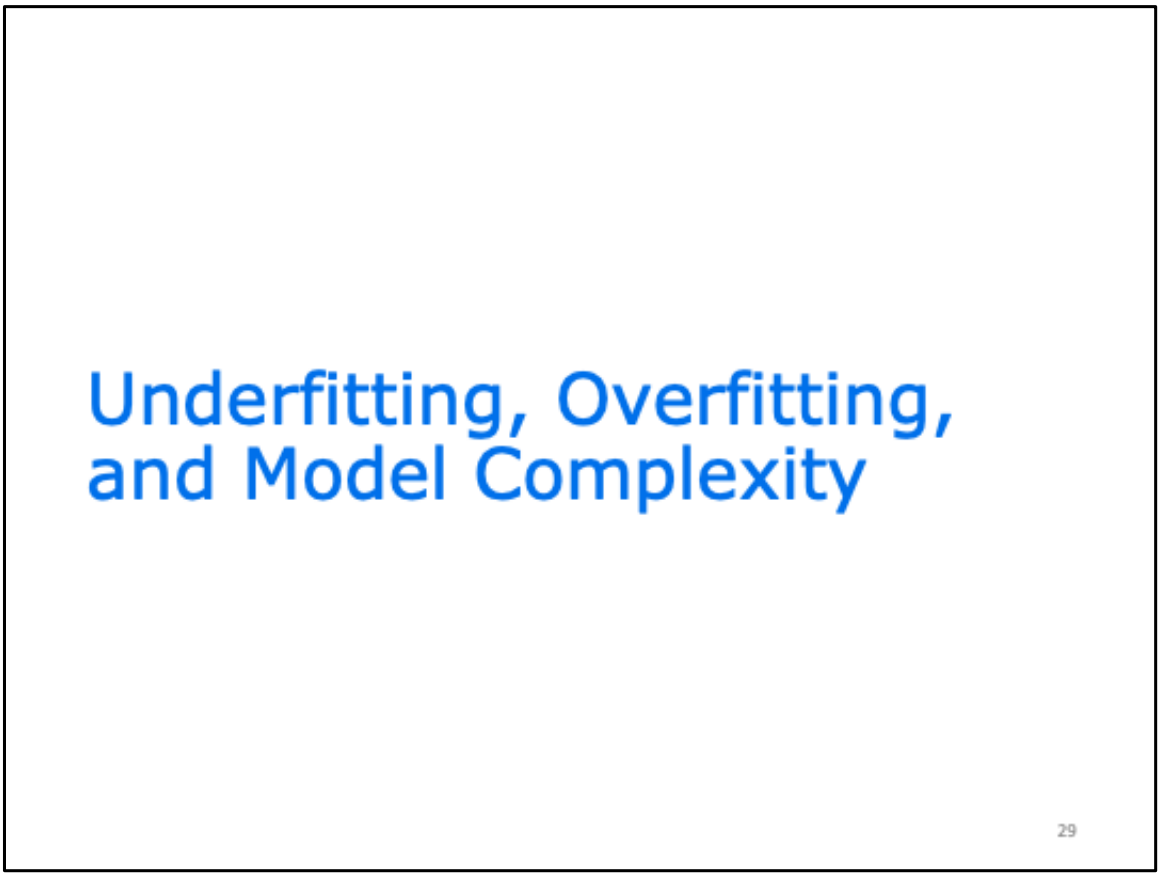
while in logistic regression,

f_w,b(x) = 1 / (1 + e^(−(w . x + b))).

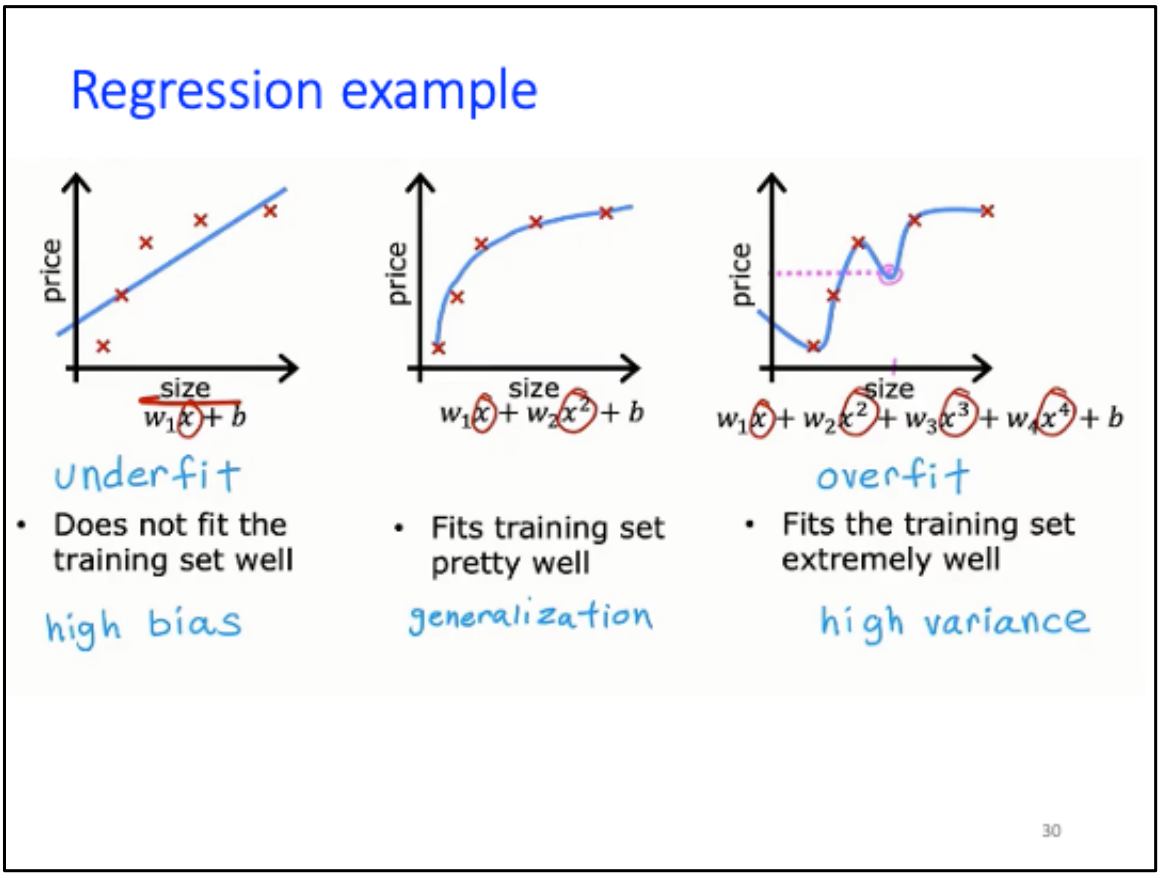
Even though the update formulas look similar, they are optimizing a **different cost function**. The same practical ideas still apply, such as monitoring convergence, using vectorized implementations, and applying feature scaling to improve training speed.



