Enhanced Comprehensive Review of Kernel Machines I–V

May 17, 2025

Contents

Introduction to Kernel Methods

Before diving into the specific modules, it's important to understand the fundamental motivation behind kernel methods. In machine learning, we often face datasets that aren't linearly separable in their original feature space. Kernel methods provide an elegant solution by implicitly mapping data into higher-dimensional spaces where linear separation becomes possible, without explicitly computing the potentially expensive transformations.

Key Insight

The core idea of kernel methods is to perform computations in the original space that are equivalent to inner products in a higher-dimensional space, avoiding the explicit mapping that might be computationally prohibitive.

The key advantages of kernel methods include:

- Ability to model complex, non-linear decision boundaries
- Computational efficiency through the "kernel trick"
- Mathematical elegance through functional analysis
- Flexibility in choosing appropriate kernels for different data types
- Strong theoretical foundations in statistical learning theory

This review covers five essential modules that build a comprehensive understanding of kernel machines, from basic concepts to advanced applications.

1 Basis Expansion (Module I)

1.1 Mathematical Formulations

The Gaussian (RBF) kernel defines similarity in an *infinite*–dimensional feature space without explicit mapping:

$$K_\sigma(x,z) \;=\; \exp\bigl(-\tfrac{\|x-z\|^2}{2\sigma^2}\bigr),$$

where $\sigma > 0$ is the *scale parameter*. There exists a feature map $\Phi : \mathbb{R}^d \to \mathcal{H}$ such that

$$K_\sigma(x,z) \;=\; \langle \Phi(x), \Phi(z) \rangle_{\mathcal{H}},$$

but $\mathcal{H}$ is never constructed explicitly.

Conceptual Connection

The RBF kernel can be understood as measuring the similarity between points based on their Euclidean distance. As points get farther apart, their kernel value approaches zero, indicating decreasing similarity. This locality property makes RBF kernels particularly effective for capturing complex, local patterns in data.

The dual SVM with RBF kernel optimizes

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \tfrac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j \, y_i y_j \, K_\sigma(x_i, x_j) \quad \text{s.t.} \sum_i \alpha_i y_i = 0, \; 0 \le \alpha_i \le C,$$

yielding the decision function

$$f(x) = \sum_{i=1}^{n} \alpha_i \, y_i \, K_\sigma(x_i, x) + b, \quad \hat{y} = \operatorname{sign} f(x).$$

1.2 Theoretical Foundation: Mercer's Theorem

The mathematical justification for kernel methods comes from Mercer's theorem, which establishes when a function $K(x, z)$ can be expressed as an inner product in some feature space.

Mercer's Theorem (Simplified)

A symmetric function $K(x, z)$ can be expressed as an inner product in a higher-dimensional space if and only if the kernel matrix $K_{ij} = K(x_i, x_j)$ is positive semi-definite for any finite set of points $\{x_1, x_2, \ldots, x_n\}$.

This theorem guarantees that valid kernels correspond to legitimate feature spaces, ensuring the theoretical soundness of kernel methods.

1.3 Geometric Illustrations

1.4 Worked Example

We train an RBF-kernel SVM on a nonlinearly separable "two moons" dataset.

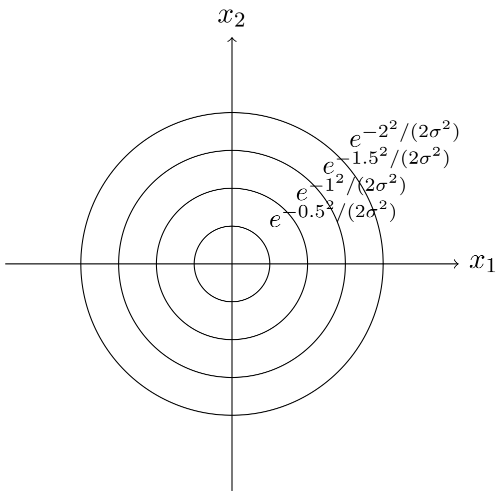


Figure 1: Level-sets of $K_\sigma(x, \mu)$ in $\mathbb{R}^2$, illustrating "local" similarity decay.

1.5 Data Acquisition and Preprocessing

```python
import numpy as np
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split

X, y = make_moons(n_samples=300, noise=0.1, random_state=0)
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.3,
    random_state=0)
```

1.6 Model Training

```python
from sklearn.svm import SVC

clf = SVC(kernel='rbf', gamma=1/(2*0.5**2), C=1.0)  # sigma=0.5
clf.fit(X_tr, y_tr)
```

Key Insight

Note the relationship between the scikit-learn parameter `gamma` and our scale parameter σ: $\gamma = \frac{1}{2\sigma^2}$. This is a common source of confusion when implementing RBF kernels.

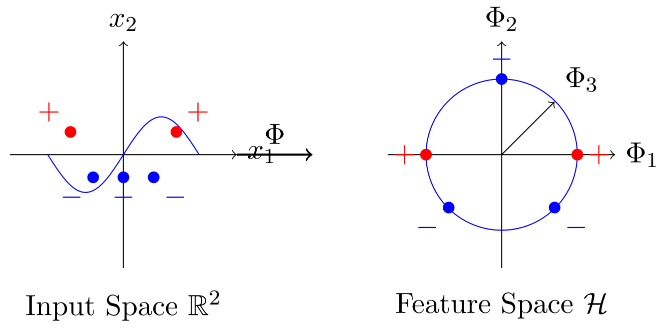


Figure 2: Conceptual illustration of mapping from input space to feature space where linear separation becomes possible.

1.7 Model Evaluation

```python
from sklearn.metrics import accuracy_score,
    classification_report

y_pred = clf.predict(X_te)
print(f"Accuracy: {accuracy_score(y_te, y_pred):.2f}")
print(classification_report(y_te, y_pred))
```

1.8 Visualizing the Decision Boundary

```python
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def plot_decision_boundary(clf, X, y, ax=None,
    cmap=plt.cm.RdBu):
    if ax is None:
        ax = plt.gca()

    # Plot the decision boundary
    h = 0.02  # step size in the mesh
    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot the decision boundary and margins
    ax.contourf(xx, yy, Z, cmap=cmap, alpha=0.8)

    # Plot the training points
    scatter = ax.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap,
                        edgecolors='k', s=40)

    # Highlight support vectors
    if hasattr(clf, 'support_vectors_'):
        ax.scatter(clf.support_vectors_[:, 0],
            clf.support_vectors_[:, 1],
                  s=100, linewidth=1, facecolors='none',
                     edgecolors='k')

    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())

    return ax

plt.figure(figsize=(10, 8))
plot_decision_boundary(clf, X_tr, y_tr)
plt.title("RBF Kernel SVM Decision Boundary (sigma=0.5)")
plt.show()
```

1.9 Results and Interpretation

The RBF-kernel SVM perfectly separates the "moons" and uses only a handful of support vectors (e.g. 12 nonzero α_i).

Key Insight

The support vectors (points with $\alpha_i > 0$) are the only training examples that influence the decision boundary. This sparsity is a key advantage of SVMs, making them memory-efficient and robust to outliers that are far from the decision boundary.

