Support Vector Classifiers: Linear vs. Kernelized (With Full Python Code)

1. Introduction

Support Vector Machines (SVMs) are one of the most powerful tools in the machine learning toolbox. While they're often used as a black-box model via libraries like scikit-learn, understanding their underlying mechanics can greatly improve your ML intuition. In this blog post, we'll focus specifically on Support Vector Classifiers (SVC)—the classification variant of SVMs.

We'll start by exploring how linear SVCs work, then move on to non-linear classification via the kernel trick. You'll also see full Python implementations of both from scratch using NumPy and CVXOPT.

2. Understanding Linear SVM

2.1 Intuition

Imagine drawing a straight line that separates two classes of data points. But not just any line—we want the one that leaves the widest possible gap between itself and the closest points from each class. This gap is called the **Margin**, the line is called the **Decision Boundary**, and the points that lie closest to the decision boundary (on or in the margin) are known as **Support Vectors**.

Maximizing this margin helps the model generalize better to new, unseen data—even when the training set has noise or outliers. That's the core idea behind Support Vector Machines.

2.2 Math Behind It

The goal of a linear Support Vector Classifier is to find a hyperplane that separates the data with the largest possible margin. Mathematically, this involves solving an optimization problem.

We want to find a weight vector w and bias b such that the decision function:

$f(x) = w^T x + b$

satisfies the following:

- $f(x) = 0$: point x lies exactly on the decision boundary.
- $f(x^+) = 1$: point x lies exactly on the positive margin.
- $f(x^-) = -1$: point x lies exactly on the negative margin.

With that in mind, let's explore how the optimization problem is derived.

Imagine you have two points from different classes, and you want to compute the distance between them. You might suggest using the Euclidean distance, and you'd be right—but here's the catch: we first need to confirm that the two points lie directly opposite each other along the direction perpendicular to the decision boundary. How do we do that?

We assume that the difference between the positive point x^+ and the negative point x^- is some scalar γ (gamma) times the weight vector w. In other words:

$x^+ - x^- = \gamma w$

This assumption aligns the vector connecting the support vectors with the normal vector to the hyperplane, which allows us to compute the margin analytically.

From that, we can say:

$x^+ = \gamma w + x^-$

And remember that:

$f(x^+) = w^T x^+ + b = 1$

Substitute $x^+ = \gamma w + x^-$ into the equation:

$w^T(\gamma w + x^-) + b = 1$

Expanding the expression:

$\gamma w^T w + w^T x^- + b = 1$

We also know that:

$f(x^-) = w^T x^- + b = -1$

Substitute $w^T x^- + b = -1$ into the equation:

$\gamma w^T w - 1 = 1$

Solving for γ:

$\gamma w^T w = 2 \implies \gamma = \frac{2}{w^T w}$

Therefore, our original equation:

$x^+ - x^- = \gamma w$

Becomes:

$x^+ - x^- = \frac{2w}{w^T w}$

Now for the Euclidean distance formula:

$\parallel x^+ - x^- \parallel = \sqrt{(x^+ - x^-)^T(x^+ - x^-)}$

We Substitute $x^+ - x^- = (\frac{2w}{w^Tw})$:

$\parallel x^+ - x^- \parallel = \sqrt{(\frac{2w}{w^Tw})^T(\frac{2w}{w^Tw})}$

$=\sqrt{\frac{4w^tw}{(w^Tw)^2}}$

$=\frac{2}{\sqrt{w^Tw}}$

This expression gives us the margin—the distance between the support vectors on either side of the hyperplane.

Our goal is to maximize this margin, which is equivalent to:

$\max_{w,b} \frac{2}{\|w\|}$

Since the constant 2 doesn't affect the optimization, we can focus on maximizing $\frac{1}{\|w\|}$. However, working with this inverse square root expression is inconvenient when taking derivatives.

To simplify optimization, we instead minimize a function that is **monotonically decreasing** with respect to $\frac{1}{\|w\|}$—specifically, we minimize $\parallel w \parallel^2$. This is valid because minimizing $\parallel w \parallel^2$ achieves the same goal as maximizing $\frac{1}{\|w\|}$.

Then, for mathematical convenience (especially when applying gradient descent), we include a factor of $\frac{1}{2}$, leading to the final objective:

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

Subject to:
$y_i(w^Tx_i + b) \geq 1$ for all $i = 1, 2, \ldots, m$

This is the primal form of the hard-margin SVM optimization problem.

But why is it called a "hard" margin? Because the current formulation tries to classify every single point perfectly, without any margin violations—meaning all data points must lie outside the margins. This is often unrealistic, as real-world data usually includes outliers. Trying to classify such data perfectly tends to lead to overfitting and poor generalization.

To address this, we introduce a slack variable $\zeta_i \geq 0$, which allows the model to "slack" a bit by tolerating some margin violations and even misclassifications. This leads to what is called a soft margin. The optimization problem becomes:

$\min_{w,b,\zeta} \frac{1}{2}||w||^2 + \sum_{i=1}^{m} \zeta_i$

Subject to:
$y_i(w^Tx_i + b) \geq 1 - \zeta_i$ and $\zeta_i \geq 0$, for all $i = 1, 2, \ldots, m$

But now we have a trade-off: we want to make ζ_i as small as possible to reduce violations, but we also want to keep $||w||^2$ as small as possible to maintain a large margin. This is where the regularization parameter C comes in. It controls the balance between these two objectives:

$\min_{w,b,\zeta} \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\zeta_i$

- A higher C imposes a harder margin (fewer violations allowed).
- A smaller C allows a softer margin (more flexibility for better generalization).

To optimize this problem using gradient-based methods, we reformulate it into an unconstrained form. This is done by expressing the slack variable ζ_i in terms of the hinge loss function:

$\zeta_i = \max(0, 1 - y_i(w^T x_i + b))$

This function penalizes violations of the margin constraint:

- If a point is correctly classified and lies outside the margin ($y_i(w^T x_i + b) \geq 1$), then the loss is 0.
- If a point is misclassified or lies within the margin ($y_i(w^T x_i + b) < 1$), then the loss increases linearly as the prediction moves further from the correct side.

Substituting this expression for ζ_i, the objective becomes:

$\min_{w,b} \frac{1}{2}||w||^2 + C\sum_{i=1}^{m} max(0, 1 - y_i(w^T x_i + b))$

This is the final form used in many SVM implementations. It is differentiable almost everywhere and can be efficiently optimized using gradient descent or specialized solvers like SGD or SMO.

3. From Theory to Code: Linear SVC

Now that we understand the theory behind linear SVMs, let's implement one from scratch using NumPy. This implementation uses batch gradient descent to minimize hinge loss with L2 regularization.

> You can find my `gradient_descent` implementation in the utils folder, but it simply applies batch updates using gradients.

```
import numpy as np

from utils.optimizers import gradient_descent

from .base_model import BaseModel

```

```
class LinearSVC(BaseModel):

    def __init__(self, C=1.0, learning_rate=1e-6, max_iter=1000, tol=1e-4) -
> None:

        self._C = C

        self._learning_rate = learning_rate

        self._max_iter = max_iter

        self._tol = tol

        self._w = None

        self._b = None

        self._support_indices = None



    def _hinge_loss_gradient(self, params, X, y):

        w, b = params



        margin = y * (X @ w + b)

        mask = margin < 1



        self._support_indices = np.where(margin <= 1)[0]



        dw = w - self._C * y[mask] @ X[mask]

        db = -self._C * np.sum(y[mask])

        return np.array([dw, db], dtype=object)
    def fit(self, X, y):

        if len(np.unique(y)) != 2:

            raise ValueError("Linear SVC only supports binary 
classification.")

        X, y = self._validate_transform_input(X, y)

        y = np.where(y <= 0, -1, 1)

        n = X.shape[1]



        self._w = np.zeros(n)

        self._b = 0.0



        self._w, self._b = gradient_descent(

            gradient_fn=self._hinge_loss_gradient,

            params=np.array([self._w, self._b], dtype=object),

            learning_rate=self._learning_rate,

            max_iter=self._max_iter,

            tol=self._tol,

            features=X,

            labels=y

        )
        return self

    def predict(self, X):

        X, _ = self._validate_transform_input(X)

        if self._w is None or self._b is None:

            raise ValueError("Model has not been fitted yet. Call 'fit'
before 'predict'.")

        predictions = X @ self._w + self._b

        return np.where(predictions >= 0, 1, 0)
```

3.1 Initializing the Model

The constructor allows us to configure:

- `C` : Regularization strength (higher = less regularization)
- `learning_rate` : Step size for gradient descent
- `max_iter` : Maximum number of iterations for gradient descent
- `tol` : Gradient norm threshold for early stopping

These hyperparameters give you control over optimization and generalization.

