Modules 6: Model Interpretability & Unbalanced Classes

> A consolidated document on machine learning model interpretability and techniques for handling imbalanced data.

---

1. Concept and importance of Model Interpretability

✅ Concept

Model interpretability is the degree to which humans can understand **how an ML model makes predictions**.

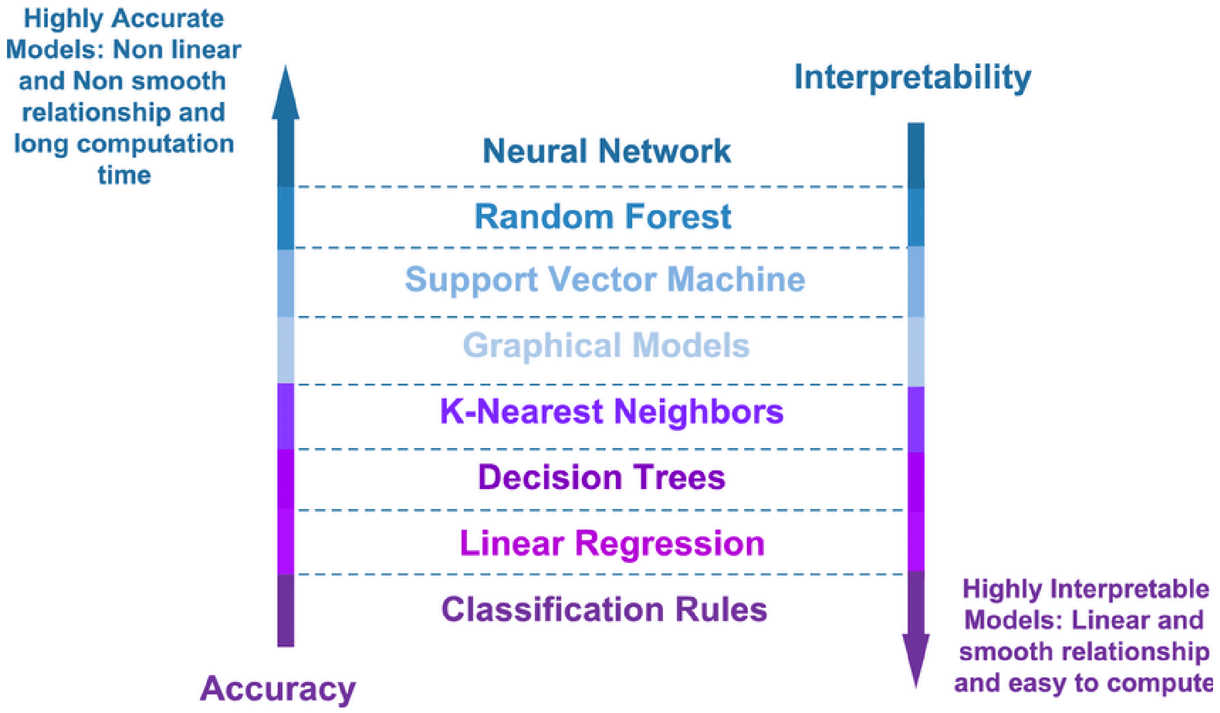
It helps build **trust**, enables **debugging**, and supports **decision-making** based on the model.

💡 Why it matters

- Understand why the model makes mistakes
- Increase transparency in sensitive domains (healthcare, finance, HR)
- Support model optimization and reduce bias

---

2. Model types by interpretability



Model type	Interpretability	Examples
Self-interpretable models	Easy to understand, clear structure	Linear Regression, Decision Tree, KNN
Non-self-interpretable models (Black-box)	Hard to understand, complex	Random Forest, Gradient Boosting, SVM, Deep Neural Network

🔷 Self-interpretable Models

1. Linear models

Formula:

$$y = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

Explanation: Each coefficient w_i reflects the influence of feature x_i on the outcome.

Example: If $w_{\text{age}} = 0.5\$$, increasing age by 1 unit increases the log-odds by 0.5.

2. Tree models

Understandable via **if–else condition branches**.

Example:

```
if income > 5000:
   predict = "Buy"
```

```
else:
    predict = "Not Buy"
```

3. K-nearest neighbors (KNN)

Explainable by **nearby data points**.

Example: If 4 out of 5 neighbors are "Yes", the new point is predicted as "Yes".

---

3. Model interpretation methods

Two main families

Family	Applies to	Goal
Intrinsic methods	Self-explaining models (Linear, Tree, KNN)	Understand directly from model structure
Post-hoc methods	Black-box models	Explain after training

---

4. Model-agnostic explanation methods

📊 (a) Permutation Feature Importance (PFI)

Measures feature importance by **shuffling a feature's values** and observing how much performance drops.

