Validation Metrics - Advanced Handout

Machine Learning for Smarter Innovation

1 Validation Metrics - 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 Statistical Learning Framework

The goal of supervised learning is to find a function $f : \mathcal{X} \to \mathcal{Y}$ that minimizes the expected risk:

$$R(f) = \mathbb{E}_{(X,Y)\sim P}[L(Y, f(X))]$$

where $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ is a loss function and P is the true data distribution. Since P is unknown, we approximate using the empirical risk:

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

The fundamental challenge is that minimizing empirical risk does not guarantee minimizing true risk.

1.1.2 Bias-Variance Decomposition

For squared error loss, the expected prediction error at point x_0 decomposes as:

$$\mathbb{E}[(Y - \hat{f}(x_0))^2] = \text{Var}(\hat{f}(x_0)) + [\text{Bias}(\hat{f}(x_0))]^2 + \text{Var}(\epsilon)$$

where: - **Variance**: $\text{Var}(\hat{f}(x_0)) = \mathbb{E}[(\hat{f}(x_0) - \mathbb{E}[\hat{f}(x_0)])^2]$ - **Bias**: $\text{Bias}(\hat{f}(x_0)) = \mathbb{E}[\hat{f}(x_0)] - f(x_0)$ - **Irreducible error**: $\text{Var}(\epsilon) = \sigma^2$

This decomposition reveals the fundamental tradeoff: complex models have low bias but high variance, while simple models have high bias but low variance.

1.1.3 Generalization Bounds

The generalization gap between empirical and true risk can be bounded. For a finite hypothesis class $\mathcal{H}$ with $|\mathcal{H}|$ elements, with probability at least $1 - \delta$:

$$R(f) \leq \hat{R}_n(f) + \sqrt{\frac{\ln|\mathcal{H}| + \ln(1/\delta)}{2n}}$$

For infinite hypothesis classes, we use VC dimension d_{VC}:

$$R(f) \leq \hat{R}_n(f) + O\left(\sqrt{\frac{d_{VC}\ln(n/d_{VC}) + \ln(1/\delta)}{n}}\right)$$

These bounds motivate Occam's razor: prefer simpler models when empirical risk is similar.

1.2 Cross-Validation Theory

1.2.1 Estimator Properties

Cross-validation provides an estimator of the true risk. For K-fold CV, let $\hat{f}^{(-k)}$ denote the model trained on all folds except k. The CV estimator is:

$$\hat{R}_{CV} = \frac{1}{K}\sum_{k=1}^{K}\frac{1}{|S_k|}\sum_{i\in S_k} L(y_i, \hat{f}^{(-k)}(x_i))$$

where S_k is the set of indices in fold k.

Bias Analysis: K-fold CV has bias $O((K-1)/n)$ because each model is trained on $(K-1)/K$ of the data. Leave-one-out CV ($K = n$) has nearly unbiased risk estimation but high variance.

Variance Analysis: As K increases, the training sets overlap more, making fold estimates correlated. The variance of CV estimator is approximately:

$$\text{Var}(\hat{R}_{CV}) \approx \frac{\sigma^2}{K} + \frac{2(K-1)\rho\sigma^2}{K}$$

where ρ is the correlation between fold estimates and σ^2 is the variance of individual loss values.

1.2.2 Nested Cross-Validation

When using CV for both model selection and performance estimation, nested CV prevents optimistic bias. The outer loop estimates generalization, while the inner loop selects hyperparameters.

Let $\hat{\theta}^{(-k)}$ be the hyperparameters selected by inner CV on fold k's training set. The nested CV estimator is:

$$\hat{R}_{nested} = \frac{1}{K_{outer}}\sum_{k=1}^{K_{outer}}\frac{1}{|S_k|}\sum_{i\in S_k} L(y_i, \hat{f}^{(-k)}_{\hat{\theta}^{(-k)}}(x_i))$$

This estimator is approximately unbiased for the expected performance of the model selection procedure.

1.3 Statistical Hypothesis Testing for Model Comparison

1.3.1 Paired t-Test

For comparing two models A and B using K-fold CV, we compute the difference $d_k = R_A^{(k)} - R_B^{(k)}$ for each fold. Under the null hypothesis $H_0 : \mathbb{E}[d] = 0$:

$$t = \frac{\bar{d}}{s_d/\sqrt{K}} \sim t_{K-1}$$

where $\bar{d} = \frac{1}{K}\sum_k d_k$ and $s_d^2 = \frac{1}{K-1}\sum_k (d_k - \bar{d})^2$.

Limitation: The paired t-test assumes independence between folds, which is violated because training sets overlap.

1.3.2 Corrected Resampled t-Test

The Nadeau-Bengio corrected test accounts for correlation:

$$t_{corr} = \frac{\bar{d}}{\sqrt{(\frac{1}{K} + \frac{n_{test}}{n_{train}})s_d^2}}$$

where n_{test}/n_{train} is the ratio of test to training set sizes.

