# ****W9D6 -- ML Integration: Testing and Debugging - Error Handling & Edge Cases****

JTC Program: Tech Pathways Cohort: S25 Lesson Plan: ML Testing and Debugging Type: Lesson Plan Week / Day: W9D6 Version Date: 06/04/2025

## ****Focus Concepts****

* Understanding the importance of robust error handling in ML applications
* Learning to identify and handle common edge cases in ML workflows
* Implementing comprehensive testing strategies for ML systems
* Building defensive programming practices for data science projects
* Creating logging and monitoring systems for ML applications
* Developing professional-grade testing frameworks for ML pipelines

## ****Learning Objectives****

By the end of this session, fellows will be able to:

* Implement comprehensive error handling in ML applications using try/except blocks
* Identify and handle common edge cases in data preprocessing and model training
* Create robust data validation functions that catch data quality issues
* Build logging systems to monitor ML pipeline execution and debug issues
* Design and implement testing frameworks for ML applications
* Apply defensive programming principles to make ML systems more reliable
* Evaluate ML system robustness through systematic testing approaches

## ****Out-of-Scope Objectives****

* Advanced unit testing frameworks (pytest, unittest) - focus on basic concepts
* Production deployment testing strategies
* Advanced monitoring and alerting systems
* Performance testing and optimization
* Database-specific error handling
* Advanced debugging tools and profilers
* Cloud-specific error handling patterns

## ****Required Competencies****

* Understanding of basic machine learning concepts (from W9D1-W9D5)
* Familiarity with Python, NumPy, Pandas, and scikit-learn
* Experience with basic data preprocessing and model training
* Comfort with Python functions and control structures
* Understanding of basic programming concepts (variables, loops, conditionals)

## ****Technical Requirements****

* Python 3.x installed
* Jupyter Notebook or code editor
* Required libraries: NumPy, Pandas, scikit-learn, Matplotlib
* Access to sample datasets (will be provided)

## ****Prerequisites****

* Completion of W9D1: ML Foundations
* Completion of W9D2: Basic Regression
* Completion of W9D3: Model Evaluation
* Completion of W9D4: Feature Engineering
* Completion of W9D5: Time Series ML
* Understanding of the machine learning workflow

## ****Assigned Reading & Pre-Class Learning****

Estimated Time: 25 minutes

Resources:

* [Python Exception Handling Best Practices](https://realpython.com/python-exceptions-handling/) - Understanding try/except and error types - 15 minutes
* [Testing Machine Learning Code](https://eugeneyan.com/writing/testing-ml/) - Overview of ML testing strategies - 10 minutes

## ****Before-Class Mini Quiz Questions (5 questions)****

1. What is the primary purpose of using try/except blocks in Python?
   * A) To make code run faster
   * \*B) To handle errors gracefully and prevent program crashes
   * C) To organize code into logical sections
   * D) To create loops that repeat until successful
2. Which of these is considered an edge case in ML data preprocessing?
   * A) Having exactly 1000 rows of data
   * B) All features being numeric
   * \*C) Having completely empty datasets or all missing values
   * D) Using standard CSV file format
3. What should you do when your ML model encounters data it has never seen before?
   * A) Ignore the data and continue processing
   * B) Always use default predictions
   * \*C) Validate the data format and handle unexpected cases gracefully
   * D) Restart the entire training process
4. Why is logging important in ML applications?
   * A) It makes the code run faster
   * \*B) It helps track what happened when errors occur and aids debugging
   * C) It's required by Python for ML libraries
   * D) It automatically fixes errors in the code
5. What is data validation in the context of ML?
   * A) Checking if the model predictions are correct
   * B) Verifying that the algorithm is mathematically sound
   * \*C) Ensuring input data meets expected format, range, and quality requirements
   * D) Confirming that the dataset is large enough

## ****Key Terms****

* **Exception Handling**: Using try/except blocks to catch and handle errors gracefully
* **Edge Cases**: Unusual or extreme input conditions that can cause system failures
* **Data Validation**: Process of ensuring data meets quality and format requirements
* **Logging**: Recording events and information during program execution for debugging
* **Defensive Programming**: Writing code that anticipates and handles potential problems
* **Error Types**: Different categories of errors (TypeError, ValueError, FileNotFoundError, etc.)
* **Testing Framework**: Systematic approach to validating code functionality
* **Graceful Degradation**: System continuing to operate despite encountering errors
* **Data Quality Issues**: Problems like missing values, invalid formats, or outliers
* **Robustness**: System's ability to handle unexpected inputs and conditions
* **Debugging**: Process of finding and fixing errors in code
* **Monitoring**: Ongoing observation of system performance and behavior
* **Input Sanitization**: Cleaning and validating input data before processing
* **Error Recovery**: Strategies for continuing operation after encountering errors
* **Test Cases**: Specific scenarios designed to validate system behavior
* **Regression Testing**: Ensuring new changes don't break existing functionality
* **Integration Testing**: Testing how different components work together
* **Unit Testing**: Testing individual functions or components in isolation

## ****Lesson Schedule & Detailed Script****

### ****6:30 PM -- 6:45 PM: Interactive Check-In****

**Instructor Script:** "Welcome to Week 9, Day 6! We've spent this week building ML foundations, learning regression, evaluation, feature engineering, and time series forecasting. Today we're focusing on a critical but often overlooked aspect of ML: making our systems robust through proper error handling and testing. In the real world, data is messy, systems fail, and unexpected things happen. Today you'll learn how to build ML systems that can handle these challenges gracefully and continue operating reliably."

**Admin Tasks:**

* Take attendance
* Ensure everyone has required libraries installed (basic Python libraries)
* Check for any issues with previous assignments

**Prompting Questions:**

* "What kinds of errors or unexpected situations have you encountered when working with data or ML models?"
* "Why do you think it's important for ML systems to handle errors gracefully instead of just crashing?"

**Poll Questions:**

* "On a scale of 1-5, how comfortable are you with Python's try/except statements?"
* "Have you ever had a data science project fail because of bad data or unexpected inputs?"

