3.3 - Support Vector Machines

Giovanni Della Lunga
giovanni.dellalunga@unibo.it

Introduction to Machine Learning for Finance

Bologna - February-March, 2025

Introduction

Introduction

- As we have seen, **supervised learning models** play a crucial role in classification tasks.
- This lecture explores two fundamental models:
  - The **Perceptron**, a foundational linear classifier.
  - **Support Vector Machines (SVM)**, an optimization-based approach for classification.

Introduction

- We will cover key concepts such as:
  - Linearly separable datasets and decision boundaries.
  - The geometry of hyperplanes and vector representations.
  - The mathematical formulation of perceptrons and their training algorithms.
  - The SVM framework, including margin maximization and kernel methods for non-linear classification.
- By the end of this lesson, you will understand how these models function and how they can be applied to real-world classification problems.

Introduction

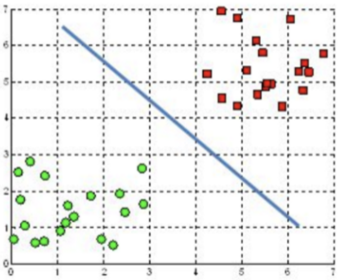
- Like **decision trees**, SVMs can be used for either classification or for the prediction of a continuous variable;
- We first consider **linear classification** where a linear function of the feature values is used to separate observations and in particular we will focus on **binary classification** where the separation is into only two categories;

Linear Separable Set

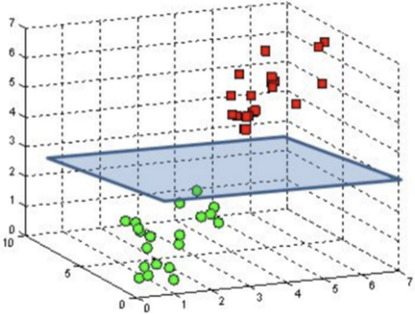
Linear Separation

- **Linearly Separable Data points**: Data points can be said to be linearly separable if a separating boundary/hyperplane can easily be drawn showing distinctively the different class groups.

A hyperplane in $\mathbb{R}^2$ is a line



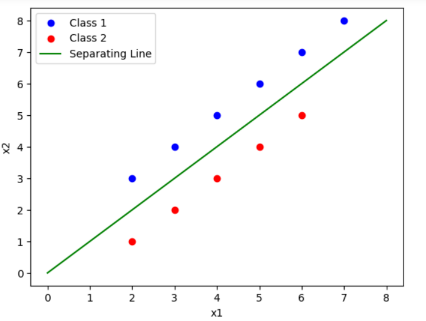
A hyperplane in $\mathbb{R}^3$ is a plane



Linear Separation

Problem Definition:

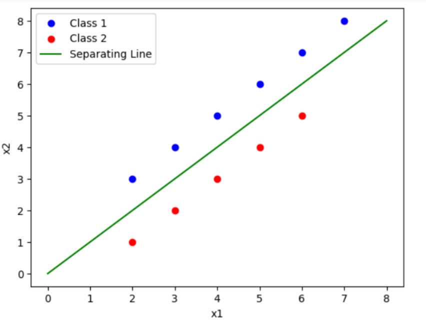
- Consider a simple 2D problem where each data point has two features (x_1 and x_2), and we want to classify them into two categories (let's call them Class 1 and Class 2).



Linear Separation

Problem Definition:

- A line can be drawn to separate the data points of Class 1 from those of Class 2.
  - Class 1: Points that lie on one side of the line (e.g., above the line)
  - Class 2: Points that lie on the other side of the line (e.g., below the line).



Linear Separation

Key Characteristics:

- **2D Case**: In a two-dimensional dataset, the two classes can be separated by a straight line.
- **Line Equation**: The separating line can be defined by the equation:

$$w_1 x_1 + w_2 x_2 + b = 0$$

Where:
- x_1 and x_2 are the features of the data points.
- w_1 and w_2 are the weights (coefficients) that define the orientation of the line.
- b is the bias, which shifts the line vertically.

Geometrical Interpretation of W

Linear Separation

- The set of weights (w_1, w_2) is nothing more than a vector W.
- But what exactly does W represent?
- We can easily verify that the vector $\mathbf{w} = (w_1, w_2)$ is perpendicular to the line $w_1x_1 + w_2x_2 + b = 0$.
- To demonstrate this relationship we can use the following argument based on the properties of the dot product.

Geometrical Interpretation of W

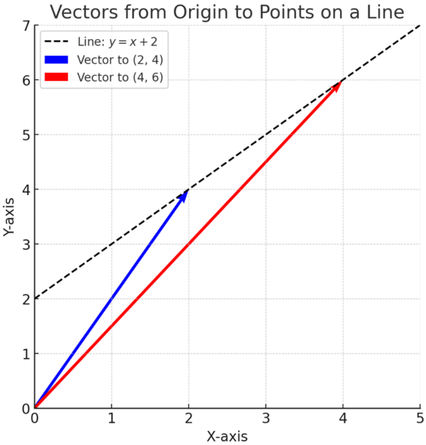
Linear Separation

Step 1: Consider Two Points on the Line

- Let $\mathbf{x} = (x_1, x_2)$ and $\mathbf{y} = (y_1, y_2)$ be any two distinct points on the line.
- Because both points lie on the line, they satisfy the equation:

$$w_1 x_1 + w_2 x_2 + b = 0$$

$$w_1 y_1 + w_2 y_2 + b = 0$$



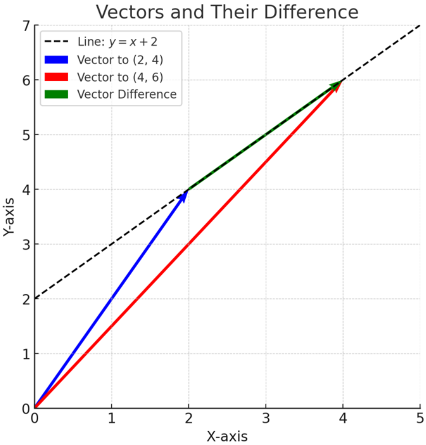
Geometrical Interpretation of W

Linear Separation

Step 2: Form a Tangent Vector to the Line

- A vector that is tangent to the line can be obtained by taking the difference between the two points:

$$\mathbf{v} = \mathbf{y} - \mathbf{x} = (y_1 - x_1,\ y_2 - x_2).$$



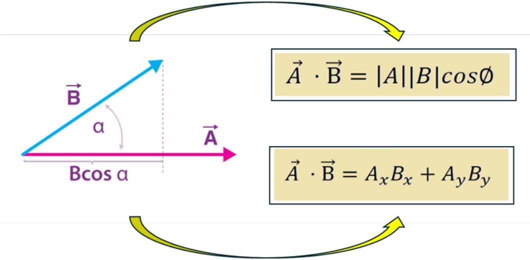
Geometrical Interpretation of W

Linear Separation

Step 3: Show That w is Perpendicular to v

- To show that **w** is perpendicular to the tangent vector **v**, we need to demonstrate that their dot product is zero:

