```
In [ ]:
```

```
In [ ]:
```

What is Support Vector Machine (SVM)?

SVM stands for **Support Vector Machine**. Imagine you're drawing a line in sand to separate two kinds of toys, say red balls and green cubes. You want this line (called a **hyperplane** in math) to be as far as possible from the nearest toys on each side. The bigger this gap (called **margin**), the better your separation.

In Data Science, these "toys" become data points, and SVM helps us separate categories, like whether an email is **spam or not spam** or whether a customer will **buy or not buy** a product.

First, let's understand the **Dot Product**:

What's a Dot Product?

Think of two arrows (vectors) drawn on paper:

- An arrow has **length** (magnitude) and **direction**.
- A **dot product** helps measure how much one arrow points in the same direction as another arrow.

Here's a simple way to understand it:

- **Arrow A** (your arm stretched out).
- **Arrow B** (a flashlight beam).

If you hold your arm straight in front, and shine a flashlight exactly in the same direction, the dot product is maximum because both point exactly the same way.

If you shine it from the side (90 degrees), your arm doesn't catch the beam, making the dot product zero.

The math formula is:

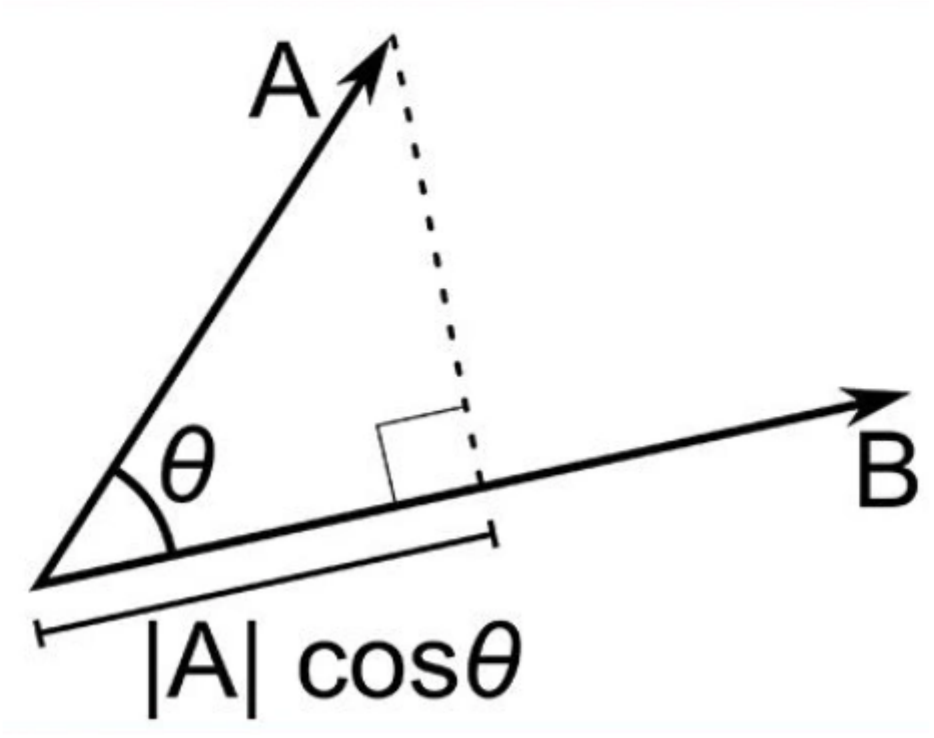
$$A \cdot B = |A| \times |B| \times \cos(\theta)$$

- $|A|$ is the length of arrow A.
- $|B|$ is the length of arrow B.
- θ (theta) is the angle between the arrows.

In **SVM**, we mostly care about **projection**, meaning how much one vector points along the other.

Why Dot Product Matters in SVM?