1.3.3 McNemar's Test

For paired binary outcomes (correct/incorrect), McNemar's test is more powerful. Let: - b = cases where A is correct and B is incorrect - c = cases where A is incorrect and B is correct

The test statistic is:

$$\chi^2 = \frac{(b-c)^2}{b+c} \sim \chi_1^2$$

With continuity correction:

$$\chi^2_{corrected} = \frac{(|b-c|-1)^2}{b+c}$$

1.3.4 Bayesian Model Comparison

Bayesian testing computes the posterior probability that model A is better:

$$P(\mu_A > \mu_B | \text{data}) = \int_{-\infty}^{\infty}\int_{\mu_B}^{\infty} p(\mu_A, \mu_B|\text{data}) d\mu_A d\mu_B$$

Using a conjugate normal-inverse-gamma prior, the posterior predictive for the difference $\delta = \mu_A - \mu_B$ follows a t-distribution:

$$\delta|\text{data} \sim t_{n_A+n_B-2}\left(\bar{d}, s_p\sqrt{\frac{1}{n_A} + \frac{1}{n_B}}\right)$$

where s_p is the pooled standard deviation.

1.4 Calibration Theory

1.4.1 Probabilistic Calibration

A probabilistic classifier is calibrated if:

$$P(Y = 1|\hat{p}(X) = p) = p \quad \forall p \in [0, 1]$$

In practice, we measure calibration using binned estimates. The reliability diagram plots:

$$\text{acc}(B_m) = \frac{1}{|B_m|} \sum_{i \in B_m} \mathbf{1}[y_i = 1]$$

against $\text{conf}(B_m) = \frac{1}{|B_m|} \sum_{i \in B_m} \hat{p}_i$ for bins B_m.

1.4.2 Expected Calibration Error

ECE quantifies miscalibration:

$$\text{ECE} = \sum_{m=1}^{M} \frac{|B_m|}{n} |\text{acc}(B_m) - \text{conf}(B_m)|$$

Maximum Calibration Error (MCE) captures worst-case miscalibration:

$$\text{MCE} = \max_m |\text{acc}(B_m) - \text{conf}(B_m)|$$

1.4.3 Calibration Methods

Platt Scaling: Fits a logistic regression on validation set:

$$p_{calibrated} = \sigma(a \cdot z + b)$$

where z is the logit (log-odds) from the original model. Parameters a, b are learned by minimizing:

$$\mathcal{L}(a, b) = -\sum_i [y_i \log p_i + (1 - y_i) \log(1 - p_i)]$$

Temperature Scaling: Special case with $a = 1/T$ and $b = 0$:

$$p_{calibrated} = \sigma(z/T)$$

Temperature $T > 1$ softens the distribution, $T < 1$ sharpens it.

Isotonic Regression: Non-parametric calibration that learns a monotonic function $m : [0, 1] \to [0, 1]$ minimizing:

$$\min_m \sum_i (y_i - m(\hat{p}_i))^2 \quad \text{s.t. } m \text{ is monotonic}$$