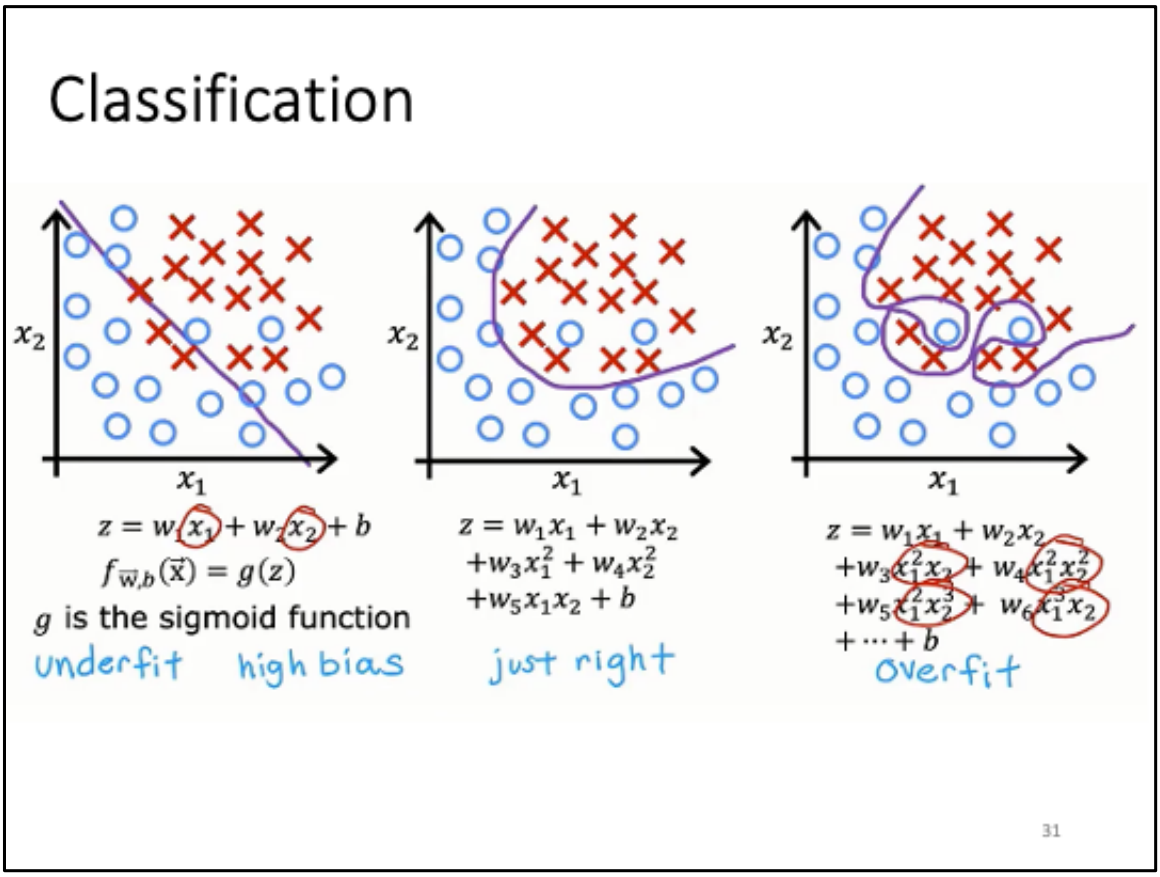
Slide Notes – Gradient Descent Components for Logistic Regression
This slide breaks gradient descent for logistic regression into clear parts. First, the **model prediction** is computed using the logistic regression function:
f_w,b(x) = 1 / (1 + e^(−(w . x + b))).
This output is a **probability between 0 and 1**, representing how likely the input belongs to class 1.
Next, the **error term** is defined as the difference between the predicted probability and the true label:
f_w,b(x(i)) − y(i).
This error tells us **how wrong the model is** for a given training example and is the key signal used to update the parameters.
The **weight update equation** adjusts each weight w_j using the error, the corresponding input feature x_j(i), and the learning rate α. Similarly, the **bias update equation** adjusts b using the average error, without involving any input feature. Finally, all updates are applied **simultaneously** and repeated many times until the model **converges**, meaning the parameter changes become very small.



Slide Notes – Underfitting, Overfitting, and Model Complexity
In this section, we study how the **complexity of a model** affects its performance. A model that is too simple may fail to capture important patterns in the data, leading to **underfitting**. A model that is too complex may fit the training data very closely but perform poorly on new data, leading to **overfitting**.
The goal of machine learning is to find a model that achieves good **generalization**, meaning it performs well on both training data and unseen data. Understanding underfitting and overfitting helps us choose the right model complexity and design models that learn meaningful patterns rather than noise.

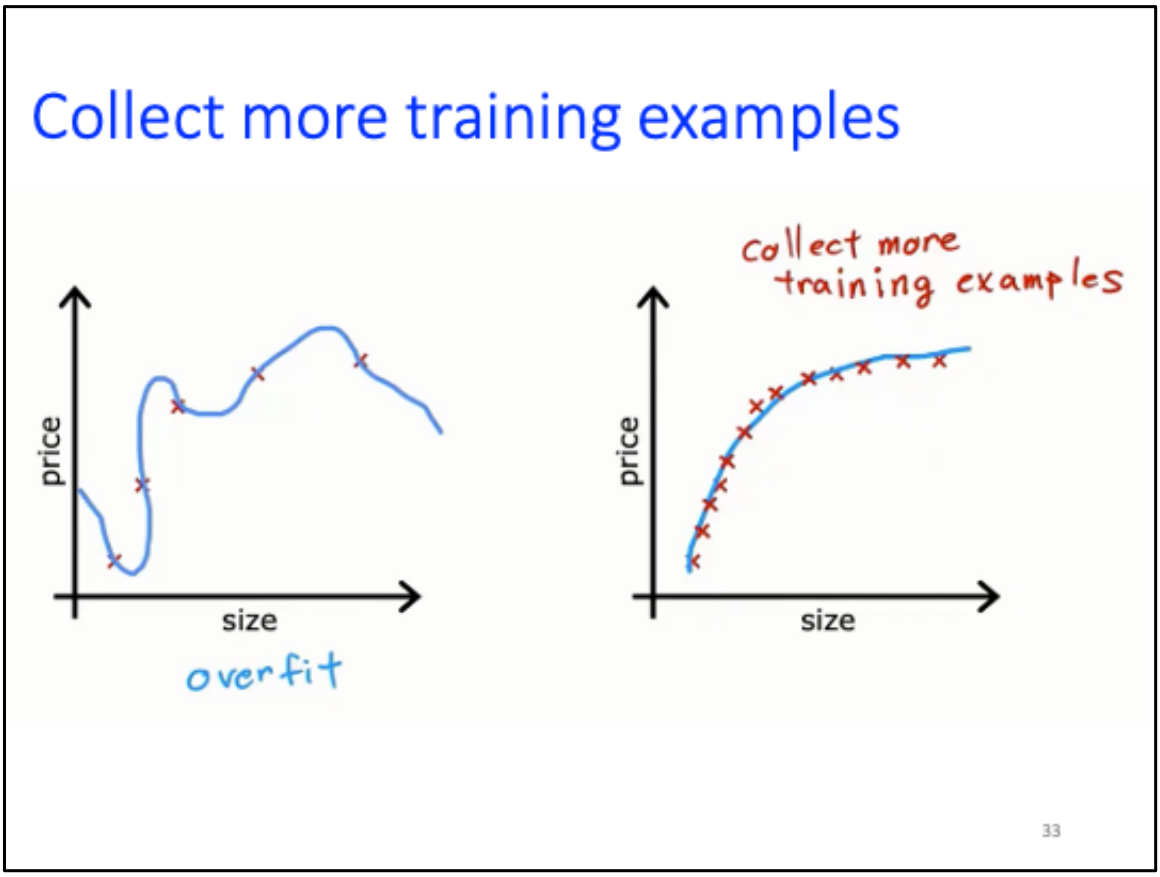


Slide Notes – Underfitting vs Overfitting (Regression Example)
This slide shows three regression models with different **levels of model complexity**. The first model is **too simple** (a straight line). It cannot capture the pattern in the data, so it **underfits** the training set. This model has **high bias**, meaning it makes strong assumptions and ignores important relationships in the data.
The middle model has **moderate complexity** and fits the training data **pretty well** without being too flexible. This is the **ideal case**, where the model captures the true pattern and can **generalize well** to new, unseen data. The last model is **too complex** and fits the training data extremely closely. This is called **overfitting**. Such a model has **high variance**, meaning it is very sensitive to the training data and may perform poorly on new data.



