Synechron

McGill | DESAUTELS

Model Robustness Evaluation Module Documentation

The **Model Robustness Evaluation** module is designed to help data scientists and machine learning engineers understand how their classification models behave under various kinds of stress. In real-world deployments, inputs may be noisy, corrupted, or even intentionally manipulated. This module provides a suite of tools for simulating those conditions and measuring how performance metrics degrade, as well as identifying vulnerabilities to adversarial attacks.

Introduction

In many mission-critical applications such as credit scoring, fraud detection, or medical diagnosis model predictions must remain reliable even when inputs are less than pristine. Minor shifts or noise in feature values can arise from data preprocessing errors, sensor drift, or deliberate tampering. By systematically perturbing input data and measuring the resulting changes in accuracy, AUC, and other key metrics, you can quantify the sensitivity of your model and take steps to harden it.

This module consolidates three main categories of robustness evaluation:

1. **Covariate Perturbations**: Applying noise, scaling shifts, or random masking to all features simultaneously.
2. **Featurewise Perturbations**: Individually perturbing each feature to identify which variables the model relies on most heavily.
3. **Adversarial Attacks**: Generating carefully crafted adversarial examples under L_0, L_2, and $L\infty$ norms to probe worst-case vulnerabilities.

Prerequisites and Installation

Before using the module, ensure you have Python 3.7 or higher installed. The required Python libraries are:

- numpy for numerical operations
- pandas for DataFrame manipulation
- scikit-learn for classification utilities and metrics
- torch for PyTorch-based attack implementations
- matplotlib for plotting results
-

You can install or upgrade these packages via pip:

pip install numpy pandas scikit-learn torch matplotlib
Next, place the file **model_robustness_eval.py** into your project directory (e.g., alongside your training and inference scripts). There is no further installation: simply import the module as needed.

Preparing Your Data

The module is designed to work with binary classification models that expose two methods:

- .predict_proba(X) returning a 2D NumPy array of shape (n_samples, 2) with class probabilities
- .predict(X) returning a 1D array of integer labels (0 or 1)

Your test data should be formatted as follows:

- **X_test**: either a pandas DataFrame or a NumPy array of shape (n_samples, n_features).
- **y_test**: a pandas Series or 1D array of length n_samples, containing the ground-truth labels.

If your model is a PyTorch nn.Module, it must implement a forward() method that returns (logits, _) and be accessible via .parameters() so that logits gradients can be computed. In that case, you will also pass your model directly to the adversarial attack functions.

Importing the Module

Once the file is in place, import the functions you need:

```
from model_robustness_eval import (
    evaluate_covariate_perturbation,
    evaluate_featurewise_perturbation,
    plot_top_features,
    gradient_topk_l0_attack,
    loss_sensitive_l0_attack,
    fgsm_l2_attack,
    pgd_l2_attack,
    fgsm_linf_attack,
    pgd_linf_attack,
    evaluate_attack
)
```

Each function is standalone, so you can pick and choose the evaluations most relevant to your use case.

Covariate Perturbations

Covariate perturbation functions apply the same form of disruption to every feature in the test set at once. This simulates global noise or drift that might occur in sensors or data pipelines.

```
# Example: add Gaussian noise and randomly zero out 10% of
evaluate_covariate_perturbation(
    X_test,                 # DataFrame or array
    y_test,                 # true labels
    model,                  # trained model
    methods=['gaussian_noise','random_mask'],
    noise_level=0.05,       # noise standard deviation
    mask_prob=0.1,          # fraction of values to mask
    verbose=True            # print detailed metrics
)
```

This call will:

1. Add Gaussian noise with standard deviation = 0.05 to all features.
2. Multiply 10% of feature values by zero (random mask).
3. Compute predictions on the perturbed data and print accuracy, AUC, F1-score, and a classification report.
4. Report how many predictions changed relative to the original test set.

If you set return_metrics=True, the function will return a dictionary of numeric metrics along with the perturbed features for further analysis.

