ML Foundations - Advanced Handout

Machine Learning for Smarter Innovation

1 ML Foundations - Advanced Handout

Target Audience: Data scientists and ML engineers **Duration**: 90 minutes reading **Level**: Advanced (mathematical foundations)

1.1 Mathematical Foundations

1.1.1 Statistical Learning Theory Framework

Machine learning problems can be formalized within the statistical learning theory framework. We assume data pairs (x, y) are drawn independently from an unknown distribution $P(x, y)$ over $\mathcal{X} \times \mathcal{Y}$. The goal is to find a function $f : \mathcal{X} \to \mathcal{Y}$ from a hypothesis class $\mathcal{H}$ that minimizes expected loss.

Definition (True Risk): The true risk (expected loss) of a function f under loss function L is:

$$R(f) = \mathbb{E}_{(x,y)\sim P}[L(f(x), y)] = \int_{\mathcal{X}\times\mathcal{Y}} L(f(x), y)\, dP(x, y)$$

Since P is unknown, we cannot compute $R(f)$ directly. Instead, we minimize the empirical risk computed from training data.

Definition (Empirical Risk): Given training data $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ drawn i.i.d. from P:

$$\hat{R}(f) = \frac{1}{n} \sum_{i=1}^{n} L(f(x_i), y_i)$$

The Empirical Risk Minimization (ERM) principle selects:

$$\hat{f} =_{f\in\mathcal{H}} \hat{R}(f)$$

The central question of statistical learning theory is: when does minimizing empirical risk lead to low true risk?

1.1.2 Bias-Variance Decomposition

For regression with squared loss $L(f(x), y) = (f(x) - y)^2$, the expected prediction error at a point x decomposes into three components.

Theorem (Bias-Variance Decomposition): Let $\hat{f}$ be an estimator trained on data D drawn from P. For a fixed test point x:

$$\mathbb{E}_D[(y - \hat{f}(x))^2] = \underbrace{(\mathbb{E}_D[\hat{f}(x)] - f^*(x))^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}_D[(\hat{f}(x) - \mathbb{E}_D[\hat{f}(x)])^2]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible}}$$

where $f^*(x) = \mathbb{E}[y|x]$ is the true regression function and $\sigma^2 = \mathbb{E}[(y - f^*(x))^2|x]$ is the noise variance.

Proof: Let $\bar{f}(x) = \mathbb{E}_D[\hat{f}(x)]$. Starting from the squared error:

$$\mathbb{E}_D[(y - \hat{f}(x))^2] = \mathbb{E}_D[(y - f^*(x) + f^*(x) - \bar{f}(x) + \bar{f}(x) - \hat{f}(x))^2]$$

Expanding and using the fact that cross-terms vanish due to independence:

$$= \mathbb{E}[(y - f^*(x))^2] + (f^*(x) - \bar{f}(x))^2 + \mathbb{E}_D[(\bar{f}(x) - \hat{f}(x))^2]$$

$$= \sigma^2 + \text{Bias}^2 + \text{Variance}$$

Implications: - Simple models (few parameters): High bias, low variance - systematic errors but stable - Complex models (many parameters): Low bias, high variance - captures signal but also noise - Optimal model complexity balances these competing effects

1.1.3 VC Dimension and Capacity Control

The Vapnik-Chervonenkis dimension quantifies the expressive power of a hypothesis class, enabling generalization bounds that do not depend on the specific distribution P.

Definition (Shattering): A hypothesis class $\mathcal{H}$ shatters a set of points $S = \{x_1, \ldots, x_m\}$ if for every possible binary labeling $(y_1, \ldots, y_m) \in \{0, 1\}^m$, there exists $h \in \mathcal{H}$ such that $h(x_i) = y_i$ for all i.

Definition (VC Dimension): The VC dimension $d_{VC}(\mathcal{H})$ is the maximum size of a set that $\mathcal{H}$ can shatter:

$$d_{VC}(\mathcal{H}) = \max\{m : \exists S \subset \mathcal{X}, |S| = m, \mathcal{H} \text{ shatters } S\}$$

Examples: - Linear classifiers in $\mathbb{R}^d$: $d_{VC} = d + 1$ - Axis-aligned rectangles in $\mathbb{R}^2$: $d_{VC} = 4$ - Neural network with W weights: $d_{VC} = O(W \log W)$

Theorem (VC Generalization Bound): For any hypothesis class $\mathcal{H}$ with VC dimension d, with probability at least $1 - \delta$:

$$R(h) \leq \hat{R}(h) + \sqrt{\frac{d(\ln(2n/d) + 1) - \ln(\delta/4)}{n}}$$

This bound shows generalization depends on: - Sample size n: More data tightens the bound - VC dimension d: Simpler models generalize better - Confidence level δ: Higher confidence requires looser bound

---

1.2 PAC Learning Framework

1.2.1 Probably Approximately Correct Learning

The PAC framework formalizes what it means for a concept to be learnable from examples.

