Knowledge Transfer Session - 2

AI
CLUB

K-Nearest Neighbors

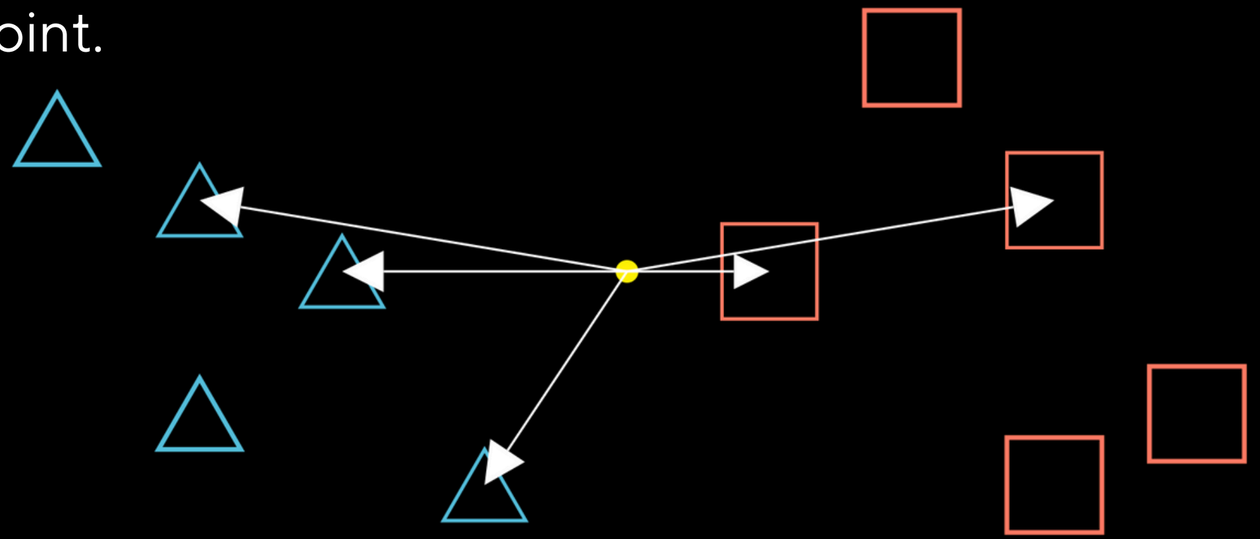
“Birds of a feather flock together”

- K-Nearest Neighbors (KNN) is a simple, instance-based learning algorithm used for classification or regression.
- It works by finding the k closest data points (neighbors) to a given input using a distance metric like Euclidean distance.
- The algorithm then assigns the most common label (in classification) or averages the values (in regression) among these neighbors.
- It’s like asking your neighbors how they dressed for today’s weather to decide what you should wear.

Working

- The KNN classifier operates by finding the K nearest neighbors to a new data point and then voting on the most common class among these neighbors.
- Here’s how it works:

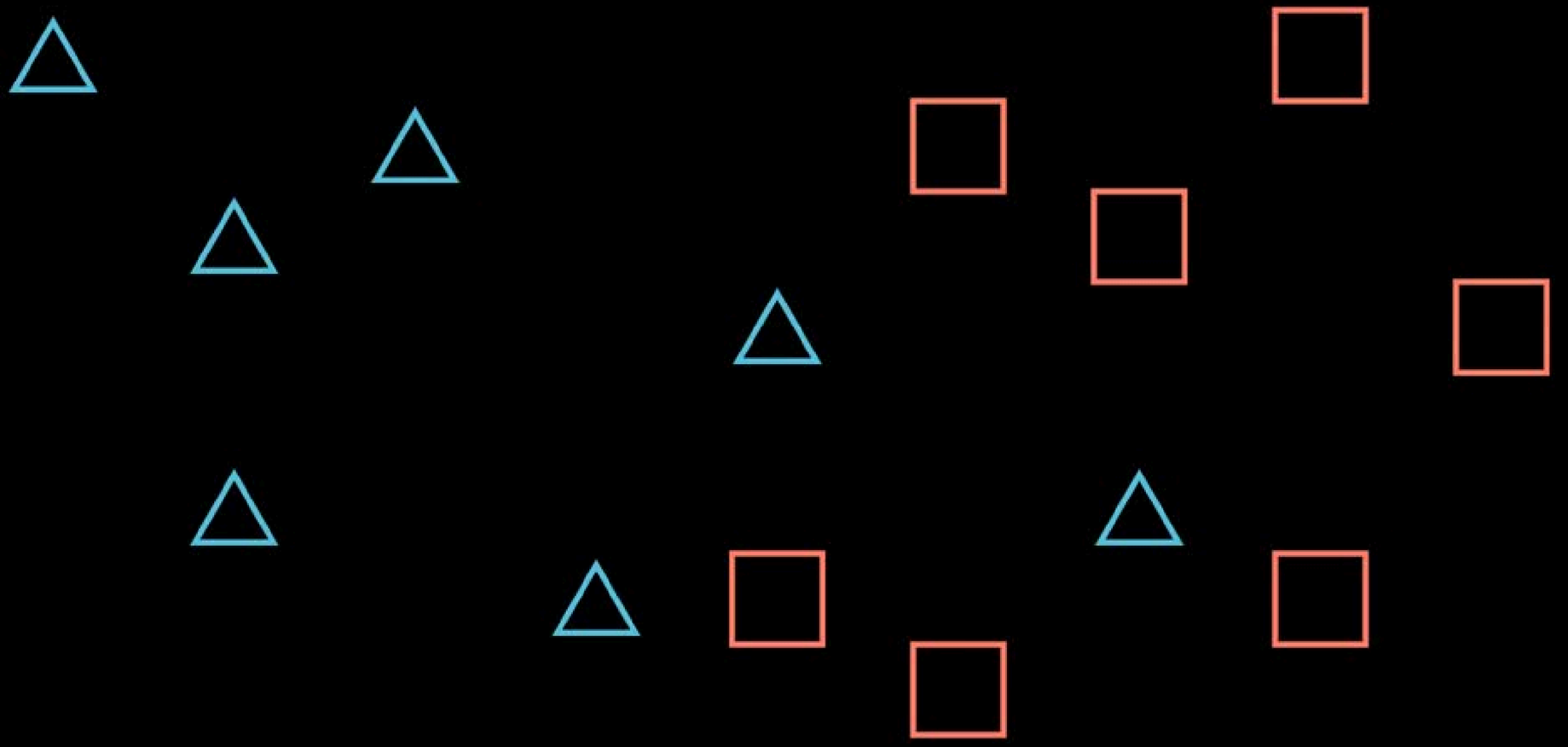
1. Calculate the distance between the new data point and all points in the training set.
2. Select the K nearest neighbors based on these distances.
3. Take a majority vote of the classes of these K neighbors.
4. Assign the majority class to the new data point.



Training Steps

- Unlike many other algorithms, KNN doesn’t have a distinct training phase. Instead, it memorizes the entire training dataset.
- Here’s the process:

1. Choose a value for K (the number of neighbors to consider).
2. Select a distance metric (e.g., Euclidean distance, Manhattan distance).
3. Store/Memorize all the training data points and their corresponding labels



No Training, Just Comparing

- What’s unique about KNN is it’s a ‘lazy’ algorithm, meaning it doesn’t try to learn a general pattern from the training data.
- It just stores the data and uses it directly to make predictions.
- It’s all about finding the nearest neighbors based on how you define ‘closeness,’ which depends on the distance method you use and the value of ‘k’ you set.

Key Parameters

- While KNN is conceptually simple, it does have a few important parameters:

1. **K:** The number of neighbors to consider. A smaller K can lead to noise-sensitive results, while a larger K may smooth out the decision boundary.
2. **Distance Metric:** This determines how similarity between points is calculated. Common options include: Euclidean, Manhattan and Minkowski distance.
3. **Weight Function:** This decides how to weight the contribution of each neighbor. Options include: ‘uniform’ (all neighbors are weighted equally) and ‘distance’ (closer neighbors have a greater influence than those farther away).

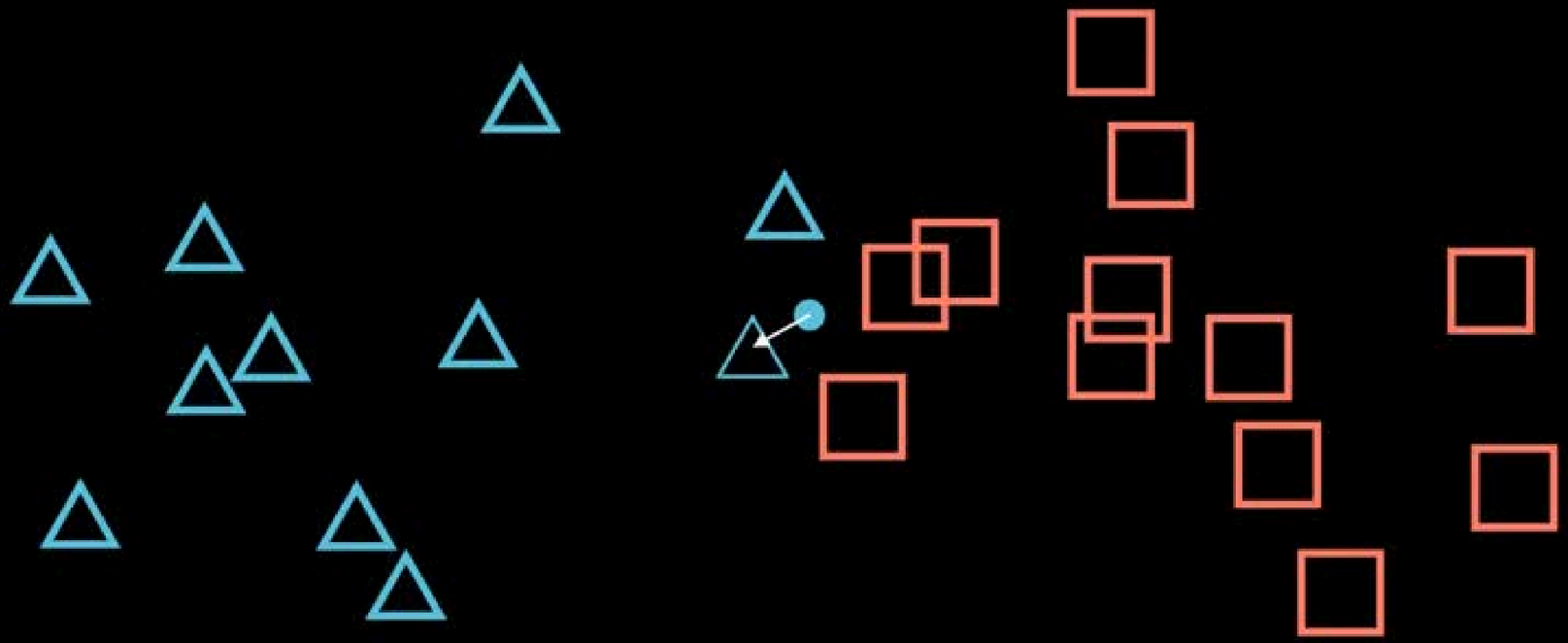
Choosing the Right ‘K’

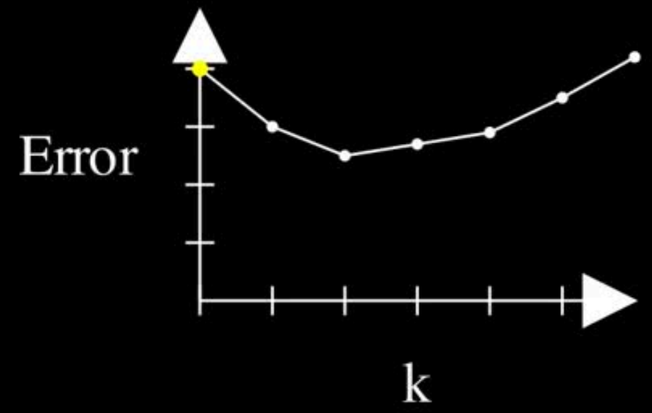
- If ‘K’ is too small, the model becomes overly sensitive to noise and outliers. It memorizes the training data, leading to overfitting.
- If ‘K’ is too large, the model smooths over important patterns and local structure. It generalizes too much, leading to underfitting.