1.5 Implementation: Statistical Model Comparison

```
"""
Rigorous statistical model comparison with multiple testing corrections.
"""

import numpy as np
from typing import List, Dict, Tuple, Optional
from dataclasses import dataclass
from scipy import stats
from sklearn.model_selection import cross_val_score, StratifiedKFold
import warnings
```

```
@dataclass
class ComparisonResult:
    """Result of statistical model comparison."""
    model_a: str
    model_b: str
    mean_diff: float
    std_diff: float
    statistic: float
    pvalue: float
    significant: bool
    effect_size: float
    confidence_interval: Tuple[float, float]
    test_type: str


class StatisticalModelComparison:
    """
    Comprehensive statistical comparison of ML models.
    Implements multiple testing procedures with corrections.
    """

    def __init__(self, alpha: float = 0.05, correction: str = "bonferroni"):
        """
        Args:
            alpha: Significance level
            correction: Multiple testing correction method
                        ("bonferroni", "holm", "fdr_bh", "none")
        """
        self.alpha = alpha
        self.correction = correction

    def paired_ttest(
        self,
        scores_a: np.ndarray,
        scores_b: np.ndarray,
        name_a: str = "Model A",
        name_b: str = "Model B"
    ) -> ComparisonResult:
        """Standard paired t-test for CV scores."""
        diff = scores_a - scores_b
        n = len(diff)

        mean_diff = np.mean(diff)
        std_diff = np.std(diff, ddof=1)

        t_stat = mean_diff / (std_diff / np.sqrt(n))
        pvalue = 2 * stats.t.sf(abs(t_stat), df=n-1)

        # Effect size (Cohen's d)
        effect_size = mean_diff / std_diff

        # Confidence interval
        t_crit = stats.t.ppf(1 - self.alpha/2, df=n-1)
        margin = t_crit * std_diff / np.sqrt(n)
        ci = (mean_diff - margin, mean_diff + margin)

        return ComparisonResult(
            model_a=name_a,
            model_b=name_b,
            mean_diff=mean_diff,
            std_diff=std_diff,
            statistic=t_stat,
```

```
            pvalue=pvalue,
            significant=pvalue < self.alpha,
            effect_size=effect_size,
            confidence_interval=ci,
            test_type="paired_ttest"
        )

    def corrected_resampled_ttest(
        self,
        scores_a: np.ndarray,
        scores_b: np.ndarray,
        n_train: int,
        n_test: int,
        name_a: str = "Model A",
        name_b: str = "Model B"
    ) -> ComparisonResult:
        """Nadeau-Bengio corrected t-test for repeated CV."""
        diff = scores_a - scores_b
        n = len(diff)

        mean_diff = np.mean(diff)
        var_diff = np.var(diff, ddof=1)

        # Correction factor for correlation
        correction = 1/n + n_test / n_train
        t_stat = mean_diff / np.sqrt(correction * var_diff)

        # Approximate degrees of freedom (conservative)
        df = n - 1
        pvalue = 2 * stats.t.sf(abs(t_stat), df=df)

        # Effect size
        effect_size = mean_diff / np.sqrt(var_diff)

        # CI (approximate)
        t_crit = stats.t.ppf(1 - self.alpha/2, df=df)
        margin = t_crit * np.sqrt(correction * var_diff)
        ci = (mean_diff - margin, mean_diff + margin)

        return ComparisonResult(
            model_a=name_a,
            model_b=name_b,
            mean_diff=mean_diff,
            std_diff=np.sqrt(var_diff),
            statistic=t_stat,
            pvalue=pvalue,
            significant=pvalue < self.alpha,
            effect_size=effect_size,
            confidence_interval=ci,
            test_type="corrected_resampled_ttest"
        )

    def mcnemar_test(
        self,
        predictions_a: np.ndarray,
        predictions_b: np.ndarray,
        y_true: np.ndarray,
        name_a: str = "Model A",
        name_b: str = "Model B"
    ) -> ComparisonResult:
        """McNemar's test for paired binary outcomes."""
        correct_a = (predictions_a == y_true)
        correct_b = (predictions_b == y_true)
```

```
        # Contingency table
        b = np.sum(correct_a & ~correct_b)  # A correct, B incorrect
        c = np.sum(~correct_a & correct_b)  # A incorrect, B correct

        # Chi-squared statistic with continuity correction
        if b + c == 0:
            statistic = 0.0
            pvalue = 1.0
        else:
            statistic = (abs(b - c) - 1)**2 / (b + c)
            pvalue = 1 - stats.chi2.cdf(statistic, df=1)

        # Effect size (odds ratio)
        effect_size = (b + 0.5) / (c + 0.5)

        # Proportion difference
