- Part II: Intermediate Level - Statistical Foundations
    - Chapter 6: Statistical Concepts for Machine Learning
    - Chapter 7: Advanced Evaluation Metrics and Techniques

Part II: Intermediate Level - Statistical Foundations

Chapter 6: Statistical Concepts for Machine Learning

User: I feel like I have a good grasp of the basics now, but I keep hearing about statistical concepts that seem important for machine learning. Things like confidence intervals, correlation coefficients, and statistical tests. Can you help me understand how these connect to what we've learned?

Expert: Absolutely! You're ready to dive into the statistical foundations that make machine learning more rigorous and reliable. These concepts will help you better understand your data, validate your models, and make more confident decisions. Let's start with the fundamentals.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import warnings
warnings.filterwarnings('ignore')
def statistical_foundations_intro():
    """
    Introduction to key statistical concepts for ML
    """
    print("STATISTICAL FOUNDATIONS FOR MACHINE LEARNING")
    print("=" * 50)

    print("Why Statistics Matter in ML:")
    print("✓ Understand data distributions and patterns")
    print("✓ Quantify uncertainty in predictions")
    print("✓ Test hypotheses about model performance")
    print("✓ Make confident business decisions")
    print("✓ Detect and avoid statistical pitfalls")

    return True
statistical_foundations_intro()
```

User: That makes sense! I want to understand how to be more confident about my model's predictions and avoid making wrong conclusions. Where should we start?

Expert: Perfect mindset! Let's start with **Pearson's Correlation Coefficient** - one of the most fundamental statistical measures for understanding relationships between variables.

```python
def pearson_correlation_deep_dive():
    """
    Comprehensive explanation of Pearson's correlation
    """
    print("PEARSON'S CORRELATION COEFFICIENT")
    print("=" * 40)

    # Generate sample data with different correlation strengths
    np.random.seed(42)
    n_samples = 100

    # Perfect positive correlation
```

```
    x1 = np.random.normal(0, 1, n_samples)
    y1 = 2 * x1 + np.random.normal(0, 0.1, n_samples)  # Almost perfect

    # Moderate positive correlation
    x2 = np.random.normal(0, 1, n_samples)
    y2 = x2 + np.random.normal(0, 1, n_samples)

    # No correlation
    x3 = np.random.normal(0, 1, n_samples)
    y3 = np.random.normal(0, 1, n_samples)

    # Negative correlation
    x4 = np.random.normal(0, 1, n_samples)
    y4 = -1.5 * x4 + np.random.normal(0, 0.5, n_samples)

    # Calculate Pearson correlations
    corr1, p_value1 = stats.pearsonr(x1, y1)
    corr2, p_value2 = stats.pearsonr(x2, y2)
    corr3, p_value3 = stats.pearsonr(x3, y3)
    corr4, p_value4 = stats.pearsonr(x4, y4)

    print("CORRELATION COEFFICIENT INTERPRETATION:")
    print("-" * 40)
    print("Range: -1 to +1")
    print("+1: Perfect positive correlation")
    print(" 0: No linear correlation")
    print("-1: Perfect negative correlation")
    print()

    print("STRENGTH GUIDELINES:")
    print("0.0 - 0.3: Weak correlation")
    print("0.3 - 0.7: Moderate correlation")
    print("0.7 - 1.0: Strong correlation")
    print()

    # Visualize different correlations
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))

    # Strong positive
    axes[0, 0].scatter(x1, y1, alpha=0.6, color='red')
    axes[0, 0].set_title(f'Strong Positive\nr = {corr1:.3f}, p = {p_value1:.3f}')
    axes[0, 0].set_xlabel('X')
    axes[0, 0].set_ylabel('Y')
    axes[0, 0].grid(True, alpha=0.3)

    # Moderate positive
    axes[0, 1].scatter(x2, y2, alpha=0.6, color='blue')
    axes[0, 1].set_title(f'Moderate Positive\nr = {corr2:.3f}, p =
```

```
{p_value2:.3f}')
    axes[0, 1].set_xlabel('X')
    axes[0, 1].set_ylabel('Y')
    axes[0, 1].grid(True, alpha=0.3)

    # No correlation
    axes[1, 0].scatter(x3, y3, alpha=0.6, color='green')
    axes[1, 0].set_title(f'No Correlation\nr = {corr3:.3f}, p = {p_value3:.3f}')
    axes[1, 0].set_xlabel('X')
    axes[1, 0].set_ylabel('Y')
    axes[1, 0].grid(True, alpha=0.3)

    # Negative correlation
    axes[1, 1].scatter(x4, y4, alpha=0.6, color='purple')
    axes[1, 1].set_title(f'Strong Negative\nr = {corr4:.3f}, p = {p_value4:.3f}')
    axes[1, 1].set_xlabel('X')
    axes[1, 1].set_ylabel('Y')
    axes[1, 1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    print("BUSINESS INTERPRETATION:")
    print("-" * 25)
    print(f"Strong Positive (r={corr1:.3f}): As X increases, Y increases predictably")
    print(f"Moderate Positive (r={corr2:.3f}): X and Y tend to increase together")
    print(f"No Correlation (r={corr3:.3f}): X and Y are independent")
    print(f"Strong Negative (r={corr4:.3f}): As X increases, Y decreases predictably")

    print("\nP-VALUES EXPLAINED:")
    print("-" * 20)
    print("p < 0.05: Statistically significant correlation")
    print("p ≥ 0.05: Could be due to random chance")

    return corr1, corr2, corr3, corr4
correlations = pearson_correlation_deep_dive()
```

User: This is really helpful! I can see how correlation strength affects the scatter plots. But what about confidence intervals? I hear this term a lot but I'm not sure what it means practically.

Expert: Excellent question! **Confidence intervals** are crucial for understanding the uncertainty in our estimates. Let me show you how they work in the context of machine learning.

```python
def confidence_intervals_explained():
    """
    Comprehensive explanation of confidence intervals
    """
    print("CONFIDENCE INTERVALS IN MACHINE LEARNING")
    print("=" * 45)

    print("WHAT IS A CONFIDENCE INTERVAL?")
    print("-" * 35)
    print("A range of values that likely contains the true parameter")
    print("95% CI means: If we repeated this study 100 times,")
    print("95 of those intervals would contain the true value")
    print()

    # Generate sample data
    np.random.seed(42)
    n_samples = 50
    true_slope = 2.5
    true_intercept = 10

    x = np.random.uniform(1, 10, n_samples)
    noise = np.random.normal(0, 2, n_samples)
    y = true_intercept + true_slope * x + noise

    # Fit linear regression
    from sklearn.linear_model import LinearRegression
    model = LinearRegression()
    X = x.reshape(-1, 1)
    model.fit(X, y)

    # Get predictions
    y_pred = model.predict(X)

    # Calculate confidence intervals manually
    def calculate_prediction_intervals(X, y, model, confidence=0.95):
        """
        Calculate prediction intervals for linear regression
        """
        n = len(y)
        y_pred = model.predict(X)

        # Calculate residuals and standard error
        residuals = y - y_pred
        mse = np.mean(residuals**2)
        se = np.sqrt(mse)

```

```python
        # Degrees of freedom
        dof = n - 2  # n - (number of parameters)

        # t-critical value
        alpha = 1 - confidence
        t_crit = stats.t.ppf(1 - alpha/2, dof)

        # Prediction intervals
        margin_error = t_crit * se * np.sqrt(1 + 1/n + (X.flatten() - np.mean(X.flatten()))**2 / np.sum((X.flatten() - np.mean(X.flatten()))**2))

        lower_bound = y_pred - margin_error
        upper_bound = y_pred + margin_error

        return lower_bound, upper_bound, se, t_crit

    lower_bound, upper_bound, se, t_crit = calculate_prediction_intervals(X, y, model)

    print("CONFIDENCE INTERVAL CALCULATION:")
    print("-" * 35)
    print(f"Sample size: {n_samples}")
    print(f"Degrees of freedom: {n_samples - 2}")
    print(f"Standard error: {se:.3f}")
    print(f"t-critical value (95% CI): {t_crit:.3f}")
    print(f"True slope: {true_slope:.2f}")
    print(f"Estimated slope: {model.coef_[0]:.2f}")
    print(f"True intercept: {true_intercept:.2f}")
    print(f"Estimated intercept: {model.intercept_:.2f}")

    # Visualize confidence intervals
    plt.figure(figsize=(12, 8))

    # Main plot with confidence intervals
    plt.subplot(2, 2, 1)
    plt.scatter(x, y, alpha=0.6, color='blue', label='Data')
    plt.plot(x, y_pred, 'r-', label='Regression Line')
    plt.fill_between(x, lower_bound, upper_bound, alpha=0.3, color='red', label='95% Prediction Interval')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Linear Regression with 95% Prediction Intervals')
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Demonstrate multiple samples
    plt.subplot(2, 2, 2)

    # Simulate multiple experiments
```

```
    slopes = []
    intercepts = []

    for i in range(100):
        # Generate new sample each time
        x_sim = np.random.uniform(1, 10, n_samples)
        y_sim = true_intercept + true_slope * x_sim + np.random.normal(0, 2, n_samples)

