Supervised Learning - Advanced Handout

Machine Learning for Smarter Innovation

1 Supervised Learning - Advanced Handout

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

1.1 Mathematical Foundations

1.1.1 Problem Formulation

Supervised learning seeks to find a function $f : \mathcal{X} \to \mathcal{Y}$ that maps inputs to outputs based on a training set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$ drawn i.i.d. from an unknown distribution $P(X, Y)$.

Risk Minimization Framework:

The true risk (expected loss) for a hypothesis h is:

$$R(h) = \mathbb{E}_{(X,Y)\sim P}[L(h(X), Y)] = \int L(h(x), y)\, dP(x, y)$$

where $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ is a loss function.

Since P is unknown, we minimize the empirical risk:

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

The Empirical Risk Minimization (ERM) principle selects:

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

Common Loss Functions:

*	*	*
0.2069	0.3851	Formula
		Func-
	0.4828	tion
*	*	*
0.20	0.38	$L(y, \hat{y}) =$
gres-	Er-	$(y -$
sion	or	$\hat{y})^2$

0.2069	0.3331	Loss Formula
	0.4828	Function
0.2069	0.3331	$L(y, \hat{y}) = \lvert y - \hat{y} \rvert$
gression	Absolute Error	
0.2069	0.3331	$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & \lvert y-\hat{y} \rvert \le \delta \\ \delta \lvert y-\hat{y} \rvert - \frac{\delta^2}{2} & \text{otherwise} \end{cases}$
gression	Huber Loss	
0.2069	0.3331	$L(y, \hat{y}) = \mathbf{1}[y \neq \hat{y}]$
sification	0-1 Loss	
0.2069	0.3331	$L(y, p) = -y \log(p) - (1-y) \log(1-p)$
sification	Cross-Entropy Loss	
0.2069	0.3331	$L(y, f) = \max(0, 1 - yf)$
sification	Hinge Loss	

1.1.2 Bias-Variance Decomposition

For squared error loss, the expected prediction error decomposes as:

$$\mathbb{E}[(Y - \hat{f}(X))^2] = \text{Bias}^2[\hat{f}(X)] + \text{Var}[\hat{f}(X)] + \sigma^2$$

Derivation:

Let $f(x) = \mathbb{E}[Y|X = x]$ be the true conditional mean and $\hat{f}$ be our estimator.

$$\mathbb{E}[(Y - \hat{f})^2] = \mathbb{E}[(Y - f + f - \hat{f})^2]$$

Expanding:

$$= \mathbb{E}[(Y - f)^2] + \mathbb{E}[(f - \hat{f})^2] + 2\mathbb{E}[(Y - f)(f - \hat{f})]$$

The cross term vanishes because $\mathbb{E}[Y - f|X] = 0$. For the second term:

$$\mathbb{E}[(f - \hat{f})^2] = \mathbb{E}[(f - \mathbb{E}[\hat{f}] + \mathbb{E}[\hat{f}] - \hat{f})^2]$$

$$= (f - \mathbb{E}[\hat{f}])^2 + \mathbb{E}[(\hat{f} - \mathbb{E}[\hat{f}])^2]$$
$$= \text{Bias}^2[\hat{f}] + \text{Var}[\hat{f}]$$

Implications: - Simple models: high bias, low variance (underfitting) - Complex models: low bias, high variance (overfitting) - Optimal models balance this tradeoff

---

1.2 Linear Methods

1.2.1 Ordinary Least Squares

Model: $y = X\beta + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$

Objective: Minimize $L(\beta) = \|y - X\beta\|_2^2 = (y - X\beta)^T(y - X\beta)$

Derivation of Normal Equations:

Taking the gradient:

$$\nabla_\beta L = -2X^T(y - X\beta) = -2X^T y + 2X^T X\beta$$

Setting to zero:

$$X^T X\beta = X^T y$$

If $X^T X$ is invertible:

$$\hat{\beta}_{OLS} = (X^T X)^{-1} X^T y$$

Properties of OLS Estimator:

1. **Unbiasedness**: $\mathbb{E}[\hat{\beta}] = \beta$
   Proof: $\mathbb{E}[(X^T X)^{-1} X^T y] = (X^T X)^{-1} X^T \mathbb{E}[y] = (X^T X)^{-1} X^T X\beta = \beta$
2. **Variance**: $\text{Var}[\hat{\beta}] = \sigma^2 (X^T X)^{-1}$
3. **Gauss-Markov Theorem**: Under the assumptions of linearity, independence, homoscedasticity, and no perfect multicollinearity, OLS is BLUE (Best Linear Unbiased Estimator).

1.2.2 Ridge Regression (L2 Regularization)

Objective: $L(\beta) = \|y - X\beta\|_2^2 + \lambda\|\beta\|_2^2$

Closed-form Solution:

$$\hat{\beta}_{ridge} = (X^T X + \lambda I)^{-1} X^T y$$

Bayesian Interpretation: Ridge regression corresponds to maximum a posteriori (MAP) estimation with a Gaussian prior $\beta \sim \mathcal{N}(0, \tau^2 I)$ where $\lambda = \sigma^2/\tau^2$.

Effect on Eigenvalues: If $X = UDV^T$ (SVD), then:

$$\hat{\beta}_{ridge} = \sum_{j=1}^{p} \frac{d_j^2}{d_j^2 + \lambda} \frac{u_j^T y}{d_j} v_j$$

Ridge shrinks all coefficients toward zero, with more shrinkage for directions with small singular values.

1.2.3 Lasso Regression (L1 Regularization)

Objective: $L(\beta) = \frac{1}{2n}\|y - X\beta\|_2^2 + \lambda\|\beta\|_1$

No Closed-form Solution: Requires iterative optimization (coordinate descent, proximal gradient).

Coordinate Descent Update:

For coordinate j:

$$\hat{\beta}_j = S_\lambda\left(\frac{1}{n}\sum_{i=1}^{n} x_{ij}(y_i - \sum_{k\neq j} x_{ik}\hat{\beta}_k)\right)$$

where $S_\lambda(z) = \text{sign}(z)(|z| - \lambda)_+$ is the soft-thresholding operator.

Sparsity: Lasso produces sparse solutions (many $\beta_j = 0$), enabling automatic feature selection.

Elastic Net: Combines L1 and L2:

$$L(\beta) = \frac{1}{2n}\|y - X\beta\|_2^2 + \lambda_1\|\beta\|_1 + \lambda_2\|\beta\|_2^2$$

1.2.4 Logistic Regression

Model: For binary classification, the log-odds are linear:

$$\log\frac{P(Y=1|X)}{P(Y=0|X)} = \beta^T X$$

Probability: $P(Y=1|X=x) = \sigma(\beta^T x) = \frac{1}{1+e^{-\beta^T x}}$

Maximum Likelihood Estimation:

Log-likelihood:

$$\ell(\beta) = \sum_{i=1}^{n}[y_i \log\sigma(\beta^T x_i) + (1-y_i)\log(1-\sigma(\beta^T x_i))]$$

Gradient:

$$\nabla_\beta \ell = \sum_{i=1}^{n}(y_i - \sigma(\beta^T x_i))x_i = X^T(y - \hat{p})$$

Hessian:

$$H = -X^T W X$$

where $W = \text{diag}[\hat{p}_i(1-\hat{p}_i)]$.

Newton-Raphson Update:

$$\beta^{(t+1)} = \beta^{(t)} - H^{-1}\nabla_\beta \ell = (X^T W X)^{-1} X^T W z$$

where $z = X\beta + W^{-1}(y - \hat{p})$ (Iteratively Reweighted Least Squares).

1.3 Tree-Based Methods

1.3.1 CART Algorithm

Objective: Recursively partition the feature space to minimize impurity within nodes.

