2025-04-28 00:43

tags :

- Machine Learning
- Hands on ML - book

Chapter 5 - Hands on

- **SVMs** are powerful models for **classification**, **regression**, and **novelty detection**.
- They work best on **small to medium-sized**, **nonlinear** datasets but don't scale well to **very large datasets**.

Linear SVM Classification

- **Goal**: Find a straight line (decision boundary) that separates two classes **with the largest margin**.
- **Large / hard Margin Classification**: SVM chooses the line that leaves the **widest possible street** between classes.
- **Support Vectors**: Only the instances on the edge of the street (circled points) determine the decision boundary.
- important Warning
  - **SVMs are sensitive to feature scales**.
  - If features aren't scaled (e.g., vertical much bigger than horizontal), the SVM decision boundary can be wrong.
  - After **feature scaling** (like using `StandardScaler`), the boundary becomes better and more balanced. illustration [1]

Soft Margin Classification

- **Hard Margin Classification** forces all points to stay outside the margin — works only if data is perfectly separable and is **very sensitive to outliers**.
- **Soft Margin Classification** allows some violations (some points inside the margin or misclassified) to balance between **margin size** and **errors** for better generalization.
- **C Hyperparameter** controls the trade-off:
  - **Small C** → Larger margin, more violations → Less overfitting but risk of underfitting.
  - **Large C** → Smaller margin, fewer violations → Risk of overfitting but fits training data better.

- **Tip**: If the SVM overfits, **reduce C** to regularize.
- **Code Example**: Using `LinearSVC` inside a `Pipeline` (with `StandardScaler`) to classify Iris Virginica flowers.

```python
from sklearn.datasets import load_iris
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 2) # Iris virginica
svm_clf = make_pipeline(StandardScaler(),
                        LinearSVC(C=1, random_state=42))
svm_clf.fit(X, y)

X_new = [[5.5, 1.7], [5.0, 1.5]]
svm_clf.predict(X_new)  # O/P  =>  array([ True, False])
```

- **Decision Function**: Outputs a score showing how far each instance is from the decision boundary.

```python
svm_clf.decision_function(X_new)  # O/P => array([ 0.66163411,
-0.22036063])
```

- **Note**: `LinearSVC` doesn't support probability predictions, but `SVC(probability=True)`
  - Normally, `LinearSVC` only gives a **score**, not a **probability**.
  - If you use `SVC(probability=True)`:
    - The model learns to **map scores to probabilities** (e.g., 80% confidence).
    - To do that, it must:
      - Run **5-fold cross-validation** during training.
      - Train a **Logistic Regression model** on the cross-validation results to map scores to probabilities.
  - ➡️ **This extra work makes training slower**.

Nonlinear SVM Classification

- Many datasets are **not linearly separable**.
- You can **add new features** (like polynomials) to make the data **linearly separable** in a higher-dimensional space.

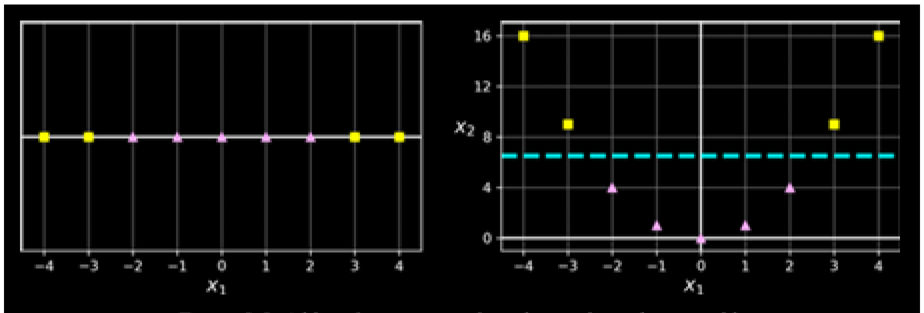
- Example: Adding a second feature like $x_2 = x_1^2$ makes a previously non-separable dataset separable.
  - 

    Figure 5-5. Adding features to make a dataset linearly separable
- **How to do it**:
  - Use `PolynomialFeatures` to create new features.
  - Then scale features with `StandardScaler`.
  - Then train a `LinearSVC` model.
  - Example code:

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.svm import LinearSVC

X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge", random_state=42))
])

polynomial_svm_clf.fit(X, y)
```

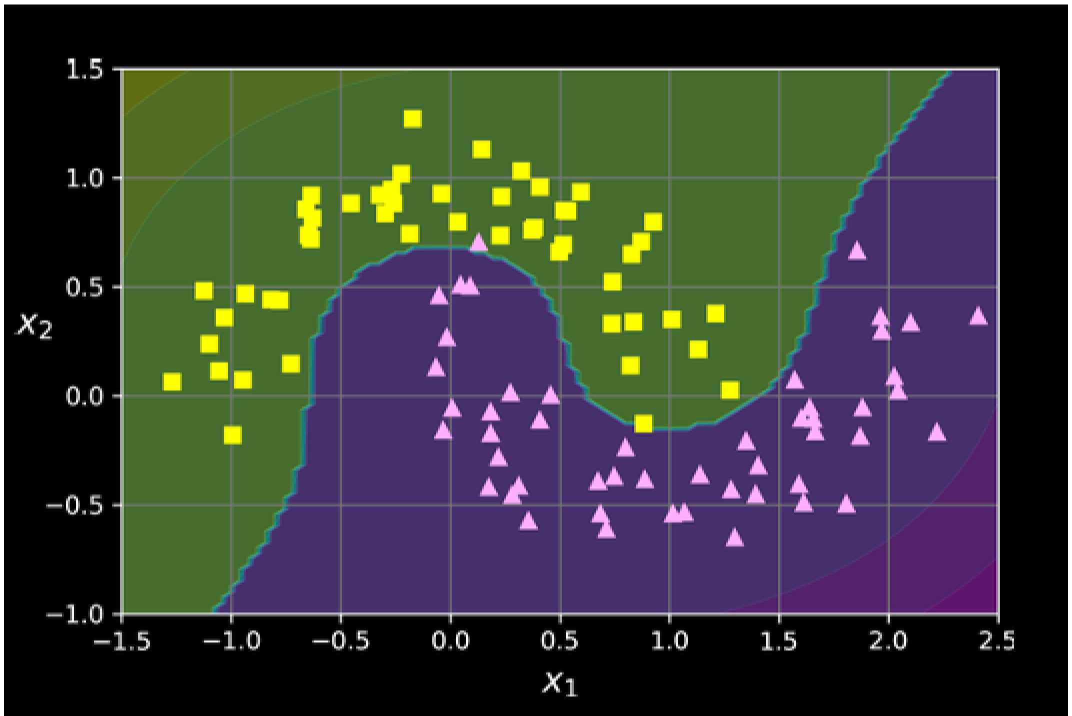
- 

Figure 5-6. Linear SVM classifier using polynomial features

Polynomial Kernel

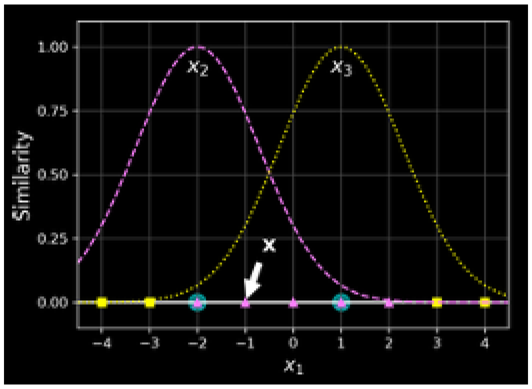
- Adding polynomial features can help models handle nonlinearity, but:
  - Low degrees can't model complex patterns well.
  - High degrees cause **too many features** → slow models.
- **Kernel trick**: SVMs can act like they added high-degree polynomial features **without** actually creating them — no feature explosion!
- Use `SVC` with `kernel="poly"` to apply the polynomial kernel easily.
- Example:

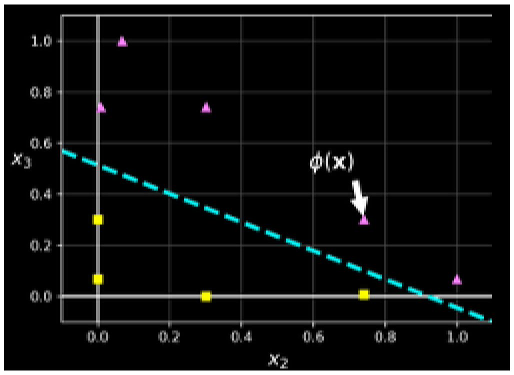
```
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