3.2 Hinge loss gradient

The `_hinge_loss_gradient` method:

- Takes in the new weights and bias
- Computes the margin for each data point
- Keeps track of support vectors
- Selects only the points that violate the margin (i.e., those with $y_i(w^T x + b) < 1$)
- Computes the gradient of the regularized hinge loss using only the violating points
- Returns these gradients so that `gradient_descent` can update the parameters

3.3 Fitting the Model

The `fit` method:

- Converts the labels from {0, 1} to {-1, 1} (as required by hinge loss)
- Initializes the weights and bias
- Runs gradient descent to optimize the parameters

3.4 Making Predictions

The `predict` method:

- Applies the learned weights and bias to new inputs
- Returns class predictions based on the sign of the decision function

4. Custom vs scikit-learn: A Side-by-Side Comparison

To validate my custom `LinearSVC`, I compared it against two scikit-learn models — each selected for a different reason:

- ✅ **`sklearn.svm.LinearSVC`** was used for evaluating accuracy and cross-validation metrics, since it optimizes the same linear SVM objective function.
- 🎯 **`sklearn.svm.SVC(kernel="linear")`** was used for visualizing decision boundary and support vectors, as `LinearSVC` does not expose them (support vectors), but `SVC` does.

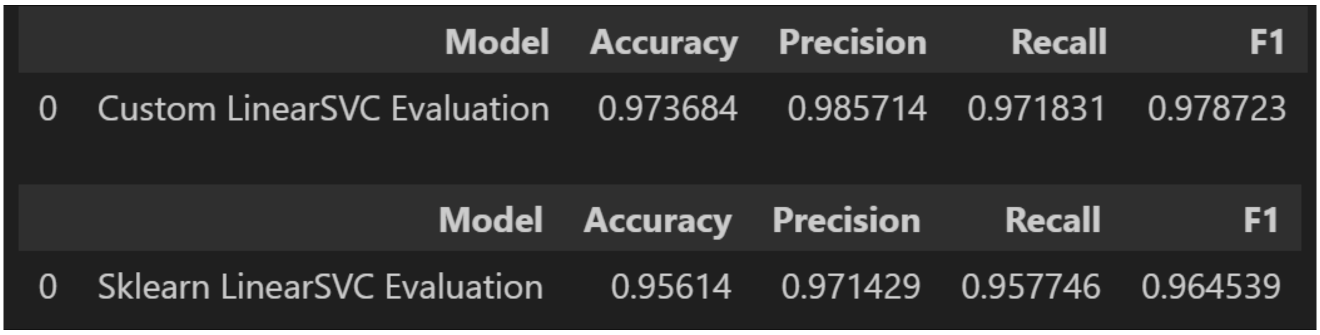
For evaluation, the models were trained on the `breast_cancer` dataset from scikit-learn. For visualization, I used a synthetic dataset generated via `make_classification`.

4.1 Evaluation and Cross-Validation (vs. `LinearSVC`)

I compared the models based on:

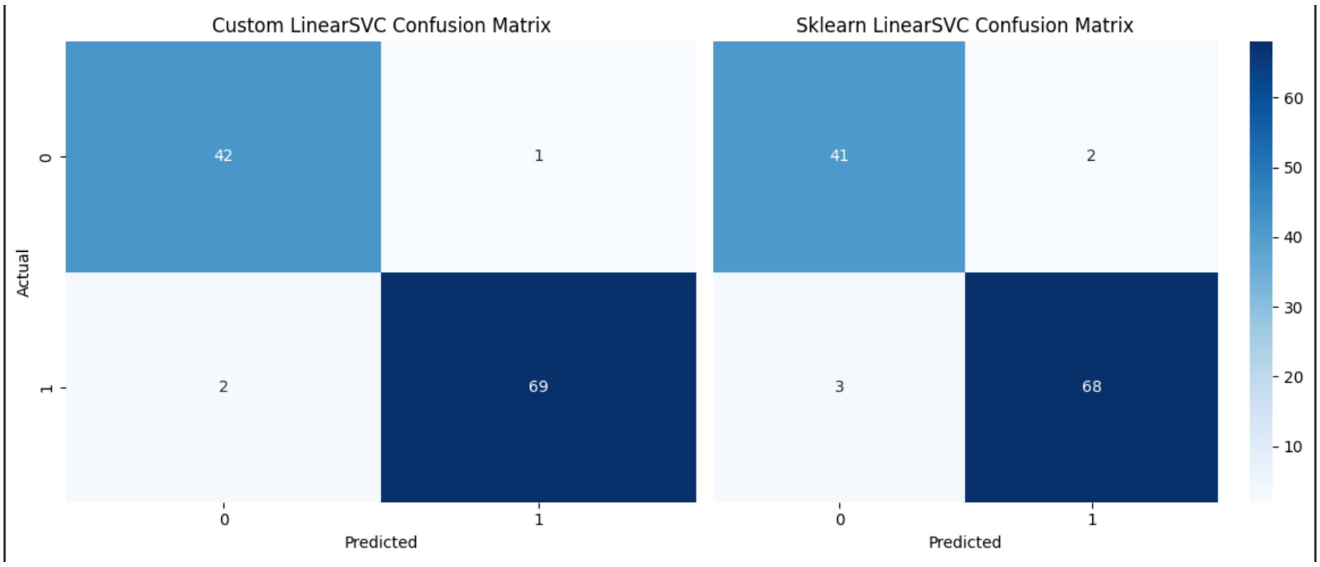
- Accuracy, Precision, Recall, and F1 Score
- 5-Fold Cross-Validation performance
- Decision Boundary Similarity