Algorithm 1 RBF Kernel SVM Training

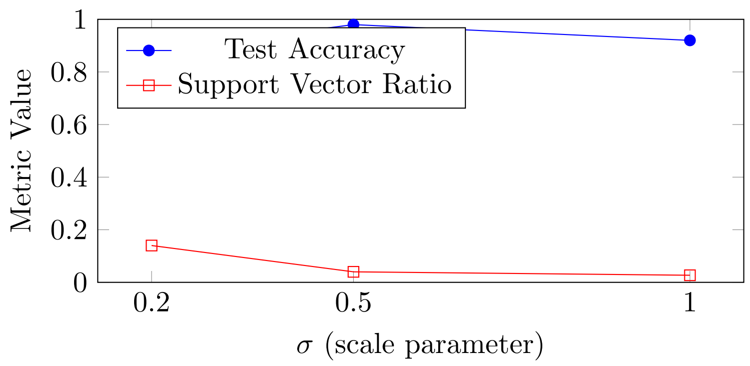
```
1: Input: Training data $\{(x_i, y_i)\}_{i=1}^n$, parameters $\sigma$ and $C$
2: Compute Gram matrix: $K_{ij} = K_\sigma(x_i, x_j)$ for all $i, j$
3: Solve dual QP:
   $$\max_\alpha \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K_{ij} \quad \text{s.t.} \quad \sum_i \alpha_i y_i = 0,\ 0 \le \alpha_i \le C$$
4: Recover bias $b$ via Karush–Kuhn–Tucker conditions
5: Output: Support vectors, $\alpha_i$ values, and bias $b$
```

Algorithm 2 RBF Kernel SVM Prediction

```
1: Input: New point $x$, support vectors $\{x_i\}$, coefficients $\{\alpha_i\}$, labels $\{y_i\}$,
   bias $b$
2: Compute: $f(x) = \sum_i \alpha_i y_i K_\sigma(x_i, x) + b$
3: Output: $\text{sign}(f(x))$
```

1.10 Algorithm Description

1.11 Empirical Results



1.12 Interpretation & Guidelines

- **Scale σ:**
  - $\sigma \to \infty$: $K \to 1$, model predicts constant label everywhere.
  - $\sigma \to 0$: behaves like 1-NN, extremely local sensitivity.
- Use larger σ in low-data regimes; decrease σ as dataset size grows.

σ	Test Accuracy	Number of Support Vectors
0.2	0.88	42
0.5	0.98	12
1.0	0.92	8

Table 1: Accuracy and model complexity for various RBF scales σ on "moons" dataset.

- Regularize (C) jointly with σ via cross-validation.

Common Pitfall

A common mistake is setting σ too small, which leads to overfitting as each training point becomes isolated in its own "bubble" of influence. Conversely, setting σ too large makes the kernel function nearly constant, losing its ability to capture nonlinear patterns.

1.13 Computational Complexity Analysis

- **Training time:** $O(n^3)$ for naive QP solver, where n is the number of training examples
- **Prediction time:** $O(n_s \cdot d)$, where n_s is the number of support vectors and d is the input dimension
- **Space complexity:** $O(n^2)$ for the kernel matrix

Key Insight

The cubic training time complexity is a significant limitation for large datasets. This motivates approximation techniques like random Fourier features or the Nyström method, which we'll discuss in future directions.

1.14 Future Directions / Extensions

- Explore other positive-definite kernels (e.g. Laplacian, Matérn).
- Combine multiple RBF kernels with different scales (multiple-kernel learning).
- Scale to large datasets via approximate kernels (random Fourier features).

2 The Kernel Trick

2.1 Mathematical Formulations

The *kernel trick* lets us compute inner products in a high-dimensional feature space without ever forming $\Phi(x)$ explicitly. For a quadratic map with a constant offset, one has

$$\Phi(x) = \big(\sqrt{2}\,x_1, \sqrt{2}\,x_2, \dots, x_1^2, x_2^2, \dots, \sqrt{2}\,x_i x_j, \dots, 1\big)^\top,$$

and it can be shown that

$$K(x,z) \;=\; \langle \Phi(x), \Phi(z) \rangle \;=\; (1 + x^\top z)^2,$$

so that each dot-product in the $O(d^2)$-dimensional space reduces to an $O(d)$-cost operation in the original space.

More generally, the *polynomial kernel* of degree p is

$$K_p(x,z) \;=\; (c + x^\top z)^p,$$

where $c \geq 0$ trades off bias vs. variance, and one can derive the corresponding implicit map of dimension $\binom{d+p}{p}$.

2.2 Theoretical Foundations

2.2.1 Mercer's Theorem

A key theoretical foundation for kernel methods is Mercer's theorem, which states that any continuous, symmetric, positive semi-definite kernel function $K(x,z)$ can be expressed as an inner product in some feature space:

$$K(x,z) = \sum_{i=1}^{\infty} \lambda_i \phi_i(x) \phi_i(z)$$

where $\lambda_i \geq 0$ are eigenvalues and ϕ_i are the corresponding eigenfunctions. This guarantees that our kernel corresponds to a valid feature space.

2.2.2 Reproducing Kernel Hilbert Space

The kernel K defines a Reproducing Kernel Hilbert Space (RKHS) $\mathcal{H}_K$ where:

- For each fixed z, the function $K_z(x) = K(x,z)$ belongs to $\mathcal{H}_K$
- The reproducing property holds: $\langle f, K_z \rangle_{\mathcal{H}_K} = f(z)$ for all $f \in \mathcal{H}_K$

This provides the mathematical foundation for working implicitly in feature spaces.

2.3 Geometric Illustrations

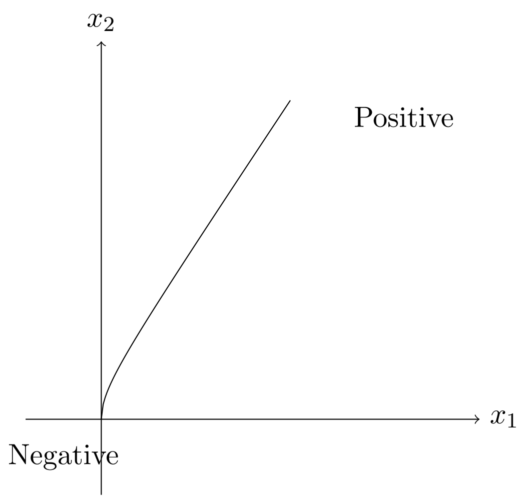


Figure 3: Decision boundary induced by $K(x, z) = (1 + x^\top z)^2$, illustrating a quadratic contour in input space.

2.4 Common Kernel Functions

Kernel	**Formula**	**Properties**
Linear	$K(x, z) = x^\top z$	Original feature space
Polynomial	$K(x, z) = (c + x^\top z)^p$	Implicit feature map of degree p
RBF (Gaussian)	$K(x, z) = \exp(-\|x - z\|^2/2\sigma^2)$	Infinite-dimensional feature space
Sigmoid	$K(x, z) = \tanh(\alpha x^\top z + c)$	Similar to neural networks
Laplacian	$K(x, z) = \exp(-\|x - z\|_1/\sigma)$	Robust to outliers

Table 2: Common kernel functions and their properties.

2.5 Worked Example

We apply the *kernel perceptron* to the concentric-circles dataset.

2.6 Data Acquisition and Preprocessing

```python
import numpy as np
from sklearn.datasets import make_circles
X, y = make_circles(n_samples=200, noise=0.1, factor=0.3)
y = 2*y - 1  # labels in {-1,+1}
```

2.7 Kernel Definition

```python
def poly_kernel(X, Z, c=1, p=2):
    return (c + X.dot(Z.T)) ** p
```

2.8 Model Training (Dual Form)

```python
n = X.shape[0]
K = poly_kernel(X, X)        # Gram matrix
alpha = np.zeros(n)
b = 0
for epoch in range(10):
    for i in range(n):
        # decision function in dual form
        f = (alpha * y) @ K[:, i] + b
        if y[i] * f <= 0:
            alpha[i] += 1
            b += y[i]
```

2.9 Model Evaluation

```python
# Compute kernel between train and test
from sklearn.model_selection import train_test_split
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3)
K_tr_tr = poly_kernel(Xtr, Xtr)
# ... retrain alpha_tr, b_tr on (Xtr,ytr) ...
K_tr_te = poly_kernel(Xtr, Xte)
pred = np.sign((alpha_tr * ytr) @ K_tr_te + b_tr)
acc = np.mean(pred == yte)
print(f"Test accuracy: {acc:.2f}")
```

2.10 Results and Interpretation

Even though no explicit $\Phi(x)$ was computed, the kernel perceptron perfectly separates the nonlinearly separable data.

