Handout 3: Statistical Model Selection Deployment

Machine Learning for Smarter Innovation

1 Handout 3: Statistical Model Selection & Deployment

1.1 Week 9 - Advanced Level

1.1.1 Introduction

You've mastered multi-metric evaluation and cross-validation. Now we tackle the hardest problems: How do you select between models when different metrics favor different choices? How do you quantify confidence in deployment decisions? How do you monitor model performance in production and detect when retraining is needed?

Target Audience: Advanced students and practitioners building production ML systems who need rigorous statistical foundations for model selection and deployment decision-making.

Prerequisites: Strong Python skills, deep understanding of classification metrics, experience with sklearn pipelines, familiarity with statistical hypothesis testing, calculus and linear algebra background.

1.2 Part 1: Advanced Cross-Validation Strategies

1.2.1 Nested Cross-Validation (Unbiased Hyperparameter Tuning)

```
from sklearn.model_selection import cross_val_score, GridSearchCV, KFold
from sklearn.ensemble import RandomForestClassifier
import numpy as np

def nested_cv(X, y, n_outer=5, n_inner=3):
    """
    Nested CV: Outer loop estimates generalization, inner loop tunes
    hyperparameters.
    Prevents hyperparameter overfitting to validation set.
    """
    outer_cv = KFold(n_splits=n_outer, shuffle=True, random_state=42)
    inner_cv = KFold(n_splits=n_inner, shuffle=True, random_state=42)

    outer_scores = []

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

        # Inner CV: Hyperparameter tuning
        param_grid = {
            'n_estimators': [50, 100, 200],
```

```
            'max_depth': [10, 20, None]
        }
        grid_search = GridSearchCV(
            RandomForestClassifier(random_state=42),
            param_grid,
            cv=inner_cv,
            scoring='f1'
        )
        grid_search.fit(X_train, y_train)

        # Outer CV: Evaluate best model on held-out fold
        best_model = grid_search.best_estimator_
        score = f1_score(y_test, best_model.predict(X_test))
        outer_scores.append(score)

    return np.array(outer_scores)

# Run nested CV
scores = nested_cv(X, y, n_outer=5, n_inner=3)
print(f"Nested CV F1: {scores.mean():.3f} (+/- {scores.std():.3f})")
print(f"95% CI: [{scores.mean() - 1.96*scores.std():.3f}, {scores.mean() +
    1.96*scores.std():.3f}]")
```

Why nested CV? - Single-level GridSearchCV overestimates performance (tuned to validation set) - Nested CV gives unbiased estimate of generalization error - Critical for papers/publications requiring rigorous evaluation

1.2.2 Time-Series Walk-Forward Validation

```
from sklearn.model_selection import TimeSeriesSplit

def walk_forward_validation(X, y, model, n_splits=5):
    """
    Walk-forward validation for time-series:
    - Train on [1, t]
    - Test on [t+1, t+k]
    - Slide forward
    """
    tscv = TimeSeriesSplit(n_splits=n_splits)
    scores = []

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

        model.fit(X_train, y_train)
        score = f1_score(y_test, model.predict(X_test))
        scores.append(score)

        print(f"Fold {i+1}: Train=[{train_idx[0]}:{train_idx[-1]}], "
              f"Test=[{test_idx[0]}:{test_idx[-1]}], F1={score:.3f}")

    return np.array(scores)

# Example usage
from sklearn.ensemble import GradientBoostingClassifier
model = GradientBoostingClassifier(n_estimators=100, random_state=42)
scores = walk_forward_validation(X, y, model, n_splits=5)
print(f"\nMean F1: {scores.mean():.3f} (+/- {scores.std():.3f})")
```

Critical for: - Financial time series - User behavior predictions - Any data with temporal dependencies

1.2.3 Blocked Cross-Validation (Grouped Data)

```
from sklearn.model_selection import GroupKFold

def grouped_cv(X, y, groups, model, n_splits=5):
    """
    Ensure all samples from same group stay together in same fold.
    Critical for: medical patients, user sessions, experimental batches.
    """
    gkf = GroupKFold(n_splits=n_splits)
    scores = []

    for train_idx, test_idx in gkf.split(X, y, groups):
        X_train, X_test = X[train_idx], X[test_idx]
        y_train, y_test = y[train_idx], y[test_idx]

        model.fit(X_train, y_train)
        score = f1_score(y_test, model.predict(X_test))
        scores.append(score)

    return np.array(scores)

# Example: Patient data (multiple samples per patient)
# groups = [1, 1, 1, 2, 2, 3, 3, 3, 4, 4, ...]
# Ensures patient 1's samples all in same fold
```