### ****6:45 PM -- 7:05 PM: Session 1 -- Introduction to Error Handling in ML****

**Objective:** Understand the importance of error handling and learn basic try/except patterns for ML applications.

**Instructor Script:** "Let's start with the fundamentals of error handling. In ML applications, we deal with data from various sources, different file formats, and user inputs - all of which can introduce errors. Learning to handle these errors gracefully is essential for building professional ML systems."

#### ****Why Error Handling Matters in ML:****

1. **Data Quality Issues**: Real-world data is often messy, incomplete, or corrupted
2. **File Operations**: Reading datasets can fail due to missing files or permissions
3. **Model Training**: Models can fail to converge or encounter mathematical errors
4. **User Input**: Interactive systems must handle invalid user inputs
5. **System Resources**: Memory or disk space limitations can cause failures
6. **External Dependencies**: APIs and external services can be unavailable

# Example: Common ML errors without proper handling

import pandas as pd

import numpy as np

# This code will crash if the file doesn't exist

# df = pd.read\_csv("nonexistent\_data.csv") # FileNotFoundError

# This code will crash with invalid input

# result = 10 / 0 # ZeroDivisionError

# This code will crash with wrong data types

# text = "hello"

# number = 5

# combined = text + number # TypeError

print("These are examples of code that would crash without error handling")

#### ****Basic Error Handling Patterns:****

print("=== Basic Error Handling Examples ===\n")

# Example 1: File operations

def safe\_load\_data(filename):

"""Safely load a CSV file with error handling"""

try:

df = pd.read\_csv(filename)

print(f"Successfully loaded {len(df)} rows from {filename}")

return df

except FileNotFoundError:

print(f"Error: File '{filename}' not found!")

print("Creating sample data instead...")

# Return sample data as fallback

return pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

except PermissionError:

print(f"Error: Permission denied for file '{filename}'")

return None

except Exception as e:

print(f"Unexpected error loading {filename}: {e}")

return None

# Test the function

sample\_df = safe\_load\_data("sample\_data.csv") # This file doesn't exist

print(f"Loaded DataFrame shape: {sample\_df.shape}\n")

# Example 2: Mathematical operations

def safe\_divide(a, b):

"""Safely divide two numbers"""

try:

result = a / b

return result

except ZeroDivisionError:

print("Warning: Division by zero! Returning 0 instead.")

return 0

except TypeError:

print("Error: Both inputs must be numbers!")

return None

# Test the function

print("Safe division examples:")

print(f"10 / 2 = {safe\_divide(10, 2)}")

print(f"10 / 0 = {safe\_divide(10, 0)}")

print(f"10 / 'hello' = {safe\_divide(10, 'hello')}")

print()

# Example 3: Data type validation

def safe\_convert\_to\_numeric(value):

"""Safely convert a value to numeric"""

try:

return float(value)

except (ValueError, TypeError):

print(f"Warning: Cannot convert '{value}' to number. Using 0 instead.")

return 0

# Test the function

print("Safe conversion examples:")

test\_values = ["123", "45.67", "hello", None, [1, 2, 3]]

for val in test\_values:

converted = safe\_convert\_to\_numeric(val)

print(f"'{val}' -> {converted}")

#### ****Error Types Common in ML:****

print("\n=== Common ML Error Types ===\n")

# Demonstrate different error types

def demonstrate\_error\_types():

"""Show common error types in ML contexts"""

# 1. FileNotFoundError - Missing data files

print("1. FileNotFoundError:")

try:

pd.read\_csv("missing\_dataset.csv")

except FileNotFoundError as e:

print(f" Caught: {type(e).\_\_name\_\_}: {e}")

# 2. ValueError - Invalid values for operations

print("\n2. ValueError:")

try:

np.sqrt(-1) # Can't take square root of negative number

except ValueError as e:

print(f" Caught: {type(e).\_\_name\_\_}: {e}")

# 3. TypeError - Wrong data types

print("\n3. TypeError:")

try:

result = "hello" + 123 # Can't add string and int

except TypeError as e:

print(f" Caught: {type(e).\_\_name\_\_}: {e}")

# 4. KeyError - Missing dictionary keys (common with DataFrames)

print("\n4. KeyError:")

try:

df = pd.DataFrame({'A': [1, 2, 3]})

missing\_column = df['nonexistent\_column']

except KeyError as e:

print(f" Caught: {type(e).\_\_name\_\_}: {e}")

# 5. IndexError - Invalid array/list indices

print("\n5. IndexError:")

try:

my\_list = [1, 2, 3]

value = my\_list[10] # Index doesn't exist

except IndexError as e:

print(f" Caught: {type(e).\_\_name\_\_}: {e}")

demonstrate\_error\_types()

**Key Points About Error Handling:**

* Always anticipate what could go wrong with your data and operations
* Use specific exception types when possible (not just generic Exception)
* Provide meaningful error messages and fallback behavior
* Log errors for debugging purposes
* Consider whether to continue execution or stop when errors occur

### ****7:05 PM -- 7:30 PM: Session 2 -- Identifying and Handling Edge Cases****

**Objective:** Learn to identify common edge cases in ML workflows and implement robust handling strategies.

**Instructor Script:** "Edge cases are scenarios that occur at the extremes of your data or system's operating parameters. They're often rare but can cause significant problems if not handled properly. Let's explore common edge cases in ML and how to handle them."

