Week - 1: Unsupervised Learning

What is Clustering?

Clustering is an **unsupervised learning** technique used to find **patterns or groups** (called **clusters**) in a dataset **without labels**.

Supervised vs. Unsupervised Learning

Aspect	Supervised Learning	Unsupervised Learning
Data includes labels?	Yes (features **x** and labels **y**)	No (only features **x**)
Goal	Learn to predict labels	Find patterns/structure
Example	Binary classification	Clustering

Applications of Clustering:

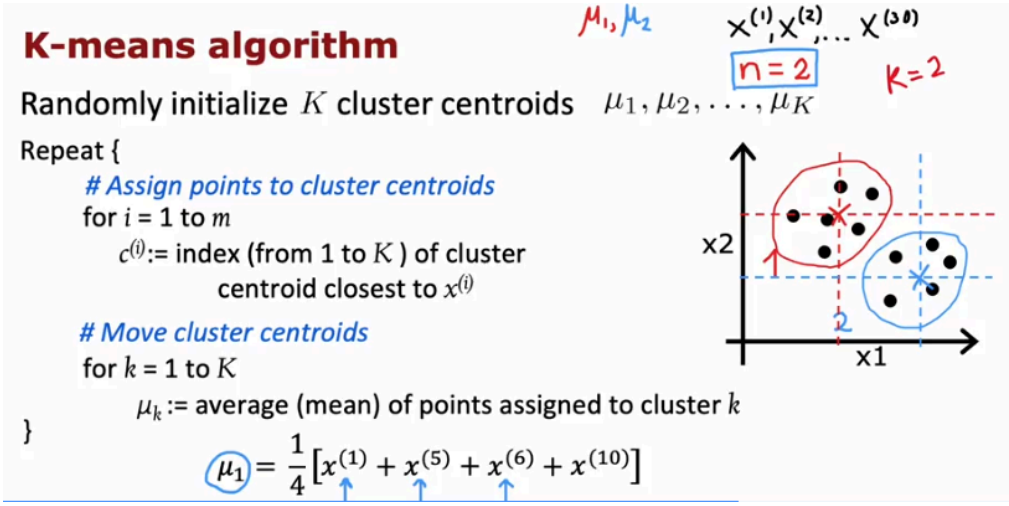
- **News article grouping** (e.g., similar topics like science or sports)
- **Market segmentation** (e.g., learners with different goals)
- **Genetic data analysis** (e.g., finding people with similar traits)
- **Astronomy** (e.g., grouping stars or galaxies)

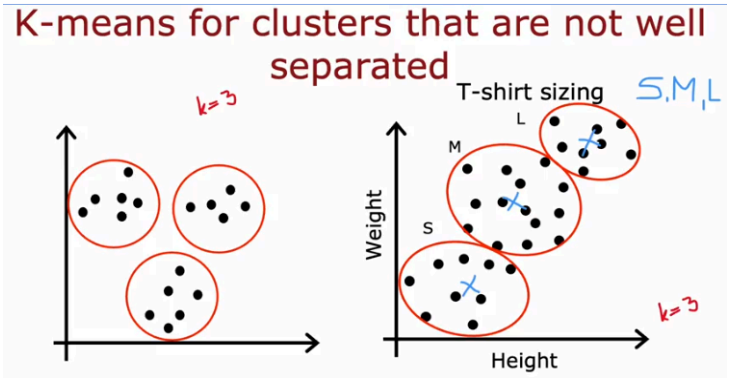
K-means Clustering

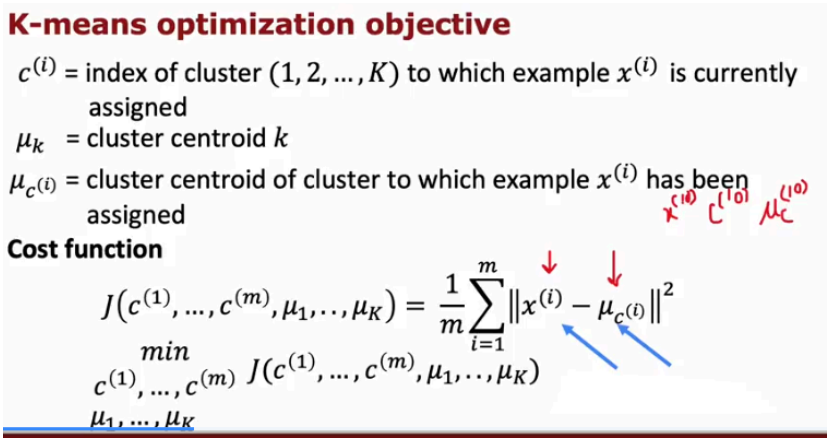
K-means is an unsupervised algorithm that groups data into clusters based on similarity.

K-means Algorithm

1. **Initialize Centroids**:
   Randomly choose K cluster centers (centroids).
2. **Repeat Until Convergence**:
   - **Assign Points**: Each data point is assigned to the nearest centroid.
   - **Update Centroids**: Move each centroid to the average of the points assigned to it.
3. **Special Case**:
   If a centroid has no points, either remove it or reinitialize it randomly.
4. **Convergence**:
   The algorithm stops when point assignments and centroid positions no longer change.







In supervised learning (like linear regression), we define a **cost function** and then **optimize** it (e.g., with gradient descent). Similarly, **K-means** also **minimizes a cost function**, but it uses a different method than gradient descent.

The Cost Function of K-means (Also Called the "Distortion Function")

The **cost function (J)** in K-means is:

$$J = \frac{1}{m} \sum_{i=1}^{m} \|x^{(i)} - \mu_{c^{(i)}}\|^2$$

Where:

- $x^{(i)}$: the i-th data point
- $c^{(i)}$: the index of the cluster assigned to $x^{(i)}$
- $\mu_{c^{(i)}}$: the centroid of the cluster that $x^{(i)}$ is assigned to

This is the **average squared distance** between each point and its assigned cluster centroid.

K-means Initialization

- K-means starts by **randomly selecting K training examples** as initial centroids.
- To get better results:
    - **Run K-means multiple times** (e.g., 100 times) with different initializations.
    - **Pick the best run** — the one with the **lowest cost (distortion J)**.

How to Choose the Number of Clusters in K-means

1. No One Correct Answer

- For the same data, some people might see **2 clusters**, others might see **4**, and both can be valid.
- This is because **clustering is unsupervised** — there are no labels to compare with.

2. Elbow Method

- One method used in research is the **elbow method**:
    - Run K-means with various values of K (e.g., 1 to 10).
    - Plot the **cost function (J)** against K. This function shows the **average squared distance** between points and their assigned centroids.
    - Initially, as K increases, the cost reduces sharply.
    - At some point, the cost starts decreasing more slowly — this point is called the **"elbow"**, and it's a suggested value for K.

But this method doesn't always work. In practice, many plots **don't show a clear elbow**, so it can still be ambiguous.

- We shouldn't pick K just by **minimizing the cost function**.
- More clusters **always reduce the cost**, but it can **overfit** or **make things unnecessarily complex**.

3. Better Strategy: Choose K Based on Your Goal

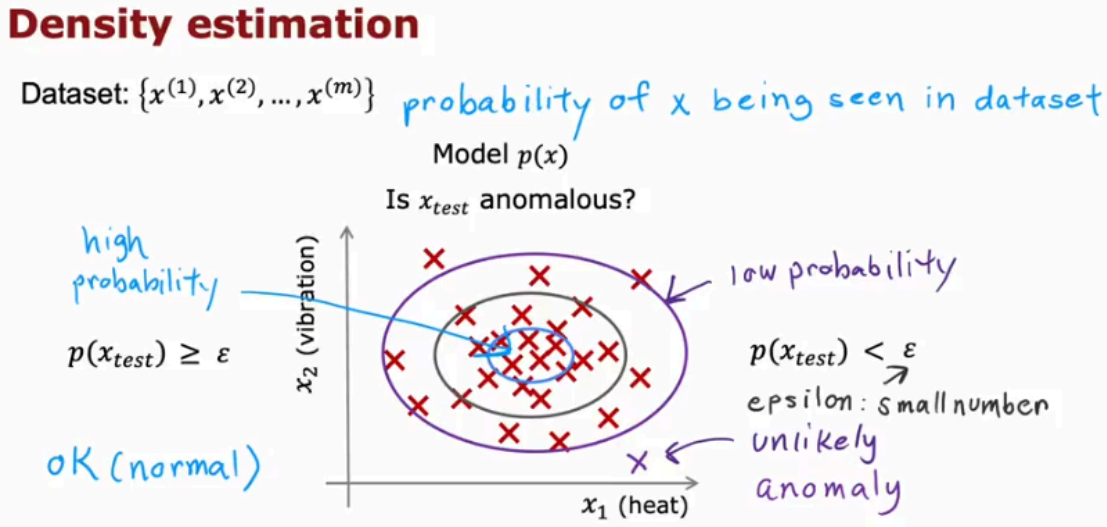
- Example: **T-shirt sizing**
  - K=3 → Small, Medium, Large
  - K=5 → XS, S, M, L, XL
- Both options are valid.

What is Anomaly Detection?

