Support Vector Machine from Scratch

Bhanu Prasanna Koppolu

Table of contents

Support Vector Machine (SVM) . . . 3
- Core Idea . . . 3
- Why Maximum Margin? . . . 3
- Support Vectors . . . 3
- Classifier . . . 3

Hard Margin SVM . . . 4
- Primal Optimization . . . 4
- Dual Formulation . . . 4
- Problem with Hard Margin . . . 4

Soft Margin SVM . . . 5
- Primal Optimization . . . 5
- Dual Formulation . . . 5
- Decision Function . . . 5

Kernel Trick . . . 6

Understanding C . . . 6

Support Vector Regression (SVR) . . . 6
- Primal . . . 7
- Dual . . . 7
- Predictor . . . 7

Linear SVC Implementation (Sub-Gradient Descent) . . . 7

Linear SVR Implementation (Sub-Gradient Descent) . . . 8

> **Note**
>
> Full implementation available at GitHub - ML from Scratch

Support Vector Machine (SVM)

You thought SGD is Stochastic Gradient Descent? No, it is **Sub Gradient Descent**.

Core Idea

The main idea of SVM is to find an optimal hyperplane that separates classes with the **maximum margin**.

The margin is nothing but the distance from the hyperplane where data points of both classes must be at a wider distance.

SVM Goal: Maximizes the margin distance to nearest points from each class.

Margin: Distance between hyperplane and nearest points.

Why Maximum Margin?

- Robustness to noise
- Proven to reduce overfitting
- Generalizes better to unseen data

In 2D it is a line, 3D it is a plane, n-D it is:

$$\mathbf{w}^T\mathbf{x} + b = 0$$

Support Vectors

The data points closest to the hyperplane (that define the margin).

Classifier

$$f(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x} + b)$$

Where the sign is the Signum function: $f(x) < 0 \Rightarrow -1$, $f(x) = 0 \Rightarrow 0$, $f(x) > 0 \Rightarrow +1$

Hard Margin SVM

The hard margin concept is for linear SVM where we can't allow any misclassifications. The decision boundary must perfectly divide all data points.

The margin is:

$$\gamma = \frac{1}{\|\mathbf{w}\|}$$

Primal Optimization

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

subject to:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \quad \forall i$$

This is a convex quadratic optimization with linear constraints.

Dual Formulation

Using Lagrangian multipliers:

$$\max_{\alpha} W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i\alpha_j y_i y_j \mathbf{x}_i^T\mathbf{x}_j$$

subject to:

$$\alpha_i \geq 0 \quad \forall i$$

$$\sum_{i=1}^{m} \alpha_i y_i = 0$$

The α_i is a Lagrange multiplier for each constraint $g_i(\mathbf{w}, b) \leq 0$ where:

$$g_i(\mathbf{w}, b) = 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b) \leq 0$$

Problem with Hard Margin

The inequality constraints demand every single training data point must be classified correctly and must lie outside the margin or exactly on its boundary. There is no tolerance for misclassification or points falling within the margin.

Soft Margin SVM

We allow some tolerance to the classes to be misclassified. We introduce slack variables ξ_i and regularization parameter C.

Primal Optimization

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

subject to:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i$$
$$\xi_i \geq 0$$

Dual Formulation

$$\mathbf{w} = \sum_{i=1}^{m}\alpha_i y_i \mathbf{x}_i$$

$$\max_{\alpha} W(\alpha) = \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \mathbf{x}_i^T\mathbf{x}_j$$

subject to:

$$0 \leq \alpha_i \leq C$$
$$\sum_{i=1}^{m}\alpha_i y_i = 0$$

The support vectors are those with $\alpha_i > 0$. The misclassified or margin-violating ones often have $\alpha_i = C$.

Decision Function

$$f(\mathbf{x}) = \sum_{i=1}^{m}\alpha_i y_i \mathbf{x}_i^T\mathbf{x} + b$$

Kernel Trick

The kernel trick maps our inner products to higher dimensions to divide the data better:

$$\mathbf{x}_i^T \mathbf{x}_j \Rightarrow K(\mathbf{x}_i, \mathbf{x}_j)$$

The dual becomes:

$$W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$$

The classifier:

$$f(\mathbf{x}) = \text{sign} \left(\sum_{i=1}^{m} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + b \right)$$

Understanding C

C is a regularization hyperparameter that controls the trade-off between:

1. Margin width ($\|\mathbf{w}\|^2$)
2. Training error (slack penalty via $\sum \xi_i$)

Small C (e.g., 0.01)	Large C (e.g., 1000)
Wider margin	Tighter margin
More regularization	Less regularization
Better generalization	Higher training accuracy
More violations allowed	Risk of overfitting
Lower training accuracy	Sensitive to outliers
Use for noisy data, many outliers	Use for clean data, confident labels

Support Vector Regression (SVR)

The concept for SVR is the same, but we use 2 slack variables and a new loss function: **epsilon-insensitive loss** which acts as a tube.

