Comprehensive Lecture: Handling Imbalanced Data in Machine Learning

1. Understanding Imbalanced Data

1.1 What is Imbalanced Data?

Imbalanced data refers to classification problems where the classes are not represented equally. For example:

- Fraud detection: 99.9% legitimate vs 0.1% fraudulent transactions
- Medical diagnosis: 98% healthy vs 2% disease cases
- Network intrusion detection: 99.9% normal vs 0.1% malicious traffic

1.2 Why is Imbalanced Data Problematic?

- **Algorithmic Bias**: Most ML algorithms optimize for accuracy, biasing predictions toward the majority class
- **Insufficient Learning**: Too few minority samples to learn meaningful patterns
- **Misleading Evaluation**: High accuracy despite poor minority class detection
- **Business Impact**: The minority class is often the class of interest (fraud, disease, etc.)

1.3 Real-World Consequences

If a fraud detection system has 99.9% accuracy but misses most fraudulent transactions, it fails its primary purpose despite appearing successful by traditional metrics.

2. Evaluation Metrics for Imbalanced Data

2.1 Confusion Matrix Fundamentals

```
                Predicted
                Neg     Pos
Actual   Neg    TN      FP
         Pos    FN      TP
```

- **True Positives (TP)**: Correctly predicted positive cases
- **True Negatives (TN)**: Correctly predicted negative cases
- **False Positives (FP)**: Incorrectly predicted positives (Type I error)
- **False Negatives (FN)**: Incorrectly predicted negatives (Type II error)

2.2 Beyond Accuracy

Accuracy = (TP + TN) / (TP + TN + FP + FN)

- Problem: In a dataset with 99% negative class, predicting everything as negative yields 99% accuracy

2.3 Better Metrics

- **Precision = TP / (TP + FP)**: Of all predicted positives, how many are actually positive?
- **Recall (Sensitivity) = TP / (TP + FN)**: Of all actual positives, how many did we predict correctly?
- **Specificity = TN / (TN + FP)**: Of all actual negatives, how many did we predict correctly?
- **F1-Score = 2 × (Precision × Recall) / (Precision + Recall)**: Harmonic mean of precision and recall
- **F-beta Score**: Weighted F-score for when precision or recall is more important

2.4 ROC and PR Curves

- **ROC (Receiver Operating Characteristic)**: Plots TPR (Recall) vs FPR at different thresholds
- **AUC-ROC**: Area under ROC curve; higher is better, with 1 being perfect
- **PR (Precision-Recall) Curve**: Plots Precision vs Recall at different thresholds
- **AUC-PR**: Area under PR curve; better for imbalanced data than AUC-ROC

2.5 Code Example: Evaluation Metrics

python

```python
import numpy as np
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, roc_auc_score, precision_recall_curve,
                             roc_curve, auc, confusion_matrix)
import matplotlib.pyplot as plt

# Example imbalanced predictions and true labels
y_true = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1])  # 25% positive class
y_pred = np.array([0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1])  # Predictions
y_pred_proba = np.array([0.1, 0.2, 0.1, 0.2, 0.3, 0.2, 0.3, 0.4, 0.7, 0.5, 0.8, 0.9])

# Basic metrics
print(f"Accuracy: {accuracy_score(y_true, y_pred):.3f}")
print(f"Precision: {precision_score(y_true, y_pred):.3f}")
print(f"Recall: {recall_score(y_true, y_pred):.3f}")
print(f"F1-Score: {f1_score(y_true, y_pred):.3f}")
print(f"AUC-ROC: {roc_auc_score(y_true, y_pred_proba):.3f}")

# Confusion matrix
cm = confusion_matrix(y_true, y_pred)
print("Confusion Matrix:")
print(cm)

# Visualize ROC curve
fpr, tpr, _ = roc_curve(y_true, y_pred_proba)
plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {auc(fpr, tpr):
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()

# Visualize Precision-Recall curve
precision, recall, _ = precision_recall_curve(y_true, y_pred_proba)
plt.figure(figsize=(10, 6))
plt.plot(recall, precision, color='green', lw=2)
plt.axhline(y=sum(y_true)/len(y_true), color='red', linestyle='--', label=f'Baseline (
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
```

```
plt.legend(loc="lower left")
plt.show()
```

3. Data-Level Approaches

3.1 Resampling Fundamentals

Resampling attempts to balance class distribution before training:

- **Advantages**: Simple, algorithm-agnostic
- **Disadvantages**: May lose information or introduce noise

3.2 Undersampling Techniques

3.2.1 Random Undersampling

Randomly removes majority class samples until classes are balanced.