Anomaly detection is a technique in **unsupervised learning** where the algorithm learns from a dataset of **normal (non-anomalous) events** to identify **unusual or abnormal instances** that deviate significantly from what's considered normal.

How Anomaly Detection Works

1. The algorithm uses the **training data** to model the **probability distribution p(x)**.
2. This distribution tells us, What values of features (x1, x2) are **common** (high probability) vs **rare** (low probability).
3. If a new engine's feature vector X_test has a **low probability**, we flag it as **anomalous**.



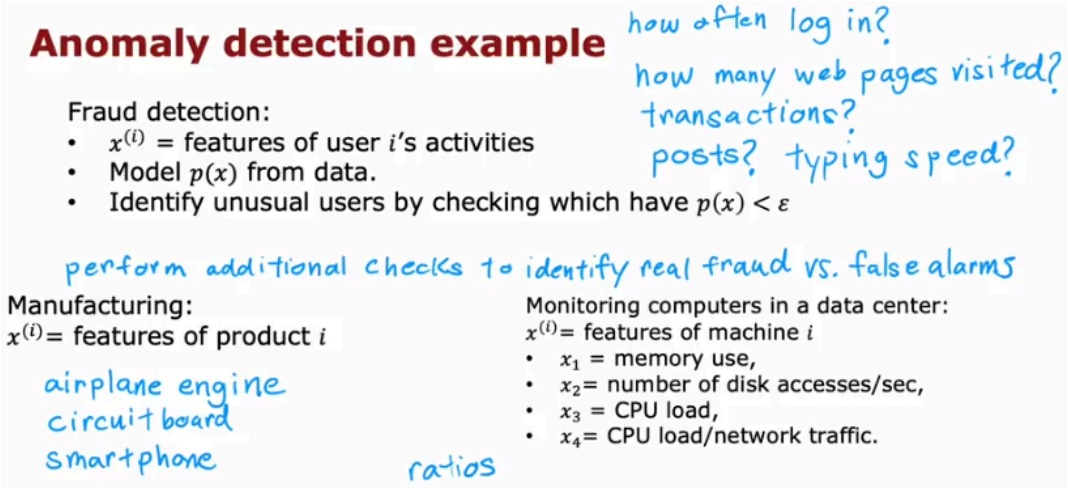
Threshold for Anomaly

- We define a small number **ε (epsilon)** as a threshold.
- If:
  p(X_test) < ε → it's **an anomaly**
  p(X_test) ≥ ε → it's **probably normal**
- Normal data clusters together (e.g., in ellipses in the center of the plot).
- Anomalies lie **far outside**, where probability p(x) is very low.

Applications of Anomaly Detection

Anomaly detection is used in **many real-world domains**, such as:

1. **Fraud Detection:**
    - Features: login frequency, typing speed, number of transactions.
    - Goal: Spot unusual behavior to flag potentially fraudulent activity.
    - Action: Flag for human review or extra verification steps.
2. **Manufacturing:**
    - Examples: engines, smartphones, motors, PCBs.
    - Goal: Detecting faulty units **before** they are shipped.
3. **Computer/Data Center Monitoring:**
    - Features: CPU load, memory usage, disk activity.
    - Goal: Detect failing or compromised machines.
4. **Telecom (Telco) Networks:**
    - Used to detect faulty **cell towers** so that technicians can fix them.



What is the Gaussian/Normal Distribution?

- It's a **probability distribution** used to describe data that clusters around a **mean (average)** value.
- Also known as the **bell-shaped curve**, because of its shape.
- It's commonly used in statistics, machine learning, and anomaly detection.

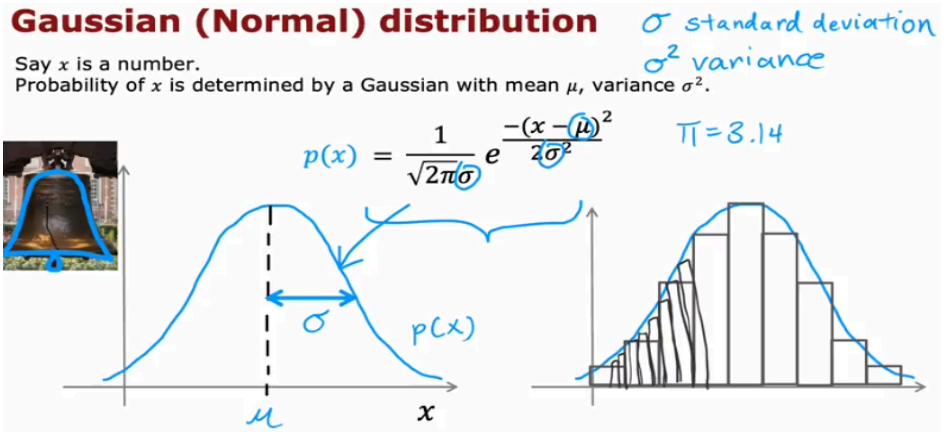
Parameters of a Gaussian Distribution:

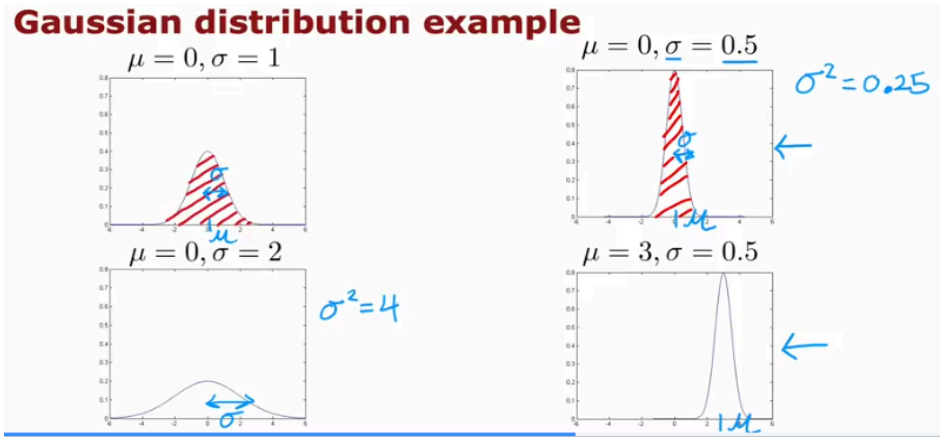
- **μ (Mu)**: the **mean** – the center of the distribution.
- **σ (Sigma)**: the **standard deviation** – measures how spread out the data is.
- **σ² (Sigma squared)**: the **variance** – it's just the square of the standard deviation./**average squared difference from the mean.**

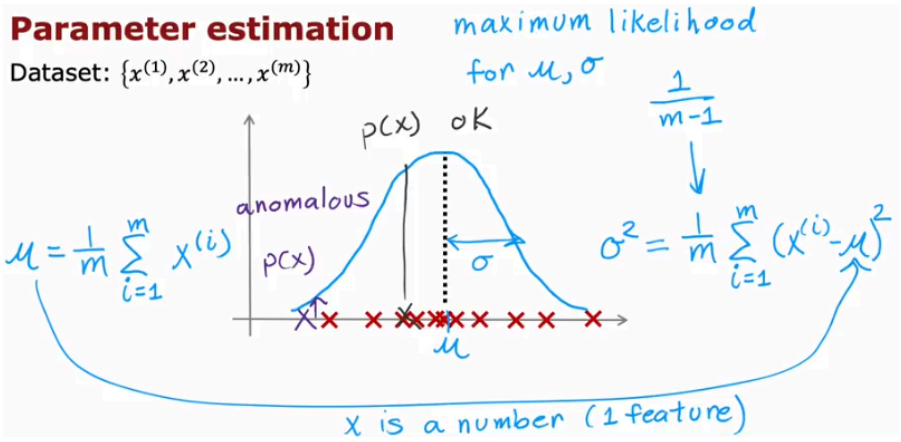
So:

- If **σ is small**, the curve is **narrow and tall**.