Splitting Criterion: At node m with samples $\mathcal{D}_m$, find split (j, s) that minimizes:

$$\min_{j,s}[n_L \cdot I(\mathcal{D}_L) + n_R \cdot I(\mathcal{D}_R)]$$

where $I(\cdot)$ is an impurity measure.

Impurity Measures:

For classification with K classes and class proportions $\hat{p}_{mk}$ in node m:

1. **Misclassification**: $I(m) = 1 - \max_k \hat{p}_{mk}$
2. **Gini Index**: $I(m) = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) = 1 - \sum_{k=1}^{K} \hat{p}_{mk}^2$
3. **Cross-Entropy**: $I(m) = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$

For regression with n_m samples and mean $\bar{y}_m$:

Mean Squared Error: $I(m) = \frac{1}{n_m} \sum_{i \in m} (y_i - \bar{y}_m)^2$

Pruning: Cost-complexity pruning minimizes:

$$C_\alpha(T) = \sum_{m=1}^{|T|} n_m \cdot I(m) + \alpha |T|$$

where $|T|$ is the number of terminal nodes.

1.3.2 Random Forests

Algorithm: 1. For $b = 1, \ldots, B$: - Draw bootstrap sample $\mathcal{D}_b$ from $\mathcal{D}$ - Train tree T_b on $\mathcal{D}_b$, at each split considering only m random features 2. Aggregate: $\hat{f}(x) = \frac{1}{B} \sum_{b=1}^{B} T_b(x)$ (regression) or majority vote (classification)

Variance Reduction Theory:

For trees with pairwise correlation ρ and individual variance σ^2:

$$\text{Var}\left[\frac{1}{B}\sum_{b=1}^{B} T_b\right] = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

As $B \to \infty$, variance approaches $\rho\sigma^2$. Feature subsampling reduces ρ.

Feature Importance:

Mean Decrease in Impurity (MDI):

$$\text{Importance}(j) = \frac{1}{B}\sum_{b=1}^{B}\sum_{t \in T_b} I(j \text{ splits at } t) \cdot \Delta I_t$$

Permutation Importance:

$$\text{Importance}(j) = \frac{1}{B}\sum_{b=1}^{B}(\text{OOB error after permuting } j) - (\text{OOB error})$$

1.3.3 Gradient Boosting

Framework: Sequentially fit models to residuals:

$$F_m(x) = F_{m-1}(x) + \nu h_m(x)$$

where h_m is fit to negative gradient of loss at current predictions.

Algorithm (for general loss L): 1. Initialize $F_0(x) =_\gamma \sum_{i=1}^{n} L(y_i, \gamma)$ 2. For $m = 1, \ldots, M$: - Compute pseudo-residuals: $r_{im} = -\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\Big|_{F=F_{m-1}}$ - Fit tree h_m to $\{(x_i, r_{im})\}$ - Update: $F_m = F_{m-1} + \nu h_m$

Gradient for Common Losses:

Loss	Pseudo-residual
Squared Error	$y_i - F(x_i)$
Absolute Error	$\text{sign}(y_i - F(x_i))$
Binomial Deviance	$y_i - \sigma(F(x_i))$

XGBoost Objective:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} L(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

where $\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$ regularizes tree complexity.

Second-order Taylor expansion:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

where $g_i = \partial_{\hat{y}} L$ and $h_i = \partial_{\hat{y}}^2 L$.

1.4 Support Vector Machines

1.4.1 Linear SVM

Hard Margin (separable case):

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

subject to: $y_i(w^T x_i + b) \geq 1, \quad \forall i$

Lagrangian:

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

KKT Conditions: - $\nabla_w \mathcal{L} = 0 \Rightarrow w = \sum_i \alpha_i y_i x_i$ - $\nabla_b \mathcal{L} = 0 \Rightarrow \sum_i \alpha_i y_i = 0$ - $\alpha_i \geq 0$ - $\alpha_i [y_i(w^T x_i + b) - 1] = 0$ (complementary slackness)

Dual Problem:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

subject to: $\alpha_i \geq 0$, $\sum_i \alpha_i y_i = 0$

Soft Margin (non-separable):

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

subject to: $y_i(w^T x_i + b) \geq 1 - \xi_i$, $\xi_i \geq 0$

Dual: same form but with $0 \leq \alpha_i \leq C$.

