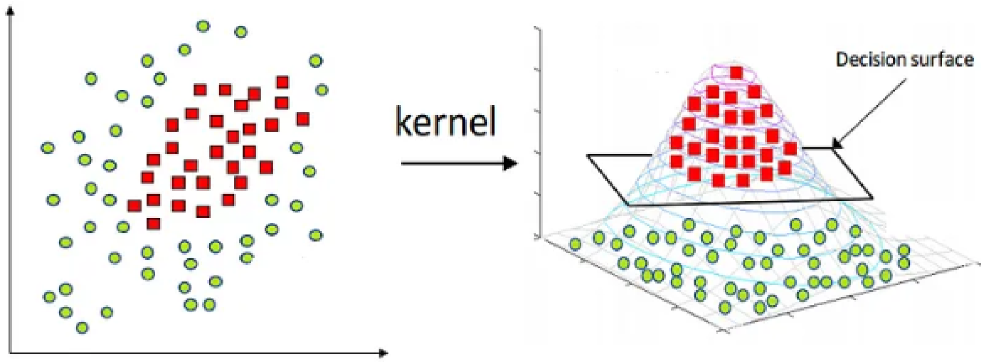
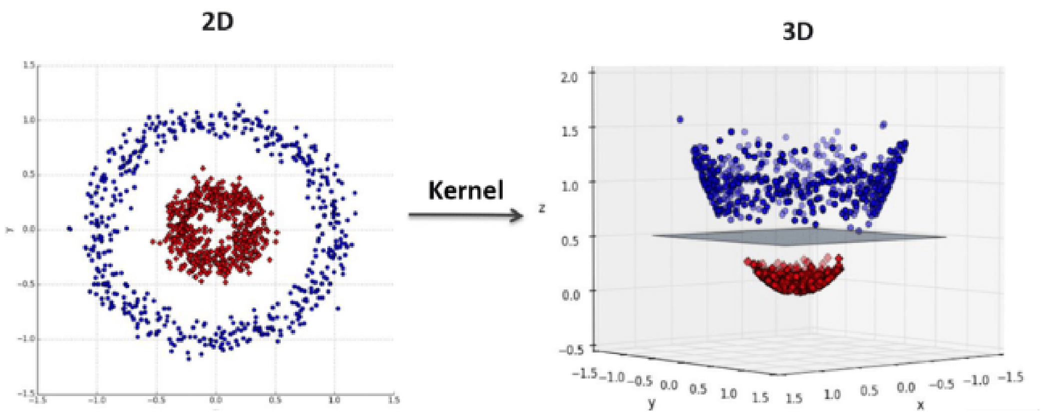
Kernels in SVM

SVM = SVC + Kernels

- Works with **non-linear data**

- Kernel takes your data to **higher dimension**:





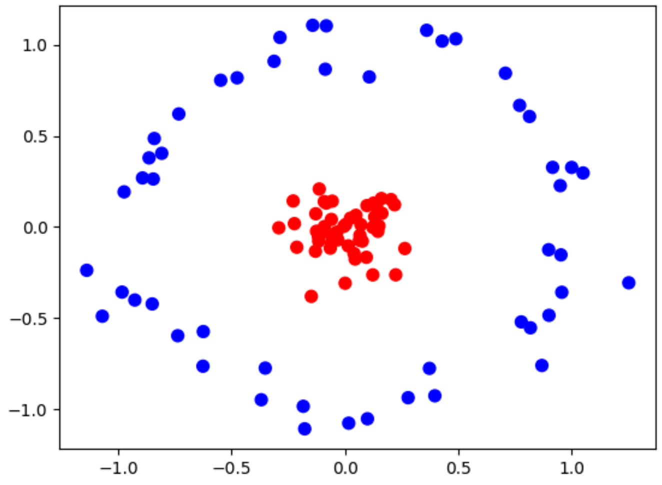
- Data becomes **linearly separable in higher dimension.**
- You apply SVM/SVC in higher dimension & bring the data back to lower dimension.