        mean_diff = (b - c) / len(y_true)

        return ComparisonResult(
            model_a=name_a,
            model_b=name_b,
            mean_diff=mean_diff,
            std_diff=0.0,  # Not applicable
            statistic=statistic,
            pvalue=pvalue,
            significant=pvalue < self.alpha,
            effect_size=effect_size,
            confidence_interval=(0, 0),  # Not applicable
            test_type="mcnemar"
        )

    def bayesian_comparison(
        self,
        scores_a: np.ndarray,
        scores_b: np.ndarray,
        rope: float = 0.01,
        name_a: str = "Model A",
        name_b: str = "Model B"
    ) -> Dict:
        """
        Bayesian comparison with ROPE (Region of Practical Equivalence).

        Args:
            rope: Threshold for practical equivalence (e.g., 0.01 = 1%)

        Returns:
            Posterior probabilities and rope analysis
        """
        diff = scores_a - scores_b
        n = len(diff)

        mean_diff = np.mean(diff)
        std_diff = np.std(diff, ddof=1)
        se = std_diff / np.sqrt(n)

        # Posterior is approximately t-distributed
        df = n - 1

        # Probability that A > B
        prob_a_better = 1 - stats.t.cdf(0, df=df, loc=mean_diff, scale=se)

        # Probability that difference is within ROPE
```

```
        prob_rope = (
            stats.t.cdf(rope, df=df, loc=mean_diff, scale=se) -
            stats.t.cdf(-rope, df=df, loc=mean_diff, scale=se)
        )

        # HDI (Highest Density Interval)
        hdi_low = mean_diff - stats.t.ppf(0.975, df=df) * se
        hdi_high = mean_diff + stats.t.ppf(0.975, df=df) * se

        return {
            "mean_diff": mean_diff,
            "std_diff": std_diff,
            "prob_a_better": prob_a_better,
            "prob_b_better": 1 - prob_a_better,
            "prob_rope": prob_rope,
            "hdi_95": (hdi_low, hdi_high),
            "rope_decision": self._rope_decision(prob_rope, rope, hdi_low,
    hdi_high)
        }

    def _rope_decision(
        self,
        prob_rope: float,
        rope: float,
        hdi_low: float,
        hdi_high: float
    ) -> str:
        """Make decision based on ROPE analysis."""
        if hdi_high < -rope:
            return "B is better"
        elif hdi_low > rope:
            return "A is better"
        elif hdi_low >= -rope and hdi_high <= rope:
            return "Practically equivalent"
        else:
            return "Undecided"

    def multiple_comparison(
        self,
        models: Dict[str, np.ndarray]
    ) -> List[ComparisonResult]:
        """
        Compare all pairs of models with multiple testing correction.
        """
        model_names = list(models.keys())
        n_comparisons = len(model_names) * (len(model_names) - 1) // 2
        results = []
        pvalues = []

        # All pairwise comparisons
        for i in range(len(model_names)):
            for j in range(i + 1, len(model_names)):
                result = self.paired_ttest(
                    models[model_names[i]],
                    models[model_names[j]],
                    model_names[i],
                    model_names[j]
                )
                results.append(result)
                pvalues.append(result.pvalue)

        # Apply correction
        adjusted_alpha = self._adjust_alpha(pvalues, n_comparisons)
```

```
        # Update significance based on corrected alpha
        for i, result in enumerate(results):
            result.significant = result.pvalue < adjusted_alpha[i]

        return results

    def _adjust_alpha(
        self,
        pvalues: List[float],
        n_comparisons: int
    ) -> List[float]:
        """Apply multiple testing correction."""
        pvalues = np.array(pvalues)

        if self.correction == "bonferroni":
            return [self.alpha / n_comparisons] * len(pvalues)

        elif self.correction == "holm":
            # Holm-Bonferroni step-down
            sorted_idx = np.argsort(pvalues)
            adjusted = np.zeros(len(pvalues))
            for rank, idx in enumerate(sorted_idx):
                adjusted[idx] = self.alpha / (n_comparisons - rank)
            return adjusted.tolist()

        elif self.correction == "fdr_bh":
            # Benjamini-Hochberg FDR
            sorted_idx = np.argsort(pvalues)
            adjusted = np.zeros(len(pvalues))
            for rank, idx in enumerate(sorted_idx):
                adjusted[idx] = self.alpha * (rank + 1) / n_comparisons
            return adjusted.tolist()

        else:  # No correction
            return [self.alpha] * len(pvalues)
```