#### ****Common ML Edge Cases:****

print("=== Common ML Edge Cases ===\n")

# Example 1: Empty or very small datasets

def handle\_small\_datasets(data):

"""Handle edge cases with dataset size"""

print(f"1. Dataset Size Edge Cases:")

if data is None:

print(" Error: No data provided!")

return None

if len(data) == 0:

print(" Warning: Empty dataset!")

return "Cannot process empty data"

if len(data) == 1:

print(" Warning: Only one data point - cannot train most models")

return "Insufficient data for training"

if len(data) < 10:

print(f" Warning: Very small dataset ({len(data)} points)")

print(" Consider collecting more data for reliable results")

return f"Dataset ready with {len(data)} points"

# Test with different scenarios

test\_datasets = [

None, # No data

[], # Empty list

[1], # Single point

[1, 2, 3], # Very small

list(range(50)) # Normal size

]

for i, dataset in enumerate(test\_datasets):

result = handle\_small\_datasets(dataset)

print(f" Test {i+1}: {result}\n")

# Example 2: Missing and invalid values

def clean\_and\_validate\_data(data\_series):

"""Handle missing and invalid values in data"""

print("2. Missing/Invalid Values Edge Cases:")

if data\_series is None or len(data\_series) == 0:

print(" No data to clean")

return []

original\_length = len(data\_series)

cleaned\_data = []

issues\_found = []

for i, value in enumerate(data\_series):

# Check for None/NaN values

if value is None or (isinstance(value, float) and np.isnan(value)):

issues\_found.append(f"Missing value at index {i}")

continue

# Check for infinite values

if isinstance(value, (int, float)) and np.isinf(value):

issues\_found.append(f"Infinite value at index {i}")

continue

# Check for extremely large values (potential outliers)

if isinstance(value, (int, float)) and abs(value) > 1e6:

issues\_found.append(f"Extreme value {value} at index {i}")

cleaned\_data.append(value)

print(f" Original data points: {original\_length}")

print(f" Cleaned data points: {len(cleaned\_data)}")

print(f" Issues found: {len(issues\_found)}")

if issues\_found:

print(" Issues detected:")

for issue in issues\_found[:5]: # Show first 5 issues

print(f" - {issue}")

if len(issues\_found) > 5:

print(f" ... and {len(issues\_found) - 5} more issues")

return cleaned\_data

# Test with problematic data

messy\_data = [1, 2, None, 4, float('inf'), -1e7, 7, float('nan'), 9, 10]

cleaned = clean\_and\_validate\_data(messy\_data)

print(f" Cleaned result: {cleaned[:5]}...\n")

# Example 3: Boundary values and extreme inputs

def validate\_input\_ranges(value, min\_val=0, max\_val=100, name="value"):

"""Validate inputs are within expected ranges"""

print(f"3. Boundary Value Validation for {name}:")

try:

# Convert to float if possible

if isinstance(value, str):

value = float(value)

# Check if it's a valid number

if not isinstance(value, (int, float)) or np.isnan(value):

return False, f"'{value}' is not a valid number"

# Check boundaries

if value < min\_val:

return False, f"{value} is below minimum ({min\_val})"

elif value > max\_val:

return False, f"{value} is above maximum ({max\_val})"

elif value == min\_val:

return True, f"{value} is at minimum boundary - check if this is expected"

elif value == max\_val:

return True, f"{value} is at maximum boundary - check if this is expected"

else:

return True, f"{value} is within valid range"

except (ValueError, TypeError) as e:

return False, f"Error validating {name}: {e}"

# Test boundary cases

test\_values = [-1, 0, 0.5, 50, 99.9, 100, 101, "50", "invalid", None]

for val in test\_values:

is\_valid, message = validate\_input\_ranges(val, 0, 100, "test\_value")

status = "✓" if is\_valid else "✗"

print(f" {status} {val}: {message}")

print()

# Example 4: Model-specific edge cases

def handle\_model\_training\_edge\_cases(X, y):

"""Handle edge cases during model training"""

print("4. Model Training Edge Cases:")

issues = []

# Check for sufficient data

if len(X) < 2:

issues.append("Insufficient samples for training (need at least 2)")

# Check for feature-target mismatch

if len(X) != len(y):

issues.append(f"Feature-target mismatch: {len(X)} samples vs {len(y)} targets")

# Check for constant features (no variation)

if len(X) > 0 and hasattr(X, 'std'):

zero\_variance\_features = (X.std() == 0).sum() if hasattr(X.std(), 'sum') else 0

if zero\_variance\_features > 0:

issues.append(f"{zero\_variance\_features} features have zero variance")

# Check for constant target (can't learn anything)

if len(set(y)) == 1:

issues.append("Target variable has only one unique value")

# Check for missing values

if hasattr(X, 'isna') and X.isna().any().any():

issues.append("Features contain missing values")

if any(pd.isna(val) for val in y):

issues.append("Target contains missing values")

print(f" Training data validation: {len(X)} samples, {len(y)} targets")

if issues:

print(" Issues found:")

for issue in issues:

print(f" - {issue}")

return False

else:

print(" ✓ Data appears suitable for training")

return True

# Test with different scenarios

print("Testing normal case:")

X\_normal = pd.DataFrame({'feature1': [1, 2, 3, 4], 'feature2': [5, 6, 7, 8]})

y\_normal = [10, 20, 30, 40]

handle\_model\_training\_edge\_cases(X\_normal, y\_normal)

print("\nTesting problematic case:")

X\_problem = pd.DataFrame({'feature1': [1, 1, 1, 1], 'feature2': [5, 6, 7, 8]}) # Constant feature

y\_problem = [10, 10, 10, 10] # Constant target

handle\_model\_training\_edge\_cases(X\_problem, y\_problem)

**Key Edge Cases to Always Consider:**

1. **Empty or minimal data**: No data, single data point, very small datasets
2. **Missing values**: None, NaN, empty strings, placeholder values
3. **Invalid formats**: Wrong data types, corrupted files, encoding issues
4. **Extreme values**: Very large/small numbers, infinite values, outliers
5. **Boundary conditions**: Minimum/maximum valid values, edge of acceptable ranges
6. **Resource limitations**: Memory constraints, processing time limits
7. **Concurrent access**: Multiple processes accessing the same data
8. **Network issues**: API timeouts, connection failures, slow responses

### ****7:30 PM -- 7:50 PM: Capstone Work Session****

**Activity:** Work on capstone project, applying error handling concepts learned so far.

### ****7:50 PM -- 8:00 PM: Break****

10-minute break

### ****8:00 PM -- 8:35 PM: Session 3 -- Building Robust ML Pipelines****

**Objective:** Learn to create comprehensive logging systems and build robust ML pipelines that can handle errors gracefully.