poly_kernel_svm_clf = make_pipeline(
    StandardScaler(),
    SVC(kernel="poly", degree=3, coef0=1, C=5)
)
poly_kernel_svm_clf.fit(X, y)
```

- **Important hyperparameters**:
  - `degree`: degree of the polynomial (higher = more flexible, but risk of overfitting).
  - `coef0`: controls balance between high-degree and low-degree features.

- **Tips**:
  - If overfitting → **lower** the degree.
  - If underfitting → **raise** the degree.
- Even when hyperparameters are tuned automatically, it's important to **understand them** to limit the search space. use randomized search

Similarity Features

- Another way to handle **nonlinear data** is by **adding features based on similarity** to certain **landmarks**.
- Example:
  - Landmarks placed at x=−2 and x=1.
  - Use **Gaussian RBF** (Radial Basis Function) with γ=0.3 to measure similarity.
  - Similarity is high (near 1) if close to a landmark, and low (near 0) if far.
  - 

- For instance x=−1:
  - Similarity to first landmark: about **0.74**.
  - Similarity to second landmark: about **0.30**.
- After transformation, the dataset can become **linearly separable**.
  - 

- **Landmark selection**:
  - Easiest way: make every training instance a landmark.
  - Downside: if you have **m instances**, you get **m features** — can be very **computationally heavy** for large datasets.

Gaussian RBF Kernel

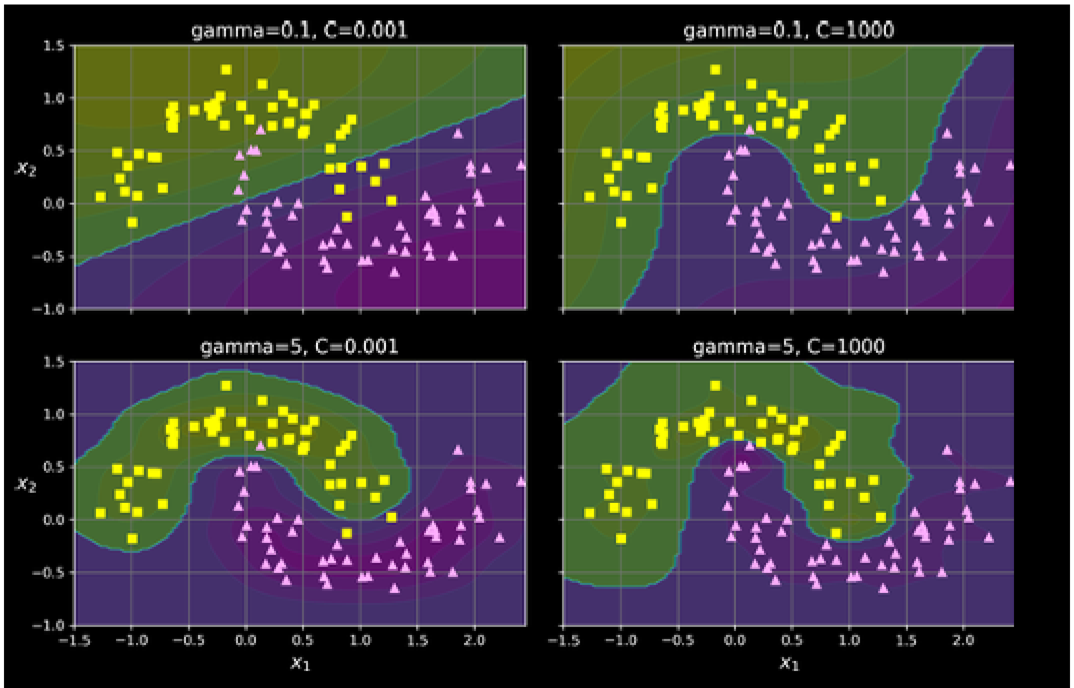
- Like polynomial features, similarity features can be useful but **computationally heavy**.
- The **kernel trick** in SVMs allows getting the same effect **without** explicitly adding features.
- Example: using `SVC(kernel="rbf", gamma=5, C=0.001)`.
- **Gamma (γ)**:
    - Higher γ → narrower bell curve → smaller influence → **more irregular (overfitting)** decision boundary.
    - Lower γ → wider bell curve → larger influence → **smoother (underfitting)** decision boundary.
    - 

Figure 5-9. SVM classifiers using an RBF kernel
- γ acts like a **regularization hyperparameter** (similar to C).
- Other kernels exist (e.g., for text or DNA sequences) but are less common.
- **Tip**:
    - Try a **linear kernel first** (preferably `LinearSVC` for speed).
    - If the dataset is small enough, try **Gaussian RBF kernel** next.
    - Experiment with other specialized kernels if needed.

SVM Classes and Computational Complexity

- **LinearSVC**:
    - Based on **liblinear**.
    - **No kernel trick**.
    - Fast: **O(m × n)** time complexity.
    - Needs scaling.

    - No out-of-core learning. explain -> [2]
- **SVC**:
    - Based on **libsvm**.
    - **Supports kernel trick** (for nonlinear problems).
    - **Slow** on large datasets: **$O(m^2 \times n)$** to **$O(m^3 \times n)$**.
    - Needs scaling.
    - No out-of-core learning.
- **SGDClassifier**:
    - Uses **stochastic gradient descent**.
    - **$O(m \times n)$** time complexity.
    - **Supports out-of-core learning** (good for huge datasets).
    - Needs scaling.
    - No kernel trick.

SVM Regression

- **SVM for Regression:**
    - Instead of separating classes, SVM tries to fit as many points **inside a margin ("street")** as possible.
    - The **width of the street** is controlled by **epsilon (ϵ)**.
    - Points inside the margin **don't affect** the model → called **ϵ-insensitive**.
- **LinearSVR**: `from sklearn.svm import LinearSVR`