General formula:

$$\text{Importance}(X_i) = \text{Score}_{\text{original}} - \text{Score}_{\text{shuffled}(X_i)}$$

Where:

- Score_original: baseline score
- Score_shuffled(X_i): score after shuffling feature X_i

Example:

Feature	Accuracy (original)	Accuracy (shuffled)	Importance
Age	0.90	0.75	0.15
Salary	0.90	0.80	0.10

→ "Age" is more important than "Salary".

📈 (b) Partial Dependence Plot (PDP)

Describes the **relationship between one or several features and the model output** while averaging over other features.

Formula:

$$\mathrm{PDP}(x_j) = \frac{1}{n}\sum_{i=1}^{n} f\left(x_j, x_{i,\neg j}\right)$$

Where:

- $f\left(x_j, x_{i,\neg j}\right)$: model output when fixing x_j and keeping other features as in sample i
- n: number of samples

Example:

The PDP of experience shows: as experience increases from 0 to 10 years, the probability of "hired" increases then saturates.

Ví dụ minh họa PDP (với scikit-learn):

Giả sử ta có bộ dữ liệu phân loại nhị phân với đặc trưng `experience` và các đặc trưng khác. Ta huấn luyện một mô hình Gradient Boosting rồi vẽ PDP cho `experience`.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.inspection import PartialDependenceDisplay
import matplotlib.pyplot as plt

# 1) Tạo dữ liệu mẫu
X, y = make_classification(
    n_samples=2000,
    n_features=5,
    n_informative=3,
    n_redundant=0,
    random_state=42,
)
feature_names = ["experience", "salary", "age", "overtime", "dept_size"]

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

```
# 2) Huấn luyện mô hình
clf = GradientBoostingClassifier(random_state=42)
clf.fit(X_train, y_train)

# 3) Vẽ PDP cho đặc trưng 'experience' (giả sử là cột 0)
fig, ax = plt.subplots(figsize=(6, 4))
PartialDependenceDisplay.from_estimator(
    clf, X_test, features=[0], feature_names=feature_names, ax=ax
)
ax.set_title("PDP cho 'experience'")
plt.tight_layout()
plt.show()
```

Diễn giải: Đường cong PDP cho biết xu hướng trung bình của xác suất dự đoán theo `experience` khi giữ các đặc trưng còn lại như ở từng mẫu trong tập kiểm thử. Nếu đường cong tăng dần rồi bão hòa, điều đó khớp với mô tả ở trên.

🪄 (c) Surrogate models

Use a **simple model (Linear/Tree)** to mimic and explain a **complex black-box model**.

Two types:

1. **Global surrogate**: mimic the entire model
2. **Local surrogate**: mimic locally around one data point

Examples:

- Use a Decision Tree to explain XGBoost predictions globally
- Use LIME to explain a specific instance locally

💬 (d) LIME (Local Interpretable Model-Agnostic Explanations)

Explains **individual predictions** by fitting a **small linear model** around the instance of interest.

Generates perturbed samples near the instance, observes output changes, and trains a simple model locally.

Example:

> 💡 Prediction "employee will quit" → LIME highlights "Overtime = Yes" and "YearsAtCompany < 2" as top contributors.

5. Models for imbalanced classes

⚠️ The problem

When class counts are skewed, models tend to favor the majority class.

🛠️ Remedies

Method	Description
Downsampling	Reduce the number of majority samples
Upsampling	Duplicate or synthesize minority samples
Combine both	Balance by adjusting both sides

Example:

- "Non-fraud": 900 samples
- "Fraud": 100 samples

→ Either upsample "Fraud" to 900 or downsample "Non-fraud" to 100.

6. Techniques for handling imbalanced data

2.1. Downsampling (reduce majority class)

Keep the minority class size, **randomly drop** majority samples.

✅ Pros

- Simple to implement
- Reduces majority bias

❌ Cons

- Information loss (discarding real data)

Python example:

```
from sklearn.utils import resample
import numpy as np

X_majority = X[y == 0]
X_minority = X[y == 1]
```

```
X_majority_downsampled = resample(
    X_majority,
    replace=False,
    n_samples=len(X_minority),
    random_state=42,
)

X_balanced = np.vstack((X_majority_downsampled, X_minority))
y_balanced = np.hstack(([0]*len(X_minority), [1]*len(X_minority)))
```

2.2. Upsampling (increase minority class)

Duplicate or generate new samples for the minority class.