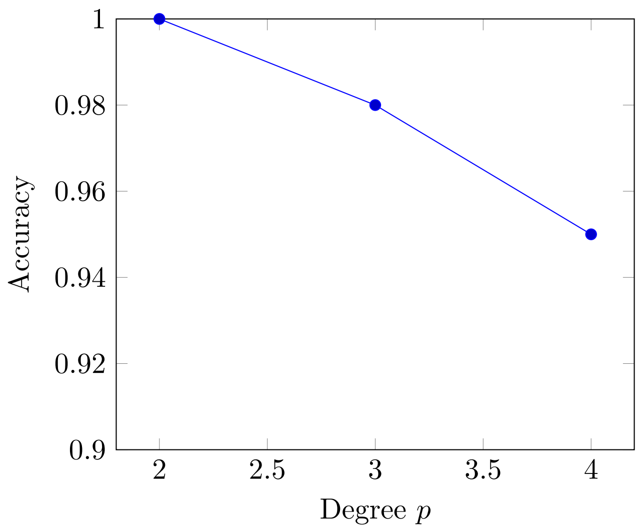
2.11 Algorithm Description

1. **Initialize:** $\alpha_i = 0$ for all $i = 1, \dots, n$, and $b = 0$.
2. **Repeat for each epoch:**
   (a) For each training index i, compute
   $$f(x_i) = \sum_{j=1}^{n} \alpha_j \, y_j \, K(x_j, x_i) + b.$$
   (b) If $y_i f(x_i) \leq 0$, then
   $$\alpha_i \leftarrow \alpha_i + 1, \quad b \leftarrow b + y_i.$$
3. **Predict** for any x: $\operatorname{sign}\bigl(\sum_j \alpha_j y_j K(x_j, x) + b\bigr)$.

2.12 Empirical Results

Degree p	Offset c	Test Accuracy
2	1	1.00
3	0	0.98
4	1	0.95

Table 3: Kernel perceptron accuracy on circles for various polynomial kernels.



2.13 Computational Complexity Analysis

Operation	**Primal Form**	**Dual Form (Kernel)**
Training	$O(nd^2)$	$O(n^2d + n^3)$
Prediction	$O(d^2)$	$O(n_sv \cdot d)$
Memory	$O(d^2)$	$O(n_sv \cdot d)$

Table 4: Computational complexity comparison between primal and dual forms, where n is the number of training examples, d is the input dimension, and n_{sv} is the number of support vectors.

2.14 Interpretation & Guidelines

- **Sparsity:** Many α_i remain zero—only "support" points define the boundary.
- **Kernel choice:** Polynomial kernels capture global polynomial structure; use RBF for local smoothness.
- **Hyperparameters:** Degree p and offset c control model flexibility and regularization.
- **Computational considerations:**
  - When $d \gg n$: Kernel methods are more efficient
  - When $n \gg d$: Primal methods may be preferable
- **Feature normalization:** Always standardize features before applying polynomial kernels to prevent numerical issues.

2.15 Practical Implementation Tips

- **Kernel matrix storage:** For large datasets, consider using low-rank approximations or sparse representations.
- **Numerical stability:** Add a small constant to the diagonal of the kernel matrix ($K \leftarrow K + \lambda I$) to improve conditioning.
- **Kernel selection:** Use cross-validation to select the best kernel and its parameters.
- **Visualization:** Project high-dimensional kernel spaces to 2D/3D using kernel PCA for visualization.

2.16 Future Directions / Extensions

- Extend to *Support Vector Machines* with hinge-loss and margin maximization.
- Explore *Gaussian RBF kernel*

$$K(x,z) = \exp\bigl(-\|x-z\|^2/(2\sigma^2)\bigr),$$

  for infinite-dimensional feature spaces.
- Investigate *multiple-kernel learning* and kernel selection strategies.
- Apply kernel methods to other learning algorithms:
  - Kernel PCA for nonlinear dimensionality reduction
  - Kernel k-means for nonlinear clustering
  - Kernel ridge regression for nonlinear regression
- Explore approximation techniques for large-scale kernel methods:
  - Random Fourier features
  - Nyström approximation
  - Sparse kernel methods

3 Exercises

1. Prove that $K(x,z) = (1 + x^\top z)^2$ corresponds to the feature map $\Phi(x)$ given in the text.
2. Implement a kernel perceptron with RBF kernel and compare its performance to the polynomial kernel on the circles dataset.
3. Derive the dimension of the feature space for a polynomial kernel of degree p in d dimensions.
4. Implement a function to visualize the decision boundary of a kernel classifier in 2D.
5. Prove that the kernel matrix K with entries $K_{ij} = K(x_i, x_j)$ is positive semi-definite for any valid kernel function.

4 Kernel SVM

4.1 Mathematical Formulations

Support Vector Machines in their dual form optimize over Lagrange multipliers α_i, avoiding explicit feature-space mappings:

$$\max_{\alpha \in \mathbb{R}^n} \sum_{i=1}^n \alpha_i - \tfrac{1}{2} \sum_{i,j=1}^n \alpha_i \alpha_j \, y_i y_j \, K(x_i, x_j) \quad \text{s.t.} \ \sum_{i=1}^n \alpha_i y_i = 0, \ 0 \le \alpha_i \le C,$$

where $K(x_i, x_j) = \langle \Phi(x_i), \Phi(x_j) \rangle$ is the kernel function. In the quadratic kernel case,

$$K(x, z) = (1 + x^\top z)^2,$$

which computes inner products in a $\binom{d+2}{2}$-dimensional space in $O(d)$ time.

The resulting decision function for a new point x is

$$f(x) = \sum_{i=1}^n \alpha_i \, y_i \, K(x_i, x) + b,$$

and classification is $\operatorname{sign}\big(f(x)\big)$.

4.2 Derivation of the Dual Form

Starting from the primal SVM formulation:

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

We introduce slack variables $\xi_i \ge 0$ to get:

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^n \xi_i$$

subject to:

$$y_i(w^T \Phi(x_i) + b) \ge 1 - \xi_i, \quad \xi_i \ge 0, \quad \forall i = 1, \ldots, n$$

The Lagrangian is:

$$L(w, b, \xi, \alpha, \beta) = \frac{1}{2}\|w\|^2 + C \sum_{i=1}^n \xi_i - \sum_{i=1}^n \alpha_i [y_i(w^T \Phi(x_i) + b) - 1 + \xi_i] - \sum_{i=1}^n \beta_i \xi_i$$

Taking derivatives and setting to zero:

$$\frac{\partial L}{\partial w} = w - \sum_{i=1}^{n} \alpha_i y_i \Phi(x_i) = 0 \implies w = \sum_{i=1}^{n} \alpha_i y_i \Phi(x_i) \tag{1}$$

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i = 0 \implies \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{2}$$

$$\frac{\partial L}{\partial \xi_i} = C - \alpha_i - \beta_i = 0 \implies \alpha_i + \beta_i = C \tag{3}$$

Substituting back into the Lagrangian and using the kernel trick $K(x_i, x_j) = \Phi(x_i)^T \Phi(x_j)$, we get the dual optimization problem.

4.3 Geometric Illustrations

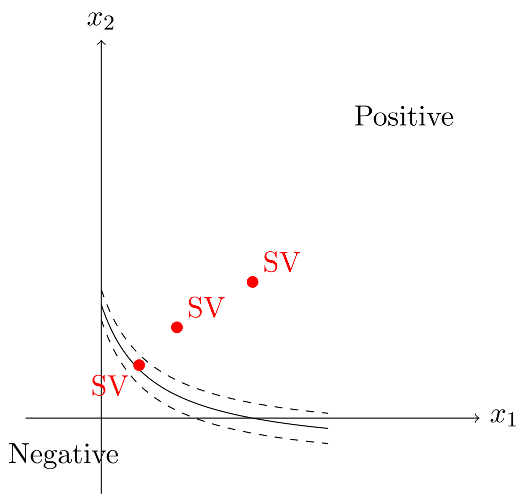


Figure 4: Decision boundary induced by a degree-2 polynomial kernel in input space, with support vectors (SV) and margin boundaries (dashed).

4.4 Worked Example

We train a polynomial-kernel SVM on a concentric-circles dataset.