- Suppose you have a decision boundary (line separating two groups).
- You have a point, and you want to know: **which side does this point belong to?**
- To do this, you measure how far along a special arrow (**perpendicular vector w**) your point goes.
- If the dot product of the point and this special arrow (w) is larger than a value (**c**), the point belongs on one side. Otherwise, it belongs on the other side.



Margin in SVM:

- Margin is how far your decision line (hyperplane) is from the nearest points (**support vectors**) on both sides.
- A good SVM maximizes this margin. Think of it as giving yourself maximum space between two groups.

Equation of hyperplane is:

$$w \cdot x + b = 0$$

- **w** is a vector perpendicular to your line.
- **x** is your point.
- **b** is just shifting your line around (offset).

The SVM decision rules are simple:

- If $w \cdot x + b > 0$, the point belongs to the positive class.
- If $w \cdot x + b < 0$, it's negative.

Why do we choose **w.x + b = 1** and **w.x + b = -1**?

- It's for simplicity. We pick these specific numbers because we want a symmetrical margin around the decision line.
- Changing the number just scales the margin, but doesn't change the actual separating line.

Optimization in SVM (Finding the best line):

SVM tries to find the best values for w and b that maximize the margin (**distance between two closest points on both sides**):

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

With constraints:

- Positive points: $w \cdot x + b \geq 1$
- Negative points: $w \cdot x + b \leq -1$

We combine these two constraints into one:

- For all points: $y(w \cdot x + b) \geq 1$, where y is either +1 or -1.

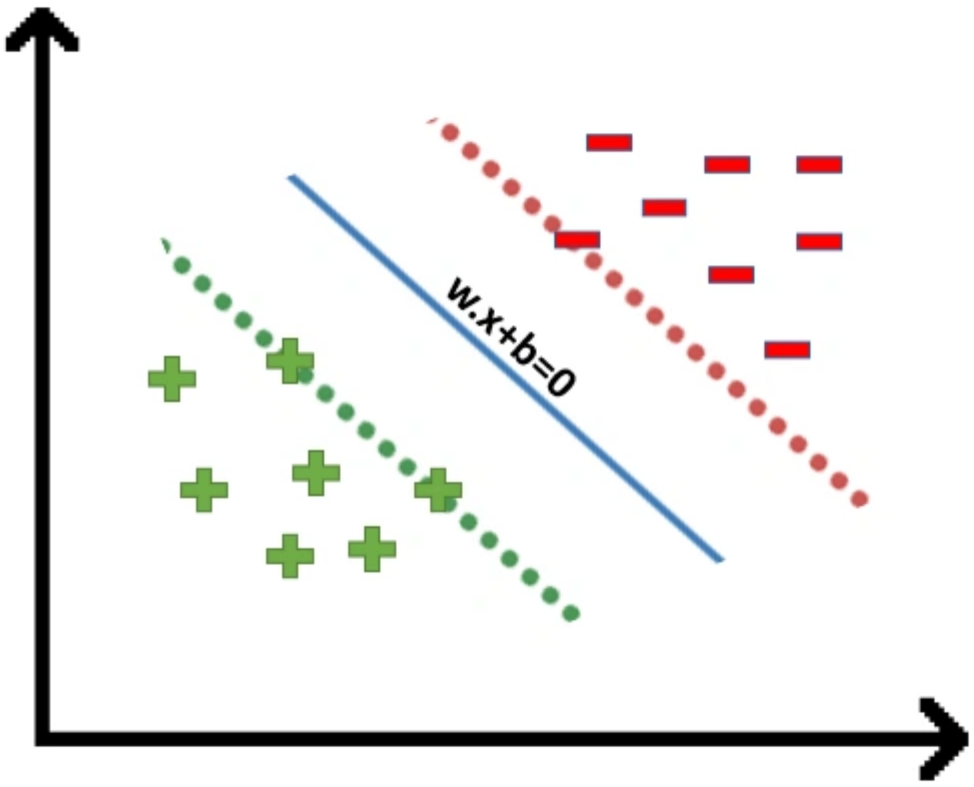
Real-Life Examples (Industry Use):

Example 1: **Spam Email Detection** (Data Science)

- SVM can classify emails as spam or not spam.
- The decision boundary separates spam emails (negative side) from real emails (positive side).
- The margin ensures you confidently classify emails.