✅ Pros

- Preserve all majority data
- Improve recall for the minority class

❌ Cons

- May cause overfitting

Python example:

```
from sklearn.utils import resample
import numpy as np

X_minority_upsampled = resample(
    X_minority,
    replace=True,
    n_samples=len(X_majority),
    random_state=42,
)

X_balanced = np.vstack((X_majority, X_minority_upsampled))
y_balanced = np.hstack(([0]*len(X_majority), [1]*len(X_majority)))
```

2.3. Hybrid methods (SMOTE, ADASYN, Tomek Links, NearMiss)

🔶 SMOTE (Synthetic Minority Oversampling Technique)

Create synthetic points by **interpolating between a minority point and one of its nearest neighbors (KNN)**.

Variants:

- **SMOTE-Regular**: pick K neighbors at random
- **Borderline-SMOTE**: synthesize near the decision boundary
- **SMOTE-SVM**: use support vectors to synthesize

Formula:

$$\mathbf{x}_{\text{new}} = \mathbf{x}_i + \lambda\,(\mathbf{x}_{z_i} - \mathbf{x}_i), \quad \lambda \in [0,1]$$

Code example:

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_res, y_res = smote.fit_resample(X, y)
```

🔶 ADASYN (Adaptive Synthetic Sampling)

- Generate more synthetic samples **where classification is harder** (more nearby opposite-class points)
- Focus on ambiguous regions

Code example:

```
from imblearn.over_sampling import ADASYN

adasyn = ADASYN(random_state=42)
X_res, y_res = adasyn.fit_resample(X, y)
```

🔶 Advanced undersampling

NearMiss:

- Keep majority samples **closest to the minority** (Euclidean distance)
- Three versions: NearMiss-1, NearMiss-2, NearMiss-3

Tomek Links:

- Remove nearest-opposite-class pairs to **clean the boundary**

Code example:

```
from imblearn.under_sampling import NearMiss, TomekLinks

nm = NearMiss(version=1)
X_res, y_res = nm.fit_resample(X, y)

tl = TomekLinks()
X_res2, y_res2 = tl.fit_resample(X, y)
```

🔶 Combinations (SMOTE + Tomek Links / SMOTE + ENN)

- **SMOTE + Tomek Links**: upsample then clean boundary
- **SMOTE + ENN (Edited Nearest Neighbors)**: remove noisy points after synthesis

```
from imblearn.combine import SMOTETomek, SMOTEENN

smote_tomek = SMOTETomek(random_state=42)
X_res, y_res = smote_tomek.fit_resample(X, y)

smote_enn = SMOTEENN(random_state=42)
X_res2, y_res2 = smote_enn.fit_resample(X, y)
```

🔶 Balanced Bagging ("Blagging")

- Each bag in bagging is constructed so **classes are balanced**
- Improves stability and reduces bias in ensembles

Example:

```
from imblearn.ensemble import BalancedBaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bbc = BalancedBaggingClassifier(
    estimator=DecisionTreeClassifier(),
    n_estimators=10,
    random_state=42,
)
```

2.4. Class weighting

Many models support **class weights**:

$$w_c = \frac{N}{N_c}$$

Where:

- N: total number of samples
- N_c: number of samples in class c

Example with Logistic Regression:

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(class_weight='balanced', max_iter=1000)
```

2.5. Stratified sampling

Use `stratify` to keep **class ratios consistent** between train and test sets.

```
from sklearn.model_selection import train_test_split

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

---

7. Evaluation for imbalanced data

⚠️ **Avoid plain Accuracy** as it can be misleading.

Prefer:

- **Precision, Recall, F1-score**
- **AUC (Area Under the ROC Curve)**
- **Cohen's Kappa**

Code example:

```
from sklearn.metrics import classification_report, roc_auc_score

y_pred = model.predict(X_test)
```

```
print(classification_report(y_test, y_pred))
print("AUC:", roc_auc_score(y_test, model.predict_proba(X_test)[:, 1]))
```

📘 Summary

Technique	Goal	Pros	Cons	Key code
Downsampling	Reduce majority samples	Simple	Information loss	`resample(..., replace=False)`
Upsampling	Increase minority samples	Keep all data	Overfitting	`resample(..., replace=True)`
SMOTE	Synthesize data	Effective	May add noise	`SMOTE()`
ADASYN	Adaptive synthesis	Good near boundary	More complex	`ADASYN()`
NearMiss	Filter majority	Sharper boundary	Possible under-representation	`NearMiss()`
Tomek Links	Clean boundary	Less noise	No data synthesis	`TomekLinks()`
Blagging	Balanced ensemble	Stable	Resource intensive	`BalancedBaggingClassifier()`

Key takeaways

Topic	Essence
Interpretability goals	Understand models, build trust, detect errors
Interpretable models	Linear, Tree, KNN
Post-hoc methods	PFI, PDP, Surrogate, LIME
Imbalanced data	Use upsampling or downsampling to improve performance