        # Fit model
        model_sim = LinearRegression()
        model_sim.fit(x_sim.reshape(-1, 1), y_sim)

        slopes.append(model_sim.coef_[0])
        intercepts.append(model_sim.intercept_)

    plt.hist(slopes, bins=20, alpha=0.7, color='green')
    plt.axvline(true_slope, color='red', linestyle='--', linewidth=2, label=f'True Slope ({true_slope})')
    plt.axvline(np.mean(slopes), color='blue', linestyle='-', linewidth=2, label=f'Mean Estimate ({np.mean(slopes):.2f})')
    plt.xlabel('Estimated Slope')
    plt.ylabel('Frequency')
    plt.title('Distribution of Slope Estimates (100 experiments)')
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Calculate confidence interval for slope
    slope_ci_lower = np.percentile(slopes, 2.5)
    slope_ci_upper = np.percentile(slopes, 97.5)

    plt.axvline(slope_ci_lower, color='orange', linestyle=':', alpha=0.7)
    plt.axvline(slope_ci_upper, color='orange', linestyle=':', alpha=0.7)

    # Business interpretation
    plt.subplot(2, 1, 2)
    plt.axis('off')

    interpretation_text = f"""
BUSINESS INTERPRETATION OF CONFIDENCE INTERVALS:
✓ SLOPE ESTIMATE: {model.coef_[0]:.2f}
✓ 95% CI for slope: [{slope_ci_lower:.2f}, {slope_ci_upper:.2f}]
WHAT THIS MEANS:
• We're 95% confident the true relationship between X and Y
  has a slope between {slope_ci_lower:.2f} and {slope_ci_upper:.2f}
• For every 1-unit increase in X, Y increases by approximately
  {model.coef_[0]:.2f} units (±{(slope_ci_upper-slope_ci_lower)/2:.2f})
```

```
BUSINESS DECISIONS:
• If planning based on this relationship, account for uncertainty
• Wider intervals = more uncertainty = more conservative planning
• Narrower intervals = more confidence = can make bolder decisions
SAMPLE SIZE EFFECT:
• Larger samples → Narrower confidence intervals
• More data → More confident estimates
    """

    plt.text(0.05, 0.95, interpretation_text, transform=plt.gca().transAxes,
             fontsize=10, verticalalignment='top', fontfamily='monospace',
             bbox=dict(boxstyle='round', facecolor='lightblue', alpha=0.8))

    plt.tight_layout()
    plt.show()

    print("PRACTICAL APPLICATIONS:")
    print("-" * 25)
    print("• Revenue forecasting: 'Sales will be $100K ± $15K'")
    print("• A/B testing: 'Treatment effect is 5% ± 2%'")
    print("• Model performance: 'Accuracy is 85% ± 3%'")
    print("• Risk assessment: 'Default rate is 2% ± 0.5%'")

    return slopes, intercepts, slope_ci_lower, slope_ci_upper
slopes, intercepts, ci_lower, ci_upper = confidence_intervals_explained()
```

User: This is incredibly insightful! I love how you showed that confidence intervals help us understand the uncertainty in our estimates. Now I'm curious about statistical tests. When would I use something like a t-test in machine learning?

Expert: Great question! Statistical tests are essential for making data-driven decisions and validating our models. Let me show you the **t-test** and other important statistical tests in ML contexts.

```
def statistical_tests_for_ml():
    """
    Statistical tests commonly used in machine learning
    """
    print("STATISTICAL TESTS IN MACHINE LEARNING")
    print("=" * 40)

    print("WHY WE NEED STATISTICAL TESTS:")
    print("-" * 35)
```

```python
    print("• Validate model improvements")
    print("• Compare different algorithms")
    print("• Test feature importance")
    print("• Validate A/B test results")
    print("• Detect significant differences")

    # Generate sample data for demonstrations
    np.random.seed(42)

    print("\n1. ONE-SAMPLE T-TEST")
    print("-" * 20)
    print("Question: Is our model's accuracy significantly different from a baseline?")

    # Simulate model accuracy scores from cross-validation
    baseline_accuracy = 0.75
    model_accuracies = np.random.normal(0.82, 0.05, 30)  # 30 CV folds

    # Perform one-sample t-test
    t_stat, p_value = stats.ttest_1samp(model_accuracies, baseline_accuracy)

    print(f"Baseline accuracy: {baseline_accuracy:.1%}")
    print(f"Our model accuracy: {np.mean(model_accuracies):.1%} ± {np.std(model_accuracies):.1%}")
    print(f"t-statistic: {t_stat:.3f}")
    print(f"p-value: {p_value:.6f}")

    if p_value < 0.05:
        print("✅ RESULT: Significant improvement over baseline!")
    else:
        print("❌ RESULT: No significant improvement over baseline")

    print("\n2. TWO-SAMPLE T-TEST")
    print("-" * 20)
    print("Question: Is Model A significantly better than Model B?")

    # Simulate performance of two models
    model_a_scores = np.random.normal(0.85, 0.04, 25)
    model_b_scores = np.random.normal(0.82, 0.045, 25)

    # Perform independent t-test
    t_stat_2, p_value_2 = stats.ttest_ind(model_a_scores, model_b_scores)

    print(f"Model A accuracy: {np.mean(model_a_scores):.1%} ± {np.std(model_a_scores):.1%}")
```

```python
    print(f"Model B accuracy: {np.mean(model_b_scores):.1%} ± {np.std(model_b_scores):.1%}")
    print(f"Difference: {np.mean(model_a_scores) - np.mean(model_b_scores):.1%}")
    print(f"t-statistic: {t_stat_2:.3f}")
    print(f"p-value: {p_value_2:.6f}")

    if p_value_2 < 0.05:
        print("✅ RESULT: Model A is significantly better than Model B!")
    else:
        print("❌ RESULT: No significant difference between models")

    print("\n3. PAIRED T-TEST")
    print("-" * 16)
    print("Question: Does feature engineering significantly improve performance?")

    # Simulate before/after feature engineering on same datasets
    before_scores = np.random.normal(0.78, 0.06, 20)
    after_scores = before_scores + np.random.normal(0.05, 0.02, 20) # Improvement

    # Perform paired t-test
    t_stat_paired, p_value_paired = stats.ttest_rel(after_scores, before_scores)

    print(f"Before feature engineering: {np.mean(before_scores):.1%} ± {np.std(before_scores):.1%}")
    print(f"After feature engineering: {np.mean(after_scores):.1%} ± {np.std(after_scores):.1%}")
    print(f"Average improvement: {np.mean(after_scores - before_scores):.1%}")
    print(f"t-statistic: {t_stat_paired:.3f}")
    print(f"p-value: {p_value_paired:.6f}")

    if p_value_paired < 0.05:
        print("✅ RESULT: Feature engineering significantly improves performance!")
    else:
        print("❌ RESULT: No significant improvement from feature engineering")

    # Visualize the tests
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))

    # One-sample t-test visualization
    axes[0, 0].hist(model_accuracies, bins=15, alpha=0.7, color='blue', density=True)
```

```python
    axes[0, 0].axvline(baseline_accuracy, color='red', linestyle='--', linewidth=2, label=f'Baseline ({baseline_accuracy:.1%})')
    axes[0, 0].axvline(np.mean(model_accuracies), color='green', linestyle='-', linewidth=2, label=f'Our Model ({np.mean(model_accuracies):.1%})')
    axes[0, 0].set_xlabel('Accuracy')
    axes[0, 0].set_ylabel('Density')
    axes[0, 0].set_title('One-Sample t-test: Model vs Baseline')
    axes[0, 0].legend()
    axes[0, 0].grid(True, alpha=0.3)

    # Two-sample t-test visualization
    axes[0, 1].hist(model_a_scores, bins=15, alpha=0.7, color='blue', label='Model A', density=True)
    axes[0, 1].hist(model_b_scores, bins=15, alpha=0.7, color='red', label='Model B', density=True)
    axes[0, 1].set_xlabel('Accuracy')
    axes[0, 1].set_ylabel('Density')
    axes[0, 1].set_title('Two-Sample t-test: Model A vs Model B')
    axes[0, 1].legend()
    axes[0, 1].grid(True, alpha=0.3)

    # Paired t-test visualization
    axes[1, 0].scatter(before_scores, after_scores, alpha=0.7, color='purple')
    axes[1, 0].plot([0.6, 0.9], [0.6, 0.9], 'k--', alpha=0.5, label='No improvement line')
    axes[1, 0].set_xlabel('Before Feature Engineering')
    axes[1, 0].set_ylabel('After Feature Engineering')
    axes[1, 0].set_title('Paired t-test: Before vs After')
    axes[1, 0].legend()
    axes[1, 0].grid(True, alpha=0.3)

    # Effect sizes and practical significance
    axes[1, 1].axis('off')

    # Calculate effect sizes (Cohen's d)
    def cohens_d(x1, x2):
        pooled_std = np.sqrt(((len(x1) - 1) * np.var(x1, ddof=1) + (len(x2) - 1) * np.var(x2, ddof=1)) / (len(x1) + len(x2) - 2))
        return (np.mean(x1) - np.mean(x2)) / pooled_std

    effect_size_ab = cohens_d(model_a_scores, model_b_scores)
    effect_size_paired = np.mean(after_scores - before_scores) / np.std(after_scores - before_scores)

    effect_text = f"""
EFFECT SIZES & PRACTICAL SIGNIFICANCE:

Cohen's d interpretation:
• Small effect: d = 0.2
```

```
• Medium effect: d = 0.5
• Large effect: d = 0.8
Model A vs B: d = {effect_size_ab:.2f}
Before vs After: d = {effect_size_paired:.2f}
REMEMBER:
✓ Statistical significance ≠ Practical significance
✓ Large samples can make tiny differences "significant"
✓ Always consider business impact
✓ Effect size matters more than p-value for decisions
BUSINESS DECISION FRAMEWORK:
1. Is the difference statistically significant? (p < 0.05)
2. Is the effect size meaningful? (d > 0.2)
3. Is the business impact worth the cost?
    """

    axes[1, 1].text(0.05, 0.95, effect_text, transform=axes[1, 1].transAxes,
                    fontsize=9, verticalalignment='top', fontfamily='monospace',
                    bbox=dict(boxstyle='round', facecolor='lightyellow', alpha=0.8))

    plt.tight_layout()
    plt.show()

    return {
        'one_sample': (t_stat, p_value),
        'two_sample': (t_stat_2, p_value_2),
        'paired': (t_stat_paired, p_value_paired),
        'effect_sizes': (effect_size_ab, effect_size_paired)
    }
test_results = statistical_tests_for_ml()
```

User: This is fantastic! I can see how statistical tests help us make confident decisions about model improvements. But I'm also curious about the error metrics we use. I know about MSE and RMSE, but can you explain the differences and when to use each one?

Expert: Excellent question! Understanding different error metrics is crucial for choosing the right evaluation approach. Let me show you **MSE, RMSE, and MAE** with their practical implications.