Code Example:

python

```
from imblearn.under_sampling import RandomUnderSampler
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

# Create imbalanced dataset (90% class 0, 10% class 1)
X, y = make_classification(n_samples=10000, n_features=10, weights=[0.9, 0.1], random_
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=

# Check original class distribution
print("Original training dataset shape:", dict(zip(*np.unique(y_train, return_counts=T

# Apply random undersampling
rus = RandomUnderSampler(random_state=42)
X_train_rus, y_train_rus = rus.fit_resample(X_train, y_train)

# Check resampled class distribution
print("Resampled training dataset shape:", dict(zip(*np.unique(y_train_rus, return_cou

# Train model on resampled data
model_rus = RandomForestClassifier(random_state=42)
model_rus.fit(X_train_rus, y_train_rus)
y_pred_rus = model_rus.predict(X_test)
print("\nClassification Report with Random Undersampling:")
print(classification_report(y_test, y_pred_rus))
```

3.2.2 Informed Undersampling

More sophisticated methods to select which majority samples to remove:

Near Miss: Selects majority samples based on their distance to minority samples.

```python
from imblearn.under_sampling import NearMiss

# Apply Near Miss undersampling (version 1)
nm = NearMiss(version=1)
X_train_nm, y_train_nm = nm.fit_resample(X_train, y_train)

# Check resampled class distribution
print("Near Miss resampled shape:", dict(zip(*np.unique(y_train_nm, return_counts=True

# Train model
model_nm = RandomForestClassifier(random_state=42)
model_nm.fit(X_train_nm, y_train_nm)
y_pred_nm = model_nm.predict(X_test)
print("\nClassification Report with Near Miss:")
print(classification_report(y_test, y_pred_nm))
```

Tomek Links: Removes majority samples that form Tomek links with minority samples.

```python
from imblearn.under_sampling import TomekLinks

# Apply Tomek Links undersampling
tl = TomekLinks()
X_train_tl, y_train_tl = tl.fit_resample(X_train, y_train)

# Check resampled class distribution
print("Tomek Links resampled shape:", dict(zip(*np.unique(y_train_tl, return_counts=Tr

# Train model
model_tl = RandomForestClassifier(random_state=42)
model_tl.fit(X_train_tl, y_train_tl)
y_pred_tl = model_tl.predict(X_test)
print("\nClassification Report with Tomek Links:")
print(classification_report(y_test, y_pred_tl))
```

Condensed Nearest Neighbor (CNN): Iteratively selects samples to preserve decision boundaries.

```python
from imblearn.under_sampling import CondensedNearestNeighbour

# Apply CNN undersampling
cnn = CondensedNearestNeighbour(random_state=42)
X_train_cnn, y_train_cnn = cnn.fit_resample(X_train, y_train)

# Check resampled class distribution
print("CNN resampled shape:", dict(zip(*np.unique(y_train_cnn, return_counts=True))))
```

3.3 Oversampling Techniques

3.3.1 Random Oversampling

Randomly duplicates minority class samples until classes are balanced.

```python
from imblearn.over_sampling import RandomOverSampler

# Apply random oversampling
ros = RandomOverSampler(random_state=42)
X_train_ros, y_train_ros = ros.fit_resample(X_train, y_train)

# Check resampled class distribution
print("Random oversampling shape:", dict(zip(*np.unique(y_train_ros, return_counts=Tru

# Train model
model_ros = RandomForestClassifier(random_state=42)
model_ros.fit(X_train_ros, y_train_ros)
y_pred_ros = model_ros.predict(X_test)
print("\nClassification Report with Random Oversampling:")
print(classification_report(y_test, y_pred_ros))
```

3.3.2 SMOTE (Synthetic Minority Oversampling Technique)

Generates synthetic samples by interpolating between minority class examples.

python

```python
from imblearn.over_sampling import SMOTE
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Visualize original data
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)
plt.figure(figsize=(10, 6))
plt.scatter(X_train_pca[y_train==0, 0], X_train_pca[y_train==0, 1], alpha=0.5, label='
plt.scatter(X_train_pca[y_train==1, 0], X_train_pca[y_train==1, 1], alpha=0.5, label='
plt.title('Original imbalanced dataset (PCA-reduced)')
plt.legend()
plt.show()

# Apply SMOTE
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Visualize SMOTE-resampled data
X_smote_pca = pca.transform(X_train_smote)
plt.figure(figsize=(10, 6))
plt.scatter(X_smote_pca[y_train_smote==0, 0], X_smote_pca[y_train_smote==0, 1], alpha=
plt.scatter(X_smote_pca[y_train_smote==1, 0], X_smote_pca[y_train_smote==1, 1], alpha=
plt.title('SMOTE-resampled dataset (PCA-reduced)')
plt.legend()
plt.show()

# Train model
model_smote = RandomForestClassifier(random_state=42)
model_smote.fit(X_train_smote, y_train_smote)
y_pred_smote = model_smote.predict(X_test)
print("\nClassification Report with SMOTE:")
print(classification_report(y_test, y_pred_smote))
```

3.3.3 ADASYN (Adaptive Synthetic Sampling)

Similar to SMOTE but focuses on generating samples near difficult-to-learn examples.

```python
from imblearn.over_sampling import ADASYN

# Apply ADASYN
adasyn = ADASYN(random_state=42)
X_train_adasyn, y_train_adasyn = adasyn.fit_resample(X_train, y_train)

# Check resampled class distribution
print("ADASYN resampled shape:", dict(zip(*np.unique(y_train_adasyn, return_counts=Tru

# Train model
model_adasyn = RandomForestClassifier(random_state=42)
model_adasyn.fit(X_train_adasyn, y_train_adasyn)
y_pred_adasyn = model_adasyn.predict(X_test)
print("\nClassification Report with ADASYN:")
print(classification_report(y_test, y_pred_adasyn))
```