- If **σ is large**, the curve is **wide and short**.
- If **μ changes**, the whole curve **shifts left or right.**







Anomaly Detection Multiple features:

Steps to Build the Algorithm

1. Each example x(i) has n features → x=[x1,x2,...,xn]

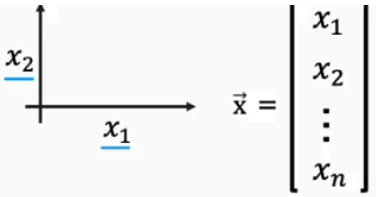
2. Model the overall probability as: **p(x) = p(x1) · p(x2) · … · p(xn)**
3. Assume each feature follows a Gaussian (Normal) distribution:

$$p(x_j) = \frac{1}{\sqrt{2\pi}\sigma_j} \cdot e^{-\frac{(x_j-\mu_j)^2}{2\sigma_j^2}}$$

4. From training data, compute:
    a. μj: mean of feature j
    b. σj2: variance of feature j
5. For a **test** point x, compute p(x) as the product of all p(xj)
6. Set a threshold ε:
    a. If p(x) < ε → Anomaly
    b. Else → Normal

Density estimation

Training set: $\{\vec{x}^{(1)}, \vec{x}^{(2)}, \dots, \vec{x}^{(m)}\}$
Each example $\vec{x}^{(i)}$ has n features



$$p(\vec{x}) = p(x_1;\mu_1,\sigma_1^2) * p(x_2;\mu_2,\sigma_2^2) * p(x_3;\mu_3,\sigma_3^2) * \cdots * p(x_n;\mu_n,\sigma_n^2)$$

$$= \prod_{j=1}^{n} p(x_j;\mu_j,\sigma_j^2)$$

"add" "multiply"

$p(x_1 = \text{high temp}) = 1/10$
$p(x_2 = \text{high vibra}) = 1/20$
$p(x_1, x_2) = p(x_1) * p(x_2)$
$= \frac{1}{10} \times \frac{1}{20} = \frac{1}{200}$

Anomaly detection algorithm

1. Choose n features x_i that you think might be indicative of anomalous examples.
2. Fit parameters $\mu_1, \dots, \mu_n, \sigma_1^2, \dots, \sigma_n^2$

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)} \qquad \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$$

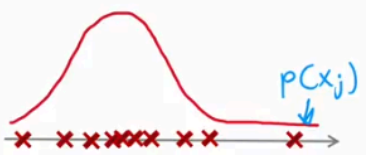
Vectorized formula

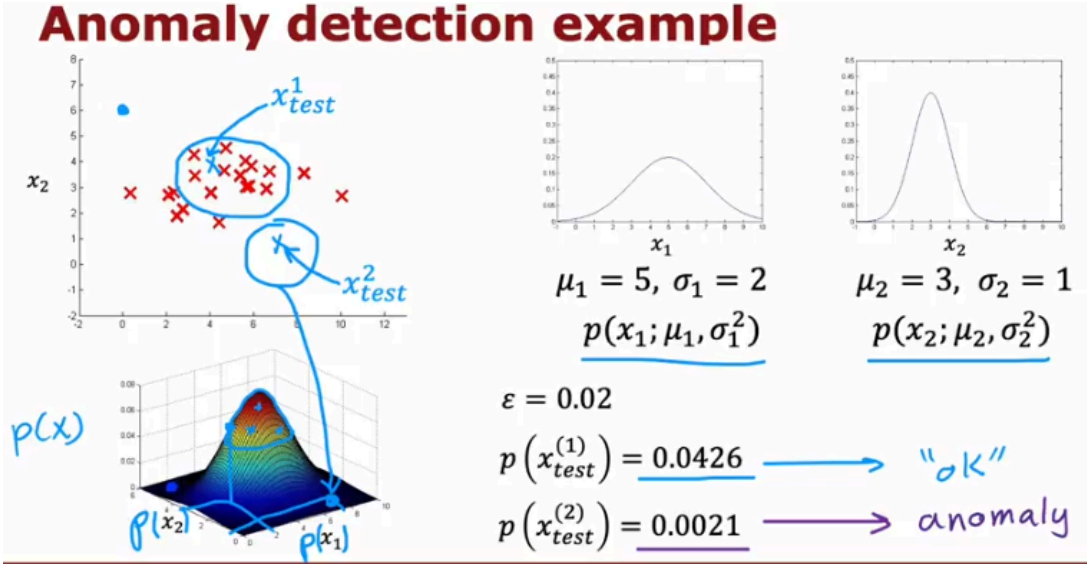
$$\vec{\mu} = \frac{1}{m}\sum_{i=1}^{m} \vec{x}^{(i)} \qquad \vec{\mu} = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \dots \\ \mu_n \end{bmatrix}$$

3. Given new example x, compute $p(x)$:

$$p(x) = \prod_{j=1}^{n} p(x_j;\mu_j,\sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} exp(-\frac{(x_j-\mu_j)^2}{2\sigma_j^2})$$

Anomaly if $p(x) < \varepsilon$





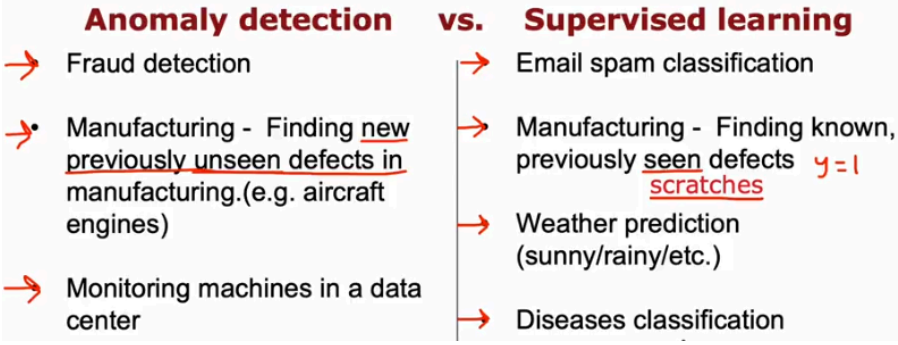
Anomaly detection vs. supervised learning

Key Points:

1. **Anomaly Detection:**
    - Best for situations where there are **very few positive examples** (0-20), and **many negative examples**.
    - It assumes **new types of anomalies** may appear that were never seen before. The algorithm learns from normal data (negative examples) and flags deviations as anomalies, even if the anomaly is entirely new.
2. **Supervised Learning:**
    - More appropriate when there are **more positive examples**.
    - Assumes **future positive examples** will resemble those in the training set. Works best if you have enough labeled data to predict outcomes based on similarities to previous examples.
3. **When to Choose Each:**
    - **Anomaly Detection**: Works well when dealing with unknown, new types of anomalies. Examples include fraud detection (e.g., financial fraud) where new fraud attempts keep emerging, and manufacturing defects where new types of failures may appear.
    - **Supervised Learning**: Works best for problems with **repeated, known patterns**. Examples include email spam detection (where spam is often similar over time) and predicting the weather (where patterns repeat).
4. **Manufacturing Example:**
    - If the defect (e.g., scratched smartphones) is well-known, supervised learning is ideal, as enough labeled data exists.
    - If the defect is unknown and could be something new, anomaly detection is more appropriate.
5. **Security Applications:**

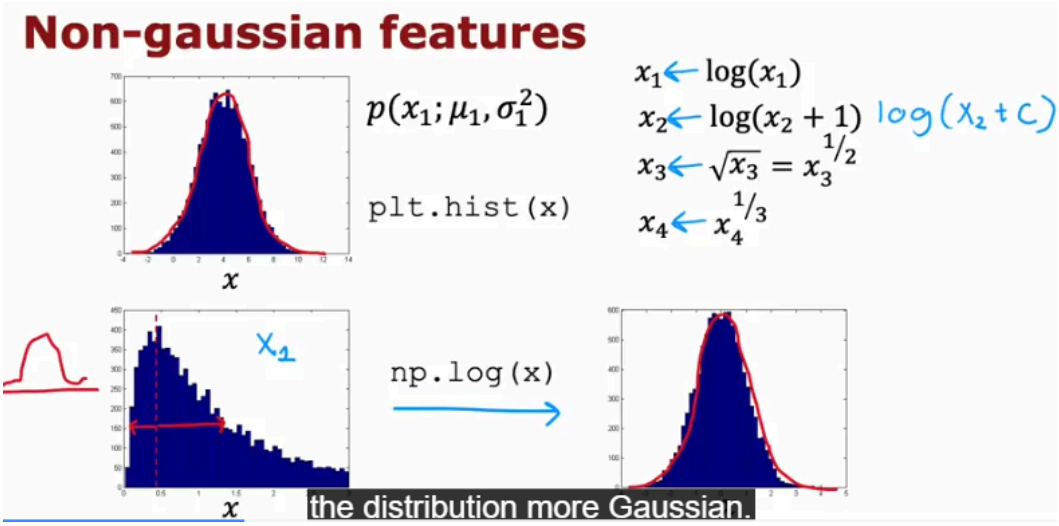
- In cybersecurity, new types of attacks emerge frequently, so anomaly detection is often used to identify abnormal patterns in machine behavior or potential hacks.

6. **Choosing Between the Two:**
   - **Anomaly Detection**: Finds novel anomalies that haven't been encountered before.
   - **Supervised Learning**: Predicts based on patterns from previous data.