Featurewise Perturbations

To discover which individual features the model is most sensitive to, use the featurewise perturbation evaluator. It loops over each column, perturbs just that column, and measures the drop in metrics.

```
# Compute impact of Gaussian noise on each feature independently
df_impacts = evaluate_featurewise_perturbation(
    X_test_df,            # must be a pandas DataFrame
    y_test,               # true labels
    model,
    method='gaussian_noise',
    noise_level=0.1       # noise scaled to each feature's std dev
)

# Visualize the top 10 features whose perturbation caused the largest AUC drop
plot_top_features(df_impacts, metric='delta_auc', top_n=10)
```

The resulting DataFrame has columns:

- feature: the column name
- delta_accuracy: accuracy drop
- delta_f1: F1-score drop
- delta_auc: AUC drop
- pred_change_pct: percent of samples whose predicted class flipped

Plotting delta_auc helps you quickly identify features that are “brittle”—small changes in these inputs have an outsized impact on performance.

Adversarial Attacks

Adversarial attacks craft worst-case perturbations under different norms. The module implements six attacks:

1. **Gradient-based L_0**: Perturbs the top-k most salient features by gradient magnitude.
2. **Loss-sensitive L_0**: Chooses top-k features that maximize the model’s loss increase.
3. **FGSM L_2**: Fast gradient sign method scaled and projected to L_2 norm.
4. **PGD L_2**: Projected gradient descent under L_2 norm.
5. **FGSM L∞**: Fast gradient sign method under L∞ norm.
6. **PGD L∞**: Projected gradient descent under L∞ norm.

Each attack function returns an adversarially perturbed NumPy array X_adv. You can then call the generic evaluator:

```
X_adv = fgsm_l2_attack(model, X_np, y_np, epsilon=3.0)
metrics = evaluate_attack(X_adv, y_np, model, label='FGSM L2 Attack')
```

The evaluate_attack() function prints a detailed classification report and returns a metrics dictionary.

Putting It All Together: Example Workflow

Below is a narrative of how you might integrate the module into your evaluation pipeline:

1. **Load your model and data**:

```
import pandas as pd
from sklearn.externals import joblib
from model_robustness_eval import *

X_test_df = pd.read_csv('data/X_test.csv')
y_test      = pd.read_csv('data/y_test.csv')['target']
model       = joblib.load('models/credit_model.pkl')
```

2. **Baseline performance**:

```
y_pred = model.predict(X_test_df.values)
print(classification_report(y_test, y_pred))
```

3. **Assess global noise sensitivity**:

```
evaluate_covariate_perturbation(
    X_test_df, y_test, model,
    methods=['gaussian_noise','feature_shift'],
    noise_level=0.02
)
```

4. **Identify fragile features**:

```
df_feat_imp = evaluate_featurewise_perturbation(
    X_test_df, y_test, model,
    method='random_mask', mask_prob=0.2
)
plot_top_features(df_feat_imp, metric='pred_change_pct')
```

5. **Probe adversarial robustness**:

```
X_np = X_test_df.values
y_np = y_test.values

for attack_fn, name in [
    (gradient_topk_l0_attack, 'Grad-L0'),
    (loss_sensitive_l0_attack, 'Loss-L0'),
    (fgsm_l2_attack, 'FGSM-L2'),
    (pgd_l2_attack, 'PGD-L2'),
    (fgsm_linf_attack, 'FGSM-Linf'),
    (pgd_linf_attack, 'PGD-Linf'),
]:
    X_adv = attack_fn(model, X_np, y_np)
    metrics = evaluate_attack(X_adv, y_np, model, label=name)
```

Through these steps, you gain a holistic view of your model's resilience to both random and adversarial perturbations. These insights can guide you in feature selection, data augmentation strategies, or adversarial training to improve real-world robustness.

Tips and Best Practices

- **Feature Standardization**: Always apply the same scaling or normalization used in training to the test data before perturbation.
- **Reproducibility**: Use random_state parameters wherever available to fix noise patterns for repeatable experiments.
- **Metric Thresholds**: You can override the default classification threshold (0.5) to match business requirements.
- **Performance**: Adversarial attacks may be computationally intensive on large datasets; consider subsampling or batching if needed.

By following this documentation, you will be able to seamlessly integrate the robustness evaluation module into your own projects and obtain actionable insights on where and how your model may fail under stress.