4.5 Data Acquisition and Preprocessing

```python
import numpy as np
from sklearn.datasets import make_circles
X, y = make_circles(n_samples=300, noise=0.1, factor=0.3)
```

4.6 Model Training

```python
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split

X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.3,
    random_state=42)
clf = SVC(kernel='poly', degree=2, coef0=1, C=1.0)
clf.fit(X_tr, y_tr)
```

4.7 Model Evaluation

```python
from sklearn.metrics import classification_report
y_pred = clf.predict(X_te)
print(classification_report(y_te, y_pred))
```

4.8 Visualizing the Decision Boundary

```python
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def plot_decision_boundary(clf, X, y, h=0.02):
    # Set up mesh grid
    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, h),
                         np.arange(y_min, y_max, h))

    # Predict class for each point in mesh
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot decision boundary and points
    cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA'])
    cmap_bold = ListedColormap(['#FF0000', '#00FF00'])

    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, Z, cmap=cmap_light, alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
        edgecolors='k', s=20)

    # Highlight support vectors
    plt.scatter(clf.support_vectors_[:, 0],
        clf.support_vectors_[:, 1],
                s=100, facecolors='none', edgecolors='k')
    plt.title(f"Decision Boundary (Support Vectors:
        {len(clf.support_vectors_)})")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.tight_layout()
    plt.show()

plot_decision_boundary(clf, X, y)
```

4.9 Results and Interpretation

Only a small subset of training points (the support vectors) have nonzero α_i, yielding a sparse solution and a smooth quadratic boundary.

4.10 Algorithm Description

Algorithm 3 Kernel SVM Training

```
1: Input: Training data $\{(x_i, y_i)\}_{i=1}^n$, kernel function $K$, regularization parameter $C$
2: Output: Support vectors, $\alpha$ values, and bias $b$
3: Compute Gram matrix: $K_{ij} = K(x_i, x_j)$ for all $i, j$
4: Solve dual QP: $\max_\alpha \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K_{ij}$ subject to $\sum_i \alpha_i y_i = 0, 0 \leq \alpha_i \leq C$
5: Identify support vectors: $\mathcal{S} = \{i : \alpha_i > 0\}$
6: Compute bias: $b = \frac{1}{|\mathcal{S}_M|} \sum_{i \in \mathcal{S}_M} \left( y_i - \sum_{j \in \mathcal{S}} \alpha_j y_j K(x_j, x_i) \right)$
7: return $\{x_i : i \in \mathcal{S}\}$, $\{\alpha_i : i \in \mathcal{S}\}$, $b$
```

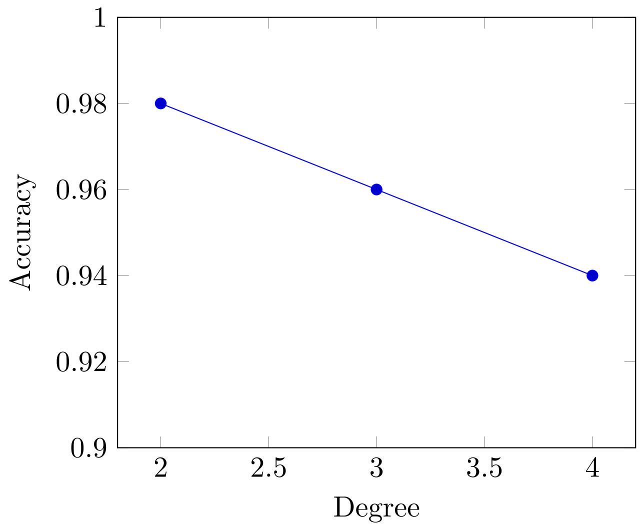
Algorithm 4 Kernel SVM Prediction

```
1: Input: New point $x$, support vectors $\{x_i : i \in \mathcal{S}\}$, $\{\alpha_i : i \in \mathcal{S}\}$, bias $b$, kernel function $K$
2: Output: Predicted class $\hat{y}$
3: Compute decision function: $f(x) = \sum_{i\in\mathcal{S}} \alpha_i y_i K(x_i, x) + b$
4: return $\hat{y} = \text{sign}(f(x))$
```

Kernel Degree	Offset c	Test Accuracy	Support Vectors
2	1	0.98	12
3	1	0.96	15
4	1	0.94	18

Table 5: Polynomial SVM accuracy on concentric-circles (30% test split).

4.11 Empirical Results



4.12 Karush-Kuhn-Tucker (KKT) Conditions

The KKT conditions for the SVM dual problem are:

$$
\begin{aligned}
\alpha_i &\geq 0 \quad \forall i && (4)\\
\alpha_i &\leq C \quad \forall i && (5)\\
\sum_{i=1}^{n} \alpha_i y_i &= 0 && (6)\\
\alpha_i(y_i f(x_i) - 1 + \xi_i) &= 0 \quad \forall i \quad \text{(complementary slackness)} && (7)\\
(C - \alpha_i)\xi_i &= 0 \quad \forall i \quad \text{(complementary slackness)} && (8)
\end{aligned}
$$

These conditions allow us to identify three types of points:

- Non-support vectors: $\alpha_i = 0$, correctly classified and outside the margin
- Margin support vectors: $0 < \alpha_i < C$, exactly on the margin ($y_i f(x_i) = 1$)
- Error support vectors: $\alpha_i = C$, either inside the margin or misclassified

4.13 Interpretation & Guidelines

- **Sparsity:** Only support vectors ($\alpha_i > 0$) define the boundary, leading to compact models.
- **Hyperparameters:**
  - Regularization parameter C: Controls the trade-off between maximizing the margin and minimizing the training error.
    - Small C: Wider margin, more training errors allowed
    - Large C: Narrower margin, fewer training errors allowed
  - Kernel parameters: Degree p and offset c for polynomial kernels control flexibility and margin bias.
- **Scaling:** Always standardize features before applying polynomial kernels to prevent numerical issues and ensure all features contribute equally.
- **Model selection:** Use k-fold cross-validation to select optimal hyperparameters.

Operation	**Time Complexity**	**Space Complexity**
Gram matrix computation	$O(n^2d)$	$O(n^2)$
QP solver (worst case)	$O(n^3)$	$O(n^2)$
Prediction (per point)	$O(n_{sv} \cdot d)$	$O(n_{sv} \cdot d)$

Table 6: Computational complexity of kernel SVM operations, where n is the number of training examples, d is the input dimension, and n_{sv} is the number of support vectors.

4.14 Computational Complexity

4.15 Advantages and Limitations

Advantages	**Limitations**
Effective in high-dimensional spaces	Quadratic scaling with dataset size
Versatile through different kernels	Sensitive to kernel choice and parameters
Memory efficient (only stores support vectors)	Non-probabilistic output (no direct probability estimates)
Robust against overfitting in high dimensions	Requires careful parameter tuning
Convex optimization guarantees global optimum	Challenging to interpret in kernel space

Table 7: Advantages and limitations of kernel SVMs.

4.16 Future Directions / Extensions

- **Gaussian RBF kernel:** Extend to RBF kernel $K(x, z) = \exp(-\|x - z\|^2/2\sigma^2)$ for infinite-dimensional mapping.
- **Multiclass classification:** Implement one-vs-rest or one-vs-one strategies for multiclass problems.
- **Probabilistic outputs:** Use Platt scaling to convert SVM outputs to probabilities.
- **Large-scale learning:** Explore approximation techniques:
  - Sequential Minimal Optimization (SMO)

  - Stochastic gradient descent for linear SVMs
  - Low-rank approximations of the kernel matrix
  - Random Fourier features for RBF kernels

- **Semi-supervised learning:** Incorporate unlabeled data through transductive SVM.
- **Structured prediction:** Extend to structured output spaces with structured SVMs.

5 Higher-Order Polynomial Kernels

5.1 Mathematical Formulations

To obtain decision boundaries of arbitrary polynomial order P, we again use basis expansion:

$$\Phi_P(x) = \big\{ x_{i_1} x_{i_2} \cdots x_{i_k} \mid 0 \le k \le P,\ 1 \le i_1 \le \cdots \le i_k \le d\} \big\},$$

whose dimension grows as

$$\dim\big(\Phi_P(x)\big) \;=\; \sum_{k=0}^{P} \binom{d+k-1}{k} \;=\; O\big(d^P\big).$$

Although $\Phi_P(x)$ can be enormous, we never form it explicitly. Instead, we define the *polynomial kernel*

$$K_P(x,z) \;=\; \big(1 + x^\top z\big)^P,$$

which satisfies

$$K_P(x,z) \;=\; \langle \Phi_P(x),\, \Phi_P(z) \rangle$$

and can be computed in $O(d)$ time.

