**Mahmoud Ayman Kharoof Smart Bricks Automated Valuation Model (AVM) - Project Documentation**

**Project Overview**

The Automated Valuation Model (AVM) leverages AI and machine learning to estimate real estate property values in the Dubai market. This project involves data preprocessing, feature selection, base model implementation, meta-learner creation, and containerization with Docker. The workflow is designed for efficient, data-driven property valuation using advanced AI algorithms.

**Environment Setup**

**1. Install WSL**

* Ensure you have WSL2 with Ubuntu installed on your system.
* Update your WSL environment:

sudo apt update && sudo apt upgrade -y

**2. Install Python**

* Install Python 3.10 or higher:

sudo apt install python3 python3-venv python3-pip -y

**3. Install Docker**

* Install Docker CE in WSL:

sudo apt install docker-ce docker-ce-cli containerd.io -y

* Allow Docker to run without sudo:

sudo usermod -aG docker $USER

newgrp docker

**Folder Structure**

The project adheres to the following structure:

├── preprocess.py # Data preprocessing pipeline

├── feature\_selection.py # Feature selection logic

├── base\_models.py # Base model implementation

├── meta\_learner.py # Meta-learner implementation

├── main.py # Orchestrates the entire pipeline

├── evaluation.py # Model evaluation metrics

├── config.py # Configuration parameters

├── Dockerfile # Docker container setup

├── requirements.txt # Python dependencies

├── test.py # Unit tests for key components

├── data/

│ ├── sales.csv # Sales data

│ ├── rentals.csv # Rentals data

└── README.md # Documentation

**Data Preprocessing**

The preprocess.py script handles:

**Comprehensive Data Processing Strategy Analysis**

**1. Data Pipeline Overview**

The preprocessing pipeline handles two main types of real estate data:

Rental Property Data

Sales Transaction Data

*# Main workflow*

1. Load raw data (CSV/Parquet)

2. Clean and standardize

3. Handle missing values

4. Process special columns (rooms, parking)

5. Engineer features (especially dates)

6. Encode categoricals

7. Scale numericals

8. Save processed data

**2. Feature Selection Analysis**

**A. Location Features**

**Kept:**

area\_en: Primary location indicator

project\_name\_en: Specific property development

master\_project\_en: Larger development context

nearest\_landmark\_en, nearest\_metro\_en, nearest\_mall\_en: Proximity features

**Removed:**

All Arabic versions (\*\_ar)

Location IDs (redundant with names)

parcel\_id (too granular)

**B. Property Characteristics**

**Kept:**

property\_size\_sqm: Essential size metric

property\_type\_en & property\_subtype\_en: Property classification

property\_usage\_en: Usage category

rooms\_en: Room configuration

parking: Parking availability

is\_freehold: Ownership type

is\_offplan: Development status

**Removed:**

Duplicate IDs

Arabic descriptions

Zero-value columns like building\_age

**C. Transaction/Contract Details**

**Kept:**

transaction\_datetime/contract\_dates: Timing information

amount/annual\_amount: Target variables

total\_buyer/total\_seller: Transaction participants

transaction\_type\_en: Transaction classification

**Removed:**

contract\_amount (using annual for standardization)

System identifiers (ejari numbers, etc.)

Internal tracking fields

**3. Data Processing Methodology**

**A. Missing Data Strategy**

Categorical Data:

- Fill with configured default value

- Ensure consistency across similar fields

Numerical Data:

- Use median for most fields

- Special handling for critical fields

- Document fill strategies

**B. Feature Engineering**

1. **Temporal Features:**

Convert dates to timestamps

Calculate durations

Remove original date columns

**Categorical Processing:**

Persistent label encoding

Handle new categories

Maintain encoding consistency

**Numerical Scaling:**

Robust scaling for outlier handling

Separate scalers for rent/sale

Preserve target variables

**5. Technical Implementation Details**

**A. File Management**

1. Input Handling:

- Support both CSV and Parquet

- Convert to Parquet for efficiency

- Maintain raw data copies

2. Output Organization:

- Separate processed files

- Maintain transformers

- Clean up temporary files

**B. Performance Considerations**

Efficient data type usage

Optimized file formats

Scalable processing approach

**6. Business Logic Integration**

**A. Rental Properties**

Key Factors:

- Annual rental value

- Contract duration

- Property characteristics

- Location details

**B. Sales Transactions**

Key Factors:

- Transaction value

- Property details

- Market conditions

- Location impact

**Feature Selection**

**Comprehensive Feature Selection Pipeline Analysis**

**1. Data Loading and Setup**

The code begins with necessary imports and a data loading function:

def load\_data(*file\_path*):

*return* pd.read\_parquet(file\_path)

This simple function is used throughout the pipeline to load Parquet files containing real estate data for both rentals and sales.

**2. Correlation Analysis**

The first feature selection method identifies highly correlated features:

def correlation\_analysis(*data*, *threshold*):

    numeric\_data = data.select\_dtypes(*exclude*=['datetime64'])

    corr\_matrix = numeric\_data.corr().abs()

*# Find highly correlated pairs*

    upper = corr\_matrix.where(np.triu(np.ones(corr\_matrix.shape), *k*=1).astype(bool))

Key aspects:

Excludes datetime columns

Creates correlation matrix

Uses upper triangle to avoid duplicate pairs

Returns features to drop based on threshold

**3. Tree-Based Feature Importance**

Uses Random Forest to determine feature importance:

def feature\_importance\_tree\_based(*data*, *target\_variable*):

    model = RandomForestRegressor(

*n\_estimators*=100,

*max\_depth*=20,

*min\_samples\_leaf*=20,

*max\_features*=0.7,

*n\_jobs*=-1

    )

Important characteristics:

Uses balanced hyperparameters

Utilizes all CPU cores

Returns sorted feature importance scores

Good balance between speed and accuracy

**4. Visualization Component**

Creates visual representations of feature importance:

def plot\_and\_save\_feature\_importance(*feature\_importances*, *model\_type*):

    plt.figure(*figsize*=EVALUATION['plot\_figsize'])

    feature\_importances.plot(*kind*='bar')

*# Save plot to configured directory*

Features:

Configurable plot size

Automatic file saving

Separate plots for rent and sale data

Clear titles and labels

**5. Recursive Feature Elimination**

Implements iterative feature selection:

def recursive\_feature\_elimination(*data*, *target\_variable*):

    model = RandomForestRegressor(

*n\_estimators*=50,  *# Simplified for speed*

*max\_depth*=15

    )

    rfe = RFE(model, *n\_features\_to\_select*=n\_features\_to\_select, *step*=2)

Key points:

Uses simplified Random Forest for speed

Small step size for accuracy

Configurable number of features to select

Returns selected feature names

**6. Univariate Feature Selection**

Evaluates features independently:

def univariate\_feature\_selection(*data*, *target\_variable*, *k*):

    selector = SelectKBest(*score\_func*=f\_regression, *k*=k)

*# Select top k features based on F-scores*

def univariate\_feature\_selection(*data*, *target\_variable*, *k*):

    selector = SelectKBest(*score\_func*=f\_regression, *k*=k)

*# Select top k features based on F-scores*

Characteristics:

Uses f\_regression for scoring

Configurable number of features

Fast computation

Independent feature evaluation

**7. Feature Combination Strategy**

Combines all methods into final selection:

def combine\_selected\_features(*data*, *target\_variable*):

*# Get features from each method*

    corr\_features = correlation\_analysis(data)

    importance\_features = feature\_importance\_tree\_based(data, target\_variable)

    rfe\_features = recursive\_feature\_elimination(data, target\_variable)

    univariate\_features = univariate\_feature\_selection(data, target\_variable)