Train-Test Split Evaluation

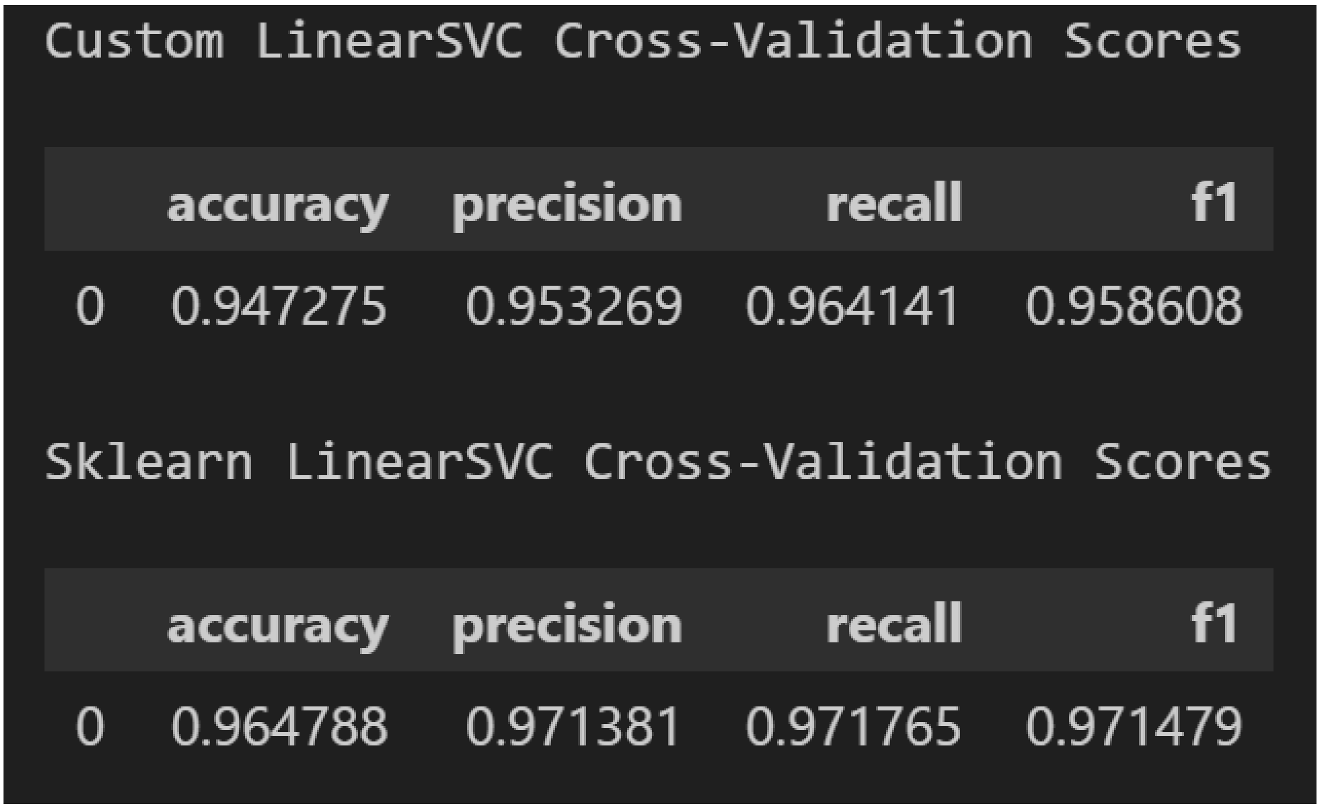


	Model	Accuracy	Precision	Recall	F1
0	Custom LinearSVC Evaluation	0.973684	0.985714	0.971831	0.978723

	Model	Accuracy	Precision	Recall	F1
0	Sklearn LinearSVC Evaluation	0.95614	0.971429	0.957746	0.964539



5-Fold Cross-Validation Evaluation



Custom LinearSVC Cross-Validation Scores

	accuracy	precision	recall	f1
0	0.947275	0.953269	0.964141	0.958608

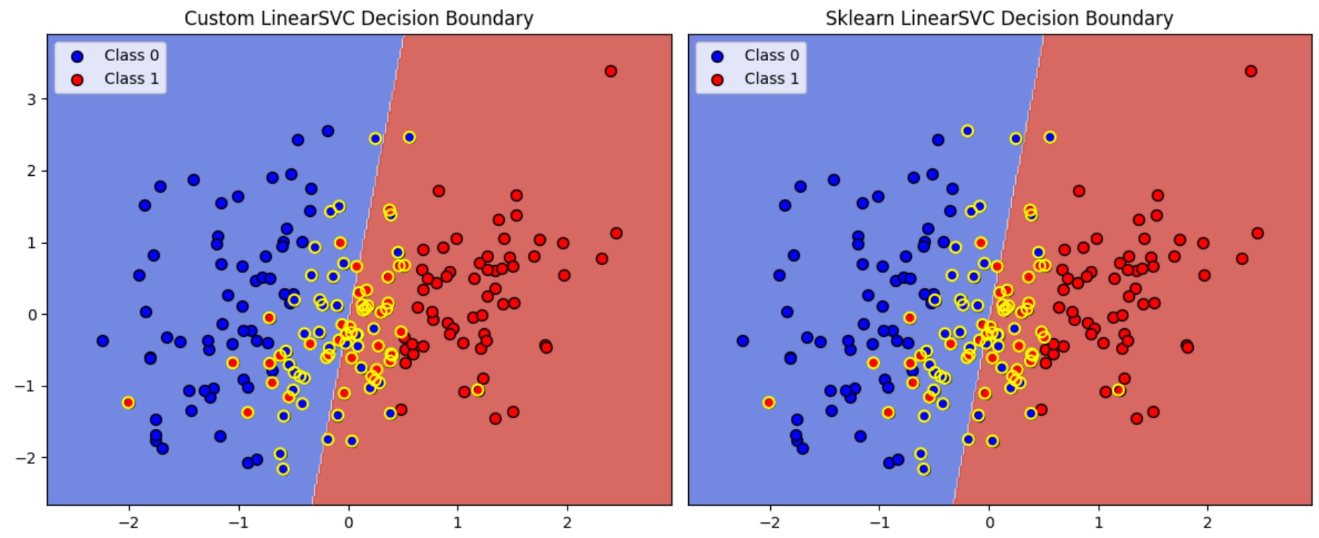
Sklearn LinearSVC Cross-Validation Scores

	accuracy	precision	recall	f1
0	0.964788	0.971381	0.971765	0.971479

As shown above, my implementation slightly outperformed scikit-learn's `LinearSVC` on the basic train-test split, but underperformed by a small margin in the cross-validation results. This suggests that while the custom model is competitive in fit, its generalization is slightly weaker — likely due to differences in optimization stability and regularization tuning.

Overall, the results validate that the custom model is functionally correct and behaves similarly to the production-grade scikit-learn version — a satisfying outcome for a scratch implementation.

4.2 Decision Boundaries & Support Vectors (vs. `SVC(kernel="linear")`



Since scikit-learn's `LinearSVC` does not expose support vectors, I used `SVC(kernel="linear")` to visualize and compare them.

Due to differences in implementation — including the underlying optimization algorithms and loss formulations — the regularization parameter `C` does not behave identically across the two models. To match decision boundaries and support vectors visually, I set `C=100` in my model while keeping `C=1` in `SVC`.

Despite these differences, the models learned nearly identical decision boundaries and support vectors (highlighted in yellow). This further reinforces the correctness of the custom implementation, even if the internal mechanics vary.

5. Wrapping Up Linear SVMs

So far, we implemented a Linear Support Vector Classifier entirely from scratch using NumPy, trained it with batch gradient descent, and evaluated it on real data. The model performed on par with scikit-learn's `LinearSVC`, showing comparable accuracy, F1 score, and decision boundaries — a strong signal that the implementation is both mathematically sound and practically useful.

But linear models have their limits.

Not all data is linearly separable — and this is where the **kernel trick** comes in. In the next section, we'll extend this implementation to support **nonlinear decision boundaries** by building a **kernelized SVM** from scratch.