**Instructor Script:** "Now that we understand basic error handling and edge cases, let's build more sophisticated systems. We'll create logging systems to track what's happening in our ML pipelines and build frameworks that can recover from errors and continue operating."

#### ****Implementing Logging for ML Applications:****

import logging

from datetime import datetime

import json

# Example: Simple logging system for ML

class MLLogger:

"""Simple logging system for ML applications"""

def \_\_init\_\_(self, name="ML\_Pipeline"):

self.name = name

self.logs = []

self.start\_time = datetime.now()

def log(self, level, message, data=None):

"""Log a message with timestamp and optional data"""

timestamp = datetime.now().strftime("%H:%M:%S")

log\_entry = {

"time": timestamp,

"level": level,

"message": message,

"data": data

}

self.logs.append(log\_entry)

# Print for immediate feedback

print(f"[{timestamp}] {level}: {message}")

if data:

print(f" Data: {data}")

def info(self, message, data=None):

self.log("INFO", message, data)

def warning(self, message, data=None):

self.log("WARNING", message, data)

def error(self, message, data=None):

self.log("ERROR", message, data)

def success(self, message, data=None):

self.log("SUCCESS", message, data)

def get\_summary(self):

"""Get a summary of all logged events"""

total\_logs = len(self.logs)

levels = {}

for log in self.logs:

level = log['level']

levels[level] = levels.get(level, 0) + 1

return {

"total\_logs": total\_logs,

"levels": levels,

"duration": str(datetime.now() - self.start\_time)

}

# Create logger instance

logger = MLLogger("Data Processing Pipeline")

# Example usage

logger.info("Starting data processing pipeline")

logger.info("Loading dataset", {"filename": "sample\_data.csv"})

logger.warning("Found missing values", {"count": 5, "columns": ["age", "income"]})

logger.error("Model training failed", {"error": "Insufficient data"})

logger.success("Pipeline completed", {"processed\_samples": 1000})

print(f"\n=== Logging Summary ===")

summary = logger.get\_summary()

for key, value in summary.items():

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

#### ****Creating Robust Data Processing Functions:****

print("\n=== Robust Data Processing Pipeline ===\n")

class RobustDataProcessor:

"""A robust data processing class with comprehensive error handling"""

def \_\_init\_\_(self):

self.logger = MLLogger("DataProcessor")

self.processed\_data = None

self.processing\_stats = {

"rows\_processed": 0,

"errors\_encountered": 0,

"warnings\_issued": 0

}

def load\_data(self, data\_source):

"""Load data with multiple fallback strategies"""

self.logger.info(f"Attempting to load data from: {data\_source}")

try:

# Primary strategy: load from file

if isinstance(data\_source, str):

if data\_source.endswith('.csv'):

data = pd.read\_csv(data\_source)

else:

raise ValueError(f"Unsupported file format: {data\_source}")

# Alternative: data is already a DataFrame or list

elif isinstance(data\_source, pd.DataFrame):

data = data\_source.copy()

elif isinstance(data\_source, list):

data = pd.DataFrame(data\_source)

else:

raise TypeError(f"Unsupported data source type: {type(data\_source)}")

self.logger.success(f"Data loaded successfully",

{"shape": data.shape, "columns": list(data.columns)})

return data

except FileNotFoundError:

self.logger.error(f"File not found: {data\_source}")

# Fallback: create sample data

self.logger.info("Creating sample data as fallback")

sample\_data = pd.DataFrame({

'feature1': np.random.randn(100),

'feature2': np.random.randn(100),

'target': np.random.randn(100)

})

return sample\_data

except Exception as e:

self.logger.error(f"Failed to load data: {e}")

return None

def validate\_and\_clean\_data(self, data):

"""Comprehensive data validation and cleaning"""

if data is None:

self.logger.error("No data to validate")

return None

self.logger.info("Starting data validation and cleaning")

cleaned\_data = data.copy()

original\_shape = data.shape

# Check for completely empty DataFrame

if data.empty:

self.logger.error("DataFrame is completely empty")

return None

# Handle missing values

missing\_counts = data.isnull().sum()

if missing\_counts.sum() > 0:

self.logger.warning("Found missing values",

{"missing\_by\_column": missing\_counts.to\_dict()})

# Strategy: fill numeric columns with median, categorical with mode

for column in data.columns:

if data[column].dtype in ['int64', 'float64']:

median\_val = data[column].median()

cleaned\_data[column].fillna(median\_val, inplace=True)

self.logger.info(f"Filled missing values in {column} with median: {median\_val}")

else:

mode\_val = data[column].mode().iloc[0] if not data[column].mode().empty else 'UNKNOWN'

cleaned\_data[column].fillna(mode\_val, inplace=True)

self.logger.info(f"Filled missing values in {column} with mode: {mode\_val}")

# Check for infinite values

numeric\_columns = cleaned\_data.select\_dtypes(include=[np.number]).columns

for column in numeric\_columns:

inf\_count = np.isinf(cleaned\_data[column]).sum()

if inf\_count > 0:

self.logger.warning(f"Found {inf\_count} infinite values in {column}")

# Replace infinite values with column median

median\_val = cleaned\_data[column].replace([np.inf, -np.inf], np.nan).median()

cleaned\_data[column].replace([np.inf, -np.inf], median\_val, inplace=True)

# Check for duplicate rows

duplicate\_count = cleaned\_data.duplicated().sum()

if duplicate\_count > 0:

self.logger.warning(f"Found {duplicate\_count} duplicate rows")

cleaned\_data = cleaned\_data.drop\_duplicates()

# Validate data ranges (example: assuming features should be reasonable)

for column in numeric\_columns:

extreme\_values = (abs(cleaned\_data[column]) > 1e6).sum()

if extreme\_values > 0:

self.logger.warning(f"Found {extreme\_values} extreme values in {column}")

final\_shape = cleaned\_data.shape

self.logger.success("Data validation completed",

{"original\_shape": original\_shape,

"final\_shape": final\_shape,

"rows\_removed": original\_shape[0] - final\_shape[0]})

self.processing\_stats["rows\_processed"] = final\_shape[0]

return cleaned\_data

def process\_pipeline(self, data\_source):