```
def error_metrics_comprehensive():
    """
    Comprehensive explanation of regression error metrics
    """
    print("REGRESSION ERROR METRICS: MSE, RMSE, MAE")
    print("=" * 45)

```

```python
    # Generate sample data with different error patterns
    np.random.seed(42)
    n_samples = 100

    # True values
    y_true = np.random.uniform(10, 100, n_samples)

    # Different prediction scenarios
    # Scenario 1: Good predictions with small errors
    y_pred_good = y_true + np.random.normal(0, 5, n_samples)

    # Scenario 2: Predictions with some outliers
    y_pred_outliers = y_true + np.random.normal(0, 3, n_samples)
    outlier_indices = np.random.choice(n_samples, 5, replace=False)
    y_pred_outliers[outlier_indices] += np.random.choice([-30, 30], 5)

    # Scenario 3: Consistently biased predictions
    y_pred_biased = y_true * 1.2 + np.random.normal(0, 2, n_samples)

    def calculate_all_metrics(y_true, y_pred):
        """Calculate all error metrics"""
        mse = np.mean((y_true - y_pred) ** 2)
        rmse = np.sqrt(mse)
        mae = np.mean(np.abs(y_true - y_pred))

        # Additional metrics
        mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100
        max_error = np.max(np.abs(y_true - y_pred))

        return {
            'MSE': mse,
            'RMSE': rmse,
            'MAE': mae,
            'MAPE': mape,
            'Max_Error': max_error
        }

    # Calculate metrics for all scenarios
    metrics_good = calculate_all_metrics(y_true, y_pred_good)
    metrics_outliers = calculate_all_metrics(y_true, y_pred_outliers)
    metrics_biased = calculate_all_metrics(y_true, y_pred_biased)

    print("METRIC DEFINITIONS:")
    print("-" * 20)
    print("MSE (Mean Squared Error): Average of squared differences")
    print("  • Formula: (1/n) Σ(actual - predicted)²")
```

```
    print("  • Units: Original units squared")
    print("  • Heavily penalizes large errors")
    print()
    print("RMSE (Root Mean Squared Error): Square root of MSE")
    print("  • Formula: √MSE")
    print("  • Units: Same as original data")
    print("  • Interpretable scale, penalizes large errors")
    print()
    print("MAE (Mean Absolute Error): Average of absolute differences")
    print("  • Formula: (1/n) Σ|actual - predicted|")
    print("  • Units: Same as original data")
    print("  • Robust to outliers")

    # Create comparison table
    print(f"\nMETRIC COMPARISON:")
    print("-" * 18)
    print(f"{'Scenario':<15} {'MSE':<8} {'RMSE':<8} {'MAE':<8} {'MAPE':<8} {'Max_Err':<8}")
    print("-" * 65)
    print(f"{'Good Model':<15} {metrics_good['MSE']:<8.1f} {metrics_good['RMSE']:<8.1f} {metrics_good['MAE']:<8.1f} {metrics_good['MAPE']:<8.1f}% {metrics_good['Max_Error']:<8.1f}")
    print(f"{'With Outliers':<15} {metrics_outliers['MSE']:<8.1f} {metrics_outliers['RMSE']:<8.1f} {metrics_outliers['MAE']:<8.1f} {metrics_outliers['MAPE']:<8.1f}% {metrics_outliers['Max_Error']:<8.1f}")
    print(f"{'Biased Model':<15} {metrics_biased['MSE']:<8.1f} {metrics_biased['RMSE']:<8.1f} {metrics_biased['MAE']:<8.1f} {metrics_biased['MAPE']:<8.1f}% {metrics_biased['Max_Error']:<8.1f}")

    # Visualize the differences
    fig, axes = plt.subplots(2, 3, figsize=(18, 12))

    scenarios = [
        (y_pred_good, 'Good Model', 'green'),
        (y_pred_outliers, 'With Outliers', 'red'),
        (y_pred_biased, 'Biased Model', 'blue')
    ]

    for i, (y_pred, title, color) in enumerate(scenarios):
        # Scatter plot
        axes[0, i].scatter(y_true, y_pred, alpha=0.6, color=color)
        axes[0, i].plot([y_true.min(), y_true.max()], [y_true.min(), y_true.max()], 'k--', alpha=0.7)
        axes[0, i].set_xlabel('True Values')
        axes[0, i].set_ylabel('Predicted Values')
        axes[0, i].set_title(f'{title}\nRMSE: {np.sqrt(np.mean((y_true - y_pred)**2)):.1f}, MAE:
```

````python
{np.mean(np.abs(y_true - y_pred)):.1f}')
        axes[0, i].grid(True, alpha=0.3)

        # Residual plot
        residuals = y_true - y_pred
        axes[1, i].scatter(y_pred, residuals, alpha=0.6, color=color)
        axes[1, i].axhline(y=0, color='k', linestyle='--', alpha=0.7)
        axes[1, i].set_xlabel('Predicted Values')
        axes[1, i].set_ylabel('Residuals')
        axes[1, i].set_title(f'Residuals - {title}')
        axes[1, i].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    print("\nWHEN TO USE EACH METRIC:")
    print("-" * 27)
    print("USE MSE WHEN:")
    print("  ✓ You want to heavily penalize large errors")
    print("  ✓ Optimizing model parameters (differentiable)")
    print("  ✓ Mathematical convenience in algorithms")
    print()
    print("USE RM
<user>: continue
<Claude Sonnet 4>: ```python
    print("USE RMSE WHEN:")
    print("  ✓ You want interpretable error in original units")
    print("  ✓ Large errors are more costly than small ones")
    print("  ✓ Reporting to stakeholders (easier to understand)")
    print("  ✓ Comparing models on the same scale as your data")
    print()
    print("USE MAE WHEN:")
    print("  ✓ You want equal weight for all errors")
    print("  ✓ Data has outliers that shouldn't dominate")
    print("  ✓ Robust evaluation is more important")
    print("  ✓ All errors have similar business impact")

    # Demonstrate sensitivity to outliers
    print(f"\nOUTLIER SENSITIVITY ANALYSIS:")
    print("-" * 32)

    # Compare metrics with and without outliers
    y_clean = y_true[:95]  # Remove outlier points
    y_pred_clean = y_pred_outliers[:95]
    y_pred_with_outliers = y_pred_outliers

    metrics_clean = calculate_all_metrics(y_clean, y_pred_clean)
    metrics_with_outliers = calculate_all_metrics(y_true,
````

```
    y_pred_with_outliers)

    print("Impact of 5 outliers on metrics:")
    print(f"RMSE: {metrics_clean['RMSE']:.1f} → {metrics_with_outliers['RMSE']:.1f} ({((metrics_with_outliers['RMSE']/metrics_clean['RMSE'])-1)*100:+.0f}%)")
    print(f"MAE:  {metrics_clean['MAE']:.1f} → {metrics_with_outliers['MAE']:.1f} ({((metrics_with_outliers['MAE']/metrics_clean['MAE'])-1)*100:+.0f}%)")
    print("→ RMSE is more sensitive to outliers than MAE")

    return {
        'good': metrics_good,
        'outliers': metrics_outliers,
        'biased': metrics_biased
    }
error_metrics = error_metrics_comprehensive()
```

User: This comparison really helps me understand when to use each metric! Now I'm curious about something I keep hearing about - sensitivity and recall. Are these the same thing? And how do they relate to the precision and recall we learned about earlier?

Expert: Great question! **Sensitivity and recall are actually the same thing** - just different names used in different contexts. Let me clarify this and show you the complete picture of classification metrics.