$$\mathbf{w} \cdot \mathbf{v} = w_1(y_1 - x_1) + w_2(y_2 - x_2).$$



Geometrical Interpretation of W

Linear Separation

Step 4: Use the Line Equations

- Since both **x** and **y** lie on the line, we have:

$$w_1x_1 + w_2x_2 + b = 0 \quad \Rightarrow \quad w_1x_1 + w_2x_2 = -b,$$

$$w_1y_1 + w_2y_2 + b = 0 \quad \Rightarrow \quad w_1y_1 + w_2y_2 = -b.$$

- Subtract the first equation from the second:

$$(w_1y_1 + w_2y_2) - (w_1x_1 + w_2x_2) = -b - (-b) = 0.$$

- This simplifies to:

$$\boxed{w_1(y_1 - x_1) + w_2(y_2 - x_2) = 0 = \vec{w} \cdot (\vec{y} - \vec{x}) \quad \blacksquare}$$

Describing Points Above or Below the Line

Linear Separation

Because the vector $\mathbf{w}$ is perpendicular to the line, we can use it to "step off" the line in a controlled way. Specifically, consider any point of the form

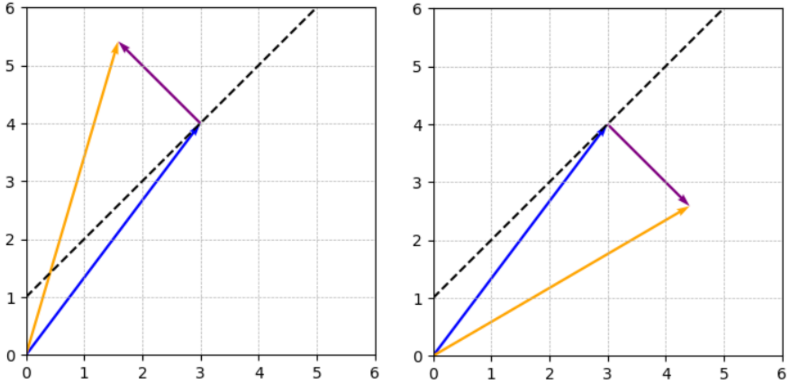
$$\mathbf{x} = \mathbf{x}_0 + \alpha\,\mathbf{u},$$

where:
- $\mathbf{x}_0$ is a point on the line (so that $w_1 x_{1,0} + w_2 x_{2,0} + b = 0$),
- $\mathbf{u}$ is a unit vector in the direction of $\mathbf{w}$, i.e.,

$$\mathbf{u} = \frac{(w_1, w_2)}{\|\mathbf{w}\|},$$

Describing Points Above or Below the Line

Linear Separation



Now, depending on the sign of α, the point $\mathbf{x}$ will lie on one side of the line or the other:

- If $\alpha > 0$, then $\mathbf{x}$ lies in the direction of $\mathbf{w}$ (which we will call "above" the line).
- If $\alpha < 0$, then $\mathbf{x}$ lies in the direction opposite to $\mathbf{w}$ (which we will call "below" the line).

Describing Points Above or Below the Line

Linear Separation

Evaluating the Linear Combination Let's compute the value of

$$f(\mathbf{x}) = w_1 x_1 + w_2 x_2 + b,$$

for the point

$$\mathbf{x} = \mathbf{x}_0 + \alpha\,\mathbf{u}.$$

Substitute $\mathbf{x}$ into f:

$$\begin{aligned} f(\mathbf{x}) &= w_1\big(x_{1,0} + \alpha u_1\big) + w_2\big(x_{2,0} + \alpha u_2\big) + b \\ &= \big(w_1 x_{1,0} + w_2 x_{2,0} + b\big) + \alpha\big(w_1 u_1 + w_2 u_2\big). \end{aligned}$$

Describing Points Above or Below the Line

Linear Separation

Since $\mathbf{x}_0$ lies on the line, the first term is zero:

$$w_1 x_{1,0} + w_2 x_{2,0} + b = 0.$$

Now, consider the second term. Because $\mathbf{u}$ is the unit vector in the direction of $\mathbf{w}$:

$$w_1 u_1 + w_2 u_2 = \mathbf{w} \cdot \mathbf{u} = \|\mathbf{w}\| \, \|\mathbf{u}\| \, \cos(0) = \|\mathbf{w}\| \cdot 1 \cdot 1 = \|\mathbf{w}\|.$$

Thus, we have:

$$f(\mathbf{x}) = \alpha \|\mathbf{w}\|.$$

Describing Points Above or Below the Line

Linear Separation

Interpreting the Result

1. For Points Above the Line:

When $\alpha > 0$, we get

$$f(\mathbf{x}) = \alpha\|\mathbf{w}\| > 0,$$

because $\|\mathbf{w}\|$ is a positive quantity. Hence, all points that are a positive distance α from the line (in the direction of $\mathbf{w}$) yield

$$w_1x_1 + w_2x_2 + b > 0.$$

Describing Points Above or Below the Line

Linear Separation

Interpreting the Result

2. For Points Below the Line:

When $\alpha < 0$, we obtain

$$f(\mathbf{x}) = \alpha \|\mathbf{w}\| < 0,$$

so all points that are a negative distance α from the line (opposite to the direction of $\mathbf{w}$) yield

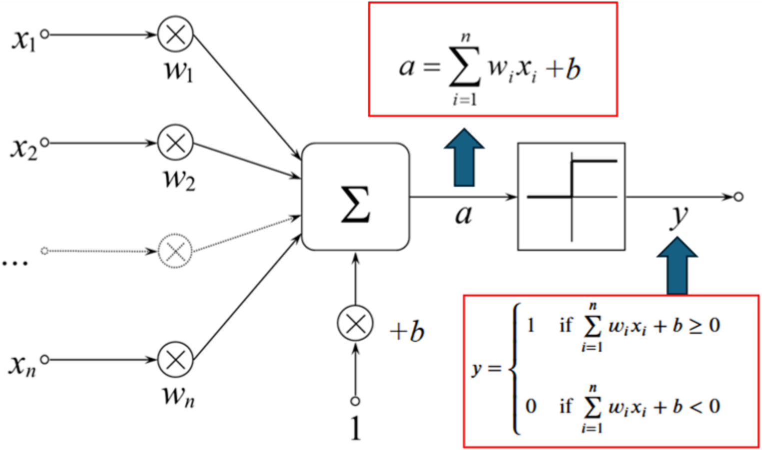
$$w_1 x_1 + w_2 x_2 + b < 0.$$

The Perceptron

The Perceptron

- The **Perceptron** is one of the simplest types of artificial neural networks and is considered a foundational algorithm in machine learning and neural network theory.
- It was introduced by Frank Rosenblatt in 1958 and is primarily used for binary classification tasks, i.e., determining whether an input belongs to one of two classes.
- A Perceptron works by classifying input vectors through a linear decision boundary, which is adjusted during the training phase to minimize classification errors.