Choosing the right ‘K’

- Use cross-validation**:** Run KNN with different K values (e.g., 1 to 20), and for each one, compute validation accuracy (or error). Plot it. Choose the K with the best score.
- Stick to odd values to avoid ties (in case of binary classification).
- Avoid overfitting to the validation set: Don’t blindly pick the K with the absolute best score if nearby Ks perform similarly, go with the simpler (usually higher) K.
- Beyond a certain point, increasing k just blurs the model. You rarely need k > sqrt(n) where n is your dataset size.

Error vs ‘K’





k = 1

Choosing the Distance Metric

1. **Euclidean Distance** (L2 norm) is the most common and works well when all features are continuous and equally scaled but it's sensitive to outliers and irrelevant features.
2. **Manhattan Distance** (L1 norm) sums absolute differences and is better when the data has high dimensionality or is sparse. It is less sensitive to outliers than Euclidean.
3. **Minkowski Distance** generalizes L1 and L2 with a parameter p; useful when tuning distance behavior but harder to interpret intuitively.
4. **Hamming Distance** is used for categorical or binary features — best when the data is non-numeric and has fixed-length vectors.

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} |x_i - y_i|$$

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} \mathbb{I}(x_i \neq y_i)$$

Regression using KNNs?

- KNN regression predicts the output for a data point by averaging the target values of its k nearest neighbors.
- It’s a non-parametric method, meaning it doesn’t assume any underlying relationship between input and output.
- While simple and effective for small datasets, KNN regression can struggle with high-dimensional data or noisy neighbors.

Pros and Cons of KNNs

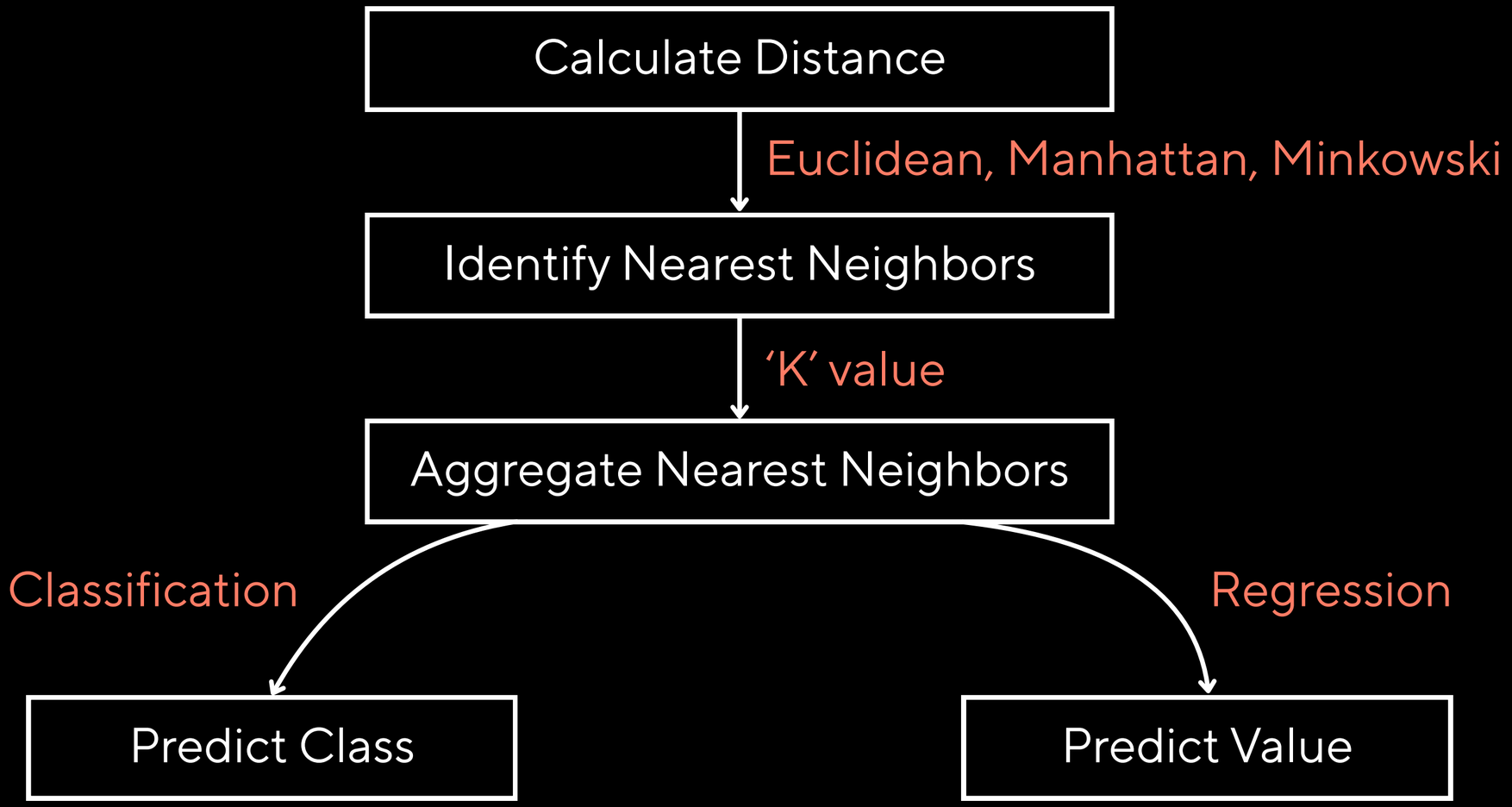
Pros

1. **Simplicity:** Easy to understand and implement.
2. **No Assumptions**: Doesn't assume anything about the data distribution.
3. **Versatility:** Can be used for both classification and regression tasks.
4. **No Training Phase:** Can quickly incorporate new data without retraining.

Cons

1. **Computationally Expensive:** Needs to compute distances to all training samples for each prediction.
2. **Memory Intensive:** Requires storing all training data.
3. **Sensitive to Irrelevant Features:** Can be thrown off by features that aren't important to the classification.
4. **Curse of Dimensionality:** Performance degrades in high-dimensional spaces.

Putting It All Together



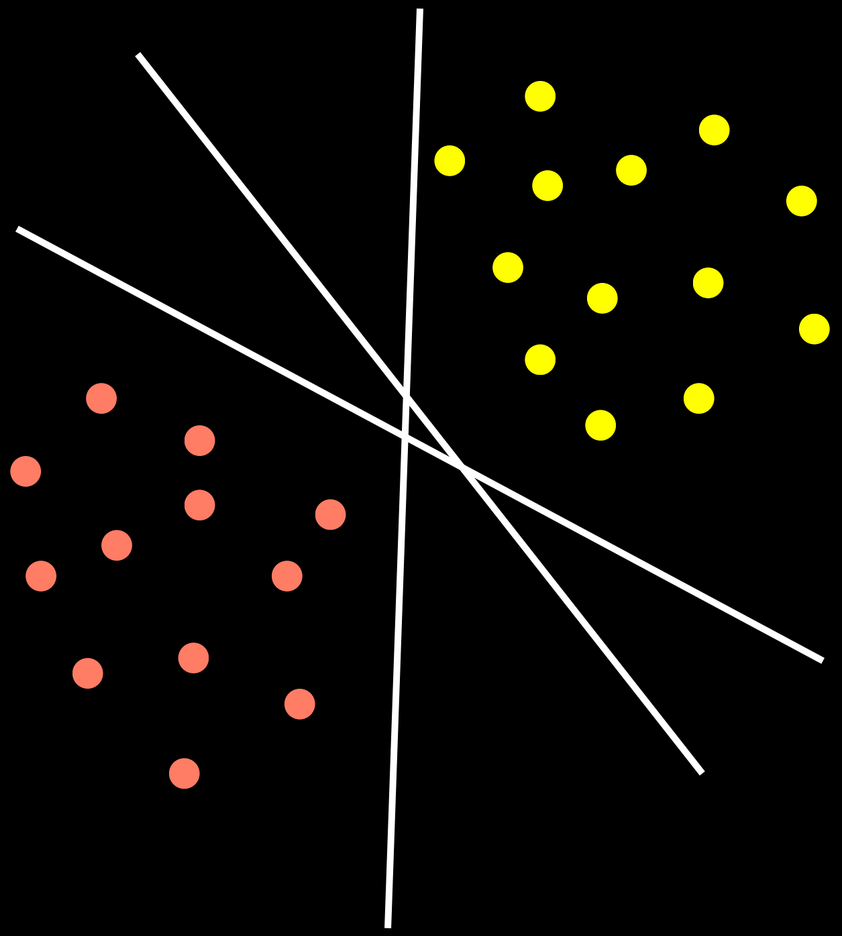
Code Implementation

Support Vector Machines

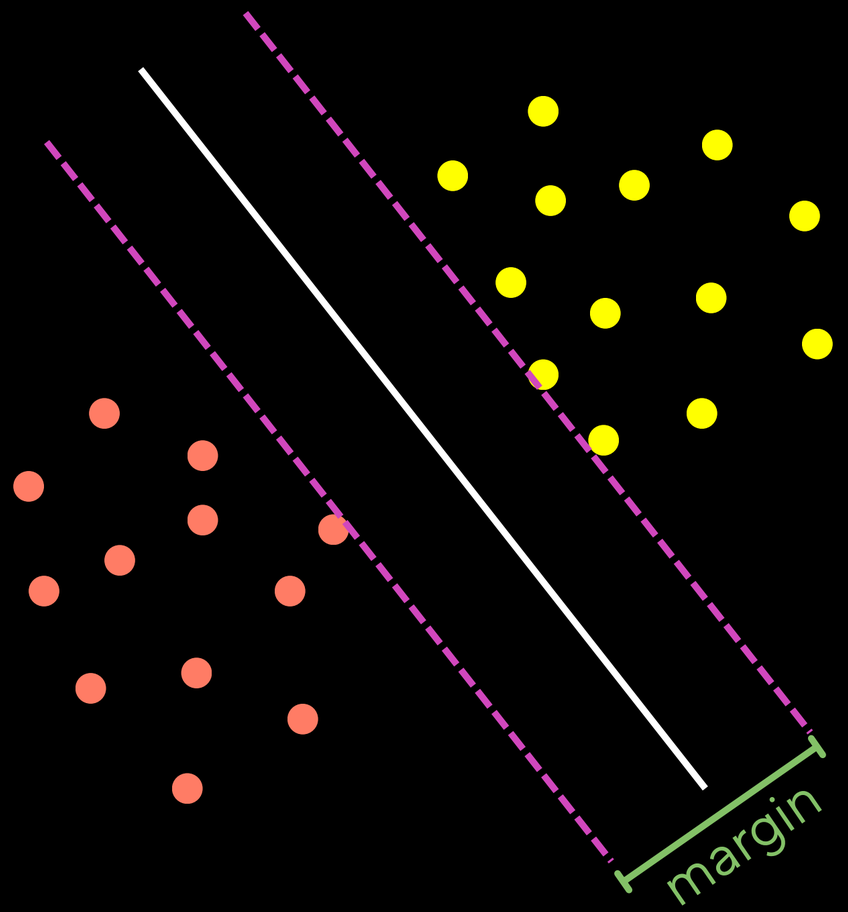
What is an SVM?

- Support Vector Machines (SVMs) are supervised classifiers that try to find the best possible decision boundary between two classes.
- Instead of just separating the classes, SVMs look for the boundary that maximizes the margin (the distance from the closest points on either side).
- These critical points are called support vectors, and they’re the only ones that influence the decision.
- SVMs can also handle nonlinear boundaries using the kernel trick, allowing them to separate even complex data.

Why Just Separating isn't Enough



What Makes a Good Boundary?



What Makes a Good Boundary?