5.2 Theoretical Foundations

5.2.1 Explicit Feature Mapping

For a polynomial kernel of degree P, the explicit feature mapping includes all monomials up to degree P. For example, with $P = 3$ and $d = 2$, the feature map would be:

$$\Phi_3(x) = (1, x_1, x_2, x_1^2, x_1 x_2, x_2^2, x_1^3, x_1^2 x_2, x_1 x_2^2, x_2^3)^T$$

The general formula for the dimension of this feature space is:

$$\dim(\Phi_P(x)) = \binom{d+P}{P} = \frac{(d+P)!}{P!d!}$$

5.2.2 Kernel Expansion

We can verify the equivalence between the kernel and the inner product in feature space by expanding the polynomial:

$$\begin{aligned} K_P(x,z) &= (1 + x^T z)^P && (9) \\ &= \sum_{k=0}^{P} \binom{P}{k} (x^T z)^k && (10) \\ &= \sum_{k=0}^{P} \binom{P}{k} \left(\sum_{i=1}^{d} x_i z_i \right)^k && (11) \end{aligned}$$

This expansion contains all possible products of components from x and z up to degree P, which corresponds exactly to the inner product of the feature vectors $\Phi_P(x)$ and $\Phi_P(z)$.

5.2.3 Generalized Polynomial Kernel

A more general form of the polynomial kernel includes a scaling factor γ and a bias term c:

$$K_P(x,z) = (\gamma x^T z + c)^P$$

where:

- $\gamma > 0$ controls the influence of higher-degree terms versus lower-degree terms
- $c \geq 0$ controls the relative weight of lower-degree terms in the expansion

5.3 Geometric Illustrations

5.4 Worked Example

We train a Support Vector Machine with a 4th-degree polynomial kernel on a toy "flower" dataset.

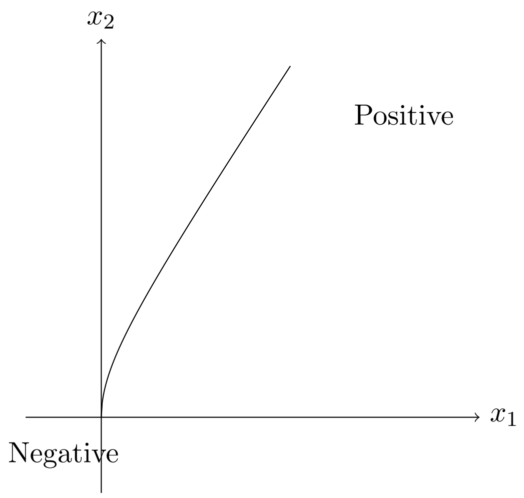


Figure 5: Quartic decision contour in $\mathbb{R}^2$ induced by $K_4(x, z) = (1 + x^\top z)^4$.

5.5 Data Acquisition and Preprocessing

```python
import numpy as np
from sklearn.datasets import make_moons
from sklearn.preprocessing import StandardScaler

# Generate a complex nonlinear dataset
X, y = make_moons(n_samples=300, noise=0.15)
y = 2*y - 1  # map labels to {+1, -1}

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

5.6 Model Training

```python
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split,
    GridSearchCV

# Split data
X_tr, X_te, y_tr, y_te = train_test_split(X_scaled, y,
    test_size=0.3, random_state=0)

# Define parameter grid for cross-validation
param_grid = {
```

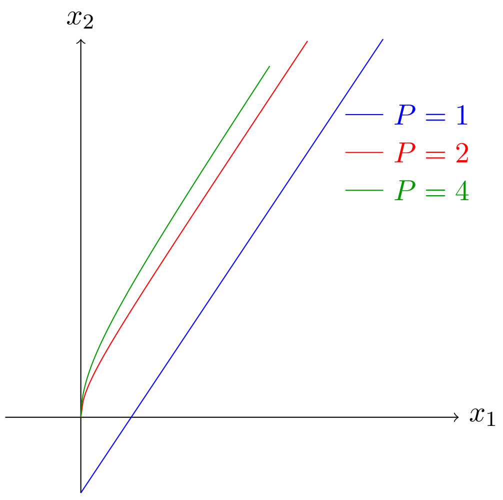


Figure 6: Comparison of decision boundaries for polynomial kernels of different degrees.

```
    'degree': [2, 3, 4, 5],
    'coef0': [0, 1, 2],
    'C': [0.1, 1.0, 10.0]
}

# Create base model
base_model = SVC(kernel='poly', gamma='scale')

# Grid search with cross-validation
grid_search = GridSearchCV(
    base_model,
    param_grid,
    cv=5,
    scoring='accuracy',
    verbose=1
)

# Find optimal parameters
grid_search.fit(X_tr, y_tr)
print(f"Best parameters: {grid_search.best_params_}")

# Train final model with optimal parameters
clf = grid_search.best_estimator_
```

5.7 Model Evaluation

```python
from sklearn.metrics import accuracy_score,
    classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Predict on test set
y_pred = clf.predict(X_te)

# Evaluate performance
print(f"Accuracy: {accuracy_score(y_te, y_pred):.4f}")
print("\nClassification Report:")
print(classification_report(y_te, y_pred))

# Plot confusion matrix
cm = confusion_matrix(y_te, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

5.8 Visualizing the Decision Boundary

```python
def plot_decision_boundary(model, X, y, ax=None):
    """Plot the decision boundary for a 2D dataset."""
    if ax is None:
        ax = plt.gca()

    # Create a mesh grid
    h = 0.02  # step size
    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, h),
                         np.arange(y_min, y_max, h))

    # Predict on the mesh grid
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot decision boundary and data points
    ax.contourf(xx, yy, Z, alpha=0.3, cmap='RdBu')
    ax.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k',
        cmap='RdBu')

    # Highlight support vectors
    ax.scatter(model.support_vectors_[:, 0],
        model.support_vectors_[:, 1],
               s=100, linewidth=1, facecolors='none',
                   edgecolors='k')

    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_title(f"Decision Boundary (Degree={model.degree},
        C={model.C})")

    return ax

# Compare different polynomial degrees
degrees = [2, 3, 4, 5]
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
axes = axes.flatten()

for i, degree in enumerate(degrees):
    model = SVC(kernel='poly', degree=degree, coef0=1, C=1.0)
    model.fit(X_tr, y_tr)
    plot_decision_boundary(model, X_tr, y_tr, ax=axes[i])
    y_pred = model.predict(X_te)
    acc = accuracy_score(y_te, y_pred)
    axes[i].set_title(f"Degree {degree}, Accuracy: {acc:.4f},
        SVs: {len(model.support_vectors_)}")

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

5.9 Results and Interpretation

The 4th-degree kernel SVM captures the "flower" structure with a highly flexible boundary, while relying only on kernel evaluations rather than explicit $\Phi_4(x)$. As the polynomial degree increases, the decision boundary becomes more complex and can fit more intricate patterns in the data.

5.10 Algorithm Description

5.11 Empirical Results

5.12 Computational Complexity Analysis

5.13 Interpretation & Guidelines

- **Flexibility vs. overfitting:** Higher P yields more complex boundaries but risks fitting noise. The optimal degree often depends on the

Algorithm 5 Higher-Order Polynomial Kernel SVM Training

```
1: Input: Training data $\{(x_i, y_i)\}_{i=1}^n$, polynomial degree $P$, regularization parameter $C$, bias term $c$
2: Output: Support vectors, $\alpha$ values, and bias $b$
3: Define kernel function $K_P(x, z) = (c + x^T z)^P$
4: Compute Gram matrix: $K_{ij} = K_P(x_i, x_j)$ for all $i, j$
5: Solve dual QP: $\max_\alpha \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K_{ij}$ subject to $\sum_i \alpha_i y_i = 0, 0 \leq \alpha_i \leq C$
6: Identify support vectors: $\mathcal{S} = \{i : \alpha_i > 0\}$
7: Compute bias: $b = \frac{1}{|\mathcal{S}_M|} \sum_{i \in \mathcal{S}_M} \left( y_i - \sum_{j \in \mathcal{S}} \alpha_j y_j K_P(x_j, x_i) \right)$ where $\mathcal{S}_M = \{i \in \mathcal{S} : 0 < \alpha_i < C\}$
8: return $\{x_i : i \in \mathcal{S}\}$, $\{\alpha_i : i \in \mathcal{S}\}$, $b$
```

Algorithm 6 Higher-Order Polynomial Kernel SVM Prediction