Definition (PAC-Learnable): A concept class $\mathcal{C}$ is PAC-learnable by hypothesis class $\mathcal{H}$ if there exists an algorithm A and a polynomial $p(\cdot, \cdot, \cdot)$ such that for any: - Target concept $c \in \mathcal{C}$ - Distribution P over $\mathcal{X}$ - Accuracy parameter $\epsilon > 0$ - Confidence parameter $\delta > 0$

When run on at least $m \geq p(1/\epsilon, 1/\delta, \text{size}(c))$ examples drawn from P, algorithm A outputs hypothesis h with:

$$P_{D \sim P^m}[R(h) \leq \epsilon] \geq 1 - \delta$$

1.2.2 Sample Complexity Bounds

Theorem (Finite Hypothesis Class): For a finite hypothesis class $\mathcal{H}$, the sample complexity for PAC learning is:

$$m(\epsilon, \delta) \geq \frac{1}{\epsilon}\left(\ln|\mathcal{H}| + \ln\frac{1}{\delta}\right)$$

Proof: Using Hoeffding's inequality and union bound. For any fixed $h \in \mathcal{H}$:

$$P(|\hat{R}(h) - R(h)| > \epsilon/2) \leq 2\exp(-2m(\epsilon/2)^2) = 2\exp(-m\epsilon^2/2)$$

By union bound over all $h \in \mathcal{H}$:

$$P(\exists h : |\hat{R}(h) - R(h)| > \epsilon/2) \leq 2|\mathcal{H}|\exp(-m\epsilon^2/2)$$

Setting this equal to δ and solving for m yields the bound.

Theorem (Infinite Class - VC): For hypothesis class with VC dimension d:

$$m(\epsilon, \delta) = O\left(\frac{d + \ln(1/\delta)}{\epsilon^2}\right)$$

1.2.3 Agnostic PAC Learning

The agnostic setting removes the assumption that the target is in the hypothesis class.

Definition (Agnostic PAC-Learnable): Class $\mathcal{H}$ is agnostic PAC-learnable if for any distribution P over $\mathcal{X} \times \mathcal{Y}$, with high probability the algorithm finds h satisfying:

$$R(h) \leq \min_{h' \in \mathcal{H}} R(h') + \epsilon$$

This is the more practical setting - we cannot expect to find the true concept, only approximate the best hypothesis in our class.

1.3 Optimization Theory

1.3.1 Convex Optimization Fundamentals

Definition (Convex Function): $f : \mathbb{R}^d \to \mathbb{R}$ is convex if for all x, y and $\lambda \in [0, 1]$:

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

Definition (Strong Convexity): f is μ-strongly convex if for all x, y:

$$f(y) \geq f(x) + \nabla f(x)^T(y - x) + \frac{\mu}{2}\|y - x\|^2$$

Definition (Lipschitz Continuity): f has L-Lipschitz gradients if:

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$$

1.3.2 Gradient Descent Convergence

Theorem (GD Convergence - Convex): For convex f with L-Lipschitz gradients, gradient descent with step size $\eta = 1/L$ satisfies:

$$f(w_T) - f(w^*) \leq \frac{L\|w_0 - w^*\|^2}{2T}$$

Theorem (GD Convergence - Strongly Convex): For μ-strongly convex f with L-Lipschitz gradients, with step size $\eta = 1/L$:

$$\|w_T - w^*\|^2 \leq \left(1 - \frac{\mu}{L}\right)^T \|w_0 - w^*\|^2$$

The condition number $\kappa = L/\mu$ determines convergence rate. Ill-conditioned problems ($\kappa \gg 1$) converge slowly.