1.6 Implementation: Nested Cross-Validation

```
"""
Production nested cross-validation with confidence intervals.
"""

from sklearn.model_selection import KFold, StratifiedKFold, GridSearchCV
from sklearn.base import clone
import numpy as np
from typing import Dict, List, Any, Callable
from dataclasses import dataclass


@dataclass
class NestedCVResult:
    """Result from nested cross-validation."""
    outer_scores: np.ndarray
    mean_score: float
    std_score: float
    confidence_interval: tuple
    best_params_per_fold: List[Dict]
    inner_scores: List[np.ndarray]
```

```
class NestedCrossValidator:
    """
    Nested cross-validation for unbiased model evaluation.
    Separates hyperparameter tuning from performance estimation.
    """

    def __init__(
        self,
        estimator,
        param_grid: Dict[str, List[Any]],
        scoring: str = "accuracy",
        n_outer: int = 5,
        n_inner: int = 3,
        stratified: bool = True,
        random_state: int = 42
    ):
        self.estimator = estimator
        self.param_grid = param_grid
        self.scoring = scoring
        self.n_outer = n_outer
        self.n_inner = n_inner
        self.stratified = stratified
        self.random_state = random_state

    def fit(self, X: np.ndarray, y: np.ndarray) -> NestedCVResult:
        """Run nested CV and return comprehensive results."""
        # Initialize CV splitters
        if self.stratified:
            outer_cv = StratifiedKFold(
                n_splits=self.n_outer,
                shuffle=True,
                random_state=self.random_state
            )
            inner_cv = StratifiedKFold(
                n_splits=self.n_inner,
                shuffle=True,
                random_state=self.random_state
            )
        else:
            outer_cv = KFold(
                n_splits=self.n_outer,
                shuffle=True,
                random_state=self.random_state
            )
            inner_cv = KFold(
                n_splits=self.n_inner,
                shuffle=True,
                random_state=self.random_state
            )

        outer_scores = []
        best_params_per_fold = []
        inner_scores = []

        for fold_idx, (train_idx, test_idx) in enumerate(outer_cv.split(X, y))
    :
            X_train, X_test = X[train_idx], X[test_idx]
            y_train, y_test = y[train_idx], y[test_idx]

            # Inner CV: hyperparameter tuning
            grid_search = GridSearchCV(
```

```
                clone(self.estimator),
                self.param_grid,
                cv=inner_cv,
                scoring=self.scoring,
                n_jobs=-1,
                refit=True
            )
            grid_search.fit(X_train, y_train)

            best_params_per_fold.append(grid_search.best_params_)
            inner_scores.append(grid_search.cv_results_["mean_test_score"])

            # Outer CV: evaluate best model on held-out fold
            best_model = grid_search.best_estimator_
            score = self._compute_score(best_model, X_test, y_test)
            outer_scores.append(score)

        outer_scores = np.array(outer_scores)

        # Compute confidence interval
        ci = self._confidence_interval(outer_scores)

        return NestedCVResult(
            outer_scores=outer_scores,
            mean_score=outer_scores.mean(),
            std_score=outer_scores.std(),
            confidence_interval=ci,
            best_params_per_fold=best_params_per_fold,
            inner_scores=inner_scores
        )

    def _compute_score(self, model, X, y) -> float:
        """Compute score based on scoring parameter."""
        from sklearn.metrics import (
            accuracy_score, f1_score, roc_auc_score,
            precision_score, recall_score
        )

        y_pred = model.predict(X)

        if self.scoring == "accuracy":
            return accuracy_score(y, y_pred)
        elif self.scoring == "f1":
            return f1_score(y, y_pred)
        elif self.scoring == "precision":
            return precision_score(y, y_pred)
        elif self.scoring == "recall":
            return recall_score(y, y_pred)
        elif self.scoring == "roc_auc":
            y_proba = model.predict_proba(X)[:, 1]
            return roc_auc_score(y, y_proba)
        else:
            return model.score(X, y)

    def _confidence_interval(
        self,
        scores: np.ndarray,
        confidence: float = 0.95
    ) -> tuple:
        """Compute confidence interval using t-distribution."""
        n = len(scores)
        mean = scores.mean()
        se = scores.std(ddof=1) / np.sqrt(n)
```

```
        t_crit = stats.t.ppf((1 + confidence) / 2, df=n-1)
        margin = t_crit * se

        return (mean - margin, mean + margin)


class TimeSeriesNestedCV:
    """
    Nested CV for time series with walk-forward validation.
    Prevents data leakage from future to past.
    """

    def __init__(
        self,
        estimator,
        param_grid: Dict[str, List[Any]],
        scoring: str = "accuracy",
        n_outer: int = 5,
        n_inner: int = 3,
        min_train_size: int = None
    ):
        self.estimator = estimator
        self.param_grid = param_grid
        self.scoring = scoring
        self.n_outer = n_outer
        self.n_inner = n_inner
        self.min_train_size = min_train_size

    def fit(self, X: np.ndarray, y: np.ndarray) -> NestedCVResult:
        """Walk-forward nested CV."""
        from sklearn.model_selection import TimeSeriesSplit

        n_samples = len(X)
        outer_cv = TimeSeriesSplit(n_splits=self.n_outer)

        outer_scores = []
        best_params_per_fold = []

        for train_idx, test_idx in outer_cv.split(X):
            # Skip if training set too small
            if self.min_train_size and len(train_idx) < self.min_train_size:
                continue

            X_train, X_test = X[train_idx], X[test_idx]
            y_train, y_test = y[train_idx], y[test_idx]

            # Inner CV with time series split
            inner_cv = TimeSeriesSplit(n_splits=self.n_inner)

            grid_search = GridSearchCV(
                clone(self.estimator),
                self.param_grid,
                cv=inner_cv,
                scoring=self.scoring,
                n_jobs=-1,
                refit=True
            )
            grid_search.fit(X_train, y_train)

            best_params_per_fold.append(grid_search.best_params_)

            # Evaluate on test fold
```

```
            best_model = grid_search.best_estimator_
            score = best_model.score(X_test, y_test)
            outer_scores.append(score)

        outer_scores = np.array(outer_scores)

        return NestedCVResult(
            outer_scores=outer_scores,
            mean_score=outer_scores.mean(),
            std_score=outer_scores.std(),
            confidence_interval=self._confidence_interval(outer_scores),
            best_params_per_fold=best_params_per_fold,
            inner_scores=[]
        )

    def _confidence_interval(self, scores, confidence=0.95):
        n = len(scores)
        if n < 2:
            return (scores.mean(), scores.mean())
        mean = scores.mean()
        se = scores.std(ddof=1) / np.sqrt(n)
        t_crit = stats.t.ppf((1 + confidence) / 2, df=n-1)
        margin = t_crit * se
        return (mean - margin, mean + margin)
```