1.3 Part 2: Custom Business Metrics and Cost-Sensitive Learning

1.3.1 Defining Custom Cost Functions

```
import numpy as np
from sklearn.metrics import make_scorer

def expected_profit_metric(y_true, y_pred, revenue_tp=1000, cost_fp=200,
    cost_fn=5000):
    """
    Calculate expected profit per prediction.

    Example: Loan approval
    - TP: Approve good customer, earn $1000 interest
    - FP: Approve bad customer, lose $200 (default)
    - FN: Reject good customer, lose $5000 opportunity cost
    - TN: Reject bad customer, $0 (no transaction)
    """
    from sklearn.metrics import confusion_matrix
    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()

    profit = (
        tp * revenue_tp +
        tn * 0 +
        fp * (-cost_fp) +
        fn * (-cost_fn)
    )

    return profit / len(y_true)  # Per-sample profit

# Convert to sklearn scorer (higher is better)
profit_scorer = make_scorer(
```

```
    expected_profit_metric,
    greater_is_better=True,
    revenue_tp=1000,
    cost_fp=200,
    cost_fn=5000
)

# Use in cross-validation
from sklearn.ensemble import RandomForestClassifier
scores = cross_val_score(
    RandomForestClassifier(n_estimators=100, random_state=42),
    X, y,
    cv=5,
    scoring=profit_scorer
)
print(f"Expected profit: ${scores.mean():.2f} (+/- ${scores.std():.2f})")
```

1.3.2 Cost-Sensitive Learning with Sample Weights

```
from sklearn.utils.class_weight import compute_sample_weight

# Compute weights proportional to misclassification cost
# If FN costs 10x more than FP, weight positive samples 10x higher
cost_ratio = 10  # Cost_FN / Cost_FP
sample_weights = compute_sample_weight(
    class_weight={0: 1, 1: cost_ratio},
    y=y_train
)

# Train with weighted loss
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train, sample_weight=sample_weights)

# Predictions now biased toward avoiding high-cost errors
y_pred = model.predict(X_test)
```

1.3.3 Threshold Optimization with Bayesian Search

```
from scipy.optimize import minimize_scalar

def expected_cost(threshold, y_true, y_proba, cost_fn, cost_fp):
    """Cost function for threshold optimization"""
    y_pred = (y_proba >= threshold).astype(int)
    cm = confusion_matrix(y_true, y_pred)
    tn, fp, fn, tp = cm.ravel()
    return fn * cost_fn + fp * cost_fp

def optimize_threshold(y_true, y_proba, cost_fn=5000, cost_fp=200):
    """Find optimal threshold using gradient-free optimization"""
    result = minimize_scalar(
        expected_cost,
        bounds=(0.01, 0.99),
        args=(y_true, y_proba, cost_fn, cost_fp),
        method='bounded'
    )
    return result.x, result.fun

# Example
```

```
optimal_threshold, optimal_cost = optimize_threshold(y_test, y_proba)
print(f"Optimal threshold: {optimal_threshold:.3f}")
print(f"Expected cost at optimal: ${optimal_cost:.2f}")
print(f"Expected cost at 0.5: ${expected_cost(0.5, y_test, y_proba, 5000, 200)
    :.2f}")
print(f"Savings: ${expected_cost(0.5, y_test, y_proba, 5000, 200) -
    optimal_cost:.2f}")
```

1.4 Part 3: Multi-Objective Optimization and Pareto Frontiers

1.4.1 Pareto Optimal Model Selection

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

def is_pareto_efficient(costs):
    """
    Find Pareto-optimal points (no other point dominates in ALL objectives).
    costs: (n_models, n_objectives) array
    """
    is_efficient = np.ones(costs.shape[0], dtype=bool)
    for i, c in enumerate(costs):
        if is_efficient[i]:
            # Point i is dominated if another point is better in ALL
    objectives
            is_efficient[is_efficient] = np.any(costs[is_efficient] < c, axis
    =1)
            is_efficient[i] = True
    return is_efficient