    - For **linear regression** (like a straight line).
    - Fast and scales well to large datasets.
- **SVR** (with kernel): `from sklearn.svm import SVR`
    - For **nonlinear regression** (curves, etc.).
    - Slow on large datasets because it uses the **kernel trick**.
- **C hyperparameter**:
    - Controls how **strict** the model is about margin violations.
    - **Small C** → more flexible (allow violations, more regularization).
    - **Large C** → less flexible (less violations, tries to fit the training data closely).
- **epsilon (ϵ) hyperparameter**:
    - Controls the **width of the margin ("street")** where no penalty is given for errors.
    - **Small ϵ** → narrow margin → **more support vectors** → **more complex model**.
    - **Large ϵ** → wide margin → simpler model.

Concept	C (penalty)	epsilon (street width)
Controls	How much you allow errors outside margin	How wide the no-penalty zone is
Small value	More regularization, allow more errors	Narrower street, more support vectors
Large value	Fit more tightly to training data	Wider street, fewer support vectors

Under the Hood of Linear SVM Classifiers

- A **linear SVM classifier** predicts the class of an input by computing a **decision function**: $w^\top x + b$
  - If the result is **positive**, it predicts class **1**.
  - If the result is **negative**, it predicts class **0**.
- This is similar to **Logistic Regression** in prediction style.
- **Two notations for parameters**:
  - **Old way**: Combine weights and bias into one vector θ (with a fake feature $x_0 = 1$).
  - **New way (standard)**:
    - **w** = weights vector
    - **b** = bias (separate)
      So, prediction = $w^\top x + b$.
- **Training a Linear SVM**:
  - **Goal**:
    - **Maximize the margin** (make the street between classes as wide as possible).
    - **Minimize margin violations** (keep instances outside the street as few as possible).
  - **Key ideas**:
    - **Smaller w → larger margin** (margin $\propto 1/\|w\|$).
    - **Bias b** only **shifts** the street but **does not change** its width.
- **Optimization Problem (Hard Margin SVM)**:
  - Minimize:
    $\frac{1}{2}w^\top w$
- Subject to:
  $t^{(i)}(w^\top x^{(i)} + b) \geq 1 \quad \text{for all instances}$
  where $t^{(i)}$ = 1 if positive class, –1 if negative class.
- We minimize $\frac{1}{2}w^\top w$ instead of $\|w\|$ because:
  - It's **differentiable** (makes optimization easier).

    - Derivative of $\frac{1}{2}w^\top w$ is simply w.

Concept	Meaning
w small	Larger margin (better separation)
b	Shifts the decision boundary (no size change)
Objective	Minimize ½‖w‖² while keeping predictions correct
Prediction Rule	Positive if w⊤x+b>0w^\top x + b > 0w⊤x+b>0, negative otherwise

- In real-world data, **perfect separation** may be impossible.
- To handle this, we introduce **slack variables** $\zeta^{(i)} \geq 0$ for each instance:
    - Each ζ measures **how much margin violation** is allowed for that instance.
- **Soft Margin SVM Objective**:
    - We want to **Minimize** both:
        - The model complexity (½‖w‖² → wider margin)
        - The total margin violations (sum of ζ's)
    - **C hyperparameter** controls the **trade-off** between margin size and violations:
        - High **C** → less tolerance for margin violations (harder margin).
        - Low **C** → more tolerance (softer margin).
