Complete Reproduction Guide - From Processed Data to Final Results

Self-Contained Guide: This document contains ALL code and instructions needed to reproduce the experimental results.

Goal: Download pre-processed datasets -> Run experiments -> Get identical `final_results.csv` -> Statistical analysis

Random Seed: All operations use `RANDOM_STATE = 42` for reproducibility

Table of Contents

1. Overview
2. Requirements
3. Project Structure
4. Phase 1: Setup Environment
5. Phase 2: Download Datasets
6. Phase 3: Create Code Files
7. Phase 4: Run Experiments
8. Phase 5: Combine Results
9. Phase 6: Statistical Analysis
    - Step 6.1: Calculate Target Standard Deviation
    - Step 6.2: Compute Normalized Metrics
    - Step 6.3: Compute Rankings (Friedman Test)
    - Step 6.4: Post-Hoc Tests (Nemenyi)
    - Step 6.5: Generate Summary Tables
    - Step 6.6: Sensitivity Analysis (Optional)
    - Step 6.7: Wilcoxon Tuning Efficiency Test
10. Validation
11. Troubleshooting
12. File Checklist

Overview

What Gets Run

- **72 datasets** (36 classification + 36 regression)
- **4 models**: Linear/Logistic Regression, SVM, XGBoost, MLP
- **2 configurations**: Default, Tuned
- **Total**: 576 experiments (72 x 4 x 2)

Time Estimates (with GPU)

- Phase 1-3: 30-60 minutes (setup)
- Phase 4: 12-48 hours (experiments)

- Phase 5-6: 15-30 minutes (analysis)

Requirements

Python Version

Python 3.9+

Install Dependencies

```
pip install numpy pandas scikit-learn torch xgboost imbalanced-learn
         matplotlib seaborn scipy joblib psutil
```

GPU Support (Recommended)

```
# Check your CUDA version first: nvidia-smi
pip install torch --index-url https://download.pytorch.org/whl/cu118
```

Verify Installation

```
import torch
print(f"PyTorch: {torch.__version__}")
print(f"CUDA available: {torch.cuda.is_available()}")
if torch.cuda.is_available():
    print(f"GPU: {torch.cuda.get_device_name(0)}")
```

Project Structure

Create this folder structure:

```
Masters_Thesis_Code/
+-- config.py
+-- batch_factorial_runner.py
+-- data/
|   +-- __init__.py
|   +-- preprocessing.py
|   +-- dataset.py
|   +-- dynamic_dataset.py
|   +-- processed_classification/    <-- FROM GOOGLE DRIVE
|   |   +-- smallxsmall/
|   |   +-- smallxmed/
|   |   +-- smallxlarge/
|   |   +-- medxsmall/
|   |   +-- medxmed/
|   |   +-- medxlarge/
|   |   +-- largexsmall/
|   |   +-- largexmed/
|   |   +-- largexlarge/
|   +-- processed_regression/        <-- FROM GOOGLE DRIVE
```

```
|       +-- smallxsmall/
|       +-- smallxmed/
|       +-- smallxlarge/
|       +-- medxsmall/
|       +-- medxmed/
|       +-- medxlarge/
|       +-- largexsmall/
|       +-- largexmed/
|       +-- largexlarge/
+-- models/
|   +-- __init__.py
|   +-- base_models.py
|   +-- advanced_models.py
|   +-- deep_learning.py
|   +-- gpu_base_models.py
+-- evaluation/
|   +-- __init__.py
|   +-- metrics.py
|   +-- hyperparameter_tuning.py
+-- scripts/
|   +-- combine_results.py
|   +-- calculate_stdy.py
|   +-- compute_normalized_metrics.py
|   +-- compute_rankings.py
|   +-- compute_rankings_senstest.py
|   +-- nemenyi_posthoc.py
|   +-- generate_summary_tables.py
|   +-- generate_summary_tables_senstest.py
|   +-- wilcoxon_tuning_efficiency.py
+-- results/                          <-- OUTPUT FOLDER
+-- logs/                             <-- LOG FILES
```

Phase 1: Setup Environment

```bash
# Create project directory
mkdir -p Masters_Thesis_Code
cd Masters_Thesis_Code

# Create folder structure
mkdir -p data/processed_classification data/processed_regression
mkdir -p models evaluation scripts results logs

# Create empty __init__.py files
touch data/__init__.py models/__init__.py evaluation/__init__.py
```

Phase 2: Download Datasets

Step 2.1: Download from Google Drive

Google Drive Link:
[https://drive.google.com/drive/folders/1LvMsrjqd0TVXAB07FpUNG06mUt0-aF4h?usp=sharing]

Step 2.2: Extract Datasets

Extract the downloaded ZIP so that: - Classification datasets go to `data/processed_classification/` - Regression datasets go to `data/processed_regression/`

Step 2.3: Verify

```
# Should show 36 files
find data/processed_classification -name "*.csv" | wc -l

# Should show 36 files
find data/processed_regression -name "*.csv" | wc -l
```

Phase 3: Create Code Files

Create each of the following files with the code provided.

3.1: config.py

Location: `config.py` (project root)

```
import os
from pathlib import Path
import torch

# Project paths
BASE_DIR = Path(__file__).parent
DATA_DIR = BASE_DIR / "data"
RAW_DATA_DIR = DATA_DIR / "raw"
PROCESSED_DATA_DIR = DATA_DIR / "processed"
MODELS_DIR = BASE_DIR / "models"
RESULTS_DIR = BASE_DIR / "results"
REPORTS_DIR = BASE_DIR / "reports"
FIGURES_DIR = REPORTS_DIR / "figures"

# Create directories if they don't exist
for directory in [DATA_DIR, RAW_DATA_DIR, PROCESSED_DATA_DIR, MODELS_DIR,
          RESULTS_DIR, REPORTS_DIR, FIGURES_DIR]:
    os.makedirs(directory, exist_ok=True)
```

```python
# Data splitting parameters
RANDOM_STATE = 42
TEST_SIZE = 0.3

# Cross-validation parameters
N_CV_FOLDS = 5
N_JOBS = -1  # Use all available cores

# Model parameters
# Logistic Regression
LOGISTIC_PARAMS = {
    'penalty': 'l2',
    'C': 1.0,
    'solver': 'lbfgs',
    'max_iter': 5000,  # Increased for complex datasets
    'class_weight': 'balanced'
}

# Linear Regression
LINEAR_PARAMS = {}

# GPU Configuration
USE_GPU = True  # Global GPU usage flag
GPU_DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

# GPU Linear/Logistic Regression Parameters
GPU_LINEAR_PARAMS = {
    'learning_rate': 0.01,
    'max_iter': 1000,
    'tol': 1e-6,
    'batch_size': 10000,
    'use_analytical': True,  # Use analytical solution for linear regression
    'use_gpu': USE_GPU
}

GPU_LOGISTIC_PARAMS = {
    'learning_rate': 0.01,
    'max_iter': 1000,
    'tol': 1e-6,
    'batch_size': 10000,
    'use_gpu': USE_GPU
}

# XGBoost Classification - Modern GPU approach (same as test.py)
XGBOOST_CLASSIFICATION_PARAMS = {
    'objective': 'binary:logistic',
    'n_estimators': 100,
    'max_depth': 6,
    'learning_rate': 0.1,
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'tree_method': 'hist',      # Modern GPU approach
    'device': 'cuda',           # GPU device specification
    'random_state': RANDOM_STATE
}
```

```python
# XGBoost Regression - Modern GPU approach (same as test.py)
XGBOOST_REGRESSION_PARAMS = {
    'objective': 'reg:squarederror',
    'n_estimators': 100,
    'max_depth': 6,
    'learning_rate': 0.1,
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'tree_method': 'hist',      # Modern GPU approach
    'device': 'cuda',           # GPU device specification
    'random_state': RANDOM_STATE
}

SVM_CLASSIFICATION_PARAMS = {
    'C': 1.0,
    'kernel': 'linear',
    'probability': True,
    'gpu_id': 0  # Use first GPU
}

# SVM Regression with GPU parameters - ALWAYS LINEAR for speed
SVM_REGRESSION_PARAMS = {
    'C': 1.0,
    'kernel': 'linear',  # Force linear kernel for all datasets
    'epsilon': 0.1,
    'gpu_id': 0  # Use first GPU
}

# Deep Learning Parameters
DL_PARAMS = {
    # Model Architecture
    'hidden_layers': [64, 32],  # Default network architecture
    'dropout_rate': 0.2,  # Dropout rate for regularization

    # Training Hyperparameters
    'learning_rate': 0.001,  # Default learning rate
    'batch_size': 32,  # Default batch size
    'epochs': 100,  # Default number of training epochs

    # Optimization
    'optimizer': 'Adam',  # Default optimizer

    # Device Configuration
    'use_gpu': torch.cuda.is_available(),  # Automatically detect GPU
        availability

    # Additional Hyperparameters
    'random_seed': 42  # For reproducibility
}

def update_dl_params(custom_params=None):
    """Update default deep learning parameters with custom parameters."""
    params = DL_PARAMS.copy()
    if custom_params:
        params.update(custom_params)
```

```
    return params

# Features for fraud classification
FRAUD_FEATURES = {
    'categorical': [
        'policy_state', 'policy_csl', 'insured_sex',
         'insured_education_level',
        'insured_occupation', 'insured_hobbies', 'insured_relationship',
        'incident_type', 'collision_type', 'incident_severity',
        'authorities_contacted', 'incident_state', 'incident_city',
        'property_damage', 'police_report_available', 'auto_make',
         'auto_model'
    ],
    'numerical': [
        'months_as_customer', 'age', 'policy_deductable',
         'policy_annual_premium',
        'umbrella_limit', 'capital-gains', 'capital-loss',
         'incident_hour_of_the_day',
        'number_of_vehicles_involved', 'bodily_injuries', 'witnesses',
        'total_claim_amount', 'injury_claim', 'property_claim',
         'vehicle_claim', 'auto_year',
        'vehicle_proportion', 'property_proportion', 'injury_proportion'
    ],
    'target': 'fraud_reported'
}

# Features for insurance premium regression
PREMIUM_FEATURES = {
    'categorical': [
        'gender', 'smoker', 'region', 'medical_history',
        'family_medical_history', 'exercise_frequency', 'occupation',
         'coverage_level'
    ],
    'numerical': [
        'age', 'bmi', 'children'
    ],
    'target': 'charges'
}

# Evaluation metrics to compute
CLASSIFICATION_METRICS = [
    'accuracy', 'precision', 'recall', 'f1', 'roc_auc', 'confusion_matrix',
         'classification_report'
]

REGRESSION_METRICS = [
    'r2', 'mean_squared_error', 'mean_absolute_error',
         'median_absolute_error', 'explained_variance'
]
```

3.2: data/preprocessing.py

Location: `data/preprocessing.py`

```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
import logging

# Set up logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %
        (message)s')
logger = logging.getLogger(__name__)

def load_data(file_path):
    """Load data from CSV file."""
    logger.info(f"Loading data from {file_path}")
    try:
        df = pd.read_csv(file_path)
        logger.info(f"Successfully loaded data with shape {df.shape}")
        return df
    except Exception as e:
        logger.error(f"Error loading data: {e}")
        raise

def check_missing_values(df):
    """Check for missing values in the DataFrame."""
    missing_count = df.isnull().sum()
    missing_percentage = (missing_count / len(df)) * 100
    missing_df = pd.DataFrame({
        'Missing Count': missing_count,
        'Missing Percentage': missing_percentage
    })

    return missing_df[missing_df['Missing Count'] >
         0].sort_values(by='Missing Count', ascending=False)

def get_data_info(df):
    """Get basic information about the DataFrame."""
    info = {
        'shape': df.shape,
        'columns': list(df.columns),
        'dtypes': df.dtypes.to_dict(),
        'numerical_columns': list(df.select_dtypes(include=['int64',
         'float64']).columns),
        'categorical_columns': list(df.select_dtypes(include=['object',
         'category']).columns),
        'missing_values': check_missing_values(df),
        'summary_statistics': df.describe()
    }

    return info

def create_preprocessing_pipeline(categorical_features, numerical_features):
    """Create preprocessing pipeline for numerical and categorical
         features."""
```

```
    numerical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())
    ])

    categorical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='most_frequent')),
        ('onehot', OneHotEncoder(handle_unknown='ignore',
         sparse_output=False))
    ])

    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numerical_transformer, numerical_features),
            ('cat', categorical_transformer, categorical_features)
        ],
        remainder='drop'  # Drop other columns not specified
    )

    return preprocessor

def preprocess_fraud_data(df):
    """Preprocess the fraud dataset."""
    logger.info("Preprocessing fraud data...")

    # Make a copy to avoid modifying the original
    df_processed = df.copy()

    # Handle unknown values
    df_processed.replace('?', np.nan, inplace=True)

    # Convert date columns to datetime
    try:
        if 'policy_bind_date' in df_processed.columns:
            df_processed['policy_bind_date'] =
         pd.to_datetime(df_processed['policy_bind_date'], errors='coerce')
            # Extract year and month from date
            df_processed['policy_bind_year'] =
         df_processed['policy_bind_date'].dt.year
            df_processed['policy_bind_month'] =
         df_processed['policy_bind_date'].dt.month

        if 'incident_date' in df_processed.columns:
            df_processed['incident_date'] =
         pd.to_datetime(df_processed['incident_date'], errors='coerce')
            df_processed['incident_year'] =
         df_processed['incident_date'].dt.year
            df_processed['incident_month'] =
         df_processed['incident_date'].dt.month
            df_processed['incident_dow'] =
         df_processed['incident_date'].dt.dayofweek
    except Exception as e:
        logger.warning(f"Error processing date columns: {e}")

    # Feature engineering - claim proportions
    try:
```

```
            df_processed['vehicle_proportion'] = df_processed['vehicle_claim'] /
             df_processed['total_claim_amount']
            df_processed['property_proportion'] = df_processed['property_claim']
             / df_processed['total_claim_amount']
            df_processed['injury_proportion'] = df_processed['injury_claim'] /
             df_processed['total_claim_amount']

            # Replace inf with 0 (case where total_claim_amount is 0)
            df_processed.replace([np.inf, -np.inf], 0, inplace=True)
        except Exception as e:
            logger.warning(f"Error creating claim proportion features: {e}")

        # Add age groups
        try:
            df_processed['age_group'] = pd.cut(
                df_processed['age'],
                bins=[0, 25, 35, 45, 55, 65, 100],
                labels=['<25', '25-35', '35-45', '45-55', '55-65', '65+']
            )
        except Exception as e:
            logger.warning(f"Error creating age groups: {e}")

        # Create binary indicator if police was contacted but no report is
             available
        try:
            df_processed['police_no_report'] = np.where(
                (df_processed['authorities_contacted'] == "Police") &
                (df_processed['police_report_available'] == "NO"),
                1, 0
            )
        except Exception as e:
            logger.warning(f"Error creating police report indicator: {e}")

        # Convert target to binary
        if 'fraud_reported' in df_processed.columns:
            df_processed['fraud_reported'] =
             df_processed['fraud_reported'].map({'Y': 1, 'N': 0})

        logger.info("Fraud data preprocessing completed")
        return df_processed

    def preprocess_insurance_data(df):
        """Preprocess the insurance premium dataset."""
        logger.info("Preprocessing insurance data...")

        # Make a copy to avoid modifying the original
        df_processed = df.copy()

        # Scale large Cost values
        scaler = StandardScaler()
        df_processed['charges'] = scaler.fit_transform(df_processed[['charges']])

        # Handle unknown values
        df_processed.replace('?', np.nan, inplace=True)

        # Create BMI categories
```

```
    try:
        df_processed['bmi_category'] = pd.cut(
            df_processed['bmi'],
            bins=[0, 18.5, 25, 30, 40, 100],
            labels=['underweight', 'normal', 'overweight', 'obese',
     'extremely_obese']
        )
    except Exception as e:
        logger.warning(f"Error creating BMI categories: {e}")

    # Create age groups
    try:
        df_processed['age_group'] = pd.cut(
            df_processed['age'],
            bins=[0, 25, 35, 45, 55, 65, 100],
            labels=['<25', '25-35', '35-45', '45-55', '55-65', '65+']
        )
    except Exception as e:
        logger.warning(f"Error creating age groups: {e}")

    # Create interaction features
    try:
        # Smoker and BMI interaction (higher BMI with smoking may lead to
     higher premiums)
        df_processed['smoker_numeric'] = df_processed['smoker'].map({'yes':
     1, 'no': 0})
        df_processed['smoker_bmi'] = df_processed['smoker_numeric'] *
     df_processed['bmi']

        # Age and smoker interaction
        df_processed['age_smoker'] = df_processed['age'] *
     df_processed['smoker_numeric']
    except Exception as e:
        logger.warning(f"Error creating interaction features: {e}")

    logger.info("Insurance data preprocessing completed")
    return df_processed

def encode_target(df, target_column, mapping=None):
    """Encode target column for classification tasks."""
    df = df.copy()

    if target_column not in df.columns:
        logger.error(f"Target column {target_column} not found in DataFrame")
        return df, None

    if mapping is None:
        # Auto-create mapping for categorical target
        if df[target_column].dtype == 'object':
            categories = df[target_column].unique()
            mapping = {cat: i for i, cat in enumerate(categories)}
        else:
            # Target is already numeric
            return df, None

    df[target_column] = df[target_column].map(mapping)
```

```
    logger.info(f"Encoded target column {target_column} using mapping:
        {mapping}")

    return df, mapping
```

3.3: data/dataset.py

Location: `data/dataset.py`

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
import logging
import sys
import os
import gc

# Add parent directory to path
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from config import TEST_SIZE, RANDOM_STATE, FRAUD_FEATURES, PREMIUM_FEATURES
from data.preprocessing import preprocess_fraud_data,
        preprocess_insurance_data, create_preprocessing_pipeline

# Set up logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %
        (message)s')
logger = logging.getLogger(__name__)

class Dataset:
    """Base class for datasets"""

    def __init__(self, file_path, task='classification', test_size=TEST_SIZE,
         random_state=RANDOM_STATE):
        self.file_path = file_path
        self.task = task
        self.test_size = test_size
        self.random_state = random_state
        self.X_train = None
        self.X_test = None
        self.y_train = None
        self.y_test = None
        self.preprocessor = None
        self.feature_names = None

    def load_data(self):
        """Load data from file"""
        try:
            self.data = pd.read_csv(self.file_path)
            logger.info(f"Loaded data from {self.file_path} with shape
         {self.data.shape}")
            return self.data
```

```
        except Exception as e:
            logger.error(f"Error loading data: {e}")
            raise

    def preprocess(self):
        """Preprocess data - to be implemented by subclasses"""
        raise NotImplementedError

    def split_data(self, X, y):
        """Split data into training and test sets."""
        self.X_train, self.X_test, self.y_train, self.y_test =
         train_test_split(
            X, y, test_size=self.test_size, random_state=self.random_state,
         stratify=y if self.task == 'classification' else None
        )

        logger.info(f"Train set shape: {self.X_train.shape}, Test set shape:
         {self.X_test.shape}")

        return self.X_train, self.X_test, self.y_train, self.y_test

    def apply_preprocessing_pipeline(self, categorical_features,
         numerical_features):
        """Create and apply preprocessing pipeline."""
        self.preprocessor =
         create_preprocessing_pipeline(categorical_features,
         numerical_features)

        # Fit on training data
        self.preprocessor.fit(self.X_train)

        # Transform both training and test data
        X_train_processed = self.preprocessor.transform(self.X_train)
        X_test_processed = self.preprocessor.transform(self.X_test)

        # Get transformed feature names
        self.feature_names = self._get_feature_names(self.preprocessor,
         categorical_features, numerical_features)

        return X_train_processed, X_test_processed

    def _get_feature_names(self, preprocessor, categorical_features,
         numerical_features):
        """Get feature names after preprocessing."""
        feature_names = []

        # Get numerical feature names (these stay the same)
        feature_names.extend(numerical_features)

        # Get one-hot encoded feature names
        if categorical_features:
            ohe =
         preprocessor.named_transformers_['cat'].named_steps['onehot']
            categories = ohe.categories_

            for i, category in enumerate(categories):
```

```
                feature_names.extend([f"{categorical_features[i]}_{cat}" for
            cat in category])

        return feature_names

    def get_data(self):
        """Get processed training and test data."""
        if self.X_train is None or self.X_test is None or self.y_train is
            None or self.y_test is None:
            logger.error("Data has not been split yet. Call split_data
            first.")
            raise ValueError("Data has not been split yet. Call split_data
            first.")

        return self.X_train, self.X_test, self.y_train, self.y_test

    def get_feature_names(self):
        """Get feature names after preprocessing."""
        return self.feature_names


class FraudDataset(Dataset):
    """Class for fraud detection dataset"""

    def __init__(self, file_path, test_size=TEST_SIZE,
            random_state=RANDOM_STATE):
        super().__init__(file_path, task='classification',
            test_size=test_size, random_state=random_state)
        self.categorical_features = FRAUD_FEATURES['categorical']
        self.numerical_features = FRAUD_FEATURES['numerical']
        self.target = FRAUD_FEATURES['target']

    def preprocess(self):
        """Preprocess fraud data."""
        self.load_data()

        # Apply fraud-specific preprocessing
        self.processed_data = preprocess_fraud_data(self.data)

        # Drop rows with missing target
        self.processed_data = self.processed_data.dropna(subset=
            [self.target])

        # Extract features and target
        X = self.processed_data[self.categorical_features +
            self.numerical_features]
        y = self.processed_data[self.target]

        logger.info(f"Preprocessed data shape: {X.shape}")

        return X, y

    def prepare_data(self):
        """Prepare data for modeling."""
        X, y = self.preprocess()
```

```
        # Split data
        self.split_data(X, y)

        # Apply preprocessing pipeline
        X_train_processed, X_test_processed =
         self.apply_preprocessing_pipeline(
            self.categorical_features, self.numerical_features
        )

        logger.info("Data preparation completed")

        return X_train_processed, X_test_processed, self.y_train, self.y_test

    def prepare_data_chunked(self, chunk_size=100000):
        """Prepare data for modeling in chunks."""
        logger.info(f"Preparing data in chunks of {chunk_size} rows")

        # Load data in chunks
        chunk_iterator = pd.read_csv(self.file_path, chunksize=chunk_size)

        # Initialize lists to store processed data
        all_X_train = []
        all_X_test = []
        all_y_train = []
        all_y_test = []

        # Process each chunk
        for i, chunk in enumerate(chunk_iterator):
            logger.info(f"Processing chunk {i+1}")

            # Apply preprocessing to chunk
            if self.task == 'classification':
                chunk_processed = preprocess_fraud_data(chunk)
            else:  # regression
                chunk_processed = preprocess_insurance_data(chunk)

            # Drop rows with missing target
            chunk_processed = chunk_processed.dropna(subset=[self.target])

            # Extract features and target
            X_chunk = chunk_processed[self.categorical_features +
         self.numerical_features]
            y_chunk = chunk_processed[self.target]

            # Split chunk data - maintain same random state for all chunks
            X_train_chunk, X_test_chunk, y_train_chunk, y_test_chunk =
         train_test_split(
                X_chunk, y_chunk, test_size=self.test_size,
                random_state=self.random_state + i  # Slightly different for
         each chunk
            )

            # Append to lists
            all_X_train.append(X_train_chunk)
            all_X_test.append(X_test_chunk)
            all_y_train.append(y_train_chunk)
```

```
            all_y_test.append(y_test_chunk)

            # Force garbage collection after each chunk
            gc.collect()

        # Combine chunks
        self.X_train = pd.concat(all_X_train, ignore_index=True)
        self.X_test = pd.concat(all_X_test, ignore_index=True)
        self.y_train = pd.concat(all_y_train, ignore_index=True)
        self.y_test = pd.concat(all_y_test, ignore_index=True)

        logger.info(f"Combined data shapes - X_train: {self.X_train.shape},
         X_test: {self.X_test.shape}")

        # Apply preprocessing pipeline
        X_train_processed, X_test_processed =
         self.apply_preprocessing_pipeline(
            self.categorical_features, self.numerical_features
        )

        logger.info("Chunked data preparation completed")

        return X_train_processed, X_test_processed, self.y_train, self.y_test



class InsuranceDataset(Dataset):
    """Class for insurance premium prediction dataset"""

    def __init__(self, file_path, test_size=TEST_SIZE,
         random_state=RANDOM_STATE):
        super().__init__(file_path, task='regression', test_size=test_size,
         random_state=random_state)
        self.categorical_features = PREMIUM_FEATURES['categorical']
        self.numerical_features = PREMIUM_FEATURES['numerical']
        self.target = PREMIUM_FEATURES['target']

    def preprocess(self):
        """Preprocess insurance data."""
        self.load_data()

        # Apply insurance-specific preprocessing
        self.processed_data = preprocess_insurance_data(self.data)

        # Drop rows with missing target
        self.processed_data = self.processed_data.dropna(subset=
         [self.target])

        # Extract features and target
        X = self.processed_data[self.categorical_features +
         self.numerical_features]
        y = self.processed_data[self.target]

        logger.info(f"Preprocessed data shape: {X.shape}")

        return X, y
```

```
    def prepare_data(self):
        """Prepare data for modeling."""
        X, y = self.preprocess()

        # Split data
        self.split_data(X, y)

        # Apply preprocessing pipeline
        X_train_processed, X_test_processed =
         self.apply_preprocessing_pipeline(
            self.categorical_features, self.numerical_features
        )

        logger.info("Data preparation completed")

        return X_train_processed, X_test_processed, self.y_train, self.y_test

    def prepare_data_chunked(self, chunk_size=100000):
        """Prepare data for modeling in chunks."""
        logger.info(f"Preparing data in chunks of {chunk_size} rows")

        # Load data in chunks
        chunk_iterator = pd.read_csv(self.file_path, chunksize=chunk_size)

        # Initialize lists to store processed data
        all_X_train = []
        all_X_test = []
        all_y_train = []
        all_y_test = []

        # Process each chunk
        for i, chunk in enumerate(chunk_iterator):
            logger.info(f"Processing chunk {i+1}")

            # Apply preprocessing to chunk
            if self.task == 'classification':
                chunk_processed = preprocess_fraud_data(chunk)
            else:  # regression
                chunk_processed = preprocess_insurance_data(chunk)

            # Drop rows with missing target
            chunk_processed = chunk_processed.dropna(subset=[self.target])

            # Extract features and target
            X_chunk = chunk_processed[self.categorical_features +
         self.numerical_features]
            y_chunk = chunk_processed[self.target]

            # Split chunk data - maintain same random state for all chunks
            X_train_chunk, X_test_chunk, y_train_chunk, y_test_chunk =
         train_test_split(
                X_chunk, y_chunk, test_size=self.test_size,
                random_state=self.random_state + i  # Slightly different for
         each chunk
            )
```

```
            # Append to lists
            all_X_train.append(X_train_chunk)
            all_X_test.append(X_test_chunk)
            all_y_train.append(y_train_chunk)
            all_y_test.append(y_test_chunk)

            # Force garbage collection after each chunk
            gc.collect()

        # Combine chunks
        self.X_train = pd.concat(all_X_train, ignore_index=True)
        self.X_test = pd.concat(all_X_test, ignore_index=True)
        self.y_train = pd.concat(all_y_train, ignore_index=True)
        self.y_test = pd.concat(all_y_test, ignore_index=True)

        logger.info(f"Combined data shapes - X_train: {self.X_train.shape},
         X_test: {self.X_test.shape}")

        # Apply preprocessing pipeline
        X_train_processed, X_test_processed =
         self.apply_preprocessing_pipeline(
            self.categorical_features, self.numerical_features
        )

        logger.info("Chunked data preparation completed")

        return X_train_processed, X_test_processed, self.y_train, self.y_test
```

3.4: data/dynamic_dataset.py

Location: `data/dynamic_dataset.py`

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import logging
import sys
import os
import gc

# Add parent directory to path
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from config import TEST_SIZE, RANDOM_STATE
from data.preprocessing import create_preprocessing_pipeline
from data.dataset import Dataset

# Set up logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %
        (message)s')
logger = logging.getLogger(__name__)
```

```python
class DynamicDataset(Dataset):
    """Dynamic dataset class that works with any CSV file structure"""

    def __init__(self, config, test_size=TEST_SIZE,
                 random_state=RANDOM_STATE):
        super().__init__(config['csv_path'], task=config['task_type'],
                         test_size=test_size, random_state=random_state)

        self.config = config
        self.categorical_features = config['categorical_features']
        self.numerical_features = config['numerical_features']
        self.feature_variables = config['feature_variables']
        self.target = config['target_variable']

        logger.info(f"Initialized dynamic dataset for {config['task_type']}
         task")
        logger.info(f"Target: {self.target}")
        logger.info(f"Features: {len(self.feature_variables)} total "
                    f"({len(self.categorical_features)} categorical, "
                    f"{len(self.numerical_features)} numerical)")

    def preprocess(self):
        """Preprocess dynamic data."""
        self.load_data()

        # Basic data cleaning
        self.processed_data = self.data.copy()

        # Handle missing values in features
        for col in self.feature_variables:
            if col in self.processed_data.columns:
                if self.processed_data[col].dtype == 'object':
                    # Fill categorical missing values with 'unknown'
                    self.processed_data[col] =
         self.processed_data[col].fillna('unknown')
                else:
                    # Fill numerical missing values with median
                    self.processed_data[col] =
         self.processed_data[col].fillna(
                        self.processed_data[col].median()
                    )

        # Drop rows with missing target values
        self.processed_data = self.processed_data.dropna(subset=
         [self.target])

        # Extract features and target
        X = self.processed_data[self.feature_variables]
        y = self.processed_data[self.target]

        # Convert target to appropriate type
        if self.task == 'classification':
            # ALWAYS encode classification targets to ensure 0-indexing
         (critical for CUDA)
```

```
            # Labels must be in range [0, num_classes) for PyTorch
         CrossEntropyLoss
            le = LabelEncoder()
            y = le.fit_transform(y)
            # Store label encoder for later use
            self.label_encoder = le
            logger.info(f"Encoded target classes from {le.classes_} to [0,
         {len(le.classes_)-1}]")

        logger.info(f"Preprocessed data shape: {X.shape}")
        if self.task == 'classification':
            logger.info(f"Target distribution:
         {pd.Series(y).value_counts().to_dict()}")
        else:
            logger.info(f"Target range: {y.min():.3f} to {y.max():.3f}")

        return X, y

    def prepare_data(self):
        """Prepare data for modeling."""
        X, y = self.preprocess()

        # Split data
        self.split_data(X, y)

        # CRITICAL FIX: Ensure targets are proper numpy arrays for sklearn
         compatibility
        if self.task == 'classification':
            # Convert to numpy arrays and ensure integer type
            self.y_train = np.array(self.y_train, dtype=int)
            self.y_test = np.array(self.y_test, dtype=int)
            logger.info(f"Target classes after encoding:
         {np.unique(self.y_train)}")
        else:
            # For regression, ensure float type
            self.y_train = np.array(self.y_train, dtype=float)
            self.y_test = np.array(self.y_test, dtype=float)

        logger.info(f"Target shapes - y_train: {self.y_train.shape}, y_test:
         {self.y_test.shape}")
        logger.info(f"Target dtypes - y_train: {self.y_train.dtype}, y_test:
         {self.y_test.dtype}")

        # Apply preprocessing pipeline
        X_train_processed, X_test_processed =
         self.apply_preprocessing_pipeline(
            self.categorical_features, self.numerical_features
        )

        # Final safety check: Remove any remaining NaN values
        if np.any(np.isnan(X_train_processed)):
            logger.warning("NaN values detected in X_train after
         preprocessing - replacing with 0")
            X_train_processed = np.nan_to_num(X_train_processed, nan=0.0)
        if np.any(np.isnan(X_test_processed)):
            logger.warning("NaN values detected in X_test after preprocessing
         - replacing with 0")
```

```
            X_test_processed = np.nan_to_num(X_test_processed, nan=0.0)
        if np.any(np.isnan(self.y_train)):
            logger.warning("NaN values detected in y_train - replacing with
         median")
            y_median = np.nanmedian(self.y_train)
            self.y_train = np.nan_to_num(self.y_train, nan=y_median)
        if np.any(np.isnan(self.y_test)):
            logger.warning("NaN values detected in y_test - replacing with
         median")
            y_median = np.nanmedian(self.y_test)
            self.y_test = np.nan_to_num(self.y_test, nan=y_median)

        logger.info("Dynamic data preparation completed")

        return X_train_processed, X_test_processed, self.y_train, self.y_test

    def prepare_data_chunked(self, chunk_size=100000):
        """Prepare data for modeling in chunks."""
        logger.info(f"Preparing dynamic data in chunks of {chunk_size} rows")

        # Initialize label encoder if classification
        if self.task == 'classification':
            self.label_encoder = LabelEncoder()
            # First pass to fit label encoder
            first_chunk = next(pd.read_csv(self.file_path,
         chunksize=chunk_size))
            first_chunk = first_chunk.dropna(subset=[self.target])
            # Always fit label encoder for classification (both string and
         numeric targets)
            self.label_encoder.fit(first_chunk[self.target])
            logger.info(f"Label encoder fitted with classes:
         {self.label_encoder.classes_}")
            logger.info(f"Label encoder will map to:
         {list(range(len(self.label_encoder.classes_)))}")

        # Load data in chunks
        chunk_iterator = pd.read_csv(self.file_path, chunksize=chunk_size)

        # Initialize lists to store processed data
        all_X_train = []
        all_X_test = []
        all_y_train = []
        all_y_test = []

        # Process each chunk
        for i, chunk in enumerate(chunk_iterator):
            logger.info(f"Processing chunk {i+1}")

            # Basic preprocessing for chunk
            chunk_processed = chunk.copy()

            # Handle missing values
            for col in self.feature_variables:
                if col in chunk_processed.columns:
                    if chunk_processed[col].dtype == 'object':
```

```
                chunk_processed[col] =
chunk_processed[col].fillna('unknown')
            else:
                chunk_processed[col] =
chunk_processed[col].fillna(chunk_processed[col].median())

    # Drop rows with missing target
    chunk_processed = chunk_processed.dropna(subset=[self.target])

    if len(chunk_processed) == 0:
        logger.warning(f"Chunk {i+1} is empty after preprocessing")
        continue

    # Extract features and target
    X_chunk = chunk_processed[self.feature_variables]
    y_chunk = chunk_processed[self.target]

    # Encode target if classification
    if self.task == 'classification' and hasattr(self,
'label_encoder'):
        # Always transform targets for classification (both string
and numeric)
        y_chunk = pd.Series(self.label_encoder.transform(y_chunk),
index=y_chunk.index)

    # Split chunk data
    stratify_chunk = y_chunk if self.task == 'classification' else
None
    try:
        X_train_chunk, X_test_chunk, y_train_chunk, y_test_chunk =
train_test_split(
            X_chunk, y_chunk, test_size=self.test_size,
            random_state=self.random_state + i,
            stratify=stratify_chunk
        )
    except ValueError as e:
        logger.warning(f"Stratification failed for chunk {i+1}, using
random split: {e}")
        X_train_chunk, X_test_chunk, y_train_chunk, y_test_chunk =
train_test_split(
            X_chunk, y_chunk, test_size=self.test_size,
            random_state=self.random_state + i
        )

    # Append to lists
    all_X_train.append(X_train_chunk)
    all_X_test.append(X_test_chunk)
    all_y_train.append(y_train_chunk)
    all_y_test.append(y_test_chunk)

    # Force garbage collection after each chunk
    gc.collect()

# Combine chunks
self.X_train = pd.concat(all_X_train, ignore_index=True)
self.X_test = pd.concat(all_X_test, ignore_index=True)
self.y_train = pd.concat(all_y_train, ignore_index=True)
```

```
        self.y_test = pd.concat(all_y_test, ignore_index=True)

        # CRITICAL FIX: Ensure targets are proper numpy arrays for sklearn
         compatibility
        if self.task == 'classification':
            # Convert to numpy arrays and ensure integer type
            self.y_train = np.array(self.y_train, dtype=int)
            self.y_test = np.array(self.y_test, dtype=int)
            logger.info(f"Target classes after encoding:
         {np.unique(self.y_train)}")
        else:
            # For regression, ensure float type
            self.y_train = np.array(self.y_train, dtype=float)
            self.y_test = np.array(self.y_test, dtype=float)

        logger.info(f"Combined data shapes - X_train: {self.X_train.shape},
         X_test: {self.X_test.shape}")
        logger.info(f"Target shapes - y_train: {self.y_train.shape}, y_test:
         {self.y_test.shape}")
        logger.info(f"Target dtypes - y_train: {self.y_train.dtype}, y_test:
         {self.y_test.dtype}")

        # Apply preprocessing pipeline
        X_train_processed, X_test_processed =
         self.apply_preprocessing_pipeline(
            self.categorical_features, self.numerical_features
        )

        logger.info("Chunked dynamic data preparation completed")

        return X_train_processed, X_test_processed, self.y_train, self.y_test

    def get_dataset_info(self):
        """Get information about the current dataset configuration"""
        return {
            'file_path': self.file_path,
            'task_type': self.task,
            'target_variable': self.target,
            'total_features': len(self.feature_variables),
            'categorical_features': len(self.categorical_features),
            'numerical_features': len(self.numerical_features),
            'dataset_shape': self.config.get('dataset_shape', 'Unknown')
        }
```

