# Robustness Testing

## Enhancing Model Stability Under Noise

July 5, 2025

# Outline


1. Introduction to Model Robustness
2. Input Perturbation Methods
3. Robustness Analysis in MoDeVa
4. Identifying Robustness Issues
5. Advanced Robustness Analysis
6. Strategies for Improving Robustness

# What is Model Robustness?

## Definition

Robustness evaluates a model's ability to perform reliably when exposed to input noise ensuring it remains accurate and stable.

**Why Is Robustness Important?**

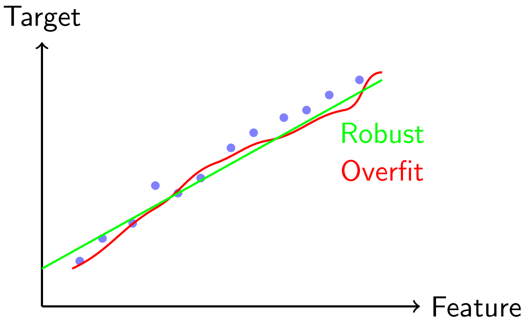
- Real-world data contains noise and variations
- Input features might have measurement errors
- Data might be corrupted during collection or processing
- Features can vary slightly but shouldn't cause drastic prediction changes
- Non-robust models may make unstable decisions in production
- Prevents benign overfitting (fitting noise rather than patterns)

## Example

A slight change in a customer's credit score ($\pm 1\%$) should not cause a dramatic shift in loan approval probability

# The Problem of Benign Overfitting

Undetected overfitting by testing data but fragile under real world.




**Characteristics:**

- Complex models may memorize noise in training data
- Performance looks excellent on training/testing random split
- Performance degrades under small input variations
- Unstable predictions in production environments

# Investigating and Improving Model Robustness

**MoDeVa's Structured Approach:**

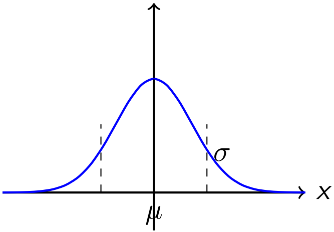
1. **Perturb Input Variables**
   - Introduce controlled perturbations to input variables
   - Simulate real-world data variations
   - Apply different noise types based on feature characteristics
2. **Assess Performance Degradation**
   - Compare metrics before and after perturbation
   - Measure how much performance drops
   - Evaluate stability under various noise levels
3. **Identify and Rank Sensitive Variables**
   - Analyze impact of perturbations on each variable
   - Rank variables by sensitivity to noise
   - Identify features contributing most to instability
4. **Enhance Model Robustness**
   - Apply regularization techniques
   - Refine or transform sensitive features
   - Use ensemble methods to stabilize predictions
   - Implement adversarial training

# Simple Perturbation with Normal Noise

**Approach:**

- Add random noise drawn from a normal distribution
- Define fraction multipliers $k$ (e.g., 0.01, 0.02) to standard deviation ($\sigma$)
- Generate noise for each feature:
  perturbed value $=$ original value $+ k \cdot \mathcal{N}(0, \sigma^2)$

Normal Distribution




**Use Case:**

- Test sensitivity to small-scale, realistic fluctuations
- Simulate measurement errors
- Straightforward and easy to implement
- Suitable for general robustness testing across features

# Quantile Perturbation with Uniform Noise

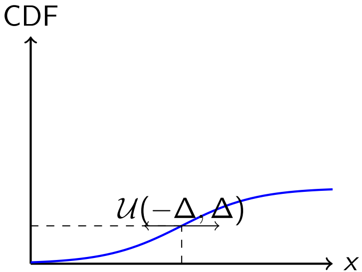
**Approach:**

- Convert input values to quantiles using CDF: $q = F(x)$
- Add uniform noise to quantiles: $q_{\text{perturbed}} = q + \mathcal{U}(-\Delta, \Delta)$
- Map perturbed quantiles back using inverse CDF: $x_{\text{perturbed}} = F^{-1}(q_{\text{perturbed}})$

**Benefits:**

- Preserves statistical properties of variables
- Ideal for non-normal distributions
- Respects variable constraints (e.g., probabilities between 0 and 1)
- Maintains the shape of the original distribution
- More realistic perturbations for many real-world

**Quantile Space**



# Categorical Variable Perturbation

**Approach:**

1. Summarize frequency of each category in the data (e.g., A: 30%, B: 30%, C: 40%)
2. Perturb each sample with probability $p$ (perturb_size)
3. Keep unchanged with probability $1 - p$
4. If a sample is perturbed, it will be changed to each category with probability proportional to the category's frequency