3.3.4 Borderline-SMOTE

Generates synthetic samples specifically along the decision boundary.

```python
from imblearn.over_sampling import BorderlineSMOTE

# Apply Borderline SMOTE
bsmote = BorderlineSMOTE(random_state=42)
X_train_bsmote, y_train_bsmote = bsmote.fit_resample(X_train, y_train)

# Train model
model_bsmote = RandomForestClassifier(random_state=42)
model_bsmote.fit(X_train_bsmote, y_train_bsmote)
y_pred_bsmote = model_bsmote.predict(X_test)
print("\nClassification Report with Borderline-SMOTE:")
print(classification_report(y_test, y_pred_bsmote))
```

3.4 Hybrid Methods

3.4.1 SMOTETomek

Combines SMOTE with Tomek links removal.

```python
from imblearn.combine import SMOTETomek

# Apply SMOTETomek
smote_tomek = SMOTETomek(random_state=42)
X_train_smt, y_train_smt = smote_tomek.fit_resample(X_train, y_train)

# Train model
model_smt = RandomForestClassifier(random_state=42)
model_smt.fit(X_train_smt, y_train_smt)
y_pred_smt = model_smt.predict(X_test)
print("\nClassification Report with SMOTETomek:")
print(classification_report(y_test, y_pred_smt))
```

3.4.2 SMOTEENN

Combines SMOTE with Edited Nearest Neighbors (ENN).

```python
from imblearn.combine import SMOTEENN

# Apply SMOTEENN
smote_enn = SMOTEENN(random_state=42)
X_train_smenn, y_train_smenn = smote_enn.fit_resample(X_train, y_train)

# Train model
model_smenn = RandomForestClassifier(random_state=42)
model_smenn.fit(X_train_smenn, y_train_smenn)
y_pred_smenn = model_smenn.predict(X_test)
print("\nClassification Report with SMOTEENN:")
print(classification_report(y_test, y_pred_smenn))
```

3.5 Resampling Strategies Comparison

python

```python
import pandas as pd

# Compare all the resampling techniques
methods = ['No resampling', 'Random Under', 'Near Miss', 'Tomek Links',
           'Random Over', 'SMOTE', 'ADASYN', 'Borderline-SMOTE',
           'SMOTETomek', 'SMOTEENN']

# Simple function to evaluate model
def evaluate_model(model, X_test, y_test):
    y_pred = model.predict(X_test)
    return {
        'accuracy': accuracy_score(y_test, y_pred),
        'precision': precision_score(y_test, y_pred),
        'recall': recall_score(y_test, y_pred),
        'f1': f1_score(y_test, y_pred),
        'roc_auc': roc_auc_score(y_test, model.predict_proba(X_test)[:, 1])
    }

# Train base model without resampling
model_base = RandomForestClassifier(random_state=42)
model_base.fit(X_train, y_train)

# Collect results
results = []
results.append(evaluate_model(model_base, X_test, y_test))
results.append(evaluate_model(model_rus, X_test, y_test))
results.append(evaluate_model(model_nm, X_test, y_test))
results.append(evaluate_model(model_tl, X_test, y_test))
results.append(evaluate_model(model_ros, X_test, y_test))
results.append(evaluate_model(model_smote, X_test, y_test))
results.append(evaluate_model(model_adasyn, X_test, y_test))
results.append(evaluate_model(model_bsmote, X_test, y_test))
results.append(evaluate_model(model_smt, X_test, y_test))
results.append(evaluate_model(model_smenn, X_test, y_test))

# Display results
df_results = pd.DataFrame(results, index=methods)
print("\nResampling Strategies Comparison:")
print(df_results)

# Visualize comparison
plt.figure(figsize=(14, 8))
df_results[['precision', 'recall', 'f1', 'roc_auc']].plot(kind='bar', ax=plt.gca())
plt.title('Performance Comparison of Different Resampling Techniques')
plt.ylabel('Score')
plt.xlabel('Resampling Method')
```

```
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

4. Algorithm-Level Approaches

4.1 Cost-Sensitive Learning

Assigns different misclassification costs to different classes.

4.1.1 Class Weights

Many algorithms accept class weights to penalize misclassification of the minority class more heavily.

```python
# Create a basic model without class weights
model_basic = RandomForestClassifier(random_state=42)
model_basic.fit(X_train, y_train)
y_pred_basic = model_basic.predict(X_test)
print("Without class weights:")
print(classification_report(y_test, y_pred_basic))

# Create a model with balanced class weights
model_balanced = RandomForestClassifier(class_weight='balanced', random_state=42)
model_balanced.fit(X_train, y_train)
y_pred_balanced = model_balanced.predict(X_test)
print("\nWith 'balanced' class weights:")
print(classification_report(y_test, y_pred_balanced))

# Create a model with custom class weights
# More weight to minority class (higher penalty for misclassification)
model_custom = RandomForestClassifier(class_weight={0: 1, 1: 10}, random_state=42)
model_custom.fit(X_train, y_train)
y_pred_custom = model_custom.predict(X_test)
print("\nWith custom class weights {0: 1, 1: 10}:")
print(classification_report(y_test, y_pred_custom))
```

4.1.2 Sample Weights

Adjusting importance of individual samples during training.