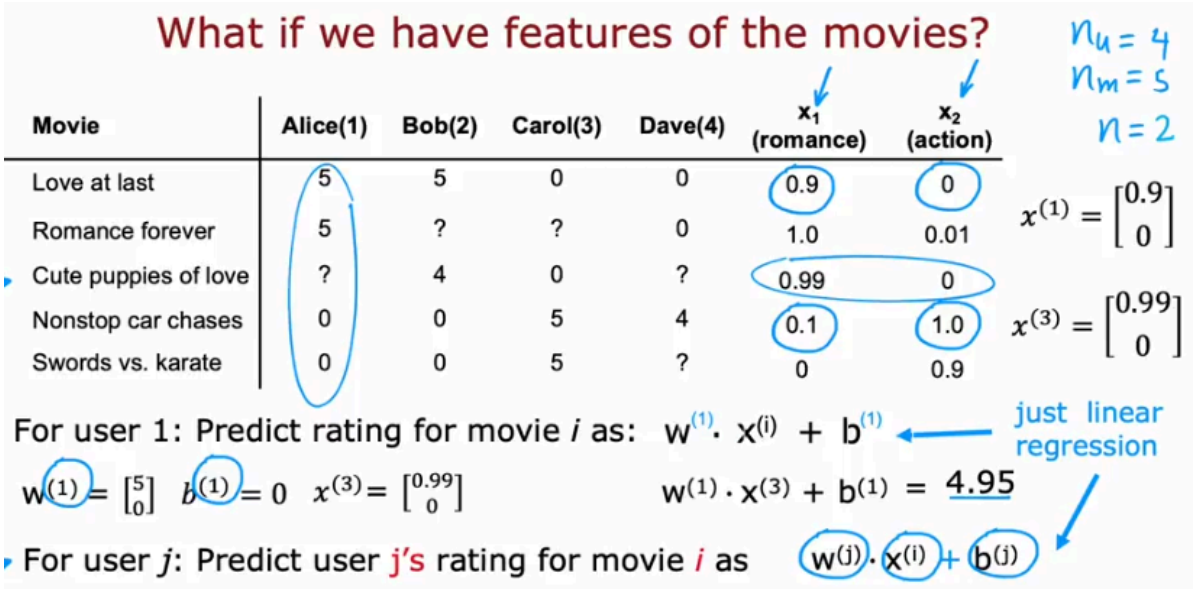
Choosing what features to use

1. **Feature Selection Matters More**
   - In anomaly detection (**unsupervised**), choosing the right features is critical since there are no labels to guide the model.
2. **Make Features Gaussian**
   - **Gaussian distribution** is ideal for anomaly detection algorithms.
   - Algorithms assume data is Gaussian. Use transformations to make feature distributions more bell-shaped:
     - **Log transformation** (e.g., log(X) or log(X + c))
     - **Square root transformation** (e.g., X^(1/2))
     - **Power transformation** (e.g., X^0.4)
3. **Plot & Transform**
   - Use histograms to visualize feature distributions.
   - Try different transformations until the distribution looks Gaussian.
4. **Error Analysis**
   - If the model fails on some anomalies, analyze those examples.
   - Add or create new features that highlight what makes them different.
5. **Create New Features**
   - Combine existing features (e.g., CPU / Network Traffic) to better capture anomalies.
6. **Apply Changes Consistently**
   - Use the same transformations on training, validation, and test sets.



Week - 2: Unsupervised Learning (Recomenders)

A **recommendation system** (or **recommender system**) is a type of software tool that suggests items to users based on various types of data, like their preferences, behavior, or past interactions.
Two types of recommender systems are **Collaborative** and **Content based** filtering.



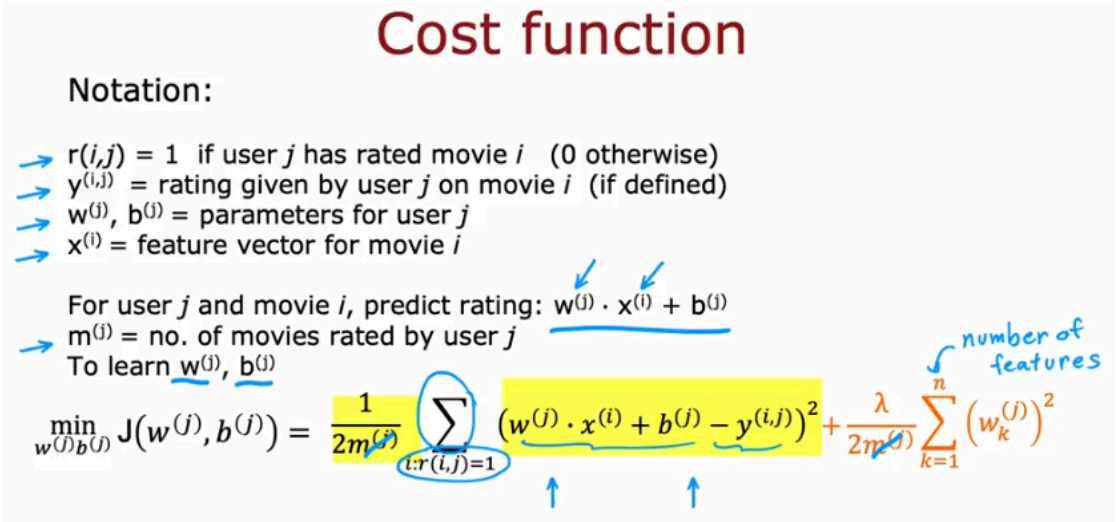
Basic Idea:

- We have **users** who rated **movies**, but not all movies are rated by every user.
- In addition to ratings, each movie also has **features** like:
    - **X1** = how romantic the movie is
    - **X2** = how action-packed the movie is

Example:

- "Love at Last" → X = [0.9, 0] (very romantic, not action)
- "Nonstop Car Chases" → X = [0.1, 1.0] (little romance, very action)

Cost Function:



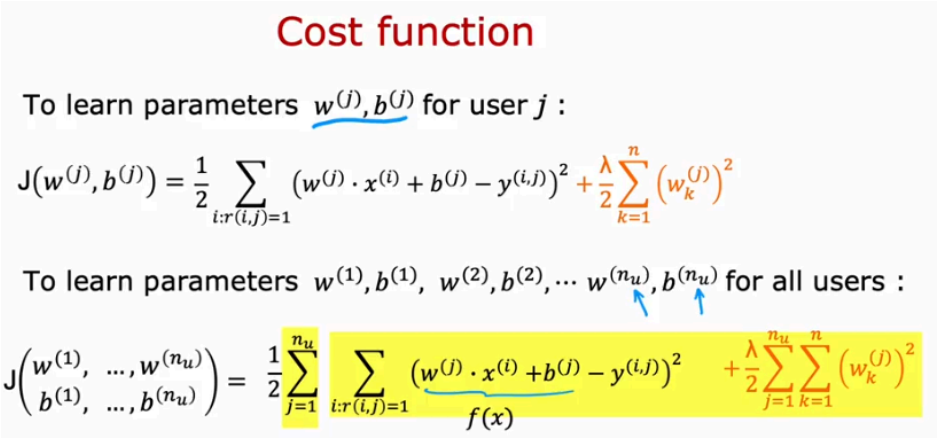
We want to find w(j) and b(j) that make our predictions as close as possible to the actual ratings.

So, we use a **Mean Squared Error (MSE)** loss:

Where:

- The sum is over all movies i rated by user j (i.e., where r(i,j) = 1)
- The second term is a **regularization** term to avoid overfitting
- λ is the regularization parameter

Now instead of training just for one user, do it for **all users**:



We can now minimize this total cost using **gradient descent** (or other optimization algorithms), and learn the best w(j) and b(j) for every user.

Collaborative Filtering (Learning Features Automatically)

What We Already Know:

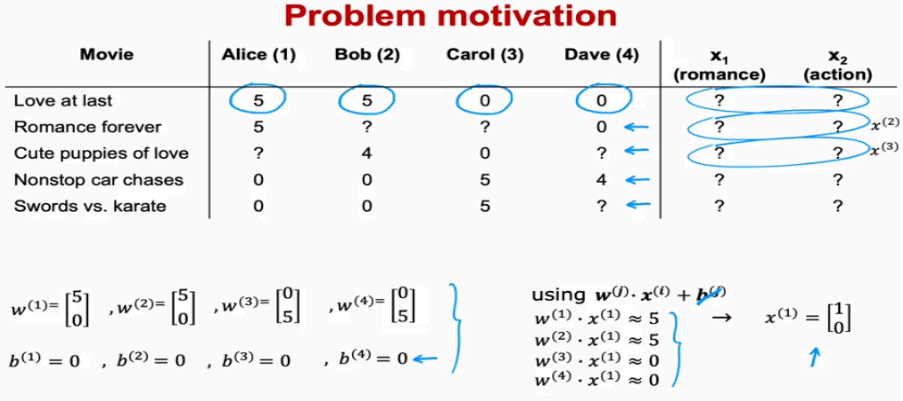
- Previously, movie features (e.g., romance, action) were **manually defined**.
- User preferences (w) + movie features (x) + bias (b) → predicted rating.

New Problem: What if Features (x) Are Unknown?

- We don't know movie features in advance.
- But we **assume we already know** user preferences (w) and biases (b).