Slide Notes – Underfitting vs Overfitting (Classification Example)
This slide shows how **model complexity affects classification performance**. In the first figure, the decision boundary is a **simple straight line**. It cannot separate the two classes well, so the model **underfits** the data. This model has **high bias** because it is too simple to capture the true structure of the data.
In the middle figure, the decision boundary is **more flexible** and separates the two classes reasonably well. This model represents the **ideal balance**, where the boundary captures the main pattern in the data and can **generalize well** to new examples. In the last figure, the decision boundary is **very complex** and closely follows the training data. This model **overfits**, has **high variance**, and may perform poorly on unseen data because it learns noise instead of the true pattern.

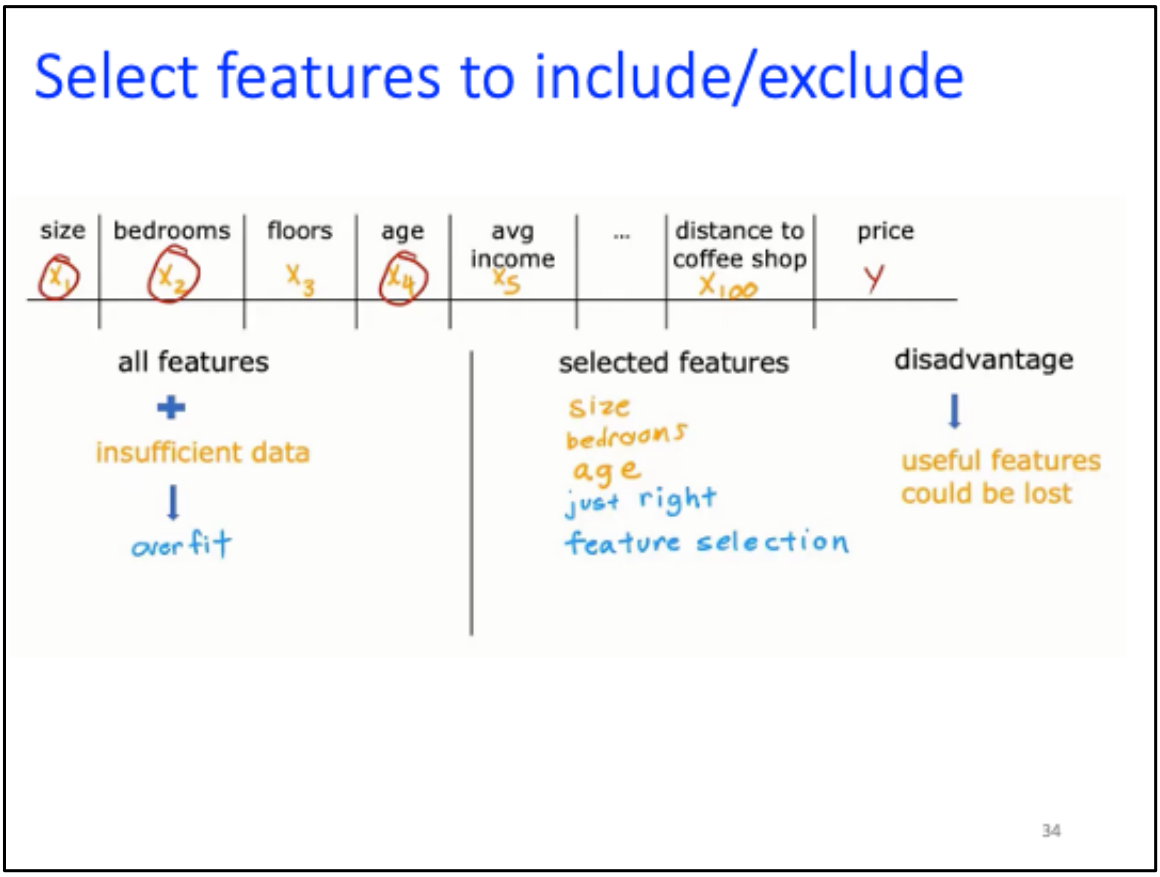
Slide Notes – Addressing Overfitting
In this section, we discuss **why overfitting happens** and how to reduce it. Overfitting occurs when a model is too complex and learns noise or small details from the training data instead of the true underlying pattern.
We will explore common techniques to address overfitting, such as **reducing model complexity**, **adding more training data**, and **using regularization**. These methods help the model generalize better and improve performance on unseen data.



Slide Notes – Collect More Training Examples

This slide shows one effective way to **reduce overfitting**: collecting more training data. In the left figure, the model overfits because it tries to follow every small fluctuation in a **small dataset**, resulting in a very irregular curve that does not represent the true relationship well.

In the right figure, after **adding more training examples**, the model sees a clearer and more stable pattern in the data. As a result, the fitted curve becomes smoother and more representative of the true relationship. More data helps reduce **variance**, making the model generalize better to new, unseen examples.

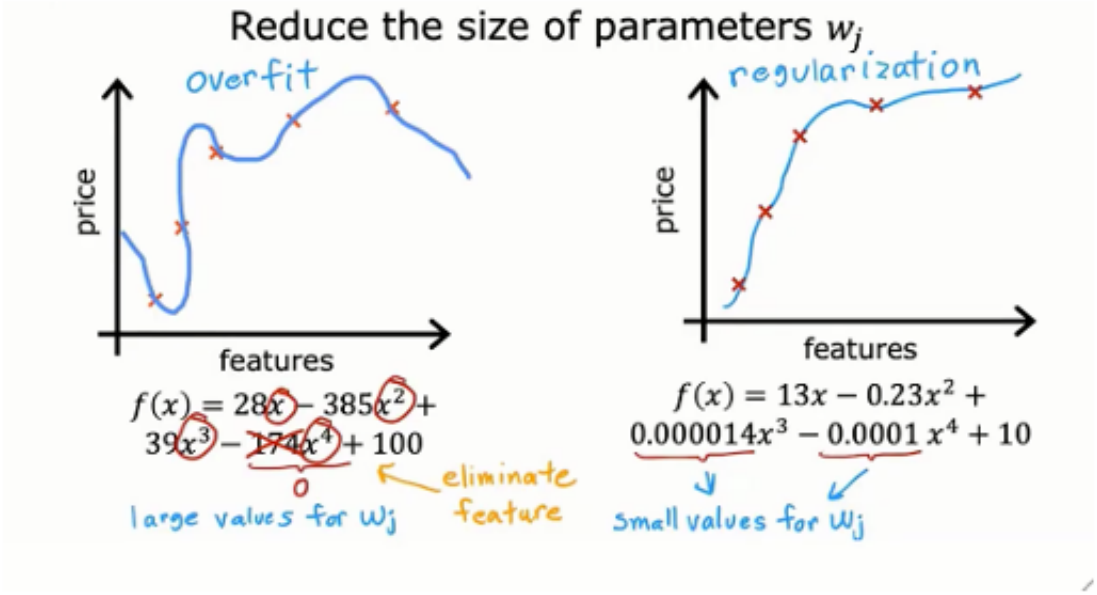


Slide Notes – Select Features to Include or Exclude

This slide explains how **feature selection** can help address overfitting. Using **all available features** may seem useful, but when the dataset is small, too many features can cause the model to overfit. The model may learn noise instead of meaningful patterns, leading to poor generalization.

By selecting a **subset of relevant features** (such as size, number of bedrooms, and age), the model becomes simpler and focuses on the most important information. This can improve performance and reduce overfitting. However, feature selection has a **trade-off**: if we remove too many features, we might lose useful information. The goal is to choose features that give the **right balance** between simplicity and predictive power.