Structure of a Perceptron



Training

The Perceptron

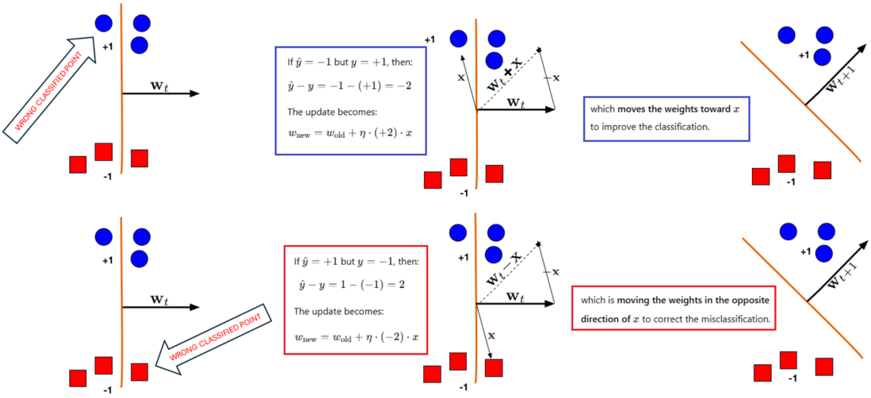
- The goal of training is to adjust the weights and bias to correctly classify the input data.
- The Perceptron uses the following update rule during training:
  1. Initialize the weights and bias to small random values
  2. For each training sample $(x^{(i)}, y^{(i)})$:
  - Calculate the predicted output $\hat{y}^{(i)} = \text{sign}(w \cdot x^{(i)} + b)$
  - If $\hat{y}^{(i)} \neq y^{(i)}$, update the weights and bias:

$$w_i \leftarrow w_i + \eta(y^{(i)} - \hat{y}^{(i)})x_i^{(i)}$$

$$b \leftarrow b + \eta(y^{(i)} - \hat{y}^{(i)})$$

Training

The Perceptron



Prediction

The Perceptron

- Once the **Perceptron** has been trained, it can be used to **make predictions** on new, unseen data.
- The prediction phase involves a **forward pass**, where the perceptron computes an output based on the learned weights.
- The perceptron follows these steps during prediction:

1. **Input Representation:**

- A new data point x is given as an input vector
- Each input feature is represented as x_i.

Prediction

The Perceptron

2. Weighted Sum Calculation:

- The perceptron computes the **weighted sum** of inputs and learned weights:

$$z = \sum w_i x_i + b$$

where:
- w_i are the **learned weights** from training
- x_i are the **input features**
- b (bias) allows shifting the decision boundary.

Prediction

The Perceptron

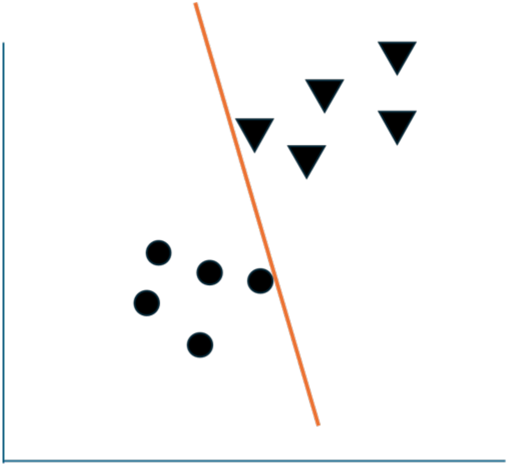
3. Activation Function (Step Function):

- The perceptron applies a **step function** (also called Heaviside function):

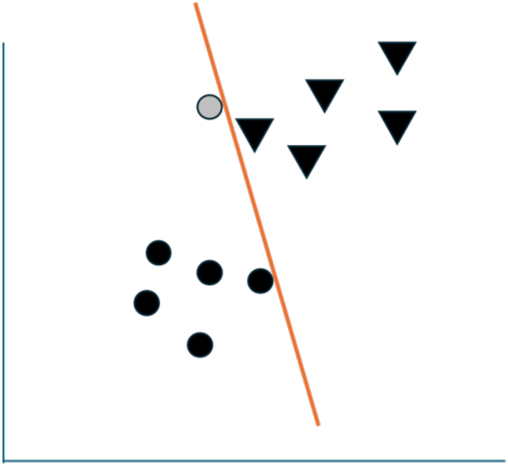
$$y = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

- If z is greater than or equal to 0, the perceptron predicts **Class 1**
- If z is less than 0, the perceptron predicts **Class 0**

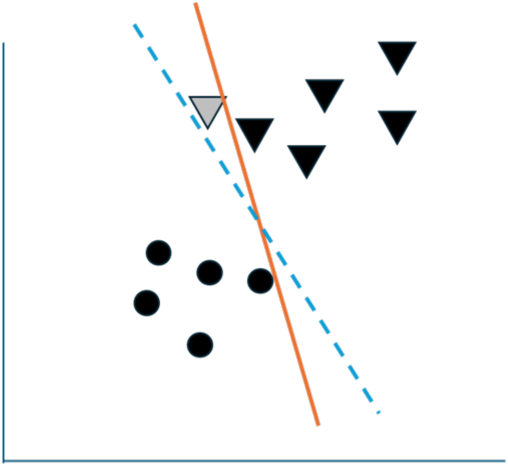
The Problem with the Perceptron



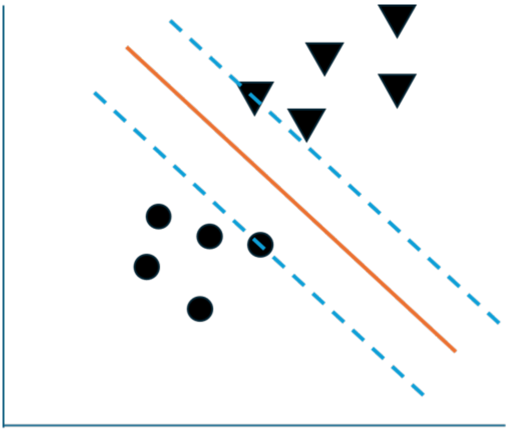
The Problem with the Perceptron



The Problem with the Perceptron



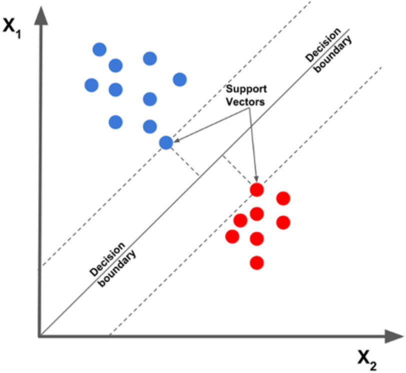
The Problem with the Perceptron



Support Vector Machines (SVM)