Kernel Trick – Handling Non-Linearly Separable Data

What if a straight line cannot separate classes?

- SVM uses **kernel functions** to **map data into a higher-dimensional space** where it **becomes separable**.
- **It's called trick because we do not transform the data to higher dimension.**

```
from sklearn.datasets import make_circles
X, y = make_circles(100, factor=.1, noise=.1)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='bwr')
```



`factor=.1`:

- This determines the ratio between the radius of the inner circle and the radius of the outer circle.
- A `factor` of 0 means the inner circle is a point.
- A factor of 1 would mean that the inner and outer circles have the same radius.
- A factor of .1 means the inner circle's radius is very small relative to the outer circle's radius.

`noise=.1`:

- This controls the amount of Gaussian noise added to the data points.
- Noise introduces randomness, making the circles less perfectly formed.

Return Values:

- `X`: A NumPy array of shape `(n_samples, 2)` containing the coordinates of the generated points.
- `y`: A NumPy array of shape `(n_samples,)` containing the class labels (0 or 1) for each point, indicating which circle it belongs to.

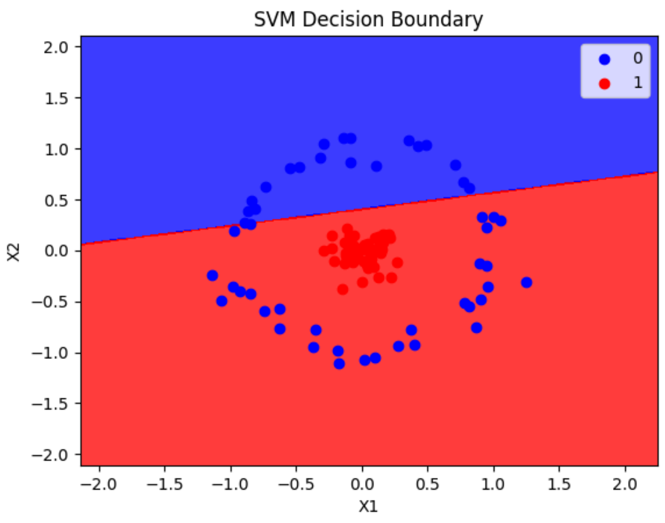
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20)
```

Calculate Accuracy:

```
classifier = SVC(kernel="linear")
classifier.fit(X_train, y_train.ravel())
y_pred = classifier.predict(X_test)

from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)

Output: 0.55
```

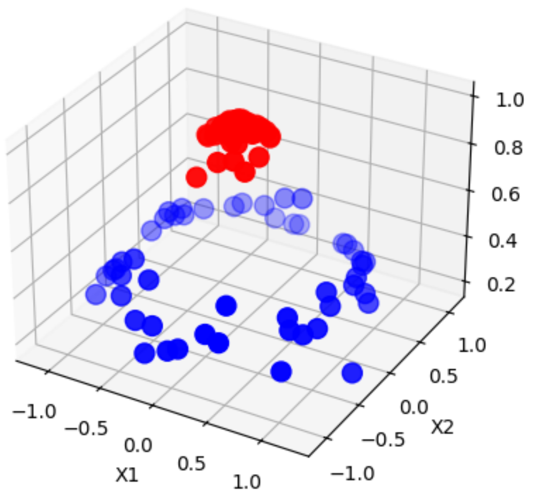
Visualize the boundary:

```
zero_one_colourmap = ListedColormap(('blue', 'red'))
def plot_decision_boundary(X, y, clf):
    X_set, y_set = X, y
    X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1,
                             stop = X_set[:, 0].max() + 1,
                             step = 0.01),
                        np.arange(start = X_set[:, 1].min() - 1,
                           stop = X_set[:, 1].max() + 1,
                           step = 0.01))

    plt.contourf(X1, X2, clf.predict(np.array([X1.ravel(),
                                    X2.ravel()]).T).reshape(X1.shape),
            alpha = 0.75,
            cmap = zero_one_colourmap)
    plt.xlim(X1.min(), X1.max())
    plt.ylim(X2.min(), X2.max())
    for i, j in enumerate(np.unique(y_set)):
        plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
              c = (zero_one_colourmap)(i), label = j)
    plt.title('SVM Decision Boundary')
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.legend()
    return plt.show()

plot_decision_boundary(X, y, classifier)
```



Plot these points in 3D:

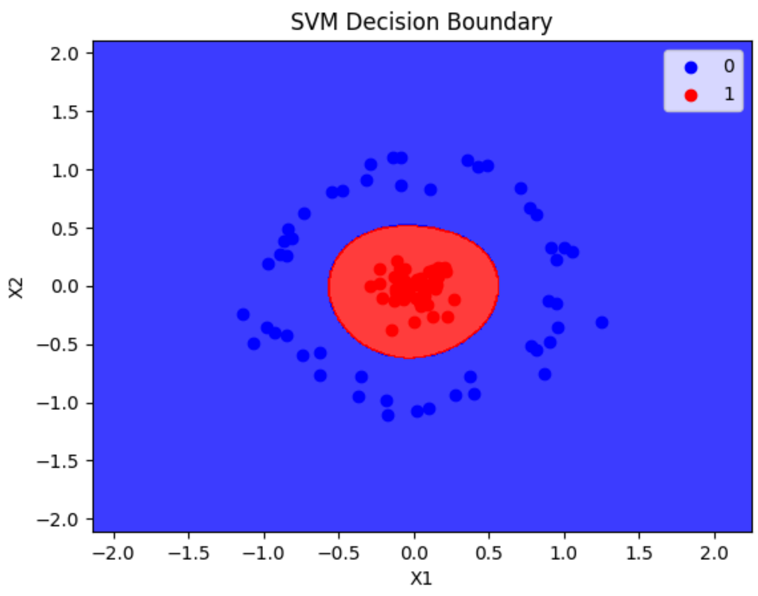
```
def plot_3d_plot(X, y):
    r = np.exp(-(X ** 2).sum(1))
    ax = plt.subplot(projection='3d')
    ax.scatter3D(X[:, 0], X[:, 1], r, c=y, s=100, cmap='bwr')
    ax.set_xlabel('X1')
    ax.set_ylabel('X2')
    ax.set_zlabel('y')
    return ax

  plot_3d_plot(X,y)
```



Now, apply rbf Kernel instead of linear:

```
rbf_classifier = SVC(kernel="rbf")
rbf_classifier.fit(X_train, y_train)
y_pred = rbf_classifier.predict(X_test)

accuracy_score(y_test, y_pred)

Output: 1
```

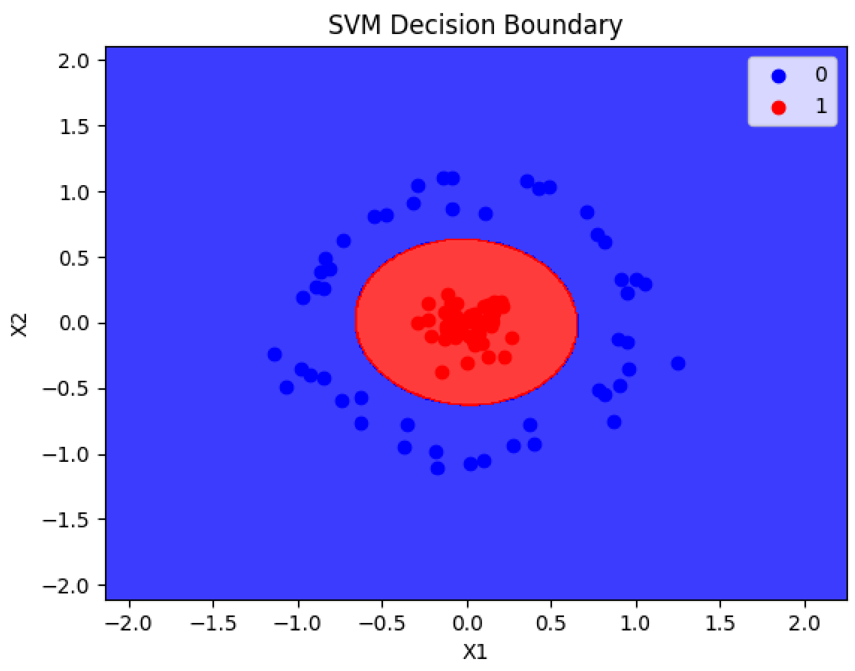
```
plot_decision_boundary(X, y, rbf_classifier)
```



- You can do the same thing with **polynomial kernel, degree 2**