```python
from sklearn.linear_model import LogisticRegression

# Create sample weights (higher for minority class)
sample_weights = np.ones(len(y_train))
sample_weights[y_train == 1] = 10  # Weight minority samples 10x more

# Train with sample weights
model_sample_weights = LogisticRegression(max_iter=1000)
model_sample_weights.fit(X_train, y_train, sample_weight=sample_weights)
y_pred_sw = model_sample_weights.predict(X_test)
print("\nWith sample weights:")
print(classification_report(y_test, y_pred_sw))
```

4.2 Ensemble Methods

4.2.1 Balanced Random Forest

```python
from imblearn.ensemble import BalancedRandomForestClassifier

# Create a Balanced Random Forest
brf = BalancedRandomForestClassifier(random_state=42)
brf.fit(X_train, y_train)
y_pred_brf = brf.predict(X_test)
print("\nClassification Report with Balanced Random Forest:")
print(classification_report(y_test, y_pred_brf))
```

4.2.2 EasyEnsemble

```python
from imblearn.ensemble import EasyEnsembleClassifier

# Create an EasyEnsemble classifier
eec = EasyEnsembleClassifier(random_state=42)
eec.fit(X_train, y_train)
y_pred_eec = eec.predict(X_test)
print("\nClassification Report with EasyEnsemble:")
print(classification_report(y_test, y_pred_eec))
```

4.2.3 RUSBoost

```python
from imblearn.ensemble import RUSBoostClassifier

# Create a RUSBoost classifier
rusboost = RUSBoostClassifier(random_state=42)
rusboost.fit(X_train, y_train)
y_pred_rusboost = rusboost.predict(X_test)
print("\nClassification Report with RUSBoost:")
print(classification_report(y_test, y_pred_rusboost))
```

4.3 Anomaly Detection

For extreme imbalance, treating the minority class as anomalies.

```python
from sklearn.ensemble import IsolationForest
from sklearn.svm import OneClassSVM

# Isolate the majority class
X_train_maj = X_train[y_train == 0]

# Train Isolation Forest on majority class only
iso_forest = IsolationForest(contamination=0.1, random_state=42)
iso_forest.fit(X_train_maj)

# Predict anomalies (-1) or normal (1)
y_pred_iso = iso_forest.predict(X_test)
# Convert to binary classification (0: normal, 1: anomaly)
y_pred_iso = np.where(y_pred_iso == 1, 0, 1)

print("\nIsolation Forest Anomaly Detection:")
print(classification_report(y_test, y_pred_iso))

# One-Class SVM
one_class_svm = OneClassSVM(nu=0.1, gamma='scale')
one_class_svm.fit(X_train_maj)
y_pred_svm = one_class_svm.predict(X_test)
# Convert predictions (1: normal, -1: anomaly)
y_pred_svm = np.where(y_pred_svm == 1, 0, 1)

print("\nOne-Class SVM Anomaly Detection:")
print(classification_report(y_test, y_pred_svm))
```

5. Threshold Moving & Probability Calibration

5.1 Decision Threshold Optimization

Finding the optimal decision threshold for classification.

python

```python
from sklearn.linear_model import LogisticRegression
import numpy as np

# Train a probability-calibrated classifier
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)

# Get probabilities
y_pred_proba = model.predict_proba(X_test)[:, 1]

# Evaluate at different thresholds
thresholds = np.arange(0.1, 1.0, 0.1)
results = []

for threshold in thresholds:
    y_pred_threshold = (y_pred_proba >= threshold).astype(int)
    precision = precision_score(y_test, y_pred_threshold)
    recall = recall_score(y_test, y_pred_threshold)
    f1 = f1_score(y_test, y_pred_threshold)
    results.append([threshold, precision, recall, f1])

# Display results
df_thresholds = pd.DataFrame(results, columns=['Threshold', 'Precision', 'Recall', 'F1
print("\nPerformance at different thresholds:")
print(df_thresholds)

# Plot precision-recall tradeoff
plt.figure(figsize=(10, 6))
plt.plot(df_thresholds['Threshold'], df_thresholds['Precision'], label='Precision')
plt.plot(df_thresholds['Threshold'], df_thresholds['Recall'], label='Recall')
plt.plot(df_thresholds['Threshold'], df_thresholds['F1'], label='F1')
plt.xlabel('Threshold')
plt.ylabel('Score')
plt.title('Precision-Recall Tradeoff at Different Thresholds')
plt.legend()
plt.grid(True)
plt.show()

# Find optimal threshold for F1
best_idx = df_thresholds['F1'].idxmax()
best_threshold = df_thresholds.loc[best_idx, 'Threshold']
print(f"\nOptimal threshold for F1: {best_threshold}")

# Apply optimal threshold
y_pred_best = (y_pred_proba >= best_threshold).astype(int)
```

```python
print("\nClassification Report with optimal threshold:")
print(classification_report(y_test, y_pred_best))
```

5.2 Probability Calibration

Ensuring predicted probabilities are well-calibrated.

python

```python
from sklearn.calibration import CalibratedClassifierCV, calibration_curve

# Original model
model_uncal = RandomForestClassifier(random_state=42)
model_uncal.fit(X_train, y_train)

# Calibrated model with sigmoid calibration (Platt scaling)
model_cal = CalibratedClassifierCV(model_uncal, method='sigmoid', cv=5)
model_cal.fit(X_train, y_train)