*# Combine with priority to RF importance*

    combined\_features = set(importance\_features.head(

        FEATURE\_SELECTION['importance\_features\_count']).index)

    combined\_features = combined\_features.union(rfe\_features, univariate\_features)

    combined\_features = combined\_features.difference(corr\_features)

Strategy details:

Prioritizes Random Forest importance

Removes highly correlated features

Combines multiple selection methods

Configurable feature counts

**8. Main Execution Flow**

The main script orchestrates the entire process:

*if* \_\_name\_\_ == "\_\_main\_\_":

*# Process rent data*

    rent\_selected\_features = combine\_selected\_features(rent\_data, rent\_target)

    rent\_importance = feature\_importance\_tree\_based(rent\_data, rent\_target)

    plot\_and\_save\_feature\_importance(rent\_importance, 'rent')

*# Process sale data*

    sale\_selected\_features = combine\_selected\_features(sale\_data, sale\_target)

    sale\_importance = feature\_importance\_tree\_based(sale\_data, sale\_target)

    plot\_and\_save\_feature\_importance(sale\_importance, 'sale')

Execution features:

Separate processing for rent and sale data

Generates visualizations

Prints selected features

Shows progress information

**Configuration and Customization**

The code uses configuration files for key parameters:

Correlation thresholds

Feature counts for each method

Plot settings

File paths

Random seed

**Advantages of This Approach**

**Comprehensive**: Combines multiple feature selection techniques

**Configurable**: Easy to adjust parameters

**Modular**: Functions can be used independently

**Scalable**: Handles large datasets efficiently

**Visual**: Includes visualization capabilities

**Robust**: Multiple selection criteria reduce bias

**Base Model Implementation**

The base\_models.py script includes:

**1. Purpose**

This is a machine learning pipeline for training and optimizing multiple regression models (XGBoost, Random Forest, and SVR) for both rent and sale price predictions.

**2. Main Components**

**Data Loading and Preparation**

def load\_and\_prepare\_data(*file\_path*, *target\_variable*, *selected\_features*):

    data = pd.read\_parquet(file\_path)

    features = [f *for* f *in* selected\_features *if* f != target\_variable]

    X = data[features]

    y = data[target\_variable]

*return* X, y

**- Loads parquet files**

Separates features and target variables

**Model Optimization**

The code includes three optimization functions for different models:

def optimize\_xgboost(*X*, *y*, *model\_type*):

    def xgb\_evaluate(*max\_depth*, *learning\_rate*, *n\_estimators*, *min\_child\_weight*, *subsample*):

        params = {

            'max\_depth': int(max\_depth),

            'learning\_rate': learning\_rate,

            'n\_estimators': int(n\_estimators),

            'min\_child\_weight': min\_child\_weight,

            'subsample': subsample

        }

*# ... optimization logic*

- Uses Bayesian Optimization for hyperparameter tuning

Includes cross-validation

Creates visualization plots for optimization process

**Visualization Functions**

def plot\_optimization\_results(*optimizer*, *model\_name*, *model\_type*):

*# Creates plots showing optimization progress*

*# Saves plots with timestamps*

**Model Training and Saving**

def train\_and\_save\_models(*X*, *y*, *model\_type*, *output\_dir*=MODELS\_DIR):

*# Trains models with optimized parameters*

*# Uses parallel processing*

*# Saves models using joblib*

**3. Key Features**

**Parallel Processing**

Uses joblib's Parallel and delayed for concurrent execution

Processes rent and sale datasets simultaneously

**Error Handling**

Comprehensive logging throughout the pipeline

Try-except blocks for robust error management

**Resource Management**

SVR model includes data sampling for large datasets

Memory management through cache size settings

**Visualization**

Creates optimization progress plots

Generates cross-validation score plots

**4. Main Execution Flow**

1. Loads configuration from external config file

Processes both rent and sale datasets in parallel

For each dataset:

Loads and prepares data

Selects features

Optimizes and trains three models (XGBoost, Random Forest, SVR)

Saves trained models

Creates visualization plots

**5. Notable Features**

Uses parquet file format for data storage

Implements Bayesian Optimization for hyperparameter tuning

Includes comprehensive logging

Handles large datasets through sampling

Creates timestamped visualization plots

Parallel processing with resource management

Modular design with separate functions for each major task

**Meta-Learner Implementation**

The meta\_learner.py script contains:

**Overview**

This code implements a meta-learning system that combines predictions from multiple base models (XGBoost, Random Forest, and SVR) using a neural network. It's designed to work with both rental and sale price predictions.