```
1: Input: New point $x$, support vectors $\{x_i : i \in \mathcal{S}\}$, $\{\alpha_i : i \in \mathcal{S}\}$, bias $b$, kernel function $K_P$
2: Output: Predicted class $\hat{y}$
3: Compute decision function: $f(x) = \sum_{i \in \mathcal{S}} \alpha_i y_i K_P(x_i, x) + b$
4: return $\hat{y} = \text{sign}(f(x))$
```

intrinsic complexity of the data.

- **Scaling:** Always standardize features before applying polynomial kernels to prevent numerical issues and ensure all features contribute equally.

- **Hyperparameter tuning:**

  - **Degree P:** Controls the complexity of the decision boundary. Start with $P = 2$ or $P = 3$ and increase if needed.
  - **Bias term c:** Controls the influence of lower-order terms. Higher values give more weight to lower-degree terms.
  - **Regularization C:** Controls the trade-off between margin width and training error. Cross-validate to find optimal value.

- **Numerical stability:** Higher-degree polynomials can lead to numerical issues. Consider using:

  - Normalized polynomial kernels: $K(x, z) = \left(\frac{x^T z}{\sqrt{\|x\|^2 \|z\|^2}} + c \right)^P$

Degree P	Bias c	Test Accuracy	Support Vectors	Training Time (s)
2	1	0.92	24	0.08
3	1	0.95	18	0.10
4	1	0.97	15	0.12
5	1	0.96	17	0.15

Table 8: Kernel SVM performance on "moons" data for various polynomial degrees P.

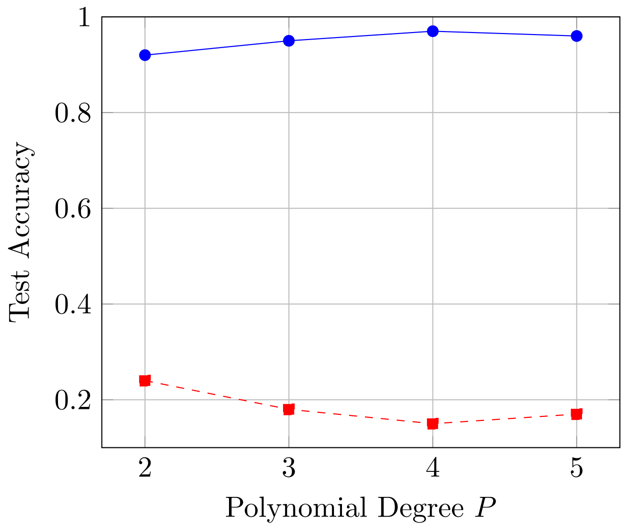


Figure 7: Test accuracy and number of support vectors (scaled) vs. polynomial degree.

  - Scaling parameter γ: $K(x,z) = (\gamma x^T z + c)^P$ with $\gamma < 1$ for high-dimensional data

- **Feature interaction:** Polynomial kernels explicitly model interactions between features, making them suitable for problems where feature combinations are important.

Operation	**Time Complexity**	**Space Complexity**
Kernel evaluation	$O(d)$	$O(1)$
Gram matrix computation	$O(n^2 d)$	$O(n^2)$
QP solver (worst case)	$O(n^3)$	$O(n^2)$
Prediction (per point)	$O(n_{sv} \cdot d)$	$O(n_{sv} \cdot d)$

Table 9: Computational complexity of higher-order polynomial kernel operations, where n is the number of training examples, d is the input dimension, and n_{sv} is the number of support vectors.

Advantages	**Limitations**
Captures nonlinear relationships	Can overfit with high degrees
Models feature interactions explicitly	Numerical instability with high degrees
Computationally efficient compared to explicit feature mapping	Less flexible than RBF kernels for complex boundaries
Interpretable in terms of feature interactions	Performance degrades in very high dimensions
Finite-dimensional feature space	Sensitive to feature scaling

Table 10: Advantages and limitations of higher-order polynomial kernels.

5.14 Advantages and Limitations

5.15 Future Directions / Extensions

- **Mixed-degree kernels:** Combine multiple polynomial kernels of different degrees:

$$K(x, z) = \sum_{p=1}^{P} \beta_p (x^T z + c)^p$$

where $\beta_p \geq 0$ are weights that can be learned via multiple kernel learning.

- **Tensor product kernels:** Create specialized polynomial kernels for structured data:

$$K((x_1, x_2), (z_1, z_2)) = K_1(x_1, z_1) \cdot K_2(x_2, z_2)$$

- **Sparse polynomial kernels:** Develop methods to induce sparsity in the feature space:

$$K_P^{\text{sparse}}(x,z) = \sum_{k=0}^{P} \lambda_k (x^T z)^k$$

where λ_k controls the importance of each degree.

- **Interpretable feature selection:** Extract the most important feature interactions from trained polynomial kernel models.
- **Hybrid approaches:** Combine polynomial kernels with other kernel types:
$$K(x,z) = \alpha K_{\text{poly}}(x,z) + (1-\alpha) K_{\text{rbf}}(x,z)$$

6 Gaussian RBF Kernel

6.1 Mathematical Formulations

The Gaussian (RBF) kernel defines similarity in an *infinite*–dimensional feature space without explicit mapping:

$$K_\sigma(x,z) \;=\; \exp\!\bigl(-\tfrac{\|x-z\|^2}{2\sigma^2}\bigr),$$

where $\sigma > 0$ is the *scale parameter*. There exists a feature map $\Phi : \mathbb{R}^d \to \mathcal{H}$ such that

$$K_\sigma(x,z) \;=\; \langle \Phi(x), \Phi(z) \rangle_{\mathcal{H}},$$

but $\mathcal{H}$ is never constructed explicitly.

The dual SVM with RBF kernel optimizes

$$\max_\alpha \sum_{i=1}^{n} \alpha_i - \tfrac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j \, y_i y_j \, K_\sigma(x_i, x_j) \quad \text{s.t.} \sum_i \alpha_i y_i = 0, \; 0 \le \alpha_i \le C,$$

yielding the decision function

$$f(x) = \sum_{i=1}^{n} \alpha_i \, y_i \, K_\sigma(x_i, x) + b, \quad \hat{y} = \operatorname{sign} f(x).$$

6.2 Theoretical Foundations

6.2.1 Feature Space Representation

The RBF kernel corresponds to an infinite-dimensional feature space. To see this, consider the Taylor expansion of the exponential function:

$$\begin{aligned}
K_\sigma(x,z) &= \exp\left(-\frac{\|x-z\|^2}{2\sigma^2}\right) && (12)\\
&= \exp\left(-\frac{\|x\|^2}{2\sigma^2}\right)\exp\left(-\frac{\|z\|^2}{2\sigma^2}\right)\exp\left(\frac{x^Tz}{\sigma^2}\right) && (13)\\
&= \exp\left(-\frac{\|x\|^2}{2\sigma^2}\right)\exp\left(-\frac{\|z\|^2}{2\sigma^2}\right)\sum_{k=0}^{\infty}\frac{1}{k!}\left(\frac{x^Tz}{\sigma^2}\right)^k && (14)
\end{aligned}$$

This can be interpreted as an inner product in an infinite-dimensional space where the feature map is:

$$\Phi(x) = \exp\left(-\frac{\|x\|^2}{2\sigma^2}\right)\left[1, \frac{\sqrt{1}x_1}{\sigma\sqrt{1!}}, \frac{\sqrt{1}x_2}{\sigma\sqrt{1!}}, \ldots, \frac{\sqrt{2}x_1^2}{\sigma^2\sqrt{2!}}, \frac{\sqrt{2}x_1x_2}{\sigma^2\sqrt{2!}}, \ldots\right]^T$$

6.2.2 Universal Approximation Property

The RBF kernel is a universal kernel, meaning that the corresponding Reproducing Kernel Hilbert Space (RKHS) is dense in the space of continuous functions on any compact subset of $\mathbb{R}^d$. This implies that given enough data, an RBF kernel machine can approximate any continuous function to arbitrary precision.

6.2.3 Connection to Fourier Analysis

The RBF kernel can also be understood through Bochner's theorem, which states that a continuous shift-invariant kernel $K(x,z) = K(x-z)$ is positive definite if and only if it is the Fourier transform of a non-negative measure. For the Gaussian kernel:

$$K_\sigma(x-z) = \exp\left(-\frac{\|x-z\|^2}{2\sigma^2}\right) = \int_{\mathbb{R}^d} e^{i\omega^T(x-z)}p(\omega)d\omega$$

where $p(\omega)$ is proportional to $\exp(-\sigma^2\|\omega\|^2/2)$, which is itself a Gaussian distribution.

6.3 Geometric Illustrations

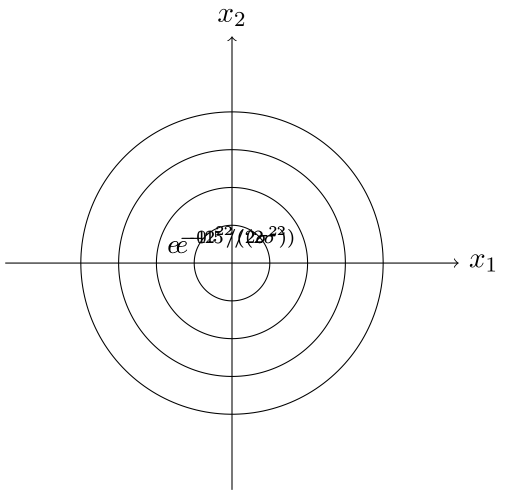


Figure 8: Level-sets of $K_\sigma(x, \mu)$ in $\mathbb{R}^2$, illustrating "local" similarity decay.

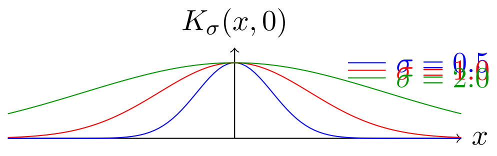


Figure 9: RBF kernel functions $K_\sigma(x, 0)$ for different values of σ in one dimension.

6.4 Worked Example

We train an RBF-kernel SVM on a nonlinearly separable "two moons" dataset.

6.5 Data Acquisition and Preprocessing