# Get probabilities
y_prob_uncal = model_uncal.predict_proba(X_test)[:, 1]
y_prob_cal = model_cal.predict_proba(X_test)[:, 1]

# Plot calibration curves
plt.figure(figsize=(10, 8))
ax1 = plt.subplot2grid((3, 1), (0, 0), rowspan=2)
ax2 = plt.subplot2grid((3, 1), (2, 0))

ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")

# Calculate calibration curves
fraction_of_positives_uncal, mean_predicted_value_uncal = calibration_curve(
    y_test, y_prob_uncal, n_bins=10)
fraction_of_positives_cal, mean_predicted_value_cal = calibration_curve(
    y_test, y_prob_cal, n_bins=10)

# Plot calibration curves
ax1.plot(mean_predicted_value_uncal, fraction_of_positives_uncal, "s-",
         label="Uncalibrated")
ax1.plot(mean_predicted_value_cal, fraction_of_positives_cal, "s-",
         label="Calibrated")

ax1.set_ylabel("Fraction of positives")
ax1.set_ylim([-0.05, 1.05])
ax1.legend(loc="lower right")
ax1.set_title('Calibration plots (reliability curve)')

# Plot histogram of probabilities
ax2.hist(y_prob_uncal, range=(0, 1), bins=10, label="Uncalibrated",
         histtype="step", lw=2)
ax2.hist(y_prob_cal, range=(0, 1), bins=10, label="Calibrated",
         histtype="step", lw=2)
ax2.set_xlabel("Mean predicted value")
ax2.set_ylabel("Count")
ax2.legend(loc="upper center")
```

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

6. Real-World Case Study: Credit Card Fraud Detection

6.1 Problem Definition

Credit card fraud detection is a classic imbalanced learning problem with fraud transactions typically being less than 0.1% of all transactions.

6.2 Mock Data Generation

python

```python
# Create a more realistic credit card fraud dataset
# With temporal patterns and feature correlations

np.random.seed(42)
n_samples = 100000
n_features = 10

# Generate transaction amounts: most are small, few are large
amounts = np.exp(np.random.normal(4, 1, n_samples))

# Generate time features (hour of day)
hours = np.random.randint(0, 24, n_samples)

# Fraud happens more at night
fraud_prob = np.zeros(n_samples)
fraud_prob = 0.001 + 0.003 * (hours >= 22) + 0.002 * (hours <= 3)
fraud_prob += 0.002 * (amounts > 500)  # Higher fraud risk for large amounts

# Generate labels
y = np.random.binomial(1, fraud_prob)

# Generate correlated features
X = np.random.normal(0, 1, (n_samples, n_features))
# Fraud transactions have slightly different feature distributions
X[y == 1] += np.random.normal(0.5, 0.5, (sum(y), n_features))

# Add hour and amount as features
X = np.hstack((X, hours.reshape(-1, 1), amounts.reshape(-1, 1)))

print(f"Dataset shape: {X.shape}")
print(f"Fraud rate: {sum(y)/len(y)*100:.3f}%")
print(f"Number of fraud cases: {sum(y)}")

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=

# Feature names for interpretation
feature_names = [f'feature_{i}' for i in range(n_features)] + ['hour', 'amount']
```

6.3 Comprehensive Fraud Detection Pipeline

python

```python
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

# Create preprocessing pipeline
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), list(range(X.shape[1])))
    ]
)

# Define multiple approaches to test
approaches = {
    'Baseline': {
        'resampler': None,
        'classifier': RandomForestClassifier(random_state=42)
    },
    'SMOTE + RF': {
        'resampler': SMOTE(random_state=42),
        'classifier': RandomForestClassifier(random_state=42)
    },
    'Weighted RF': {
        'resampler': None,
        'classifier': RandomForestClassifier(class_weight='balanced', random_state=42)
    },
    'Balanced RF': {
        'resampler': None,
        'classifier': BalancedRandomForestClassifier(random_state=42)
    },
    'SMOTE + XGBoost': {
        'resampler': SMOTE(random_state=42),
        'classifier': None  # We'll use a simple logistic regression as proxy
    }
}

# Evaluate all approaches
results_fraud = {}

for approach_name, config in approaches.items():
    print(f"\nEvaluating {approach_name}...")

    # Prepare training data
    if config['resampler'] is not None:
        X_train_resampled, y_train_resampled = config['resampler'].fit_resample(X_trai
    else:
        X_train_resampled, y_train_resampled = X_train, y_train
```

```python
    # Train model
    if config['classifier'] is not None:
        model = Pipeline([
            ('preprocessor', preprocessor),
            ('classifier', config['classifier'])
        ])
    else:  # SMOTE + XGBoost proxy (using LogisticRegression)
        model = Pipeline([
            ('preprocessor', preprocessor),
            ('classifier', LogisticRegression(class_weight='balanced', max_iter=1000))
        ])

    model.fit(X_train_resampled, y_train_resampled)