Regularization

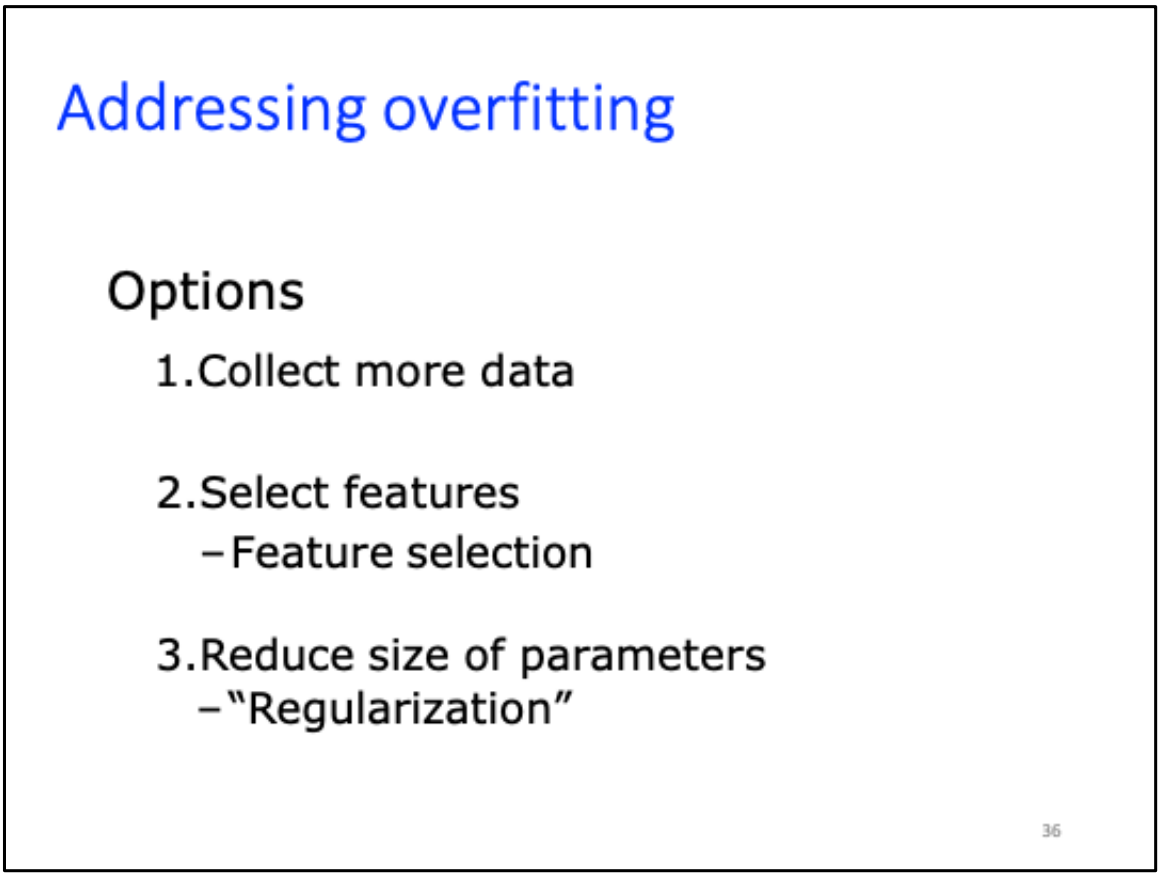


35

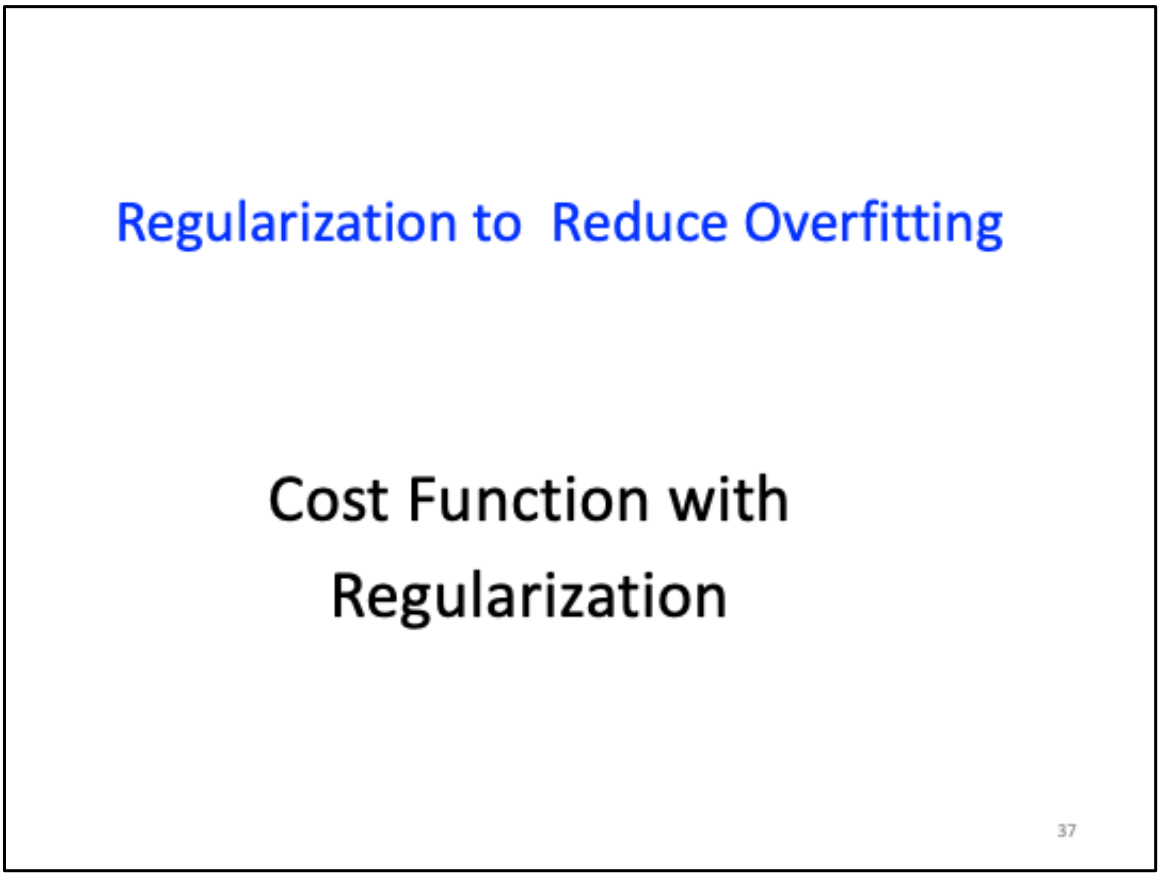
Slide Notes – Regularization

This slide introduces **regularization**, a powerful technique used to reduce overfitting. Overfitting often happens when model parameters (weights) become **very large**, allowing the model to fit noise in the training data. In the left figure, large parameter values lead to a very complex curve that fits the training data too closely.

Regularization works by **penalizing large weights**, encouraging the model to keep parameter values small. As shown in the right figure, smaller weights result in a **smoother and simpler model** that captures the main trend in the data. This helps the model generalize better to new data while still maintaining good predictive performance.