```
def sensitivity_recall_comprehensive():
    """
    Comprehensive explanation of sensitivity, recall, and related metrics
    """
    print("SENSITIVITY, RECALL, AND CLASSIFICATION METRICS")
    print("=" * 50)

    print("TERMINOLOGY CLARIFICATION:")
    print("-" * 28)
    print("SENSITIVITY = RECALL = TRUE POSITIVE RATE")
    print("• Medical/Scientific context: 'Sensitivity'")
    print("• Machine Learning context: 'Recall'")
    print("• Signal Detection context: 'True Positive Rate'")
    print("• Same formula: TP / (TP + FN)")
    print()

    # Generate sample medical diagnosis data
    np.random.seed(42)
    n_patients = 1000

```

```python
    # True disease status (1 = has disease, 0 = healthy)
    true_disease = np.random.choice([0, 1], n_patients, p=[0.9, 0.1])  # 10% have disease

    # Simulate test results with realistic performance
    predicted_disease = []
    for i in range(n_patients):
        if true_disease[i] == 1:  # Actually has disease
            # 85% chance test detects it (sensitivity = 0.85)
            pred = np.random.choice([0, 1], p=[0.15, 0.85])
        else:  # Actually healthy
            # 90% chance test correctly identifies healthy (specificity = 0.90)
            pred = np.random.choice([0, 1], p=[0.90, 0.10])
        predicted_disease.append(pred)

    predicted_disease = np.array(predicted_disease)

    # Calculate confusion matrix
    from sklearn.metrics import confusion_matrix, classification_report
    cm = confusion_matrix(true_disease, predicted_disease)
    tn, fp, fn, tp = cm.ravel()

    # Calculate all metrics
    sensitivity_recall = tp / (tp + fn)  # Same as recall
    specificity = tn / (tn + fp)
    precision_ppv = tp / (tp + fp)  # Positive Predictive Value
    npv = tn / (tn + fn)  # Negative Predictive Value
    accuracy = (tp + tn) / (tp + tn + fp + fn)
    f1_score = 2 * (precision_ppv * sensitivity_recall) / (precision_ppv + sensitivity_recall)

    print("COMPLETE METRICS BREAKDOWN:")
    print("-" * 30)
    print(f"True Positives (TP):  {tp:3d} - Correctly identified disease")
    print(f"True Negatives (TN):  {tn:3d} - Correctly identified healthy")
    print(f"False Positives (FP): {fp:3d} - Incorrectly flagged as disease")
    print(f"False Negatives (FN): {fn:3d} - Missed disease cases")
    print()

    print("SENSITIVITY/RECALL FAMILY:")
    print("-" * 28)
    print(f"Sensitivity/Recall:     {sensitivity_recall:.3f} - Of diseased patients, % correctly identified")
    print(f"Specificity:            {specificity:.3f} - Of healthy patients, % correctly identified")
```

```python
    print(f"Precision/PPV:          {precision_ppv:.3f} - Of positive tests, % actually have disease")
    print(f"Negative Pred. Value:   {npv:.3f} - Of negative tests, % actually healthy")
    print(f"Accuracy:               {accuracy:.3f} - Overall correct predictions")
    print(f"F1-Score:               {f1_score:.3f} - Harmonic mean of precision & recall")

    # Visualize the confusion matrix and metrics
    fig, axes = plt.subplots(2, 3, figsize=(18, 12))

    # Confusion Matrix
    im = axes[0, 0].imshow(cm, interpolation='nearest', cmap='Blues')
    axes[0, 0].set_title('Confusion Matrix\n(Medical Test Example)')

    # Add text annotations
    thresh = cm.max() / 2.
    for i in range(2):
        for j in range(2):
            axes[0, 0].text(j, i, f'{cm[i, j]}\n({cm[i, j]/n_patients:.1%})',
                           ha="center", va="center",
                           color="white" if cm[i, j] > thresh else "black",
                           fontsize=12, fontweight='bold')

    axes[0, 0].set_ylabel('True Label')
    axes[0, 0].set_xlabel('Predicted Label')
    axes[0, 0].set_xticks([0, 1])
    axes[0, 0].set_xticklabels(['Healthy', 'Disease'])
    axes[0, 0].set_yticks([0, 1])
    axes[0, 0].set_yticklabels(['Healthy', 'Disease'])

    # Sensitivity vs Specificity Trade-off
    # Simulate different threshold values
    thresholds = np.linspace(0, 1, 100)
    sensitivities = []
    specificities = []

    for threshold in thresholds:
        # Simulate how changing threshold affects predictions
        sens = 1 - threshold * 0.8  # As threshold increases, sensitivity decreases
        spec = threshold * 0.9      # As threshold increases, specificity increases
        sensitivities.append(max(0, min(1, sens)))
        specificities.append(max(0, min(1, spec)))
```

```python

    axes[0, 1].plot(1 - np.array(specificities), sensitivities, 'b-', linewidth=2)
    axes[0, 1].plot([0, 1], [0, 1], 'k--', alpha=0.5)
    axes[0, 1].scatter(1 - specificity, sensitivity_recall, color='red', s=100, zorder=5)
    axes[0, 1].set_xlabel('1 - Specificity (False Positive Rate)')
    axes[0, 1].set_ylabel('Sensitivity (True Positive Rate)')
    axes[0, 1].set_title('ROC Curve\n(Receiver Operating Characteristic)')
    axes[0, 1].grid(True, alpha=0.3)
    axes[0, 1].annotate(f'Current Test\n(Sens={sensitivity_recall:.2f}, Spec={specificity:.2f})',
                        xy=(1-specificity, sensitivity_recall), xytext=(0.6, 0.3),
                        arrowprops=dict(arrowstyle='->', color='red'))

    # Precision-Recall Trade-off
    precisions = []
    recalls = []

    for threshold in thresholds:
        # Simulate precision-recall trade-off
        recall = 1 - threshold * 0.8
        precision = 0.3 + threshold * 0.6  # Higher threshold → higher precision
        recalls.append(max(0, min(1, recall)))
        precisions.append(max(0, min(1, precision)))

    axes[0, 2].plot(recalls, precisions, 'g-', linewidth=2)
    axes[0, 2].scatter(sensitivity_recall, precision_ppv, color='red', s=100, zorder=5)
    axes[0, 2].set_xlabel('Recall (Sensitivity)')
    axes[0, 2].set_ylabel('Precision (PPV)')
    axes[0, 2].set_title('Precision-Recall Curve')
    axes[0, 2].grid(True, alpha=0.3)
    axes[0, 2].annotate(f'Current Test\n(Prec={precision_ppv:.2f}, Rec={sensitivity_recall:.2f})',
                        xy=(sensitivity_recall, precision_ppv), xytext=(0.3, 0.8),
                        arrowprops=dict(arrowstyle='->', color='red'))

    # Context-specific interpretations
    contexts = [
        ('Medical Screening', 'High Sensitivity', 'Don\'t miss any diseases'),
        ('Spam Detection', 'High Precision', 'Don\'t block important emails'),
```

```python
        ('Fraud Detection', 'Balanced F1', 'Balance catching fraud vs false alarms')
    ]

    context_colors = ['red', 'blue', 'green']

    axes[1, 0].axis('off')
    axes[1, 0].set_title('Context-Specific Priorities', fontweight='bold')

    for i, (context, priority, reason) in enumerate(contexts):
        y_pos = 0.8 - i * 0.25
        axes[1, 0].text(0.1, y_pos, f"{context}:", fontweight='bold', fontsize=12, color=context_colors[i])
        axes[1, 0].text(0.1, y_pos-0.05, f"Priority: {priority}", fontsize=10)
        axes[1, 0].text(0.1, y_pos-0.10, f"Reason: {reason}", fontsize=10, style='italic')

    # Metrics comparison chart
    metrics_names = ['Sensitivity\n(Recall)', 'Specificity', 'Precision\n(PPV)', 'NPV', 'Accuracy', 'F1-Score']
    metrics_values = [sensitivity_recall, specificity, precision_ppv, npv, accuracy, f1_score]

    bars = axes[1, 1].bar(metrics_names, metrics_values,
                          color=['red', 'blue', 'green', 'orange', 'purple', 'brown'],
                          alpha=0.7)
    axes[1, 1].set_ylabel('Score')
    axes[1, 1].set_title('All Classification Metrics')
    axes[1, 1].set_ylim(0, 1)
    axes[1, 1].tick_params(axis='x', rotation=45)
    axes[1, 1].grid(True, alpha=0.3)

    # Add value labels on bars
    for bar, value in zip(bars, metrics_values):
        axes[1, 1].text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
                        f'{value:.3f}', ha='center', va='bottom', fontweight='bold')

    # Business decision framework
    axes[1, 2].axis('off')

    decision_text = f"""
BUSINESS DECISION FRAMEWORK:

MEDICAL DIAGNOSIS:
✓ High Sensitivity (Recall): {sensitivity_recall:.1%}
  → Catch most disease cases
```

```
✓ Accept lower Specificity: {specificity:.1%}
  → Some false alarms OK

SPAM DETECTION:
✓ High Precision: Minimize false positives
✓ Accept lower Recall: Some spam gets through
  → Better than blocking important emails

FRAUD DETECTION:
✓ Balance Sensitivity & Precision
✓ Use F1-Score: {f1_score:.1%}
  → Optimize overall performance

KEY INSIGHT:
The "best" metric depends on the cost of
different types of errors in your specific
business context.
    """

    axes[1, 2].text(0.05, 0.95, decision_text, transform=axes[1, 2].transAxes,
                    fontsize=10, verticalalignment='top', fontfamily='monospace',
                    bbox=dict(boxstyle='round', facecolor='lightcyan', alpha=0.8))

    plt.tight_layout()
    plt.show()

    print("\nPRACTICAL GUIDELINES:")
    print("-" * 22)
    print("PRIORITIZE SENSITIVITY/RECALL WHEN:")
    print("  • Missing positives is very costly (medical diagnosis)")
    print("  • You want to catch as many cases as possible")
    print("  • False negatives are worse than false positives")
    print()
    print("PRIORITIZE SPECIFICITY WHEN:")
    print("  • False positives are very costly")
    print("  • You need to be sure about negative cases")
    print("  • Resources for follow-up are limited")
    print()
    print("PRIORITIZE PRECISION WHEN:")
    print("  • Acting on positive predictions is expensive")
    print("  • You need high confidence in positive predictions")
    print("  • False alarms cause significant problems")

    return {
        'confusion_matrix': cm,
        'sensitivity': sensitivity_recall,
        'specificity': specificity,
        'precision': precision_ppv,
        'npv': npv,
```

```
        'accuracy': accuracy,
        'f1_score': f1_score
    }
classification_metrics = sensitivity_recall_comprehensive()
```

User: This is incredibly comprehensive! I love how you showed that sensitivity and recall are the same thing, just used in different contexts. Now I'm really curious about ROC curves - I see them mentioned a lot but I'm not sure how to interpret them properly.

Expert: Excellent! **ROC curves** (Receiver Operating Characteristic) are one of the most important tools for evaluating classification models. Let me give you a complete understanding of how to create, interpret, and use them effectively.