    # Predictions
    y_pred = model.predict(X_test)
    y_pred_proba = model.predict_proba(X_test)[:, 1]

    # Evaluate
    results_fraud[approach_name] = {
        'accuracy': accuracy_score(y_test, y_pred),
        'precision': precision_score(y_test, y_pred),
        'recall': recall_score(y_test, y_pred),
        'f1': f1_score(y_test, y_pred),
        'roc_auc': roc_auc_score(y_test, y_pred_proba),
        'confusion_matrix': confusion_matrix(y_test, y_pred)
    }

    print(f"Precision: {results_fraud[approach_name]['precision']:.3f}")
    print(f"Recall: {results_fraud[approach_name]['recall']:.3f}")
    print(f"F1-Score: {results_fraud[approach_name]['f1']:.3f}")
    print(f"ROC-AUC: {results_fraud[approach_name]['roc_auc']:.3f}")

# Create comprehensive results comparison
df_fraud_results = pd.DataFrame({
    method: {metric: values[metric] for metric in ['accuracy', 'precision', 'recall',
    for method, values in results_fraud.items()
}).T

print("\n" + "="*50)
print("FRAUD DETECTION RESULTS COMPARISON")
print("="*50)
print(df_fraud_results)

# Visualize results
plt.figure(figsize=(14, 10))
```

```python
# Plot 1: Performance metrics
plt.subplot(2, 2, 1)
df_fraud_results[['precision', 'recall', 'f1', 'roc_auc']].plot(kind='bar', ax=plt.gca
plt.title('Performance Comparison - Fraud Detection')
plt.ylabel('Score')
plt.xticks(rotation=45)
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')

# Plot 2: Precision vs Recall
plt.subplot(2, 2, 2)
plt.scatter(df_fraud_results['recall'], df_fraud_results['precision'], s=100, alpha=0.
for i, method in enumerate(df_fraud_results.index):
    plt.annotate(method, (df_fraud_results['recall'][i], df_fraud_results['precision']
                xytext=(5, 5), textcoords='offset points', fontsize=8)
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision vs Recall Trade-off')
plt.grid(True, alpha=0.3)

# Plot 3: Confusion matrices for best methods
best_methods = df_fraud_results.nlargest(2, 'f1').index

for i, method in enumerate(best_methods):
    plt.subplot(2, 2, 3+i)
    cm = results_fraud[method]['confusion_matrix']
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title(f'Confusion Matrix - {method}')
    plt.colorbar()

    # Add text annotations
    thresh = cm.max() / 2.
    for j in range(cm.shape[0]):
        for k in range(cm.shape[1]):
            plt.text(k, j, format(cm[j, k], 'd'),
                    horizontalalignment="center",
                    color="white" if cm[j, k] > thresh else "black")

    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')

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

6.4 Business Impact Analysis

python

```python
# Business impact analysis for fraud detection
def calculate_business_impact(cm, avg_transaction=100, fraud_loss_multiplier=10):
    """
    Calculate business impact of fraud detection model

    Args:
        cm: confusion matrix [[TN, FP], [FN, TP]]
        avg_transaction: average transaction amount
        fraud_loss_multiplier: how much more a fraud costs vs legitimate transaction
    """
    tn, fp, fn, tp = cm.ravel()

    # Costs
    false_positive_cost = fp * avg_transaction * 0.01  # 1% cost for blocking legitima
    false_negative_cost = fn * avg_transaction * fraud_loss_multiplier  # Full fraud a
    investigation_cost = tp * 10  # Cost to investigate true positives

    total_cost = false_positive_cost + false_negative_cost + investigation_cost

    # Savings (frauds caught)
    fraud_prevented = tp * avg_transaction * fraud_loss_multiplier

    net_benefit = fraud_prevented - total_cost

    return {
        'false_positive_cost': false_positive_cost,
        'false_negative_cost': false_negative_cost,
        'investigation_cost': investigation_cost,
        'total_cost': total_cost,
        'fraud_prevented': fraud_prevented,
        'net_benefit': net_benefit,
        'roi': (net_benefit / total_cost * 100) if total_cost > 0 else 0
    }

print("\n" + "="*50)
print("BUSINESS IMPACT ANALYSIS")
print("="*50)

for method, results in results_fraud.items():
    impact = calculate_business_impact(results['confusion_matrix'])
    print(f"\n{method}:")
    print(f"  Net Benefit: ${impact['net_benefit']:,.2f}")
    print(f"  ROI: {impact['roi']:.1f}%")
    print(f"  False Positive Cost: ${impact['false_positive_cost']:,.2f}")
    print(f"  False Negative Cost: ${impact['false_negative_cost']:,.2f}")
```

7. Advanced Techniques and Considerations

7.1 Deep Learning for Imbalanced Data

7.1.1 Focal Loss

Modifies cross-entropy loss to focus on hard examples.

```python
# Simplified focal loss implementation (conceptual)
def focal_loss(y_true, y_pred, alpha=0.25, gamma=2.0):
    """
    Focal Loss for addressing class imbalance

    FL(pt) = -alpha * (1-pt)^gamma * log(pt)
    """
    # This is a conceptual implementation
    # In practice, you would use frameworks like TensorFlow or PyTorch
    pass

print("Focal Loss concept:")
print("- Reduces loss for well-classified examples")
print("- Focuses training on hard, misclassified examples")
print("- Particularly effective for extreme imbalance")
```

7.1.2 Autoencoders for Anomaly Detection

```python
# Conceptual autoencoder for fraud detection
print("Autoencoder approach for anomaly detection:")
print("1. Train autoencoder on majority class (normal transactions)")
print("2. Measure reconstruction error for new samples")
print("3. High reconstruction error indicates potential fraud")
print("4. Particularly useful for high-dimensional data")
```