- A good boundary isn’t just one that separates the classes, it’s one that separates them with confidence.
- SVMs aim to find the boundary that is as far away as possible from the nearest points on either side. This distance is called the margin.
- A wider margin means better robustness to noise and better generalization to unseen data.
- Points that lie exactly on the margin are called support vectors, they define the boundary.

The Hyperplane

- A hyperplane is just a flat decision boundary — a line in 2D, a plane in 3D, etc.
- It's equation is given by:

$$\vec{w} \cdot \vec{x} + b = 0$$

where,

w : normal vector (perpendicular to the hyperplane)

x : input vector

b : bias/offset

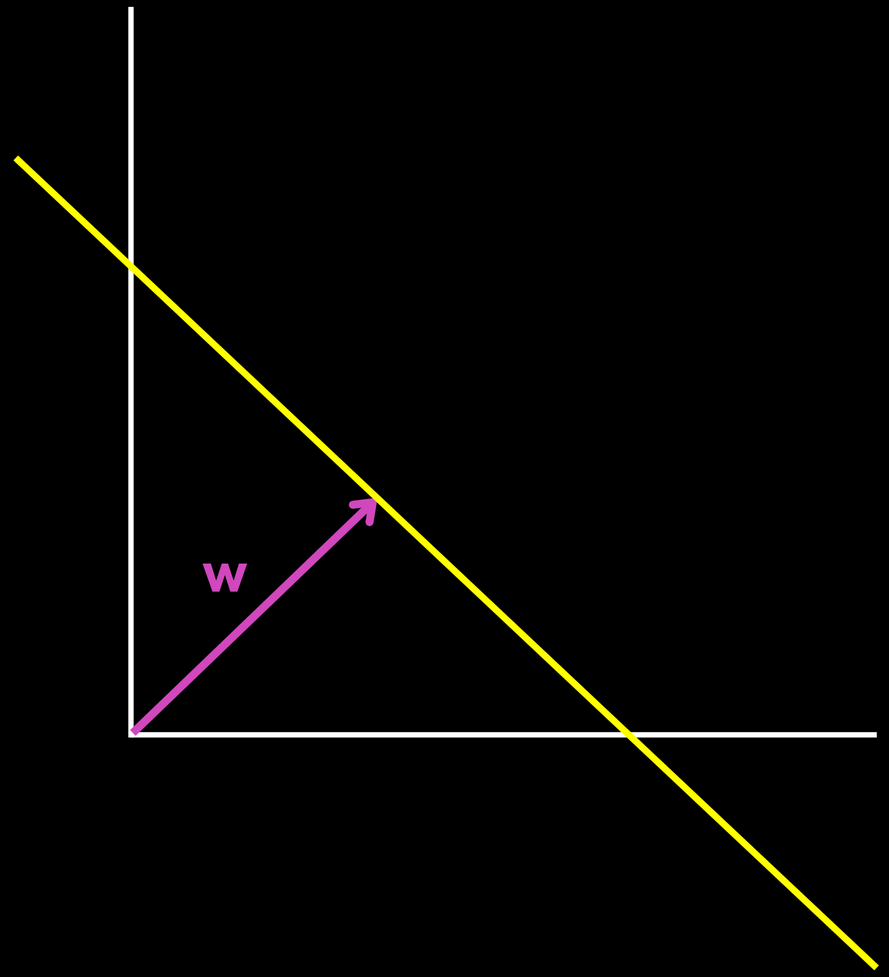
Hyperplane in 2D

- A hyperplane in 2D is just a line.

$$x = \begin{bmatrix} x \\ y \end{bmatrix}, \quad w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}, \quad b \in \mathbb{R}$$

Hyperplane equation:

$$w^T x + b = 0 \Rightarrow w_1 x + w_2 y + b = 0$$



The Math Behind SVMs

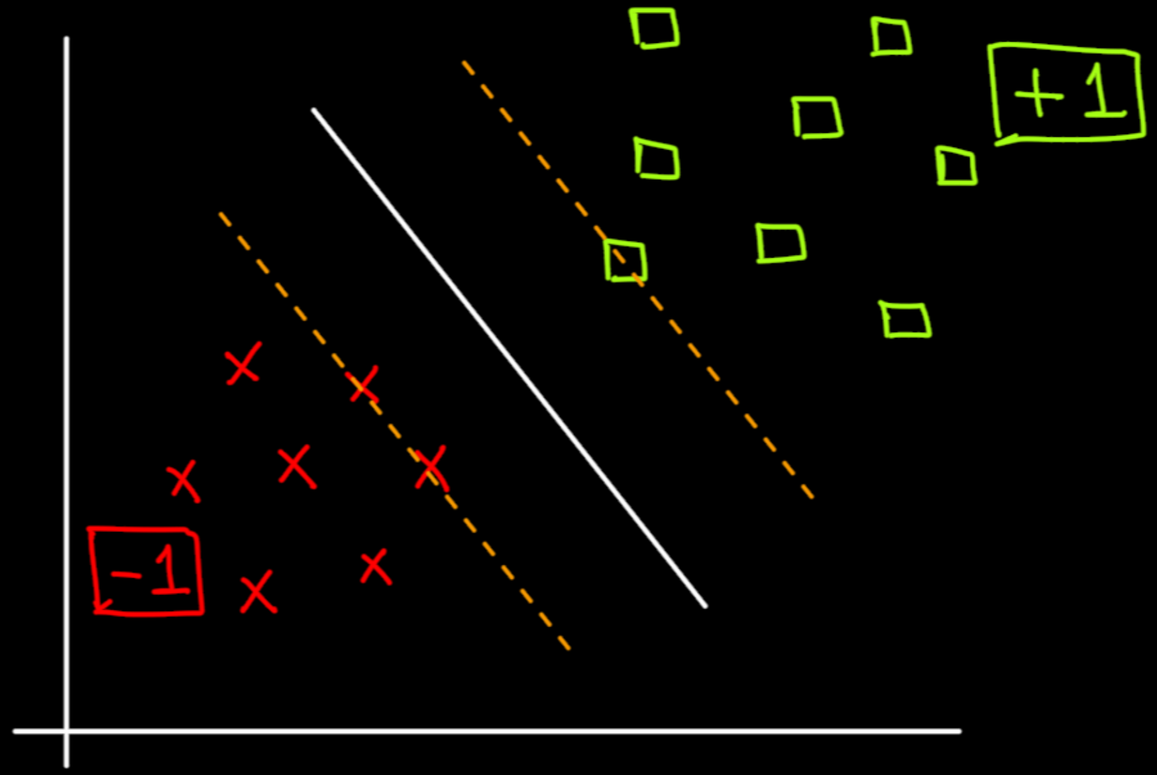
Note:

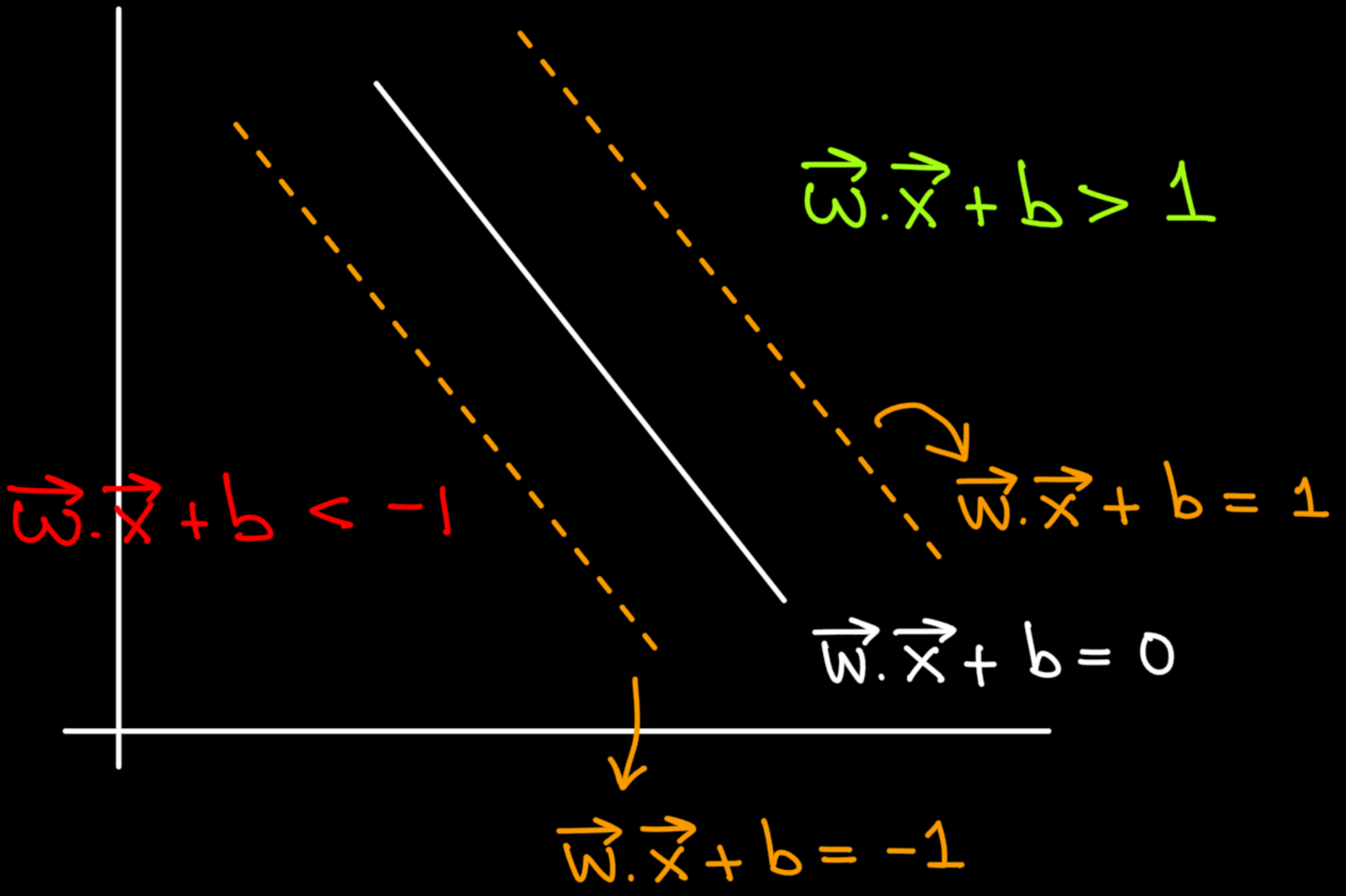
I'm gonna call $w \cdot x_i + b$ the "score" so that you don't have to listen to me say the whole expression 769 times.