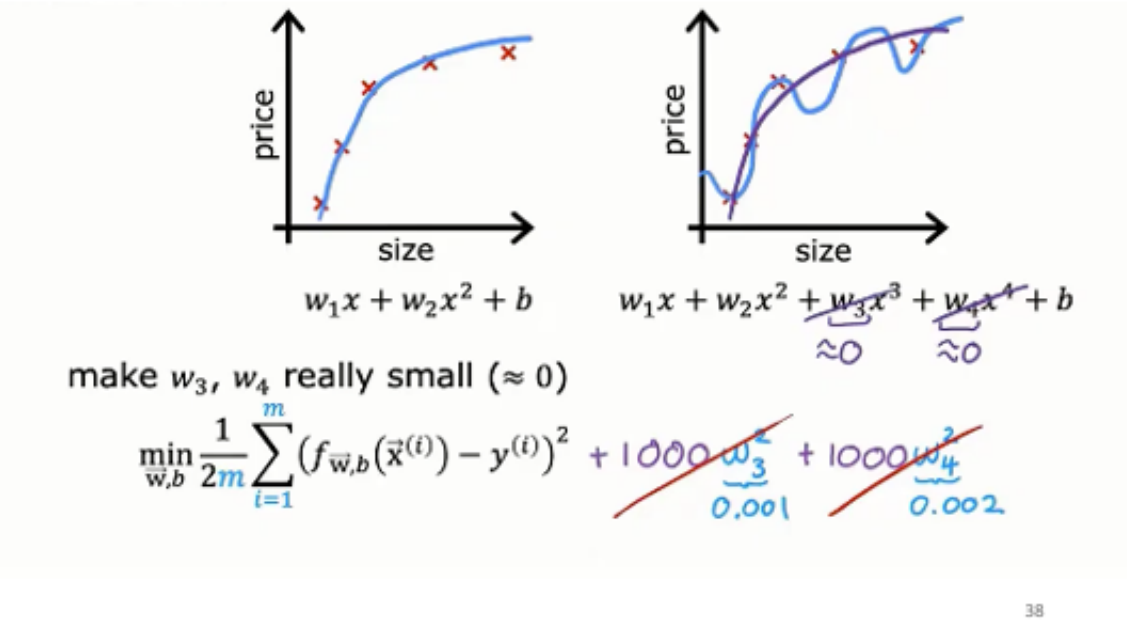


Slide Notes – Addressing Overfitting (Summary of Options)
This slide summarizes the **main strategies used to address overfitting** in machine learning models. The first option is to **collect more training data**, which helps the model see a wider variety of examples and reduces its tendency to memorize noise.
The second option is **feature selection**, where we choose only the most relevant input features and remove unnecessary ones. This reduces model complexity and improves generalization. The third option is **regularization**, which reduces the size of model parameters by penalizing large weights. Regularization encourages simpler models that focus on the main patterns in the data rather than overfitting to small details.



Slide Notes – Cost Function with Regularization
In this section, we introduce **regularization in the cost function** as a way to reduce overfitting. Regularization works by adding an extra term to the cost function that **penalizes large parameter values**, especially large weights.
By discouraging very large weights, the model becomes **simpler and smoother**, focusing on the most important patterns in the data. This helps improve **generalization** and reduces the risk of overfitting, especially when using complex models or many features.

Intuition behind Regularization

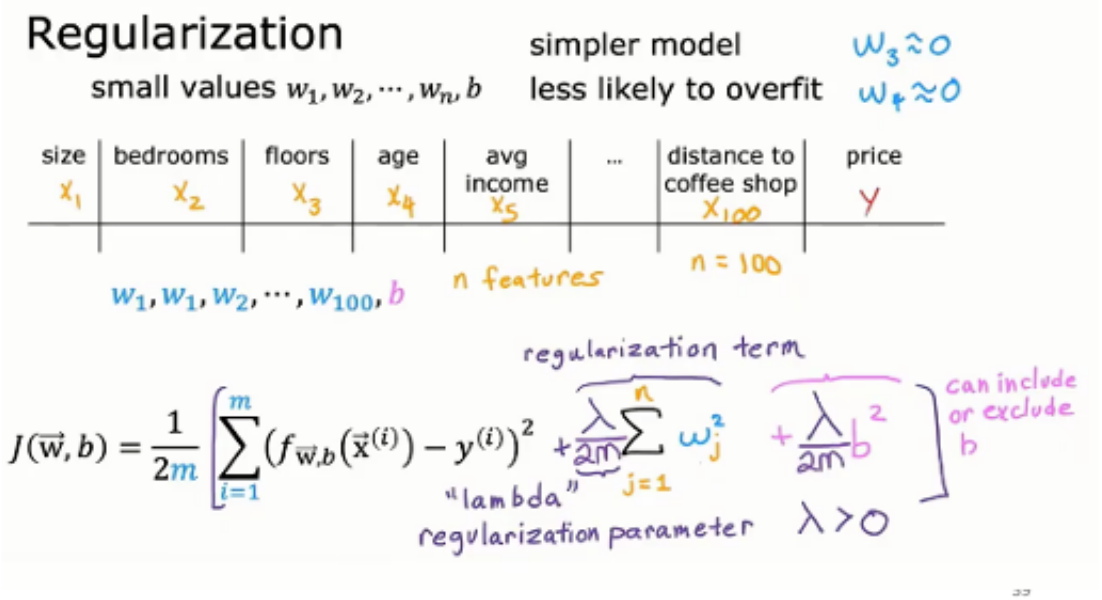


Slide Notes – Intuition Behind Regularization

This slide explains the **intuition behind regularization**. On the left, the model uses only a few important terms and produces a **smooth curve** that captures the main trend in the data. On the right, the model includes many higher-order terms, which allows it to fit the training data too closely and causes **overfitting**.

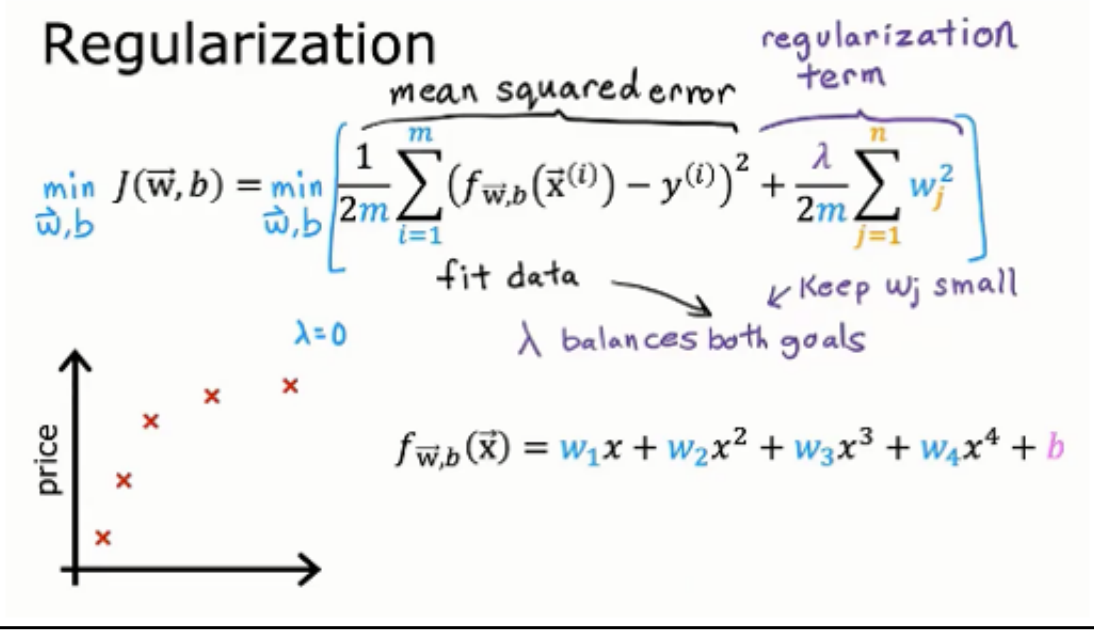
Regularization works by making the weights of less important features (such as w3, w4, and higher terms) **very small, close to zero**. This is done by adding a penalty term to the cost function that discourages large weights. As a result, the model keeps the important features and ignores unnecessary complexity, leading to better **generalization** on new data.

