Comprehensive Scikit-learn Guide for Machine Learning

Machine Learning Team

December 23, 2025

Contents

1 1. Cheatsheet - Quick Reference

1.1 1.1 Basic Import and Setup

```
# Essential imports
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import accuracy_score, classification_report
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# Basic setup
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

1.2 1.2 Data Preprocessing

```
# Handling missing values
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='mean')  # or 'median', 'most_frequent'
X_imputed = imputer.fit_transform(X)

# Encoding categorical variables
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
le = LabelEncoder()
y_encoded = le.fit_transform(y)

# One-hot encoding
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder(sparse=False)
X_encoded = ohe.fit_transform(X_categorical)

# Scaling features
from sklearn.preprocessing import StandardScaler, MinMaxScaler
scaler = StandardScaler()  # or MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

1.3 1.3 Model Training and Evaluation

```
# Basic model training
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)

```

```
# Cross-validation
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model, X, y, cv=5)

# Grid search
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.1, 1, 10], 'penalty': ['l1', 'l2']}
grid_search = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

1.4 1.4 Model Selection and Validation

```
# Train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# Cross-validation
from sklearn.model_selection import cross_val_score, StratifiedKFold
cv_scores = cross_val_score(model, X, y, cv=5)
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Validation curve
from sklearn.model_selection import validation_curve
train_scores, val_scores = validation_curve(model, X, y, param_name='param
    ', param_range=param_range)
```

1.5 1.5 Performance Metrics

```
# Classification metrics
from sklearn.metrics import accuracy_score, precision_score, recall_score,
     f1_score, roc_auc_score
accuracy = accuracy_score(y_test, predictions)
precision = precision_score(y_test, predictions, average='weighted')
recall = recall_score(y_test, predictions, average='weighted')
f1 = f1_score(y_test, predictions, average='weighted')
roc_auc = roc_auc_score(y_test, predictions_proba)

# Regression metrics
from sklearn.metrics import mean_squared_error, mean_absolute_error,
    r2_score
mse = mean_squared_error(y_test, predictions)
mae = mean_absolute_error(y_test, predictions)
r2 = r2_score(y_test, predictions)

# Confusion matrix
from sklearn.metrics import confusion_matrix, classification_report
cm = confusion_matrix(y_test, predictions)
report = classification_report(y_test, predictions)
```

1.6 1.6 Feature Selection and Engineering

```python
# Feature selection
from sklearn.feature_selection import SelectKBest, f_classif, RFE
selector = SelectKBest(score_func=f_classif, k=10)
X_selected = selector.fit_transform(X, y)

# Recursive Feature Elimination
from sklearn.feature_selection import RFE
rfe = RFE(estimator=LogisticRegression(), n_features_to_select=5)
X_rfe = rfe.fit_transform(X, y)

# Principal Component Analysis
from sklearn.decomposition import PCA
pca = PCA(n_components=0.95)  # Retain 95% variance
X_pca = pca.fit_transform(X)
```

1.7 1.7 Pipeline Construction

```python
# Basic pipeline
from sklearn.pipeline import Pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression())
])
pipeline.fit(X_train, y_train)
predictions = pipeline.predict(X_test)

# Column transformer pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features),
        ('cat', OneHotEncoder(), categorical_features)
    ])
full_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier())
])
```

1.8 1.8 Model Persistence

```python
# Save and load model
import joblib
joblib.dump(model, 'model.pkl')
loaded_model = joblib.load('model.pkl')

# Using pickle
import pickle
with open('model.pkl', 'wb') as f:
    pickle.dump(model, f)
```

```
with open('model.pkl', 'rb') as f:
    loaded_model = pickle.load(f)
```

2 2. Glossary - Terms Defined

2.1 2.1 A

Accuracy - The proportion of correctly predicted instances among all predictions.

Activation Function - A function that transforms the weighted sum of inputs in neural network nodes.

AdaBoost - Adaptive Boosting, an ensemble method that combines weak learners into a strong learner.

Agglomerative Clustering - A hierarchical clustering method that starts with individual points and merges them.

Algorithm - A step-by-step procedure for solving a problem or completing a task.

Anomaly Detection - The identification of rare items, events or observations which raise suspicions.

API (Application Programming Interface) - A set of protocols and tools for building software applications.

Area Under Curve (AUC) - A performance measurement for classification problems at various threshold settings.

Artificial Intelligence - The simulation of human intelligence processes by machines, especially computer systems.

Association Rules - If-then statements that help to show the probability of relationships between data items.

2.2 2.2 B

Bagging - Bootstrap Aggregating, an ensemble method that trains multiple models on different subsets of data.

Bias - Error introduced by approximating a real-world problem, which may be complex, by a simplified model.

Boosting - An ensemble technique which converts weak learners to strong learners sequentially.

Bootstrap - A statistical method for estimating the sampling distribution of an estimator.

Box Plot - A standardized way of displaying the distribution of data based on a five number summary.

2.3 2.3 C

Calibration - The process of adjusting model outputs to better match observed frequencies.

CART (Classification and Regression Trees) - A decision tree learning technique.

Categorical Data - Data that represents categories or groups rather than numerical values.

Classification - A supervised learning task where the goal is to predict discrete class labels.

Classifier - A machine learning model that performs classification tasks.

Clustering - An unsupervised learning task that groups similar data points together.

Coefficient of Determination (R^2) - A statistical measure of how well regression predictions approximate real data.

Collinearity - A phenomenon where two or more predictor variables are highly correlated.

Confidence Interval - A range of values that's likely to contain a population parameter with a certain probability.

Confusion Matrix - A table used to describe the performance of a classification model.

Convergence - The property of a sequence or algorithm approaching a limit or stable state.

Correlation - A statistical measure that describes the extent to which two variables change together.

Cross-Entropy - A measure of the difference between two probability distributions.

Cross-Validation - A model validation technique for assessing how results will generalize to an independent data set.

Curse of Dimensionality - Various phenomena that arise when analyzing data in high-dimensional spaces.

2.4 2.4 D

Data Cleaning - The process of detecting and correcting corrupt or inaccurate records in a dataset.

Data Mining - The process of discovering patterns in large data sets involving methods at the intersection of machine learning, statistics, and database systems.

Data Preprocessing - Techniques used to transform raw data into a format suitable for modeling.

Data Wrangling - The process of cleaning, structuring and enriching raw data into a desired format.

Decision Boundary - A hypersurface that partitions the underlying vector space into sets.

Decision Tree - A tree-like model of decisions and their possible consequences.

Deep Learning - A subset of machine learning based on artificial neural networks with multiple layers.

Density Estimation - The construction of an estimate of an unknown probability density function.

Dependent Variable - The variable being predicted or explained in a statistical model.

Dimensionality Reduction - The process of reducing the number of random variables under consideration.

Discriminant Analysis - A statistical technique used to classify observations into predefined groups.

Distance Metrics - Mathematical functions that define the distance between two data points.

Distribution - A function that shows the possible values a variable can take and how often.

2.5 2.5 E

Ensemble Methods - Techniques that combine multiple learning algorithms to obtain better predictive performance.

Entropy - A measure of uncertainty or disorder in a dataset used in decision tree algorithms.

Epoch - One complete pass through the entire training dataset in machine learning.

Error Rate - The proportion of incorrect predictions made by a model.

Estimator - An object that fits a statistical model to data and provides predictions.

Euclidean Distance - The straight-line distance between two points in Euclidean space.

Evaluation Metrics - Quantitative measures used to assess model performance.

Explained Variance - The proportion of the dataset's variance that is accounted for by the model.

Exploratory Data Analysis (EDA) - An approach to analyzing data sets to summarize their main characteristics.

Exponential Smoothing - A time series forecasting method for univariate data.

2.6 2.6 F

F1-Score - The harmonic mean of precision and recall used in classification evaluation.

False Negative - An outcome where the model incorrectly predicts the negative class.

False Positive - An outcome where the model incorrectly predicts the positive class.

Feature Engineering - The process of using domain knowledge to create new features from raw data.

Feature Extraction - The process of reducing the dimensionality of data by transforming it.

Feature Importance - A technique to determine which features contribute most to model predictions.

Feature Scaling - Methods used to standardize the range of independent variables or features.

Feature Selection - The process of selecting a subset of relevant features for model construction.

Fitting - The process of training a model on data to learn the underlying patterns.

Forest - A collection of decision trees used in ensemble methods like Random Forest.

Forward Selection - A feature selection method that starts with no features and adds them one by one.

Fuzzy Logic - A form of many-valued logic in which the truth values of variables may be any real number between 0 and 1.

2.7 2.7 G

Gaussian Distribution - A probability distribution that is symmetric about the mean, showing that data near the mean are more frequent.

Generalization - The ability of a model to perform well on unseen data.

Gradient Descent - An optimization algorithm used to minimize a function by iteratively moving in the direction of steepest descent.

Grid Search - A hyperparameter tuning technique that exhaustively considers all parameter combinations.

Gini Impurity - A measure of how often a randomly chosen element would be incorrectly labeled.

Goodness of Fit - A statistical test describing how well a regression line fits a set of observations.

2.8 2.8 H

Hierarchical Clustering - A clustering method that builds a hierarchy of clusters.

Holdout Validation - A validation technique that splits data into training and testing sets.

Homoscedasticity - A condition where the variance of the error term is constant across all levels of the independent variables.

Hyperparameter - A parameter whose value is set before the learning process begins.

Hypothesis Testing - A statistical method that uses sample data to evaluate a hypothesis about a population parameter.

2.9 2.9 I

Imputation - The process of replacing missing data with substituted values.

Independent Variable - A variable that is manipulated or controlled in an experiment.

Information Gain - The reduction in entropy or surprise by transforming a dataset.

Initialization - The process of setting initial values for model parameters before training.

Instance-Based Learning - A family of learning algorithms that compare new problem instances with instances seen in training.

Interpolation - The estimation of unknown values that fall between known values.

Iteration - A single execution of the training process on a batch of data.

Iris Dataset - A classic dataset in machine learning containing measurements of iris flowers.

2.10 2.10 J

Jaccard Index - A statistic used for comparing the similarity and diversity of sample sets.

Joint Probability - The probability of two events occurring simultaneously.

2.11 2.11 K

K-Means Clustering - An unsupervised learning algorithm that partitions data into k clusters.

K-Nearest Neighbors (KNN) - A non-parametric method used for classification and regression.

Kernel - A function used in support vector machines to transform data into higher-dimensional space.

K-Fold Cross-Validation - A resampling procedure used to evaluate machine learning models.

2.12 2.12 L

Label Encoding - The process of converting categorical labels into numeric form.

Lasso Regression - A regression analysis method that performs both variable selection and regularization.

Latent Variables - Variables that are not directly observed but are inferred from other variables.

Learning Rate - A hyperparameter that controls how much to change the model in response to the estimated error.

Leave-One-Out Cross-Validation - A special case of cross-validation where each observation is used as a test set once.

Linear Discriminant Analysis (LDA) - A method used in statistics and machine learning to find a linear combination of features.

Linear Regression - A linear approach to modeling the relationship between a scalar response and explanatory variables.

Logistic Regression - A statistical model that uses a logistic function to model a binary dependent variable.

Loss Function - A function that maps an event or values of one or more variables onto a real number representing cost.

2.13 2.13 M

Machine Learning - A method of data analysis that automates analytical model building.

Manhattan Distance - The distance between two points measured along axes at right angles.

Margin - The distance between the hyperplane and the observations closest to the hyperplane in SVM.

Matrix Factorization - A class of techniques used to reduce the dimensionality of data.

Maximum Likelihood Estimation - A method of estimating the parameters of a probability distribution.

Mean Absolute Error (MAE) - The average of the absolute differences between predicted and actual values.

Mean Squared Error (MSE) - The average of the squares of the differences between predicted and actual values.

Median - The middle value in a sorted list of numbers.

Memory-Based Learning - Learning algorithms that store training data and use it directly for prediction.

Metric Learning - The process of learning a distance function over objects.

Mini-Batch - A small subset of the training data used in one iteration of model training.

Mode - The value that appears most frequently in a data set.

Model Selection - The process of selecting the best model from a set of candidate models.

Multicollinearity - The occurrence of high correlation among two or more independent variables.

Multinomial Distribution - A generalization of the binomial distribution for experiments with more than two outcomes.

2.14 2.14 N

Naive Bayes - A classification technique based on Bayes' theorem with an assumption of independence.

Neural Network - A computing system inspired by the biological neural networks that constitute animal brains.

Noise - Random variation in data that does not carry meaningful information.

Normalization - The process of scaling individual samples to have unit norm.

Numpy - A fundamental package for scientific computing in Python.

Nystroem Method - A technique for approximating the feature map of an RBF kernel.

2.15 2.15 O

One-Hot Encoding - A process of converting categorical data into a binary vector representation.

One-Class SVM - A variant of SVM used for novelty detection.

One-Vs-All - A strategy for multi-class classification using binary classifiers.

One-Vs-One - A strategy for multi-class classification that trains a binary classifier for each pair of classes.

Outlier - An observation point that is distant from other observations.

Overfitting - When a model learns the detail and noise in the training data to the extent that it negatively impacts performance on new data.

2.16 2.16 P

Pandas - A software library for data manipulation and analysis in Python.

Parameter - A configuration variable that is internal to the model and whose value can be estimated from data.

Partial Fit - A method that allows incremental learning on chunks of data.

Perceptron - An algorithm for supervised learning of binary classifiers.

Pipeline - A sequence of data processing components chained together.

Polynomial Features - Features created by raising existing features to various powers.

Pooling - An operation in neural networks that reduces the spatial dimensions of the input.

Precision - The ratio of correctly predicted positive observations to the total predicted positives.

Principal Component Analysis (PCA) - A statistical procedure that uses an orthogonal transformation to convert correlated variables into linearly uncorrelated variables.

Probability - A measure of the likelihood that an event will occur.

Probability Calibration - The process of transforming classifier scores into calibrated probabilities.

Python - A high-level programming language widely used in data science and machine learning.

2.17 2.17 Q

Quantile - Values that divide a probability distribution into continuous intervals with equal probabilities.

Quantization - The process of mapping input values from a large set to output values in a smaller set.

2.18 2.18 R

Random Forest - An ensemble learning method that operates by constructing multiple decision trees.

Random Sampling - A sampling method where each member of the population has an equal chance of being selected.

Random State - A parameter that controls the shuffling applied to the data before applying the split.

Range - The difference between the highest and lowest values in a dataset.

Recall - The ratio of correctly predicted positive observations to all observations in actual class.

Regression - A supervised learning task where the goal is to predict continuous values.

Regularization - A technique used to prevent overfitting by adding a penalty term to the loss function.

Reinforcement Learning - An area of machine learning concerned with how intelligent agents ought to take actions.

Resampling - A statistical method of repeatedly drawing samples from a dataset.

Ridge Regression - A technique for analyzing multiple regression data that suffer from multicollinearity.

ROC Curve - A graphical plot that illustrates the diagnostic ability of a binary classifier system.

Root Mean Square Error (RMSE) - The square root of the average of squared differences between prediction and actual observation.

2.19 2.19 S

Sample - A subset of a statistical population in which a survey or census is conducted.

Sampling - The process of selecting a subset of individuals from a population to estimate characteristics of the whole population.

Scikit-learn - A free software machine learning library for Python.

Score - A function that evaluates the quality of a model's predictions.

Scoring - The process of applying a trained model to new data to make predictions.

Sensitivity - The proportion of actual positives that are correctly identified.

Separability - The ability to distinguish between different classes in a classification problem.

Shuffle - The process of randomly reordering data.

Silhouette Score - A measure of how similar an object is to its own cluster compared to other clusters.

Skewness - A measure of the asymmetry of the probability distribution of a real-valued random variable.

Smoothing - Techniques used to reduce noise in data.

Specificity - The proportion of actual negatives that are correctly identified.

Spline - A special function defined piecewise by polynomials.

Standard Deviation - A measure of the amount of variation or dispersion of a set of values.

Standardization - The process of rescaling data to have a mean of 0 and standard deviation of 1.

Statistical Learning - The framework for studying and understanding the performance of machine learning algorithms.

Stratified Sampling - A method of sampling that involves dividing a population into homogeneous subgroups.

Support Vector Machine (SVM) - A supervised learning model that analyzes data for classification and regression analysis.

2.20 2.20 T

Target Variable - The variable that is being predicted in a supervised learning problem.

Test Set - A dataset used to provide an unbiased evaluation of a final model fit on the training dataset.

Time Series - A series of data points indexed in time order.

Training Set - A dataset used to train a machine learning model.

Tree Pruning - The process of reducing the size of decision trees to prevent overfitting.

True Negative - An outcome where the model correctly predicts the negative class.

True Positive - An outcome where the model correctly predicts the positive class.

Type I Error - The incorrect rejection of a true null hypothesis.

Type II Error - The failure to reject a false null hypothesis.

2.21 2.21 U

Underfitting - When a model is too simple to capture the underlying structure of the data.

Unsupervised Learning - A type of machine learning that looks for previously undetected patterns in a data set.

Upsampling - A technique used to increase the frequency of a signal or to balance imbalanced datasets.

2.22 2.22 V

Validation Curve - A plot that shows the training and validation scores for different parameter values.

Validation Set - A dataset used to tune hyperparameters and prevent overfitting.

Variance - A measure of how far a set of numbers are spread out from their average value.

Variance Inflation Factor (VIF) - A measure of multicollinearity in regression analysis.

Vector - A mathematical object that has both magnitude and direction.

Voting Classifier - An ensemble method that combines several classifiers and uses majority voting.

2.23 2.23 W

Weight - A parameter in a neural network that transforms input data within the network's hidden layers.

Weight Decay - A regularization technique that adds a penalty term to the loss function.

Within-Cluster Sum of Squares - A measure used in clustering to evaluate the compactness of clusters.

2.24 2.24 X

XGBoost - An optimized distributed gradient boosting library designed to be highly efficient, flexible and portable.

2.25 2.25 Y

Yield - A keyword in Python used to create generator functions.

2.26 2.26 Z

Z-Score - A statistical measurement that describes a value's relationship to the mean of a group of values.

3 3. Code Snippets

3.1 3.1 Data Loading and Preprocessing

3.1.1 3.1.1 Loading Data

```
# Load from CSV
import pandas as pd
data = pd.read_csv('data.csv')

# Load from Excel
data = pd.read_excel('data.xlsx', sheet_name='Sheet1')

# Load from JSON
data = pd.read_json('data.json')

# Load from database
import sqlite3
```

```
conn = sqlite3.connect('database.db')
data = pd.read_sql_query('SELECT * FROM table_name', conn)

# Load from URL
data = pd.read_csv('https://example.com/data.csv')

# Load built-in datasets
from sklearn.datasets import load_iris, load_boston, load_breast_cancer
iris = load_iris()
boston = load_boston()  # Note: deprecated in newer versions
cancer = load_breast_cancer()
```

Listing 1: Load data from various sources

3.1.2 3.1.2 Data Exploration

```
# Basic dataset information
print(data.shape)
print(data.info())
print(data.describe())
print(data.head())
print(data.tail())

# Check for missing values
print(data.isnull().sum())
print(data.isnull().mean() * 100)  # Percentage of missing values

# Check data types
print(data.dtypes)

# Check unique values in categorical columns
for col in data.select_dtypes(include=['object']).columns:
    print(f {col}: {data[col].nunique()} unique values )

# Correlation matrix
import seaborn as sns
import matplotlib.pyplot as plt
correlation_matrix = data.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.show()
```

Listing 2: Explore dataset characteristics

3.1.3 3.1.3 Handling Missing Values

```
from sklearn.impute import SimpleImputer, KNNImputer, IterativeImputer
import pandas as pd
import numpy as np

# Simple imputation strategies
# Mean imputation
mean_imputer = SimpleImputer(strategy='mean')
```

```
data_mean_imputed = mean_imputer.fit_transform(data_numeric)

# Median imputation
median_imputer = SimpleImputer(strategy='median')
data_median_imputed = median_imputer.fit_transform(data_numeric)

# Mode imputation for categorical
mode_imputer = SimpleImputer(strategy='most_frequent')
data_mode_imputed = mode_imputer.fit_transform(data_categorical)

# Forward fill for time series
data_ffill = data.fillna(method='ffill')

# Backward fill for time series
data_bfill = data.fillna(method='bfill')

# KNN imputation
knn_imputer = KNNImputer(n_neighbors=5)
data_knn_imputed = knn_imputer.fit_transform(data_numeric)

# Advanced imputation with IterativeImputer
from sklearn.experimental import enable_iterative_imputer
iterative_imputer = IterativeImputer(random_state=42)
data_iterative_imputed = iterative_imputer.fit_transform(data_numeric)

# Custom imputation function
def custom_impute(df):
    df_imputed = df.copy()
    for column in df.columns:
        if df[column].dtype in ['int64', 'float64']:
            # For numeric columns, use median
            df_imputed[column].fillna(df[column].median(), inplace=True)
        else:
            # For categorical columns, use mode
            mode_value = df[column].mode()
            if len(mode_value) > 0:
                df_imputed[column].fillna(mode_value[0], inplace=True)
    return df_imputed

data_custom_imputed = custom_impute(data)
```

Listing 3: Handle missing data comprehensively

3.1.4 3.1.4 Feature Encoding

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder,
    OrdinalEncoder
from sklearn.compose import ColumnTransformer
import pandas as pd

# Label encoding for target variable
le = LabelEncoder()
y_encoded = le.fit_transform(y)
```

```

# One-hot encoding for categorical features
ohe = OneHotEncoder(sparse=False, drop='first')  # drop='first' to avoid multicollinearity
X_encoded = ohe.fit_transform(X_categorical)

# Ordinal encoding for ordered categorical variables
oe = OrdinalEncoder(categories=[['Low', 'Medium', 'High']])
X_ordinal = oe.fit_transform(X_ordinal_categorical)

# Column transformer for mixed data types
numeric_features = ['age', 'income', 'score']
categorical_features = ['gender', 'education', 'city']

preprocessor = ColumnTransformer(
    transformers=[
        ('num', 'passthrough', numeric_features),
        ('cat', OneHotEncoder(drop='first'), categorical_features)
    ])

X_processed = preprocessor.fit_transform(X)

# Pandas get_dummies (alternative to OneHotEncoder)
X_dummies = pd.get_dummies(X, columns=categorical_features, drop_first=True)

# Target encoding (mean encoding)
def target_encode(df, categorical_col, target_col):
    encoding_map = df.groupby(categorical_col)[target_col].mean().to_dict()
    return df[categorical_col].map(encoding_map)

X_target_encoded = target_encode(data, 'category', 'target')
```

Listing 4: Encode categorical variables

3.1.5 3.1.5 Feature Scaling

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler, Normalizer
import numpy as np

# Standardization (Z-score normalization)
scaler = StandardScaler()
X_standardized = scaler.fit_transform(X)

# Min-Max scaling (0-1 range)
minmax_scaler = MinMaxScaler()
X_minmax = minmax_scaler.fit_transform(X)

# Robust scaling (using median and IQR)
robust_scaler = RobustScaler()
X_robust = robust_scaler.fit_transform(X)
```

```

# Normalization (unit vector)
normalizer = Normalizer()
X_normalized = normalizer.fit_transform(X)

# Custom scaling function
def custom_scale(X, method='standard'):
    if method == 'standard':
        return (X - np.mean(X, axis=0)) / np.std(X, axis=0)
    elif method == 'minmax':
        return (X - np.min(X, axis=0)) / (np.max(X, axis=0) - np.min(X,
   axis=0))
    elif method == 'robust':
        median = np.median(X, axis=0)
        q75, q25 = np.percentile(X, [75, 25], axis=0)
        iqr = q75 - q25
        return (X - median) / iqr

# Apply inverse transform to get original scale
X_original = scaler.inverse_transform(X_standardized)
```

Listing 5: Scale numerical features

3.1.6 3.1.6 Feature Engineering

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# Create polynomial features
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(X)

# Create interaction features
def create_interactions(df, feature_pairs):
    df_new = df.copy()
    for feat1, feat2 in feature_pairs:
        df_new[f'{feat1}_{feat2}_interaction'] = df[feat1] * df[feat2]
    return df_new

interaction_pairs = [('feature1', 'feature2'), ('feature1', 'feature3')]
data_with_interactions = create_interactions(data, interaction_pairs)

# Create binned features
data['age_group'] = pd.cut(data['age'], bins=[0, 18, 35, 50, 100],
                           labels=['Child', 'Young', 'Adult', 'Senior'])

# Create date-based features
data['date'] = pd.to_datetime(data['date'])
data['year'] = data['date'].dt.year
data['month'] = data['date'].dt.month
data['day_of_week'] = data['date'].dt.dayofweek
data['is_weekend'] = data['day_of_week'].isin([5, 6]).astype(int)
```

```

# Create ratio features
data['income_to_age_ratio'] = data['income'] / (data['age'] + 1)  # Add 1
    to avoid division by zero

# Create binary flags
data['high_income'] = (data['income'] > data['income'].median()).astype(
    int)

# Create rolling window features (for time series)
data['rolling_mean_7'] = data['value'].rolling(window=7).mean()
data['rolling_std_7'] = data['value'].rolling(window=7).std()

# Create lag features
data['lag_1'] = data['value'].shift(1)
data['lag_7'] = data['value'].shift(7)

# Create cyclical features (for periodic data)
data['month_sin'] = np.sin(2 * np.pi * data['month'] / 12)
data['month_cos'] = np.cos(2 * np.pi * data['month'] / 12)
```

Listing 6: Create new features from existing data

3.2 3.2 Model Training and Evaluation

3.2.1 3.2.1 Classification Models

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# Logistic Regression
lr = LogisticRegression(random_state=42)
lr.fit(X_train, y_train)
lr_pred = lr.predict(X_test)
lr_proba = lr.predict_proba(X_test)

# Random Forest
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)
rf_pred = rf.predict(X_test)
rf_proba = rf.predict_proba(X_test)

```

```
# Support Vector Machine
svm = SVC(probability=True, random_state=42)
svm.fit(X_train, y_train)
svm_pred = svm.predict(X_test)
svm_proba = svm.predict_proba(X_test)

# Gradient Boosting
gb = GradientBoostingClassifier(random_state=42)
gb.fit(X_train, y_train)
gb_pred = gb.predict(X_test)
gb_proba = gb.predict_proba(X_test)

# Naive Bayes
nb = GaussianNB()
nb.fit(X_train, y_train)
nb_pred = nb.predict(X_test)
nb_proba = nb.predict_proba(X_test)

# K-Nearest Neighbors
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
knn_pred = knn.predict(X_test)
knn_proba = knn.predict_proba(X_test)

# Decision Tree
dt = DecisionTreeClassifier(random_state=42)
dt.fit(X_train, y_train)
dt_pred = dt.predict(X_test)
dt_proba = dt.predict_proba(X_test)

# Evaluate models
models = {
    'Logistic Regression': lr_pred,
    'Random Forest': rf_pred,
    'SVM': svm_pred,
    'Gradient Boosting': gb_pred,
    'Naive Bayes': nb_pred,
    'KNN': knn_pred,
    'Decision Tree': dt_pred
}

for name, predictions in models.items():
    accuracy = accuracy_score(y_test, predictions)
    print(f {name} Accuracy: {accuracy:.4f} )
    print(f {name} Classification Report: )
    print(classification_report(y_test, predictions))
    print( -  * 50)
```

Listing 7: Train various classification models

3.2.2 3.2.2 Regression Models

```
from sklearn.linear_model import LinearRegression, Ridge, Lasso,
    ElasticNet
```

```
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Linear Regression
lr = LinearRegression()
lr.fit(X_train, y_train)
lr_pred = lr.predict(X_test)

# Ridge Regression
ridge = Ridge(alpha=1.0)
ridge.fit(X_train, y_train)
ridge_pred = ridge.predict(X_test)

# Lasso Regression
lasso = Lasso(alpha=1.0)
lasso.fit(X_train, y_train)
lasso_pred = lasso.predict(X_test)

# Elastic Net
elastic = ElasticNet(alpha=1.0, l1_ratio=0.5)
elastic.fit(X_train, y_train)
elastic_pred = elastic.predict(X_test)

# Random Forest Regressor
rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)
rf_pred = rf.predict(X_test)

# Support Vector Regression
svr = SVR(kernel='rbf')
svr.fit(X_train, y_train)
svr_pred = svr.predict(X_test)

# Gradient Boosting Regressor
gb = GradientBoostingRegressor(random_state=42)
gb.fit(X_train, y_train)
gb_pred = gb.predict(X_test)

# K-Nearest Neighbors Regressor
knn = KNeighborsRegressor(n_neighbors=5)
knn.fit(X_train, y_train)
knn_pred = knn.predict(X_test)

# Decision Tree Regressor
dt = DecisionTreeRegressor(random_state=42)
dt.fit(X_train, y_train)
dt_pred = dt.predict(X_test)

# Evaluate regression models
```

```
def evaluate_regression(y_true, y_pred, model_name):
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    mae = mean_absolute_error(y_true, y_pred)
    r2 = r2_score(y_true, y_pred)

    print(f {model_name}: )
    print(f   RMSE: {rmse:.4f} )
    print(f   MAE: {mae:.4f} )
    print(f   R  : {r2:.4f} )
    print( -  * 30)

# Evaluate all models
models_regression = {
    'Linear Regression': lr_pred,
    'Ridge': ridge_pred,
    'Lasso': lasso_pred,
    'Elastic Net': elastic_pred,
    'Random Forest': rf_pred,
    'SVR': svr_pred,
    'Gradient Boosting': gb_pred,
    'KNN': knn_pred,
    'Decision Tree': dt_pred
}

for name, predictions in models_regression.items():
    evaluate_regression(y_test, predictions, name)
```

Listing 8: Train various regression models

3.2.3 3.2.3 Cross-Validation

```
from sklearn.model_selection import cross_val_score, StratifiedKFold,
    KFold
from sklearn.model_selection import LeaveOneOut, ShuffleSplit
import numpy as np

# Basic cross-validation
cv_scores = cross_val_score(model, X, y, cv=5)
print(f CV Scores: {cv_scores} )
print(f Mean CV Score: {cv_scores.mean():.4f} (+/- {cv_scores.std() * 2:.4
    f}) )

# Stratified K-Fold (for classification)
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
skf_scores = cross_val_score(model, X, y, cv=skf)
print(f Stratified CV Scores: {skf_scores} )

# K-Fold (for regression)
kf = KFold(n_splits=5, shuffle=True, random_state=42)
kf_scores = cross_val_score(model, X, y, cv=kf, scoring='
    neg_mean_squared_error')
print(f KFold CV Scores: {-kf_scores} )
```

```

# Leave-One-Out Cross-Validation
loo = LeaveOneOut()
loo_scores = cross_val_score(model, X_small, y_small, cv=loo)
print(f LOO CV Score: {loo_scores.mean():.4f} )

# Shuffle Split
ss = ShuffleSplit(n_splits=10, test_size=0.2, random_state=42)
ss_scores = cross_val_score(model, X, y, cv=ss)
print(f ShuffleSplit CV Score: {ss_scores.mean():.4f} )

# Custom cross-validation with multiple metrics
from sklearn.model_selection import cross_validate
scoring = ['accuracy', 'precision_macro', 'recall_macro']
cv_results = cross_validate(model, X, y, cv=5, scoring=scoring)
for metric in scoring:
    scores = cv_results[f'test_{metric}']
    print(f {metric}: {scores.mean():.4f} (+/- {scores.std() * 2:.4f}) )

# Time Series Cross-Validation
from sklearn.model_selection import TimeSeriesSplit
tscv = TimeSeriesSplit(n_splits=5)
tscv_scores = cross_val_score(model, X_time_series, y_time_series, cv=tscv
    )
print(f Time Series CV Score: {tscv_scores.mean():.4f} )
```

Listing 9: Implement cross-validation strategies

3.2.4 3.2.4 Hyperparameter Tuning

```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.model_selection import validation_curve, learning_curve
from scipy.stats import randint, uniform
import numpy as np

# Grid Search
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 7, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

grid_search = GridSearchCV(
    estimator=RandomForestClassifier(random_state=42),
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1,
    verbose=1
)

grid_search.fit(X_train, y_train)
```

```
print(f Best parameters: {grid_search.best_params_} )
print(f Best cross-validation score: {grid_search.best_score_:.4f} )

# Randomized Search
param_dist = {
    'C': uniform(0.1, 10),
    'gamma': uniform(0.001, 1),
    'kernel': ['rbf', 'poly', 'sigmoid']
}

random_search = RandomizedSearchCV(
    estimator=SVC(random_state=42),
    param_distributions=param_dist,
    n_iter=100,
    cv=5,
    scoring='accuracy',
    n_jobs=-1,
    random_state=42
)

random_search.fit(X_train, y_train)
print(f Best parameters: {random_search.best_params_} )
print(f Best cross-validation score: {random_search.best_score_:.4f} )

# Validation Curve
from sklearn.model_selection import validation_curve
param_range = [1, 10, 100, 1000]
train_scores, val_scores = validation_curve(
    SVC(), X, y, param_name='C', param_range=param_range, cv=5
)

train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
val_mean = np.mean(val_scores, axis=1)
val_std = np.std(val_scores, axis=1)

# Nested Cross-Validation
from sklearn.model_selection import cross_val_score
nested_scores = cross_val_score(grid_search, X, y, cv=5)
print(f Nested CV Score: {nested_scores.mean():.4f} (+/- {nested_scores.
    std() * 2:.4f}) )

# Bayesian Optimization (using scikit-optimize)
# pip install scikit-optimize
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer

search_space = {
    'n_estimators': Integer(10, 1000),
    'max_depth': Integer(1, 50),
    'min_samples_split': Integer(2, 20),
    'min_samples_leaf': Integer(1, 10)
}

```

```
bayes_search = BayesSearchCV(
    estimator=RandomForestClassifier(random_state=42),
    search_spaces=search_space,
    n_iter=50,
    cv=5,
    scoring='accuracy',
    n_jobs=-1,
    random_state=42
)

bayes_search.fit(X_train, y_train)
print(f Bayesian Optimization Best Score: {bayes_search.best_score_:.4f} )
```

Listing 10: Optimize model hyperparameters

3.2.5 3.2.5 Model Evaluation Metrics

```
from sklearn.metrics import (accuracy_score, precision_score, recall_score
    , f1_score,
                             roc_auc_score, roc_curve,
    precision_recall_curve,
                             confusion_matrix, classification_report,
                             mean_squared_error, mean_absolute_error,
    r2_score,
                             silhouette_score, adjusted_rand_score)
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Classification Metrics
def evaluate_classification(y_true, y_pred, y_proba=None):
    # Basic metrics
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')

    print(f Accuracy: {accuracy:.4f} )
    print(f Precision: {precision:.4f} )
    print(f Recall: {recall:.4f} )
    print(f F1-Score: {f1:.4f} )

    # Confusion Matrix
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title('Confusion Matrix')
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.show()

    # Classification Report
    print( \nClassification Report: )
```

```
    print(classification_report(y_true, y_pred))

    # ROC Curve and AUC (for binary classification)
    if y_proba is not None and len(np.unique(y_true)) == 2:
        fpr, tpr, _ = roc_curve(y_true, y_proba[:, 1])
        auc = roc_auc_score(y_true, y_proba[:, 1])

        plt.figure(figsize=(8, 6))
        plt.plot(fpr, tpr, label=f'ROC Curve (AUC = {auc:.4f})')
        plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier')
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('ROC Curve')
        plt.legend()
        plt.show()

        print(f AUC Score: {auc:.4f} )

    # Precision-Recall Curve
    if y_proba is not None and len(np.unique(y_true)) == 2:
        precision_vals, recall_vals, _ = precision_recall_curve(y_true,
   y_proba[:, 1])

        plt.figure(figsize=(8, 6))
        plt.plot(recall_vals, precision_vals)
        plt.xlabel('Recall')
        plt.ylabel('Precision')
        plt.title('Precision-Recall Curve')
        plt.show()

# Regression Metrics
def evaluate_regression(y_true, y_pred):
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    mae = mean_absolute_error(y_true, y_pred)
    r2 = r2_score(y_true, y_pred)

    print(f Mean Squared Error: {mse:.4f} )
    print(f Root Mean Squared Error: {rmse:.4f} )
    print(f Mean Absolute Error: {mae:.4f} )
    print(f  R   Score: {r2:.4f} )

    # Residuals plot
    residuals = y_true - y_pred
    plt.figure(figsize=(10, 4))

    plt.subplot(1, 2, 1)
    plt.scatter(y_pred, residuals)
    plt.xlabel('Predicted Values')
    plt.ylabel('Residuals')
    plt.title('Residuals vs Predicted')
    plt.axhline(y=0, color='r', linestyle='--')

    plt.subplot(1, 2, 2)
```

```
    plt.hist(residuals, bins=30)
    plt.xlabel('Residuals')
    plt.ylabel('Frequency')
    plt.title('Residuals Distribution')

    plt.tight_layout()
    plt.show()

# Clustering Metrics
def evaluate_clustering(X, labels):
    if len(np.unique(labels)) > 1:
        silhouette = silhouette_score(X, labels)
        print(f Silhouette Score: {silhouette:.4f} )

    # Plot clusters
    if X.shape[1] >= 2:
        plt.figure(figsize=(8, 6))
        plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
        plt.title('Clustering Results')
        plt.colorbar()
        plt.show()

# Custom scoring function
from sklearn.metrics import make_scorer

def custom_f1_score(y_true, y_pred):
    return f1_score(y_true, y_pred, average='macro')

custom_scorer = make_scorer(custom_f1_score)

# Multi-class specific metrics
def multiclass_evaluation(y_true, y_pred):
    # Per-class metrics
    precision_per_class = precision_score(y_true, y_pred, average=None)
    recall_per_class = recall_score(y_true, y_pred, average=None)
    f1_per_class = f1_score(y_true, y_pred, average=None)

    classes = np.unique(y_true)
    for i, class_label in enumerate(classes):
        print(f Class {class_label}: )
        print(f   Precision: {precision_per_class[i]:.4f} )
        print(f   Recall: {recall_per_class[i]:.4f} )
        print(f   F1-Score: {f1_per_class[i]:.4f} )
        print()
```

Listing 11: Comprehensive model evaluation

3.3 3.3 Feature Selection and Engineering

3.3.1 3.3.1 Feature Selection Methods

```
from sklearn.feature_selection import (SelectKBest, SelectPercentile,
    f_classif, f_regression,
```

```
                                    chi2, mutual_info_classif,
    mutual_info_regression,
                                    RFE, RFECV, SelectFromModel)
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LassoCV
import pandas as pd
import numpy as np

# Univariate Feature Selection
# For classification
selector_f_classif = SelectKBest(score_func=f_classif, k=10)
X_selected_f_classif = selector_f_classif.fit_transform(X, y)

# For regression
selector_f_regression = SelectKBest(score_func=f_regression, k=10)
X_selected_f_regression = selector_f_regression.fit_transform(X, y)

# Using Chi-square test (for non-negative features)
selector_chi2 = SelectKBest(score_func=chi2, k=10)
X_selected_chi2 = selector_chi2.fit_transform(X_non_negative, y)

# Using mutual information
selector_mi_classif = SelectKBest(score_func=mutual_info_classif, k=10)
X_selected_mi_classif = selector_mi_classif.fit_transform(X, y)

# Select Percentile
selector_percentile = SelectPercentile(score_func=f_classif, percentile
    =20)
X_selected_percentile = selector_percentile.fit_transform(X, y)

# Recursive Feature Elimination (RFE)
estimator = RandomForestClassifier(n_estimators=100, random_state=42)
rfe = RFE(estimator=estimator, n_features_to_select=10)
X_selected_rfe = rfe.fit_transform(X, y)

# RFE with Cross-Validation
rfecv = RFECV(estimator=estimator, step=1, cv=5, scoring='accuracy')
X_selected_rfecv = rfecv.fit_transform(X, y)
print(f Optimal number of features: {rfecv.n_features_} )

# Feature selection based on model coefficients
lasso = LassoCV(cv=5, random_state=42)
lasso.fit(X, y)
selector_lasso = SelectFromModel(lasso, prefit=True)
X_selected_lasso = selector_lasso.transform(X)

# Feature selection based on feature importance
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X, y)
selector_rf = SelectFromModel(rf, prefit=True)
X_selected_rf = selector_rf.transform(X)

# Variance Threshold (remove low-variance features)
from sklearn.feature_selection import VarianceThreshold
```

```
selector_variance = VarianceThreshold(threshold=0.01)
X_selected_variance = selector_variance.fit_transform(X)

# Correlation-based feature selection
def correlation_filter(X, threshold=0.95):
    corr_matrix = pd.DataFrame(X).corr().abs()
    upper_triangle = corr_matrix.where(
        np.triu(np.ones(corr_matrix.shape), k=1).astype(bool)
    )
    to_drop = [column for column in upper_triangle.columns
               if any(upper_triangle[column] > threshold)]
    return np.delete(X, to_drop, axis=1)

X_selected_corr = correlation_filter(X, threshold=0.95)

# Wrapper methods
from mlxtend.feature_selection import SequentialFeatureSelector

# Forward selection
sfs_forward = SequentialFeatureSelector(
    RandomForestClassifier(n_estimators=100, random_state=42),
    k_features=10,
    forward=True,
    floating=False,
    scoring='accuracy',
    cv=5
)
sfs_forward.fit(X, y)
X_selected_sfs = sfs_forward.transform(X)

# Backward elimination
sfs_backward = SequentialFeatureSelector(
    RandomForestClassifier(n_estimators=100, random_state=42),
    k_features=10,
    forward=False,
    floating=False,
    scoring='accuracy',
    cv=5
)
sfs_backward.fit(X, y)
X_selected_sfs_backward = sfs_backward.transform(X)
```

Listing 12: Implement various feature selection techniques

3.3.2 3.3.2 Dimensionality Reduction

```
from sklearn.decomposition import PCA, TruncatedSVD, FastICA, NMF
from sklearn.manifold import TSNE, MDS, Isomap, LocallyLinearEmbedding
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
import matplotlib.pyplot as plt
import numpy as np

# Principal Component Analysis (PCA)
```

```
pca = PCA(n_components=0.95)  # Retain 95% of variance
X_pca = pca.fit_transform(X)
print(f Original dimensions: {X.shape} )
print(f PCA dimensions: {X_pca.shape} )
print(f Explained variance ratio: {pca.explained_variance_ratio_} )

# Plot PCA components
plt.figure(figsize=(10, 6))
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('PCA Explained Variance')
plt.show()

# Linear Discriminant Analysis (LDA)
lda = LinearDiscriminantAnalysis(n_components=2)
X_lda = lda.fit_transform(X, y)
print(f LDA dimensions: {X_lda.shape} )

# Truncated SVD (for sparse matrices)
svd = TruncatedSVD(n_components=50)
X_svd = svd.fit_transform(X_sparse)

# Independent Component Analysis (ICA)
ica = FastICA(n_components=10, random_state=42)
X_ica = ica.fit_transform(X)

# Non-negative Matrix Factorization (NMF)
nmf = NMF(n_components=10, random_state=42)
X_nmf = nmf.fit_transform(X_non_negative)

# t-SNE (for visualization)
tsne = TSNE(n_components=2, random_state=42, perplexity=30)
X_tsne = tsne.fit_transform(X_sample)  # Use sample for large datasets

# Plot t-SNE
plt.figure(figsize=(8, 6))
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y_sample, cmap='viridis')
plt.title('t-SNE Visualization')
plt.colorbar()
plt.show()

# Multidimensional Scaling (MDS)
mds = MDS(n_components=2, random_state=42)
X_mds = mds.fit_transform(X_sample)

# Isomap
isomap = Isomap(n_components=2, n_neighbors=5)
X_isomap = isomap.fit_transform(X)

# Locally Linear Embedding (LLE)
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=5)
X_lle = lle.fit_transform(X)

```

```
# Kernel PCA
from sklearn.decomposition import KernelPCA
kpca = KernelPCA(n_components=10, kernel='rbf', gamma=0.1)
X_kpca = kpca.fit_transform(X)

# Autoencoders (using TensorFlow/Keras)
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

def create_autoencoder(input_dim, encoding_dim):
    input_layer = Input(shape=(input_dim,))
    encoded = Dense(encoding_dim, activation='relu')(input_layer)
    decoded = Dense(input_dim, activation='sigmoid')(encoded)

    autoencoder = Model(input_layer, decoded)
    encoder = Model(input_layer, encoded)

    autoencoder.compile(optimizer='adam', loss='mse')
    return autoencoder, encoder

# Use autoencoder for dimensionality reduction
input_dim = X.shape[1]
encoding_dim = 50
autoencoder, encoder = create_autoencoder(input_dim, encoding_dim)
autoencoder.fit(X, X, epochs=100, batch_size=256, shuffle=True, verbose=0)
X_autoencoded = encoder.predict(X)
```

Listing 13: Apply dimensionality reduction techniques

3.4 3.4 Model Selection and Pipeline

3.4.1 3.4.1 Pipeline Construction

```
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder,
    FunctionTransformer
from sklearn.impute import SimpleImputer
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np

# Basic Pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('selector', SelectKBest(f_classif, k=10)),
    ('classifier', RandomForestClassifier(random_state=42))
])

pipeline.fit(X_train, y_train)
predictions = pipeline.predict(X_test)
```

```

# Column Transformer Pipeline (for mixed data types)
numeric_features = ['age', 'income', 'score']
categorical_features = ['gender', 'education', 'city']

preprocessor = ColumnTransformer(
    transformers=[
        ('num', Pipeline([
            ('imputer', SimpleImputer(strategy='median')),
            ('scaler', StandardScaler())
        ]), numeric_features),
        ('cat', Pipeline([
            ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
            ('onehot', OneHotEncoder(drop='first', handle_unknown='ignore'))
        ]), categorical_features)
    ])

# Full pipeline with preprocessing
full_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('feature_selection', SelectKBest(f_classif, k=20)),
    ('classifier', RandomForestClassifier(n_estimators=100, random_state=42))
])

full_pipeline.fit(X_train, y_train)
predictions = full_pipeline.predict(X_test)

# Feature Union Pipeline (combine multiple feature extraction methods)
from sklearn.base import BaseEstimator, TransformerMixin

class CustomFeatureExtractor(BaseEstimator, TransformerMixin):
    def __init__(self, feature_indices):
        self.feature_indices = feature_indices

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        return X[:, self.feature_indices]

# Feature union combining different transformations
feature_union = FeatureUnion([
    ('pca_features', Pipeline([
        ('scaler', StandardScaler()),
        ('pca', PCA(n_components=10))
    ])),
    ('statistical_features', Pipeline([
        ('scaler', StandardScaler()),
        ('selector', SelectKBest(f_classif, k=5))
    ]))
])
```

```

union_pipeline = Pipeline([
    ('features', feature_union),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Custom Pipeline with Custom Transformers
class OutlierRemover(BaseEstimator, TransformerMixin):
    def __init__(self, columns=None, threshold=3):
        self.columns = columns
        self.threshold = threshold

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        if isinstance(X, pd.DataFrame):
            X_copy = X.copy()
            if self.columns is None:
                self.columns = X_copy.select_dtypes(include=[np.number]).columns

            for col in self.columns:
                z_scores = np.abs((X_copy[col] - X_copy[col].mean()) / X_copy[col].std())
                X_copy = X_copy[z_scores < self.threshold]
            return X_copy
        return X

# Pipeline with outlier removal
outlier_pipeline = Pipeline([
    ('outlier_remover', OutlierRemover(threshold=2.5)),
    ('scaler', StandardScaler()),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Pipeline with custom feature engineering
def create_interaction_features(X):
    if isinstance(X, pd.DataFrame):
        X_new = X.copy()
        numeric_cols = X_new.select_dtypes(include=[np.number]).columns
        for i in range(len(numeric_cols)):
            for j in range(i+1, len(numeric_cols)):
                col1, col2 = numeric_cols[i], numeric_cols[j]
                X_new[f'{col1}_{col2}_interaction'] = X_new[col1] * X_new[col2]
        return X_new
    return X

interaction_transformer = FunctionTransformer(create_interaction_features, validate=False)

feature_engineering_pipeline = Pipeline([
    ('interactions', interaction_transformer),
```

```
    ('scaler', StandardScaler()),
    ('selector', SelectKBest(f_classif, k=15)),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Pipeline with parameter tuning
from sklearn.model_selection import GridSearchCV

param_grid = {
    'classifier__n_estimators': [50, 100, 200],
    'classifier__max_depth': [3, 5, 7, None],
    'selector__k': [10, 15, 20]
}

pipeline_grid = GridSearchCV(
    full_pipeline,
    param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

pipeline_grid.fit(X_train, y_train)
print(f Best parameters: {pipeline_grid.best_params_} )
print(f Best score: {pipeline_grid.best_score_:.4f} )

# Save and load pipeline
import joblib
joblib.dump(full_pipeline, 'ml_pipeline.pkl')
loaded_pipeline = joblib.load('ml_pipeline.pkl')
```

Listing 14: Build comprehensive machine learning pipelines

3.4.2 3.4.2 Model Comparison and Selection

```
from sklearn.model_selection import cross_validate
from sklearn.metrics import accuracy_score, make_scorer
import pandas as pd
import numpy as np

# Define models to compare
models = {
    'Logistic Regression': LogisticRegression(random_state=42, max_iter
   =1000),
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state
   =42),
    'SVM': SVC(random_state=42, probability=True),
    'Gradient Boosting': GradientBoostingClassifier(random_state=42),
    'KNN': KNeighborsClassifier(n_neighbors=5),
    'Naive Bayes': GaussianNB(),
    'Decision Tree': DecisionTreeClassifier(random_state=42)
}

```

```python
# Define scoring metrics
scoring = {
    'accuracy': 'accuracy',
    'precision': 'precision_macro',
    'recall': 'recall_macro',
    'f1': 'f1_macro',
    'roc_auc': 'roc_auc_ovr'  # One-vs-Rest for multiclass
}

# Compare models using cross-validation
results = {}
for name, model in models.items():
    print(f Evaluating {name}... )
    cv_results = cross_validate(model, X, y, cv=5, scoring=scoring, n_jobs
   =-1)

    results[name] = {
        'accuracy_mean': cv_results['test_accuracy'].mean(),
        'accuracy_std': cv_results['test_accuracy'].std(),
        'precision_mean': cv_results['test_precision'].mean(),
        'precision_std': cv_results['test_precision'].std(),
        'recall_mean': cv_results['test_recall'].mean(),
        'recall_std': cv_results['test_recall'].std(),
        'f1_mean': cv_results['test_f1'].mean(),
        'f1_std': cv_results['test_f1'].std(),
        'roc_auc_mean': cv_results['test_roc_auc'].mean(),
        'roc_auc_std': cv_results['test_roc_auc'].std()
    }

# Convert results to DataFrame for easy comparison
results_df = pd.DataFrame(results).T
print(results_df.round(4))

# Visualize model comparison
import matplotlib.pyplot as plt

metrics = ['accuracy', 'precision', 'recall', 'f1', 'roc_auc']
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
axes = axes.ravel()

for i, metric in enumerate(metrics):
    means = results_df[f'{metric}_mean']
    stds = results_df[f'{metric}_std']

    axes[i].bar(range(len(means)), means, yerr=stds, capsize=5)
    axes[i].set_xticks(range(len(means)))
    axes[i].set_xticklabels(means.index, rotation=45, ha='right')
    axes[i].set_title(f'{metric.capitalize()} Comparison')
    axes[i].set_ylabel(metric.capitalize())

# Remove empty subplot
axes[5].remove()
plt.tight_layout()
plt.show()
```

```python

# Statistical significance testing
from scipy import stats

def compare_models_statistically(model1_name, model2_name, cv_scores1, cv_scores2):
    """Compare two models using paired t-test"""
    t_stat, p_value = stats.ttest_rel(cv_scores1, cv_scores2)

    print(f"{model1_name} vs {model2_name}:")
    print(f"  t-statistic: {t_stat:.4f}")
    print(f"  p-value: {p_value:.4f}")

    if p_value < 0.05:
        print("  Result: Statistically significant difference")
        if np.mean(cv_scores1) > np.mean(cv_scores2):
            print(f"  {model1_name} is significantly better")
        else:
            print(f"  {model2_name} is significantly better")
    else:
        print("  Result: No statistically significant difference")
    print()

# Compare top two models
top_models = results_df['accuracy_mean'].nlargest(2).index.tolist()
if len(top_models) >= 2:
    model1, model2 = top_models[0], top_models[1]

    # Get CV scores for top models
    cv_scores1 = cross_validate(models[model1], X, y, cv=5, scoring='accuracy')['test_score']
    cv_scores2 = cross_validate(models[model2], X, y, cv=5, scoring='accuracy')['test_score']

    compare_models_statistically(model1, model2, cv_scores1, cv_scores2)

# Ensemble methods for model combination
from sklearn.ensemble import VotingClassifier, StackingClassifier
from sklearn.linear_model import LogisticRegression

# Voting Classifier (Hard Voting)
voting_clf = VotingClassifier(
    estimators=[(name, model) for name, model in models.items()],
    voting='hard'
)

# Voting Classifier (Soft Voting)
voting_clf_soft = VotingClassifier(
    estimators=[(name, model) for name, model in models.items()
                if hasattr(model, 'predict_proba')],
    voting='soft'
)

# Stacking Classifier
```

```
stacking_clf = StackingClassifier(
    estimators=[(name, model) for name, model in models.items()],
    final_estimator=LogisticRegression(random_state=42),
    cv=5
)