# Example: Optimize for both precision and recall
models = {
    'LogReg': LogisticRegression(max_iter=1000),
    'RF': RandomForestClassifier(n_estimators=100, random_state=42),
    'GBM': GradientBoostingClassifier(n_estimators=100, random_state=42),
    'SVM': SVC(probability=True, random_state=42)
}

results = []
for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    results.append({
        'model': name,
        'precision': precision,
        'recall': recall,
        # Negate because we want to MAXIMIZE both (Pareto finds minima)
        'cost_precision': -precision,
        'cost_recall': -recall
    })

df_results = pd.DataFrame(results)
```

```
# Find Pareto frontier
costs = df_results[['cost_precision', 'cost_recall']].values
pareto_mask = is_pareto_efficient(costs)

# Visualize
plt.figure(figsize=(10, 6))
plt.scatter(df_results['precision'], df_results['recall'],
            s=100, alpha=0.6, label='All models')
plt.scatter(df_results.loc[pareto_mask, 'precision'],
            df_results.loc[pareto_mask, 'recall'],
            s=200, c='red', marker='*', label='Pareto optimal')

for i, row in df_results.iterrows():
    plt.annotate(row['model'], (row['precision'], row['recall']),
                 xytext=(5, 5), textcoords='offset points')

plt.xlabel('Precision')
plt.ylabel('Recall')
plt.title('Pareto Frontier: Precision vs Recall')
plt.legend()
plt.grid(alpha=0.3)
plt.show()

print("Pareto-optimal models:")
print(df_results.loc[pareto_mask, ['model', 'precision', 'recall']])
```

Interpretation: Any model on the Pareto frontier is a valid choice depending on business priorities. Models not on the frontier are strictly dominated.

1.4.2 Weighted Multi-Objective Scoring

```
def weighted_multi_objective_score(y_true, y_pred, y_proba, weights):
    """
    Combine multiple objectives with user-specified weights.
    weights: dict like {'precision': 0.6, 'recall': 0.3, 'auc': 0.1}
    """
    metrics = {
        'precision': precision_score(y_true, y_pred),
        'recall': recall_score(y_true, y_pred),
        'f1': f1_score(y_true, y_pred),
        'auc': roc_auc_score(y_true, y_proba)
    }

    score = sum(metrics[k] * weights.get(k, 0) for k in metrics)
    return score

# Example: Business wants 60% precision, 30% recall, 10% AUC
custom_scorer = make_scorer(
    weighted_multi_objective_score,
    needs_proba=True,
    weights={'precision': 0.6, 'recall': 0.3, 'auc': 0.1}
)

scores = cross_val_score(model, X, y, cv=5, scoring=custom_scorer)
print(f"Weighted score: {scores.mean():.3f} (+/- {scores.std():.3f})")
```

1.5 Part 4: Bayesian Model Comparison

1.5.1 Bayesian Hypothesis Testing

```
import pymc3 as pm
import arviz as az

def bayesian_model_comparison(scores_a, scores_b):
    """
    Bayesian test: Is Model A better than Model B?
    Returns posterior probability that A > B.
    """
    with pm.Model() as model:
        # Priors
        mu_a = pm.Normal('mu_a', mu=0.8, sigma=0.1)
        mu_b = pm.Normal('mu_b', mu=0.8, sigma=0.1)
        sigma = pm.HalfNormal('sigma', sigma=0.1)

        # Likelihoods
        pm.Normal('scores_a', mu=mu_a, sigma=sigma, observed=scores_a)
        pm.Normal('scores_b', mu=mu_b, sigma=sigma, observed=scores_b)

        # Difference
        diff = pm.Deterministic('diff', mu_a - mu_b)

        # Sample
        trace = pm.sample(2000, return_inferencedata=True, progressbar=False)

    # Posterior probability that A > B
    prob_a_better = (trace.posterior['diff'].values > 0).mean()

    print(f"P(Model A > Model B): {prob_a_better:.3f}")

    # Visualize
    az.plot_posterior(trace, var_names=['diff'], ref_val=0)
    plt.title('Difference in Performance (A - B)')
    plt.show()

    return prob_a_better

# Example: Compare two models via CV scores
scores_rf = cross_val_score(RandomForestClassifier(), X, y, cv=5, scoring='f1'
    )
scores_gb = cross_val_score(GradientBoostingClassifier(), X, y, cv=5, scoring=
    'f1')

prob_rf_better = bayesian_model_comparison(scores_rf, scores_gb)
```

Advantages over frequentist tests: - Directly quantifies probability one model is better - No p-value misinterpretation - Incorporates prior knowledge - Handles small sample sizes better

1.5.2 Bayesian Optimization for Hyperparameters