1.4.2 Kernel Methods

Kernel Trick: Replace inner products $x_i^T x_j$ with $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ for implicit feature map ϕ.

Common Kernels:

Kernel	Formula	Properties
Linear	$K(x,z) = x^T z$	Baseline
Polynomial	$K(x,z) = (x^T z + c)^d$	Degree-d interactions
RBF (Gaussian)	$K(x,z) = \exp(-\gamma \|x - z\|^2)$	Infinite-dimensional
Sigmoid	$K(x,z) = \tanh(\gamma x^T z + c)$	Neural network analogy

Mercer's Theorem: K is a valid kernel iff it is symmetric and positive semi-definite (the Gram matrix $K_{ij} = K(x_i, x_j)$ has non-negative eigenvalues).

1.5 Learning Theory

1.5.1 PAC Learning

Definition: A concept class $\mathcal{C}$ is PAC-learnable if there exists an algorithm A such that for any $\epsilon, \delta > 0$ and any distribution P over $\mathcal{X}$, given $m \geq \text{poly}(1/\epsilon, 1/\delta, \text{size}(c))$ samples, A outputs hypothesis h with:

$$P[R(h) - R(c^*) > \epsilon] < \delta$$

in time polynomial in the same quantities.

Finite Hypothesis Class: For $|\mathcal{H}| < \infty$, with probability $\geq 1 - \delta$:

$$R(\hat{h}) - R(h^*) \leq 2\sqrt{\frac{\log |\mathcal{H}| + \log(2/\delta)}{2n}}$$

1.5.2 VC Dimension

Definition: The VC dimension $d_{VC}(\mathcal{H})$ is the largest n such that $\mathcal{H}$ can shatter some set of n points (achieve all 2^n labelings).

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: $d_{VC} = O(W \log W)$

Generalization Bound: With probability $\geq 1 - \delta$:

$$R(h) \leq \hat{R}_n(h) + \sqrt{\frac{d_{VC}(\log(2n/d_{VC}) + 1) + \log(4/\delta)}{n}}$$

1.5.3 Rademacher Complexity

Definition: For function class $\mathcal{F}$ and sample $S = \{x_1, \ldots, x_n\}$:

$$\hat{\mathcal{R}}_S(\mathcal{F}) = \mathbb{E}_\sigma \left[\sup_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \sigma_i f(x_i) \right]$$

where σ_i are i.i.d. Rademacher variables (± 1 with probability $1/2$).

Generalization Bound: For bounded loss $L \in [0, 1]$:

$$R(h) \leq \hat{R}_n(h) + 2\mathcal{R}_n(\mathcal{H}) + 3\sqrt{\frac{\log(2/\delta)}{2n}}$$

1.6 Model Selection and Validation

1.6.1 Cross-Validation Theory

Leave-One-Out CV:

$$\text{CV}_n = \frac{1}{n} \sum_{i=1}^{n} L(y_i, \hat{f}^{(-i)}(x_i))$$

Bias-Variance of CV Estimates:

- LOOCV: Nearly unbiased for true risk, high variance
- k-fold CV: Slight bias (trains on $(k-1)/k$ of data), lower variance
- Typical choice: $k = 5$ or $k = 10$ balances bias-variance

Computational Shortcut for OLS (LOOCV):

$$\text{CV}_n = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{y_i - \hat{y}_i}{1 - h_{ii}} \right)^2$$

where h_{ii} are diagonal elements of the hat matrix $H = X(X^T X)^{-1} X^T$.

1.6.2 Information Criteria

Akaike Information Criterion (AIC):

$$\text{AIC} = -2\ell(\hat{\theta}) + 2p$$

where ℓ is log-likelihood and p is number of parameters.