Primal

$$\min_{\mathbf{w},b,\xi,\xi^*} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m}(\xi_i + \xi_i^*)$$

subject to:

$$y_i - (\mathbf{w}^T\mathbf{x}_i + b) \leq \epsilon + \xi_i$$
$$(\mathbf{w}^T\mathbf{x}_i + b) - y_i \leq \epsilon + \xi_i^*$$
$$\xi_i, \xi_i^* \geq 0$$

Dual

$$\max_{\alpha,\alpha^*} -\frac{1}{2}\sum_{i,j}(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)K(\mathbf{x}_i, \mathbf{x}_j) - \epsilon\sum_i(\alpha_i + \alpha_i^*) + \sum_i y_i(\alpha_i - \alpha_i^*)$$

subject to:

$$\sum_{i=1}^{m}(\alpha_i - \alpha_i^*) = 0$$
$$0 \leq \alpha_i, \alpha_i^* \leq C$$

Predictor

$$f(\mathbf{x}) = \sum_{i=1}^{m}(\alpha_i - \alpha_i^*)K(\mathbf{x}_i, \mathbf{x}) + b$$

Linear SVC Implementation (Sub-Gradient Descent)

```
class linear_svc:
    def __init__(self, C=0.01, learning_rate=0.01):
        self.C = C                    # Regularization parameter
        self.learning_rate = learning_rate
        self.loss_history = []

    def fit(self, X_train, y_train, iterations=10000, threshold=1e-6):
        """Train SVC using sub-gradient descent on hinge loss."""
        # Convert labels to -1, +1
        y_train[y_train == 0] = -1
        y_train[y_train == 1] = +1
```

```
        w = np.random.randn(X_train.shape[1])
        b = 0

        for epoch in range(iterations):
            for i in range(X_train.shape[0]):
                # Compute margin: y_i * (w.x_i + b)
                m_i = y_train[i] * (np.dot(w, X_train[i]) + b)

                if m_i >= 1:  # Correct side, no hinge loss
                    gradient_w = w
                    gradient_b = 0
                else:  # Margin violation
                    gradient_w = w - (self.C * X_train[i] * y_train[i])
                    gradient_b = -self.C * y_train[i]

                w = w - self.learning_rate * gradient_w
                b = b - self.learning_rate * gradient_b

        self.w, self.b = w, b
        return w, b

    def predict(self, X_test):
        """Predict class labels (-1 or +1)."""
        return np.sign(np.dot(X_test, self.w) + self.b)
```

Linear SVR Implementation (Sub-Gradient Descent)

```
class linear_svr:
    def __init__(self, C=0.1, epsilon=10.0, learning_rate=0.01):
        self.C = C                    # Regularization parameter
        self.epsilon = epsilon        # Epsilon-tube width
        self.learning_rate = learning_rate

    def fit(self, X_train, y_train, iterations=1000):
        """Train SVR using epsilon-insensitive loss."""
        w = np.zeros(X_train.shape[1])
        b = 0

        for epoch in range(iterations):
```

```
            for i in range(X_train.shape[0]):
                f_i = np.dot(w, X_train[i]) + b
                residual = y_train[i] - f_i

                if np.abs(residual) <= self.epsilon:
                    # Inside tube: only regularization
                    dw, db = w, 0
                elif residual > self.epsilon:
                    # Prediction too low
                    dw = w - self.C * X_train[i]
                    db = -self.C
                else:
                    # Prediction too high
                    dw = w + self.C * X_train[i]
                    db = self.C

                w = w - self.learning_rate * dw
                b = b - self.learning_rate * db

        self.w, self.b = w, b
        return w, b

    def predict(self, X_test):
        """Predict continuous values."""
        return np.dot(X_test, self.w) + self.b
```