# Compare ensemble methods
ensemble_models = {
    'Hard Voting': voting_clf,
    'Soft Voting': voting_clf_soft,
    'Stacking': stacking_clf
}

ensemble_results = {}
for name, model in ensemble_models.items():
    cv_scores = cross_validate(model, X, y, cv=5, scoring='accuracy',
   n_jobs=-1)
    ensemble_results[name] = {
        'mean': cv_scores['test_score'].mean(),
        'std': cv_scores['test_score'].std()
    }

ensemble_df = pd.DataFrame(ensemble_results).T
print( Ensemble Methods Comparison: )
print(ensemble_df.round(4))
```

Listing 15: Compare multiple models systematically

3.5 3.5 Advanced Topics

3.5.1 3.5.1 Imbalanced Dataset Handling

```
from sklearn.utils.class_weight import compute_class_weight
from imblearn.over_sampling import SMOTE, ADASYN, RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler, TomekLinks
from imblearn.combine import SMOTETomek, SMOTEENN
from imblearn.pipeline import Pipeline as ImbPipeline
from collections import Counter
import numpy as np

# Analyze class distribution
class_distribution = Counter(y)
print( Original class distribution: , class_distribution)

# Compute class weights
class_weights = compute_class_weight('balanced', classes=np.unique(y), y=y
   )
class_weight_dict = dict(zip(np.unique(y), class_weights))
print( Class weights: , class_weight_dict)

# Apply to model
weighted_model = RandomForestClassifier(class_weight='balanced',
   random_state=42)

```

```
# Oversampling techniques
# SMOTE (Synthetic Minority Oversampling Technique)
smote = SMOTE(random_state=42)
X_smote, y_smote = smote.fit_resample(X, y)
print( SMOTE class distribution: , Counter(y_smote))

# ADASYN (Adaptive Synthetic Sampling)
adasyn = ADASYN(random_state=42)
X_adasyn, y_adasyn = adasyn.fit_resample(X, y)
print( ADASYN class distribution: , Counter(y_adasyn))

# Random Oversampling
random_oversampler = RandomOverSampler(random_state=42)
X_random_over, y_random_over = random_oversampler.fit_resample(X, y)
print( Random oversampling distribution: , Counter(y_random_over))

# Undersampling techniques
# Random Undersampling
random_undersampler = RandomUnderSampler(random_state=42)
X_random_under, y_random_under = random_undersampler.fit_resample(X, y)
print( Random undersampling distribution: , Counter(y_random_under))

# Tomek Links
tomek = TomekLinks()
X_tomek, y_tomek = tomek.fit_resample(X, y)
print( Tomek Links distribution: , Counter(y_tomek))

# Combined techniques
# SMOTE + Tomek Links
smote_tomek = SMOTETomek(random_state=42)
X_smote_tomek, y_smote_tomek = smote_tomek.fit_resample(X, y)
print( SMOTE + Tomek Links distribution: , Counter(y_smote_tomek))

# SMOTE + Edited Nearest Neighbors
from imblearn.combine import SMOTEENN
smote_enn = SMOTEENN(random_state=42)
X_smote_enn, y_smote_enn = smote_enn.fit_resample(X, y)
print( SMOTE + ENN distribution: , Counter(y_smote_enn))

# Pipeline with resampling
resampling_pipeline = ImbPipeline([
    ('smote', SMOTE(random_state=42)),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Advanced resampling with custom sampling strategy
from imblearn.over_sampling import SMOTE

# Custom sampling strategy
sampling_strategy = {0: 1000, 1: 1000, 2: 1000}  # Equalize all classes to
    1000 samples
custom_smote = SMOTE(sampling_strategy=sampling_strategy, random_state=42)
X_custom, y_custom = custom_smote.fit_resample(X, y)

```

```python
# Evaluate different resampling techniques
from sklearn.model_selection import cross_val_score

techniques = {
    'Original': (X, y),
    'SMOTE': (X_smote, y_smote),
    'ADASYN': (X_adasyn, y_adasyn),
    'Random Oversample': (X_random_over, y_random_over),
    'Random Undersample': (X_random_under, y_random_under)
}

for name, (X_data, y_data) in techniques.items():
    scores = cross_val_score(RandomForestClassifier(random_state=42),
                             X_data, y_data, cv=5, scoring='f1_macro')
    print(f {name} - F1 Score: {scores.mean():.4f} (+/- {scores.std() *
    2:.4f}) )

# Cost-sensitive learning with custom weights
def custom_class_weight(y, multiplier=2):
       Create custom class weights based on class frequency
    unique, counts = np.unique(y, return_counts=True)
    total = len(y)
    weights = {}
    for i, (cls, count) in enumerate(zip(unique, counts)):
        # Higher weight for minority classes
        weight = (total / (len(unique) * count)) * multiplier
        weights[cls] = weight
    return weights

custom_weights = custom_class_weight(y, multiplier=3)
weighted_model_custom = RandomForestClassifier(class_weight=custom_weights
    , random_state=42)

# Threshold tuning for imbalanced datasets
from sklearn.metrics import precision_recall_curve

def find_optimal_threshold(y_true, y_proba):
       Find optimal threshold to maximize F1 score
    precisions, recalls, thresholds = precision_recall_curve(y_true,
    y_proba)
    f1_scores = 2 * (precisions * recalls) / (precisions + recalls)
    optimal_idx = np.argmax(f1_scores)
    optimal_threshold = thresholds[optimal_idx]
    return optimal_threshold

# Example usage
model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)
y_proba = model.predict_proba(X_test)[:, 1]
optimal_threshold = find_optimal_threshold(y_test, y_proba)
print(f Optimal threshold: {optimal_threshold:.4f} )

# Apply optimal threshold
```

```
y_pred_optimal = (y_proba >= optimal_threshold).astype(int)
```

Listing 16: Handle imbalanced datasets effectively

3.5.2 3.5.2 Time Series Analysis

```
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
import pandas as pd
import numpy as np

# Time series cross-validation
def time_series_cv(model, X, y, n_splits=5):
    """Perform time series cross-validation"""
    tscv = TimeSeriesSplit(n_splits=n_splits)
    scores = []

    for train_index, test_index in tscv.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        score = mean_squared_error(y_test, y_pred)
        scores.append(score)

    return np.array(scores)

# Create time series features
def create_time_series_features(df, target_col, lags=[1, 2, 3, 7],
    window_sizes=[3, 7, 14]):
    """Create lag and rolling window features for time series"""
    df_features = df.copy()

    # Lag features
    for lag in lags:
        df_features[f'{target_col}_lag_{lag}'] = df_features[target_col].
    shift(lag)

    # Rolling window features
    for window in window_sizes:
        df_features[f'{target_col}_rolling_mean_{window}'] = df_features[
    target_col].rolling(window=window).mean()
        df_features[f'{target_col}_rolling_std_{window}'] = df_features[
    target_col].rolling(window=window).std()
        df_features[f'{target_col}_rolling_min_{window}'] = df_features[
    target_col].rolling(window=window).min()
        df_features[f'{target_col}_rolling_max_{window}'] = df_features[
    target_col].rolling(window=window).max()

    # Time-based features
    if 'date' in df_features.columns:
        df_features['date'] = pd.to_datetime(df_features['date'])
```

```
        df_features['year'] = df_features['date'].dt.year
        df_features['month'] = df_features['date'].dt.month
        df_features['day'] = df_features['date'].dt.day
        df_features['dayofweek'] = df_features['date'].dt.dayofweek
        df_features['quarter'] = df_features['date'].dt.quarter
        df_features['is_weekend'] = df_features['dayofweek'].isin([5, 6]).
    astype(int)

    return df_features

# Example usage
# Assuming you have a time series dataframe with 'date' and 'value'
    columns
# ts_features = create_time_series_features(time_series_df, 'value')

# Time series forecasting pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor

def time_series_forecast_pipeline(X_train, y_train, X_test, model_type='rf
    '):
        Pipeline for time series forecasting

    # Scale features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Choose model
    if model_type == 'lr':
        model = LinearRegression()
    elif model_type == 'rf':
        model = RandomForestRegressor(n_estimators=100, random_state=42)
    else:
        raise ValueError( model_type must be 'lr' or 'rf' )

    # Train model
    model.fit(X_train_scaled, y_train)

    # Make predictions
    y_pred = model.predict(X_test_scaled)

    return y_pred, model, scaler

# Multi-step forecasting
def multi_step_forecast(model, scaler, X_last, steps=5):
        Multi-step forecasting for time series
    predictions = []
    X_current = X_last.copy()

    for _ in range(steps):
        X_scaled = scaler.transform(X_current.reshape(1, -1))
        pred = model.predict(X_scaled)[0]
```

```
        predictions.append(pred)

        # Update X_current for next prediction (simplified approach)
        # In practice, you'd need to update the lag features properly
        X_current = np.roll(X_current, -1)
        X_current[-1] = pred

    return np.array(predictions)

# Seasonal decomposition features
from statsmodels.tsa.seasonal import seasonal_decompose

def add_seasonal_features(df, target_col, period=12):
    """Add seasonal decomposition features"""
    decomposition = seasonal_decompose(df[target_col], model='additive',
    period=period)

    df['trend'] = decomposition.trend
    df['seasonal'] = decomposition.seasonal
    df['residual'] = decomposition.resid

    return df

# Anomaly detection in time series
from sklearn.ensemble import IsolationForest
from sklearn.svm import OneClassSVM

def detect_time_series_anomalies(X, method='isolation_forest'):
    """Detect anomalies in time series data"""

    if method == 'isolation_forest':
        detector = IsolationForest(contamination=0.1, random_state=42)
    elif method == 'one_class_svm':
        detector = OneClassSVM(nu=0.1)
    else:
        raise ValueError("method must be 'isolation_forest' or '
    one_class_svm'")

    # Fit and predict
    anomalies = detector.fit_predict(X)

    # Convert to boolean (True for anomalies)
    return anomalies == -1

# Time series clustering
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

def cluster_time_series_patterns(X_sequences, n_clusters=3):
    """Cluster time series patterns"""

    # Flatten sequences for clustering
    X_flat = X_sequences.reshape(X_sequences.shape[0], -1)

```

```
    # Apply K-means clustering
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    cluster_labels = kmeans.fit_predict(X_flat)

    # Calculate silhouette score
    sil_score = silhouette_score(X_flat, cluster_labels)

    return cluster_labels, sil_score, kmeans

# Rolling window cross-validation
def rolling_window_cv(model, X, y, window_size=0.8, step_size=0.1):
        Rolling window cross-validation for time series
    n_samples = len(X)
    window_length = int(window_size * n_samples)
    step_length = int(step_size * n_samples)

    scores = []
    start = 0

    while start + window_length < n_samples:
        end = start + window_length

        # Train on window
        X_train, y_train = X[start:end], y[start:end]
        X_test, y_test = X[end:end+step_length], y[end:end+step_length]

        if len(X_test) > 0:
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
            score = mean_squared_error(y_test, y_pred)
            scores.append(score)

        start += step_length

    return np.array(scores)
```

Listing 17: Time series machine learning approaches

4 4. Best Practices

4.1 4.1 Data Preprocessing Best Practices

Best Practice 4.1.1: Always Split Data Before Preprocessing

Why: Prevents data leakage by ensuring test data doesn't influence training preprocessing.
How: Split data first, then apply preprocessing separately to train and test sets.

```
# WRONG WAY - Data leakage risk
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)  # Using entire dataset
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size
    =0.2)
```

```

# CORRECT WAY - No data leakage
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# Fit preprocessing on training data only
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)  # Only transform, don't fit
```

Listing 18: Correct data splitting and preprocessing

Best Practice 4.1.2: Handle Missing Values Strategically

Why: Different missing data patterns require different handling approaches.
How: Analyze missing patterns and choose appropriate imputation strategies.

```
# Analyze missing data patterns
def analyze_missing_data(df):
        Analyze missing data patterns in dataframe
    missing_stats = df.isnull().sum()
    missing_percent = 100 * missing_stats / len(df)

    missing_table = pd.DataFrame({
        'Missing Count': missing_stats,
        'Missing Percentage': missing_percent
    }).sort_values('Missing Percentage', ascending=False)

    return missing_table[missing_table['Missing Count'] > 0]

# Choose imputation strategy based on data characteristics
def smart_impute(df):
        Smart imputation based on data types and missing patterns
    df_imputed = df.copy()

    for column in df.columns:
        missing_percent = df[column].isnull().sum() / len(df) * 100

        if missing_percent > 50:
            print(f Warning: {column} has {missing_percent:.1f}% missing
    values )
            # Consider dropping column
        elif missing_percent > 5:
            print(f High missing rate in {column}: {missing_percent:.1f}%
    )
            # Use advanced imputation
        else:
            # Standard imputation
            if df[column].dtype in ['int64', 'float64']:
                df_imputed[column].fillna(df[column].median(), inplace=
    True)
            else:
                mode_value = df[column].mode()
```

```
                if len(mode_value) > 0:
                    df_imputed[column].fillna(mode_value[0], inplace=True)

    return df_imputed

# Advanced imputation with domain knowledge
def domain_aware_impute(df, domain_rules):
        Impute missing values using domain-specific rules
    df_imputed = df.copy()

    for column, rule in domain_rules.items():
        if column in df.columns:
            if rule['method'] == 'constant':
                df_imputed[column].fillna(rule['value'], inplace=True)
            elif rule['method'] == 'conditional':
                condition_col = rule['condition_column']
                mapping = rule['mapping']
                for cond_value, fill_value in mapping.items():
                    mask = (df_imputed[condition_col] == cond_value) &
   df_imputed[column].isnull()
                    df_imputed.loc[mask, column] = fill_value

    return df_imputed
```

Listing 19: Comprehensive missing data handling

Best Practice 4.1.3: Validate Data Quality Before Modeling

Why: Poor data quality leads to poor model performance and unreliable results.
How: Implement comprehensive data validation checks before training models.

```
def validate_data_quality(df, validation_rules=None):
        Comprehensive data quality validation

    validation_report = {
        'timestamp': pd.Timestamp.now(),
        'total_rows': len(df),
        'total_columns': len(df.columns),
        'issues': []
    }

    # Basic structure validation
    if not isinstance(df, pd.DataFrame):
        validation_report['issues'].append( Data is not a pandas DataFrame
    )
        return validation_report

    # Check for duplicate rows
    duplicates = df.duplicated().sum()
    if duplicates > 0:
        validation_report['issues'].append(f Found {duplicates} duplicate
   rows )

```

```
    # Check for completely empty columns
    empty_cols = df.columns[df.isnull().all()].tolist()
    if empty_cols:
        validation_report['issues'].append(f Empty columns: {empty_cols} )

    # Data type validation
    for column in df.columns:
        # Check for mixed data types
        unique_types = df[column].apply(type).unique()
        if len(unique_types) > 1:
            validation_report['issues'].append(f Mixed types in column {column}: {unique_types} )

        # Check for extreme values in numeric columns
        if df[column].dtype in ['int64', 'float64']:
            Q1 = df[column].quantile(0.25)
            Q3 = df[column].quantile(0.75)
            IQR = Q3 - Q1
            outliers = df[(df[column] < (Q1 - 1.5 * IQR)) | (df[column] > (Q3 + 1.5 * IQR))]
            if len(outliers) > 0:
                outlier_percent = len(outliers) / len(df) * 100
                if outlier_percent > 5:  # More than 5% outliers
                    validation_report['issues'].append(
                        f High outlier rate in {column}: {outlier_percent:.1f}%
                    )

    # Custom validation rules
    if validation_rules:
        for rule in validation_rules:
            try:
                result = df.eval(rule['condition'])
                failed_count = (~result).sum()
                if failed_count > 0:
                    validation_report['issues'].append(
                        f Rule '{rule['name']}' failed for {failed_count} rows
                    )
            except Exception as e:
                validation_report['issues'].append(f Rule '{rule['name']}' error: {str(e)} )

    return validation_report

# Example usage
validation_rules = [
    {'name': 'age_positive', 'condition': 'age > 0'},
    {'name': 'income_non_negative', 'condition': 'income >= 0'},
    {'name': 'valid_category', 'condition':  category.isin(['A', 'B', 'C']) }
]

quality_report = validate_data_quality(data, validation_rules)
```

```
if quality_report['issues']:
    print( Data Quality Issues Found: )
    for issue in quality_report['issues']:
        print(f   - {issue} )
else:
    print( Data quality validation passed! )
```

Listing 20: Data quality validation framework

4.2 4.2 Model Training Best Practices

Best Practice 4.2.1: Use Proper Cross-Validation Strategies

Why: Proper validation prevents overfitting and provides reliable performance estimates.
How: Choose appropriate cross-validation methods based on data characteristics.

```
from sklearn.model_selection import (StratifiedKFold, TimeSeriesSplit,
                                     LeaveOneOut, ShuffleSplit, GroupKFold)

def choose_cv_strategy(data_type, n_samples, groups=None):
       Choose appropriate cross-validation strategy based on data
    characteristics

    if data_type == 'time_series':
        return TimeSeriesSplit(n_splits=5)
    elif data_type == 'grouped':
        if groups is None:
            raise ValueError( Groups required for grouped data )
        return GroupKFold(n_splits=5)
    elif data_type == 'classification':
        if n_samples < 1000:
            return StratifiedKFold(n_splits=5, shuffle=True, random_state
    =42)
        else:
            return StratifiedKFold(n_splits=10, shuffle=True, random_state
    =42)
    elif data_type == 'regression':
        if n_samples < 1000:
            return KFold(n_splits=5, shuffle=True, random_state=42)
        else:
            return KFold(n_splits=10, shuffle=True, random_state=42)
    else:
        return ShuffleSplit(n_splits=5, test_size=0.2, random_state=42)

# Nested cross-validation for unbiased model evaluation
def nested_cross_validation(model, param_grid, X, y, outer_cv=None,
    inner_cv=None):
       Perform nested cross-validation for unbiased model evaluation

    if outer_cv is None:
        outer_cv = KFold(n_splits=5, shuffle=True, random_state=42)
    if inner_cv is None:
        inner_cv = KFold(n_splits=3, shuffle=True, random_state=42)
```

```
    # Outer loop for performance evaluation
    outer_scores = []

    for train_idx, test_idx in outer_cv.split(X):
        X_train_outer, X_test_outer = X[train_idx], X[test_idx]
        y_train_outer, y_test_outer = y[train_idx], y[test_idx]

        # Inner loop for hyperparameter tuning
        grid_search = GridSearchCV(
            model, param_grid, cv=inner_cv, scoring='accuracy', n_jobs=-1
        )
        grid_search.fit(X_train_outer, y_train_outer)

        # Evaluate best model on outer test set
        best_model = grid_search.best_estimator_
        score = best_model.score(X_test_outer, y_test_outer)
        outer_scores.append(score)

    return np.array(outer_scores)

# Example usage
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 7, None]
}

nested_scores = nested_cross_validation(
    RandomForestClassifier(random_state=42),
    param_grid,
    X,
    y
)

print(f Nested CV Score: {nested_scores.mean():.4f} (+/- {nested_scores.
    std() * 2:.4f}) )
```

Listing 21: Advanced cross-validation strategies

Best Practice 4.2.2: Implement Early Stopping and Regularization

Why: Prevents overfitting and reduces training time for iterative algorithms.
How: Use early stopping criteria and regularization parameters appropriately.

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.neural_network import MLPClassifier

# Early stopping for gradient boosting
gb_model = GradientBoostingClassifier(
    n_estimators=1000,
    learning_rate=0.1,
    max_depth=3,
```

```
    random_state=42,
    validation_fraction=0.1,  # Use 10% of training data for validation
    n_iter_no_change=10,      # Stop if no improvement for 10 iterations
    tol=1e-4                  # Tolerance for early stopping
)

# Regularization for linear models
ridge_model = Ridge(alpha=1.0)      # L2 regularization
lasso_model = Lasso(alpha=1.0)      # L1 regularization
elastic_model = ElasticNet(alpha=1.0, l1_ratio=0.5)  # Combined L1/L2

# Early stopping for neural networks
nn_model = MLPClassifier(
    hidden_layer_sizes=(100, 50),
    max_iter=1000,
    early_stopping=True,       # Enable early stopping
    validation_fraction=0.1,   # Validation split
    n_iter_no_change=10,       # Patience
    tol=1e-4,
    random_state=42
)

# Custom early stopping callback
class EarlyStoppingCallback:
    def __init__(self, patience=10, min_delta=1e-4):
        self.patience = patience
        self.min_delta = min_delta
        self.best_score = None
        self.wait = 0

    def __call__(self, current_score):
        if self.best_score is None:
            self.best_score = current_score
        elif current_score - self.best_score < self.min_delta:
            self.wait += 1
            if self.wait >= self.patience:
                return True  # Stop training
        else:
            self.best_score = current_score
            self.wait = 0
        return False  # Continue training

# Adaptive regularization strength selection
def find_optimal_regularization(X, y, model_type='ridge'):
    """Find optimal regularization strength using cross-validation"""

    if model_type == 'ridge':
        alphas = np.logspace(-4, 4, 50)
        scores = []

        for alpha in alphas:
            model = Ridge(alpha=alpha)
            cv_scores = cross_val_score(model, X, y, cv=5, scoring='
   neg_mean_squared_error')
```

```
            scores.append(cv_scores.mean())

        optimal_alpha = alphas[np.argmax(scores)]
        return optimal_alpha

    elif model_type == 'lasso':
        # Use LassoCV for automatic selection
        lasso_cv = LassoCV(cv=5, random_state=42)
        lasso_cv.fit(X, y)
        return lasso_cv.alpha_

# Example usage
optimal_alpha = find_optimal_regularization(X_train, y_train, 'ridge')
print(f Optimal regularization strength: {optimal_alpha:.6f} )
```

Listing 22: Early stopping and regularization implementation

Best Practice 4.2.3: Monitor Training Progress and Diagnose Issues

Why: Helps identify problems early and optimize model performance.
How: Track metrics during training and implement diagnostic tools.

```
import matplotlib.pyplot as plt
from sklearn.model_selection import learning_curve, validation_curve
import time

# Learning curves to diagnose bias/variance
def plot_learning_curve(estimator, X, y, cv=5, train_sizes=np.linspace
    (0.1, 1.0, 10)):
        Plot learning curves to diagnose bias/variance problems

    train_sizes, train_scores, val_scores = learning_curve(
        estimator, X, y, cv=cv, train_sizes=train_sizes, n_jobs=-1
    )

    train_mean = np.mean(train_scores, axis=1)
    train_std = np.std(train_scores, axis=1)
    val_mean = np.mean(val_scores, axis=1)
    val_std = np.std(val_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_mean, 'o-', color='blue', label='Training
    score')
    plt.fill_between(train_sizes, train_mean - train_std, train_mean +
    train_std, alpha=0.1, color='blue')

    plt.plot(train_sizes, val_mean, 'o-', color='red', label='Cross-
    validation score')
    plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std,
    alpha=0.1, color='red')

    plt.xlabel('Training Set Size')
    plt.ylabel('Score')
```

```
    plt.title('Learning Curve')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()

    # Diagnose problems
    final_train_score = train_mean[-1]
    final_val_score = val_mean[-1]

    if final_train_score < 0.8:
        print( High bias (underfitting) - Model is too simple )
    elif final_train_score - final_val_score > 0.1:
        print( High variance (overfitting) - Model is too complex )
    else:
        print( Good fit - Balanced bias and variance )

# Validation curves to optimize hyperparameters
def plot_validation_curve(estimator, X, y, param_name, param_range, cv=5):
       Plot validation curves for hyperparameter tuning

    train_scores, val_scores = validation_curve(
        estimator, X, y, param_name=param_name, param_range=param_range,
    cv=cv, n_jobs=-1
    )

    train_mean = np.mean(train_scores, axis=1)
    train_std = np.std(train_scores, axis=1)
    val_mean = np.mean(val_scores, axis=1)
    val_std = np.std(val_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.semilogx(param_range, train_mean, 'o-', color='blue', label='
    Training score')
    plt.fill_between(param_range, train_mean - train_std, train_mean +
    train_std, alpha=0.1, color='blue')

    plt.semilogx(param_range, val_mean, 'o-', color='red', label='Cross-
    validation score')
    plt.fill_between(param_range, val_mean - val_std, val_mean + val_std,
    alpha=0.1, color='red')

    plt.xlabel(param_name)
    plt.ylabel('Score')
    plt.title(f'Validation Curve for {param_name}')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()

    # Find optimal parameter
    optimal_idx = np.argmax(val_mean)
    optimal_param = param_range[optimal_idx]
    print(f Optimal {param_name}: {optimal_param} )

    return optimal_param
```

```python

# Training time monitoring
def time_model_training(model, X, y, n_repeats=3):
    """Measure model training time"""
    times = []
    for _ in range(n_repeats):
        start_time = time.time()
        model.fit(X, y)
        end_time = time.time()
        times.append(end_time - start_time)

    avg_time = np.mean(times)
    std_time = np.std(times)

    print(f"Average training time: {avg_time:.4f} seconds (+/- {std_time:.4f})")
    return avg_time, std_time

# Model complexity analysis
def analyze_model_complexity(X, y, model_class, param_name, param_values):
    """Analyze how model complexity affects performance"""
    train_scores = []
    val_scores = []
    complexities = []

    for param_value in param_values:
        model = model_class(**{param_name: param_value})

        # Training score
        model.fit(X, y)
        train_score = model.score(X, y)
        train_scores.append(train_score)

        # Validation score
        val_score = cross_val_score(model, X, y, cv=5).mean()
        val_scores.append(val_score)

        # Model complexity (example: number of parameters for tree-based models)
        if hasattr(model, 'n_estimators'):
            complexity = model.n_estimators
        elif hasattr(model, 'max_depth'):
            complexity = model.max_depth if model.max_depth else float('inf')
        else:
            complexity = param_value

        complexities.append(complexity)

    # Plot complexity vs performance
    plt.figure(figsize=(12, 4))

```

```
    plt.subplot(1, 2, 1)
    plt.plot(complexities, train_scores, 'o-', label='Training Score')
    plt.plot(complexities, val_scores, 'o-', label='Validation Score')
    plt.xlabel('Model Complexity')
    plt.ylabel('Score')
    plt.title('Model Complexity vs Performance')
    plt.legend()
    plt.grid(True)

    plt.subplot(1, 2, 2)
    plt.plot(complexities, np.array(train_scores) - np.array(val_scores),
    'o-')
    plt.xlabel('Model Complexity')
    plt.ylabel('Train-Validation Gap')
    plt.title('Overfitting Analysis')
    plt.grid(True)

    plt.tight_layout()
    plt.show()

# Example usage
plot_learning_curve(RandomForestClassifier(random_state=42), X, y)
optimal_n_estimators = plot_validation_curve(
    RandomForestClassifier(random_state=42),
    X, y,
    'n_estimators',
    [10, 50, 100, 200, 500]
)
time_model_training(RandomForestClassifier(n_estimators=100), X, y)
```

Listing 23: Training monitoring and diagnostics

4.3 4.3 Model Evaluation Best Practices

Best Practice 4.3.1: Use Multiple Evaluation Metrics

Why: Single metrics can be misleading; multiple metrics provide comprehensive model assessment.
How: Select appropriate metrics based on problem type and business requirements.

```
from sklearn.metrics import (accuracy_score, precision_score, recall_score
    , f1_score,
                             roc_auc_score, precision_recall_curve,
    roc_curve,
                             mean_squared_error, mean_absolute_error,
    r2_score,
                             confusion_matrix, classification_report)
import matplotlib.pyplot as plt
import seaborn as sns

def comprehensive_classification_evaluation(y_true, y_pred, y_proba=None,
    class_names=None):
        Comprehensive evaluation for classification models
```

```
    results = {}

    # Basic metrics
    results['accuracy'] = accuracy_score(y_true, y_pred)
    results['precision_macro'] = precision_score(y_true, y_pred, average='macro')
    results['recall_macro'] = recall_score(y_true, y_pred, average='macro')
    results['f1_macro'] = f1_score(y_true, y_pred, average='macro')
    results['precision_weighted'] = precision_score(y_true, y_pred, average='weighted')
    results['recall_weighted'] = recall_score(y_true, y_pred, average='weighted')
    results['f1_weighted'] = f1_score(y_true, y_pred, average='weighted')

    # Per-class metrics
    if class_names is None:
        class_names = [f'Class_{i}' for i in range(len(np.unique(y_true)))]

    precision_per_class = precision_score(y_true, y_pred, average=None)
    recall_per_class = recall_score(y_true, y_pred, average=None)
    f1_per_class = f1_score(y_true, y_pred, average=None)

    for i, class_name in enumerate(class_names):
        results[f'precision_{class_name}'] = precision_per_class[i]
        results[f'recall_{class_name}'] = recall_per_class[i]
        results[f'f1_{class_name}'] = f1_per_class[i]

    # Probability-based metrics (if probabilities provided)
    if y_proba is not None:
        if len(np.unique(y_true)) == 2:
            # Binary classification
            results['roc_auc'] = roc_auc_score(y_true, y_proba[:, 1])

            # Find optimal threshold
            precision_vals, recall_vals, thresholds = precision_recall_curve(y_true, y_proba[:, 1])
            f1_scores = 2 * (precision_vals * recall_vals) / (precision_vals + recall_vals)
            optimal_idx = np.argmax(f1_scores)
            results['optimal_threshold'] = thresholds[optimal_idx]
            results['optimal_f1'] = f1_scores[optimal_idx]
        else:
            # Multi-class
            results['roc_auc_ovr'] = roc_auc_score(y_true, y_proba, multi_class='ovr')
            results['roc_auc_ovo'] = roc_auc_score(y_true, y_proba, multi_class='ovo')

    # Print results
    print( === CLASSIFICATION EVALUATION RESULTS === )
    print(f Accuracy: {results['accuracy']:.4f} )
```

```
    print(f Macro Precision: {results['precision_macro']:.4f} )
    print(f Macro Recall: {results['recall_macro']:.4f} )
    print(f Macro F1-Score: {results['f1_macro']:.4f} )
    print(f Weighted Precision: {results['precision_weighted']:.4f} )
    print(f Weighted Recall: {results['recall_weighted']:.4f} )
    print(f Weighted F1-Score: {results['f1_weighted']:.4f} )

    if 'roc_auc' in results:
        print(f AUC-ROC: {results['roc_auc']:.4f} )
        print(f Optimal Threshold: {results['optimal_threshold']:.4f} )
        print(f Optimal F1-Score: {results['optimal_f1']:.4f} )

    # Per-class results
    print( \nPer-Class Results: )
    for class_name in class_names:
        print(f   {class_name}: )
        print(f     Precision: {results[f'precision_{class_name}']:.4f} )
        print(f     Recall: {results[f'recall_{class_name}']:.4f} )
        print(f     F1-Score: {results[f'f1_{class_name}']:.4f} )

    return results

def comprehensive_regression_evaluation(y_true, y_pred):
       Comprehensive evaluation for regression models

    results = {}

    # Error metrics
    results['mse'] = mean_squared_error(y_true, y_pred)
    results['rmse'] = np.sqrt(results['mse'])
    results['mae'] = mean_absolute_error(y_true, y_pred)
    results['r2'] = r2_score(y_true, y_pred)

    # Relative errors
    results['mape'] = np.mean(np.abs((y_true - y_pred) / y_true)) * 100  #
    Mean Absolute Percentage Error

    # Residual analysis
    residuals = y_true - y_pred
    results['residual_std'] = np.std(residuals)
    results['residual_mean'] = np.mean(residuals)

    # Print results
    print( === REGRESSION EVALUATION RESULTS === )
    print(f Mean Squared Error: {results['mse']:.4f} )
    print(f Root Mean Squared Error: {results['rmse']:.4f} )
    print(f Mean Absolute Error: {results['mae']:.4f} )
    print(f  R   Score: {results['r2']:.4f} )
    print(f Mean Absolute Percentage Error: {results['mape']:.2f}% )
    print(f Residual Standard Deviation: {results['residual_std']:.4f} )
    print(f Residual Mean: {results['residual_mean']:.4f} )

    # Visualization
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))
```

```
    # Predicted vs Actual
    axes[0, 0].scatter(y_true, y_pred, alpha=0.6)
    axes[0, 0].plot([y_true.min(), y_true.max()], [y_true.min(), y_true.max()], 'r--', lw=2)
    axes[0, 0].set_xlabel('Actual Values')
    axes[0, 0].set_ylabel('Predicted Values')
    axes[0, 0].set_title('Predicted vs Actual')

    # Residuals vs Predicted
    axes[0, 1].scatter(y_pred, residuals, alpha=0.6)
    axes[0, 1].axhline(y=0, color='r', linestyle='--')
    axes[0, 1].set_xlabel('Predicted Values')
    axes[0, 1].set_ylabel('Residuals')
    axes[0, 1].set_title('Residuals vs Predicted')

    # Residuals histogram
    axes[1, 0].hist(residuals, bins=30, edgecolor='black')
    axes[1, 0].set_xlabel('Residuals')
    axes[1, 0].set_ylabel('Frequency')
    axes[1, 0].set_title('Residuals Distribution')

    # Q-Q plot
    from scipy import stats
    stats.probplot(residuals, dist= norm , plot=axes[1, 1])
    axes[1, 1].set_title('Q-Q Plot of Residuals')

    plt.tight_layout()
    plt.show()

    return results

def plot_confusion_matrix(y_true, y_pred, class_names=None, normalize=False):
        Plot confusion matrix with optional normalization

    cm = confusion_matrix(y_true, y_pred)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        fmt = '.2f'
        title = 'Normalized Confusion Matrix'
    else:
        fmt = 'd'
        title = 'Confusion Matrix'

    if class_names is None:
        class_names = [f'Class_{i}' for i in range(len(cm))]

    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt=fmt, cmap='Blues',
                xticklabels=class_names, yticklabels=class_names)
    plt.title(title)
    plt.ylabel('True Label')
```

```
    plt.xlabel('Predicted Label')
    plt.show()

def plot_roc_curves(y_true, y_proba, class_names=None):
    """Plot ROC curves for multi-class classification"""
    from sklearn.preprocessing import label_binarize
    from itertools import cycle

    n_classes = len(np.unique(y_true))

    if n_classes == 2:
        # Binary classification
        fpr, tpr, _ = roc_curve(y_true, y_proba[:, 1])
        roc_auc = roc_auc_score(y_true, y_proba[:, 1])

        plt.figure(figsize=(8, 6))
        plt.plot(fpr, tpr, color='darkorange', lw=2,
                 label=f'ROC curve (AUC = {roc_auc:.2f})')
        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')
        plt.legend(loc="lower right")
        plt.show()

    else:
        # Multi-class classification
        y_bin = label_binarize(y_true, classes=np.unique(y_true))

        if class_names is None:
            class_names = [f'Class_{i}' for i in range(n_classes)]

        fpr = dict()
        tpr = dict()
        roc_auc = dict()

        for i in range(n_classes):
            fpr[i], tpr[i], _ = roc_curve(y_bin[:, i], y_proba[:, i])
            roc_auc[i] = roc_auc_score(y_bin[:, i], y_proba[:, i])

        # Plot ROC curves
        plt.figure(figsize=(10, 8))
        colors = cycle(['aqua', 'darkorange', 'cornflowerblue', 'red', '
    green'])

        for i, color in zip(range(n_classes), colors):
            plt.plot(fpr[i], tpr[i], color=color, lw=2,
                     label=f'ROC curve of class {class_names[i]} (AUC = {
    roc_auc[i]:.2f})')

        plt.plot([0, 1], [0, 1], 'k--', lw=2)
```

```
        plt.xlim([0.0, 1.0])
        plt.ylim([0.0, 1.05])
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('Multi-class ROC Curves')
        plt.legend(loc= lower right )
        plt.show()

# Example usage
# For classification
# results = comprehensive_classification_evaluation(y_test, y_pred,
   y_proba, class_names=['Class_A', 'Class_B'])

# For regression
# results = comprehensive_regression_evaluation(y_test, y_pred)

# Additional visualizations
# plot_confusion_matrix(y_test, y_pred, class_names=['Class_A', 'Class_B
   '])
# plot_roc_curves(y_test, y_proba, class_names=['Class_A', 'Class_B'])
```

Listing 24: Comprehensive model evaluation with multiple metrics

Best Practice 4.3.2: Validate Model Assumptions and Check Robustness

Why: Ensures model reliability and identifies potential failure modes.
How: Test model assumptions and evaluate performance under various conditions.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from scipy import stats
import matplotlib.pyplot as plt

def validate_linear_regression_assumptions(X, y, model):
       Validate assumptions for linear regression

    # Fit model
    model.fit(X, y)
    y_pred = model.predict(X)
    residuals = y - y_pred

    assumptions_validated = {}

    # 1. Linearity
    # Check if relationship between features and target is linear
    plt.figure(figsize=(15, 5))

    plt.subplot(1, 3, 1)
    plt.scatter(y_pred, y, alpha=0.6)
    plt.plot([y_pred.min(), y_pred.max()], [y_pred.min(), y_pred.max()], '
```

```
r--', lw=2)
    plt.xlabel('Predicted Values')
    plt.ylabel('Actual Values')
    plt.title('Linearity Check')

    # 2. Homoscedasticity (Constant Variance)
    plt.subplot(1, 3, 2)
    plt.scatter(y_pred, residuals, alpha=0.6)
    plt.axhline(y=0, color='r', linestyle='--')
    plt.xlabel('Predicted Values')
    plt.ylabel('Residuals')
    plt.title('Homoscedasticity Check')

    # 3. Normality of Residuals
    plt.subplot(1, 3, 3)
    stats.probplot(residuals, dist= norm , plot=plt)
    plt.title('Normality Check (Q-Q Plot)')

    plt.tight_layout()
    plt.show()

    # Statistical tests
    # Shapiro-Wilk test for normality
    shapiro_stat, shapiro_p = stats.shapiro(residuals)
    assumptions_validated['normality'] = {
        'statistic': shapiro_stat,
        'p_value': shapiro_p,
        'valid': shapiro_p > 0.05  # Fail to reject normality
    }

    # Breusch-Pagan test for homoscedasticity (simplified)
    # Correlation between squared residuals and predicted values
    bp_stat = np.corrcoef(residuals**2, y_pred)[0, 1]
    assumptions_validated['homoscedasticity'] = {
        'correlation': bp_stat,
        'valid': abs(bp_stat) < 0.3  # Low correlation indicates
homoscedasticity
    }

    # Linearity check using correlation
    linearity_corr = np.corrcoef(y_pred, y)[0, 1]
    assumptions_validated['linearity'] = {
        'correlation': linearity_corr,
        'valid': linearity_corr > 0.7  # High correlation indicates good
fit
    }

    # Print results
    print( === LINEAR REGRESSION ASSUMPTIONS VALIDATION === )
    for assumption, result in assumptions_validated.items():
        status =      VALID  if result['valid'] else      INVALID
        print(f {assumption.capitalize()}: {status} )
        for key, value in result.items():
            if key != 'valid':
```

```
                print(f   {key}: {value} )
        print()

    return assumptions_validated

def validate_logistic_regression_assumptions(X, y, model):
       Validate assumptions for logistic regression

    # Fit model
    model.fit(X, y)
    y_pred_proba = model.predict_proba(X)[:, 1]

    assumptions_validated = {}

    # 1. Linearity of logit
    # Check if log-odds are linearly related to predictors
    log_odds = np.log(y_pred_proba / (1 - y_pred_proba + 1e-15))  # Add
   small value to avoid log(0)

    plt.figure(figsize=(12, 4))

    # Plot log-odds vs each feature
    n_features = min(X.shape[1], 6)  # Limit to 6 features for
   visualization
    for i in range(n_features):
        plt.subplot(1, n_features, i + 1)
        plt.scatter(X[:, i], log_odds, alpha=0.6)
        plt.xlabel(f'Feature {i}')
        plt.ylabel('Log-Odds')
        plt.title(f'Log-Odds vs Feature {i}')

    plt.tight_layout()
    plt.show()

    # 2. Absence of multicollinearity
    correlation_matrix = np.corrcoef(X.T)
    max_corr = np.max(np.abs(correlation_matrix - np.eye(
   correlation_matrix.shape[0])))
    assumptions_validated['multicollinearity'] = {
        'max_correlation': max_corr,
        'valid': max_corr < 0.8  # Low multicollinearity
    }

    # 3. Large sample size
    sample_size_ratio = len(y) / X.shape[1]
    assumptions_validated['sample_size'] = {
        'ratio': sample_size_ratio,
        'valid': sample_size_ratio > 10  # At least 10 observations per
   feature
    }

    # Print results
    print( === LOGISTIC REGRESSION ASSUMPTIONS VALIDATION === )
    for assumption, result in assumptions_validated.items():
```

```
        status = "VALID" if result['valid'] else "INVALID"
        print(f"{assumption.replace('_', ' ').title()}: {status}")
        for key, value in result.items():
            if key != 'valid':
                print(f"  {key}: {value}")
        print()

    return assumptions_validated

def test_model_robustness(model, X_train, y_train, X_test, y_test,
    noise_levels=[0.01, 0.05, 0.1]):
    """Test model robustness to input noise"""

    # Fit model on original data
    model.fit(X_train, y_train)
    original_score = model.score(X_test, y_test)

    robustness_results = {
        'original_score': original_score,
        'noise_levels': noise_levels,
        'noisy_scores': [],
        'score_degradation': []
    }

    print("=== MODEL ROBUSTNESS TEST ===")
    print(f"Original performance: {original_score:.4f}")
    print()

    for noise_level in noise_levels:
        # Add noise to test data
        noise = np.random.normal(0, noise_level, X_test.shape)
        X_test_noisy = X_test + noise

        # Evaluate on noisy data
        noisy_score = model.score(X_test_noisy, y_test)
        degradation = (original_score - noisy_score) / original_score *
    100

        robustness_results['noisy_scores'].append(noisy_score)
        robustness_results['score_degradation'].append(degradation)

        print(f"Noise level {noise_level}:")
        print(f"  Performance: {noisy_score:.4f}")
        print(f"  Degradation: {degradation:.2f}%")
        print()

    # Plot results
    plt.figure(figsize=(10, 6))
    plt.plot([0] + noise_levels, [original_score] + robustness_results['
    noisy_scores'], 'o-')
    plt.xlabel('Noise Level')
    plt.ylabel('Model Performance')
    plt.title('Model Robustness to Input Noise')
    plt.grid(True)
```

```
    plt.show()

    # Robustness assessment
    avg_degradation = np.mean(robustness_results['score_degradation'])
    if avg_degradation < 10:
        robustness_level = "High"
    elif avg_degradation < 25:
        robustness_level = "Medium"
    else:
        robustness_level = "Low"

    print(f"Overall Robustness Level: {robustness_level}")
    print(f"Average Performance Degradation: {avg_degradation:.2f}%")

    return robustness_results

def test_model_stability(model, X, y, n_bootstrap=100):
    """Test model stability using bootstrap sampling"""

    original_model = model.__class__(**model.get_params())
    original_model.fit(X, y)
    original_coef = getattr(original_model, 'coef_', None)

    bootstrap_scores = []
    bootstrap_coefs = []

    print("=== MODEL STABILITY TEST (BOOTSTRAP) ===")

    for i in range(n_bootstrap):
        # Bootstrap sample
        indices = np.random.choice(len(X), len(X), replace=True)
        X_boot = X[indices]
        y_boot = y[indices]

        # Fit model on bootstrap sample
        boot_model = model.__class__(**model.get_params())
        boot_model.fit(X_boot, y_boot)
        bootstrap_scores.append(boot_model.score(X, y))

        # Store coefficients if available
        if hasattr(boot_model, 'coef_'):
            bootstrap_coefs.append(boot_model.coef_)

    # Calculate statistics
    mean_score = np.mean(bootstrap_scores)
    std_score = np.std(bootstrap_scores)

    print(f"Original performance: {original_model.score(X, y):.4f}")
    print(f"Bootstrap mean performance: {mean_score:.4f} (+/- {std_score
    :.4f})")
    print(f"Performance stability: {((1 - std_score/mean_score) * 100):.1f
    }%")

    # Coefficient stability (if applicable)
```

```
    if bootstrap_coefs and original_coef is not None:
        coef_matrix = np.array(bootstrap_coefs)
        coef_std = np.std(coef_matrix, axis=0)
        coef_cv = np.mean(coef_std / np.abs(np.mean(coef_matrix, axis=0) + 1e-10))  # Coefficient of variation

        print(f Coefficient stability (CV): {coef_cv:.4f} )
        if coef_cv < 0.1:
            print( Coefficients are stable )
        elif coef_cv < 0.3:
            print( Coefficients are moderately stable )
        else:
            print( Coefficients are unstable )

    # Plot score distribution
    plt.figure(figsize=(10, 6))
    plt.hist(bootstrap_scores, bins=30, alpha=0.7, edgecolor='black')
    plt.axvline(original_model.score(X, y), color='red', linestyle='--',
                label=f'Original Score: {original_model.score(X, y):.4f}')
    plt.axvline(mean_score, color='blue', linestyle='--',
                label=f'Mean Bootstrap Score: {mean_score:.4f}')
    plt.xlabel('Model Performance')
    plt.ylabel('Frequency')
    plt.title('Bootstrap Performance Distribution')
    plt.legend()
    plt.grid(True)
    plt.show()

    return {
        'bootstrap_scores': bootstrap_scores,
        'bootstrap_coefs': bootstrap_coefs,
        'mean_score': mean_score,
        'std_score': std_score
    }

# Example usage
# For linear regression
# lr_model = LinearRegression()
# assumptions = validate_linear_regression_assumptions(X, y, lr_model)

# For logistic regression
# log_model = LogisticRegression()
# assumptions = validate_logistic_regression_assumptions(X, y, log_model)

# Robustness testing
# rf_model = RandomForestClassifier(random_state=42)
# robustness_results = test_model_robustness(rf_model, X_train, y_train, X_test, y_test)

# Stability testing
# stability_results = test_model_stability(rf_model, X, y)
```

Listing 25: Model assumption validation and robustness testing

4.4 4.4 Deployment and Production Best Practices

Best Practice 4.4.1: Implement Proper Model Versioning and Tracking

Why: Enables reproducibility, rollback capabilities, and experiment tracking.
How: Use version control for code and models, and implement experiment tracking.