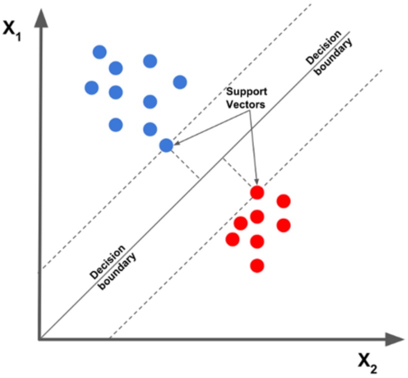
Support Vector Machines

- SVM is an algorithm that takes the data as an input and outputs a line that separates those classes if possible but in this case our optimization objective is **to maximize the margin**.
- The margin is defined as the distance between the separating hyperplane (decision boundary) and the training examples that are closest to this hyperplane, which are the so-called **support vectors**.



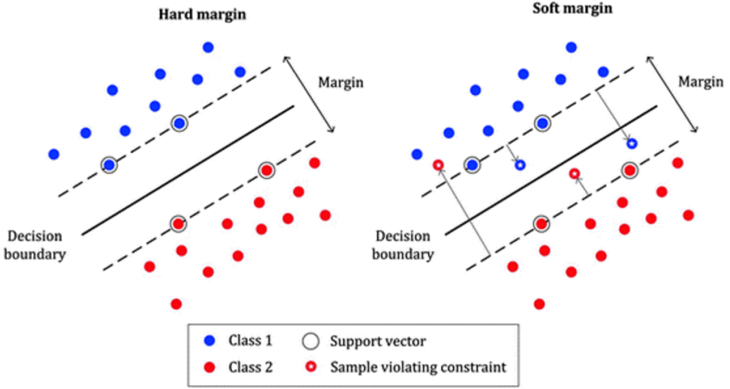
Support Vector Machines

- The rationale behind having decision boundaries with large margins is that they tend to have a lower generalization error, whereas models with small margins are more prone to overfitting.



Support Vector Machines

Hard and Soft Margins. When the data is linearly separable, and we don't want to have any misclassifications, we use SVM with a **hard margin**. However, when a linear boundary is not feasible, or we want to allow some misclassifications in the hope of achieving better generality, we can opt for a **soft margin** for our classifier

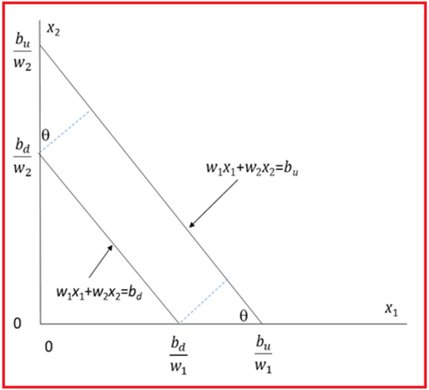


The Algorithm

Support Vector Machine

In the SVM framework, the hyperplanes are defined as:

- **Upper margin**:
  $w_1x_1 + w_2x_2 = b_u,$
- **Lower margin**:
  $w_1x_1 + w_2x_2 = b_d,$
- **Decision boundary** (center):
  $w_1x_1 + w_2x_2 + b = 0.$



The Algorithm
Support Vector Machines

From the equation for the distance between two parallel lines we can write:

$$P = \frac{b_u - b_d}{\sqrt{w_1^2 + w_2^2}}$$

We can scale w_1, w_2, b_u, and b_d by the same constant without changing the model. We can therefore choose a constant α such that $\alpha b_u = b + 1$ and $\alpha b_d = b - 1$. With a bit of algebra we easily find

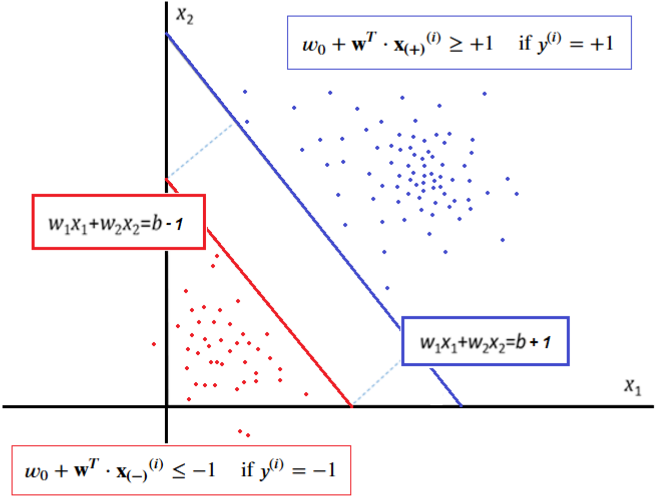
$$\alpha = \frac{2}{b_u - b_d} \quad b = \frac{b_d + b_u}{b_d - b_u}$$

using this scaling we finally have

$$\boxed{P = \frac{2}{\sqrt{w_1^2 + w_2^2}} = \frac{2}{||\mathbf{w}||^2}}$$

The Algorithm

Support Vector Machines



The Algorithm

Support Vector Machines

- In the **hard margin** case the algorithm minimizes $w_1^2 + w_2^2$ subject to **perfect separation** being achieved
- Now, the objective function of the SVM becomes the maximization of this margin by maximizing P under the constraint that the examples are classified correctly, which can be written as:

$$w_0 + \mathbf{w}^T \cdot \mathbf{x}_{(+)}{}^{(i)} \geq +1 \quad \text{if } y^{(i)} = +1$$

$$w_0 + \mathbf{w}^T \cdot \mathbf{x}_{(-)}{}^{(i)} \leq -1 \quad \text{if } y^{(i)} = -1$$

for $i = 1, \ldots, N$. Here, N is the number of examples in our dataset and $w_0 = b$.