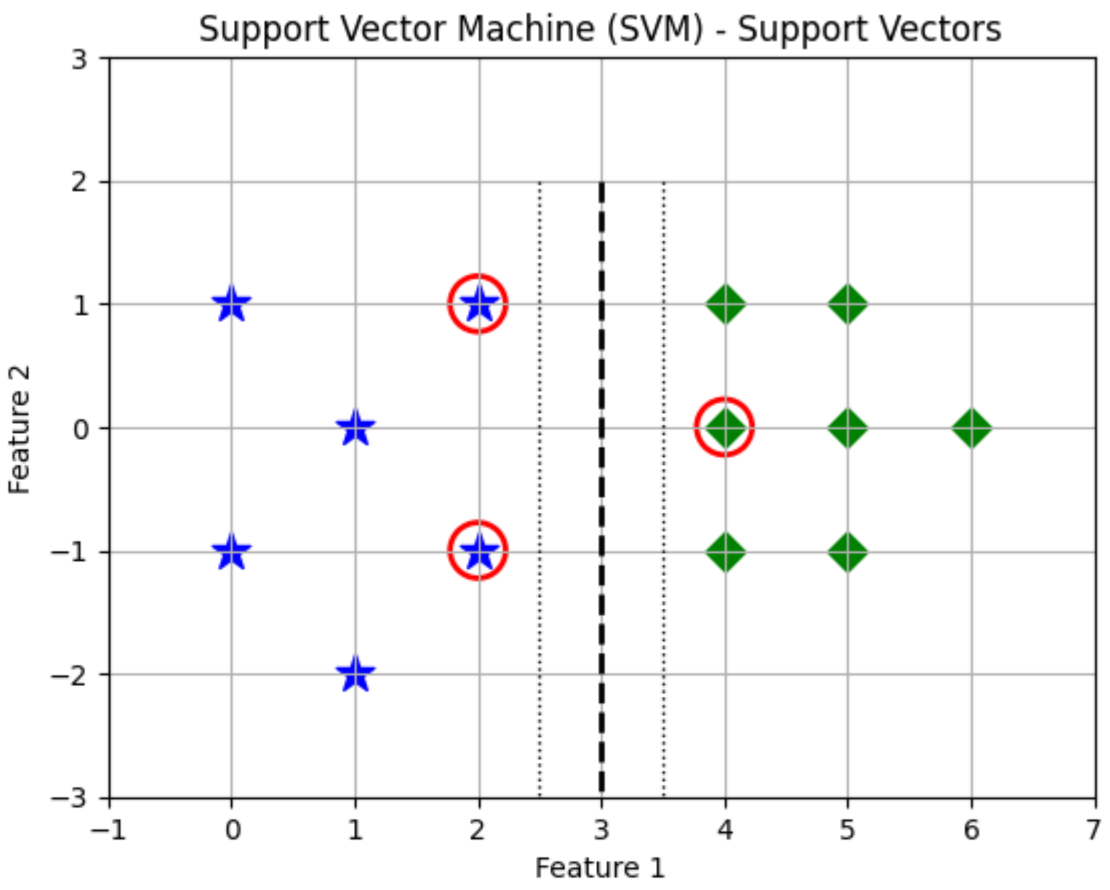
Example 2: **Credit Approval** (Finance)

- Banks use SVM to decide whether to approve loans.
- Data like income, credit history, and debt levels create points.
- The decision boundary separates people who should or shouldn't get loans.