Bayesian Information Criterion (BIC):

$$\text{BIC} = -2\ell(\hat{\theta}) + p \log n$$

BIC penalizes complexity more heavily for large n.

Mallows' Cp (for linear regression):

$$C_p = \frac{RSS_p}{\hat{\sigma}^2} - n + 2p$$

where RSS_p is residual sum of squares with p predictors.

1.7 Production Considerations

1.7.1 Feature Engineering at Scale

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer

def create_production_pipeline(numerical_features, categorical_features, model
    ):
    """
    Create a production-ready feature engineering pipeline.

    Handles:
    - Missing value imputation
    - Numerical scaling
    - Categorical encoding
    - Model fitting
    """
    numerical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())
    ])

    categorical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
        ('onehot', OneHotEncoder(handle_unknown='ignore', sparse_output=False)
    )
    ])

    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numerical_transformer, numerical_features),
            ('cat', categorical_transformer, categorical_features)
        ],
        remainder='drop'
    )

    pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('model', model)
    ])

    return pipeline
```

1.7.2 Model Serialization and Versioning

```
import joblib
import hashlib
import json
from datetime import datetime
from pathlib import Path

class ModelRegistry:
    """
    Version and track trained models for production deployment.
    """

    def __init__(self, registry_path: str):
        self.registry_path = Path(registry_path)
        self.registry_path.mkdir(parents=True, exist_ok=True)
        self.manifest_path = self.registry_path / 'manifest.json'
        self.manifest = self._load_manifest()

    def _load_manifest(self):
        if self.manifest_path.exists():
            return json.loads(self.manifest_path.read_text())
        return {'models': []}

    def _save_manifest(self):
        self.manifest_path.write_text(json.dumps(self.manifest, indent=2))

    def register_model(self, model, model_name: str, metrics: dict,
                       training_data_hash: str, hyperparameters: dict):
        """
        Register a trained model with full provenance tracking.
        """
        timestamp = datetime.now().isoformat()
        version = len([m for m in self.manifest['models']
                       if m['name'] == model_name]) + 1

        # Serialize model
        model_filename = f"{model_name}_v{version}.joblib"
        model_path = self.registry_path / model_filename
        joblib.dump(model, model_path)

        # Compute model hash
        model_hash = hashlib.md5(model_path.read_bytes()).hexdigest()

        # Record metadata
        entry = {
            'name': model_name,
            'version': version,
            'timestamp': timestamp,
            'path': str(model_path),
            'model_hash': model_hash,
            'training_data_hash': training_data_hash,
            'hyperparameters': hyperparameters,
            'metrics': metrics
        }

        self.manifest['models'].append(entry)
        self._save_manifest()

        return entry

    def load_model(self, model_name: str, version: int = None):
        """
```

```
        Load a registered model by name and version.
        """
        candidates = [m for m in self.manifest['models']
                      if m['name'] == model_name]

        if not candidates:
            raise ValueError(f"No model found with name: {model_name}")

        if version:
            model_entry = next((m for m in candidates if m['version'] ==
    version), None)
        else:
            model_entry = max(candidates, key=lambda m: m['version'])

        if not model_entry:
            raise ValueError(f"Version {version} not found for model: {
    model_name}")

        return joblib.load(model_entry['path']), model_entry
```

1.7.3 Online Monitoring System

```
import numpy as np
from collections import deque
from scipy import stats

class ModelMonitor:
    """
    Real-time model performance and data drift monitoring.
    """

    def __init__(self, reference_data, window_size=1000):
        self.reference_data = reference_data
        self.reference_stats = self._compute_statistics(reference_data)
        self.window_size = window_size
        self.prediction_buffer = deque(maxlen=window_size)
        self.feature_buffer = deque(maxlen=window_size)
        self.label_buffer = deque(maxlen=window_size)

    def _compute_statistics(self, data):
        return {
            'mean': np.mean(data, axis=0),
            'std': np.std(data, axis=0),
            'quantiles': np.percentile(data, [25, 50, 75], axis=0)
        }

    def log_prediction(self, features, prediction, label=None):
        """
        Log a prediction for monitoring.
        """
        self.feature_buffer.append(features)
        self.prediction_buffer.append(prediction)
        if label is not None:
            self.label_buffer.append(label)

    def detect_data_drift(self, significance=0.05):
        """
        Detect distribution shift using Kolmogorov-Smirnov test.
        """
        if len(self.feature_buffer) < 100:
            return None
```