6. Beyond Linearity: Introducing Kernelized SVMs

6.1 Intuition

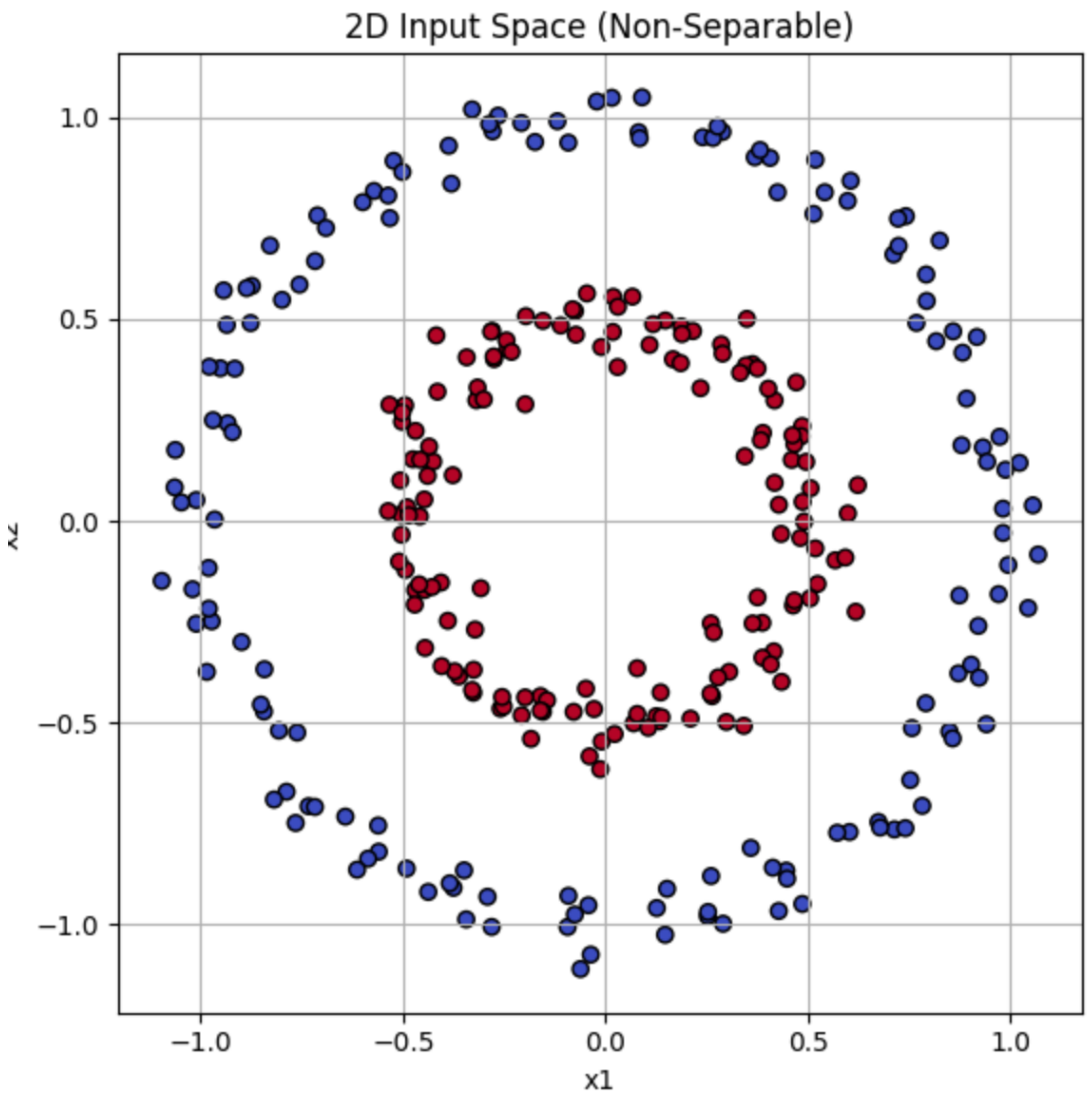
Linear decision boundaries are elegant and efficient — but what if your data simply *can't* be separated by a straight line?

Consider datasets where classes curve around each other, like spirals or concentric circles. A linear SVM will struggle, no matter how well it's optimized.

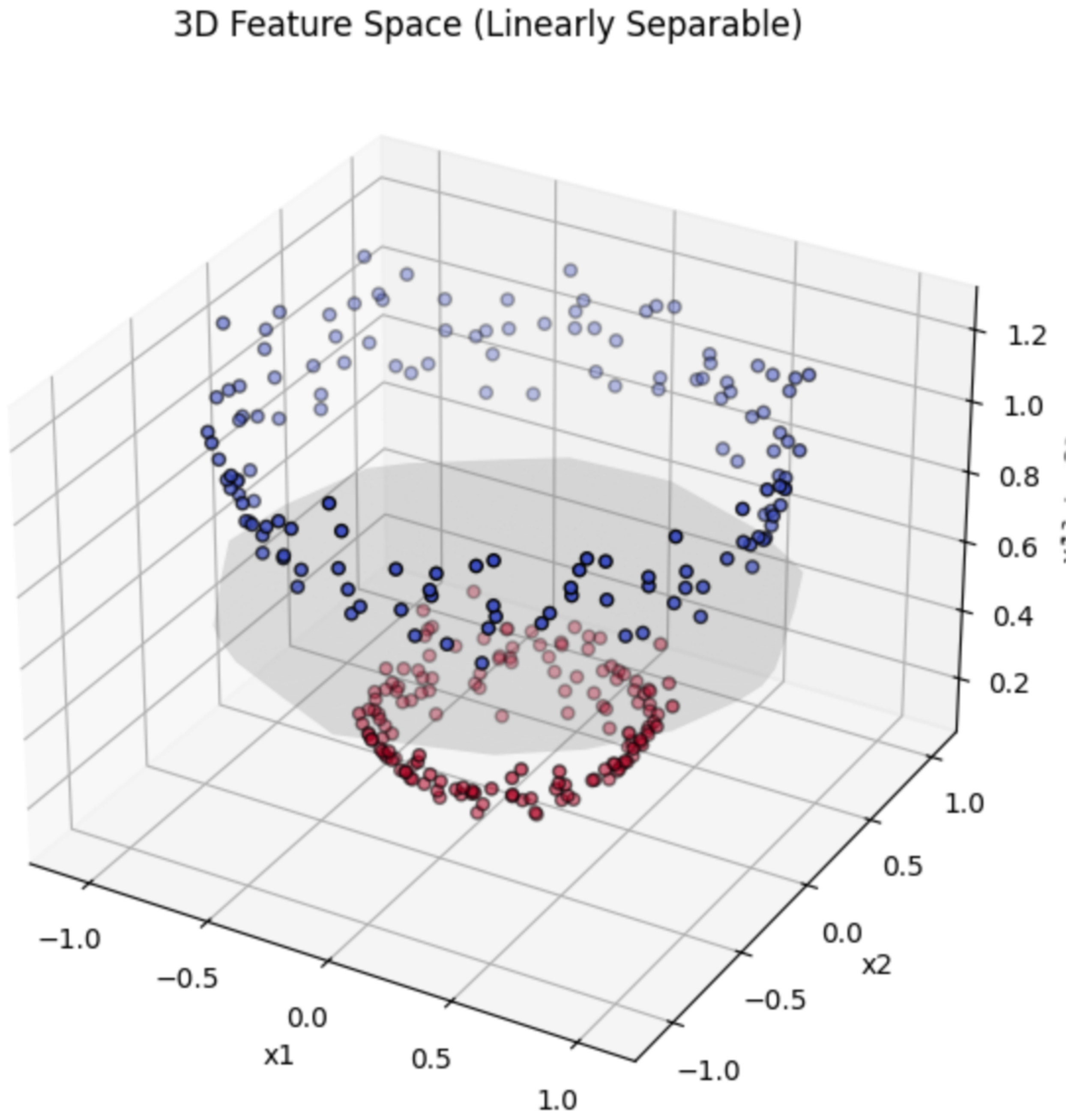
This is where SVMs truly shine. Using a technique known as the **kernel trick**, they can implicitly project data into higher-dimensional spaces — where a linear hyperplane *can* separate the classes. Remarkably, this is done without ever computing the actual transformation, making the process both powerful and computationally efficient.

Here's a visual demonstration of the problem — and how the kernel trick solves it.

In the first plot, we see two concentric clusters of points in a 2D input space. It's immediately clear that **no straight line** can separate these two classes:



Now, if we introduce a third axis $\mathbf{x_3}$ and apply a transformation (e.g., $\mathbf{x_3 = x_1^2 + x_2^2}$), we effectively "lift" the data into a 3D space. The result is shown below:



In this transformed space, the same data points are now **linearly separable** — a flat hyperplane is all that's needed to draw a perfect decision boundary.