1.7 Implementation: Calibration Analysis

```
"""
Probability calibration analysis and correction methods.
"""

import numpy as np
from typing import Tuple, List, Optional
from sklearn.isotonic import IsotonicRegression
from sklearn.linear_model import LogisticRegression
from scipy.special import expit, logit
from dataclasses import dataclass


@dataclass
class CalibrationMetrics:
    """Calibration evaluation metrics."""
    ece: float  # Expected Calibration Error
    mce: float  # Maximum Calibration Error
    brier: float  # Brier score
    bin_accuracies: np.ndarray
    bin_confidences: np.ndarray
    bin_counts: np.ndarray


class CalibrationAnalyzer:
    """
    Comprehensive probability calibration analysis.
    """

    def __init__(self, n_bins: int = 10, strategy: str = "uniform"):
        """
        Args:
            n_bins: Number of bins for calibration analysis
```

```
            strategy: "uniform" (equal width) or "quantile" (equal count)
        """
        self.n_bins = n_bins
        self.strategy = strategy

    def analyze(
        self,
        y_true: np.ndarray,
        y_prob: np.ndarray
    ) -> CalibrationMetrics:
        """Compute calibration metrics."""
        # Create bins
        if self.strategy == "uniform":
            bins = np.linspace(0, 1, self.n_bins + 1)
        else:  # quantile
            bins = np.percentile(y_prob, np.linspace(0, 100, self.n_bins + 1))
            bins[0] = 0
            bins[-1] = 1

        bin_indices = np.digitize(y_prob, bins[1:-1])

        # Compute per-bin statistics
        bin_accuracies = np.zeros(self.n_bins)
        bin_confidences = np.zeros(self.n_bins)
        bin_counts = np.zeros(self.n_bins)

        for b in range(self.n_bins):
            mask = (bin_indices == b)
            if mask.sum() > 0:
                bin_accuracies[b] = y_true[mask].mean()
                bin_confidences[b] = y_prob[mask].mean()
                bin_counts[b] = mask.sum()

        # Expected Calibration Error
        weights = bin_counts / bin_counts.sum()
        ece = np.sum(weights * np.abs(bin_accuracies - bin_confidences))

        # Maximum Calibration Error
        nonzero_mask = bin_counts > 0
        if nonzero_mask.any():
            mce = np.max(np.abs(bin_accuracies[nonzero_mask] -
                                bin_confidences[nonzero_mask]))
        else:
            mce = 0.0

        # Brier score
        brier = np.mean((y_prob - y_true) ** 2)

        return CalibrationMetrics(
            ece=ece,
            mce=mce,
            brier=brier,
            bin_accuracies=bin_accuracies,
            bin_confidences=bin_confidences,
            bin_counts=bin_counts
        )


class PlattScaling:
    """Platt scaling for probability calibration."""

    def __init__(self):
        self.a = None
```

```
        self.b = None

    def fit(self, y_true: np.ndarray, y_prob: np.ndarray):
        """Fit Platt scaling parameters."""
        # Convert to logits
        eps = 1e-10
        y_prob = np.clip(y_prob, eps, 1 - eps)
        logits = logit(y_prob)

        # Fit logistic regression
        lr = LogisticRegression(solver='lbfgs')
        lr.fit(logits.reshape(-1, 1), y_true)

        self.a = lr.coef_[0, 0]
        self.b = lr.intercept_[0]

        return self

    def predict(self, y_prob: np.ndarray) -> np.ndarray:
        """Apply calibration."""
        eps = 1e-10
        y_prob = np.clip(y_prob, eps, 1 - eps)
        logits = logit(y_prob)
        return expit(self.a * logits + self.b)


class TemperatureScaling:
    """Temperature scaling for probability calibration."""

    def __init__(self):
        self.temperature = 1.0

    def fit(
        self,
        y_true: np.ndarray,
        y_prob: np.ndarray,
        method: str = "nll"
    ):
        """
        Fit temperature parameter.

        Args:
            method: "nll" (negative log-likelihood) or "ece" (ECE)
        """
        from scipy.optimize import minimize_scalar

        eps = 1e-10
        y_prob = np.clip(y_prob, eps, 1 - eps)
        logits = logit(y_prob)

        def objective(T):
            scaled_prob = expit(logits / T)
            if method == "nll":
                # Negative log-likelihood
                return -np.mean(
                    y_true * np.log(scaled_prob + eps) +
                    (1 - y_true) * np.log(1 - scaled_prob + eps)
                )
            else:  # ECE
                analyzer = CalibrationAnalyzer(n_bins=10)
                metrics = analyzer.analyze(y_true, scaled_prob)
                return metrics.ece
```

```
        result = minimize_scalar(
            objective,
            bounds=(0.1, 10.0),
            method='bounded'
        )
        self.temperature = result.x

        return self

    def predict(self, y_prob: np.ndarray) -> np.ndarray:
        """Apply temperature scaling."""
        eps = 1e-10
        y_prob = np.clip(y_prob, eps, 1 - eps)
        logits = logit(y_prob)
        return expit(logits / self.temperature)


class IsotonicCalibration:
    """Isotonic regression for probability calibration."""

    def __init__(self):
        self.iso = IsotonicRegression(out_of_bounds='clip')

    def fit(self, y_true: np.ndarray, y_prob: np.ndarray):
        """Fit isotonic regression."""
        self.iso.fit(y_prob, y_true)
        return self

    def predict(self, y_prob: np.ndarray) -> np.ndarray:
        """Apply calibration."""
        return self.iso.predict(y_prob)


class CalibratedClassifier:
    """
    Wrapper that adds calibration to any classifier.
    """

    def __init__(
        self,
        base_classifier,
        method: str = "platt",
        cv: int = 5
    ):
        """
        Args:
            base_classifier: Classifier with predict_proba method
            method: "platt", "temperature", or "isotonic"
            cv: Number of CV folds for calibration
        """
        self.base = base_classifier
        self.method = method
        self.cv = cv
        self.calibrator = None

    def fit(self, X: np.ndarray, y: np.ndarray):
        """Fit classifier and calibrator using CV."""
        from sklearn.model_selection import cross_val_predict
        from sklearn.base import clone

        # Get CV predictions for calibration
        y_prob = cross_val_predict(
            clone(self.base),
```

```
            X, y,
            cv=self.cv,
            method='predict_proba'
        )[:, 1]

        # Fit calibrator on CV predictions
        if self.method == "platt":
            self.calibrator = PlattScaling().fit(y, y_prob)
        elif self.method == "temperature":
            self.calibrator = TemperatureScaling().fit(y, y_prob)
        elif self.method == "isotonic":
            self.calibrator = IsotonicCalibration().fit(y, y_prob)

        # Fit base classifier on full data
        self.base.fit(X, y)

        return self

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """Return calibrated probabilities."""
        y_prob = self.base.predict_proba(X)[:, 1]
        y_calibrated = self.calibrator.predict(y_prob)

        # Return 2D array for sklearn compatibility
        return np.column_stack([1 - y_calibrated, y_calibrated])

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Return predictions using calibrated probabilities."""
        return (self.predict_proba(X)[:, 1] >= 0.5).astype(int)
```