3.5: models/base_models.py

Location: `models/base_models.py`

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression as
         SklearnLogisticRegression, LinearRegression as
         SklearnLinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
```

```
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, confusion_matrix, classification_report,
    r2_score, mean_squared_error, mean_absolute_error, median_absolute_error,
        explained_variance_score
)
import torch
import torch.nn as nn
import torch.optim as optim
import logging
import sys
import os

# Set up logging first
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %
        (message)s')
logger = logging.getLogger(__name__)

# Add parent directory to path
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from config import LOGISTIC_PARAMS, LINEAR_PARAMS, USE_GPU,
        GPU_LINEAR_PARAMS, GPU_LOGISTIC_PARAMS

# Import GPU models
try:
    from models.gpu_base_models import GPULogisticRegression,
        GPULinearRegression
    GPU_MODELS_AVAILABLE = True
    logger.info("GPU models successfully imported")
except ImportError as e:
    GPU_MODELS_AVAILABLE = False
    logger.warning(f"GPU models not available. Using CPU models only. Error:
        {e}")

class BaseClassifier:
    """Base classifier using logistic regression with GPU support"""

    def __init__(self, params=None, use_gpu=None, **kwargs):
        if params is None:
            params = LOGISTIC_PARAMS.copy()
        else:
            params = params.copy()

        # Merge kwargs into params (sklearn clone compatibility)
        params.update(kwargs)

        # Auto-detect GPU usage if not specified - DEFAULT TO GPU for
         performance
        if use_gpu is None:
            use_gpu = GPU_MODELS_AVAILABLE and torch.cuda.is_available()  #
         Always prefer GPU

        # Choose implementation
        if use_gpu and GPU_MODELS_AVAILABLE:
            # Use GPU implementation
            gpu_params = {
```

```
                'learning_rate': params.get('learning_rate', 0.01),
                'max_iter': params.get('max_iter', 1000),
                'batch_size': params.get('batch_size', 10000),
                'use_gpu': True
            }
            self.model = GPULogisticRegression(params=gpu_params)
            self.is_gpu = True
            logger.info("Using GPU-accelerated Logistic Regression")
        else:
            # Use CPU implementation
            self.model = SklearnLogisticRegression(**params)
            self.is_gpu = False
            logger.info("Using CPU Logistic Regression")

        self.params = params
        self.feature_importance = None

    def get_params(self, deep=True):
        """Get parameters for this estimator (sklearn compatibility)"""
        return self.params.copy()

    def set_params(self, **params):
        """Set parameters for this estimator (sklearn compatibility)"""
        self.params.update(params)
        # Reinitialize model with new parameters
        self.__init__(self.params, getattr(self, 'is_gpu', False))
        return self

    def fit(self, X_train, y_train):
        """Fit model to training data."""
        logger.info("Fitting logistic regression model...")
        self.model.fit(X_train, y_train)

        # Calculate feature importance (coefficients)
        if self.is_gpu:
            # GPU model stores feature importance internally
            self.feature_importance = self.model.feature_importance
        else:
            # sklearn model uses coef_ attribute
            self.feature_importance = np.abs(self.model.coef_[0])

        return self

    def predict(self, X):
        """Make predictions."""
        return self.model.predict(X)

    def predict_proba(self, X):
        """Predict class probabilities."""
        if self.is_gpu:
            # GPU model returns single probabilities, need to format for
         sklearn compatibility
            proba_pos = self.model.predict_proba(X)
            proba_neg = 1 - proba_pos
            return np.column_stack([proba_neg, proba_pos])
```

```
        else:
            return self.model.predict_proba(X)

    def evaluate(self, X_test, y_test):
        """Evaluate model on test data."""
        from evaluation.metrics import evaluate_classification_model
        return evaluate_classification_model(self.model, X_test, y_test,
         "Logistic Regression")

    def get_feature_importance(self, feature_names=None):
        """Get feature importance."""
        if self.feature_importance is None:
            logger.warning("Feature importance not available. Fit model
         first.")
            return None

        if feature_names is None:
            feature_names = [f"feature_{i}" for i in
         range(len(self.feature_importance))]

        importance_df = pd.DataFrame({
            'Feature': feature_names,
            'Importance': self.feature_importance
        })

        return importance_df.sort_values('Importance', ascending=False)


class BaseRegressor:
    """Base regressor using linear regression"""

    def __init__(self, params=None, use_gpu=None, **kwargs):
        if params is None:
            params = LINEAR_PARAMS.copy()
        else:
            params = params.copy()

        # Merge kwargs into params (sklearn clone compatibility)
        params.update(kwargs)

        # Linear Regression: Always use CPU (analytical solution doesn't
         benefit from GPU)
        # GPU version uses analytical solver which ignores
         learning_rate/max_iter,
        # making hyperparameter tuning pointless. CPU sklearn is simpler and
         more stable.
        if use_gpu is None:
            use_gpu = False  # Force CPU for Linear Regression

        # Choose implementation
        if use_gpu and GPU_MODELS_AVAILABLE:
            # Use GPU implementation
            gpu_params = {
                'learning_rate': params.get('learning_rate', 0.01),
                'max_iter': params.get('max_iter', 1000),
                'use_gpu': True
```

```
            }
            self.model = GPULinearRegression(params=gpu_params)
            self.is_gpu = True
            logger.info("Using GPU-accelerated Linear Regression")
        else:
            # Use CPU implementation
            self.model = SklearnLinearRegression(**params)
            self.is_gpu = False
            logger.info("Using CPU Linear Regression")

        self.params = params
        self.feature_importance = None

    def get_params(self, deep=True):
        """Get parameters for this estimator (sklearn compatibility)"""
        return self.params.copy()

    def set_params(self, **params):
        """Set parameters for this estimator (sklearn compatibility)"""
        self.params.update(params)
        # Reinitialize model with new parameters
        self.__init__(self.params)
        return self

    def fit(self, X_train, y_train):
        """Fit model to training data."""
        logger.info("Fitting linear regression model...")
        self.model.fit(X_train, y_train)

        # Calculate feature importance (coefficients)
        if self.is_gpu:
            # GPU model stores feature importance internally
            self.feature_importance = self.model.feature_importance
        else:
            # sklearn model uses coef_ attribute
            self.feature_importance = np.abs(self.model.coef_)

        return self

    def predict(self, X):
        """Make predictions."""
        return self.model.predict(X)

    def evaluate(self, X_test, y_test):
        """Evaluate model on test data."""
        from evaluation.metrics import evaluate_regression_model
        return evaluate_regression_model(self, X_test, y_test, "Linear
         Regression")

    def get_feature_importance(self, feature_names=None):
        """Get feature importance."""
        if self.feature_importance is None:
            logger.warning("Feature importance not available. Fit model
         first.")
            return None
```

```
        if feature_names is None:
            feature_names = [f"feature_{i}" for i in
             range(len(self.feature_importance))]

        importance_df = pd.DataFrame({
            'Feature': feature_names,
            'Importance': self.feature_importance
        })

        return importance_df.sort_values('Importance', ascending=False)


# Convenience classes that automatically choose between GPU and CPU versions
class AutoLogisticRegression:
    """Automatically chooses between GPU and CPU Logistic Regression based on
         configuration"""

    def __new__(cls, params=None, force_cpu=False):
        if USE_GPU and GPU_MODELS_AVAILABLE and not force_cpu and
         torch.cuda.is_available():
            logger.info("Using GPU-accelerated Logistic Regression")
            gpu_params = GPU_LOGISTIC_PARAMS.copy()
            if params:
                gpu_params.update(params)
            return GPULogisticRegression(params=gpu_params)
        else:
            logger.info("Using CPU Logistic Regression")
            cpu_params = LOGISTIC_PARAMS.copy()
            if params:
                cpu_params.update(params)
            return BaseClassifier(params=cpu_params)


class AutoLinearRegression:
    """Automatically chooses between GPU and CPU Linear Regression based on
         configuration"""

    def __new__(cls, params=None, force_cpu=False):
        if USE_GPU and GPU_MODELS_AVAILABLE and not force_cpu and
         torch.cuda.is_available():
            logger.info("Using GPU-accelerated Linear Regression")
            gpu_params = GPU_LINEAR_PARAMS.copy()
            if params:
                gpu_params.update(params)
            return GPULinearRegression(params=gpu_params)
        else:
            logger.info("Using CPU Linear Regression")
            cpu_params = LINEAR_PARAMS.copy()
            if params:
                cpu_params.update(params)
            return BaseRegressor(params=cpu_params)


# Aliases for backward compatibility and easy switching
```

```
LogisticRegression = AutoLogisticRegression
LinearRegression = AutoLinearRegression
```

3.6: models/advanced_models.py

Location: `models/advanced_models.py`

```
import numpy as np
import pandas as pd
from xgboost import XGBClassifier, XGBRegressor
from sklearn.svm import SVC, SVR, LinearSVC, LinearSVR
import logging
import sys
import os
import gc
import psutil
import xgboost
import torch
from joblib import dump, load

# Try to import GPU acceleration for scikit-learn
try:
    from sklearnex import patch_sklearn
    patch_sklearn()
    from sklearnex.svm import SVC, SVR  # Import GPU-accelerated SVM
        implementations
    GPU_ACCELERATION = True
    print("scikit-learn-intelex GPU acceleration enabled!")
except ImportError:
    from sklearn.svm import SVC, SVR  # Fall back to CPU implementation
    GPU_ACCELERATION = False
    print("Using standard CPU-based scikit-learn. Consider installing scikit-
        learn-intelex for GPU acceleration.")

# Try to import ThunderSVM for dedicated GPU SVM acceleration
try:
    from thundersvm import SVC as ThunderSVC, SVR as ThunderSVR
    THUNDERSVM_AVAILABLE = True
    print("ThunderSVM GPU acceleration available!")
except ImportError:
    THUNDERSVM_AVAILABLE = False
    print("ThunderSVM not available. Consider installing for better GPU SVM
        performance.")

# Try to import cuML for RAPIDS GPU acceleration
try:
    import cuml
    from cuml.svm import SVC as cuMLSVC, SVR as cuMLSVR
    CUML_AVAILABLE = True
    print("cuML GPU acceleration available!")
except ImportError:
    CUML_AVAILABLE = False
```

```python
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, confusion_matrix, classification_report,
    r2_score, mean_squared_error, mean_absolute_error, median_absolute_error,
         explained_variance_score
)

# Add parent directory to path
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from config import (
    XGBOOST_CLASSIFICATION_PARAMS, XGBOOST_REGRESSION_PARAMS,
    SVM_CLASSIFICATION_PARAMS, SVM_REGRESSION_PARAMS
)

# Set up logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %
         (message)s')
logger = logging.getLogger(__name__)

# Memory monitoring utilities
def get_memory_usage():
    """Get current memory usage in MB"""
    process = psutil.Process(os.getpid())
    mem_info = process.memory_info()
    return mem_info.rss / 1024 / 1024  # Convert to MB

def log_memory_usage(operation):
    """Log current memory usage"""
    memory_mb = get_memory_usage()
    logger.info(f"Memory usage after {operation}: {memory_mb:.2f} MB")

def optimize_array_memory(array):
    """Optimize numpy array memory usage by downcasting to appropriate
         types"""
    if not isinstance(array, np.ndarray):
        return array

    # For float arrays, try to downcast to float32
    if np.issubdtype(array.dtype, np.floating):
        # Check if float32 is sufficient precision
        if np.min(array) > np.finfo(np.float32).min and np.max(array) <
         np.finfo(np.float32).max:
            return array.astype(np.float32)

    # For integer arrays, find the smallest possible integer type
    elif np.issubdtype(array.dtype, np.integer):
        min_val = np.min(array)
        max_val = np.max(array)

        if min_val >= 0:  # Unsigned integers
            if max_val <= 255:
                return array.astype(np.uint8)
            elif max_val <= 65535:
                return array.astype(np.uint16)
            elif max_val <= 4294967295:
```

```
                return array.astype(np.uint32)
            else:  # Signed integers
                if min_val >= -128 and max_val <= 127:
                    return array.astype(np.int8)
                elif min_val >= -32768 and max_val <= 32767:
                    return array.astype(np.int16)
                elif min_val >= -2147483648 and max_val <= 2147483647:
                    return array.astype(np.int32)

        return array

    class XGBoostClassifier:
        """XGBoost classifier for fraud detection with memory optimization"""

        def __init__(self, params=None, use_gpu=True, **kwargs):
            if params is None:
                params = {
                    'n_estimators': 100,
                    'learning_rate': 0.1,
                    'max_depth': 6,
                    'subsample': 0.8,
                    'colsample_bytree': 0.8,
                    'random_state': 42
                    # objective and eval_metric will be set dynamically in fit()
                }
            else:
                params = params.copy()
                # objective and eval_metric will be set dynamically in fit()
             based on number of classes

            # Merge kwargs into params (sklearn clone compatibility)
            params.update(kwargs)

            # Set up GPU or CPU usage (modern approach like test.py)
            if use_gpu:
                params['tree_method'] = 'hist'      # Modern GPU approach
                params['device'] = 'cuda'           # GPU device specification
                logger.info("Using XGBoost with GPU acceleration
             (tree_method=hist, device=cuda)")
            else:
                # Use histogram-based algorithm on CPU
                params['tree_method'] = 'hist'
                params['device'] = 'cpu'            # Explicit CPU specification
                logger.info("Using XGBoost on CPU (tree_method=hist,
             device=cpu)")

            self.model = XGBClassifier(**params)
            self.params = params
            self.feature_importance = None
            self.use_gpu = use_gpu

        def get_params(self, deep=True):
            """Get parameters for this estimator (sklearn compatibility)"""
            return self.params.copy()
```

```
    def set_params(self, **params):
        """Set parameters for this estimator (sklearn compatibility)"""
        self.params.update(params)
        # Reinitialize model with new parameters
        self.__init__(self.params, self.use_gpu)
        return self

    def fit(self, X_train, y_train, eval_set=None, use_dmatrix=True,
             memory_efficient=True):
        """Fit model to training data with GPU support."""
        logger.info(f"Fitting XGBoost model on {'GPU' if self.use_gpu else
             'CPU'}...")

        # Set objective and eval_metric based on number of classes (for both
             DMatrix and sklearn API)
        num_classes = len(np.unique(y_train))
        if num_classes > 2:
            # Multi-class classification
            if 'objective' not in self.params:
                self.params['objective'] = 'multi:softprob'
            if 'eval_metric' not in self.params:
                self.params['eval_metric'] = 'mlogloss'
        else:
            # Binary classification
            if 'objective' not in self.params:
                self.params['objective'] = 'binary:logistic'
            if 'eval_metric' not in self.params:
                self.params['eval_metric'] = 'logloss'

        # Reinitialize model with correct objective
        self.model = XGBClassifier(**self.params)

        # Optimize memory usage if requested
        if memory_efficient and not self.use_gpu:  # Skip for GPU as we'll
             convert to DMatrix
            if isinstance(X_train, pd.DataFrame):
                # For pandas DataFrame, optimize numeric columns
                for col in X_train.select_dtypes(include=['float']).columns:
                    X_train[col] = pd.to_numeric(X_train[col],
             downcast='float')
                for col in X_train.select_dtypes(include=
             ['integer']).columns:
                    X_train[col] = pd.to_numeric(X_train[col],
             downcast='integer')
            elif isinstance(X_train, np.ndarray):
                # For numpy array, optimize array dtype
                X_train = optimize_array_memory(X_train)

            if isinstance(y_train, np.ndarray):
                y_train = optimize_array_memory(y_train)

        # For GPU training, we should always use DMatrix
        if self.use_gpu:
            logger.info("XGBoost GPU training configuration:")
            logger.info(f"Tree method: {self.params.get('tree_method')}")
            logger.info(f"Device: {self.params.get('device')}")
```

```
            logger.info(f"GPU ID: {self.params.get('gpu_id')}")
            use_dmatrix = True

        if use_dmatrix:
            # Using DMatrix for better memory efficiency and performance
            from xgboost import DMatrix, train

            # Create DMatrix object - this automatically handles device
         transfer for GPU training
            dtrain = DMatrix(X_train, label=y_train)

            # Setup evaluation set if provided
            eval_sets = []
            if eval_set is not None:
                for i, (X_eval, y_eval) in enumerate(eval_set):
                    deval = DMatrix(X_eval, label=y_eval)
                    eval_sets.append((deval, f'eval-{i}'))

            eval_sets.insert(0, (dtrain, 'train'))

            # Extract parameters for xgb.train
            train_params = self.params.copy()

            # Handle specific parameters not used by xgb.train
            if 'n_estimators' in train_params:
                num_boost_round = train_params.pop('n_estimators')
            else:
                num_boost_round = 100

            # Set objective and num_class based on number of classes
            num_classes = len(np.unique(y_train))
            if num_classes > 2:
                # Multi-class classification
                train_params['objective'] = 'multi:softprob'
                train_params['eval_metric'] = 'mlogloss'
                train_params['num_class'] = num_classes
            else:
                # Binary classification
                train_params['objective'] = 'binary:logistic'
                train_params['eval_metric'] = 'logloss'
                # Don't set num_class for binary classification

            # Train the model
            logger.info(f"Training with {'GPU' if self.use_gpu else 'CPU'}
         using DMatrix API")
            self.model = train(
                train_params,
                dtrain,
                num_boost_round=num_boost_round,
                evals=eval_sets,
                early_stopping_rounds=50 if eval_set else None,
                verbose_eval=True  # Set to False if you don't want progress
         reports
            )
```

```
        # Get feature importance
            self.feature_importance =
         self.model.get_score(importance_type='gain')

        else:
            # Using standard sklearn API (not recommended for GPU)
            if self.use_gpu:
                logger.warning("Using sklearn API with GPU may cause device
         mismatch warnings. Consider use_dmatrix=True.")

            if eval_set is not None:
                # With evaluation set, use early stopping
                self.model.fit(
                    X_train, y_train,
                    eval_set=eval_set,
                    eval_metric='auc',  # Adjust based on
         classification/regression
                    early_stopping_rounds=50,
                    verbose=True  # Set to False if you don't want progress
         reports
                )
            else:
                # Without evaluation set, simple fit
                self.model.fit(X_train, y_train)

            # Get feature importance
            self.feature_importance = self.model.feature_importances_

        # Force garbage collection to free memory
        gc.collect()

        return self

    def predict(self, X, batch_size=None):
        """Make predictions, optionally in batches."""
        # Check if model is a Booster (trained with DMatrix API)
        is_booster = hasattr(self.model, 'feature_names')

        if batch_size is None:
            if is_booster:
                # For Booster objects
                X_dmatrix = xgboost.DMatrix(X)
                # Get probabilities
                probas = self.model.predict(X_dmatrix)
                # For multi-class, return class with highest probability
                if probas.ndim > 1:
                    return np.argmax(probas, axis=1)
                else:
                    # Binary classification - convert probabilities to binary
         labels
                    return (probas > 0.5).astype(int)
            else:
                # For sklearn API
                return self.model.predict(X)

        # Batch processing for large datasets
```

```
        n_samples = X.shape[0]
        n_batches = int(np.ceil(n_samples / batch_size))

        predictions = []
        for i in range(n_batches):
            start_idx = i * batch_size
            end_idx = min((i + 1) * batch_size, n_samples)

            logger.info(f"Processing prediction batch {i+1}/{n_batches}")
            batch_X = X[start_idx:end_idx]

            if is_booster:
                # For Booster objects
                batch_X_dmatrix = xgboost.DMatrix(batch_X)
                batch_probas = self.model.predict(batch_X_dmatrix)
                # For multi-class, return class with highest probability
                if batch_probas.ndim > 1:
                    batch_pred = np.argmax(batch_probas, axis=1)
                else:
                    # Binary classification - convert probabilities to binary
     labels
                    batch_pred = (batch_probas > 0.5).astype(int)
            else:
                # For sklearn API
                batch_pred = self.model.predict(batch_X)

            predictions.append(batch_pred)

            # Force garbage collection after each batch
            gc.collect()

        return np.concatenate(predictions)

    def predict_proba(self, X, batch_size=None):
        """Predict class probabilities, optionally in batches."""
        is_booster = hasattr(self.model, 'feature_names')

        if batch_size is None:
            if is_booster:
                # For Booster objects, use predict with appropriate
     parameters
                X_dmatrix = xgboost.DMatrix(X)
                # Get probabilities
                probas = self.model.predict(X_dmatrix)
                # Handle multi-class vs binary classification
                if probas.ndim > 1:
                    # Multi-class: probabilities are already in correct
     format
                    return probas
                else:
                    # Binary classification: convert to 2-column format
     expected by sklearn
                    return np.vstack((1 - probas, probas)).T
            else:
                # For sklearn API
                return self.model.predict_proba(X)
```

```
        # Batch processing for large datasets
        n_samples = X.shape[0]
        n_batches = int(np.ceil(n_samples / batch_size))

        probabilities = []
        for i in range(n_batches):
            start_idx = i * batch_size
            end_idx = min((i + 1) * batch_size, n_samples)

            logger.info(f"Processing probability batch {i+1}/{n_batches}")
            batch_X = X[start_idx:end_idx]

            if is_booster:
                # For Booster objects
                batch_X_dmatrix = xgboost.DMatrix(batch_X)
                batch_probas = self.model.predict(batch_X_dmatrix)
                # Handle multi-class vs binary classification
                if batch_probas.ndim > 1:
                    # Multi-class: probabilities are already in correct
         format
                    pass  # batch_probas is already correct
                else:
                    # Binary classification: convert to 2-column format
                    batch_probas = np.vstack((1 - batch_probas,
         batch_probas)).T
            else:
                # For sklearn API
                batch_probas = self.model.predict_proba(batch_X)

            probabilities.append(batch_probas)

            # Force garbage collection after each batch
            gc.collect()

        return np.concatenate(probabilities)

    def evaluate(self, X_test, y_test, batch_size=None):
        """Evaluate model on test data."""
        from evaluation.metrics import evaluate_classification_model
        return evaluate_classification_model(self, X_test, y_test, "XGBoost")

    def get_feature_importance(self, feature_names=None):
        """Get feature importance."""
        if self.feature_importance is None:
            logger.warning("Feature importance not available. Fit model
         first.")
            return None

        if isinstance(self.feature_importance, dict):
            # For models trained with DMatrix API
            if feature_names is None:
                importance_df = pd.DataFrame({
                    'Feature': list(self.feature_importance.keys()),
                    'Importance': list(self.feature_importance.values())
```

```
                })
            else:
                # Match feature names with importance scores
                importance = [self.feature_importance.get(f'f{i}', 0) for i
 in range(len(feature_names))]
                importance_df = pd.DataFrame({
                    'Feature': feature_names,
                    'Importance': importance
                })
        else:
            # For models trained with sklearn API
            if feature_names is None:
                feature_names = [f"feature_{i}" for i in
 range(len(self.feature_importance))]

            importance_df = pd.DataFrame({
                'Feature': feature_names,
                'Importance': self.feature_importance
            })

        return importance_df.sort_values('Importance', ascending=False)

    def save_model(self, filepath):
        """Save the model to disk"""
        dump(self.model, filepath)
        logger.info(f"Model saved to {filepath}")

    def load_model(self, filepath):
        """Load a model from disk"""
        self.model = load(filepath)
        logger.info(f"Model loaded from {filepath}")
        return self


class XGBoostRegressor:
    """XGBoost regressor for premium prediction with memory optimization"""

    def __init__(self, params=None, use_gpu=True, **kwargs):
        if params is None:
            params = XGBOOST_REGRESSION_PARAMS.copy()
        else:
            params = params.copy()

        # Merge kwargs into params (sklearn clone compatibility)
        params.update(kwargs)

        # Set up GPU or CPU usage (modern approach like test.py)
        if use_gpu:
            params['tree_method'] = 'hist'      # Modern GPU approach
            params['device'] = 'cuda'           # GPU device specification
            logger.info("Using XGBoost with GPU acceleration
 (tree_method=hist, device=cuda)")
        else:
            # Use histogram-based algorithm on CPU
            params['tree_method'] = 'hist'
```

```
            params['device'] = 'cpu'           # Explicit CPU specification
            logger.info("Using XGBoost on CPU (tree_method=hist,
         device=cpu)")

        self.model = XGBRegressor(**params)
        self.params = params
        self.feature_importance = None
        self.use_gpu = use_gpu

    def get_params(self, deep=True):
        """Get parameters for this estimator (sklearn compatibility)"""
        return self.params.copy()

    def set_params(self, **params):
        """Set parameters for this estimator (sklearn compatibility)"""
        self.params.update(params)
        # Reinitialize model with new parameters
        self.__init__(self.params, self.use_gpu)
        return self

    def fit(self, X_train, y_train, eval_set=None, use_dmatrix=True,
         memory_efficient=True):
        """Fit model to training data with GPU support."""
        logger.info(f"Fitting XGBoost model on {'GPU' if self.use_gpu else
         'CPU'}...")

        # Set objective and eval_metric based on number of classes (for both
         DMatrix and sklearn API)
        num_classes = len(np.unique(y_train))
        if num_classes > 2:
            # Multi-class classification
            if 'objective' not in self.params:
                self.params['objective'] = 'multi:softprob'
            if 'eval_metric' not in self.params:
                self.params['eval_metric'] = 'mlogloss'
        else:
            # Binary classification
            if 'objective' not in self.params:
                self.params['objective'] = 'binary:logistic'
            if 'eval_metric' not in self.params:
                self.params['eval_metric'] = 'logloss'

        # Reinitialize model with correct objective
        self.model = XGBClassifier(**self.params)

        # Optimize memory usage if requested
        if memory_efficient and not self.use_gpu:  # Skip for GPU as we'll
         convert to DMatrix
            if isinstance(X_train, pd.DataFrame):
                # For pandas DataFrame, optimize numeric columns
                for col in X_train.select_dtypes(include=['float']).columns:
                    X_train[col] = pd.to_numeric(X_train[col],
         downcast='float')
                for col in X_train.select_dtypes(include=
         ['integer']).columns:
```

```
            X_train[col] = pd.to_numeric(X_train[col],
 downcast='integer')
    elif isinstance(X_train, np.ndarray):
        # For numpy array, optimize array dtype
        X_train = optimize_array_memory(X_train)

    if isinstance(y_train, np.ndarray):
        y_train = optimize_array_memory(y_train)

# For GPU training, we should always use DMatrix
if self.use_gpu:
    use_dmatrix = True

if use_dmatrix:
    # Using DMatrix for better memory efficiency and performance
    from xgboost import DMatrix, train

    # Create DMatrix object - this automatically handles device
 transfer for GPU training
    dtrain = DMatrix(X_train, label=y_train)

    # Setup evaluation set if provided
    eval_sets = []
    if eval_set is not None:
        for i, (X_eval, y_eval) in enumerate(eval_set):
            deval = DMatrix(X_eval, label=y_eval)
            eval_sets.append((deval, f'eval-{i}'))

    eval_sets.insert(0, (dtrain, 'train'))

    # Extract parameters for xgb.train
    train_params = self.params.copy()

    # Handle specific parameters not used by xgb.train
    if 'n_estimators' in train_params:
        num_boost_round = train_params.pop('n_estimators')
    else:
        num_boost_round = 100

    # Ensure regression-specific parameters
    train_params['objective'] = 'reg:squarederror'
    train_params['eval_metric'] = 'rmse'

    # Train the model
    logger.info(f"Training with {'GPU' if self.use_gpu else 'CPU'}
 using DMatrix API")
    self.model = train(
        train_params,
        dtrain,
        num_boost_round=num_boost_round,
        evals=eval_sets,
        early_stopping_rounds=50 if eval_set else None,
        verbose_eval=False  # Disable verbose to avoid metric errors
    )
```

```
        # Get feature importance
            self.feature_importance =
         self.model.get_score(importance_type='gain')

        else:
            # Using standard sklearn API (not recommended for GPU)
            if self.use_gpu:
                logger.warning("Using sklearn API with GPU may cause device
         mismatch warnings. Consider use_dmatrix=True.")

            if eval_set is not None:
                # With evaluation set, use early stopping
                self.model.fit(
                X_train, y_train,
                eval_set=eval_set,
                eval_metric='rmse',  # Change from 'auc' to 'rmse' for
         regression
                early_stopping_rounds=50,
                verbose=True
                )
            else:
                # Without evaluation set, simple fit
                self.model.fit(X_train, y_train)

            # Get feature importance
            self.feature_importance = self.model.feature_importances_

        # Force garbage collection to free memory
        gc.collect()

        return self

    def predict(self, X, batch_size=None):
        """Make predictions, optionally in batches."""
        is_booster = hasattr(self.model, 'feature_names')

        if batch_size is None:
            if is_booster:
                # For Booster objects
                from xgboost import DMatrix
                X_dmatrix = DMatrix(X)
                return self.model.predict(X_dmatrix)
            else:
                # For sklearn API
                return self.model.predict(X)

        # Batch processing for large datasets
        n_samples = X.shape[0]
        n_batches = int(np.ceil(n_samples / batch_size))

        predictions = []
        for i in range(n_batches):
            start_idx = i * batch_size
            end_idx = min((i + 1) * batch_size, n_samples)
```

```
            logger.info(f"Processing prediction batch {i+1}/{n_batches}")
            batch_X = X[start_idx:end_idx]

            if is_booster:
                # For Booster objects
                from xgboost import DMatrix
                batch_X_dmatrix = DMatrix(batch_X)
                batch_pred = self.model.predict(batch_X_dmatrix)
            else:
                # For sklearn API
                batch_pred = self.model.predict(batch_X)

            predictions.append(batch_pred)

            # Force garbage collection after each batch
            gc.collect()

        return np.concatenate(predictions)

    def evaluate(self, X_test, y_test, batch_size=None):
        """Evaluate model on test data."""
        from evaluation.metrics import evaluate_regression_model
        return evaluate_regression_model(self, X_test, y_test, "XGBoost
         Regressor")

    def get_feature_importance(self, feature_names=None):
        """Get feature importance."""
        if self.feature_importance is None:
            logger.warning("Feature importance not available. Fit model
         first.")
            return None

        if isinstance(self.feature_importance, dict):
            # For models trained with DMatrix API
            if feature_names is None:
                importance_df = pd.DataFrame({
                    'Feature': list(self.feature_importance.keys()),
                    'Importance': list(self.feature_importance.values())
                })
            else:
                # Match feature names with importance scores
                importance = [self.feature_importance.get(f'f{i}', 0) for i
         in range(len(feature_names))]
                importance_df = pd.DataFrame({
                    'Feature': feature_names,
                    'Importance': importance
                })
        else:
            # For models trained with sklearn API
            if feature_names is None:
                feature_names = [f"feature_{i}" for i in
         range(len(self.feature_importance))]

            importance_df = pd.DataFrame({
                'Feature': feature_names,
                'Importance': self.feature_importance
```

```
            })

        return importance_df.sort_values('Importance', ascending=False)

    def save_model(self, filepath):
        """Save the model to disk"""
        dump(self.model, filepath)
        logger.info(f"Model saved to {filepath}")

    def load_model(self, filepath):
        """Load a model from disk"""
        self.model = load(filepath)
        logger.info(f"Model loaded from {filepath}")
        return self


class SVMClassifier:
    """SVM classifier for fraud detection with GPU acceleration via scikit-
        learn-intelex and memory optimization"""

    def __init__(self, force_cpu=False, large_dataset_threshold=5000,
        **svm_params):
        """Initialize SVM classifier."""
        self.force_cpu = force_cpu
        self.large_dataset_threshold = large_dataset_threshold

        # Set default SVM parameters if none provided
        default_params = SVM_CLASSIFICATION_PARAMS.copy()
        self.svm_params = {**default_params, **svm_params}

        self.model = None
        self.is_linear_svm = False

        # Model will be initialized in fit() based on dataset size

    def get_params(self, deep=True):
        """Get parameters for this estimator (sklearn compatibility)"""
        params = {
            'force_cpu': self.force_cpu,
            'large_dataset_threshold': self.large_dataset_threshold
        }
        # Add SVM parameters directly (not nested)
        params.update(self.svm_params)
        return params

    def set_params(self, **params):
        """Set parameters for this estimator (sklearn compatibility)"""
        # Extract our custom parameters
        if 'force_cpu' in params:
            self.force_cpu = params.pop('force_cpu')
        if 'large_dataset_threshold' in params:
            self.large_dataset_threshold =
        params.pop('large_dataset_threshold')

        # Strip 'estimator__' prefix from parameter names (added by
        GridSearchCV/Pipeline)
```

```
        cleaned_params = {}
        for key, value in params.items():
            if key.startswith('estimator__'):
                cleaned_params[key.replace('estimator__', '')] = value
            else:
                cleaned_params[key] = value

        # Remaining parameters are SVM parameters
        self.svm_params.update(cleaned_params)
        return self

    def fit(self, X_train, y_train, memory_efficient=True):
        """Fit model to training data."""
        n_samples = X_train.shape[0]
        n_features = X_train.shape[1]

        logger.info(f"Dataset size: {n_samples} samples, {n_features}
         features")

        # Always use RBF kernel SVM for all datasets
        logger.info(f"Using SVM with RBF kernel for {n_samples} samples")
        self.is_linear_svm = False

        # Try GPU acceleration in order of preference
        if not self.force_cpu:
            # First try scikit-learn-intelex GPU acceleration
            if GPU_ACCELERATION:
                self.model = SVC(**self.svm_params)
                logger.info("Using GPU-accelerated SVM via scikit-learn-
         intelex")
            # Second try ThunderSVM
            elif THUNDERSVM_AVAILABLE:
                try:
                    from thundersvm import SVC as ThunderSVC
                    # Add GPU ID if not specified
                    thunder_params = self.svm_params.copy()
                    if 'gpu_id' not in thunder_params:
                        thunder_params['gpu_id'] = 0
                    self.model = ThunderSVC(**thunder_params)
                    logger.info("Using GPU-accelerated SVM via ThunderSVM")
                except Exception as e:
                    logger.warning(f"ThunderSVM failed, falling back to CPU:
         {e}")
                    params_cpu = self.svm_params.copy()
                    if 'gpu_id' in params_cpu:
                        del params_cpu['gpu_id']
                    self.model = SVC(**params_cpu)
                    logger.info("Using standard CPU-based SVM")
            # Fallback to CPU
            else:
                params_cpu = self.svm_params.copy()
                if 'gpu_id' in params_cpu:
                    del params_cpu['gpu_id']
                self.model = SVC(**params_cpu)
                logger.info("Using standard CPU-based SVM")
```

```
else:
    # Force CPU usage
    params_cpu = self.svm_params.copy()
    if 'gpu_id' in params_cpu:
        del params_cpu['gpu_id']
    self.model = SVC(**params_cpu)
    logger.info("Using standard CPU-based SVM (forced)")

logger.info("Fitting SVM classification model...")
log_memory_usage("before SVM fit")