**Example:**

- Three categories: A (30%), B (30%), C (40%)
- If $p = 0.3$, then 30% of the samples will be perturbed
- A perturbed sample will become A with 30% probability, B with 30% probability, or C with 40% probability

**Key Insight**

This approach preserves the overall distribution of categorical variables while introducing realistic noise

# Choosing Methods

**Choosing the Approach**

- Simple Perturbation: Straightforward, general-purpose
- Quantile Perturbation: Preserves distributions, ideal for non-normal data
- Feature-specific design: Choose techniques based on feature types
- Consider domain knowledge when selecting perturbation methods

**Interpretation Considerations**

- Different features may have different sensitivity levels
- Consider practical impact of performance degradation
- Evaluate trade-offs between complexity and robustness
- Ensure consistent evaluation across model comparisons

# Choosing Perturbation Size

**Perturbation Magnitude**

- Ensure perturbations reflect realistic scenarios
- Too small: May not reveal robustness issues
- Too large: Unrealistic and less meaningful testing
- Consider typical measurement errors in your domain

**Maximum Perturbation Size**

- Use a linear model as benchmark
- Apply increasing perturbation levels
- Monitor performance degradation
- Maximum size: Point where performance drops below linear benchmark

# Basic Robustness Assessment

```python
# Create a testsuite that bundles dataset and model
from modeva import TestSuite
ts = TestSuite(ds, model_lgbm) # store bundle of dataset and model

# robustness analysis with different noise levels
results = ts.diagnose_robustness(
    perturb_features=None,          # perturb all features
    noise_levels=(0.1, 0.2, 0.3, 0.4),  # magnitudes
    metric="MSE")                   # evaluation metric
results.plot()

```

*This generates a plot showing performance degradation under increasing noise levels*

## Key Output

The plot shows MSE on the y-axis against noise levels on the x-axis, visualizing how model performance degrades as noise increases

# Univariate Robustness Slicing

```python
# Analyze robustness across a specific feature
results = ts.diagnose_slicing_robustness(
    features="hr",                        # feature to analyze
    perturb_features=("hum", "atemp"),  # features to perturb
    noise_levels=0.1, metric="MAE",
    method="auto-xgb1", threshold=0.8)  # weak region threshold
results.table
results.plot()

```

*This shows how robustness varies across different hours of the day*

**What to look for:**

- Features or feature values with high sensitivity (large performance degradation)
- Patterns in sensitivity across feature ranges

# Distribution Analysis for Sensitive Regions

```python
# import get_data_info from TestSuite utility
from modeva.testsuite.utils.slicing_utils import get_data_info
# retrieve result for "hr" feature
data_info = get_data_info(res_value=results.value)["hr"]
# apply drift test to rank distributional difference
data_results = ds.data_drift_test(
    **data_info,
    distance_metric="PSI", psi_method="uniform", psi_bins=10)
data_results.plot("summary")

```

*This analyzes the feature distribution patterns in sensitive regions*

**Key outputs:**

- PSI summary: Features ranked by distribution difference
- Density plots: Distribution comparison between sensitive and normal regions
- Insights into which feature values are associated with reduced robustness

# Feature Interaction Analysis

```python
# Bivariate slicing to analyze feature interactions
results = ts.diagnose_slicing_robustness(
    features=("hr", "atemp"),      # feature pair to analyze
    perturb_features=("hum", "atemp"),  # features to perturb
    noise_levels=0.1, metric="MSE", threshold=0.7)
results.table
results.plot()

```

*This visualizes how combinations of feature values affect robustness*

**Benefits:**

- Identifies complex interactions affecting model robustness
- Reveals conditional dependencies in sensitivity
- Shows where feature combinations create vulnerable regions
- Helps detect subtle robustness patterns missed in univariate analysis

# Multiple Feature Analysis

```python
# Analyze multiple features independently
results = ts.diagnose_slicing_robustness(
    features=(("hr",), ("atemp",), ("weekday",)),
    perturb_features=("atemp", "hum"),
    noise_levels=0.1, perturb_method="quantile",
    metric="MSE", threshold=0.7)
results.table
results.plot()

```

*This analyzes robustness across multiple features independently*

**Insights from multiple feature analysis:**

- Compare robustness patterns across different features
- Identify which features show the greatest sensitivity to perturbations
- Prioritize features for robustness improvement efforts
- Understand how perturbation method affects different feature types

# Model Comparison

```python
# Compare robustness between models
tsc = TestSuite(ds, models=[model_lgbm, model_xgb])

# Compare overall robustness across noise levels
results = tsc.compare_robustness(
    perturb_features=("hr", "atemp"),
    noise_levels=(0.1, 0.2, 0.3, 0.4),
    perturb_method="quantile", metric="MSE")
results.plot()

```

*This compares robustness between different models across noise levels*

**What to compare:**

- Which model degrades more gracefully as noise increases?
- Do models show different sensitivity to specific noise levels?
- Are there crossover points where one model becomes better than another?
- How do different model architectures affect robustness properties?