```
import joblib
import json
import os
from datetime import datetime
import git
import hashlib

class ModelVersionTracker:
       Track model versions, parameters, and performance metrics

    def __init__(self, tracking_dir='model_tracking'):
        self.tracking_dir = tracking_dir
        os.makedirs(tracking_dir, exist_ok=True)
        self.experiments_file = os.path.join(tracking_dir, 'experiments.
   json')

        # Initialize experiments file if it doesn't exist
        if not os.path.exists(self.experiments_file):
            with open(self.experiments_file, 'w') as f:
                json.dump([], f)

    def get_git_info(self):
           Get current git commit information
        try:
            repo = git.Repo(search_parent_directories=True)
            commit_hash = repo.head.object.hexsha
            commit_message = repo.head.object.message.strip()
            branch = repo.active_branch.name
            return {
                'commit_hash': commit_hash,
                'commit_message': commit_message,
                'branch': branch
            }
        except:
            return {
                'commit_hash': 'unknown',
                'commit_message': 'unknown',
                'branch': 'unknown'
            }

    def calculate_data_hash(self, X, y):
           Calculate hash of training data for version tracking
        # Convert to numpy arrays and flatten
        X_flat = X.flatten() if hasattr(X, 'flatten') else np.array(X).
   flatten()
        y_flat = y.flatten() if hasattr(y, 'flatten') else np.array(y).
```

```
    flatten()

        # Combine and hash
        combined = np.concatenate([X_flat, y_flat])
        return hashlib.md5(combined.tobytes()).hexdigest()

    def log_experiment(self, model, X, y, params, metrics, model_name,
    notes=  ):
            Log experiment details

        experiment = {
            'experiment_id': datetime.now().strftime('%Y%m%d_%H%M%S'),
            'timestamp': datetime.now().isoformat(),
            'model_name': model_name,
            'parameters': params,
            'metrics': metrics,
            'data_hash': self.calculate_data_hash(X, y),
            'git_info': self.get_git_info(),
            'notes': notes
        }

        # Load existing experiments
        with open(self.experiments_file, 'r') as f:
            experiments = json.load(f)

        # Add new experiment
        experiments.append(experiment)

        # Save updated experiments
        with open(self.experiments_file, 'w') as f:
            json.dump(experiments, f, indent=2)

        print(f Experiment logged: {experiment['experiment_id']} )
        return experiment['experiment_id']

    def save_model(self, model, experiment_id, model_name):
            Save model with version tracking
        model_dir = os.path.join(self.tracking_dir, 'models')
        os.makedirs(model_dir, exist_ok=True)

        model_filename = f {model_name}_{experiment_id}.pkl
        model_path = os.path.join(model_dir, model_filename)

        joblib.dump(model, model_path)
        print(f Model saved: {model_path} )
        return model_path

    def load_model(self, model_name, experiment_id=None):
            Load specific model version
        model_dir = os.path.join(self.tracking_dir, 'models')

        if experiment_id:
            model_filename = f {model_name}_{experiment_id}.pkl
            model_path = os.path.join(model_dir, model_filename)
```

```
            if os.path.exists(model_path):
                return joblib.load(model_path)
            else:
                raise FileNotFoundError(f Model {model_filename} not found
    )
        else:
            # Load latest model
            model_files = [f for f in os.listdir(model_dir) if f.
   startswith(model_name)]
            if not model_files:
                raise FileNotFoundError(f No models found for {model_name}
    )

            # Sort by timestamp and load latest
            model_files.sort(reverse=True)
            latest_model = model_files[0]
            model_path = os.path.join(model_dir, latest_model)
            return joblib.load(model_path)

    def get_experiment_history(self, model_name=None):
           Get experiment history
        with open(self.experiments_file, 'r') as f:
            experiments = json.load(f)

        if model_name:
            experiments = [exp for exp in experiments if exp['model_name']
    == model_name]

        return experiments

    def compare_experiments(self, experiment_ids):
           Compare multiple experiments
        with open(self.experiments_file, 'r') as f:
            all_experiments = json.load(f)

        experiments = [exp for exp in all_experiments if exp['
   experiment_id'] in experiment_ids]

        comparison = {}
        for exp in experiments:
            exp_id = exp['experiment_id']
            comparison[exp_id] = {
                'timestamp': exp['timestamp'],
                'model_name': exp['model_name'],
                'metrics': exp['metrics'],
                'parameters': exp['parameters']
            }

        return comparison

# Advanced model versioning with MLflow-like functionality
class SimpleMLFlow:
       Simple experiment tracking inspired by MLflow

```

```
    def __init__(self, experiment_name= default ):
        self.experiment_name = experiment_name
        self.runs_dir = f mlruns/{experiment_name}
        os.makedirs(self.runs_dir, exist_ok=True)
        self.current_run = None

    def start_run(self, run_name=None):
            Start a new experiment run
        run_id = datetime.now().strftime('%Y%m%d_%H%M%S_%f')
        run_dir = os.path.join(self.runs_dir, run_id)
        os.makedirs(run_dir, exist_ok=True)

        self.current_run = {
            'run_id': run_id,
            'run_name': run_name or f run_{run_id} ,
            'start_time': datetime.now().isoformat(),
            'params': {},
            'metrics': {},
            'artifacts': {}
        }

        print(f Started run: {run_name or run_id} )
        return self.current_run

    def log_param(self, key, value):
            Log a parameter
        if self.current_run:
            self.current_run['params'][key] = value
            print(f Logged parameter: {key} = {value} )

    def log_metric(self, key, value):
            Log a metric
        if self.current_run:
            self.current_run['metrics'][key] = value
            print(f Logged metric: {key} = {value} )

    def log_artifact(self, local_path, artifact_path=None):
            Log an artifact (file)
        if self.current_run:
            import shutil
            artifact_name = artifact_path or os.path.basename(local_path)
            run_artifacts_dir = os.path.join(self.runs_dir, self.
    current_run['run_id'], 'artifacts')
            os.makedirs(run_artifacts_dir, exist_ok=True)

            artifact_dest = os.path.join(run_artifacts_dir, artifact_name)
            shutil.copy2(local_path, artifact_dest)

            self.current_run['artifacts'][artifact_name] = artifact_dest
            print(f Logged artifact: {artifact_name} )

    def end_run(self):
            End current run and save metadata
        if self.current_run:
```

```
            # Save run metadata
            run_metadata_path = os.path.join(self.runs_dir, self.
    current_run['run_id'], 'meta.json')
            self.current_run['end_time'] = datetime.now().isoformat()

            with open(run_metadata_path, 'w') as f:
                json.dump(self.current_run, f, indent=2)

            print(f Ended run: {self.current_run['run_name']} )
            self.current_run = None

    def list_runs(self):
           List all runs in experiment
        runs = []
        for run_id in os.listdir(self.runs_dir):
            run_dir = os.path.join(self.runs_dir, run_id)
            if os.path.isdir(run_dir):
                meta_file = os.path.join(run_dir, 'meta.json')
                if os.path.exists(meta_file):
                    with open(meta_file, 'r') as f:
                        meta = json.load(f)
                    runs.append(meta)
        return sorted(runs, key=lambda x: x['start_time'], reverse=True)

# Production model deployment utilities
class ModelDeployer:
       Utilities for model deployment

    def __init__(self, model_path, version_tracker=None):
        self.model = joblib.load(model_path)
        self.model_path = model_path
        self.version_tracker = version_tracker
        self.load_time = datetime.now()

    def predict(self, X):
           Make predictions with error handling
        try:
            predictions = self.model.predict(X)
            return predictions
        except Exception as e:
            print(f Prediction error: {str(e)} )
            raise

    def predict_proba(self, X):
           Make probability predictions if supported
        if hasattr(self.model, 'predict_proba'):
            try:
                probabilities = self.model.predict_proba(X)
                return probabilities
            except Exception as e:
                print(f Probability prediction error: {str(e)} )
                raise
        else:
            raise AttributeError( Model does not support predict_proba )
```

```
252
253     def get_model_info(self):
254         """Get model information"""
255         info = {
256             'model_type': type(self.model).__name__,
257             'load_time': self.load_time.isoformat(),
258             'model_path': self.model_path
259         }
260
261         # Add model-specific information
262         if hasattr(self.model, 'get_params'):
263             info['parameters'] = self.model.get_params()
264
265         if hasattr(self.model, 'feature_importances_'):
266             info['feature_importance_available'] = True
267
268         return info
269
270     def health_check(self, X_sample, y_sample=None):
271         """Perform health check on loaded model"""
272         try:
273             # Basic prediction test
274             predictions = self.predict(X_sample[:5])  # Test with first 5
    samples
275
276             health_status = {
277                 'status': 'healthy',
278                 'timestamp': datetime.now().isoformat(),
279                 'prediction_shape': predictions.shape,
280                 'prediction_types': [type(pred).__name__ for pred in
    predictions[:3]]
281             }
282
283             # If sample labels provided, check accuracy
284             if y_sample is not None:
285                 sample_accuracy = np.mean(predictions[:len(y_sample[:5])]
    == y_sample[:5])
286                 health_status['sample_accuracy'] = sample_accuracy
287
288             print("✓ Health check passed")
289             return health_status
290
291         except Exception as e:
292             health_status = {
293                 'status': 'unhealthy',
294                 'timestamp': datetime.now().isoformat(),
295                 'error': str(e)
296             }
297             print(f"✗ Health check failed: {str(e)}")
298             return health_status
299
300 # Example usage
301 def example_model_tracking():
302     """Example of comprehensive model tracking
```

```
    # Initialize trackers
    version_tracker = ModelVersionTracker()
    mlflow = SimpleMLFlow( customer_churn_prediction )

    # Start experiment
    mlflow.start_run( random_forest_baseline )

    # Log parameters
    params = {
        'n_estimators': 100,
        'max_depth': 10,
        'random_state': 42
    }

    for key, value in params.items():
        mlflow.log_param(key, value)

    # Train model
    model = RandomForestClassifier(**params)
    model.fit(X_train, y_train)

    # Evaluate model
    y_pred = model.predict(X_test)
    metrics = {
        '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)
    }

    for key, value in metrics.items():
        mlflow.log_metric(key, value)

    # Save model
    model_path = version_tracker.save_model(model, mlflow.current_run['
run_id'], 'churn_model')
    mlflow.log_artifact(model_path)

    # Log experiment
    experiment_id = version_tracker.log_experiment(
        model, X_train, y_train, params, metrics,
        'churn_model', 'Baseline Random Forest model'
    )

    # End run
    mlflow.end_run()

    # Compare experiments
    recent_experiments = version_tracker.get_experiment_history('
churn_model')[:3]
    experiment_ids = [exp['experiment_id'] for exp in recent_experiments]
    comparison = version_tracker.compare_experiments(experiment_ids)
```

```
    print( Recent experiments comparison: )
    for exp_id, exp_data in comparison.items():
        print(f   {exp_id}: Accuracy = {exp_data['metrics']['accuracy']:.4
   f} )

# Model monitoring in production
class ModelMonitor:
       Monitor model performance in production

    def __init__(self, model_deployer, alert_threshold=0.1):
        self.model_deployer = model_deployer
        self.alert_threshold = alert_threshold
        self.performance_history = []
        self.prediction_history = []

    def log_prediction(self, X, y_true=None, prediction_time=None):
           Log prediction for monitoring
        prediction_record = {
            'timestamp': prediction_time or datetime.now().isoformat(),
            'n_samples': len(X),
            'prediction_time': datetime.now().isoformat()
        }

        # Store prediction data (be careful with data privacy)
        self.prediction_history.append(prediction_record)

        # If true labels provided, evaluate performance
        if y_true is not None:
            try:
                y_pred = self.model_deployer.predict(X)
                accuracy = accuracy_score(y_true, y_pred)

                performance_record = {
                    'timestamp': prediction_time or datetime.now().
   isoformat(),
                    'accuracy': accuracy,
                    'n_samples': len(y_true)
                }

                self.performance_history.append(performance_record)

                # Check for performance degradation
                if len(self.performance_history) > 5:
                    recent_accuracy = np.mean([r['accuracy'] for r in self
   .performance_history[-5:]])
                    historical_accuracy = np.mean([r['accuracy'] for r in
   self.performance_history[:-5]])

                    if historical_accuracy - recent_accuracy > self.
   alert_threshold:
                        self.send_alert(
                            f Performance degradation detected!
                            f Accuracy dropped from {historical_accuracy
   :.4f} to {recent_accuracy:.4f}
```

```
                        )

                return accuracy
            except Exception as e:
                print(f Performance monitoring error: {str(e)} )

    def send_alert(self, message):
        Send alert (simplified - in practice, integrate with alerting systems)
        print(f ALERT: {message} )
        # In practice: send email, Slack message, PagerDuty alert, etc.

    def get_performance_report(self, window_hours=24):
        Get performance report for specified time window
        cutoff_time = datetime.now() - pd.Timedelta(hours=window_hours)
        cutoff_str = cutoff_time.isoformat()

        recent_performance = [
            record for record in self.performance_history
            if record['timestamp'] >= cutoff_str
        ]

        if recent_performance:
            avg_accuracy = np.mean([r['accuracy'] for r in recent_performance])
            total_predictions = sum([r['n_samples'] for r in recent_performance])

            report = {
                'period': f Last {window_hours} hours ,
                'average_accuracy': avg_accuracy,
                'total_predictions': total_predictions,
                'n_evaluation_batches': len(recent_performance)
            }

            return report
        else:
            return {
                'period': f Last {window_hours} hours ,
                'message': 'No performance data available'
            }

# Example usage of production deployment
def production_deployment_example():
    Example of production model deployment

    # Load model
    deployer = ModelDeployer('models/churn_model_latest.pkl')

    # Health check
    health_status = deployer.health_check(X_test[:5], y_test[:5])
    print( Model health: , health_status)

    # Initialize monitoring
```

```
    monitor = ModelMonitor(deployer, alert_threshold=0.05)

    # Simulate production predictions
    for i in range(10):
        # Get batch of data
        batch_size = np.random.randint(10, 100)
        X_batch = X_test[i*batch_size:(i+1)*batch_size]

        # Make predictions
        predictions = deployer.predict(X_batch)

        # Log predictions for monitoring
        monitor.log_prediction(X_batch)

        print(f Batch {i+1}: Processed {len(predictions)} predictions )

    # Get performance report
    report = monitor.get_performance_report(window_hours=1)
    print( Performance report: , report)

# Run examples
# example_model_tracking()
# production_deployment_example()
```

Listing 26: Model versioning and experiment tracking

Best Practice 4.4.2: Implement Proper Error Handling and Logging

Why: Ensures system reliability and enables debugging in production environments.
How: Use comprehensive logging, error handling, and monitoring throughout the ML pipeline.

```
import logging
import traceback
from functools import wraps
import time
from datetime import datetime
import json
import os

# Configure logging
def setup_logging(log_level=logging.INFO, log_file='ml_pipeline.log'):
        Setup comprehensive logging for ML pipeline

    # Create logger
    logger = logging.getLogger('ml_pipeline')
    logger.setLevel(log_level)

    # Create formatters
    detailed_formatter = logging.Formatter(
        '%(asctime)s - %(name)s - %(levelname)s - %(funcName)s:%(lineno)d
   - %(message)s'
    )
```

```
    simple_formatter = logging.Formatter(
        '%(asctime)s - %(levelname)s - %(message)s'
    )

    # Console handler
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.INFO)
    console_handler.setFormatter(simple_formatter)

    # File handler
    file_handler = logging.FileHandler(log_file)
    file_handler.setLevel(log_level)
    file_handler.setFormatter(detailed_formatter)

    # Add handlers to logger
    logger.addHandler(console_handler)
    logger.addHandler(file_handler)

    return logger

# Global logger
logger = setup_logging()

# Decorator for function timing and error handling
def monitor_function(func):
       Decorator to monitor function execution time and handle errors

    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        function_name = func.__name__

        logger.info(f Starting execution of {function_name} )

        try:
            result = func(*args, **kwargs)
            execution_time = time.time() - start_time

            logger.info(f Successfully completed {function_name} in {
   execution_time:.2f} seconds )
            return result

        except Exception as e:
            execution_time = time.time() - start_time
            error_msg = f Error in {function_name} after {execution_time
   :.2f} seconds: {str(e)}
            logger.error(error_msg)
            logger.error(f Traceback: {traceback.format_exc()} )

            # Re-raise the exception
            raise

    return wrapper

```

```
# Custom exception classes
class DataValidationError(Exception):
        Exception raised for data validation errors
    def __init__(self, message, details=None):
        self.message = message
        self.details = details
        super().__init__(self.message)

class ModelTrainingError(Exception):
        Exception raised for model training errors
    def __init__(self, message, model_params=None):
        self.message = message
        self.model_params = model_params
        super().__init__(self.message)

class PredictionError(Exception):
        Exception raised for prediction errors
    def __init__(self, message, input_shape=None):
        self.message = message
        self.input_shape = input_shape
        super().__init__(self.message)

# Data validation with comprehensive error handling
@monitor_function
def validate_input_data(X, y=None, expected_features=None):
        Validate input data with comprehensive error checking

    errors = []

    # Check if X is None or empty
    if X is None:
        errors.append( Input data X is None )
    elif len(X) == 0:
        errors.append( Input data X is empty )

    # Check data types
    if not isinstance(X, (list, np.ndarray, pd.DataFrame)):
        errors.append(f X must be array-like, got {type(X)} )

    # Convert to numpy array for consistency
    try:
        X_array = np.array(X)
    except Exception as e:
        errors.append(f Cannot convert X to numpy array: {str(e)} )
        X_array = None

    # Check dimensions
    if X_array is not None:
        if X_array.ndim != 2:
            errors.append(f X must be 2-dimensional, got {X_array.ndim}
   dimensions )

        # Check for expected number of features
        if expected_features and X_array.shape[1] != expected_features:
```

```
            errors.append(f Expected {expected_features} features, got {X_array.shape[1]} )

        # Check for infinite values
        if np.any(np.isinf(X_array)):
            inf_count = np.sum(np.isinf(X_array))
            errors.append(f Found {inf_count} infinite values in X )

        # Check for NaN values
        if np.any(np.isnan(X_array)):
            nan_count = np.sum(np.isnan(X_array))
            errors.append(f Found {nan_count} NaN values in X )

    # Validate target variable if provided
    if y is not None:
        if len(y) != len(X):
            errors.append(f Length mismatch: X has {len(X)} samples, y has {len(y)} samples )

        # Check for NaN in target
        y_array = np.array(y)
        if np.any(np.isnan(y_array)):
            nan_count = np.sum(np.isnan(y_array))
            errors.append(f Found {nan_count} NaN values in y )

    # Raise exception if errors found
    if errors:
        error_details = {
            'timestamp': datetime.now().isoformat(),
            'input_shape': X_array.shape if X_array is not None else None,
            'error_count': len(errors)
        }
        raise DataValidationError(
             Data validation failed:   +  ;  .join(errors),
            details=error_details
        )

    logger.info( Data validation passed )
    return X_array, y_array if y is not None else None

# Robust model training with error handling
@monitor_function
def robust_model_training(X_train, y_train, model_class, model_params=None):
        Train model with comprehensive error handling

    if model_params is None:
        model_params = {}

    try:
        # Validate input data
        X_train_valid, y_train_valid = validate_input_data(X_train, y_train)

```

```
        # Create model instance
        logger.info(f Creating model: {model_class.__name__} )
        model = model_class(**model_params)

        # Train model
        logger.info( Starting model training )
        model.fit(X_train_valid, y_train_valid)

        # Validate trained model
        if not hasattr(model, 'predict'):
            raise ModelTrainingError(
                 Trained model does not have predict method ,
                model_params=model_params
            )

        logger.info( Model training completed successfully )
        return model

    except DataValidationError as e:
        logger.error(f Data validation error during training: {e.message}
    )
        raise
    except Exception as e:
        error_details = {
            'timestamp': datetime.now().isoformat(),
            'model_class': model_class.__name__,
            'model_params': model_params,
            'error_type': type(e).__name__
        }
        raise ModelTrainingError(
            f Model training failed: {str(e)} ,
            model_params=model_params
        ) from e

# Safe prediction with error handling
@monitor_function
def safe_predict(model, X, return_proba=False):
       Make predictions with comprehensive error handling

    try:
        # Validate input data
        X_valid, _ = validate_input_data(X)

        # Check if model has required methods
        if not hasattr(model, 'predict'):
            raise PredictionError(
                 Model does not have predict method ,
                input_shape=X_valid.shape
            )

        # Make predictions
        logger.info(f Making predictions for {X_valid.shape[0]} samples )
        predictions = model.predict(X_valid)

```

```
        # Return probabilities if requested
        if return_proba:
            if not hasattr(model, 'predict_proba'):
                logger.warning( Model does not support predict_proba, returning only predictions )
                return predictions, None
            else:
                probabilities = model.predict_proba(X_valid)
                return predictions, probabilities

        return predictions

    except DataValidationError as e:
        logger.error(f Data validation error during prediction: {e.message} )
        raise
    except Exception as e:
        error_details = {
            'timestamp': datetime.now().isoformat(),
            'input_shape': X.shape if hasattr(X, 'shape') else None,
            'model_type': type(model).__name__,
            'error_type': type(e).__name__
        }
        raise PredictionError(
            f Prediction failed: {str(e)} ,
            input_shape=X.shape if hasattr(X, 'shape') else None
        ) from e

# Model performance monitoring with error handling
@monitor_function
def monitor_model_performance(y_true, y_pred, metrics=None):
       Monitor model performance with error handling

    if metrics is None:
        metrics = ['accuracy', 'precision', 'recall', 'f1']

    results = {}
    errors = []

    try:
        # Validate inputs
        if len(y_true) != len(y_pred):
            raise ValueError(f Length mismatch: y_true={len(y_true)}, y_pred={len(y_pred)} )

        # Calculate metrics
        for metric in metrics:
            try:
                if metric == 'accuracy':
                    results[metric] = accuracy_score(y_true, y_pred)
                elif metric == 'precision':
                    results[metric] = precision_score(y_true, y_pred, average='weighted')
                elif metric == 'recall':
```

```
                    results[metric] = recall_score(y_true, y_pred, average='weighted')
                elif metric == 'f1':
                    results[metric] = f1_score(y_true, y_pred, average='weighted')
                elif metric == 'roc_auc':
                    if len(np.unique(y_true)) == 2:
                        results[metric] = roc_auc_score(y_true, y_pred)
                    else:
                        results[metric] = 'Not applicable for multiclass'
                else:
                    logger.warning(f Unknown metric: {metric} )
            except Exception as e:
                errors.append(f Error calculating {metric}: {str(e)} )
                results[metric] = None

        # Log results
        logger.info( Model performance metrics: )
        for metric, value in results.items():
            if value is not None:
                logger.info(f   {metric}: {value:.4f} )

        if errors:
            logger.warning(f Errors in metric calculation: {errors} )

        return results

    except Exception as e:
        logger.error(f Error in performance monitoring: {str(e)} )
        raise

# Context manager for safe model operations
class SafeModelOperation:
       Context manager for safe model operations with automatic cleanup

    def __init__(self, operation_name, timeout_seconds=300):
        self.operation_name = operation_name
        self.timeout_seconds = timeout_seconds
        self.start_time = None

    def __enter__(self):
        self.start_time = time.time()
        logger.info(f Starting {self.operation_name} )
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        elapsed_time = time.time() - self.start_time

        if exc_type is not None:
            # An exception occurred
            logger.error(f Error in {self.operation_name} after {elapsed_time:.2f} seconds )
            logger.error(f Exception type: {exc_type.__name__} )
            logger.error(f Exception message: {str(exc_value)} )
```

```
            if traceback:
                logger.error(f"Traceback: {traceback.format_exc()}")
            return False  # Don't suppress the exception
        else:
            # Operation completed successfully
            logger.info(f"Completed {self.operation_name} in {elapsed_time:.2f} seconds")
            return True

# Retry mechanism for unreliable operations
def retry_on_failure(max_retries=3, delay_seconds=1, backoff_factor=2):
    """Decorator to retry function execution on failure"""

    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            retries = 0
            current_delay = delay_seconds

            while retries < max_retries:
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    retries += 1
                    if retries >= max_retries:
                        logger.error(f"Function {func.__name__} failed after {max_retries} retries")
                        raise

                    logger.warning(
                        f"Function {func.__name__} failed (attempt {retries}/{max_retries}): {str(e)}"
                    )
                    logger.info(f"Retrying in {current_delay} seconds...")
                    time.sleep(current_delay)
                    current_delay *= backoff_factor

        return wrapper
    return decorator

# Example usage of error handling and logging
@retry_on_failure(max_retries=3, delay_seconds=2)
@monitor_function
def example_ml_pipeline(X_train, y_train, X_test, y_test):
    """Example ML pipeline with comprehensive error handling"""

    with SafeModelOperation("Data Validation") as op:
        # Validate training data
        X_train_valid, y_train_valid = validate_input_data(X_train, y_train)
        X_test_valid, y_test_valid = validate_input_data(X_test, y_test)

    with SafeModelOperation("Model Training") as op:
        # Train model
```

```
        model = robust_model_training(
            X_train_valid, y_train_valid,
            RandomForestClassifier,
            {'n_estimators': 100, 'random_state': 42}
        )

    with SafeModelOperation( Model Evaluation ) as op:
        # Make predictions
        y_pred = safe_predict(model, X_test_valid)

        # Evaluate performance
        metrics = monitor_model_performance(y_test_valid, y_pred)

        return model, metrics

# Error handling for data loading
@monitor_function
def safe_data_loading(file_path, file_type='csv'):
       Safely load data with comprehensive error handling

    try:
        if not os.path.exists(file_path):
            raise FileNotFoundError(f File not found: {file_path} )

        if file_type == 'csv':
            data = pd.read_csv(file_path)
        elif file_type == 'json':
            data = pd.read_json(file_path)
        elif file_type == 'excel':
            data = pd.read_excel(file_path)
        else:
            raise ValueError(f Unsupported file type: {file_type} )

        logger.info(f Successfully loaded data from {file_path} )
        logger.info(f Data shape: {data.shape} )

        return data

    except FileNotFoundError as e:
        logger.error(f File not found error: {str(e)} )
        raise
    except pd.errors.EmptyDataError:
        logger.error( File is empty )
        raise DataValidationError( Data file is empty )
    except pd.errors.ParserError as e:
        logger.error(f Data parsing error: {str(e)} )
        raise DataValidationError(f Data parsing failed: {str(e)} )
    except Exception as e:
        logger.error(f Unexpected error loading data: {str(e)} )
        raise DataValidationError(f Failed to load data: {str(e)} )

# Graceful degradation for model services
class GracefulModelService:
       Model service with graceful degradation
```

```
    def __init__(self, primary_model, fallback_model=None):
        self.primary_model = primary_model
        self.fallback_model = fallback_model
        self.primary_model_failed = False

    def predict(self, X, use_fallback_if_needed=True):
        """Make predictions with fallback capability"""

        # Try primary model first
        if not self.primary_model_failed:
            try:
                logger.info("Using primary model for prediction")
                return safe_predict(self.primary_model, X)
            except Exception as e:
                logger.warning(f"Primary model failed: {str(e)}")
                self.primary_model_failed = True

                # If fallback available and allowed, use it
                if self.fallback_model and use_fallback_if_needed:
                    logger.info("Switching to fallback model")
                    return safe_predict(self.fallback_model, X)
                else:
                    raise
        else:
            # Primary model already failed, use fallback if available
            if self.fallback_model and use_fallback_if_needed:
                logger.info("Using fallback model for prediction")
                return safe_predict(self.fallback_model, X)
            else:
                raise PredictionError("Primary model failed and no fallback available")

# Example usage
def example_error_handling():
    """Example demonstrating comprehensive error handling"""

    try:
        # Load data safely
        data = safe_data_loading('data.csv')

        # Prepare data
        X = data.drop('target', axis=1)
        y = data['target']

        # Split data
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

        # Run pipeline with error handling
        model, metrics = example_ml_pipeline(X_train, y_train, X_test, y_test)

        logger.info("ML pipeline completed successfully")
```

```
        return model, metrics

    except DataValidationError as e:
        logger.error(f Data validation error: {e.message} )
        if e.details:
            logger.error(f Details: {e.details} )
    except ModelTrainingError as e:
        logger.error(f Model training error: {e.message} )
        if e.model_params:
            logger.error(f Model parameters: {e.model_params} )
    except PredictionError as e:
        logger.error(f Prediction error: {e.message} )
        if e.input_shape:
            logger.error(f Input shape: {e.input_shape} )
    except Exception as e:
        logger.error(f Unexpected error: {str(e)} )
        logger.error(f Traceback: {traceback.format_exc()} )

# Run example
# example_error_handling()
```

Listing 27: Production-ready error handling and logging

5 5. Learning from Errors

5.1 5.1 Common Data Preprocessing Errors

```
# Buggy: Data leakage during preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# WRONG WAY - Data leakage
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)  # Using entire dataset!
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size
    =0.2)

# This causes data leakage because test data influences the scaling
    parameters

# CORRECT WAY - No data leakage
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# Fit preprocessing on training data only
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)  # Fit only on training
    data
X_test_scaled = scaler.transform(X_test)        # Transform test data
    using training parameters

print( Correct preprocessing completed without data leakage )
```

Listing 28: Error 5.1.1: Data Leakage During Preprocessing - Buggy Code

```python
# Buggy: Incorrect categorical variable handling
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# WRONG WAY - Problems with LabelEncoder for features
data = pd.DataFrame({
    'category': ['Low', 'Medium', 'High', 'Low', 'Medium'],
    'target': [0, 1, 1, 0, 1]
})

# This creates artificial ordering: Low=0, Medium=1, High=2
le = LabelEncoder()
data['category_encoded'] = le.fit_transform(data['category'])

print( Buggy encoding result: )
print(data)

# PROBLEMS:
# 1. Artificial ordinal relationship (Low < Medium < High)
# 2. Model might interpret this as numerical relationship
# 3. Can lead to incorrect predictions

# CORRECT WAY - Use OneHotEncoder for features
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(sparse=False, drop='first')  # drop='first' to avoid
    multicollinearity
category_encoded = ohe.fit_transform(data[['category']])

print( \nCorrect encoding result: )
print(category_encoded)
print( Categories: , ohe.categories_)

# For target variables, LabelEncoder is appropriate
target_le = LabelEncoder()
data['target_encoded'] = target_le.fit_transform(data['target'])
print( \nTarget encoding: , data['target_encoded'])
```

Listing 29: Error 5.1.2: Incorrect Handling of Categorical Variables - Buggy Code

```python
# Buggy: Improper missing value handling
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer

# WRONG WAY - Dropping important data
data = pd.DataFrame({
    'feature1': [1, 2, np.nan, 4, 5],
    'feature2': [10, np.nan, 30, 40, 50],
    'target': [0, 1, 1, 0, 1]
})

print( Original data: )
print(data)
print(f Missing values: {data.isnull().sum().sum()} )
```

```

# Buggy approach - Dropping rows with missing values
data_dropped = data.dropna()
print(f \nAfter dropping rows: {len(data_dropped)} rows remaining )
print( This loses 40% of data! )

# Another buggy approach - Filling with zeros
data_zero_filled = data.fillna(0)
print( \nZero-filled data: )
print(data_zero_filled)
# Problem: Zeros might not be meaningful and can skew results

# CORRECT WAY - Strategic imputation
# For numerical features, use median or mean
num_imputer = SimpleImputer(strategy='median')
data['feature1'] = num_imputer.fit_transform(data[['feature1']])

# For features with specific patterns, use domain knowledge
# Or use advanced imputation methods
from sklearn.impute import KNNImputer
knn_imputer = KNNImputer(n_neighbors=2)
data_imputed = pd.DataFrame(
    knn_imputer.fit_transform(data[['feature1', 'feature2']]),
    columns=['feature1', 'feature2']
)
data_imputed['target'] = data['target']

print( \nCorrectly imputed data: )
print(data_imputed)
```

Listing 30: Error 5.1.3: Improper Missing Value Handling - Buggy Code

```
# Buggy: Incorrect feature scaling on sparse data
import numpy as np
from scipy.sparse import csr_matrix
from sklearn.preprocessing import StandardScaler

# Create sparse data
sparse_data = np.array([
    [1, 0, 0, 0, 5],
    [0, 2, 0, 0, 0],
    [0, 0, 3, 0, 0],
    [0, 0, 0, 4, 0]
])

sparse_matrix = csr_matrix(sparse_data)
print( Original sparse matrix: )
print(sparse_matrix.toarray())
print(f Sparsity: {1 - sparse_matrix.nnz / (sparse_matrix.shape[0] *
    sparse_matrix.shape[1]):.2%} )

# WRONG WAY - Using StandardScaler on sparse data
scaler = StandardScaler()
try:
    scaled_data_wrong = scaler.fit_transform(sparse_matrix)
```

```
    print( \nAfter StandardScaler (WRONG): )
    print(scaled_data_wrong)
    print( Result is dense matrix - lost sparsity! )
except Exception as e:
    print(f Error: {e} )

# CORRECT WAY - Use appropriate scaling for sparse data
from sklearn.preprocessing import MaxAbsScaler

# MaxAbsScaler preserves sparsity
maxabs_scaler = MaxAbsScaler()
scaled_data_correct = maxabs_scaler.fit_transform(sparse_matrix)

print( \nAfter MaxAbsScaler (CORRECT): )
print(scaled_data_correct.toarray())
print(f Sparsity preserved: {1 - scaled_data_correct.nnz / (
    scaled_data_correct.shape[0] * scaled_data_correct.shape[1]):.2%} )

# Alternative: Use MinMaxScaler with feature_range=(-1, 1)
from sklearn.preprocessing import MinMaxScaler

minmax_scaler = MinMaxScaler(feature_range=(-1, 1))
scaled_data_minmax = minmax_scaler.fit_transform(sparse_matrix)

print( \nAfter MinMaxScaler (Alternative): )
print(scaled_data_minmax.toarray())
```

Listing 31: Error 5.1.4: Feature Scaling on Sparse Data - Buggy Code

```
# Buggy: Incorrect train-test split approaches
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

# Create time series data
dates = pd.date_range('2020-01-01', periods=1000, freq='D')
data = pd.DataFrame({
    'date': dates,
    'value': np.random.randn(1000).cumsum() + 100,
    'feature1': np.random.randn(1000),
    'feature2': np.random.randn(1000)
})

print( Data shape: , data.shape)
print( Date range: , data['date'].min(),  to , data['date'].max())

# WRONG WAY 1 - Random split on time series data
X = data[['feature1', 'feature2']]
y = data['value']

X_train_wrong, X_test_wrong, y_train_wrong, y_test_wrong =
    train_test_split(
    X, y, test_size=0.2, random_state=42
)

```

```
print( \nWRONG WAY 1 - Random split: )
print(f Train date range: {data.iloc[X_train_wrong.index]['date'].min()} to {data.iloc[X_train_wrong.index]['date'].max()} )
print(f Test date range: {data.iloc[X_test_wrong.index]['date'].min()} to {data.iloc[X_test_wrong.index]['date'].max()} )
print( Problem: Future data leaks into training! )

# WRONG WAY 2 - Incorrect stratification
categorical_data = pd.DataFrame({
    'feature': np.random.randn(1000),
    'category': np.random.choice(['A', 'B', 'C'], 1000, p=[0.95, 0.03, 0.02])
})

X_cat = categorical_data[['feature']]
y_cat = categorical_data['category']

# Not using stratification for imbalanced data
X_train_no_strat, X_test_no_strat, y_train_no_strat, y_test_no_strat = train_test_split(
    X_cat, y_cat, test_size=0.2, random_state=42
)

print(f \nWRONG WAY 2 - No stratification: )
print(f Original distribution: A={sum(y_cat=='A')}, B={sum(y_cat=='B')}, C={sum(y_cat=='C')} )
print(f Train distribution: A={sum(y_train_no_strat=='A')}, B={sum(y_train_no_strat=='B')}, C={sum(y_train_no_strat=='C')} )
print( Problem: Rare classes might be underrepresented in train/test sets! )

# CORRECT WAY 1 - Time series split
# Use temporal order for time series
split_date = data['date'].quantile(0.8)  # 80th percentile
train_mask = data['date'] <= split_date
test_mask = data['date'] > split_date

X_train_correct = X[train_mask]
X_test_correct = X[test_mask]
y_train_correct = y[train_mask]
y_test_correct = y[test_mask]

print( \nCORRECT WAY 1 - Temporal split: )
print(f Train date range: {data[train_mask]['date'].min()} to {data[train_mask]['date'].max()} )
print(f Test date range: {data[test_mask]['date'].min()} to {data[test_mask]['date'].max()} )

# CORRECT WAY 2 - Stratified split
X_train_strat, X_test_strat, y_train_strat, y_test_strat = train_test_split(
    X_cat, y_cat, test_size=0.2, random_state=42, stratify=y_cat
)

```

```
print(f \nCORRECT WAY 2 - Stratified split: )
print(f Train distribution: A={sum(y_train_strat=='A')}, B={sum(y_train_strat=='B')}, C={sum(y_train_strat=='C')} )
print(f Test distribution: A={sum(y_test_strat=='A')}, B={sum(y_test_strat=='B')}, C={sum(y_test_strat=='C')} )
```

Listing 32: Error 5.1.5: Incorrect Train-Test Split - Buggy Code

```
# Buggy: Data type conversion problems
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder

# Create problematic data
data = pd.DataFrame({
    'numeric_string': ['1', '2', '3', '4', '5'],
    'mixed_numeric': [1, 2.5, 3, 4.7, 5],
    'categorical_numeric': [1, 2, 3, 1, 2],  # Actually categories
    'boolean_like': ['True', 'False', 'True', 'False', 'True'],
    'date_string': ['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-04', '2020-01-05']
})

print( Original data types: )
print(data.dtypes)
print( \nOriginal data: )
print(data)

# WRONG WAY 1 - Incorrect numeric conversion
try:
    # This might work but can cause issues
    data['numeric_string_converted'] = pd.to_numeric(data['numeric_string'])
    print( \nNumeric string conversion: )
    print(data['numeric_string_converted'].dtype)
except Exception as e:
    print(f Error in numeric conversion: {e} )

# WRONG WAY 2 - Treating categorical as numeric
# Problem: categorical_numeric should be treated as categories, not numbers
le = LabelEncoder()
data['categorical_numeric_encoded'] = le.fit_transform(data['categorical_numeric'])
print(f \nCategorical numeric encoding: {data['categorical_numeric_encoded'].tolist()} )
print( Problem: Model might treat these as ordinal relationships! )

# WRONG WAY 3 - Boolean conversion issues
try:
    data['boolean_converted'] = data['boolean_like'].astype(bool)
    print(f \nBoolean conversion: {data['boolean_converted'].tolist()} )
    print( Problem: All strings convert to True! )
except Exception as e:
    print(f Error in boolean conversion: {e} )
```

```
43
44 # WRONG WAY 4 - Date parsing without proper format
45 try:
46     data['date_parsed_wrong'] = pd.to_datetime(data['date_string'])
47     print(f \nDate parsing (might work): {data['date_parsed_wrong'].dtype}
       )
48 except Exception as e:
49     print(f Error in date parsing: {e} )
50
51 # CORRECT WAY - Proper data type handling
52 # 1. Explicit numeric conversion with error handling
53 data['numeric_string_correct'] = pd.to_numeric(data['numeric_string'],
       errors='coerce')
54
55 # 2. Proper categorical handling
56 data['categorical_numeric_correct'] = data['categorical_numeric'].astype('
       category')
57
58 # 3. Boolean conversion with proper mapping
59 bool_mapping = {'True': True, 'False': False}
60 data['boolean_correct'] = data['boolean_like'].map(bool_mapping)
61
62 # 4. Explicit date parsing with format
63 data['date_correct'] = pd.to_datetime(data['date_string'], format='%Y-%m-%
       d')
64
65 print( \nCorrect data types: )
66 print(data.dtypes)
67
68 print( \nCorrectly processed data: )
69 print(data[['numeric_string_correct', 'categorical_numeric_correct',
70             'boolean_correct', 'date_correct']].head())
71
72 # Additional safety checks
73 def safe_convert_types(df):
74         Safely convert data types with proper error handling
75     df_converted = df.copy()
76
77     for column in df.columns:
78         # Try to infer and convert appropriate types
79         if df[column].dtype == 'object':
80             # Check if it's numeric
81             numeric_converted = pd.to_numeric(df[column], errors='coerce')
82             if not numeric_converted.isna().all():
83                 # Check if conversion lost too much data
84                 if numeric_converted.isna().sum() / len(df) < 0.1:  # Less
       than 10% loss
85                     df_converted[column] = numeric_converted
86                     continue
87
88             # Check if it's boolean-like
89             unique_values = df[column].unique()
90             if set(unique_values).issubset({True, False, 'True', 'False',
       'true', 'false', 1, 0}):
```

```
                bool_mapping = {True: True, False: False, 'True': True, '
    False': False,
                                'true': True, 'false': False, 1: True, 0:
    False}
                df_converted[column] = df[column].map(bool_mapping)
                continue

            # Check if it's date-like
            try:
                pd.to_datetime(df[column].iloc[:5])  # Test first 5 values
                df_converted[column] = pd.to_datetime(df[column])
                continue
            except:
                pass

            # Otherwise, keep as categorical if few unique values
            if df[column].nunique() / len(df) < 0.05:  # Less than 5%
    unique values
                df_converted[column] = df[column].astype('category')

     return df_converted

# Apply safe conversion
data_safe = safe_convert_types(data)
print( \nSafely converted data types: )
print(data_safe.dtypes)
```

Listing 33: Error 5.1.6: Data Type Conversion Issues - Buggy Code

```
# Buggy: Common feature engineering mistakes
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler

# Create sample data
np.random.seed(42)
data = pd.DataFrame({
    'feature1': np.random.randn(1000),
    'feature2': np.random.randn(1000),
    'category': np.random.choice(['A', 'B', 'C'], 1000),
    'date': pd.date_range('2020-01-01', periods=1000, freq='D')
})

print( Original data shape: , data.shape)

# WRONG WAY 1 - Creating data leakage features
# Buggy: Using future information to create features
def create_leaky_features_buggy(df):
       Buggy feature engineering that creates data leakage
    df_new = df.copy()

    # WRONG: Using mean of entire dataset (data leakage!)
    df_new['feature1_normalized'] = df_new['feature1'] / df_new['feature1'
    ].mean()

```

```
    # WRONG: Using global statistics instead of training set statistics
    df_new['feature2_scaled'] = (df_new['feature2'] - df_new['feature2'].min()) / (df_new['feature2'].max() - df_new['feature2'].min())

    # WRONG: Creating features that wouldn't be available at prediction time
    df_new['future_trend'] = df_new['feature1'].rolling(window=10).mean().shift(-5)  # Future data!

    return df_new

# Apply buggy feature engineering
data_leaky = create_leaky_features_buggy(data)
print( Buggy feature engineering completed )
print( Problems: Data leakage, future data usage )

# WRONG WAY 2 - Overfitting through excessive feature creation
def create_overfitting_features_buggy(df):
       Buggy: Creating too many features leading to overfitting
    df_new = df.copy()

    # WRONG: Creating polynomial features for all combinations
    for i in range(10):  # Too many features!
        for j in range(10):
            df_new[f'feature1_poly_{i}_feature2_poly_{j}'] = (df_new['feature1'] ** i) * (df_new['feature2'] ** j)

    # WRONG: Creating features for every possible combination
    categories = df_new['category'].unique()
    for cat1 in categories:
        for cat2 in categories:
            df_new[f'category_interaction_{cat1}_{cat2}'] = ((df_new['category'] == cat1) & (df_new['category'] == cat2)).astype(int)

    return df_new

# WRONG WAY 3 - Inconsistent feature engineering
def inconsistent_feature_engineering(df_train, df_test):
       Buggy: Inconsistent feature engineering between train and test
    # WRONG: Different transformations on train vs test
    df_train_new = df_train.copy()
    df_test_new = df_test.copy()

    # Train gets log transformation
    df_train_new['feature1_transformed'] = np.log(df_train['feature1'] + 1)

    # Test gets square root transformation (INCONSISTENT!)
    df_test_new['feature1_transformed'] = np.sqrt(df_test['feature1'])

    return df_train_new, df_test_new

# CORRECT WAY - Proper feature engineering practices
def create_proper_features(df_train, df_test=None):
```

```
    Correct feature engineering with proper practices

# 1. Feature engineering on training data only
df_train_new = df_train.copy()

# Safe feature creation using training statistics only
feature1_mean = df_train['feature1'].mean()
feature1_std = df_train['feature1'].std()
feature2_min = df_train['feature2'].min()
feature2_max = df_train['feature2'].max()

df_train_new['feature1_normalized'] = (df_train['feature1'] - feature1_mean) / feature1_std
df_train_new['feature2_scaled'] = (df_train['feature2'] - feature2_min) / (feature2_max - feature2_min)

# 2. Safe date features (no future data)
df_train_new['year'] = df_train['date'].dt.year
df_train_new['month'] = df_train['date'].dt.month
df_train_new['day_of_week'] = df_train['date'].dt.dayofweek
df_train_new['is_weekend'] = df_train['date'].dt.dayofweek.isin([5, 6]).astype(int)

# 3. Safe categorical encoding
df_train_new = pd.get_dummies(df_train_new, columns=['category'], prefix='cat')

# 4. Lag features (past information only)
df_train_new['feature1_lag_1'] = df_train['feature1'].shift(1)
df_train_new['feature1_rolling_mean_3'] = df_train['feature1'].rolling(window=3).mean()

# 5. If test data provided, apply same transformations
if df_test is not None:
    df_test_new = df_test.copy()

    # Apply same transformations using training statistics
    df_test_new['feature1_normalized'] = (df_test['feature1'] - feature1_mean) / feature1_std
    df_test_new['feature2_scaled'] = (df_test['feature2'] - feature2_min) / (feature2_max - feature2_min)

    # Date features
    df_test_new['year'] = df_test['date'].dt.year
    df_test_new['month'] = df_test['date'].dt.month
    df_test_new['day_of_week'] = df_test['date'].dt.dayofweek
    df_test_new['is_weekend'] = df_test['date'].dt.dayofweek.isin([5, 6]).astype(int)

    # Categorical encoding (ensure same columns)
    df_test_new = pd.get_dummies(df_test_new, columns=['category'], prefix='cat')

    # Ensure same columns in both datasets
```

```
        train_cols = set(df_train_new.columns) - {'feature1', 'feature2', 'date', 'category'}
        test_cols = set(df_test_new.columns) - {'feature1', 'feature2', 'date', 'category'}

        # Add missing columns with zeros
        for col in train_cols - test_cols:
            df_test_new[col] = 0
        for col in test_cols - train_cols:
            df_train_new[col] = 0

        # Reorder columns to match
        common_cols = sorted(list(train_cols & test_cols))
        df_train_new = df_train_new[common_cols]
        df_test_new = df_test_new[common_cols]

        return df_train_new, df_test_new

    return df_train_new

# Demonstrate correct approach
train_data = data.iloc[:800]  # First 800 rows for training
test_data = data.iloc[800:]   # Last 200 rows for testing

train_features, test_features = create_proper_features(train_data, test_data)

print(f \nCorrect feature engineering: )
print(f Training features shape: {train_features.shape} )
print(f Testing features shape: {test_features.shape} )
print(f Feature columns: {list(train_features.columns)} )

# Additional best practices for feature engineering
class FeatureEngineer:
       Robust feature engineering class

    def __init__(self):
        self.feature_stats = {}
        self.categorical_mappings = {}

    def fit_transform(self, df, target_col=None):
           Fit feature engineering parameters and transform training data

        df_transformed = df.copy()

        # Store statistics for numerical features
        numerical_cols = df.select_dtypes(include=[np.number]).columns
        if target_col in numerical_cols:
            numerical_cols = numerical_cols.drop(target_col)

        for col in numerical_cols:
            self.feature_stats[col] = {
                'mean': df[col].mean(),
                'std': df[col].std(),
```

```
                'min': df[col].min(),
                'max': df[col].max()
            }

            # Apply standardization
            mean_val = self.feature_stats[col]['mean']
            std_val = self.feature_stats[col]['std']
            if std_val > 0:
                df_transformed[f'{col}_standardized'] = (df[col] -
    mean_val) / std_val

        # Handle categorical features
        categorical_cols = df.select_dtypes(include=['object', 'category'
    ]).columns
        for col in categorical_cols:
            # Store value counts for reference
            self.categorical_mappings[col] = df[col].value_counts().
    to_dict()

            # One-hot encoding
            dummies = pd.get_dummies(df[col], prefix=col)
            df_transformed = pd.concat([df_transformed, dummies], axis=1)

        return df_transformed

    def transform(self, df):
            Transform new data using fitted parameters
        df_transformed = df.copy()

        # Apply same transformations using stored parameters
        for col, stats in self.feature_stats.items():
            if col in df.columns:
                mean_val = stats['mean']
                std_val = stats['std']
                if std_val > 0:
                    df_transformed[f'{col}_standardized'] = (df[col] -
    mean_val) / std_val

        # Handle categorical features consistently
        for col, mappings in self.categorical_mappings.items():
            if col in df.columns:
                dummies = pd.get_dummies(df[col], prefix=col)
                # Ensure same columns as training
                for dummy_col in [f'{col}_{val}' for val in mappings.keys
    ()]:
                    if dummy_col not in dummies.columns:
                        dummies[dummy_col] = 0
                df_transformed = pd.concat([df_transformed, dummies], axis
    =1)

        return df_transformed

# Example usage
engineer = FeatureEngineer()
```

```
train_transformed = engineer.fit_transform(train_data)
test_transformed = engineer.transform(test_data)

print(f \nUsing FeatureEngineer class: )
print(f Training transformed shape: {train_transformed.shape} )
print(f Testing transformed shape: {test_transformed.shape} )
```

Listing 34: Error 5.1.7: Feature Engineering Mistakes - Buggy Code

```
# Buggy: Ignoring data quality issues
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

# Create problematic data with quality issues
np.random.seed(42)
data = pd.DataFrame({
    'feature1': np.random.randn(1000),
    'feature2': np.random.randn(1000),
    'feature3': np.random.choice([1, 2, 3, 999], 1000, p=[0.3, 0.3, 0.3, 0.1]),  # 999 is outlier
    'category': np.random.choice(['A', 'B', 'C', ' '], 1000, p=[0.4, 0.4, 0.1, 0.1]),  # Empty strings
    'target': np.random.choice([0, 1], 1000, p=[0.7, 0.3])
})

# Add some missing values and duplicates
data.loc[100:105, 'feature1'] = np.nan
data.loc[200:202] = data.loc[100:102].values  # Duplicate rows

print( Original data shape: , data.shape)
print( Missing values: )
print(data.isnull().sum())
print( \nDuplicate rows: , data.duplicated().sum())
print( \nValue counts for category: )
print(data['category'].value_counts())

# WRONG WAY - Ignoring data quality issues
def train_model_buggy(X, y):
        Buggy model training ignoring data quality
    # No data quality checks!
    model = RandomForestClassifier(random_state=42)
    model.fit(X, y)
    return model

# Prepare data the buggy way
X_buggy = data.drop('target', axis=1)
y_buggy = data['target']

# This will cause issues:
# 1. Missing values in X
# 2. Categorical variables not encoded
# 3. Outliers not handled
# 4. Duplicates not removed
```

```
45
46 try:
47     model_buggy = train_model_buggy(X_buggy, y_buggy)
48     print( Buggy model training completed (but with issues!) )
49 except Exception as e:
50     print(f Buggy approach failed: {e} )
51
52 # CORRECT WAY - Comprehensive data quality handling
53 def comprehensive_data_quality_check(df):
54        Comprehensive data quality assessment
55     quality_report = {
56         'shape': df.shape,
57         'missing_values': df.isnull().sum().to_dict(),
58         'duplicate_rows': df.duplicated().sum(),
59         'data_types': df.dtypes.to_dict(),
60         'issues': []
61     }
62
63     # Check for missing values
64     missing_cols = df.isnull().sum()
65     for col, count in missing_cols.items():
66         if count > 0:
67             percentage = (count / len(df)) * 100
68             quality_report['issues'].append(f Column '{col}' has {count}
    ({percentage:.1f}%) missing values )
69
70     # Check for duplicates
71     if quality_report['duplicate_rows'] > 0:
72         quality_report['issues'].append(f Found {quality_report['
    duplicate_rows']} duplicate rows )
73
74     # Check for suspicious values (outliers, invalid categories)
75     for col in df.select_dtypes(include=[np.number]).columns:
76         Q1 = df[col].quantile(0.25)
77         Q3 = df[col].quantile(0.75)
78         IQR = Q3 - Q1
79         outliers = df[(df[col] < (Q1 - 1.5 * IQR)) | (df[col] > (Q3 + 1.5
    * IQR))]
80         if len(outliers) > 0:
81             outlier_percentage = (len(outliers) / len(df)) * 100
82             if outlier_percentage > 5:  # More than 5% outliers
83                 quality_report['issues'].append(f Column '{col}' has {len(
    outliers)} ({outlier_percentage:.1f}%) outliers )
84
85     for col in df.select_dtypes(include=['object']).columns:
86         empty_values = df[col].isin(['', ' ', 'None', 'null']).sum()
87         if empty_values > 0:
88             percentage = (empty_values / len(df)) * 100
89             quality_report['issues'].append(f Column '{col}' has {
    empty_values} ({percentage:.1f}%) empty/invalid values )
90
91     return quality_report
92
93 def clean_data_properly(df, target_col):
```

```
    Properly clean data with quality checks
    df_clean = df.copy()

    # Remove duplicates
    duplicates_before = df_clean.duplicated().sum()
    df_clean = df_clean.drop_duplicates()
    print(f Removed {duplicates_before - df_clean.duplicated().sum()} duplicate rows )

    # Handle missing values
    missing_before = df_clean.isnull().sum().sum()

    # For numerical columns, fill with median
    numerical_cols = df_clean.select_dtypes(include=[np.number]).columns
    if target_col in numerical_cols:
        numerical_cols = numerical_cols.drop(target_col)

    for col in numerical_cols:
        if df_clean[col].isnull().sum() > 0:
            median_val = df_clean[col].median()
            df_clean[col].fillna(median_val, inplace=True)

    # For categorical columns, fill with mode or create 'Unknown' category
    categorical_cols = df_clean.select_dtypes(include=['object']).columns
    for col in categorical_cols:
        if df_clean[col].isnull().sum() > 0:
            mode_val = df_clean[col].mode()
            if len(mode_val) > 0:
                df_clean[col].fillna(mode_val[0], inplace=True)
            else:
                df_clean[col].fillna('Unknown', inplace=True)

    # Handle empty strings
    for col in categorical_cols:
        df_clean[col] = df_clean[col].replace(['', ' ', 'None', 'null'], 'Unknown')

    # Handle outliers (capping approach)
    for col in numerical_cols:
        Q1 = df_clean[col].quantile(0.25)
        Q3 = df_clean[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        # Cap outliers instead of removing them
        df_clean[col] = df_clean[col].clip(lower=lower_bound, upper=upper_bound)

    print(f Handled {missing_before - df_clean.isnull().sum().sum()} missing values )

    return df_clean

```

```
# Apply proper data quality handling
print( === DATA QUALITY ASSESSMENT === )
quality_report = comprehensive_data_quality_check(data)
for issue in quality_report['issues']:
    print(f   - {issue} )

print( \n=== CLEANING DATA === )
data_clean = clean_data_properly(data, 'target')

print(f \nAfter cleaning: )
print(f   Shape: {data_clean.shape} )
print(f   Missing values: {data_clean.isnull().sum().sum()} )
print(f   Duplicates: {data_clean.duplicated().sum()} )

# Proper model training with clean data
def train_model_properly(df, target_col):
       Proper model training with data quality checks

    # Separate features and target
    X = df.drop(target_col, axis=1)
    y = df[target_col]

    # Encode categorical variables
    X_encoded = pd.get_dummies(X, drop_first=True)

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X_encoded, y, test_size=0.2, random_state=42, stratify=y
    )

    # Train model
    model = RandomForestClassifier(random_state=42, n_estimators=100)
    model.fit(X_train, y_train)

    # Evaluate
    train_score = model.score(X_train, y_train)
    test_score = model.score(X_test, y_test)

    print(f Training accuracy: {train_score:.4f} )
    print(f Testing accuracy: {test_score:.4f} )

    return model, X_train, X_test, y_train, y_test

# Train model with clean data
print( \n=== TRAINING MODEL WITH CLEAN DATA === )
model_proper, X_train, X_test, y_train, y_test = train_model_properly(
   data_clean, 'target')

# Advanced data quality monitoring
class DataQualityMonitor:
       Monitor data quality in production

    def __init__(self):
        self.baseline_stats = {}
```

```

    def set_baseline(self, df):
           Set baseline statistics for monitoring
        self.baseline_stats = {}

        for col in df.columns:
            if df[col].dtype in ['int64', 'float64']:
                self.baseline_stats[col] = {
                    'mean': df[col].mean(),
                    'std': df[col].std(),
                    'min': df[col].min(),
                    'max': df[col].max()
                }
            else:
                self.baseline_stats[col] = {
                    'value_counts': df[col].value_counts().to_dict()
                }

    def check_drift(self, new_df, threshold=0.1):
           Check for data drift
        drift_alerts = []

        for col in new_df.columns:
            if col in self.baseline_stats and new_df[col].dtype in ['int64
    ', 'float64']:
                current_mean = new_df[col].mean()
                baseline_mean = self.baseline_stats[col]['mean']

                if baseline_mean != 0:
                    drift = abs(current_mean - baseline_mean) / abs(
    baseline_mean)
                    if drift > threshold:
                        drift_alerts.append(f Column '{col}' drift: {drift
    :.4f} )

        return drift_alerts

# Example usage of data quality monitoring
monitor = DataQualityMonitor()
monitor.set_baseline(data_clean)

# Simulate new data
new_data = data_clean.sample(100)
drift_alerts = monitor.check_drift(new_data)

if drift_alerts:
    print( \n=== DATA DRIFT ALERTS === )
    for alert in drift_alerts:
        print(f   - {alert} )
else:
    print( \nNo significant data drift detected )
```

Listing 35: Error 5.1.8: Data Quality Issues - Buggy Code

5.2 5.2 Model Training and Evaluation Errors

```
# Buggy: Incorrect cross-validation usage
import numpy as np
import pandas as pd
from sklearn.model_selection import cross_val_score, KFold,
    train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Create time series data
np.random.seed(42)
dates = pd.date_range('2020-01-01', periods=1000, freq='D')
data = pd.DataFrame({
    'date': dates,
    'feature1': np.random.randn(1000).cumsum(),
    'feature2': np.random.randn(1000),
    'target': np.random.choice([0, 1], 1000, p=[0.7, 0.3])
})

X = data[['feature1', 'feature2']]
y = data['target']

print( Data shape: , X.shape)
print( Date range: , data['date'].min(),  to , data['date'].max())

# WRONG WAY 1 - Using regular CV on time series data
print( \n=== WRONG WAY 1: Regular CV on Time Series === )

# This is wrong because it mixes future and past data
model = RandomForestClassifier(random_state=42)
cv_scores_wrong = cross_val_score(model, X, y, cv=5)

print( Cross-validation scores (WRONG for time series): )
print(f   Mean: {cv_scores_wrong.mean():.4f} (+/- {cv_scores_wrong.std() *
    2:.4f}) )
print( Problem: Future data leaks into training! )

# WRONG WAY 2 - Data leakage in CV
print( \n=== WRONG WAY 2: Data Leakage in CV === )

# Split data first, then do CV on entire dataset (data leakage!)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# WRONG: Doing CV on entire dataset after splitting
cv_scores_leakage = cross_val_score(model, X, y, cv=5)  # Uses test data!

print( CV scores with data leakage: )
print(f   Mean: {cv_scores_leakage.mean():.4f} )
print( Problem: Test data used in cross-validation! )

# WRONG WAY 3 - Inconsistent preprocessing in CV
```

```
print( \n=== WRONG WAY 3: Inconsistent Preprocessing === )

from sklearn.preprocessing import StandardScaler

# WRONG: Fitting scaler on entire dataset before CV
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)  # Data leakage!

cv_scores_inconsistent = cross_val_score(model, X_scaled, y, cv=5)

print( CV scores with inconsistent preprocessing: )
print(f   Mean: {cv_scores_inconsistent.mean():.4f} )
print( Problem: Scaler fitted on entire dataset! )