# Optimize memory usage if requested
if memory_efficient:
    if isinstance(X_train, pd.DataFrame):
        # For pandas DataFrame, optimize numeric columns
        for col in X_train.select_dtypes(include=['float']).columns:
            X_train[col] = pd.to_numeric(X_train[col],
 downcast='float')
        for col in X_train.select_dtypes(include=
 ['integer']).columns:
            X_train[col] = pd.to_numeric(X_train[col],
 downcast='integer')

        # Convert to numpy array
        X_train = X_train.to_numpy()
    elif isinstance(X_train, np.ndarray):
        # For numpy array, optimize array dtype if not needed for SVM
        if not self.is_linear_svm and not
 self.svm_params.get('kernel', 'rbf') == 'linear':
            # For non-linear kernels, keep float64 for precision
            pass
        else:
            X_train = optimize_array_memory(X_train)

    if isinstance(y_train, np.ndarray):
        y_train = optimize_array_memory(y_train)

    # Convert to numpy arrays (required for SVM)
    if not isinstance(X_train, np.ndarray):
        X_train = np.array(X_train)
    if not isinstance(y_train, np.ndarray):
        y_train = np.array(y_train)

    # Ensure 64-bit float precision for better GPU performance with
 SVM (not needed for LinearSVC)
    if X_train.dtype != np.float64 and GPU_ACCELERATION and not
 self.is_linear_svm:
        X_train = X_train.astype(np.float64)

# Fit model
self.model.fit(X_train, y_train)

# Force garbage collection to free memory
gc.collect()
log_memory_usage("after SVM fit")
```

```
        return self

    def predict(self, X, batch_size=None):
        """Make predictions, optionally in batches."""
        if not isinstance(X, np.ndarray):
            X = np.array(X)

        # Ensure 64-bit float precision for better GPU performance with SVM
        if X.dtype != np.float64 and GPU_ACCELERATION:
            X = X.astype(np.float64)

        if batch_size is None:
            return self.model.predict(X)

        # Batch processing for large datasets
        n_samples = X.shape[0]
        n_batches = int(np.ceil(n_samples / batch_size))

        predictions = []
        for i in range(n_batches):
            start_idx = i * batch_size
            end_idx = min((i + 1) * batch_size, n_samples)

            logger.info(f"Processing SVM prediction batch {i+1}/{n_batches}")
            batch_pred = self.model.predict(X[start_idx:end_idx])
            predictions.append(batch_pred)

            # Force garbage collection after each batch
            gc.collect()

        return np.concatenate(predictions)

    def predict_proba(self, X, batch_size=None):
        """Predict class probabilities, optionally in batches."""
        if not isinstance(X, np.ndarray):
            X = np.array(X)

        # Ensure 64-bit float precision for better GPU performance
        if X.dtype != np.float64 and GPU_ACCELERATION:
            X = X.astype(np.float64)

        # LinearSVC doesn't support predict_proba, use decision function
        if self.is_linear_svm:
            logger.info("Using decision function for LinearSVC probability
         estimates")

            if batch_size is None:
                decision = self.model.decision_function(X)
            else:
                # Batch processing for decision function
                n_samples = X.shape[0]
                n_batches = int(np.ceil(n_samples / batch_size))

                decisions = []
                for i in range(n_batches):
```

```
                start_idx = i * batch_size
                end_idx = min((i + 1) * batch_size, n_samples)

                logger.info(f"Processing LinearSVC decision batch
 {i+1}/{n_batches}")
                batch_decision =
 self.model.decision_function(X[start_idx:end_idx])
                decisions.append(batch_decision)

                # Force garbage collection after each batch
                gc.collect()

            decision = np.concatenate(decisions)

        # Convert decision function to probabilities using sigmoid
        from scipy.special import expit
        prob_pos = expit(decision)
        prob_neg = 1 - prob_pos
        return np.column_stack((prob_neg, prob_pos))

    # For regular SVM, try predict_proba
    try:
        if batch_size is None:
            return self.model.predict_proba(X)

        # Batch processing for large datasets
        n_samples = X.shape[0]
        n_batches = int(np.ceil(n_samples / batch_size))

        probabilities = []
        for i in range(n_batches):
            start_idx = i * batch_size
            end_idx = min((i + 1) * batch_size, n_samples)

            logger.info(f"Processing SVM probability batch
 {i+1}/{n_batches}")
            batch_proba = self.model.predict_proba(X[start_idx:end_idx])
            probabilities.append(batch_proba)

            # Force garbage collection after each batch
            gc.collect()

        return np.concatenate(probabilities)

    except AttributeError:
        # In case probability was not enabled
        logger.warning("Probability estimates not enabled. Using decision
 function.")

        if batch_size is None:
            decision = self.model.decision_function(X)
        else:
            # Batch processing for decision function
            n_samples = X.shape[0]
            n_batches = int(np.ceil(n_samples / batch_size))
```

```
                decisions = []
                for i in range(n_batches):
                    start_idx = i * batch_size
                    end_idx = min((i + 1) * batch_size, n_samples)

                    logger.info(f"Processing SVM decision batch
          {i+1}/{n_batches}")
                    batch_decision =
          self.model.decision_function(X[start_idx:end_idx])
                    decisions.append(batch_decision)

                    # Force garbage collection after each batch
                    gc.collect()

                decision = np.concatenate(decisions)

# Convert decision function to pseudo-probabilities
            proba = 1 / (1 + np.exp(-decision))
            return np.column_stack((1 - proba, proba))

    def evaluate(self, X_test, y_test, batch_size=None):
        """Evaluate model on test data."""
        from evaluation.metrics import evaluate_classification_model
        return evaluate_classification_model(self, X_test, y_test, "SVM")

    def save_model(self, filepath):
        """Save the model to disk"""
        dump(self.model, filepath)
        logger.info(f"Model saved to {filepath}")

    def load_model(self, filepath):
        """Load a model from disk"""
        self.model = load(filepath)
        logger.info(f"Model loaded from {filepath}")
        return self


class SVMRegressor:
    """SVM regressor for premium prediction with GPU acceleration via scikit-
         learn-intelex and memory optimization"""

    def __init__(self, force_cpu=False, large_dataset_threshold=50000,
         **svm_params):
        """Initialize SVM Regressor."""
        self.force_cpu = force_cpu
        self.large_dataset_threshold = large_dataset_threshold

        # Set default SVM parameters if none provided
        default_params = SVM_REGRESSION_PARAMS.copy()
        self.svm_params = {**default_params, **svm_params}

        self.model = None
        self.is_linear_svm = False

        # Model will be initialized in fit() based on dataset size
```

```
    def get_params(self, deep=True):
        """Get parameters for this estimator (sklearn compatibility)"""
        params = {
            'force_cpu': self.force_cpu,
            'large_dataset_threshold': self.large_dataset_threshold
        }
        # Add SVM parameters directly (not nested)
        params.update(self.svm_params)
        return params

    def set_params(self, **params):
        """Set parameters for this estimator (sklearn compatibility)"""
        # Extract our custom parameters
        if 'force_cpu' in params:
            self.force_cpu = params.pop('force_cpu')
        if 'large_dataset_threshold' in params:
            self.large_dataset_threshold =
         params.pop('large_dataset_threshold')

        # Strip 'estimator__' prefix from parameter names (added by
         GridSearchCV/Pipeline)
        cleaned_params = {}
        for key, value in params.items():
            if key.startswith('estimator__'):
                cleaned_params[key.replace('estimator__', '')] = value
            else:
                cleaned_params[key] = value

        # Remaining parameters are SVM parameters
        self.svm_params.update(cleaned_params)
        return self

    def fit(self, X_train, y_train, memory_efficient=True):
        """Fit model to training data."""
        n_samples = X_train.shape[0]
        n_features = X_train.shape[1]

        logger.info(f"Dataset size: {n_samples} samples, {n_features}
         features")

        # ALWAYS use LinearSVR for speed - regardless of dataset size
        if True:  # Force LinearSVR for all datasets
            logger.info(f"Using LinearSVR for optimal speed (dataset:
         {n_samples} samples, {n_features} features)")
            self.is_linear_svm = True

            # Create LinearSVR parameters (remove kernel-specific and
         classification-specific params)
            linear_params = self.svm_params.copy()
            if 'kernel' in linear_params:
                del linear_params['kernel']
            if 'gamma' in linear_params:
                del linear_params['gamma']
            if 'degree' in linear_params:
                del linear_params['degree']
            if 'gpu_id' in linear_params:
```

```
            del linear_params['gpu_id']
        if 'class_weight' in linear_params:
            del linear_params['class_weight']
        if 'probability' in linear_params:
            del linear_params['probability']

        # Map SVR parameters to LinearSVR
        if 'C' not in linear_params:
            linear_params['C'] = 1.0
        linear_params['max_iter'] = linear_params.get('max_iter', 5000)
        linear_params['dual'] = 'auto'  # Use auto dual selection for
    LinearSVR

        self.model = LinearSVR(**linear_params)
        logger.info("Using LinearSVR for large dataset optimization")

    else:
        # Use regular SVR for smaller datasets
        logger.info("Using standard SVR with kernel methods")
        self.is_linear_svm = False

        # Try GPU acceleration in order of preference
        if not self.force_cpu:
            # First try scikit-learn-intelex GPU acceleration
            if GPU_ACCELERATION:
                self.model = SVR(**self.svm_params)
                logger.info("Using GPU-accelerated SVR via scikit-learn-
    intelex")
            # Second try ThunderSVM
            elif THUNDERSVM_AVAILABLE:
                try:
                    from thundersvm import SVR as ThunderSVR
                    # Add GPU ID if not specified
                    thunder_params = self.svm_params.copy()
                    if 'gpu_id' not in thunder_params:
                        thunder_params['gpu_id'] = 0
                    self.model = ThunderSVR(**thunder_params)
                    logger.info("Using GPU-accelerated SVR via
    ThunderSVM")
                except Exception as e:
                    logger.warning(f"ThunderSVM failed, falling back to
    CPU: {e}")
                    params_cpu = self.svm_params.copy()
                    if 'gpu_id' in params_cpu:
                        del params_cpu['gpu_id']
                    self.model = SVR(**params_cpu)
                    logger.info("Using standard CPU-based SVR")
            # Fallback to CPU
            else:
                params_cpu = self.svm_params.copy()
                if 'gpu_id' in params_cpu:
                    del params_cpu['gpu_id']
                self.model = SVR(**params_cpu)
                logger.info("Using standard CPU-based SVR")
        else:
            # Force CPU usage
```

```
            params_cpu = self.svm_params.copy()
            if 'gpu_id' in params_cpu:
                del params_cpu['gpu_id']
            self.model = SVR(**params_cpu)
            logger.info("Using standard CPU-based SVR (forced)")

    logger.info("Fitting SVM regression model...")
    log_memory_usage("before SVM fit")

    # Optimize memory usage if requested
    if memory_efficient:
        if isinstance(X_train, pd.DataFrame):
            # For pandas DataFrame, optimize numeric columns
            for col in X_train.select_dtypes(include=['float']).columns:
                X_train[col] = pd.to_numeric(X_train[col],
     downcast='float')
            for col in X_train.select_dtypes(include=
     ['integer']).columns:
                X_train[col] = pd.to_numeric(X_train[col],
     downcast='integer')

            # Convert to numpy array
            X_train = X_train.to_numpy()
        elif isinstance(X_train, np.ndarray):
            # For numpy array, optimize array dtype if not needed for SVM
            if not self.is_linear_svm and not
     self.svm_params.get('kernel', 'rbf') == 'linear':
                # For non-linear kernels, keep float64 for precision
                pass
            else:
                X_train = optimize_array_memory(X_train)

        if isinstance(y_train, np.ndarray):
            y_train = optimize_array_memory(y_train)

        # Convert to numpy arrays (required for SVM)
        if not isinstance(X_train, np.ndarray):
            X_train = np.array(X_train)
        if not isinstance(y_train, np.ndarray):
            y_train = np.array(y_train)

        # Ensure 64-bit float precision for better GPU performance with
     SVM (not needed for LinearSVR)
        if X_train.dtype != np.float64 and GPU_ACCELERATION and not
     self.is_linear_svm:
            X_train = X_train.astype(np.float64)

    # Fit model
    self.model.fit(X_train, y_train)

    # Force garbage collection to free memory
    gc.collect()
    log_memory_usage("after SVM fit")

    return self
```

```
    def predict(self, X, batch_size=None):
        """Make predictions, optionally in batches."""
        if not isinstance(X, np.ndarray):
            X = np.array(X)

        # Ensure 64-bit float precision for better GPU performance with SVM
        if X.dtype != np.float64 and GPU_ACCELERATION:
            X = X.astype(np.float64)

        if batch_size is None:
            return self.model.predict(X)

        # Batch processing for large datasets
        n_samples = X.shape[0]
        n_batches = int(np.ceil(n_samples / batch_size))

        predictions = []
        for i in range(n_batches):
            start_idx = i * batch_size
            end_idx = min((i + 1) * batch_size, n_samples)

            logger.info(f"Processing SVM prediction batch {i+1}/{n_batches}")
            batch_pred = self.model.predict(X[start_idx:end_idx])
            predictions.append(batch_pred)

            # Force garbage collection after each batch
            gc.collect()

        return np.concatenate(predictions)

    def evaluate(self, X_test, y_test, batch_size=None):
        """Evaluate model on test data."""
        from evaluation.metrics import evaluate_regression_model
        return evaluate_regression_model(self, X_test, y_test, "SVM
         Regressor")

    def save_model(self, filepath):
        """Save the model to disk"""
        dump(self.model, filepath)
        logger.info(f"Model saved to {filepath}")

    def load_model(self, filepath):
        """Load a model from disk"""
        self.model = load(filepath)
        logger.info(f"Model loaded from {filepath}")
        return self


# Helper function to create a DataLoader for chunked dataset processing
def create_data_chunker(file_path, chunk_size=100000, target_column=None,
         feature_columns=None):
    """Create a generator for processing large datasets in chunks."""
    try:
        total_rows = sum(1 for _ in open(file_path)) - 1  # Subtract header
        logger.info(f"Total rows in dataset: {total_rows}")
```

```
        except Exception as e:
            logger.warning(f"Could not count rows: {e}")
            total_rows = None

        # Process file in chunks
        chunk_iter = pd.read_csv(file_path, chunksize=chunk_size)

        for i, chunk in enumerate(chunk_iter):
            if total_rows:
                progress = min(100, (i * chunk_size + len(chunk)) / total_rows *
             100)
                logger.info(f"Processing chunk {i+1}, progress: {progress:.1f}%")
            else:
                logger.info(f"Processing chunk {i+1}")

            # Optimize memory usage
            for col in chunk.select_dtypes(include=['float']).columns:
                chunk[col] = pd.to_numeric(chunk[col], downcast='float')
            for col in chunk.select_dtypes(include=['integer']).columns:
                chunk[col] = pd.to_numeric(chunk[col], downcast='integer')

            # Select features and target
            if target_column is not None:
                if feature_columns is None:
                    feature_columns = [col for col in chunk.columns if col !=
             target_column]

                X_chunk = chunk[feature_columns]
                y_chunk = chunk[target_column]

                yield X_chunk, y_chunk
            else:
                yield chunk, None

            # Force garbage collection
            del chunk
            gc.collect()


    class TabNetRegressorWrapper:
        """
        Wrapper for TabNet regressor to integrate with existing pipeline.
        """

        def __init__(self, params=None):
            default_params = {
                'n_d': 8,
                'n_a': 8,
                'n_steps': 3,
                'gamma': 1.3,
                'n_independent': 2,
                'n_shared': 2,
                'epsilon': 1e-15,
                'momentum': 0.02,
                'lambda_sparse': 1e-3,
```

```
            'learning_rate': 0.02,
            'batch_size': 256,
            'epochs': 100,
            'patience': 10,
            'device': 'cuda' if torch.cuda.is_available() else 'cpu',
            'random_seed': 42
        }

        if params is not None:
            default_params.update(params)

        self.model = TabNetRegressor(default_params)
        self.params = default_params
        self.feature_importance = None

    def fit(self, X_train, y_train):
        """Fit TabNet model to training data."""
        logger.info("Fitting TabNet regressor...")

        # Monitor memory usage
        initial_memory = get_memory_usage()
        logger.info(f"Initial memory usage: {initial_memory:.2f} MB")

        try:
            # Fit the model
            self.model.fit(X_train, y_train)

            # Store feature importance
            self.feature_importance = self.model.feature_importance

            # Final memory check
            final_memory = get_memory_usage()
            logger.info(f"Final memory usage: {final_memory:.2f} MB")
            logger.info(f"Memory increase: {final_memory -
         initial_memory:.2f} MB")

            logger.info("TabNet regressor training completed successfully")

        except Exception as e:
            logger.error(f"Error during TabNet training: {str(e)}")
            # Try to free memory
            gc.collect()
            raise

        return self

    def predict(self, X_test):
        """Make predictions using the trained TabNet model."""
        if self.model.model is None:
            raise ValueError("Model has not been trained yet. Call fit()
         first.")

        try:
            predictions = self.model.predict(X_test)
            return predictions
```

```
        except Exception as e:
            logger.error(f"Error during TabNet prediction: {str(e)}")
            raise

    def evaluate(self, X_test, y_test):
        """Evaluate model performance on test data."""
        from evaluation.metrics import evaluate_regression_model
        return evaluate_regression_model(self, X_test, y_test, "TabNet
         Regressor")

    def get_feature_importance(self, feature_names=None):
        """Get feature importance scores."""
        if self.feature_importance is None:
            logger.warning("Feature importance not available. Model may not
         be trained.")
            return None

        return self.model.get_feature_importance(feature_names)

    def get_training_history(self):
        """Get training history (loss curves)."""
        return self.model.training_history

    def save_model(self, filepath):
        """Save the trained TabNet model."""
        try:
            # Save using PyTorch's native saving
            import torch
            torch.save({
                'model_state_dict': self.model.model.state_dict(),
                'params': self.params,
                'feature_importance': self.feature_importance,
                'training_history': self.model.training_history
            }, filepath)
            logger.info(f"TabNet model saved to {filepath}")
        except Exception as e:
            logger.error(f"Error saving TabNet model: {str(e)}")
            raise

    def load_model(self, filepath):
        """Load a trained TabNet model."""
        try:
            import torch
            checkpoint = torch.load(filepath)

            # Restore parameters and initialize model
            self.params = checkpoint['params']
            self.model = TabNetRegressor(self.params)

            # Load model state
            # Note: This requires the model structure to be initialized first
            logger.warning("Model loading requires manual structure
         initialization")

            self.feature_importance = checkpoint.get('feature_importance')
```

```
            self.model.training_history = checkpoint.get('training_history',
         {'train_loss': [], 'val_loss': []})

            logger.info(f"TabNet model loaded from {filepath}")
        except Exception as e:
            logger.error(f"Error loading TabNet model: {str(e)}")
            raise


class ResNetClassifierWrapper:
    """
    Wrapper for ResNet classifier to integrate with existing pipeline.
    """

    def __init__(self, params=None):
        default_params = {
            'hidden_size': 256,
            'num_blocks': 4,
            'block_depth': 2,
            'dropout_rate': 0.3,
            'learning_rate': 0.001,
            'batch_size': 64,
            'epochs': 100,
            'patience': 15,
            'weight_decay': 1e-4,
            'optimizer': 'adam',
            'scheduler': 'cosine',
            'device': 'cuda' if torch.cuda.is_available() else 'cpu',
            'random_seed': 42,
            'class_weight': True
        }

        if params is not None:
            default_params.update(params)

        self.model = ResNetClassifier(default_params)
        self.params = default_params
        self.feature_importance = None

    def fit(self, X_train, y_train):
        """Fit ResNet model to training data."""
        logger.info("Fitting ResNet classifier...")

        # Monitor memory usage
        initial_memory = get_memory_usage()
        logger.info(f"Initial memory usage: {initial_memory:.2f} MB")

        try:
            # Fit the model
            self.model.fit(X_train, y_train)

            # Store feature importance
            self.feature_importance = self.model.feature_importance

            # Final memory check
```

```
            final_memory = get_memory_usage()
            logger.info(f"Final memory usage: {final_memory:.2f} MB")
            logger.info(f"Memory increase: {final_memory -
         initial_memory:.2f} MB")

            logger.info("ResNet classifier training completed successfully")

        except Exception as e:
            logger.error(f"Error during ResNet training: {str(e)}")
            # Try to free memory
            gc.collect()
            raise

        return self

    def predict(self, X_test):
        """Make predictions using the trained ResNet model."""
        if self.model.model is None:
            raise ValueError("Model has not been trained yet. Call fit()
         first.")

        try:
            predictions = self.model.predict(X_test)
            return predictions
        except Exception as e:
            logger.error(f"Error during ResNet prediction: {str(e)}")
            raise

    def predict_proba(self, X_test):
        """Predict class probabilities."""
        if self.model.model is None:
            raise ValueError("Model has not been trained yet. Call fit()
         first.")

        try:
            probabilities = self.model.predict_proba(X_test)
            return probabilities
        except Exception as e:
            logger.error(f"Error during ResNet probability prediction:
         {str(e)}")
            raise

    def evaluate(self, X_test, y_test):
        """Evaluate model performance on test data."""
        try:
            metrics = self.model.evaluate(X_test, y_test)

            logger.info("ResNet evaluation metrics:")
            for metric, value in metrics.items():
                if isinstance(value, (int, float)):
                    logger.info(f"  {metric}: {value:.6f}")

            return metrics

        except Exception as e:
            logger.error(f"Error during ResNet evaluation: {str(e)}")
```

```
            raise

    def get_feature_importance(self, feature_names=None):
        """Get feature importance scores."""
        if self.feature_importance is None:
            logger.warning("Feature importance not available. Model may not
         be trained.")
            return None

        return self.model.get_feature_importance(feature_names)

    def get_training_history(self):
        """Get training history (loss and accuracy curves)."""
        return self.model.training_history

    def save_model(self, filepath):
        """Save the trained ResNet model."""
        try:
            import torch
            torch.save({
                'model_state_dict': self.model.model.state_dict(),
                'params': self.params,
                'feature_importance': self.feature_importance,
                'training_history': self.model.training_history,
                'label_encoder': self.model.label_encoder,
                'num_classes': self.model.num_classes
            }, filepath)
            logger.info(f"ResNet model saved to {filepath}")
        except Exception as e:
            logger.error(f"Error saving ResNet model: {str(e)}")
            raise

    def load_model(self, filepath):
        """Load a trained ResNet model."""
        try:
            import torch
            checkpoint = torch.load(filepath)

            # Restore parameters and initialize model
            self.params = checkpoint['params']
            self.model = ResNetClassifier(self.params)

            # Load model state
            # Note: This requires the model structure to be initialized first
            logger.warning("Model loading requires manual structure
         initialization")

            self.feature_importance = checkpoint.get('feature_importance')
            self.model.training_history = checkpoint.get('training_history',
         {
                'train_loss': [], 'val_loss': [], 'train_acc': [], 'val_acc':
         []
            })
            self.model.label_encoder = checkpoint.get('label_encoder')
            self.model.num_classes = checkpoint.get('num_classes')
```

```
            logger.info(f"ResNet model loaded from {filepath}")
        except Exception as e:
            logger.error(f"Error loading ResNet model: {str(e)}")
            raise
```

3.7: models/deep_learning.py

Location: `models/deep_learning.py`

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, TensorDataset
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, confusion_matrix, classification_report,
    r2_score, mean_squared_error, mean_absolute_error, median_absolute_error,
         explained_variance_score
)
import logging
import sys
import os

# Add parent directory to path
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from config import DL_PARAMS

# Set up logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %
         (message)s')
logger = logging.getLogger(__name__)

OPTIMIZER_MAP = {
    'sgd': 'SGD',
    'adam': 'Adam',
    'rmsprop': 'RMSprop',
    'adagrad': 'Adagrad',
    'adadelta': 'Adadelta'
}


# Check if CUDA is available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
logger.info(f"Using device: {device}")

# Check GPU availability
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Example of checking specific GPU details
if torch.cuda.is_available():
```

```
    print(f"GPU Name: {torch.cuda.get_device_name(0)}")
    print(f"Number of GPUs: {torch.cuda.device_count()}")

class MLP(nn.Module):
    """Multilayer Perceptron neural network"""

    def __init__(self, input_dim, hidden_layers, output_dim,
                 dropout_rate=0.3):
        super(MLP, self).__init__()

        layers = []
        prev_dim = input_dim

        # Add hidden layers
        for hidden_dim in hidden_layers:
            layers.append(nn.Linear(prev_dim, hidden_dim))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout_rate))
            prev_dim = hidden_dim

        # Add output layer
        layers.append(nn.Linear(prev_dim, output_dim))

        # Combine all layers
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        """Forward pass."""
        return self.model(x)


class DeepClassifier:
    """Deep neural network classifier for fraud detection"""

    def __init__(self, params=None):
        if params is None:
            params = DL_PARAMS

        self.params = params
        self.model = None
        self.input_dim = None
        self.output_dim = None  # Will be set dynamically based on number of
                                classes

    def _create_model(self, input_dim):
        """Create neural network model."""
        self.input_dim = input_dim
        model = MLP(
            input_dim=input_dim,
            hidden_layers=self.params['hidden_layers'],
            output_dim=self.output_dim,
            dropout_rate=self.params['dropout_rate']
        )

        # Move model to device (CPU or GPU)
```

```
        if self.params['use_gpu'] and torch.cuda.is_available():
            model = model.to(device)

        return model

    def _create_data_loader(self, X, y=None, batch_size=None, shuffle=True):
        """Create PyTorch DataLoader."""
        if batch_size is None:
            batch_size = self.params['batch_size']

        # Convert to numpy if pandas
        if hasattr(X, 'values'):
            X = X.values

        # Convert to PyTorch tensors
        X_tensor = torch.FloatTensor(X)

        # Move tensors to device
        if self.params['use_gpu'] and torch.cuda.is_available():
            X_tensor = X_tensor.to(device)

        if y is not None:
            # Convert to numpy if pandas
            if hasattr(y, 'values'):
                y = y.values

            # Handle multi-class vs binary classification
            if hasattr(self, 'num_classes') and self.num_classes > 2:
                # Multi-class: use LongTensor for class indices
                y_tensor = torch.LongTensor(y.flatten())  # Flatten to 1D for
         CrossEntropyLoss
            else:
                # Binary classification: use FloatTensor and reshape
                if len(y.shape) == 1:
                    y = y.reshape(-1, 1)
                y_tensor = torch.FloatTensor(y)
            if self.params['use_gpu'] and torch.cuda.is_available():
                y_tensor = y_tensor.to(device)
            dataset = TensorDataset(X_tensor, y_tensor)
        else:
            dataset = TensorDataset(X_tensor)

        return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)

    def fit(self, X_train, y_train, X_val=None, y_val=None):
        """Fit model to training data."""
        logger.info("Fitting MLP classification model...")

        # Determine number of classes dynamically
        self.num_classes = len(np.unique(y_train))
        self.output_dim = self.num_classes if self.num_classes > 2 else 1
        logger.info(f"Detected {self.num_classes} classes, output_dim=
         {self.output_dim}")

        # Create model
```

```
self.model = self._create_model(X_train.shape[1])

# Create data loaders
train_loader = self._create_data_loader(X_train, y_train,
 shuffle=True)

if X_val is not None and y_val is not None:
    val_loader = self._create_data_loader(X_val, y_val,
 shuffle=False)
    has_validation = True
else:
    has_validation = False

# Define loss function and optimizer based on number of classes
if self.num_classes > 2:
    # Multi-class classification
    criterion = nn.CrossEntropyLoss()
    # Convert targets to long integers for CrossEntropyLoss
    y_train = y_train.astype(np.int64)
    if y_val is not None:
        y_val = y_val.astype(np.int64)
else:
    # Binary classification
    criterion = nn.BCEWithLogitsLoss()

optimizer_name = OPTIMIZER_MAP.get(
self.params['optimizer'].lower(),
self.params['optimizer'].capitalize()
)
optimizer = getattr(optim, optimizer_name)(
self.model.parameters(), lr=self.params['learning_rate']
)

# Training loop
epochs = self.params['epochs']
best_val_loss = float('inf')
patience_counter = 0
patience = 10  # Early stopping patience

for epoch in range(epochs):
    # Training phase
    self.model.train()
    running_loss = 0.0

    for batch_X, batch_y in train_loader:
        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = self.model(batch_X)
        loss = criterion(outputs, batch_y)

        # Backward pass and optimize
        loss.backward()
        optimizer.step()
```

```
                # Accumulate loss
                running_loss += loss.item() * batch_X.size(0)

            epoch_loss = running_loss / len(train_loader.dataset)

            # Validation phase
            if has_validation:
                self.model.eval()
                val_loss = 0.0

                with torch.no_grad():
                    for batch_X, batch_y in val_loader:
                        outputs = self.model(batch_X)
                        loss = criterion(outputs, batch_y)
                        val_loss += loss.item() * batch_X.size(0)

                val_loss = val_loss / len(val_loader.dataset)

                # Early stopping
                if val_loss < best_val_loss:
                    best_val_loss = val_loss
                    patience_counter = 0
                    # Save best model
                    best_model_state = self.model.state_dict().copy()
                else:
                    patience_counter += 1

                if patience_counter >= patience:
                    logger.info(f"Early stopping at epoch {epoch+1}")
                    # Restore best model
                    self.model.load_state_dict(best_model_state)
                    break

                logger.info(f"Epoch {epoch+1}/{epochs} - Loss:
 {epoch_loss:.4f} - Val Loss: {val_loss:.4f}")
            else:
                logger.info(f"Epoch {epoch+1}/{epochs} - Loss:
 {epoch_loss:.4f}")

        return self

    def predict(self, X):
        """Make predictions."""
        if self.model is None:
            logger.error("Model not fitted yet.")
            return None

        # Create data loader
        test_loader = self._create_data_loader(X, shuffle=False)

        # Set model to evaluation mode
        self.model.eval()

        # Make predictions
```

```
        predictions = []

        with torch.no_grad():
            for batch_X in test_loader:
                outputs = self.model(batch_X[0])
                if hasattr(self, 'num_classes') and self.num_classes > 2:
                    # Multi-class: use argmax to get predicted class
                    preds = torch.argmax(outputs, dim=1)
                else:
                    # Binary classification: use sigmoid and threshold
                    preds = torch.sigmoid(outputs) >= 0.5
                predictions.append(preds.cpu().numpy())

        # Concatenate batches
        predictions = np.concatenate(predictions).flatten()

        return predictions

    def predict_proba(self, X):
        """Predict class probabilities."""
        if self.model is None:
            logger.error("Model not fitted yet.")
            return None

        # Create data loader
        test_loader = self._create_data_loader(X, shuffle=False)

        # Set model to evaluation mode
        self.model.eval()

        # Make predictions
        probabilities = []

        with torch.no_grad():
            for batch_X in test_loader:
                outputs = self.model(batch_X[0])
                if hasattr(self, 'num_classes') and self.num_classes > 2:
                    # Multi-class: use softmax to get class probabilities
                    probs = torch.softmax(outputs, dim=1).cpu().numpy()
                else:
                    # Binary classification: use sigmoid
                    probs = torch.sigmoid(outputs).cpu().numpy()
                probabilities.append(probs)

        # Concatenate batches
        probabilities = np.concatenate(probabilities)

        # Return probabilities in appropriate format
        if hasattr(self, 'num_classes') and self.num_classes > 2:
            # Multi-class: probabilities already in correct shape
            return probabilities
        else:
            # Binary classification: convert to 2-column format
            probabilities = probabilities.flatten()
            return np.column_stack((1 - probabilities, probabilities))
```

```
    def evaluate(self, X_test, y_test):
        """Evaluate model on test data."""
        from evaluation.metrics import evaluate_classification_model
        return evaluate_classification_model(self, X_test, y_test, "MLP")

    def get_params(self, deep=True):
        """Get parameters for this estimator (sklearn compatibility)"""
        return self.params.copy()

    def set_params(self, **params):
        """Set parameters for this estimator (sklearn compatibility)"""
        self.params.update(params)
        # Reinitialize model with new parameters
        self.__init__(self.params)
        return self


class DeepRegressor:
    """Deep neural network regressor for premium prediction"""

    def __init__(self, params=None):
        if params is None:
            params = DL_PARAMS

        self.params = params
        self.model = None
        self.input_dim = None
        self.output_dim = 1  # Regression

    def _create_model(self, input_dim):
        """Create neural network model."""
        self.input_dim = input_dim
        model = MLP(
            input_dim=input_dim,
            hidden_layers=self.params['hidden_layers'],
            output_dim=self.output_dim,
            dropout_rate=self.params['dropout_rate']
        )

        # Move model to device (CPU or GPU)
        if self.params['use_gpu'] and torch.cuda.is_available():
            model = model.to(device)

        return model

    def _create_data_loader(self, X, y=None, batch_size=None, shuffle=True):
        """Create PyTorch DataLoader."""
        if batch_size is None:
            batch_size = self.params['batch_size']

        # Convert to numpy if pandas
        if hasattr(X, 'values'):
            X = X.values
```

```
        # Convert to PyTorch tensors
        X_tensor = torch.FloatTensor(X)

        # Move tensors to device
        if self.params['use_gpu'] and torch.cuda.is_available():
            X_tensor = X_tensor.to(device)

        if y is not None:
            # Convert to numpy if pandas
            if hasattr(y, 'values'):
                y = y.values

            # Handle multi-class vs binary classification
            if hasattr(self, 'num_classes') and self.num_classes > 2:
                # Multi-class: use LongTensor for class indices
                y_tensor = torch.LongTensor(y.flatten())  # Flatten to 1D for
         CrossEntropyLoss
            else:
                # Binary classification: use FloatTensor and reshape
                if len(y.shape) == 1:
                    y = y.reshape(-1, 1)
                y_tensor = torch.FloatTensor(y)
            if self.params['use_gpu'] and torch.cuda.is_available():
                y_tensor = y_tensor.to(device)
            dataset = TensorDataset(X_tensor, y_tensor)
        else:
            dataset = TensorDataset(X_tensor)

        return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)

    def fit(self, X_train, y_train, X_val=None, y_val=None):
        """Fit model to training data."""
        logger.info("Fitting MLP regression model...")

        # Create model
        self.model = self._create_model(X_train.shape[1])

        # Create data loaders
        train_loader = self._create_data_loader(X_train, y_train,
         shuffle=True)

        if X_val is not None and y_val is not None:
            val_loader = self._create_data_loader(X_val, y_val,
         shuffle=False)
            has_validation = True
        else:
            has_validation = False

        # Define loss function and optimizer
        criterion = nn.MSELoss()
        optimizer_name = OPTIMIZER_MAP.get(
        self.params['optimizer'].lower(),
        self.params['optimizer'].capitalize()
        )
        optimizer = getattr(optim, optimizer_name)(
```

```
self.model.parameters(), lr=self.params['learning_rate']
)

# Training loop
epochs = self.params['epochs']
best_val_loss = float('inf')
patience_counter = 0
patience = 10  # Early stopping patience

for epoch in range(epochs):
    # Training phase
    self.model.train()
    running_loss = 0.0

    for batch_X, batch_y in train_loader:
        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = self.model(batch_X)
        loss = criterion(outputs, batch_y)

        # Backward pass and optimize
        loss.backward()
        optimizer.step()

        # Accumulate loss
        running_loss += loss.item() * batch_X.size(0)

    epoch_loss = running_loss / len(train_loader.dataset)

    # Validation phase
    if has_validation:
        self.model.eval()
        val_loss = 0.0

        with torch.no_grad():
            for batch_X, batch_y in val_loader:
                outputs = self.model(batch_X)
                loss = criterion(outputs, batch_y)
                val_loss += loss.item() * batch_X.size(0)

        val_loss = val_loss / len(val_loader.dataset)

        # Early stopping
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            patience_counter = 0
            # Save best model
            best_model_state = self.model.state_dict().copy()
        else:
            patience_counter += 1

        if patience_counter >= patience:
            logger.info(f"Early stopping at epoch {epoch+1}")
```

```
                # Restore best model
                self.model.load_state_dict(best_model_state)
                break
            if torch.isnan(loss).any():
                print("NaN detected in loss!")
                break

            logger.info(f"Epoch {epoch+1}/{epochs} - Loss:
 {epoch_loss:.4f} - Val Loss: {val_loss:.4f}")
        else:
            logger.info(f"Epoch {epoch+1}/{epochs} - Loss:
 {epoch_loss:.4f}")

    return self

def predict(self, X):
    """Make predictions."""
    if self.model is None:
        logger.error("Model not fitted yet.")
        return None

    # Create data loader
    test_loader = self._create_data_loader(X, shuffle=False)

    # Set model to evaluation mode
    self.model.eval()

    # Make predictions
    predictions = []

    with torch.no_grad():
        for batch_X in test_loader:
            outputs = self.model(batch_X[0])
            predictions.append(outputs.cpu().numpy())

    # Concatenate batches
    predictions = np.concatenate(predictions).flatten()

    return predictions

def evaluate(self, X_test, y_test):
    """Evaluate model on test data."""
    from evaluation.metrics import evaluate_regression_model
    return evaluate_regression_model(self, X_test, y_test, "MLP")

def get_params(self, deep=True):
    """Get parameters for this estimator (sklearn compatibility)"""
    return self.params.copy()

def set_params(self, **params):
    """Set parameters for this estimator (sklearn compatibility)"""
    self.params.update(params)
    # Reinitialize model with new parameters
    self.__init__(self.params)
    return self
```