In [ ]:
```
import matplotlib.pyplot as plt

# Data points
blue_stars = [(2,1), (2,-1), (1,0), (1,-2), (0,1), (0,-1)]
green_diamonds = [(4,0), (5,1), (5,-1), (6,0), (4,1), (4,-1), (5,0)]

# Support vectors
support_vectors = [(2,1), (2,-1), (4,0)]

# Plot points
for point in blue_stars:
    plt.scatter(*point, color='blue', marker='*', s=200)

for point in green_diamonds:
    plt.scatter(*point, color='green', marker='D', s=100)

# Highlight support vectors
for sv in support_vectors:
    plt.scatter(*sv, facecolors='none', edgecolors='red', marker='o', s=400,

# Decision boundary line (example)
plt.plot([3,3],[2,-3],'k--', linewidth=2)
```

```
# Margin lines (example)
plt.plot([2.5,2.5],[2,-3],'k:', linewidth=1)
plt.plot([3.5,3.5],[2,-3],'k:', linewidth=1)

# Labels and title
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Support Vector Machine (SVM) - Support Vectors')
plt.grid(True)
plt.xlim(-1,7)
plt.ylim(-3,3)
plt.gca().set_aspect('equal', adjustable='box')

plt.show()
```



🎯Context of the Example

We are building an **SVM classification model** to separate two categories of data:

- **Blue stars (★)** → Negative class (−1)
- **Green diamonds (♦)** → Positive class (+1)

There are **13 observations** total:

- 6 blue stars (−1)

- 7 green diamonds (+1)

From this dataset, we select **3 special data points** called **Support Vectors**, which help define the **best separating boundary** (the hyperplane).

👇The 3 Chosen Support Vectors

These are the points closest to the decision boundary (i.e., they lie on the edges of the margin):

Support Vector	Coordinates	Class
SV_1	(2, 1)	-1 (blue star)
SV_2	(2, -1)	-1 (blue star)
SV_3	(4, 0)	+1 (green diamond)

🧠Step-by-Step: What Are We Trying to Do?

We want to **find the equation of the plane** (line in 2D) that separates the classes and **maximizes the margin**. The plane has the general form:

$$w \cdot x + b = 0$$

To calculate `w` and `b`, we use the **3 support vectors** and solve a system of equations.

But first...

✍️Step 1: Augment the Vectors

We add a third component `1` to each vector to include the **bias** term `b` directly in the vector math. This turns:

- SV_1 → (2, 1, 1)
- SV_2 → (2, -1, 1)
- SV_3 → (4, 0, 1)

Now all our operations (dot products) will naturally include the bias.

🧮Step 2: Write the Linear Equations

We want to find weights **X_1, X_2, X_3** such that:

$$X_1(\mathrm{SV}_1) + X_2(\mathrm{SV}_2) + X_3(\mathrm{SV}_3) = \text{final vector defining the hyperplane}$$

To do this, we build **3 equations** using dot products:

Each equation is:

$$X_1(\mathrm{SV}_1 \cdot \mathrm{SV}_i) + X_2(\mathrm{SV}_2 \cdot \mathrm{SV}_i) + X_3(\mathrm{SV}_3 \cdot \mathrm{SV}_i) = \text{class of } \mathrm{SV}_i$$

🔢Step 3: Calculate Dot Products

Let's calculate each dot product:

📌First equation: using $SV_1 = (2, 1, 1)$

$$\begin{array}{r} X_1(2,1,1)\cdot(2,1,1) = 4+1+1 = 6 \\ X_2(2,-1,1)\cdot(2,1,1) = 4-1+1 = 4 \\ X_3(4,0,1)\cdot(2,1,1) = 8+0+1 = 9 \end{array} \Rightarrow 6X_1 + 4X_2 + 9X_3 = -1$$

