Validation Metrics - Intermediate Handout

Machine Learning for Smarter Innovation

1 Validation Metrics - Intermediate Handout

Target Audience: Practitioners with Python knowledge **Duration**: 60 minutes reading + coding **Level**: Intermediate (implementation focused)

1.1 Setup

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import (
    train_test_split, cross_val_score, cross_validate,
    StratifiedKFold, TimeSeriesSplit, GridSearchCV
)
from sklearn.ensemble import RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score, fbeta_score,
    roc_auc_score, average_precision_score, cohen_kappa_score,
    matthews_corrcoef,
    confusion_matrix, classification_report, roc_curve, precision_recall_curve
    ,
    make_scorer
)
from scipy import stats
import warnings
warnings.filterwarnings('ignore')

# Visualization settings
plt.rcParams.update({
    'font.size': 10,
    'axes.labelsize': 10,
    'figure.figsize': (10, 6)
})
```

This handout covers comprehensive model validation using multiple metrics, cross-validation strategies, and statistical significance testing. Proper validation ensures model performance estimates are reliable and deployment decisions are sound. The techniques apply across classification problems in fraud detection, medical diagnosis, customer churn prediction, and any domain where model performance must be rigorously assessed.

1.2 1. Multi-Metric Classification Evaluation

1.2.1 Concept Overview

No single metric tells the complete story of model performance. Accuracy misleads on imbalanced data, precision and recall trade off against each other, and threshold-dependent metrics behave differently than ranking metrics like AUC. Production evaluation requires computing multiple metrics and understanding their relationships. Each metric answers a different question about model behavior.

1.2.2 Implementation: Complete Metric Suite

```
def evaluate_classifier(y_true, y_pred, y_proba):
    """Compute comprehensive classification metrics."""
    metrics = {
        'Accuracy': accuracy_score(y_true, y_pred),
        'Precision': precision_score(y_true, y_pred),
        'Recall': recall_score(y_true, y_pred),
        'F1': f1_score(y_true, y_pred),
        'F2 (favor recall)': fbeta_score(y_true, y_pred, beta=2),
        'F0.5 (favor precision)': fbeta_score(y_true, y_pred, beta=0.5),
        'ROC-AUC': roc_auc_score(y_true, y_proba),
        'PR-AUC': average_precision_score(y_true, y_proba),
        'Cohen Kappa': cohen_kappa_score(y_true, y_pred),
        'Matthews Corr': matthews_corrcoef(y_true, y_pred)
    }
    return metrics


# Create imbalanced dataset
X, y = make_classification(
    n_samples=10000, n_features=20, n_informative=15,
    n_redundant=5, weights=[0.9, 0.1], random_state=42
)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

# Train and evaluate
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
y_proba = model.predict_proba(X_test)[:, 1]

metrics = evaluate_classifier(y_test, y_pred, y_proba)
print(pd.DataFrame([metrics]).T.round(3))
```

1.2.3 Multi-Class Averaging Strategies

```
def multiclass_metrics(y_true, y_pred):
    """Compute metrics with different averaging strategies."""
    results = {}
    for avg in ['macro', 'micro', 'weighted']:
        results[avg] = {
            'Precision': precision_score(y_true, y_pred, average=avg),
```

```
            'Recall': recall_score(y_true, y_pred, average=avg),
            'F1': f1_score(y_true, y_pred, average=avg)
        }
    return pd.DataFrame(results).T

# Example with multi-class data
y_true_multi = [0, 1, 2, 2, 1, 0, 1, 2, 0]
y_pred_multi = [0, 1, 2, 1, 1, 0, 2, 2, 0]

print("Multi-class metrics:")
print(multiclass_metrics(y_true_multi, y_pred_multi).round(3))
```

Averaging strategies: - **Macro**: Unweighted mean (treats all classes equally) - **Micro**: Aggregate TP/FP/FN globally (favors frequent classes) - **Weighted**: Weighted by class frequency (balances both)

1.3 2. Cross-Validation Strategies

1.3.1 Concept Overview

Single train-test splits provide unstable performance estimates. Cross-validation averages over multiple splits, giving more reliable estimates with confidence intervals. Different CV strategies suit different data types: stratified K-fold maintains class balance, time series split respects temporal ordering, and nested CV prevents test set overfitting during hyperparameter tuning.

1.3.2 Implementation: Comprehensive Cross-Validation

```
def cross_validate_model(model, X, y, cv_strategy='stratified', n_splits=5):
    """Perform cross-validation with multiple metrics."""