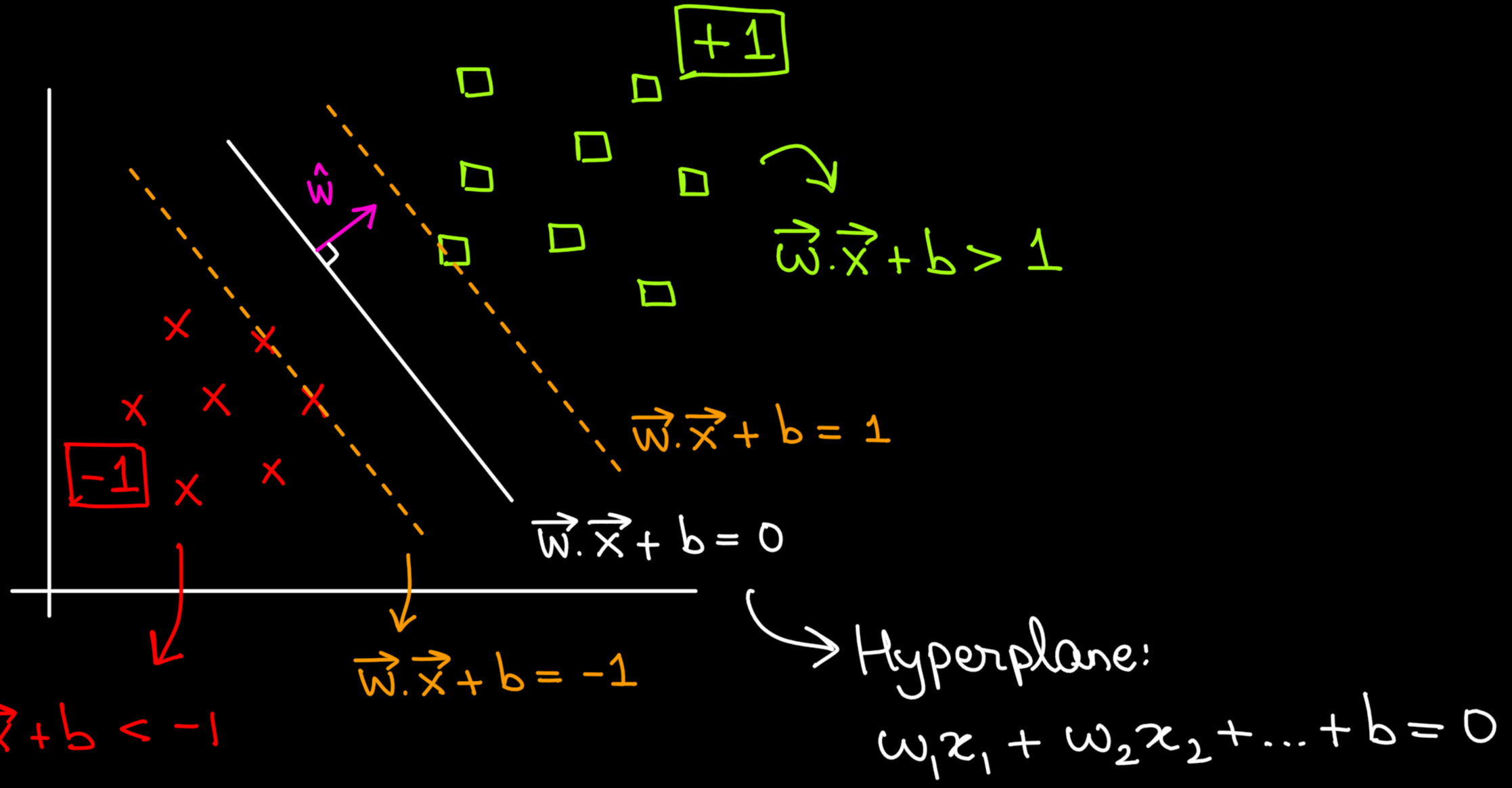
Training data:

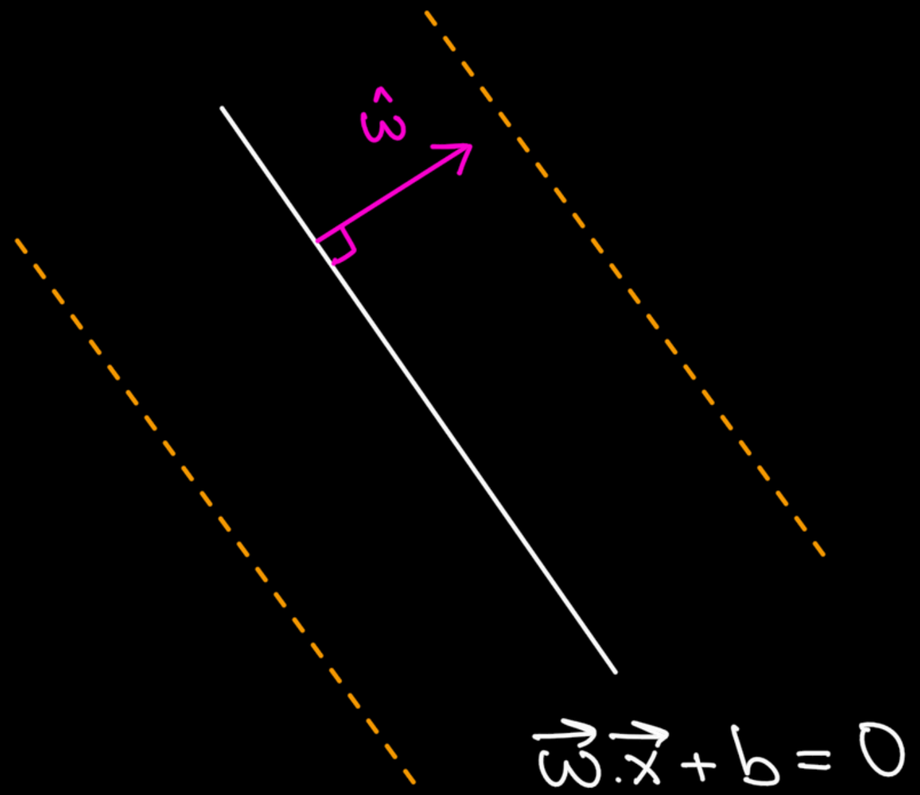
$$(\vec{x}_1, y_1), (\vec{x}_2, y_2) \dots (\vec{x}_N, y_N)$$

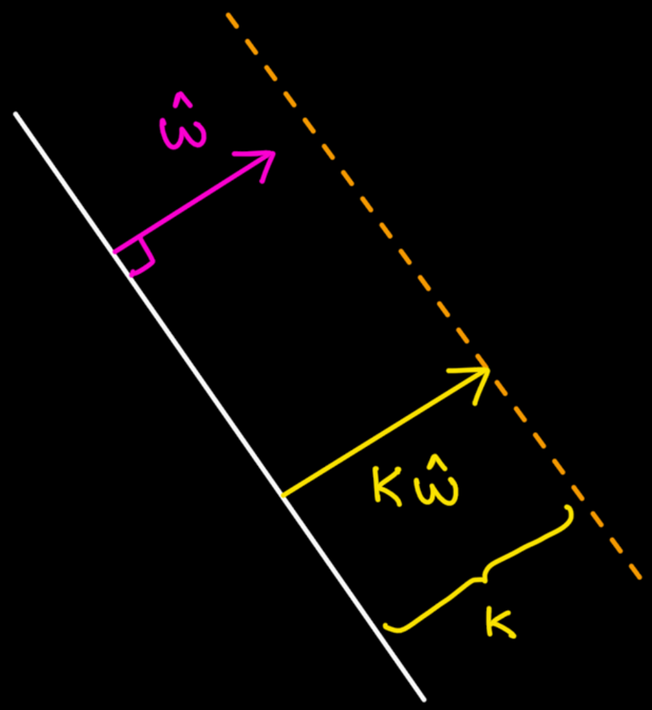
where $y_i \in \{-1, 1\}$




$\vec{w}.\vec{x} + b > 1$
$\vec{w}.\vec{x} + b = 1$
$\vec{w}.\vec{x} + b = 0$
$\vec{w}.\vec{x} + b = -1$
$\vec{w}.\vec{x} + b < -1$


+1
$\hat{w}$
$\vec{w}.\vec{x}+b>1$
$\vec{w}.\vec{x}+b=1$
-1
$\vec{w}.\vec{x}+b=0$
$\vec{w}.\vec{x}+b=-1$
$\vec{w}.\vec{x}+b<-1$
Hyperplane:
$w_1x_1+w_2x_2+\ldots+b=0$


$\hat{w}$
$\vec{w}.\vec{x} + b = 0$



$$\vec{w}.(\vec{x}+k\hat{w})+b=1$$

$$\vec{w}.\vec{x}+b=0$$

What is the margin size?