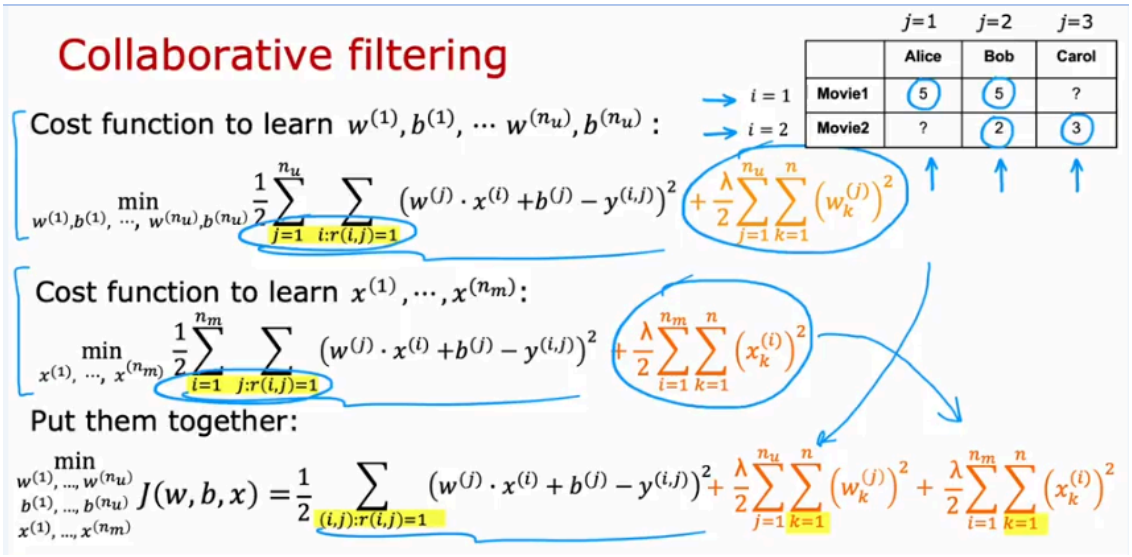
Idea: Learn Movie Features (x) from Ratings

- Use observed ratings to **guess** feature values (x) that match user preferences.
- For each movie, minimize the **squared error** between predicted and actual ratings.



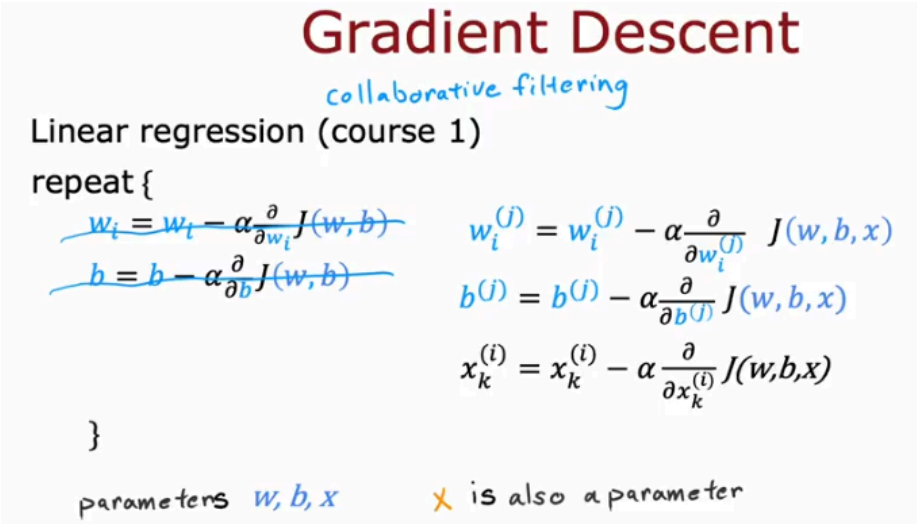
Learning Both Users and Movies

- In practice, we **don't know** either w, b, or x.
- So, we **jointly learn**:
  - Movie features x^i
  - User preferences w^j
  - User bias b^j



Optimization: Use Gradient Descent

- Update w, b, and x iteratively to minimize the cost.
- Learn everything from the data — no hand-crafted features needed.



Key Insight:

- Ratings from **multiple users** of the same movie allow us to **collaboratively** infer features.
- This is why it's called **Collaborative Filtering**.

The main goal is to generalize the collaborative filtering algorithm, typically used for predicting ratings, to work with binary labels, which are simpler and reflect user engagement more directly.

Here's a breakdown of the process:

1. **Binary Labels**:
   - **1**: User liked or engaged with the item (e.g., watched a movie, clicked on a product).
   - **0**: User did not engage with the item (e.g., skipped a movie, ignored a product).
   - **?**: User has not yet been exposed to the item.
2. **Logistic Regression**:
   - The model predicts the probability of user engagement using the logistic function:

$$P(y_{ij} = 1) = g(w_j \cdot x_i + b)$$

   - This transforms the prediction from a continuous value to a probability between 0 and 1.
3. **Cost Function**:
   - The binary cross-entropy loss function is used:

$$\text{Loss} = -y \log(f(x)) - (1 - y) \log(1 - f(x))$$

   - The cost function is summed over all user-item pairs where ratings exist.

Adapting the Algorithm:

- Previously, the algorithm used a linear regression model where the prediction was in the form $y_{ij} = w_j \cdot x_i + b$. This would predict a numerical rating.
- To work with binary labels, we shift to **logistic regression**, which predicts probabilities. The formula becomes:

$$P(y_{ij} = 1) = g(w_j \cdot x_i + b)$$

  where $g(z) = \frac{1}{1+e^{-z}}$ is the **logistic function** (also known as the sigmoid function).
- This transforms the prediction from a continuous value to a probability between 0 and 1, indicating the likelihood that a user will engage with the item.

5. **Application in Various Domains**:
   - **E-commerce**: Whether a user purchases a product after being shown it (binary 1 or 0).
   - **Social Media**: Whether a user likes or favorites a post after being shown it.
   - **Advertising**: Whether a user clicks on an ad after seeing it.
6. **Generalization**:
   - By using the logistic function and binary cross-entropy loss, we generalize collaborative filtering to handle binary interactions, which is suitable for many real-world applications like recommending products, movies, or ads based on user engagement.

Mean normalization in collaborative filtering

Problem Without Mean Normalization

- Collaborative filtering uses parameters w and b for each user.
- For a new user (like Eve) who hasn't rated any movies:
    - Their parameters will be initialized as w = [0, 0] and b = 0.
    - The model predicts all ratings as 0, which is unrealistic.

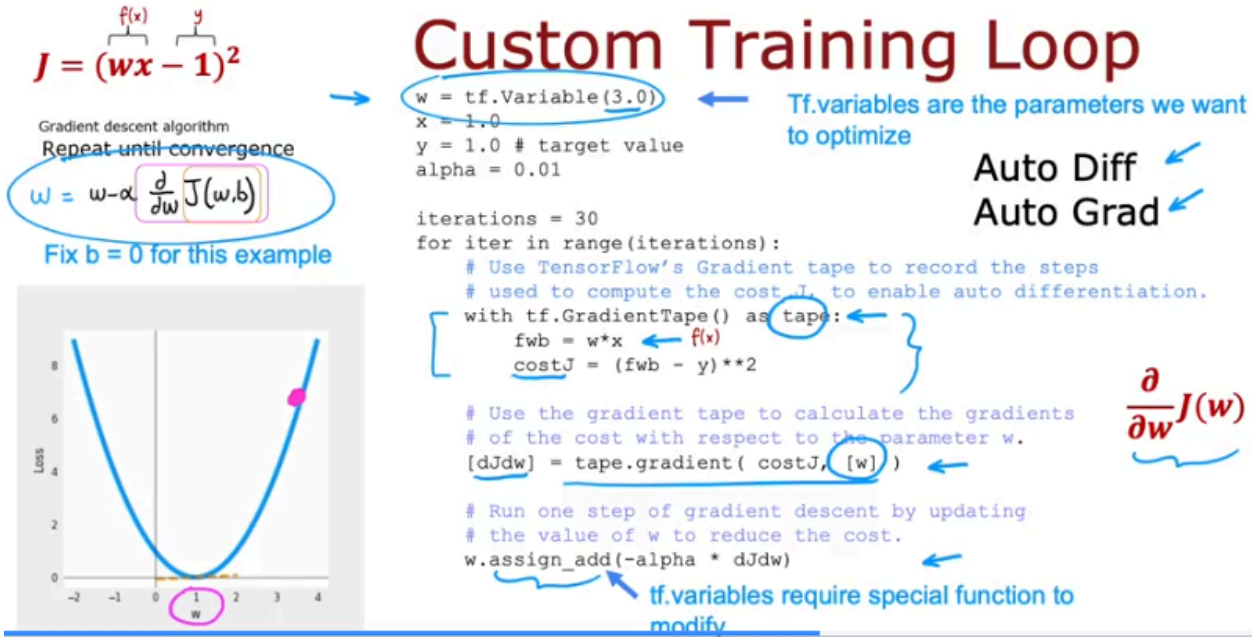
What is Mean Normalization?

- For each **movie**, calculate its **average rating** (e.g., 2.5 stars).
- Subtract this average (mean) from each user's rating for that movie.
- This shifts all movie ratings to have a mean of **0**.
- These adjusted ratings are used for training the model.