"""Execute the complete processing pipeline"""

self.logger.info("Starting complete data processing pipeline")

try:

# Step 1: Load data

raw\_data = self.load\_data(data\_source)

if raw\_data is None:

raise ValueError("Failed to load data from any source")

# Step 2: Validate and clean

cleaned\_data = self.validate\_and\_clean\_data(raw\_data)

if cleaned\_data is None:

raise ValueError("Data validation failed")

# Step 3: Additional processing (feature engineering, etc.)

processed\_data = self.additional\_processing(cleaned\_data)

self.processed\_data = processed\_data

self.logger.success("Pipeline completed successfully")

return processed\_data

except Exception as e:

self.logger.error(f"Pipeline failed: {e}")

self.processing\_stats["errors\_encountered"] += 1

return None

def additional\_processing(self, data):

"""Additional processing steps with error handling"""

self.logger.info("Performing additional processing")

try:

processed = data.copy()

# Example: Create interaction features

numeric\_columns = processed.select\_dtypes(include=[np.number]).columns

if len(numeric\_columns) >= 2:

col1, col2 = numeric\_columns[0], numeric\_columns[1]

processed[f'{col1}\_x\_{col2}'] = processed[col1] \* processed[col2]

self.logger.info(f"Created interaction feature: {col1}\_x\_{col2}")

# Example: Normalize numeric features

for column in numeric\_columns:

if processed[column].std() > 0: # Avoid division by zero

processed[f'{column}\_normalized'] = (

processed[column] - processed[column].mean()

) / processed[column].std()

self.logger.info("Additional processing completed")

return processed

except Exception as e:

self.logger.error(f"Additional processing failed: {e}")

return data # Return original data if processing fails

# Test the robust processor

processor = RobustDataProcessor()

# Test with sample data

sample\_data = pd.DataFrame({

'feature1': [1, 2, np.nan, 4, 5, 1000000], # Has missing and extreme values

'feature2': [10, 20, 30, np.inf, 50, 60], # Has infinite value

'target': [100, 200, 300, 400, 500, 600]

})

print("Testing with problematic sample data:")

result = processor.process\_pipeline(sample\_data)

if result is not None:

print(f"\n✓ Processing successful! Final data shape: {result.shape}")

print(f"Columns: {list(result.columns)}")

else:

print("\n✗ Processing failed!")

print(f"\nProcessing Statistics: {processor.processing\_stats}")

#### ****Building a Complete Testing Framework:****

print("\n=== Complete ML Testing Framework ===\n")

class MLTestFramework:

"""Comprehensive testing framework for ML applications"""

def \_\_init\_\_(self, name="ML Test Suite"):

self.name = name

self.logger = MLLogger(f"TestFramework\_{name}")

self.test\_results = []

self.setup\_complete = False

def setup\_test\_environment(self):

"""Setup testing environment"""

try:

self.logger.info("Setting up test environment")

# Simulate environment checks

self.setup\_complete = True

self.logger.success("Test environment ready")

return True

except Exception as e:

self.logger.error(f"Setup failed: {e}")

return False

def run\_test(self, test\_name, test\_function, \*args, \*\*kwargs):

"""Execute a single test with error handling"""

if not self.setup\_complete:

self.logger.warning("Test environment not properly set up")

try:

self.logger.info(f"Running test: {test\_name}")

# Execute test function

result = test\_function(\*args, \*\*kwargs)

# Interpret results

if isinstance(result, bool):

passed = result

message = "Test passed" if passed else "Test failed"

elif isinstance(result, tuple) and len(result) == 2:

passed, message = result

else:

passed = True

message = f"Test completed: {result}"

# Record result

self.test\_results.append({

'name': test\_name,

'passed': passed,

'message': message

})

status = "✓ PASS" if passed else "✗ FAIL"

self.logger.info(f"{status}: {message}")

return passed

except Exception as e:

self.test\_results.append({

'name': test\_name,

'passed': False,

'message': f"Test error: {e}"

})

self.logger.error(f"Test '{test\_name}' failed with error: {e}")

return False

def test\_data\_quality(self, data):

"""Test data quality and format"""

if data is None:

return False, "Data is None"

if len(data) == 0:

return False, "Data is empty"

if hasattr(data, 'isnull'):

missing\_ratio = data.isnull().sum().sum() / (data.shape[0] \* data.shape[1])

if missing\_ratio > 0.5:

return False, f"Too many missing values: {missing\_ratio:.2%}"

return True, f"Data quality check passed: {len(data)} samples"

def test\_model\_training(self, X, y):

"""Test that model training works with given data"""

try:

from sklearn.linear\_model import LinearRegression

if len(X) != len(y):

return False, "Feature-target length mismatch"

if len(X) < 2:

return False, "Insufficient training data"

# Try to train a simple model

model = LinearRegression()

model.fit(X, y)

# Test prediction

prediction = model.predict(X[:1])

if np.isnan(prediction).any():

return False, "Model produces NaN predictions"

return True, "Model training test passed"

except Exception as e:

return False, f"Model training failed: {e}"

def test\_edge\_cases(self):

"""Test system behavior with edge cases"""

edge\_cases\_passed = 0

total\_edge\_cases = 0

# Test 1: Empty data

total\_edge\_cases += 1

try:

processor = RobustDataProcessor()

result = processor.validate\_and\_clean\_data(pd.DataFrame())

if result is None: # Expected behavior

edge\_cases\_passed += 1

except:

pass

# Test 2: Single row data

total\_edge\_cases += 1

try:

single\_row = pd.DataFrame({'A': [1], 'B': [2]})

processor = RobustDataProcessor()

result = processor.validate\_and\_clean\_data(single\_row)

if result is not None and len(result) == 1:

edge\_cases\_passed += 1

except:

pass

# Test 3: All missing values

total\_edge\_cases += 1

try:

all\_missing = pd.DataFrame({'A': [np.nan, np.nan], 'B': [np.nan, np.nan]})

processor = RobustDataProcessor()

result = processor.validate\_and\_clean\_data(all\_missing)