# WRONG WAY 4 - Wrong CV for imbalanced data
print( \n=== WRONG WAY 4: Wrong CV for Imbalanced Data === )

# Create imbalanced dataset
imbalanced_data = pd.DataFrame({
    'feature1': np.random.randn(1000),
    'feature2': np.random.randn(1000),
    'target': np.random.choice([0, 1], 1000, p=[0.95, 0.05])  # 95% class
   0
})

X_imb = imbalanced_data[['feature1', 'feature2']]
y_imb = imbalanced_data['target']

print(f Class distribution: {np.bincount(y_imb)} )

# WRONG: Regular CV on imbalanced data
cv_scores_imbalanced = cross_val_score(model, X_imb, y_imb, cv=5)

print( CV scores on imbalanced data (regular CV): )
print(f   Mean: {cv_scores_imbalanced.mean():.4f} )
print( Problem: May not represent minority class performance! )

# CORRECT WAY 1 - Time Series Cross-Validation
print( \n=== CORRECT WAY 1: Time Series CV === )

from sklearn.model_selection import TimeSeriesSplit

# Use TimeSeriesSplit for time series data
tscv = TimeSeriesSplit(n_splits=5)
cv_scores_correct_ts = cross_val_score(model, X, y, cv=tscv)

print( Time series cross-validation scores: )
print(f   Mean: {cv_scores_correct_ts.mean():.4f} (+/- {
   cv_scores_correct_ts.std() * 2:.4f}) )

# CORRECT WAY 2 - Proper data splitting
print( \n=== CORRECT WAY 2: Proper Data Splitting === )

# Split first, then do CV only on training data
```

```
X_train_correct, X_test_correct, y_train_correct, y_test_correct =
    train_test_split(
    X, y, test_size=0.2, random_state=42
)

# CV only on training data
cv_scores_correct_split = cross_val_score(model, X_train_correct,
    y_train_correct, cv=5)

print( CV scores on training data only: )
print(f   Mean: {cv_scores_correct_split.mean():.4f} (+/- {
    cv_scores_correct_split.std() * 2:.4f}) )

# Final evaluation on test set
model_final = RandomForestClassifier(random_state=42)
model_final.fit(X_train_correct, y_train_correct)
test_score = model_final.score(X_test_correct, y_test_correct)
print(f Final test score: {test_score:.4f} )

# CORRECT WAY 3 - Pipeline with proper preprocessing
print( \n=== CORRECT WAY 3: Pipeline with Preprocessing === )

from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_validate

# Create pipeline that handles preprocessing within CV
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', RandomForestClassifier(random_state=42))
])

# CV will properly handle preprocessing within each fold
cv_results = cross_validate(pipeline, X, y, cv=5, return_train_score=True)

print( Pipeline CV scores: )
print(f   Train mean: {cv_results['train_score'].mean():.4f} )
print(f   Test mean: {cv_results['test_score'].mean():.4f} )

# CORRECT WAY 4 - Stratified CV for imbalanced data
print( \n=== CORRECT WAY 4: Stratified CV for Imbalanced Data === )

from sklearn.model_selection import StratifiedKFold

# Use StratifiedKFold to maintain class distribution
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cv_scores_stratified = cross_val_score(model, X_imb, y_imb, cv=skf)

print( Stratified CV scores on imbalanced data: )
print(f   Mean: {cv_scores_stratified.mean():.4f} (+/- {
    cv_scores_stratified.std() * 2:.4f}) )

# Compare with regular CV
cv_scores_regular = cross_val_score(model, X_imb, y_imb, cv=5)
print(f Regular CV scores: {cv_scores_regular.mean():.4f} )
```

```
152
153 print( Stratified CV better represents minority class performance! )
154
155 # Advanced CV techniques
156 class AdvancedCrossValidation:
157         Advanced cross-validation techniques
158
159     @staticmethod
160     def nested_cv(model, param_grid, X, y, outer_cv=5, inner_cv=3):
161             Nested cross-validation for unbiased model evaluation
162         from sklearn.model_selection import GridSearchCV
163
164         outer_scores = []
165
166         if isinstance(outer_cv, int):
167             outer_cv = KFold(n_splits=outer_cv, shuffle=True, random_state
    =42)
168
169         for train_idx, test_idx in outer_cv.split(X):
170             X_train_outer, X_test_outer = X.iloc[train_idx], X.iloc[
    test_idx]
171             y_train_outer, y_test_outer = y.iloc[train_idx], y.iloc[
    test_idx]
172
173             # Inner CV for hyperparameter tuning
174             grid_search = GridSearchCV(
175                 model, param_grid, cv=inner_cv, scoring='accuracy'
176             )
177             grid_search.fit(X_train_outer, y_train_outer)
178
179             # Evaluate best model on outer test set
180             best_model = grid_search.best_estimator_
181             score = best_model.score(X_test_outer, y_test_outer)
182             outer_scores.append(score)
183
184         return np.array(outer_scores)
185
186     @staticmethod
187     def group_cv(model, X, y, groups, cv=5):
188             Cross-validation with group constraints
189         from sklearn.model_selection import GroupKFold
190
191         group_kfold = GroupKFold(n_splits=cv)
192         scores = cross_val_score(model, X, y, cv=group_kfold, groups=
    groups)
193         return scores
194
195 # Example of nested CV
196 print( \n=== ADVANCED: Nested Cross-Validation === )
197
198 param_grid = {
199     'n_estimators': [50, 100, 200],
200     'max_depth': [3, 5, 7]
201 }
```

```

nested_scores = AdvancedCrossValidation.nested_cv(
    RandomForestClassifier(random_state=42),
    param_grid,
    X_train_correct,
    y_train_correct
)

print( Nested CV scores: )
print(f   Mean: {nested_scores.mean():.4f} (+/- {nested_scores.std() *
   2:.4f}) )
print( Unbiased estimate of model performance! )
```

Listing 36: Error 5.2.1: Incorrect Cross-Validation Usage - Buggy Code

```
# Buggy: Common hyperparameter tuning mistakes
import numpy as np
import pandas as pd
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV,
   train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler

# Create sample data
np.random.seed(42)
X = np.random.randn(1000, 10)
y = np.random.choice([0, 1], 1000, p=[0.7, 0.3])

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

print( Data shape: , X.shape)
print( Train/Test split: , X_train.shape, X_test.shape)

# WRONG WAY 1 - Tuning on test set (data leakage)
print( \n=== WRONG WAY 1: Tuning on Test Set === )

# This is completely wrong - using test set for hyperparameter tuning!
param_grid_wrong = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 7]
}

grid_search_wrong = GridSearchCV(
    RandomForestClassifier(random_state=42),
    param_grid_wrong,
    cv=5,
    scoring='accuracy'
)

# WRONG: Fitting on entire dataset including test data
grid_search_wrong.fit(np.vstack([X_train, X_test]), np.hstack([y_train,
   y_test]))
```

```
38
39 print( Best parameters (WRONG - data leakage): , grid_search_wrong.
       best_params_)
40 print( Best score: , grid_search_wrong.best_score_)
41
42 # Test on same data - overly optimistic!
43 test_score_wrong = grid_search_wrong.score(X_test, y_test)
44 print( Test score (overly optimistic): , test_score_wrong)
45 print( Problem: Test set used in hyperparameter tuning! )
46
47 # WRONG WAY 2 - No proper validation set
48 print( \n=== WRONG WAY 2: No Proper Validation Set === )
49
50 # WRONG: Using GridSearchCV without proper separation
51 grid_search_no_val = GridSearchCV(
52     RandomForestClassifier(random_state=42),
53     param_grid_wrong,
54     cv=3,  # Too few folds
55     scoring='accuracy'
56 )
57
58 # Fitting on training data is correct, but...
59 grid_search_no_val.fit(X_train, y_train)
60
61 # But then evaluating on test set without considering overfitting to
       validation folds
62 best_model_no_val = grid_search_no_val.best_estimator_
63 test_score_no_val = best_model_no_val.score(X_test, y_test)
64
65 print( Best parameters: , grid_search_no_val.best_params_)
66 print( CV score: , grid_search_no_val.best_score_)
67 print( Test score: , test_score_no_val)
68 print( Problem: May be overfitting to validation folds! )
69
70 # WRONG WAY 3 - Exhaustive search with too many parameters
71 print( \n=== WRONG WAY 3: Exhaustive Search with Too Many Parameters === )
72
73 # WRONG: Too many parameter combinations
74 param_grid_exhaustive = {
75     'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],  # 7 values
76     'gamma': [0.001, 0.01, 0.1, 1, 10, 100],     # 6 values
77     'kernel': ['rbf', 'poly', 'sigmoid']          # 3 values
78 }
79
80 # Total combinations: 7    6    3 = 126 combinations!
81 # With 5-fold CV: 126    5 = 630 model fits!
82
83 print(f Total combinations: {7 * 6 * 3} )
84 print(f Total model fits with 5-fold CV: {7 * 6 * 3 * 5} )
85 print( Problem: Computationally expensive, may not be necessary! )
86
87 # WRONG WAY 4 - Inappropriate scoring metric
88 print( \n=== WRONG WAY 4: Inappropriate Scoring Metric === )
89
```

```
# For imbalanced dataset, accuracy might not be the best metric
imbalanced_y = np.random.choice([0, 1], 1000, p=[0.95, 0.05])

param_grid_metric = {
    'n_estimators': [50, 100],
    'max_depth': [3, 5]
}

# WRONG: Using accuracy for imbalanced data
grid_search_accuracy = GridSearchCV(
    RandomForestClassifier(random_state=42),
    param_grid_metric,
    cv=5,
    scoring='accuracy'  # Not appropriate for imbalanced data
)

grid_search_accuracy.fit(X_train, imbalanced_y[:800])

print( Best parameters with accuracy scoring: , grid_search_accuracy.
   best_params_)
print( Best accuracy score: , grid_search_accuracy.best_score_)
print( Problem: May favor models that predict majority class! )

# CORRECT WAY 1 - Proper train/validation/test split
print( \n=== CORRECT WAY 1: Proper Data Splitting === )

# Split into train/validation/test
X_temp, X_test_final, y_temp, y_test_final = train_test_split(
    X, y, test_size=0.2, random_state=42
)

X_train_final, X_val_final, y_train_final, y_val_final = train_test_split(
    X_temp, y_temp, test_size=0.25, random_state=42  # 0.25 * 0.8 = 0.2 of
    total
)

print(f Train: {X_train_final.shape}, Val: {X_val_final.shape}, Test: {
   X_test_final.shape} )

# Hyperparameter tuning on validation set
param_grid_correct = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 7, None]
}

# Grid search on training data with cross-validation
grid_search_correct = GridSearchCV(
    RandomForestClassifier(random_state=42),
    param_grid_correct,
    cv=5,
    scoring='accuracy',
    n_jobs=-1  # Use all processors
)

```

```
grid_search_correct.fit(X_train_final, y_train_final)

# Evaluate on validation set
val_score = grid_search_correct.score(X_val_final, y_val_final)
print("Best parameters:", grid_search_correct.best_params_)
print("CV score:", grid_search_correct.best_score_)
print("Validation score:", val_score)

# Final evaluation on test set
best_model = grid_search_correct.best_estimator_
test_score_final = best_model.score(X_test_final, y_test_final)
print("Final test score:", test_score_final)

# CORRECT WAY 2 - Randomized search for efficiency
print("\n=== CORRECT WAY 2: Randomized Search ===")

from scipy.stats import randint, uniform

# Use RandomizedSearchCV for efficiency
param_dist = {
    'n_estimators': randint(50, 300),
    'max_depth': randint(3, 20),
    'min_samples_split': randint(2, 20),
    'min_samples_leaf': randint(1, 10)
}

random_search = RandomizedSearchCV(
    RandomForestClassifier(random_state=42),
    param_distributions=param_dist,
    n_iter=50,  # Try 50 random combinations instead of all
    cv=5,
    scoring='accuracy',
    n_jobs=-1,
    random_state=42
)

random_search.fit(X_train, y_train)

print("Random search best parameters:", random_search.best_params_)
print("Random search best score:", random_search.best_score_)

# CORRECT WAY 3 - Appropriate scoring for imbalanced data
print("\n=== CORRECT WAY 3: Appropriate Scoring ===")

# For imbalanced data, use F1-score or AUC
from sklearn.metrics import make_scorer, f1_score

# Custom scorer for F1-score
f1_scorer = make_scorer(f1_score, average='weighted')

grid_search_f1 = GridSearchCV(
    RandomForestClassifier(random_state=42),
    param_grid_correct,
    cv=5,
```

```
    scoring=f1_scorer,  # Appropriate for imbalanced data
    n_jobs=-1
)

grid_search_f1.fit(X_train, imbalanced_y[:800])

print("Best parameters with F1 scoring:", grid_search_f1.best_params_)
print("Best F1 score:", grid_search_f1.best_score_)

# CORRECT WAY 4 - Pipeline with preprocessing
print("\n=== CORRECT WAY 4: Pipeline with Preprocessing ===")

from sklearn.pipeline import Pipeline

# Create pipeline that includes preprocessing
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', SVC(random_state=42))
])

param_grid_pipeline = {
    'classifier__C': [0.1, 1, 10],
    'classifier__gamma': [0.001, 0.01, 0.1],
    'classifier__kernel': ['rbf', 'linear']
}

grid_search_pipeline = GridSearchCV(
    pipeline,
    param_grid_pipeline,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

grid_search_pipeline.fit(X_train, y_train)

print("Pipeline best parameters:", grid_search_pipeline.best_params_)
print("Pipeline best score:", grid_search_pipeline.best_score_)

# Advanced hyperparameter tuning techniques
class AdvancedHyperparameterTuning:
    """Advanced hyperparameter tuning techniques"""

    @staticmethod
    def bayesian_optimization(X, y, model_class, param_space, n_iter=50):
        """Bayesian optimization using scikit-optimize"""
        try:
            from skopt import BayesSearchCV
            from skopt.space import Real, Integer, Categorical

            # Convert parameter space
            skopt_space = {}
            for param, values in param_space.items():
                if isinstance(values, list) and all(isinstance(v, (int, np
```

```
    .integer)) for v in values):
                    skopt_space[param] = Integer(min(values), max(values))
                elif isinstance(values, list) and all(isinstance(v, (float
    , np.floating)) for v in values):
                    skopt_space[param] = Real(min(values), max(values))
                elif isinstance(values, list):
                    skopt_space[param] = Categorical(values)
                else:
                    skopt_space[param] = values

            bayes_search = BayesSearchCV(
                estimator=model_class(random_state=42),
                search_spaces=skopt_space,
                n_iter=n_iter,
                cv=5,
                scoring='accuracy',
                n_jobs=-1,
                random_state=42
            )

            bayes_search.fit(X, y)
            return bayes_search

        except ImportError:
            print( scikit-optimize not installed. Using RandomizedSearchCV
     instead. )
            from scipy.stats import randint, uniform

            # Convert to scipy distributions
            param_dist = {}
            for param, values in param_space.items():
                if isinstance(values, list) and all(isinstance(v, (int, np
    .integer)) for v in values):
                    param_dist[param] = randint(min(values), max(values) +
     1)
                elif isinstance(values, list) and all(isinstance(v, (float
    , np.floating)) for v in values):
                    param_dist[param] = uniform(min(values), max(values) -
     min(values))
                else:
                    param_dist[param] = values

            random_search = RandomizedSearchCV(
                model_class(random_state=42),
                param_distributions=param_dist,
                n_iter=n_iter,
                cv=5,
                scoring='accuracy',
                n_jobs=-1,
                random_state=42
            )

            random_search.fit(X, y)
            return random_search
```

```

    @staticmethod
    def multi_metric_tuning(X, y, model, param_grid, scoring_metrics):
            Tune hyperparameters using multiple metrics
        from sklearn.model_selection import GridSearchCV

        multi_search = GridSearchCV(
            model,
            param_grid,
            cv=5,
            scoring=scoring_metrics,
            refit='f1_weighted',  # Which metric to use for final model
            n_jobs=-1
        )

        multi_search.fit(X, y)
        return multi_search

# Example of advanced tuning
print( \n=== ADVANCED: Bayesian Optimization === )

param_space = {
    'n_estimators': [50, 100, 200, 300],
    'max_depth': [3, 5, 7, 10, None],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 4, 8]
}

# This would use Bayesian optimization if scikit-optimize is installed
# advanced_search = AdvancedHyperparameterTuning.bayesian_optimization(
#     X_train, y_train, RandomForestClassifier, param_space, n_iter=30
# )

print( Advanced tuning techniques available for more efficient
   hyperparameter optimization! )
```

Listing 37: Error 5.2.2: Hyperparameter Tuning Mistakes - Buggy Code

```
# Buggy: Common model selection mistakes
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.ensemble import RandomForestClassifier,
   GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import StandardScaler

# Create different types of datasets to demonstrate various issues
np.random.seed(42)

# 1. Linearly separable data
X_linear = np.random.randn(1000, 2)
```

```
y_linear = (X_linear[:, 0] + X_linear[:, 1] > 0).astype(int)

# 2. Non-linear data (XOR pattern)
X_nonlinear = np.random.randn(1000, 2)
y_nonlinear = ((X_nonlinear[:, 0] > 0) ^ (X_nonlinear[:, 1] > 0)).astype(
    int)

# 3. High-dimensional sparse data
X_sparse = np.random.choice([0, 1], size=(1000, 1000), p=[0.95, 0.05])
y_sparse = np.random.choice([0, 1], 1000, p=[0.7, 0.3])

# 4. Imbalanced data
X_imbalanced = np.random.randn(1000, 5)
y_imbalanced = np.random.choice([0, 1], 1000, p=[0.95, 0.05])

print( Dataset shapes: )
print(f Linear: {X_linear.shape} )
print(f Non-linear: {X_nonlinear.shape} )
print(f Sparse: {X_sparse.shape} )
print(f Imbalanced: {X_imbalanced.shape} )

# WRONG WAY 1 - Always using the same model
print( \n=== WRONG WAY 1: Always Using Same Model === )

# WRONG: Always using Random Forest regardless of data characteristics
def wrong_model_selection(X, y):
        Always use Random Forest - not considering data characteristics
    model = RandomForestClassifier(random_state=42)
    model.fit(X, y)
    return model

# This works but might not be optimal
model_linear_wrong = wrong_model_selection(X_linear, y_linear)
model_nonlinear_wrong = wrong_model_selection(X_nonlinear, y_nonlinear)

print( Random Forest accuracy on linear data: , model_linear_wrong.score(
    X_linear, y_linear))
print( Random Forest accuracy on non-linear data: , model_nonlinear_wrong.
    score(X_nonlinear, y_nonlinear))
print( Problem: Not considering that linear models might be better for
    linear data! )

# WRONG WAY 2 - No proper model comparison
print( \n=== WRONG WAY 2: No Proper Model Comparison === )

# WRONG: Just trying a few models without systematic comparison
models_quick = {
    'RF': RandomForestClassifier(random_state=42),
    'SVM': SVC(random_state=42),
    'LR': LogisticRegression(random_state=42)
}

def quick_model_comparison(X, y, models):
        Quick comparison without proper validation
```

```
    results = {}
    for name, model in models.items():
        model.fit(X, y)  # WRONG: Fitting on entire dataset
        score = model.score(X, y)  # WRONG: Evaluating on training data
        results[name] = score
    return results

# This gives overly optimistic results
quick_results = quick_model_comparison(X_linear, y_linear, models_quick)
print( Quick comparison results (OVERLY OPTIMISTIC): )
for name, score in quick_results.items():
    print(f   {name}: {score:.4f} )
print( Problem: No proper validation, results are overly optimistic! )

# WRONG WAY 3 - Ignoring data preprocessing requirements
print( \n=== WRONG WAY 3: Ignoring Preprocessing === )

# WRONG: Using SVM without scaling on data that needs it
X_scaled_train, X_scaled_test, y_scaled_train, y_scaled_test =
    train_test_split(
    X_linear, y_linear, test_size=0.2, random_state=42
)

# Without scaling (WRONG for SVM)
svm_no_scaling = SVC(random_state=42)
svm_no_scaling.fit(X_scaled_train, y_scaled_train)
score_no_scaling = svm_no_scaling.score(X_scaled_test, y_scaled_test)

# With proper scaling
scaler = StandardScaler()
X_scaled_proper = scaler.fit_transform(X_linear)
X_scaled_train_proper, X_scaled_test_proper, y_scaled_train_proper,
    y_scaled_test_proper = train_test_split(
    X_scaled_proper, y_linear, test_size=0.2, random_state=42
)

svm_with_scaling = SVC(random_state=42)
svm_with_scaling.fit(X_scaled_train_proper, y_scaled_train_proper)
score_with_scaling = svm_with_scaling.score(X_scaled_test_proper,
    y_scaled_test_proper)

print( SVM without scaling: , score_no_scaling)
print( SVM with scaling: , score_with_scaling)
print( Problem: SVM performance significantly affected by scaling! )

# WRONG WAY 4 - Inappropriate models for data types
print( \n=== WRONG WAY 4: Inappropriate Models === )

# WRONG: Using tree-based models on high-dimensional sparse data
print( High-dimensional sparse data shape: , X_sparse.shape)

# Tree-based models don't work well with sparse high-dimensional data
rf_sparse_wrong = RandomForestClassifier(random_state=42)
rf_sparse_wrong.fit(X_sparse, y_sparse)
```

```
sparse_score = rf_sparse_wrong.score(X_sparse, y_sparse)

print( Random Forest on sparse data: , sparse_score)
print( Problem: Tree-based models often perform poorly on high-dimensional
    sparse data! )

# CORRECT WAY 1 - Systematic model selection based on data characteristics
print( \n=== CORRECT WAY 1: Systematic Model Selection === )

def analyze_data_characteristics(X, y):
       Analyze data characteristics for model selection
    characteristics = {
        'n_samples': X.shape[0],
        'n_features': X.shape[1],
        'feature_to_sample_ratio': X.shape[1] / X.shape[0],
        'sparsity': np.mean(X == 0) if X.shape[1] > 100 else 0,
        'class_balance': min(np.bincount(y)) / max(np.bincount(y)),
        'linearity_indicator': None  # Would require more complex analysis
    }

    return characteristics

def recommend_models(characteristics):
       Recommend appropriate models based on data characteristics
    recommendations = []

    # Feature-to-sample ratio analysis
    if characteristics['feature_to_sample_ratio'] > 0.5:
        recommendations.append( Use regularization (Ridge, Lasso) or
   dimensionality reduction )
        recommendations.append( Avoid tree-based models )

    # Sparsity analysis
    if characteristics['sparsity'] > 0.8:
        recommendations.append( Use linear models with L1 regularization )
        recommendations.append( Consider Naive Bayes for text-like data )

    # Class balance analysis
    if characteristics['class_balance'] < 0.1:
        recommendations.append( Use appropriate metrics (F1, AUC) )
        recommendations.append( Consider resampling techniques )

    # General recommendations
    recommendations.append( Use cross-validation for model comparison )
    recommendations.append( Consider preprocessing requirements )

    return recommendations

# Analyze datasets
char_linear = analyze_data_characteristics(X_linear, y_linear)
char_sparse = analyze_data_characteristics(X_sparse, y_sparse)

print( Linear data characteristics: )
for key, value in char_linear.items():
```

```
    print(f   {key}: {value} )

print( \nSparse data characteristics: )
for key, value in char_sparse.items():
    print(f   {key}: {value} )

print( \nModel recommendations for linear data: )
for rec in recommend_models(char_linear):
    print(f   - {rec} )

print( \nModel recommendations for sparse data: )
for rec in recommend_models(char_sparse):
    print(f   - {rec} )

# CORRECT WAY 2 - Proper model comparison with cross-validation
print( \n=== CORRECT WAY 2: Proper Model Comparison === )

def proper_model_comparison(X, y, test_size=0.2):
       Proper model comparison with cross-validation

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=test_size, random_state=42
    )

    # Define models
    models = {
        'Logistic Regression': LogisticRegression(random_state=42,
   max_iter=1000),
        'Random Forest': RandomForestClassifier(random_state=42,
   n_estimators=100),
        'SVM': SVC(random_state=42, probability=True),
        'Naive Bayes': GaussianNB()
    }

    # Preprocessing pipeline
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    results = {}

    for name, model in models.items():
        # Use appropriate data (scaled for SVM and LR)
        if name in ['Logistic Regression', 'SVM']:
            X_train_use = X_train_scaled
            X_test_use = X_test_scaled
        else:
            X_train_use = X_train
            X_test_use = X_test

        # Cross-validation
        cv_scores = cross_val_score(model, X_train_use, y_train, cv=5)

```

```
        # Fit and test
        model.fit(X_train_use, y_train)
        test_score = model.score(X_test_use, y_test)

        results[name] = {
            'cv_mean': cv_scores.mean(),
            'cv_std': cv_scores.std(),
            'test_score': test_score
        }

    return results

# Compare models properly
print( Proper model comparison on linear data: )
linear_results = proper_model_comparison(X_linear, y_linear)

for name, scores in linear_results.items():
    print(f   {name}: )
    print(f     CV: {scores['cv_mean']:.4f} (+/- {scores['cv_std'] * 2:.4f
   }) )
    print(f     Test: {scores['test_score']:.4f} )

# CORRECT WAY 3 - Model selection for specific data types
print( \n=== CORRECT WAY 3: Data-Type Specific Selection === )

def select_model_for_data_type(X, y):
       Select appropriate model based on data type

    n_samples, n_features = X.shape

    # High-dimensional data
    if n_features > n_samples or n_features > 100:
        print( High-dimensional data detected )
        if np.mean(X == 0) > 0.8:  # Sparse data
            print( Using Naive Bayes (good for sparse data) )
            return GaussianNB()
        else:
            print( Using Logistic Regression with L1 regularization )
            return LogisticRegression(penalty='l1', solver='liblinear',
   random_state=42)

    # Low-dimensional data
    elif n_features < 20:
        print( Low-dimensional data detected )
        print( Using Random Forest (good general-purpose model) )
        return RandomForestClassifier(random_state=42, n_estimators=100)

    # Medium-dimensional data
    else:
        print( Medium-dimensional data detected )
        print( Using Gradient Boosting (good for structured data) )
        return GradientBoostingClassifier(random_state=42)

# Test model selection
```

```
print( Model selection for linear data: )
model_linear_correct = select_model_for_data_type(X_linear, y_linear)

print( \nModel selection for sparse data: )
model_sparse_correct = select_model_for_data_type(X_sparse, y_sparse)

# CORRECT WAY 4 - Ensemble methods for robust selection
print( \n=== CORRECT WAY 4: Ensemble Methods === )

from sklearn.ensemble import VotingClassifier

def create_ensemble_model(X, y):
       Create ensemble model combining multiple approaches

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    # Scale data for models that need it
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Create diverse set of models
    models = [
        ('lr', LogisticRegression(random_state=42, max_iter=1000)),
        ('rf', RandomForestClassifier(random_state=42, n_estimators=100)),
        ('svm', SVC(probability=True, random_state=42)),
        ('gb', GradientBoostingClassifier(random_state=42))
    ]

    # Create voting classifier
    ensemble = VotingClassifier(
        estimators=models,
        voting='soft'  # Use probability averaging
    )

    # Fit ensemble
    ensemble.fit(X_train_scaled, y_train)  # Use scaled data for all

    return ensemble, scaler

# Create ensemble for linear data
ensemble_linear, scaler_linear = create_ensemble_model(X_linear, y_linear)
ensemble_score = ensemble_linear.score(scaler_linear.transform(X_linear),
   y_linear)
print( Ensemble model accuracy on linear data: , ensemble_score)

# Advanced model selection techniques
class AdvancedModelSelector:
       Advanced model selection techniques

    @staticmethod
```

```
    def automated_model_selection(X, y, time_limit=300):
        """Automated model selection with time constraints"""
        import time
        from sklearn.model_selection import cross_validate

        models = {
            'LogisticRegression': LogisticRegression(random_state=42, max_iter=1000),
            'RandomForest': RandomForestClassifier(random_state=42, n_estimators=100),
            'SVM': SVC(random_state=42, probability=True),
            'GradientBoosting': GradientBoostingClassifier(random_state=42),
            'NaiveBayes': GaussianNB()
        }

        results = {}
        start_time = time.time()

        for name, model in models.items():
            # Check time limit
            if time.time() - start_time > time_limit:
                print(f"Time limit reached. Skipping {name}")
                continue

            try:
                # Preprocessing if needed
                if name in ['LogisticRegression', 'SVM']:
                    scaler = StandardScaler()
                    X_processed = scaler.fit_transform(X)
                else:
                    X_processed = X

                # Quick cross-validation
                cv_results = cross_validate(
                    model, X_processed, y,
                    cv=3,  # Quick CV
                    scoring='accuracy',
                    return_train_score=True
                )

                results[name] = {
                    'mean_cv_score': cv_results['test_score'].mean(),
                    'std_cv_score': cv_results['test_score'].std(),
                    'training_time': time.time() - start_time
                }

            except Exception as e:
                print(f"Error with {name}: {e}")
                continue

        # Select best model
        if results:
            best_model_name = max(results.keys(), key=lambda x: results[x
```

```
    ]['mean_cv_score'])
            print(f Best model: {best_model_name} )
            print(f CV Score: {results[best_model_name]['mean_cv_score
    ']:.4f} )
            return best_model_name, results[best_model_name]
        else:
            return None, None

    @staticmethod
    def model_complexity_analysis(X, y, model_class, complexity_param,
    param_values):
        Analyze model complexity vs performance
        from sklearn.model_selection import validation_curve

        train_scores, val_scores = validation_curve(
            model_class(random_state=42), X, y,
            param_name=complexity_param,
            param_range=param_values,
            cv=5, scoring='accuracy'
        )

        train_mean = np.mean(train_scores, axis=1)
        train_std = np.std(train_scores, axis=1)
        val_mean = np.mean(val_scores, axis=1)
        val_std = np.std(val_scores, axis=1)

        # Find optimal complexity
        optimal_idx = np.argmax(val_mean)
        optimal_value = param_values[optimal_idx]

        return {
            'train_mean': train_mean,
            'train_std': train_std,
            'val_mean': val_mean,
            'val_std': val_std,
            'optimal_value': optimal_value,
            'optimal_score': val_mean[optimal_idx]
        }

# Example of advanced model selection
print( \n=== ADVANCED: Automated Model Selection === )

# This would run automated selection
# best_model, best_results = AdvancedModelSelector.
    automated_model_selection(X_linear, y_linear)

print( Advanced model selection techniques provide automated and robust
    model selection! )

# Model selection based on problem requirements
def select_model_by_requirements(X, y, requirements):
        Select model based on specific requirements

    recommended_models = []
```

```

    if requirements.get('interpretability', False):
        recommended_models.extend(['LogisticRegression', 'DecisionTree'])

    if requirements.get('speed', False):
        recommended_models.extend(['LogisticRegression', 'NaiveBayes'])

    if requirements.get('accuracy', False):
        recommended_models.extend(['RandomForest', 'GradientBoosting'])

    if requirements.get('scalability', False):
        recommended_models.extend(['LogisticRegression', 'SGDClassifier'])

    return list(set(recommended_models))  # Remove duplicates

# Example usage
requirements = {
    'interpretability': True,
    'speed': True,
    'accuracy': True
}

recommended = select_model_by_requirements(X_linear, y_linear,
    requirements)
print(f \nModels recommended for requirements {requirements}: {recommended
    } )
```

Listing 38: Error 5.2.3: Model Selection Mistakes - Buggy Code

```
# Buggy: Common evaluation metric mistakes
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (accuracy_score, precision_score, recall_score
    , f1_score,
                             roc_auc_score, confusion_matrix,
    classification_report)
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler

# Create different types of datasets to demonstrate issues
np.random.seed(42)

# 1. Balanced dataset
X_balanced, y_balanced = make_classification(
    n_samples=1000, n_features=10, n_classes=2,
    weights=[0.5, 0.5], random_state=42
)

# 2. Imbalanced dataset (95% class 0, 5% class 1)
X_imbalanced, y_imbalanced = make_classification(
    n_samples=1000, n_features=10, n_classes=2,
    weights=[0.95, 0.05], random_state=42
```

```
)

# 3. Multi-class dataset
X_multiclass, y_multiclass = make_classification(
    n_samples=1000, n_features=10, n_classes=3,
    n_informative=8, random_state=42
)

print( Dataset shapes: )
print(f Balanced: {X_balanced.shape}, classes: {np.bincount(y_balanced)} )
print(f Imbalanced: {X_imbalanced.shape}, classes: {np.bincount(
    y_imbalanced)} )
print(f Multiclass: {X_multiclass.shape}, classes: {np.bincount(
    y_multiclass)} )

# WRONG WAY 1 - Using accuracy for imbalanced data
print( \n=== WRONG WAY 1: Accuracy for Imbalanced Data === )

# Train models
rf_balanced = RandomForestClassifier(random_state=42)
rf_imbalanced = RandomForestClassifier(random_state=42)

rf_balanced.fit(X_balanced, y_balanced)
rf_imbalanced.fit(X_imbalanced, y_imbalanced)

# WRONG: Using accuracy for imbalanced dataset
y_pred_imbalanced = rf_imbalanced.predict(X_imbalanced)
accuracy_wrong = accuracy_score(y_imbalanced, y_pred_imbalanced)

print( Imbalanced dataset results: )
print(f Class distribution: {np.bincount(y_imbalanced)} )
print(f Accuracy (MISLEADING): {accuracy_wrong:.4f} )

# The model might just predict the majority class all the time!
majority_class = np.bincount(y_imbalanced).argmax()
naive_accuracy = np.mean(y_imbalanced == majority_class)
print(f Naive classifier accuracy: {naive_accuracy:.4f} )
print( Problem: High accuracy doesn't mean good performance on minority
    class! )

# WRONG WAY 2 - Inappropriate metrics for multi-class
print( \n=== WRONG WAY 2: Inappropriate Multi-class Metrics === )

# Train on multi-class data
rf_multiclass = RandomForestClassifier(random_state=42)
rf_multiclass.fit(X_multiclass, y_multiclass)

y_pred_multiclass = rf_multiclass.predict(X_multiclass)

# WRONG: Using binary metrics without proper averaging
try:
    precision_wrong = precision_score(y_multiclass, y_pred_multiclass)  #
    Error!
except ValueError as e:
```

```
    print(f Error with precision_score: {e} )

# WRONG: Not specifying averaging method
try:
    recall_wrong = recall_score(y_multiclass, y_pred_multiclass)  # Error!
except ValueError as e:
    print(f Error with recall_score: {e} )

print( Problem: Multi-class requires averaging method specification! )

# WRONG WAY 3 - ROC AUC without probability scores
print( \n=== WRONG WAY 3: ROC AUC without Probabilities === )

# WRONG: Using predictions instead of probabilities for ROC AUC
try:
    auc_wrong = roc_auc_score(y_balanced, y_pred_balanced)  # Using
   predictions!
    print(f AUC with predictions (WRONG): {auc_wrong:.4f} )
except Exception as e:
    print(f Error with AUC calculation: {e} )

print( Problem: ROC AUC needs probability scores, not class predictions! )

# WRONG WAY 4 - Single metric evaluation
print( \n=== WRONG WAY 4: Single Metric Evaluation === )

# WRONG: Only looking at one metric
model1 = RandomForestClassifier(random_state=42)
model2 = LogisticRegression(random_state=42)

model1.fit(X_balanced, y_balanced)
model2.fit(X_balanced, y_balanced)

y_pred1 = model1.predict(X_balanced)
y_pred2 = model2.predict(X_balanced)

acc1 = accuracy_score(y_balanced, y_pred1)
acc2 = accuracy_score(y_balanced, y_pred2)

print( Single metric comparison: )
print(f Model 1 accuracy: {acc1:.4f} )
print(f Model 2 accuracy: {acc2:.4f} )
print( Problem: Single metric doesn't tell the full story! )

# CORRECT WAY 1 - Appropriate metrics for imbalanced data
print( \n=== CORRECT WAY 1: Appropriate Imbalanced Metrics === )

# Use multiple metrics for imbalanced data
def evaluate_imbalanced_model(y_true, y_pred, y_proba=None):
       Proper evaluation for imbalanced datasets

    results = {
        'accuracy': accuracy_score(y_true, y_pred),
        'precision': precision_score(y_true, y_pred, average='weighted'),
```

```
        'recall': recall_score(y_true, y_pred, average='weighted'),
        'f1': f1_score(y_true, y_pred, average='weighted')
    }

    # Per-class metrics for minority class
    precision_per_class = precision_score(y_true, y_pred, average=None)
    recall_per_class = recall_score(y_true, y_pred, average=None)

    # Focus on minority class (class 1 in our case)
    results['minority_precision'] = precision_per_class[1]
    results['minority_recall'] = recall_per_class[1]

    # ROC AUC if probabilities provided
    if y_proba is not None:
        results['auc'] = roc_auc_score(y_true, y_proba[:, 1])

    return results

# Proper evaluation
y_pred_imb_correct = rf_imbalanced.predict(X_imbalanced)
y_proba_imb_correct = rf_imbalanced.predict_proba(X_imbalanced)

results_imb = evaluate_imbalanced_model(
    y_imbalanced, y_pred_imb_correct, y_proba_imb_correct
)

print( Proper imbalanced evaluation: )
for metric, value in results_imb.items():
    print(f   {metric}: {value:.4f} )

print(f Majority class accuracy: {np.mean(y_imbalanced == 0):.4f} )
print( Notice the difference between overall accuracy and minority class
   performance! )

# CORRECT WAY 2 - Proper multi-class metrics
print( \n=== CORRECT WAY 2: Proper Multi-class Metrics === )

def evaluate_multiclass_model(y_true, y_pred, y_proba=None):
       Proper evaluation for multi-class datasets

    results = {
        'accuracy': accuracy_score(y_true, y_pred),
        'precision_macro': precision_score(y_true, y_pred, average='macro'
   ),
        'precision_weighted': precision_score(y_true, y_pred, average='
   weighted'),
        'recall_macro': recall_score(y_true, y_pred, average='macro'),
        'recall_weighted': recall_score(y_true, y_pred, average='weighted'
   ),
        'f1_macro': f1_score(y_true, y_pred, average='macro'),
        'f1_weighted': f1_score(y_true, y_pred, average='weighted')
    }

    # Per-class metrics
```

```
    precision_per_class = precision_score(y_true, y_pred, average=None)
    recall_per_class = recall_score(y_true, y_pred, average=None)

    for i in range(len(precision_per_class)):
        results[f'class_{i}_precision'] = precision_per_class[i]
        results[f'class_{i}_recall'] = recall_per_class[i]

    # ROC AUC for multi-class
    if y_proba is not None:
        results['auc_ovr'] = roc_auc_score(y_true, y_proba, multi_class='ovr')
        results['auc_ovo'] = roc_auc_score(y_true, y_proba, multi_class='ovo')

    return results

# Proper multi-class evaluation
y_pred_mc_correct = rf_multiclass.predict(X_multiclass)
y_proba_mc_correct = rf_multiclass.predict_proba(X_multiclass)

results_mc = evaluate_multiclass_model(
    y_multiclass, y_pred_mc_correct, y_proba_mc_correct
)

print( Proper multi-class evaluation: )
for metric, value in results_mc.items():
    if not metric.startswith('class_'):  # Skip per-class for brevity
        print(f   {metric}: {value:.4f} )

print( \nPer-class performance: )
for i in range(3):
    print(f   Class {i}: Precision={results_mc[f'class_{i}_precision']:.4f},
          f Recall={results_mc[f'class_{i}_recall']:.4f} )

# CORRECT WAY 3 - Comprehensive evaluation with proper ROC AUC
print( \n=== CORRECT WAY 3: Proper ROC AUC === )

# Correct way to calculate ROC AUC
def proper_auc_calculation(model, X, y):
       Proper ROC AUC calculation

    # Get probability predictions
    if hasattr(model, 'predict_proba'):
        y_proba = model.predict_proba(X)
        if y_proba.shape[1] == 2:  # Binary classification
            auc = roc_auc_score(y, y_proba[:, 1])
        else:  # Multi-class
            auc = roc_auc_score(y, y_proba, multi_class='ovr')
    else:
        # For models without predict_proba, use decision function
        if hasattr(model, 'decision_function'):
            y_scores = model.decision_function(X)
            auc = roc_auc_score(y, y_scores)
```

```
        else:
            raise ValueError( Model doesn't support probability or decision function )

    return auc

# Calculate AUC properly
auc_balanced = proper_auc_calculation(rf_balanced, X_balanced, y_balanced)
auc_imbalanced = proper_auc_calculation(rf_imbalanced, X_imbalanced, y_imbalanced)

print( Proper AUC calculation: )
print(f Balanced dataset AUC: {auc_balanced:.4f} )
print(f Imbalanced dataset AUC: {auc_imbalanced:.4f} )

# CORRECT WAY 4 - Multi-metric evaluation with context
print( \n=== CORRECT WAY 4: Multi-metric Evaluation === )

def comprehensive_model_evaluation(models, X_train, y_train, X_test, y_test):
    Comprehensive model evaluation with multiple metrics

    results = {}

    for name, model in models.items():
        # Train model
        model.fit(X_train, y_train)

        # Make predictions
        y_pred = model.predict(X_test)
        if hasattr(model, 'predict_proba'):
            y_proba = model.predict_proba(X_test)
        else:
            y_proba = None

        # Calculate multiple metrics
        metrics = {
            'accuracy': accuracy_score(y_test, y_pred),
            'precision_weighted': precision_score(y_test, y_pred, average='weighted'),
            'recall_weighted': recall_score(y_test, y_pred, average='weighted'),
            'f1_weighted': f1_score(y_test, y_pred, average='weighted'),
            'confusion_matrix': confusion_matrix(y_test, y_pred)
        }

        # Add AUC if probabilities available
        if y_proba is not None:
            if y_proba.shape[1] == 2:  # Binary
                metrics['auc'] = roc_auc_score(y_test, y_proba[:, 1])
            else:  # Multi-class
                metrics['auc'] = roc_auc_score(y_test, y_proba, multi_class='ovr')

```

```
        results[name] = metrics

    return results

# Compare models properly
X_train, X_test, y_train, y_test = train_test_split(
    X_balanced, y_balanced, test_size=0.2, random_state=42
)

models_to_compare = {
    'RandomForest': RandomForestClassifier(random_state=42),
    'LogisticRegression': LogisticRegression(random_state=42, max_iter
   =1000)
}

comprehensive_results = comprehensive_model_evaluation(
    models_to_compare, X_train, y_train, X_test, y_test
)

print( Comprehensive model comparison: )
for model_name, metrics in comprehensive_results.items():
    print(f \n{model_name}: )
    print(f   Accuracy: {metrics['accuracy']:.4f} )
    print(f   Precision: {metrics['precision_weighted']:.4f} )
    print(f   Recall: {metrics['recall_weighted']:.4f} )
    print(f   F1-Score: {metrics['f1_weighted']:.4f} )
    if 'auc' in metrics:
        print(f   AUC: {metrics['auc']:.4f} )
    print(f   Confusion Matrix:\n{metrics['confusion_matrix']} )

# Advanced evaluation techniques
class AdvancedEvaluator:
       Advanced model evaluation techniques

    @staticmethod
    def cost_sensitive_evaluation(y_true, y_pred, cost_matrix):
           Evaluate model with custom cost matrix
        cm = confusion_matrix(y_true, y_pred)
        total_cost = np.sum(cm * cost_matrix)
        return total_cost

    @staticmethod
    def bootstrap_confidence_intervals(y_true, y_pred, metric_func,
   n_bootstrap=1000, confidence=0.95):
           Calculate confidence intervals using bootstrap
        bootstrap_scores = []

        for _ in range(n_bootstrap):
            indices = np.random.choice(len(y_true), len(y_true), replace=
   True)
            score = metric_func(y_true[indices], y_pred[indices])
            bootstrap_scores.append(score)

        alpha = 1 - confidence
```

```
        lower_percentile = (alpha/2) * 100
        upper_percentile = (1 - alpha/2) * 100

        lower_bound = np.percentile(bootstrap_scores, lower_percentile)
        upper_bound = np.percentile(bootstrap_scores, upper_percentile)

        return {
            'mean': np.mean(bootstrap_scores),
            'std': np.std(bootstrap_scores),
            'confidence_interval': (lower_bound, upper_bound)
        }

    @staticmethod
    def evaluate_model_stability(model, X, y, n_bootstrap=100):
            Evaluate model stability using bootstrap
        scores = []

        for _ in range(n_bootstrap):
            indices = np.random.choice(len(X), len(X), replace=True)
            X_boot, y_boot = X[indices], y[indices]

            model.fit(X_boot, y_boot)
            score = model.score(X, y)
            scores.append(score)

        return {
            'mean_score': np.mean(scores),
            'std_score': np.std(scores),
            'stability': 1 - np.std(scores) / np.mean(scores)  # Higher is
    better
        }

# Example of advanced evaluation
print( \n=== ADVANCED: Bootstrap Confidence Intervals === )

# Calculate confidence intervals for accuracy
def accuracy_metric(y_true, y_pred):
    return accuracy_score(y_true, y_pred)

bootstrap_results = AdvancedEvaluator.bootstrap_confidence_intervals(
    y_test, rf_balanced.predict(X_test), accuracy_metric
)

print( Bootstrap confidence intervals for accuracy: )
print(f   Mean: {bootstrap_results['mean']:.4f} )
print(f   Std: {bootstrap_results['std']:.4f} )
print(f   95% CI: ({bootstrap_results['confidence_interval'][0]:.4f},
      f {bootstrap_results['confidence_interval'][1]:.4f}) )

# Example of cost-sensitive evaluation
print( \n=== ADVANCED: Cost-Sensitive Evaluation === )

# Define cost matrix (higher cost for false negatives in medical diagnosis
   )
```

```
# [TN, FP]
# [FN, TP]
cost_matrix = np.array([
    [0, 1],    # TN cost = 0, FP cost = 1
    [10, 0]    # FN cost = 10, TP cost = 0
])

cost_sensitive_score = AdvancedEvaluator.cost_sensitive_evaluation(
    y_test, rf_balanced.predict(X_test), cost_matrix
)

print(f Cost-sensitive evaluation score: {cost_sensitive_score} )

# Business impact metrics
def calculate_business_metrics(y_true, y_pred, value_per_tp=100,
   cost_per_fp=10):
       Calculate business impact metrics
    cm = confusion_matrix(y_true, y_pred)
    tn, fp, fn, tp = cm.ravel()

    revenue = tp * value_per_tp
    cost = fp * cost_per_fp
    net_benefit = revenue - cost

    return {
        'true_positives': tp,
        'false_positives': fp,
        'revenue': revenue,
        'cost': cost,
        'net_benefit': net_benefit
    }

business_metrics = calculate_business_metrics(y_test, rf_balanced.predict(
   X_test))
print( \nBusiness impact metrics: )
for metric, value in business_metrics.items():
    print(f   {metric}: {value} )
```

Listing 39: Error 5.2.4: Evaluation Metric Mistakes - Buggy Code

```
# Buggy: Common overfitting and underfitting issues
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, validation_curve,
   learning_curve
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.svm import SVR
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.metrics import mean_squared_error, accuracy_score
from sklearn.datasets import make_classification, make_regression
import matplotlib.pyplot as plt

# Create datasets to demonstrate overfitting/underfitting
np.random.seed(42)
```

```

# 1. Simple linear relationship (for underfitting demo)
X_simple, y_simple = make_regression(n_samples=100, n_features=1, noise
    =10, random_state=42)

# 2. Complex non-linear relationship (for overfitting demo)
X_complex = np.linspace(0, 2*np.pi, 100).reshape(-1, 1)
y_complex = np.sin(X_complex).ravel() + np.random.normal(0, 0.1, 100)

# 3. High-dimensional dataset (for overfitting demo)
X_high_dim, y_high_dim = make_classification(
    n_samples=100, n_features=50, n_informative=5,
    n_redundant=10, random_state=42
)

print( Dataset shapes: )
print(f Simple: {X_simple.shape} )
print(f Complex: {X_complex.shape} )
print(f High-dimensional: {X_high_dim.shape} )

# WRONG WAY 1 - Underfitting with too simple model
print( \n=== WRONG WAY 1: Underfitting with Too Simple Model === )

# WRONG: Using linear model for non-linear data
X_train_complex, X_test_complex, y_train_complex, y_test_complex =
    train_test_split(
    X_complex, y_complex, test_size=0.2, random_state=42
)

# Linear model on non-linear data (underfitting)
linear_model = LinearRegression()
linear_model.fit(X_train_complex, y_train_complex)

train_score_linear = linear_model.score(X_train_complex, y_train_complex)
test_score_linear = linear_model.score(X_test_complex, y_test_complex)

print( Linear model on non-linear data: )
print(f   Training score: {train_score_linear:.4f} )
print(f   Test score: {test_score_linear:.4f} )
print( Problem: High bias, underfitting - model too simple! )

# WRONG WAY 2 - Overfitting with too complex model
print( \n=== WRONG WAY 2: Overfitting with Too Complex Model === )

# WRONG: High-degree polynomial on small dataset
poly_features_overfit = PolynomialFeatures(degree=15)  # Too high!
X_poly_overfit = poly_features_overfit.fit_transform(X_train_complex)

# Complex model on small dataset (overfitting)
complex_model = LinearRegression()
complex_model.fit(X_poly_overfit, y_train_complex)

# Training score will be very high
train_pred_overfit = complex_model.predict(X_poly_overfit)
```

```
train_mse_overfit = mean_squared_error(y_train_complex, train_pred_overfit)

# Test score will be poor
X_test_poly_overfit = poly_features_overfit.transform(X_test_complex)
test_pred_overfit = complex_model.predict(X_test_poly_overfit)
test_mse_overfit = mean_squared_error(y_test_complex, test_pred_overfit)

print( High-degree polynomial on small dataset: )
print(f   Training MSE: {train_mse_overfit:.4f} )
print(f   Test MSE: {test_mse_overfit:.4f} )
print( Problem: High variance, overfitting - model too complex! )

# WRONG WAY 3 - No regularization on high-dimensional data
print( \n=== WRONG WAY 3: No Regularization on High-dimensional Data === )

X_train_hd, X_test_hd, y_train_hd, y_test_hd = train_test_split(
    X_high_dim, y_high_dim, test_size=0.2, random_state=42
)

# WRONG: No regularization on high-dimensional data
rf_no_reg = RandomForestClassifier(random_state=42, min_samples_split=2, min_samples_leaf=1)
rf_no_reg.fit(X_train_hd, y_train_hd)

train_score_no_reg = rf_no_reg.score(X_train_hd, y_train_hd)
test_score_no_reg = rf_no_reg.score(X_test_hd, y_test_hd)

print( Random Forest without regularization on high-dimensional data: )
print(f   Training accuracy: {train_score_no_reg:.4f} )
print(f   Test accuracy: {test_score_no_reg:.4f} )
print( Problem: Overfitting due to high variance in high-dimensional space! )

# WRONG WAY 4 - Ignoring learning curves
print( \n=== WRONG WAY 4: Ignoring Learning Curves === )

# WRONG: Not checking if more data would help
def quick_model_evaluation(X, y):
        Quick evaluation without learning curve analysis
    model = RandomForestClassifier(random_state=42)
    model.fit(X, y)
    return model.score(X, y)  # WRONG: Only training score

quick_score = quick_model_evaluation(X_high_dim, y_high_dim)
print(f Quick evaluation score: {quick_score:.4f} )
print( Problem: No insight into bias-variance tradeoff! )

# CORRECT WAY 1 - Diagnosing bias-variance with validation curves
print( \n=== CORRECT WAY 1: Validation Curves for Diagnosis === )

def plot_validation_curve_analysis(X, y, model, param_name, param_range, cv=5):
        Plot validation curves to diagnose bias-variance
```

```
    train_scores, val_scores = validation_curve(
        model, X, y, param_name=param_name, param_range=param_range, cv=cv
    )

    train_mean = np.mean(train_scores, axis=1)
    train_std = np.std(train_scores, axis=1)
    val_mean = np.mean(val_scores, axis=1)
    val_std = np.std(val_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(param_range, train_mean, 'o-', color='blue', label='Training score')
    plt.fill_between(param_range, train_mean - train_std, train_mean + train_std, alpha=0.1, color='blue')

    plt.plot(param_range, val_mean, 'o-', color='red', label='Cross-validation score')
    plt.fill_between(param_range, val_mean - val_std, val_mean + val_std, alpha=0.1, color='red')

    plt.xlabel(param_name)
    plt.ylabel('Score')
    plt.title(f'Validation Curve for {param_name}')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Diagnosis
    final_train = train_mean[-1]
    final_val = val_mean[-1]

    if final_train < 0.8:
        print( Diagnosis: High bias (underfitting) - Model is too simple )
    elif final_train - final_val > 0.1:
        print( Diagnosis: High variance (overfitting) - Model is too complex )
    else:
        print( Diagnosis: Good fit - Balanced bias and variance )

    return final_train, final_val

# Analyze polynomial degree complexity
print( Analyzing polynomial degree complexity: )
poly_model = LinearRegression()
degrees = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
train_scores_poly, val_scores_poly = [], []

for degree in degrees:
    poly_features = PolynomialFeatures(degree=degree)
    X_poly = poly_features.fit_transform(X_train_complex)

    # Cross-validation score
    from sklearn.model_selection import cross_val_score
```

```
    cv_scores = cross_val_score(LinearRegression(), X_poly,
    y_train_complex, cv=5)
    val_scores_poly.append(cv_scores.mean())

    # Training score
    model = LinearRegression()
    model.fit(X_poly, y_train_complex)
    train_scores_poly.append(model.score(X_poly, y_train_complex))