3.8: models/gpu_base_models.py

Location: `models/gpu_base_models.py`

```
"""
GPU-accelerated versions of base models using PyTorch
"""

import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, r2_score, mean_squared_error, mean_absolute_error,
         explained_variance_score
)
import torch
import torch.nn as nn
import torch.optim as optim
import logging
import sys
import os

# Add parent directory to path
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

# Set up logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %
         (message)s')
logger = logging.getLogger(__name__)


class GPULogisticRegression:
    """GPU-accelerated Logistic Regression using PyTorch"""

    def __init__(self, params=None, use_gpu=True):
        if params is None:
            params = {}

        self.learning_rate = params.get('learning_rate', 0.01)
        self.max_iter = params.get('max_iter', 1000)
        self.tol = params.get('tol', 1e-6)
        self.batch_size = params.get('batch_size', 10000)
        self.use_gpu = use_gpu and torch.cuda.is_available()

        self.device = torch.device('cuda' if self.use_gpu else 'cpu')
        self.model = None
        self.scaler = StandardScaler()
        self.feature_importance = None

        if self.use_gpu:
            logger.info("Using GPU acceleration for Logistic Regression")
        else:
```

```
            logger.info("Using CPU for Logistic Regression")

    def fit(self, X_train, y_train):
        """Fit model to training data."""
        logger.info("Fitting GPU logistic regression model...")

        # Normalize features
        X_scaled = self.scaler.fit_transform(X_train)

        # Detect number of classes
        self.classes_ = np.unique(y_train)
        n_classes = len(self.classes_)

        # Convert to tensors and move to device
        X_tensor = torch.FloatTensor(X_scaled).to(self.device)

        # Create model based on binary vs multi-class
        if n_classes == 2:
            # Binary classification
            y_tensor = torch.FloatTensor(y_train).to(self.device)

            class LogisticRegressionModel(nn.Module):
                def __init__(self, input_dim):
                    super(LogisticRegressionModel, self).__init__()
                    self.linear = nn.Linear(input_dim, 1)

                def forward(self, x):
                    return torch.sigmoid(self.linear(x)).squeeze()

            self.model =
         LogisticRegressionModel(X_tensor.shape[1]).to(self.device)
            criterion = nn.BCELoss()
            self.is_binary = True
        else:
            # Multi-class classification
            y_tensor = torch.LongTensor(y_train).to(self.device)

            class LogisticRegressionModel(nn.Module):
                def __init__(self, input_dim, num_classes):
                    super(LogisticRegressionModel, self).__init__()
                    self.linear = nn.Linear(input_dim, num_classes)

                def forward(self, x):
                    return self.linear(x)  # No activation - CrossEntropyLoss
         includes softmax

            self.model = LogisticRegressionModel(X_tensor.shape[1],
         n_classes).to(self.device)
            criterion = nn.CrossEntropyLoss()
            self.is_binary = False

        optimizer = optim.Adam(self.model.parameters(),
         lr=self.learning_rate)

        # Training loop with early stopping
        prev_loss = float('inf')
```

```
        patience = 10
        patience_counter = 0

        for epoch in range(self.max_iter):
            epoch_loss = 0
            num_batches = 0

            # Mini-batch training
            for i in range(0, len(X_tensor), self.batch_size):
                batch_X = X_tensor[i:i+self.batch_size]
                batch_y = y_tensor[i:i+self.batch_size]

                optimizer.zero_grad()
                outputs = self.model(batch_X)
                loss = criterion(outputs, batch_y)
                loss.backward()
                optimizer.step()

                epoch_loss += loss.item()
                num_batches += 1

            avg_loss = epoch_loss / num_batches

            # Early stopping check
            if abs(prev_loss - avg_loss) < self.tol:
                patience_counter += 1
                if patience_counter >= patience:
                    logger.info(f"Converged at epoch {epoch}")
                    break
            else:
                patience_counter = 0

            prev_loss = avg_loss

        # Calculate feature importance (weights magnitude)
        with torch.no_grad():
            weights = self.model.linear.weight.cpu().numpy().flatten()
            self.feature_importance = np.abs(weights)

        return self

    def predict(self, X_test):
        """Predict class labels."""
        if self.is_binary:
            # Binary: threshold at 0.5
            probabilities = self.predict_proba(X_test)
            return (probabilities > 0.5).astype(int)
        else:
            # Multi-class: argmax of probabilities
            probabilities = self.predict_proba(X_test)
            return np.argmax(probabilities, axis=1)

    def predict_proba(self, X_test):
        """Predict class probabilities."""
        if self.model is None:
```

```
            raise ValueError("Model not fitted yet. Call fit() first.")

        # Normalize features using fitted scaler
        X_scaled = self.scaler.transform(X_test)

        self.model.eval()
        with torch.no_grad():
            X_tensor = torch.FloatTensor(X_scaled).to(self.device)

            # Handle large datasets in batches
            all_probs = []
            for i in range(0, len(X_tensor), self.batch_size):
                batch_X = X_tensor[i:i+self.batch_size]
                batch_output = self.model(batch_X)

                if self.is_binary:
                    # Binary: output is already probabilities from sigmoid
                    batch_probs = batch_output.cpu().numpy()
                else:
                    # Multi-class: apply softmax to get probabilities
                    batch_probs = torch.softmax(batch_output,
     dim=1).cpu().numpy()

                all_probs.extend(batch_probs)

            return np.array(all_probs)

    def evaluate(self, X_test, y_test):
        """Evaluate model performance."""
        predictions = self.predict(X_test)
        probabilities = self.predict_proba(X_test)

        metrics = {
            'accuracy': accuracy_score(y_test, predictions),
            'precision': precision_score(y_test, predictions,
     average='weighted', zero_division=0),
            'recall': recall_score(y_test, predictions, average='weighted',
     zero_division=0),
            'f1': f1_score(y_test, predictions, average='weighted',
     zero_division=0),
            'auc': roc_auc_score(y_test, probabilities) if
     len(np.unique(y_test)) > 1 else 0
        }

        return metrics

    def get_feature_importance(self, feature_names=None):
        """Get feature importance."""
        if self.feature_importance is None:
            logger.warning("Feature importance not available. Fit model
     first.")
            return None

        if feature_names is None:
            feature_names = [f"feature_{i}" for i in
     range(len(self.feature_importance))]
```

```
        importance_df = pd.DataFrame({
            'Feature': feature_names,
            'Importance': self.feature_importance
        })

        return importance_df.sort_values('Importance', ascending=False)


class GPULinearRegression:
    """GPU-accelerated Linear Regression using PyTorch"""

    def __init__(self, params=None, use_gpu=True):
        if params is None:
            params = {}

        self.learning_rate = params.get('learning_rate', 0.01)
        self.max_iter = params.get('max_iter', 1000)
        self.tol = params.get('tol', 1e-6)
        self.batch_size = params.get('batch_size', 10000)
        self.use_analytical = params.get('use_analytical', True)  # Use
         analytical solution when possible
        self.use_gpu = use_gpu and torch.cuda.is_available()

        self.device = torch.device('cuda' if self.use_gpu else 'cpu')
        self.model = None
        self.scaler = StandardScaler()
        self.feature_importance = None

        if self.use_gpu:
            logger.info("Using GPU acceleration for Linear Regression")
        else:
            logger.info("Using CPU for Linear Regression")

    def fit(self, X_train, y_train):
        """Fit model to training data."""
        logger.info("Fitting GPU linear regression model...")

        # Normalize features
        X_scaled = self.scaler.fit_transform(X_train)

        if self.use_analytical:
            # Use analytical solution for better accuracy
            return self._fit_analytical(X_scaled, y_train)
        else:
            # Use SGD for very large datasets
            return self._fit_sgd(X_scaled, y_train)

    def _fit_analytical(self, X_scaled, y_train):
        """Fit using analytical least squares solution"""
        logger.info("Using analytical solution...")

        # Convert to tensors and move to device
        X_tensor = torch.FloatTensor(X_scaled).to(self.device)
        y_tensor = torch.FloatTensor(y_train).to(self.device)
```

```
        # Add bias column (intercept)
        X_bias = torch.cat([torch.ones(X_tensor.shape[0], 1,
         device=self.device), X_tensor], dim=1)

        # Analytical solution: w = (X^T X)^(-1) X^T y
        XtX = torch.matmul(X_bias.T, X_bias)
        Xty = torch.matmul(X_bias.T, y_tensor)

        # Solve using Cholesky decomposition (more stable)
        try:
            self.weights = torch.cholesky_solve(Xty.unsqueeze(1),
         torch.linalg.cholesky(XtX)).squeeze()
        except:
            # Fallback to pseudo-inverse if Cholesky fails
            self.weights = torch.matmul(torch.pinverse(XtX), Xty)

        # Feature importance (weight magnitudes, excluding bias)
        self.feature_importance = torch.abs(self.weights[1:]).cpu().numpy()

        return self

    def _fit_sgd(self, X_scaled, y_train):
        """Fit using SGD for very large datasets"""
        logger.info("Using SGD training...")

        # Convert to tensors and move to device
        X_tensor = torch.FloatTensor(X_scaled).to(self.device)
        y_tensor = torch.FloatTensor(y_train).to(self.device)

        # Create model
        class LinearRegressionModel(nn.Module):
            def __init__(self, input_dim):
                super(LinearRegressionModel, self).__init__()
                self.linear = nn.Linear(input_dim, 1)

            def forward(self, x):
                return self.linear(x).squeeze()

        self.model = LinearRegressionModel(X_tensor.shape[1]).to(self.device)
        criterion = nn.MSELoss()
        optimizer = optim.Adam(self.model.parameters(),
         lr=self.learning_rate)

        # Training loop with early stopping
        prev_loss = float('inf')
        patience = 10
        patience_counter = 0

        for epoch in range(self.max_iter):
            epoch_loss = 0
            num_batches = 0

            # Mini-batch training
            for i in range(0, len(X_tensor), self.batch_size):
```

```
                batch_X = X_tensor[i:i+self.batch_size]
                batch_y = y_tensor[i:i+self.batch_size]

                optimizer.zero_grad()
                outputs = self.model(batch_X)
                loss = criterion(outputs, batch_y)
                loss.backward()
                optimizer.step()

                epoch_loss += loss.item()
                num_batches += 1

            avg_loss = epoch_loss / num_batches

            # Early stopping check
            if abs(prev_loss - avg_loss) < self.tol:
                patience_counter += 1
                if patience_counter >= patience:
                    logger.info(f"Converged at epoch {epoch}")
                    break
            else:
                patience_counter = 0

            prev_loss = avg_loss

        # Calculate feature importance (weights magnitude)
        with torch.no_grad():
            weights = self.model.linear.weight.cpu().numpy().flatten()
            self.feature_importance = np.abs(weights)

        return self

    def predict(self, X_test):
        """Predict target values."""
        if hasattr(self, 'weights') and self.weights is not None:
            # Analytical solution prediction
            X_scaled = self.scaler.transform(X_test)
            X_tensor = torch.FloatTensor(X_scaled).to(self.device)
            X_bias = torch.cat([torch.ones(X_tensor.shape[0], 1,
         device=self.device), X_tensor], dim=1)

            with torch.no_grad():
                predictions = torch.matmul(X_bias,
         self.weights).cpu().numpy()
            return predictions

        elif self.model is not None:
            # SGD model prediction
            X_scaled = self.scaler.transform(X_test)

            self.model.eval()
            with torch.no_grad():
                X_tensor = torch.FloatTensor(X_scaled).to(self.device)

                # Handle large datasets in batches
```

```
                all_preds = []
                for i in range(0, len(X_tensor), self.batch_size):
                    batch_X = X_tensor[i:i+self.batch_size]
                    batch_preds = self.model(batch_X).cpu().numpy()
                    all_preds.extend(batch_preds)

                return np.array(all_preds)

        else:
            raise ValueError("Model not fitted yet. Call fit() first.")

    def evaluate(self, X_test, y_test):
        """Evaluate model performance."""
        predictions = self.predict(X_test)

        metrics = {
            'r2': r2_score(y_test, predictions),
            'mse': mean_squared_error(y_test, predictions),
            'mae': mean_absolute_error(y_test, predictions),
            'rmse': np.sqrt(mean_squared_error(y_test, predictions)),
            'explained_variance': explained_variance_score(y_test,
         predictions)
        }

        return metrics

    def get_feature_importance(self, feature_names=None):
        """Get feature importance."""
        if self.feature_importance is None:
            logger.warning("Feature importance not available. Fit model
         first.")
            return None

        if feature_names is None:
            feature_names = [f"feature_{i}" for i in
         range(len(self.feature_importance))]

        importance_df = pd.DataFrame({
            'Feature': feature_names,
            'Importance': self.feature_importance
        })

        return importance_df.sort_values('Importance', ascending=False)
```

3.9: evaluation/metrics.py

Location: `evaluation/metrics.py`

```
import numpy as np
import pandas as pd
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, confusion_matrix, classification_report,
    balanced_accuracy_score, matthews_corrcoef, cohen_kappa_score,
```

```
    r2_score, mean_squared_error, mean_absolute_error, median_absolute_error,
         explained_variance_score
)
import logging

# Set up logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %
         (message)s')
logger = logging.getLogger(__name__)

def evaluate_classification_model(model, X_test, y_test, model_name=None):
    """Evaluate classification model with metrics for CSV export."""
    if model_name is None:
        model_name = type(model).__name__

    logger.info(f"Evaluating {model_name}...")

    # Make predictions
    y_pred = model.predict(X_test)

    # Calculate core metrics
    metrics = {
        # Overall accuracy
        'accuracy': accuracy_score(y_test, y_pred),

        # Weighted averages (account for class imbalance)
        'precision': precision_score(y_test, y_pred, average='weighted',
         zero_division=0),
        'recall': recall_score(y_test, y_pred, average='weighted',
         zero_division=0),
        'f1': f1_score(y_test, y_pred, average='weighted', zero_division=0),

        # Macro averages (treat all classes equally)
        'precision_macro': precision_score(y_test, y_pred, average='macro',
         zero_division=0),
        'recall_macro': recall_score(y_test, y_pred, average='macro',
         zero_division=0),
        'f1_macro': f1_score(y_test, y_pred, average='macro',
         zero_division=0),

        # Robustness metrics
        'balanced_accuracy': balanced_accuracy_score(y_test, y_pred),
        'mcc': matthews_corrcoef(y_test, y_pred),
        'cohens_kappa': cohen_kappa_score(y_test, y_pred)
    }

    # Calculate ROC-AUC if predict_proba is available
    try:
        y_pred_proba = model.predict_proba(X_test)
        n_classes = len(np.unique(y_test))

        if n_classes == 2:
            # Binary classification - use positive class probabilities
            metrics['roc_auc'] = roc_auc_score(y_test, y_pred_proba[:, 1])
        else:
```

```
            # Multi-class classification - use one-vs-rest weighted
            metrics['roc_auc'] = roc_auc_score(y_test, y_pred_proba,
                                               multi_class='ovr',
         average='weighted')
    except (AttributeError, ValueError, IndexError) as e:
        # Model doesn't support predict_proba or other issue
        logger.warning(f"{model_name}: Could not calculate ROC-AUC
         ({str(e)})")
        metrics['roc_auc'] = np.nan

    logger.info(f"{model_name} - Accuracy: {metrics['accuracy']:.4f}, "
                f"F1 (weighted): {metrics['f1']:.4f}, "
                f"F1 (macro): {metrics['f1_macro']:.4f}")

    return metrics

def evaluate_regression_model(model, X_test, y_test, model_name=None):
    """Evaluate regression model with metrics for CSV export."""
    if model_name is None:
        model_name = type(model).__name__

    logger.info(f"Evaluating {model_name}...")

    # Make predictions
    y_pred = model.predict(X_test)

    # Calculate core required metrics
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    mae = mean_absolute_error(y_test, y_pred)

    metrics = {
        # Primary metrics (required)
        'r2': r2_score(y_test, y_pred),
        'rmse': rmse,
        'mae': mae,
        'median_absolute_error': median_absolute_error(y_test, y_pred),
        'explained_variance': explained_variance_score(y_test, y_pred),

        # Also include MSE for completeness (commonly used)
        'mse': mse,
    }

    # Optional: MAPE and SMAPE (exclude if near-zero targets exist)
    try:
        # MAPE - only calculate if no near-zero values
        if np.all(np.abs(y_test) > 1e-3):
            mape = np.mean(np.abs((y_test - y_pred) / y_test)) * 100
            metrics['mape'] = mape
        else:
            metrics['mape'] = np.nan

        # SMAPE - handles zero values better
        smape = 100 * np.mean(2 * np.abs(y_pred - y_test) / (np.abs(y_test) +
         np.abs(y_pred) + 1e-8))
```

```
            metrics['smape'] = smape
        except (ZeroDivisionError, RuntimeWarning):
            metrics['mape'] = np.nan
            metrics['smape'] = np.nan

        # Optional: RMSLE (for positive targets only)
        if np.all(y_test >= 0) and np.all(y_pred >= 0):
            try:
                rmsle = np.sqrt(np.mean((np.log1p(y_test) -
             np.log1p(y_pred))**2))
                metrics['rmsle'] = rmsle
            except (ValueError, RuntimeWarning):
                metrics['rmsle'] = np.nan
        else:
            metrics['rmsle'] = np.nan

        logger.info(f"{model_name} - R^2: {metrics['r2']:.4f}, "
                    f"RMSE: {metrics['rmse']:.4f}, "
                    f"MAE: {metrics['mae']:.4f}")

        return metrics

    def compare_classification_models(models_metrics, sort_by='f1'):
        """Compare multiple classification models."""
        comparison = []

        for model_name, metrics in models_metrics.items():
            model_comparison = {
                'Model': model_name,
                'Accuracy': metrics['accuracy'],
                'F1 (weighted)': metrics['f1'],
                'F1 (macro)': metrics['f1_macro'],
                'Precision (weighted)': metrics['precision'],
                'Precision (macro)': metrics['precision_macro'],
                'Recall (weighted)': metrics['recall'],
                'Recall (macro)': metrics['recall_macro'],
                'Balanced Accuracy': metrics['balanced_accuracy'],
                'MCC': metrics['mcc'],
                "Cohen's kappa": metrics['cohens_kappa']
            }

            if 'roc_auc' in metrics and not np.isnan(metrics['roc_auc']):
                model_comparison['ROC-AUC'] = metrics['roc_auc']

            comparison.append(model_comparison)

        # Create comparison DataFrame
        comparison_df = pd.DataFrame(comparison)

        # Sort by specified metric
        sort_mapping = {
            'f1': 'F1 (weighted)',
            'f1_macro': 'F1 (macro)',
            'accuracy': 'Accuracy',
            'balanced_accuracy': 'Balanced Accuracy',
```

```
        'mcc': 'MCC'
    }
    sort_col = sort_mapping.get(sort_by, sort_by)

    if sort_col in comparison_df.columns:
        comparison_df = comparison_df.sort_values(by=sort_col,
         ascending=False)

    return comparison_df

def compare_regression_models(models_metrics, sort_by='r2'):
    """Compare multiple regression models."""
    comparison = []

    for model_name, metrics in models_metrics.items():
        model_comparison = {
            'Model': model_name,
            'R^2': metrics['r2'],
            'RMSE': metrics['rmse'],
            'MAE': metrics['mae'],
            'Median AE': metrics['median_absolute_error'],
            'Explained Variance': metrics['explained_variance'],
            'MSE': metrics['mse']
        }

        # Optional metrics (may be NaN for some datasets)
        if 'mape' in metrics and not np.isnan(metrics['mape']):
            model_comparison['MAPE'] = metrics['mape']
        if 'smape' in metrics and not np.isnan(metrics['smape']):
            model_comparison['SMAPE'] = metrics['smape']
        if 'rmsle' in metrics and not np.isnan(metrics['rmsle']):
            model_comparison['RMSLE'] = metrics['rmsle']

        comparison.append(model_comparison)

    # Create comparison DataFrame
    comparison_df = pd.DataFrame(comparison)

    # Sort by specified metric
    sort_mapping = {
        'r2': 'R^2',
        'rmse': 'RMSE',
        'mae': 'MAE',
        'median_ae': 'Median AE',
        'explained_variance': 'Explained Variance'
    }
    sort_col = sort_mapping.get(sort_by, sort_by)

    if sort_col in comparison_df.columns:
        # Lower is better for error metrics, higher is better for R^2 and
         explained variance
        ascending = sort_col in ['RMSE', 'MAE', 'Median AE', 'MSE', 'MAPE',
         'SMAPE', 'RMSLE']
        comparison_df = comparison_df.sort_values(by=sort_col,
         ascending=ascending)
```

```
    return comparison_df
```