Regularization: Controlling Model Complexity with Small Weights



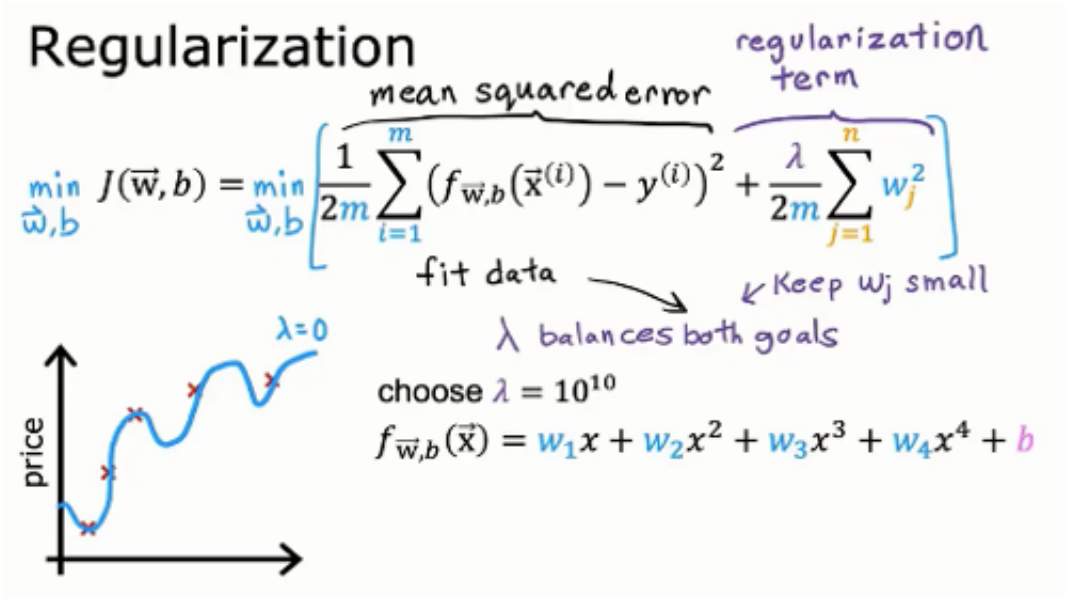
Suggested Slide Title (modified to reflect content):
Regularization: Controlling Model Complexity with Small Weights
Slide Notes – Regularization
This slide explains how **regularization** helps control model complexity by encouraging **small values for the parameters** w_1, w_2, …, w_n (and sometimes b). When a model has many features (for example n = 100), large weight values can make the model too sensitive to the training data, which leads to **overfitting**. Regularization reduces this risk by pushing unnecessary weights toward **zero**, resulting in a **simpler model** that is less likely to overfit.
This is done by adding a **regularization term** to the cost function. For example, in regularized linear regression, the cost function becomes:
J(w, b)
= (1 / (2m)) . Σ (f_w,b(x(i)) − y(i))² + (λ / (2m)) . Σ w_j²
Here, λ (lambda) is the **regularization parameter** that controls how strong the penalty is. A larger λ forces the weights to be smaller, while λ = 0 means no regularization. By choosing an appropriate λ, we balance fitting the data well and keeping the model simple, which improves **generalization**.

Regularization: Balancing Data Fit and Model Complexity



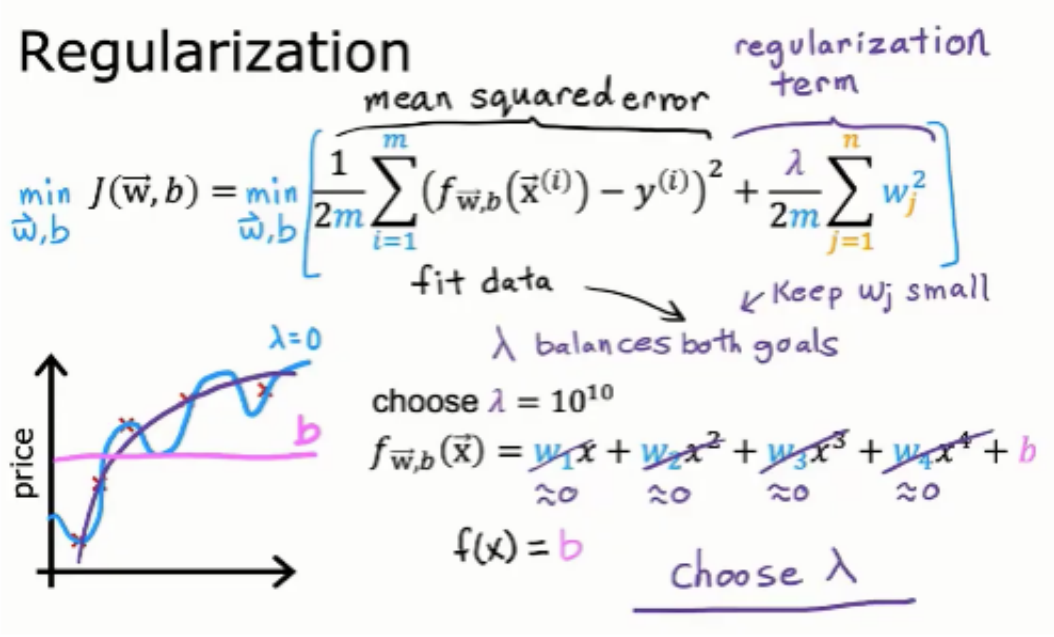
Slide Notes – Regularization: Balancing Data Fit and Model Complexity
This slide explains how **regularization modifies the cost function** to balance two goals at the same time. The first goal is to **fit the training data well**, which is measured by the mean squared error term. The second goal is to **keep the model simple** by penalizing large weight values. This is done by adding a regularization term that depends on the sum of the squared weights.
The regularized cost function can be written as:
$J(w, b) = (1 / (2m)) \cdot \Sigma (f_{w,b}(x(i)) - y(i))^2 + (\lambda / (2m)) \cdot \Sigma w_j^2$
Here, **λ (lambda)** is the regularization parameter that controls the trade-off between fitting the data and keeping the weights small. When λ is large, the model strongly penalizes large weights, resulting in a simpler and smoother model. When λ is small or zero, the model focuses more on fitting the training data, which can lead to overfitting.

Effect of Regularization (Role of λ)



Slide Notes – Effect of Regularization (Role of λ)
This slide shows how the **regularization parameter λ (lambda)** controls the behavior of the model. The cost function has two parts: the **data fitting term**, which tries to reduce prediction error, and the **regularization term**, which tries to keep the weights w_1, w_2, … small. Lambda controls the balance between these two goals.
When **λ = 0**, there is no regularization, so the model focuses only on fitting the training data. This can lead to a **very complex curve** and overfitting. When **λ is very large** (for example $\lambda = 10^{10}$), the regularization term dominates, forcing most weights to be very close to zero. As a result, higher-order terms become negligible and the model becomes **much simpler and smoother**, which helps reduce overfitting and improves generalization.

Choosing the Regularization Parameter λ



Slide Notes – Choosing the Regularization Parameter λ

This slide highlights the importance of **choosing an appropriate value for the regularization parameter λ**. The cost function includes a term that fits the data and a regularization term that keeps the weights small. Lambda controls how strongly we penalize large weights and therefore how simple the model becomes.