- if $\vec{x}$ is on the decision boundary,

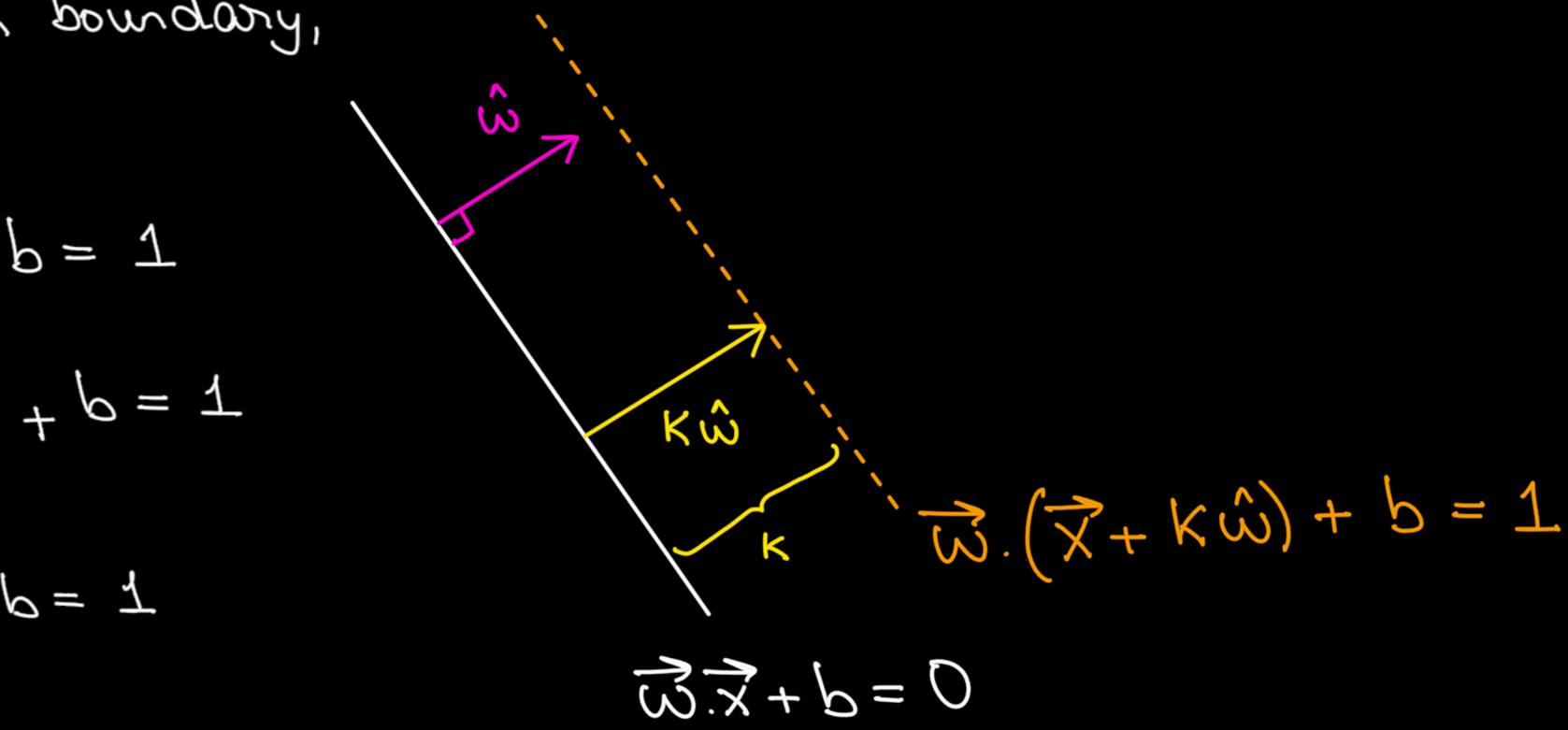
$$\vec{w} \cdot \vec{x} + b = 0$$

So, $\vec{w} \cdot (\vec{x} + k\hat{w}) + b = 1$

$$\vec{w} \cdot \vec{x} + k\, \vec{w} \cdot \frac{\vec{w}}{\|w\|} + b = 1$$

$$k\,\|w\| + \vec{w} \cdot \vec{x} + b = 1$$

$$\therefore k = \frac{1}{\|w\|}$$

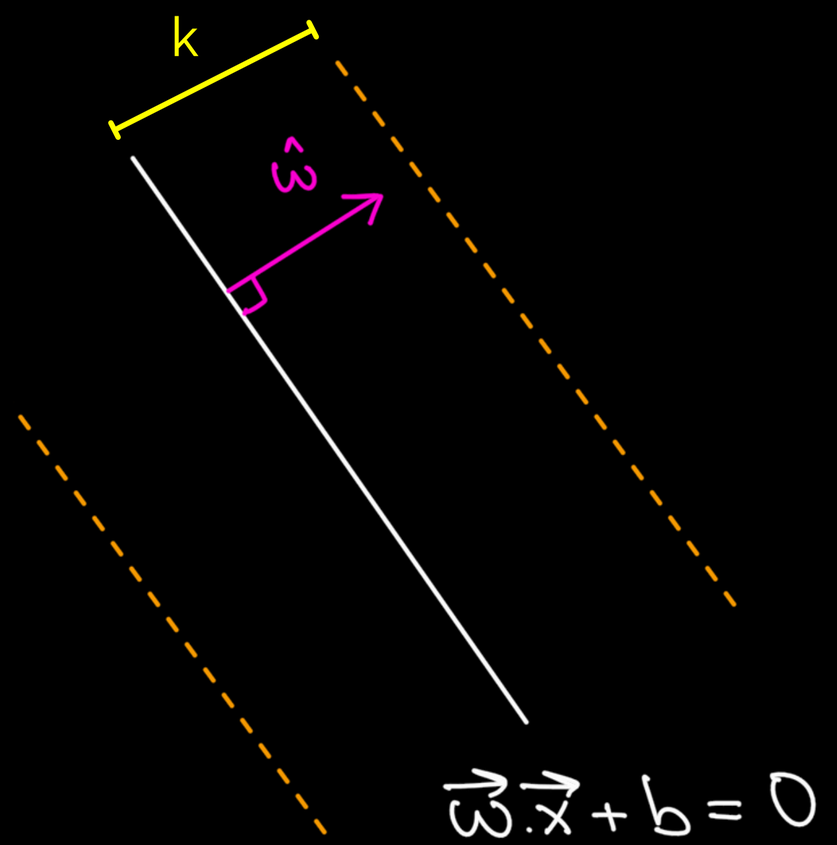


$$k = \frac{1}{\|w\|}$$

$$\text{margin} = 2k = \frac{2}{\|w\|}$$

max margin $\Rightarrow$ min $\|w\|$

for Convenience, we try to minimize $\frac{1}{2}\|w\|^2$



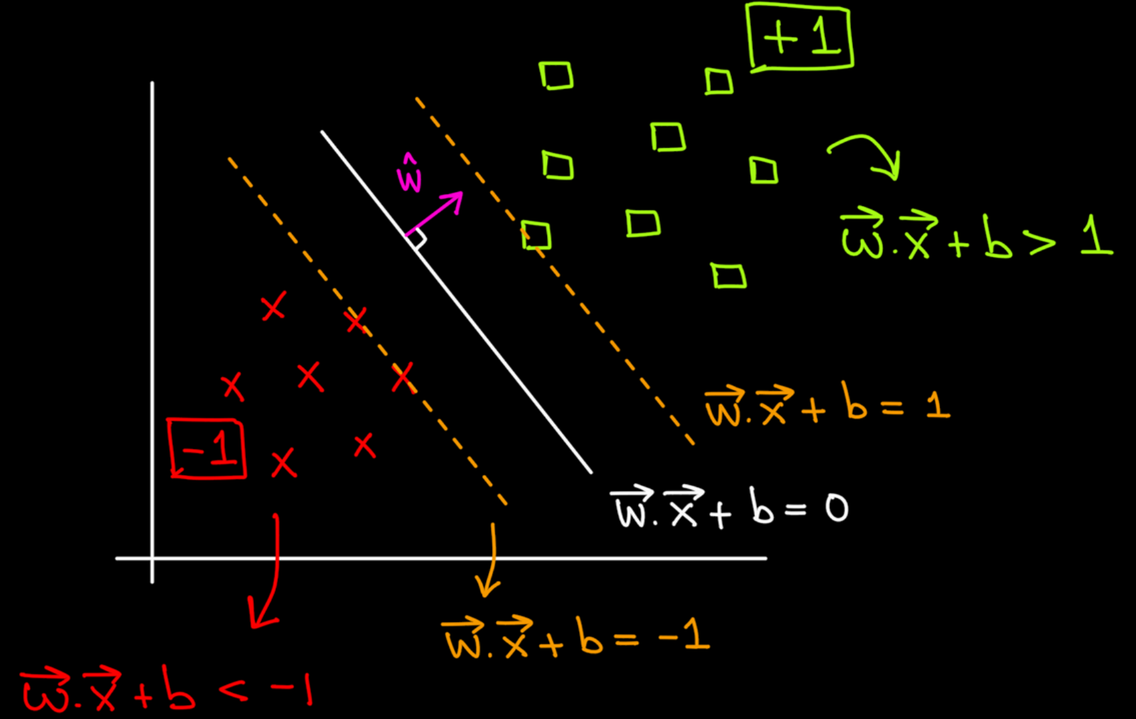
Constraint:

$$y_i = 1 \Rightarrow \vec{w}.\vec{x}_i + b \geq 1$$

$$y_i = -1 \Rightarrow \vec{w}.\vec{x}_i + b \leq -1$$

OR

$$y_i(\vec{w}.\vec{x}_i + b) \geq 1 \quad \forall \; i = 1, 2, \ldots n$$



Finally, our goal is to:

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{such that} \quad y_i(\vec{w}.\vec{x}_i + b) \geq 1$$

minimize

constraint

This is called the "Primal"

"get the widest margin S.T. all data points are classified correctly"

Constrained Optimization

Lagrangian: $L = \frac{1}{2}w^T w - \sum_{i=1}^{N} \alpha_i \left[y_i(\vec{w}.\vec{x}_i + b) - 1 \right]$

$$\frac{\partial L}{\partial \vec{w}} = \vec{w} - \sum \alpha_i y_i \vec{x}_i = 0$$

$$\therefore \vec{w} = \sum \alpha_i y_i \vec{x}_i$$

→ Since $y_i \neq 0$ for non-support vectors, $\alpha_i = 0$

$$\frac{\partial L}{\partial b} = -\sum \alpha_i y_i = 0$$

$$\therefore \sum \alpha_i y_i = 0$$

Now, we have

① $$L = \frac{1}{2} w^T w - \sum_{i=1}^{N} \alpha_i \left[y_i (\vec{w} \cdot \vec{x}_i - b) - 1 \right]$$

② $$\vec{w} = \sum \alpha_i y_i \vec{x}_i$$

③ $$\sum \alpha_i y_i = 0$$

① $L = \frac{1}{2}\omega^T\omega - \sum_{i=1}^{N} \alpha_i \left[y_i(\vec{\omega}\cdot\vec{x}_i - b) - 1 \right]$

② $\vec{\omega} = \sum \alpha_i y_i \vec{x}_i$

③ $\sum \alpha_i y_i = 0$

$$L = \frac{1}{2}\left(\sum_i \alpha_i y_i \vec{x}_i\right)\left(\sum_j \alpha_j y_j \vec{x}_j\right)$$

$$- \sum_i \alpha_i y_i \vec{x}_i \cdot \left(\sum_j \alpha_j y_j \vec{x}_j\right) - \sum_i (\alpha_i y_i) b + \sum_i \alpha_i$$

$$L = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

Now our goal becomes to:

$$\max_{\alpha} L(\alpha) = \max_{\alpha} \left[\sum \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (\bar{x}_i \cdot \bar{x}_j) \right]$$

such that $\alpha_i \geq 0$

$$\sum \alpha_i y_i = 0$$

This is called the "Dual"

Notice:

while classifying: $\vec{w} \cdot \vec{u} + b$

while training: $\frac{1}{2} \sum \alpha_i \alpha_j y_i y_j \vec{x}_i \cdot \vec{x}_j - \sum \alpha_i$

During training and inference, the model only depends on dot products.

Primal

$$\begin{aligned} \min_{\vec{w},b} \quad & \frac{1}{2}\|\vec{w}\|^2 \\ \text{subject to} \quad & y_i(\vec{w}\cdot\vec{x}_i + b) \geq 1, \quad \forall i \end{aligned}$$

Dual

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j (\vec{x}_i \cdot \vec{x}_j) \\ \text{subject to} \quad & \alpha_i \geq 0, \quad \forall i \\ & \sum_{i=1}^{n} \alpha_i y_i = 0 \end{aligned}$$

Why Do We Care About the Dual?

- **The Dual Gives Sparsity (Only SVs Matter):**

$$\max_{\alpha} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (x_i \cdot x_j)$$

In the primal, w depends on all training points.

Whereas, in the dual: $w = \sum_i \alpha_i y_i x_i$

But most alpha's are zero so only support vectors contribute.

- **Dual Enables the Kernel Trick:**

In the dual, data only appears as dot products. So we can use kernels without computing ϕ(x) explicitly. This makes non-linear classification possible and efficient.

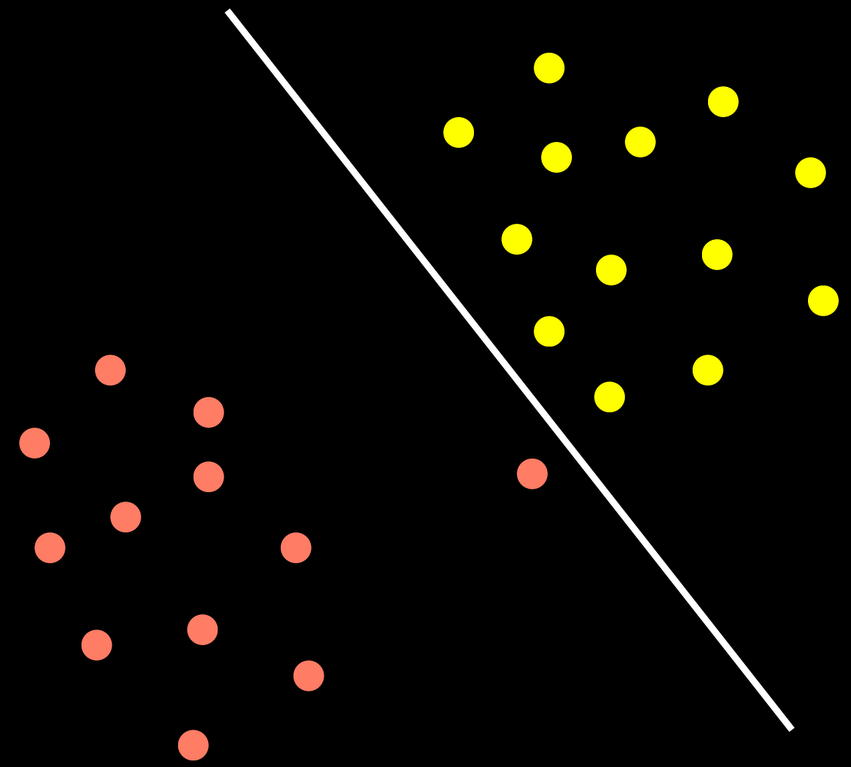
- **Dual Handles Constraints Cleanly:**

Primal constraint: $y_i(w \cdot x_i + b) \geq 1$

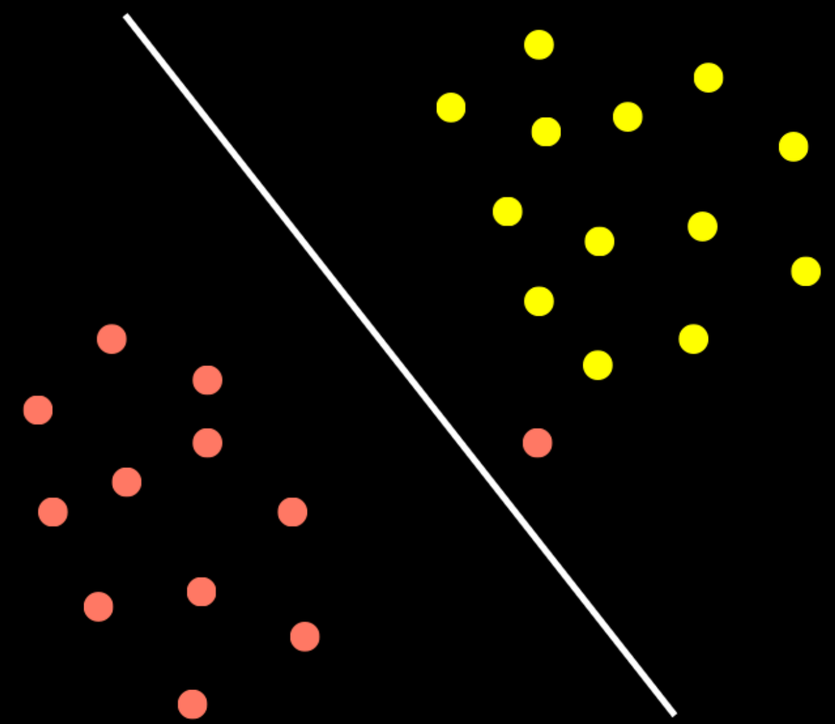
Dual constraint: $0 \leq \alpha_i \leq C, \quad \sum_i \alpha_i y_i = 0$ → Easier to solve using quadratic programming.