1.3.3 Stochastic Gradient Descent Analysis

SGD replaces the full gradient with an unbiased estimate:

$$w_{t+1} = w_t - \eta_t g_t, \quad \mathbb{E}[g_t | w_t] = \nabla f(w_t)$$

Theorem (SGD Convergence): For convex f with bounded stochastic gradients $\mathbb{E}[\|g_t\|^2] \leq G^2$, using step size $\eta_t = \frac{1}{\sqrt{t}}$:

$$\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} f(w_t)\right] - f(w^*) \leq \frac{\|w_0 - w^*\|^2 + G^2 \sum_{t=1}^{T} \eta_t^2}{2\sum_{t=1}^{T} \eta_t}$$

With $\eta_t = 1/\sqrt{t}$, this gives $O(1/\sqrt{T})$ convergence.

1.3.4 Momentum and Acceleration

Momentum SGD:

$$v_{t+1} = \beta v_t + g_t$$
$$w_{t+1} = w_t - \eta v_{t+1}$$

Nesterov Accelerated Gradient: For μ-strongly convex, L-smooth functions:

$$v_{t+1} = w_t - \frac{1}{L}\nabla f(w_t)$$
$$w_{t+1} = \left(1 + \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right) v_{t+1} - \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} v_t$$

Achieves $O((1 - 1/\sqrt{\kappa})^t)$ convergence, improving over GD's $O((1 - 1/\kappa)^t)$.

1.4 Regularization Theory

1.4.1 Structural Risk Minimization

Regularization controls model complexity to prevent overfitting. The regularized objective is:

$$\hat{f}_\lambda = _{f \in \mathcal{H}} \hat{R}(f) + \lambda \Omega(f)$$

where $\Omega(f)$ is a complexity penalty and $\lambda > 0$ controls the tradeoff.

1.4.2 Tikhonov Regularization (Ridge)

For linear models $f(x) = w^T x$:

$$\min_w \|Xw - y\|_2^2 + \lambda \|w\|_2^2$$

Closed-form solution:

$$w^* = (X^T X + \lambda I)^{-1} X^T y$$

Derivation: Setting gradient to zero:

$$\nabla_w L = 2X^T(Xw - y) + 2\lambda w = 0$$
$$X^T X w + \lambda w = X^T y$$
$$(X^T X + \lambda I) w = X^T y$$

Bayesian interpretation: Ridge regression is equivalent to MAP estimation with Gaussian prior:

$$p(w) = \mathcal{N}(0, \tau^2 I), \quad \lambda = \sigma^2 / \tau^2$$

Singular Value Analysis: If $X = U \Sigma V^T$ (SVD), then:

$$w^* = V(\Sigma^2 + \lambda I)^{-1} \Sigma U^T y = \sum_{j=1}^{p} \frac{\sigma_j}{\sigma_j^2 + \lambda} u_j^T y \cdot v_j$$

Ridge shrinks coefficients proportionally more for directions with small singular values.

1.4.3 Lasso Regularization (L1)

$$\min_w \|Xw - y\|_2^2 + \lambda \|w\|_1$$

Properties: - Produces sparse solutions ($w_i = 0$ for many i) - Performs automatic feature selection - No closed-form solution (subdifferential involved)

Subdifferential Characterization: Optimal w satisfies:

$$X^T(Xw - y) + \lambda \partial \|w\|_1 \ni 0$$

where $\partial \|w\|_1$ is the subdifferential:

$$\partial |w_i| = \begin{cases} \{1\} & w_i > 0 \\ [-1, 1] & w_i = 0 \\ \{-1\} & w_i < 0 \end{cases}$$

Coordinate Descent Update:

$$w_j \leftarrow S_{\lambda/L} \left(w_j - \frac{1}{L} [\nabla_j f(w)] \right)$$

where $S_\tau(z) = \text{sign}(z) \max(|z| - \tau, 0)$ is the soft-thresholding operator.

1.4.4 Elastic Net

Combines L1 and L2 penalties:

$$\min_w \|Xw - y\|_2^2 + \lambda_1\|w\|_1 + \lambda_2\|w\|_2^2$$

Advantages: - Handles correlated features better than Lasso alone - Retains grouping effect (selects correlated features together) - Convex, unique solution

---

1.5 Kernel Methods and Reproducing Kernel Hilbert Spaces

1.5.1 The Kernel Trick

Many algorithms depend on data only through inner products $\langle x_i, x_j\rangle$. The kernel trick replaces these with a kernel function:

$$K(x, x') = \langle\phi(x), \phi(x')\rangle_{\mathcal{H}}$$

where $\phi : \mathcal{X} \to \mathcal{H}$ maps inputs to a (possibly infinite-dimensional) feature space.