📌Second equation: using $SV_2 = (2, -1, 1)$

$$\begin{array}{r} X_1(2,1,1)\cdot(2,-1,1) = 4-1+1 = 4 \\ X_2(2,-1,1)\cdot(2,-1,1) = 4+1+1 = 6 \\ X_3(4,0,1)\cdot(2,-1,1) = 8+0+1 = 9 \end{array} \Rightarrow 4X_1 + 6X_2 + 9X_3 = -1$$

📌Third equation: using $SV_3 = (4, 0, 1)$

$$\begin{array}{r} X_1(2,1,1)\cdot(4,0,1) = 8+0+1 = 9 \\ X_2(2,-1,1)\cdot(4,0,1) = 8+0+1 = 9 \\ X_3(4,0,1)\cdot(4,0,1) = 16+0+1 = 17 \end{array} \quad \Rightarrow 9X_1 + 9X_2 + 17X_3 = +1$$

🧮Step 4: Solve the System of Equations

We now solve the system:

```
6X₁ + 4X₂ + 9X₃  = −1
4X₁ + 6X₂ + 9X₃  = −1
9X₁ + 9X₂ + 17X₃ = +1
```

Solving (from the problem):

$$X_1 = -3.25, \quad X_2 = -3.25, \quad X_3 = 3.5$$

🧠Step 5: Combine Support Vectors with Weights

Now we calculate the final plane:

$$\text{Resulting Vector} = X_1 \cdot SV_1 + X_2 \cdot SV_2 + X_3 \cdot SV_3$$

Substituting:

$$= (-3.25)(2,1,1) + (-3.25)(2,-1,1) + (3.5)(4,0,1)$$

Break it down:

- (-3.25)(2, 1, 1) = (-6.5, -3.25, -3.25)
- (-3.25)(2, -1, 1) = (-6.5, +3.25, -3.25)
- (3.5)(4, 0, 1) = (14, 0, 3.5)

Now add all:

- X-component: -6.5 - 6.5 + 14 = **1**
- Y-component: -3.25 + 3.25 + 0 = **0**
- Bias term: -3.25 -3.25 + 3.5 = **-3**

So final vector:

$$(1, 0, -3) \rightarrow \text{meaning } w = (1, 0),\ b = -3$$

📏Step 6: Interpret the Hyperplane

From the standard SVM equation:

$$w \cdot x + b = 0 \Rightarrow 1 \cdot x + 0 \cdot y - 3 = 0 \Rightarrow x = 3$$

This is a **vertical line** at **x = 3** that separates the two classes.

✅Final Summary

Step	What We Did
Selected 3 Support Vectors	Points that lie on the margin from both classes
Augmented Vectors	Added bias component for easier math
Built Dot Product Equations	Based on similarity of points
Solved for Weights	Found X_1, X_2, X_3

Step	What We Did
Combined to Form Hyperplane	Used weighted sum of vectors
Derived Final Plane	Equation: x = 3 → perfectly separates classes

---

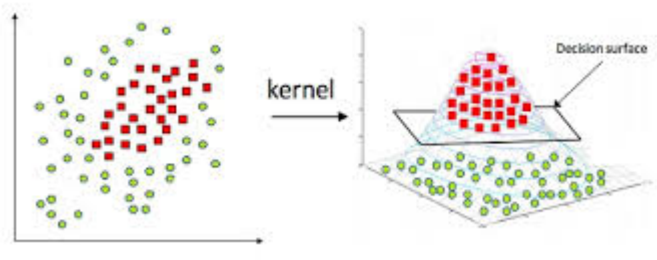
📌Real-world Interpretation (HR Example):

You might use SVM like this:

- Employees with features like (Years of experience, Productivity Score)
- SVM finds the **perfect dividing rule**, like: "Anyone with experience below 3 years → Not ready for leadership" "Anyone with experience above 3 years → Ready for leadership"

This decision boundary is **interpretable and robust**, which is why **SVM is still widely used in classification**.

```
In [ ]:
```



🌀What is a Kernel in SVM?