The Algorithm

Support Vector Machines

$$w_0 + \mathbf{w}^T \cdot \mathbf{x}_{(+)}{}^{(i)} \geq +1 \quad \text{if } y^{(i)} = +1$$
$$w_0 + \mathbf{w}^T \cdot \mathbf{x}_{(-)}{}^{(i)} \leq -1 \quad \text{if } y^{(i)} = -1$$

These two equations basically say that all negative-class examples should fall on one side of the negative hyperplane, whereas all the positive-class examples should fall behind the positive hyperplane, which can also be written more compactly as follows:

$$y^{(i)} \left(w_0 + \mathbf{w}^T \cdot \mathbf{x}^{(i)} \right) \geq 1 \quad \forall i \tag{1}$$

Step 1: Primal Formulation

Support Vector Machines

The **hard-margin SVM** optimization problem:

$$\min_{w,b} \quad \frac{1}{2}\|w\|^2 \tag{2}$$

subject to:

$$y_i(w^T x_i + b) \geq 1, \quad \forall i = 1, \ldots, n. \tag{3}$$

Where:

- w is the weight vector
- b is the bias
- $y_i \in \{-1, +1\}$ are class labels
- x_i are training samples

The constraint ensures correct classification.

Step 2: Construct the Lagrangian

Support Vector Machines

Introducing **Lagrange multipliers** $\alpha_i \geq 0$:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{n} \alpha_i \left(y_i(w^T x_i + b) - 1 \right). \qquad (4)$$

Why? The constraints are incorporated into the optimization problem.

Step 3: Solve for w and b (Primal Variables)

Optimal w: Differentiate $\mathcal{L}$ w.r.t. w:

$$\frac{\partial \mathcal{L}}{\partial w} = w - \sum_{i=1}^{n} \alpha_i y_i x_i = 0. \tag{5}$$

Solving for w:

$$w = \sum_{i=1}^{n} \alpha_i y_i x_i. \tag{6}$$

Optimal b: Differentiate w.r.t. b:

$$\sum_{i=1}^{n} \alpha_i y_i = 0. \tag{7}$$

Step 4: Substitute into the Lagrangian

Support Vector Machines

Substituting w:

$$\mathcal{L}(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j (x_i^T x_j). \tag{8}$$

This function depends only on α_i.

Step 5: Dual Formulation (Maximization)

Support Vector Machines

Final dual problem:

$$\max_{\alpha} \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j (x_i^T x_j). \tag{9}$$

Subject to:

$$\sum_{i=1}^{n} \alpha_i y_i = 0, \quad \alpha_i \geq 0 \quad \forall i. \tag{10}$$

Step 6: Why Does Minimization Become Maximization?

Support Vector Machines

- The **primal problem** minimizes a quadratic function with constraints.
- Using Lagrange multipliers, constraints are incorporated into a single function.
- **Minimizing over primal variables** transforms the problem into **maximization** over dual variables (α_i).
- The final dual problem is a **quadratic maximization** problem, easier to solve.

The Algorithm

Support Vector Machines

Once we obtain α_n, the **support vectors** correspond to the nonzero values of α_n. The weight vector is then computed as:

$$\mathbf{w} = \sum_{n=1}^{N} \alpha_n y^{(n)} \mathbf{x}^{(n)}$$

To compute w_0, we use any support vector $\mathbf{x}^{(s)}$ such that $\alpha_s > 0$:

$$w_0 = y^{(s)} - \mathbf{w}^T \mathbf{x}^{(s)}$$

The Algorithm

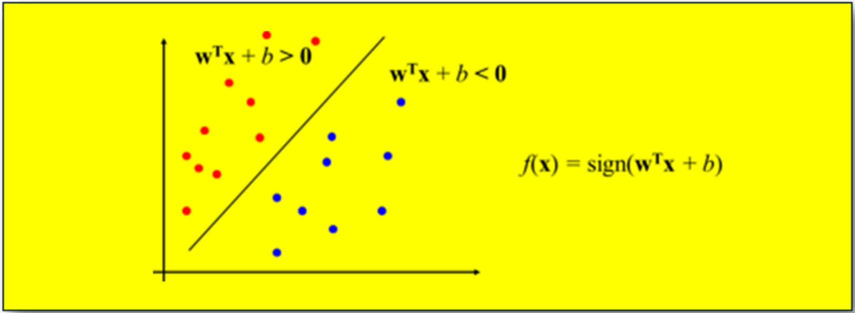
Support Vector Machines

Prediction Phase

To classify a new point $\mathbf{x}_{\text{new}}$, we compute:

$$f(\mathbf{x}_{\text{new}}) = \mathbf{w}^T \mathbf{x}_{\text{new}} + w_0$$

If $f(\mathbf{x}_{\text{new}}) > 0$, classify as $+1$, otherwise classify as -1.



The Algorithm

Support Vector Machines

The decision function for a linear SVM is:

$$f(x) = \sum_i \alpha_i y_i (x_i \cdot x) + b$$

where: x_i are the support vectors, y_i are their class labels ($+1$ or -1), α_i are the Lagrange multipliers obtained from solving the optimization problem, x is the new point we want to classify and b is the bias term

The **only mathematical operation involving data points is the dot product** $x_i \cdot x$

> Nowhere in the formulation do we explicitly require the coordinates of vectors in some space!

Support Vector Machines

Soft Margin Optimization

- Let's briefly mention the slack variable, ξ , which was introduced by Vladimir Vapnik in 1995 and led to the so-called soft-margin classification.
- The motivation for introducing the slack variable, ξ , was that the linear constraints need to be relaxed for nonlinearly separable data to allow the convergence of the optimization in the presence of misclassifications, under appropriate cost penalization.

Support Vector Machines

The positive-valued slack variable is simply added to the linear constraints:

$$b + \mathbf{w}^T \cdot \mathbf{x}^{(i)} \geq 1 - \xi^{(i)} \quad \text{if } y^{(i)} = 1$$

$$b + \mathbf{w}^T \cdot \mathbf{x}^{(i)} \leq -1 + \xi^{(i)} \quad \text{if } y^{(i)} = -1$$

So, the new objective to be minimized (subject to the contraints) becomes

$$\frac{1}{2}||\mathbf{w}||^2 + C\left(\sum_i \xi^{(i)}\right) \tag{11}$$

Via the variable, C, we can then control the penalty for misclassification. Large values of C correspond to large error penalties, whereas we are less strict about misclassification errors if we choose smaller values for C. We can then use the C parameter to control the width of the margin and therefore tune the bias-variance tradeoff.