# Plot results
plt.figure(figsize=(10, 6))
plt.plot(degrees, train_scores_poly, 'o-', label='Training Score')
plt.plot(degrees, val_scores_poly, 'o-', label='Validation Score')
plt.xlabel('Polynomial Degree')
plt.ylabel('Score')
plt.title('Polynomial Degree vs Model Performance')
plt.legend()
plt.grid(True)
plt.show()

# CORRECT WAY 2 - Proper regularization
print( \n=== CORRECT WAY 2: Proper Regularization === )

# Ridge regression with proper regularization
def find_optimal_regularization(X, y, alpha_range=np.logspace(-4, 4, 50)):
       Find optimal regularization strength

    train_scores = []
    val_scores = []

    for alpha in alpha_range:
        ridge = Ridge(alpha=alpha)

        # Cross-validation
        cv_scores = cross_val_score(ridge, X, y, cv=5)
        val_scores.append(cv_scores.mean())

        # Training score
        ridge.fit(X, y)
        train_scores.append(ridge.score(X, y))

    optimal_idx = np.argmax(val_scores)
    optimal_alpha = alpha_range[optimal_idx]

    return optimal_alpha, train_scores, val_scores, alpha_range

# Find optimal regularization for high-dimensional data
optimal_alpha, train_reg, val_reg, alphas = find_optimal_regularization(
    X_train_hd, y_train_hd)

print(f Optimal regularization strength: {optimal_alpha:.6f} )

# Compare with no regularization
ridge_optimal = Ridge(alpha=optimal_alpha)
```

```
ridge_optimal.fit(X_train_hd, y_train_hd)

rf_regularized = RandomForestClassifier(
    random_state=42,
    min_samples_split=10,  # Increase minimum samples
    min_samples_leaf=5,    # Increase minimum leaf samples
    max_depth=10           # Limit depth
)
rf_regularized.fit(X_train_hd, y_train_hd)

print( Regularized models performance: )
print(f   Ridge (  ={optimal_alpha:.4f}) - Train: {ridge_optimal.score(X_train_hd, y_train_hd):.4f},
      f Test: {ridge_optimal.score(X_test_hd, y_test_hd):.4f} )

print(f   Random Forest (regularized) - Train: {rf_regularized.score(X_train_hd, y_train_hd):.4f},
      f Test: {rf_regularized.score(X_test_hd, y_test_hd):.4f} )

# CORRECT WAY 3 - Learning curve analysis
print( \n=== CORRECT WAY 3: Learning Curve Analysis === )

def plot_learning_curve_analysis(estimator, X, y, cv=5, train_sizes=np.linspace(0.1, 1.0, 10)):
       Plot learning curves to understand bias-variance tradeoff

    train_sizes, train_scores, val_scores = learning_curve(
        estimator, X, y, cv=cv, train_sizes=train_sizes, n_jobs=-1
    )

    train_mean = np.mean(train_scores, axis=1)
    train_std = np.std(train_scores, axis=1)
    val_mean = np.mean(val_scores, axis=1)
    val_std = np.std(val_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_mean, 'o-', color='blue', label='Training score')
    plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.1, color='blue')

    plt.plot(train_sizes, val_mean, 'o-', color='red', label='Cross-validation score')
    plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std, alpha=0.1, color='red')

    plt.xlabel('Training Set Size')
    plt.ylabel('Score')
    plt.title('Learning Curve')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Diagnosis based on learning curves
```

```
    final_train = train_mean[-1]
    final_val = val_mean[-1]
    gap = final_train - final_val

    print( Learning Curve Analysis: )
    print(f   Final training score: {final_train:.4f} )
    print(f   Final validation score: {final_val:.4f} )
    print(f   Gap: {gap:.4f} )

    if final_train < 0.8:
        print(       High bias problem (underfitting) )
        print(       Consider more complex model or feature engineering )
    elif gap > 0.1:
        print(       High variance problem (overfitting) )
        print(       Consider more data, regularization, or simpler model
   )
    else:
        print(       Good bias-variance balance )

    # Data sufficiency analysis
    if final_val < 0.8 and val_mean[-1] - val_mean[-3] > 0.01:
        print(       More data might improve performance )
    elif final_val < 0.8:
        print(       More data unlikely to help significantly )

    return train_sizes, train_mean, val_mean

# Analyze learning curves for different models
print( Learning curve analysis for Random Forest: )
plot_learning_curve_analysis(
    RandomForestClassifier(random_state=42, n_estimators=100),
    X_high_dim, y_high_dim
)

# CORRECT WAY 4 - Early stopping and model complexity control
print( \n=== CORRECT WAY 4: Early Stopping and Complexity Control === )

class EarlyStoppingModel:
       Model with early stopping capability

    def __init__(self, base_model, patience=10, min_delta=1e-4):
        self.base_model = base_model
        self.patience = patience
        self.min_delta = min_delta
        self.best_score = None
        self.wait = 0
        self.stopped_epoch = 0

    def fit_with_early_stopping(self, X, y, X_val, y_val, max_epochs=1000)
   :
           Fit model with early stopping

        scores = []

```

```
        for epoch in range(max_epochs):
            # Fit model for this epoch (simplified for demonstration)
            self.base_model.fit(X, y)

            # Evaluate on validation set
            score = self.base_model.score(X_val, y_val)
            scores.append(score)

            # Early stopping logic
            if self.best_score is None:
                self.best_score = score
            elif score - self.best_score < self.min_delta:
                self.wait += 1
                if self.wait >= self.patience:
                    self.stopped_epoch = epoch
                    print(f Early stopping at epoch {epoch} )
                    break
            else:
                self.best_score = score
                self.wait = 0

        return scores

# Demonstrate early stopping concept
from sklearn.ensemble import GradientBoostingClassifier

# Split data for validation
X_train_es, X_val_es, y_train_es, y_val_es = train_test_split(
    X_high_dim, y_high_dim, test_size=0.2, random_state=42
)

# Model with potential overfitting
gb_model = GradientBoostingClassifier(
    n_estimators=1000,  # Many estimators
    learning_rate=0.1,
    max_depth=3,
    random_state=42
)

# Early stopping approach
gb_model_early = GradientBoostingClassifier(
    n_estimators=1000,
    learning_rate=0.1,
    max_depth=3,
    validation_fraction=0.1,  # Use 10% for validation
    n_iter_no_change=10,      # Patience
    tol=1e-4,                 # Minimum improvement
    random_state=42
)

# Fit both models
gb_model.fit(X_train_es, y_train_es)
gb_model_early.fit(X_train_es, y_train_es)

```

```
print( Early stopping comparison: )
print(f   Regular GB - Train: {gb_model.score(X_train_es, y_train_es):.4f},
      f Test: {gb_model.score(X_val_es, y_val_es):.4f} )

print(f   Early stopping GB - Train: {gb_model_early.score(X_train_es, y_train_es):.4f},
      f Test: {gb_model_early.score(X_val_es, y_val_es):.4f} )

print(f   Regular GB estimators used: {gb_model.n_estimators} )
print(f   Early stopping GB estimators used: {gb_model_early.n_estimators_} )

# Advanced bias-variance analysis
class BiasVarianceAnalyzer:
       Advanced bias-variance decomposition analysis

    @staticmethod
    def bias_variance_decomposition(model, X, y, n_bootstrap=100):
           Decompose error into bias, variance, and noise components

        predictions = []

        # Bootstrap sampling
        for _ in range(n_bootstrap):
            indices = np.random.choice(len(X), len(X), replace=True)
            X_boot, y_boot = X[indices], y[indices]

            model.fit(X_boot, y_boot)
            pred = model.predict(X)
            predictions.append(pred)

        predictions = np.array(predictions)

        # Calculate components
        avg_prediction = np.mean(predictions, axis=0)
        bias_squared = np.mean((avg_prediction - y) ** 2)
        variance = np.mean(np.var(predictions, axis=0))
        noise = np.mean(np.var(predictions, axis=0))  # Simplified

        return {
            'bias_squared': bias_squared,
            'variance': variance,
            'noise': noise,
            'total_error': bias_squared + variance + noise
        }

    @staticmethod
    def model_complexity_analysis(X, y, model_class, complexity_param, param_values, cv=5):
           Analyze how model complexity affects bias-variance tradeoff

        results = {
            'param_values': param_values,
```

```
            'bias_squared': [],
            'variance': [],
            'total_error': []
        }

        for param_value in param_values:
            model = model_class(**{complexity_param: param_value})

            # Cross-validation for variance estimation
            from sklearn.model_selection import cross_val_predict
            cv_predictions = cross_val_predict(model, X, y, cv=cv)

            # Bias calculation
            model.fit(X, y)
            train_predictions = model.predict(X)

            bias_sq = np.mean((np.mean(cv_predictions) - y) ** 2)
            var = np.var(cv_predictions)

            results['bias_squared'].append(bias_sq)
            results['variance'].append(var)
            results['total_error'].append(bias_sq + var)

        return results

# Example of bias-variance analysis
print( \n=== ADVANCED: Bias-Variance Analysis === )

# This would perform detailed bias-variance decomposition
# bias_var_results = BiasVarianceAnalyzer.bias_variance_decomposition(
#     RandomForestClassifier(random_state=42), X_high_dim, y_high_dim
# )

print( Bias-variance analysis helps understand the sources of prediction
    error! )

# Practical overfitting prevention strategies
def prevent_overfitting_checklist():
       Checklist for preventing overfitting

    checklist = [
         1. Use cross-validation for model evaluation ,
         2. Split data properly (train/validation/test) ,
         3. Apply appropriate regularization ,
         4. Use early stopping for iterative algorithms ,
         5. Limit model complexity (depth, features, etc.) ,
         6. Increase minimum samples for splits/leaves ,
         7. Use ensemble methods to reduce variance ,
         8. Analyze learning curves to understand data needs ,
         9. Check validation curves for optimal hyperparameters ,
         10. Monitor training vs validation performance gap
    ]

    print( Overfitting Prevention Checklist: )
```

```
    for item in checklist:
        print(f   {item} )

prevent_overfitting_checklist()

# Model selection based on bias-variance tradeoff
def select_model_by_bias_variance(X, y, models_dict):
       Select model based on bias-variance analysis

    results = {}

    for name, model in models_dict.items():
        # Cross-validation scores
        cv_scores = cross_val_score(model, X, y, cv=5)

        # Training score
        model.fit(X, y)
        train_score = model.score(X, y)

        # Bias-variance indicators
        bias_indicator = 1 - train_score  # Higher = more bias
        variance_indicator = train_score - cv_scores.mean()  # Higher =
   more variance

        results[name] = {
            'train_score': train_score,
            'cv_mean': cv_scores.mean(),
            'cv_std': cv_scores.std(),
            'bias_indicator': bias_indicator,
            'variance_indicator': variance_indicator
        }

    return results

# Example model selection
models_for_analysis = {
    'Simple Model': LogisticRegression(random_state=42),
    'Moderate Model': RandomForestClassifier(n_estimators=100,
   random_state=42),
    'Complex Model': RandomForestClassifier(n_estimators=500, max_depth
   =20, random_state=42)
}

analysis_results = select_model_by_bias_variance(X_train_hd, y_train_hd,
   models_for_analysis)

print( \nModel Selection Based on Bias-Variance Tradeoff: )
for name, metrics in analysis_results.items():
    print(f \n{name}: )
    print(f   Train Score: {metrics['train_score']:.4f} )
    print(f   CV Score: {metrics['cv_mean']:.4f} (+/- {metrics['cv_std'] *
    2:.4f}) )
    print(f   Bias Indicator: {metrics['bias_indicator']:.4f} )
```

```
    print(f   Variance Indicator: {metrics['variance_indicator']:.4f} )
```

Listing 40: Error 5.2.5: Overfitting and Underfitting Issues - Buggy Code

5.3 5.3 Deployment and Production Errors

```
# Buggy: Common model versioning issues
import joblib
import pickle
import os
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification

# Create sample data and model
X, y = make_classification(n_samples=1000, n_features=10, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# Train a model
model_v1 = RandomForestClassifier(n_estimators=100, random_state=42)
model_v1.fit(X_train, y_train)

print( Model V1 trained )
print(f Model V1 accuracy: {model_v1.score(X_test, y_test):.4f} )

# WRONG WAY 1 - No versioning at all
print( \n=== WRONG WAY 1: No Versioning === )

# WRONG: Overwriting model file without version control
joblib.dump(model_v1, 'model.pkl')  # No version information
print( Model saved without versioning )

# Train improved model
model_v2 = RandomForestClassifier(n_estimators=200, random_state=42)  #
    Better parameters
model_v2.fit(X_train, y_train)

# WRONG: Overwrite previous model
joblib.dump(model_v2, 'model.pkl')  # Previous model lost!
print( Model V2 saved, V1 overwritten )

# Problem: No way to rollback or compare versions
print( Problem: Previous model version lost forever! )

# WRONG WAY 2 - Inconsistent naming
print( \n=== WRONG WAY 2: Inconsistent Naming === )

# WRONG: Inconsistent version naming
joblib.dump(model_v1, 'my_model_v1.pkl')
joblib.dump(model_v2, 'model_version_2.pkl')  # Different naming
    convention
```

```
joblib.dump(model_v1, 'final_model.pkl')      # No version info

print( Models saved with inconsistent naming )
print( Problem: Hard to track and manage versions! )

# WRONG WAY 3 - No metadata tracking
print( \n=== WRONG WAY 3: No Metadata Tracking === )

# WRONG: Only saving model, no metadata
joblib.dump(model_v2, 'model_v2.pkl')

# Later, when loading...
loaded_model = joblib.load('model_v2.pkl')
print( Model loaded, but no information about: )
print(   - Training parameters )
print(   - Performance metrics )
print(   - Training data characteristics )
print(   - Creation date )
print( Problem: No context about the model! )

# WRONG WAY 4 - Manual version management
print( \n=== WRONG WAY 4: Manual Version Management === )

# WRONG: Manual file management
def save_model_manual(model, version):
       Manual model saving - error prone
    filename = f'model_v{version}.pkl'
    joblib.dump(model, filename)
    print(f Model saved as {filename} )

save_model_manual(model_v1, 1)
save_model_manual(model_v2, 2)

# Manual rollback - error prone
def load_model_manual(version):
       Manual model loading - error prone
    filename = f'model_v{version}.pkl'
    if os.path.exists(filename):
        return joblib.load(filename)
    else:
        print(f Model version {version} not found )
        return None

# This approach is fragile and error-prone
print( Problem: Manual management is error-prone and inconsistent! )

# CORRECT WAY 1 - Systematic versioning
print( \n=== CORRECT WAY 1: Systematic Versioning === )

import datetime
import json

class ModelVersionManager:
       Systematic model version management
```

```
99
100     def __init__(self, model_dir='models'):
101         self.model_dir = model_dir
102         os.makedirs(model_dir, exist_ok=True)
103         self.versions_file = os.path.join(model_dir, 'versions.json')
104 
105         # Initialize versions file if it doesn't exist
106         if not os.path.exists(self.versions_file):
107             with open(self.versions_file, 'w') as f:
108                 json.dump([], f)
109 
110     def save_model(self, model, version_name, metadata=None):
111            Save model with version tracking
112 
113         # Create version identifier
114         timestamp = datetime.datetime.now().strftime('%Y%m%d_%H%M%S')
115         version_id = f {version_name}_{timestamp}
116 
117         # Save model
118         model_filename = f {version_id}.pkl
119         model_path = os.path.join(self.model_dir, model_filename)
120         joblib.dump(model, model_path)
121 
122         # Create metadata
123         version_info = {
124             'version_id': version_id,
125             'version_name': version_name,
126             'timestamp': timestamp,
127             'model_path': model_filename,
128             'metadata': metadata or {}
129         }
130 
131         # Update versions file
132         with open(self.versions_file, 'r') as f:
133             versions = json.load(f)
134 
135         versions.append(version_info)
136 
137         with open(self.versions_file, 'w') as f:
138             json.dump(versions, f, indent=2)
139 
140         print(f Model saved: {version_id} )
141         return version_id
142 
143     def load_model(self, version_id=None, version_name=None):
144            Load specific model version
145 
146         with open(self.versions_file, 'r') as f:
147             versions = json.load(f)
148 
149         # Find version
150         if version_id:
151             version_info = next((v for v in versions if v['version_id'] ==
    version_id), None)
```

```
        elif version_name:
            # Get latest version with this name
            matching_versions = [v for v in versions if v['version_name'] == version_name]
            version_info = max(matching_versions, key=lambda x: x['timestamp']) if matching_versions else None
        else:
            # Get latest version
            version_info = max(versions, key=lambda x: x['timestamp']) if versions else None

        if not version_info:
            raise ValueError( Model version not found )

        model_path = os.path.join(self.model_dir, version_info['model_path'])
        model = joblib.load(model_path)

        return model, version_info

    def list_versions(self):
           List all model versions
        with open(self.versions_file, 'r') as f:
            versions = json.load(f)
        return versions

# Use proper version management
version_manager = ModelVersionManager()

# Save models with proper versioning
metadata_v1 = {
    'n_estimators': 100,
    'accuracy': model_v1.score(X_test, y_test),
    'training_samples': len(X_train)
}

metadata_v2 = {
    'n_estimators': 200,
    'accuracy': model_v2.score(X_test, y_test),
    'training_samples': len(X_train)
}

version_id_1 = version_manager.save_model(model_v1, 'random_forest', metadata_v1)
version_id_2 = version_manager.save_model(model_v2, 'random_forest', metadata_v2)

print(f Saved versions: {version_id_1}, {version_id_2} )

# List all versions
print( \nAll model versions: )
versions = version_manager.list_versions()
for version in versions:
    print(f   {version['version_id']}: {version['metadata']} )
```

```
200
201 # Load specific version
202 loaded_model_v1, info_v1 = version_manager.load_model(version_id=
    version_id_1)
203 print(f \nLoaded model {info_v1['version_id']} )
204 print(f Accuracy: {loaded_model_v1.score(X_test, y_test):.4f} )
205
206 # CORRECT WAY 2 - Semantic versioning
207 print( \n=== CORRECT WAY 2: Semantic Versioning === )
208
209 class SemanticModelVersionManager:
210         Model version management with semantic versioning
211
212     def __init__(self, model_dir='models_semantic'):
213         self.model_dir = model_dir
214         os.makedirs(model_dir, exist_ok=True)
215         self.versions_file = os.path.join(model_dir, 'semantic_versions.
    json')
216
217         if not os.path.exists(self.versions_file):
218             # Initialize with version 1.0.0
219             initial_info = {
220                 'latest_major': 1,
221                 'latest_minor': 0,
222                 'latest_patch': 0,
223                 'versions': []
224             }
225             with open(self.versions_file, 'w') as f:
226                 json.dump(initial_info, f)
227
228     def _get_next_version(self, version_type='patch'):
229            Get next version number
230         with open(self.versions_file, 'r') as f:
231             version_info = json.load(f)
232
233         major = version_info['latest_major']
234         minor = version_info['latest_minor']
235         patch = version_info['latest_patch']
236
237         if version_type == 'major':
238             major += 1
239             minor = 0
240             patch = 0
241         elif version_type == 'minor':
242             minor += 1
243             patch = 0
244         elif version_type == 'patch':
245             patch += 1
246
247         version_info['latest_major'] = major
248         version_info['latest_minor'] = minor
249         version_info['latest_patch'] = patch
250
251         return f {major}.{minor}.{patch} , version_info
```

```
252
253     def save_model_semantic(self, model, change_type='patch', description=
    ''):
254             Save model with semantic versioning
255
256         version, version_info = self._get_next_version(change_type)
257         timestamp = datetime.datetime.now().strftime('%Y%m%d_%H%M%S')
258         version_id = f v{version}_{timestamp}
259
260         # Save model
261         model_filename = f {version_id}.pkl
262         model_path = os.path.join(self.model_dir, model_filename)
263         joblib.dump(model, model_path)
264
265         # Create version record
266         version_record = {
267             'version': version,
268             'version_id': version_id,
269             'timestamp': timestamp,
270             'model_path': model_filename,
271             'change_type': change_type,
272             'description': description
273         }
274
275         version_info['versions'].append(version_record)
276
277         with open(self.versions_file, 'w') as f:
278             json.dump(version_info, f, indent=2)
279
280         print(f Model saved with semantic version: {version} )
281         return version_id, version
282
283     def get_version_history(self):
284            Get version history
285         with open(self.versions_file, 'r') as f:
286             version_info = json.load(f)
287         return version_info['versions']
288
289 # Use semantic versioning
290 semantic_manager = SemanticModelVersionManager()
291
292 # Save models with semantic versioning
293 version_id_1, sem_version_1 = semantic_manager.save_model_semantic(
294     model_v1, 'major', 'Initial release with basic Random Forest'
295 )
296
297 version_id_2, sem_version_2 = semantic_manager.save_model_semantic(
298     model_v2, 'minor', 'Improved with more estimators'
299 )
300
301 print(f Semantic versions: {sem_version_1}, {sem_version_2} )
302
303 # View version history
304 print( \nSemantic version history: )
```

```
history = semantic_manager.get_version_history()
for record in history:
    print(f   v{record['version']} ({record['change_type']}): {record['
   description']} )

# CORRECT WAY 3 - Model registry with comprehensive metadata
print( \n=== CORRECT WAY 3: Comprehensive Model Registry === )

class ModelRegistry:
       Comprehensive model registry with metadata

    def __init__(self, registry_dir='model_registry'):
        self.registry_dir = registry_dir
        os.makedirs(registry_dir, exist_ok=True)
        self.registry_file = os.path.join(registry_dir, 'registry.json')

        if not os.path.exists(self.registry_file):
            with open(self.registry_file, 'w') as f:
                json.dump({'models': {}}, f)

    def register_model(self, model, model_name, version, metrics=None,
                       training_data_info=None, hyperparameters=None,
                       feature_names=None, target_name=None):
           Register model with comprehensive metadata

        timestamp = datetime.datetime.now().strftime('%Y%m%d_%H%M%S')
        model_id = f {model_name}_v{version}_{timestamp}

        # Save model
        model_filename = f {model_id}.pkl
        model_path = os.path.join(self.registry_dir, model_filename)
        joblib.dump(model, model_path)

        # Create comprehensive metadata
        model_info = {
            'model_id': model_id,
            'model_name': model_name,
            'version': version,
            'timestamp': timestamp,
            'model_path': model_filename,
            'model_type': type(model).__name__,
            'hyperparameters': hyperparameters or {},
            'metrics': metrics or {},
            'training_data_info': training_data_info or {},
            'feature_names': feature_names or [],
            'target_name': target_name,
            'python_version': '.'.join(map(str, [x for x in __import__('
   sys').version_info[:3]])),
            'scikit_learn_version': __import__('sklearn').__version__
        }

        # Update registry
        with open(self.registry_file, 'r') as f:
            registry = json.load(f)
```

```
        if model_name not in registry['models']:
            registry['models'][model_name] = []

        registry['models'][model_name].append(model_info)

        with open(self.registry_file, 'w') as f:
            json.dump(registry, f, indent=2)

        print(f Model registered: {model_id} )
        return model_id

    def get_model(self, model_name, version=None):
           Retrieve model by name and optional version
        with open(self.registry_file, 'r') as f:
            registry = json.load(f)

        if model_name not in registry['models']:
            raise ValueError(f Model {model_name} not found )

        model_versions = registry['models'][model_name]

        if version:
            model_info = next((m for m in model_versions if m['version']
   == version), None)
        else:
            # Get latest version
            model_info = max(model_versions, key=lambda x: x['timestamp'])

        if not model_info:
            raise ValueError(f Model version not found )

        model_path = os.path.join(self.registry_dir, model_info['
   model_path'])
        model = joblib.load(model_path)

        return model, model_info

    def compare_models(self, model_name):
           Compare different versions of a model
        with open(self.registry_file, 'r') as f:
            registry = json.load(f)

        if model_name not in registry['models']:
            return None

        versions = registry['models'][model_name]
        return sorted(versions, key=lambda x: x['timestamp'])

# Use comprehensive model registry
registry = ModelRegistry()

# Register models with full metadata
model_v1_id = registry.register_model(
```

```
    model_v1,
    model_name='customer_churn_rf',
    version='1.0.0',
    metrics={'accuracy': model_v1.score(X_test, y_test)},
    training_data_info={'samples': len(X_train), 'features': X_train.shape[1]},
    hyperparameters={'n_estimators': 100, 'random_state': 42},
    feature_names=[f'feature_{i}' for i in range(X.shape[1])],
    target_name='churn'
)

model_v2_id = registry.register_model(
    model_v2,
    model_name='customer_churn_rf',
    version='1.1.0',
    metrics={'accuracy': model_v2.score(X_test, y_test)},
    training_data_info={'samples': len(X_train), 'features': X_train.shape[1]},
    hyperparameters={'n_estimators': 200, 'random_state': 42},
    feature_names=[f'feature_{i}' for i in range(X.shape[1])],
    target_name='churn'
)

print(f Registered models: {model_v1_id}, {model_v2_id} )

# Compare model versions
print( \nModel comparison: )
comparison = registry.compare_models('customer_churn_rf')
for model_info in comparison:
    print(f   v{model_info['version']}: Accuracy={model_info['metrics'].get('accuracy', 'N/A')} )

# Retrieve specific model version
retrieved_model, model_info = registry.get_model('customer_churn_rf', '1.0.0')
print(f \nRetrieved model: {model_info['model_id']} )
print(f Model type: {model_info['model_type']} )
print(f Hyperparameters: {model_info['hyperparameters']} )

# Advanced versioning with Git integration
class GitModelVersioning:
       Model versioning with Git integration

    def __init__(self, repo_path='.'):
        try:
            import git
            self.repo = git.Repo(repo_path)
        except ImportError:
            print( GitPython not installed. Git integration disabled. )
            self.repo = None
        except Exception as e:
            print(f Git repository not found: {e} )
            self.repo = None

```

```
    def get_git_info(self):
        """Get current Git information"""
        if not self.repo:
            return {}

        try:
            return {
                'commit_hash': self.repo.head.object.hexsha,
                'commit_message': self.repo.head.object.message.strip(),
                'branch': self.repo.active_branch.name,
                'timestamp': datetime.datetime.now().isoformat()
            }
        except Exception as e:
            print(f"Error getting Git info: {e}")
            return {}

    def save_model_with_git(self, model, model_name, description=''):
        """Save model with Git integration"""

        git_info = self.get_git_info()
        timestamp = datetime.datetime.now().strftime('%Y%m%d_%H%M%S')
        model_id = f"{model_name}_{timestamp}"

        # Save model
        model_filename = f"{model_id}.pkl"
        joblib.dump(model, model_filename)

        # Create metadata with Git info
        metadata = {
            'model_id': model_id,
            'timestamp': timestamp,
            'model_name': model_name,
            'description': description,
            'git_info': git_info
        }

        metadata_filename = f"{model_id}_metadata.json"
        with open(metadata_filename, 'w') as f:
            json.dump(metadata, f, indent=2)

        print(f"Model saved with Git integration: {model_id}")
        return model_id

# Example of Git-integrated versioning
print("\n=== ADVANCED: Git-Integrated Versioning ===")

# This would work if GitPython is installed and in a Git repository
# git_versioning = GitModelVersioning()
# git_model_id = git_versioning.save_model_with_git(
#     model_v1, 'git_tracked_model', 'Model saved with Git integration'
# )

```

```
print( Git-integrated versioning provides commit-level tracking! )
```

Listing 41: Error 5.3.1: Model Versioning Issues - Buggy Code

```
# Buggy: Common data drift and model monitoring issues
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt
from datetime import datetime, timedelta

# Create initial training data
np.random.seed(42)
X_train_orig, y_train_orig = make_classification(
    n_samples=1000, n_features=10, n_classes=2, random_state=42
)

# Train initial model
model = RandomForestClassifier(random_state=42)
model.fit(X_train_orig, y_train_orig)

print( Initial model trained )
print(f Training accuracy: {model.score(X_train_orig, y_train_orig):.4f} )

# Simulate production data over time
def generate_production_data(base_date, n_samples=200, drift_factor=0):
       Generate production data with potential drift
    # Add some drift to feature distributions
    X = np.random.randn(n_samples, 10)
    X[:, 0] += drift_factor * 0.5  # Drift in first feature
    X[:, 1] += drift_factor * 0.3  # Drift in second feature

    # Change relationship between features and target (concept drift)
    if drift_factor > 2:
        y = (X[:, 0] + X[:, 2] > 0).astype(int)  # Different relationship
    else:
        y = (X[:, 0] + X[:, 1] > 0).astype(int)  # Original relationship

    dates = [base_date + timedelta(days=i) for i in range(n_samples)]
    return X, y, dates

# WRONG WAY 1 - No monitoring at all
print( \n=== WRONG WAY 1: No Monitoring === )

# WRONG: Deploy model and ignore monitoring
production_dates = [datetime(2024, 1, 1) + timedelta(days=i) for i in
    range(30)]
X_prod, y_prod, dates_prod = generate_production_data(datetime(2024, 1, 1)
    , 300, drift_factor=3)

# Make predictions without monitoring
predictions = model.predict(X_prod)
```

```
accuracy = accuracy_score(y_prod, predictions)

print(f Production accuracy: {accuracy:.4f} )
print( Problem: No detection of performance degradation due to data drift! )
print( The model performance has likely degraded due to concept drift. )

# WRONG WAY 2 - Manual monitoring with static thresholds
print( \n=== WRONG WAY 2: Static Thresholds === )

# WRONG: Using fixed thresholds that don't adapt
def manual_monitoring(predictions, actuals, threshold=0.8):
       Manual monitoring with static threshold
    current_accuracy = accuracy_score(actuals, predictions)

    if current_accuracy < threshold:
        print(f ALERT: Accuracy {current_accuracy:.4f} below threshold {threshold} )
        return True
    else:
        print(f Normal: Accuracy {current_accuracy:.4f} above threshold )
        return False

# This approach fails when data distribution changes
alert_triggered = manual_monitoring(predictions, y_prod, threshold=0.8)
print(f Alert triggered: {alert_triggered} )
print( Problem: Static thresholds don't account for changing data distributions! )

# WRONG WAY 3 - No concept drift detection
print( \n=== WRONG WAY 3: No Concept Drift Detection === )

# WRONG: Only monitoring input features, not target relationships
def simple_drift_detection(X_current, X_reference):
       Simple drift detection - only looks at input features
    # Compare means
    current_means = np.mean(X_current, axis=0)
    reference_means = np.mean(X_reference, axis=0)

    drift_scores = np.abs(current_means - reference_means)
    total_drift = np.mean(drift_scores)

    return total_drift

# This misses concept drift (changes in P(Y|X))
drift_score = simple_drift_detection(X_prod, X_train_orig)
print(f Feature drift score: {drift_score:.4f} )
print( Problem: Feature drift detected, but concept drift (P(Y|X)) not monitored! )

# WRONG WAY 4 - No statistical significance testing
print( \n=== WRONG WAY 4: No Statistical Testing === )

# WRONG: Not using statistical tests for significant changes
```

```
def naive_performance_comparison(old_perf, new_perf):
       Naive performance comparison without statistical testing
    if new_perf < old_perf:
        print( Performance decreased! )
        return True
    else:
        print( Performance stable )
        return False

# This can lead to false alarms due to random variation
old_performance = 0.85
new_performance = 0.82
decrease_detected = naive_performance_comparison(old_performance,
   new_performance)
print(f Performance decrease detected: {decrease_detected} )
print( Problem: No statistical significance testing for performance
   changes! )

# CORRECT WAY 1 - Statistical drift detection
print( \n=== CORRECT WAY 1: Statistical Drift Detection === )

from scipy import stats

class StatisticalDriftDetector:
       Statistical approach to drift detection

    def __init__(self, reference_data, alpha=0.05):
        self.reference_data = reference_data
        self.alpha = alpha
        self.reference_stats = self._calculate_reference_stats()

    def _calculate_reference_stats(self):
           Calculate reference statistics
        return {
            'means': np.mean(self.reference_data, axis=0),
            'stds': np.std(self.reference_data, axis=0),
            'medians': np.median(self.reference_data, axis=0)
        }

    def detect_feature_drift(self, current_data):
           Detect drift in input features using statistical tests
        drift_results = {}

        for i in range(current_data.shape[1]):
            current_feature = current_data[:, i]
            reference_feature = self.reference_data[:, i]

            # Two-sample t-test for mean comparison
            t_stat, p_value = stats.ttest_ind(current_feature,
   reference_feature)

            # Kolmogorov-Smirnov test for distribution comparison
            ks_stat, ks_p_value = stats.ks_2samp(current_feature,
   reference_feature)
```

```
150
151             drift_results[f'feature_{i}'] = {
152                 't_test_p_value': p_value,
153                 'ks_test_p_value': ks_p_value,
154                 'drift_detected': p_value < self.alpha or ks_p_value <
    self.alpha,
155                 'mean_change': np.mean(current_feature) - self.
    reference_stats['means'][i]
156             }
157
158         return drift_results
159
160     def detect_multivariate_drift(self, current_data):
161             Detect multivariate drift using Hotelling's T-squared
162         # Simplified approach - in practice, use more sophisticated
    methods
163         reference_mean = np.mean(self.reference_data, axis=0)
164         current_mean = np.mean(current_data, axis=0)
165
166         # This is a simplified version - real implementation would be more
     complex
167         mean_diff = np.linalg.norm(current_mean - reference_mean)
168
169         return {
170             'mean_difference': mean_diff,
171             'drift_magnitude': mean_diff / np.linalg.norm(reference_mean)
172         }
173
174 # Use statistical drift detection
175 drift_detector = StatisticalDriftDetector(X_train_orig)
176
177 # Generate data with different drift levels
178 X_no_drift, y_no_drift, _ = generate_production_data(datetime(2024, 1, 1),
     200, drift_factor=0)
179 X_moderate_drift, y_moderate_drift, _ = generate_production_data(datetime
    (2024, 1, 1), 200, drift_factor=1)
180 X_heavy_drift, y_heavy_drift, _ = generate_production_data(datetime(2024,
    1, 1), 200, drift_factor=3)
181
182 print( Statistical drift detection results: )
183
184 # No drift case
185 no_drift_results = drift_detector.detect_feature_drift(X_no_drift)
186 drift_count_no = sum(1 for r in no_drift_results.values() if r['
    drift_detected'])
187 print(f   No drift: {drift_count_no} features showing drift )
188
189 # Moderate drift case
190 moderate_drift_results = drift_detector.detect_feature_drift(
    X_moderate_drift)
191 drift_count_moderate = sum(1 for r in moderate_drift_results.values() if r
    ['drift_detected'])
192 print(f   Moderate drift: {drift_count_moderate} features showing drift )
193
```

```
# Heavy drift case
heavy_drift_results = drift_detector.detect_feature_drift(X_heavy_drift)
drift_count_heavy = sum(1 for r in heavy_drift_results.values() if r['
    drift_detected'])
print(f   Heavy drift: {drift_count_heavy} features showing drift )

# CORRECT WAY 2 - Concept drift detection
print( \n=== CORRECT WAY 2: Concept Drift Detection === )

class ConceptDriftDetector:
       Detect changes in the relationship between features and target

    def __init__(self, window_size=100):
        self.window_size = window_size
        self.performance_history = []
        self.prediction_history = []

    def add_predictions(self, y_true, y_pred, timestamps=None):
           Add prediction results to history
        if timestamps is None:
            timestamps = [datetime.now()] * len(y_true)

        for i in range(len(y_true)):
            self.prediction_history.append({
                'timestamp': timestamps[i],
                'y_true': y_true[i],
                'y_pred': y_pred[i],
                'correct': y_true[i] == y_pred[i]
            })

        # Calculate current window accuracy
        if len(self.prediction_history) >= self.window_size:
            recent_predictions = self.prediction_history[-self.window_size
    :]
            current_accuracy = np.mean([p['correct'] for p in
    recent_predictions])
            self.performance_history.append(current_accuracy)

    def detect_concept_drift(self, significance_level=0.05):
           Detect concept drift using performance degradation
        if len(self.performance_history) < 2:
            return False, 0.0

        # Compare recent performance with historical baseline
        if len(self.performance_history) > 5:
            baseline_performance = np.mean(self.performance_history[:-3])
            recent_performance = np.mean(self.performance_history[-3:])

            # Statistical test for performance degradation
            if len(self.performance_history) > 6:
                # Use t-test to check if difference is significant
                t_stat, p_value = stats.ttest_rel(
                    self.performance_history[-6:-3],  # Previous period
                    self.performance_history[-3:]     # Current period
```

```
                )

                performance_degradation = baseline_performance - recent_performance
                significant_degradation = p_value < significance_level and performance_degradation > 0.05

                return significant_degradation, performance_degradation

        return False, 0.0

    def get_performance_trend(self):
            Get performance trend analysis
        if len(self.performance_history) < 2:
            return  Insufficient data

        recent_avg = np.mean(self.performance_history[-3:])
        overall_avg = np.mean(self.performance_history)

        if recent_avg < overall_avg - 0.05:
            return  Degrading
        elif recent_avg > overall_avg + 0.05:
            return  Improving
        else:
            return  Stable

# Use concept drift detection
concept_drift_detector = ConceptDriftDetector(window_size=50)

# Simulate predictions over time with concept drift
base_date = datetime(2024, 1, 1)
all_predictions = []
all_actuals = []
all_timestamps = []

# Period 1: No drift
X_period1, y_period1, dates_period1 = generate_production_data(base_date, 100, drift_factor=0)
pred_period1 = model.predict(X_period1)
concept_drift_detector.add_predictions(y_period1, pred_period1, dates_period1)

# Period 2: Concept drift
X_period2, y_period2, dates_period2 = generate_production_data(base_date + timedelta(days=100), 100, drift_factor=3)
pred_period2 = model.predict(X_period2)
concept_drift_detector.add_predictions(y_period2, pred_period2, dates_period2)

print( Concept drift detection results: )
drift_detected, degradation = concept_drift_detector.detect_concept_drift()
trend = concept_drift_detector.get_performance_trend()

```

```
print(f   Concept drift detected: {drift_detected} )
print(f   Performance degradation: {degradation:.4f} )
print(f   Performance trend: {trend} )

# CORRECT WAY 3 - Adaptive monitoring thresholds
print( \n=== CORRECT WAY 3: Adaptive Monitoring === )

class AdaptivePerformanceMonitor:
       Adaptive performance monitoring with dynamic thresholds

    def __init__(self, initial_window_size=100, confidence_level=0.95):
        self.initial_window_size = initial_window_size
        self.confidence_level = confidence_level
        self.performance_history = []
        self.baseline_stats = None
        self.alert_threshold = None

    def establish_baseline(self, performances):
           Establish baseline performance statistics
        self.performance_history.extend(performances)

        # Calculate baseline statistics
        mean_perf = np.mean(performances)
        std_perf = np.std(performances)

        # Calculate adaptive threshold using confidence intervals
        alpha = 1 - self.confidence_level
        margin_of_error = stats.norm.ppf(1 - alpha/2) * (std_perf / np.
   sqrt(len(performances)))

        self.baseline_stats = {
            'mean': mean_perf,
            'std': std_perf,
            'margin_of_error': margin_of_error
        }

        # Set alert threshold (mean - 2 * margin of error)
        self.alert_threshold = mean_perf - 2 * margin_of_error

        print(f Baseline established: Mean={mean_perf:.4f}, Std={std_perf
   :.4f} )
        print(f Alert threshold: {self.alert_threshold:.4f} )

    def monitor_performance(self, current_performance):
           Monitor current performance against adaptive thresholds
        self.performance_history.append(current_performance)

        if self.baseline_stats is None:
            print( Warning: Baseline not established yet )
            return False

        # Check if performance is below threshold
        alert_triggered = current_performance < self.alert_threshold

```

```
        # Update threshold adaptively (moving average approach)
        if len(self.performance_history) > self.initial_window_size:
            recent_performances = self.performance_history[-self.
    initial_window_size:]
            recent_mean = np.mean(recent_performances)
            recent_std = np.std(recent_performances)

            # Update threshold based on recent performance
            alpha = 1 - self.confidence_level
            margin_of_error = stats.norm.ppf(1 - alpha/2) * (recent_std /
    np.sqrt(len(recent_performances)))
            self.alert_threshold = recent_mean - 2 * margin_of_error

        return alert_triggered

    def get_performance_summary(self):
           Get performance summary statistics
        if not self.performance_history:
            return {}

        return {
            'current_performance': self.performance_history[-1],
            'historical_mean': np.mean(self.performance_history),
            'historical_std': np.std(self.performance_history),
            'alert_threshold': self.alert_threshold,
            'samples_monitored': len(self.performance_history)
        }

# Use adaptive monitoring
adaptive_monitor = AdaptivePerformanceMonitor()

# Establish baseline with initial good performance
baseline_performances = [0.85, 0.87, 0.86, 0.88, 0.84]  # Good performance
adaptive_monitor.establish_baseline(baseline_performances)

# Monitor degrading performance
degrading_performances = [0.83, 0.81, 0.79, 0.75, 0.72, 0.68]  # Degrading

print( Adaptive monitoring results: )
for i, perf in enumerate(degrading_performances):
    alert = adaptive_monitor.monitor_performance(perf)
    summary = adaptive_monitor.get_performance_summary()

    print(f   Sample {i+1}: Performance={perf:.4f},
          f Threshold={summary['alert_threshold']:.4f},
          f Alert={alert} )

# CORRECT WAY 4 - Comprehensive monitoring dashboard
print( \n=== CORRECT WAY 4: Comprehensive Monitoring Dashboard === )

class ModelMonitoringDashboard:
       Comprehensive model monitoring dashboard

    def __init__(self):
```

```
        self.drift_detector = None
        self.concept_drift_detector = ConceptDriftDetector()
        self.performance_monitor = AdaptivePerformanceMonitor()
        self.data_quality_monitor = None
        self.alerts = []

    def initialize_with_training_data(self, X_train, y_train):
        """Initialize monitoring with training data"""
        self.drift_detector = StatisticalDriftDetector(X_train)
        print("Monitoring dashboard initialized with training data")

    def process_batch(self, X_batch, y_batch, timestamps=None, model=None):
        """Process a batch of predictions"""
        if model is not None:
            y_pred = model.predict(X_batch)
        else:
            # Assume predictions are provided
            y_pred = y_batch  # Simplified for demo

        # Feature drift detection
        if self.drift_detector:
            drift_results = self.drift_detector.detect_feature_drift(X_batch)
            drifted_features = sum(1 for r in drift_results.values() if r['drift_detected'])

            if drifted_features > 0:
                self._add_alert(f"Feature drift detected in {drifted_features} features")

        # Concept drift detection
        self.concept_drift_detector.add_predictions(y_batch, y_pred, timestamps)
        drift_detected, degradation = self.concept_drift_detector.detect_concept_drift()

        if drift_detected:
            self._add_alert(f"Concept drift detected (degradation: {degradation:.4f})")

        # Performance monitoring
        current_accuracy = accuracy_score(y_batch, y_pred)
        alert_triggered = self.performance_monitor.monitor_performance(current_accuracy)

        if alert_triggered:
            self._add_alert(f"Performance below threshold: {current_accuracy:.4f}")

        return {
            'accuracy': current_accuracy,
            'feature_drift_count': drifted_features if 'drifted_features' in locals() else 0,
```

```
            'concept_drift': drift_detected,
            'performance_alert': alert_triggered
        }

    def _add_alert(self, message):
        """Add alert to monitoring system"""
        alert = {
            'timestamp': datetime.now(),
            'message': message,
            'severity': 'HIGH' if 'drift' in message.lower() or 'performance' in message.lower() else 'MEDIUM'
        }
        self.alerts.append(alert)
        print(f"ALERT ({alert['severity']}): {message}")

    def get_dashboard_summary(self):
        """Get comprehensive monitoring summary"""
        summary = {
            'total_alerts': len(self.alerts),
            'recent_alerts': self.alerts[-5:] if self.alerts else [],
            'performance_trend': self.concept_drift_detector.get_performance_trend(),
            'samples_processed': len(self.concept_drift_detector.prediction_history)
        }

        if self.alerts:
            severity_counts = {}
            for alert in self.alerts:
                severity = alert['severity']
                severity_counts[severity] = severity_counts.get(severity, 0) + 1
            summary['alert_severity_distribution'] = severity_counts

        return summary

    def generate_report(self):
        """Generate monitoring report"""
        summary = self.get_dashboard_summary()

        print("\n=== MODEL MONITORING DASHBOARD ===")
        print(f"Samples Processed: {summary['samples_processed']}")
        print(f"Performance Trend: {summary['performance_trend']}")
        print(f"Total Alerts: {summary['total_alerts']}")

        if 'alert_severity_distribution' in summary:
            print("Alert Severity Distribution:")
            for severity, count in summary['alert_severity_distribution'].items():
                print(f"  {severity}: {count}")

        if summary['recent_alerts']:
            print("\nRecent Alerts:")
            for alert in summary['recent_alerts']:
```

```
                print(f   {alert['timestamp'].strftime('%Y-%m-%d %H:%M:%S')} -
                      f {alert['severity']} - {alert['message']} )

# Use comprehensive monitoring dashboard
dashboard = ModelMonitoringDashboard()
dashboard.initialize_with_training_data(X_train_orig, y_train_orig)

# Simulate production monitoring
print( Simulating production monitoring... )

# Batch 1: Normal performance
X_batch1, y_batch1, dates_batch1 = generate_production_data(datetime(2024, 1, 1), 50, drift_factor=0)
results_batch1 = dashboard.process_batch(X_batch1, y_batch1, dates_batch1, model)

# Batch 2: Feature drift
X_batch2, y_batch2, dates_batch2 = generate_production_data(datetime(2024, 1, 2), 50, drift_factor=1)
results_batch2 = dashboard.process_batch(X_batch2, y_batch2, dates_batch2, model)

# Batch 3: Concept drift + performance degradation
X_batch3, y_batch3, dates_batch3 = generate_production_data(datetime(2024, 1, 3), 50, drift_factor=3)
results_batch3 = dashboard.process_batch(X_batch3, y_batch3, dates_batch3, model)

print( \nBatch Results: )
print(f Batch 1: {results_batch1} )
print(f Batch 2: {results_batch2} )
print(f Batch 3: {results_batch3} )

# Generate final report
dashboard.generate_report()

# Advanced monitoring techniques
class AdvancedMonitoring:
       Advanced monitoring techniques

    @staticmethod
    def cusum_monitoring(baseline_mean, current_values, threshold=5, drift=1):
            CUSUM (Cumulative Sum) control chart for drift detection
        # CUSUM algorithm for detecting small shifts
        cusum_pos = 0
        cusum_neg = 0
        alarms = []

        for i, value in enumerate(current_values):
            # Calculate deviation from baseline
            deviation = value - baseline_mean

```

```
            # Update CUSUM statistics
            cusum_pos = max(0, cusum_pos + (deviation - drift/2))
            cusum_neg = max(0, cusum_neg - (deviation + drift/2))

            # Check for alarms
            if cusum_pos > threshold or cusum_neg > threshold:
                alarms.append({
                    'index': i,
                    'value': value,
                    'cusum_pos': cusum_pos,
                    'cusum_neg': cusum_neg
                })

        return alarms

    @staticmethod
    def ewma_monitoring(baseline_mean, current_values, lambda_param=0.2, threshold=2):
        """EWMA (Exponentially Weighted Moving Average) for monitoring"""
        ewma_values = []
        alarms = []

        # Initialize EWMA
        ewma = baseline_mean

        for i, value in enumerate(current_values):
            # Update EWMA
            ewma = lambda_param * value + (1 - lambda_param) * ewma
            ewma_values.append(ewma)

            # Calculate standardized EWMA (simplified)
            std_ewma = abs(ewma - baseline_mean)

            # Check for alarms
            if std_ewma > threshold:
                alarms.append({
                    'index': i,
                    'value': value,
                    'ewma': ewma,
                    'std_ewma': std_ewma
                })

        return ewma_values, alarms

# Example of advanced monitoring techniques
print("\n=== ADVANCED: Statistical Process Control ===")

# Simulate performance data with gradual degradation
baseline_performance = 0.85
performance_data = [baseline_performance - 0.01 * i + np.random.normal(0, 0.02)
                    for i in range(100)]

# CUSUM monitoring
```

```
cusum_alarms = AdvancedMonitoring.cusum_monitoring(baseline_performance, performance_data)
print(f CUSUM alarms detected: {len(cusum_alarms)} )

# EWMA monitoring
ewma_values, ewma_alarms = AdvancedMonitoring.ewma_monitoring(baseline_performance, performance_data)
print(f EWMA alarms detected: {len(ewma_alarms)} )

# Real-time alerting system
class RealTimeAlerting:
       Real-time alerting system with multiple channels

    def __init__(self):
        self.alert_rules = []
        self.alert_channels = []

    def add_alert_rule(self, name, condition_func, severity='MEDIUM'):
           Add alert rule
        self.alert_rules.append({
            'name': name,
            'condition': condition_func,
            'severity': severity
        })

    def add_alert_channel(self, channel_func):
           Add alert channel (email, Slack, etc.)
        self.alert_channels.append(channel_func)

    def check_alerts(self, monitoring_data):
           Check all alert rules
        triggered_alerts = []

        for rule in self.alert_rules:
            if rule['condition'](monitoring_data):
                alert = {
                    'name': rule['name'],
                    'severity': rule['severity'],
                    'timestamp': datetime.now(),
                    'data': monitoring_data
                }
                triggered_alerts.append(alert)

                # Send alerts through all channels
                for channel in self.alert_channels:
                    try:
                        channel(alert)
                    except Exception as e:
                        print(f Failed to send alert through channel: {e} )

        return triggered_alerts

# Example alert channels
```

```
def console_alert_channel(alert):
        Console alert channel
    print(f      ALERT: {alert['name']} ({alert['severity']}) at {alert['
    timestamp']} )

def email_alert_channel(alert):
        Email alert channel (simplified)
    print(f      Email alert sent: {alert['name']} )

# Setup real-time alerting
alerting_system = RealTimeAlerting()
alerting_system.add_alert_channel(console_alert_channel)

# Add alert rules
alerting_system.add_alert_rule(
    'High Drift',
    lambda data: data.get('feature_drift_count', 0) > 3,
    'HIGH'
)

alerting_system.add_alert_rule(
    'Performance Drop',
    lambda data: data.get('accuracy', 1.0) < 0.7,
    'CRITICAL'
)

# Test alerting system
test_data = {'feature_drift_count': 5, 'accuracy': 0.65}
alerts = alerting_system.check_alerts(test_data)
print(f Real-time alerts triggered: {len(alerts)} )
```

Listing 42: Error 5.3.2: Data Drift and Model Monitoring Issues - Buggy Code

```
# Buggy: Common feature pipeline inconsistencies
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler, LabelEncoder,
    OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
import warnings
warnings.filterwarnings('ignore')

# Create sample data with various data types and issues
np.random.seed(42)

# Create realistic dataset with mixed data types
data = pd.DataFrame({
    'numeric_feature1': np.random.randn(1000),
    'numeric_feature2': np.random.randn(1000),
    'categorical_feature1': np.random.choice(['A', 'B', 'C'], 1000),
    'categorical_feature2': np.random.choice(['X', 'Y'], 1000),
```

```
    'binary_feature': np.random.choice([0, 1], 1000),
    'date_feature': pd.date_range('2020-01-01', periods=1000, freq='D'),
    'target': np.random.choice([0, 1], 1000)
})

# Add some missing values and data issues
data.loc[100:105, 'numeric_feature1'] = np.nan
data.loc[200:205, 'categorical_feature1'] = np.nan
data.loc[300:305, 'date_feature'] = pd.NaT

print( Original data shape: , data.shape)
print( Missing values: )
print(data.isnull().sum())

# WRONG WAY 1 - Inconsistent preprocessing between train/test
print( \n=== WRONG WAY 1: Inconsistent Train/Test Preprocessing === )

# Split data
X = data.drop('target', axis=1)
y = data['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# WRONG: Different preprocessing for train and test
# Training preprocessing
numeric_features = ['numeric_feature1', 'numeric_feature2']
categorical_features = ['categorical_feature1', 'categorical_feature2']

# WRONG: Fit scaler on training data
scaler_train = StandardScaler()
X_train_numeric_scaled = scaler_train.fit_transform(X_train[
    numeric_features])

# WRONG: Fit scaler on test data (INCONSISTENT!)
scaler_test = StandardScaler()  # Different scaler!
X_test_numeric_scaled = scaler_test.fit_transform(X_test[numeric_features
    ])  # WRONG: fit on test!

print( Training scaler mean: , scaler_train.mean_)
print( Test scaler mean: , scaler_test.mean_)
print( Problem: Different scaling parameters - data leakage and
    inconsistency! )

# WRONG WAY 2 - Fitting preprocessing on entire dataset
print( \n=== WRONG WAY 2: Fitting on Entire Dataset === )

# WRONG: Fitting preprocessing on entire dataset (data leakage)
scaler_leakage = StandardScaler()
X_numeric_scaled_leakage = scaler_leakage.fit_transform(X[numeric_features
    ])  # WRONG!