```
        current_data = np.array(self.feature_buffer)
        drift_detected = []

        for i in range(current_data.shape[1]):
            statistic, p_value = stats.ks_2samp(
                self.reference_data[:, i],
                current_data[:, i]
            )
            if p_value < significance:
                drift_detected.append({
                    'feature_index': i,
                    'ks_statistic': statistic,
                    'p_value': p_value
                })

        return drift_detected

    def compute_psi(self, bins=10):
        """
        Population Stability Index for prediction distribution shift.
        """
        if len(self.prediction_buffer) < 100:
            return None

        reference_preds = self.reference_data[:, -1]
        current_preds = np.array(self.prediction_buffer)

        # Create bins from reference distribution
        bin_edges = np.percentile(reference_preds,
                                  np.linspace(0, 100, bins + 1))

        ref_counts, _ = np.histogram(reference_preds, bins=bin_edges)
        cur_counts, _ = np.histogram(current_preds, bins=bin_edges)

        ref_pct = (ref_counts + 1) / (len(reference_preds) + bins)
        cur_pct = (cur_counts + 1) / (len(current_preds) + bins)

        psi = np.sum((cur_pct - ref_pct) * np.log(cur_pct / ref_pct))

        return {
            'psi': psi,
            'interpretation': 'significant shift' if psi > 0.2 else
                              'moderate shift' if psi > 0.1 else 'stable'
        }

    def get_performance_metrics(self):
        """
        Compute current performance metrics if labels available.
        """
        if len(self.label_buffer) < 50:
            return None

        predictions = np.array(self.prediction_buffer)[-len(self.label_buffer)
:]
        labels = np.array(self.label_buffer)

        if len(np.unique(labels)) == 2:
            from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score
            pred_labels = (predictions > 0.5).astype(int)
            return {
                'accuracy': accuracy_score(labels, pred_labels),
```

```
                'precision': precision_score(labels, pred_labels,
    zero_division=0),
                'recall': recall_score(labels, pred_labels, zero_division=0),
                'f1': f1_score(labels, pred_labels, zero_division=0)
            }
        else:
            from sklearn.metrics import mean_squared_error,
    mean_absolute_error, r2_score
            return {
                'rmse': np.sqrt(mean_squared_error(labels, predictions)),
                'mae': mean_absolute_error(labels, predictions),
                'r2': r2_score(labels, predictions)
            }
```

1.7.4 A/B Testing Framework

```
from scipy import stats
import numpy as np

class ABTestEvaluator:
    """
    Statistical evaluation of model A/B tests.
    """

    def __init__(self, metric_type='conversion'):
        self.metric_type = metric_type
        self.control_results = []
        self.treatment_results = []

    def add_observation(self, group: str, outcome: float):
        if group == 'control':
            self.control_results.append(outcome)
        else:
            self.treatment_results.append(outcome)

    def compute_sample_size(self, baseline_rate, mde, alpha=0.05, power=0.8):
        """
        Compute required sample size for detecting minimum detectable effect.
        """
        from scipy.stats import norm

        z_alpha = norm.ppf(1 - alpha/2)
        z_beta = norm.ppf(power)

        p1 = baseline_rate
        p2 = baseline_rate * (1 + mde)

        pooled_var = p1*(1-p1) + p2*(1-p2)
        effect = abs(p2 - p1)

        n = ((z_alpha + z_beta)**2 * pooled_var) / (effect**2)

        return int(np.ceil(n))

    def evaluate(self, alpha=0.05):
        """
        Evaluate A/B test results with appropriate statistical test.
        """
        control = np.array(self.control_results)
        treatment = np.array(self.treatment_results)
```