3.10: evaluation/hyperparameter_tuning.py

Location: `evaluation/hyperparameter_tuning.py`

```
import numpy as np
import pandas as pd
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV,
         train_test_split, cross_val_score
from sklearn.metrics import make_scorer, f1_score, r2_score,
         mean_squared_error
from sklearn.utils.multiclass import check_classification_targets
from sklearn.linear_model import Ridge, Lasso, ElasticNet
import logging
import sys
import os
import gc
from models.deep_learning import DeepClassifier, DeepRegressor
import warnings
from sklearn.exceptions import ConvergenceWarning

# Add parent directory to path
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from config import N_CV_FOLDS, N_JOBS, update_dl_params

# Set up logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %
         (message)s')
logger = logging.getLogger(__name__)

def check_gpu_availability():
    """Check if GPU is available for XGBoost and PyTorch"""
    gpu_available = False
    try:
        import torch
        if torch.cuda.is_available():
            gpu_available = True
            logger.info(f"GPU detected: {torch.cuda.get_device_name(0)}")
        else:
            logger.info("PyTorch CUDA not available")
    except ImportError:
        logger.info("PyTorch not installed")

    try:
        import xgboost as xgb
        if hasattr(xgb, 'gpu') and xgb.gpu.device_count() > 0:
            logger.info(f"XGBoost GPU support detected:
         {xgb.gpu.device_count()} devices")
        else:
            logger.info("XGBoost GPU support not available")
    except ImportError:
```

```
            logger.info("XGBoost not installed")

    return gpu_available

def prepare_xgboost_params(param_grid, force_cpu=False):
    """Prepare XGBoost parameters with GPU fallback"""
    if force_cpu:
        # Force CPU usage
        if 'tree_method' in param_grid:
            param_grid['tree_method'] = ['hist']
        if 'device' in param_grid:
            param_grid['device'] = ['cpu']
    else:
        # Check GPU availability and adjust accordingly
        try:
            import xgboost as xgb
            if hasattr(xgb, 'gpu') and xgb.gpu.device_count() > 0:
                logger.info("XGBoost will attempt GPU usage")
                # Keep both hist and gpu_hist for fallback
                if 'tree_method' in param_grid and 'gpu_hist' not in
     param_grid['tree_method']:
                    if isinstance(param_grid['tree_method'], list):
                        param_grid['tree_method'].append('gpu_hist')
                    else:
                        param_grid['tree_method'] =
     [param_grid['tree_method'], 'gpu_hist']
            else:
                logger.info("GPU not available, using CPU for XGBoost")
                if 'tree_method' in param_grid:
                    param_grid['tree_method'] = ['hist']
        except ImportError:
            logger.warning("XGBoost not available")
            if 'tree_method' in param_grid:
                param_grid['tree_method'] = ['hist']

    return param_grid

def tune_linear_regression(X_train, y_train, param_grid=None, cv=5):
    """Tune Linear Regression by selecting the best model from Ridge, Lasso,
         and Elastic Net."""
    if param_grid is None:
        param_grid = LINEAR_PARAM_GRID

    logger.info("Tuning Linear Regression: testing Ridge, Lasso, and Elastic
         Net...")

    best_model = None
    best_params = {}
    best_score = -np.inf
    best_model_type = None

    # Test Ridge
    logger.info(f"Testing Ridge with {len(param_grid['Ridge']['alpha'])}
         alpha values...")
    for alpha in param_grid['Ridge']['alpha']:
        model = Ridge(alpha=alpha, random_state=42)
```

```
            scores = cross_val_score(model, X_train, y_train, cv=cv,
             scoring='r2')
            mean_score = scores.mean()

            if mean_score > best_score:
                best_score = mean_score
                best_model = model
                best_model_type = 'Ridge'
                best_params = {'model_type': 'Ridge', 'alpha': alpha}

        # Test Lasso
        logger.info(f"Testing Lasso with {len(param_grid['Lasso']['alpha'])}
             alpha values...")
        for alpha in param_grid['Lasso']['alpha']:
            model = Lasso(alpha=alpha, random_state=42, max_iter=10000)
            scores = cross_val_score(model, X_train, y_train, cv=cv,
             scoring='r2')
            mean_score = scores.mean()

            if mean_score > best_score:
                best_score = mean_score
                best_model = model
                best_model_type = 'Lasso'
                best_params = {'model_type': 'Lasso', 'alpha': alpha}

        # Test Elastic Net
        logger.info(f"Testing Elastic Net with {len(param_grid['ElasticNet']
             ['alpha'])} alpha x {len(param_grid['ElasticNet']['l1_ratio'])}
             l1_ratio combinations...")
        for alpha in param_grid['ElasticNet']['alpha']:
            for l1_ratio in param_grid['ElasticNet']['l1_ratio']:
                model = ElasticNet(alpha=alpha, l1_ratio=l1_ratio,
             random_state=42, max_iter=10000)
                scores = cross_val_score(model, X_train, y_train, cv=cv,
             scoring='r2')
                mean_score = scores.mean()

                if mean_score > best_score:
                    best_score = mean_score
                    best_model = model
                    best_model_type = 'ElasticNet'
                    best_params = {'model_type': 'ElasticNet', 'alpha': alpha,
             'l1_ratio': l1_ratio}

        logger.info(f"Best model: {best_model_type} with score:
             {best_score:.4f}")
        logger.info(f"Best parameters: {best_params}")

        # Refit best model on full training set
        best_model.fit(X_train, y_train)

        return best_model, best_params, best_score

    def tune_logistic_regression(X_train, y_train, param_grid=None, cv=5):
        """Tune Logistic Regression using SGDClassifier with mini-batch gradient
             descent."""
        if param_grid is None:
```

```
    param_grid = LOGISTIC_PARAM_GRID_NEW

logger.info("Tuning Logistic Regression: using SGDClassifier with mini-
     batch gradient descent...")

from sklearn.linear_model import SGDClassifier
from sklearn.metrics import f1_score

best_model = None
best_params = {}
best_score = -np.inf

# Define scorer for multi-class support
scorer = make_scorer(f1_score, average='weighted')

# Calculate total combinations
n_combinations = (len(param_grid['learning_rate']) *
                  len(param_grid['max_iter']) *
                  len(param_grid['batch_size']))
logger.info(f"Testing {n_combinations} combinations: "
            f"{len(param_grid['learning_rate'])} learning_rate x "
            f"{len(param_grid['max_iter'])} max_iter x "
            f"{len(param_grid['batch_size'])} batch_size")

# Grid search over all combinations
for learning_rate in param_grid['learning_rate']:
    for max_iter in param_grid['max_iter']:
        for batch_size in param_grid['batch_size']:
            # SGDClassifier uses 'log_loss' for logistic regression
            # eta0 is the initial learning rate when
     learning_rate='constant'
            model = SGDClassifier(
                loss='log_loss',
                penalty='l2',
                alpha=0.0001,  # Regularization strength
                learning_rate='constant',
                eta0=learning_rate,
                max_iter=max_iter,
                random_state=42,
                n_jobs=-1
            )

            # Note: SGDClassifier doesn't have batch_size directly
            # We use partial_fit approach or just track it as a parameter
            # For standard CV, we fit on full batches but record the
     intended batch_size

            scores = cross_val_score(model, X_train, y_train, cv=cv,
     scoring=scorer)
            mean_score = scores.mean()

            if mean_score > best_score:
                best_score = mean_score
                best_model = model
                best_params = {
                    'learning_rate': learning_rate,
```

```
                    'max_iter': max_iter,
                    'batch_size': batch_size
                }

    logger.info(f"Best score: {best_score:.4f}")
    logger.info(f"Best parameters: {best_params}")

    # Refit best model on full training set
    best_model.fit(X_train, y_train)

    return best_model, best_params, best_score

def tune_classification_model(estimator, param_grid, X_train, y_train,
         method='random', n_iter=10, task='classification', cv=3):
    """Tune classification model with error handling and early stopping."""
    warnings.filterwarnings('ignore', category=ConvergenceWarning)
    warnings.filterwarnings('ignore', category=UserWarning)

    try:
        # Split for early stopping validation set if using XGBoost
        is_xgboost = 'xgboost' in str(type(estimator)).lower()

        if is_xgboost:
            X_train_main, X_val_es, y_train_main, y_val_es =
         train_test_split(
                X_train, y_train, test_size=0.2, random_state=42
            )
            eval_set = [(X_val_es, y_val_es)]
        else:
            X_train_main = X_train
            y_train_main = y_train
            eval_set = None

        # Determine search method
        if method == 'random':
            # Create RandomizedSearchCV with appropriate parameters
            search_params = {
                'estimator': estimator,
                'param_distributions': param_grid,
                'n_iter': n_iter,
                'cv': cv,  # Use the cv parameter here
                'scoring': make_scorer(f1_score, average='weighted'),  # Use
         weighted averaging for multi-class
                'n_jobs': -1,
                'random_state': 42,
                'verbose': 3  # Increased verbosity to show progress during
         long SVM tuning
            }

            # Note: fit_params removed due to sklearn compatibility issues
            # Early stopping for XGBoost will be handled in the model's fit
         method

            search = RandomizedSearchCV(**search_params)

        else:  # Grid search
```

```
            search_params = {
                'estimator': estimator,
                'param_grid': param_grid,
                'cv': 3,  # 3-fold cross-validation
                'scoring': make_scorer(f1_score, average='weighted'),  # Use
        weighted averaging for multi-class
                'n_jobs': -1,  # Use all available cores
                'verbose': 3  # Increased verbosity to show progress during
        tuning
            }

            # Note: fit_params removed due to sklearn compatibility issues
            # Early stopping for XGBoost will be handled in the model's fit
        method

            search = GridSearchCV(**search_params)

        # Fit the search
        logger.info(f"Starting {'randomized' if method == 'random' else
         'grid'} search with {n_iter if method == 'random' else
         len(param_grid)} parameter combinations")

        if is_xgboost:
            search.fit(X_train_main, y_train_main)
        else:
            search.fit(X_train, y_train)

        # Log best parameters and score
        logger.info("Best parameters found:")
        for param, value in search.best_params_.items():
            logger.info(f"{param}: {value}")
        logger.info(f"Best cross-validation score: {search.best_score_:.4f}")

        # Clean up to free memory
        gc.collect()

        return (
            search.best_estimator_,
            search.best_params_,
            search.best_score_
        )

    except Exception as e:
        logger.error(f"Error during hyperparameter tuning: {e}")

        # Fallback to simpler search or default parameters
        logger.warning("Falling back to simplified parameter search")

        try:
            # Create a reduced parameter grid
            if method == 'random':
                reduced_grid = {}
                for param, values in param_grid.items():
                    if isinstance(values, list) and len(values) > 2:
                        reduced_grid[param] = [values[0], values[-1]]  # Take
        first and last values
```

```
                    else:
                        reduced_grid[param] = values

                search = RandomizedSearchCV(
                    estimator=estimator,
                    param_distributions=reduced_grid,
                    n_iter=min(5, n_iter),  # Reduced iterations
                    cv=2,  # Reduced CV
                    scoring=make_scorer(f1_score, average='weighted'),  # Use
        weighted averaging for multi-class
                    n_jobs=1,  # Reduce to single core
                    random_state=42
                )
            else:
                reduced_grid = {}
                for param, values in param_grid.items():
                    if isinstance(values, list) and len(values) > 2:
                        reduced_grid[param] = [values[0]]  # Take just the
        first value
                    else:
                        reduced_grid[param] = values[:1]  # Take just the
        first value

                search = GridSearchCV(
                    estimator=estimator,
                    param_grid=reduced_grid,
                    cv=2,  # Reduced CV
                    scoring=make_scorer(f1_score, average='weighted'),  # Use
        weighted averaging for multi-class
                    n_jobs=1  # Reduce to single core
                )

            search.fit(X_train, y_train)
            logger.info("Fallback search completed successfully")
            return (
                search.best_estimator_,
                search.best_params_,
                search.best_score_
            )
        except Exception as fallback_error:
            logger.error(f"Fallback tuning failed: {fallback_error}")

            # Return original estimator with default parameters
            logger.warning("Using default parameters")
            return (
                estimator,
                {},
                0.0
            )

def tune_regression_model(model, param_grid, X_train, y_train, X_val=None,
        y_val=None,
                          scoring='neg_root_mean_squared_error', cv=3,
        n_jobs=N_JOBS, method='grid'):
    """Tune hyperparameters for regression model with early stopping."""
    logger.info(f"Tuning hyperparameters for {type(model).__name__}...")
```

```
# Define scorer
if scoring == 'r2':
    scorer = make_scorer(r2_score)
elif scoring == 'neg_rmse':
    scorer = make_scorer(lambda y, y_pred: -np.sqrt(mean_squared_error(y,
     y_pred)))
else:
    scorer = scoring

try:
    # Check if this is an XGBoost model
    is_xgboost = 'xgboost' in str(type(model)).lower()

    # Create validation split for early stopping if model supports it
    if is_xgboost:
        if X_val is None or y_val is None:
            X_train_main, X_val_es, y_train_main, y_val_es =
     train_test_split(
                X_train, y_train, test_size=0.2, random_state=42
            )
            eval_set = [(X_val_es, y_val_es)]
        else:
            X_train_main = X_train
            y_train_main = y_train
            eval_set = [(X_val, y_val)]
    else:
        X_train_main = X_train
        y_train_main = y_train
        eval_set = None

    # Create search object
    if method == 'grid':
        search_params = {
            'estimator': model,
            'param_grid': param_grid,
            'scoring': scorer,
            'cv': cv,
            'n_jobs': n_jobs,
            'verbose': 1
        }

        # Note: fit_params removed due to sklearn compatibility issues
        # Early stopping for XGBoost will be handled in the model's fit
     method

        search = GridSearchCV(**search_params)

    elif method == 'random':
        search_params = {
            'estimator': model,
            'param_distributions': param_grid,
            'scoring': scorer,
            'cv': cv,
            'n_jobs': n_jobs,
```

```
                'n_iter': 20,
                'verbose': 1,
                'random_state': 42
            }

            # Note: fit_params removed due to sklearn compatibility issues
            # Early stopping for XGBoost will be handled in the model's fit
        method

            search = RandomizedSearchCV(**search_params)

        else:
            raise ValueError(f"Unknown method: {method}")

        # Fit search
        if is_xgboost:
            search.fit(X_train_main, y_train_main)
        else:
            search.fit(X_train, y_train)

        # Get best model
        best_model = search.best_estimator_
        best_params = search.best_params_
        cv_results = pd.DataFrame(search.cv_results_)

        logger.info(f"Best parameters: {best_params}")
        logger.info(f"Best CV score: {search.best_score_:.4f}")

        # Clean up to free memory
        gc.collect()

        # Evaluate on validation set if provided
        if X_val is not None and y_val is not None:
            y_pred = best_model.predict(X_val)
            val_r2 = r2_score(y_val, y_pred)
            val_rmse = np.sqrt(mean_squared_error(y_val, y_pred))
            logger.info(f"Validation R^2: {val_r2:.4f}, RMSE:
        {val_rmse:.4f}")

        return best_model, best_params, cv_results

    except Exception as e:
        logger.error(f"Error during hyperparameter tuning: {e}")

        # Fallback to simpler model or default parameters
        try:
            logger.warning("Falling back to simplified parameter search")

            # Create a reduced parameter grid
            if method == 'random':
                reduced_grid = {}
                for param, values in param_grid.items():
                    if isinstance(values, list) and len(values) > 2:
                        reduced_grid[param] = [values[0], values[-1]]  # Take
        first and last values
```

```
                    else:
                        reduced_grid[param] = values

                search = RandomizedSearchCV(
                    estimator=model,
                    param_distributions=reduced_grid,
                    scoring=scorer,
                    cv=2,  # Reduced CV
                    n_jobs=1,  # Single core
                    n_iter=5,  # Reduced iterations
                    verbose=1,
                    random_state=42
                )
            else:
                reduced_grid = {}
                for param, values in param_grid.items():
                    if isinstance(values, list) and len(values) > 2:
                        reduced_grid[param] = [values[0]]  # Take just one
        value
                    else:
                        reduced_grid[param] = values[:1]  # Take just the
        first option

                search = GridSearchCV(
                    estimator=model,
                    param_grid=reduced_grid,
                    scoring=scorer,
                    cv=2,  # Reduced CV
                    n_jobs=1,  # Single core
                    verbose=1
                )

            search.fit(X_train, y_train)

            logger.info("Fallback search completed successfully")
            return search.best_estimator_, search.best_params_,
        pd.DataFrame(search.cv_results_)

        except Exception as fallback_error:
            logger.error(f"Fallback tuning failed: {fallback_error}")

            # Return original model with default parameters
            logger.warning("Using default parameters")
            return model, {}, None


# Logistic Regression - 20x range standardization
LOGISTIC_PARAM_GRID = [
    # L2 penalty with compatible solvers
    {
        'C': [0.1, 1, 2],  # 20x range (0.1 to 2)
        'penalty': ['l2'],
        'solver': ['lbfgs'],
        'class_weight': [None, 'balanced']
    },
```

```
    # L1 penalty with compatible solvers
    {
        'C': [0.1, 1],  # Keep 2 values for L1
        'penalty': ['l1'],
        'solver': ['saga'],
        'class_weight': ['balanced']
    }
]

# Linear Regression - Regularization variants (Ridge, Lasso, Elastic Net)
# Default uses plain OLS; Tuned selects best regularization model via CV
LINEAR_PARAM_GRID = {
    'Ridge': {
        'alpha': [1e-4, 1e-3, 1e-2, 1e-1, 1, 10]  # 6 values
    },
    'Lasso': {
        'alpha': [1e-4, 1e-3, 1e-2, 1e-1, 1]  # 5 values
    },
    'ElasticNet': {
        'alpha': [1e-4, 1e-3, 1e-2, 1e-1, 1],  # 5 values
        'l1_ratio': [0.2, 0.5, 0.8]  # 3 values (eta in thesis notation)
    }
}
# Total: 6 Ridge + 5 Lasso + 15 Elastic Net = 26 combinations
# With 5-fold CV: 26 x 5 = 130 total fits

# Logistic Regression - SGDClassifier with mini-batch gradient descent
# Default: learning_rate=0.01, max_iter=1000, batch_size=10000
# Tuned: Grid search over learning_rate, max_iter, batch_size
LOGISTIC_PARAM_GRID_NEW = {
    'learning_rate': [0.001, 0.01, 0.1],  # 3 values
    'max_iter': [500, 1000, 2000],  # 3 values
    'batch_size': [5000, 10000]  # 2 values
}
# Total: 3 x 3 x 2 = 18 combinations
# With 5-fold CV: 18 x 5 = 90 total fits

# XGBoost Classification - Light tuning (9 combinations)
XGBOOST_CLASSIFICATION_PARAM_GRID = {
    'n_estimators': [100, 300],  # 2 values - most important
    'max_depth': [3, 6],  # 2 values - key complexity control
    'learning_rate': [0.01, 0.05, 0.1],  # 3 values - critical parameter
    'subsample': [0.8],  # 1 option - less critical
    'tree_method': ['hist'],  # Single option
    'device': ['cuda'],  # Single option for GPU
    'colsample_bytree': [0.8],  # 1 option - less critical
    'gamma': [0],  # 1 option - often not needed
    'min_child_weight': [1],  # 1 option - less critical
}

# XGBoost Regression - Medium tuning (18 combinations = 90 total fits with 5-
        fold CV)
XGBOOST_REGRESSION_PARAM_GRID = {
    'n_estimators': [100, 300],  # 2 values - most important
```

```
    'max_depth': [3, 6, 9],  # 3 values - key complexity control (added 9 for
        deeper trees)
    'learning_rate': [0.01, 0.05, 0.1],  # 3 values - critical parameter
    'reg_alpha': [0, 0.1],  # 2 values - L1 regularization (NEW - important
        for regression)
    'subsample': [0.8],  # 1 option - less critical
    'colsample_bytree': [0.8],  # 1 option - less critical
    'gamma': [0],  # 1 option - often not needed
    'min_child_weight': [1],  # 1 option - less critical
    'tree_method': ['hist'],  # Single option
    'device': ['cuda'],  # Single option for GPU
}
# Total: 2 x 3 x 3 x 2 = 18 combinations

# SVM Classification - 20x range standardization
SVM_CLASSIFICATION_PARAM_GRID = {
    'estimator__C': [0.1, 1, 2],  # 20x range (0.1 to 2)
    'estimator__class_weight': [None, 'balanced']  # Keep 2 for balance
}

# SVM Regression - Wider range exploration (12 combinations = 60 total fits
        with 5-fold CV)
SVM_REGRESSION_PARAM_GRID = {
    'C': [0.1, 1, 10],  # 3 values - 100x range (0.1 to 10) for better
        exploration
    'epsilon': [0.01, 0.1, 0.5, 1.0]  # 4 values - wider range for epsilon
        sensitivity
}
# Total: 3 x 4 = 12 combinations


# Neural Network - Medium tuning (18 combinations for 5-fold CV = 90 total
        fits)
NN_PARAM_GRID = {
    'hidden_layers': [
        [64], [128], [64, 32]  # 3 most important architectures
    ],
    'learning_rate': [0.0001, 0.001, 0.01],  # 3 values - added more
        conservative 0.0001 to prevent divergence
    'batch_size': [64],  # 1 option - less critical for light tuning
    'dropout_rate': [0.2, 0.4],  # 2 options - slightly lower range to reduce
        over-regularization
    'optimizer': ['adam']  # 1 option - adam usually best
}
# Total: 3 x 3 x 1 x 2 x 1 = 18 combinations

################################################################################
# EXHAUSTIVE PARAMETER GRIDS (OLD METHODOLOGY - NOT USED)
# New methodology uses only Default (Level 0) and Medium-Tuned (Level 1)
# These exhaustive grids are kept for reference but not called in new
        workflow
################################################################################

# XGBoost Classification - Added CPU fallback
XGBOOST_CLASSIFICATION_PARAM_GRID_EXHAUSTIVE = {
    'n_estimators': [100, 300, 500, 1000],  # Reduced from 7 to 4
    'max_depth': [3, 5, 7, 10],  # Reduced from 10 to 4
```

```
    'learning_rate': [0.01, 0.05, 0.1, 0.2],  # Reduced from 7 to 4
    'subsample': [0.7, 0.8, 1.0],  # Reduced from 6 to 3
    'colsample_bytree': [0.7, 0.8, 1.0],  # Reduced from 6 to 3
    'gamma': [0, 0.1, 0.3],  # Reduced from 7 to 3
    'min_child_weight': [1, 3, 5],  # Reduced from 6 to 3
    'reg_alpha': [0, 0.01, 0.1, 1.0],  # Reduced from 8 to 4
    'reg_lambda': [0.001, 0.1, 1.0],  # Reduced from 7 to 3
    'scale_pos_weight': [1, 10, 50],  # Reduced from 7 to 3 for imbalanced
        datasets
    'tree_method': ['hist'],  # Single option
    'device': ['cuda']  # Single option for GPU
}

# SVM Classification
SVM_CLASSIFICATION_PARAM_GRID_EXHAUSTIVE = {
    'estimator__C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
    'estimator__kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
    'estimator__gamma': ['scale', 'auto', 0.0001, 0.001, 0.01, 0.1, 1],
    'estimator__degree': [2, 3, 4, 5],  # for poly kernel
    'estimator__coef0': [0.0, 0.1, 0.5, 1.0],  # for poly and sigmoid kernels
    'estimator__class_weight': [None, 'balanced'],
    'probability': [True],
    'tol': [1e-3, 1e-4, 1e-5],
    'cache_size': [1000]  # Larger cache size for faster computation
}


# Exhaustive parameter grids for regression models

# Linear Regression
LINEAR_PARAM_GRID_EXHAUSTIVE = {
    'fit_intercept': [True, False],
    'copy_X': [True],
    'positive': [True, False],
    'n_jobs': [-1]  # Use all available processors
}

# XGBoost Regression - Added CPU fallback
XGBOOST_REGRESSION_PARAM_GRID_EXHAUSTIVE = {
    'n_estimators': [100, 300, 500, 1000],  # Reduced from 7 to 4
    'max_depth': [3, 5, 7, 10],  # Reduced from 10 to 4
    'learning_rate': [0.01, 0.05, 0.1, 0.2],  # Reduced from 7 to 4
    'subsample': [0.7, 0.8, 1.0],  # Reduced from 6 to 3
    'colsample_bytree': [0.7, 0.8, 1.0],  # Reduced from 6 to 3
    'gamma': [0, 0.1, 0.3],  # Reduced from 7 to 3
    'min_child_weight': [1, 3, 5],  # Reduced from 6 to 3
    'reg_alpha': [0, 0.01, 0.1, 1.0],  # Reduced from 8 to 4
    'reg_lambda': [0.001, 0.1, 1.0],  # Reduced from 7 to 3
    'tree_method': ['hist'],  # Single option
    'device': ['cuda']  # Single option for GPU
    # Removed device parameter - will be handled dynamically
}

# SVM Regression
SVM_REGRESSION_PARAM_GRID_EXHAUSTIVE = {
```

```
    'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
    'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
    'gamma': ['scale', 'auto', 0.0001, 0.001, 0.01, 0.1, 1],
    'degree': [2, 3, 4, 5],  # for poly kernel
    'coef0': [0.0, 0.1, 0.5, 1.0],  # for poly and sigmoid kernels
    'epsilon': [0.01, 0.05, 0.1, 0.2, 0.5],
    'tol': [1e-3, 1e-4, 1e-5],
    'cache_size': [1000]  # Larger cache size for faster computation
}


# Neural Network grid for Exhaustion
NN_PARAM_GRID_EXHAUSTIVE = {
    'hidden_layers': [
        [32], [64], [128], [256], [512],
        [32, 16], [64, 32], [128, 64], [256, 128], [512, 256],
        [64, 32, 16], [128, 64, 32], [256, 128, 64], [512, 256, 128],
        [128, 64, 32, 16], [256, 128, 64, 32]
    ],
    'learning_rate': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05],
    'batch_size': [16, 32, 64, 128, 256],
    'dropout_rate': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6],
    'optimizer': ['adam', 'sgd', 'rmsprop', 'adagrad'],
    'epochs': [50, 100, 200, 300, 500]
}

# TabNet - Exhaustive parameter grid
TABNET_PARAM_GRID_EXHAUSTIVE = {
    'n_d': [4, 8, 12, 16, 24, 32],  # Width of decision prediction layer
    'n_a': [4, 8, 12, 16, 24, 32],  # Width of attention embedding
    'n_steps': [2, 3, 4, 5, 6, 7],  # Number of decision steps
    'gamma': [1.0, 1.2, 1.3, 1.5, 1.7, 2.0],  # Coefficient for feature
        reusage
    'n_independent': [1, 2, 3, 4],  # Independent GLU layers
    'n_shared': [1, 2, 3, 4],  # Shared GLU layers
    'learning_rate': [0.005, 0.01, 0.02, 0.03, 0.05],  # Learning rate
    'batch_size': [64, 128, 256, 512],  # Batch size
    'lambda_sparse': [1e-5, 1e-4, 1e-3, 1e-2],  # Sparsity regularization
    'momentum': [0.01, 0.02, 0.05, 0.1],  # Ghost batch normalization
        momentum
    'epochs': [50, 100, 150, 200, 300]  # Number of training epochs
}

# ResNet - Exhaustive parameter grid
RESNET_PARAM_GRID_EXHAUSTIVE = {
    'hidden_size': [128, 256],  # Reduced from 4 to 2
    'num_blocks': [3, 4],  # Reduced from 5 to 2
    'block_depth': [2, 3],  # Reduced from 4 to 2
    'dropout_rate': [0.2, 0.3, 0.4],  # Reduced from 5 to 3
    'learning_rate': [0.001, 0.005],  # Reduced from 5 to 2
    'batch_size': [32, 64],  # Reduced from 4 to 2
    'weight_decay': [1e-4, 1e-3],  # Reduced from 4 to 2
    'optimizer': ['adam'],  # Reduced from 2 to 1
    'scheduler': ['cosine', None],  # Reduced from 3 to 2
    'epochs': [100, 150]  # Reduced from 5 to 2
```

```
}

# Parameter grids for different models - Optimized for better performance/speed tradeoff
def get_standard_param_grid(model_name):
    """Get standard parameter grid for the given model"""
    if model_name == 'Logistic Regression':
        return LOGISTIC_PARAM_GRID
    elif model_name == 'Linear Regression':
        return LINEAR_PARAM_GRID
    elif model_name == 'XGBoost' or model_name == 'XGBoost Classification':
        return XGBOOST_CLASSIFICATION_PARAM_GRID
    elif model_name == 'XGBoost Regression':
        return XGBOOST_REGRESSION_PARAM_GRID
    elif model_name == 'SVM' or model_name == 'SVM Classification':
        return SVM_CLASSIFICATION_PARAM_GRID
    elif model_name == 'SVM Regression':
        return SVM_REGRESSION_PARAM_GRID
    elif model_name == 'MLP':
        return NN_PARAM_GRID
    elif model_name == 'TabNet':
        return TABNET_PARAM_GRID
    elif model_name == 'ResNet':
        return RESNET_PARAM_GRID
    else:
        logger.warning(f"No standard parameter grid defined for {model_name}")
        return {}

def get_exhaustive_param_grid(model_name):
    """Get exhaustive parameter grid for the given model"""
    if model_name == 'Logistic Regression':
        return LOGISTIC_PARAM_GRID_EXHAUSTIVE
    elif model_name == 'Linear Regression':
        return LINEAR_PARAM_GRID_EXHAUSTIVE
    elif model_name == 'XGBoost' or model_name == 'XGBoost Classification':
        return XGBOOST_CLASSIFICATION_PARAM_GRID_EXHAUSTIVE
    elif model_name == 'XGBoost Regression':
        return XGBOOST_REGRESSION_PARAM_GRID_EXHAUSTIVE
    elif model_name == 'SVM' or model_name == 'SVM Classification':
        return SVM_CLASSIFICATION_PARAM_GRID_EXHAUSTIVE
    elif model_name == 'SVM Regression':
        return SVM_REGRESSION_PARAM_GRID_EXHAUSTIVE
    elif model_name == 'Deep Learning' or model_name == 'MLP':
        return NN_PARAM_GRID_EXHAUSTIVE
    elif model_name == 'TabNet':
        return TABNET_PARAM_GRID_EXHAUSTIVE
    elif model_name == 'ResNet':
        return RESNET_PARAM_GRID_EXHAUSTIVE
    else:
        logger.warning(f"No exhaustive parameter grid defined for {model_name}")
        return get_standard_param_grid(model_name)  # Fall back to standard grid
```

```python
def tune_nn_model(model_class, param_grid=None, X_train=None, y_train=None,
                  X_val=None, y_val=None, task='classification'):
    """Tune neural network hyperparameters."""
    if param_grid is None:
        param_grid = {
            'hidden_layers': [[64], [64, 32]],  # Reduced options
            'learning_rate': [0.001],  # Single option for faster tuning
            'batch_size': [32],  # Single option for faster tuning
            'dropout_rate': [0.3],  # Single option for faster tuning
            'optimizer': ['Adam']  # Single option for faster tuning
        }

    # If validation split not provided, split training data
    if X_val is None or y_val is None:
        X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
         test_size=0.2, random_state=42)

    best_model = None
    best_score = float('-inf') if task == 'classification' else float('inf')
    best_params = None

    # Log total number of combinations
    total_combinations = (
        len(param_grid['hidden_layers']) *
        len(param_grid['learning_rate']) *
        len(param_grid['batch_size']) *
        len(param_grid['dropout_rate']) *
        len(param_grid['optimizer'])
    )
    logger.info(f"Tuning neural network with {total_combinations} parameter
         combinations")

    # Track progress
    current_combination = 0

    # Iterate through parameter combinations
    for hidden_layers in param_grid['hidden_layers']:
        for lr in param_grid['learning_rate']:
            for batch_size in param_grid['batch_size']:
                for dropout_rate in param_grid['dropout_rate']:
                    for optimizer in param_grid['optimizer']:
                        # Create parameter set
                        params = {
                            'hidden_layers': hidden_layers,
                            'learning_rate': lr,
                            'batch_size': batch_size,
                            'dropout_rate': dropout_rate,
                            'optimizer': optimizer,
                            'use_gpu': True,  # Ensure this is included
                            'epochs': 50,  # Reduced from 100 for faster
         tuning
                        }

                        # Update progress
                        current_combination += 1
```

```
                    logger.info(f"Combination
    {current_combination}/{total_combinations} - {params}")

                    # Select appropriate model class based on task
                    if task == 'classification':
                        model_class_to_use = DeepClassifier
                    elif task == 'regression':
                        model_class_to_use = DeepRegressor
                    else:
                        raise ValueError(f"Invalid task: {task}. Must be
    'classification' or 'regression'.")

                    try:
                        # Create and train model
                        model =
    model_class_to_use(update_dl_params(params))
                        model.fit(X_train, y_train, X_val, y_val)

                        # Evaluate model based on task
                        if task == 'classification':
                            metrics = model.evaluate(X_val, y_val)
                            score = metrics['f1']  # Using F1 score as
    the metric
                            comparison = score > best_score
                            logger.info(f"Model F1 score: {score:.4f}")
                        else:  # regression
                            metrics = model.evaluate(X_val, y_val)
                            score = metrics['r2']  # Using R^2 as the
    metric
                            comparison = score > best_score
                            logger.info(f"Model R^2 score: {score:.4f}")

                        # Update best model if current model is better
                        if comparison:
                            best_score = score
                            best_model = model
                            best_params = params
                            logger.info(f"New best model found! Score:
    {best_score:.4f}")

                    except Exception as e:
                        logger.error(f"Error training model with
    parameters {params}: {e}")

                    # Force garbage collection to free memory
                    gc.collect()

logger.info(f"Best Parameters: {best_params}")
logger.info(f"Best Score: {best_score}")

if best_model is None:
    logger.warning("No successful model was found during tuning. Using
     default parameters.")
    default_params = update_dl_params()
    if task == 'classification':
        best_model = DeepClassifier(default_params)
```

```
        else:
            best_model = DeepRegressor(default_params)

        try:
            best_model.fit(X_train, y_train, X_val, y_val)
            best_params = default_params
        except Exception as e:
            logger.error(f"Error training model with default parameters:
             {e}")

    return best_model, best_params, best_score


def tune_resnet_model(model_class, param_grid=None, X_train=None,
         y_train=None,
                      optimization_level=1, n_iter=20, cv=3, scoring='f1'):
    """Tune ResNet model hyperparameters."""
    logger.info(f"Tuning ResNet model with optimization level
         {optimization_level}")

    if param_grid is None:
        param_grid = RESNET_PARAM_GRID.copy()

    if X_train is None or y_train is None:
        logger.error("X_train and y_train are required for ResNet tuning")
        return None, {}, 0.0

    # Create validation split
    X_train_main, X_val, y_train_main, y_val = train_test_split(
        X_train, y_train, test_size=0.2, random_state=42, stratify=y_train
    )

    # Generate parameter combinations
    import itertools
    param_names = list(param_grid.keys())
    param_values = list(param_grid.values())
    param_combinations = list(itertools.product(*param_values))

    # Limit combinations based on optimization level
    if optimization_level == 0:
        param_combinations = param_combinations[:1]  # Just first combination
    elif optimization_level == 1:
        param_combinations = param_combinations[:min(8,
         len(param_combinations))]  # Up to 8 combinations
    # optimization_level == 2 uses all combinations

    best_score = 0.0
    best_params = {}
    best_model = None

    logger.info(f"Testing {len(param_combinations)} parameter combinations")

    for i, param_values in enumerate(param_combinations):
        current_params = dict(zip(param_names, param_values))
        logger.info(f"Combination {i+1}/{len(param_combinations)} -
         {current_params}")
```

```
            try:
                # Create model with current parameters
                model = model_class(params=current_params)

                # Train model
                model.fit(X_train_main, y_train_main)

                # Evaluate on validation set
                metrics = model.evaluate(X_val, y_val)
                current_score = metrics.get('f1', 0.0)

                logger.info(f"Model F1 score: {current_score:.4f}")

                if current_score > best_score:
                    best_score = current_score
                    best_params = current_params.copy()
                    best_model = model
                    logger.info(f"New best model found! Score: {best_score:.4f}")

            except Exception as e:
                logger.warning(f"Failed to train ResNet with params
             {current_params}: {e}")
                continue

        if best_model is None:
            logger.error("No ResNet model trained successfully")
            return None, {}, 0.0

        logger.info(f"Best Parameters: {best_params}")
        logger.info(f"Best Score: {best_score}")

        return best_model, best_params, best_score


    def tune_tabnet_model(model_class, param_grid=None, X_train=None,
             y_train=None,
                         X_val=None, y_val=None, optimization_level=1,
             scoring='r2'):
        """Tune TabNet model hyperparameters."""
        logger.info(f"Tuning TabNet model with optimization level
             {optimization_level}")

        if param_grid is None:
            if optimization_level <= 1:
                param_grid = TABNET_PARAM_GRID.copy()
            else:
                param_grid = TABNET_PARAM_GRID_EXHAUSTIVE.copy()

        if X_train is None or y_train is None:
            logger.error("X_train and y_train are required for TabNet tuning")
            return None, {}

        # If no validation set provided, create one
        if X_val is None or y_val is None:
            X_train_main, X_val, y_train_main, y_val = train_test_split(
```

```
            X_train, y_train, test_size=0.2, random_state=42
        )
    else:
        X_train_main = X_train
        y_train_main = y_train

    # Generate parameter combinations
    import itertools
    param_names = list(param_grid.keys())
    param_values = list(param_grid.values())
    param_combinations = list(itertools.product(*param_values))

    # Limit combinations based on optimization level
    if optimization_level == 0:
        param_combinations = param_combinations[:1]  # Just first combination
    elif optimization_level == 1:
        param_combinations = param_combinations[:min(8,
         len(param_combinations))]  # Up to 8 combinations
    # optimization_level == 2 uses all combinations

    best_score = -float('inf')
    best_params = {}
    best_model = None

    logger.info(f"Testing {len(param_combinations)} parameter combinations")

    for i, param_values in enumerate(param_combinations):
        current_params = dict(zip(param_names, param_values))
        logger.info(f"Combination {i+1}/{len(param_combinations)} -
         {current_params}")

        try:
            # Create model with current parameters
            # TabNet wrapper expects params as a dictionary
            model = model_class(params=current_params)

            # Train model
            model.fit(X_train_main, y_train_main)

            # Evaluate on validation set
            y_val_pred = model.predict(X_val)

            # Calculate score based on scoring metric
            if scoring == 'r2':
                from sklearn.metrics import r2_score
                current_score = r2_score(y_val, y_val_pred)
            elif scoring == 'f1':
                from sklearn.metrics import f1_score
                current_score = f1_score(y_val, y_val_pred,
         average='weighted')
            else:
                # Default to r2 for regression
                from sklearn.metrics import r2_score
                current_score = r2_score(y_val, y_val_pred)
```

```
            logger.info(f"Model {scoring} score: {current_score:.4f}")

            if current_score > best_score:
                best_score = current_score
                best_params = current_params.copy()
                best_model = model
                logger.info(f"New best model found! Score: {best_score:.4f}")

        except Exception as e:
            logger.warning(f"Failed to train TabNet with params
         {current_params}: {e}")
            continue

    if best_model is None:
        logger.error("No TabNet model trained successfully")
        return None, {}

    logger.info(f"Best Parameters: {best_params}")
    logger.info(f"Best Score: {best_score}")

    return best_model, best_params
```