Subsection 1

SVM in Scikit-Learn

Creating a Dataset

SVM in Scikit-Learn

Scikit-Learn provides utilities to generate synthetic datasets. This is useful for testing models before applying them to real data. We can create a classification dataset with:

```
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=100, n_features=2,
                           n_classes=2, random_state=42)
```

This will create a dataset with 100 samples and 2 features, ideal for visualization. To split the dataset into training and testing:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test
= train_test_split(X, y, test_size=0.2, random_state=42)
```

We now have 80 samples for training and 20 for testing.

Training an SVM Model

SVM in Scikit-Learn

To train an SVM model, we use the ‘SVC‘ class from Scikit-Learn:

```
from sklearn.svm import SVC
svm_model = SVC(kernel='linear', C=1.0)
svm_model.fit(X_train, y_train)
```

This fits the model to the training data. The ‘kernel‘ parameter defines how the decision boundary is computed. The ‘C‘ parameter controls regularization, where a lower value allows more misclassifications but results in a smoother boundary, while a higher value enforces stricter separation of classes.

Making Predictions and Evaluating the Model

SVM in Scikit-Learn

Once trained, the model can predict on new data:

```
y_pred = svm_model.predict(X_test)
```

To evaluate performance, we measure accuracy:

```
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')
```

This gives the percentage of correctly classified test samples. Additional metrics such as precision, recall, and confusion matrix can provide deeper insights.

Visualizing Decision Boundaries

SVM in Scikit-Learn

A useful way to understand how SVM classifies data is by plotting decision boundaries:

```
import numpy as np
import matplotlib.pyplot as plt

xx, yy = np.meshgrid(np.linspace(X[:, 0].min(), X[:, 0].max(), 100),
                     np.linspace(X[:, 1].min(), X[:, 1].max(), 100))
Z = svm_model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, alpha=0.3)
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')
plt.show()
```

This graphically represents how the SVM separates the two classes.

Tuning SVM Hyperparameters

SVM in Scikit-Learn

SVM performance depends on hyperparameters such as ‘C‘, ‘kernel‘, and ‘gamma‘ (for non-linear kernels). A good way to find optimal values is using grid search:

```
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.1, 1, 10], 'gamma': [0.1, 1, 10], 'kernel': ['rbf']}
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
print(grid_search.best_params_)
```

Grid search automates the process of testing multiple hyperparameter combinations and finds the best performing one based on cross-validation.

Conclusion

SVM in Scikit-Learn

Scikit-Learn simplifies the implementation of SVMs, allowing for quick experimentation and tuning. To summarize:

- Use ‘SVC‘ to define and train an SVM model.
- Adjust ‘C‘ and ‘kernel‘ to improve classification results.
- Evaluate using accuracy, precision, recall, and visualization.
- Fine-tune using ‘GridSearchCV‘ for better performance.

By experimenting with these steps, you can optimize an SVM model for your specific dataset.

Subsection 2

The Kernel Trick

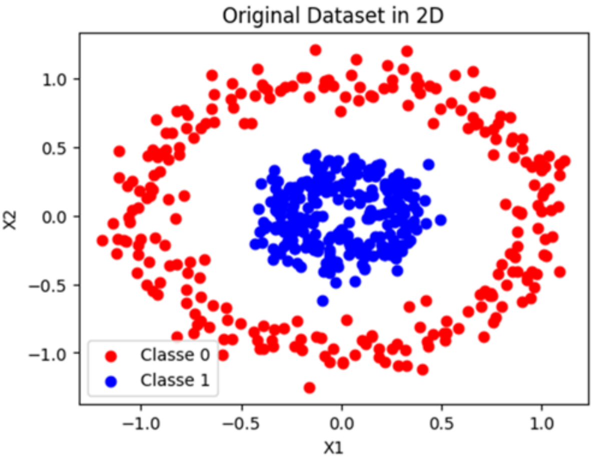
Non Linear Separable Dataset

The Kernel Trick

- As we have seen, the effectiveness of SVMs largely depends on their ability to find an optimal decision boundary, which is typically a hyperplane in a high-dimensional feature space.
- However, many real-world problems involve data that is not linearly separable in its original space.
- As we are going to study in this section, to address this limitation, we can transform the problem into a higher-dimensional space where a linear separation becomes possible.
- This transformation is achieved through a mapping function $\phi(x)$ that projects the data into a new feature space.

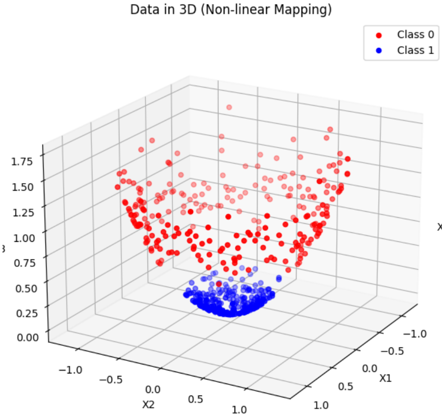
Non Linear Separable Dataset

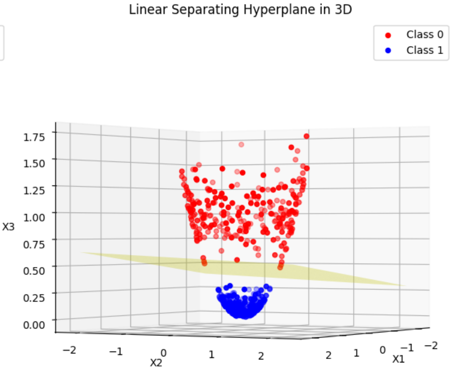
The Kernel Trick



Non Linear Separable Dataset

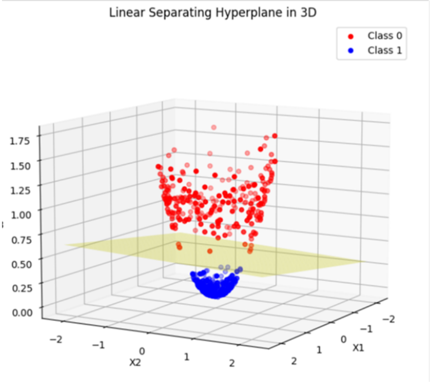
The Kernel Trick

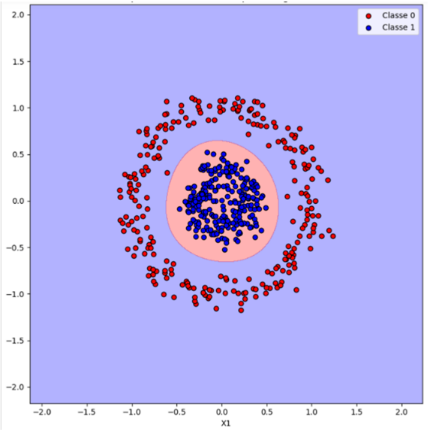




Non Linear Separable Dataset

The Kernel Trick





Projecting back on the Original Space

Non Linear Separable Dataset

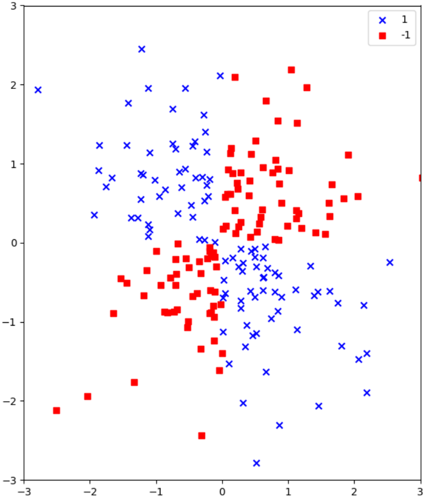
The Kernel Trick

- The challenge with this approach is that explicitly computing the transformation can be **computationally expensive** or even infeasible.
- Furthermore, for the problem we just tackled, it was not difficult to find a third feature that allowed us to separate the data into two groups. But what happens if the 2D data looks like the figure on the right?

Non Linear Separable Dataset

The Kernel Trick

- The challenge with this approach is that explicitly computing the transformation can be **computationally expensive** or even infeasible.
- Furthermore, for the problem we just tackled, it was not difficult to find a third feature that allowed us to separate the data into two groups. But what happens if the 2D data looks like the figure on the right?



Mapping to Higher Dimensions

The Kernel Trick

- Now it's not immediately clear what to choose as the third feature
- $x_1^2 + x_2^2$ no longer works.
- We need a rigorous method to project the data into higher dimensions.
- It must work even if the low-dimensional space is itself much higher than 2D (and thus impossible to visualize).
- Moreover, once the data is projected into higher dimensions, finding a linearly separable hyperplane in the augmented space requires **computing the dot product of vectors in higher dimensions**, which, as we have said, can become computationally unmanageable.

Two Main Goals

The Kernel Trick

So, in some way, the algorithm must simultaneously achieve two things:

1. **Create new features** so that they can be mapped into a higher-dimensional space, and
2. **Avoid performing dot products** in this new space while still being able to find the separating hyperplane.

Mapping to Higher Dimensions

The Kernel Trick

- Let's start with data in two dimensions and map it into three dimensions using three features.
- Given a vector x in the lower-dimensional space (2D in our case), it is mapped to the vector $\phi(x)$ in the higher-dimensional space (3D in our case):

$$x \rightarrow \phi(x)$$

- Our mapping is as follows:

$$[x_1,\ x_2] \rightarrow [x_1^2,\ x_2^2,\ \sqrt{2}x_1x_2]$$

- Thus, if a point a in 2D is given by $[a_1,\ a_2]$ and a point b is given by $[b_1,\ b_2]$, the same points, projected into 3D space, become:

$$[a_1^2,\ a_2^2,\ \sqrt{2}a_1a_2], \quad [b_1^2,\ b_2^2,\ \sqrt{2}b_1b_2].$$

Computational Challenge

The Kernel Trick

- To find a linearly separable hyperplane, we would need to compute the dot products of the points' vectors in the higher-dimensional space.
- In this example, it is not a problem to compute the dot products of all vectors in 3D space.
- Unfortunately, in the real world, the dimensionality of the augmented space can be enormous, making such calculations computationally prohibitive in terms of resources (both time and memory)
- However, Aizerman, Braverman, and Rozonoer devised a very useful trick that completely bypassed this difficulty...

Dot Products in Higher-Dimensional Space

The Kernel Trick

- What if we could perform these calculations using only the lower-dimensional vectors x_i and x_j?
- In other words, what if we could find a function K such that:

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

- This means that passing two lower-dimensional vectors to K should yield the dot product of their higher-dimensional projections.

Example of Feature Mapping

The Kernel Trick

- Given two 2D vectors:

$$a = [a_1 \ a_2], \quad b = [b_1 \ b_2]$$

- Their higher-dimensional transformation is:

$$\phi(a) = [a_1^2, a_2^2, \sqrt{2}a_1 a_2]$$
$$\phi(b) = [b_1^2, b_2^2, \sqrt{2}b_1 b_2]$$

- The dot product of the transformed vectors is:

$$\begin{aligned}\phi(a) \cdot \phi(b) &= [a_1^2, a_2^2, \sqrt{2}a_1 a_2] \cdot [b_1^2, b_2^2, \sqrt{2}b_1 b_2] \\ &= a_1^2 b_1^2 + a_2^2 b_2^2 + 2a_1 a_2 b_1 b_2\end{aligned}$$

Kernel Function Definition

The Kernel Trick

We need a function K that produces the same result:

$$K(x, y) = (x \cdot y)^2$$

Applying this function to a and b:

$$\begin{aligned} K(a, b) &= (a \cdot b)^2 = ([a_1, a_2] \cdot [b_1, b_2])^2 \\ &= (a_1 b_1 + a_2 b_2)^2 \\ &= a_1^2 b_1^2 + a_2^2 b_2^2 + 2 a_1 a_2 b_1 b_2 \end{aligned}$$

So:

$$K(a, b) = \phi(a) \cdot \phi(b)$$

Polynomial Kernel

The Kernel Trick

- If a and b were 100-dimensional and $\phi(a), \phi(b)$ had a million dimensions, we could compute dot products without explicitly working in the large space. **The function K is called a "kernel function."**