This is the essence of the kernel trick: transforming data to make the problem easier, without ever explicitly computing the high-dimensional mapping.

6.2 A Word on Kernels and Mercer's Theorem

Before we dive into the dual formulation, it's important to understand what makes a function a "valid" kernel.

The **kernel trick** relies on replacing dot products with a kernel function $K(x, x\prime)$ — but how do we know whether this function corresponds to a legitimate transformation into a higher-dimensional space?

That's where **Mercer's Theorem** comes in.

> **Mercer's Theorem** states that a symmetric, positive semi-definite function $K(x, x\prime)$ corresponds to an inner product in some (possibly infinite-dimensional) Hilbert space.

In simple terms, this means:

- If your kernel function satisfies certain mathematical properties (symmetry and positive semi-definiteness),
- Then it is **guaranteed** to be equivalent to computing the dot product between two transformed vectors: $K(x, x') = \langle \phi(x), \phi(x') \rangle$

We never need to know or compute $\phi(x)$ directly — we just use $K(x, x')$ in the optimization.

Here are some of the most commonly used kernel functions that (mostly) satisfy Mercer's conditions:

- **Linear Kernel**: $K(x, x') = x^T x'$
  The standard inner product — equivalent to no transformation (i.e., the standard linear svc).
- **Polynomial Kernel** : $K(x, x') = (\gamma x^T x' + r)^d$
  Projects inputs into a space of polynomial combinations of features.
- **Gaussian RBF** : $K(x, x') = \exp(-\gamma ||x - x'||^2)$
  Projects data into an infinite-dimensional space — very powerful and widely used.
- **Sigmoid** : $K(x, x') = tanh(\gamma x^T x' + r)$
  Inspired by neural networks — behaves like a two-layer perceptron.

> Note: While the Sigmoid kernel does not always satisfy Mercer's conditions for all values of γ and r, it can still work well in practice when properly tuned. Other kernels may also violate the conditions under certain settings, but the ones listed above (except Sigmoid) are generally safe for most applications.

Now that we know when a kernel is valid, let's see **how** kernels get used in SVMs through the dual formulation.

6.3 Theoretical Foundations: Duality and the Kernel Trick

Now that we know what makes a function a valid **kernel**, let's look at how kernels actually come into play in SVMs — and that takes us to the concept of **duality**.

The SVM we've implemented so far is based on the **primal form** of the optimization problem: we directly learn the weight vector w and bias b by minimizing a hinge loss with L2 regularization. This approach works well for linear decision boundaries.

However, the primal problem has an **equivalent formulation** known as the **dual form**. Instead of solving for the weight vector w directly, the dual form rewrites the optimization in terms of a new set of parameters called **Lagrange multipliers**.

> ⚠️ Why do we care? Because the **kernel trick only works in the dual form** — it relies on replacing inner products between data points with kernel evaluations.

There are two major flavors of dual formulations:

- **Partial Dual**: A relaxed or simplified version of the primal problem. It's useful for analysis and can be easier to solve, but it may not match the primal solution exactly.
- **Complete Dual**: A fully equivalent reformulation of the primal optimization. This is the version we need for kernelized SVMs, since it allows us to express the solution entirely in terms of inner products — which we can then replace with kernel functions.

By working in the dual form, we never need to compute the feature mapping $\phi(x)$ explicitly. We just use the kernel function $K(x, x') = \langle \phi(x), \phi(x') \rangle$ to operate as if we were in the higher-dimensional space.

6.4 Solving the Dual: Why We Need Quadratic Programming

We've seen that kernelized SVMs rely on the **dual formulation** of the optimization problem. Let's now take a closer look at this dual form.

For binary classification, the **dual form of the soft-margin SVM** can be written as:

$\min_\alpha \frac{1}{2} \sum_{i,j=1}^m \alpha_i \alpha_j y_i y_j \langle xi, xj \rangle - \sum_{i=1}^m \alpha_i$

Subject to:
$0 \leq \alpha_i \leq C$ for i = 1,2,.....,m,
$\sum_{i=1}^m \alpha_i y_i = 0$

This is a **convex quadratic objective** with both **box constraints** $(0 \leq \alpha i \leq C)$ and a **linear equality constraint**.

This type of optimization problem — minimizing a **quadratic function** subject to linear constraints — belongs to a class of problems called **Quadratic Programming (QP)**.

Here is the general formulation of a Quadratic Programming problem:

$\min_p \frac{1}{2} p^T H p + f^T p$

Subject to:
$Ap \leq b$ and $A_{eq} p = b_{eq}$

Fortunately, we don't need to solve this optimization problem from scratch. Once we map the elements of our dual problem into the standard QP form, we can plug them into any off-the-shelf **Quadratic Programming solver** — such as `cvxopt`, `quadprog`, or `scipy.optimize.minimize`.

Here's how the components of our soft-margin SVM dual correspond to the standard QP form:

QP Component	SVM Dual Equivalent
p	$\alpha \in \mathbb{R}^m$ (one variable per training instance)
H	$Q \in \mathbb{R}^{m \times m}$, where $Q_{ij} = y_i y_j K(x_i, x_j)$
f	Vector of −1s: $f_i = -1$
A	$2m{\times}m$ matrix: first m rows are $-I$ (for $\alpha_i \geq 0$), next m rows are I (for $\alpha_i \leq C$)
b	$2m$-dimensional vector: first m elements are 0, next m are C
A_{eq}	Single row vector of labels: y^T
b_{eq}	Scalar 0

Once we map the soft-margin SVM dual to the standard QP form, we can simply plug these components into a **QP solver** to find the optimal vector of Lagrange multipliers $\hat{\alpha}$**.

From these values, we can recover the parameters of the original (primal) problem:

- The optimal weight vector:

  $\hat{w} = \sum_{i=1}^{m} \hat{\alpha}_i y_i x_i$
- The optimal bias $\hat{b}$ can be computed using any **support vector** x_i that lies exactly on the margin (i.e., with $0 < \hat{\alpha_i} < C$). A common approach is to average over all such support vectors:

  $\hat{b} = \frac{1}{n_s} \sum_{i \in SV} (y_i - \hat{w}^T x_i)$

  where n_s is the number of support vectors used in the average.

Till now, we've been discussing the dual solution for a **linear** SVM. But the real power of SVMs comes when we apply the **kernel trick** — which allows us to handle non-linear decision boundaries efficiently.

When applying the kernel trick, we **never compute $\hat{w}$ explicitly. Instead, we express the decision function in terms of inner products, which we replace with a kernel function.

Here's how the **kernelized decision function** is derived:

$$\hat{y}_{\hat{w}\hat{b}}(\phi(x_n)) = \hat{w}^T \phi(x_n) + \hat{b}$$
$$= (\sum_{i=1}^{m} \hat{\alpha}_i y_i \phi(x_i))^T \phi(x^n) + \hat{b}$$
$$= (\sum_{i=1}^{m} \hat{\alpha}_i y_i (\phi(x_i)^T \phi(x^n))) + \hat{b}$$
$$= \sum_{i \in SV} \hat{\alpha}_i y_i K(x_i, x_n) + \hat{b}$$

We typically sum over **support vectors** only (i.e., those with $\hat{\alpha}_i$ > 0) for efficiency.