Definition (Positive Definite Kernel): $K : \mathcal{X}\times\mathcal{X} \to \mathbb{R}$ is positive definite if for any n and $x_1, \ldots, x_n \in \mathcal{X}$, the Gram matrix $K_{ij} = K(x_i, x_j)$ is positive semidefinite.

Common Kernels:

Kernel	Formula	Feature Dimension
Linear	$K(x, x') = x^T x'$	d
Polynomial	$K(x, x') = (x^T x' + c)^p$	$\binom{d+p}{p}$
RBF/Gaussian	$K(x, x') = \exp(-\gamma\|x - x'\|^2)$	∞
Laplacian	$K(x, x') = \exp(-\gamma\|x - x'\|_1)$	∞

1.5.2 Representer Theorem

Theorem: For regularized risk minimization in an RKHS $\mathcal{H}$:

$$\min_{f\in\mathcal{H}} \frac{1}{n}\sum_{i=1}^{n} L(f(x_i), y_i) + \lambda\|f\|_{\mathcal{H}}^2$$

The optimal solution has the form:

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

Proof sketch: Any $f \in \mathcal{H}$ can be decomposed as $f = f_\parallel + f_\perp$ where $f_\parallel$ is in the span of $\{K(x_i, \cdot)\}$ and $f_\perp$ is orthogonal. Since $f_\perp(x_i) = 0$ (by reproducing property), only $f_\parallel$ affects the loss, while $\|f\|^2 = \|f_\parallel\|^2 + \|f_\perp\|^2$. Thus $f_\perp = 0$ at optimum.

1.5.3 Kernel Ridge Regression

Substituting the representer form:

$$\alpha^* = (K + \lambda I)^{-1} y$$

where K is the $n \times n$ Gram matrix.

Prediction:

$$f(x) = k(x)^T (K + \lambda I)^{-1} y$$

where $k(x) = [K(x_1, x), \ldots, K(x_n, x)]^T$.

1.6 Information Theory in Machine Learning

1.6.1 Cross-Entropy and Log Loss

For classification with predicted probabilities $q(y|x)$ and true distribution $p(y|x)$:

$$H(p, q) = -\sum_y p(y|x) \log q(y|x) = -\mathbb{E}_p[\log q]$$

Relation to KL Divergence:

$$H(p, q) = H(p) + D_{KL}(p\|q)$$

Minimizing cross-entropy is equivalent to minimizing KL divergence (since $H(p)$ is constant).

1.6.2 KL Divergence Properties

Definition:

$$D_{KL}(P\|Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} = \mathbb{E}_P \left[\log \frac{P}{Q}\right]$$

Properties: - **Non-negativity**: $D_{KL}(P\|Q) \geq 0$ with equality iff $P = Q$ (Gibbs' inequality) - **Asymmetry**: $D_{KL}(P\|Q) \neq D_{KL}(Q\|P)$ in general - **Chain rule**: $D_{KL}(P(X, Y)\|Q(X, Y)) = D_{KL}(P(X)\|Q(X)) + \mathbb{E}_{P(X)}[D_{KL}(P(Y|X)\|Q(Y|X))]$

1.6.3 Mutual Information

$$I(X; Y) = D_{KL}(P(X, Y)\|P(X)P(Y)) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$

Applications in ML: - **Feature selection**: Select features maximizing $I(X_i; Y)$ - **Information bottleneck**: Find representation Z maximizing $I(Z; Y)$ while minimizing $I(Z; X)$ - **Neural network analysis**: Track $I(X; T)$ and $I(T; Y)$ through layers

---

1.7 Concentration Inequalities

1.7.1 Hoeffding's Inequality

Theorem: Let $X_1, \ldots, X_n$ be independent bounded random variables with $a_i \leq X_i \leq b_i$. Then:

$$P\left(\left|\frac{1}{n}\sum_{i=1}^n X_i - \mathbb{E}\left[\frac{1}{n}\sum_{i=1}^n X_i\right]\right| \geq t\right) \leq 2\exp\left(-\frac{2n^2t^2}{\sum_{i=1}^n (b_i - a_i)^2}\right)$$

For $X_i \in [0, 1]$:

$$P(|\hat{\mu} - \mu| \geq t) \leq 2\exp(-2nt^2)$$

1.7.2 McDiarmid's Inequality

Theorem: Let $X_1, \ldots, X_n$ be independent and f satisfy the bounded differences condition:

$$\sup_{x_1, \ldots, x_n, x_i'} |f(x_1, \ldots, x_i, \ldots, x_n) - f(x_1, \ldots, x_i', \ldots, x_n)| \leq c_i$$

Then:

$$P(|f(X_1, \ldots, X_n) - \mathbb{E}[f]| \geq t) \leq 2\exp\left(-\frac{2t^2}{\sum_{i=1}^n c_i^2}\right)$$

Application: Generalization bounds for functions of data.