- One kernel function used in SVC is the **polynomial kernel**, introduced by Tomaso Poggio in 1975:

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

where c and d are constants. If $c = 0$ and $d = 2$, we recover:

$$K(x, y) = (x \cdot y)^2$$

Example: Polynomial Kernel with $c = 1$, $d = 2$

The Kernel Trick

Let's use:

$$K(x, y) = (1 + x \cdot y)^2$$

For 2D vectors:

$$a = [a_1, a_2], \quad b = [b_1, b_2]$$

We compute:

$$\begin{aligned} K(a, b) &= (1 + a_1 b_1 + a_2 b_2)^2 \\ &= 1 + a_1^2 b_1^2 + a_2^2 b_2^2 + 2a_1 a_2 b_1 b_2 + 2a_1 b_1 + 2a_2 b_2 \end{aligned}$$

Feature Mapping for Polynomial Kernel

The Kernel Trick

- The required mapping $\phi(x)$ is:

$$[x_1, x_2] \rightarrow [1, x_1^2, x_2^2, \sqrt{2}x_1 x_2, \sqrt{2}x_1, \sqrt{2}x_2]$$

- Thus:

$$a \rightarrow [1, a_1^2, a_2^2, \sqrt{2}a_1 a_2, \sqrt{2}a_1, \sqrt{2}a_2]$$
$$b \rightarrow [1, b_1^2, b_2^2, \sqrt{2}b_1 b_2, \sqrt{2}b_1, \sqrt{2}b_2]$$

- Compute $\phi(a) \cdot \phi(b)$:

$$\begin{aligned}
&[1, a_1^2, a_2^2, \sqrt{2}a_1 a_2, \sqrt{2}a_1, \sqrt{2}a_2] \cdot [1, b_1^2, b_2^2, \sqrt{2}b_1 b_2, \sqrt{2}b_1, \sqrt{2}b_2] \\
&= 1 + a_1^2 b_1^2 + a_2^2 b_2^2 + 2a_1 a_2 b_1 b_2 + 2a_1 b_1 + 2a_2 b_2 \\
&= K(a, b)
\end{aligned}$$

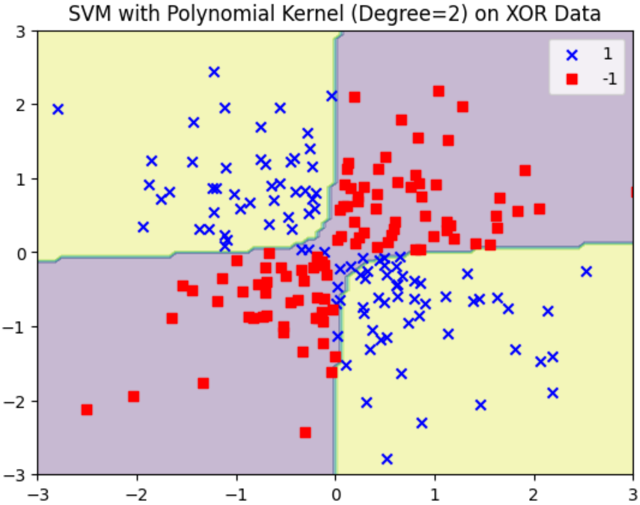
The Kernel Trick

- The kernel function allows us to compute dot products in higher dimensional space efficiently
- The math of Support Vector Machines (SVMs) **depends only on the dot product (scalar product) of vectors and not on their explicit representation in the original or higher-dimensional space**

When using the kernel trick in the context of Support Vector Machines (SVMs), there is no need to explicitly compute the mapping function that transforms data into a higher-dimensional space because the kernel trick leverages the mathematical properties of inner products.

Non Linear Separable Dataset

The Kernel Trick



RBF Kernel

The Kernel Trick

- The Radial Basis Function (RBF) kernel is one of the most widely used kernels in Support Vector Machines (SVMs).
- The RBF kernel **maps data into an infinite-dimensional space**, making it possible to find a linear separation in this transformed space.
- It adapts to complex decision boundaries better than polynomial kernels.

Mathematical Definition of the RBF Kernel

The Kernel Trick

The RBF kernel is defined as:

$$K(x, y) = \exp\left(-\gamma \|x - y\|^2\right)$$

where:
- x and y are input vectors
- γ (gamma) is a hyperparameter that controls the spread of the kernel

The function measures **how similar** two points are: the closer they are, the larger the kernel value.

Intuition Behind the RBF Kernel

The Kernel Trick

- The kernel function gives **high values for nearby points** and **low values for distant points**
- If x and y are close, then $\|x - y\|^2$ is small, making $K(x, y)$ close to 1
- If x and y are far apart, $K(x, y)$ approaches 0
- This creates a **localized influence**, meaning that SVMs with RBF kernels adapt well to non-linear structures.

Effect of the Hyperparameter γ

The Kernel Trick

- **Small** γ: The kernel has a **large spread**, meaning distant points influence each other, leading to smoother decision boundaries.
- **Large** γ: The kernel has a **small spread**, meaning each point mostly affects its local neighborhood, leading to highly flexible decision boundaries that may overfit.

Choosing the Right γ

The Kernel Trick

- A very **small** γ leads to **underfitting** (too simple, missing patterns in data).
- A very **large** γ leads to **overfitting** (too complex, fitting noise in data).
- Grid search and cross-validation are commonly used to tune γ

Decision Boundaries with RBF Kernel

The Kernel Trick

- Unlike linear SVMs, RBF-kernel SVMs can create **non-linear decision boundaries**.
- These boundaries adapt to the structure of the data, making them useful in many applications, such as **image recognition and bioinformatics**.

Comparison with Other Kernels

The Kernel Trick

- **Linear Kernel**: Works well if data is already linearly separable
- **Polynomial Kernel**: More flexible but can be computationally expensive for high-degree polynomials
- **RBF Kernel**: Provides **a good balance** between flexibility and efficiency, making it widely used.

Conclusion

The Kernel Trick

- The **RBF kernel** is a powerful tool in SVMs for handling **non-linearly separable data**.
- The **choice of** γ is crucial for model performance.
- By leveraging the **kernel trick**, we efficiently compute in high-dimensional spaces **without explicitly working in them**.
- RBF-kernel SVMs are widely used in **classification and regression tasks** where complex patterns need to be captured.

Conclusions

The Perceptron Model - Recap

- The Perceptron is one of the simplest neural networks, introduced by Frank Rosenblatt in 1958.
- It is a binary classifier that makes predictions using a weighted sum of input features.
- The perceptron algorithm works by iteratively updating weights to minimize classification errors.
- The update rule follows:

$$w \leftarrow w + \eta(y - \hat{y})x$$

  where:
  - w are the model weights.
  - η is the learning rate.
  - y is the true label, and $\hat{y}$ is the predicted label.
- It is effective for linearly separable problems but struggles with non-linearly separable data.

Support Vector Machines - Recap

- The SVM is a powerful classification model that finds the optimal separating hyperplane.
- It maximizes the margin, the distance between the hyperplane and the nearest data points (support vectors).
- The optimization problem for SVM is formulated as: $\min \frac{1}{2}||w||^2$ subject to: $y_i(w \cdot x_i + b) \geq 1, \forall i$
- For non-linearly separable data, SVM uses the kernel trick to map data into a higher-dimensional space.
- Popular kernels include:
  - Linear Kernel: $K(x, x') = x \cdot x'$
  - Polynomial Kernel: $K(x, x') = (x \cdot x' + c)^d$
  - Radial Basis Function (RBF) Kernel: $K(x, x') = e^{-\gamma||x-x'||^2}$

Comparison and Applications

- **Comparison:**
  - Perceptron: Simple, fast, works only for linearly separable data.
  - SVM: More powerful, can handle non-linear data with kernels, but computationally expensive.
- **Applications:**
  - Perceptron: Early neural networks, simple binary classification tasks.
  - SVM: Image classification, text categorization, bioinformatics (e.g., protein classification).
- Choosing between the two depends on the dataset and computational constraints.