7.2 Multi-class Imbalanced Problems

python

```python
# Extend to multi-class imbalanced scenario
from sklearn.datasets import make_classification

# Create multi-class imbalanced dataset
X_multi, y_multi = make_classification(
    n_samples=10000,
    n_features=10,
    n_classes=4,
    weights=[0.7, 0.2, 0.08, 0.02],  # Highly imbalanced
    random_state=42
)

print("Multi-class distribution:")
unique, counts = np.unique(y_multi, return_counts=True)
for cls, count in zip(unique, counts):
    print(f"Class {cls}: {count} samples ({count/len(y_multi)*100:.1f}%)")

# Apply SMOTE for multi-class
X_train_multi, X_test_multi, y_train_multi, y_test_multi = train_test_split(
    X_multi, y_multi, test_size=0.3, random_state=42, stratify=y_multi
)

smote_multi = SMOTE(random_state=42)
X_train_multi_smote, y_train_multi_smote = smote_multi.fit_resample(X_train_multi, y_t

print("\nAfter SMOTE:")
unique, counts = np.unique(y_train_multi_smote, return_counts=True)
for cls, count in zip(unique, counts):
    print(f"Class {cls}: {count} samples")

# Train multi-class model
model_multi = RandomForestClassifier(random_state=42)
model_multi.fit(X_train_multi_smote, y_train_multi_smote)
y_pred_multi = model_multi.predict(X_test_multi)

print("\nMulti-class Classification Report:")
print(classification_report(y_test_multi, y_pred_multi))
```

7.3 Time Series Imbalanced Data

```python
# Special considerations for time series data
print("Time Series Imbalanced Data Considerations:")
print("1. Temporal dependencies: Can't randomly resample")
print("2. Data leakage: Future information shouldn't influence past predictions")
print("3. Concept drift: Patterns may change over time")
print("4. Seasonal patterns: Imbalance may vary by time period")
print("\nRecommended approaches:")
print("- Use time-aware cross-validation")
print("- Apply resampling within time windows")
print("- Consider ensemble methods with temporal awareness")
print("- Monitor model performance over time")
```

8. Best Practices and Guidelines

8.1 Choosing the Right Technique

python

```python
def recommend_technique(imbalance_ratio, dataset_size, problem_type):
    """
    Recommend techniques based on problem characteristics
    """
    recommendations = []

    if imbalance_ratio < 10:  # Mild imbalance
        recommendations.append("Class weights")
        recommendations.append("Threshold tuning")
    elif imbalance_ratio < 100:  # Moderate imbalance
        recommendations.append("SMOTE")
        recommendations.append("Balanced Random Forest")
        recommendations.append("Cost-sensitive learning")
    else:  # Extreme imbalance
        recommendations.append("Anomaly detection")
        recommendations.append("Ensemble methods")
        recommendations.append("Focal loss (if using deep learning)")

    if dataset_size < 1000:  # Small dataset
        recommendations.append("Avoid oversampling (risk of overfitting)")
        recommendations.append("Use cross-validation carefully")

    if problem_type == "fraud_detection":
        recommendations.append("Prioritize recall over precision")
        recommendations.append("Consider business costs in evaluation")
    elif problem_type == "medical_diagnosis":
        recommendations.append("Minimize false negatives")
        recommendations.append("Use ensemble methods for robustness")

    return recommendations

# Example usage
print("Recommendations for fraud detection (ratio 1:1000, large dataset):")
recs = recommend_technique(1000, 100000, "fraud_detection")
for i, rec in enumerate(recs, 1):
    print(f"{i}. {rec}")
```

8.2 Evaluation Strategy

python

```python
def comprehensive_evaluation(y_true, y_pred, y_pred_proba=None):
    """
    Comprehensive evaluation for imbalanced datasets
    """
    results = {}

    # Basic metrics
    results['accuracy'] = accuracy_score(y_true, y_pred)
    results['precision'] = precision_score(y_true, y_pred)
    results['recall'] = recall_score(y_true, y_pred)
    results['f1'] = f1_score(y_true, y_pred)
    results['specificity'] = recall_score(y_true, y_pred, pos_label=0)

    # Advanced metrics
    if y_pred_proba is not None:
        results['roc_auc'] = roc_auc_score(y_true, y_pred_proba)
        precision_curve, recall_curve, _ = precision_recall_curve(y_true, y_pred_proba
        results['pr_auc'] = auc(recall_curve, precision_curve)

    # Class-specific metrics
    cm = confusion_matrix(y_true, y_pred)
    results['confusion_matrix'] = cm

    # Balanced accuracy
    results['balanced_accuracy'] = (results['recall'] + results['specificity']) / 2

    return results

print("Comprehensive evaluation framework includes:")
print("- Multiple metrics (precision, recall, F1, AUC-ROC, AUC-PR)")
print("- Confusion matrix analysis")
print("- Class-specific performance")
print("- Balanced accuracy")
print("- Business impact assessment")
```