    # Choose CV strategy
    if cv_strategy == 'stratified':
        cv = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=42)
    elif cv_strategy == 'timeseries':
        cv = TimeSeriesSplit(n_splits=n_splits)
    else:
        cv = n_splits  # Regular K-fold

    scoring = ['accuracy', 'precision', 'recall', 'f1', 'roc_auc']
    results = cross_validate(model, X, y, cv=cv, scoring=scoring, n_jobs=-1)

    # Organize results
    cv_results = {}
    for metric in scoring:
        scores = results[f'test_{metric}']
        cv_results[metric] = {
            'mean': scores.mean(),
            'std': scores.std(),
            'scores': scores
        }

    return cv_results


def compute_confidence_interval(scores, confidence=0.95):
    """Compute confidence interval from CV scores."""
    n = len(scores)
    mean = scores.mean()
```

```
    std_err = scores.std() / np.sqrt(n)
    ci = stats.t.interval(confidence, n-1, loc=mean, scale=std_err)
    return mean, ci


# Cross-validation
model = RandomForestClassifier(n_estimators=100, random_state=42)
cv_results = cross_validate_model(model, X, y, cv_strategy='stratified',
    n_splits=5)

print("Cross-Validation Results:")
for metric, values in cv_results.items():
    mean, ci = compute_confidence_interval(values['scores'])
    print(f"{metric:12s}: {mean:.3f} (95% CI: [{ci[0]:.3f}, {ci[1]:.3f}])")
```

1.3.3 Time Series Cross-Validation

```
def time_series_validation(model, X, y, n_splits=5):
    """Cross-validation for temporal data."""
    tscv = TimeSeriesSplit(n_splits=n_splits)
    results = []

    for fold, (train_idx, test_idx) in enumerate(tscv.split(X)):
        X_train_cv, X_test_cv = X[train_idx], X[test_idx]
        y_train_cv, y_test_cv = y[train_idx], y[test_idx]

        model.fit(X_train_cv, y_train_cv)
        y_pred = model.predict(X_test_cv)
        y_proba = model.predict_proba(X_test_cv)[:, 1]

        results.append({
            'fold': fold + 1,
            'train_size': len(train_idx),
            'test_size': len(test_idx),
            'f1': f1_score(y_test_cv, y_pred),
            'roc_auc': roc_auc_score(y_test_cv, y_proba)
        })

    return pd.DataFrame(results)


# Note: For time series, no shuffling - train on past, test on future
ts_results = time_series_validation(model, X, y, n_splits=5)
print("\nTime Series CV Results:")
print(ts_results)
```

1.4 3. ROC and Precision-Recall Curves

1.4.1 Concept Overview

ROC curves plot true positive rate against false positive rate across all thresholds, with AUC measuring overall ranking ability. Precision-Recall curves are more informative for imbalanced data, focusing on positive class performance. Comparing curves across models reveals which performs better at different operating points, not just overall.

1.4.2 Implementation: Curve Analysis

```
def plot_model_curves(models, X_train, X_test, y_train, y_test):
    """Plot ROC and PR curves for multiple models."""
    fig, axes = plt.subplots(1, 2, figsize=(14, 5))

    for name, model in models.items():
        model.fit(X_train, y_train)
        y_proba = model.predict_proba(X_test)[:, 1]

        # ROC curve
        fpr, tpr, _ = roc_curve(y_test, y_proba)
        roc_auc = roc_auc_score(y_test, y_proba)
        axes[0].plot(fpr, tpr, lw=2, label=f'{name} (AUC={roc_auc:.2f})')

        # PR curve
        precision, recall, _ = precision_recall_curve(y_test, y_proba)
        pr_auc = average_precision_score(y_test, y_proba)
        axes[1].plot(recall, precision, lw=2, label=f'{name} (AUC={pr_auc:.2f
    })')

    # ROC plot formatting
    axes[0].plot([0, 1], [0, 1], 'k--', lw=2)
    axes[0].set_xlabel('False Positive Rate')
    axes[0].set_ylabel('True Positive Rate')
    axes[0].set_title('ROC Curves')
    axes[0].legend(loc='lower right')
    axes[0].grid(alpha=0.3)

    # PR plot formatting
    baseline = y_test.sum() / len(y_test)
    axes[1].axhline(baseline, color='k', linestyle='--', label=f'Baseline ({
    baseline:.2f})')
    axes[1].set_xlabel('Recall')
    axes[1].set_ylabel('Precision')
    axes[1].set_title('Precision-Recall Curves')
    axes[1].legend(loc='lower left')
    axes[1].grid(alpha=0.3)

    plt.tight_layout()
    plt.savefig('model_curves.pdf', bbox_inches='tight')
    plt.show()