1.7.3 Chernoff Bounds

For sum of independent Bernoulli variables $X = \sum_{i=1}^{n} X_i$ with $\mathbb{E}[X] = \mu$:

Upper tail: $P(X \geq (1+\delta)\mu) \leq \exp\left(-\frac{\delta^2\mu}{2+\delta}\right)$

Lower tail: $P(X \leq (1-\delta)\mu) \leq \exp\left(-\frac{\delta^2\mu}{2}\right)$

1.8 Production Implementation

1.8.1 Scalable Training Pipeline

```
import numpy as np
from scipy import sparse
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.utils.validation import check_X_y, check_array, check_is_fitted
from sklearn.preprocessing import StandardScaler
from typing import Optional, Callable, Tuple, List
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


class RegularizedLinearModel(BaseEstimator):
    """
    Production-grade regularized linear model with multiple optimization
    algorithms.

    Supports L1, L2, and Elastic Net regularization with convergence
    guarantees.
    """

    def __init__(
        self,
        penalty: str = 'l2',
        alpha: float = 1.0,
        l1_ratio: float = 0.5,
        fit_intercept: bool = True,
        max_iter: int = 1000,
        tol: float = 1e-6,
        optimizer: str = 'sgd',
        learning_rate: float = 0.01,
        batch_size: int = 32,
        momentum: float = 0.9,
        random_state: Optional[int] = None,
        verbose: bool = False
    ):
        """
        Initialize regularized linear model.

        Parameters
        ----------
        penalty : str, {'l1', 'l2', 'elasticnet'}
            Type of regularization
        alpha : float
            Regularization strength (lambda)
        l1_ratio : float
            For elasticnet, ratio of L1 penalty (0=ridge, 1=lasso)
```

```
        fit_intercept : bool
            Whether to fit intercept term
        max_iter : int
            Maximum number of iterations
        tol : float
            Convergence tolerance
        optimizer : str, {'gd', 'sgd', 'adam'}
            Optimization algorithm
        learning_rate : float
            Initial learning rate
        batch_size : int
            Batch size for SGD/Adam
        momentum : float
            Momentum coefficient
        random_state : int, optional
            Random seed for reproducibility
        verbose : bool
            Whether to print progress
        """
        self.penalty = penalty
        self.alpha = alpha
        self.l1_ratio = l1_ratio
        self.fit_intercept = fit_intercept
        self.max_iter = max_iter
        self.tol = tol
        self.optimizer = optimizer
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.momentum = momentum
        self.random_state = random_state
        self.verbose = verbose

    def _add_intercept(self, X: np.ndarray) -> np.ndarray:
        """Add intercept column to feature matrix."""
        if self.fit_intercept:
            intercept = np.ones((X.shape[0], 1))
            return np.hstack([intercept, X])
        return X

    def _compute_loss(self, X: np.ndarray, y: np.ndarray, w: np.ndarray) ->
    float:
        """Compute regularized loss."""
        n = X.shape[0]
        residuals = X @ w - y
        data_loss = np.sum(residuals ** 2) / (2 * n)

        # Skip intercept in regularization
        w_reg = w[1:] if self.fit_intercept else w

        if self.penalty == 'l2':
            reg_loss = self.alpha * np.sum(w_reg ** 2) / 2
        elif self.penalty == 'l1':
            reg_loss = self.alpha * np.sum(np.abs(w_reg))
        elif self.penalty == 'elasticnet':
            reg_loss = self.alpha * (
                self.l1_ratio * np.sum(np.abs(w_reg)) +
                (1 - self.l1_ratio) * np.sum(w_reg ** 2) / 2
            )
        else:
            reg_loss = 0

        return data_loss + reg_loss
```

```
    def _compute_gradient(self, X: np.ndarray, y: np.ndarray, w: np.ndarray)
    -> np.ndarray:
        """Compute gradient of regularized loss."""
        n = X.shape[0]
        residuals = X @ w - y
        grad = X.T @ residuals / n

        # Add regularization gradient (skip intercept)
        if self.penalty == 'l2':
            grad_reg = self.alpha * w.copy()
            if self.fit_intercept:
                grad_reg[0] = 0
            grad += grad_reg
        elif self.penalty == 'elasticnet':
            grad_reg = self.alpha * (1 - self.l1_ratio) * w.copy()
            if self.fit_intercept:
                grad_reg[0] = 0
            grad += grad_reg

        return grad

    def _proximal_l1(self, w: np.ndarray, step: float) -> np.ndarray:
        """Proximal operator for L1 penalty (soft thresholding)."""
        threshold = self.alpha * step
        if self.penalty == 'elasticnet':
            threshold *= self.l1_ratio

        w_prox = w.copy()
        if self.fit_intercept:
            # Don't threshold intercept
            w_prox[1:] = np.sign(w[1:]) * np.maximum(np.abs(w[1:]) - threshold
    , 0)
        else:
            w_prox = np.sign(w) * np.maximum(np.abs(w) - threshold, 0)

        return w_prox

    def _fit_gd(self, X: np.ndarray, y: np.ndarray) -> np.ndarray:
        """Fit using gradient descent."""
        n, d = X.shape
        w = np.zeros(d)

        history = []

        for iteration in range(self.max_iter):
            grad = self._compute_gradient(X, y, w)
            w_new = w - self.learning_rate * grad

            # Apply proximal operator for L1
            if self.penalty in ['l1', 'elasticnet']:
                w_new = self._proximal_l1(w_new, self.learning_rate)

            loss = self._compute_loss(X, y, w_new)
            history.append(loss)

            # Check convergence
            if np.linalg.norm(w_new - w) < self.tol:
                if self.verbose:
                    logger.info(f"Converged at iteration {iteration}")
                break

            w = w_new
```

```
            if self.verbose and iteration % 100 == 0:
                logger.info(f"Iteration {iteration}: loss = {loss:.6f}")

        self.loss_history_ = history
        return w

    def _fit_sgd(self, X: np.ndarray, y: np.ndarray) -> np.ndarray:
        """Fit using stochastic gradient descent with momentum."""
        rng = np.random.RandomState(self.random_state)
        n, d = X.shape
        w = np.zeros(d)
        v = np.zeros(d)  # Momentum velocity

        history = []

        for iteration in range(self.max_iter):
            # Sample mini-batch
            idx = rng.choice(n, size=min(self.batch_size, n), replace=False)
            X_batch = X[idx]
            y_batch = y[idx]

            grad = self._compute_gradient(X_batch, y_batch, w)

            # Momentum update
            v = self.momentum * v + grad
            w_new = w - self.learning_rate * v

            # Apply proximal operator for L1
            if self.penalty in ['l1', 'elasticnet']:
                w_new = self._proximal_l1(w_new, self.learning_rate)

            # Compute full loss periodically
            if iteration % 10 == 0:
                loss = self._compute_loss(X, y, w_new)
                history.append(loss)

                if len(history) > 1 and abs(history[-1] - history[-2]) < self.
tol:
                    if self.verbose:
                        logger.info(f"Converged at iteration {iteration}")
                    break

            w = w_new

            if self.verbose and iteration % 100 == 0:
                loss = self._compute_loss(X, y, w)
                logger.info(f"Iteration {iteration}: loss = {loss:.6f}")

        self.loss_history_ = history
        return w

    def _fit_adam(self, X: np.ndarray, y: np.ndarray) -> np.ndarray:
        """Fit using Adam optimizer."""
        rng = np.random.RandomState(self.random_state)
        n, d = X.shape
        w = np.zeros(d)

        # Adam parameters
        beta1, beta2 = 0.9, 0.999
        eps = 1e-8
        m = np.zeros(d)  # First moment
        v = np.zeros(d)  # Second moment
```

```
        history = []

        for iteration in range(1, self.max_iter + 1):
            # Sample mini-batch
            idx = rng.choice(n, size=min(self.batch_size, n), replace=False)
            X_batch = X[idx]
            y_batch = y[idx]

            grad = self._compute_gradient(X_batch, y_batch, w)

            # Update biased moment estimates
            m = beta1 * m + (1 - beta1) * grad
            v = beta2 * v + (1 - beta2) * grad ** 2

            # Bias-corrected estimates
            m_hat = m / (1 - beta1 ** iteration)
            v_hat = v / (1 - beta2 ** iteration)

            # Adam update
            w_new = w - self.learning_rate * m_hat / (np.sqrt(v_hat) + eps)

            # Apply proximal operator for L1
            if self.penalty in ['l1', 'elasticnet']:
                w_new = self._proximal_l1(w_new, self.learning_rate)

            # Compute full loss periodically
            if iteration % 10 == 0:
                loss = self._compute_loss(X, y, w_new)
                history.append(loss)

                if len(history) > 1 and abs(history[-1] - history[-2]) < self.
    tol:
                    if self.verbose:
                        logger.info(f"Converged at iteration {iteration}")
                    break

            w = w_new

            if self.verbose and iteration % 100 == 0:
                loss = self._compute_loss(X, y, w)
                logger.info(f"Iteration {iteration}: loss = {loss:.6f}")

        self.loss_history_ = history
        return w

    def fit(self, X: np.ndarray, y: np.ndarray):
        """
        Fit the model to training data.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
            Training features
        y : array-like of shape (n_samples,)
            Target values

        Returns
        -------
        self : object
            Fitted estimator
        """
        X, y = check_X_y(X, y)