```
poly_classifier = SVC(kernel="poly",degree=2)
poly_classifier.fit(X_train, y_train)
y_pred = poly_classifier.predict(X_test)

accuracy_score(y_test, y_pred)

Output: 1
```

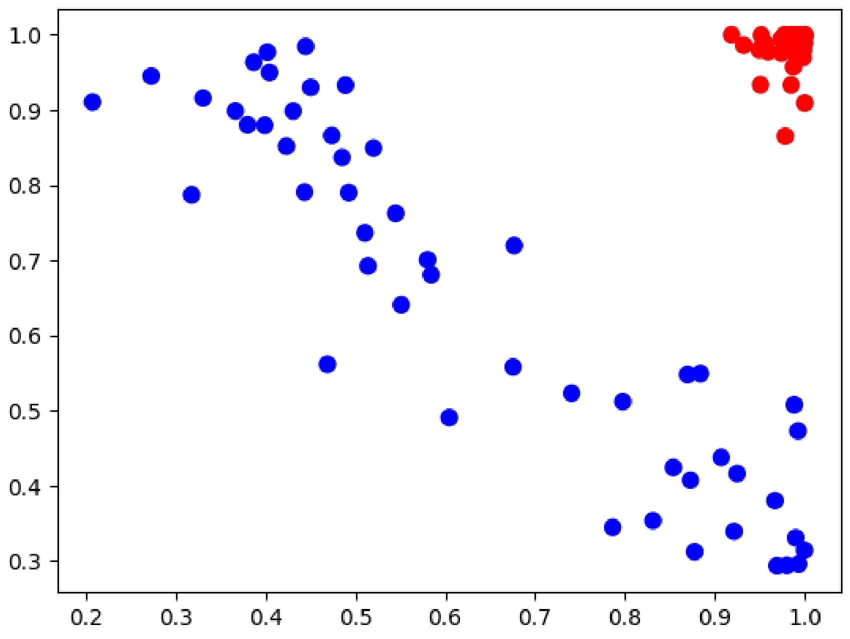
```
plot_decision_boundary(X, y, poly_classifier)
```



```
X_new=np.exp(-(X**2))
```

it computes `e^(-x^2)` for each element `x` in `X`.

```
plt.scatter(X_new[:, 0], X_new[:, 1], c=y, s=50, cmap='bwr')
```



Types of Kernels

1. Linear Kernel
2. Polynomial Kernel
3. Radial Basis Function (RBF) Kernel (**Most Common**)
4. Sigmoid Kernel

1. Linear Kernel

`kernel = 'linear'`

- Simplest kernel, **used when data is linearly separable**.
- No transformation needed, just computes the **dot product**.

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

✅

Fast, simple, and works well when data is already separated by a straight line.

🔷
Use Case: Text classification, linearly separable datasets.

- **Advantages**: Computationally efficient; less prone to overfitting.
- **Disadvantages**: Only suitable for linearly separable data.

2. Polynomial Kernel:

```
SVC(kernel='poly', degree=2)
```

- It maps the data into a higher-dimensional space where it can be linearly separable.

Formula:

$$K(x_i, x_j) = (x_i^T x_j + c)^d$$

where:

- d is the degree of the polynomial.
- c is a constant term (often set to 1).

✅**Captures polynomial relationships between features.**

🔷Use Case:

- Handwritten digit recognition
- Image classification.

Disadvantages: Can be computationally expensive for higher-degree polynomials and prone to overfitting for higher **d**.

3. Radial Basis Function (RBF) Kernel (Gaussian Kernel):

```
SVC(kernel='rbf', C=1.0, gamma=0.1)
```

The RBF kernel is one of the most commonly used kernels in SVM.

- It transforms the data into an infinite-dimensional space, making it **effective for complex, non-linear data.**
- **Maps features to infinite dimensions** for complex decision boundaries.
- Works well when **no prior knowledge about data structure exists**.

Formula:

$$K(x, y) = e^{-\gamma||x-y||^2}$$

✅**Works in most real-world problems.**

🔷Use Case:

- Facial recognition
- Bioinformatics
- Financial predictions

- **Advantages**: Very powerful for non-linear problems, as it works well for data that is not linearly separable.
- **Disadvantages**: Can lead to overfitting if the parameter σ is not properly tuned.

σ\sigma

Hyperparameters:

- `gamma (γ)`: Controls how far the influence of a point extends.
    - **Small γ** → smoother decision boundary (generalizes more).
    - **Large γ** → complex boundary, but risk of overfitting.

4. Sigmoid Kernel

```
kernel = 'sigmoid'
```

- Similar to the activation function used in neural networks.

$$K(x_i, x_j) = \tanh(\alpha x_i^T x_j + c)$$

where:

- α and c are parameters.

✅**Works well for data with neural network-like patterns.**

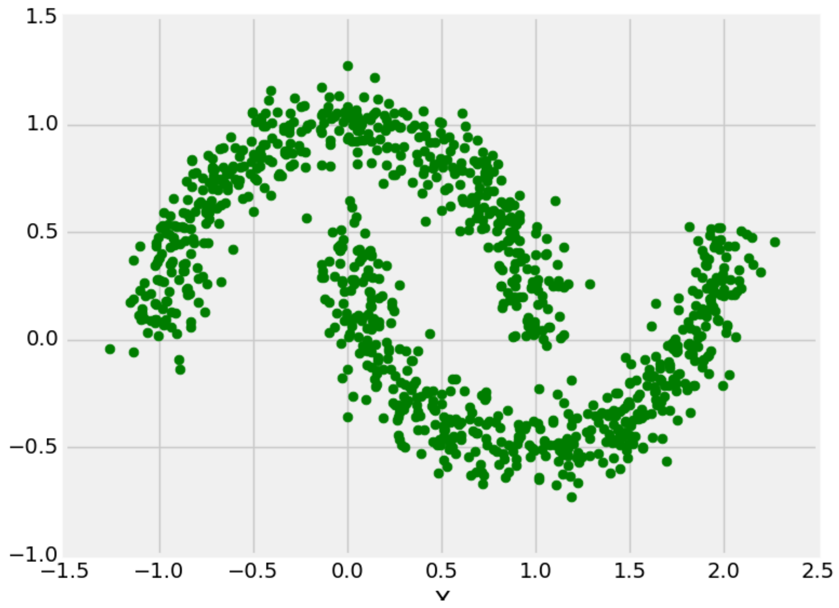
🔷Use Case:

- Biological sequence classification.

Disadvantages: Can be harder to tune and may not perform as well as the RBF kernel in practice.

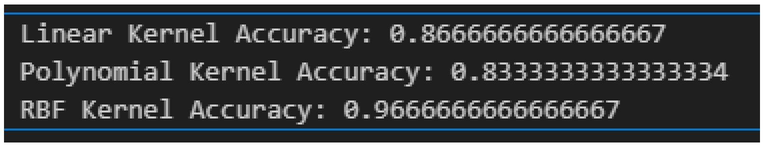
Python Code:

Dataset: `make_moons`