# Split after preprocessing
X_train_leak, X_test_leak, y_train_leak, y_test_leak = train_test_split(
    X_numeric_scaled_leakage, y, test_size=0.2, random_state=42
```

```
)

print( Scaler fitted on entire dataset )
print( Problem: Test data influences training preprocessing - data leakage ! )

# WRONG WAY 3 - Inconsistent categorical encoding
print( \n=== WRONG WAY 3: Inconsistent Categorical Encoding === )

# WRONG: Different categories in train vs test
X_train_cat = X_train.copy()
X_test_cat = X_test.copy()

# Add new category to test set that doesn't exist in training
X_test_cat.loc[X_test_cat.index[0], 'categorical_feature1'] = 'D'  # New category!

# WRONG: Fit encoder on training data
ohe_train = OneHotEncoder(sparse=False, handle_unknown='error')
X_train_encoded = ohe_train.fit_transform(X_train_cat[categorical_features])

# WRONG: Transform test data with new category
try:
    X_test_encoded = ohe_train.transform(X_test_cat[categorical_features])
except ValueError as e:
    print(f Error with inconsistent categories: {e} )
    print( Problem: New categories in test set not seen during training! )

# WRONG WAY 4 - Manual feature engineering inconsistencies
print( \n=== WRONG WAY 4: Manual Feature Engineering Inconsistencies === )

def manual_feature_engineering_train(df):
       Manual feature engineering for training data
    df_new = df.copy()

    # WRONG: Hard-coded values based on training data
    df_new['numeric_feature1_normalized'] = (df_new['numeric_feature1'] - df_new['numeric_feature1'].mean()) / df_new['numeric_feature1'].std()

    # WRONG: Date feature engineering with training-specific logic
    df_new['month'] = df_new['date_feature'].dt.month
    df_new['day_of_year'] = df_new['date_feature'].dt.dayofyear

    # WRONG: Categorical encoding with training-specific mappings
    category_mapping = {'A': 1, 'B': 2, 'C': 3}  # Hard-coded
    df_new['categorical_feature1_encoded'] = df_new['categorical_feature1'].map(category_mapping)

    return df_new

def manual_feature_engineering_test(df):
       Manual feature engineering for test data - INCONSISTENT!
    df_new = df.copy()
```

```
    # WRONG: Different normalization using test data statistics
    df_new['numeric_feature1_normalized'] = (df_new['numeric_feature1'] - df_new['numeric_feature1'].mean()) / df_new['numeric_feature1'].std()

    # WRONG: Different date processing
    df_new['month'] = df_new['date_feature'].dt.month
    # Missing day_of_year feature!

    # WRONG: Different categorical encoding
    category_mapping = {'A': 10, 'B': 20, 'C': 30}  # Different mapping!
    df_new['categorical_feature1_encoded'] = df_new['categorical_feature1'].map(category_mapping)

    return df_new

# Apply inconsistent feature engineering
X_train_manual = manual_feature_engineering_train(X_train)
X_test_manual = manual_feature_engineering_test(X_test)

print( Manual feature engineering applied )
print( Training features: , list(X_train_manual.columns))
print( Test features: , list(X_test_manual.columns))
print( Problem: Inconsistent feature engineering between train and test! )

# CORRECT WAY 1 - Proper pipeline with consistent preprocessing
print( \n=== CORRECT WAY 1: Proper Pipeline with Consistent Preprocessing === )

from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.base import BaseEstimator, TransformerMixin

class DateFeatureExtractor(BaseEstimator, TransformerMixin):
       Custom transformer for date feature extraction

    def __init__(self, date_columns):
        self.date_columns = date_columns

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_transformed = X.copy()
        for col in self.date_columns:
            if col in X_transformed.columns:
                X_transformed[f'{col}_year'] = X_transformed[col].dt.year
                X_transformed[f'{col}_month'] = X_transformed[col].dt.month
                X_transformed[f'{col}_day'] = X_transformed[col].dt.day
                X_transformed[f'{col}_dayofweek'] = X_transformed[col].dt.dayofweek
        return X_transformed

```

```
# Define feature types
numeric_features = ['numeric_feature1', 'numeric_feature2']
categorical_features = ['categorical_feature1', 'categorical_feature2']
binary_features = ['binary_feature']
date_features = ['date_feature']

# Create preprocessing pipeline
preprocessor = ColumnTransformer(
    transformers=[
        ('num', Pipeline([
            ('imputer', SimpleImputer(strategy='median')),
            ('scaler', StandardScaler())
        ]), numeric_features),

        ('cat', Pipeline([
            ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
            ('encoder', OneHotEncoder(sparse=False, handle_unknown='ignore'))
        ]), categorical_features),

        ('binary', 'passthrough', binary_features)
    ],
    remainder='drop'
)

# Full pipeline including date feature extraction
full_pipeline = Pipeline([
    ('date_features', DateFeatureExtractor(date_features)),
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Split data properly
X_train_correct, X_test_correct, y_train_correct, y_test_correct = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Fit pipeline on training data only
full_pipeline.fit(X_train_correct, y_train_correct)

# Transform test data using fitted pipeline
predictions_correct = full_pipeline.predict(X_test_correct)
accuracy_correct = accuracy_score(y_test_correct, predictions_correct)

print( Proper pipeline accuracy: , accuracy_correct)
print( Problem solved: Consistent preprocessing applied to both train and test! )

# CORRECT WAY 2 - Custom feature engineering pipeline
print( \n=== CORRECT WAY 2: Custom Feature Engineering Pipeline === )

class FeatureEngineer(BaseEstimator, TransformerMixin):
```

```
        Custom feature engineering transformer

    def __init__(self):
        self.numeric_stats = {}
        self.category_mappings = {}
        self.date_stats = {}

    def fit(self, X, y=None):
            Fit feature engineering parameters on training data
        # Store numeric statistics
        for col in ['numeric_feature1', 'numeric_feature2']:
            if col in X.columns:
                self.numeric_stats[col] = {
                    'mean': X[col].mean(),
                    'std': X[col].std(),
                    'min': X[col].min(),
                    'max': X[col].max()
                }

        # Store categorical mappings
        for col in ['categorical_feature1', 'categorical_feature2']:
            if col in X.columns:
                unique_values = X[col].dropna().unique()
                self.category_mappings[col] = {val: idx for idx, val in
    enumerate(unique_values)}

        # Store date statistics
        for col in ['date_feature']:
            if col in X.columns and not X[col].isna().all():
                self.date_stats[col] = {
                    'min_date': X[col].min(),
                    'max_date': X[col].max()
                }

        return self

    def transform(self, X):
            Apply feature engineering using fitted parameters
        X_transformed = X.copy()

        # Apply numeric transformations
        for col, stats in self.numeric_stats.items():
            if col in X_transformed.columns:
                # Standardization using training statistics
                X_transformed[f'{col}_standardized'] = (
                    X_transformed[col] - stats['mean']
                ) / stats['std']

                # Min-max scaling
                range_val = stats['max'] - stats['min']
                if range_val > 0:
                    X_transformed[f'{col}_normalized'] = (
                        X_transformed[col] - stats['min']
                    ) / range_val
```

```
272
273         # Apply categorical transformations
274         for col, mapping in self.category_mappings.items():
275             if col in X_transformed.columns:
276                 # Use training mappings, handle unknown categories
277                 X_transformed[f'{col}_encoded'] = X_transformed[col].map(
    mapping).fillna(-1)
278
279         # Apply date transformations
280         for col, stats in self.date_stats.items():
281             if col in X_transformed.columns and not X_transformed[col].
    isna().all():
282                 # Extract date components
283                 X_transformed[f'{col}_year'] = X_transformed[col].dt.year
284                 X_transformed[f'{col}_month'] = X_transformed[col].dt.
    month
285                 X_transformed[f'{col}_day'] = X_transformed[col].dt.day
286                 X_transformed[f'{col}_dayofweek'] = X_transformed[col].dt.
    dayofweek
287
288                 # Days since reference date
289                 if 'min_date' in stats:
290                     X_transformed[f'{col}_days_since_start'] = (
291                         X_transformed[col] - stats['min_date']
292                     ).dt.days
293
294         return X_transformed
295
296 # Use custom feature engineering pipeline
297 feature_engineer = FeatureEngineer()
298 feature_engineer.fit(X_train)
299
300 # Transform both training and test data consistently
301 X_train_engineered = feature_engineer.transform(X_train)
302 X_test_engineered = feature_engineer.transform(X_test)
303
304 print( Custom feature engineering applied consistently )
305 print( Training engineered features: , list(X_train_engineered.columns))
306 print( Test engineered features: , list(X_test_engineered.columns))
307
308 # CORRECT WAY 3 - Pipeline with proper handling of missing categories
309 print( \n=== CORRECT WAY 3: Pipeline with Missing Category Handling === )
310
311 # Create pipeline that properly handles unseen categories
312 robust_preprocessor = ColumnTransformer(
313     transformers=[
314         ('num', Pipeline([
315             ('imputer', SimpleImputer(strategy='median')),
316             ('scaler', StandardScaler())
317         ]), numeric_features),
318
319         ('cat', Pipeline([
320             ('imputer', SimpleImputer(strategy='constant', fill_value='
    missing')),
```

```
            ('encoder', OneHotEncoder(sparse=False, handle_unknown='ignore'))
        ]), categorical_features),

        ('binary', 'passthrough', binary_features)
    ],
    remainder='drop'
)

# Test with unseen categories
X_test_with_new_cat = X_test.copy()
X_test_with_new_cat.loc[X_test_with_new_cat.index[0], 'categorical_feature1'] = 'NEW_CATEGORY'

# Fit on training data
robust_preprocessor.fit(X_train)

# Transform test data with new category - should handle gracefully
X_test_transformed = robust_preprocessor.transform(X_test_with_new_cat)
print( Pipeline handles unseen categories gracefully )
print( Transformed shape: , X_test_transformed.shape)

# CORRECT WAY 4 - Feature validation and consistency checking
print( \n=== CORRECT WAY 4: Feature Validation and Consistency === )

class FeatureValidator(BaseEstimator, TransformerMixin):
       Feature validator to ensure consistency

    def __init__(self, expected_features=None):
        self.expected_features = expected_features
        self.feature_stats = {}

    def fit(self, X, y=None):
           Store feature statistics for validation
        self.feature_stats = {}
        for col in X.columns:
            if X[col].dtype in ['int64', 'float64']:
                self.feature_stats[col] = {
                    'mean': X[col].mean(),
                    'std': X[col].std(),
                    'min': X[col].min(),
                    'max': X[col].max()
                }
        return self

    def transform(self, X):
           Validate and transform features
        # Check feature consistency
        if self.expected_features and set(X.columns) != set(self.expected_features):
            missing_features = set(self.expected_features) - set(X.columns)
            extra_features = set(X.columns) - set(self.expected_features)

```

```
            if missing_features:
                print(f Warning: Missing features: {missing_features} )
            if extra_features:
                print(f Warning: Extra features: {extra_features} )

        # Check for extreme values (data drift detection)
        for col in X.columns:
            if col in self.feature_stats and X[col].dtype in ['int64', '
    float64']:
                stats = self.feature_stats[col]
                outliers = X[(X[col] < stats['mean'] - 3*stats['std']) |
                             (X[col] > stats['mean'] + 3*stats['std'])]
                if len(outliers) > 0:
                    print(f Warning: {len(outliers)} outliers detected in
    {col} )

        return X

# Use feature validator in pipeline
validated_pipeline = Pipeline([
    ('feature_engineer', FeatureEngineer()),
    ('validator', FeatureValidator()),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Fit and validate
validated_pipeline.fit(X_train, y_train)
predictions_validated = validated_pipeline.predict(X_test)
accuracy_validated = accuracy_score(y_test, predictions_validated)

print( Validated pipeline accuracy: , accuracy_validated)

# Advanced feature pipeline with error handling
class RobustFeaturePipeline:
       Robust feature pipeline with comprehensive error handling

    def __init__(self):
        self.pipeline_steps = []
        self.fitted_params = {}

    def add_step(self, name, transformer):
           Add processing step to pipeline
        self.pipeline_steps.append((name, transformer))

    def fit_transform(self, X, y=None):
           Fit and transform data through pipeline
        X_processed = X.copy()

        for name, transformer in self.pipeline_steps:
            try:
                if hasattr(transformer, 'fit_transform'):
                    X_processed = transformer.fit_transform(X_processed, y
    )
                elif hasattr(transformer, 'fit') and hasattr(transformer,
```

```
    'transform'):
                    transformer.fit(X_processed, y)
                    X_processed = transformer.transform(X_processed)
                else:
                    X_processed = transformer(X_processed)

                print(f Step '{name}' completed successfully )

            except Exception as e:
                print(f Error in step '{name}': {e} )
                # Continue with previous state or apply fallback
                continue

        return X_processed

    def transform(self, X):
           Transform new data using fitted pipeline
        X_processed = X.copy()

        for name, transformer in self.pipeline_steps:
            try:
                if hasattr(transformer, 'transform'):
                    X_processed = transformer.transform(X_processed)
                else:
                    X_processed = transformer(X_processed)

            except Exception as e:
                print(f Error transforming with step '{name}': {e} )
                continue

        return X_processed

# Example of robust pipeline
print( \n=== ADVANCED: Robust Feature Pipeline === )

robust_pipeline = RobustFeaturePipeline()
robust_pipeline.add_step('imputer', SimpleImputer(strategy='median'))
robust_pipeline.add_step('scaler', StandardScaler())
robust_pipeline.add_step('feature_selection', SelectKBest(k=5))  # This
    would need to be defined

print( Robust pipeline framework ready for production use! )

# Feature pipeline versioning and rollback
class VersionedFeaturePipeline:
       Feature pipeline with versioning and rollback capabilities

    def __init__(self):
        self.versions = {}
        self.current_version = None

    def save_version(self, version_name, pipeline):
           Save pipeline version
        import pickle
```

```
        import datetime

        version_info = {
            'pipeline': pickle.dumps(pipeline),
            'timestamp': datetime.datetime.now(),
            'version_name': version_name
        }

        self.versions[version_name] = version_info
        self.current_version = version_name
        print(f Pipeline version '{version_name}' saved )

    def load_version(self, version_name):
           Load specific pipeline version
        import pickle

        if version_name not in self.versions:
            raise ValueError(f Version {version_name} not found )

        version_info = self.versions[version_name]
        pipeline = pickle.loads(version_info['pipeline'])
        self.current_version = version_name

        print(f Pipeline version '{version_name}' loaded )
        return pipeline

    def rollback(self, version_name=None):
           Rollback to previous version
        if not version_name and len(self.versions) > 1:
            # Get previous version
            versions_list = list(self.versions.keys())
            current_idx = versions_list.index(self.current_version)
            if current_idx > 0:
                version_name = versions_list[current_idx - 1]

        if version_name:
            return self.load_version(version_name)
        else:
            raise ValueError( No previous version to rollback to )

# Example usage would require proper versioning setup
print( \nFeature pipeline versioning system ready for production
   deployment! )
```

Listing 43: Error 5.3.3: Feature Pipeline Inconsistencies - Buggy Code

```
# Buggy: Common error handling and graceful degradation issues
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
import time
import logging
```

```
from datetime import datetime

# Setup logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Create sample data
np.random.seed(42)
X, y = make_classification(n_samples=1000, n_features=10, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

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

print( Model trained successfully )
print(f Training accuracy: {model.score(X_train, y_train):.4f} )

# WRONG WAY 1 - No error handling at all
print( \n=== WRONG WAY 1: No Error Handling === )

# WRONG: No error handling - will crash on any issue
def predict_no_error_handling(model, X):
        Prediction without error handling
    predictions = model.predict(X)  # No validation, no error handling
    return predictions

# This works fine with good data
try:
    good_predictions = predict_no_error_handling(model, X_test)
    print(f Predictions successful: {len(good_predictions)} predictions )
except Exception as e:
    print(f Error occurred: {e} )

# But what if there's an issue?
print( Problem: No protection against data issues, model failures, or
    system problems! )

# WRONG WAY 2 - Basic try-except without proper handling
print( \n=== WRONG WAY 2: Basic Error Handling === )

# WRONG: Generic exception handling that doesn't address specific issues
def predict_basic_error_handling(model, X):
        Basic error handling - too generic
    try:
        predictions = model.predict(X)
        return predictions
    except Exception as e:
        print(f Error: {e} )  # Just print and continue
        return None  # Return None - caller must handle this!

# This can lead to cascading failures
result = predict_basic_error_handling(model, X_test)
```

```
if result is None:
    print( Prediction failed - need to handle None result everywhere! )
print( Problem: Generic error handling doesn't solve the underlying issues ! )

# WRONG WAY 3 - Silent failures
print( \n=== WRONG WAY 3: Silent Failures === )

# WRONG: Silently handling errors without notification
def predict_silent_failure(model, X):
    Silent failure handling
    try:
        predictions = model.predict(X)
        return predictions
    except:
        return np.zeros(len(X))  # Return zeros silently!

# This can mask serious problems
silent_result = predict_silent_failure(model, X_test)
print(f Silent failure result shape: {silent_result.shape} )
print( Problem: Serious errors masked as successful predictions! )

# WRONG WAY 4 - No fallback mechanisms
print( \n=== WRONG WAY 4: No Fallback Mechanisms === )

# WRONG: No backup plan when primary model fails
def predict_no_fallback(model, X):
    No fallback when model fails
    predictions = model.predict(X)
    return predictions

# What if the model file is corrupted or missing?
# In production, this would crash the entire system
print( Problem: Single point of failure - no redundancy or fallback! )

# CORRECT WAY 1 - Comprehensive error handling with logging
print( \n=== CORRECT WAY 1: Comprehensive Error Handling === )

import traceback

def predict_with_comprehensive_error_handling(model, X, model_name= default_model ):
    Comprehensive error handling with logging and context

    try:
        # Input validation
        if X is None:
            raise ValueError( Input data X cannot be None )

        if len(X) == 0:
            raise ValueError( Input data X is empty )

        # Check if model exists and is fitted
        if model is None:
```

```
            raise ValueError( Model is None )

        if not hasattr(model, 'predict'):
            raise AttributeError( Model does not have predict method )

        # Make predictions
        logger.info(f Starting prediction with {model_name} )
        start_time = time.time()

        predictions = model.predict(X)

        end_time = time.time()
        logger.info(f Prediction completed in {end_time - start_time:.4f} seconds )
        logger.info(f Generated {len(predictions)} predictions )

        return predictions

    except ValueError as e:
        logger.error(f ValueError in {model_name} prediction: {e} )
        logger.error(f Input shape: {X.shape if hasattr(X, 'shape') else 'Unknown'} )
        raise
    except AttributeError as e:
        logger.error(f AttributeError in {model_name} prediction: {e} )
        raise
    except Exception as e:
        logger.error(f Unexpected error in {model_name} prediction: {e} )
        logger.error(f Traceback: {traceback.format_exc()} )
        raise

# Test comprehensive error handling
try:
    comprehensive_result = predict_with_comprehensive_error_handling(model, X_test,  random_forest )
    print(f Comprehensive error handling successful: {len(comprehensive_result)} predictions )
except Exception as e:
    print(f Comprehensive error handling caught: {e} )

# CORRECT WAY 2 - Graceful degradation with fallback
print( \n=== CORRECT WAY 2: Graceful Degradation === )

from sklearn.linear_model import LogisticRegression
from sklearn.dummy import DummyClassifier

class GracefulPredictionService:
       Prediction service with graceful degradation

    def __init__(self):
        self.primary_model = None
        self.backup_model = None
        self.fallback_model = None
        self.model_status = {
```

```
            'primary': 'unknown',
            'backup': 'unknown',
            'fallback': 'unknown'
        }

    def initialize_models(self, X_train, y_train):
            Initialize models with different complexity levels
        try:
            # Primary model - complex
            self.primary_model = RandomForestClassifier(n_estimators=100,
random_state=42)
            self.primary_model.fit(X_train, y_train)
            self.model_status['primary'] = 'ready'
            logger.info( Primary model initialized successfully )
        except Exception as e:
            logger.error(f Failed to initialize primary model: {e} )
            self.model_status['primary'] = 'failed'

        try:
            # Backup model - simpler
            self.backup_model = LogisticRegression(random_state=42,
max_iter=1000)
            self.backup_model.fit(X_train, y_train)
            self.model_status['backup'] = 'ready'
            logger.info( Backup model initialized successfully )
        except Exception as e:
            logger.error(f Failed to initialize backup model: {e} )
            self.model_status['backup'] = 'failed'

        try:
            # Fallback model - very simple
            self.fallback_model = DummyClassifier(strategy='most_frequent'
)
            self.fallback_model.fit(X_train, y_train)
            self.model_status['fallback'] = 'ready'
            logger.info( Fallback model initialized successfully )
        except Exception as e:
            logger.error(f Failed to initialize fallback model: {e} )
            self.model_status['fallback'] = 'failed'

    def predict_with_fallback(self, X, timeout=30):
            Predict with automatic fallback on failure
        start_time = time.time()

        # Try primary model first
        if self.model_status['primary'] == 'ready':
            try:
                logger.info( Attempting prediction with primary model )
                predictions = self._safe_predict(self.primary_model, X,
timeout)
                logger.info( Primary model prediction successful )
                return predictions, 'primary'
            except Exception as e:
                logger.warning(f Primary model failed: {e} )
```

```
                self.model_status['primary'] = 'failed'

        # Try backup model
        if self.model_status['backup'] == 'ready':
            try:
                logger.info( Attempting prediction with backup model )
                predictions = self._safe_predict(self.backup_model, X,
    timeout)
                logger.info( Backup model prediction successful )
                return predictions, 'backup'
            except Exception as e:
                logger.warning(f Backup model failed: {e} )
                self.model_status['backup'] = 'failed'

        # Try fallback model
        if self.model_status['fallback'] == 'ready':
            try:
                logger.info( Attempting prediction with fallback model )
                predictions = self._safe_predict(self.fallback_model, X,
    timeout)
                logger.info( Fallback model prediction successful )
                return predictions, 'fallback'
            except Exception as e:
                logger.error(f Fallback model failed: {e} )
                self.model_status['fallback'] = 'failed'

        # All models failed
        raise RuntimeError( All prediction models failed )

    def _safe_predict(self, model, X, timeout):
           Safe prediction with timeout
        # In practice, you might use threading or other timeout mechanisms
        # This is a simplified version
        return model.predict(X)

# Test graceful degradation
graceful_service = GracefulPredictionService()
graceful_service.initialize_models(X_train, y_train)

try:
    predictions, model_used = graceful_service.predict_with_fallback(
    X_test)
    print(f Graceful prediction successful using {model_used} model )
    print(f Predictions: {len(predictions)} )
except Exception as e:
    print(f Graceful prediction failed: {e} )

# CORRECT WAY 3 - Circuit breaker pattern
print( \n=== CORRECT WAY 3: Circuit Breaker Pattern === )

import time
from enum import Enum

class CircuitState(Enum):
```

```
    CLOSED =  closed
    OPEN =  open
    HALF_OPEN =  half_open

class CircuitBreaker:
       Circuit breaker for preventing cascading failures

    def __init__(self, failure_threshold=5, timeout=60):
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.failure_count = 0
        self.last_failure_time = None
        self.state = CircuitState.CLOSED

    def call(self, func, *args, **kwargs):
           Call function with circuit breaker protection

        if self.state == CircuitState.OPEN:
            if self.last_failure_time and time.time() - self.
   last_failure_time > self.timeout:
                self.state = CircuitState.HALF_OPEN
                logger.info( Circuit breaker half-open - testing service )
            else:
                raise Exception( Circuit breaker is OPEN - service
   unavailable )

        try:
            result = func(*args, **kwargs)
            self._on_success()
            return result
        except Exception as e:
            self._on_failure()
            raise e

    def _on_success(self):
           Handle successful call
        self.failure_count = 0
        self.state = CircuitState.CLOSED
        logger.info( Circuit breaker closed - service restored )

    def _on_failure(self):
           Handle failed call
        self.failure_count += 1
        self.last_failure_time = time.time()

        if self.failure_count >= self.failure_threshold:
            self.state = CircuitState.OPEN
            logger.warning(f Circuit breaker OPEN - {self.failure_count}
   failures )
        else:
            logger.warning(f Circuit breaker failure count: {self.
   failure_count} )

# Test circuit breaker
```

```
circuit_breaker = CircuitBreaker(failure_threshold=3, timeout=30)

def unreliable_service(X):
       Simulate unreliable service
    # Randomly fail 30% of the time
    if np.random.random() < 0.3:
        raise Exception( Service temporarily unavailable )
    return model.predict(X)

print( Testing circuit breaker with unreliable service: )
for i in range(10):
    try:
        result = circuit_breaker.call(unreliable_service, X_test[:10])
        print(f Call {i+1}: Success )
    except Exception as e:
        print(f Call {i+1}: Failed - {e} )
        print(f   Circuit state: {circuit_breaker.state} )
    time.sleep(1)  # Small delay between calls

# CORRECT WAY 4 - Health checks and monitoring
print( \n=== CORRECT WAY 4: Health Checks and Monitoring === )

class ModelHealthMonitor:
       Comprehensive model health monitoring

    def __init__(self):
        self.health_metrics = {
            'prediction_count': 0,
            'error_count': 0,
            'last_prediction_time': None,
            'average_prediction_time': 0,
            'prediction_success_rate': 1.0
        }
        self.error_log = []

    def record_prediction(self, success, prediction_time, error=None):
           Record prediction outcome for monitoring
        self.health_metrics['prediction_count'] += 1

        if success:
            # Update timing metrics
            if self.health_metrics['last_prediction_time']:
                self.health_metrics['average_prediction_time'] = (
                    self.health_metrics['average_prediction_time'] * 0.9 +
                    prediction_time * 0.1
                )
            else:
                self.health_metrics['average_prediction_time'] =
   prediction_time
        else:
            self.health_metrics['error_count'] += 1
            if error:
                self.error_log.append({
                    'timestamp': datetime.now(),
```

```
                    'error': str(error)
                })
                # Keep only last 100 errors
                if len(self.error_log) > 100:
                    self.error_log = self.error_log[-100:]

        # Update success rate
        total_predictions = self.health_metrics['prediction_count']
        successful_predictions = total_predictions - self.health_metrics['error_count']
        self.health_metrics['prediction_success_rate'] = successful_predictions / total_predictions

        self.health_metrics['last_prediction_time'] = datetime.now()

    def get_health_status(self):
        """Get current health status"""
        return {
            'status': self._calculate_status(),
            'metrics': self.health_metrics.copy(),
            'recent_errors': self.error_log[-5:] if self.error_log else []
        }

    def _calculate_status(self):
        """Calculate overall health status"""
        success_rate = self.health_metrics['prediction_success_rate']
        error_count = self.health_metrics['error_count']

        if success_rate >= 0.95 and error_count < 10:
            return "HEALTHY"
        elif success_rate >= 0.80 and error_count < 50:
            return "DEGRADED"
        else:
            return "UNHEALTHY"

    def is_healthy(self):
        """Check if service is healthy"""
        return self._calculate_status() == "HEALTHY"

# Test health monitoring
health_monitor = ModelHealthMonitor()

def monitored_prediction(model, X, health_monitor):
    """Prediction with health monitoring"""
    start_time = time.time()

    try:
        predictions = model.predict(X)
        prediction_time = time.time() - start_time

        health_monitor.record_prediction(True, prediction_time)
        return predictions
    except Exception as e:
        prediction_time = time.time() - start_time
```

```
        health_monitor.record_prediction(False, prediction_time, e)
        raise

# Test monitored predictions
print( Testing monitored predictions: )
for i in range(20):
    try:
        # Simulate some failures
        if i == 5 or i == 15:  # Force some failures
            raise Exception( Simulated prediction error )

        result = monitored_prediction(model, X_test[:5], health_monitor)
        print(f Prediction {i+1}: Success )
    except Exception as e:
        print(f Prediction {i+1}: Failed - {e} )

    # Check health status periodically
    if (i + 1) % 5 == 0:
        health_status = health_monitor.get_health_status()
        print(f   Health status: {health_status['status']} )
        print(f   Success rate: {health_status['metrics']['
   prediction_success_rate']:.2%} )

# Advanced error handling patterns
class AdvancedErrorHandling:
       Advanced error handling patterns for production systems

    @staticmethod
    def retry_with_exponential_backoff(func, max_retries=3, base_delay=1):
           Retry function with exponential backoff
        import random

        for attempt in range(max_retries + 1):
            try:
                return func()
            except Exception as e:
                if attempt == max_retries:
                    raise e

                delay = base_delay * (2 ** attempt) + random.uniform(0, 1)
                logger.warning(f Attempt {attempt + 1} failed: {e}.
   Retrying in {delay:.2f} seconds... )
                time.sleep(delay)

    @staticmethod
    def bulk_prediction_with_error_handling(model, X_batches, batch_size
   =100):
           Bulk prediction with individual batch error handling
        all_predictions = []
        batch_results = []

        for i, batch in enumerate(X_batches):
            try:
                batch_predictions = model.predict(batch)
```

```
                all_predictions.extend(batch_predictions)
                batch_results.append({
                    'batch': i,
                    'status': 'success',
                    'count': len(batch_predictions)
                })
                logger.info(f Batch {i}: Success - {len(batch_predictions)
    } predictions )
            except Exception as e:
                logger.error(f Batch {i}: Failed - {e} )
                batch_results.append({
                    'batch': i,
                    'status': 'failed',
                    'error': str(e)
                })
                # Use fallback predictions for this batch
                fallback_predictions = np.full(len(batch), 0)  # Default
    to class 0
                all_predictions.extend(fallback_predictions)

        return np.array(all_predictions), batch_results

    @staticmethod
    def prediction_with_timeout(func, timeout_seconds=30):
           Prediction with timeout protection
        import signal

        def timeout_handler(signum, frame):
            raise TimeoutError(f Prediction timed out after {
    timeout_seconds} seconds )

        # Set up timeout
        old_handler = signal.signal(signal.SIGALRM, timeout_handler)
        signal.alarm(timeout_seconds)

        try:
            result = func()
            signal.alarm(0)  # Cancel timeout
            return result
        except TimeoutError:
            raise
        except Exception as e:
            signal.alarm(0)  # Cancel timeout
            raise e
        finally:
            signal.signal(signal.SIGALRM, old_handler)  # Restore old
    handler

# Example usage of advanced error handling
print( \n=== ADVANCED: Production-Ready Error Handling === )

# Retry with exponential backoff
def unreliable_api_call():
       Simulate unreliable API call
```

```
    if np.random.random() < 0.7:  # 70% failure rate
        raise Exception( API temporarily unavailable )
    return  API call successful

try:
    result = AdvancedErrorHandling.retry_with_exponential_backoff(
        unreliable_api_call, max_retries=3, base_delay=0.5
    )
    print(f Retry successful: {result} )
except Exception as e:
    print(f Retry failed after all attempts: {e} )

# Bulk prediction with error handling
X_batches = [X_test[i:i+50] for i in range(0, len(X_test), 50)]
bulk_predictions, batch_results = AdvancedErrorHandling.
    bulk_prediction_with_error_handling(
    model, X_batches
)

successful_batches = sum(1 for r in batch_results if r['status'] == '
    success')
failed_batches = sum(1 for r in batch_results if r['status'] == 'failed')

print(f Bulk prediction results: )
print(f   Total batches: {len(batch_results)} )
print(f   Successful batches: {successful_batches} )
print(f   Failed batches: {failed_batches} )

# Error handling best practices summary
def error_handling_best_practices():
       Summary of error handling best practices

    practices = [
         1. Always validate inputs before processing ,
         2. Use specific exception handling, not generic except clauses ,
         3. Log errors with context and stack traces ,
         4. Implement graceful degradation with fallback mechanisms ,
         5. Use circuit breakers to prevent cascading failures ,
         6. Monitor service health and performance metrics ,
         7. Implement retry logic with exponential backoff ,
         8. Set timeouts to prevent hanging operations ,
         9. Use bulk processing with individual error handling ,
         10. Maintain error logs for debugging and monitoring
    ]

    print( \nError Handling Best Practices: )
    for practice in practices:
        print(f   {practice} )

error_handling_best_practices()

# Production deployment considerations
class ProductionDeploymentChecklist:
       Checklist for production deployment error handling
```

```
573
574     def __init__(self):
575         self.checks = {
576             'input_validation': False,
577             'error_logging': False,
578             'fallback_mechanisms': False,
579             'circuit_breaker': False,
580             'health_monitoring': False,
581             'timeout_handling': False,
582             'retry_logic': False,
583             'bulk_processing': False
584         }
585
586     def run_checklist(self):
587            Run production deployment checklist
588         print( \n=== PRODUCTION DEPLOYMENT CHECKLIST === )
589
590         for check, completed in self.checks.items():
591             status =       COMPLETED  if completed else       PENDING
592             print(f   {check.replace('_', ' ').title()}: {status} )
593
594         completed_checks = sum(self.checks.values())
595         total_checks = len(self.checks)
596         print(f \nProgress: {completed_checks}/{total_checks} checks
    completed )
597
598         if completed_checks == total_checks:
599             print(     All production readiness checks passed! )
600         else:
601             print(     Some checks still pending - review before
    production deployment )
602
603 # Run production checklist
604 deployment_checklist = ProductionDeploymentChecklist()
605 deployment_checklist.run_checklist()
```

Listing 44: Error 5.3.4: Error Handling and Graceful Degradation - Buggy Code

```
1  # Buggy: Common performance and scalability issues
2  import numpy as np
3  import pandas as pd
4  from sklearn.ensemble import RandomForestClassifier
5  from sklearn.model_selection import train_test_split
6  from sklearn.metrics import accuracy_score
7  from sklearn.datasets import make_classification
8  import time
9  import memory_profiler
10 from functools import wraps
11
12 # Create large dataset to demonstrate performance issues
13 np.random.seed(42)
14 print( Creating large dataset for performance testing... )
15
16 # Large dataset (100K samples, 100 features)
```

```
X_large, y_large = make_classification(n_samples=100000, n_features=100, random_state=42)
X_train_large, X_test_large, y_train_large, y_test_large = train_test_split(
    X_large, y_large, test_size=0.2, random_state=42
)

print(f Large dataset created: {X_large.shape} )

# WRONG WAY 1 - No performance optimization
print( \n=== WRONG WAY 1: No Performance Optimization === )

# WRONG: Using default parameters without optimization
def train_model_no_optimization(X, y):
        Train model without performance considerations
    model = RandomForestClassifier(random_state=42)  # Default parameters
    model.fit(X, y)
    return model

# This will be slow and memory-intensive
start_time = time.time()
model_slow = train_model_no_optimization(X_train_large, y_train_large)
training_time_slow = time.time() - start_time

print(f Slow training time: {training_time_slow:.2f} seconds )
print( Problem: Default parameters may not be optimal for large datasets! )

# WRONG WAY 2 - Loading entire dataset into memory
print( \n=== WRONG WAY 2: Loading Entire Dataset === )

# WRONG: Loading all data at once without batching
def process_entire_dataset_at_once(X, y):
        Process entire dataset at once - memory inefficient
    # This approach loads everything into memory
    processed_data = []
    for i in range(len(X)):
        # Expensive processing for each sample
        processed_sample = X[i] * 2 + np.random.normal(0, 0.1, len(X[i]))
        processed_data.append(processed_sample)
    return np.array(processed_data)

# This can cause memory issues with large datasets
print( Processing large dataset without batching... )
start_time = time.time()
try:
    processed_data = process_entire_dataset_at_once(X_large[:10000], y_large[:10000])
    processing_time = time.time() - start_time
    print(f Processing time: {processing_time:.2f} seconds )
except MemoryError:
    print( Memory error occurred - dataset too large for available memory! )
print( Problem: Loading entire dataset can cause memory issues! )
```

```

# WRONG WAY 3 - No parallel processing
print( \n=== WRONG WAY 3: No Parallel Processing === )

# WRONG: Sequential processing without parallelization
def sequential_feature_engineering(X):
       Sequential feature engineering - slow
    engineered_features = np.zeros((X.shape[0], X.shape[1] * 3))

    for i in range(X.shape[0]):  # Process each sample sequentially
        for j in range(X.shape[1]):
            engineered_features[i, j*3] = X[i, j] ** 2
            engineered_features[i, j*3 + 1] = np.sqrt(np.abs(X[i, j]))
            engineered_features[i, j*3 + 2] = np.log(np.abs(X[i, j]) + 1)

    return engineered_features

# This is very slow for large datasets
print( Sequential feature engineering on small subset... )
start_time = time.time()
sequential_result = sequential_feature_engineering(X_large[:1000])  #
   Small subset
sequential_time = time.time() - start_time
print(f Sequential processing time: {sequential_time:.2f} seconds )
print( Problem: Sequential processing doesn't utilize multiple CPU cores!
   )

# WRONG WAY 4 - No caching or memoization
print( \n=== WRONG WAY 4: No Caching === )

# WRONG: Recomputing expensive operations
def expensive_computation_no_cache(data):
       Expensive computation without caching
    # Simulate expensive computation
    time.sleep(0.01)  # Artificial delay
    return np.sum(data ** 2)

def process_data_no_cache(X):
       Process data without caching expensive operations
    results = []
    for i in range(len(X)):
        # Same computation repeated multiple times
        result1 = expensive_computation_no_cache(X[i])
        result2 = expensive_computation_no_cache(X[i])  # Same computation
    again!
        result3 = expensive_computation_no_cache(X[i])  # And again!
        results.append(result1 + result2 + result3)
    return results

print( Processing without caching... )
start_time = time.time()
no_cache_results = process_data_no_cache(X_large[:100])  # Small subset
no_cache_time = time.time() - start_time
print(f No caching processing time: {no_cache_time:.2f} seconds )
```

```
print( Problem: Redundant computations waste time and resources! )

# CORRECT WAY 1 - Optimized model parameters
print( \n=== CORRECT WAY 1: Optimized Model Parameters === )

# CORRECT: Using optimized parameters for large datasets
def train_model_optimized(X, y):
       Train model with performance-optimized parameters
    model = RandomForestClassifier(
        n_estimators=100,        # Reasonable number
        max_depth=10,            # Limit depth to prevent overfitting
        min_samples_split=100,   # Increase to reduce overfitting on large
    data
        min_samples_leaf=50,     # Increase to reduce overfitting
        max_features='sqrt',     # Use subset of features
        n_jobs=-1,               # Use all available cores
        random_state=42
    )
    model.fit(X, y)
    return model

# Compare performance
start_time = time.time()
model_optimized = train_model_optimized(X_train_large, y_train_large)
training_time_optimized = time.time() - start_time

print(f Optimized training time: {training_time_optimized:.2f} seconds )
print(f Performance improvement: {training_time_slow/
   training_time_optimized:.2f}x faster )

# CORRECT WAY 2 - Batch processing for large datasets
print( \n=== CORRECT WAY 2: Batch Processing === )

def process_data_in_batches(X, y, batch_size=1000):
       Process data in batches to manage memory
    n_samples = X.shape[0]
    results = []

    for i in range(0, n_samples, batch_size):
        batch_end = min(i + batch_size, n_samples)
        X_batch = X[i:batch_end]
        y_batch = y[i:batch_end]

        # Process batch
        batch_result = {
            'batch_start': i,
            'batch_end': batch_end,
            'batch_size': batch_end - i,
            'processing_time': None
        }

        # Simulate batch processing
        start_time = time.time()
        # Actual processing would go here
```

```
        batch_processing_time = time.time() - start_time
        batch_result['processing_time'] = batch_processing_time

        results.append(batch_result)

        # Progress indicator
        if (i // batch_size) % 10 == 0:
            print(f Processed batch {(i // batch_size) + 1}/{(n_samples + batch_size - 1) // batch_size} )

    return results

# Process large dataset in batches
print( Processing large dataset in batches... )
batch_results = process_data_in_batches(X_large, y_large, batch_size=5000)
print(f Processed {len(batch_results)} batches successfully )
print( Problem solved: Memory usage controlled through batching! )

# CORRECT WAY 3 - Parallel processing
print( \n=== CORRECT WAY 3: Parallel Processing === )

from multiprocessing import Pool
import multiprocessing as mp

def parallel_feature_computation(sample):
       Compute features for a single sample
    features = []
    for value in sample:
        features.extend([value ** 2, np.sqrt(np.abs(value)), np.log(np.abs(value) + 1)])
    return features

def parallel_feature_engineering(X, n_processes=None):
       Parallel feature engineering using multiprocessing
    if n_processes is None:
        n_processes = mp.cpu_count()

    print(f Using {n_processes} processes for parallel processing )

    with Pool(processes=n_processes) as pool:
        results = pool.map(parallel_feature_computation, X)

    return np.array(results)

# Compare sequential vs parallel processing
print( Parallel feature engineering on subset... )
start_time = time.time()
parallel_result = parallel_feature_engineering(X_large[:1000], n_processes=4)
parallel_time = time.time() - start_time

print(f Parallel processing time: {parallel_time:.2f} seconds )
print(f Speedup: {sequential_time/parallel_time:.2f}x faster )

```

```
# CORRECT WAY 4 - Caching and memoization
print( \n=== CORRECT WAY 4: Caching and Memoization === )

from functools import lru_cache

# Cache expensive computations
@lru_cache(maxsize=1000)
def expensive_computation_cached(data_tuple):
        Expensive computation with caching
    # Convert tuple back to array for computation
    data = np.array(data_tuple)
    time.sleep(0.001)  # Artificial delay
    return np.sum(data ** 2)

def process_data_with_cache(X):
        Process data with caching
    results = []
    for i in range(len(X)):
        # Convert to tuple for hashing (cache key)
        data_tuple = tuple(X[i])

        # These calls will be cached after first computation
        result1 = expensive_computation_cached(data_tuple)
        result2 = expensive_computation_cached(data_tuple)  # Cached!
        result3 = expensive_computation_cached(data_tuple)  # Cached!

        results.append(result1 + result2 + result3)
    return results

# Compare caching vs no caching
print( Processing with caching... )
start_time = time.time()
cache_results = process_data_with_cache(X_large[:100])
cache_time = time.time() - start_time

print(f Caching processing time: {cache_time:.2f} seconds )
print(f Caching speedup: {no_cache_time/cache_time:.2f}x faster )

# Advanced performance optimization techniques
class PerformanceOptimizer:
        Advanced performance optimization techniques

    @staticmethod
    def memory_efficient_data_loading(file_path, chunk_size=10000):
            Memory-efficient data loading with chunking
        # This would be implemented for actual file loading
        # For demo, simulate chunked processing
        print(f Loading data in chunks of {chunk_size} )

        # Simulate chunks
        total_samples = 100000
        n_chunks = (total_samples + chunk_size - 1) // chunk_size

        for chunk_idx in range(n_chunks):
```

```
            start_idx = chunk_idx * chunk_size
            end_idx = min((chunk_idx + 1) * chunk_size, total_samples)
            chunk_size_actual = end_idx - start_idx

            # Process chunk
            yield {
                'chunk_index': chunk_idx,
                'start_index': start_idx,
                'end_index': end_idx,
                'chunk_size': chunk_size_actual
            }

    @staticmethod
    def optimized_model_training(X, y, model_type='random_forest'):
            Optimized model training with performance considerations

        if model_type == 'random_forest':
            # Use optimized parameters for large datasets
            from sklearn.ensemble import RandomForestClassifier

            model = RandomForestClassifier(
                n_estimators=100,
                max_depth=15,
                min_samples_split=100,
                min_samples_leaf=50,
                max_features='sqrt',
                n_jobs=-1,  # Use all cores
                random_state=42,
                verbose=1  # Show progress
            )

        elif model_type == 'gradient_boosting':
            from sklearn.ensemble import HistGradientBoostingClassifier

            # HistGradientBoosting is more memory efficient for large
    datasets
            model = HistGradientBoostingClassifier(
                max_iter=100,
                random_state=42,
                verbose=1
            )

        # Train model
        start_time = time.time()
        model.fit(X, y)
        training_time = time.time() - start_time

        return model, training_time

    @staticmethod
    def model_prediction_optimization(model, X, batch_size=1000):
            Optimized model prediction with batching
        n_samples = X.shape[0]
        predictions = []
```

```
        for i in range(0, n_samples, batch_size):
            batch_end = min(i + batch_size, n_samples)
            X_batch = X[i:batch_end]

            batch_predictions = model.predict(X_batch)
            predictions.extend(batch_predictions)

            # Progress indicator for large datasets
            if n_samples > 10000 and (i // batch_size) % 10 == 0:
                progress = (i + batch_size) / n_samples * 100
                print(f Prediction progress: {progress:.1f}% )

        return np.array(predictions)

# Test advanced optimization techniques
print( \n=== ADVANCED: Performance Optimization Techniques === )

# Memory-efficient data loading
print( Memory-efficient data loading: )
chunks = list(PerformanceOptimizer.memory_efficient_data_loading('
    dummy_path.csv', chunk_size=25000))
print(f Generated {len(chunks)} chunks )

# Optimized model training comparison
print( \nComparing model training optimization: )

# Standard Random Forest
rf_standard = RandomForestClassifier(n_estimators=100, random_state=42,
    n_jobs=1)
start_time = time.time()
rf_standard.fit(X_train_large[:10000], y_train_large[:10000])  # Smaller
    subset for demo
standard_time = time.time() - start_time

# Optimized Random Forest
rf_optimized, optimized_time = PerformanceOptimizer.
    optimized_model_training(
    X_train_large[:10000], y_train_large[:10000], 'random_forest'
)

print(f Standard RF training time: {standard_time:.2f} seconds )
print(f Optimized RF training time: {optimized_time:.2f} seconds )
if optimized_time > 0:
    print(f Optimization speedup: {standard_time/optimized_time:.2f}x )

# Batch prediction optimization
print( \nBatch prediction optimization: )
batch_predictions = PerformanceOptimizer.model_prediction_optimization(
    rf_optimized, X_test_large[:5000], batch_size=1000
)
print(f Batch predictions completed: {len(batch_predictions)} predictions
    )

```

```
# Scalability testing framework
class ScalabilityTester:
       Framework for testing model scalability

    def __init__(self):
        self.results = []

    def test_scalability(self, model, X_train, y_train, X_test, y_test,
                         sample_sizes=[1000, 5000, 10000, 50000]):
           Test model scalability with different sample sizes

        for size in sample_sizes:
            if size > len(X_train):
                continue

            print(f \nTesting with {size} samples... )

            # Training scalability
            X_train_subset = X_train[:size]
            y_train_subset = y_train[:size]

            start_time = time.time()
            model.fit(X_train_subset, y_train_subset)
            train_time = time.time() - start_time

            # Prediction scalability
            start_time = time.time()
            predictions = model.predict(X_test[:min(1000, len(X_test))])
            pred_time = time.time() - start_time

            # Accuracy
            accuracy = accuracy_score(y_test[:len(predictions)],
   predictions)

            result = {
                'sample_size': size,
                'training_time': train_time,
                'prediction_time': pred_time,
                'accuracy': accuracy
            }

            self.results.append(result)
            print(f   Training time: {train_time:.2f}s )
            print(f   Prediction time: {pred_time:.4f}s )
            print(f   Accuracy: {accuracy:.4f} )

        return self.results

    def plot_scalability_results(self):
           Plot scalability results
        if not self.results:
            print( No results to plot )
            return

```

```
        import matplotlib.pyplot as plt

        sample_sizes = [r['sample_size'] for r in self.results]
        train_times = [r['training_time'] for r in self.results]
        pred_times = [r['prediction_time'] for r in self.results]
        accuracies = [r['accuracy'] for r in self.results]

        fig, axes = plt.subplots(1, 3, figsize=(15, 5))

        # Training time vs sample size
        axes[0].plot(sample_sizes, train_times, 'o-')
        axes[0].set_xlabel('Sample Size')
        axes[0].set_ylabel('Training Time (s)')
        axes[0].set_title('Training Time vs Sample Size')
        axes[0].grid(True)

        # Prediction time vs sample size
        axes[1].plot(sample_sizes, pred_times, 'o-')
        axes[1].set_xlabel('Sample Size')
        axes[1].set_ylabel('Prediction Time (s)')
        axes[1].set_title('Prediction Time vs Sample Size')
        axes[1].grid(True)

        # Accuracy vs sample size
        axes[2].plot(sample_sizes, accuracies, 'o-')
        axes[2].set_xlabel('Sample Size')
        axes[2].set_ylabel('Accuracy')
        axes[2].set_title('Accuracy vs Sample Size')
        axes[2].grid(True)

        plt.tight_layout()
        plt.show()

# Test scalability
print( \n=== SCALABILITY TESTING === )

scaler = ScalabilityTester()
test_model = RandomForestClassifier(n_estimators=50, random_state=42,
    n_jobs=-1)

# Test with smaller dataset for demo
scalability_results = scaler.test_scalability(
    test_model, X_large, y_large, X_large, y_large,
    sample_sizes=[1000, 5000, 10000, 20000]
)

print( \nScalability test completed! )
print( Performance scales reasonably with dataset size )

# Resource monitoring and optimization
class ResourceMonitor:
       Monitor system resources during model operations

    def __init__(self):
```

```
        self.metrics = []

    def monitor_resources(self, func, *args, **kwargs):
        """Monitor resources during function execution"""
        import psutil
        import os

        # Get initial resource usage
        process = psutil.Process(os.getpid())
        initial_memory = process.memory_info().rss / 1024 / 1024  # MB
        initial_cpu = process.cpu_percent()

        start_time = time.time()

        # Execute function
        result = func(*args, **kwargs)

        end_time = time.time()
        final_memory = process.memory_info().rss / 1024 / 1024  # MB
        final_cpu = process.cpu_percent()

        metrics = {
            'execution_time': end_time - start_time,
            'memory_used': final_memory - initial_memory,
            'cpu_usage': final_cpu - initial_cpu,
            'peak_memory': final_memory
        }

        self.metrics.append(metrics)
        return result, metrics

    def get_resource_summary(self):
        """Get resource usage summary"""
        if not self.metrics:
            return {}

        return {
            'avg_execution_time': np.mean([m['execution_time'] for m in self.metrics]),
            'max_memory_usage': np.max([m['peak_memory'] for m in self.metrics]),
            'avg_memory_increase': np.mean([m['memory_used'] for m in self.metrics])
        }

# Example resource monitoring
print("\n=== RESOURCE MONITORING ===")

monitor = ResourceMonitor()

# Monitor a function execution
def sample_processing(X, y):
    """Sample processing function"""
    model = RandomForestClassifier(n_estimators=50, random_state=42)
```

```
    model.fit(X[:5000], y[:5000])
    return model.predict(X[:1000])

result, metrics = monitor.monitor_resources(
    sample_processing, X_large, y_large
)

print( Resource usage metrics: )
for key, value in metrics.items():
    print(f   {key}: {value:.2f} )

# Performance optimization checklist
def performance_optimization_checklist():
       Checklist for performance optimization

    checklist = [
         1. Use appropriate model parameters for dataset size ,
         2. Implement batch processing for large datasets ,
         3. Utilize parallel processing when possible ,
         4. Apply caching for expensive repeated computations ,
         5. Monitor memory usage and implement garbage collection ,
         6. Use memory-efficient data structures ,
         7. Optimize feature engineering pipelines ,
         8. Consider approximate algorithms for very large datasets ,
         9. Implement proper data loading strategies ,
         10. Profile code to identify bottlenecks
    ]

    print( \nPerformance Optimization Checklist: )
    for item in checklist:
        print(f   {item} )

performance_optimization_checklist()

# Production scalability considerations
class ProductionScalability:
       Production scalability considerations and best practices

    @staticmethod
    def scalability_assessment(X, y):
           Assess scalability requirements

        n_samples, n_features = X.shape
        data_size_gb = (X.nbytes + y.nbytes) / (1024**3)

        assessment = {
            'dataset_size': (n_samples, n_features),
            'data_size_gb': data_size_gb,
            'scalability_level': None,
            'recommendations': []
        }

        if n_samples < 10000:
            assessment['scalability_level'] = 'Small'
```

```
            assessment['recommendations'].extend([
                'Standard scikit-learn models should work fine',
                'Consider using cross-validation for better estimates'
            ])
        elif n_samples < 100000:
            assessment['scalability_level'] = 'Medium'
            assessment['recommendations'].extend([
                'Use n_jobs=-1 for parallel processing',
                'Consider batch processing for predictions',
                'Monitor memory usage during training'
            ])
        elif n_samples < 1000000:
            assessment['scalability_level'] = 'Large'
            assessment['recommendations'].extend([
                'Use HistGradientBoosting for better memory efficiency',
                'Implement data streaming/batching',
                'Consider dimensionality reduction',
                'Use incremental learning algorithms when possible'
            ])
        else:
            assessment['scalability_level'] = 'Very Large'
            assessment['recommendations'].extend([
                'Consider distributed computing frameworks',
                'Use online/incremental learning algorithms',
                'Implement data sampling strategies',
                'Consider approximate algorithms',
                'Use specialized libraries like Dask or Vaex'
            ])

        return assessment

    @staticmethod
    def distributed_computing_recommendations():
           Recommendations for distributed computing

        recommendations = {
            'libraries': [
                'Dask - Parallel computing library',
                'Vaex - Out-of-core dataframes',
                'Ray - Distributed computing framework',
                'Spark - Big data processing'
            ],
            'strategies': [
                'Data parallelism - distribute data across nodes',
                'Model parallelism - distribute model training',
                'Pipeline parallelism - parallelize pipeline steps',
                'Ensemble parallelism - train ensemble members in parallel
    '
            ],
            'considerations': [
                'Network overhead vs computation time',
                'Data serialization costs',
                'Fault tolerance and recovery',
                'Resource allocation and scheduling'
```