**Key Components**

**Initialization and Base Model Loading**

class MetaLearner:

    def \_\_init\_\_(*self*, *base\_models\_dir*=MODELS\_DIR, *model\_type*='rent'):

        self.model\_type = model\_type

        self.base\_models = self.\_load\_base\_models(base\_models\_dir)

        self.scaler = StandardScaler()

        self.meta\_model = None

- Loads pre-trained base models (XGBoost, Random Forest, SVR)

Initializes a scaler for normalizing predictions

Supports both rent and sale price predictions

**Base Model Predictions**

def \_get\_base\_predictions(*self*, *X*):

    X\_clean = X.copy()

*# Remove target column if present*

*if* self.model\_type == 'rent' and 'annual\_amount' in X\_clean.columns:

        X\_clean = X\_clean.drop('annual\_amount', *axis*=1)

    predictions = np.column\_stack([

        model.predict(X\_ordered) *for* model *in* self.base\_models.values()

    ])

*return* predictions

- Gets predictions from all base models

Combines them into a single array

**Meta-Model Architecture**

def \_create\_meta\_model(*self*, *num\_hidden\_layers*, *hidden\_units*, *dropout\_rate*, *learning\_rate*):

    model = models.Sequential()

    model.add(layers.Input(*shape*=(3,)))  *# 3 base models*

*for* \_ *in* range(int(num\_hidden\_layers)):

        model.add(layers.Dense(int(hidden\_units), *activation*='relu'))

        model.add(layers.Dropout(dropout\_rate))

Creates a neural network that takes base model predictions as input

Configurable number of layers, units, and dropout rate

Uses Adam optimizer with MSE loss

**Hyperparameter Optimization**

Uses Bayesian Optimization to find optimal hyperparameters

Optimizes:

Number of hidden layers

Number of hidden units

Dropout rate

Learning rate

Includes visualization of optimization results

**Training and Visualization**

def fit(*self*, *X*, *y*):

*# Optimize hyperparameters*

    best\_params = self.optimize\_meta\_model(X, y)

*# Train final model with best parameters*

    self.meta\_model = self.\_create\_meta\_model(...)

*# Add early stopping*

    early\_stopping = tf.keras.callbacks.EarlyStopping(

*monitor*='loss',

*patience*=3,

*restore\_best\_weights*=True

    )

- Includes early stopping to prevent overfitting

Generates training history plots

Saves optimization and training visualizations

**Model Persistence**

def save(*self*, *output\_dir*='models'):

    self.meta\_model.save(f'{output\_dir}/meta\_learner\_{self.model\_type}.h5')

    joblib.dump(self.scaler, f'{output\_dir}/meta\_learner\_scaler\_{self.model\_type}.joblib')

- Saves trained meta-model and scaler for later use

7. **Main Execution**

Processes both rent and sale data

Includes error handling and debugging information

Uses feature selection to prepare input data

**Key Features**

Ensemble learning approach

Automated hyperparameter optimization

Visualization of training and optimization

Error handling and logging

Support for both rental and sale price predictions

Feature selection integration

Model persistence

**Config.py**

The config.py script evaluates models using:

**Directory Structure Setup**

The code establishes a base directory structure for the project

Creates essential directories for data, models, logs, plots, and reports

Automatically creates these directories if they don't exist

**Data Configuration**

Defines two main data sources: rent and sale data

For each data source, it specifies:

Input and output file paths

Target variable for prediction

List of columns to be removed during preprocessing

Uses parquet format for cleaned data storage

**Model Parameters Configuration**

Configures three different models: XGBoost, Random Forest, and SVR

For each model, defines:

Parameter bounds for optimization (pbounds)

Default parameters that remain constant

XGBoost parameters focus on tree structure and learning

Random Forest parameters focus on ensemble characteristics

SVR parameters focus on kernel and margin settings

**Meta-Learner Configuration**

Defines parameters for a neural network meta-learner

Includes optimization settings:

Network architecture bounds (layers, units)

Training parameters (dropout, learning rate)

Specifies training configuration:

Validation split ratio

Number of epochs

Batch size

Verbosity level

**Feature Selection Settings**

Defines thresholds and parameters for feature selection:

Correlation threshold for removing highly correlated features

Number of features to select using importance-based methods

Settings for Recursive Feature Elimination (RFE)

Parameters for univariate feature selection

**Preprocessing Configuration**

Defines strategies for handling missing data:

Categorical missing values filled with "Unknown"

Numerical missing values filled with median

Specifies robust scaling as the scaling method

**Logging Configuration**

Sets up logging parameters:

Log level set to INFO

Defines log format with timestamp

Specifies log file location

**Evaluation Settings**

Configures model evaluation parameters:

Defines evaluation metrics (RMSE, R2, MAE)

Sets number of cross-validation folds

Specifies plot dimensions

**Reproducibility**

Sets a random seed (42) for reproducible results

This configuration file serves as a central control panel for the entire machine learning pipeline, making it easy to modify parameters and settings without changing the core code. It's well-organized and separates different aspects of the ML workflow into distinct sections.

**Evaluation Metrics**

The evaluation.py script evaluates models using:

**Model Evaluation Code Overview**

This code implements a comprehensive model evaluation system for real estate price predictions (both rental and sale prices). Here's a detailed breakdown:

**1. Main Class: ModelEvaluator**

The core class ModelEvaluator handles all evaluation tasks. It's initialized with a model type:

evaluator = ModelEvaluator(*model\_type*='rent')  *# or 'sale'*

evaluator = ModelEvaluator(*model\_type*='rent')  *# or 'sale'*

**2. Key Components**

**a. Metrics Calculation**

The class calculates several important regression metrics:

RMSE (Root Mean Square Error)

R² Score (Coefficient of Determination)

MAE (Mean Absolute Error)

MSE (Mean Square Error)

MAPE (Mean Absolute Percentage Error)

Explained Variance

def calculate\_metrics(*self*, *y\_true*, *y\_pred*):

    mse = mean\_squared\_error(y\_true, y\_pred)

    rmse = np.sqrt(mse)

    r2 = r2\_score(y\_true, y\_pred)

    mae = mean\_absolute\_error(y\_true, y\_pred)

    mape = np.mean(np.abs((y\_true - y\_pred) / y\_true)) \* 100

    explained\_variance = 1 - (np.var(y\_true - y\_pred) / np.var(y\_true))

*# ... returns dictionary of metrics*

**b. Visualization**

The class creates three types of plots:

1. Residual Plot (actual - predicted vs predicted)

Residuals Distribution (histogram with KDE)

Actual vs Predicted Scatter Plot

Example of plotting actual vs predicted:

def plot\_actual\_vs\_predicted(*self*, *y\_true*, *y\_pred*, *save\_dir*=PLOTS\_DIR):

    plt.scatter(y\_true, y\_pred, *alpha*=0.5)

    plt.plot([min\_val, max\_val], [min\_val, max\_val], 'r--')  *# Perfect prediction line*

    plt.xlabel('Actual Values')

    plt.ylabel('Predicted Values')

*# ... saves plot to specified directory*

**c. Reporting**

Generates a comprehensive evaluation report including:

All calculated metrics

Basic statistics

References to generated plots

**3. Logging System**

The code implements robust logging:

Creates separate log files for rent and sale models

Logs success and error messages

Uses standard Python logging module

self.logger = logging.getLogger(f'{model\_type}\_evaluator')

handler = logging.FileHandler(os.path.join(LOGS\_DIR, f'{model\_type}\_evaluation.log'))

**4. Configuration**

The code uses external configuration (from config.py) for:

Plot sizes (EVALUATION['plot\_figsize'])

Directory paths (PLOTS\_DIR, REPORTS\_DIR, LOGS\_DIR)

Metrics to include in reports (EVALUATION['metrics'])

**5. Convenience Function**

Provides a simple interface for quick evaluation:

metrics = evaluate\_model(y\_true, y\_pred, *model\_type*='rent')

**Key Features**

1. **Error Handling**: All methods include try-except blocks with proper error logging

**Flexibility**: Works with both rental and sale price predictions

**Visualization**: Multiple plot types for comprehensive analysis

**Detailed Reporting**: Generates detailed text reports with metrics and statistics

**Configurability**: Uses external configuration for easy customization

**Logging**: Comprehensive logging system for debugging and monitoring

**Usage Example**

*# Create evaluator instance*

evaluator = ModelEvaluator(*model\_type*='rent')

*# Generate full evaluation*

report = evaluator.generate\_evaluation\_report(y\_true, y\_pred)

*# Or use convenience function*

metrics = evaluate\_model(y\_true, y\_pred, *model\_type*='rent')

This code is particularly useful for:

Model performance assessment

Model comparison

Result visualization

Documentation of model performance

Debugging model issues through residual analysis

**Main.py**

The main.py script evaluates models using:

**Real Estate Price Prediction Pipeline**

This code implements a machine learning pipeline for predicting real estate prices for both rental and sale properties. Here's a detailed breakdown:

**1. Project Structure and Setup**

The code uses a modular structure with separate modules for different functionalities:

preprocess.py: Data preprocessing

feature\_selection.py: Feature selection and data loading

base\_models.py: Training base models

meta\_learner.py: Meta-learning implementation

evaluation.py: Model evaluation

config.py: Configuration settings

**Logging Setup**

logging.basicConfig(

*level*=LOGGING\_CONFIG['level'],

*format*=LOGGING\_CONFIG['format'],

*handlers*=[

        logging.FileHandler(os.path.join(LOGS\_DIR, 'pipeline.log')),

        logging.StreamHandler()

    ]

)

**This sets up logging to both file and console output.**

**2. Preprocessing Pipeline**

The preprocessing function handles both rent and sale data:

def run\_preprocessing\_pipeline():

*for* data\_type, params *in* DATA\_FILES.items():

        preprocess\_data(

            params['input'],

            params['output'],

            params['columns\_to\_remove'],

            data\_type

        )

This processes raw data files according to configuration parameters.

**3. Model Training**

The training pipeline consists of three main steps:

**a. Data Preparation**

def train\_models(*model\_type*):

    data = load\_data(data\_path)

    selected\_features = combine\_selected\_features(data, target)

    X, y = load\_and\_prepare\_data(data\_path, target, list(selected\_features))

**b. Base Models Training**

The pipeline trains multiple base models and saves them to disk.

**c. Meta-Learner Training**

meta\_learner = MetaLearner(MODELS\_DIR, model\_type)

meta\_learner.fit(X, y)

meta\_learner.save()

A meta-learner combines predictions from base models for better performance.

**4. Model Evaluation**

The evaluation process:

def evaluate\_models(*X*, *y*, *meta\_learner*, *model\_type*):

    predictions = meta\_learner.predict(X)

    metrics = evaluate\_model(y, predictions, model\_type)

- Makes predictions using the meta-learner

Calculates performance metrics

Saves results to a file

Logs the results

**5. Main Pipeline**

The main function orchestrates the entire process:

Creates necessary directories

Runs preprocessing

Trains models for both rent and sale data

Evaluates models and logs results

Handles errors and tracks execution time

**Key Features:**

**Error Handling**: Comprehensive error logging and exception handling

**Modularity**: Clear separation of concerns across different modules

**Configurability**: Uses external configuration files

**Logging**: Detailed logging of process steps and results

**Dual Purpose**: Handles both rental and sale price predictions

**Meta-Learning**: Uses ensemble learning through a meta-learner

**Usage**

To run the pipeline:

*if* \_\_name\_\_ == "\_\_main\_\_":

    main()

This will execute the entire pipeline from preprocessing to evaluation for both rental and sale price predictions.