What if the data isn’t linearly separable?

When Hard Margins Fail



What we get

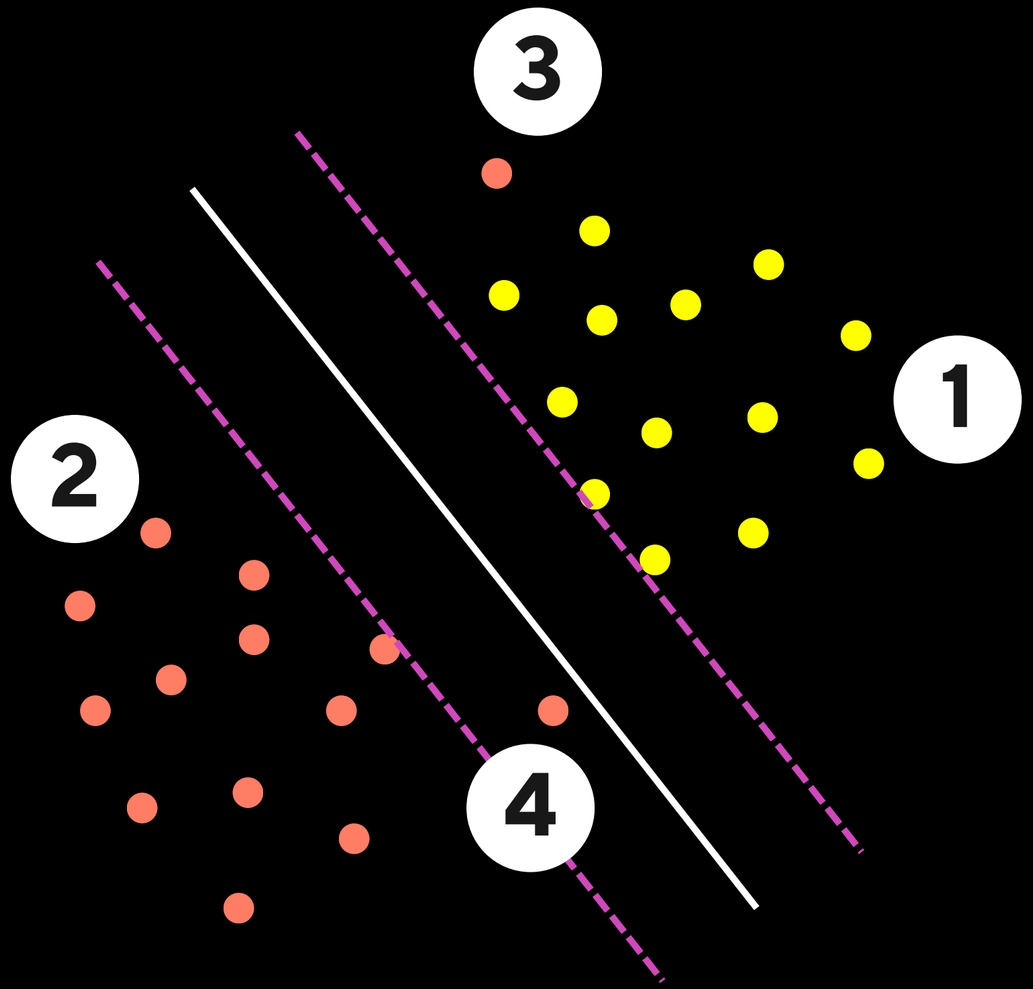


What we want

When Hard Margins Fail

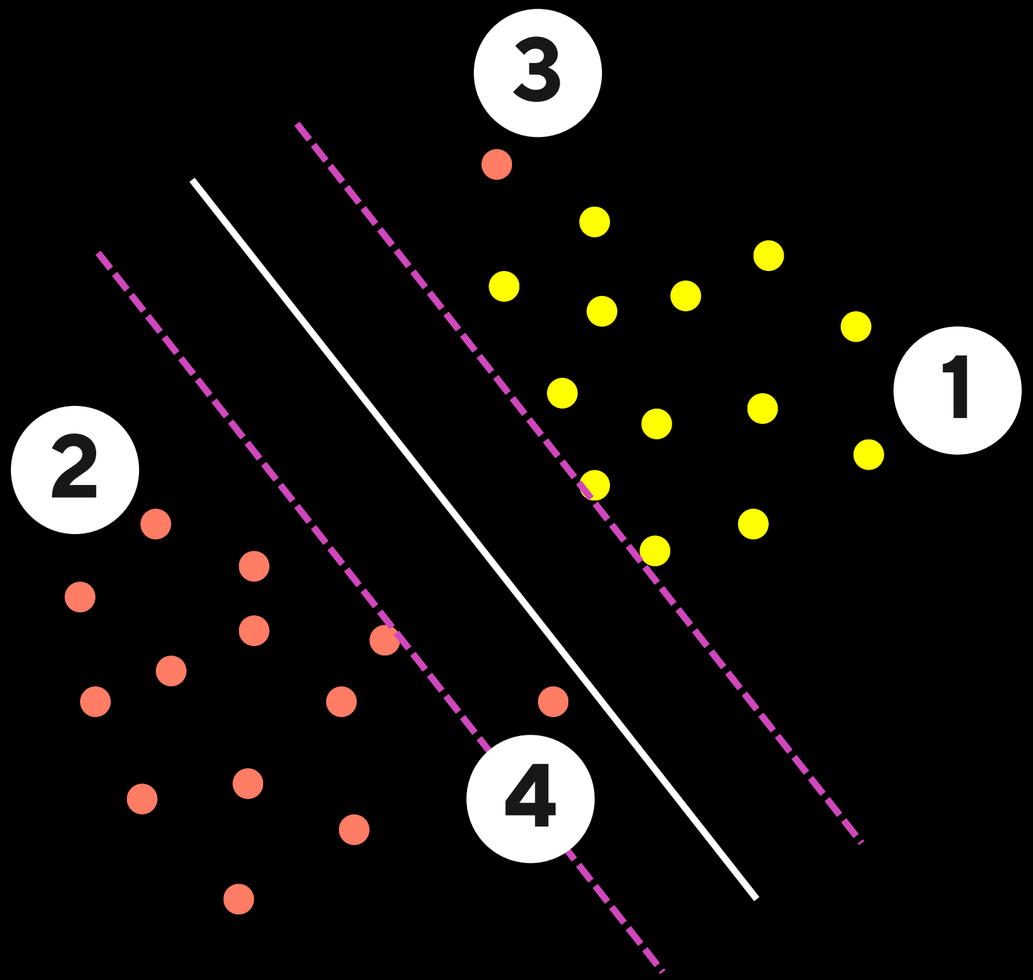
- So far, we’ve assumed that our data is perfectly separable — that there's a clear gap between the two classes, and we can draw a clean margin with no points on the wrong side.
- But real-world data isn't that clean. It’s messy. You’ll have outliers, overlapping classes, label noise and even inseparable data.
- If you still try to force a hard-margin SVM here, the optimization problem becomes infeasible there's literally no ‘w’ and ‘b’ that can satisfy all the constraints.
- So we need to relax the problem and allow some violations, but control how much we allow.

Cases of Classification



1. Zero error (correctly classified)
2. Zero error (correctly classified)
3. High error (in wrong cluster)
4. Low error (within margin boundaries)

Hinge Loss



$$L = \max(0,\ 1 - y_i\,[w \cdot x_i + b])$$

Case		Hinge Loss
1. $y = +1,$	$w \cdot x + b > 1$	0
2. $y = -1,$	$w \cdot x + b < -1$	0
3. $y = -1,$	$w \cdot x + b > 1$	>1
4. $y = -1,$	$w \cdot x + b \in (-1,\ 0)$	$\in (0,\ 1)$

New Objective

$$\min_{w,b} \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \max(0,\ 1 - y_i\,[w \cdot x_i + b])$$

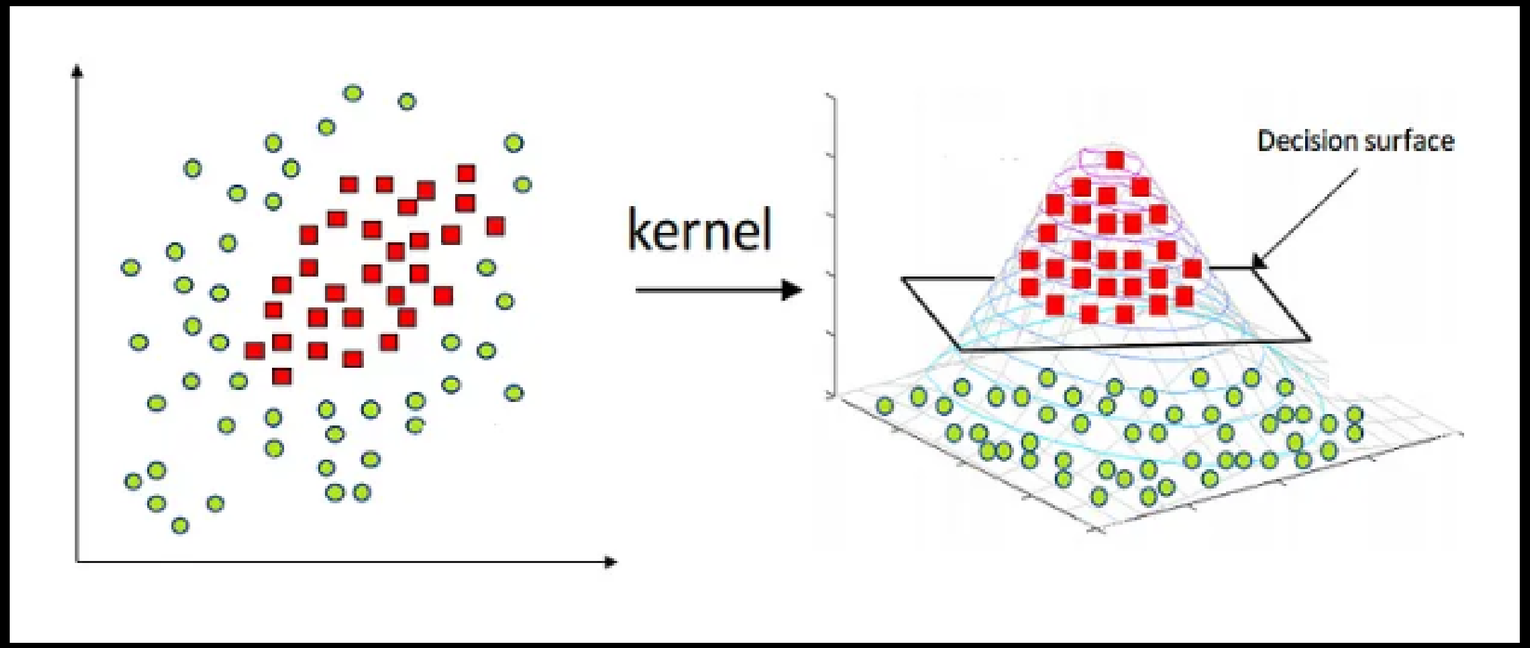
- In soft-margin SVM, we allow misclassifications by penalizing them using hinge loss.
- The first term keeps the margin large; the second term penalizes points violating it.