When computing $\hat{b}$ in the kernel case we still us the support vectors only:

$\hat{b} = \frac{1}{n_s} \sum_{i \in SV} (y_i - \hat{w}^T \phi(x_i))$
$= \frac{1}{n_s} \sum_{i \in SV} (y_i - (\sum_{j=1}^{m} \hat{\alpha}_j y_j \phi(x_j))^T \phi(x_i))$
$= \frac{1}{n_s} \sum_{i \in SV} (y_i - \sum_{j \in SV}^{m} \hat{\alpha}_j y_j K(x_i, x_j))$

7. From Theory to Code: Kernelized SVC

Now that we've extended our understanding to kernelized SVMs, let's implement a binary **kernelized soft-margin SVC** from scratch. This implementation solves the dual optimization problem using **Quadratic Programming (QP)** via `cvxopt`, and supports `linear`, `polynomial`, and `RBF` kernels.

```
import numpy as np

from cvxopt import matrix, solvers

from utils.helpers import validate_transform_input

from .base_model import BaseModel



class KernelizedSVC(BaseModel):



    def __init__(self, kernel='rbf', C=1.0, gamma='scale', degree=3,
coef0=0.0):

        self._X = None

        self._y = None

        self._kernel = kernel

        self._C = C

        self._gamma = gamma

        self._degree = degree

        self._coef0 = coef0

        self._kernel_matrix = None
        self._alphas = None

        self._b = None

        self._support_indices_vis = None

        self.__support_indices = None



    def __compute_kernel_matrix(self, X1, X2):



        if self._kernel == 'linear':

            return X1 @ X2.T

        elif self._kernel == 'poly':

            return (self._gamma * (X1 @ X2.T) + self._coef0) ** self._degree

        elif self._kernel == 'rbf':

            sq_dists = np.sum(X1**2, axis=1).reshape(-1, 1) + np.sum(X2**2,
axis=1) - 2 * (X1 @ X2.T)

            sq_dists = np.maximum(sq_dists, 0)

            return np.exp(-self._gamma * sq_dists)



        else:

            raise ValueError("Unsupported kernel type. Use 'linear', 'poly',
'rbf'.")




    def __compute_alpha(self):
        m = self._X.shape[0]

        self._kernel_matrix = K = self.__compute_kernel_matrix(self._X,
self._X)

        H = np.outer(self._y, self._y) * K

        f = -np.ones(m)

        A = np.vstack([np.eye(m), -np.eye(m)])

        b = np.hstack([self._C * np.ones(m), np.zeros(m)])

        A_eq = self._y.reshape(1, -1)

        b_eq = np.array([0.0])



        H = matrix(H)

        f = matrix(f)

        A = matrix(A)

        b = matrix(b)

        A_eq = matrix(A_eq.astype('double'))

        b_eq = matrix(b_eq.astype('double'))



        solvers.options['show_progress'] = False

        solution = solvers.qp(H, f, A, b, A_eq, b_eq)

        alphas = np.array(solution['x']).flatten()

        return alphas
```

```
    def fit(self, X, y):



        self._X, self._y = self._validate_transform_input(X, y)

        if len(np.unique(y)) != 2:

            raise ValueError("Kernelized SVC only supports binary
classification.")

        if self._gamma == 'scale':

            self._gamma = 1.0 / (self._X.shape[1] * self._X.var())



        self._y = np.where(self._y <= 0, -1, 1)

        self._alphas = self.__compute_alpha()

        self.__support_indices = np.where((self._alphas > 1e-5) &
(self._alphas < self._C - 1e-5))[0]

        self._support_indices_vis = np.where(self._alphas > 1e-6)[0]

        b_vals = []

        for i in self.__support_indices:

            b_i = self._y[i] - np.sum(self._alphas * self._y *
self._kernel_matrix[i])

            b_vals.append(b_i)

        self._b = np.mean(b_vals)




    def predict(self, X):
        X, _ = self._validate_transform_input(X)

        if self._alphas is None or self._b is None:

            raise ValueError("Model has not been fitted yet. Call 'fit'
before 'predict'.")

        kernel_matrix = self.__compute_kernel_matrix(self._X, X)

        predictions = kernel_matrix.T @ (self._alphas * self._y) + self._b

        return np.where(predictions >= 0, 1, 0)
```

7.1 Initializing the Model

The constructor allows us to configure:

- `kernel` : The kernel type to use (`'linear'` , `'poly'` , `'rbf'`)
- `C` : Soft-margin regularization strength (higher = less regularization)
- `gamma` : Controls the **spread** of the RBF kernel and the influence of individual points.
  - **Low gamma** → smoother decision boundaries (wider influence).
  - **High gamma** → tighter boundaries (points influence only nearby space), risking overfitting.
- `degree` : Relevant for the **polynomial kernel**.
  - Higher degrees create more complex, curved decision boundaries.
  - Often, degree 2 or 3 is enough in practice.
- `coef0` : Affects both **polynomial** and **sigmoid** kernels.
  - Shifts the kernel function output by a constant before applying non-linearity.
  - Controls the influence of **higher-order terms** in poly kernels — useful for tuning model flexibility.

These hyperparameters let you fine-tune how flexible or smooth the model's decision boundary is in transformed space.

7.2 Computing the Kernel (Gram) Matrix

The private method `__compute_kernel_matrix(X1, X2)` computes the pairwise kernel values between samples in `X1` and `X2` . This is where we apply the **kernel trick** to avoid explicit feature transformation:

- **Linear Kernel**: $K(x, x') = x^T x'$
- **Polynomial Kernel** : $K(x, x') = (\gamma x^T x' + r)^d$

- **Gaussian RBF** : $K(x, x') = \exp(-\gamma||x - x'||^2)$

7.3 Solving the Dual Problem

The `__compute_alpha()` method constructs and solves the dual optimization problem:

```
$\min_{\alpha}\frac{1}{2}\alpha^TH\alpha-1^T\alpha$
```

Subject to:

- $0 \leq \alpha_i \leq C$
- $\sum \alpha_i y_i = 0$

Here:

- $H_{ij} = y_i y_j K(x_i, x_j)$
- `cvxopt.qp()` is used to solve this quadratic program
- The solution gives us the optimal α_i values

7.4 Fitting the Model

The `fit` method:

- Validates the input and converts labels from {0, 1} to {-1, 1}
- Computes the kernel matrix and solves for $\hat{\alpha}$
- Identifies support vectors as those with $0 < \alpha_i < C$ (points that are exactly on the margin)
- Identifies support vectors for visualization purposes as those with $\alpha_i < 10^{-6}$ (points that violate the margin or misclassified)
- Computes the bias $\hat{b}$ as the average over support vectors

  Now, you might be wondering why I have **two different sets of support vector indices** — one for evaluation and one for visualization — instead of just sticking with one. Well, here's an interesting observation:

  When I used the traditional definition of support vectors — those with $0 < \alpha_i < C$ — I got exactly the same evaluation metrics as scikit-learn's `SVC` , even across different scoring methods and cross-validation splits. But the actual support vectors were quite different.

  On the other hand, when I defined support vectors simply as those with $\alpha_i > 10^{-6}$, I got **support vector indices that matched scikit-learn's** — but the performance was **slightly worse**.

  So in the end, I decided to keep both:

  - One set for **evaluating and computing the bias term**

    - Another, looser set for **visualization**

  I’m not sure if this is what scikit-learn actually does under the hood — I might have to dig into their source code at some point — but for now, this hybrid approach works well for both consistency and interpretability.

7.5 Making Predictions

The `predict` method uses the decision function:
$f(x) = \sum_{i \in SV} \alpha_i yi K(x_i, x) + b$

- It computes the kernel matrix between the training set and test points
- Uses the the learned α_i, y_i to compute the decision function
- Predicts 1 if $f(x) \geq 0$, else 0

8. Custom vs scikit-learn: A Side-by-Side Comparison

To evaluate my custom `KernelizedSVC`, I compared it against `sklearn.svm.SVC`, which is the de facto implementation for kernel-based SVMs.

Since both models solve the same dual optimization problem with kernels, they are directly comparable in terms of decision boundaries, support vectors, and classification performance.