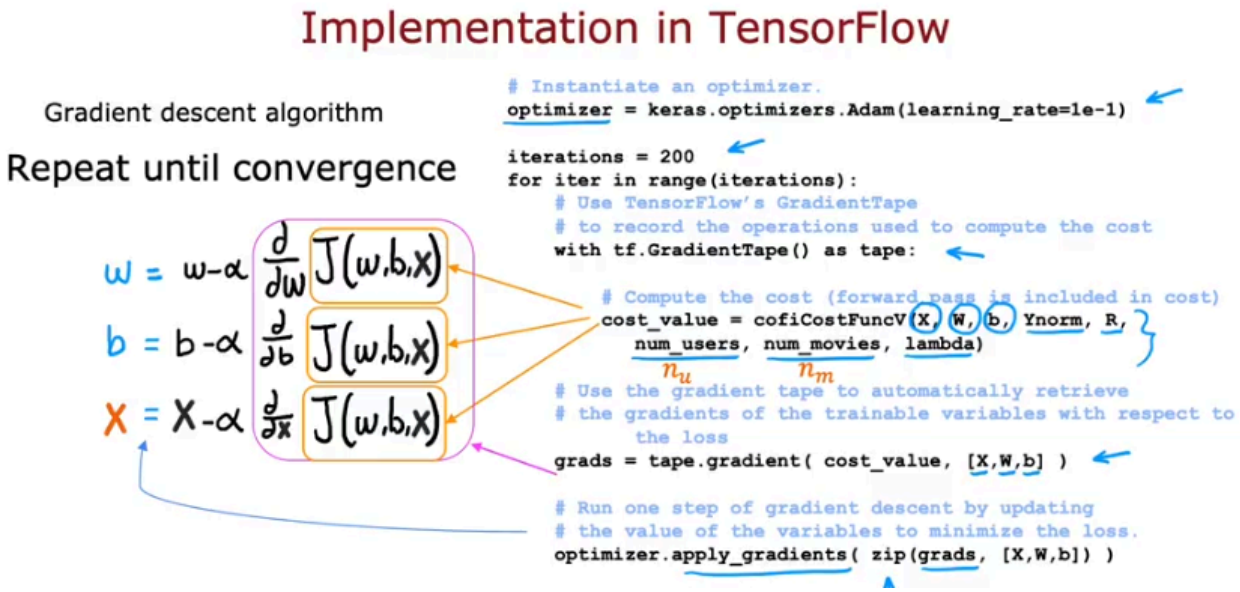
Prediction with Mean Normalization

- During training: Model predicts w(j) = x(i) + b(j) using normalized ratings.

Tensorflow Implementation of Collaborative Filtering:



TensorFlow does all the calculus work for us. We just need to write the cost function!
The code is for gradient descent that calculates derivatives automatically, and updates weights.
Using this type of code, we can also implement **collaborative filtering** using tensorflow.



Also we can update **'b'** and **'x'** using **Auto Diff.**
TensorFlow's model.compile() and model.fit() work well for **standard neural networks**, like dense layers.
But **collaborative** filtering needs **custom cost functions** that don't fit well into those standard layers.

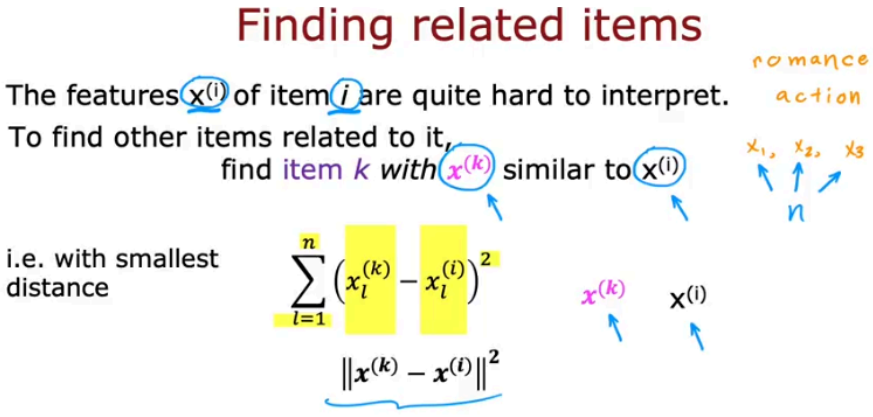
Finding Related items:

If we want to find movies similar to a given movie i, we need to follow the steps:

- Take the **feature vector** x^(i) of movie i.
- Compute the **distance** between x^(i) and the feature vectors of all other movies x^(k) using this formula:

$$\text{Distance} = \sum_{l=1}^{n} (x_l^{(k)} - x_l^{(i)})^2 = \|x^{(k)} - x^{(i)}\|^2$$

- Find the movies with the **smallest distances** — they're the **most similar**.



:

Limitations:

Cold Start Problem: Collaborative filtering works well when we already have ratings. But:

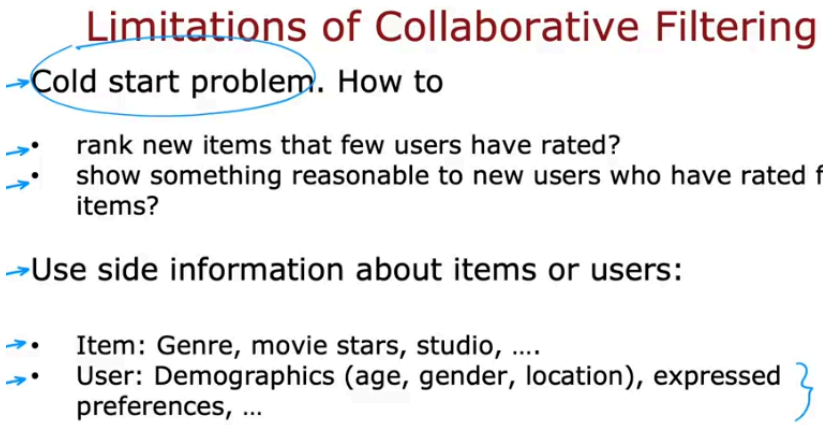
- **New item** : Nobody rated it yet → model doesn't know where to place it.
- **New user**: They rated only a few movies → model doesn't know their preferences well.

This is called the **cold start problem**.

Another Limitation: Can't Use Side Information Easily:
Collaborative filtering mostly depends on **user-item ratings**. But sometimes we have **extra info**:

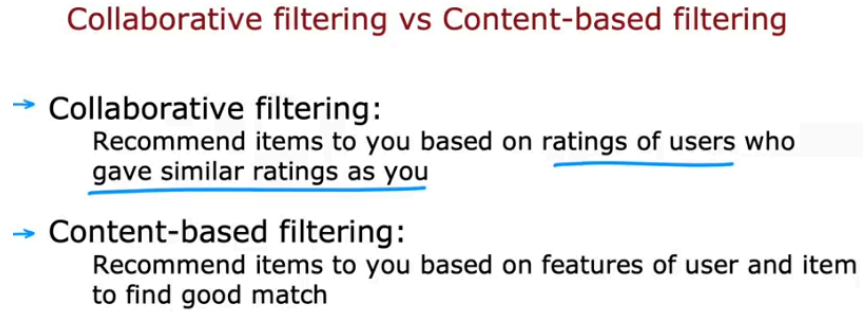
- For movies: genre, director, actors, release date
- For users: age, location, device/browser used, etc.

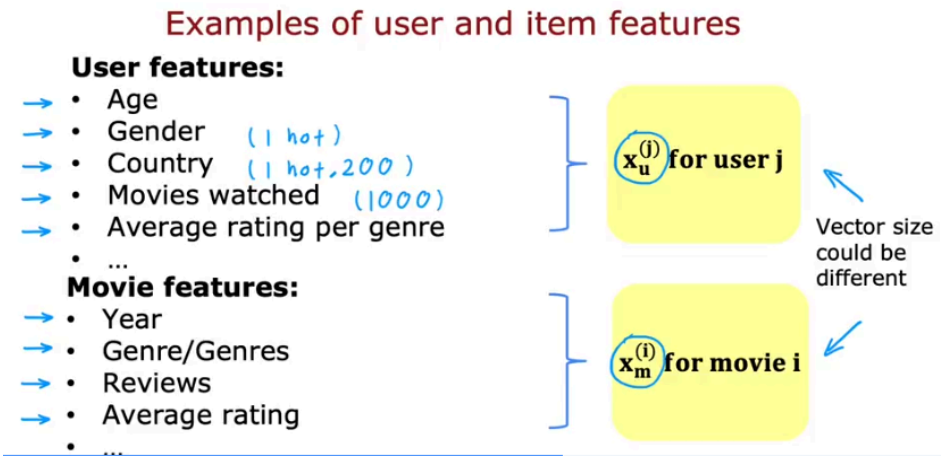
These could be **very useful** in making better recommendations! But collaborative filtering doesn't naturally incorporate that info.