```
def roc_curve_comprehensive():
    """
    Comprehensive explanation of ROC curves and AUC
    """
    print("ROC CURVES AND AUC: COMPLETE GUIDE")
    print("=" * 40)

    print("WHAT IS A ROC CURVE?")
    print("-" * 22)
    print("• Plots True Positive Rate (Sensitivity) vs False Positive Rate (1-Specificity)")
    print("• Shows performance across all classification thresholds")
    print("• Helps choose optimal threshold for your business needs")
    print("• AUC (Area Under Curve) summarizes overall performance")
    print()

    # Generate sample data with different model qualities
    np.random.seed(42)
    n_samples = 1000

    # True labels (20% positive class)
    y_true = np.random.choice([0, 1], n_samples, p=[0.8, 0.2])

    # Simulate different model prediction probabilities

    # Model 1: Excellent model
    y_prob_excellent = []
    for label in y_true:
        if label == 1:
            prob = np.random.beta(8, 2)  # High probabilities for positive class
```

```
        else:
            prob = np.random.beta(2, 8)  # Low probabilities for negative class
        y_prob_excellent.append(prob)
    y_prob_excellent = np.array(y_prob_excellent)

    # Model 2: Good model
    y_prob_good = []
    for label in y_true:
        if label == 1:
            prob = np.random.beta(5, 3)
        else:
            prob = np.random.beta(3, 5)
        y_prob_good.append(prob)
    y_prob_good = np.array(y_prob_good)

    # Model 3: Poor model
    y_prob_poor = []
    for label in y_true:
        if label == 1:
            prob = np.random.beta(3, 4)
        else:
            prob = np.random.beta(4, 3)
        y_prob_poor.append(prob)
    y_prob_poor = np.array(y_prob_poor)

    # Model 4: Random model
    y_prob_random = np.random.uniform(0, 1, n_samples)

    # Calculate ROC curves
    from sklearn.metrics import roc_curve, auc

    models = {
        'Excellent Model': y_prob_excellent,
        'Good Model': y_prob_good,
        'Poor Model': y_prob_poor,
        'Random Model': y_prob_random
    }

    colors = ['green', 'blue', 'orange', 'red']

    fig, axes = plt.subplots(2, 3, figsize=(18, 12))

    # Plot ROC curves
    axes[0, 0].plot([0, 1], [0, 1], 'k--', alpha=0.5, label='Random Classifier')

    roc_data = {}

    for i, (model_name, y_prob) in enumerate(models.items()):
```

```
        fpr, tpr, thresholds = roc_curve(y_true, y_prob)
        roc_auc = auc(fpr, tpr)

        axes[0, 0].plot(fpr, tpr, color=colors[i], linewidth=2,
                       label=f'{model_name} (AUC = {roc_auc:.3f})')

        roc_data[model_name] = {
            'fpr': fpr,
            'tpr': tpr,
            'thresholds': thresholds,
            'auc': roc_auc
        }

    axes[0, 0].set_xlabel('False Positive Rate (1 - Specificity)')
    axes[0, 0].set_ylabel('True Positive Rate (Sensitivity)')
    axes[0, 0].set_title('ROC Curves Comparison')
    axes[0, 0].legend(loc='lower right')
    axes[0, 0].grid(True, alpha=0.3)

    # Show probability distributions
    axes[0, 1].hist(y_prob_excellent[y_true == 0], bins=30, alpha=0.7,
                   label='Negative Class', color='red', density=True)
    axes[0, 1].hist(y_prob_excellent[y_true == 1], bins=30, alpha=0.7,
                   label='Positive Class', color='green', density=True)
    axes[0, 1].axvline(0.5, color='black', linestyle='--', alpha=0.7, label='Threshold = 0.5')
    axes[0, 1].set_xlabel('Predicted Probability')
    axes[0, 1].set_ylabel('Density')
    axes[0, 1].set_title('Excellent Model: Probability Distributions')
    axes[0, 1].legend()
    axes[0, 1].grid(True, alpha=0.3)

    # Show threshold selection
    excellent_fpr = roc_data['Excellent Model']['fpr']
    excellent_tpr = roc_data['Excellent Model']['tpr']
    excellent_thresholds = roc_data['Excellent Model']['thresholds']

    # Find optimal threshold (closest to top-left corner)
    distances = np.sqrt((excellent_fpr - 0)**2 + (excellent_tpr - 1)**2)
    optimal_idx = np.argmin(distances)
    optimal_threshold = excellent_thresholds[optimal_idx]
    optimal_fpr = excellent_fpr[optimal_idx]
    optimal_tpr = excellent_tpr[optimal_idx]
```

```python

    axes[0, 2].plot(excellent_fpr, excellent_tpr, 'g-', linewidth=2, label='ROC Curve')
    axes[0, 2].scatter(optimal_fpr, optimal_tpr, color='red', s=100, zorder=5,
                       label=f'Optimal Threshold = {optimal_threshold:.3f}')
    axes[0, 2].plot([0, 1], [0, 1], 'k--', alpha=0.5)
    axes[0, 2].set_xlabel('False Positive Rate')
    axes[0, 2].set_ylabel('True Positive Rate')
    axes[0, 2].set_title('Threshold Selection')
    axes[0, 2].legend()
    axes[0, 2].grid(True, alpha=0.3)

    # AUC interpretation
    axes[1, 0].axis('off')

    auc_text = f"""
AUC INTERPRETATION GUIDE:

AUC = 1.0: Perfect classifier
AUC = 0.9-1.0: Excellent
AUC = 0.8-0.9: Good
AUC = 0.7-0.8: Fair
AUC = 0.6-0.7: Poor
AUC = 0.5: Random (no skill)
AUC < 0.5: Worse than random

OUR MODELS:
• Excellent: {roc_data['Excellent Model']['auc']:.3f}
• Good: {roc_data['Good Model']['auc']:.3f}
• Poor: {roc_data['Poor Model']['auc']:.3f}
• Random: {roc_data['Random Model']['auc']:.3f}

BUSINESS MEANING:
AUC = Probability that model ranks
a random positive example higher
than a random negative example.
    """

    axes[1, 0].text(0.05, 0.95, auc_text, transform=axes[1, 0].transAxes,
                    fontsize=11, verticalalignment='top', fontfamily='monospace',
                    bbox=dict(boxstyle='round', facecolor='lightgreen', alpha=0.8))

    # Threshold analysis table
    axes[1, 1].axis('off')

    # Calculate metrics at different thresholds for excellent model
    thresholds_to_analyze = [0.3, 0.5, 0.7, optimal_threshold]

```

```
    threshold_analysis = []
    for thresh in thresholds_to_analyze:
        y_pred = (y_prob_excellent >= thresh).astype(int)

        from sklearn.metrics import confusion_matrix, precision_score, recall_score
        cm = confusion_matrix(y_true, y_pred)
        tn, fp, fn, tp = cm.ravel()

        precision = precision_score(y_true, y_pred)
        recall = recall_score(y_true, y_pred)
        specificity = tn / (tn + fp)

        threshold_analysis.append({
            'threshold': thresh,
            'precision': precision,
            'recall': recall,
            'specificity': specificity,
            'fpr': 1 - specificity
        })

    # Create table
    table_data = []
    table_data.append(['Threshold', 'Precision', 'Recall', 'Specificity', 'FPR'])

    for analysis in threshold_analysis:
        row = [
            f"{analysis['threshold']:.3f}",
            f"{analysis['precision']:.3f}",
            f"{analysis['recall']:.3f}",
            f"{analysis['specificity']:.3f}",
            f"{analysis['fpr']:.3f}"
        ]
        table_data.append(row)

    table = axes[1, 1].table(cellText=table_data[1:],
                            colLabels=table_data[0],
                            cellLoc='center',
                            loc='center',
                            bbox=[0, 0.3, 1, 0.6])

    table.auto_set_font_size(False)
    table.set_fontsize(10)
    table.scale(1, 2)

    # Color code the header and optimal row
    for i in range(5):
        table[(0, i)].set_facecolor('#4CAF50')
        table[(0, i)].set_text_props(weight='bold', color='white')
```