3.11: batch_factorial_runner.py

Location: `batch_factorial_runner.py` (project root)

```
"""
Batch Factorial Design Runner

This script runs the complete factorial design experiment across all 72
         datasets:
- 9 dimensionality conditions (3x3: small/medium/large obs x
         small/medium/large features)
- 4 quality variants per dimensionality
- 2 tasks (classification, regression)
- 2 configurations (Default, Tuned)
- 4 models (Linear/Logistic, SVM, XGBoost, MLP)

Total: 72 datasets x 2 configs x 4 models = 576 experiments

Usage:
    # Run all datasets with default and tuned configurations
    python batch_factorial_runner.py --tune

    # Run only classification datasets
    python batch_factorial_runner.py --task classification --tune

    # Run only regression datasets
    python batch_factorial_runner.py --task regression --tune

    # Run specific dimensionality category
    python batch_factorial_runner.py --dimensionality medxmed --tune
```

```
    # Run multiple dimensionality categories (e.g., all large observation
        datasets)
    python batch_factorial_runner.py --dimensionality largexlarge largexmed
        largexsmall --tune

    # Dry run to see what would be processed
    python batch_factorial_runner.py --dry-run
"""

import os
import pandas as pd
import numpy as np
import random
import logging
import argparse
import time
from pathlib import Path
from datetime import datetime
import json
import torch

# Configure logging with timestamped log files
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
log_filename = f'batch_factorial_runner_{timestamp}.log'

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler(log_filename),  # Timestamped log file
        logging.StreamHandler()  # Still output to console
    ]
)
logger = logging.getLogger(__name__)
logger.info(f"Logging to: {log_filename}")

# Paths
BASE_DIR = Path(__file__).parent
DATA_DIR = BASE_DIR / "data"
CLASSIFICATION_DIR = DATA_DIR / "processed_classification"
REGRESSION_DIR = DATA_DIR / "processed_regression"
RESULTS_DIR = BASE_DIR / "results"
FIGURES_DIR = BASE_DIR / "reports" / "figures"

# Import after path setup
from config import RANDOM_STATE
from data.dynamic_dataset import DynamicDataset
from models.base_models import BaseClassifier, BaseRegressor
from models.advanced_models import XGBoostClassifier, XGBoostRegressor,
        SVMClassifier, SVMRegressor
from models.deep_learning import DeepClassifier, DeepRegressor
from evaluation.metrics import evaluate_classification_model,
        evaluate_regression_model
from evaluation.hyperparameter_tuning import (
    tune_classification_model, tune_regression_model, tune_nn_model,
    tune_linear_regression,
```

```
    LOGISTIC_PARAM_GRID, LINEAR_PARAM_GRID,
    XGBOOST_CLASSIFICATION_PARAM_GRID, XGBOOST_REGRESSION_PARAM_GRID,
    NN_PARAM_GRID
)

# ============================================================================
# SET GLOBAL RANDOM SEEDS FOR REPRODUCIBILITY
# ============================================================================
logger.info("=" * 80)
logger.info("SETTING GLOBAL RANDOM SEEDS FOR REPRODUCIBILITY")
logger.info("=" * 80)
logger.info(f"Random seed: {RANDOM_STATE}")

# Set Python's built-in random module
random.seed(RANDOM_STATE)
logger.info("[OK] Python random seed set")

# Set NumPy random seed
np.random.seed(RANDOM_STATE)
logger.info("[OK] NumPy random seed set")

# Set PyTorch random seeds
torch.manual_seed(RANDOM_STATE)
logger.info("[OK] PyTorch CPU random seed set")

if torch.cuda.is_available():
    torch.cuda.manual_seed(RANDOM_STATE)
    torch.cuda.manual_seed_all(RANDOM_STATE)  # For multi-GPU setups
    logger.info("[OK] PyTorch CUDA random seed set")

    # Make CUDA operations deterministic (may reduce performance slightly)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    logger.info("[OK] CUDA deterministic mode enabled")
    logger.warning("  Note: Deterministic CUDA may reduce performance for
        reproducibility")
else:
    logger.info("  CUDA not available - skipping GPU seed initialization")

logger.info("=" * 80)
logger.info("")

# GPU-compatible parameter grids for base models (PyTorch-based)
GPU_LOGISTIC_PARAM_GRID = {
    'learning_rate': [0.001, 0.01, 0.1],  # Learning rate for gradient
        descent
    'max_iter': [500, 1000, 2000],  # Training iterations
    'batch_size': [5000, 10000]  # Batch size for training
}

GPU_LINEAR_PARAM_GRID = {
    'learning_rate': [0.001, 0.01, 0.1],  # Learning rate for gradient
        descent
    'max_iter': [500, 1000, 2000],  # Training iterations
    'batch_size': [5000, 10000],  # Batch size for training
```

```
    'use_analytical': [True]  # Use analytical solution when possible (more
        accurate)
}

# Corrected parameter grids for batch tuning (without estimator__ prefix)
SVM_CLASSIFICATION_PARAM_GRID_BATCH = {
    'C': [0.1, 1, 2],  # 20x range (0.1 to 2)
    'class_weight': [None, 'balanced']  # Keep 2 for balance
}

SVM_REGRESSION_PARAM_GRID_BATCH = {
    'C': [0.1, 1, 10],  # 100x range (0.1 to 10) for better exploration
    'epsilon': [0.01, 0.1, 0.5, 1.0]  # Wider range for epsilon sensitivity
}
# Total: 3 x 4 = 12 combinations


class FactorialBatchRunner:
    """Runner for factorial design experiments"""

    def __init__(self, include_deep_learning=True, checkpoint_file=None):
        self.include_deep_learning = include_deep_learning
        self.results_summary = []
        self.checkpoint_file = checkpoint_file
        self.completed_datasets = set()

        # Load checkpoint if it exists
        if checkpoint_file and Path(checkpoint_file).exists():
            self._load_checkpoint()
            logger.info(f"Loaded checkpoint from {checkpoint_file}")
            logger.info(f"Already completed: {len(self.completed_datasets)}
         datasets")

    def discover_datasets(self, task=None, dimensionality=None):
        """Discover all datasets in the processed folders."""
        datasets = []

        # Determine which directories to search
        search_dirs = []
        if task in [None, 'classification']:
            search_dirs.append(('classification', CLASSIFICATION_DIR))
        if task in [None, 'regression']:
            search_dirs.append(('regression', REGRESSION_DIR))

        for task_type, base_dir in search_dirs:
            if not base_dir.exists():
                logger.warning(f"{task_type.capitalize()} directory not
         found: {base_dir}")
                continue

            # Get dimensionality folders
            dim_folders = sorted([d for d in base_dir.iterdir() if
         d.is_dir()])

            for dim_folder in dim_folders:
                # Filter by dimensionality if specified
```

```
            if dimensionality and dim_folder.name not in dimensionality:
                continue

            # Find CSV files (excluding _reduced15k versions)
            csv_files = sorted([
                f for f in dim_folder.glob("*.csv")
                if '_reduced15k' not in f.name
            ])

            for csv_file in csv_files:
                datasets.append({
                    'path': csv_file,
                    'task': task_type,
                    'dimensionality': dim_folder.name,
                    'name': csv_file.stem
                })

    logger.info(f"Discovered {len(datasets)} datasets")
    return datasets

def _load_checkpoint(self):
    """Load checkpoint file to resume from previous run."""
    try:
        with open(self.checkpoint_file, 'r') as f:
            checkpoint_data = json.load(f)
            self.completed_datasets =
     set(checkpoint_data.get('completed_datasets', []))

            # Also load previous results if they exist
            if 'results' in checkpoint_data:
                self.results_summary = checkpoint_data['results']
                logger.info(f"Loaded {len(self.results_summary)} previous
     results")
    except Exception as e:
        logger.warning(f"Could not load checkpoint: {e}")
        self.completed_datasets = set()

def _save_checkpoint(self):
    """Save current progress to checkpoint file."""
    if not self.checkpoint_file:
        return

    try:
        checkpoint_data = {
            'completed_datasets': list(self.completed_datasets),
            'results': self.results_summary,
            'timestamp': datetime.now().isoformat()
        }

        # Save to temp file first, then rename (atomic operation)
        temp_file = f"{self.checkpoint_file}.tmp"
        with open(temp_file, 'w') as f:
            json.dump(checkpoint_data, f, indent=2)

        # Atomic rename
```

```
            Path(temp_file).replace(self.checkpoint_file)
            logger.info(f"[OK] Checkpoint saved
         ({len(self.completed_datasets)} datasets completed)")

        except Exception as e:
            logger.error(f"Failed to save checkpoint: {e}")

    def load_and_prepare_dataset(self, dataset_info):
        """Load and prepare a dataset for training."""
        logger.info(f"Loading dataset: {dataset_info['name']}")

        # Read CSV
        df = pd.read_csv(dataset_info['path'])

        # Check if TARGET_VARIABLE exists
        if 'TARGET_VARIABLE' not in df.columns:
            raise ValueError(f"TARGET_VARIABLE column not found in
         {dataset_info['path']}. "
                             "Please run standardize_target_variables.py
         first.")

        # Separate features and target
        X = df.drop('TARGET_VARIABLE', axis=1)
        y = df['TARGET_VARIABLE']

        # Get feature lists
        numerical_features = X.select_dtypes(include=
         [np.number]).columns.tolist()
        categorical_features = X.select_dtypes(exclude=
         [np.number]).columns.tolist()
        feature_variables = numerical_features + categorical_features

        # Create dynamic dataset config
        config = {
            'csv_path': str(dataset_info['path']),
            'target_variable': 'TARGET_VARIABLE',
            'task_type': dataset_info['task'],
            'feature_variables': feature_variables,
            'numerical_features': numerical_features,
            'categorical_features': categorical_features
        }

        # Use DynamicDataset for preprocessing
        dynamic_dataset = DynamicDataset(config)
        X_train, X_test, y_train, y_test = dynamic_dataset.prepare_data()
        feature_names = dynamic_dataset.get_feature_names()

        logger.info(f"Dataset loaded: {X_train.shape[0]} train samples,
         {X_test.shape[0]} test samples")

        return X_train, X_test, y_train, y_test, feature_names

    def get_model_classes(self, task_type):
        """Get model classes based on task type"""
        if task_type == 'classification':
            models = {
```

```
            'Logistic Regression': BaseClassifier,
            'SVM': SVMClassifier,
            'XGBoost': XGBoostClassifier
        }
        if self.include_deep_learning:
            models['MLP'] = DeepClassifier
    else:  # regression
        models = {
            'Linear Regression': BaseRegressor,
            'SVM': SVMRegressor,
            'XGBoost': XGBoostRegressor
        }
        if self.include_deep_learning:
            models['MLP'] = DeepRegressor

    return models

def run_single_experiment(self, dataset_info, model_name, model_class,
                          X_train, X_test, y_train, y_test,
                          tune=False):
    """
    Run a single model experiment.

    Returns:
        List of result dictionaries (one for Default, optionally one for
     Tuned)
    """
    task_type = dataset_info['task']
    results = []

    # ===== ALWAYS RUN DEFAULT CONFIGURATION =====
    logger.info(f"Training {model_name} (Default)...")

    default_result = {
        'dataset': dataset_info['name'],
        'dimensionality': dataset_info['dimensionality'],
        'task': task_type,
        'model': model_name,
        'configuration': 'Default'
    }

    try:
        # Train default model
        start_time = time.time()
        model = model_class()
        model.fit(X_train, y_train)
        training_time = time.time() - start_time

        # Evaluate default model
        start_time = time.time()
        if task_type == 'classification':
            metrics = evaluate_classification_model(model, X_test,
     y_test, model_name=model_name)
        else:
```

```
            metrics = evaluate_regression_model(model, X_test, y_test,
 model_name=model_name)
        prediction_time = time.time() - start_time

        default_result.update(metrics)
        default_result['training_time'] = training_time
        default_result['prediction_time'] = prediction_time
        default_result['status'] = 'success'

    except Exception as e:
        logger.error(f"Error training {model_name} (Default): {str(e)}")
        default_result['status'] = 'failed'
        default_result['error'] = str(e)

    results.append(default_result)

    # ===== RUN TUNED CONFIGURATION IF REQUESTED =====
    if tune and default_result['status'] == 'success':
        logger.info(f"Training {model_name} (Tuned)...")

        tuned_result = {
            'dataset': dataset_info['name'],
            'dimensionality': dataset_info['dimensionality'],
            'task': task_type,
            'model': model_name,
            'configuration': 'Tuned'
        }

        try:
            # Get parameter grid
            if model_name == 'Logistic Regression':
                if hasattr(model, 'is_gpu') and model.is_gpu:
                    param_grid = GPU_LOGISTIC_PARAM_GRID
                else:
                    param_grid = LOGISTIC_PARAM_GRID
            elif model_name == 'Linear Regression':
                if hasattr(model, 'is_gpu') and model.is_gpu:
                    param_grid = GPU_LINEAR_PARAM_GRID
                else:
                    param_grid = LINEAR_PARAM_GRID
            elif model_name == 'XGBoost':
                param_grid = XGBOOST_CLASSIFICATION_PARAM_GRID if
 task_type == 'classification' else XGBOOST_REGRESSION_PARAM_GRID
            elif model_name == 'SVM':
                param_grid = SVM_CLASSIFICATION_PARAM_GRID_BATCH if
 task_type == 'classification' else SVM_REGRESSION_PARAM_GRID_BATCH
            elif model_name == 'MLP':
                param_grid = NN_PARAM_GRID
            else:
                param_grid = None

            if param_grid:
                tuning_start = time.time()

                # Perform tuning
```

```
            if model_name == 'MLP':
                from sklearn.model_selection import train_test_split
                X_train_nn, X_val_nn, y_train_nn, y_val_nn =
train_test_split(
                    X_train, y_train, test_size=0.2,
random_state=RANDOM_STATE
                )
                tuned_model, best_params, _ = tune_nn_model(
                    model_class, param_grid,
                    X_train_nn, y_train_nn, X_val_nn, y_val_nn,
                    task=task_type
                )
            elif model_name == 'Linear Regression' and task_type ==
'regression':
                # Special handling: select best from
Ridge/Lasso/ElasticNet
                tuned_model, best_params, _ = tune_linear_regression(
                    X_train, y_train,
                    param_grid=param_grid,
                    cv=5
                )
            elif model_name in ['Logistic Regression', 'Linear
Regression', 'XGBoost']:
                tune_func = tune_classification_model if task_type ==
'classification' else tune_regression_model
                tuning_wrapper = model_class()
                tuned_model, best_params, _ = tune_func(
                    tuning_wrapper, param_grid,
                    X_train, y_train,
                    method='random' if model_name == 'XGBoost' else
'grid',
                    cv=5
                )
            else:  # SVM
                tune_func = tune_classification_model if task_type ==
'classification' else tune_regression_model
                tuned_model_obj, best_params, _ = tune_func(
                    model.model, param_grid,
                    X_train, y_train,
                    method='random',
                    cv=5
                )
                tuned_model = model_class()
                tuned_model.model = tuned_model_obj

            tuning_time = time.time() - tuning_start

            # Refit tuned model on full training set (skip if already
fitted)
            if model_name == 'Linear Regression' and task_type ==
'regression':
                # tune_linear_regression already refits the best
model
                training_time = 0.0
                logger.info(f"{model_name} (Tuned) already fitted
during tuning")
            else:
```

```
                logger.info(f"Refitting {model_name} (Tuned) with
best parameters on full training set...")
                start_time = time.time()
                tuned_model.fit(X_train, y_train)
                training_time = time.time() - start_time
                logger.info(f"Refitting complete in
{training_time:.2f}s")

            # Evaluate tuned model
            start_time = time.time()
            if task_type == 'classification':
                tuned_metrics =
evaluate_classification_model(tuned_model, X_test, y_test,
model_name=model_name)
            else:
                tuned_metrics =
evaluate_regression_model(tuned_model, X_test, y_test,
model_name=model_name)
            prediction_time = time.time() - start_time

            tuned_result.update(tuned_metrics)
            tuned_result['training_time'] = training_time
            tuned_result['prediction_time'] = prediction_time
            tuned_result['tuning_time'] = tuning_time
            tuned_result['best_params'] = json.dumps(best_params)
            tuned_result['status'] = 'success'

            # Log improvement
            primary_metric = 'f1_macro' if task_type ==
'classification' else 'rmse'
            if task_type == 'classification':
                improvement = ((tuned_metrics[primary_metric] -
default_result[primary_metric]) /
                               abs(default_result[primary_metric])) *
100
                logger.info(f"Tuning successful: {improvement:.2f}%
improvement in {primary_metric}")
            else:
                # For RMSE, lower is better
                improvement = ((default_result[primary_metric] -
tuned_metrics[primary_metric]) /
                               abs(default_result[primary_metric])) *
100
                logger.info(f"Tuning successful: {improvement:.2f}%
improvement in {primary_metric}")

        else:
            logger.warning(f"No parameter grid for {model_name}")
            tuned_result['status'] = 'skipped'
            tuned_result['error'] = 'No parameter grid available'

    except Exception as e:
        logger.error(f"Error tuning {model_name}: {str(e)}")
        import traceback
        logger.error(f"Traceback: {traceback.format_exc()}")
        tuned_result['status'] = 'failed'
        tuned_result['error'] = str(e)
```

```
            results.append(tuned_result)

        return results

    def run_dataset(self, dataset_info, tune=False):
        """
        Run all models on a single dataset.

        Returns:
            List of results dictionaries
        """
        dataset_name = dataset_info['name']

        # Check if already completed
        if dataset_name in self.completed_datasets:
            logger.info(f"[SKIP] {dataset_name} (already completed)")
            return []

        logger.info("=" * 80)
        logger.info(f"Processing: {dataset_name}")
        logger.info(f"Task: {dataset_info['task']}, Dimensionality:
         {dataset_info['dimensionality']}")
        logger.info("=" * 80)

        try:
            # Load and prepare data
            X_train, X_test, y_train, y_test, feature_names =
         self.load_and_prepare_dataset(dataset_info)

            # Get models
            model_classes = self.get_model_classes(dataset_info['task'])

            # Run experiments
            results = []
            for model_name, model_class in model_classes.items():
                # run_single_experiment returns a list (Default + optionally
         Tuned)
                experiment_results = self.run_single_experiment(
                    dataset_info, model_name, model_class,
                    X_train, X_test, y_train, y_test,
                    tune=tune
                )
                results.extend(experiment_results)
                self.results_summary.extend(experiment_results)

            # Mark as completed and save checkpoint
            self.completed_datasets.add(dataset_name)
            self._save_checkpoint()

            return results

        except Exception as e:
            logger.error(f"Failed to process dataset {dataset_name}:
         {str(e)}")
```

```
            logger.error(f"Checkpoint NOT saved for this dataset - can retry
         on next run")
            return []

    def run_batch(self, task=None, dimensionality=None, tune=False,
         dry_run=False):
        """Run batch experiments across all datasets."""
        logger.info("=" * 80)
        logger.info("FACTORIAL DESIGN BATCH RUNNER")
        logger.info("=" * 80)
        logger.info(f"Task: {task or 'both'}")
        logger.info(f"Dimensionality: {dimensionality or 'all'}")
        logger.info(f"Tuning: {tune}")
        logger.info(f"Deep Learning: {self.include_deep_learning}")
        logger.info(f"Dry Run: {dry_run}")
        logger.info("=" * 80)

        # Discover datasets
        datasets = self.discover_datasets(task=task,
         dimensionality=dimensionality)

        # Filter out already completed datasets
        if self.completed_datasets:
            remaining_datasets = [ds for ds in datasets if ds['name'] not in
         self.completed_datasets]
            logger.info(f"Total datasets: {len(datasets)}")
            logger.info(f"Already completed: {len(datasets) -
         len(remaining_datasets)}")
            logger.info(f"Remaining: {len(remaining_datasets)}")
            datasets = remaining_datasets
        else:
            logger.info(f"Total datasets: {len(datasets)}")

        if dry_run:
            logger.info(f"\nWould process {len(datasets)} datasets:")
            for ds in datasets:
                logger.info(f"  - {ds['name']} ({ds['task']},
         {ds['dimensionality']})")
            return

        if not datasets:
            logger.info("No datasets to process (all completed!)")
            return

        # Run experiments
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        total_datasets = len(datasets)

        for idx, dataset_info in enumerate(datasets, 1):
            logger.info(f"\n\nDataset {idx}/{total_datasets}")
            self.run_dataset(dataset_info, tune=tune)

        # Save summary results
        if self.results_summary:
            summary_df = pd.DataFrame(self.results_summary)
```

```
    # Standardize column order for consistency between classification
and regression
    base_columns = ['dataset', 'dimensionality', 'task', 'model',
'configuration']
    timing_columns = ['training_time', 'prediction_time']
    status_columns = ['status']
    optional_columns = ['error', 'tuning_time', 'best_params']

    # Get metric columns (everything not in the predefined lists)
    all_predefined = set(base_columns + timing_columns +
status_columns + optional_columns)
    metric_columns = [col for col in summary_df.columns if col not in
all_predefined]

    # Build final column order: base + metrics + timing + status +
optional (if present)
    final_order = base_columns + metric_columns + timing_columns +
status_columns
    for col in optional_columns:
        if col in summary_df.columns:
            final_order.append(col)

    # Reorder columns
    summary_df = summary_df[final_order]

    # Determine task type for folder organization
    if task is None:
        # Mixed tasks - save to root
        task_folder = RESULTS_DIR
        task_suffix = "mixed"
    else:
        # Single task - save to task-specific folder
        task_folder = RESULTS_DIR / task
        task_suffix = task

    os.makedirs(task_folder, exist_ok=True)
    summary_path = task_folder /
f"factorial_design_results_{task_suffix}_{timestamp}.csv"
    summary_df.to_csv(summary_path, index=False)
    logger.info(f"\nResults saved to: {summary_path}")

    # Print summary statistics
    logger.info("\n" + "=" * 80)
    logger.info("SUMMARY STATISTICS")
    logger.info("=" * 80)
    logger.info(f"Total experiments: {len(summary_df)}")
    logger.info(f"Successful: {(summary_df['status'] ==
'success').sum()}")
    logger.info(f"Failed: {(summary_df['status'] ==
'failed').sum()}")

    # Log failed experiments
    failed_df = summary_df[summary_df['status'] == 'failed']
    if len(failed_df) > 0:
        logger.info("\n" + "=" * 80)
        logger.info("FAILED EXPERIMENTS")
        logger.info("=" * 80)
```

```
        for idx, row in failed_df.iterrows():
            logger.info(f"Dataset: {row['dataset']}, Model:
{row['model']}")
            if 'error' in row and pd.notna(row['error']):
                logger.info(f"  Error: {row['error']}")

    # Calculate tuning performance (if tuning was requested)
    if tune:
        logger.info("\n" + "=" * 80)
        logger.info("TUNING PERFORMANCE")
        logger.info("=" * 80)

        # Calculate improvements by comparing Default and Tuned rows
        improvements = []
        default_df = summary_df[summary_df['configuration'] ==
'Default']
        tuned_df = summary_df[summary_df['configuration'] == 'Tuned']

        for _, default_row in default_df.iterrows():
            # Find matching tuned row
            matching_tuned = tuned_df[
                (tuned_df['dataset'] == default_row['dataset']) &
                (tuned_df['model'] == default_row['model'])
            ]

            if len(matching_tuned) > 0:
                tuned_row = matching_tuned.iloc[0]

                # Calculate improvement based on task type
                if default_row['task'] == 'classification':
                    # Use f1_macro (as per methodology)
                    default_score = default_row['f1_macro']
                    tuned_score = tuned_row['f1_macro']
                    improvement_pct = ((tuned_score - default_score)
/ abs(default_score)) * 100
                else:  # regression
                    # Use RMSE (lower is better)
                    default_score = default_row['rmse']
                    tuned_score = tuned_row['rmse']
                    improvement_pct = ((default_score - tuned_score)
/ abs(default_score)) * 100

                improvements.append({
                    'model': default_row['model'],
                    'dataset': default_row['dataset'],
                    'improvement_pct': improvement_pct,
                    'tuning_time': tuned_row.get('tuning_time', 0)
                })

        if improvements:
            improvements_df = pd.DataFrame(improvements)
            logger.info(f"Average improvement from tuning:
{improvements_df['improvement_pct'].mean():.2f}%")
            logger.info(f"Median improvement from tuning:
{improvements_df['improvement_pct'].median():.2f}%")
```

```
                    # Log improvement by model
                    for model in improvements_df['model'].unique():
                        model_improvements =
            improvements_df[improvements_df['model'] == model]
            ['improvement_pct']
                        if len(model_improvements) > 0:
                            logger.info(f"{model}: avg=
            {model_improvements.mean():.2f}%, median=
            {model_improvements.median():.2f}%")
                else:
                    logger.info("No tuning comparisons available")


def main():
    """Main execution"""
    parser = argparse.ArgumentParser(description='Run factorial design batch
         experiments')
    parser.add_argument('--task', choices=['classification', 'regression'],
         default=None,
                        help='Task type to run (default: both)')
    parser.add_argument('--dimensionality', type=str, nargs='+',
         default=None,
                        help='Specific dimensionality(ies) to run (e.g.,
         medxmed or largexlarge largexmed largexsmall)')
    parser.add_argument('--tune', action='store_true',
                        help='Perform hyperparameter tuning')
    parser.add_argument('--deep-learning', action='store_true', default=True,
                        help='Include deep learning models (MLP)')
    parser.add_argument('--dry-run', action='store_true',
                        help='Show what would be done without running')
    parser.add_argument('--checkpoint', type=str, default=None,
                        help='Checkpoint file path (auto-generated if not
         specified)')
    parser.add_argument('--no-checkpoint', action='store_true',
                        help='Disable checkpoint saving (not recommended for
         large runs)')

    args = parser.parse_args()

    # Create checkpoint file path
    checkpoint_file = None
    if not args.no_checkpoint:
        if args.checkpoint:
            checkpoint_file = args.checkpoint
        else:
            # Auto-generate checkpoint filename
            checkpoint_parts = ['checkpoint']
            if args.task:
                checkpoint_parts.append(args.task)
            if args.dimensionality:
                # Join multiple dimensionalities with underscore
                checkpoint_parts.append('_'.join(args.dimensionality))
            if args.tune:
                checkpoint_parts.append('tuned')
            else:
                checkpoint_parts.append('default')
```

```
            checkpoint_file = f"{'_'.join(checkpoint_parts)}.json"
            logger.info(f"Checkpoint file: {checkpoint_file}")

    # Create runner
    runner = FactorialBatchRunner(
        include_deep_learning=args.deep_learning,
        checkpoint_file=checkpoint_file
    )

    # Run batch
    runner.run_batch(
        task=args.task,
        dimensionality=args.dimensionality,
        tune=args.tune,
        dry_run=args.dry_run
    )

    logger.info("\nBatch processing complete!")

    # Show final checkpoint info
    if checkpoint_file and not args.dry_run:
        logger.info(f"\n[OK] Progress saved to: {checkpoint_file}")
        logger.info(f"[OK] To resume this run later, use the same command")
        logger.info(f"[OK] Completed datasets:
         {len(runner.completed_datasets)}")


if __name__ == "__main__":
    main()
```

3.12: scripts/combine_results.py

Location: `scripts/combine_results.py`

3.13: results/scripts/calculate_stdy.py

Location: `results/scripts/calculate_stdy.py`

```
"""
Combine all factorial design results into a single final_results.csv file.

This script:
1. Loads all individual result files from the results/ folder
2. Removes duplicates (temp files are subsets of combined files)
3. Validates we have all expected experiments
4. Saves to results/final_results.csv
"""

import pandas as pd
import os
from pathlib import Path
```

```
script_dir = Path(__file__).parent
results_dir = script_dir.parent
print(script_dir)
print(results_dir)

def combine_results():
    """Combine all result files into final_results.csv"""

    # Get the results directory (parent of scripts folder)
    script_dir = Path(__file__).parent
    results_dir = script_dir.parent
    print(script_dir)
    print(results_dir)

    # Define files to load (avoiding duplicates)
    # large_obs_fin already contains largexlarge/med/small, so skip temp
        files
    files_to_load = [
        "factorial_design_results_mixed_large_obs_fin.csv",  # Contains
         largexlarge, largexmed, largexsmall
        "factorial_design_results_mixed_medxlarge.csv",
        "factorial_design_results_mixed_medxmed.csv",
        "factorial_design_results_mixed_medxsmall.csv",
        "factorial_design_results_mixed_smallxlarge.csv",
        "factorial_design_results_mixed_smallxmed.csv",
        "factorial_design_results_mixed_smallxsmall.csv",
    ]

    # Load and combine
    dfs = []
    for filename in files_to_load:
        filepath = results_dir / filename
        if filepath.exists():
            df = pd.read_csv(filepath)
            print(f"Loaded {filename}: {len(df)} rows")
            dfs.append(df)
        else:
            print(f"WARNING: {filename} not found!")

    # Combine all dataframes
    combined = pd.concat(dfs, ignore_index=True)
    print(f"\nTotal combined rows: {len(combined)}")

    # Validation checks
    print("\n=== Validation ===")

    # Check dimensionality coverage
    dims = combined['dimensionality'].value_counts().sort_index()
    print(f"\nRows per dimensionality:")
    for dim, count in dims.items():
        status = "[OK]" if count == 64 else "[X]"
        print(f"  {dim}: {count} {status}")

    # Check expected totals
    expected_dims = 9
```

```
    expected_per_dim = 64  # 8 datasets x 4 models x 2 configs
    expected_total = expected_dims * expected_per_dim

    print(f"\nExpected: {expected_total} rows ({expected_dims} dims x
         {expected_per_dim} per dim)")
    print(f"Actual: {len(combined)} rows")

    if len(combined) == expected_total:
        print("[OK] All experiments accounted for!")
    else:
        print(f"[X] Missing {expected_total - len(combined)} experiments")

    # Additional checks
    print(f"\nTasks: {combined['task'].value_counts().to_dict()}")
    print(f"Models: {combined['model'].unique().tolist()}")
    print(f"Configurations:
         {combined['configuration'].value_counts().to_dict()}")
    print(f"Datasets: {combined['dataset'].nunique()} unique")

    # Save to final_results.csv
    output_path = results_dir / "final_results.csv"
    combined.to_csv(output_path, index=False)
    print(f"\n[OK] Saved to {output_path}")

    return combined

if __name__ == "__main__":
    df = combine_results()
```

3.14: scripts/compute_normalized_metrics.py

Location: `scripts/compute_normalized_metrics.py`

```
"""
Compute normalized error metrics for regression results.

Usage:
    python scripts/compute_normalized_metrics.py
"""

import pandas as pd
import numpy as np
from pathlib import Path


def compute_normalized_metrics():
    """Compute NRMSE, NMAE, NMSE for regression results."""

    # Load results with stdy
    results_path = Path('results/final_results_with_stdy.csv')
    if not results_path.exists():
        print("ERROR: Run calculate_stdy.py first!")
        return None
```

```
    df = pd.read_csv(results_path)

    # Filter to regression only
    df_reg = df[df['task'] == 'regression'].copy()

    print(f"Computing normalized metrics for {len(df_reg)} regression
        experiments...")

    # Compute normalized metrics
    df_reg['NRMSE'] = df_reg['rmse'] / df_reg['stdy']
    df_reg['NMAE'] = df_reg['mae'] / df_reg['stdy']
    df_reg['NMSE'] = df_reg['mse'] / (df_reg['stdy'] ** 2)

    # Check for any issues
    print(f"\nMetric statistics:")
    print(f"  NRMSE - mean: {df_reg['NRMSE'].mean():.4f}, max:
        {df_reg['NRMSE'].max():.4f}")
    print(f"  NMAE  - mean: {df_reg['NMAE'].mean():.4f}, max:
        {df_reg['NMAE'].max():.4f}")
    print(f"  NMSE  - mean: {df_reg['NMSE'].mean():.4f}, max:
        {df_reg['NMSE'].max():.4f}")

    # Flag potential model failures (NRMSE > 3 indicates poor fit)
    failures = df_reg[df_reg['NRMSE'] > 3]
    if len(failures) > 0:
        print(f"\nWarning: {len(failures)} experiments have NRMSE > 3
         (potential failures)")
        print(failures[['dataset', 'model', 'configuration',
         'NRMSE']].to_string())

    # Merge back into full results
    df.loc[df['task'] == 'regression', 'NRMSE'] = df_reg['NRMSE'].values
    df.loc[df['task'] == 'regression', 'NMAE'] = df_reg['NMAE'].values
    df.loc[df['task'] == 'regression', 'NMSE'] = df_reg['NMSE'].values

    # Save
    output_path = Path('results/final_results_normalized.csv')
    df.to_csv(output_path, index=False)

    print(f"\nSaved normalized results to: {output_path}")

    return df


if __name__ == '__main__':
    compute_normalized_metrics()
```

3.15: scripts/compute_rankings.py

Location: `scripts/compute_rankings.py`

```
"""
Compute model rankings and perform Friedman test.

Usage:
```

```
    python scripts/compute_rankings.py --task regression --metric NRMSE
    python scripts/compute_rankings.py --task classification --metric
         f1_macro
    python scripts/compute_rankings.py --all    # Auto-generate ALL rankings
"""

import pandas as pd
import numpy as np
from scipy.stats import friedmanchisquare
from pathlib import Path
import argparse


# Define all subset filters for auto-generation
SUBSET_FILTERS = {
    # By observation size
    'obs_small': 'smallx',
    'obs_medium': 'medx',
    'obs_large': 'largex',

    # By noise
    'noisy': 'Noisy_True',
    'clean': 'Noisy_False',

    # By balance (classification only)
    'balanced': 'Bal_True',
    'imbalanced': 'Bal_False',

    # By linearity (regression only)
    'linear': 'Nonlinear_False',
    'nonlinear': 'Nonlinear_True',

    # Combined
    'large_obs_clean': ['largex', 'Noisy_False'],
    'large_obs_noisy': ['largex', 'Noisy_True'],
}

# Default metrics per task
DEFAULT_METRICS = {
    'regression': ['NRMSE', 'NMAE', 'NMSE'],
    'classification': ['f1_macro', 'accuracy', 'roc_auc']
}


def apply_filters(df, filters):
    """Apply one or more filters to dataframe."""
    if isinstance(filters, str):
        filters = [filters]

    mask = pd.Series([True] * len(df), index=df.index)
    for f in filters:
        mask &= df['dataset'].str.contains(f, case=False, na=False)

    return df[mask]
```

```python
def compute_rankings(task='regression', metric='NRMSE', subset_filter=None,
        df=None, output_suffix=''):
    """Compute model rankings and Friedman test."""

    if df is None:
        results_path = Path('results/final_results_normalized.csv')
        if not results_path.exists():
            results_path = Path('results/final_results.csv')
        df = pd.read_csv(results_path)

    # Filter by task
    df = df[df['task'] == task].copy()

    # Apply subset filter if specified
    filter_desc = None
    if subset_filter:
        if isinstance(subset_filter, list):
            df = apply_filters(df, subset_filter)
            filter_desc = ' & '.join(subset_filter)
        else:
            df = df[df['dataset'].str.contains(subset_filter)]
            filter_desc = subset_filter

    if len(df) == 0:
        print(f"  WARNING: No data for {task} with filter {subset_filter}")
        return None, None, None, None

    # Check if metric exists
    if metric not in df.columns:
        # Try alternative column names
        alt_metrics = {'NRMSE': 'nrmse', 'NMAE': 'nmae', 'NMSE': 'nmse',
                       'f1_macro': 'f1', 'roc_auc': 'auc'}
        if metric in alt_metrics and alt_metrics[metric] in df.columns:
            metric = alt_metrics[metric]
        else:
            print(f"  WARNING: Metric {metric} not found in data")
            return None, None, None, None

    print(f"\nTask: {task}")
    print(f"Metric: {metric}")
    print(f"Datasets: {df['dataset'].nunique()}")
    print(f"Subset filter: {filter_desc or 'None'}")

    # Create model+config label
    df['model_config'] = df['model'] + ' (' + df['configuration'] + ')'

    # For regression metrics (NRMSE, NMAE, NMSE), lower is better
    # For classification metrics (f1_macro, accuracy, roc_auc), higher is
        better
    ascending = metric.lower() in ['nrmse', 'nmae', 'nmse', 'rmse', 'mae',
        'mse']

    # Rank each dataset separately
    df['rank'] = df.groupby('dataset')[metric].rank(method='average',
        ascending=ascending)
```

```python
# Average ranks per model
avg_ranks = df.groupby(['model', 'configuration'])
     ['rank'].mean().sort_values()

print("\n" + "=" * 60)
print(f"AVERAGE RANKS (by {metric})")
print("=" * 60)
for (model, config), rank in avg_ranks.items():
    print(f"  {model} ({config}): {rank:.3f}")

# Pivot for Friedman test
pivot = df.pivot_table(
    index='dataset',
    columns='model_config',
    values='rank'
)

# Friedman test (requires at least 3 groups and 2 samples)
if pivot.shape[1] >= 3 and pivot.shape[0] >= 2:
    try:
        friedman_stat, friedman_p = friedmanchisquare(*pivot.T.values)

        print("\n" + "=" * 60)
        print("FRIEDMAN TEST")
        print("=" * 60)
        print(f"  Chi-square statistic: {friedman_stat:.4f}")
        print(f"  p-value: {friedman_p:.6f}")

        if friedman_p < 0.05:
            print("  -> Significant difference between models (p <
     0.05)")
            print("  -> Proceed to post-hoc Nemenyi test")
        else:
            print("  -> No significant difference between models (p >=
     0.05)")
    except Exception as e:
        print(f"  WARNING: Friedman test failed: {e}")
        friedman_stat, friedman_p = None, None
else:
    print(f"  WARNING: Not enough data for Friedman test (need >= 3
     groups, >= 2 datasets)")
    friedman_stat, friedman_p = None, None

# Save rankings
output_dir = Path('analysis_outputs/rankings')
output_dir.mkdir(parents=True, exist_ok=True)

suffix = output_suffix if output_suffix else (f"_{subset_filter.replace('
     ', '_')}" if isinstance(subset_filter, str) and subset_filter else
     "")
avg_ranks.to_csv(output_dir / f'{task}_rankings_{metric}{suffix}.csv')

# Save Friedman results
if friedman_stat is not None:
    friedman_results = pd.DataFrame([{
```

```
            'task': task,
            'metric': metric,
            'subset': filter_desc or 'all',
            'n_datasets': df['dataset'].nunique(),
            'n_models': pivot.shape[1],
            'chi_square': friedman_stat,
            'p_value': friedman_p,
            'significant': friedman_p < 0.05
        }])
        friedman_results.to_csv(output_dir / f'{task}_friedman_{metric}
         {suffix}.csv', index=False)

    return avg_ranks, friedman_stat, friedman_p, pivot


def generate_all_rankings():
    """Auto-generate rankings for ALL predefined subsets and metrics."""

    print("=" * 80)
    print("AUTO-GENERATING RANKINGS FOR ALL SUBSETS AND METRICS")
    print("=" * 80)

    # Load data once
    results_path = Path('results/final_results_normalized.csv')
    if not results_path.exists():
        results_path = Path('results/final_results.csv')
    df = pd.read_csv(results_path)
    print(f"Loaded {len(df)} rows from {results_path}")

    output_dir = Path('analysis_outputs/rankings')
    output_dir.mkdir(parents=True, exist_ok=True)

    all_results = []

    # Process each task
    for task in ['classification', 'regression']:
        task_df = df[df['task'] == task].copy()
        if len(task_df) == 0:
            continue

        # Determine available metrics
        if task == 'classification':
            metrics = [m for m in ['f1_macro', 'f1', 'accuracy', 'roc_auc',
         'auc']
                       if m in task_df.columns]
        else:
            metrics = [m for m in ['NRMSE', 'nrmse', 'NMAE', 'nmae', 'NMSE',
         'nmse', 'r2']
                       if m in task_df.columns]

        print(f"\n{'='*80}")
        print(f"TASK: {task.upper()}")
        print(f"Available metrics: {metrics}")
        print(f"{'='*80}")
```

```
    # Overall rankings (no filter)
    for metric in metrics:
        print(f"\n--- Overall ({metric}) ---")
        avg_ranks, stat, p, _ = compute_rankings(
            task=task, metric=metric, subset_filter=None,
            df=task_df.copy(), output_suffix='_overall'
        )
        if stat is not None:
            all_results.append({
                'task': task, 'metric': metric, 'subset': 'overall',
                'chi_square': stat, 'p_value': p, 'significant': p < 0.05
            })

    # Subset rankings
    for subset_name, subset_filter in SUBSET_FILTERS.items():
        # Skip classification-only filters for regression and vice versa
        if task == 'regression' and subset_name in ['balanced',
     'imbalanced']:
            continue
        if task == 'classification' and subset_name in ['linear',
     'nonlinear']:
            continue

        for metric in metrics[:2]:  # Limit to top 2 metrics per subset
            print(f"\n--- {subset_name} ({metric}) ---")
            avg_ranks, stat, p, _ = compute_rankings(
                task=task, metric=metric, subset_filter=subset_filter,
                df=task_df.copy(), output_suffix=f'_{subset_name}'
            )
            if stat is not None:
                all_results.append({
                    'task': task, 'metric': metric, 'subset':
     subset_name,
                    'chi_square': stat, 'p_value': p, 'significant': p <
     0.05
                })

# Save consolidated Friedman results
if all_results:
    consolidated = pd.DataFrame(all_results)
    consolidated.to_csv(output_dir / 'friedman_all_results.csv',
     index=False)
    print(f"\n\nConsolidated Friedman results saved to:
     {output_dir}/friedman_all_results.csv")

    # Print summary
    print("\n" + "=" * 80)
    print("FRIEDMAN TEST SUMMARY")
    print("=" * 80)
    print(consolidated.to_string(index=False))

print("\n" + "=" * 80)
print("ALL RANKINGS GENERATED!")
print("=" * 80)
print(f"\nOutput directory: {output_dir}/")
```

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Compute model rankings and
        Friedman test')
    parser.add_argument('--task', choices=['regression', 'classification'],
        default='regression')
    parser.add_argument('--metric', type=str, default='NRMSE')
    parser.add_argument('--subset', type=str, default=None)
    parser.add_argument('--all', action='store_true',
                        help='Auto-generate rankings for ALL subsets and
        metrics')
    args = parser.parse_args()

    if args.all:
        generate_all_rankings()
    else:
        compute_rankings(args.task, args.metric, args.subset)
```

3.16: scripts/nemenyi_posthoc.py

Location: `scripts/nemenyi_posthoc.py`

```
"""
Perform Nemenyi post-hoc test and compute pairwise significance.

Usage:
    python scripts/nemenyi_posthoc.py --task regression --metric NRMSE
    python scripts/nemenyi_posthoc.py --task classification --metric f1_macro
    python scripts/nemenyi_posthoc.py --all    # Auto-generate ALL post-hoc
        tests
"""

import pandas as pd
import numpy as np
from scipy.stats import friedmanchisquare
from math import sqrt
import itertools
from pathlib import Path
import argparse


# Define all subset filters for auto-generation
SUBSET_FILTERS = {
    # By observation size
    'obs_small': 'smallx',
    'obs_medium': 'medx',
    'obs_large': 'largex',

    # By noise
    'noisy': 'Noisy_True',
    'clean': 'Noisy_False',

    # By balance (classification only)
    'balanced': 'Bal_True',
```

```
    'imbalanced': 'Bal_False',

    # By linearity (regression only)
    'linear': 'Nonlinear_False',
    'nonlinear': 'Nonlinear_True',

    # Combined
    'large_obs_clean': ['largex', 'Noisy_False'],
    'large_obs_noisy': ['largex', 'Noisy_True'],
}


def apply_filters(df, filters):
    """Apply one or more filters to dataframe."""
    if isinstance(filters, str):
        filters = [filters]

    mask = pd.Series([True] * len(df), index=df.index)
    for f in filters:
        mask &= df['dataset'].str.contains(f, case=False, na=False)

    return df[mask]


def nemenyi_cd(k, N, alpha=0.05):
    """Compute Nemenyi critical difference."""
    # Critical values from Demsar (2006) Table 5
    # Using normal approximation for large k
    q_alpha = {
        0.10: 1.645,
        0.05: 1.960,
        0.01: 2.576
    }.get(alpha, 1.960)

    return q_alpha * sqrt(k * (k + 1) / (6 * N))


def nemenyi_posthoc(task='regression', metric='NRMSE', subset_filter=None,
         alpha=0.05,
                    df=None, output_suffix=''):
    """Perform Nemenyi post-hoc test."""

    if df is None:
        results_path = Path('results/final_results_normalized.csv')
        if not results_path.exists():
            results_path = Path('results/final_results.csv')
        df = pd.read_csv(results_path)

    # Filter by task
    df = df[df['task'] == task].copy()

    # Apply subset filter if specified
    filter_desc = None
    if subset_filter:
        if isinstance(subset_filter, list):
```

```
        df = apply_filters(df, subset_filter)
        filter_desc = ' & '.join(subset_filter)
    else:
        df = df[df['dataset'].str.contains(subset_filter)]
        filter_desc = subset_filter

if len(df) == 0:
    print(f"  WARNING: No data for {task} with filter {subset_filter}")
    return None, None, None, None

# Check if metric exists
if metric not in df.columns:
    alt_metrics = {'NRMSE': 'nrmse', 'NMAE': 'nmae', 'NMSE': 'nmse',
                   'f1_macro': 'f1', 'roc_auc': 'auc'}
    if metric in alt_metrics and alt_metrics[metric] in df.columns:
        metric = alt_metrics[metric]
    else:
        print(f"  WARNING: Metric {metric} not found in data")
        return None, None, None, None

# Create model+config label
df['model_config'] = df['model'] + ' (' + df['configuration'] + ')'

# For regression metrics, lower is better
ascending = metric.lower() in ['nrmse', 'nmae', 'nmse', 'rmse', 'mae',
     'mse']

# Rank each dataset separately
df['rank'] = df.groupby('dataset')[metric].rank(method='average',
     ascending=ascending)

# Average ranks per model
avg_ranks = df.groupby('model_config')['rank'].mean().sort_values()

# Pivot for test
pivot = df.pivot_table(
    index='dataset',
    columns='model_config',
    values='rank'
)

# Compute critical difference
k = len(avg_ranks)  # Number of model-config combinations
N = len(pivot)       # Number of datasets

if k < 2 or N < 2:
    print(f"  WARNING: Not enough data for Nemenyi test (k={k}, N={N})")
    return None, None, None, None

CD = nemenyi_cd(k, N, alpha)

print("=" * 70)
print(f"NEMENYI POST-HOC TEST")
print(f"Task: {task} | Metric: {metric} | Alpha: {alpha}")
print(f"Subset: {filter_desc or 'All'}")
```

```
print("=" * 70)
print(f"\nNumber of classifiers (k): {k}")
print(f"Number of datasets (N): {N}")
print(f"Critical Difference (CD): {CD:.4f}")

# Print average ranks
print("\n" + "-" * 70)
print("AVERAGE RANKS")
print("-" * 70)
for model, rank in avg_ranks.items():
    print(f"  {model}: {rank:.3f}")

# Compute all pairwise comparisons
pairs = []
for (m1, r1), (m2, r2) in itertools.combinations(avg_ranks.items(), 2):
    diff = abs(r1 - r2)
    significant = diff > CD
    pairs.append({
        'Model A': m1,
        'Model B': m2,
        'Rank A': round(r1, 3),
        'Rank B': round(r2, 3),
        'Rank Diff': round(diff, 3),
        'CD': round(CD, 3),
        'Significant': significant
    })

df_pairs = pd.DataFrame(pairs)

# Filter to significant pairs
df_sig = df_pairs[df_pairs['Significant']].copy()
df_sig = df_sig.sort_values('Rank Diff', ascending=False)

print(f"\n" + "-" * 70)
print(f"SIGNIFICANT PAIRWISE DIFFERENCES ({len(df_sig)} of
     {len(df_pairs)} pairs)")
print("-" * 70)

if len(df_sig) > 0:
    pd.set_option('display.max_colwidth', 40)
    pd.set_option('display.width', 200)
    print(df_sig[['Model A', 'Model B', 'Rank Diff',
     'Significant']].to_string(index=False))
else:
    print("No significant pairwise differences found.")