```
            ]
        }

        return recommendations

# Run scalability assessment
print( \n=== PRODUCTION SCALABILITY ASSESSMENT === )

assessment = ProductionScalability.scalability_assessment(X_large, y_large
    )
print(f Dataset size: {assessment['dataset_size']} )
print(f Data size: {assessment['data_size_gb']:.2f} GB )
print(f Scalability level: {assessment['scalability_level']} )

print( \nRecommendations: )
for rec in assessment['recommendations']:
    print(f   - {rec} )

# Distributed computing recommendations
dist_recommendations = ProductionScalability.
    distributed_computing_recommendations()
print( \nDistributed Computing Recommendations: )
print( Libraries: )
for lib in dist_recommendations['libraries']:
    print(f   - {lib} )

print( \nStrategies: )
for strategy in dist_recommendations['strategies']:
    print(f   - {strategy} )

print( \nConsiderations: )
for consideration in dist_recommendations['considerations']:
    print(f   - {consideration} )
```

Listing 45: Error 5.3.5: Performance and Scalability Issues - Buggy Code

6 6. Counter-Intuitive Behaviors

6.1 6.1 Model Behavior Surprises

```
# Counter-Intuitive: Random Forest feature importance can be misleading
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt

# Create dataset with correlated features
np.random.seed(42)

# Create correlated features
```

```
n_samples = 1000
X_base = np.random.randn(n_samples, 3)

# Create correlated copies
X_correlated = np.hstack([
    X_base,
    X_base[:, 0:1] + np.random.normal(0, 0.1, (n_samples, 1)),  # Correlated with feature 1
    X_base[:, 1:2] + np.random.normal(0, 0.1, (n_samples, 1)),  # Correlated with feature 2
    X_base[:, 2:3] + np.random.normal(0, 0.1, (n_samples, 1))   # Correlated with feature 3
])

# Create target with only first original feature being important
y = (X_base[:, 0] > 0).astype(int)

feature_names = [f'feature_{i}' for i in range(X_correlated.shape[1])]

print( Dataset created with correlated features )
print( Only feature_0 is truly important for prediction )

# Train Random Forest
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_correlated, y)

# Get feature importances
importances = rf.feature_importances_
feature_importance_df = pd.DataFrame({
    'feature': feature_names,
    'importance': importances
}).sort_values('importance', ascending=False)

print( \nRandom Forest Feature Importances: )
print(feature_importance_df)

# Counter-intuitive result: Correlated features split importance
print( \n=== COUNTER-INTUITIVE BEHAVIOR === )
print( Observation: Features 0 and 3 are highly correlated (both derived from same base) )
print( Feature 0 (original): Importance = , importances[0])
print( Feature 3 (correlated copy): Importance = , importances[3])
print( Problem: Importance is split between correlated features! )
print( The true importance of feature 0 is diluted across its correlated copies. )

# Demonstrate the effect with permutation importance
from sklearn.inspection import permutation_importance

perm_importance = permutation_importance(rf, X_correlated, y, n_repeats=10, random_state=42)

perm_importance_df = pd.DataFrame({
    'feature': feature_names,
```

```
    'perm_importance': perm_importance.importances_mean
}).sort_values('perm_importance', ascending=False)

print( \nPermutation Importances (more reliable): )
print(perm_importance_df)

# Visualize the difference
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Feature importances
ax1.barh(feature_importance_df['feature'], feature_importance_df['
    importance'])
ax1.set_xlabel('Feature Importance')
ax1.set_title('Random Forest Feature Importances\n(Split among correlated
    features)')

# Permutation importances
ax2.barh(perm_importance_df['feature'], perm_importance_df['
    perm_importance'])
ax2.set_xlabel('Permutation Importance')
ax2.set_title('Permutation Importances\n(Shows true predictive power)')

plt.tight_layout()
plt.show()

# Another counter-intuitive example: High cardinality categorical features
print( \n=== ANOTHER COUNTER-INTUITIVE EXAMPLE === )

# Create dataset with high cardinality categorical feature
np.random.seed(42)
n_samples = 1000

# Create meaningful feature
meaningful_feature = np.random.randn(n_samples)
target = (meaningful_feature > 0).astype(int)

# Create high cardinality categorical feature (random noise)
high_cardinality_feature = np.random.choice(range(100), n_samples)  # 100
    categories

# Combine features
X_mixed = np.column_stack([meaningful_feature, high_cardinality_feature])

# Train model
rf_mixed = RandomForestClassifier(n_estimators=100, random_state=42)
rf_mixed.fit(X_mixed, target)

importances_mixed = rf_mixed.feature_importances_
print(f Meaningful feature importance: {importances_mixed[0]:.4f} )
print(f High cardinality feature importance: {importances_mixed[1]:.4f} )
print( Counter-intuitive: Random noise with many categories can appear
    important! )
print( Reason: High cardinality features create many splits, appearing
    more important. )
```

```

# Demonstrate with different random states
print( \nTesting with different random states: )
for rs in [1, 42, 123, 456]:
    rf_test = RandomForestClassifier(n_estimators=100, random_state=rs)
    rf_test.fit(X_mixed, target)
    imp = rf_test.feature_importances_
    print(f Random state {rs}: Meaningful={imp[0]:.4f}, High cardinality={
   imp[1]:.4f} )
```

Listing 46: Behavior 6.1.1: Random Forest Feature Importance Misconceptions

```
# Counter-Intuitive: Cross-validation can give misleading results
import numpy as np
import pandas as pd
from sklearn.model_selection import (cross_val_score, KFold,
   StratifiedKFold,
                                     TimeSeriesSplit, LeaveOneOut)
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# Create time series data to demonstrate CV issues
np.random.seed(42)

# Generate time series data with trend
n_samples = 1000
time_points = np.arange(n_samples)
trend = 0.01 * time_points
seasonal = 2 * np.sin(2 * np.pi * time_points / 50)
noise = np.random.normal(0, 0.5, n_samples)
y_values = trend + seasonal + noise

# Create binary classification target
y_ts = (y_values > np.median(y_values)).astype(int)

# Create features with temporal correlation
X_ts = np.column_stack([
    np.roll(y_values, 1),  # Lag 1
    np.roll(y_values, 2),  # Lag 2
    np.roll(y_values, 3),  # Lag 3
])

# Remove first few rows with NaN from rolling
X_ts = X_ts[3:]
y_ts = y_ts[3:]

print( Time series dataset created )
print( Features are temporally correlated - future values can predict past
   ! )

# WRONG WAY: Regular K-Fold on time series
print( \n=== WRONG WAY: Regular K-Fold on Time Series === )
```

```
42
43 # This violates temporal ordering
44 kfold = KFold(n_splits=5, shuffle=True, random_state=42)
45 rf_model = RandomForestClassifier(n_estimators=50, random_state=42)
46
47 # Cross-validation that ignores temporal order
48 cv_scores_kfold = cross_val_score(rf_model, X_ts, y_ts, cv=kfold)
49
50 print( Regular K-Fold CV scores: , cv_scores_kfold)
51 print(f Mean CV score: {cv_scores_kfold.mean():.4f} (+/- {cv_scores_kfold.
    std() * 2:.4f}) )
52 print( Counter-intuitive: Very high scores due to data leakage! )
53
54 # CORRECT WAY: Time Series Split
55 print( \n=== CORRECT WAY: Time Series Split === )
56
57 # Proper temporal cross-validation
58 tscv = TimeSeriesSplit(n_splits=5)
59 cv_scores_tscv = cross_val_score(rf_model, X_ts, y_ts, cv=tscv)
60
61 print( Time Series CV scores: , cv_scores_tscv)
62 print(f Mean CV score: {cv_scores_tscv.mean():.4f} (+/- {cv_scores_tscv.
    std() * 2:.4f}) )
63 print( Much more realistic performance estimate! )
64
65 # Demonstrate the difference
66 print(f \nDifference in estimates: )
67 print(f K-Fold overestimates by: {cv_scores_kfold.mean() - cv_scores_tscv.
    mean():.4f} )
68
69 # Another counter-intuitive example: Stratified CV with imbalanced small
    datasets
70 print( \n=== ANOTHER COUNTER-INTUITIVE EXAMPLE: Small Imbalanced Datasets
    === )
71
72 # Create small, highly imbalanced dataset
73 np.random.seed(42)
74 X_imb, y_imb = make_classification(
75     n_samples=50,           # Very small
76     n_features=10,
77     n_classes=2,
78     weights=[0.9, 0.1],     # 90% class 0, 10% class 1
79     random_state=42
80 )
81
82 print(f Imbalanced dataset: {len(y_imb)} samples, class distribution: {np.
    bincount(y_imb)} )
83
84 # Regular K-Fold
85 kfold_imb = KFold(n_splits=5, shuffle=True, random_state=42)
86 cv_scores_regular = cross_val_score(LogisticRegression(random_state=42),
87                                     X_imb, y_imb, cv=kfold_imb)
88
89 # Stratified K-Fold
```

```
strat_kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cv_scores_strat = cross_val_score(LogisticRegression(random_state=42),
                                  X_imb, y_imb, cv=strat_kfold)

print(f \nRegular K-Fold scores: {cv_scores_regular} )
print(f Stratified K-Fold scores: {cv_scores_strat} )
print(f Regular mean: {cv_scores_regular.mean():.4f} )
print(f Stratified mean: {cv_scores_strat.mean():.4f} )

# Counter-intuitive: Stratified CV can be more variable with small datasets
print( Counter-intuitive: Stratified CV has higher variance with small datasets! )
print( Reason: Forced class balance in small folds can create extreme cases. )

# Demonstrate with multiple random states
print( \nTesting variance with different random states: )

regular_scores_all = []
strat_scores_all = []

for rs in range(10):
    kfold_temp = KFold(n_splits=5, shuffle=True, random_state=rs)
    strat_kfold_temp = StratifiedKFold(n_splits=5, shuffle=True, random_state=rs)

    scores_reg = cross_val_score(LogisticRegression(random_state=42, max_iter=1000),
                                 X_imb, y_imb, cv=kfold_temp)
    scores_strat = cross_val_score(LogisticRegression(random_state=42, max_iter=1000),
                                   X_imb, y_imb, cv=strat_kfold_temp)

    regular_scores_all.append(scores_reg.mean())
    strat_scores_all.append(scores_strat.mean())

print(f Regular K-Fold variance: {np.var(regular_scores_all):.6f} )
print(f Stratified K-Fold variance: {np.var(strat_scores_all):.6f} )
print( Counter-intuitive: Stratified CV can have higher variance on small datasets! )

# Leave-One-Out CV surprises
print( \n=== LEAVE-ONE-OUT CV SURPRISES === )

# LOO CV on small dataset
loo = LeaveOneOut()
loo_scores = cross_val_score(LogisticRegression(random_state=42, max_iter=1000),
                             X_imb[:20], y_imb[:20], cv=loo)

print(f LOO CV scores (first 10): {loo_scores[:10]} )
print(f LOO CV mean: {loo_scores.mean():.4f} )
print(f LOO CV std: {loo_scores.std():.4f} )
```

```
136
137 # Counter-intuitive: LOO CV can be overly optimistic
138 print( Counter-intuitive: LOO CV often gives overly optimistic estimates!
    )
139 print( Reason: Each model is trained on almost the full dataset. )
140
141 # Visualize CV score distributions
142 fig, axes = plt.subplots(1, 3, figsize=(15, 5))
143
144 axes[0].hist(cv_scores_kfold, bins=10, alpha=0.7, label='K-Fold')
145 axes[0].axvline(cv_scores_kfold.mean(), color='red', linestyle='--',
146                 label=f'Mean: {cv_scores_kfold.mean():.4f}')
147 axes[0].set_xlabel('CV Score')
148 axes[0].set_ylabel('Frequency')
149 axes[0].set_title('Regular K-Fold CV Scores\n(Overly optimistic)')
150 axes[0].legend()
151
152 axes[1].hist(cv_scores_tscv, bins=10, alpha=0.7, label='Time Series CV')
153 axes[1].axvline(cv_scores_tscv.mean(), color='red', linestyle='--',
154                 label=f'Mean: {cv_scores_tscv.mean():.4f}')
155 axes[1].set_xlabel('CV Score')
156 axes[1].set_ylabel('Frequency')
157 axes[1].set_title('Time Series CV Scores\n(Realistic)')
158 axes[1].legend()
159
160 axes[2].hist(loo_scores, bins=10, alpha=0.7, label='LOO CV')
161 axes[2].axvline(loo_scores.mean(), color='red', linestyle='--',
162                 label=f'Mean: {loo_scores.mean():.4f}')
163 axes[2].set_xlabel('CV Score')
164 axes[2].set_ylabel('Frequency')
165 axes[2].set_title('Leave-One-Out CV Scores\n(Can be overly optimistic)')
166 axes[2].legend()
167
168 plt.tight_layout()
169 plt.show()
170
171 # Advanced CV behavior demonstration
172 class CVBehaviorAnalyzer:
173         Analyze counter-intuitive CV behaviors
174
175     @staticmethod
176     def analyze_cv_variance(X, y, cv_methods, n_trials=20):
177             Analyze variance across different CV methods
178
179         results = {}
180
181         for name, cv_method in cv_methods.items():
182             trial_scores = []
183
184             for trial in range(n_trials):
185                 # Use different random state for each trial
186                 scores = cross_val_score(
187                     LogisticRegression(random_state=trial, max_iter=1000),
188                     X, y, cv=cv_method
```

```
                )
                trial_scores.append(scores.mean())

            results[name] = {
                'mean': np.mean(trial_scores),
                'std': np.std(trial_scores),
                'scores': trial_scores
            }

        return results

    @staticmethod
    def demonstrate_data_leakage():
           Demonstrate how CV can leak data

        # Create dataset where future predicts past
        np.random.seed(42)
        n = 100
        data = np.random.randn(n)

        # Features are future values, target is past values
        X_leak = data[2:].reshape(-1, 1)  # Future values
        y_leak = (data[:-2] > 0).astype(int)  # Past values as target

        # Regular CV will give overly optimistic results
        kfold = KFold(n_splits=5, shuffle=True)
        rf = RandomForestClassifier(n_estimators=50, random_state=42)

        scores = cross_val_score(rf, X_leak, y_leak, cv=kfold)

        print( Data leakage example: )
        print(f CV scores with data leakage: {scores} )
        print(f Mean: {scores.mean():.4f} )
        print( Counter-intuitive: High scores due to using future to
    predict past! )

# Run advanced analysis
print( \n=== ADVANCED CV BEHAVIOR ANALYSIS === )

cv_methods = {
    'K-Fold': KFold(n_splits=5, shuffle=True, random_state=42),
    'Stratified': StratifiedKFold(n_splits=5, shuffle=True, random_state
    =42),
    'Time Series': TimeSeriesSplit(n_splits=5)
}

analyzer = CVBehaviorAnalyzer()
cv_analysis = analyzer.analyze_cv_variance(X_imb, y_imb, cv_methods)

print( CV Method Comparison (Small Imbalanced Dataset): )
for method, results in cv_analysis.items():
    print(f   {method}: Mean={results['mean']:.4f}, Std={results['std']:.4
    f} )

```

```
# Demonstrate data leakage
analyzer.demonstrate_data_leakage()

print( \nKey Counter-Intuitive CV Behaviors: )
print( 1. Regular CV on time series data causes data leakage )
print( 2. Stratified CV can have higher variance on small datasets )
print( 3. LOO CV often gives overly optimistic estimates )
print( 4. Feature engineering before CV causes data leakage )
print( 5. CV scores can be misleading without proper context )
```

Listing 47: Behavior 6.1.2: Cross-Validation Surprises

```
# Counter-Intuitive: Feature scaling can have unexpected effects
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
    RobustScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt

# Create dataset with features of different scales
np.random.seed(42)

# Create features with very different scales
n_samples = 1000

# Feature 1: Small scale (0-1 range)
feature1 = np.random.uniform(0, 1, n_samples)

# Feature 2: Medium scale (0-100 range)
feature2 = np.random.uniform(0, 100, n_samples)

# Feature 3: Large scale (0-10000 range)
feature3 = np.random.uniform(0, 10000, n_samples)

# Feature 4: Very large scale with outliers
feature4 = np.random.normal(0, 1, n_samples) * 100000
feature4[::100] = np.random.normal(0, 1, 10) * 1000000  # Add outliers

# Create target based on all features
target = ((feature1 * 10 + feature2 * 0.1 + feature3 * 0.001 + feature4 *
    0.00001) > 500).astype(int)

# Combine features
X_mixed = np.column_stack([feature1, feature2, feature3, feature4])
feature_names = ['small_scale', 'medium_scale', 'large_scale', '
    very_large_with_outliers']

print( Dataset created with features of different scales )
print( Feature statistics: )
```

```
for i, name in enumerate(feature_names):
    print(f   {name}: Mean={X_mixed[:, i].mean():.2f}, Std={X_mixed[:, i].std():.2f} )

# Split data
X_train, X_test, y_train, y_test = train_test_split(X_mixed, target, test_size=0.2, random_state=42)

# Test different models with and without scaling
models = {
    'LogisticRegression': LogisticRegression(random_state=42, max_iter=1000),
    'SVM': SVC(random_state=42),
    'RandomForest': RandomForestClassifier(random_state=42, n_estimators=100)
}

scalers = {
    'No Scaling': None,
    'StandardScaler': StandardScaler(),
    'MinMaxScaler': MinMaxScaler(),
    'RobustScaler': RobustScaler()
}

# Counter-intuitive result 1: Different models react differently to scaling
print( \n=== COUNTER-INTUITIVE: Models React Differently to Scaling === )

results = {}

for model_name, model in models.items():
    results[model_name] = {}

    for scaler_name, scaler in scalers.items():
        try:
            if scaler is not None:
                X_train_scaled = scaler.fit_transform(X_train)
                X_test_scaled = scaler.transform(X_test)
            else:
                X_train_scaled = X_train
                X_test_scaled = X_test

            # Train model
            model.fit(X_train_scaled, y_train)

            # Predict
            y_pred = model.predict(X_test_scaled)
            accuracy = accuracy_score(y_test, y_pred)

            results[model_name][scaler_name] = accuracy

        except Exception as e:
            results[model_name][scaler_name] = f Error: {e}

```

```
# Display results
print( \nModel Performance with Different Scaling: )
for model_name, scaler_results in results.items():
    print(f \n{model_name}: )
    for scaler_name, accuracy in scaler_results.items():
        print(f   {scaler_name}: {accuracy} )

print( \nCounter-intuitive observations: )
print( 1. Random Forest performance is similar with/without scaling )
print(   Reason: Tree-based models are scale-invariant )
print( 2. Logistic Regression and SVM performance varies significantly
    with scaling )
print(   Reason: Distance-based algorithms are sensitive to feature scales
    )
print( 3. Different scalers can give different results )
print(   StandardScaler: Mean=0, Std=1 )
print(   MinMaxScaler: Range=[0,1] )
print(   RobustScaler: Uses median and IQR (robust to outliers) )

# Counter-intuitive result 2: Scaling can hurt performance
print( \n=== COUNTER-INTUITIVE: Scaling Can Hurt Performance === )

# Create dataset where original scale is meaningful
np.random.seed(42)

# Features where scale represents importance
X_meaningful = np.array([
    [1, 100, 1000],   # Feature 3 (large scale) is most important
    [2, 200, 2000],
    [3, 300, 3000],
    [4, 400, 4000],
    [5, 500, 5000]
])

y_meaningful = (X_meaningful[:, 2] > 2500).astype(int)  # Only feature 3
    matters

# Train models
lr_original = LogisticRegression(random_state=42)
lr_scaled = LogisticRegression(random_state=42)

# Without scaling
lr_original.fit(X_meaningful, y_meaningful)
coef_original = lr_original.coef_[0]

# With scaling
scaler_meaningful = StandardScaler()
X_meaningful_scaled = scaler_meaningful.fit_transform(X_meaningful)
lr_scaled.fit(X_meaningful_scaled, y_meaningful)
coef_scaled = lr_scaled.coef_[0]

print( Feature coefficients comparison: )
print( Original scale coefficients: , coef_original)
print( Scaled coefficients: , coef_scaled)
```

```
141
142 print( \nCounter-intuitive: Scaling can obscure feature importance! )
143 print( In original scale: Feature 3 coefficient is largest (correct) )
144 print( After scaling: All coefficients similar (feature importance hidden)
        )
145
146 # Counter-intuitive result 3: Different scalers handle outliers
       differently
147 print( \n=== COUNTER-INTUITIVE: Different Scaler Outlier Handling === )
148
149 # Create data with outliers
150 np.random.seed(42)
151 data_with_outliers = np.random.normal(0, 1, 100)
152 data_with_outliers[::20] = np.random.normal(0, 10, 5)  # Add outliers
153
154 print( Data with outliers statistics: )
155 print(f   Mean: {data_with_outliers.mean():.4f} )
156 print(f   Std: {data_with_outliers.std():.4f} )
157 print(f   Min: {data_with_outliers.min():.4f} )
158 print(f   Max: {data_with_outliers.max():.4f} )
159
160 # Apply different scalers
161 scalers_comparison = {
162     'StandardScaler': StandardScaler(),
163     'MinMaxScaler': MinMaxScaler(),
164     'RobustScaler': RobustScaler()
165 }
166
167 print( \nEffect of different scalers on outliers: )
168 for name, scaler in scalers_comparison.items():
169     scaled_data = scaler.fit_transform(data_with_outliers.reshape(-1, 1)).
       flatten()
170     print(f {name}: )
171     print(f   Mean: {scaled_data.mean():.4f} )
172     print(f   Std: {scaled_data.std():.4f} )
173     print(f   Min: {scaled_data.min():.4f} )
174     print(f   Max: {scaled_data.max():.4f} )
175
176 print( \nCounter-intuitive: Different scalers treat outliers very
       differently! )
177 print( StandardScaler: Outliers affect mean/std significantly )
178 print( MinMaxScaler: Outliers dominate the range )
179 print( RobustScaler: Outliers have minimal effect (uses median/IQR) )
180
181 # Visualize the scaling effects
182 fig, axes = plt.subplots(2, 2, figsize=(12, 10))
183
184 # Original data
185 axes[0, 0].hist(data_with_outliers, bins=30, alpha=0.7)
186 axes[0, 0].set_title('Original Data with Outliers')
187 axes[0, 0].set_xlabel('Value')
188 axes[0, 0].set_ylabel('Frequency')
189
190 # StandardScaler
```

```
standard_scaled = StandardScaler().fit_transform(data_with_outliers.
    reshape(-1, 1)).flatten()
axes[0, 1].hist(standard_scaled, bins=30, alpha=0.7)
axes[0, 1].set_title('StandardScaler (Mean=0, Std=1)')
axes[0, 1].set_xlabel('Value')
axes[0, 1].set_ylabel('Frequency')

# MinMaxScaler
minmax_scaled = MinMaxScaler().fit_transform(data_with_outliers.reshape
    (-1, 1)).flatten()
axes[1, 0].hist(minmax_scaled, bins=30, alpha=0.7)
axes[1, 0].set_title('MinMaxScaler (Range=[0,1])')
axes[1, 0].set_xlabel('Value')
axes[1, 0].set_ylabel('Frequency')

# RobustScaler
robust_scaled = RobustScaler().fit_transform(data_with_outliers.reshape
    (-1, 1)).flatten()
axes[1, 1].hist(robust_scaled, bins=30, alpha=0.7)
axes[1, 1].set_title('RobustScaler (Robust to Outliers)')
axes[1, 1].set_xlabel('Value')
axes[1, 1].set_ylabel('Frequency')

plt.tight_layout()
plt.show()

# Counter-intuitive result 4: Scaling order matters in pipelines
print( \n=== COUNTER-INTUITIVE: Scaling Order in Pipelines === )

from sklearn.pipeline import Pipeline
from sklearn.feature_selection import SelectKBest, f_classif

# Create pipeline where order matters
X_pipeline, y_pipeline = make_classification(n_samples=1000, n_features
    =20, random_state=42)

# Pipeline 1: Scale then select features
pipeline1 = Pipeline([
    ('scaler', StandardScaler()),
    ('selector', SelectKBest(f_classif, k=10)),
    ('classifier', LogisticRegression(random_state=42))
])

# Pipeline 2: Select features then scale
pipeline2 = Pipeline([
    ('selector', SelectKBest(f_classif, k=10)),
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression(random_state=42))
])

# Split data
X_train_p, X_test_p, y_train_p, y_test_p = train_test_split(
    X_pipeline, y_pipeline, test_size=0.2, random_state=42
)
```

```
241
242 # Fit pipelines
243 pipeline1.fit(X_train_p, y_train_p)
244 pipeline2.fit(X_train_p, y_train_p)
245
246 # Predict
247 pred1 = pipeline1.predict(X_test_p)
248 pred2 = pipeline2.predict(X_test_p)
249
250 acc1 = accuracy_score(y_test_p, pred1)
251 acc2 = accuracy_score(y_test_p, pred2)
252
253 print( Pipeline performance comparison: )
254 print(f Scale then select: {acc1:.4f} )
255 print(f Select then scale: {acc2:.4f} )
256
257 print( \nCounter-intuitive: Order of operations affects results! )
258 print( Scale then select: Uses all features for scaling, then selects )
259 print( Select then scale: Selects features first, then scales selected
    features )
260 print( The scaling parameters will be different in each case! )
261
262 # Advanced scaling behavior analysis
263 class ScalingBehaviorAnalyzer:
264         Analyze counter-intuitive scaling behaviors
265
266     @staticmethod
267     def analyze_model_sensitivity(models, scalers, X, y):
268             Analyze how different models respond to scaling
269
270         results = {}
271
272         for model_name, model in models.items():
273             results[model_name] = {}
274
275             for scaler_name, scaler in scalers.items():
276                 try:
277                     if scaler_name == 'No Scaling':
278                         X_processed = X
279                     else:
280                         X_processed = scaler.fit_transform(X)
281
282                     # Cross-validation to get stable estimates
283                     from sklearn.model_selection import cross_val_score
284                     scores = cross_val_score(model, X_processed, y, cv=5)
285
286                     results[model_name][scaler_name] = {
287                         'mean': scores.mean(),
288                         'std': scores.std(),
289                         'scores': scores
290                     }
291
292                 except Exception as e:
293                     results[model_name][scaler_name] = {'error': str(e)}
```

```
294
295         return results
296
297     @staticmethod
298     def demonstrate_scaling_impact_on_coefficients():
299             Demonstrate how scaling affects model coefficients
300
301         # Create dataset where feature scale represents importance
302         np.random.seed(42)
303         n_samples = 1000
304
305         # Features with different scales but same predictive power
306         X_impact = np.column_stack([
307             np.random.normal(0, 1, n_samples),      # Unit scale
308             np.random.normal(0, 10, n_samples),     # 10x scale
309             np.random.normal(0, 100, n_samples)     # 100x scale
310         ])
311
312         # Target depends equally on all features
313         y_impact = (X_impact[:, 0] + X_impact[:, 1]/10 + X_impact[:,
    2]/100 > 0).astype(int)
314
315         # Models
316         lr_unscaled = LogisticRegression(random_state=42)
317         lr_scaled = LogisticRegression(random_state=42)
318
319         # Fit without scaling
320         lr_unscaled.fit(X_impact, y_impact)
321         coef_unscaled = lr_unscaled.coef_[0]
322
323         # Fit with scaling
324         scaler_impact = StandardScaler()
325         X_impact_scaled = scaler_impact.fit_transform(X_impact)
326         lr_scaled.fit(X_impact_scaled, y_impact)
327         coef_scaled = lr_scaled.coef_[0]
328
329         print( Coefficient analysis: )
330         print( True importance: All features equally important )
331         print(f Unscaled coefficients: {coef_unscaled} )
332         print(f Scaled coefficients: {coef_scaled} )
333
334         print( \nCounter-intuitive: Unscaled coefficients reflect data
    scale, not importance! )
335         print( After scaling: Coefficients reflect true feature importance
     )
336
337 # Run advanced analysis
338 print( \n=== ADVANCED SCALING BEHAVIOR ANALYSIS === )
339
340 # Analyze model sensitivity
341 models_analysis = {
342     'LinearSVM': SVC(kernel='linear', random_state=42),
343     'LogisticRegression': LogisticRegression(random_state=42),
344     'RandomForest': RandomForestClassifier(random_state=42, n_estimators
```

```
    =50)
}

scalers_analysis = {
    'No Scaling': 'No Scaling',
    'StandardScaler': StandardScaler(),
    'MinMaxScaler': MinMaxScaler(),
    'RobustScaler': RobustScaler()
}

# Use a smaller dataset for demonstration
X_small, y_small = make_classification(n_samples=500, n_features=10,
    random_state=42)

analyzer = ScalingBehaviorAnalyzer()
sensitivity_results = analyzer.analyze_model_sensitivity(
    models_analysis, scalers_analysis, X_small, y_small
)

print( Model sensitivity to scaling: )
for model_name, scaler_results in sensitivity_results.items():
    print(f \n{model_name}: )
    for scaler_name, results in scaler_results.items():
        if 'error' not in results:
            print(f   {scaler_name}: {results['mean']:.4f} (+/- {results['
    std']:.4f}) )
        else:
            print(f   {scaler_name}: Error - {results['error']} )

# Demonstrate coefficient impact
analyzer.demonstrate_scaling_impact_on_coefficients()

print( \nKey Counter-Intuitive Scaling Behaviors: )
print( 1. Tree-based models are scale-invariant )
print( 2. Distance-based models are highly scale-sensitive )
print( 3. Scaling can obscure true feature importance )
print( 4. Different scalers handle outliers differently )
print( 5. Order of operations in pipelines matters )
print( 6. Scaling parameters should only be fit on training data )
```

Listing 48: Behavior 6.1.3: Feature Scaling Surprises

```
# Counter-Intuitive: Ensemble methods can behave unexpectedly
import numpy as np
import pandas as pd
from sklearn.ensemble import (RandomForestClassifier,
    GradientBoostingClassifier,
                              AdaBoostClassifier, VotingClassifier,
    BaggingClassifier)
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt
```

```
12
13 # Create challenging dataset to demonstrate ensemble behaviors
14 np.random.seed(42)
15
16 # Create dataset with different difficulty levels for different models
17 X_complex, y_complex = make_classification(
18     n_samples=2000,
19     n_features=20,
20     n_informative=10,
21     n_redundant=5,
22     n_clusters_per_class=2,
23     class_sep=0.8,
24     flip_y=0.1,  # Add some noise
25     random_state=42
26 )
27
28 # Add some highly correlated features to make it more challenging
29 correlated_features = X_complex[:, :5] + np.random.normal(0, 0.1, (2000,
       5))
30 X_complex = np.hstack([X_complex, correlated_features])
31
32 X_train, X_test, y_train, y_test = train_test_split(
33     X_complex, y_complex, test_size=0.3, random_state=42
34 )
35
36 print( Complex dataset created for ensemble testing )
37 print(f Dataset shape: {X_complex.shape} )
38
39 # Individual models for comparison
40 models_individual = {
41     'LogisticRegression': LogisticRegression(random_state=42, max_iter
       =1000),
42     'DecisionTree': DecisionTreeClassifier(random_state=42),
43     'RandomForest': RandomForestClassifier(random_state=42, n_estimators
       =100)
44 }
45
46 # Test individual models
47 print( \n=== INDIVIDUAL MODEL PERFORMANCE === )
48 individual_results = {}
49
50 for name, model in models_individual.items():
51     model.fit(X_train, y_train)
52     y_pred = model.predict(X_test)
53     accuracy = accuracy_score(y_test, y_pred)
54     individual_results[name] = accuracy
55     print(f {name}: {accuracy:.4f} )
56
57 # Counter-intuitive result 1: Ensemble not always better than best
       individual model
58 print( \n=== COUNTER-INTUITIVE: Ensemble Not Always Better === )
59
60 # Create voting ensemble
61 voting_clf = VotingClassifier(
```

```
    estimators=[
        ('lr', LogisticRegression(random_state=42, max_iter=1000)),
        ('dt', DecisionTreeClassifier(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42, n_estimators=50))
    ],
    voting='hard'
)

voting_clf.fit(X_train, y_train)
y_pred_voting = voting_clf.predict(X_test)
voting_accuracy = accuracy_score(y_test, y_pred_voting)

print( Voting Ensemble Results: )
print(f Voting Classifier: {voting_accuracy:.4f} )
print( Individual model accuracies: )
for name, acc in individual_results.items():
    print(f   {name}: {acc:.4f} )

best_individual = max(individual_results.values())
print(f \nBest individual model: {best_individual:.4f} )
print(f Ensemble vs best individual: {voting_accuracy - best_individual:.4
    f} )

print( \nCounter-intuitive: Ensemble can be worse than best individual
    model! )
print( Reasons: )
print( 1. Poor individual models can drag down ensemble performance )
print( 2. Correlated errors among ensemble members )
print( 3. Voting can average away the strengths of good models )

# Counter-intuitive result 2: Bagging can hurt performance on simple
    problems
print( \n=== COUNTER-INTUITIVE: Bagging Can Hurt Performance === )

# Simple, well-behaved dataset
X_simple, y_simple = make_classification(
    n_samples=1000,
    n_features=5,
    n_informative=5,
    n_redundant=0,
    class_sep=2.0,  # Well separated
    random_state=42
)

X_train_simple, X_test_simple, y_train_simple, y_test_simple =
    train_test_split(
    X_simple, y_simple, test_size=0.3, random_state=42
)

# Simple model (should work well on simple data)
simple_model = LogisticRegression(random_state=42)
simple_model.fit(X_train_simple, y_train_simple)
simple_accuracy = simple_model.score(X_test_simple, y_test_simple)

```

```
# Bagged version of simple model
bagged_model = BaggingClassifier(
    base_estimator=LogisticRegression(random_state=42),
    n_estimators=10,
    random_state=42
)
bagged_model.fit(X_train_simple, y_train_simple)
bagged_accuracy = bagged_model.score(X_test_simple, y_test_simple)

print( Simple Dataset Results: )
print(f Simple LogisticRegression: {simple_accuracy:.4f} )
print(f Bagged LogisticRegression: {bagged_accuracy:.4f} )
print(f Difference: {bagged_accuracy - simple_accuracy:.4f} )

print( \nCounter-intuitive: Bagging can hurt performance on simple
   problems! )
print( Reasons: )
print( 1. Simple problems don't benefit from variance reduction )
print( 2. Bagging introduces additional complexity without benefit )
print( 3. Small datasets may not support effective bagging )

# Counter-intuitive result 3: More estimators doesn't always mean better
print( \n=== COUNTER-INTUITIVE: More Estimators Doesn't Always Mean Better
    === )

# Test Random Forest with different numbers of estimators
n_estimators_range = [1, 5, 10, 50, 100, 200, 500]
rf_accuracies = []

print( Testing Random Forest with different n_estimators: )
for n_est in n_estimators_range:
    rf = RandomForestClassifier(n_estimators=n_est, random_state=42)
    rf.fit(X_train, y_train)
    accuracy = rf.score(X_test, y_test)
    rf_accuracies.append(accuracy)
    print(f   n_estimators={n_est}: {accuracy:.4f} )

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(n_estimators_range, rf_accuracies, 'o-')
plt.xlabel('Number of Estimators')
plt.ylabel('Accuracy')
plt.title('Random Forest Performance vs Number of Estimators')
plt.grid(True)
plt.show()

# Find optimal point
optimal_idx = np.argmax(rf_accuracies)
optimal_n_est = n_estimators_range[optimal_idx]
optimal_accuracy = rf_accuracies[optimal_idx]

print(f \nOptimal n_estimators: {optimal_n_est} )
print(f Optimal accuracy: {optimal_accuracy:.4f} )
print(f Diminishing returns after n_estimators={optimal_n_est} )
```

```
164
165 print( \nCounter-intuitive: More estimators can lead to diminishing
       returns! )
166 print( Reasons: )
167 print( 1. Each additional estimator provides marginal improvement )
168 print( 2. Computational cost increases linearly )
169 print( 3. Overfitting risk with too many estimators )
170 print( 4. Random variation can cause performance to plateau or decrease )
171
172 # Counter-intuitive result 4: Ensemble diversity vs performance trade-off
173 print( \n=== COUNTER-INTUITIVE: Ensemble Diversity Trade-off === )
174
175 from sklearn.metrics import accuracy_score
176
177 # Create diverse ensemble members
178 ensemble_members = [
179     LogisticRegression(random_state=42),
180     DecisionTreeClassifier(random_state=42),
181     RandomForestClassifier(n_estimators=50, random_state=42),
182     GradientBoostingClassifier(random_state=42, n_estimators=50)
183 ]
184
185 # Train individual members
186 individual_predictions = []
187 individual_accuracies = []
188
189 for i, model in enumerate(ensemble_members):
190     model.fit(X_train, y_train)
191     y_pred = model.predict(X_test)
192     accuracy = accuracy_score(y_test, y_pred)
193     individual_predictions.append(y_pred)
194     individual_accuracies.append(accuracy)
195     print(f Model {i+1} ({type(model).__name__}): {accuracy:.4f} )
196
197 # Calculate diversity (disagreement rate)
198 def calculate_diversity(predictions):
199         Calculate average disagreement rate among predictions
200     n_samples = len(predictions[0])
201     disagreements = 0
202
203     for i in range(n_samples):
204         sample_predictions = [pred[i] for pred in predictions]
205         if len(set(sample_predictions)) > 1:  # If predictions differ
206             disagreements += 1
207
208     return disagreements / n_samples
209
210 diversity = calculate_diversity(individual_predictions)
211 print(f \nEnsemble diversity (disagreement rate): {diversity:.4f} )
212
213 # Test different ensemble strategies
214 print( \nDifferent Ensemble Strategies: )
215
216 # Simple average
```

```
avg_predictions = np.mean(individual_predictions, axis=0)
avg_predictions_binary = (avg_predictions > 0.5).astype(int)
avg_accuracy = accuracy_score(y_test, avg_predictions_binary)
print(f Simple average: {avg_accuracy:.4f} )

# Majority vote
majority_vote = []
for i in range(len(y_test)):
    votes = [pred[i] for pred in individual_predictions]
    majority_vote.append(1 if sum(votes) > len(votes)/2 else 0)
majority_accuracy = accuracy_score(y_test, majority_vote)
print(f Majority vote: {majority_accuracy:.4f} )

# Weighted average (weight by individual performance)
weights = np.array(individual_accuracies)
weights = weights / np.sum(weights)  # Normalize
weighted_predictions = np.average(individual_predictions, axis=0, weights=
    weights)
weighted_predictions_binary = (weighted_predictions > 0.5).astype(int)
weighted_accuracy = accuracy_score(y_test, weighted_predictions_binary)
print(f Weighted average: {weighted_accuracy:.4f} )

print(f \nCounter-intuitive: High diversity doesn't always mean better
    performance! )
print( Sometimes diverse models make different types of errors that cancel
     out )
print( Sometimes diverse models make correlated errors that compound )

# Advanced ensemble behavior analysis
class EnsembleBehaviorAnalyzer:
       Analyze counter-intuitive ensemble behaviors

    @staticmethod
    def analyze_correlation_effects(models, X_train, y_train, X_test,
    y_test):
            Analyze how model correlation affects ensemble performance

        # Train models
        predictions = []
        accuracies = []

        for model in models:
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
            predictions.append(y_pred)
            accuracies.append(accuracy_score(y_test, y_pred))

        # Calculate pairwise correlations
        correlations = []
        for i in range(len(predictions)):
            for j in range(i+1, len(predictions)):
                corr = np.corrcoef(predictions[i], predictions[j])[0, 1]
                correlations.append(corr)

```

```
        avg_correlation = np.mean(correlations)

        # Ensemble performance
        ensemble_pred = np.mean(predictions, axis=0)
        ensemble_pred_binary = (ensemble_pred > 0.5).astype(int)
        ensemble_accuracy = accuracy_score(y_test, ensemble_pred_binary)

        return {
            'individual_accuracies': accuracies,
            'average_correlation': avg_correlation,
            'ensemble_accuracy': ensemble_accuracy,
            'correlations': correlations
        }

    @staticmethod
    def demonstrate_overfitting_in_ensembles():
           Demonstrate how ensembles can overfit

        # Create dataset with noise
        np.random.seed(42)
        X_noise, y_noise = make_classification(
            n_samples=500,
            n_features=100,  # High dimensional
            n_informative=10,
            n_redundant=20,
            flip_y=0.3,  # High noise
            random_state=42
        )

        X_train_n, X_test_n, y_train_n, y_test_n = train_test_split(
            X_noise, y_noise, test_size=0.3, random_state=42
        )

        # Test with increasing ensemble size
        ensemble_sizes = [1, 5, 10, 50, 100, 200]
        train_scores = []
        test_scores = []

        for size in ensemble_sizes:
            rf_overfit = RandomForestClassifier(
                n_estimators=size,
                max_depth=20,  # Deep trees to encourage overfitting
                min_samples_split=2,
                random_state=42
            )

            rf_overfit.fit(X_train_n, y_train_n)
            train_score = rf_overfit.score(X_train_n, y_train_n)
            test_score = rf_overfit.score(X_test_n, y_test_n)

            train_scores.append(train_score)
            test_scores.append(test_score)

            print(f Ensemble size {size}: Train={train_score:.4f}, Test={
```

```
    test_score:.4f} )

        # Plot overfitting
        plt.figure(figsize=(10, 6))
        plt.plot(ensemble_sizes, train_scores, 'o-', label='Training Score')
        plt.plot(ensemble_sizes, test_scores, 'o-', label='Test Score')
        plt.xlabel('Ensemble Size')
        plt.ylabel('Accuracy')
        plt.title('Ensemble Overfitting Demonstration')
        plt.legend()
        plt.grid(True)
        plt.show()

        print( \nCounter-intuitive: Larger ensembles can overfit! )
        print( Reason: More complex models with more parameters )

# Run advanced analysis
print( \n=== ADVANCED ENSEMBLE BEHAVIOR ANALYSIS === )

# Analyze correlation effects
diverse_models = [
    LogisticRegression(random_state=42),
    DecisionTreeClassifier(random_state=42),
    RandomForestClassifier(n_estimators=50, random_state=42),
    GradientBoostingClassifier(random_state=42, n_estimators=50)
]

analyzer = EnsembleBehaviorAnalyzer()
correlation_analysis = analyzer.analyze_correlation_effects(
    diverse_models, X_train, y_train, X_test, y_test
)

print( Correlation Analysis Results: )
print(f Average model correlation: {correlation_analysis['average_correlation']:.4f} )
print(f Individual accuracies: {[f'{acc:.4f}' for acc in correlation_analysis['individual_accuracies']]} )
print(f Ensemble accuracy: {correlation_analysis['ensemble_accuracy']:.4f} )

# Demonstrate overfitting
print( \nDemonstrating ensemble overfitting: )
analyzer.demonstrate_overfitting_in_ensembles()

print( \nKey Counter-Intuitive Ensemble Behaviors: )
print( 1. Ensembles are not always better than the best individual model )
print( 2. Bagging can hurt performance on simple problems )
print( 3. More estimators lead to diminishing returns )
print( 4. High diversity doesn't always improve performance )
print( 5. Ensembles can overfit with too many members )
print( 6. Correlated errors can reduce ensemble effectiveness )
```

```
print( 7. Weighted ensembles may not always outperform simple voting )
```

Listing 49: Behavior 6.1.4: Ensemble Method Surprises

```
# Counter-Intuitive: Hyperparameter tuning can have unexpected behaviors
import numpy as np
import pandas as pd
from sklearn.model_selection import (GridSearchCV, RandomizedSearchCV,
                                     cross_val_score, validation_curve)
from sklearn.ensemble import RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from scipy.stats import randint, uniform
import time

# Create challenging dataset for hyperparameter tuning
np.random.seed(42)
X, y = make_classification(
    n_samples=2000,
    n_features=20,
    n_informative=15,
    n_redundant=5,
    n_clusters_per_class=2,
    class_sep=1.0,
    flip_y=0.1,
    random_state=42
)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    random_state=42)

print( Dataset created for hyperparameter tuning experiments )
print(f Dataset shape: {X.shape} )

# Counter-intuitive result 1: Grid search can miss optimal parameters
print( \n=== COUNTER-INTUITIVE: Grid Search Can Miss Optimal Parameters
    === )

# Define coarse grid
param_grid_coarse = {
    'n_estimators': [10, 50, 100, 200],
    'max_depth': [3, 5, 7, 10, None],
    'min_samples_split': [2, 5, 10]
}

# Define fine grid around suspected optimal region
param_grid_fine = {
    'n_estimators': [40, 45, 50, 55, 60],
    'max_depth': [4, 5, 6],
    'min_samples_split': [4, 5, 6]
```

```
}

rf_coarse = RandomForestClassifier(random_state=42)
rf_fine = RandomForestClassifier(random_state=42)

# Coarse grid search
print( Performing coarse grid search... )
start_time = time.time()
grid_coarse = GridSearchCV(rf_coarse, param_grid_coarse, cv=5, n_jobs=-1)
grid_coarse.fit(X_train, y_train)
coarse_time = time.time() - start_time

print(f Coarse grid search completed in {coarse_time:.2f} seconds )
print(f Best parameters (coarse): {grid_coarse.best_params_} )
print(f Best score (coarse): {grid_coarse.best_score_:.4f} )

# Fine grid search around coarse best
print( \nPerforming fine grid search... )
start_time = time.time()
grid_fine = GridSearchCV(rf_fine, param_grid_fine, cv=5, n_jobs=-1)
grid_fine.fit(X_train, y_train)
fine_time = time.time() - start_time

print(f Fine grid search completed in {fine_time:.2f} seconds )
print(f Best parameters (fine): {grid_fine.best_params_} )
print(f Best score (fine): {grid_fine.best_score_:.4f} )

# Test both on test set
coarse_model = grid_coarse.best_estimator_
fine_model = grid_fine.best_estimator_

coarse_test_score = coarse_model.score(X_test, y_test)
fine_test_score = fine_model.score(X_test, y_test)

print(f \nTest set performance: )
print(f Coarse grid model: {coarse_test_score:.4f} )
print(f Fine grid model: {fine_test_score:.4f} )

print( \nCounter-intuitive: Fine grid search might not find better
    parameters! )
print( Reasons: )
print( 1. Coarse grid might have skipped the optimal region )
print( 2. Cross-validation variance can mask true performance differences
    )
print( 3. The 'optimal' parameters might overfit to validation set )

# Counter-intuitive result 2: Random search can outperform grid search
print( \n=== COUNTER-INTUITIVE: Random Search Can Outperform Grid Search
    === )

# Parameter distributions for random search
param_dist = {
    'n_estimators': randint(10, 300),
    'max_depth': randint(3, 20),
```

```
    'min_samples_split': randint(2, 20),
    'min_samples_leaf': randint(1, 10)
}

rf_random = RandomForestClassifier(random_state=42)

# Random search with same budget as grid search
print( Performing random search... )
start_time = time.time()
random_search = RandomizedSearchCV(
    rf_random, param_dist, n_iter=100, cv=5, n_jobs=-1, random_state=42
)
random_search.fit(X_train, y_train)
random_time = time.time() - start_time

print(f Random search completed in {random_time:.2f} seconds )
print(f Best parameters (random): {random_search.best_params_} )
print(f Best score (random): {random_search.best_score_:.4f} )

# Compare with grid search
print(f \nComparison with grid search: )
print(f Grid search time: {coarse_time:.2f}s, Score: {grid_coarse.
    best_score_:.4f} )
print(f Random search time: {random_time:.2f}s, Score: {random_search.
    best_score_:.4f} )

random_model = random_search.best_estimator_
random_test_score = random_model.score(X_test, y_test)
print(f Random search test score: {random_test_score:.4f} )

print( \nCounter-intuitive: Random search can be more efficient than grid
    search! )
print( Reasons: )
print( 1. Random search explores parameter space more effectively )
print( 2. Grid search can waste evaluations on redundant parameter
    combinations )
print( 3. Random search is less sensitive to parameter scaling )

# Counter-intuitive result 3: More hyperparameter combinations don't
    always help
print( \n=== COUNTER-INTUITIVE: More Combinations Don't Always Help === )

# Test with different numbers of parameter combinations
param_grids = {
    'Small': {
        'n_estimators': [50, 100],
        'max_depth': [5, 10]
    },
    'Medium': {
        'n_estimators': [25, 50, 100, 200],
        'max_depth': [3, 5, 7, 10]
    },
    'Large': {
        'n_estimators': [10, 25, 50, 100, 200, 300],
```

```
        'max_depth': [3, 5, 7, 10, 15, 20, None]
    }
}

results_complexity = {}

for grid_name, param_grid in param_grids.items():
    n_combinations = 1
    for param_values in param_grid.values():
        n_combinations *= len(param_values)

    print(f \nTesting {grid_name} grid ({n_combinations} combinations)...
    )

    rf_test = RandomForestClassifier(random_state=42)
    start_time = time.time()

    grid_test = GridSearchCV(rf_test, param_grid, cv=3)  # Reduced CV for
    speed
    grid_test.fit(X_train[:500], y_train[:500])  # Smaller dataset for
    demo

    search_time = time.time() - start_time
    results_complexity[grid_name] = {
        'combinations': n_combinations,
        'time': search_time,
        'best_score': grid_test.best_score_
    }

    print(f   Combinations: {n_combinations} )
    print(f   Time: {search_time:.2f}s )
    print(f   Best score: {grid_test.best_score_:.4f} )

print( \nComplexity Analysis Results: )
for name, results in results_complexity.items():
    print(f {name} Grid: {results['combinations']} combinations,
          f {results['time']:.2f}s, Score: {results['best_score']:.4f} )

print( \nCounter-intuitive: More parameter combinations don't always
    improve results! )
print( Reasons: )
print( 1. Diminishing returns - many combinations are similar )
print( 2. Computational cost increases exponentially )
print( 3. Overfitting to validation set with too many combinations )
print( 4. Some parameters have little impact on performance )

# Counter-intuitive result 4: Validation curves can be misleading
print( \n=== COUNTER-INTUITIVE: Validation Curves Can Be Misleading === )

# Generate validation curves for different parameters
def plot_validation_analysis(model, X, y, param_name, param_range, cv=5):
       Plot validation curve and analyze behavior

    train_scores, val_scores = validation_curve(
```

```
        model, X, y, param_name=param_name, param_range=param_range, cv=cv
    )

    train_mean = np.mean(train_scores, axis=1)
    train_std = np.std(train_scores, axis=1)
    val_mean = np.mean(val_scores, axis=1)
    val_std = np.std(val_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(param_range, train_mean, 'o-', color='blue', label='Training score')
    plt.fill_between(param_range, train_mean - train_std, train_mean + train_std, alpha=0.1, color='blue')

    plt.plot(param_range, val_mean, 'o-', color='red', label='Cross-validation score')
    plt.fill_between(param_range, val_mean - val_std, val_mean + val_std, alpha=0.1, color='red')

    plt.xlabel(param_name)
    plt.ylabel('Score')
    plt.title(f'Validation Curve for {param_name}')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Find optimal parameter
    optimal_idx = np.argmax(val_mean)
    optimal_param = param_range[optimal_idx]
    optimal_score = val_mean[optimal_idx]

    # Check for overfitting
    train_at_optimal = train_mean[optimal_idx]
    gap = train_at_optimal - optimal_score

    print(f Optimal {param_name}: {optimal_param} )
    print(f Optimal CV score: {optimal_score:.4f} )
    print(f Training score at optimum: {train_at_optimal:.4f} )
    print(f Train-CV gap: {gap:.4f} )

    if gap > 0.1:
        print( Warning: Significant overfitting detected! )
    elif gap < 0.01:
        print( Good: Low bias and variance )
    else:
        print( Acceptable: Reasonable bias-variance tradeoff )

    return optimal_param, optimal_score

# Test validation curves
print( Analyzing validation curves... )

# Max depth validation curve
print( \n1. Max Depth Analysis: )
```

```
optimal_depth, score_depth = plot_validation_analysis(
    RandomForestClassifier(n_estimators=100, random_state=42),
    X_train[:1000], y_train[:1000],
    'max_depth', [3, 5, 7, 10, 15, 20, None]
)

# n_estimators validation curve
print( \n2. n_estimators Analysis: )
optimal_n_est, score_n_est = plot_validation_analysis(
    RandomForestClassifier(max_depth=10, random_state=42),
    X_train[:1000], y_train[:1000],
    'n_estimators', [10, 50, 100, 200, 300, 500]
)

print( \nCounter-intuitive: Validation curves can suggest suboptimal parameters! )
print( Reasons: )
print( 1. Cross-validation variance can create false peaks )
print( 2. Small validation sets may not represent true performance )
print( 3. Parameter interactions not captured in single-parameter curves )
print( 4. Overfitting to validation set in the curve analysis )

# Advanced hyperparameter tuning analysis
class HyperparameterTuningAnalyzer:
        Analyze counter-intuitive hyperparameter tuning behaviors

    @staticmethod
    def analyze_parameter_interactions(model_class, X, y, param_grid, cv=5):
            Analyze interactions between hyperparameters

        # Perform grid search
        model = model_class()
        grid_search = GridSearchCV(model, param_grid, cv=cv, n_jobs=-1)
        grid_search.fit(X, y)

        # Get results
        results_df = pd.DataFrame(grid_search.cv_results_)

        # Analyze parameter importance
        param_importance = {}
        for param in param_grid.keys():
            param_name = f'param_{param}'
            if param_name in results_df.columns:
                grouped = results_df.groupby(param_name)['mean_test_score'].mean()
                param_importance[param] = grouped.std()

        return {
            'best_params': grid_search.best_params_,
            'best_score': grid_search.best_score_,
            'param_importance': param_importance,
            'results_df': results_df
        }
```