Sometimes, data points are mixed up in such a way that you can't easily draw a straight line (or hyperplane) to separate them. Think about having green and blue candies scattered randomly. If they're mixed together closely, a straight line won't help much.

The solution? Lift the candies into a new "world" (a higher dimension) where separating them becomes easier. Kernels help us do exactly this.

🚀Kernel Trick (Simple Explanation)

The Kernel Trick is like a magic lens:

- You start with points that you can't separate easily.
- You put on your magic lens (use a kernel).
- Now, the points appear in a different, simpler world, where drawing a straight line

between them is easy.

This new view makes classification straightforward!

🎨 Common Kernel Types (Made Easy)

Here are three common kernel types, explained simply:

1. Polynomial Kernel 🌟

This kernel makes new features by multiplying existing features together.

Simple Formula:

$$\text{Kernel}(X, Y) = (X \cdot Y + 1)^d$$

- d is how complicated (curvy) the boundary becomes.
- Example: If $d = 2$, you turn 2 features into new features like X_1^2, X_2^2, and $X_1 X_2$.

Intuition: Imagine folding paper. More folds mean more complexity, helping separate points better.

---

2. Sigmoid Kernel 💡

This kernel acts like a brain cell (neuron), mapping your features onto a curve that separates data easily.

Simple Formula:

$$\text{Kernel}(X, Y) = \tanh(a(X \cdot Y) + b)$$

- The formula "squishes" the values between -1 and +1.
- It helps separate data similar to neural networks.

Intuition: It's like bending your data gently until two groups can be separated by a simple line.

---

3. Radial Basis Function (RBF) Kernel 🌀 (Most Popular!)

This kernel is like creating bubbles around points:

Simple Formula:

$$\text{Kernel}(X, Y) = e^{-\gamma ||X - Y||^2}$$

- $||X - Y||$ is just the distance between two points.
- γ (gamma) controls how big the bubbles are.

Intuition: Each point becomes a small bubble, clearly separating different classes easily.

🔍 Why RBF Kernel is Popular?

- It works very well with most real-world data.
- It creates smooth, flexible boundaries that neatly separate complicated, messy data.

Imagine using RBF like dropping bubbles around points, naturally separating them into groups.

🌈 Choosing Kernels (Simple Guide)