# Feature-Specific Robustness Comparison

```python
# Compare model robustness for specific features
results = tsc.compare_slicing_robustness(
    features="hr", noise_levels=0.1,
    method="quantile", metric="MSE")
results.plot()

```

*This compares how different models respond to perturbations across feature values*

**Benefits:**

- Identify features where models show different robustness properties
- Reveal model-specific sensitivities to particular feature ranges
- Guide feature-specific model selection
- Inform ensemble strategies based on complementary robustness profiles

# Random Forest Clustering for Sensitivity Patterns

```python
# Use Random Forest clustering with residual change as target
results = ts.diagnose_residual_cluster(
    dataset="test", response_type="abs_residual_perturb",
    metric="MAE", n_clusters=10, cluster_method="pam",
    sample_size=2000, rf_n_estimators=100, rf_max_depth=5)
results.table
results.plot()

```

*This uses Random Forest proximity to cluster similar samples based on sensitivity to perturbations*

**Key outputs:**

- Cluster table: Performance metrics for each sensitivity cluster
- Feature importance: Variables driving sensitivity clusters
- Cluster visualization: Similarity patterns in sensitivity

# Detailed Cluster Analysis

```python
# Analyze a specific high-sensitivity cluster
cluster_id = 2  # cluster with high sensitivity
# Compare cluster distribution to overall distribution
data_results = ds.data_drift_test(
    **results.value["clusters"][cluster_id]["data_info"],
    distance_metric="PSI", psi_method="uniform", psi_bins=10)
data_results.plot("summary")
data_results.plot(name=('density', 'mnth'))

```

*This analyzes the feature distribution patterns of a specific high-sensitivity cluster*

**Insights from cluster analysis:**

- Distinct feature patterns in high-sensitivity regions
- Feature interaction effects not visible in univariate analysis
- Natural groupings of similar sensitivity patterns
- Key drivers of prediction sensitivity to perturbations

# Remediation: Data-Centric Approaches

**Feature Engineering:**

- Transform features that show high sensitivity
- Apply smoothing techniques to reduce noise impact
- Create derived features that are more stable
- Use binning or discretization for highly sensitive continuous features
- Apply normalization techniques that improve robustness
- Remove or downweight extremely noisy features
- Create ensemble features that combine multiple inputs

**Key Principle**

Good feature engineering can significantly enhance model robustness before any model-specific techniques are applied

# Remediation: Model-Centric Approaches

### 1. Architecture Modifications

- Choose more robust model architectures
- Control tree depth in GBDT models
- Limit number of nodes in neural networks
- Use constant or linear basis functions at terminal nodes
- Incorporate domain knowledge via constraints (e.g., monotonicity)
- Implement simpler models for highly sensitive regions

### 2. Loss Function Adjustments

- Apply regularization to reduce overfitting
- L1 Regularization: Promotes sparsity
- L2 Regularization: Smooths decision boundaries
- Local regularization for specific regions
- Early stopping in training phase
- Adversarial training with perturbed samples
- Robust loss functions less sensitive to outliers

# Implementation Framework

**Diagnose**

- Apply input perturbation
- Analyze performance degradation
- Identify sensitive features
- Analyze feature interactions

**Prioritize**

- Rank features by sensitivity
- Consider business impact
- Focus on high-leverage improvements
- Balance performance vs. robustness

**Implement & Validate**

- Apply targeted remediation
- Rerun robustness testing
- Compare before/after metrics
- Iterate as needed

**Systematic Approach**

Improving robustness requires understanding sensitivity patterns, applying targeted interventions, and validating improvements under perturbation testing

# Summary: Model Robustness Testing

- **Understanding Robustness**: A model's ability to perform reliably when exposed to input noise or changes
- **Input Perturbation Methods**: Simple, Quantile, and Categorical perturbations to test sensitivity
- **Robustness Analysis**: Measuring performance degradation under different noise levels
- **Feature Analysis**: Using slicing and clustering to understand sensitivity patterns
- **Model Comparison**: Different models may show varying robustness properties
- **Targeted Remediation**: Combining feature engineering and model-centric approaches

**Key Takeaway**

Robust models maintain stable performance under realistic input variations, ensuring reliable predictions in noisy real-world environments