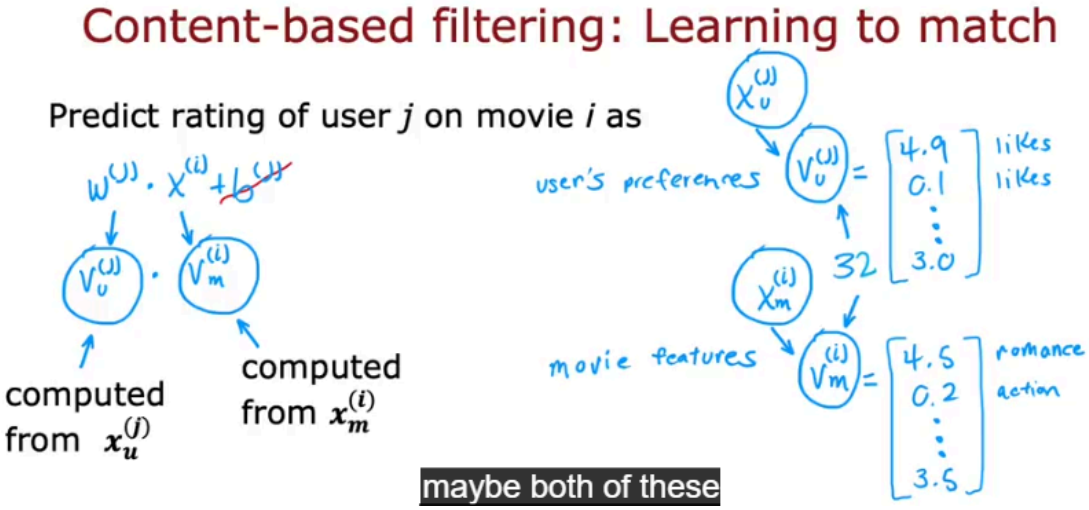


Content Based Filtering:

Content-Based Filtering recommends items by comparing the **content (features or attributes**) of **items** with a **user's past preferences**.

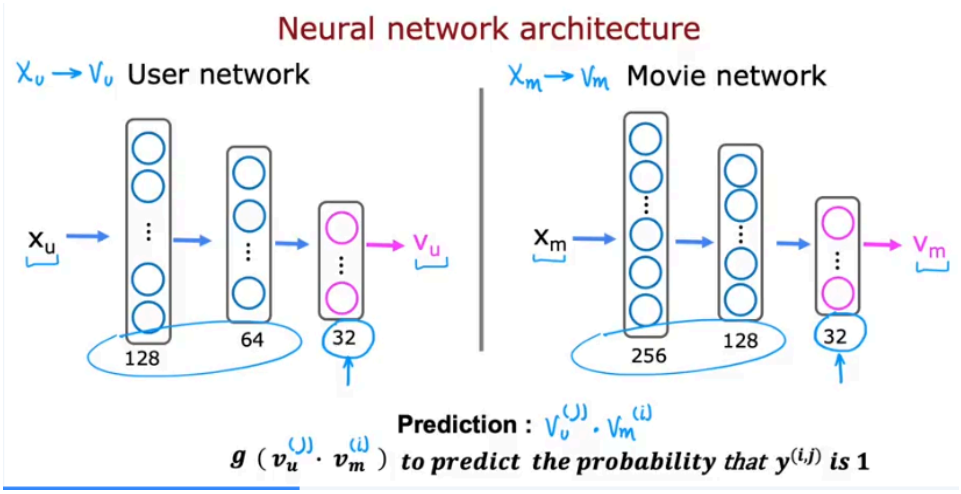




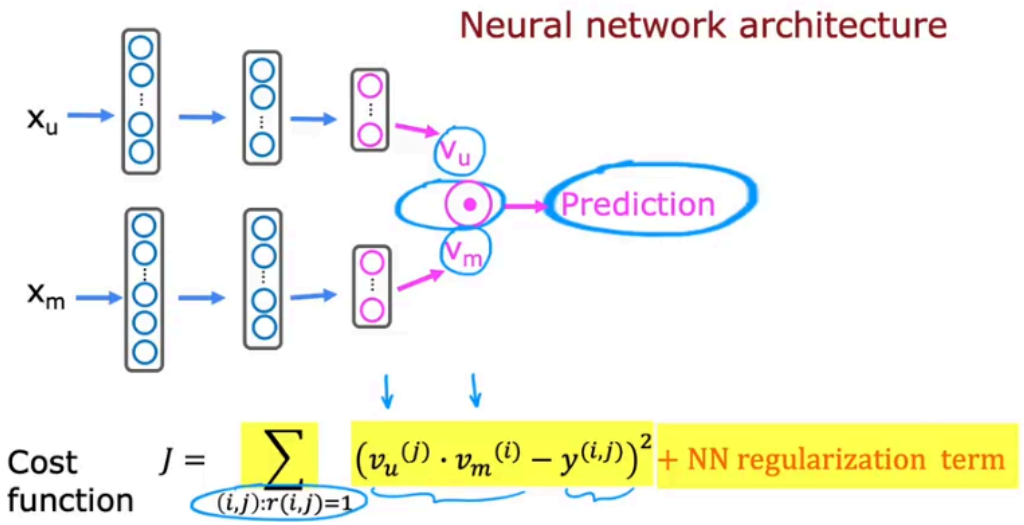


Let, **Xu** is user features and **Xm** is movie (item) features, then we need to calculate **Vu** (user vector) and **Vm** (movie vector) from user and movie features. Feature size may vary, but feature vector size must be the same.

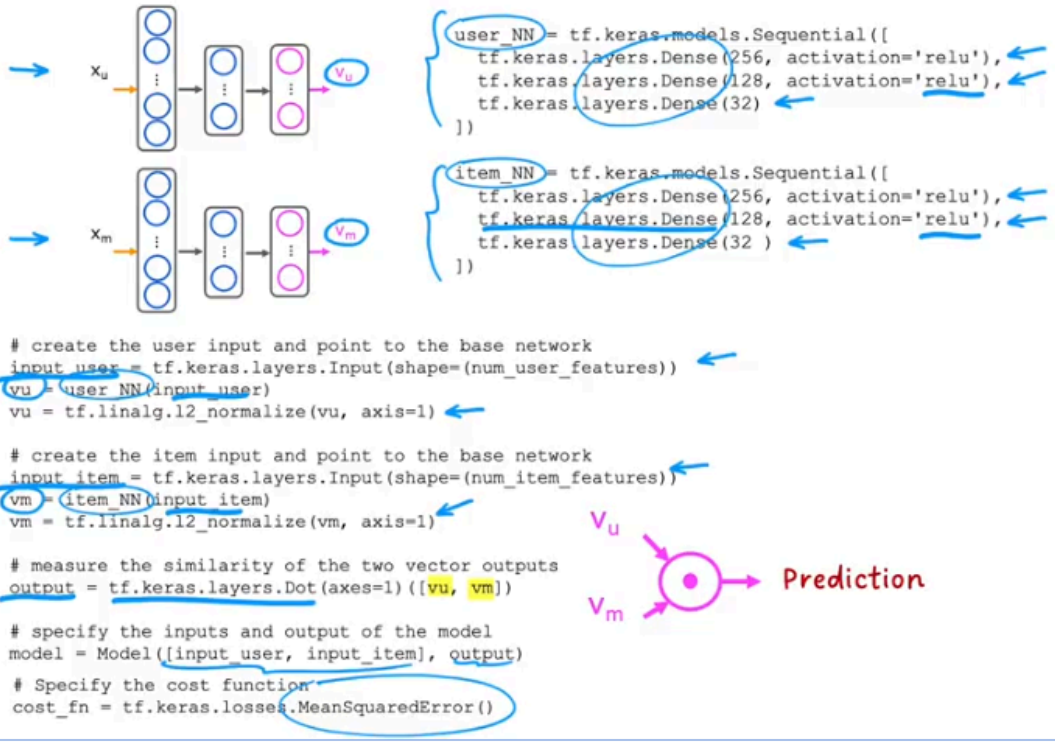
We can remove **bj,** as it won't affect the performance of content based filtering.



Using **two neural** networks, we can calculate **Vu** and **Vm**, then using dot products, we can calculate the movie rating. Also using **sigmoid** function we can get the probability of movie rating.
Also we can update the model minimising the following cost function.



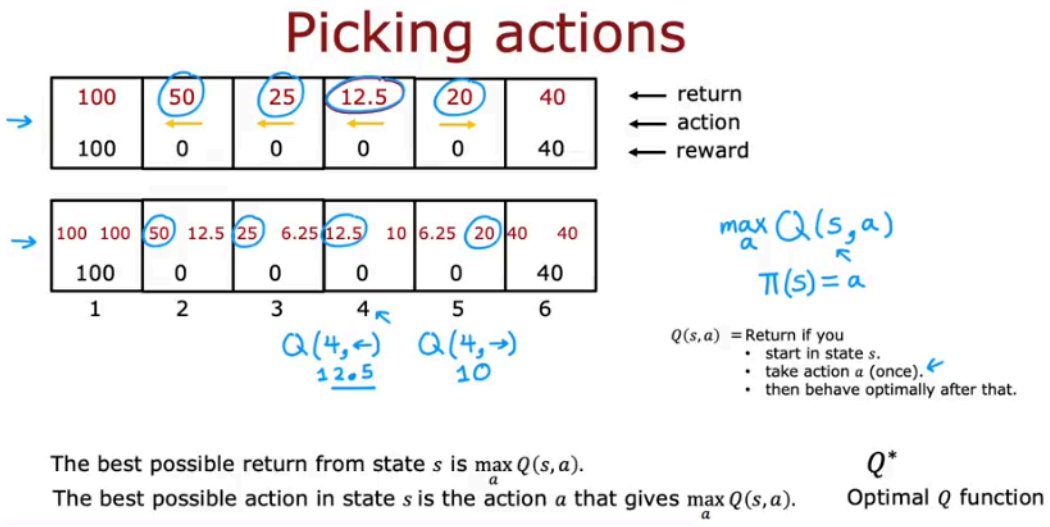
Tensorflow implementation of Content based filtering:



ssue

Week - 03: Reinforcement Learning:

Reinforcement Learning (RL) is a type of machine learning where an **agent learns how to behave in an environment** by performing actions and receiving feedback in the form of **rewards or penalties**.