8.3 Common Pitfalls and How to Avoid Them

python

```python
print("COMMON PITFALLS IN IMBALANCED LEARNING:")
print()
print("1. RELYING SOLELY ON ACCURACY")
print("   Problem: 99% accuracy might mean 0% minority class detection")
print("   Solution: Use precision, recall, F1, AUC-PR")
print()
print("2. IMPROPER CROSS-VALIDATION")
print("   Problem: Random splits may not preserve class distribution")
print("   Solution: Use stratified cross-validation")
print()
print("3. RESAMPLING BEFORE SPLITTING")
print("   Problem: Data leakage between train and test sets")
print("   Solution: Always split first, then resample training data only")
print()
print("4. IGNORING BUSINESS COSTS")
print("   Problem: False positives and false negatives may have different costs")
print("   Solution: Incorporate business metrics in evaluation")
print()
print("5. OVERFITTING WITH OVERSAMPLING")
print("   Problem: Synthetic samples may not represent real patterns")
print("   Solution: Use cross-validation, ensemble methods, regularization")
print()
print("6. IGNORING TEMPORAL ASPECTS")
print("   Problem: In time series, past and future data get mixed")
print("   Solution: Use time-aware validation, respect temporal order")
```

9. Practical Implementation Checklist

9.1 Step-by-Step Implementation Guide

python

```python
print("IMBALANCED DATA HANDLING CHECKLIST:")
print()
print("□ 1. EXPLORE AND UNDERSTAND THE DATA")
print("   - Calculate imbalance ratio")
print("   - Understand business context")
print("   - Identify if it's truly imbalanced or just rare events")
print()
print("□ 2. CHOOSE APPROPRIATE EVALUATION METRICS")
print("   - Avoid accuracy for severe imbalance")
print("   - Use precision, recall, F1, AUC-PR")
print("   - Consider business-specific metrics")
print()
print("□ 3. SET UP PROPER VALIDATION STRATEGY")
print("   - Use stratified cross-validation")
print("   - Ensure temporal consistency for time series")
print("   - Never resample before splitting")
print()
print("□ 4. TRY MULTIPLE APPROACHES")
print("   - Start with class weights")
print("   - Try resampling techniques (SMOTE, undersampling)")
print("   - Consider ensemble methods")
print("   - Test anomaly detection for extreme cases")
print()
print("□ 5. OPTIMIZE DECISION THRESHOLD")
print("   - Don't use default 0.5 threshold")
print("   - Optimize based on business requirements")
print("   - Consider precision-recall trade-offs")
print()
print("□ 6. VALIDATE WITH BUSINESS STAKEHOLDERS")
print("   - Ensure metrics align with business goals")
print("   - Test edge cases and failure modes")
print("   - Plan for model monitoring and updating")
```

10. Summary and Key Takeaways

10.1 When to Use Each Technique

Technique	Best For	Avoid When
Class Weights	Quick baseline, mild imbalance	Need interpretable probabilities
Random Oversampling	Small datasets, as baseline	Risk of overfitting
SMOTE	Moderate imbalance, tabular data	High-dimensional, categorical data
Random Undersampling	Large datasets, fast training	Small datasets, information loss
Ensemble Methods	Robust performance needed	Interpretability required
Anomaly Detection	Extreme imbalance (>1:1000)	Moderate imbalance
Threshold Tuning	Business cost considerations	Balanced datasets

10.2 Final Recommendations

python

```python
print("KEY TAKEAWAYS:")
print()
print("1. UNDERSTAND YOUR PROBLEM DOMAIN")
print("   - Not all imbalanced datasets require special handling")
print("   - Business context determines success metrics")
print()
print("2. START SIMPLE, THEN ITERATE")
print("   - Begin with class weights or threshold tuning")
print("   - Add complexity only if needed")
print()
print("3. EVALUATION IS CRUCIAL")
print("   - Use multiple metrics")
print("   - Focus on minority class performance")
print("   - Consider business impact")
print()
print("4. COMBINE TECHNIQUES")
print("   - Data-level + algorithm-level approaches")
print("   - Ensemble different strategies")
print()
print("5. MONITOR IN PRODUCTION")
print("   - Class distribution may drift over time")
print("   - Performance may degrade")
print("   - Plan for model updates")
```

Practice Problems

Problem 1: Medical Diagnosis

You're building a model to detect a rare disease that affects 0.5% of the population. False negatives are 10x more costly than false positives. Which techniques would you use and why?

Problem 2: Email Spam Detection

Your spam detection system has 95% accuracy but users complain that too many legitimate emails are marked as spam. The spam rate is about 20%. How would you improve this?

Problem 3: Quality Control

In manufacturing, defective products occur in 2% of cases. You have 100,000 samples. Detection cost is $1 per item, missed defects cost $100 each, and false alarms cost $10 each. Design an optimal strategy.

Problem 4: Time Series Anomaly Detection

You're monitoring server logs where anomalies occur in 0.01% of records. The data has strong temporal patterns. How would you handle this extreme imbalance while respecting temporal dependencies?

Problem 5: Multi-class Imbalance

A customer segmentation problem has 5 classes with distribution [60%, 25%, 10%, 4%, 1%]. You need to predict all classes accurately. What's your approach?

This comprehensive lecture covers the fundamental concepts, practical implementations, and real-world considerations for handling imbalanced data in machine learning. Each technique includes both theoretical understanding and hands-on coding examples to reinforce learning.