```
from skopt import BayesSearchCV
from skopt.space import Real, Integer

# Define search space
search_space = {
    'n_estimators': Integer(50, 300),
    'max_depth': Integer(5, 50),
```

```
    'min_samples_split': Integer(2, 20),
    'min_samples_leaf': Integer(1, 10)
}

# Bayesian optimization (more efficient than grid search)
bayes_search = BayesSearchCV(
    RandomForestClassifier(random_state=42),
    search_space,
    n_iter=50,  # Much fewer than grid search
    cv=5,
    scoring='f1',
    n_jobs=-1,
    random_state=42
)

bayes_search.fit(X_train, y_train)

print(f"Best parameters: {bayes_search.best_params_}")
print(f"Best F1: {bayes_search.best_score_:.3f}")

# Compare to random search efficiency
import matplotlib.pyplot as plt
results = pd.DataFrame(bayes_search.cv_results_)
plt.plot(results['mean_test_score'])
plt.xlabel('Iteration')
plt.ylabel('F1 Score')
plt.title('Bayesian Optimization Convergence')
plt.show()
```

1.6 Part 5: Statistical Power Analysis

1.6.1 Sample Size Calculation

```
from statsmodels.stats.power import tt_ind_solve_power

def calculate_required_samples(effect_size=0.5, alpha=0.05, power=0.8):
    """
    Calculate required sample size for detecting effect.

    effect_size: Cohen's d (0.2=small, 0.5=medium, 0.8=large)
    alpha: Type I error rate (false positive)
    power: 1 - Type II error rate (1 - false negative)
    """
    n = tt_ind_solve_power(
        effect_size=effect_size,
        alpha=alpha,
        power=power,
        alternative='two-sided'
    )
    return int(np.ceil(n))

# Example: Detect 0.05 difference in F1 score
# Standard deviation of F1 scores ~0.05 from CV
effect_size = 0.05 / 0.05  # Mean diff / std
n_required = calculate_required_samples(effect_size=effect_size)
print(f"Required CV folds: {n_required}")
```

1.6.2 Bootstrap Confidence Intervals

```python
from scipy.stats import bootstrap

def bootstrap_ci(y_true, y_pred, metric_func, n_bootstrap=1000, confidence
    =0.95):
    """
    Calculate bootstrap confidence interval for any metric.
    """
    def metric_wrapper(y_true, y_pred):
        # Resample indices
        idx = np.random.choice(len(y_true), size=len(y_true), replace=True)
        return metric_func(y_true[idx], y_pred[idx])

    scores = [metric_wrapper(y_true, y_pred) for _ in range(n_bootstrap)]
    scores = np.array(scores)

    alpha = 1 - confidence
    lower = np.percentile(scores, alpha/2 * 100)
    upper = np.percentile(scores, (1 - alpha/2) * 100)

    return scores.mean(), lower, upper

# Example
mean_f1, lower_ci, upper_ci = bootstrap_ci(
    y_test, y_pred,
    metric_func=f1_score,
    n_bootstrap=1000
)
print(f"F1: {mean_f1:.3f} (95% CI: [{lower_ci:.3f}, {upper_ci:.3f}])")
```

1.7 Part 6: Fairness-Aware Model Selection

1.7.1 Demographic Parity

```python
def demographic_parity(y_true, y_pred, sensitive_feature):
    """
    Measure demographic parity: P(Y_pred=1 | A=0) vs P(Y_pred=1 | A=1)
    where A is sensitive attribute (e.g., gender, race)
    """
    groups = np.unique(sensitive_feature)
    positive_rates = {}

    for group in groups:
        mask = (sensitive_feature == group)
        positive_rate = (y_pred[mask] == 1).mean()
        positive_rates[group] = positive_rate

    # Disparity: max difference
    disparity = max(positive_rates.values()) - min(positive_rates.values())
    return disparity, positive_rates

# Example
# sensitive = [0, 0, 1, 1, 0, 1, ...]  # 0=female, 1=male
disparity, rates = demographic_parity(y_test, y_pred, sensitive_test)
print(f"Positive rate by group: {rates}")
print(f"Disparity: {disparity:.3f} (closer to 0 = more fair)")
```

1.7.2 Equalized Odds