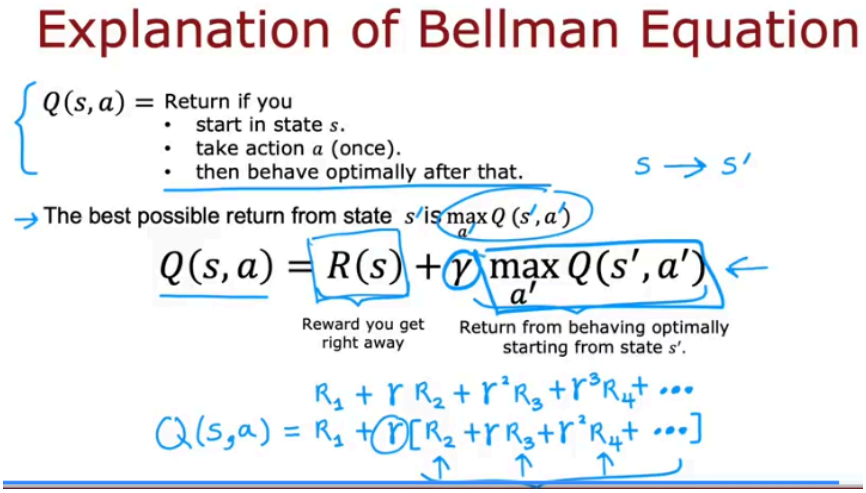


Q(s, a) is the **expected total return** if you start in state **s**, take action **a**, and then **follow the optimal policy π** from the next step onward.
The **Q-function** is defined as:

$$Q(s,a) = \mathbb{E}\left[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + \cdots \middle| s_t = s, a_t = a, \pi^*\right]$$

If **γ = 0**, the agent only cares about **immediate rewards**.
If **γ is close to 1**, the agent cares about **long-term rewards** more.

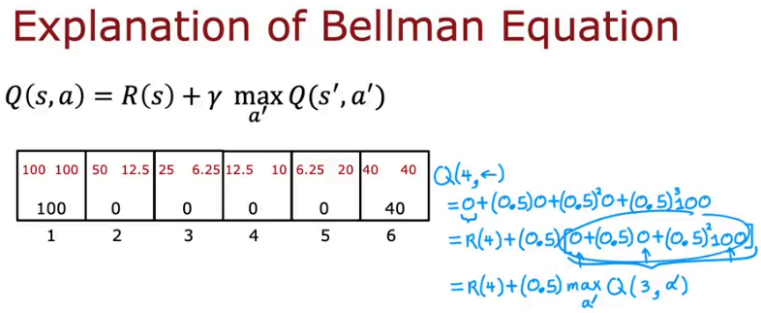
Bellman Equation:



In this equation:

- **R(s)**: Reward from current state.
- **γ (gamma)**: Discount factor (e.g., 0.5).
- **s′**: Next state after taking action **a** from **s**.
- **a′**: Next possible action.

Example of bellman equation (return from state 4 and action left**):**



Stochastic Reinforcement Learning (RL) Summary

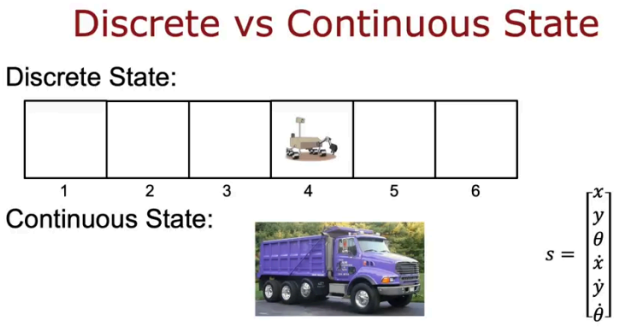
In real-world applications like commanding a Mars rover, actions often have uncertain outcomes due to unpredictable factors like slipping, wind, or terrain. This randomness makes the environment **stochastic** rather than deterministic.

Rewards can differ each time due to randomness, so the **expected return** is the average over many trials of the same policy.

$$\text{Expected Return} = \text{Average}(R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots)$$
$$= \mathrm{E}[R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots]$$

Discrete Vs Continuous State

- A **discrete state** space has a limited number of possible states.
- A **continuous state** space allows the state to be any value within a range — **infinitely many possibilities**.



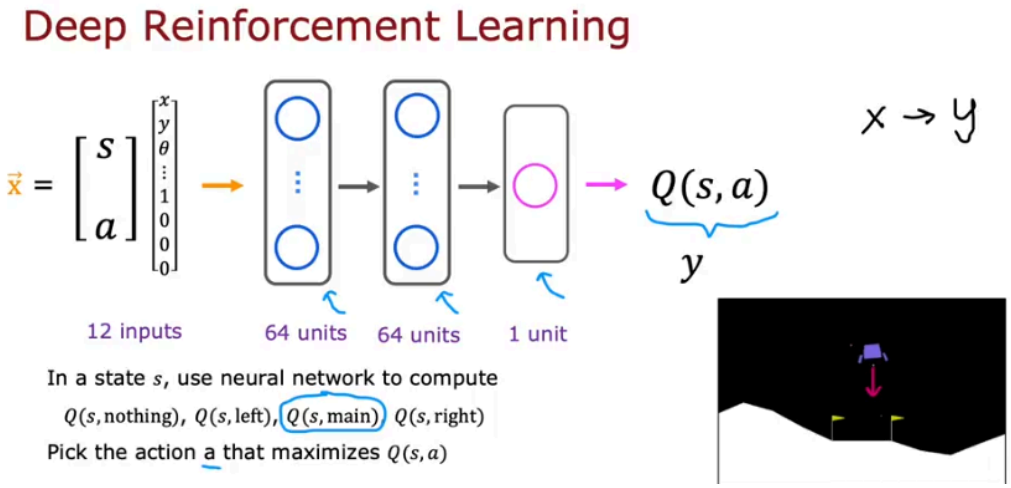
States for Self-driving Car or Truck (These are continuous values**)**

- Position: x,y (left-right or front-back)
- Orientation: θ (Rotation clockwise or anticlockwise)
- Velocity: x', y',
- Angular velocity: θ'

States for Autonomous Helicopter (These are continuous values**)**

- **Position:** x, y, z
- **Orientation (Rotation):** Roll (ϕ), Pitch (θ), Yaw (ω)
- **Velocity:** $x^{\cdot}$,$y^{\cdot}$,$z^{\cdot}$
- **Angular velocity:** $\phi^{\cdot}$,$\theta^{\cdot}$,$\omega^{\cdot}$
- Total: **12 continuous values**

DQN (Deep Q-Network)(Lunar Lander)



Neural Network Input & Output

- **Input x** = [state (8 values) + one-hot action (4 values)] → total 12 values.
- **Output y** = approximated Q(s, a) (a single scalar).

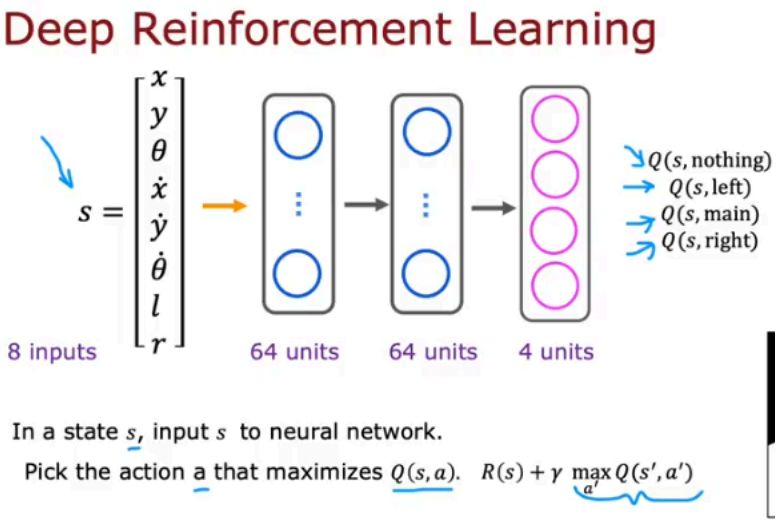
Creating training set:

Input x: concatenate s and one-hot a.

Target y:

$$y = r + \gamma \cdot a' \max Q(s', a')$$

Although it looks like supervised learning, this is **reinforcement learning**, because the targets y are computed using the **Bellman equation.**



In the previous network we needed to inference Four times for Q(s,a). We can improve it by following a network, where 4 neurons are in the output layer, so Inference is required one time for four values.

How should an agent pick actions while it's still learning:

Option 1: Always Pick the Best According to Current Q(s, a)

Option 2: **Epsilon-Greedy Policy(ε)**

Most of the time (say 95%): pick the action with the highest Q(s, a) → this is called **greedy** or **exploitation**.

Some of the time (say 5%): pick a random action → this is called **exploration**.