# Compare models
models = {
    'Logistic Regression': LogisticRegression(max_iter=1000),
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42)
    ,
    'Gradient Boosting': GradientBoostingClassifier(n_estimators=100,
    random_state=42)
}

plot_model_curves(models, X_train, X_test, y_train, y_test)
```

When to use which curve: - **ROC**: Balanced datasets, threshold-independent evaluation - **PR**: Imbalanced datasets where positive class is the focus

---

1.5 4. Threshold Optimization

1.5.1 Concept Overview

The default 0.5 classification threshold is arbitrary and often suboptimal. Business contexts have different costs for false positives versus false negatives. Threshold optimization finds the operating point that minimizes total cost or maximizes the relevant metric for your application. This is one of the most impactful yet overlooked model tuning steps.

1.5.2 Implementation: Cost-Sensitive Thresholds

```
class ThresholdOptimizer:
    """Find optimal classification threshold."""

    def __init__(self, y_true, y_proba):
        self.y_true = y_true
        self.y_proba = y_proba

    def optimize_for_cost(self, cost_fn, cost_fp):
        """Find threshold minimizing total cost."""
        thresholds = np.linspace(0.01, 0.99, 99)
        costs = []

        for thresh in thresholds:
            y_pred = (self.y_proba >= thresh).astype(int)
            cm = confusion_matrix(self.y_true, y_pred)
            tn, fp, fn, tp = cm.ravel()
            total_cost = fn * cost_fn + fp * cost_fp
            costs.append(total_cost)

        optimal_idx = np.argmin(costs)
        return thresholds[optimal_idx], costs[optimal_idx], thresholds, costs

    def optimize_for_metric(self, metric='f1'):
        """Find threshold maximizing specified metric."""
        thresholds = np.linspace(0.01, 0.99, 99)
        scores = []

        for thresh in thresholds:
            y_pred = (self.y_proba >= thresh).astype(int)
            if metric == 'f1':
                score = f1_score(self.y_true, y_pred)
            elif metric == 'precision':
                score = precision_score(self.y_true, y_pred)
            elif metric == 'recall':
                score = recall_score(self.y_true, y_pred)
            scores.append(score)

        optimal_idx = np.argmax(scores)
        return thresholds[optimal_idx], scores[optimal_idx], thresholds,
    scores

    def analyze_thresholds(self, thresholds):
        """Show metric values at specific thresholds."""
        results = []
        for thresh in thresholds:
            y_pred = (self.y_proba >= thresh).astype(int)
            results.append({
                'Threshold': thresh,
                'Precision': precision_score(self.y_true, y_pred),
                'Recall': recall_score(self.y_true, y_pred),
```

```
                'F1': f1_score(self.y_true, y_pred)
            })
        return pd.DataFrame(results)


# Example: Credit risk (FN costs $50K, FP costs $5K)
optimizer = ThresholdOptimizer(y_test, y_proba)

optimal_thresh, optimal_cost, thresholds, costs = optimizer.optimize_for_cost(
    cost_fn=50000, cost_fp=5000
)
print(f"Cost-optimal threshold: {optimal_thresh:.2f} (total cost: ${
    optimal_cost:,.0f})")

f1_thresh, f1_score_val, _, _ = optimizer.optimize_for_metric('f1')
print(f"F1-optimal threshold: {f1_thresh:.2f} (F1: {f1_score_val:.3f})")

# Compare thresholds
print("\nMetrics at different thresholds:")
print(optimizer.analyze_thresholds([0.3, 0.4, 0.5, 0.6, 0.7]).round(3))
```

1.6 5. Model Comparison Framework

1.6.1 Concept Overview

Systematic model comparison requires evaluating multiple models on the same data splits with the same metrics. A comparison framework organizes results, identifies the best model per metric, and visualizes tradeoffs. Statistical tests confirm whether observed differences are significant or could arise from random variation.

1.6.2 Implementation: Comparison Pipeline

```
class ModelComparator:
    """Compare multiple models systematically."""

    def __init__(self, models, X_train, X_test, y_train, y_test):
        self.models = models
        self.X_train = X_train
        self.X_test = X_test
        self.y_train = y_train
        self.y_test = y_test
        self.results = []

    def evaluate_all(self):
        """Train and evaluate all models."""
        for name, model in self.models.items():
            model.fit(self.X_train, self.y_train)
            y_pred = model.predict(self.X_test)
            y_proba = model.predict_proba(self.X_test)[:, 1]

            self.results.append({
                'Model': name,
                '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),
```