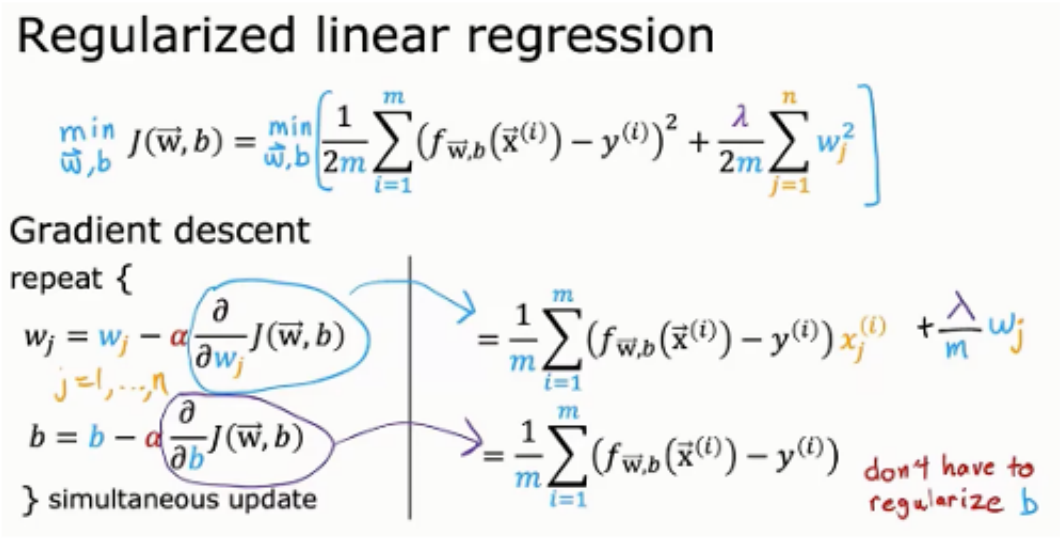
If **λ is too large**, almost all weights are pushed very close to zero. In this case, the model becomes **too simple** and may underfit the data. As shown in the slide, higher-order terms disappear and the model may reduce to something close to a constant function f(x) = b. If **λ is too small**, regularization has little effect and the model may overfit. Therefore, choosing λ carefully is essential to achieve a good balance between fitting the data and generalizing well to new examples.

Regularization to Reduce Overfitting

43

Slide Notes – Regularization to Reduce Overfitting
This slide introduces **regularization** as a key technique for reducing overfitting in machine learning models. Overfitting happens when a model becomes too complex and fits noise in the training data rather than the true underlying pattern.
Regularization works by **adding a penalty to the cost function** that discourages large parameter values. By keeping the weights small, the model becomes simpler and smoother, which improves **generalization** and leads to better performance on unseen data.

Regularized Linear Regression



Slide Notes – Regularized Linear Regression

This slide shows how **regularization is added to linear regression** and how it affects the **gradient descent update rules**. The regularized cost function includes the usual mean squared error term plus a penalty on the weights:

J(w, b)
= (1 / (2m)) . Σ (f_w,b(x(i)) − y(i))² + (λ / (2m)) . Σ w_j²

The goal is still to **minimize J(w, b)** with respect to w and b.

When we apply gradient descent, the **weight update rule changes** because of the regularization term. Each weight w_j is updated using:

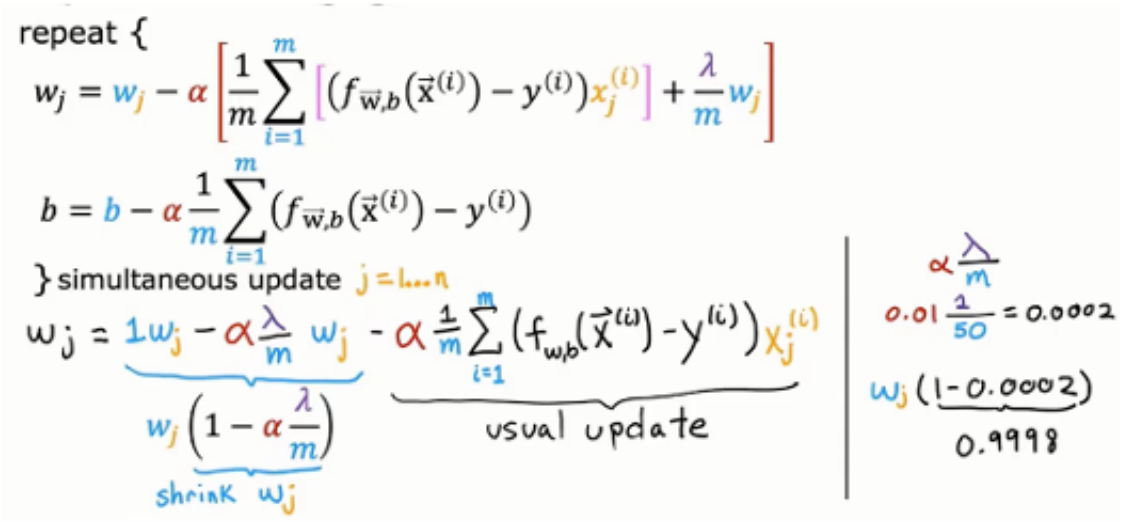
w_j = w_j − α . [(1 / m) . Σ (f_w,b(x(i)) − y(i)) . x_j(i) + (λ / m) . w_j]

This extra term pushes w_j toward zero. The **bias term b is not regularized**, so its update rule stays the same:

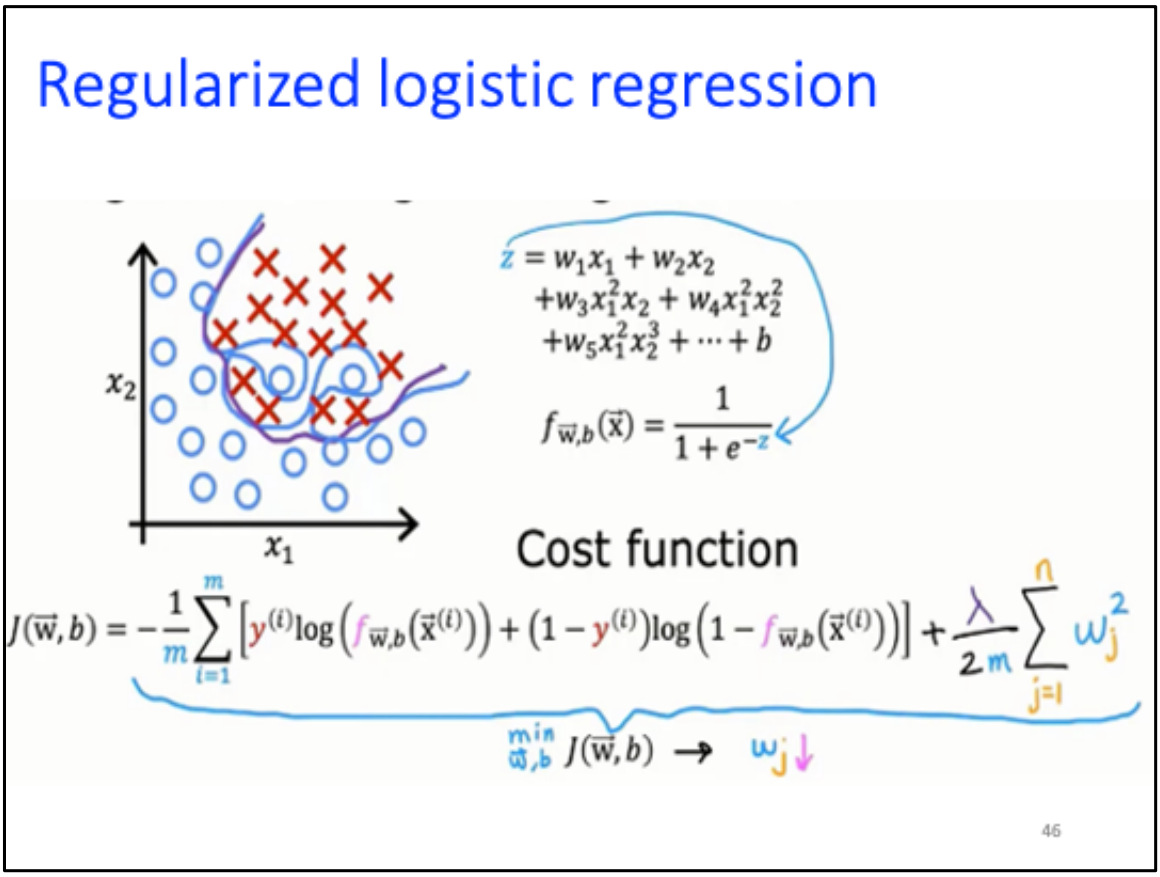
b = b − α . (1 / m) . Σ (f_w,b(x(i)) − y(i))

Regularization therefore reduces overfitting by shrinking weights while still allowing the model to fit the data.