```

    # Highlight optimal threshold row
    optimal_row = 4  # Last row is optimal
    for i in range(5):
        table[(optimal_row, i)].set_facecolor('#FFE082')

    axes[1, 1].set_title('Threshold Analysis for Excellent Model', pad=20, fontweight='bold')

    # Business decision guide
    axes[1, 2].axis('off')

    decision_guide = f"""
BUSINESS DECISION GUIDE:
CHOOSE THRESHOLD BASED ON:
1. COST OF FALSE POSITIVES:
   High cost → Higher threshold
   Low cost → Lower threshold
2. COST OF FALSE NEGATIVES:
   High cost → Lower threshold
   Low cost → Higher threshold
3. BUSINESS CONTEXT:
MEDICAL SCREENING:
• Low threshold (0.3)
• High sensitivity, catch all cases
• Accept more false alarms
SPAM DETECTION:
• High threshold (0.7)
• High precision, avoid blocking
• Accept some spam getting through
FRAUD DETECTION:
• Optimal threshold ({optimal_threshold:.3f})
• Balance sensitivity & precision
• Minimize total cost
REMEMBER:
ROC curve shows ALL possible
trade-offs. Choose the point
that best fits your business needs!
    """

    axes[1, 2].text(0.05, 0.95, decision_guide, transform=axes[1, 2].transAxes,
                    fontsize=9, verticalalignment='top', fontfamily='monospace',
                    bbox=dict(boxstyle='round', facecolor='lightblue', alpha=0.8))

    plt.tight_layout()
    plt.show()

```

```
    print("KEY INSIGHTS:")
    print("-" * 15)
    print("✓ ROC curve shows performance across all thresholds")
    print("✓ AUC summarizes overall discriminative ability")
    print("✓ Closer to top-left corner = better performance")
    print("✓ Diagonal line = random performance")
    print("✓ Choose threshold based on business costs")
    print("✓ ROC curves are threshold-independent evaluation")

    return roc_data, threshold_analysis

roc_results, threshold_results = roc_curve_comprehensive()
```

Chapter 7: Advanced Evaluation Metrics and Techniques

User: The ROC curves really helped me understand model evaluation better! But I feel like there are still some advanced techniques I should know about. What about cross-validation? And are there other important evaluation methods I should be aware of?

Expert: Absolutely! You're ready for the advanced evaluation techniques that separate good data scientists from great ones. Let's dive into **cross-validation**, **bootstrap sampling**, and other sophisticated evaluation methods.

```
def cross_validation_comprehensive():
    """
    Comprehensive guide to cross-validation techniques
    """
    print("CROSS-VALIDATION: ADVANCED MODEL EVALUATION")
    print("=" * 50)

    print("WHY CROSS-VALIDATION?")
    print("-" * 23)
    print("• Single train/test split can be misleading")
    print("• Results depend on which data points end up in test set")
    print("• Cross-validation gives more robust performance estimates")
    print("• Helps detect overfitting and model instability")
    print()

    # Generate sample data
    from sklearn.datasets import make_classification
    from sklearn.model_selection import cross_val_score, StratifiedKFold, TimeSeriesSplit
    from sklearn.linear_model import LogisticRegression
    from sklearn.ensemble import RandomForestClassifier
```

```python
    from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

    np.random.seed(42)
    X, y = make_classification(n_samples=1000, n_features=20, n_informative=10,
                              n_redundant=10, n_clusters_per_class=1, random_state=42)

    # Models to compare
    models = {
        'Logistic Regression': LogisticRegression(random_state=42),
        'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42)
    }

    print("TYPES OF CROSS-VALIDATION:")
    print("-" * 30)

    # 1. K-Fold Cross-Validation
    print("1. K-FOLD CROSS-VALIDATION")
    print("   • Split data into k equal folds")
    print("   • Train on k-1 folds, test on 1 fold")
    print("   • Repeat k times, average results")
    print()

    from sklearn.model_selection import KFold

    kf = KFold(n_splits=5, shuffle=True, random_state=42)

    fig, axes = plt.subplots(3, 2, figsize=(15, 18))

    # Visualize K-Fold splits
    fold_colors = ['red', 'blue', 'green', 'orange', 'purple']

    for i, (train_idx, test_idx) in enumerate(kf.split(X)):
        if i < 5:  # Show first 5 folds
            y_pos = [i] * len(train_idx) + [i] * len(test_idx)
            colors = ['blue'] * len(train_idx) + ['red'] * len(test_idx)
            x_pos = list(train_idx) + list(test_idx)

            if i == 0:  # Only show for first subplot
                axes[0, 0].scatter(x_pos[:len(train_idx)], y_pos[:len(train_idx)],
                                  c='blue', alpha=0.6, s=1, label='Training')
                axes[0, 0].scatter(x_pos[len(train_idx):], y_pos[len(train_idx):],
                                  c='red', alpha=0.6, s=1,
```

````
label='Testing')
<user>: continue
<Claude Sonnet 4>: ```python
            else:
                axes[0, 0].scatter(x_pos[:len(train_idx)],
y_pos[:len(train_idx)],
                                  c='blue', alpha=0.6, s=1)
                axes[0, 0].scatter(x_pos[len(train_idx):],
y_pos[len(train_idx):],
                                  c='red', alpha=0.6, s=1)

    axes[0, 0].set_xlabel('Sample Index')
    axes[0, 0].set_ylabel('Fold Number')
    axes[0, 0].set_title('5-Fold Cross-Validation Splits')
    axes[0, 0].legend()
    axes[0, 0].grid(True, alpha=0.3)

    # Perform cross-validation for all models
    cv_results = {}

    for model_name, model in models.items():
        scores = cross_val_score(model, X, y, cv=5,
scoring='accuracy')
        cv_results[model_name] = {
            'scores': scores,
            'mean': scores.mean(),
            'std': scores.std()
        }

    # Plot cross-validation results
    model_names = list(cv_results.keys())
    means = [cv_results[name]['mean'] for name in model_names]
    stds = [cv_results[name]['std'] for name in model_names]

    bars = axes[0, 1].bar(model_names, means, yerr=stds, capsize=5,
                         alpha=0.7, color=['blue', 'green'])
    axes[0, 1].set_ylabel('Accuracy')
    axes[0, 1].set_title('Cross-Validation Results (5-Fold)')
    axes[0, 1].grid(True, alpha=0.3)

    # Add value labels
    for i, (bar, mean, std) in enumerate(zip(bars, means, stds)):
        axes[0, 1].text(bar.get_x() + bar.get_width()/2,
bar.get_height() + std + 0.01,
                       f'{mean:.3f}±{std:.3f}', ha='center',
va='bottom', fontweight='bold')

    print(f"5-Fold CV Results:")
    for name in model_names:
        mean_acc = cv_results[name]['mean']
````

```
        std_acc = cv_results[name]['std']
        print(f"  {name}: {mean_acc:.3f} ± {std_acc:.3f}")

    # 2. Stratified K-Fold
    print(f"\n2. STRATIFIED K-FOLD CROSS-VALIDATION")
    print("   • Maintains class distribution in each fold")
    print("   • Important for imbalanced datasets")
    print("   • Ensures each fold is representative")
    print()

    skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

    # Compare class distributions
    fold_distributions = []
    for i, (train_idx, test_idx) in enumerate(skf.split(X, y)):
        train_dist = np.bincount(y[train_idx]) / len(train_idx)
        test_dist = np.bincount(y[test_idx]) / len(test_idx)
        fold_distributions.append({
            'fold': i+1,
            'train_class_0': train_dist[0],
            'train_class_1': train_dist[1],
            'test_class_0': test_dist[0],
            'test_class_1': test_dist[1]
        })

    # Visualize class distributions
    folds = [d['fold'] for d in fold_distributions]
    train_0 = [d['train_class_0'] for d in fold_distributions]
    train_1 = [d['train_class_1'] for d in fold_distributions]
    test_0 = [d['test_class_0'] for d in fold_distributions]
    test_1 = [d['test_class_1'] for d in fold_distributions]

    x = np.arange(len(folds))
    width = 0.35

    axes[1, 0].bar(x - width/2, train_0, width, label='Train Class 0', alpha=0.7, color='lightblue')
    axes[1, 0].bar(x - width/2, train_1, width, bottom=train_0, label='Train Class 1', alpha=0.7, color='lightcoral')
    axes[1, 0].bar(x + width/2, test_0, width, label='Test Class 0', alpha=0.7, color='blue')
    axes[1, 0].bar(x + width/2, test_1, width, bottom=test_0, label='Test Class 1', alpha=0.7, color='red')

    axes[1, 0].set_xlabel('Fold')
    axes[1, 0].set_ylabel('Class Proportion')
    axes[1, 0].set_title('Stratified K-Fold: Class Distribution')
    axes[1, 0].set_xticks(x)
    axes[1, 0].set_xticklabels([f'Fold {i}' for i in folds])
```

```python
    axes[1, 0].legend()
    axes[1, 0].grid(True, alpha=0.3)

    # 3. Time Series Cross-Validation
    print("3. TIME SERIES CROSS-VALIDATION")
    print("   • Respects temporal order of data")
    print("   • No future data leakage")
    print("   • Expanding or sliding window approach")
    print()

    # Generate time series data
    np.random.seed(42)
    n_time_points = 100
    time_X = np.random.randn(n_time_points, 5)
    time_y = np.random.randn(n_time_points)

    tscv = TimeSeriesSplit(n_splits=5)