```
                'PR-AUC': average_precision_score(self.y_test, y_proba)
            })

        return pd.DataFrame(self.results).set_index('Model')

    def get_best_models(self, df):
        """Identify best model for each metric."""
        best = {}
        for col in df.columns:
            best_model = df[col].idxmax()
            best_score = df[col].max()
            best[col] = {'model': best_model, 'score': best_score}
        return best

    def plot_comparison(self, df):
        """Visualize model comparison."""
        import seaborn as sns

        fig, axes = plt.subplots(1, 2, figsize=(14, 5))

        # Heatmap
        sns.heatmap(df, annot=True, fmt='.3f', cmap='RdYlGn',
                    vmin=0, vmax=1, ax=axes[0])
        axes[0].set_title('Model Comparison Heatmap')

        # Bar chart for F1
        df['F1'].plot(kind='bar', ax=axes[1], color='steelblue', alpha=0.8)
        axes[1].set_title('F1 Score Comparison')
        axes[1].set_ylabel('F1 Score')
        axes[1].set_xticklabels(axes[1].get_xticklabels(), rotation=45, ha='
    right')

        plt.tight_layout()
        plt.savefig('model_comparison.pdf', bbox_inches='tight')
        plt.show()


# Run comparison
comparator = ModelComparator(models, X_train, X_test, y_train, y_test)
results_df = comparator.evaluate_all()

print("Model Comparison Results:")
print(results_df.round(3))

print("\nBest model per metric:")
for metric, info in comparator.get_best_models(results_df).items():
    print(f"  {metric}: {info['model']} ({info['score']:.3f})")

comparator.plot_comparison(results_df)
```

1.7 6. Error Analysis

1.7.1 Concept Overview

Understanding where and why models fail is as important as measuring overall performance. Error analysis examines false positives and false negatives separately, looking for patterns in misclassified examples. High-confidence errors are particularly concerning and may indicate data quality issues or model blind spots.

1.7.2 Implementation: Error Deep Dive

```
def analyze_errors(y_true, y_pred, y_proba, X_test, feature_names=None):
    """Detailed analysis of model errors."""

    # Confusion matrix components
    cm = confusion_matrix(y_true, y_pred)
    tn, fp, fn, tp = cm.ravel()

    print("Confusion Matrix Breakdown:")
    print(f"  True Negatives:  {tn:4d} ({tn/len(y_true)*100:5.1f}%)")
    print(f"  False Positives: {fp:4d} ({fp/len(y_true)*100:5.1f}%)")
    print(f"  False Negatives: {fn:4d} ({fn/len(y_true)*100:5.1f}%)")
    print(f"  True Positives:  {tp:4d} ({tp/len(y_true)*100:5.1f}%)")

    # Error indices
    fp_idx = np.where((y_pred == 1) & (y_true == 0))[0]
    fn_idx = np.where((y_pred == 0) & (y_true == 1))[0]

    # Confidence analysis
    fp_confidence = y_proba[fp_idx]
    fn_confidence = 1 - y_proba[fn_idx]  # Confidence in negative prediction

    print(f"\nError Confidence Analysis:")
    print(f"  FP mean confidence: {fp_confidence.mean():.3f} (+/- {
    fp_confidence.std():.3f})")
    print(f"  FN mean confidence: {fn_confidence.mean():.3f} (+/- {
    fn_confidence.std():.3f})")

    # High-confidence errors (most concerning)
    high_conf_fp = np.sum(fp_confidence > 0.8)
    high_conf_fn = np.sum(fn_confidence > 0.8)
    print(f"\nHigh-confidence errors (>0.8):")
    print(f"  False Positives: {high_conf_fp}")
    print(f"  False Negatives: {high_conf_fn}")

    return {'fp_idx': fp_idx, 'fn_idx': fn_idx,
            'fp_confidence': fp_confidence, 'fn_confidence': fn_confidence}


# Analyze errors
error_analysis = analyze_errors(y_test, y_pred, y_proba, X_test)
```

1.8 7. Production Validation Pipeline

1.8.1 Concept Overview

Production validation requires preventing data leakage, using proper pipelines that combine preprocessing with model training, and implementing custom scorers for business metrics. The sklearn Pipeline ensures preprocessing fits only on training data during cross-validation.

1.8.2 Implementation: Complete Pipeline