Implementing Gradient Descent with Regularization



Slide Notes – Implementing Gradient Descent with Regularization
This slide explains how **gradient descent is implemented when regularization is included**. The weight update rule now has two parts: the usual error-based update and an additional regularization term. The update for each weight w_j is:
w_j = w_j − α . [(1 / m) . Σ (f_w,b(x(i)) − y(i)) . x_j(i) + (λ / m) . w_j]
The bias term is updated as usual and is **not regularized**:
b = b − α . (1 / m) . Σ (f_w,b(x(i)) − y(i))
The key intuition is that regularization causes each weight to be **slightly shrunk toward zero at every step**. This can be seen by rewriting the update as:
w_j ← w_j . (1 − α . λ / m) − α . (1 / m) . Σ (f_w,b(x(i)) − y(i)) . x_j(i)
The factor (1 − α . λ / m) is called **weight decay**. It gradually reduces the size of w_j, preventing large weights and helping reduce overfitting while still learning from the data.



Slide Notes – Regularized Logistic Regression

This slide shows how **regularization is applied to logistic regression** to reduce overfitting, especially when using many features or higher-order terms. The model still computes a linear combination of features:

z = w1 . x1 + w2 . x2 + w3 . x1² + w4 . x2² + w5 . x1 . x2 + … + b

This value is passed through the sigmoid function:

f_w,b(x) = 1 / (1 + e^(−z))

To control model complexity, a **regularization term** is added to the cost function. The regularized logistic regression cost becomes:

J(w, b)

= − (1 / m) . Σ [y(i) . log(f_w,b(x(i))) + (1 − y(i)) . log(1 − f_w,b(x(i)))]

+ (λ / (2m)) . Σ w_j²

The regularization term penalizes large weights, forcing unnecessary parameters toward zero. This produces a **smoother decision boundary**, reduces overfitting, and improves **generalization** while keeping the logistic regression model effective for classification.

Gradient Descent for Regularized Logistic Regression

$$J(\vec{w},b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1-y^{(i)})\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$

$\min_{\vec{w},b}$

Gradient descent

Looks same as for linear regression!

repeat {

$$w_j = w_j - \alpha\frac{\partial}{\partial w_j}J(\vec{w},b) \quad j=1\ldots n$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)} + \frac{\lambda}{m}w_j$$

logistic regression

$$b = b - \alpha\frac{\partial}{\partial b}J(\vec{w},b)$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)$$

don't have to regularize

}

47

Slide Notes – Gradient Descent for Regularized Logistic Regression
This slide shows how **gradient descent is applied to regularized logistic regression**. The cost function includes the usual logistic loss plus a **regularization term** that penalizes large weights. The objective is to **minimize J(w, b)** with respect to the parameters.
The **weight update rule** looks very similar to linear regression with regularization:
w_j = w_j − α . [(1 / m) . Σ (f_w,b(x(i)) − y(i)) . x_j(i) + (λ / m) . w_j]
The first part comes from the logistic loss, and the second part is the regularization term that pushes w_j toward zero. The **bias term b is not regularized**, so its update remains:
b = b − α . (1 / m) . Σ (f_w,b(x(i)) − y(i))
Even though the update equations look similar to linear regression, the key difference is that **f_w,b(x)** comes from the **sigmoid function**. Regularization helps keep the decision boundary smooth, reduces overfitting, and improves generalization.

Stage	Problem	Solution (Next Concept)
1	We need to classify data into two classes (y = 0 or 1)	Use a **linear model**: z = w . x + b
2	z can be any real number (not a probability)	Apply the **sigmoid function**
3	We now have a probability, but need a final decision	Use a **threshold (0.5)**
4	We need a rule that separates classes	Define a **decision boundary**: w . x + b = 0
5	We don't know the best values of w and b	Define a **loss function (log loss)**
6	We need to measure overall model performance	Use a **cost function** (average loss)
7	We must minimize the cost function	Apply **gradient descent**
8	Model may fit noise in training data	Risk of **overfitting**
9	Overfitting hurts generalization	Add **regularization**
10	Final goal	A **simple, accurate, generalizable classifier** 48

Slide Notes – Logistic Regression as a Problem-Solving Story

We begin with a **classification problem**: given some input features **x**, we want to decide whether the output belongs to class **0 or 1**. A simple idea is to use a **linear model** to combine the inputs, computing z = w . x + b.

However, this value can be any real number, so it **cannot be used directly** to represent a class or a probability.

To solve this, we apply the **sigmoid function**, which maps any real value into the range **0 to 1**. This allows us to interpret the output as a **probability** that y = 1. But now we face a new question: *how do we turn this probability into a final class decision?*

This leads to the idea of a **threshold**, usually 0.5. Using this threshold creates a **decision boundary**, defined by w . x + b = 0, which separates the two classes. Now the model can make predictions—but we still don't know **how to choose the best values of w and b**.

To learn these parameters, we need a way to measure how wrong our predictions are. Squared error does not work well for probabilities, so we introduce the **logistic loss**, which penalizes confident but wrong predictions. Averaging this loss over all training examples gives us a **cost function**.

Once we have a cost function, we still need a method to **minimize it**. This problem is solved using **gradient descent**, which updates the

parameters step by step to reduce the cost. However, if the model becomes too complex, it may fit the training data too closely.
This creates the final problem: **overfitting**. To address it, we use **regularization**, which penalizes large weights and keeps the model simple. With regularization, the model generalizes better to new data.
In this way, each component of logistic regression exists to **solve a specific problem introduced by the previous step**, forming a complete and coherent classification pipeline.

Logistic Regression: Formula and Intuition

Formula / Concept	Intuition (What It Means)
$y \in \{0, 1\}$	Binary classification problem
$z = w \cdot x + b$	Combine input features into a single score
$f_{w,b}(x) = 1 / (1 + e^{(-z)})$	Sigmoid converts score into a probability
$f_{w,b}(x) = P(y = 1 \mid x)$	Output is probability of class 1
Threshold = 0.5	Decide between class 0 and class 1
$w \cdot x + b = 0$	Decision boundary where model is uncertain
$L = -y \cdot \log(f) - (1 - y) \cdot \log(1 - f)$	Penalizes confident wrong predictions
$J(w, b) = (1 / m) \cdot \Sigma L$	Measures average error over all data
Gradient descent	Adjusts w and b to reduce the cost
$+ (\lambda / 2m) \cdot \Sigma w^2$	Regularization keeps weights small
λ (lambda)	Controls model complexity
Final model	Accurate and generalizes well

49

Linear Regression vs Logistic Regression

Aspect	Linear Regression	Logistic Regression
Learning task	Regression	Classification
Output y	Continuous value	Binary value (0 or 1)
Model formula	f_w,b(x) = w . x + b	z = w . x + bf_w,b(x) = 1 / (1 + e^(−z))
Output range	(−∞ , +∞)	(0 , 1)
Interpretation	Predicted numeric value	Probability that y = 1
Decision boundary	Not defined	w . x + b = 0
Loss function	Squared error	Logistic (log) loss
Cost function	Mean squared error	Cross-entropy loss
Convex cost	Yes	Yes
Typical use	Predict prices, temperatures	Spam detection, disease prediction
Threshold needed	No	Yes (usually 0.5)
Regularization	Optional	Very important to avoid overfitting

50