```

```
        # Scale features for numerical stability
        self.scaler_ = StandardScaler()
        X_scaled = self.scaler_.fit_transform(X)

        # Add intercept column
        X_aug = self._add_intercept(X_scaled)

        # Select optimizer
        if self.optimizer == 'gd':
            self.coef_ = self._fit_gd(X_aug, y)
        elif self.optimizer == 'sgd':
            self.coef_ = self._fit_sgd(X_aug, y)
        elif self.optimizer == 'adam':
            self.coef_ = self._fit_adam(X_aug, y)
        else:
            raise ValueError(f"Unknown optimizer: {self.optimizer}")

        # Extract intercept and coefficients
        if self.fit_intercept:
            self.intercept_ = self.coef_[0]
            self.coef_ = self.coef_[1:]
        else:
            self.intercept_ = 0.0

        self.n_features_in_ = X.shape[1]

        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """
        Predict target values.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
            Samples to predict

        Returns
        -------
        y_pred : array of shape (n_samples,)
            Predicted values
        """
        check_is_fitted(self)
        X = check_array(X)

        X_scaled = self.scaler_.transform(X)
        return X_scaled @ self.coef_ + self.intercept_


class KernelRidgeRegression(BaseEstimator):
    """
    Kernel ridge regression with support for custom kernels.
    """

    def __init__(
        self,
        kernel: str = 'rbf',
        alpha: float = 1.0,
        gamma: float = 1.0,
        degree: int = 3,
        coef0: float = 1.0
    ):
        """
```

```
        Initialize kernel ridge regression.

        Parameters
        ----------
        kernel : str, {'linear', 'polynomial', 'rbf'}
            Kernel type
        alpha : float
            Regularization strength
        gamma : float
            RBF kernel width parameter
        degree : int
            Polynomial kernel degree
        coef0 : float
            Polynomial kernel offset
        """
        self.kernel = kernel
        self.alpha = alpha
        self.gamma = gamma
        self.degree = degree
        self.coef0 = coef0

    def _compute_kernel(self, X1: np.ndarray, X2: np.ndarray) -> np.ndarray:
        """Compute kernel matrix."""
        if self.kernel == 'linear':
            return X1 @ X2.T
        elif self.kernel == 'polynomial':
            return (X1 @ X2.T + self.coef0) ** self.degree
        elif self.kernel == 'rbf':
            # Efficient RBF computation
            X1_sq = np.sum(X1 ** 2, axis=1, keepdims=True)
            X2_sq = np.sum(X2 ** 2, axis=1, keepdims=True)
            dist_sq = X1_sq + X2_sq.T - 2 * X1 @ X2.T
            return np.exp(-self.gamma * dist_sq)
        else:
            raise ValueError(f"Unknown kernel: {self.kernel}")

    def fit(self, X: np.ndarray, y: np.ndarray):
        """Fit kernel ridge regression."""
        X, y = check_X_y(X, y)
        self.X_train_ = X

        # Compute kernel matrix
        K = self._compute_kernel(X, X)

        # Solve (K + alpha*I) @ alpha = y
        n = K.shape[0]
        self.alpha_ = np.linalg.solve(K + self.alpha * np.eye(n), y)

        self.n_features_in_ = X.shape[1]
        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict using kernel ridge regression."""
        check_is_fitted(self)
        X = check_array(X)

        K_test = self._compute_kernel(X, self.X_train_)
        return K_test @ self.alpha_


# Usage demonstration
if __name__ == "__main__":
    from sklearn.datasets import make_regression
```

```
    from sklearn.model_selection import train_test_split
    from sklearn.metrics import mean_squared_error, r2_score

    # Generate synthetic data
    X, y = make_regression(n_samples=1000, n_features=20,
                           n_informative=10, noise=10, random_state=42)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    # Test different configurations
    models = {
        'Ridge (GD)': RegularizedLinearModel(penalty='l2', alpha=1.0,
    optimizer='gd'),
        'Ridge (SGD)': RegularizedLinearModel(penalty='l2', alpha=1.0,
    optimizer='sgd'),
        'Ridge (Adam)': RegularizedLinearModel(penalty='l2', alpha=1.0,
    optimizer='adam'),
        'Lasso': RegularizedLinearModel(penalty='l1', alpha=0.1, optimizer='
    sgd'),
        'ElasticNet': RegularizedLinearModel(penalty='elasticnet', alpha=0.5,
                                             l1_ratio=0.5, optimizer='sgd'),
        'Kernel Ridge (RBF)': KernelRidgeRegression(kernel='rbf', alpha=1.0,
    gamma=0.1)
    }

    print("Model Comparison:")
    print("-" * 50)
    for name, model in models.items():
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        mse = mean_squared_error(y_test, y_pred)
        r2 = r2_score(y_test, y_pred)
        print(f"{name:20s} MSE: {mse:.4f}  R2: {r2:.4f}")
```