```python
def equalized_odds(y_true, y_pred, sensitive_feature):
    """
    Measure equalized odds: TPR and FPR should be equal across groups.
    """
    from sklearn.metrics import confusion_matrix

    groups = np.unique(sensitive_feature)
    results = {}

    for group in groups:
        mask = (sensitive_feature == group)
        cm = confusion_matrix(y_true[mask], y_pred[mask])
        tn, fp, fn, tp = cm.ravel()

        tpr = tp / (tp + fn) if (tp + fn) > 0 else 0
        fpr = fp / (fp + tn) if (fp + tn) > 0 else 0

        results[group] = {'TPR': tpr, 'FPR': fpr}

    # Calculate max disparity
    tpr_disparity = max(r['TPR'] for r in results.values()) - min(r['TPR'] for
     r in results.values())
    fpr_disparity = max(r['FPR'] for r in results.values()) - min(r['FPR'] for
     r in results.values())

    return results, tpr_disparity, fpr_disparity

# Example
results, tpr_disp, fpr_disp = equalized_odds(y_test, y_pred, sensitive_test)
print(f"Group-wise metrics: {results}")
print(f"TPR disparity: {tpr_disp:.3f}")
print(f"FPR disparity: {fpr_disp:.3f}")
```

1.8 Part 7: Production Deployment Framework

1.8.1 Pre-Deployment Checklist (Code-Based)

```python
class ModelValidator:
    """
    Comprehensive validation before deployment.
    """
    def __init__(self, model, X_test, y_test, thresholds):
        self.model = model
        self.X_test = X_test
        self.y_test = y_test
        self.thresholds = thresholds
        self.checks_passed = {}

    def check_minimum_performance(self):
        """All metrics must exceed thresholds"""
        y_pred = self.model.predict(self.X_test)
        y_proba = self.model.predict_proba(self.X_test)[:, 1]

        metrics = {
            'accuracy': accuracy_score(self.y_test, y_pred),
            'precision': precision_score(self.y_test, y_pred),
```

```
            'recall': recall_score(self.y_test, y_pred),
            'f1': f1_score(self.y_test, y_pred),
            'roc_auc': roc_auc_score(self.y_test, y_proba)
        }

        for metric, value in metrics.items():
            threshold = self.thresholds.get(metric, 0)
            passed = value >= threshold
            self.checks_passed[f'min_{metric}'] = passed
            print(f"{metric}: {value:.3f} (threshold: {threshold:.3f}) {'  '
    if passed else '  '}")

        return all(metrics[k] >= self.thresholds.get(k, 0) for k in metrics)

    def check_class_balance(self):
        """Performance should be reasonable on both classes"""
        report = classification_report(self.y_test, self.model.predict(self.
    X_test),
                                       output_dict=True)

        for class_label in ['0', '1']:
            f1_class = report[class_label]['f1-score']
            passed = f1_class >= 0.6
            self.checks_passed[f'class_{class_label}_f1'] = passed
            print(f"Class {class_label} F1: {f1_class:.3f} {'  ' if passed
    else '  '}")

        return all(report[c]['f1-score'] >= 0.6 for c in ['0', '1'])

    def check_no_data_leakage(self):
        """Train and test metrics shouldn't be too different"""
        # Assume we have train predictions available
        # train_score = ...
        # test_score = ...
        # return abs(train_score - test_score) < 0.1
        pass

    def validate(self):
        """Run all checks"""
        print("=== Pre-Deployment Validation ===\n")
        perf_ok = self.check_minimum_performance()
        balance_ok = self.check_class_balance()

        all_passed = perf_ok and balance_ok
        print(f"\n{'PASSED' if all_passed else 'FAILED'}: Model ready for
    deployment")
        return all_passed

# Usage
thresholds = {
    'accuracy': 0.90,
    'precision': 0.85,
    'recall': 0.75,
    'f1': 0.80,
    'roc_auc': 0.90
}

validator = ModelValidator(model, X_test, y_test, thresholds)
ready_for_deployment = validator.validate()
```

1.8.2 A/B Test Planning

```python
def calculate_ab_test_size(baseline_conversion=0.10, mde=0.02, alpha=0.05,
    power=0.80):
    """
    Calculate required sample size for A/B test.