```
from sklearn.svm import SVC
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# Generate non-linear data (moons dataset)
X, y = make_moons(n_samples=100, noise=0.2, random_state=42)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_stat
e=42)

# Create SVM models with different kernels
linear_svm = SVC(kernel='linear')
poly_svm = SVC(kernel='poly', degree=3)
rbf_svm = SVC(kernel='rbf', gamma=0.7)

# Train the models
linear_svm.fit(X_train, y_train)
```

```
poly_svm.fit(X_train, y_train)
rbf_svm.fit(X_train, y_train)

# Make predictions
y_pred_linear = linear_svm.predict(X_test)
y_pred_poly = poly_svm.predict(X_test)
y_pred_rbf = rbf_svm.predict(X_test)

# Evaluate the models
print("Linear Kernel Accuracy:", accuracy_score(y_test, y_pred_linear))
print("Polynomial Kernel Accuracy:", accuracy_score(y_test, y_pred_poly))
print("RBF Kernel Accuracy:", accuracy_score(y_test, y_pred_rbf))
```



```
Linear Kernel Accuracy: 0.8666666666666667
Polynomial Kernel Accuracy: 0.8333333333333334
RBF Kernel Accuracy: 0.9666666666666667
```

Choosing the Right Kernel

Condition	**Best Kernel**
Data is **linearly separable**	**Linear Kernel**
Data has **curved patterns**	**Polynomial Kernel** (low-degree)
Data is **highly complex & non-linear**	**RBF Kernel**
Data behaves like a **neural network**	**Sigmoid Kernel**

Understanding Hyperparameters and Tuning Strategies

Hyperparameter	Effect	Tuning Strategy
C (Regularization Parameter)	Controls how much the model allows misclassifications. Higher C → strict decision	Try logarithmic scale values: 0.01, 0.1, 1, 10, 100

Hyperparameter	Effect	Tuning Strategy
	boundary, lower C → flexible boundary.	
gamma (RBF Kernel Parameter)	Controls how far influence of a single training example extends. Higher gamma → more complex decision boundary.	Try 'scale', 0.01, 0.1, 1
degree (Polynomial Kernel Parameter)	Defines the degree of the polynomial kernel.	Choose 2, 3, 4 (higher degrees increase complexity)

1. **C**:
    - Start with values like `[0.1, 1, 10, 100]`.
    - If the model overfits, reduce CC. If it underfits, increase CC.
2. **Kernel**:
    - Try `linear`, `poly`, and `rbf` kernels.
    - Use `rbf` for highly non-linear data.
3. **Gamma**:
    - For `rbf`, `poly`, and `sigmoid` kernels, try values like `[0.1, 1, 10]` or use `'scale'` and `'auto'`.
    - A smaller γ means a larger influence radius (smoother decision boundary).
    - A larger γ means a smaller influence radius (more complex decision boundary).
        - If **gamma is large (e.g., 10)**, the model will only care about points that are **very close** to the decision boundary.
            - This makes the boundary very **detailed and wiggly.**
            - **Overfitting**
        - If **gamma is small (e.g., 0.01)**, the model will consider points that are **farther away**.
            - This makes the boundary **smoother and less detailed.**

        - **Underfitting**

- `"scale"` **&** `"auto"` **:** These are just **shortcuts** for setting the value of **gamma** without you having to pick a number.

`gamma='scale'` :

- The model automatically sets **gamma** based on the **spread of your data**.
- If your data points are spread out, it will use a smaller gamma.
- If your data points are close together, it will use a larger gamma.
- This is usually a good default choice.

`gamma='auto'` :

- The model sets **gamma** to a fixed value: ***γ= 1/Number of features***
- This is simpler but **may not work as well as `'scale'` for all datasets**.

4. **Degree**:
    - For the polynomial kernel, try degrees like `[2, 3, 4]`.
    - Higher degrees can lead to overfitting.