```
    @staticmethod
    def demonstrate_local_optima():
        """Demonstrate how local optima can trap optimization"""

        # Create a parameter landscape with multiple optima
        np.random.seed(42)

        # Simulate parameter space evaluation
        n_estimators_range = range(10, 200, 10)
        max_depth_range = range(3, 20)

        scores = {}
        true_optimum = (100, 10)  # True optimal parameters

        for n_est in n_estimators_range:
            for max_depth in max_depth_range:
                # Simulate score with some noise and multiple optima
                base_score = 0.8 - 0.1 * ((n_est - true_optimum[0])**2 + (max_depth - true_optimum[1])**2) / 10000
                noise = np.random.normal(0, 0.02)
                local_boost = 0.05 * np.exp(-((n_est - 50)**2 + (max_depth - 5)**2) / 1000)  # Local optimum
                scores[(n_est, max_depth)] = base_score + noise + local_boost

        # Find local and global optima
        best_params = max(scores.keys(), key=lambda x: scores[x])
        best_score = scores[best_params]

        print("Parameter Landscape Analysis:")
        print(f"Global optimum at: n_estimators={true_optimum[0]}, max_depth={true_optimum[1]}")
        print(f"Found optimum at: n_estimators={best_params[0]}, max_depth={best_params[1]}")
        print(f"Best score: {best_score:.4f}")

        # This demonstrates how optimization can get trapped in local optima
        print("\nCounter-intuitive: Optimization can find local rather than global optima!")
        print("Reason: Greedy search strategies may miss better regions of parameter space")

# Run advanced analysis
print("\n=== ADVANCED HYPERPARAMETER TUNING ANALYSIS ===")

# Analyze parameter interactions
param_grid_analysis = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 10, 15],
    'min_samples_split': [2, 5, 10]
}
```

```
analyzer = HyperparameterTuningAnalyzer()
interaction_analysis = analyzer.analyze_parameter_interactions(
    RandomForestClassifier, X_train[:500], y_train[:500],
    param_grid_analysis, cv=3
)

print( Parameter Interaction Analysis: )
print(f Best parameters: {interaction_analysis['best_params']} )
print(f Best score: {interaction_analysis['best_score']:.4f} )
print( Parameter importance (std of CV scores): )
for param, importance in interaction_analysis['param_importance'].items():
    print(f   {param}: {importance:.4f} )

# Demonstrate local optima
print( \nDemonstrating local optima in parameter space: )
analyzer.demonstrate_local_optima()

print( \nKey Counter-Intuitive Hyperparameter Tuning Behaviors: )
print( 1. Grid search can miss optimal parameters due to coarse
    discretization )
print( 2. Random search can be more efficient than systematic grid search
    )
print( 3. More parameter combinations don't always lead to better results
    )
print( 4. Validation curves can be misleading due to variance or
    interactions )
print( 5. Optimization can get trapped in local optima )
print( 6. Cross-validation scores can overfit to validation sets )
print( 7. Best parameters on validation may not be best on test set )
print( 8. Parameter importance varies by dataset and problem type )
```

Listing 50: Behavior 6.1.5: Hyperparameter Tuning Surprises

```
# Counter-Intuitive: Data preprocessing can have unexpected effects
import numpy as np
import pandas as pd
from sklearn.preprocessing import (StandardScaler, MinMaxScaler,
    RobustScaler,
                                   LabelEncoder, OneHotEncoder,
    PolynomialFeatures)
from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt
from datetime import datetime, timedelta

# Create complex dataset with various data issues
np.random.seed(42)

# Create dataset with mixed data types and issues
n_samples = 1000

```

```
# Numerical features with different scales and issues
numerical_data = {
    'small_range': np.random.uniform(0, 1, n_samples),
    'large_range': np.random.uniform(0, 10000, n_samples),
    'with_outliers': np.random.normal(0, 1, n_samples),
    'with_missing': np.random.normal(50, 10, n_samples)
}

# Add outliers to some features
outlier_indices = np.random.choice(n_samples, 20, replace=False)
numerical_data['with_outliers'][outlier_indices] = np.random.normal(0, 10,
    20)

# Add missing values
missing_indices = np.random.choice(n_samples, 50, replace=False)
numerical_data['with_missing'][missing_indices] = np.nan

# Categorical features
categorical_data = {
    'low_cardinality': np.random.choice(['A', 'B', 'C'], n_samples),
    'high_cardinality': np.random.choice([f'Cat_{i}' for i in range(50)],
   n_samples),
    'with_rare_categories': np.random.choice(
        ['Common1', 'Common2', 'Common3'] + [f'Rare_{i}' for i in range
   (20)],
        n_samples, p=[0.3, 0.3, 0.3] + [0.1/20]*20
    )
}

# Date/time features
base_date = datetime(2020, 1, 1)
date_data = [base_date + timedelta(days=i) for i in range(n_samples)]
# Add some missing dates
date_missing_indices = np.random.choice(n_samples, 30, replace=False)
for idx in date_missing_indices:
    date_data[idx] = pd.NaT

# Create target variable based on features
target = (
    numerical_data['small_range'] * 10 +
    numerical_data['large_range'] / 1000 +
    (numerical_data['with_outliers'] > 0).astype(int) +
    (categorical_data['low_cardinality'] == 'A').astype(int)
) > 2

# Combine into DataFrame
data = pd.DataFrame(numerical_data)
for col, values in categorical_data.items():
    data[col] = values
data['date_feature'] = date_data
data['target'] = target.astype(int)

print( Complex dataset created with various data issues )
print(f Dataset shape: {data.shape} )
```

```
print(f Missing values per column: )
print(data.isnull().sum())

# Counter-intuitive result 1: Imputation can introduce bias
print( \n=== COUNTER-INTUITIVE: Imputation Can Introduce Bias === )

# Test different imputation strategies
X_impute = data[['small_range', 'large_range', 'with_outliers', '
    with_missing']].copy()
y_impute = data['target']

# Split data
X_train_imp, X_test_imp, y_train_imp, y_test_imp = train_test_split(
    X_impute, y_impute, test_size=0.3, random_state=42
)

imputers = {
    'Mean': SimpleImputer(strategy='mean'),
    'Median': SimpleImputer(strategy='median'),
    'KNN': KNNImputer(n_neighbors=5)
}

results_imputation = {}

print( Testing different imputation strategies: )
for name, imputer in imputers.items():
    # Fit imputer on training data
    X_train_imputed = imputer.fit_transform(X_train_imp)
    X_test_imputed = imputer.transform(X_test_imp)

    # Train model
    model = LogisticRegression(random_state=42, max_iter=1000)
    model.fit(X_train_imputed, y_train_imp)

    # Evaluate
    y_pred = model.predict(X_test_imputed)
    accuracy = accuracy_score(y_test_imp, y_pred)

    results_imputation[name] = accuracy
    print(f   {name} imputation: {accuracy:.4f} )

# Show the bias introduced by different imputation methods
print(f \nOriginal data statistics for 'with_missing' column: )
print(f   Mean: {X_train_imp['with_missing'].mean():.4f} )
print(f   Std: {X_train_imp['with_missing'].std():.4f} )

# Calculate imputed values
for name, imputer in imputers.items():
    if name == 'Mean':
        imputed_values = imputer.fit_transform(X_train_imp[:, 3].reshape
   (-1, 1))
        print(f   {name} imputed value: {imputed_values[0][0]:.4f} )
    elif name == 'Median':
        imputed_values = imputer.fit_transform(X_train_imp[:, 3].reshape
```

```
    (-1, 1))
        print(f   {name} imputed value: {imputed_values[0][0]:.4f} )

print( \nCounter-intuitive: Imputation can introduce systematic bias! )
print( Reasons: )
print( 1. Mean imputation reduces variance )
print( 2. Imputed values may not represent true missing data distribution )
print( 3. Different imputation methods can lead to different model performances )
print( 4. Imputation assumes data is missing at random (often not true) )

# Counter-intuitive result 2: One-hot encoding high cardinality features can hurt performance
print( \n=== COUNTER-INTUITIVE: One-Hot Encoding High Cardinality Features === )

# Test categorical encoding strategies
X_cat = data[['low_cardinality', 'high_cardinality', 'with_rare_categories']].copy()
y_cat = data['target']

X_train_cat, X_test_cat, y_train_cat, y_test_cat = train_test_split(
    X_cat, y_cat, test_size=0.3, random_state=42
)

# Test different encoding approaches
print( Testing categorical encoding strategies: )

# Approach 1: One-hot encode everything
ohe_all = OneHotEncoder(sparse=False, handle_unknown='ignore')
X_train_ohe = ohe_all.fit_transform(X_train_cat)
X_test_ohe = ohe_all.transform(X_test_cat)

model_ohe = RandomForestClassifier(random_state=42, n_estimators=50)
model_ohe.fit(X_train_ohe, y_train_cat)
accuracy_ohe = model_ohe.score(X_test_ohe, y_test_cat)
print(f   One-hot all features: {accuracy_ohe:.4f} )
print(f   Feature dimensions after OHE: {X_train_ohe.shape[1]} )

# Approach 2: Handle high cardinality differently
from sklearn.preprocessing import LabelEncoder

X_train_smart = X_train_cat.copy()
X_test_smart = X_test_cat.copy()

# Low cardinality: One-hot encode
ohe_low = OneHotEncoder(sparse=False, handle_unknown='ignore')
low_card_features = ['low_cardinality']
X_train_low = ohe_low.fit_transform(X_train_smart[low_card_features])
X_test_low = ohe_low.transform(X_test_smart[low_card_features])

# High cardinality: Label encode or target encode
le_high = LabelEncoder()
```

```
for col in ['high_cardinality', 'with_rare_categories']:
    X_train_smart[col] = le_high.fit_transform(X_train_smart[col].astype(str))
    X_test_smart[col] = le_high.transform(X_test_smart[col].astype(str))

# Combine features
X_train_smart_combined = np.column_stack([
    X_train_low,
    X_train_smart[['high_cardinality', 'with_rare_categories']].values
])
X_test_smart_combined = np.column_stack([
    X_test_low,
    X_test_smart[['high_cardinality', 'with_rare_categories']].values
])

model_smart = RandomForestClassifier(random_state=42, n_estimators=50)
model_smart.fit(X_train_smart_combined, y_train_cat)
accuracy_smart = model_smart.score(X_test_smart_combined, y_test_cat)

print(f"  Smart encoding: {accuracy_smart:.4f}")
print(f"  Feature dimensions after smart encoding: {X_train_smart_combined.shape[1]}")

print("\nCounter-intuitive: One-hot encoding high cardinality features can hurt performance!")
print("Reasons:")
print("1. Creates too many dimensions (curse of dimensionality)")
print("2. Increases computational complexity")
print("3. Can lead to overfitting")
print("4. Sparse features may not be informative")

# Counter-intuitive result 3: Feature engineering can decrease performance
print("\n=== COUNTER-INTUITIVE: Feature Engineering Can Decrease Performance ===")

# Test feature engineering impact
X_fe = data[['small_range', 'large_range', 'with_outliers']].copy()
y_fe = data['target']

X_train_fe, X_test_fe, y_train_fe, y_test_fe = train_test_split(
    X_fe, y_fe, test_size=0.3, random_state=42
)

# Baseline model
model_baseline = LogisticRegression(random_state=42, max_iter=1000)
model_baseline.fit(X_train_fe, y_train_fe)
baseline_accuracy = model_baseline.score(X_test_fe, y_test_fe)
print(f"Baseline accuracy: {baseline_accuracy:.4f}")

# Add polynomial features (can overfit)
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_train_poly = poly_features.fit_transform(X_train_fe)
X_test_poly = poly_features.transform(X_test_fe)

```

```
print(f Feature dimensions increased from {X_train_fe.shape[1]} to {X_train_poly.shape[1]} )

model_poly = LogisticRegression(random_state=42, max_iter=1000)
model_poly.fit(X_train_poly, y_train_fe)
poly_accuracy = model_poly.score(X_test_poly, y_test_fe)
print(f Polynomial features accuracy: {poly_accuracy:.4f} )

# Add interaction features that don't help
X_train_interact = X_train_fe.copy()
X_train_interact['useless_interaction'] = (
    X_train_fe['small_range'] * np.random.normal(0, 1, len(X_train_fe))
)
X_test_interact = X_test_fe.copy()
X_test_interact['useless_interaction'] = (
    X_test_fe['small_range'] * np.random.normal(0, 1, len(X_test_fe))
)

model_interact = LogisticRegression(random_state=42, max_iter=1000)
model_interact.fit(X_train_interact, y_train_fe)
interact_accuracy = model_interact.score(X_test_interact, y_test_fe)
print(f Useless interaction accuracy: {interact_accuracy:.4f} )

print( \nCounter-intuitive: Feature engineering can decrease performance! )
print( Reasons: )
print( 1. Overfitting to training data )
print( 2. Adding noise instead of signal )
print( 3. Increasing dimensionality without adding information )
print( 4. Creating correlated features that confuse the model )

# Counter-intuitive result 4: Scaling can make models worse
print( \n=== COUNTER-INTUITIVE: Scaling Can Make Models Worse === )

# Test scaling impact on different models
X_scale = data[['small_range', 'large_range', 'with_outliers']].copy()
y_scale = data['target']

X_train_scale, X_test_scale, y_train_scale, y_test_scale = train_test_split(
    X_scale, y_scale, test_size=0.3, random_state=42
)

models_scale = {
    'RandomForest': RandomForestClassifier(random_state=42, n_estimators=50),
    'LogisticRegression': LogisticRegression(random_state=42, max_iter=1000)
}

scalers_scale = {
    'No Scaling': None,
    'StandardScaler': StandardScaler(),
    'MinMaxScaler': MinMaxScaler(),
```

```
    'RobustScaler': RobustScaler()
}

results_scaling = {}

print( Testing scaling impact on different models: )
for model_name, model in models_scale.items():
    results_scaling[model_name] = {}

    for scaler_name, scaler in scalers_scale.items():
        if scaler is None:
            X_train_proc = X_train_scale
            X_test_proc = X_test_scale
        else:
            X_train_proc = scaler.fit_transform(X_train_scale)
            X_test_proc = scaler.transform(X_test_scale)

        model.fit(X_train_proc, y_train_scale)
        accuracy = model.score(X_test_proc, y_test_scale)
        results_scaling[model_name][scaler_name] = accuracy

# Display results
for model_name, scaler_results in results_scaling.items():
    print(f \n{model_name}: )
    for scaler_name, accuracy in scaler_results.items():
        print(f   {scaler_name}: {accuracy:.4f} )

print( \nCounter-intuitive: Scaling can make some models worse! )
print( Reasons: )
print( 1. Tree-based models are scale-invariant )
print( 2. Scaling can obscure feature importance )
print( 3. Different scalers handle outliers differently )
print( 4. Scaling parameters fitted on wrong data (data leakage) )

# Advanced preprocessing behavior analysis
class PreprocessingBehaviorAnalyzer:
       Analyze counter-intuitive preprocessing behaviors

    @staticmethod
    def analyze_missing_data_patterns(X, y):
           Analyze how different missing data patterns affect imputation


        # Create different missing data patterns
        patterns = {
            'MCAR': 'Missing Completely at Random',
            'MAR': 'Missing at Random',
            'MNAR': 'Missing Not at Random'
        }

        results = {}

        for pattern_name, description in patterns.items():
            print(f \nAnalyzing {pattern_name}: {description} )
```

```
            # Simulate missing data pattern
            X_missing = X.copy()

            if pattern_name == 'MCAR':
                # Random missing
                missing_mask = np.random.random(X.shape) < 0.1
                X_missing = np.where(missing_mask, np.nan, X_missing)

            elif pattern_name == 'MAR':
                # Missing related to other features
                missing_prob = 0.1 * (X[:, 0] > np.median(X[:, 0]))
                missing_mask = np.random.random(X.shape[0]) < missing_prob
                X_missing[missing_mask, 1] = np.nan

            elif pattern_name == 'MNAR':
                # Missing related to missing values themselves
                missing_mask = X[:, 1] > np.percentile(X[:, 1], 90)
                X_missing[missing_mask, 1] = np.nan

            # Test imputation
            imputer = SimpleImputer(strategy='mean')
            try:
                X_imputed = imputer.fit_transform(X_missing)

                # Check if imputation introduced bias
                original_mean = np.nanmean(X[:, 1])
                imputed_mean = np.mean(X_imputed[:, 1])
                bias = abs(original_mean - imputed_mean)

                results[pattern_name] = {
                    'bias': bias,
                    'description': description
                }

                print(f   Original mean: {original_mean:.4f} )
                print(f   Imputed mean: {imputed_mean:.4f} )
                print(f   Bias: {bias:.4f} )

            except Exception as e:
                print(f   Error: {e} )
                results[pattern_name] = {'error': str(e)}

        return results

    @staticmethod
    def demonstrate_feature_selection_pitfalls():
           Demonstrate how feature selection can go wrong

        # Create dataset where feature selection might remove important
    features
        np.random.seed(42)
        n_samples = 1000

```

```
        # Create features with different importance levels
        X_fs = np.column_stack([
            np.random.normal(0, 1, n_samples),      # Low importance
            np.random.normal(0, 1, n_samples),      # Low importance
            np.random.normal(0, 0.1, n_samples),    # Very low importance (but necessary interaction)
            np.random.normal(0, 1, n_samples)       # Medium importance
        ])

        # Target depends on interaction that individual feature selection might miss
        y_fs = ((X_fs[:, 0] * X_fs[:, 2]) + X_fs[:, 3] > 0).astype(int)

        print( Feature Selection Pitfall Demonstration: )
        print( Target depends on interaction between features 1 and 3 )
        print( But individually, feature 3 has low variance and might be removed )

        # Individual feature importance
        from sklearn.feature_selection import f_classif
        f_scores, p_values = f_classif(X_fs, y_fs)

        print( \nIndividual feature F-scores: )
        for i, (f_score, p_value) in enumerate(zip(f_scores, p_values)):
            print(f   Feature {i}: F-score={f_score:.4f}, p-value={p_value:.4f} )

        # This shows how univariate feature selection can miss important interactions
        print( \nCounter-intuitive: Univariate feature selection can remove important features! )
        print( Reason: It doesn't consider feature interactions )

# Run advanced analysis
print( \n=== ADVANCED PREPROCESSING BEHAVIOR ANALYSIS === )

# Analyze missing data patterns
X_missing_test, y_missing_test = make_classification(n_samples=500, n_features=5, random_state=42)
analyzer = PreprocessingBehaviorAnalyzer()
missing_analysis = analyzer.analyze_missing_data_patterns(X_missing_test, y_missing_test)

print( \nMissing Data Pattern Analysis: )
for pattern, results in missing_analysis.items():
    if 'error' not in results:
        print(f   {pattern}: Bias={results['bias']:.4f} )
    else:
        print(f   {pattern}: Error={results['error']} )

# Demonstrate feature selection pitfalls
analyzer.demonstrate_feature_selection_pitfalls()

# Demonstrate date feature engineering surprises
```

```
print( \n=== DATE FEATURE ENGINEERING SURPRISES === )

# Create date features and show common pitfalls
date_series = pd.date_range('2020-01-01', periods=1000, freq='D')

# Common date features
date_features = pd.DataFrame({
    'date': date_series,
    'year': date_series.year,
    'month': date_series.month,
    'day': date_series.day,
    'dayofweek': date_series.dayofweek,
    'quarter': date_series.quarter,
    'is_weekend': date_series.dayofweek.isin([5, 6]).astype(int)
})

print( Date feature engineering considerations: )
print( 1. Cyclical features (month, dayofweek) should use sine/cosine
    encoding )
print( 2. High cardinality (daily dates) can create too many features )
print( 3. Seasonal patterns might be better captured with continuous
    features )

# Show cyclical encoding
date_features['month_sin'] = np.sin(2 * np.pi * date_features['month'] /
    12)
date_features['month_cos'] = np.cos(2 * np.pi * date_features['month'] /
    12)
date_features['dayofweek_sin'] = np.sin(2 * np.pi * date_features['
    dayofweek'] / 7)
date_features['dayofweek_cos'] = np.cos(2 * np.pi * date_features['
    dayofweek'] / 7)

print(f \nCyclical encoding example: )
print(f January (1) encoded as: sin={date_features['month_sin'].iloc[0]:.4
    f}, cos={date_features['month_cos'].iloc[0]:.4f} )
print(f December (12) encoded as: sin={date_features['month_sin'].iloc
    [11]:.4f}, cos={date_features['month_cos'].iloc[11]:.4f} )

print( \nKey Counter-Intuitive Preprocessing Behaviors: )
print( 1. Imputation can introduce systematic bias )
print( 2. One-hot encoding high cardinality features can hurt performance
    )
print( 3. Feature engineering can decrease rather than increase
    performance )
print( 4. Scaling can make some models worse, not better )
print( 5. Different missing data patterns require different imputation
    strategies )
print( 6. Univariate feature selection can miss important interactions )
print( 7. Cyclical features need special encoding (sine/cosine) )
print( 8. Outlier handling methods can significantly impact results )
print( 9. Data leakage in preprocessing is easy to accidentally introduce
    )
```

```
print( 10. The 'best' preprocessing for one model may hurt another model )
```

Listing 51: Behavior 6.1.6: Data Preprocessing Surprises

```
# Counter-Intuitive: Model interpretation can be misleading
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.datasets import make_classification
from sklearn.inspection import permutation_importance
import matplotlib.pyplot as plt
import shap

# Create dataset with correlated and interacting features
np.random.seed(42)

# Create base features
n_samples = 1000
base_feature1 = np.random.normal(0, 1, n_samples)
base_feature2 = np.random.normal(0, 1, n_samples)

# Create correlated features
correlated_feature1 = base_feature1 + np.random.normal(0, 0.1, n_samples)
correlated_feature2 = base_feature2 + np.random.normal(0, 0.1, n_samples)

# Create interaction features
interaction_feature = base_feature1 * base_feature2

# Create target with complex relationships
target = (
    base_feature1 * 2 +  # Strong effect
    base_feature2 * 0.5 +  # Weak effect
    interaction_feature * 1.5 +  # Strong interaction
    np.random.normal(0, 0.5, n_samples)  # Noise
) > 0

# Combine into dataset
X_interpret = np.column_stack([
    base_feature1,
    base_feature2,
    correlated_feature1,
    correlated_feature2,
    interaction_feature
])

feature_names = [
    'base_feature1',
    'base_feature2',
    'correlated_feature1',
    'correlated_feature2',
    'interaction_feature'
```

```
]

y_interpret = target.astype(int)

print( Dataset created with correlated and interacting features )
print( True importance: base_feature1 > interaction_feature >
    base_feature2 )

# Split data
X_train_int, X_test_int, y_train_int, y_test_int = train_test_split(
    X_interpret, y_interpret, test_size=0.3, random_state=42
)

# Counter-intuitive result 1: Feature importance can be misleading with
    correlated features
print( \n=== COUNTER-INTUITIVE: Feature Importance with Correlated
    Features === )

# Train Random Forest
rf_interpret = RandomForestClassifier(n_estimators=100, random_state=42)
rf_interpret.fit(X_train_int, y_train_int)

# Get feature importances
rf_importances = rf_interpret.feature_importances_
importance_df = pd.DataFrame({
    'feature': feature_names,
    'importance': rf_importances
}).sort_values('importance', ascending=False)

print( Random Forest Feature Importances: )
print(importance_df)

print( \nCounter-intuitive observation: )
print( base_feature1 and correlated_feature1 should have similar
    importance )
print( But the importance is split between them! )
print( The true importance of base_feature1 is diluted. )

# Compare with permutation importance
print( \nPermutation Importances (more reliable): )
perm_importance = permutation_importance(
    rf_interpret, X_test_int, y_test_int, n_repeats=10, random_state=42
)

perm_importance_df = pd.DataFrame({
    'feature': feature_names,
    'perm_importance': perm_importance.importances_mean,
    'perm_std': perm_importance.importances_std
}).sort_values('perm_importance', ascending=False)

print(perm_importance_df)

# Visualize the difference
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
```

```

ax1.barh(importance_df['feature'], importance_df['importance'])
ax1.set_xlabel('Feature Importance')
ax1.set_title('Random Forest Feature Importances\n(Split among correlated features)')

ax2.barh(perm_importance_df['feature'], perm_importance_df['perm_importance'])
ax2.set_xlabel('Permutation Importance')
ax2.set_title('Permutation Importances\n(Shows true predictive power)')

plt.tight_layout()
plt.show()

# Counter-intuitive result 2: Linear model coefficients can be misleading
print( \n=== COUNTER-INTUITIVE: Linear Model Coefficients === )

# Train Logistic Regression
lr_interpret = LogisticRegression(random_state=42, max_iter=1000)
lr_interpret.fit(X_train_int, y_train_int)

# Get coefficients
lr_coefficients = lr_interpret.coef_[0]
coef_df = pd.DataFrame({
    'feature': feature_names,
    'coefficient': lr_coefficients,
    'abs_coefficient': np.abs(lr_coefficients)
}).sort_values('abs_coefficient', ascending=False)

print( Logistic Regression Coefficients: )
print(coef_df)

print( \nCounter-intuitive observation: )
print( Coefficients don't reflect true feature importance due to scale! )
print( Features with larger scales appear more important. )
print( Correlated features can have opposite signs due to multicollinearity. )

# Compare with standardized coefficients
from sklearn.preprocessing import StandardScaler

scaler_int = StandardScaler()
X_train_scaled = scaler_int.fit_transform(X_train_int)
X_test_scaled = scaler_int.transform(X_test_int)

lr_scaled = LogisticRegression(random_state=42, max_iter=1000)
lr_scaled.fit(X_train_scaled, y_train_int)

lr_coefficients_scaled = lr_scaled.coef_[0]
coef_scaled_df = pd.DataFrame({
    'feature': feature_names,
    'coefficient': lr_coefficients_scaled,
    'abs_coefficient': np.abs(lr_coefficients_scaled)
}).sort_values('abs_coefficient', ascending=False)
```

```
152
153 print( \nStandardized Logistic Regression Coefficients: )
154 print(coef_scaled_df)
155
156 print( \nCounter-intuitive: Even standardized coefficients can be
    misleading! )
157 print( Reason: Multicollinearity still affects coefficient interpretation
    )
158
159 # Counter-intuitive result 3: SHAP values can be counter-intuitive
160 print( \n=== COUNTER-INTUITIVE: SHAP Values === )
161
162 try:
163     # Calculate SHAP values
164     explainer = shap.TreeExplainer(rf_interpret)
165     shap_values = explainer.shap_values(X_test_int[:100])  # Use subset
    for speed
166
167     # Summary plot
168     shap.summary_plot(shap_values[1], X_test_int[:100], feature_names=
    feature_names, show=False)
169     plt.title('SHAP Summary Plot')
170     plt.show()
171
172     # Mean absolute SHAP values
173     mean_shap = np.mean(np.abs(shap_values[1]), axis=0)
174     shap_df = pd.DataFrame({
175         'feature': feature_names,
176         'mean_abs_shap': mean_shap
177     }).sort_values('mean_abs_shap', ascending=False)
178
179     print( SHAP Feature Importance: )
180     print(shap_df)
181
182     print( \nCounter-intuitive: SHAP values show individual contributions
    )
183     print( But global importance might differ from individual explanations
    )
184
185 except ImportError:
186     print( SHAP not installed. Install with: pip install shap )
187     print( SHAP provides game-theoretic approach to explain model
    predictions )
188
189 # Counter-intuitive result 4: Partial dependence plots can be misleading
190 print( \n=== COUNTER-INTUITIVE: Partial Dependence Plots === )
191
192 from sklearn.inspection import partial_dependence, plot_partial_dependence
193
194 # Create PDP for features
195 try:
196     features_pdp = [0, 1, 4]  # base_feature1, base_feature2,
    interaction_feature
197     plot_partial_dependence(
```

```
        rf_interpret, X_train_int, features_pdp,
        feature_names=feature_names, random_state=42
    )
    plt.suptitle('Partial Dependence Plots')
    plt.show()

    print( Partial Dependence Plots created )
    print( Counter-intuitive: PDPs show marginal effects )
    print( But ignore interactions between features )
    print( For interaction_feature, PDP might show little effect )
    print( Even though it's actually very important in combination with
   others )

except Exception as e:
    print(f Error creating PDPs: {e} )

# Counter-intuitive result 5: Model confidence doesn't equal accuracy
print( \n=== COUNTER-INTUITIVE: Model Confidence vs Accuracy === )

# Get prediction probabilities
y_proba = rf_interpret.predict_proba(X_test_int)

# Separate high and low confidence predictions
high_confidence_mask = np.max(y_proba, axis=1) > 0.8
low_confidence_mask = np.max(y_proba, axis=1) <= 0.8

y_pred = rf_interpret.predict(X_test_int)

high_conf_accuracy = accuracy_score(
    y_test_int[high_confidence_mask],
    y_pred[high_confidence_mask]
)
low_conf_accuracy = accuracy_score(
    y_test_int[low_confidence_mask],
    y_pred[low_confidence_mask]
)

print(f High confidence predictions (>{0.8*100}%): )
print(f   Count: {np.sum(high_confidence_mask)} )
print(f   Accuracy: {high_conf_accuracy:.4f} )

print(f Low confidence predictions (   {0.8*100}%): )
print(f   Count: {np.sum(low_confidence_mask)} )
print(f   Accuracy: {low_conf_accuracy:.4f} )

print( \nCounter-intuitive: High model confidence doesn't guarantee high
   accuracy! )
print( Reasons: )
print( 1. Model can be confidently wrong )
print( 2. Calibration issues )
print( 3. Distribution shift between training and test data )
print( 4. Model overfitting leading to false confidence )

# Advanced interpretation analysis
```

```
class ModelInterpretationAnalyzer:
        Analyze counter-intuitive model interpretation behaviors

    @staticmethod
    def analyze_feature_interaction_effects(model, X, y, feature_pairs):
            Analyze how feature interactions affect interpretation

        # Train model
        model.fit(X, y)

        # Individual feature importance
        if hasattr(model, 'feature_importances_'):
            individual_importance = model.feature_importances_
        else:
            # Use permutation importance
            perm_imp = permutation_importance(model, X, y, n_repeats=5,
    random_state=42)
            individual_importance = perm_imp.importances_mean

        # Create interaction features and retrain
        X_with_interactions = X.copy()
        interaction_names = []

        for i, (f1, f2) in enumerate(feature_pairs):
            interaction_name = f interaction_{f1}_{f2}
            X_with_interactions = np.column_stack([
                X_with_interactions,
                X[:, f1] * X[:, f2]
            ])
            interaction_names.append(interaction_name)

        # Retrain with interactions
        model_interactions = type(model)(**model.get_params())
        model_interactions.fit(X_with_interactions, y)

        # Compare importances
        if hasattr(model_interactions, 'feature_importances_'):
            interaction_importance = model_interactions.
    feature_importances_
        else:
            perm_imp_int = permutation_importance(
                model_interactions, X_with_interactions, y, n_repeats=5,
    random_state=42
            )
            interaction_importance = perm_imp_int.importances_mean

        return {
            'individual_importance': individual_importance,
            'interaction_importance': interaction_importance,
            'interaction_names': interaction_names
        }

    @staticmethod
    def demonstrate_calibration_issues():
```

```
        Demonstrate model calibration problems

    # Create dataset where model confidence is misleading
    np.random.seed(42)
    X_cal, y_cal = make_classification(
        n_samples=1000, n_features=10, n_informative=5,
        n_redundant=2, flip_y=0.3, random_state=42
    )

    # Train overconfident model
    rf_overconfident = RandomForestClassifier(
        n_estimators=50, max_depth=20, random_state=42
    )
    rf_overconfident.fit(X_cal[:800], y_cal[:800])

    # Get probabilities
    y_proba_cal = rf_overconfident.predict_proba(X_cal[800:])
    y_pred_cal = rf_overconfident.predict(X_cal[800:])
    y_true_cal = y_cal[800:]

    # Analyze calibration
    from sklearn.calibration import calibration_curve

    fraction_of_positives, mean_predicted_value = calibration_curve(
        y_true_cal, y_proba_cal[:, 1], n_bins=10
    )

    # Plot calibration curve
    plt.figure(figsize=(8, 6))
    plt.plot(mean_predicted_value, fraction_of_positives,  s- , label= Random Forest )
    plt.plot([0, 1], [0, 1],  k: , label= Perfectly calibrated )
    plt.xlabel( Mean Predicted Probability )
    plt.ylabel( Fraction of Positives )
    plt.title( Calibration Plot - Overconfident Model )
    plt.legend()
    plt.grid(True)
    plt.show()

    print( Calibration Analysis: )
    print( Perfect calibration: points should follow diagonal line )
    print( Overconfident model: points below diagonal (predictions too high) )
    print( Underconfident model: points above diagonal (predictions too low) )

    # Calculate calibration error
    calibration_error = np.mean(np.abs(fraction_of_positives - mean_predicted_value))
    print(f Calibration error: {calibration_error:.4f} )

    print( \nCounter-intuitive: Well-calibrated models are often less confident! )
    print( Reason: They honestly represent uncertainty in predictions
```

```
    )

# Run advanced analysis
print( \n=== ADVANCED MODEL INTERPRETATION ANALYSIS === )

# Analyze feature interactions
feature_pairs = [(0, 1)]  # base_feature1 and base_feature2
analyzer = ModelInterpretationAnalyzer()

interaction_analysis = analyzer.analyze_feature_interaction_effects(
    RandomForestClassifier(n_estimators=50, random_state=42),
    X_train_int, y_train_int, feature_pairs
)

print( Feature Interaction Analysis: )
print( Individual feature importance may miss important interactions )
print( Adding explicit interaction features can reveal hidden
    relationships )

# Demonstrate calibration issues
print( \nDemonstrating calibration issues: )
analyzer.demonstrate_calibration_issues()

# Demonstrate model-specific interpretation surprises
print( \n=== MODEL-SPECIFIC INTERPRETATION SURPRISES === )

# Tree-based models
print( Tree-based models (Random Forest, XGBoost): )
print( 1. Feature importance splits importance among correlated features )
print( 2. Deep trees can memorize training data, making importance
    misleading )
print( 3. Individual trees can have very different feature importance )

# Linear models
print( \nLinear models (Logistic Regression, Linear SVM): )
print( 1. Coefficients reflect scale, not necessarily importance )
print( 2. Multicollinearity can flip coefficient signs )
print( 3. Regularization can shrink important features to zero )

# Neural networks
print( \nNeural networks: )
print( 1. Individual weights are not interpretable )
print( 2. Feature importance methods are approximations )
print( 3. Hidden layer activations are difficult to interpret )

# Ensemble models
print( \nEnsemble models: )
print( 1. Ensemble importance is average of individual model importances )
print( 2. Different ensemble members may disagree on feature importance )
print( 3. Voting can obscure individual model insights )

print( \nKey Counter-Intuitive Model Interpretation Behaviors: )
print( 1. Feature importance splits among correlated features )
print( 2. Linear coefficients reflect scale, not importance )
```

```
print( 3. SHAP values show individual contributions, not global importance )
print( 4. Partial dependence ignores feature interactions )
print( 5. High confidence doesn't equal high accuracy )
print( 6. Models can be confidently wrong )
print( 7. Calibration and discrimination are different concepts )
print( 8. Individual model insights may not apply to ensembles )
print( 9. Feature importance can change with data distribution )
print( 10. The most important feature may not be the most actionable )
```

Listing 52: Behavior 6.1.7: Model Interpretation Surprises

```
# Counter-Intuitive: Evaluation metrics can be misleading
import numpy as np
import pandas as pd
from sklearn.metrics import (accuracy_score, precision_score, recall_score, f1_score,
                             roc_auc_score, precision_recall_curve, roc_curve,
                             confusion_matrix, classification_report)
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt
from sklearn.calibration import calibration_curve

# Create imbalanced dataset to demonstrate metric surprises
np.random.seed(42)

# Highly imbalanced dataset (95% class 0, 5% class 1)
X_imb, y_imb = make_classification(
    n_samples=2000,
    n_features=10,
    n_classes=2,
    weights=[0.95, 0.05],  # 95% class 0, 5% class 1
    random_state=42
)

X_train_imb, X_test_imb, y_train_imb, y_test_imb = train_test_split(
    X_imb, y_imb, test_size=0.3, random_state=42
)

print( Highly imbalanced dataset created )
print(f Class distribution: {np.bincount(y_imb)} )
print(f Class 0: {np.sum(y_imb == 0) / len(y_imb) * 100:.1f}% )
print(f Class 1: {np.sum(y_imb == 1) / len(y_imb) * 100:.1f}% )

# Train models
rf_imb = RandomForestClassifier(random_state=42, n_estimators=100)
lr_imb = LogisticRegression(random_state=42, max_iter=1000)

rf_imb.fit(X_train_imb, y_train_imb)
lr_imb.fit(X_train_imb, y_train_imb)

```

```
# Get predictions
y_pred_rf = rf_imb.predict(X_test_imb)
y_pred_lr = lr_imb.predict(X_test_imb)

y_proba_rf = rf_imb.predict_proba(X_test_imb)[:, 1]
y_proba_lr = lr_imb.predict_proba(X_test_imb)[:, 1]

# Counter-intuitive result 1: Accuracy can be misleading for imbalanced data
print( \n=== COUNTER-INTUITIVE: Accuracy on Imbalanced Data === )

# Naive classifier that always predicts majority class
y_naive = np.zeros(len(y_test_imb))  # Always predict class 0

# Calculate metrics
accuracy_naive = accuracy_score(y_test_imb, y_naive)
accuracy_rf = accuracy_score(y_test_imb, y_pred_rf)
accuracy_lr = accuracy_score(y_test_imb, y_pred_lr)

print( Accuracy Comparison: )
print(f Naive classifier (always 0): {accuracy_naive:.4f} )
print(f Random Forest: {accuracy_rf:.4f} )
print(f Logistic Regression: {accuracy_lr:.4f} )

print( \nCounter-intuitive: High accuracy doesn't mean good performance! )
print( Naive classifier achieves 95% accuracy by always predicting majority class )
print( But it completely fails to identify the minority class (class 1) )

# Show detailed confusion matrices
print( \nConfusion Matrices: )

cm_naive = confusion_matrix(y_test_imb, y_naive)
cm_rf = confusion_matrix(y_test_imb, y_pred_rf)
cm_lr = confusion_matrix(y_test_imb, y_pred_lr)

print(f Naive classifier:\n{cm_naive} )
print(f Random Forest:\n{cm_rf} )
print(f Logistic Regression:\n{cm_lr} )

# Counter-intuitive result 2: F1-score can be misleading without context
print( \n=== COUNTER-INTUITIVE: F1-Score Without Context === )

# Calculate detailed metrics
def detailed_metrics(y_true, y_pred, y_proba=None, model_name=  ):
       Calculate comprehensive metrics

    metrics = {
        'accuracy': accuracy_score(y_true, y_pred),
        'precision': precision_score(y_true, y_pred, zero_division=0),
        'recall': recall_score(y_true, y_pred, zero_division=0),
        'f1': f1_score(y_true, y_pred, zero_division=0)
    }

```

```
    if y_proba is not None:
        metrics['auc'] = roc_auc_score(y_true, y_proba)

    # Per-class metrics
    precision_per_class = precision_score(y_true, y_pred, average=None, zero_division=0)
    recall_per_class = recall_score(y_true, y_pred, average=None, zero_division=0)

    metrics['class_0_precision'] = precision_per_class[0]
    metrics['class_1_precision'] = precision_per_class[1]
    metrics['class_0_recall'] = recall_per_class[0]
    metrics['class_1_recall'] = recall_per_class[1]

    return metrics

# Calculate metrics for all models
naive_metrics = detailed_metrics(y_test_imb, y_naive, model_name= Naive )
rf_metrics = detailed_metrics(y_test_imb, y_pred_rf, y_proba_rf, model_name= Random Forest )
lr_metrics = detailed_metrics(y_test_imb, y_pred_lr, y_proba_lr, model_name= Logistic Regression )

print( Detailed Metrics Comparison: )
models_metrics = [
    ( Naive , naive_metrics),
    ( Random Forest , rf_metrics),
    ( Logistic Regression , lr_metrics)
]

for model_name, metrics in models_metrics:
    print(f \n{model_name}: )
    print(f   Accuracy: {metrics['accuracy']:.4f} )
    print(f   F1-Score: {metrics['f1']:.4f} )
    print(f   Class 1 Precision: {metrics['class_1_precision']:.4f} )
    print(f   Class 1 Recall: {metrics['class_1_recall']:.4f} )
    if 'auc' in metrics:
        print(f   AUC: {metrics['auc']:.4f} )

print( \nCounter-intuitive: F1-score can hide poor minority class performance! )
print( A model with high overall F1 might have terrible recall for minority class )
print( Context matters: What's more important - precision or recall? )

# Counter-intuitive result 3: ROC-AUC can be misleading for imbalanced data
print( \n=== COUNTER-INTUITIVE: ROC-AUC for Imbalanced Data === )

# Create ROC curves
fpr_rf, tpr_rf, _ = roc_curve(y_test_imb, y_proba_rf)
fpr_lr, tpr_lr, _ = roc_curve(y_test_imb, y_proba_lr)
fpr_naive, tpr_naive, _ = roc_curve(y_test_imb, np.zeros_like(y_test_imb))

```

```
plt.figure(figsize=(10, 6))
plt.plot(fpr_rf, tpr_rf, label=f'Random Forest (AUC = {rf_metrics['auc']:.4f})')
plt.plot(fpr_lr, tpr_lr, label=f'Logistic Regression (AUC = {lr_metrics['auc']:.4f})')
plt.plot(fpr_naive, tpr_naive, label='Naive Classifier')
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves - Imbalanced Dataset')
plt.legend()
plt.grid(True)
plt.show()

print("ROC-AUC Analysis:")
print("High ROC-AUC doesn't mean good practical performance on imbalanced data")
print("ROC-AUC is insensitive to class distribution")
print("A model can have high AUC but poor precision on minority class")

# Counter-intuitive result 4: Precision-Recall curves tell a different story
print("\n=== COUNTER-INTUITIVE: Precision-Recall vs ROC ===")

# Create Precision-Recall curves
precision_rf, recall_rf, _ = precision_recall_curve(y_test_imb, y_proba_rf)
precision_lr, recall_lr, _ = precision_recall_curve(y_test_imb, y_proba_lr)

# Calculate average precision
from sklearn.metrics import average_precision_score
ap_rf = average_precision_score(y_test_imb, y_proba_rf)
ap_lr = average_precision_score(y_test_imb, y_proba_lr)

plt.figure(figsize=(10, 6))
plt.plot(recall_rf, precision_rf, label=f'Random Forest (AP = {ap_rf:.4f})')
plt.plot(recall_lr, precision_lr, label=f'Logistic Regression (AP = {ap_lr:.4f})')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves - Imbalanced Dataset')
plt.legend()
plt.grid(True)
plt.show()

print("Precision-Recall Analysis:")
print(f"Random Forest Average Precision: {ap_rf:.4f}")
print(f"Logistic Regression Average Precision: {ap_lr:.4f}")
print("AP is more informative than AUC for imbalanced datasets")
print("PR curves focus on minority class performance")

# Counter-intuitive result 5: Threshold selection can dramatically change
```

```
    results
print( \n=== COUNTER-INTUITIVE: Threshold Selection Impact === )

def evaluate_threshold(y_true, y_proba, threshold):
        Evaluate model at specific threshold
    y_pred_thresh = (y_proba >= threshold).astype(int)
    return {
        'threshold': threshold,
        'accuracy': accuracy_score(y_true, y_pred_thresh),
        'precision': precision_score(y_true, y_pred_thresh, zero_division=0),
        'recall': recall_score(y_true, y_pred_thresh, zero_division=0),
        'f1': f1_score(y_true, y_pred_thresh, zero_division=0)
    }

# Test different thresholds
thresholds = [0.1, 0.3, 0.5, 0.7, 0.9]
threshold_results = []

print( Impact of Different Classification Thresholds: )
print( Random Forest with probability threshold: )

for thresh in thresholds:
    results = evaluate_threshold(y_test_imb, y_proba_rf, thresh)
    threshold_results.append(results)
    print(f   Threshold {thresh}: Acc={results['accuracy']:.4f},
          f Precision={results['precision']:.4f}, Recall={results['recall']:.4f},
          f F1={results['f1']:.4f} )

# Find optimal thresholds
def find_optimal_threshold(y_true, y_proba, metric='f1'):
        Find threshold that optimizes specific metric
    thresholds_search = np.arange(0.01, 1.0, 0.01)
    best_score = 0
    best_threshold = 0.5

    for thresh in thresholds_search:
        y_pred_thresh = (y_proba >= thresh).astype(int)
        if metric == 'f1':
            score = f1_score(y_true, y_pred_thresh, zero_division=0)
        elif metric == 'precision':
            score = precision_score(y_true, y_pred_thresh, zero_division=0)
        elif metric == 'recall':
            score = recall_score(y_true, y_pred_thresh, zero_division=0)

        if score > best_score:
            best_score = score
            best_threshold = thresh

    return best_threshold, best_score

opt_f1_thresh, opt_f1_score = find_optimal_threshold(y_test_imb,
```

```
    y_proba_rf, 'f1')
opt_prec_thresh, opt_prec_score = find_optimal_threshold(y_test_imb,
    y_proba_rf, 'precision')
opt_rec_thresh, opt_rec_score = find_optimal_threshold(y_test_imb,
    y_proba_rf, 'recall')

print(f \nOptimal thresholds: )
print(f   F1-optimized: {opt_f1_thresh:.3f} (F1 = {opt_f1_score:.4f}) )
print(f   Precision-optimized: {opt_prec_thresh:.3f} (Precision = {
    opt_prec_score:.4f}) )
print(f   Recall-optimized: {opt_rec_thresh:.3f} (Recall = {opt_rec_score
    :.4f}) )

print( \nCounter-intuitive: Default 0.5 threshold is often suboptimal! )
print( The 'best' threshold depends on your specific requirements )
print( Optimizing for F1 might give different threshold than optimizing
    for precision )

# Counter-intuitive result 6: Calibration affects reliability of
    probabilities
print( \n=== COUNTER-INTUITIVE: Model Calibration === )

# Check calibration
fraction_positives_rf, mean_pred_rf = calibration_curve(
    y_test_imb, y_proba_rf, n_bins=10
)
fraction_positives_lr, mean_pred_lr = calibration_curve(
    y_test_imb, y_proba_lr, n_bins=10
)

plt.figure(figsize=(10, 6))
plt.plot(mean_pred_rf, fraction_positives_rf,  s- , label= Random Forest )
plt.plot(mean_pred_lr, fraction_positives_lr,  s- , label= Logistic
    Regression )
plt.plot([0, 1], [0, 1],  k: , label= Perfectly calibrated )
plt.xlabel( Mean Predicted Probability )
plt.ylabel( Fraction of Positives )
plt.title( Calibration Curves )
plt.legend()
plt.grid(True)
plt.show()

# Calculate calibration error
def calibration_error(fraction_positives, mean_predicted):
        Calculate mean calibration error
    return np.mean(np.abs(fraction_positives - mean_predicted))

cal_error_rf = calibration_error(fraction_positives_rf, mean_pred_rf)
cal_error_lr = calibration_error(fraction_positives_lr, mean_pred_lr)

print( Calibration Analysis: )
print(f Random Forest calibration error: {cal_error_rf:.4f} )
print(f Logistic Regression calibration error: {cal_error_lr:.4f} )

```

```
print( \nCounter-intuitive: Well-calibrated models may have lower discrimination! )
print( A perfectly calibrated model might be less confident but more reliable )
print( Overconfident models can have better metrics but poor real-world performance )

# Advanced evaluation analysis
class EvaluationAnalyzer:
       Analyze counter-intuitive evaluation behaviors

    @staticmethod
    def analyze_cost_sensitive_metrics(y_true, y_pred_proba, cost_matrix):
           Analyze cost-sensitive evaluation

        # Cost matrix: [TN, FP, FN, TP costs]
        tn_cost, fp_cost, fn_cost, tp_cost = cost_matrix

        # Calculate expected costs for different thresholds
        thresholds = np.arange(0.01, 1.0, 0.01)
        costs = []

        for thresh in thresholds:
            y_pred = (y_pred_proba >= thresh).astype(int)
            cm = confusion_matrix(y_true, y_pred)

            if cm.shape == (2, 2):
                tn, fp, fn, tp = cm.ravel()
            else:
                # Handle edge cases
                tn, fp, fn, tp = 0, 0, 0, 0
                if len(np.unique(y_true)) == 1:
                    if np.unique(y_true)[0] == 0:
                        tn = len(y_true)
                    else:
                        tp = len(y_true)

            total_cost = tn * tn_cost + fp * fp_cost + fn * fn_cost + tp * tp_cost
            costs.append(total_cost)

        optimal_idx = np.argmin(costs)
        optimal_threshold = thresholds[optimal_idx]
        optimal_cost = costs[optimal_idx]

        return {
            'optimal_threshold': optimal_threshold,
            'optimal_cost': optimal_cost,
            'costs': list(zip(thresholds, costs))
        }

    @staticmethod
    def demonstrate_metric_sensitivity():
           Demonstrate how small changes can dramatically affect metrics
```

```

        # Create scenario where small changes have big metric impacts
        np.random.seed(42)

        # Create predictions where one wrong prediction flips F1 score
        y_true_demo = np.array([0, 0, 0, 0, 1, 1, 1, 1])  # 4 of each class
        y_pred_good = np.array([0, 0, 0, 1, 1, 1, 1, 1])  # 1 FP, 0 FN
        y_pred_bad = np.array([0, 0, 1, 1, 0, 1, 1, 1])   # 2 FP, 1 FN (1 more wrong)

        metrics_good = {
            'accuracy': accuracy_score(y_true_demo, y_pred_good),
            'precision': precision_score(y_true_demo, y_pred_good, zero_division=0),
            'recall': recall_score(y_true_demo, y_pred_good, zero_division=0),
            'f1': f1_score(y_true_demo, y_pred_good, zero_division=0)
        }

        metrics_bad = {
            'accuracy': accuracy_score(y_true_demo, y_pred_bad),
            'precision': precision_score(y_true_demo, y_pred_bad, zero_division=0),
            'recall': recall_score(y_true_demo, y_pred_bad, zero_division=0),
            'f1': f1_score(y_true_demo, y_pred_bad, zero_division=0)
        }

        print( Metric Sensitivity Demonstration: )
        print( Small change (1 additional error) causing large metric differences: )
        print(f Good predictions: {y_pred_good} )
        print(f Bad predictions:  {y_pred_bad} )
        print( \nMetrics comparison: )
        for metric in metrics_good.keys():
            diff = metrics_bad[metric] - metrics_good[metric]
            print(f   {metric}: Good={metrics_good[metric]:.4f}, Bad={metrics_bad[metric]:.4f}, Diff={diff:.4f} )

        print( \nCounter-intuitive: Small absolute changes can cause large relative metric changes! )
        print( Especially problematic with small datasets or critical thresholds )

# Run advanced analysis
print( \n=== ADVANCED EVALUATION ANALYSIS === )

# Cost-sensitive analysis
print( Cost-Sensitive Evaluation: )
# Example: Medical diagnosis where false negatives are very costly
cost_matrix = [0, 10, 1000, 0]  # TN=0, FP=10, FN=1000, TP=0
analyzer = EvaluationAnalyzer()
```

```

cost_analysis = analyzer.analyze_cost_sensitive_metrics(
    y_test_imb, y_proba_rf, cost_matrix
)

print(f Optimal threshold for cost matrix {cost_matrix}: {cost_analysis['
    optimal_threshold']:.3f} )
print(f Minimum expected cost: {cost_analysis['optimal_cost']:.2f} )

# Demonstrate metric sensitivity
print( \nDemonstrating metric sensitivity: )
analyzer.demonstrate_metric_sensitivity()

# Demonstrate different evaluation scenarios
print( \n=== DIFFERENT EVALUATION SCENARIOS === )

# Scenario 1: Medical diagnosis (high cost of false negatives)
print( Scenario 1: Medical Diagnosis )
print( High cost of missing actual cases (false negatives) )
print( Prioritize recall over precision )

# Scenario 2: Spam detection (high cost of false positives)
print( \nScenario 2: Spam Detection )
print( High cost of marking legitimate emails as spam (false positives) )
print( Prioritize precision over recall )

# Scenario 3: Recommendation system (balanced approach)
print( \nScenario 3: Recommendation System )
print( Balanced cost of false positives and false negatives )
print( Optimize for F1-score or AUC )

print( \nKey Counter-Intuitive Evaluation Behaviors: )
print( 1. Accuracy is misleading for imbalanced datasets )
print( 2. F1-score can hide poor performance on minority class )
print( 3. ROC-AUC can be overly optimistic for imbalanced data )
print( 4. Precision-Recall curves are more informative for imbalanced data
    )
print( 5. Default 0.5 threshold is often suboptimal )
print( 6. Model calibration affects reliability of probability estimates )
print( 7. Small changes in predictions can cause large metric fluctuations
    )
print( 8. The 'best' metric depends on the specific business context )
print( 9. Cost-sensitive evaluation can reveal different optimal models )
print( 10. Cross-validation scores can be misleading without proper
    stratification )
```

Listing 53: Behavior 6.1.8: Evaluation Metric Surprises