```
        if self.metric_type == 'conversion':
            # Chi-squared test for proportions
            successes = [control.sum(), treatment.sum()]
            totals = [len(control), len(treatment)]
            chi2, p_value = stats.chi2_contingency(
                [[successes[0], totals[0] - successes[0]],
                 [successes[1], totals[1] - successes[1]]]
            )[:2]

            effect = treatment.mean() - control.mean()
            relative_effect = effect / control.mean() if control.mean() > 0
    else 0

        else:
            # Welch's t-test for continuous metrics
            t_stat, p_value = stats.ttest_ind(treatment, control, equal_var=
    False)
            effect = treatment.mean() - control.mean()
            relative_effect = effect / control.mean() if control.mean() > 0
    else 0

        # Confidence interval for effect
        se = np.sqrt(control.var()/len(control) + treatment.var()/len(
    treatment))
        ci_lower = effect - 1.96 * se
        ci_upper = effect + 1.96 * se

        return {
            'control_mean': control.mean(),
            'treatment_mean': treatment.mean(),
            'absolute_effect': effect,
            'relative_effect': relative_effect,
            'p_value': p_value,
            'significant': p_value < alpha,
            'ci_95': (ci_lower, ci_upper),
            'n_control': len(control),
            'n_treatment': len(treatment)
        }
```

1.7.5 Inference Optimization

```
import numpy as np
from functools import lru_cache

class OptimizedPredictor:
    """
    Optimized inference for production deployment.
    """

    def __init__(self, model, feature_names):
        self.model = model
        self.feature_names = feature_names

        # Pre-compile model if possible
        self._optimize_model()

    def _optimize_model(self):
        """
        Apply model-specific optimizations.
        """
        model_type = type(self.model).__name__
```

```
        if model_type in ['RandomForestClassifier', 'RandomForestRegressor']:
            # Reduce number of jobs for single predictions
            self.model.n_jobs = 1

        elif model_type in ['GradientBoostingClassifier', '
    GradientBoostingRegressor']:
            # Pre-allocate prediction arrays
            pass

    def predict_single(self, features: dict):
        """
        Optimized single-sample prediction.
        """
        # Convert dict to array in correct order
        x = np.array([[features.get(f, 0) for f in self.feature_names]])
        return self.model.predict(x)[0]

    def predict_batch(self, features_list: list, batch_size=1000):
        """
        Batched prediction for throughput optimization.
        """
        all_predictions = []

        for i in range(0, len(features_list), batch_size):
            batch = features_list[i:i + batch_size]
            X = np.array([[f.get(name, 0) for name in self.feature_names]
                          for f in batch])
            predictions = self.model.predict(X)
            all_predictions.extend(predictions)

        return all_predictions

    @lru_cache(maxsize=10000)
    def predict_cached(self, features_tuple):
        """
        Cached prediction for repeated identical inputs.
        """
        features = dict(features_tuple)
        return self.predict_single(features)
```

1.8 References

1. Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.
2. Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
3. Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press.
4. Vapnik, V. N. (1998). *Statistical Learning Theory*. Wiley.
5. Breiman, L. (2001). "Random Forests." *Machine Learning*, 45(1), 5-32.
6. Friedman, J. H. (2001). "Greedy Function Approximation: A Gradient Boosting Machine." *Annals of Statistics*, 29(5), 1189-1232.
7. Chen, T., & Guestrin, C. (2016). "XGBoost: A Scalable Tree Boosting System." *KDD*.
8. Shalev-Shwartz, S., & Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.

9. Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). *Foundations of Machine Learning*. MIT Press.
10. Sculley, D., et al. (2015). "Hidden Technical Debt in Machine Learning Systems." *NeurIPS*.

The gap between a working prototype and a production system is vast. Mathematical understanding provides the foundation for debugging unexpected behavior, while software engineering discipline ensures reliability at scale. Master both to build systems that deliver value consistently.