```
def create_validation_pipeline(model, param_grid):
    """Create sklearn pipeline with grid search."""

    pipeline = Pipeline([
```

```
        ('scaler', StandardScaler()),
        ('classifier', model)
    ])

    grid_search = GridSearchCV(
        pipeline,
        param_grid,
        cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42),
        scoring='f1',
        n_jobs=-1,
        return_train_score=True
    )

    return grid_search


# Custom business scorer
def business_cost_scorer(y_true, y_pred, cost_fn=50000, cost_fp=5000):
    """Custom scorer minimizing business cost."""
    cm = confusion_matrix(y_true, y_pred)
    tn, fp, fn, tp = cm.ravel()
    return -(fn * cost_fn + fp * cost_fp)  # Negative because sklearn
    maximizes


cost_scorer = make_scorer(business_cost_scorer, greater_is_better=True)

# Example with Random Forest
param_grid = {
    'classifier__n_estimators': [50, 100, 200],
    'classifier__max_depth': [10, 20, None]
}

rf_search = create_validation_pipeline(
    RandomForestClassifier(random_state=42),
    param_grid
)

rf_search.fit(X_train, y_train)

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

# Final test evaluation
y_pred_final = rf_search.predict(X_test)
print(f"Test F1: {f1_score(y_test, y_pred_final):.3f}")
```

1.9 Common Metric Selection Guidelines

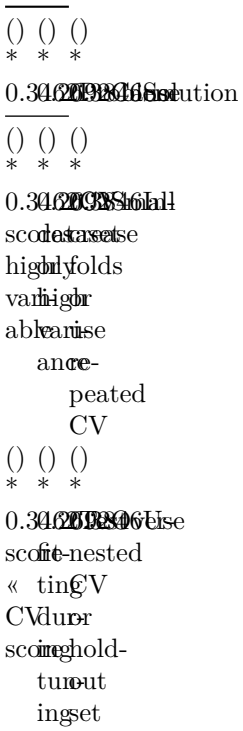
Metric	Use When	Avoid When
Accuracy	Balanced classes	Imbalanced data
Precision	FP costly (spam filter)	FN costly
Recall	FN costly (disease detection)	FP costly
F1	Balance precision/recall	Asymmetric costs
F2	Recall more important	Precision critical
ROC-AUC	Overall ranking quality	Severe imbalance
PR-AUC	Imbalanced data	Balanced classes

Metric	Use When	Avoid When
Cohen Kappa	Agreement vs chance	Continuous targets
Matthews Corr	Binary with imbalance	Multi-class

1.10 Practice Projects

1. **Credit Risk Evaluation**: Build a validation pipeline for loan default prediction. Implement cost-sensitive threshold optimization where false negatives (approved defaults) cost 10x more than false positives (rejected good loans).
2. **Medical Diagnosis Validation**: Evaluate a disease classifier prioritizing recall. Compute PR-AUC, optimize F2 score, and analyze false negative patterns to understand which patient profiles are missed.
3. **Multi-Model Tournament**: Compare 5+ models on a dataset using nested cross-validation. Generate statistical significance tests and a comprehensive comparison report with visualizations.
4. **Drift Detection System**: Build a validation pipeline that monitors model performance over time, detects performance degradation, and triggers retraining alerts.

1.11 Troubleshooting



() *	() *	() *
0.34620993246 [illegible]	[illegible]	[illegible] Solution
() *	() *	() *
[illegible] metrics look good but production fails	[illegible] tribution shift	[illegible] Monitor production metrics
() *	() *	() *
[illegible] high but precision low	[illegible] Imbalance	[illegible] Use PR-AUC instead
() *	() *	() *
[illegible] old optimization unstable	[illegible] Limited positive examples	[illegible] Use CV for threshold selection
() *	() *	() *
[illegible] leakage suspected	[illegible] processing before split	[illegible] Use sklearn Pipeline
() *	() *	() *
[illegible] reproduce results	[illegible] random seed not set	[illegible] Set random_state everywhere

1.12 Next Steps

- Read the advanced handout for Bayesian model selection and multi-objective optimization
- Implement production monitoring dashboards for deployed models
- Explore calibration techniques (Platt scaling, isotonic regression)
- Build automated validation pipelines with MLflow or similar tools

---

Model validation is not a checkbox but a systematic practice. Robust validation combines multiple metrics, proper cross-validation, statistical testing, and error analysis to build confidence that models will perform as expected in production. The effort invested in validation pays dividends in avoiding costly deployment failures.