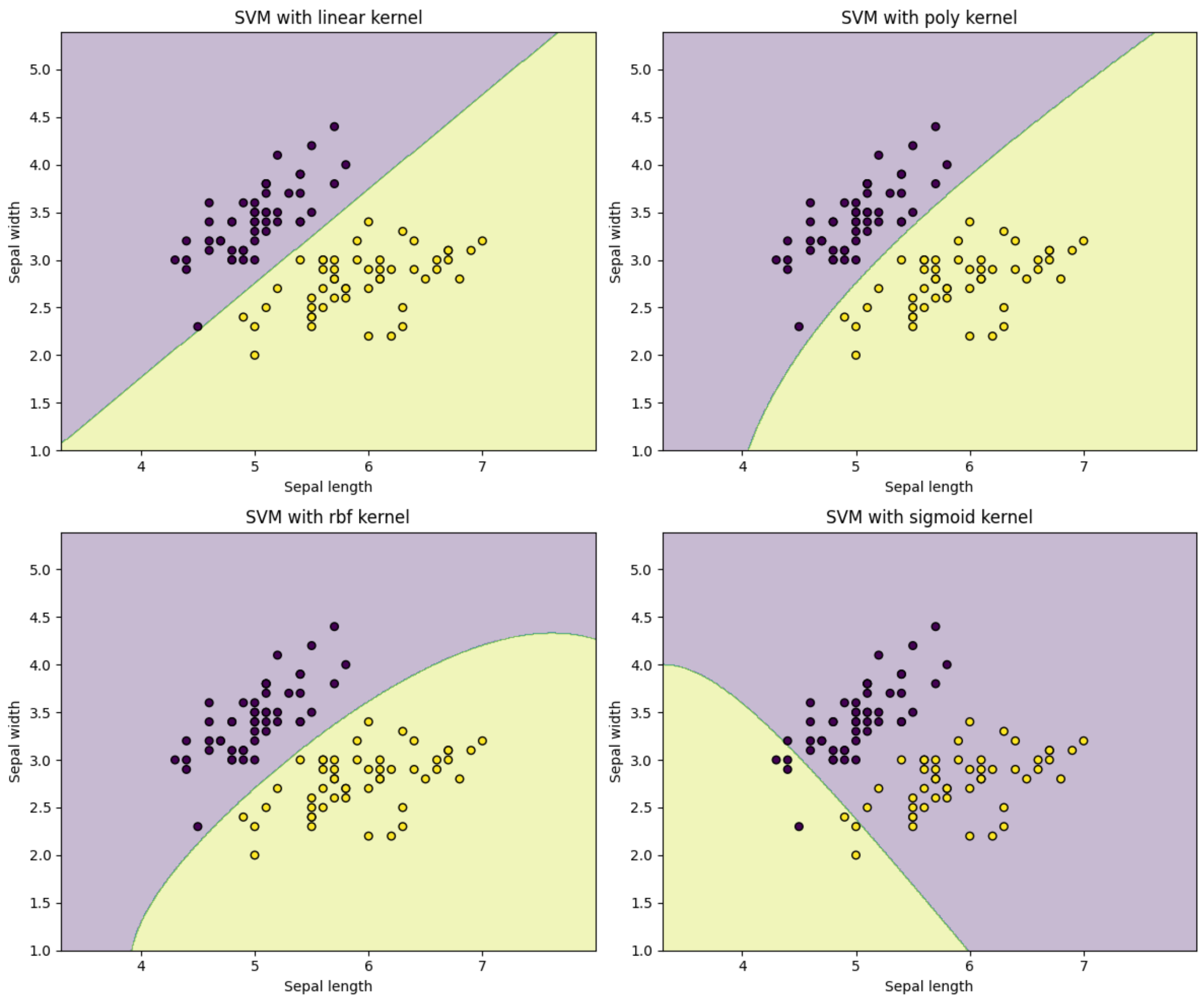
- Start with a **linear kernel** (straight line) first.
- If data can't be separated easily, use **RBF kernel** next (common).
- Polynomial and Sigmoid kernels are used less often, usually when experimenting with special datasets.

📌 Real-life Example: Classifying Emails

- **Linear Kernel**: For simple email data, where spam is clearly separate from regular emails.
- **RBF Kernel**: For complex data, where spam emails can be similar to regular emails. RBF helps clearly divide them.

In [ ]:
```
# Importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.svm import SVC

# Loading Iris dataset
iris = datasets.load_iris()
X = iris.data[:, :2]  # Only the first two features for visualization
y = iris.target

# We will only consider two classes for simplicity
X = X[y != 2]
y = y[y != 2]

# Creating SVM models with different kernels
kernels = ['linear', 'poly', 'rbf', 'sigmoid']
models = [SVC(kernel=k).fit(X, y) for k in kernels]
```

```python
# Plotting decision boundaries
fig, axs = plt.subplots(2, 2, figsize=(12, 10))

x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                     np.arange(y_min, y_max, 0.01))

for ax, clf, kernel in zip(axs.flatten(), models, kernels):
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    ax.contourf(xx, yy, Z, alpha=0.3)
    ax.scatter(X[:, 0], X[:, 1], c=y, s=30, edgecolor='k', marker='o')
    ax.set_title(f'SVM with {kernel} kernel')
    ax.set_xlabel('Sepal length')
    ax.set_ylabel('Sepal width')

plt.tight_layout()
plt.show()
```


SVM with linear kernel
SVM with poly kernel
SVM with rbf kernel
SVM with sigmoid kernel
Sepal length
Sepal width