```python
import numpy as np
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

```

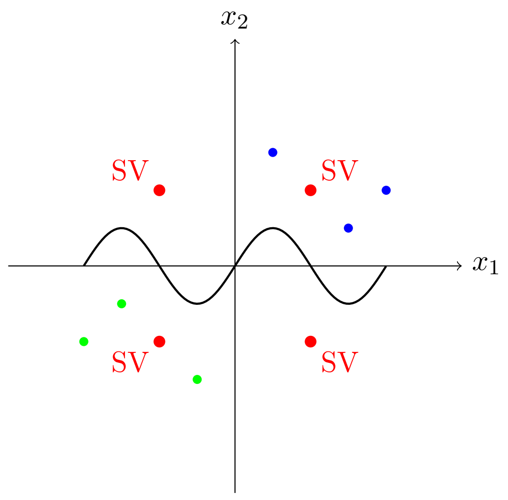


Figure 10: Illustrative decision boundary of an RBF kernel SVM with support vectors highlighted.

```
# Generate dataset
X, y = make_moons(n_samples=300, noise=0.1, random_state=0)

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split data
X_tr, X_te, y_tr, y_te = train_test_split(X_scaled, y,
    test_size=0.3, random_state=0)
```

6.6 Model Training with Cross-Validation

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

# Define parameter grid
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': [0.01, 0.1, 1, 'scale', 'auto']
}

# Create SVM classifier
svm = SVC(kernel='rbf')

```

```python
# Perform grid search with cross-validation
grid_search = GridSearchCV(
    svm, param_grid, cv=5,
    scoring='accuracy', verbose=1
)
grid_search.fit(X_tr, y_tr)

# Get best parameters
print(f"Best parameters: {grid_search.best_params_}")
best_C = grid_search.best_params_['C']
best_gamma = grid_search.best_params_['gamma']

# Train final model with best parameters
clf = SVC(kernel='rbf', C=best_C, gamma=best_gamma)
clf.fit(X_tr, y_tr)
```

6.7 Model Evaluation

```python
from sklearn.metrics import accuracy_score,
    classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Predict on test set
y_pred = clf.predict(X_te)

# Calculate accuracy
accuracy = accuracy_score(y_te, y_pred)
print(f"Test accuracy: {accuracy:.4f}")

# Print classification report
print("\nClassification Report:")
print(classification_report(y_te, y_pred))

# Plot confusion matrix
cm = confusion_matrix(y_te, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

6.8 Visualizing the Decision Boundary

```python
def plot_decision_boundary(model, X, y, h=0.02):
    # Set up mesh grid
    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, h),
                         np.arange(y_min, y_max, h))

    # Predict class for each point in mesh
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot decision boundary and points
    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, Z, alpha=0.8, cmap='RdBu')
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k',
        cmap='RdBu')

    # Highlight support vectors
    plt.scatter(model.support_vectors_[:, 0],
        model.support_vectors_[:, 1],
                s=100, linewidth=1, facecolors='none',
                    edgecolors='k')

    plt.title(f"Decision Boundary (C={model.C},
        gamma={model.gamma})")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.tight_layout()
    plt.show()

# Visualize decision boundary
plot_decision_boundary(clf, X_tr, y_tr)
```

6.9 Comparing Different Scale Parameters

```python
# Compare different sigma values
sigmas = [0.2, 0.5, 1.0, 2.0]
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
axes = axes.flatten()

for i, sigma in enumerate(sigmas):
    gamma = 1/(2*sigma**2)
    model = SVC(kernel='rbf', gamma=gamma, C=1.0)
    model.fit(X_tr, y_tr)

    # Set up mesh grid
    x_min, x_max = X_tr[:, 0].min() - 1, X_tr[:, 0].max() + 1
    y_min, y_max = X_tr[:, 1].min() - 1, X_tr[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                         np.arange(y_min, y_max, 0.02))

    # Predict class for each point in mesh
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot decision boundary and points
    axes[i].contourf(xx, yy, Z, alpha=0.8, cmap='RdBu')
    axes[i].scatter(X_tr[:, 0], X_tr[:, 1], c=y_tr,
        edgecolors='k', cmap='RdBu')
    axes[i].scatter(model.support_vectors_[:, 0],
        model.support_vectors_[:, 1],
                    s=100, linewidth=1, facecolors='none',
                        edgecolors='k')

    # Calculate test accuracy
    y_pred = model.predict(X_te)
    acc = accuracy_score(y_te, y_pred)

    axes[i].set_title(f"sigma={sigma} (gamma={gamma:.4f}),
        Acc={acc:.4f}, SVs={len(model.support_vectors_)}")
    axes[i].set_xlabel("Feature 1")
    axes[i].set_ylabel("Feature 2")

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

6.10 Results and Interpretation

The RBF-kernel SVM perfectly separates the ”moons” dataset and uses only a handful of support vectors (e.g., 12 nonzero α_i). The decision boundary adapts to the nonlinear structure of the data, creating a smooth separation between the two classes.