# Save results
output_dir = Path('analysis_outputs/posthoc')
output_dir.mkdir(parents=True, exist_ok=True)

suffix = output_suffix if output_suffix else (f"_{subset_filter.replace('
     ', '_')}" if isinstance(subset_filter, str) and subset_filter else
     "")
df_pairs.to_csv(output_dir / f'{task}_nemenyi_{metric}{suffix}_all.csv',
     index=False)
```

```
    df_sig.to_csv(output_dir / f'{task}_nemenyi_{metric}
        {suffix}_significant.csv', index=False)

    # Save average ranks
    avg_ranks.to_csv(output_dir / f'{task}_avg_ranks_{metric}{suffix}.csv')

    # Save summary
    summary = pd.DataFrame([{
        'task': task,
        'metric': metric,
        'subset': filter_desc or 'all',
        'n_classifiers': k,
        'n_datasets': N,
        'critical_difference': CD,
        'alpha': alpha,
        'n_significant_pairs': len(df_sig),
        'n_total_pairs': len(df_pairs)
    }])
    summary.to_csv(output_dir / f'{task}_nemenyi_{metric}
        {suffix}_summary.csv', index=False)

    print(f"\nResults saved to: {output_dir}/")

    return df_pairs, df_sig, CD, avg_ranks


def generate_all_posthoc():
    """Auto-generate Nemenyi post-hoc tests for ALL predefined subsets and
        metrics."""

    print("=" * 80)
    print("AUTO-GENERATING NEMENYI POST-HOC TESTS FOR ALL SUBSETS AND
        METRICS")
    print("=" * 80)

    # Load data once
    results_path = Path('results/final_results_normalized.csv')
    if not results_path.exists():
        results_path = Path('results/final_results.csv')
    df = pd.read_csv(results_path)
    print(f"Loaded {len(df)} rows from {results_path}")

    output_dir = Path('analysis_outputs/posthoc')
    output_dir.mkdir(parents=True, exist_ok=True)

    all_results = []

    # Process each task
    for task in ['classification', 'regression']:
        task_df = df[df['task'] == task].copy()
        if len(task_df) == 0:
            continue

        # Determine available metrics
        if task == 'classification':
```

```
            metrics = [m for m in ['f1_macro', 'f1', 'accuracy', 'roc_auc',
        'auc']
                       if m in task_df.columns]
        else:
            metrics = [m for m in ['NRMSE', 'nrmse', 'NMAE', 'nmae', 'NMSE',
        'nmse', 'r2']
                       if m in task_df.columns]

        print(f"\n{'='*80}")
        print(f"TASK: {task.upper()}")
        print(f"Available metrics: {metrics}")
        print(f"{'='*80}")

        # Overall post-hoc (no filter)
        for metric in metrics[:2]:  # Top 2 metrics
            print(f"\n--- Overall ({metric}) ---")
            df_pairs, df_sig, CD, avg_ranks = nemenyi_posthoc(
                task=task, metric=metric, subset_filter=None,
                df=task_df.copy(), output_suffix='_overall'
            )
            if CD is not None:
                all_results.append({
                    'task': task, 'metric': metric, 'subset': 'overall',
                    'CD': CD, 'n_significant': len(df_sig) if df_sig is not
        None else 0,
                    'n_total_pairs': len(df_pairs) if df_pairs is not None
        else 0
                })

        # Subset post-hoc tests
        for subset_name, subset_filter in SUBSET_FILTERS.items():
            # Skip classification-only filters for regression and vice versa
            if task == 'regression' and subset_name in ['balanced',
        'imbalanced']:
                continue
            if task == 'classification' and subset_name in ['linear',
        'nonlinear']:
                continue

            for metric in metrics[:1]:  # Top 1 metric per subset to reduce
        output
                print(f"\n--- {subset_name} ({metric}) ---")
                df_pairs, df_sig, CD, avg_ranks = nemenyi_posthoc(
                    task=task, metric=metric, subset_filter=subset_filter,
                    df=task_df.copy(), output_suffix=f'_{subset_name}'
                )
                if CD is not None:
                    all_results.append({
                        'task': task, 'metric': metric, 'subset':
        subset_name,
                        'CD': CD, 'n_significant': len(df_sig) if df_sig is
        not None else 0,
                        'n_total_pairs': len(df_pairs) if df_pairs is not
        None else 0
                    })

    # Save consolidated results
```

```
    if all_results:
        consolidated = pd.DataFrame(all_results)
        consolidated.to_csv(output_dir / 'nemenyi_all_results.csv',
         index=False)
        print(f"\n\nConsolidated Nemenyi results saved to:
         {output_dir}/nemenyi_all_results.csv")

        # Print summary
        print("\n" + "=" * 80)
        print("NEMENYI POST-HOC TEST SUMMARY")
        print("=" * 80)
        print(consolidated.to_string(index=False))

    print("\n" + "=" * 80)
    print("ALL POST-HOC TESTS GENERATED!")
    print("=" * 80)
    print(f"\nOutput directory: {output_dir}/")


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Perform Nemenyi post-hoc
         test')
    parser.add_argument('--task', choices=['regression', 'classification'],
         default='regression')
    parser.add_argument('--metric', type=str, default='NRMSE')
    parser.add_argument('--subset', type=str, default=None)
    parser.add_argument('--alpha', type=float, default=0.05)
    parser.add_argument('--all', action='store_true',
                        help='Auto-generate post-hoc tests for ALL subsets and
         metrics')
    args = parser.parse_args()

    if args.all:
        generate_all_posthoc()
    else:
        nemenyi_posthoc(args.task, args.metric, args.subset, args.alpha)
```

3.17: scripts/generate_summary_tables.py

Location: `scripts/generate_summary_tables.py`

```
"""
Generate summary tables for dissertation.

Usage:
    python scripts/generate_summary_tables.py                    # Basic
         summary
    python scripts/generate_summary_tables.py --subset "Noisy_True"  # Single
         subset
    python scripts/generate_summary_tables.py --all              # Auto-
         generate ALL subsets
"""

import pandas as pd
import numpy as np
```

```python
from pathlib import Path
import argparse


# Define all subset filters for auto-generation
SUBSET_FILTERS = {
    # By observation size
    'obs_small': 'smallx',
    'obs_medium': 'medx',
    'obs_large': 'largex',

    # By feature size
    'feat_small': 'xsmall',
    'feat_medium': 'xmed',
    'feat_large': 'xlarge',

    # By noise
    'noisy': 'Noisy_True',
    'clean': 'Noisy_False',

    # By balance (classification)
    'balanced': 'Bal_True',
    'imbalanced': 'Bal_False',

    # By linearity (regression)
    'linear': 'Nonlinear_False',
    'nonlinear': 'Nonlinear_True',

    # Combined - Large observation datasets
    'large_obs_clean': ['largex', 'Noisy_False'],
    'large_obs_noisy': ['largex', 'Noisy_True'],
    'large_obs_balanced': ['largex', 'Bal_True'],
    'large_obs_imbalanced': ['largex', 'Bal_False'],
    'large_obs_linear': ['largex', 'Nonlinear_False'],
    'large_obs_nonlinear': ['largex', 'Nonlinear_True'],

    # Combined - Medium observation datasets
    'medium_obs_clean': ['medx', 'Noisy_False'],
    'medium_obs_noisy': ['medx', 'Noisy_True'],

    # Combined - Small observation datasets
    'small_obs_clean': ['smallx', 'Noisy_False'],
    'small_obs_noisy': ['smallx', 'Noisy_True'],
}


def apply_filters(df, filters):
    """Apply one or more filters to dataframe."""
    if isinstance(filters, str):
        filters = [filters]

    mask = pd.Series([True] * len(df), index=df.index)
    for f in filters:
        mask &= df['dataset'].str.contains(f, case=False, na=False)
```

```
    return df[mask]


def generate_summary_tables(subset_filter=None, df=None, output_suffix=''):
    """Generate summary statistics tables."""

    if df is None:
        results_path = Path('results/final_results_normalized.csv')
        if not results_path.exists():
            results_path = Path('results/final_results.csv')
        df = pd.read_csv(results_path)

    # Apply subset filter if specified
    if subset_filter:
        if isinstance(subset_filter, list):
            df = apply_filters(df, subset_filter)
            filter_desc = ' & '.join(subset_filter)
        else:
            df = df[df['dataset'].str.contains(subset_filter)]
            filter_desc = subset_filter
        print(f"Filtered to subset: {filter_desc} ({len(df)} rows)")

    if len(df) == 0:
        print(f"  WARNING: No data after filtering!")
        return None, None

    # Create model+config label
    df['model_config'] = df['model'] + ' (' + df['configuration'] + ')'

    output_dir = Path('analysis_outputs/summary_tables')
    output_dir.mkdir(parents=True, exist_ok=True)

    summary_reg = None
    summary_class = None

    # ===== REGRESSION SUMMARY =====
    df_reg = df[df['task'] == 'regression'].copy()

    if len(df_reg) > 0:
        # Determine available metrics
        agg_dict = {}
        for metric, col_options in [('NRMSE', ['NRMSE', 'nrmse']),
                                    ('NMAE', ['NMAE', 'nmae']),
                                    ('NMSE', ['NMSE', 'nmse']),
                                    ('R2', ['r2', 'R2'])]:
            for col in col_options:
                if col in df_reg.columns:
                    agg_dict[f'mean_{metric}'] = (col, 'mean')
                    agg_dict[f'std_{metric}'] = (col, 'std')
                    break

        if agg_dict:
            summary_reg = (
                df_reg
                .groupby(['model', 'configuration'], as_index=False)
```

```
            .agg(**agg_dict)
        )
        if 'mean_NRMSE' in summary_reg.columns:
            summary_reg = summary_reg.sort_values('mean_NRMSE')

        suffix = output_suffix if output_suffix else
    (f"_{subset_filter.replace(' ', '_')}" if isinstance(subset_filter,
    str) and subset_filter else "")
        summary_reg.to_csv(output_dir /
    f'regression_summary{suffix}.csv', index=False)

        print("\n" + "=" * 60)
        print(f"REGRESSION SUMMARY{' (' + filter_desc + ')' if
    subset_filter else ''}")
        print("=" * 60)
        print(summary_reg.to_string(index=False))

# ===== CLASSIFICATION SUMMARY =====
df_class = df[df['task'] == 'classification'].copy()

if len(df_class) > 0:
    agg_dict = {}
    for metric, col_options in [('F1_macro', ['f1_macro', 'f1']),
                                ('Accuracy', ['accuracy']),
                                ('ROC_AUC', ['roc_auc', 'auc'])]:
        for col in col_options:
            if col in df_class.columns:
                agg_dict[f'mean_{metric}'] = (col, 'mean')
                agg_dict[f'std_{metric}'] = (col, 'std')
                break

    if agg_dict:
        summary_class = (
            df_class
            .groupby(['model', 'configuration'], as_index=False)
            .agg(**agg_dict)
        )
        if 'mean_F1_macro' in summary_class.columns:
            summary_class = summary_class.sort_values('mean_F1_macro',
    ascending=False)

        suffix = output_suffix if output_suffix else
    (f"_{subset_filter.replace(' ', '_')}" if isinstance(subset_filter,
    str) and subset_filter else "")
        summary_class.to_csv(output_dir /
    f'classification_summary{suffix}.csv', index=False)

        print("\n" + "=" * 60)
        print(f"CLASSIFICATION SUMMARY{' (' + filter_desc + ')' if
    subset_filter else ''}")
        print("=" * 60)
        print(summary_class.to_string(index=False))

print(f"\nSummary tables saved to: {output_dir}/")

return summary_reg, summary_class
```

```python
def generate_all_subsets():
    """Auto-generate summary tables for ALL predefined subsets."""

    print("=" * 80)
    print("AUTO-GENERATING SUMMARY TABLES FOR ALL SUBSETS")
    print("=" * 80)

    # Load data once
    results_path = Path('results/final_results_normalized.csv')
    if not results_path.exists():
        results_path = Path('results/final_results.csv')
    df = pd.read_csv(results_path)
    print(f"Loaded {len(df)} rows from {results_path}")

    # Generate overall summary first
    print("\n" + "=" * 80)
    print("OVERALL SUMMARY (no filter)")
    print("=" * 80)
    generate_summary_tables(subset_filter=None, df=df.copy(),
         output_suffix='_overall')

    # Generate for each subset
    for name, filters in SUBSET_FILTERS.items():
        print("\n" + "=" * 80)
        print(f"SUBSET: {name}")
        print("=" * 80)
        generate_summary_tables(subset_filter=filters, df=df.copy(),
         output_suffix=f'_{name}')

    # Generate consolidated report
    generate_consolidated_report(df)

    print("\n" + "=" * 80)
    print("ALL SUMMARY TABLES GENERATED!")
    print("=" * 80)
    print("\nOutput directory: analysis_outputs/summary_tables/")


def generate_consolidated_report(df):
    """Generate a single consolidated report with all breakdowns."""

    output_dir = Path('analysis_outputs/summary_tables')
    output_dir.mkdir(parents=True, exist_ok=True)

    # Parse properties from dataset names
    df = df.copy()
    df['obs_size'] = df['dimensionality'].apply(lambda x: x.split('x')[0] if
         'x' in str(x) else 'unknown')
    df['feat_size'] = df['dimensionality'].apply(lambda x: x.split('x')[1] if
         'x' in str(x) else 'unknown')
    df['is_noisy'] = df['dataset'].str.contains('Noisy_True', case=False,
         na=False)
    df['is_balanced'] = df['dataset'].str.contains('Bal_True', case=False,
         na=False)
```

```
df['is_nonlinear'] = df['dataset'].str.contains('Nonlinear_True',
     case=False, na=False)

# ===== CLASSIFICATION CONSOLIDATED =====
df_class = df[df['task'] == 'classification'].copy()

if len(df_class) > 0:
    print("\n" + "=" * 80)
    print("CONSOLIDATED CLASSIFICATION REPORT")
    print("=" * 80)

    rows = []
    for obs in ['small', 'med', 'large']:
        for balanced in [True, False]:
            for noisy in [True, False]:
                subset = df_class[
                    (df_class['obs_size'] == obs) &
                    (df_class['is_balanced'] == balanced) &
                    (df_class['is_noisy'] == noisy)
                ]

                bal_label = 'Balanced' if balanced else 'Imbalanced'
                noise_label = 'Noisy' if noisy else 'Clean'

                for config in ['Default', 'Tuned']:
                    config_subset = subset[subset['configuration'] ==
     config]
                    for model in sorted(df_class['model'].unique()):
                        model_subset =
     config_subset[config_subset['model'] == model]
                        if len(model_subset) > 0:
                            row = {
                                'Obs_Size': obs.capitalize(),
                                'Balance': bal_label,
                                'Noise': noise_label,
                                'Config': config,
                                'Model': model,
                                'N': len(model_subset)
                            }
                            for metric in ['f1_macro', 'accuracy',
     'roc_auc']:
                                if metric in model_subset.columns:
                                    row[f'{metric}_mean'] =
     model_subset[metric].mean()
                                    row[f'{metric}_std'] =
     model_subset[metric].std()
                            rows.append(row)

    if rows:
        consolidated_class = pd.DataFrame(rows)
        consolidated_class.to_csv(output_dir /
     'classification_consolidated.csv', index=False)
        print(f"Saved: classification_consolidated.csv
     ({len(consolidated_class)} rows)")

# ===== REGRESSION CONSOLIDATED =====
```

```
    df_reg = df[df['task'] == 'regression'].copy()

    if len(df_reg) > 0:
        print("\n" + "=" * 80)
        print("CONSOLIDATED REGRESSION REPORT")
        print("=" * 80)

        rows = []
        for obs in ['small', 'med', 'large']:
            for nonlinear in [True, False]:
                for noisy in [True, False]:
                    subset = df_reg[
                        (df_reg['obs_size'] == obs) &
                        (df_reg['is_nonlinear'] == nonlinear) &
                        (df_reg['is_noisy'] == noisy)
                    ]

                    lin_label = 'Non-linear' if nonlinear else 'Linear'
                    noise_label = 'Noisy' if noisy else 'Clean'

                    for config in ['Default', 'Tuned']:
                        config_subset = subset[subset['configuration'] ==
        config]
                        for model in sorted(df_reg['model'].unique()):
                            model_subset =
        config_subset[config_subset['model'] == model]
                            if len(model_subset) > 0:
                                row = {
                                    'Obs_Size': obs.capitalize(),
                                    'Linearity': lin_label,
                                    'Noise': noise_label,
                                    'Config': config,
                                    'Model': model,
                                    'N': len(model_subset)
                                }
                                for metric in ['NRMSE', 'nrmse', 'NMAE',
        'nmae', 'NMSE', 'nmse', 'r2']:
                                    if metric in model_subset.columns:
                                        row[f'{metric}_mean'] =
        model_subset[metric].mean()
                                        row[f'{metric}_std'] =
        model_subset[metric].std()
                                rows.append(row)

        if rows:
            consolidated_reg = pd.DataFrame(rows)
            consolidated_reg.to_csv(output_dir /
        'regression_consolidated.csv', index=False)
            print(f"Saved: regression_consolidated.csv
        ({len(consolidated_reg)} rows)")


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Generate summary tables for
         dissertation')
    parser.add_argument('--subset', type=str, default=None,
```

```
                    help='Filter datasets by substring (e.g.,
        "Noisy_True", "obs_large")')
    parser.add_argument('--all', action='store_true',
                    help='Auto-generate tables for ALL predefined
        subsets')
    args = parser.parse_args()

    if args.all:
        generate_all_subsets()
    else:
        generate_summary_tables(args.subset)
```

3.18: scripts/generate_summary_tables_senstest.py

Location: `scripts/generate_summary_tables_senstest.py`

Sensitivity test version that filters out extreme failures (NRMSE > 3).

```
"""
Generate summary tables for dissertation - SENSITIVITY TEST VERSION.

Filters out extreme failures (NRMSE > 3) before computing summaries.

Usage:
    python scripts/generate_summary_tables_senstest.py                    #
        Basic summary
    python scripts/generate_summary_tables_senstest.py --subset "Noisy_True"
        # Single subset
    python scripts/generate_summary_tables_senstest.py --all              #
        Auto-generate ALL subsets
    python scripts/generate_summary_tables_senstest.py --threshold 2.0    #
        Custom threshold
"""

import pandas as pd
import numpy as np
from pathlib import Path
import argparse


# Default threshold for filtering extreme failures
DEFAULT_NRMSE_THRESHOLD = 3.0

# Define all subset filters for auto-generation
SUBSET_FILTERS = {
    # By observation size
    'obs_small': 'smallx',
    'obs_medium': 'medx',
    'obs_large': 'largex',

    # By feature size
    'feat_small': 'xsmall',
    'feat_medium': 'xmed',
    'feat_large': 'xlarge',

    # By noise
```

```
    'noisy': 'Noisy_True',
    'clean': 'Noisy_False',

    # By balance (classification)
    'balanced': 'Bal_True',
    'imbalanced': 'Bal_False',

    # By linearity (regression)
    'linear': 'Nonlinear_False',
    'nonlinear': 'Nonlinear_True',

    # Combined - Large observation datasets
    'large_obs_clean': ['largex', 'Noisy_False'],
    'large_obs_noisy': ['largex', 'Noisy_True'],
    'large_obs_balanced': ['largex', 'Bal_True'],
    'large_obs_imbalanced': ['largex', 'Bal_False'],
    'large_obs_linear': ['largex', 'Nonlinear_False'],
    'large_obs_nonlinear': ['largex', 'Nonlinear_True'],

    # Combined - Medium observation datasets
    'medium_obs_clean': ['medx', 'Noisy_False'],
    'medium_obs_noisy': ['medx', 'Noisy_True'],

    # Combined - Small observation datasets
    'small_obs_clean': ['smallx', 'Noisy_False'],
    'small_obs_noisy': ['smallx', 'Noisy_True'],
}


def apply_filters(df, filters):
    """Apply one or more filters to dataframe."""
    if isinstance(filters, str):
        filters = [filters]

    mask = pd.Series([True] * len(df), index=df.index)
    for f in filters:
        mask &= df['dataset'].str.contains(f, case=False, na=False)

    return df[mask]


def filter_extreme_failures(df, threshold=DEFAULT_NRMSE_THRESHOLD):
    """Filter out extreme failures from regression results."""
    original_count = len(df)

    # Find NRMSE column
    nrmse_col = None
    for col in ['NRMSE', 'nrmse']:
        if col in df.columns:
            nrmse_col = col
            break

    if nrmse_col is None:
        print("  WARNING: NRMSE column not found, no filtering applied")
        return df, 0
```

```
    # Only filter regression results
    regression_mask = df['task'] == 'regression'
    extreme_mask = regression_mask & (df[nrmse_col] > threshold)

    removed_count = extreme_mask.sum()
    df_filtered = df[~extreme_mask].copy()

    if removed_count > 0:
        print(f"  SENSITIVITY FILTER: Removed {removed_count} rows with NRMSE
         > {threshold}")
        # Show which model-configs were affected
        affected = df[extreme_mask].groupby(['model',
         'configuration']).size()
        print(f"  Affected model-configs:")
        for (model, config), count in affected.items():
            print(f"    - {model} ({config}): {count} rows removed")

    return df_filtered, removed_count


def generate_summary_tables(subset_filter=None, df=None, output_suffix='',
         threshold=DEFAULT_NRMSE_THRESHOLD):
    """Generate summary statistics tables with sensitivity filtering."""

    if df is None:
        results_path = Path('results/final_results_normalized.csv')
        if not results_path.exists():
            results_path = Path('results/final_results.csv')
        df = pd.read_csv(results_path)

    # Apply sensitivity filter (remove extreme NRMSE values)
    df, removed_count = filter_extreme_failures(df, threshold)

    # Apply subset filter if specified
    filter_desc = None
    if subset_filter:
        if isinstance(subset_filter, list):
            df = apply_filters(df, subset_filter)
            filter_desc = ' & '.join(subset_filter)
        else:
            df = df[df['dataset'].str.contains(subset_filter)]
            filter_desc = subset_filter
        print(f"Filtered to subset: {filter_desc} ({len(df)} rows)")

    if len(df) == 0:
        print(f"  WARNING: No data after filtering!")
        return None, None

    # Create model+config label
    df['model_config'] = df['model'] + ' (' + df['configuration'] + ')'

    output_dir = Path('analysis_outputs/summary_tables_senstest')
    output_dir.mkdir(parents=True, exist_ok=True)
```

```
summary_reg = None
summary_class = None

# ===== REGRESSION SUMMARY =====
df_reg = df[df['task'] == 'regression'].copy()

if len(df_reg) > 0:
    # Determine available metrics
    agg_dict = {}
    for metric, col_options in [('NRMSE', ['NRMSE', 'nrmse']),
                                ('NMAE', ['NMAE', 'nmae']),
                                ('NMSE', ['NMSE', 'nmse']),
                                ('R2', ['r2', 'R2'])]:
        for col in col_options:
            if col in df_reg.columns:
                agg_dict[f'mean_{metric}'] = (col, 'mean')
                agg_dict[f'std_{metric}'] = (col, 'std')
                agg_dict[f'count_{metric}'] = (col, 'count')
                break

    if agg_dict:
        summary_reg = (
            df_reg
            .groupby(['model', 'configuration'], as_index=False)
            .agg(**agg_dict)
        )
        if 'mean_NRMSE' in summary_reg.columns:
            summary_reg = summary_reg.sort_values('mean_NRMSE')

        suffix = output_suffix if output_suffix else
     (f"_{subset_filter.replace(' ', '_')}" if isinstance(subset_filter,
     str) and subset_filter else "")
        summary_reg.to_csv(output_dir /
     f'regression_summary{suffix}.csv', index=False)

        print("\n" + "=" * 60)
        print(f"REGRESSION SUMMARY - SENSITIVITY TEST (NRMSE <=
     {threshold})")
        print(f"{' (' + filter_desc + ')' if subset_filter else ''}")
        print("=" * 60)
        print(summary_reg.to_string(index=False))

# ===== CLASSIFICATION SUMMARY =====
df_class = df[df['task'] == 'classification'].copy()

if len(df_class) > 0:
    agg_dict = {}
    for metric, col_options in [('F1_macro', ['f1_macro', 'f1']),
                                ('Accuracy', ['accuracy']),
                                ('ROC_AUC', ['roc_auc', 'auc'])]:
        for col in col_options:
            if col in df_class.columns:
                agg_dict[f'mean_{metric}'] = (col, 'mean')
                agg_dict[f'std_{metric}'] = (col, 'std')
                agg_dict[f'count_{metric}'] = (col, 'count')
                break
```

```
        if agg_dict:
            summary_class = (
                df_class
                .groupby(['model', 'configuration'], as_index=False)
                .agg(**agg_dict)
            )
            if 'mean_F1_macro' in summary_class.columns:
                summary_class = summary_class.sort_values('mean_F1_macro',
         ascending=False)

            suffix = output_suffix if output_suffix else
         (f"_{subset_filter.replace(' ', '_')}" if isinstance(subset_filter,
         str) and subset_filter else "")
            summary_class.to_csv(output_dir /
         f'classification_summary{suffix}.csv', index=False)

            print("\n" + "=" * 60)
            print(f"CLASSIFICATION SUMMARY - SENSITIVITY TEST")
            print(f"{' (' + filter_desc + ')' if subset_filter else ''}")
            print("=" * 60)
            print(summary_class.to_string(index=False))

    print(f"\nSummary tables saved to: {output_dir}/")

    return summary_reg, summary_class


def generate_all_subsets(threshold=DEFAULT_NRMSE_THRESHOLD):
    """Auto-generate summary tables for ALL predefined subsets with
         sensitivity filtering."""

    print("=" * 80)
    print(f"SENSITIVITY TEST: AUTO-GENERATING SUMMARY TABLES (NRMSE <=
         {threshold})")
    print("=" * 80)

    # Load data once
    results_path = Path('results/final_results_normalized.csv')
    if not results_path.exists():
        results_path = Path('results/final_results.csv')
    df_original = pd.read_csv(results_path)
    print(f"Loaded {len(df_original)} rows from {results_path}")

    # Apply sensitivity filter once
    df, total_removed = filter_extreme_failures(df_original.copy(),
         threshold)
    print(f"\nAfter sensitivity filtering: {len(df)} rows remaining")

    output_dir = Path('analysis_outputs/summary_tables_senstest')
    output_dir.mkdir(parents=True, exist_ok=True)

    # Save filtering summary
    filtering_summary = {
        'threshold': threshold,
        'original_rows': len(df_original),
```

```
        'filtered_rows': len(df),
        'removed_rows': total_removed
    }
    pd.DataFrame([filtering_summary]).to_csv(output_dir /
         'filtering_summary.csv', index=False)

    # Generate overall summary first
    print("\n" + "=" * 80)
    print("OVERALL SUMMARY (no subset filter)")
    print("=" * 80)
    generate_summary_tables(subset_filter=None, df=df.copy(),
         output_suffix='_overall', threshold=threshold)

    # Generate for each subset
    for name, filters in SUBSET_FILTERS.items():
        print("\n" + "=" * 80)
        print(f"SUBSET: {name}")
        print("=" * 80)
        generate_summary_tables(subset_filter=filters, df=df.copy(),
         output_suffix=f'_{name}', threshold=threshold)

    # Generate consolidated report
    generate_consolidated_report(df, threshold)

    print("\n" + "=" * 80)
    print("ALL SENSITIVITY TEST SUMMARY TABLES GENERATED!")
    print("=" * 80)
    print(f"\nOutput directory: {output_dir}/")
    print(f"Threshold used: NRMSE <= {threshold}")


def generate_consolidated_report(df, threshold):
    """Generate a single consolidated report with all breakdowns."""

    output_dir = Path('analysis_outputs/summary_tables_senstest')
    output_dir.mkdir(parents=True, exist_ok=True)

    # Parse properties from dataset names
    df = df.copy()
    df['obs_size'] = df['dimensionality'].apply(lambda x: x.split('x')[0] if
         'x' in str(x) else 'unknown')
    df['feat_size'] = df['dimensionality'].apply(lambda x: x.split('x')[1] if
         'x' in str(x) else 'unknown')
    df['is_noisy'] = df['dataset'].str.contains('Noisy_True', case=False,
         na=False)
    df['is_balanced'] = df['dataset'].str.contains('Bal_True', case=False,
         na=False)
    df['is_nonlinear'] = df['dataset'].str.contains('Nonlinear_True',
         case=False, na=False)

    # ===== CLASSIFICATION CONSOLIDATED =====
    df_class = df[df['task'] == 'classification'].copy()

    if len(df_class) > 0:
        print("\n" + "=" * 80)
        print("CONSOLIDATED CLASSIFICATION REPORT (SENSITIVITY TEST)")
```

```
    print("=" * 80)

    rows = []
    for obs in ['small', 'med', 'large']:
        for balanced in [True, False]:
            for noisy in [True, False]:
                subset = df_class[
                    (df_class['obs_size'] == obs) &
                    (df_class['is_balanced'] == balanced) &
                    (df_class['is_noisy'] == noisy)
                ]

                bal_label = 'Balanced' if balanced else 'Imbalanced'
                noise_label = 'Noisy' if noisy else 'Clean'

                for config in ['Default', 'Tuned']:
                    config_subset = subset[subset['configuration'] ==
 config]
                    for model in sorted(df_class['model'].unique()):
                        model_subset =
 config_subset[config_subset['model'] == model]
                        if len(model_subset) > 0:
                            row = {
                                'Obs_Size': obs.capitalize(),
                                'Balance': bal_label,
                                'Noise': noise_label,
                                'Config': config,
                                'Model': model,
                                'N': len(model_subset)
                            }
                            for metric in ['f1_macro', 'accuracy',
 'roc_auc']:
                                if metric in model_subset.columns:
                                    row[f'{metric}_mean'] =
 model_subset[metric].mean()
                                    row[f'{metric}_std'] =
 model_subset[metric].std()
                            rows.append(row)

    if rows:
        consolidated_class = pd.DataFrame(rows)
        consolidated_class.to_csv(output_dir /
 'classification_consolidated.csv', index=False)
        print(f"Saved: classification_consolidated.csv
 ({len(consolidated_class)} rows)")

# ===== REGRESSION CONSOLIDATED =====
df_reg = df[df['task'] == 'regression'].copy()

if len(df_reg) > 0:
    print("\n" + "=" * 80)
    print(f"CONSOLIDATED REGRESSION REPORT (SENSITIVITY TEST, NRMSE <=
 {threshold})")
    print("=" * 80)

    rows = []
```

```
        for obs in ['small', 'med', 'large']:
            for nonlinear in [True, False]:
                for noisy in [True, False]:
                    subset = df_reg[
                        (df_reg['obs_size'] == obs) &
                        (df_reg['is_nonlinear'] == nonlinear) &
                        (df_reg['is_noisy'] == noisy)
                    ]

                    lin_label = 'Non-linear' if nonlinear else 'Linear'
                    noise_label = 'Noisy' if noisy else 'Clean'

                    for config in ['Default', 'Tuned']:
                        config_subset = subset[subset['configuration'] ==
    config]
                        for model in sorted(df_reg['model'].unique()):
                            model_subset =
    config_subset[config_subset['model'] == model]
                            if len(model_subset) > 0:
                                row = {
                                    'Obs_Size': obs.capitalize(),
                                    'Linearity': lin_label,
                                    'Noise': noise_label,
                                    'Config': config,
                                    'Model': model,
                                    'N': len(model_subset)
                                }
                                for metric in ['NRMSE', 'nrmse', 'NMAE',
    'nmae', 'NMSE', 'nmse', 'r2']:
                                    if metric in model_subset.columns:
                                        row[f'{metric}_mean'] =
    model_subset[metric].mean()
                                        row[f'{metric}_std'] =
    model_subset[metric].std()
                                rows.append(row)

        if rows:
            consolidated_reg = pd.DataFrame(rows)
            consolidated_reg.to_csv(output_dir /
    'regression_consolidated.csv', index=False)
            print(f"Saved: regression_consolidated.csv
    ({len(consolidated_reg)} rows)")


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Generate summary tables -
        SENSITIVITY TEST (filters NRMSE > threshold)')
    parser.add_argument('--subset', type=str, default=None,
                        help='Filter datasets by substring (e.g.,
        "Noisy_True", "obs_large")')
    parser.add_argument('--all', action='store_true',
                        help='Auto-generate tables for ALL predefined
        subsets')
    parser.add_argument('--threshold', type=float,
        default=DEFAULT_NRMSE_THRESHOLD,
                        help=f'NRMSE threshold for filtering extreme failures
        (default: {DEFAULT_NRMSE_THRESHOLD})')
```

```
    args = parser.parse_args()

    if args.all:
        generate_all_subsets(args.threshold)
    else:
        generate_summary_tables(args.subset, threshold=args.threshold)
```

3.19: scripts/compute_rankings_senstest.py

Location: `scripts/compute_rankings_senstest.py`

Sensitivity test version that filters out extreme failures (NRMSE > 3).

```
"""
Compute model rankings and perform Friedman test - SENSITIVITY TEST VERSION.

Filters out extreme failures (NRMSE > 3) before computing rankings.