- C is a hyperparameter that controls how strictly the model penalizes violations.
- A high C tries to classify every point correctly, even if it means a smaller margin → risk of overfitting.
- A low C allows some violations to keep the margin wide → better generalization.

What if the data is not linearly separable... even with a soft margin?

Idea: Make the data linearly separable... by lifting it

- In 2D, this data is not linearly separable.
- But by applying a kernel, we implicitly lift the points into a higher-dimensional space where a simple linear surface (a plane) can now separate the classes.



Mapping into a higher space

- A mapping ϕ(x) transforms your original data point into a new, higher-dimensional space; often to make it linearly separable.
- For example:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \xrightarrow{\phi(x)} \phi(x) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{bmatrix}$$

- But computing these transformed vectors explicitly is expensive — especially in very high or infinite dimensions.
- This slows down both training and prediction.

The Kernel Trick

- In the dual form of SVM, predictions and training depend only on dot products between data points: $f(x) = \sum_i \alpha_i y_i (x_i \cdot x) + b$
- Now imagine we mapped each point into a higher-dimensional space using ϕ(x) — to make the data linearly separable.
  But then every dot product becomes: $\phi(x_i) \cdot \phi(x_j)$

- The trick: we don't need to compute ϕ(x) at all — we just use a kernel function:

$$K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$$

This is the kernel trick: it gives us the power of high-dimensional spaces — without ever computing the transformation.

How SVM Changes With Kernels

- In kernel SVM, we don’t need a new algorithm — we just replace the dot product with a kernel function: $K(x_i, x_j)$, everything else remains the same.
- The dual optimization problem becomes:

$$\max_{\alpha} \sum_{i} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

- Subject to the same constraints:

$$0 \leq \alpha_i \leq C, \quad \sum_{i} \alpha_i y_i = 0$$

- And the decision function used for prediction is:

$$f(x) = \sum_{i} \alpha_i y_i K(x_i, x) + b$$

Popular SVM Kernels

- **Linear:** Use when data is linearly separable.

$$K(x, x') = x \cdot x'$$

- **Polynomial:** Captures curved boundaries. Degree controls complexity.

$$K(x, x') = (x \cdot x' + c)^d$$

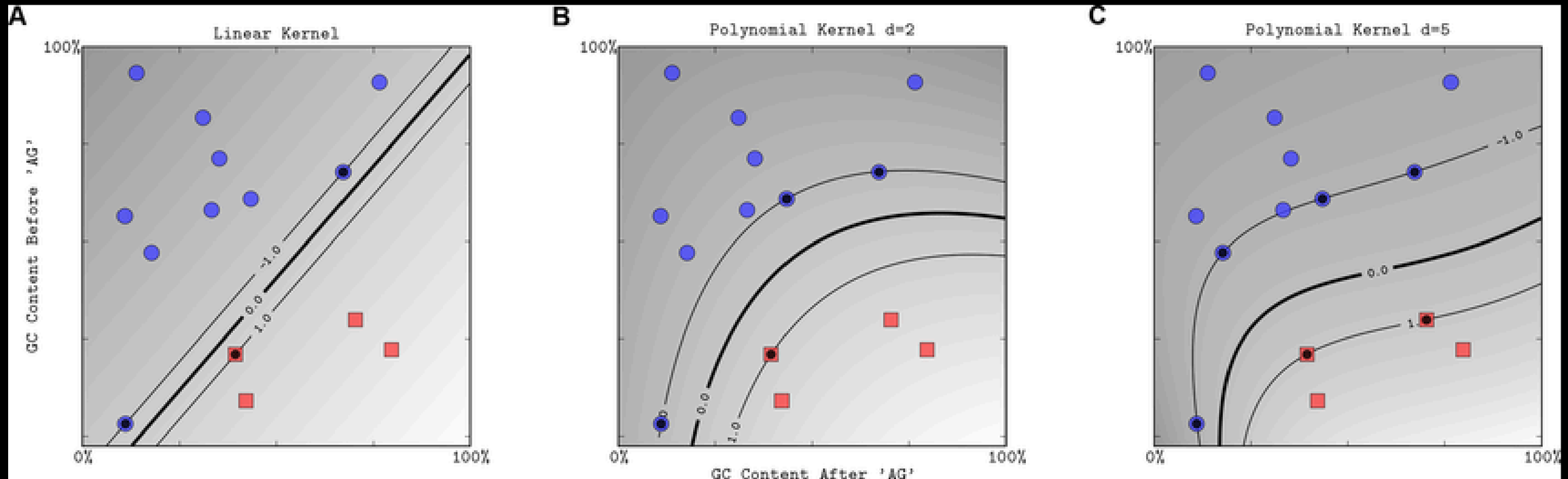
- **RBF/Gaussian Kernel:** Works well for most non-linear data. Highly flexible.

$$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$$

Polynomial Kernel

$$K(x, x') = (x \cdot x' + c)^d$$

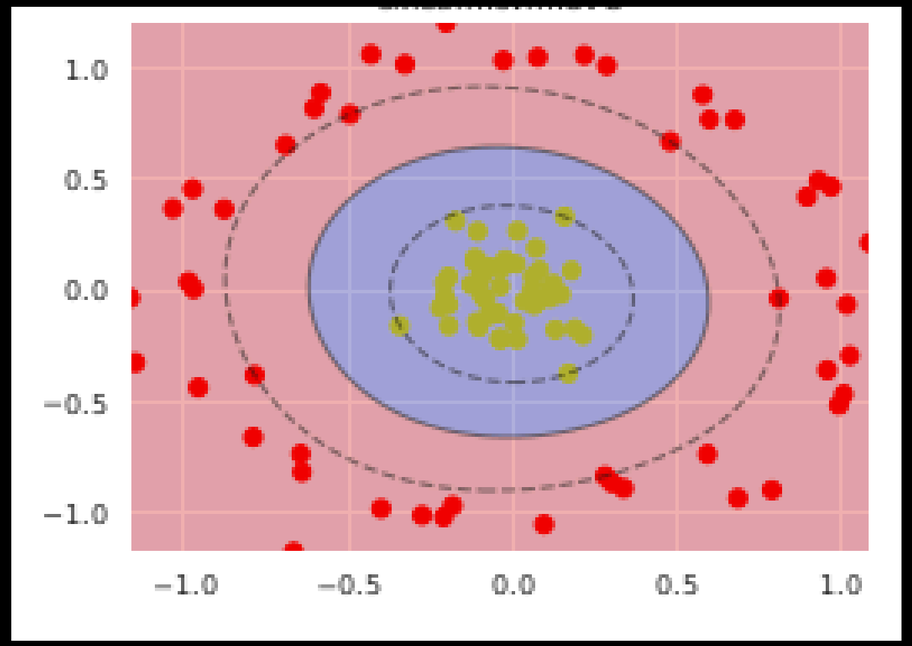
- d: the degree of the polynomial (controls flexibility).
- c: a constant that adjusts the influence of higher-order terms.
- When d=1, this becomes the linear kernel.
- Higher d captures more complex patterns, but can overfit if too large.



Radial Basis Kernel

$$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$$

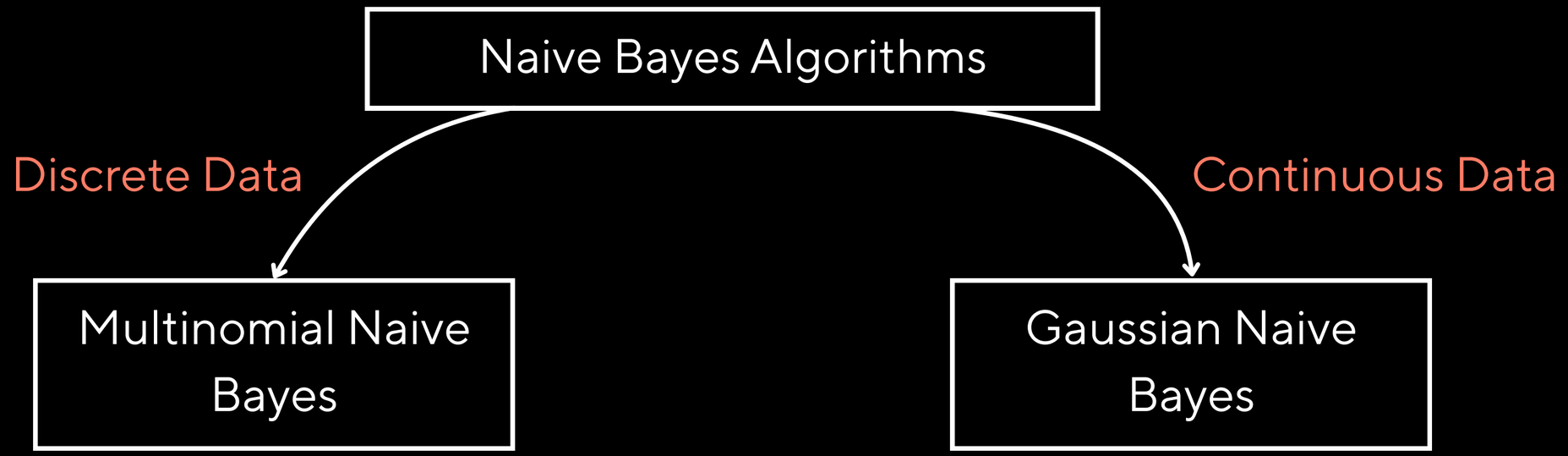
σ controls the spread:
Small σ → tight clusters → complex boundaries (can overfit)
Large σ → smooth boundaries (can underfit)



Code Implementation

Naive Bayes Classifiers

Most Popular Naive Bayes Algorithms



Based on the Popular Bayes Theorem!

- For any events A and B:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Multinomial Naive Bayes

- Used when the features of the dataset are **Discreet.**
- Used in tasks such as classification of text data (Spam/Not Spam emails, Positive/Negative feedback and so on).
- To dive in further, lets take an example of a Spam/Not Spam Dataset:

Not Spam	**Spam**
Hi Friend	Money Money Friend
Dear	Dear Take Money
Friend, Lunch	
Give Money!	

Goal

Given the Spam/Not Spam dataset, lets try to classify whether the sentence “Dear Friend” is Spam/Not Spam.

How we do this?

- We try to compute two probabilities : P(Spam | Test Set) and P(Not Spam | Test Set)
- These are computed using the Bayes Theorem.

Not Spam	**Spam**
Hi Friend	Money Money Friend
Dear	Dear Take Money
Friend, Lunch	
Give Money!	

Algorithm

We start by calculating:

1.P(Class):

- P(Spam) = 2/6
- P(Not Spam) = 4/6

2.P(Word | Class)

3. P(Class | Test Set)

Using the data we get from above, we then calculate, using Bayes Theorem:

- P(Spam | Dear Friend) = P(Dear Friend | Spam) * P(Spam)
- P(Not Spam | Dear Friend) = P(Dear Friend | Not Spam) * P(Not Spam)

We choose to omit the denominator (=1) for the sake of simplicity in calculation, since both of them will have the same denominator P(Dear Friend) and hence wouldn’t affect their scores during comparision.

Calculations

- P(Spam | Dear Friend) = P(Dear Friend | Spam) * P(Spam)
  = P(Dear | Spam) * P(Friend | Spam) * P(Spam)
  = (1/7) * (1/7) * (2/6)
  = 0.0068

- P(Not Spam | Dear Friend) = P(Dear Friend | Not Spam) * P(Not Spam)
  = P(Dear | Not Spam) * P(Friend | Not Spam) * P(Not Spam)
  = (1/7) * (2/7) * (4/6)
  = 0.027

Naive??