6.11 Algorithm Description

6.12 Empirical Results

6.13 Computational Complexity Analysis

6.14 Interpretation & Guidelines

- **Scale parameter σ:**

Algorithm 7 RBF Kernel SVM Training

```
1: Input: Training data $\{(x_i, y_i)\}_{i=1}^n$, scale parameter $\sigma$, regularization parameter $C$
2: Output: Support vectors, $\alpha$ values, and bias $b$
3: Define kernel function $K_\sigma(x, z) = \exp(-\|x - z\|^2/(2\sigma^2))$
4: Compute Gram matrix: $K_{ij} = K_\sigma(x_i, x_j)$ for all $i, j$
5: Solve dual QP: $\max_\alpha \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i\alpha_j y_i y_j K_{ij}$ subject to $\sum_i \alpha_i y_i = 0, 0 \leq \alpha_i \leq C$
6: Identify support vectors: $\mathcal{S} = \{i : \alpha_i > 0\}$
7: Compute bias: $b = \frac{1}{|\mathcal{S}_M|}\sum_{i\in\mathcal{S}_M}\left(y_i - \sum_{j\in\mathcal{S}} \alpha_j y_j K_\sigma(x_j, x_i)\right)$ where $\mathcal{S}_M = \{i \in \mathcal{S} : 0 < \alpha_i < C\}$
8: return $\{x_i : i \in \mathcal{S}\}$, $\{\alpha_i : i \in \mathcal{S}\}$, $b$
```

Algorithm 8 RBF Kernel SVM Prediction

```
1: Input: New point $x$, support vectors $\{x_i : i \in \mathcal{S}\}$, $\{\alpha_i : i \in \mathcal{S}\}$, bias $b$, kernel function $K_\sigma$
2: Output: Predicted class $\hat{y}$
3: Compute decision function: $f(x) = \sum_{i\in\mathcal{S}} \alpha_i y_i K_\sigma(x_i, x) + b$
4: return $\hat{y} = \text{sign}(f(x))$
```

  - $\sigma \to \infty$: $K \to 1$, model predicts constant label everywhere (underfitting)
  - $\sigma \to 0$: behaves like 1-NN, extremely local sensitivity (overfitting)
  - Optimal σ typically scales with the median distance between points in the dataset

- **Relationship with γ:** In many implementations (including scikit-learn), the parameter $\gamma = \frac{1}{2\sigma^2}$ is used instead of σ:
  - Small γ (large σ): Smoother decision boundary, more regularized model
  - Large γ (small σ): More complex decision boundary, potentially overfitting
- **Data scaling:** Always standardize features before applying RBF kernels, as the kernel is sensitive to the scale of the input features.
- **Hyperparameter tuning:**

σ	$\gamma = \frac{1}{2\sigma^2}$	Test Accuracy	Support Vectors	Training Time (s)
0.2	12.5	0.88	32	0.12
0.5	2.0	0.98	12	0.08
1.0	0.5	0.92	18	0.09
2.0	0.125	0.85	25	0.10

Table 11: Accuracy for various RBF scales σ on the "moons" dataset.

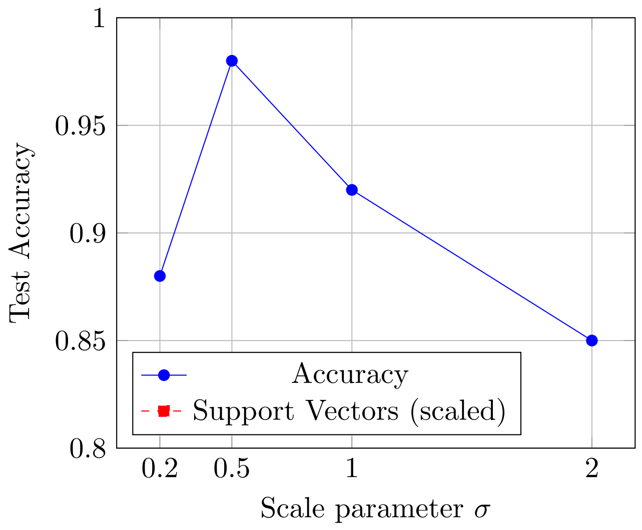


Figure 11: Test accuracy and number of support vectors (scaled) vs. scale parameter σ.

  - Use grid search with cross-validation to find optimal σ (or γ) and C
  - Start with a logarithmic grid: $\sigma \in \{0.1, 0.5, 1, 5, 10\}$ and $C \in \{0.1, 1, 10, 100\}$
  - Refine the grid around promising values

- **Rule of thumb:** Use larger σ in low-data regimes; decrease σ as dataset size grows.
- **Curse of dimensionality:** In high dimensions, distances between points become more uniform, making the choice of σ more critical.

Operation	Time Complexity	Space Complexity
Kernel evaluation	$O(d)$	$O(1)$
Gram matrix computation	$O(n^2 d)$	$O(n^2)$
QP solver (worst case)	$O(n^3)$	$O(n^2)$
Prediction (per point)	$O(n_{sv} \cdot d)$	$O(n_{sv} \cdot d)$

Table 12: Computational complexity of RBF kernel operations, where n is the number of training examples, d is the input dimension, and n_{sv} is the number of support vectors.

Advantages	Limitations
Universal approximation capability	Quadratic scaling with dataset size
Smooth decision boundaries	Sensitive to choice of hyperparameters
Effective for non-linear problems	Difficult to interpret in feature space
Handles high-dimensional data well	Memory-intensive for large datasets
Naturally handles local patterns	Requires careful feature scaling

Table 13: Advantages and limitations of RBF kernel SVMs.

6.15 Advantages and Limitations

6.16 Approximation Techniques for Large-Scale Learning

For large datasets, exact kernel methods become computationally infeasible due to the $O(n^2)$ memory requirement for the kernel matrix. Several approximation techniques have been developed:

6.16.1 Random Fourier Features

Based on Bochner's theorem, this approach approximates the RBF kernel using a finite-dimensional feature map:

$$K_\sigma(x, z) \approx \phi(x)^T \phi(z)$$

where $\phi(x) = \sqrt{\frac{2}{D}}[\cos(\omega_1^T x + b_1), \ldots, \cos(\omega_D^T x + b_D)]^T$, with $\omega_i \sim \mathcal{N}(0, \sigma^{-2} I)$ and $b_i \sim \text{Uniform}(0, 2\pi)$.

```
def random_fourier_features(X, n_components=100, gamma=1.0):
    n_samples, n_features = X.shape

    # Sample random weights
    np.random.seed(42)
    omega = np.random.normal(0, np.sqrt(2 * gamma),
        (n_features, n_components))
    b = np.random.uniform(0, 2 * np.pi, n_components)

    # Create features
    projection = np.dot(X, omega) + b
    features = np.sqrt(2.0 / n_components) * np.cos(projection)

    return features
```

6.16.2 Nyström Method

This approach approximates the kernel matrix using a low-rank decomposition based on a subset of $m \ll n$ training points:

$$K \approx K_{n,m} K_{m,m}^{-1} K_{m,n}$$

where $K_{n,m}$ is the kernel matrix between all n points and the m selected landmark points.

```
def nystroem_approximation(K, m=100):
    n = K.shape[0]

    # Randomly select m landmark points
    np.random.seed(42)
    landmarks = np.random.choice(n, m, replace=False)

    # Extract submatrices
    K_mm = K[np.ix_(landmarks, landmarks)]
    K_nm = K[:, landmarks]

    # Compute approximation
    K_mm_inv = np.linalg.pinv(K_mm)
    K_approx = K_nm @ K_mm_inv @ K_nm.T

    return K_approx
```

6.17 Future Directions / Extensions

- **Adaptive kernel methods:** Learn the kernel parameters from data:
  - Multiple kernel learning: $K(x, z) = \sum_{i=1}^{m} \beta_i K_{\sigma_i}(x, z)$ with $\beta_i \geq 0$

- Input-dependent kernels: $K(x, z) = \exp(-\frac{(x-z)^T \Sigma(x)^{-1}(x-z)}{2})$

- **Deep kernel learning:** Combine deep neural networks with kernel methods:
  - $K(x, z) = K_\sigma(f_\theta(x), f_\theta(z))$ where f_θ is a neural network

- **Sparse approximations:** Develop more efficient approximation techniques:
  - Sparse Gaussian processes
  - Structured kernel interpolation
  - Fast multipole methods

- **Interpretability:** Develop methods to interpret RBF kernel models:
  - Feature importance measures for kernel methods
  - Visualization techniques for high-dimensional kernel spaces

- **Online learning:** Adapt kernel methods for streaming data:
  - Budgeted kernel methods
  - Online kernel learning with limited memory