1.8 Implementation: Production Monitoring

```
"""
Production model monitoring with drift detection and alerting.
"""

from typing import Dict, List, Callable, Optional
from dataclasses import dataclass, field
from datetime import datetime, timedelta
from collections import deque
import numpy as np
from scipy import stats


@dataclass
class DriftResult:
    """Result of drift detection."""
    feature: str
    statistic: float
    pvalue: float
    drifted: bool
    severity: str  # "none", "warning", "critical"


class DriftDetector:
    """
    Statistical drift detection for model monitoring.
    """
```

```
    def __init__(
        self,
        reference_data: np.ndarray,
        feature_names: List[str],
        warning_threshold: float = 0.05,
        critical_threshold: float = 0.01
    ):
        self.reference = reference_data
        self.feature_names = feature_names
        self.warning_threshold = warning_threshold
        self.critical_threshold = critical_threshold

    def detect_feature_drift(
        self,
        current_data: np.ndarray
    ) -> List[DriftResult]:
        """Detect drift in each feature using KS test."""
        results = []

        for i, feature in enumerate(self.feature_names):
            stat, pvalue = stats.ks_2samp(
                self.reference[:, i],
                current_data[:, i]
            )

            if pvalue < self.critical_threshold:
                severity = "critical"
                drifted = True
            elif pvalue < self.warning_threshold:
                severity = "warning"
                drifted = True
            else:
                severity = "none"
                drifted = False

            results.append(DriftResult(
                feature=feature,
                statistic=stat,
                pvalue=pvalue,
                drifted=drifted,
                severity=severity
            ))

        return results

    def detect_label_drift(
        self,
        reference_labels: np.ndarray,
        current_labels: np.ndarray
    ) -> DriftResult:
        """Detect drift in label distribution."""
        # Chi-squared test for categorical
        if len(np.unique(reference_labels)) <= 10:
            ref_counts = np.bincount(reference_labels.astype(int))
            cur_counts = np.bincount(current_labels.astype(int))

            # Align lengths
            max_len = max(len(ref_counts), len(cur_counts))
            ref_counts = np.pad(ref_counts, (0, max_len - len(ref_counts)))
            cur_counts = np.pad(cur_counts, (0, max_len - len(cur_counts)))

            # Expected under no drift
            total_ref = ref_counts.sum()
```

```
            total_cur = cur_counts.sum()
            expected = ref_counts * (total_cur / total_ref)

            # Chi-squared statistic
            mask = expected > 0
            stat = np.sum((cur_counts[mask] - expected[mask])**2 / expected[
    mask])
            pvalue = 1 - stats.chi2.cdf(stat, df=mask.sum() - 1)
        else:
            # KS test for continuous
            stat, pvalue = stats.ks_2samp(reference_labels, current_labels)

        if pvalue < self.critical_threshold:
            severity = "critical"
        elif pvalue < self.warning_threshold:
            severity = "warning"
        else:
            severity = "none"

        return DriftResult(
            feature="label",
            statistic=stat,
            pvalue=pvalue,
            drifted=pvalue < self.warning_threshold,
            severity=severity
        )


class PerformanceMonitor:
    """
    Continuous performance monitoring with degradation detection.
    """

    def __init__(
        self,
        baseline_metrics: Dict[str, float],
        window_size: int = 100,
        degradation_threshold: float = 0.05
    ):
        self.baseline = baseline_metrics
        self.window_size = window_size
        self.threshold = degradation_threshold
        self.metric_history: Dict[str, deque] = {
            metric: deque(maxlen=window_size)
            for metric in baseline_metrics
        }

    def update(
        self,
        y_true: np.ndarray,
        y_pred: np.ndarray,
        y_prob: Optional[np.ndarray] = None
    ) -> Dict[str, Dict]:
        """Update metrics and check for degradation."""
        from sklearn.metrics import (
            accuracy_score, precision_score, recall_score,
            f1_score, roc_auc_score
        )

        # Compute current metrics
        current_metrics = {
            "accuracy": accuracy_score(y_true, y_pred),
            "precision": precision_score(y_true, y_pred, zero_division=0),
```

```
            "recall": recall_score(y_true, y_pred, zero_division=0),
            "f1": f1_score(y_true, y_pred, zero_division=0)
        }

        if y_prob is not None and len(np.unique(y_true)) > 1:
            current_metrics["roc_auc"] = roc_auc_score(y_true, y_prob)

        # Update history and check degradation
        alerts = {}
        for metric, value in current_metrics.items():
            if metric in self.metric_history:
                self.metric_history[metric].append(value)

                # Check for degradation
                if metric in self.baseline:
                    baseline_val = self.baseline[metric]
                    window_mean = np.mean(list(self.metric_history[metric]))
                    degradation = baseline_val - window_mean

                    alerts[metric] = {
                        "current": value,
                        "window_mean": window_mean,
                        "baseline": baseline_val,
                        "degradation": degradation,
                        "alert": degradation > self.threshold
                    }

        return alerts

    def get_trends(self) -> Dict[str, Dict]:
        """Get trend analysis for each metric."""
        trends = {}

        for metric, history in self.metric_history.items():
            if len(history) < 10:
                continue

            values = np.array(list(history))
            x = np.arange(len(values))

            # Linear regression for trend
            slope, intercept, r_value, p_value, std_err = stats.linregress(x,
values)

            trends[metric] = {
                "slope": slope,
                "r_squared": r_value ** 2,
                "p_value": p_value,
                "trend": "improving" if slope > 0 else "degrading",
                "significant": p_value < 0.05
            }

        return trends
```