I used two datasets for validation:

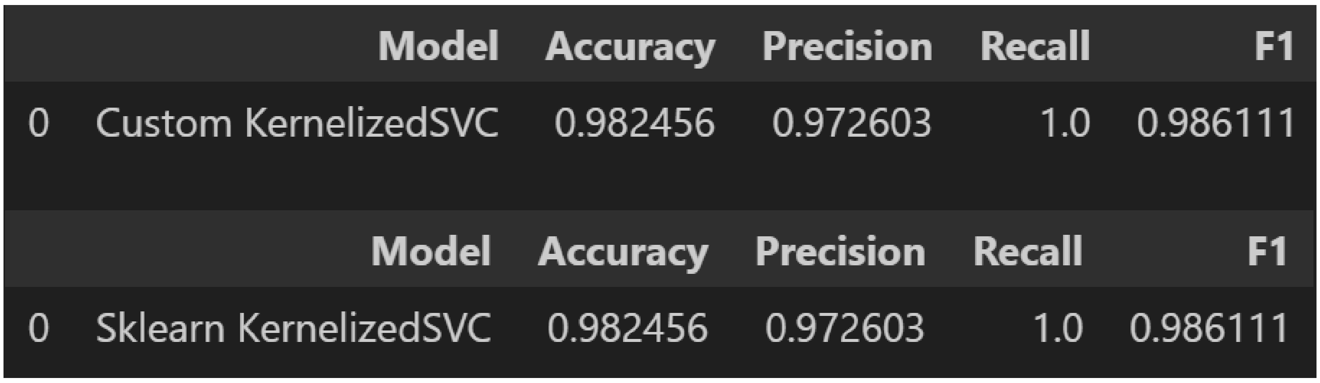
- 🧪 The `make_moons` dataset for visualizing decision boundaries in a non-linear setting.
- 🩺 The `breast_cancer` dataset for evaluating classification metrics and generalization.

8.1 Evaluation and Cross-Validation (vs. `sklearn.svm.SVC`)

I compared the models based on:

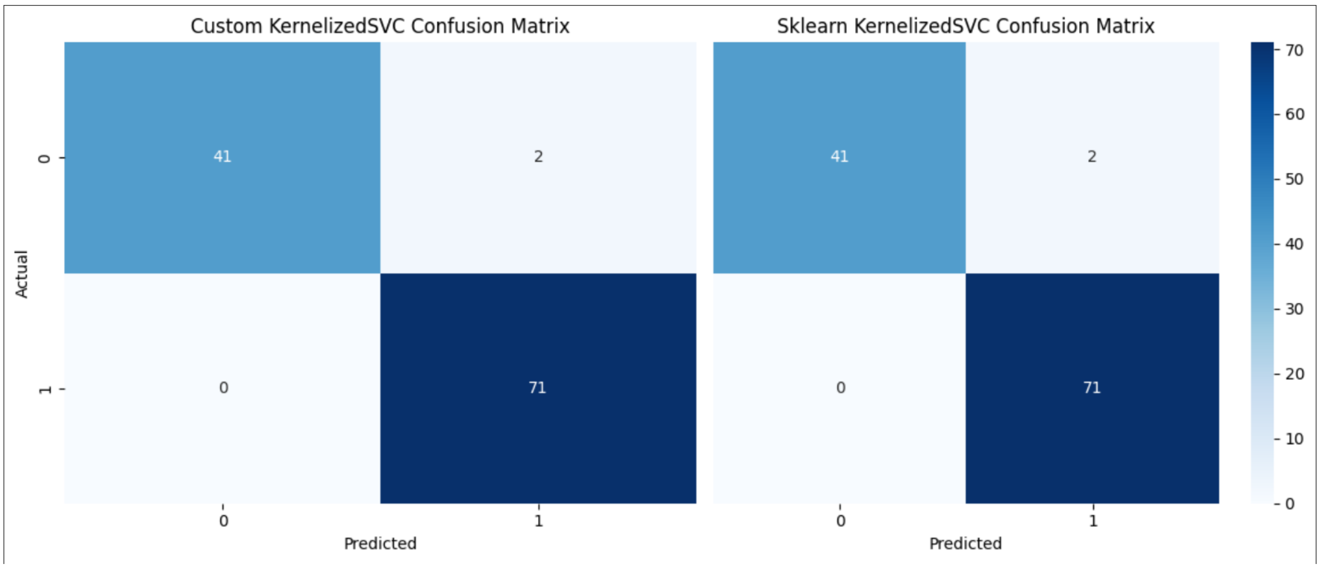
- Accuracy, Precision, Recall, and F1 Score
- 5-Fold Cross-Validation performance
- Decision Boundary Similarity

Train-Test Split Evaluation (kernel='rbf')

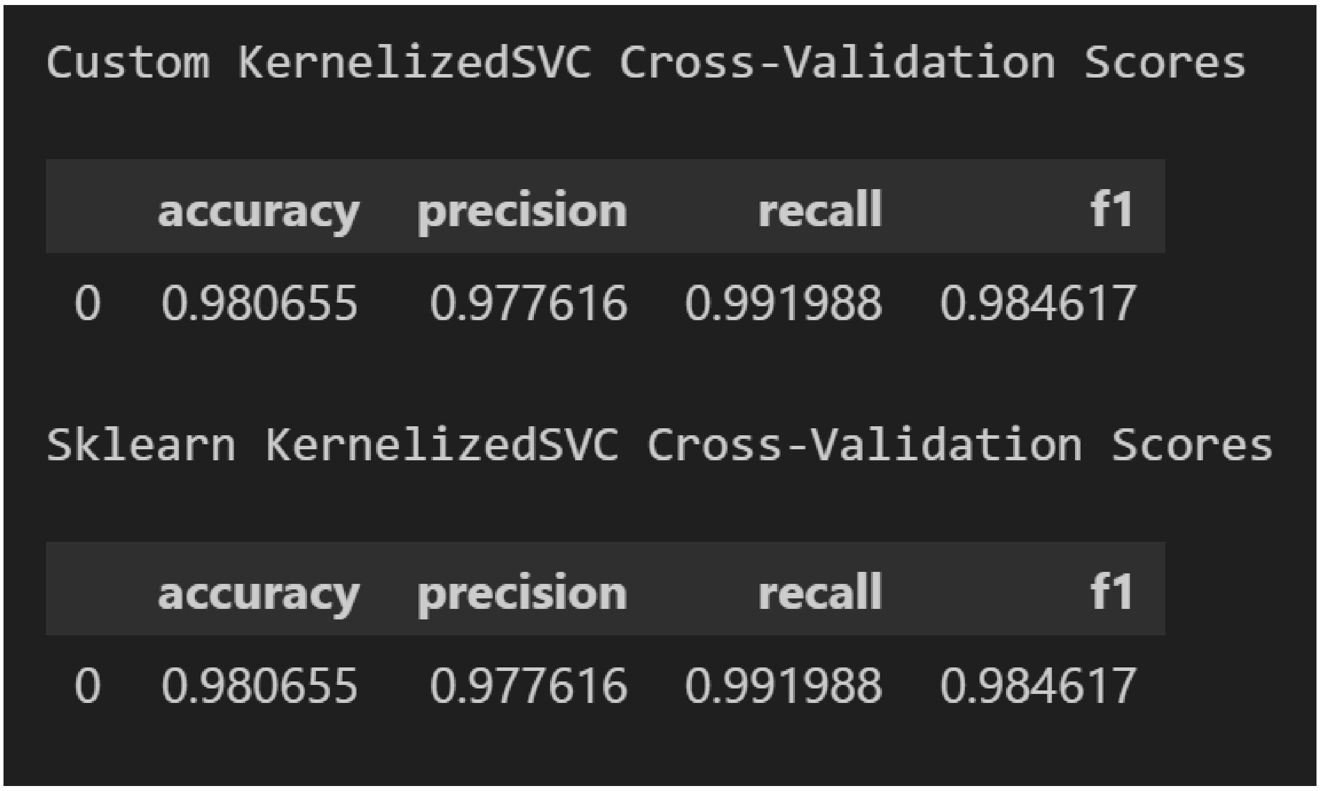


	Model	Accuracy	Precision	Recall	F1
0	Custom KernelizedSVC	0.982456	0.972603	1.0	0.986111