# Should handle gracefully (fill with defaults)

if result is not None:

edge\_cases\_passed += 1

except:

pass

success\_rate = edge\_cases\_passed / total\_edge\_cases

if success\_rate >= 0.8:

return True, f"Edge cases handled well: {edge\_cases\_passed}/{total\_edge\_cases} passed"

else:

return False, f"Too many edge case failures: {edge\_cases\_passed}/{total\_edge\_cases} passed"

def run\_comprehensive\_test\_suite(self):

"""Run all tests in the framework"""

self.logger.info(f"Starting comprehensive test suite: {self.name}")

# Setup

if not self.setup\_test\_environment():

self.logger.error("Cannot proceed without proper setup")

return False

# Generate test data

test\_data = pd.DataFrame({

'feature1': np.random.randn(50),

'feature2': np.random.randn(50),

'target': np.random.randn(50)

})

# Run tests

tests = [

("Data Quality Check", self.test\_data\_quality, test\_data),

("Model Training Test", self.test\_model\_training,

test\_data[['feature1', 'feature2']], test\_data['target']),

("Edge Cases Test", self.test\_edge\_cases),

]

passed\_tests = 0

for test\_name, test\_func, \*test\_args in tests:

if self.run\_test(test\_name, test\_func, \*test\_args):

passed\_tests += 1

# Generate report

self.generate\_test\_report()

return passed\_tests == len(tests)

def generate\_test\_report(self):

"""Generate comprehensive test report"""

print(f"\n{'='\*50}")

print(f"TEST REPORT: {self.name}")

print(f"{'='\*50}")

total\_tests = len(self.test\_results)

passed\_tests = sum(1 for result in self.test\_results if result['passed'])

failed\_tests = total\_tests - passed\_tests

print(f"Total Tests: {total\_tests}")

print(f"Passed: {passed\_tests}")

print(f"Failed: {failed\_tests}")

print(f"Success Rate: {(passed\_tests/total\_tests)\*100:.1f}%")

if failed\_tests > 0:

print(f"\nFAILED TESTS:")

for result in self.test\_results:

if not result['passed']:

print(f" ✗ {result['name']}: {result['message']}")

print(f"\nALL TEST RESULTS:")

for result in self.test\_results:

status = "✓" if result['passed'] else "✗"

print(f" {status} {result['name']}: {result['message']}")

# Create and run the test framework

test\_framework = MLTestFramework("ML Pipeline Tests")

success = test\_framework.run\_comprehensive\_test\_suite()

print(f"\n{'='\*60}")

print(f"OVERALL TEST RESULT: {'SUCCESS' if success else 'NEEDS ATTENTION'}")

print(f"{'='\*60}")

**Key Points About Building Robust ML Pipelines:**

1. **Comprehensive Logging**: Track all operations, errors, and warnings
2. **Graceful Degradation**: System continues operating even when parts fail
3. **Multiple Fallback Strategies**: Have backup plans when primary approaches fail
4. **Systematic Testing**: Regularly verify system behavior under various conditions
5. **Error Recovery**: Implement strategies to recover from common failures
6. **Monitoring and Alerting**: Track system health and performance over time

### ****8:35 PM -- 9:05 PM: Session 4 -- Advanced Testing Strategies****

**Objective:** Learn advanced testing concepts including integration testing, performance testing, and building comprehensive test suites.

**Instructor Script:** "Now let's explore more advanced testing strategies. We'll learn how to test complete ML workflows, handle performance requirements, and build test suites that can catch problems before they reach production."

#### ****Integration Testing for ML Pipelines:****

print("=== Integration Testing for ML Pipelines ===\n")

class MLIntegrationTester:

"""Advanced integration testing for complete ML workflows"""

def \_\_init\_\_(self):

self.logger = MLLogger("IntegrationTester")

self.test\_scenarios = []

def test\_complete\_workflow(self, data\_source, expected\_output\_shape=None):

"""Test a complete ML workflow from data loading to prediction"""

self.logger.info("Starting complete workflow integration test")

workflow\_steps = []

try:

# Step 1: Data Loading

self.logger.info("Testing data loading step")

processor = RobustDataProcessor()

loaded\_data = processor.load\_data(data\_source)

if loaded\_data is None:

workflow\_steps.append(("Data Loading", False, "Failed to load data"))

return False, workflow\_steps

else:

workflow\_steps.append(("Data Loading", True, f"Loaded {loaded\_data.shape[0]} rows"))

# Step 2: Data Processing

self.logger.info("Testing data processing step")

processed\_data = processor.validate\_and\_clean\_data(loaded\_data)

if processed\_data is None:

workflow\_steps.append(("Data Processing", False, "Data processing failed"))

return False, workflow\_steps

else:

workflow\_steps.append(("Data Processing", True, f"Processed to {processed\_data.shape}"))

# Step 3: Feature Preparation

self.logger.info("Testing feature preparation")

# Ensure we have numeric features for modeling

numeric\_cols = processed\_data.select\_dtypes(include=[np.number]).columns

if len(numeric\_cols) < 2:

workflow\_steps.append(("Feature Preparation", False, "Insufficient numeric features"))

return False, workflow\_steps

# Prepare features and target

feature\_cols = numeric\_cols[:-1] # Use all but last column as features

target\_col = numeric\_cols[-1] # Use last column as target

X = processed\_data[feature\_cols]

y = processed\_data[target\_col]

workflow\_steps.append(("Feature Preparation", True, f"Prepared {len(feature\_cols)} features"))

# Step 4: Model Training

self.logger.info("Testing model training step")

from sklearn.linear\_model import LinearRegression

from sklearn.model\_selection import train\_test\_split

# Split data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Train model

model = LinearRegression()

model.fit(X\_train, y\_train)

workflow\_steps.append(("Model Training", True, f"Trained on {len(X\_train)} samples"))