- This algorithm makes an assumption.
- This algorithm assumes the features are conditionally independent of each other.
- It does: P(Dear Friend | Spam) * P(Spam) = P(Dear | Spam) * P(Friend | Spam) * P(Spam)
- This is why its called “Naive” (For making this assumption).
- It assumes knowing that the message contains "Dear" gives you no extra clue about the chance of it containing “Friend”, as long as the class is known.
- This is “Naive” in reality. In real spam data, words can co-occur. The presence of one word may cause the presence of another word to change.
- In Multinomial Naive Bayes, “Dear Friend, I ate your lunch” and “Friend Dear, lunch I your ate” will have the same score, even though the grammatical inconsistency in the second case indicates it could be more towards spam!

Laplace Smoothing

- Say we had to find whether the sentence “Lunch Lunch Lunch Lunch Lunch Lunch Money” is Spam or Not Spam. Calculating via the same formula, we would get P(Spam | Sentence) = 0. P(Not Spam | Sentence) = 1. This is obviously wrong.
- This is because the word Lunch doesn’t occur in the Spam dataset, resulting in P(Lunch|Spam) = 0 during calculations.
- To prevent this, we use Laplace Smoothing/Correction:
  - If the word is present in any of the classes, we assume it to be a part of the entire vocabulary. We say that there is a chance that the test data should have this.
  - We increase the count of every single word by 1.

$$P(w_i|\text{class}) = \frac{\text{freq}(w_i, \text{class})}{N_{\text{class}}} \qquad \text{class} \in \{\text{Positive, Negative}\}$$

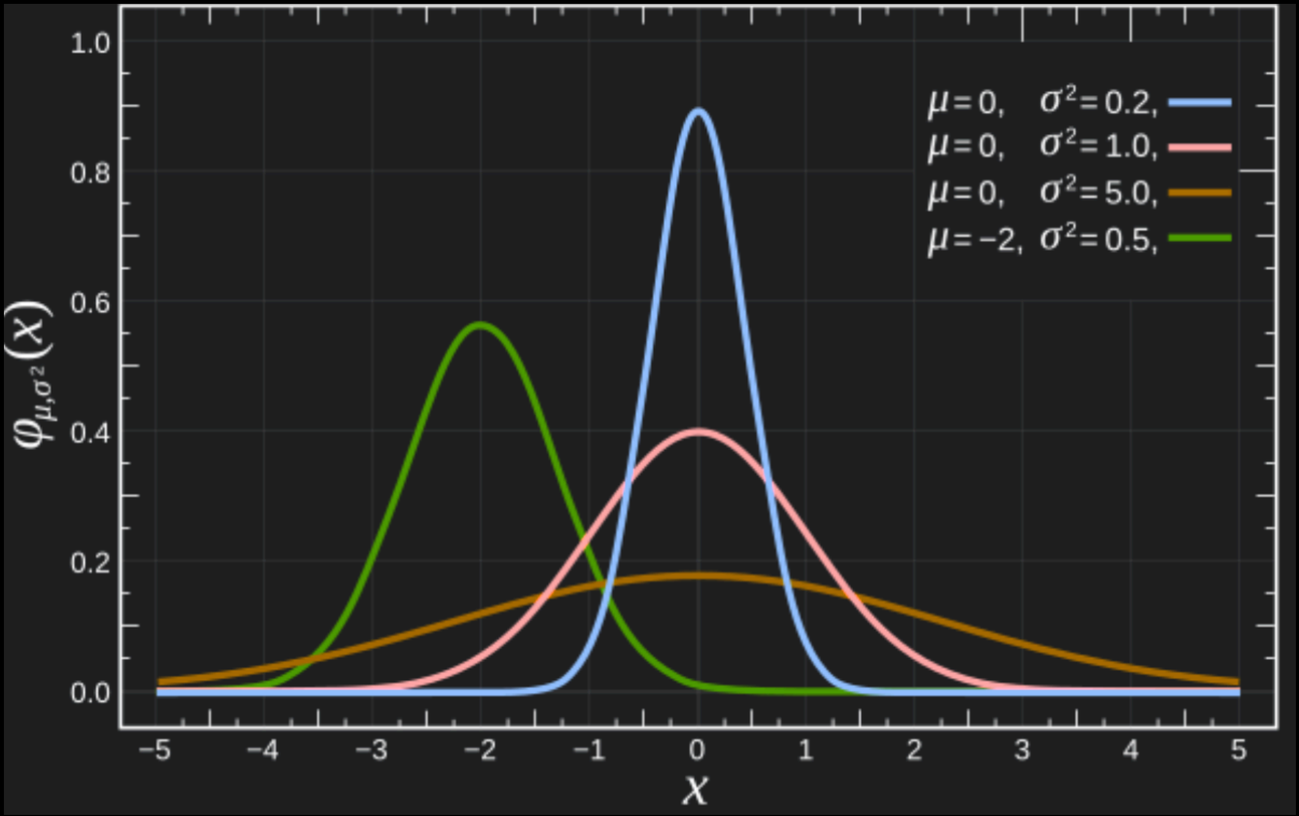
$$P(w_i|\text{class}) = \frac{\text{freq}(w_i, \text{class}) + 1}{N_{\text{class}} + V}$$

N_{class} = frequency of all words in class

V = number of unique words in vocabulary

Gaussian Naive Bayes

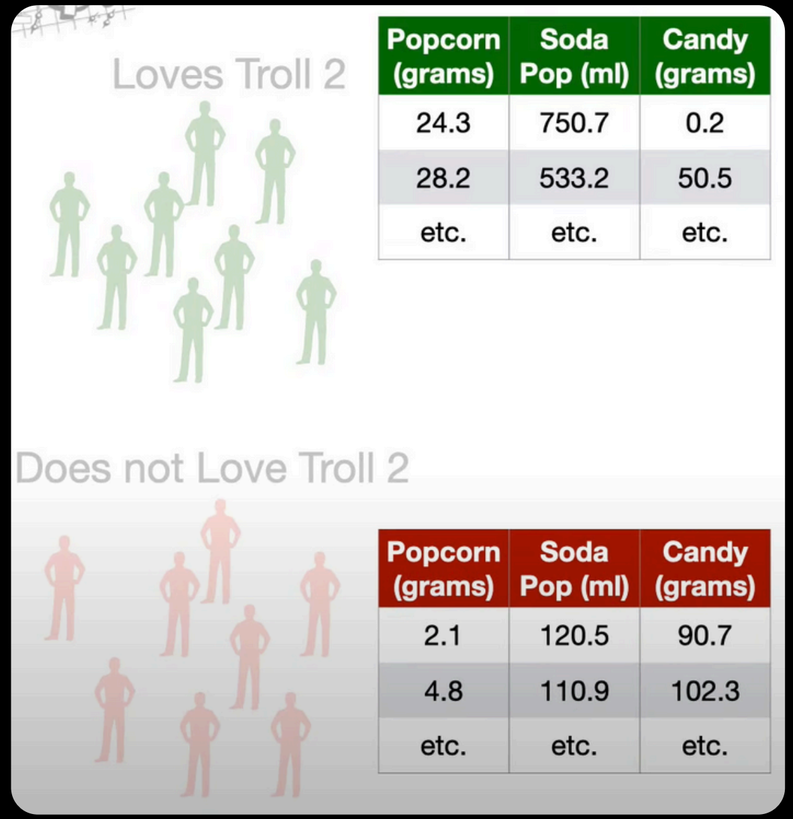
- Used when the features of the dataset have some **Continuity.**
- Based on the Gaussian Distribution Curve.



$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Example!

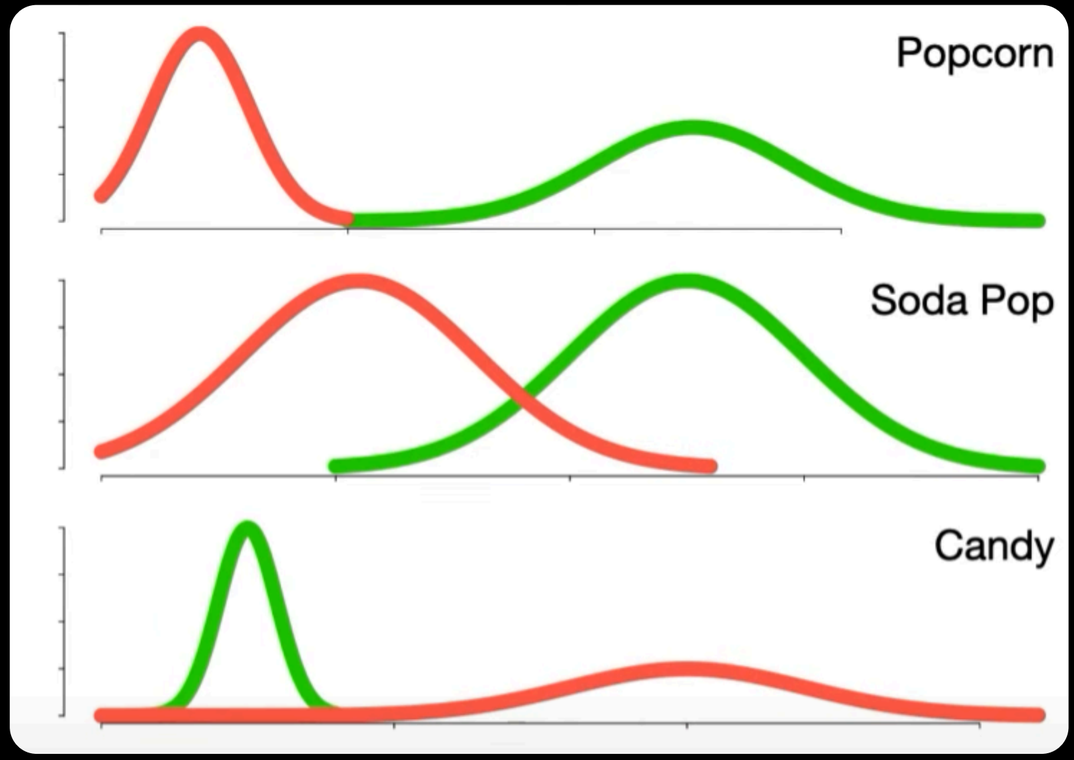
- Say we have data on the quantity of Popcorn, Soda, Candy (P, C, S) bought during a movie show - Troll 2.
- The data is classified into 2 types - Likes the movie, Dislikes the movie.
- Now if we're given new Popcorn, Soda and Candy data for a person, we have to predict whether the person likes the movie or not.



Example!

- How is this related to Gaussian Naive Bayes? We assume that these features follow a Gaussian distribution.

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_{y,i}^2}} \exp\left(-\frac{(x_i - \mu_{y,i})^2}{2\sigma_{y,i}^2}\right)$$



Example!

- P(Likes Troll 2 | Given values of P, C, S) = P(Likes troll 2) * P(P | Likes troll 2) * P(C | Likes Troll 2) * (S | Likes Troll 2)
- This also assumes conditional independence amongst the features.
- P(Likes Troll 2) = ½ (Either likes the movie or dislikes the movie)
- P(P | Likes Troll 2) = Calculated via the Gaussian Formula.
- P(C | Likes Troll 2) = Calculated via the Gaussian Formula.
- P(S | Likes Troll 2) = Calculated via the Gaussian Formula.

- In the same way P(Doesn’t Like Troll 2 | Given Values of P, C, S) is calculated.

We run into a problem

- The values further away from the bell peak of the curve are very close to zero.
- When we try to compute the Liklihood of a person liking or disliking the movie via multiplication of the probabilities, there can be cases where each probability is very close to zero, resulting in the multiplication to be very close to 0.
- UNDERFLOW: Computer approximates these extremely small value multiplications to 0.

- To avoid this issue, we compute the log of probabilities.
- log(P(Loves Troll 2 | Given Values of P, C, S)) and
- log(P(Doesnt like Troll 2 | Given Values of P, C, S)) are computed.
- Whichever one has a higher value, it belongs to that class.

Code Implementation

Thank You!

AI
CLUB