    - The optimization problem becomes: $\text{Minimize} \quad \frac{1}{2}w^\top w + C\sum_{i=1}^{m}\zeta^{(i)}$
    - subject to: $t^{(i)}(w^\top x^{(i)} + b) \geq 1 - \zeta^{(i)}, \quad \zeta^{(i)} \geq 0$
- **Optimization Techniques**:
    - These are **convex quadratic programming (QP)** problems (special type of optimization with linear constraints).
    - **Training options**:
        - Use a **QP solver** (specialized software).
        - Use **gradient descent** by minimizing:
            - **Hinge loss** (linear penalty)
            - **Squared hinge loss** (quadratic penalty, more sensitive to outliers).
- **Hinge Loss Behavior**:

	Positive class (t=1)	Negative class (t=−1)
Correctly classified & far from margin	Loss = 0	Loss = 0
Inside margin or wrong side	Loss > 0	Loss > 0

- **Hinge loss** grows **linearly** with error.

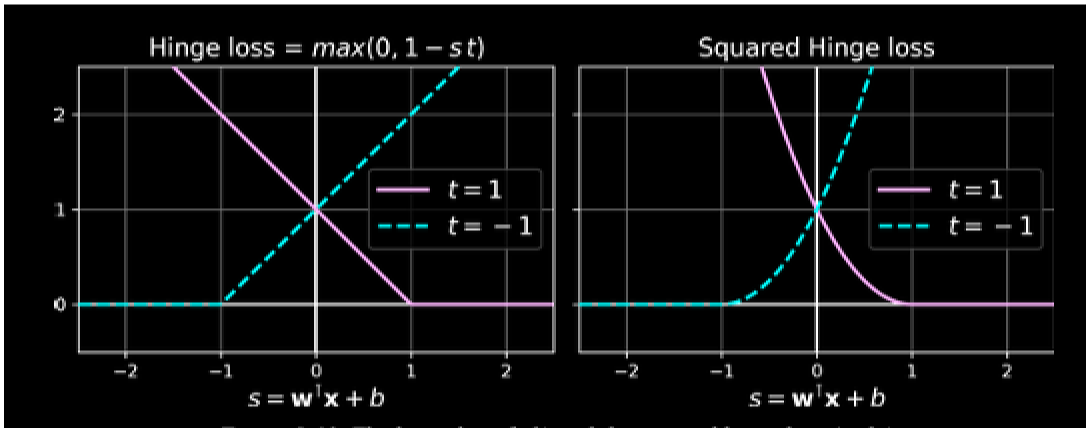
- **Squared hinge loss** grows **quadratically** with error (faster convergence if clean data).
- 

Figure 5-13. The hinge loss (left) and the squared hinge loss (right) *
- **In Scikit-Learn**:
  - **LinearSVC**: uses **squared hinge loss** by default.
  - **SGDClassifier**: uses **hinge loss** by default.
  - **SVC**: similar to minimizing hinge loss but with kernels.

Name	What it Is	Controls	Used in
C (capital letter)	Hyperparameter chosen by you	Trade-off between margin size and margin violations	Classification & Regression
ε (epsilon)	Width of the "street" (margin) where no penalty happens in **regression**	Tolerance for errors inside the margin (in regression)	**SVM Regression (SVR)**
ζ (zeta)	Slack variables computed during training	How much each individual training point **violates** the margin (in classification)	**SVM Classification (Soft margin)**

The Dual Problem

➔ What is happening?

- **SVM training** can be done using the **primal problem** or the **dual problem**.
- In SVM, the **dual** and **primal** give **the same solution** (because of special conditions).
- The **dual problem** is often easier when:
  - The number of training samples **m** is **less than** the number of features.
  - You want to use the **kernel trick** later.
- Dual Problem Equation
  We **solve for α by minimizing:

$$\text{minimize} \quad \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}\left(x^{(i)}\cdot x^{(j)}\right)-\sum_{i=1}^{m}\alpha^{(i)}$$

subject to: $\alpha^{(i)} \geq 0 \quad \text{for all } i = 1, 2, \ldots, m$

$$\sum_{i=1}^{m} \alpha^{(i)} t^{(i)} = 0$$

- $\alpha^{(i)}$ = slack related values (Lagrange multipliers).
- $t^{(i)}$ = class label (+1 or -1).
- $x^{(i)}$ = feature vectors.

- From Dual Solution to Primal Solution
  Once you find the optimal $\hat{\alpha}$, you can **rebuild** your model:
  - **Weights vector $\hat{w}$** :

$$\hat{w} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} x^{(i)}$$

  - **Bias term $\hat{b}$:

$$\hat{b} = \frac{1}{n_s} \sum_{\text{support vectors}} \left(t^{(i)} - \hat{w}^{\top} x^{(i)} \right)$$

  - where:
    - n_s = number of **support vectors** (i.e., data points where $\hat{\alpha}^{(i)} > 0$.
- Quick notes:
  - **Support vectors** are the important data points that "touch" the margin.
  - If the dataset is very big, **dual** can be slower (but necessary for kernels).
  - **Primal** is faster for very large datasets with simple (linear) relationships.

Kernelized SVMs

- Problem:
  - You want to apply a **second-degree polynomial transformation** ϕ(x)\phi(x)ϕ(x) to your 2D data to make it easier to separate with a **linear SVM**.
    Transformation (Equation 5-5):
    $\phi(\mathbf{x}) = \begin{pmatrix} x_1^2 \\ \sqrt{2} x_1 x_2 \\ x_2^2 \end{pmatrix}$
    After transformation, the data becomes **3D** instead of 2D.
- Kernel Trick:
  - Instead of explicitly computing ϕ(x)\phi(x)ϕ(x), notice:
    $\phi(a)^{\top} \phi(b) = (a^{\top} b)^2$
    Key insight:

    - You can **just compute $(a^\top b)^2$ — without ever computing $\phi(x)$ directly! (Thus saving a lot of computation**.)
- Common Kernels (Equation 5-7):
    - **Linear kernel**: $K(a, b) = a^\top b$
    - **Polynomial kernel**: $K(a, b) = (\gamma a^\top b + r)^d$
    - **Gaussian RBF kernel**: $K(a, b) = \exp\left(-\gamma \|a - b\|^2\right)$
    - **Sigmoid kernel**: $K(a, b) = \tanh(\gamma a^\top b + r)$
- Mercer's Theorem:
    - If $K(a, b)$ satisfies some properties (continuous, symmetric, positive semi-definite), then there exists a mapping ϕ such that: $K(a, b) = \phi(a)^\top \phi(b)$
      This guarantees that you can safely use kernels **without needing to know** ϕ\phiϕ!
    - **Example**:
      For Gaussian RBF, $\phi(x)$ is infinite-dimensional — but we never actually compute it.
    - **Note**:
      Some kernels (like sigmoid) **violate Mercer's conditions**, but **work well in practice**.
- Predictions with Kernelized SVMs:
    - When you train with kernels, you **can't compute** $\hat{w}$ explicitly anymore (because $\phi(x)$ may be huge/infinite).
      Instead, you use only dot products to make predictions!
    - Equ(decision function):

$$f(x) = \sum_{i=1}^{m} \hat{\alpha}(i)\, t(i)\, K(x(i), x) + \hat{b}$$

      Only involves **kernel evaluations**, no explicit transformation.
- Final Note:
    - For very large-scale nonlinear problems, **Random Forests** or **Neural Networks** can sometimes be better choices than SVMs.

Resources :

-

Related notes :

-

References :

- ## Internal :
  -
  -
  -
- ## External :
  - hegab videos
  - the book
  - the notebook
  -

---

1. **Without scaling** (left plot in Figure 5-2):

   - Suppose one feature (like `x₀`) is between **0 and 6**, but another feature (`x₁`) is between **0 and 80**.
   - The SVM looks for the widest street *based on these raw values*.
   - Because `x₁` is so much bigger than `x₀`, the SVM **thinks** `x₁` **is more important**, and the decision boundary becomes almost horizontal — **biased**.

   With scaling (right plot in Figure 5-2):

   - After scaling, both features have similar ranges (like -2 to +2).
   - Now, the SVM treats `x₀` and `x₁` **equally**.
   - As a result, the **street is more natural**, better separating the points.

   In short:
   ⚡ **Scaling fixes the balance between features** so SVM can find the *true* best boundary without being tricked by big numbers. ↩
2. Out-of-core learning means **training a model without loading all the data into RAM at once**.

   🔵 Instead, it **loads small chunks of the data little by little**, processes them, and updates the model gradually.
   ➔ This way, you can **train on very large datasets** that **don't fit into memory**.

Simple example:

- Your data is 500 GB.
- Your RAM is only 8 GB.
- ➔ Out-of-core learning reads, say, 100 MB at a time, trains on it, then moves to the next 100 MB.

↩