# Step 5: Prediction

self.logger.info("Testing prediction step")

predictions = model.predict(X\_test)

# Validate predictions

if np.isnan(predictions).any():

workflow\_steps.append(("Prediction", False, "Model produced NaN predictions"))

return False, workflow\_steps

if np.isinf(predictions).any():

workflow\_steps.append(("Prediction", False, "Model produced infinite predictions"))

return False, workflow\_steps

workflow\_steps.append(("Prediction", True, f"Generated {len(predictions)} predictions"))

# Step 6: Evaluation

self.logger.info("Testing evaluation step")

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error

mse = mean\_squared\_error(y\_test, predictions)

mae = mean\_absolute\_error(y\_test, predictions)

# Check if metrics are reasonable

if np.isnan(mse) or np.isinf(mse):

workflow\_steps.append(("Evaluation", False, "Invalid evaluation metrics"))

return False, workflow\_steps

workflow\_steps.append(("Evaluation", True, f"MSE: {mse:.4f}, MAE: {mae:.4f}"))

self.logger.success("Complete workflow integration test passed")

return True, workflow\_steps

except Exception as e:

self.logger.error(f"Integration test failed: {e}")

workflow\_steps.append(("Error", False, str(e)))

return False, workflow\_steps

def test\_error\_recovery(self):

"""Test system's ability to recover from various error conditions"""

self.logger.info("Testing error recovery capabilities")

recovery\_tests = []

# Test 1: Recovery from corrupted data

try:

corrupted\_data = pd.DataFrame({

'feature1': [1, 'corrupted', 3, 4],

'feature2': [1, 2, 3, 'bad\_data'],

'target': [10, 20, 30, 40]

})

processor = RobustDataProcessor()

result = processor.process\_pipeline(corrupted\_data)

recovery\_tests.append({

'test': 'Corrupted Data Recovery',

'passed': result is not None,

'message': 'Handled corrupted data' if result is not None else 'Failed to handle corrupted data'

})

except Exception as e:

recovery\_tests.append({

'test': 'Corrupted Data Recovery',

'passed': False,

'message': f'Exception: {e}'

})

# Test 2: Recovery from extreme values

try:

extreme\_data = pd.DataFrame({

'feature1': [1e10, -1e10, 1, 2],

'feature2': [1, 2, 1e15, -1e15],

'target': [10, 20, 30, 40]

})

processor = RobustDataProcessor()

result = processor.process\_pipeline(extreme\_data)

recovery\_tests.append({

'test': 'Extreme Values Recovery',

'passed': result is not None,

'message': 'Handled extreme values' if result is not None else 'Failed to handle extreme values'

})

except Exception as e:

recovery\_tests.append({

'test': 'Extreme Values Recovery',

'passed': False,

'message': f'Exception: {e}'

})

# Test 3: Recovery from insufficient data

try:

minimal\_data = pd.DataFrame({

'feature1': [1],

'target': [10]

})

# This should be handled gracefully

success, workflow\_steps = self.test\_complete\_workflow(minimal\_data)

recovery\_tests.append({

'test': 'Insufficient Data Recovery',

'passed': True, # Should handle gracefully, even if workflow fails

'message': 'Gracefully handled insufficient data'

})

except Exception as e:

recovery\_tests.append({

'test': 'Insufficient Data Recovery',

'passed': False,

'message': f'Exception: {e}'

})

# Summary

passed\_recovery\_tests = sum(1 for test in recovery\_tests if test['passed'])

total\_recovery\_tests = len(recovery\_tests)

self.logger.info(f"Error recovery testing completed: {passed\_recovery\_tests}/{total\_recovery\_tests} passed")

for test in recovery\_tests:

status = "✓" if test['passed'] else "✗"

self.logger.info(f" {status} {test['test']}: {test['message']}")

return passed\_recovery\_tests / total\_recovery\_tests >= 0.7 # 70% success rate required

def performance\_stress\_test(self, max\_data\_size=1000):

"""Test system performance under stress conditions"""

self.logger.info(f"Starting performance stress test (max size: {max\_data\_size})")

performance\_results = []

data\_sizes = [10, 50, 100, 500, max\_data\_size]

for size in data\_sizes:

try:

import time

start\_time = time.time()

# Generate test data of specified size

test\_data = pd.DataFrame({

'feature1': np.random.randn(size),

'feature2': np.random.randn(size),

'feature3': np.random.randn(size),

'target': np.random.randn(size)

})

# Process the data

processor = RobustDataProcessor()

result = processor.process\_pipeline(test\_data)

end\_time = time.time()

processing\_time = end\_time - start\_time

performance\_results.append({

'size': size,

'success': result is not None,

'time': processing\_time,

'rate': size / processing\_time if processing\_time > 0 else float('inf')

})

self.logger.info(f"Size {size}: {processing\_time:.3f}s ({size/processing\_time:.1f} samples/sec)")

except Exception as e:

performance\_results.append({

'size': size,

'success': False,

'time': None,

'rate': 0,

'error': str(e)

})

self.logger.error(f"Performance test failed at size {size}: {e}")

# Analyze performance

successful\_tests = [r for r in performance\_results if r['success']]

if len(successful\_tests) > 0:

avg\_rate = np.mean([r['rate'] for r in successful\_tests])

self.logger.info(f"Average processing rate: {avg\_rate:.1f} samples/second")

# Check for performance degradation

if len(successful\_tests) > 1:

early\_rate = successful\_tests[0]['rate']

late\_rate = successful\_tests[-1]['rate']

degradation = (early\_rate - late\_rate) / early\_rate

if degradation > 0.5: # More than 50% slowdown

self.logger.warning(f"Significant performance degradation: {degradation:.1%}")

return False

success\_rate = len(successful\_tests) / len(performance\_results)

return success\_rate >= 0.8 # 80% of tests should succeed

# Run integration tests

integration\_tester = MLIntegrationTester()

print("=== Complete Workflow Test ===")