    baseline_conversion: Current model's positive rate
    mde: Minimum detectable effect (e.g., 0.02 = 2 percentage points)
    alpha: Significance level
    power: Statistical power
    """
    from statsmodels.stats.proportion import proportions_ztest
    from statsmodels.stats.power import zt_ind_solve_power

    effect_size = mde / np.sqrt(baseline_conversion * (1 - baseline_conversion
    ))
    n_per_group = zt_ind_solve_power(
        effect_size=effect_size,
        alpha=alpha,
        power=power,
        alternative='two-sided'
    )

    return int(np.ceil(n_per_group))

# Example: Detect 2% improvement in conversion
n_required = calculate_ab_test_size(baseline_conversion=0.10, mde=0.02)
print(f"Required samples per group: {n_required}")
print(f"Total samples needed: {2 * n_required}")
print(f"At 1000 users/day: {2 * n_required / 1000:.1f} days")
```

1.9 Part 8: Post-Deployment Monitoring

1.9.1 Concept Drift Detection

```python
from scipy.stats import ks_2samp

def detect_feature_drift(X_train, X_production, feature_names, threshold=0.05)
    :
    """
    Detect distribution shift using Kolmogorov-Smirnov test.
    """
    drift_detected = {}

    for i, feature in enumerate(feature_names):
        stat, pvalue = ks_2samp(X_train[:, i], X_production[:, i])
        drifted = pvalue < threshold
        drift_detected[feature] = {
            'statistic': stat,
            'pvalue': pvalue,
            'drifted': drifted
        }

        if drifted:
            print(f"DRIFT DETECTED: {feature} (p={pvalue:.4f})")

    return drift_detected

# Example
```

```
# X_production: New data from last week
drift_results = detect_feature_drift(
    X_train,
    X_production,
    feature_names=['age', 'income', 'credit_score'],
    threshold=0.05
)
```

1.9.2 Performance Degradation Monitoring

```
class PerformanceMonitor:
    """
    Monitor model performance over time and trigger alerts.
    """
    def __init__(self, baseline_metrics, alert_thresholds):
        self.baseline = baseline_metrics
        self.thresholds = alert_thresholds
        self.history = []

    def check_performance(self, y_true, y_pred, timestamp):
        """Check current performance against baseline"""
        current_f1 = f1_score(y_true, y_pred)
        current_precision = precision_score(y_true, y_pred)
        current_recall = recall_score(y_true, y_pred)

        # Calculate degradation
        f1_drop = self.baseline['f1'] - current_f1
        precision_drop = self.baseline['precision'] - current_precision
        recall_drop = self.baseline['recall'] - current_recall

        alerts = []
        if f1_drop > self.thresholds['f1']:
            alerts.append(f"F1 dropped by {f1_drop:.3f}")
        if precision_drop > self.thresholds['precision']:
            alerts.append(f"Precision dropped by {precision_drop:.3f}")
        if recall_drop > self.thresholds['recall']:
            alerts.append(f"Recall dropped by {recall_drop:.3f}")

        # Log
        self.history.append({
            'timestamp': timestamp,
            'f1': current_f1,
            'precision': current_precision,
            'recall': current_recall,
            'alerts': alerts
        })

        return alerts

    def plot_trends(self):
        """Visualize performance over time"""
        df = pd.DataFrame(self.history)
        df.plot(x='timestamp', y=['f1', 'precision', 'recall'],
                figsize=(12, 6), marker='o')
        plt.axhline(self.baseline['f1'], color='r', linestyle='--', label='
    Baseline F1')
        plt.title('Model Performance Over Time')
        plt.ylabel('Score')
        plt.legend()
        plt.show()
```

```
# Example usage
baseline = {'f1': 0.85, 'precision': 0.90, 'recall': 0.80}
thresholds = {'f1': 0.05, 'precision': 0.05, 'recall': 0.05}

monitor = PerformanceMonitor(baseline, thresholds)

# Check performance weekly
for week in range(12):
    # Simulate performance degradation
    y_true_week = ...  # New labels from production
    y_pred_week = ...  # Model predictions
    alerts = monitor.check_performance(y_true_week, y_pred_week, timestamp=
    week)

    if alerts:
        print(f"Week {week} ALERTS: {alerts}")

monitor.plot_trends()
```

1.10 Part 9: Advanced Topics

1.10.1 Conformal Prediction (Prediction Intervals)