1.9 Advanced Topics

1.9.1 Online Learning and Regret Bounds

In online learning, data arrives sequentially and the algorithm must make predictions before seeing the true label.

Regret Definition:

$$\text{Regret}_T = \sum_{t=1}^{T} L(w_t, x_t, y_t) - \min_{w \in \mathcal{W}} \sum_{t=1}^{T} L(w, x_t, y_t)$$

Online Gradient Descent:

$$w_{t+1} = \Pi_{\mathcal{W}} \left(w_t - \eta \nabla_w L(w_t, x_t, y_t) \right)$$

Theorem: For convex, G-Lipschitz losses over domain with diameter D, OGD with $\eta = \frac{D}{G\sqrt{T}}$ achieves:

$$\text{Regret}_T \leq DG\sqrt{T}$$

1.9.2 Multi-Armed Bandits

The explore-exploit tradeoff formalized.

Upper Confidence Bound (UCB):

$$a_t =_a \left(\hat{\mu}_a + \sqrt{\frac{2 \ln t}{n_a}} \right)$$

Regret Bound: UCB achieves $O(\sqrt{KT \ln T})$ regret for K arms.

1.9.3 Boosting Theory

AdaBoost minimizes exponential loss:

$$L(\alpha, h) = \sum_{i=1}^{n} \exp\left(-y_i \sum_{t=1}^{T} \alpha_t h_t(x_i) \right)$$

Margin Theory: The margin of example (x_i, y_i) is:

$$\text{margin}_i = \frac{y_i \sum_t \alpha_t h_t(x_i)}{\sum_t \alpha_t}$$

Boosting increases the minimum margin, explaining generalization even with many weak learners.

1.10 Practice Problems

1. **Derive** the gradient of logistic loss $L(w) = \log(1 + \exp(-y \cdot w^T x))$
2. **Prove** the bias-variance decomposition for a general loss function (extend beyond squared loss)
3. **Show** that SVMs solve the dual problem using KKT conditions and identify the support vectors
4. **Compute** the VC dimension of neural networks with ReLU activations
5. **Derive** the coordinate descent update for Lasso regression
6. **Prove** Hoeffding's inequality using the Chernoff bounding technique
7. **Analyze** the convergence rate of Nesterov accelerated gradient descent

---

1.11 References

1. Shalev-Shwartz, S., & Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.
2. Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning* (2nd ed.). Springer.
3. Vapnik, V. N. (1998). *Statistical Learning Theory*. Wiley.
4. Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
5. Boyd, S., & Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press.
6. Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). *Foundations of Machine Learning* (2nd ed.). MIT Press.
7. Cover, T. M., & Thomas, J. A. (2006). *Elements of Information Theory* (2nd ed.). Wiley.

Statistical learning theory provides the mathematical foundations for understanding when and why machine learning works. These theoretical tools - generalization bounds, optimization guarantees, and complexity measures - guide principled algorithm design and hyperparameter selection.