# Test with sample data

sample\_data = pd.DataFrame({

'feature1': np.random.randn(100),

'feature2': np.random.randn(100),

'feature3': np.random.randn(100),

'target': np.random.randn(100)

})

workflow\_success, workflow\_steps = integration\_tester.test\_complete\_workflow(sample\_data)

print(f"Workflow Success: {workflow\_success}")

print("Workflow Steps:")

for step\_name, step\_success, step\_message in workflow\_steps:

status = "✓" if step\_success else "✗"

print(f" {status} {step\_name}: {step\_message}")

print(f"\n=== Error Recovery Test ===")

recovery\_success = integration\_tester.test\_error\_recovery()

print(f"Error Recovery Success: {recovery\_success}")

print(f"\n=== Performance Stress Test ===")

performance\_success = integration\_tester.performance\_stress\_test(500)

print(f"Performance Test Success: {performance\_success}")

print(f"\n{'='\*60}")

overall\_success = workflow\_success and recovery\_success and performance\_success

print(f"OVERALL INTEGRATION TEST: {'SUCCESS' if overall\_success else 'NEEDS IMPROVEMENT'}")

print(f"{'='\*60}")

**Key Advanced Testing Concepts:**

1. **Integration Testing**: Testing complete workflows end-to-end
2. **Error Recovery Testing**: Verifying system can recover from failures
3. **Performance Testing**: Ensuring system meets speed and scalability requirements
4. **Stress Testing**: Testing system behavior under extreme conditions
5. **Regression Testing**: Ensuring new changes don't break existing functionality
6. **Automated Test Suites**: Running comprehensive tests automatically

### ****9:05 PM -- 9:25 PM: Breakout #2: Capstone Working Session****

**Activity:** Apply the testing and error handling concepts learned today to your capstone project. Focus on:

* Adding try/except blocks to handle potential errors
* Implementing data validation in your preprocessing steps
* Creating basic logging to track your model's performance
* Testing edge cases specific to your project's data

**Instructor Guidance:**

* Walk around and help students identify potential failure points in their projects
* Suggest specific error handling strategies based on their data types and models
* Help students implement basic logging to track their model training progress

### ****9:25 PM -- 9:30 PM: Wrap-Up & Final Questions****

**Instructor Script:** "Today we've learned how to make ML systems more robust and reliable through comprehensive error handling, edge case management, and systematic testing. These skills are crucial for building production-ready ML applications that can handle real-world data and unexpected situations gracefully."

**Review Key Points:**

* Error handling prevents crashes and provides better user experience
* Edge cases are common in real-world data and must be anticipated
* Logging helps with debugging and monitoring system health
* Systematic testing catches problems before they reach production
* Robust systems degrade gracefully rather than failing catastrophically

**Prompting Question:** "What's one error handling or testing strategy you'll implement in your current or future ML projects?"

**Next Steps:**

* Apply these concepts to your capstone projects
* Practice building robust data processing pipelines
* Consider how these principles apply to other programming contexts

## ****After-Class Quiz (5 questions)****

1. What is the main advantage of using try/except blocks in ML applications?
   * A) They make code run faster
   * B) They reduce memory usage
   * \*C) They prevent crashes and allow graceful error handling
   * D) They automatically fix data quality issues
2. Which of these is the best practice when handling missing values in a dataset?
   * A) Always delete rows with missing values
   * B) Always fill missing values with zeros
   * \*C) Choose an appropriate strategy based on the data type and context
   * D) Ignore missing values and proceed with training
3. What should a robust ML system do when it encounters data outside its expected range?
   * A) Crash immediately to alert the user
   * B) Ignore the problematic data silently
   * \*C) Log a warning and handle the data appropriately (clip, replace, or reject)
   * D) Automatically retrain the model
4. Why is logging important in ML applications?
   * A) It's required by scikit-learn
   * B) It makes models more accurate
   * \*C) It helps track system behavior and debug issues when they occur
   * D) It automatically prevents errors from happening
5. What is the purpose of integration testing in ML pipelines?
   * A) To test individual functions in isolation
   * B) To optimize model performance
   * \*C) To ensure all components work together correctly in the complete workflow
   * D) To validate mathematical formulas used in algorithms

## ****Homework Assignment****

**Assignment: Robust ML Pipeline Implementation**

Create a robust version of a simple ML pipeline that includes:

1. **Error Handling** (20 points):
   * Implement try/except blocks for data loading
   * Handle common data processing errors gracefully
   * Provide meaningful error messages
2. **Edge Case Management** (25 points):
   * Handle empty datasets
   * Manage missing values appropriately
   * Deal with extreme or invalid values
   * Test with boundary conditions
3. **Logging System** (20 points):
   * Implement basic logging for major pipeline steps
   * Log warnings for data quality issues
   * Track processing statistics
4. **Testing Framework** (25 points):
   * Create at least 5 test cases for your pipeline
   * Test both normal and edge case scenarios
   * Include performance or stress testing
5. **Documentation** (10 points):
   * Document your error handling strategies
   * Explain your edge case management approach
   * Provide examples of how to use your robust pipeline

**Deliverables:**

* Python script with robust ML pipeline
* Test suite that validates your pipeline
* README explaining your approach and how to run the code
* Brief report (1-2 pages) discussing the errors and edge cases you handled

**Due Date:** Next class session

**Evaluation Criteria:**

* Code quality and error handling implementation
* Comprehensive edge case coverage
* Effective logging and monitoring
* Thorough testing approach
* Clear documentation and explanations

## ****Additional Resources****

**Recommended Reading:**

* "Effective Python" by Brett Slatkin - Chapters on exceptions and testing
* "Clean Code" by Robert Martin - Principles of defensive programming
* "Building Machine Learning Powered Applications" by Emmanuel Ameisen

**Online Resources:**

* Python Exception Handling Documentation
* scikit-learn Best Practices Guide
* ML Testing Guidelines (Google's ML Testing Best Practices)

**Tools to Explore:**

* pytest for advanced testing
* logging module for production logging
* pandas profiling for data quality assessment
* Great Expectations for data validation