    # Visualize time series splits
    for i, (train_idx, test_idx) in enumerate(tscv.split(time_X)):
        y_pos = [i] * len(range(n_time_points))
        colors = ['lightgray'] * n_time_points

        for idx in train_idx:
            colors[idx] = 'blue'
        for idx in test_idx:
            colors[idx] = 'red'

        axes[1, 1].scatter(range(n_time_points), y_pos, c=colors, alpha=0.7, s=10)

    axes[1, 1].set_xlabel('Time Point')
    axes[1, 1].set_ylabel('Fold Number')
    axes[1, 1].set_title('Time Series Cross-Validation')

    # Add legend
    from matplotlib.patches import Patch
    legend_elements = [Patch(facecolor='blue', label='Training'),
                       Patch(facecolor='red', label='Testing'),
                       Patch(facecolor='lightgray', label='Not Used')]
    axes[1, 1].legend(handles=legend_elements)
    axes[1, 1].grid(True, alpha=0.3)

    # 4. Leave-One-Out Cross-Validation (LOOCV)
    print("4. LEAVE-ONE-OUT CROSS-VALIDATION (LOOCV)")
    print("   • Each sample is test set once")
    print("   • n-fold CV where n = number of samples")
    print("   • Maximum use of data, but computationally expensive")
```

```
    print()

    # Demonstrate with smaller dataset
    from sklearn.model_selection import LeaveOneOut

    # Use smaller subset for LOOCV demonstration
    X_small = X[:50]
    y_small = y[:50]

    loo = LeaveOneOut()
    loo_scores = cross_val_score(LogisticRegression(random_state=42), X_small, y_small, cv=loo)

    print(f"LOOCV Results on 50 samples:")
    print(f"  Mean Accuracy: {loo_scores.mean():.3f}")
    print(f"  Std Accuracy: {loo_scores.std():.3f}")
    print(f"  Number of folds: {len(loo_scores)}")

    # 5. Nested Cross-Validation
    print(f"\n5. NESTED CROSS-VALIDATION")
    print("   • Outer loop: Model evaluation")
    print("   • Inner loop: Hyperparameter tuning")
    print("   • Unbiased estimate of model performance")
    print()

    from sklearn.model_selection import GridSearchCV

    # Nested CV example
    def nested_cross_validation(X, y, model, param_grid, outer_cv=5, inner_cv=3):
        outer_scores = []

        outer_kf = StratifiedKFold(n_splits=outer_cv, shuffle=True, random_state=42)

        for train_idx, test_idx in outer_kf.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 for hyperparameter tuning
            inner_cv = StratifiedKFold(n_splits=inner_cv, shuffle=True, random_state=42)
            grid_search = GridSearchCV(model, param_grid, cv=inner_cv, scoring='accuracy')
            grid_search.fit(X_train, y_train)

            # Test best model on outer test set
            best_model = grid_search.best_estimator_
            score = best_model.score(X_test, y_test)
```

```
            outer_scores.append(score)

        return np.array(outer_scores)

    # Perform nested CV
    param_grid = {'C': [0.1, 1, 10], 'max_iter': [100, 1000]}
    nested_scores = nested_cross_validation(X, y, LogisticRegression(random_state=42), param_grid)

    print(f"Nested CV Results:")
    print(f"  Mean Accuracy: {nested_scores.mean():.3f}")
    print(f"  Std Accuracy: {nested_scores.std():.3f}")

    # Visualize different CV methods comparison
    cv_methods = ['5-Fold', 'Stratified 5-Fold', 'LOOCV (50 samples)', 'Nested CV']
    cv_means = [
        cv_results['Logistic Regression']['mean'],
        cv_results['Logistic Regression']['mean'],  # Similar for stratified
        loo_scores.mean(),
        nested_scores.mean()
    ]
    cv_stds = [
        cv_results['Logistic Regression']['std'],
        cv_results['Logistic Regression']['std'],
        loo_scores.std(),
        nested_scores.std()
    ]

    bars = axes[2, 0].bar(cv_methods, cv_means, yerr=cv_stds, capsize=5,
                         alpha=0.7, color=['blue', 'green', 'orange', 'red'])
    axes[2, 0].set_ylabel('Accuracy')
    axes[2, 0].set_title('Cross-Validation Methods Comparison')
    axes[2, 0].tick_params(axis='x', rotation=45)
    axes[2, 0].grid(True, alpha=0.3)

    # Add value labels
    for bar, mean, std in zip(bars, cv_means, cv_stds):
        axes[2, 0].text(bar.get_x() + bar.get_width()/2, bar.get_height() + std + 0.01,
                       f'{mean:.3f}±{std:.3f}', ha='center', va='bottom', fontweight='bold', fontsize=9)

    # Cross-validation best practices
    axes[2, 1].axis('off')

    best_practices = """
```

```
CROSS-VALIDATION BEST PRACTICES:
✓ CHOOSE THE RIGHT METHOD:
  • Standard: K-Fold (k=5 or k=10)
  • Imbalanced data: Stratified K-Fold
  • Time series: TimeSeriesSplit
  • Small datasets: LOOCV
  • Hyperparameter tuning: Nested CV
✓ CONSIDERATIONS:
  • More folds = less bias, more variance
  • Fewer folds = more bias, less variance
  • Computational cost increases with folds
✓ COMMON MISTAKES TO AVOID:
  • Data leakage between folds
  • Not stratifying imbalanced data
  • Using future data in time series
  • Tuning hyperparameters on test set
✓ REPORTING RESULTS:
  • Always report mean ± standard deviation
  • Consider confidence intervals
  • Test statistical significance of differences
  • Validate on truly held-out test set
    """

    axes[2, 1].text(0.05, 0.95, best_practices, transform=axes[2, 1].transAxes,
                    fontsize=10, verticalalignment='top', fontfamily='monospace',
                    bbox=dict(boxstyle='round', facecolor='lightcyan', alpha=0.8))

    plt.tight_layout()
    plt.show()

    return cv_results, nested_scores
cv_results, nested_scores = cross_validation_comprehensive()
```

User: Cross-validation makes so much sense now! I can see how it gives us much more confidence in our model evaluation. But what about bootstrap sampling? I've heard it's another important technique for understanding uncertainty.

Expert: Excellent question! **Bootstrap sampling** is a powerful statistical technique that complements cross-validation beautifully. It's particularly useful for estimating confidence intervals and understanding the stability of your model performance.

```
def bootstrap_sampling_comprehensive():
    """
    Comprehensive guide to bootstrap sampling for model evaluation
```

```
    """
    print("BOOTSTRAP SAMPLING FOR MODEL EVALUATION")
    print("=" * 45)

    print("WHAT IS BOOTSTRAP SAMPLING?")
    print("-" * 32)
    print("• Sampling WITH replacement from your dataset")
    print("• Creates multiple 'bootstrap samples' of same size as original")
    print("• Each sample is slightly different due to random sampling")
    print("• Estimates sampling distribution of any statistic")
    print("• Provides confidence intervals without assumptions")
    print()

    # Generate sample data
    from sklearn.datasets import make_regression
    from sklearn.linear_model import LinearRegression
    from sklearn.metrics import mean_squared_error, r2_score

    np.random.seed(42)
    X, y = make_regression(n_samples=200, n_features=5, noise=10, random_state=42)

    # Original model performance
    model = LinearRegression()
    model.fit(X, y)
    y_pred = model.predict(X)
    original_r2 = r2_score(y, y_pred)
    original_mse = mean_squared_error(y, y_pred)

    print("BOOTSTRAP PROCEDURE:")
    print("-" * 21)
    print("1. Sample n observations WITH replacement")
    print("2. Train model on bootstrap sample")
    print("3. Evaluate model performance")
    print("4. Repeat B times (typically B = 1000)")
    print("5. Analyze distribution of performance metrics")
    print()

    # Perform bootstrap sampling
    def bootstrap_model_evaluation(X, y, model, n_bootstrap=1000):
        """
        Perform bootstrap evaluation of model performance
        """
        n_samples = len(X)
        bootstrap_scores = {
            'r2_scores': [],
            'mse_scores': [],
            'coefficients': [],
```

```python
            'intercepts': []
        }

        for i in range(n_bootstrap):
            # Bootstrap sample
            bootstrap_indices = np.random.choice(n_samples, size=n_samples, replace=True)
            X_bootstrap = X[bootstrap_indices]
            y_bootstrap = y[bootstrap_indices]

            # Train model on bootstrap sample
            bootstrap_model = LinearRegression()
            bootstrap_model.fit(X_bootstrap, y_bootstrap)

            # Evaluate on bootstrap sample
            y_pred_bootstrap = bootstrap_model.predict(X_bootstrap)
            r2_bootstrap = r2_score(y_bootstrap, y_pred_bootstrap)
            mse_bootstrap = mean_squared_error(y_bootstrap, y_pred_bootstrap)

            # Store results
            bootstrap_scores['r2_scores'].append(r2_bootstrap)
            bootstrap_scores['mse_scores'].append(mse_bootstrap)
            bootstrap_scores['coefficients'].append(bootstrap_model.coef_)
            bootstrap_scores['intercepts'].append(bootstrap_model.intercept_)

        return bootstrap_scores

    print("Performing 1000 bootstrap samples...")
    bootstrap_results = bootstrap_model_evaluation(X, y, model, n_bootstrap=1000)