1.9 Common Parameters

Component	Parameter	Typical Range	Notes
Cross-validation	n_splits	5-10	Balance bias-variance tradeoff
Nested CV	n_outer	5-10	More = lower variance, higher cost
Nested CV	n_inner	3-5	For hyperparameter tuning
Statistical test	alpha	0.01-0.05	Type I error rate
Power analysis	power	0.8-0.9	1 - Type II error rate
Calibration	n_bins	10-20	For ECE computation
Temperature scaling	T	0.5-5.0	Learned from validation set
Drift detection	threshold	0.01-0.05	KS test significance level
Monitoring	window_size	50-500	Recent samples for averaging

1.10 Practice Problems

1. **Nested CV Comparison**: Compare nested CV vs single-level CV for hyperparameter tuning. Show that single-level CV produces optimistically biased estimates.
2. **Statistical Power**: Design a model comparison experiment. Calculate required sample size to detect a 2% difference in F1 score with 80% power.
3. **Calibration Study**: Train multiple classifiers on imbalanced data. Compute ECE before and after temperature scaling. Visualize reliability diagrams.
4. **Drift Simulation**: Simulate gradual covariate drift by shifting feature distributions over time. Implement sliding-window KS test to detect drift onset.
5. **Multi-objective Selection**: Train models optimizing different metrics. Identify Pareto frontier and implement weighted scoring for final selection.

1.11 References

1. Cawley, G. C., & Talbot, N. L. (2010). "On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation." JMLR, 11, 2079-2107.
2. Nadeau, C., & Bengio, Y. (2003). "Inference for the Generalization Error." Machine Learning, 52(3), 239-281.
3. Guo, C., Pleiss, G., Sun, Y., & Weinberger, K. Q. (2017). "On Calibration of Modern Neural Networks." ICML.
4. Raschka, S. (2018). "Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning." arXiv:1811.12808.
5. Dietterich, T. G. (1998). "Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms." Neural Computation, 10(7), 1895-1923.
6. Vovk, V., Gammerman, A., & Shafer, G. (2005). "Algorithmic Learning in a Random World." Springer.

Model validation transforms intuition into evidence. Rigorous statistical procedures ensure deployed models meet business requirements with quantified confidence.