	Model	Accuracy	Precision	Recall	F1
0	Sklearn KernelizedSVC	0.982456	0.972603	1.0	0.986111



5-Fold Cross-Validation Evaluation (kernel='rbf')



Custom KernelizedSVC Cross-Validation Scores

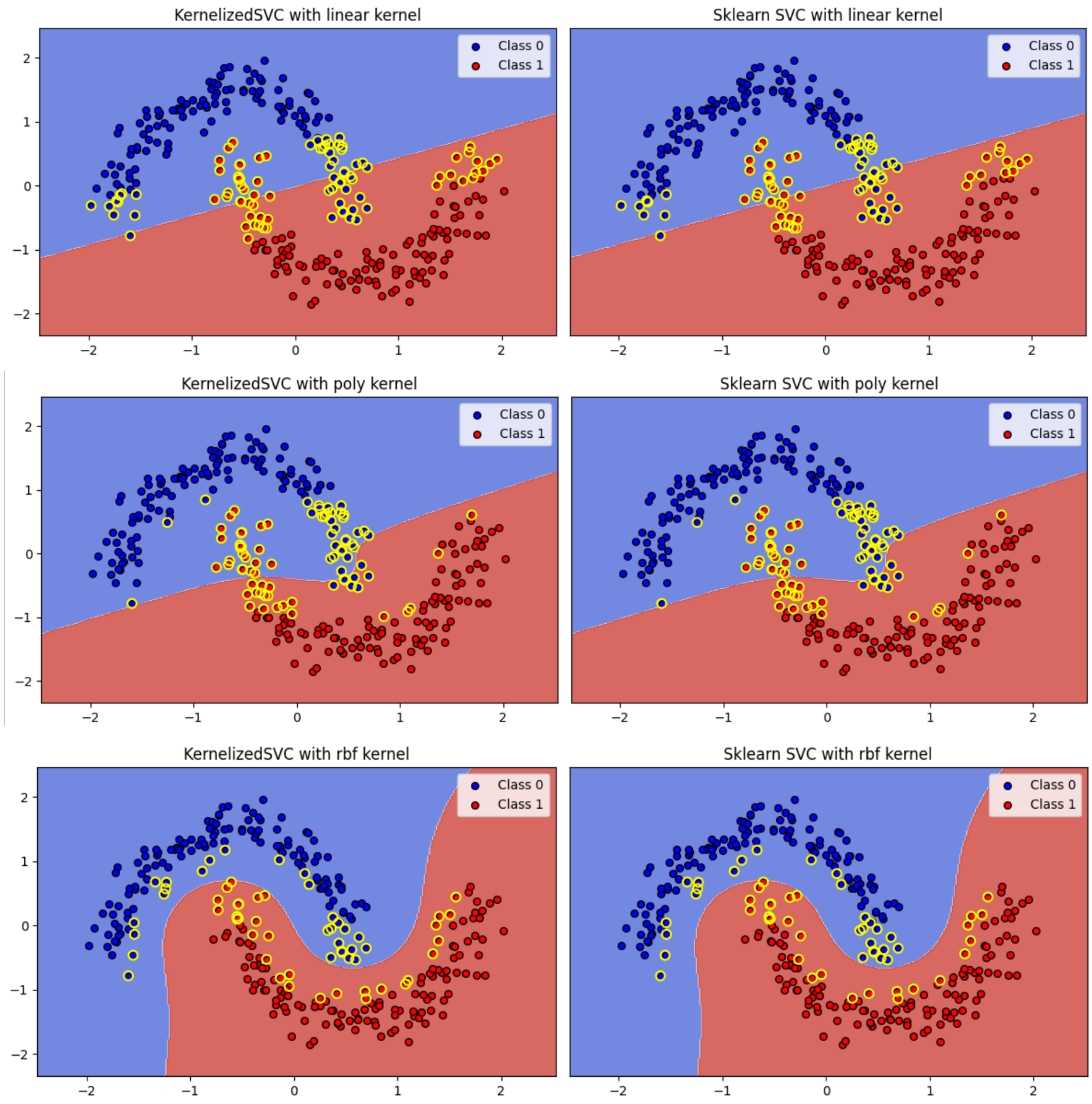
	accuracy	precision	recall	f1
0	0.980655	0.977616	0.991988	0.984617

Sklearn KernelizedSVC Cross-Validation Scores

	accuracy	precision	recall	f1
0	0.980655	0.977616	0.991988	0.984617

You can see that my implementation performs identically to scikit-learn's `SVC` across all evaluation metrics, including cross-validation. This consistency is expected, as both models solve **the same dual optimization problem** under the hood.

8.2 Decision Boundaries & Support Vectors (vs. `SVC()`)



Visually, the decision boundaries from my implementation align perfectly with those of scikit-learn's `SVC()` across all three kernels. The differences in support vectors are there — but barely noticeable — and don't affect the geometry of the boundary in any meaningful way.

I am sure that with a bit more tweaking — for example, adjusting how support vectors are selected— I'm confident those gaps could be eliminated entirely.

All in all, this comparison has been a really insightful sanity check. It's encouraging to see that a scratch-built SVM using `cvxopt` can mirror a production-level model so closely, both visually and in evaluation metrics.

Of course, this doesn't mean the model is on the same caliber as scikit-learn's `SVC` when it comes to scaling to large datasets or high-performance applications — it lacks the engineering optimizations, numerical stability tricks, and runtime efficiency required for production-grade workloads. But as a learning tool and a proof of concept, it holds up remarkably well.

9. Summary

In this post, we took a deep dive into **support vector classifiers**, starting with their theoretical foundations and building up to a fully functional **kernelized soft-margin SVC** from scratch using `numpy` and `cvxopt`. We explored:

- The intuition and mathematics behind **linear and kernelized SVMs**
- How the **kernel trick** allows us to classify non-linearly separable data in high-dimensional feature spaces
- The structure of the **dual optimization problem** and how to solve it with a QP solver
- Implementing and visualizing **linear**, **polynomial**, and **RBF** kernels
- A detailed comparison with scikit-learn’s `SVC()` across decision boundaries, support vectors, and evaluation metrics

Through careful experimentation, we found that our implementation performs **nearly identically** to scikit-learn's — even matching it **to the decimal** in cross-validation accuracy. Subtle implementation details (like the precision cutoff for support vectors) explain any visual or performance differences.

10. Wrapping Up

Writing this kernelized SVM from scratch was both a technical challenge and a rewarding way to reinforce the theory. While this model isn’t optimized for large-scale, real-world deployment like scikit-learn’s version, it **faithfully captures the mathematical heart** of the SVM algorithm.

More importantly, it gave me an opportunity to think deeply about:

- How optimization formulations translate into code,
- The role of each hyperparameter and kernel function,
- And how subtle implementation details can influence performance and behavior.

This won’t be the last scratch-built model I share — stay tuned for more as I continue to build out my own machine learning library, one algorithm at a time.

Connect & Explore More

If you found this implementation interesting or want to explore the full code, feel free to check out the repository:

🔗 **GitHub**: https://github.com/EyadMostafa/ml-from-scratch
🧑‍💻 **LinkedIn**: www.linkedin.com/in/eyad-mostafa-813b11206

I’m always happy to connect, chat machine learning, or get feedback!