    # Convert to numpy arrays for easier analysis
    r2_scores = np.array(bootstrap_results['r2_scores'])
    mse_scores = np.array(bootstrap_results['mse_scores'])
    coefficients = np.array(bootstrap_results['coefficients'])
    intercepts = np.array(bootstrap_results['intercepts'])

    # Calculate confidence intervals
    def calculate_confidence_intervals(data, confidence_level=0.95):
        """
        Calculate confidence intervals using percentile method
        """
        alpha = 1 - confidence_level
        lower_percentile = (alpha/2) * 100
        upper_percentile = (1 - alpha/2) * 100

```

```
        lower_bound = np.percentile(data, lower_percentile)
        upper_bound = np.percentile(data, upper_percentile)

        return lower_bound, upper_bound

    # Calculate CIs for performance metrics
    r2_ci_lower, r2_ci_upper = calculate_confidence_intervals(r2_scores)
    mse_ci_lower, mse_ci_upper = calculate_confidence_intervals(mse_scores)

    print("BOOTSTRAP RESULTS:")
    print("-" * 18)
    print(f"Original R² Score: {original_r2:.4f}")
    print(f"Bootstrap R² Mean: {np.mean(r2_scores):.4f}")
    print(f"Bootstrap R² Std:  {np.std(r2_scores):.4f}")
    print(f"95% CI for R²:     [{r2_ci_lower:.4f}, {r2_ci_upper:.4f}]")
    print()
    print(f"Original MSE:      {original_mse:.2f}")
    print(f"Bootstrap MSE Mean: {np.mean(mse_scores):.2f}")
    print(f"Bootstrap MSE Std:  {np.std(mse_scores):.2f}")
    print(f"95% CI for MSE:     [{mse_ci_lower:.2f}, {mse_ci_upper:.2f}]")

    # Visualize bootstrap results
    fig, axes = plt.subplots(3, 2, figsize=(15, 18))

    # R² distribution
    axes[0, 0].hist(r2_scores, bins=50, alpha=0.7, color='blue', density=True)
    axes[0, 0].axvline(original_r2, color='red', linestyle='--', linewidth=2, label=f'Original R² ({original_r2:.3f})')
    axes[0, 0].axvline(np.mean(r2_scores), color='green', linestyle='-', linewidth=2, label=f'Bootstrap Mean ({np.mean(r2_scores):.3f})')
    axes[0, 0].axvline(r2_ci_lower, color='orange', linestyle=':', alpha=0.7, label=f'95% CI')
    axes[0, 0].axvline(r2_ci_upper, color='orange', linestyle=':', alpha=0.7)
    axes[0, 0].set_xlabel('R² Score')
    axes[0, 0].set_ylabel('Density')
    axes[0, 0].set_title('Bootstrap Distribution of R² Scores')
    axes[0, 0].legend()
    axes[0, 0].grid(True, alpha=0.3)

    # MSE distribution
    axes[0, 1].hist(mse_scores, bins=50, alpha=0.7, color='red', density=True)
    axes[0, 1].axvline(original_mse, color='blue', linestyle='--',
```

```python
linewidth=2, label=f'Original MSE ({original_mse:.1f})')
    axes[0, 1].axvline(np.mean(mse_scores), color='green', linestyle='-', linewidth=2, label=f'Bootstrap Mean ({np.mean(mse_scores):.1f})')
    axes[0, 1].axvline(mse_ci_lower, color='orange', linestyle=':', alpha=0.7, label=f'95% CI')
    axes[0, 1].axvline(mse_ci_upper, color='orange', linestyle=':', alpha=0.7)
    axes[0, 1].set_xlabel('MSE')
    axes[0, 1].set_ylabel('Density')
    axes[0, 1].set_title('Bootstrap Distribution of MSE')
    axes[0, 1].legend()
    axes[0, 1].grid(True, alpha=0.3)

    # Coefficient stability
    feature_names = [f'Feature_{i+1}' for i in range(coefficients.shape[1])]

    # Box plot of coefficients
    axes[1, 0].boxplot(coefficients, labels=feature_names)
    axes[1, 0].set_ylabel('Coefficient Value')
    axes[1, 0].set_title('Bootstrap Distribution of Model Coefficients')
    axes[1, 0].tick_params(axis='x', rotation=45)
    axes[1, 0].grid(True, alpha=0.3)

    # Add original coefficients as red dots
    for i, coef in enumerate(model.coef_):
        axes[1, 0].scatter(i+1, coef, color='red', s=50, zorder=5)

    # Coefficient confidence intervals
    coef_stats = []
    for i in range(coefficients.shape[1]):
        coef_mean = np.mean(coefficients[:, i])
        coef_std = np.std(coefficients[:, i])
        coef_ci_lower, coef_ci_upper = calculate_confidence_intervals(coefficients[:, i])

        coef_stats.append({
            'feature': feature_names[i],
            'original': model.coef_[i],
            'bootstrap_mean': coef_mean,
            'bootstrap_std': coef_std,
            'ci_lower': coef_ci_lower,
            'ci_upper': coef_ci_upper
        })

    # Coefficient comparison plot
    x_pos = np.arange(len(feature_names))
    bootstrap_means = [stat['bootstrap_mean'] for stat in
```

```
coef_stats]
    bootstrap_stds = [stat['bootstrap_std'] for stat in coef_stats]
    original_coefs = [stat['original'] for stat in coef_stats]

    axes[1, 1].errorbar(x_pos, bootstrap_means, yerr=bootstrap_stds,
                       fmt='o', capsize=5, label='Bootstrap Mean ± Std')
    axes[1, 1].scatter(x_pos, original_coefs, color='red', s=50,
                      label='Original Coefficients', zorder=5)
    axes[1, 1].set_xlabel('Features')
    axes[1, 1].set_ylabel('Coefficient Value')
    axes[1, 1].set_title('Coefficient Stability Analysis')
    axes[1, 1].set_xticks(x_pos)
    axes[1, 1].set_xticklabels(feature_names, rotation=45)
    axes[1, 1].legend()
    axes[1, 1].grid(True, alpha=0.3)

    # Bootstrap vs Cross-Validation comparison
    print(f"\nBOOTSTRAP vs CROSS-VALIDATION:")
    print("-" * 35)

    # Perform 5-fold CV for comparison
    from sklearn.model_selection import cross_val_score
    cv_r2_scores = cross_val_score(model, X, y, cv=5, scoring='r2')
    cv_mse_scores = -cross_val_score(model, X, y, cv=5, scoring='neg_mean_squared_error')

    print(f"Cross-Validation R² Mean: {np.mean(cv_r2_scores):.4f} ± {np.std(cv_r2_scores):.4f}")
    print(f"Bootstrap R² Mean:        {np.mean(r2_scores):.4f} ± {np.std(r2_scores):.4f}")
    print()
    print(f"Cross-Validation MSE Mean: {np.mean(cv_mse_scores):.2f} ± {np.std(cv_mse_scores):.2f}")
    print(f"Bootstrap MSE Mean:        {np.mean(mse_scores):.2f} ± {np.std(mse_scores):.2f}")

    # Comparison visualization
    methods = ['Cross-Validation', 'Bootstrap']
    r2_means_comp = [np.mean(cv_r2_scores), np.mean(r2_scores)]
    r2_stds_comp = [np.std(cv_r2_scores), np.std(r2_scores)]

    bars = axes[2, 0].bar(methods, r2_means_comp, yerr=r2_stds_comp,
                         capsize=5, alpha=0.7, color=['blue', 'green'])
    axes[2, 0].set_ylabel('R² Score')
    axes[2, 0].set_title('Bootstrap vs Cross-Validation: R² Comparison')
```

```python
    axes[2, 0].grid(True, alpha=0.3)

    # Add value labels
    for bar, mean, std in zip(bars, r2_means_comp, r2_stds_comp):
        axes[2, 0].text(bar.get_x() + bar.get_width()/2, bar.get_height() + std + 0.01,
                       f'{mean:.3f}±{std:.3f}', ha='center', va='bottom', fontweight='bold')

    # Bootstrap applications and best practices
    axes[2, 1].axis('off')

    applications_text = """
BOOTSTRAP APPLICATIONS & BEST PRACTICES:

✓ WHEN TO USE BOOTSTRAP:
 • Estimate confidence intervals for any metric
 • Assess model stability and robustness
 • Compare models statistically
 • Understand parameter uncertainty
 • Small datasets where CV is unstable

✓ ADVANTAGES:
 • Works with any statistic or metric
 • No distributional assumptions
 • Provides full sampling distribution
 • Easy to implement and interpret

✓ LIMITATIONS:
 • Assumes sample represents population
 • Can be computationally expensive
 • May not work well with very small samples
 • Doesn't account for model selection bias

✓ BEST PRACTICES:
 • Use B ≥ 1000 bootstrap samples
 • Report confidence intervals
 • Check for stability across runs
 • Combine with other validation methods
 • Consider stratified bootstrap for classification

✓ BUSINESS VALUE:
 • Quantify uncertainty in predictions
 • Make risk-informed decisions
 • Communicate model reliability
 • Support regulatory requirements
    """

    axes[2, 1].text(0.05, 0.95, applications_text, transform=axes[2, 1].transAxes,
                    fontsize=9, verticalalignment='top', fontfamily='monospace',
                    bbox=dict(boxstyle='round', facecolor='lightyellow', alpha=0.8))

```

```
    plt.tight_layout()
    plt.show()

    return bootstrap_results, coef_stats

bootstrap_results, coefficient_stats = bootstrap_sampling_comprehensive()
```