```
from sklearn.model_selection import train_test_split

def conformal_prediction(model, X_train, y_train, X_cal, y_cal, X_test, alpha
    =0.1):
    """
    Conformal prediction: Provide prediction sets with guaranteed coverage.

    alpha: Miscoverage rate (0.1 = 90% coverage)
    """
    # Train model
    model.fit(X_train, y_train)

    # Calibration: Compute nonconformity scores
    y_cal_proba = model.predict_proba(X_cal)[:, 1]
    cal_scores = np.abs(y_cal - y_cal_proba)

    # Quantile of calibration scores
    q = np.quantile(cal_scores, 1 - alpha)

    # Prediction intervals
    y_test_proba = model.predict_proba(X_test)[:, 1]
    lower = np.clip(y_test_proba - q, 0, 1)
    upper = np.clip(y_test_proba + q, 0, 1)

    return y_test_proba, lower, upper

# Example
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4)
X_cal, X_test, y_cal, y_test = train_test_split(X_temp, y_temp, test_size=0.5)

proba, lower, upper = conformal_prediction(
    RandomForestClassifier(n_estimators=100),
    X_train, y_train, X_cal, y_cal, X_test,
    alpha=0.1
)
```

```
print(f"Prediction intervals (90% coverage):")
for i in range(5):
    print(f"Sample {i}: [{lower[i]:.3f}, {upper[i]:.3f}], True: {y_test[i]}")
```

1.10.2 Adversarial Validation

```
def adversarial_validation(X_train, X_test):
    """
    Check if train and test distributions are different.
    If AUC > 0.5, distributions differ (potential train-test mismatch).
    """
    # Label train=0, test=1
    y_train_labels = np.zeros(len(X_train))
    y_test_labels = np.ones(len(X_test))

    X_combined = np.vstack([X_train, X_test])
    y_combined = np.concatenate([y_train_labels, y_test_labels])

    # Train model to distinguish train from test
    from sklearn.ensemble import RandomForestClassifier
    model = RandomForestClassifier(n_estimators=100, random_state=42)
    scores = cross_val_score(model, X_combined, y_combined, cv=5, scoring='
    roc_auc')

    print(f"Adversarial Validation AUC: {scores.mean():.3f}")
    if scores.mean() > 0.75:
        print("WARNING: Train and test distributions significantly different!"
    )
    else:
        print("Train and test distributions similar (good)")

    return scores.mean()

# Example
auc = adversarial_validation(X_train, X_test)
```

1.11 Key Takeaways

1. **Nested CV for unbiased hyperparameter tuning**: Single-level CV overestimates performance.
2. **Custom business metrics**: Align evaluation with actual business value, not just ML metrics.
3. **Multi-objective optimization**: Use Pareto frontiers when metrics conflict.
4. **Bayesian methods**: Quantify uncertainty in model comparisons and hyperparameter search.
5. **Fairness metrics**: Ensure equitable performance across demographic groups.
6. **Statistical power**: Calculate required sample sizes before experiments.
7. **Production monitoring**: Detect drift and performance degradation early.
8. **Conformal prediction**: Provide prediction intervals with statistical guarantees.

1.12 Production Checklist

- ☐ Nested CV for hyperparameter tuning
- ☐ Custom business metric defined and validated
- ☐ Multi-metric comparison (Pareto analysis if needed)
- ☐ Statistical significance tests passed
- ☐ Fairness metrics evaluated
- ☐ Pre-deployment validation passed
- ☐ A/B test plan with power analysis
- ☐ Monitoring system for drift detection
- ☐ Performance degradation alerts configured
- ☐ Rollback criteria defined
- ☐ Model card with all validation results
- ☐ Stakeholder sign-off obtained

1.13 Resources

Advanced Cross-Validation: - Cawley, G. & Talbot, N. (2010). "On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation"

Cost-Sensitive Learning: - Elkan, C. (2001). "The Foundations of Cost-Sensitive Learning"

Multi-Objective Optimization: - Deb, K. (2001). "Multi-Objective Optimization Using Evolutionary Algorithms"

Bayesian Methods: - Kruschke, J. (2014). "Doing Bayesian Data Analysis"

Fairness in ML: - Barocas, S., Hardt, M., & Narayanan, A. (2019). "Fairness and Machine Learning"

Production ML: - Huyen, C. (2022). "Designing Machine Learning Systems"