Usage:
    python scripts/compute_rankings_senstest.py --task regression --metric
        NRMSE
    python scripts/compute_rankings_senstest.py --task classification --
        metric f1_macro
    python scripts/compute_rankings_senstest.py --all              # Auto-
        generate ALL rankings
    python scripts/compute_rankings_senstest.py --threshold 2.0    # Custom
        threshold
"""

import pandas as pd
import numpy as np
from scipy.stats import friedmanchisquare
from pathlib import Path
import argparse


# Default threshold for filtering extreme failures
DEFAULT_NRMSE_THRESHOLD = 3.0

# Define all subset filters for auto-generation
SUBSET_FILTERS = {
    # By observation size
    'obs_small': 'smallx',
    'obs_medium': 'medx',
    'obs_large': 'largex',

    # By feature size
    'feat_small': 'xsmall',
    'feat_medium': 'xmed',
    'feat_large': 'xlarge',

    # By noise
    'noisy': 'Noisy_True',
    'clean': 'Noisy_False',
```

```
    # By balance (classification only)
    'balanced': 'Bal_True',
    'imbalanced': 'Bal_False',

    # By linearity (regression only)
    'linear': 'Nonlinear_False',
    'nonlinear': 'Nonlinear_True',

    # Combined - Large observation datasets
    'large_obs_clean': ['largex', 'Noisy_False'],
    'large_obs_noisy': ['largex', 'Noisy_True'],
    'large_obs_balanced': ['largex', 'Bal_True'],
    'large_obs_imbalanced': ['largex', 'Bal_False'],
    'large_obs_linear': ['largex', 'Nonlinear_False'],
    'large_obs_nonlinear': ['largex', 'Nonlinear_True'],

    # Combined - Medium observation datasets
    'medium_obs_clean': ['medx', 'Noisy_False'],
    'medium_obs_noisy': ['medx', 'Noisy_True'],

    # Combined - Small observation datasets
    'small_obs_clean': ['smallx', 'Noisy_False'],
    'small_obs_noisy': ['smallx', 'Noisy_True'],
}

# Default metrics per task
DEFAULT_METRICS = {
    'regression': ['NRMSE', 'NMAE', 'NMSE'],
    'classification': ['f1_macro', 'accuracy', 'roc_auc']
}


def apply_filters(df, filters):
    """Apply one or more filters to dataframe."""
    if isinstance(filters, str):
        filters = [filters]

    mask = pd.Series([True] * len(df), index=df.index)
    for f in filters:
        mask &= df['dataset'].str.contains(f, case=False, na=False)

    return df[mask]


def filter_extreme_failures(df, threshold=DEFAULT_NRMSE_THRESHOLD):
    """Filter out extreme failures from regression results."""
    original_count = len(df)

    # Find NRMSE column
    nrmse_col = None
    for col in ['NRMSE', 'nrmse']:
        if col in df.columns:
            nrmse_col = col
            break
```

```
    if nrmse_col is None:
        print("  WARNING: NRMSE column not found, no filtering applied")
        return df, 0

    # Only filter regression results
    regression_mask = df['task'] == 'regression'
    extreme_mask = regression_mask & (df[nrmse_col] > threshold)

    removed_count = extreme_mask.sum()
    df_filtered = df[~extreme_mask].copy()

    if removed_count > 0:
        print(f"  SENSITIVITY FILTER: Removed {removed_count} rows with NRMSE
         > {threshold}")
        # Show which model-configs were affected
        affected = df[extreme_mask].groupby(['model',
         'configuration']).size()
        print(f"  Affected model-configs:")
        for (model, config), count in affected.items():
            print(f"    - {model} ({config}): {count} rows removed")

    return df_filtered, removed_count


def compute_rankings(task='regression', metric='NRMSE', subset_filter=None,
         df=None,
                     output_suffix='', threshold=DEFAULT_NRMSE_THRESHOLD):
    """Compute model rankings and Friedman test with sensitivity
         filtering."""

    if df is None:
        results_path = Path('results/final_results_normalized.csv')
        if not results_path.exists():
            results_path = Path('results/final_results.csv')
        df = pd.read_csv(results_path)

    # Apply sensitivity filter (remove extreme NRMSE values)
    df, removed_count = filter_extreme_failures(df, threshold)

    # Filter by task
    df = df[df['task'] == task].copy()

    # Apply subset filter if specified
    filter_desc = None
    if subset_filter:
        if isinstance(subset_filter, list):
            df = apply_filters(df, subset_filter)
            filter_desc = ' & '.join(subset_filter)
        else:
            df = df[df['dataset'].str.contains(subset_filter)]
            filter_desc = subset_filter

    if len(df) == 0:
        print(f"  WARNING: No data for {task} with filter {subset_filter}")
        return None, None, None, None
```

```python
# Check if metric exists
if metric not in df.columns:
    # Try alternative column names
    alt_metrics = {'NRMSE': 'nrmse', 'NMAE': 'nmae', 'NMSE': 'nmse',
                   'f1_macro': 'f1', 'roc_auc': 'auc'}
    if metric in alt_metrics and alt_metrics[metric] in df.columns:
        metric = alt_metrics[metric]
    else:
        print(f"  WARNING: Metric {metric} not found in data")
        return None, None, None, None

print(f"\nTask: {task}")
print(f"Metric: {metric}")
print(f"Datasets: {df['dataset'].nunique()}")
print(f"Subset filter: {filter_desc or 'None'}")
print(f"Sensitivity threshold: NRMSE <= {threshold}")

# Create model+config label
df['model_config'] = df['model'] + ' (' + df['configuration'] + ')'

# For regression metrics (NRMSE, NMAE, NMSE), lower is better
# For classification metrics (f1_macro, accuracy, roc_auc), higher is
    better
ascending = metric.lower() in ['nrmse', 'nmae', 'nmse', 'rmse', 'mae',
    'mse']

# Rank each dataset separately
df['rank'] = df.groupby('dataset')[metric].rank(method='average',
    ascending=ascending)

# Average ranks per model
avg_ranks = df.groupby(['model', 'configuration'])
    ['rank'].mean().sort_values()

print("\n" + "=" * 60)
print(f"AVERAGE RANKS (by {metric}) - SENSITIVITY TEST")
print("=" * 60)
for (model, config), rank in avg_ranks.items():
    print(f"  {model} ({config}): {rank:.3f}")

# Pivot for Friedman test
pivot = df.pivot_table(
    index='dataset',
    columns='model_config',
    values='rank'
)

# Friedman test (requires at least 3 groups and 2 samples)
friedman_stat, friedman_p = None, None
if pivot.shape[1] >= 3 and pivot.shape[0] >= 2:
    try:
        friedman_stat, friedman_p = friedmanchisquare(*pivot.T.values)

        print("\n" + "=" * 60)
        print("FRIEDMAN TEST - SENSITIVITY TEST")
        print("=" * 60)
```

```
            print(f"  Chi-square statistic: {friedman_stat:.4f}")
            print(f"  p-value: {friedman_p:.6f}")

            if friedman_p < 0.05:
                print("  -> Significant difference between models (p <
     0.05)")
                print("  -> Proceed to post-hoc Nemenyi test")
            else:
                print("  -> No significant difference between models (p >=
     0.05)")
        except Exception as e:
            print(f"  WARNING: Friedman test failed: {e}")
    else:
        print(f"  WARNING: Not enough data for Friedman test (need >= 3
     groups, >= 2 datasets)")

    # Save rankings
    output_dir = Path('analysis_outputs/rankings_senstest')
    output_dir.mkdir(parents=True, exist_ok=True)

    suffix = output_suffix if output_suffix else (f"_{subset_filter.replace('
     ', '_')}" if isinstance(subset_filter, str) and subset_filter else
     "")
    avg_ranks.to_csv(output_dir / f'{task}_rankings_{metric}{suffix}.csv')

    # Save Friedman results
    if friedman_stat is not None:
        friedman_results = pd.DataFrame([{
            'task': task,
            'metric': metric,
            'subset': filter_desc or 'all',
            'threshold': threshold,
            'n_datasets': df['dataset'].nunique(),
            'n_models': pivot.shape[1],
            'chi_square': friedman_stat,
            'p_value': friedman_p,
            'significant': friedman_p < 0.05
        }])
        friedman_results.to_csv(output_dir / f'{task}_friedman_{metric}
     {suffix}.csv', index=False)

    return avg_ranks, friedman_stat, friedman_p, pivot


def generate_all_rankings(threshold=DEFAULT_NRMSE_THRESHOLD):
    """Auto-generate rankings for ALL predefined subsets and metrics with
     sensitivity filtering."""

    print("=" * 80)
    print(f"SENSITIVITY TEST: AUTO-GENERATING RANKINGS (NRMSE <=
     {threshold})")
    print("=" * 80)

    # Load data once
    results_path = Path('results/final_results_normalized.csv')
    if not results_path.exists():
```

```
    results_path = Path('results/final_results.csv')
df_original = pd.read_csv(results_path)
print(f"Loaded {len(df_original)} rows from {results_path}")

# Apply sensitivity filter once
df, total_removed = filter_extreme_failures(df_original.copy(),
     threshold)
print(f"\nAfter sensitivity filtering: {len(df)} rows remaining")

output_dir = Path('analysis_outputs/rankings_senstest')
output_dir.mkdir(parents=True, exist_ok=True)

# Save filtering summary
filtering_summary = {
    'threshold': threshold,
    'original_rows': len(df_original),
    'filtered_rows': len(df),
    'removed_rows': total_removed
}
pd.DataFrame([filtering_summary]).to_csv(output_dir /
     'filtering_summary.csv', index=False)

all_results = []

# Process each task
for task in ['classification', 'regression']:
    task_df = df[df['task'] == task].copy()
    if len(task_df) == 0:
        continue

    # Determine available metrics
    if task == 'classification':
        metrics = [m for m in ['f1_macro', 'f1', 'accuracy', 'roc_auc',
     'auc']
                   if m in task_df.columns]
    else:
        metrics = [m for m in ['NRMSE', 'nrmse', 'NMAE', 'nmae', 'NMSE',
     'nmse', 'r2']
                   if m in task_df.columns]

    print(f"\n{'='*80}")
    print(f"TASK: {task.upper()}")
    print(f"Available metrics: {metrics}")
    print(f"{'='*80}")

    # Overall rankings (no filter)
    for metric in metrics:
        print(f"\n--- Overall ({metric}) ---")
        avg_ranks, stat, p, _ = compute_rankings(
            task=task, metric=metric, subset_filter=None,
            df=task_df.copy(), output_suffix='_overall',
     threshold=threshold
        )
        if stat is not None:
            all_results.append({
                'task': task, 'metric': metric, 'subset': 'overall',
```

```
                    'threshold': threshold,
                    'chi_square': stat, 'p_value': p, 'significant': p < 0.05
                })

        # Subset rankings
        for subset_name, subset_filter in SUBSET_FILTERS.items():
            # Skip classification-only filters for regression and vice versa
            if task == 'regression' and subset_name in ['balanced',
         'imbalanced']:
                continue
            if task == 'classification' and subset_name in ['linear',
         'nonlinear']:
                continue

            for metric in metrics[:2]:  # Limit to top 2 metrics per subset
                print(f"\n--- {subset_name} ({metric}) ---")
                avg_ranks, stat, p, _ = compute_rankings(
                    task=task, metric=metric, subset_filter=subset_filter,
                    df=task_df.copy(), output_suffix=f'_{subset_name}',
         threshold=threshold
                )
                if stat is not None:
                    all_results.append({
                        'task': task, 'metric': metric, 'subset':
         subset_name,
                        'threshold': threshold,
                        'chi_square': stat, 'p_value': p, 'significant': p <
         0.05
                    })

    # Save consolidated Friedman results
    if all_results:
        consolidated = pd.DataFrame(all_results)
        consolidated.to_csv(output_dir / 'friedman_all_results.csv',
         index=False)
        print(f"\n\nConsolidated Friedman results saved to:
         {output_dir}/friedman_all_results.csv")

        # Print summary
        print("\n" + "=" * 80)
        print(f"FRIEDMAN TEST SUMMARY - SENSITIVITY TEST (NRMSE <=
         {threshold})")
        print("=" * 80)
        print(consolidated.to_string(index=False))

    print("\n" + "=" * 80)
    print("ALL SENSITIVITY TEST RANKINGS GENERATED!")
    print("=" * 80)
    print(f"\nOutput directory: {output_dir}/")
    print(f"Threshold used: NRMSE <= {threshold}")


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Compute model rankings -
         SENSITIVITY TEST (filters NRMSE > threshold)')
    parser.add_argument('--task', choices=['regression', 'classification'],
         default='regression')
```

```
    parser.add_argument('--metric', type=str, default='NRMSE')
    parser.add_argument('--subset', type=str, default=None)
    parser.add_argument('--threshold', type=float,
         default=DEFAULT_NRMSE_THRESHOLD,
                        help=f'NRMSE threshold for filtering extreme failures
         (default: {DEFAULT_NRMSE_THRESHOLD})')
    parser.add_argument('--all', action='store_true',
                        help='Auto-generate rankings for ALL subsets and
         metrics')
    args = parser.parse_args()

    if args.all:
        generate_all_rankings(args.threshold)
    else:
        compute_rankings(args.task, args.metric, args.subset,
         threshold=args.threshold)
```

3.20: scripts/wilcoxon_tuning_efficiency.py

Location: `scripts/wilcoxon_tuning_efficiency.py`

Wilcoxon signed-rank test comparing Default vs Tuned configurations.

```
"""
Wilcoxon Signed-Rank Test for Tuning Efficiency Analysis.

Compares Default vs Tuned configurations to determine if tuning provides
statistically significant improvement.

Delta_NRMSE = Default - Tuned (positive values indicate improvement)
Positive outcome = Tuned model performs significantly better

Usage:
    python scripts/wilcoxon_tuning_efficiency.py                          #
         All models
    python scripts/wilcoxon_tuning_efficiency.py --model "XGBoost"        #
         Single model
    python scripts/wilcoxon_tuning_efficiency.py --task classification    #
         Classification only
    python scripts/wilcoxon_tuning_efficiency.py --all                    #
         All subsets
    python scripts/wilcoxon_tuning_efficiency.py --threshold 3.0          #
         Filter NRMSE > 3
    python scripts/wilcoxon_tuning_efficiency.py --all --threshold 3.0    #
         Filter NRMSE > 3 for ALL data complexities and Overall
"""

import pandas as pd
import numpy as np
from scipy.stats import wilcoxon
from pathlib import Path
import argparse


# Default threshold for filtering extreme failures (set to None for no
         filtering)
```

```python
DEFAULT_NRMSE_THRESHOLD = None

# Define all subset filters for auto-generation
SUBSET_FILTERS = {
    # By observation size
    'obs_small': 'smallx',
    'obs_medium': 'medx',
    'obs_large': 'largex',

    # By feature size
    'feat_small': 'xsmall',
    'feat_medium': 'xmed',
    'feat_large': 'xlarge',

    # By noise
    'noisy': 'Noisy_True',
    'clean': 'Noisy_False',

    # By balance (classification only)
    'balanced': 'Bal_True',
    'imbalanced': 'Bal_False',

    # By linearity (regression only)
    'linear': 'Nonlinear_False',
    'nonlinear': 'Nonlinear_True',

    # Combined
    'large_obs_clean': ['largex', 'Noisy_False'],
    'large_obs_noisy': ['largex', 'Noisy_True'],
    'small_obs_clean': ['smallx', 'Noisy_False'],
    'small_obs_noisy': ['smallx', 'Noisy_True'],
}


def apply_filters(df, filters):
    """Apply one or more filters to dataframe."""
    if isinstance(filters, str):
        filters = [filters]

    mask = pd.Series([True] * len(df), index=df.index)
    for f in filters:
        mask &= df['dataset'].str.contains(f, case=False, na=False)

    return df[mask]


def filter_extreme_failures(df, threshold):
    """Filter out extreme failures from regression results."""
    if threshold is None:
        return df, 0

    original_count = len(df)

    # Find NRMSE column
    nrmse_col = None
```

```
    for col in ['NRMSE', 'nrmse']:
        if col in df.columns:
            nrmse_col = col
            break

    if nrmse_col is None:
        return df, 0

    # Only filter regression results
    regression_mask = df['task'] == 'regression'
    extreme_mask = regression_mask & (df[nrmse_col] > threshold)

    removed_count = extreme_mask.sum()
    df_filtered = df[~extreme_mask].copy()

    if removed_count > 0:
        print(f"  FILTER: Removed {removed_count} rows with NRMSE >
         {threshold}")

    return df_filtered, removed_count


def wilcoxon_tuning_test(task='regression', model=None, subset_filter=None,
         df=None,
                         output_suffix='', threshold=DEFAULT_NRMSE_THRESHOLD,
         alpha=0.05):
    """Perform Wilcoxon signed-rank test comparing Default vs Tuned
         configurations."""

    if df is None:
        results_path = Path('results/final_results_normalized.csv')
        if not results_path.exists():
            results_path = Path('results/final_results.csv')
        df = pd.read_csv(results_path)

    # Apply sensitivity filter if specified
    if threshold is not None:
        df, _ = filter_extreme_failures(df, threshold)

    # Filter by task
    df = df[df['task'] == task].copy()

    # Apply subset filter if specified
    filter_desc = None
    if subset_filter:
        if isinstance(subset_filter, list):
            df = apply_filters(df, subset_filter)
            filter_desc = ' & '.join(subset_filter)
        else:
            df = df[df['dataset'].str.contains(subset_filter)]
            filter_desc = subset_filter

    if len(df) == 0:
        print(f"  WARNING: No data for {task} with filter {subset_filter}")
        return None
```

```
# Determine metric based on task
if task == 'regression':
    # Find NRMSE column
    metric_col = None
    for col in ['NRMSE', 'nrmse']:
        if col in df.columns:
            metric_col = col
            break
    if metric_col is None:
        print("  WARNING: NRMSE column not found")
        return None
    metric_name = 'NRMSE'
    # For NRMSE: lower is better, so Delta_ = Default - Tuned (positive =
     improvement)
    improvement_direction = 'lower_better'
else:
    # Find F1 column
    metric_col = None
    for col in ['f1_macro', 'f1']:
        if col in df.columns:
            metric_col = col
            break
    if metric_col is None:
        print("  WARNING: F1 column not found")
        return None
    metric_name = 'F1_macro'
    # For F1: higher is better, so Delta_ = Tuned - Default (positive =
     improvement)
    improvement_direction = 'higher_better'

# Get models to test
if model:
    models = [model]
else:
    models = sorted(df['model'].unique())

print("=" * 80)
print(f"WILCOXON SIGNED-RANK TEST: TUNING EFFICIENCY")
print(f"Task: {task} | Metric: {metric_name} | Alpha: {alpha}")
print(f"Subset: {filter_desc or 'All'}")
if threshold:
    print(f"Sensitivity filter: NRMSE <= {threshold}")
print("=" * 80)

results = []

for model_name in models:
    model_df = df[df['model'] == model_name].copy()

    # Get Default and Tuned results
    default_df = model_df[model_df['configuration'] == 'Default'].copy()
    tuned_df = model_df[model_df['configuration'] == 'Tuned'].copy()

    # Find common datasets
```

```
common_datasets = set(default_df['dataset']) &
 set(tuned_df['dataset'])

if len(common_datasets) < 2:
    print(f"\n{model_name}: Insufficient paired data (n=
 {len(common_datasets)})")
    results.append({
        'model': model_name,
        'n_datasets': len(common_datasets),
        'metric': metric_name,
        'mean_default': np.nan,
        'mean_tuned': np.nan,
        'mean_delta': np.nan,
        'median_delta': np.nan,
        'positive_improvements': np.nan,
        'negative_improvements': np.nan,
        'statistic': np.nan,
        'p_value': np.nan,
        'significant': False,
        'conclusion': 'Insufficient data'
    })
    continue

# Align data by dataset
default_df =
 default_df[default_df['dataset'].isin(common_datasets)].set_index('dataset')
tuned_df =
 tuned_df[tuned_df['dataset'].isin(common_datasets)].set_index('dataset')

# Ensure same order
common_datasets = sorted(common_datasets)
default_values = default_df.loc[common_datasets, metric_col].values
tuned_values = tuned_df.loc[common_datasets, metric_col].values

# Calculate delta based on metric direction
if improvement_direction == 'lower_better':
    # Delta_NRMSE = Default - Tuned (positive = tuned is better)
    delta = default_values - tuned_values
else:
    # Delta_F1 = Tuned - Default (positive = tuned is better)
    delta = tuned_values - default_values

# Count improvements
n_positive = np.sum(delta > 0)  # Tuned better
n_negative = np.sum(delta < 0)  # Default better
n_zero = np.sum(delta == 0)     # No change

# Perform Wilcoxon signed-rank test
# alternative='greater' tests if tuned is significantly better
try:
    # Remove zeros for Wilcoxon test
    delta_nonzero = delta[delta != 0]
    if len(delta_nonzero) < 2:
        statistic, p_value = np.nan, np.nan
        conclusion = 'All differences are zero'
    else:
```

```
            statistic, p_value = wilcoxon(delta_nonzero,
 alternative='greater')

            if p_value < alpha:
                conclusion = f'Tuned SIGNIFICANTLY BETTER (p=
 {p_value:.4f})'
            else:
                conclusion = f'No significant difference (p=
 {p_value:.4f})'
        except Exception as e:
            statistic, p_value = np.nan, np.nan
            conclusion = f'Test failed: {str(e)}'

        # Store results
        result = {
            'model': model_name,
            'n_datasets': len(common_datasets),
            'metric': metric_name,
            'mean_default': np.mean(default_values),
            'mean_tuned': np.mean(tuned_values),
            'mean_delta': np.mean(delta),
            'median_delta': np.median(delta),
            'std_delta': np.std(delta),
            'positive_improvements': n_positive,
            'negative_improvements': n_negative,
            'ties': n_zero,
            'pct_improved': 100 * n_positive / len(delta),
            'statistic': statistic,
            'p_value': p_value,
            'significant': p_value < alpha if not np.isnan(p_value) else
 False,
            'conclusion': conclusion
        }
        results.append(result)

        # Print results
        print(f"\n{'-'*70}")
        print(f"MODEL: {model_name}")
        print(f"{'-'*70}")
        print(f"  Paired datasets: {len(common_datasets)}")
        print(f"  Mean Default {metric_name}: {np.mean(default_values):.4f}")
        print(f"  Mean Tuned {metric_name}: {np.mean(tuned_values):.4f}")
        print(f"  Mean Delta_{metric_name}: {np.mean(delta):.4f} (positive =
 tuned better)")
        print(f"  Median Delta_{metric_name}: {np.median(delta):.4f}")
        print(f"  Improvements: {n_positive}/{len(delta)}
 ({100*n_positive/len(delta):.1f}%)")
        print(f"  Degradations: {n_negative}/{len(delta)}
 ({100*n_negative/len(delta):.1f}%)")
        if not np.isnan(statistic):
            print(f"  Wilcoxon statistic: {statistic:.2f}")
            print(f"  p-value (one-sided): {p_value:.6f}")
        print(f"  CONCLUSION: {conclusion}")

# Create results DataFrame
results_df = pd.DataFrame(results)
```

```
    # Save results
    output_dir = Path('analysis_outputs/wilcoxon_tuning')
    output_dir.mkdir(parents=True, exist_ok=True)

    suffix = output_suffix if output_suffix else ''
    if threshold:
        suffix += f'_thresh{threshold}'

    results_df.to_csv(output_dir / f'{task}_wilcoxon_tuning{suffix}.csv',
         index=False)

    # Print summary table
    print("\n" + "=" * 80)
    print("SUMMARY TABLE")
    print("=" * 80)
    summary_cols = ['model', 'n_datasets', 'mean_delta', 'pct_improved',
         'p_value', 'significant']
    print(results_df[summary_cols].to_string(index=False))

    print(f"\nResults saved to: {output_dir}/")

    return results_df


def generate_delta_table(task='regression', df=None,
         threshold=DEFAULT_NRMSE_THRESHOLD,
                         output_suffix=''):
    """Generate detailed delta table showing Default vs Tuned for each
         dataset."""

    if df is None:
        results_path = Path('results/final_results_normalized.csv')
        if not results_path.exists():
            results_path = Path('results/final_results.csv')
        df = pd.read_csv(results_path)

    # Apply sensitivity filter if specified
    if threshold is not None:
        df, _ = filter_extreme_failures(df, threshold)

    # Filter by task
    df = df[df['task'] == task].copy()

    if len(df) == 0:
        print(f"  WARNING: No data for {task}")
        return None

    # Determine metric
    if task == 'regression':
        metric_col = 'NRMSE' if 'NRMSE' in df.columns else 'nrmse'
        metric_name = 'NRMSE'
    else:
        metric_col = 'f1_macro' if 'f1_macro' in df.columns else 'f1'
        metric_name = 'F1_macro'
```

```
print("=" * 80)
print(f"DELTA TABLE: {task.upper()} - Delta_{metric_name} = Default -
     Tuned")
print("=" * 80)

# Pivot to get Default and Tuned side by side
rows = []
for model in sorted(df['model'].unique()):
    model_df = df[df['model'] == model]
    default_df = model_df[model_df['configuration'] ==
     'Default'].set_index('dataset')
    tuned_df = model_df[model_df['configuration'] ==
     'Tuned'].set_index('dataset')

    common = set(default_df.index) & set(tuned_df.index)
    for dataset in sorted(common):
        default_val = default_df.loc[dataset, metric_col]
        tuned_val = tuned_df.loc[dataset, metric_col]

        if task == 'regression':
            delta = default_val - tuned_val  # Positive = improvement
        else:
            delta = tuned_val - default_val  # Positive = improvement

        rows.append({
            'dataset': dataset,
            'model': model,
            f'{metric_name}_Default': default_val,
            f'{metric_name}_Tuned': tuned_val,
            f'Delta_{metric_name}': delta,
            'improved': delta > 0
        })

delta_df = pd.DataFrame(rows)

# Save
output_dir = Path('analysis_outputs/wilcoxon_tuning')
output_dir.mkdir(parents=True, exist_ok=True)

suffix = output_suffix if output_suffix else ''
if threshold:
    suffix += f'_thresh{threshold}'

delta_df.to_csv(output_dir / f'{task}_delta_table{suffix}.csv',
     index=False)

# Print summary by model
print("\nDelta Summary by Model:")
print("-" * 70)
summary = delta_df.groupby('model').agg({
    f'Delta_{metric_name}': ['mean', 'median', 'std'],
    'improved': ['sum', 'count']
})
summary.columns = ['mean_delta', 'median_delta', 'std_delta',
     'n_improved', 'n_total']
```

```
    summary['pct_improved'] = 100 * summary['n_improved'] /
         summary['n_total']
    print(summary.round(4).to_string())

    print(f"\nDelta table saved to:
         {output_dir}/{task}_delta_table{suffix}.csv")

    return delta_df


def generate_all_wilcoxon(threshold=DEFAULT_NRMSE_THRESHOLD):
    """Auto-generate Wilcoxon tests for ALL tasks, models, and subsets."""

    print("=" * 80)
    print("AUTO-GENERATING WILCOXON TUNING EFFICIENCY TESTS")
    if threshold:
        print(f"Sensitivity filter: NRMSE <= {threshold}")
    print("=" * 80)

    # Load data once
    results_path = Path('results/final_results_normalized.csv')
    if not results_path.exists():
        results_path = Path('results/final_results.csv')
    df = pd.read_csv(results_path)
    print(f"Loaded {len(df)} rows from {results_path}")

    output_dir = Path('analysis_outputs/wilcoxon_tuning')
    output_dir.mkdir(parents=True, exist_ok=True)

    all_results = []

    # Process each task
    for task in ['regression', 'classification']:
        print(f"\n{'='*80}")
        print(f"TASK: {task.upper()}")
        print(f"{'='*80}")

        # Overall test (no filter)
        print("\n--- Overall ---")
        results_df = wilcoxon_tuning_test(
            task=task, df=df.copy(), output_suffix='_overall',
         threshold=threshold
        )
        if results_df is not None:
            results_df['subset'] = 'overall'
            all_results.append(results_df)

        # Generate delta table
        generate_delta_table(task=task, df=df.copy(), threshold=threshold,
         output_suffix='_overall')

        # Subset tests
        for subset_name, subset_filter in SUBSET_FILTERS.items():
            # Skip classification-only filters for regression and vice versa
            if task == 'regression' and subset_name in ['balanced',
         'imbalanced']:
```

```
                continue
            if task == 'classification' and subset_name in ['linear',
        'nonlinear']:
                continue

            print(f"\n--- {subset_name} ---")
            results_df = wilcoxon_tuning_test(
                task=task, subset_filter=subset_filter, df=df.copy(),
                output_suffix=f'_{subset_name}', threshold=threshold
            )
            if results_df is not None:
                results_df['subset'] = subset_name
                all_results.append(results_df)

    # Consolidate all results
    if all_results:
        consolidated = pd.concat(all_results, ignore_index=True)
        suffix = f'_thresh{threshold}' if threshold else ''
        consolidated.to_csv(output_dir / f'wilcoxon_all_results{suffix}.csv',
         index=False)

        # Print final summary
        print("\n" + "=" * 80)
        print("CONSOLIDATED WILCOXON TEST SUMMARY")
        print("=" * 80)

        # Summary: significant improvements by task and model
        sig_summary = consolidated[consolidated['subset'] == 'overall'][
            ['model', 'metric', 'mean_delta', 'pct_improved', 'p_value',
         'significant']
        ].copy()
        sig_summary = sig_summary.sort_values(['metric', 'p_value'])
        print("\nOverall Results (all datasets):")
        print(sig_summary.to_string(index=False))

        print(f"\nAll results saved to: {output_dir}/")

    print("\n" + "=" * 80)
    print("ALL WILCOXON TESTS GENERATED!")
    print("=" * 80)


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='Wilcoxon signed-rank test for tuning efficiency
         analysis'
    )
    parser.add_argument('--task', choices=['regression', 'classification'],
                        default='regression', help='Task type')
    parser.add_argument('--model', type=str, default=None,
                        help='Specific model to test (default: all models)')
    parser.add_argument('--subset', type=str, default=None,
                        help='Filter datasets by substring')
    parser.add_argument('--threshold', type=float,
         default=DEFAULT_NRMSE_THRESHOLD,
```

```
                    help='NRMSE threshold for filtering extreme failures
        (default: no filtering)')
    parser.add_argument('--alpha', type=float, default=0.05,
                        help='Significance level (default: 0.05)')
    parser.add_argument('--all', action='store_true',
                        help='Auto-generate tests for ALL tasks and subsets')
    parser.add_argument('--delta-table', action='store_true',
                        help='Generate detailed delta table')
    args = parser.parse_args()

    if args.all:
        generate_all_wilcoxon(args.threshold)
    elif args.delta_table:
        generate_delta_table(args.task, threshold=args.threshold)
    else:
        wilcoxon_tuning_test(
            task=args.task,
            model=args.model,
            subset_filter=args.subset,
            threshold=args.threshold,
            alpha=args.alpha
        )
```

Phase 4: Run Experiments

Step 4.1: Run All at Once

```
python batch_factorial_runner.py --tune > logs/tuned_all.log 2>&1
```

Expected Output

After completion, you should have these files in `results/`:

```
factorial_design_results_mixed_smallxsmall.csv
factorial_design_results_mixed_smallxmed.csv
factorial_design_results_mixed_smallxlarge.csv
factorial_design_results_mixed_medxsmall.csv
factorial_design_results_mixed_medxmed.csv
factorial_design_results_mixed_medxlarge.csv
factorial_design_results_mixed_largexsmall.csv
factorial_design_results_mixed_largexmed.csv
factorial_design_results_mixed_largexlarge.csv
```

Phase 5: Combine Results

Step 5.1: Run Combine Script

```
python scripts/combine_results.py
```

Step 5.2: Verify Output

```
# Should show 577 lines (576 data + 1 header)
wc -l results/final_results.csv
```

Expected Output

`results/final_results.csv` with 576 rows containing all experiment results.

Phase 6: Statistical Analysis

Step 6.1: Calculate Target Standard Deviation

```
python scripts/calculate_stdy.py
```

Output: `results/final_results_with_stdy.csv`

Step 6.2: Compute Normalized Metrics

```
python scripts/compute_normalized_metrics.py
```

Output: `results/final_results_normalized.csv` (adds NRMSE, NMAE, NMSE columns)

Step 6.3: Compute Rankings (Friedman Test)

```
python scripts/compute_rankings.py --task regression --metric NRMSE
python scripts/compute_rankings.py --task classification --metric f1_macro
```

Output: `analysis_outputs/rankings/`

Step 6.4: Post-Hoc Tests (Nemenyi)

```
python scripts/nemenyi_posthoc.py --task regression --metric NRMSE
python scripts/nemenyi_posthoc.py --task classification --metric f1_macro
```

Output: `analysis_outputs/posthoc/`

Step 6.5: Generate Summary Tables

```
# Generate summary tables for all subsets
python scripts/generate_summary_tables.py --all
```

Output: `analysis_outputs/summary_tables/`

This generates: - `regression_summary_*.csv` - Mean/std of NRMSE, NMAE, NMSE by model+config - `classification_summary_*.csv` - Mean/std of F1, Accuracy, ROC-AUC by model+config - `*_consolidated.csv` - Full breakdown by observation size, noise, balance/linearity

Step 6.6: Sensitivity Analysis (Optional)

Filter out extreme failures (NRMSE > 3) before computing statistics:

```
# Summary tables with sensitivity filtering
python scripts/generate_summary_tables_senstest.py --all

# Rankings with sensitivity filtering
python scripts/compute_rankings_senstest.py --all

# Custom threshold (e.g., NRMSE > 2)
python scripts/generate_summary_tables_senstest.py --all --threshold 2.0
```

Output: - `analysis_outputs/summary_tables_senstest/` - `analysis_outputs/rankings_senstest/`

Step 6.7: Wilcoxon Tuning Efficiency Test

Compare Default vs Tuned configurations using Wilcoxon signed-rank test:

```
# Run all Wilcoxon tests
python scripts/wilcoxon_tuning_efficiency.py --all

# With sensitivity filtering
python scripts/wilcoxon_tuning_efficiency.py --all --threshold 3.0

# Generate delta table (Delta_NRMSE = Default - Tuned for each dataset)
python scripts/wilcoxon_tuning_efficiency.py --delta-table
```

Output: `analysis_outputs/wilcoxon_tuning/`

This generates: - `regression_wilcoxon_tuning_*.csv` - Test results per model - `classification_wilcoxon_tuning_*.csv` - Test results per model - `*_delta_table_*.csv` - Per-dataset delta values - `wilcoxon_all_results.csv` - Consolidated results

Interpretation: - NRMSE = Default - Tuned (positive values indicate tuning improved performance) - p-value < 0.05 indicates statistically significant improvement from tuning

---

Validation

Check Experiment Count

```
import pandas as pd
df = pd.read_csv('results/final_results.csv')
print(f"Total experiments: {len(df)}")           # Expected: 576
print(f"Unique datasets: {df['dataset'].nunique()}")  # Expected: 72
print(f"Models: {sorted(df['model'].unique())}")
print(f"Configurations: {sorted(df['configuration'].unique())}")
```

Expected Output

```
Total experiments: 576
Unique datasets: 72
Models: ['Linear Regression', 'Logistic Regression', 'MLP', 'SVM', 'XGBoost']
Configurations: ['Default', 'Tuned']
```

Check Dimensionality Coverage

```
print(df['dimensionality'].value_counts().sort_index())
```

Each dimensionality should have exactly 64 rows.

Troubleshooting

“CUDA out of memory”

- Run one dimensionality at a time
- Reduce batch sizes in config.py

“Module not found”

- Ensure all `__init__.py` files exist
- Check you’re running from project root

Checkpointing

- The batch runner creates checkpoint files automatically
- Re-running the same command will skip completed datasets

File Checklist

Before running, verify you have all 20 Python files:

Core Files (2): - [ ] `config.py` - [ ] `batch_factorial_runner.py`

Data Module (4): - [ ] `data/__init__.py` - [ ] `data/preprocessing.py` - [ ] `data/dataset.py` - [ ] `data/dynamic_dataset.py`

Models Module (5): - [ ] `models/__init__.py` - [ ] `models/base_models.py` - [ ] `models/advanced_models.py` - [ ] `models/deep_learning.py` - [ ] `models/gpu_base_models.py`

Evaluation Module (3): - [ ] `evaluation/__init__.py` - [ ] `evaluation/metrics.py` - [ ] `evaluation/hyperparameter_tuning.py`

Scripts (9): - [ ] `scripts/combine_results.py` - [ ] `scripts/calculate_stdy.py` - [ ] `scripts/compute_normalized_metrics.py` - [ ] `scripts/compute_rankings.py` - [ ] `scripts/compute_rankings_senstest.py` - [ ] `scripts/nemenyi_posthoc.py` - [ ]

`scripts/generate_summary_tables.py` - [ ]
`scripts/generate_summary_tables_senstest.py` - [ ]
`scripts/wilcoxon_tuning_efficiency.py`

Datasets: - [ ] `data/processed_classification/` (36 CSV files across 9 folders) - [ ] `data/processed_regression/` (36 CSV files across 9 folders)

Document Version: 2025-12-08 **Random Seed**: 42 throughout all operations