The code follows best practices for machine learning pipelines:

Clear separation of concerns

Comprehensive logging

Error handling

Configuration management

Modular design

Reusable components

This makes it maintainable, scalable, and suitable for production environments.

**Test File Overview**

This is a pytest-based test file for what appears to be a machine learning pipeline focused on real estate price prediction. The pipeline handles both rental and sale price predictions.

**Key Components**

**1. Imports and Dependencies**

*import* pytest

*import* pandas *as* pd

*import* numpy *as* np

*from* unittest.mock *import* patch, MagicMock

**The file uses pytest for testing, pandas/numpy for data handling, and unittest.mock for mocking dependencies.**

**2. Test Fixtures**

Two main fixtures are defined:

@pytest.fixture

def sample\_data():

    """Create sample dataset for testing."""

*return* pd.DataFrame({

        'price': [100000, 200000, 300000],

        'area': [50, 75, 100],

        'rooms': [2, 3, 4],

        'location': ['A', 'B', 'C']

    })

This fixture creates a sample DataFrame with real estate properties and their features.

**3. Directory Setup Testing**

Tests the creation of necessary directories for models and logs:

def test\_setup\_directories(*tmp\_path*):

*with* patch('notebooks.config.MODELS\_DIR', str(tmp\_path / 'models')):

        setup\_directories()

*assert* os.path.exists(str(tmp\_path / 'models'))

**4. Preprocessing Pipeline Testing**

Tests the data preprocessing pipeline using parametrize to test both rent and sale scenarios:

@pytest.mark.parametrize("data\_type", ["rent", "sale"])

def test\_run\_preprocessing\_pipeline(*data\_type*, *sample\_data*):

**5. Model Training Testing**

Tests the model training pipeline with extensive mocking of dependencies:

Mocks data loading

Mocks feature selection

Mocks model training

Mocks meta-learner operations

**6. Model Evaluation Testing**

Tests the evaluation metrics calculation:

def test\_evaluate\_models(*sample\_data*, *mock\_meta\_learner*):

    """Test model evaluation pipeline."""

    metrics = evaluate\_models(X, y, mock\_meta\_learner, model\_type)

*assert* isinstance(metrics, dict)

*assert* all(key *in* metrics *for* key *in* ['mae', 'mse', 'r2'])

**7. Error Handling Testing**

Tests how the pipeline handles exceptions:

def test\_main\_error\_handling():

*with* patch('notebooks.main.run\_preprocessing\_pipeline',

*side\_effect*=Exception("Test error")):

*with* pytest.raises(Exception) *as* exc\_info:

            main()

**8. Integration Testing**

Tests the full pipeline integration by mocking major components and ensuring they're called in the correct order.

**Key Testing Patterns Used**

1. **Fixture Usage**: Reusable test data and mock objects

**Parametrization**: Testing multiple scenarios with the same test function

3. **Mocking**: Extensive use of patch and MagicMock to isolate components

**Context Managers**: Using with statements for temporary modifications

**Assertion Patterns**: Checking both function calls and return values

**Pipeline Structure**

The tests reveal a machine learning pipeline with these main steps:

1. Directory Setup

Data Preprocessing

Model Training

Model Evaluation

Each step is tested independently and as part of the full integration test.

This test suite demonstrates good testing practices:

Independent test functions

Proper isolation through mocking

Clear test names and docstrings

Comprehensive coverage of happy path and error cases

Both unit and integration testing approaches

**Docker Setup**

**1. Create Dockerfile**

Add the following content to the Dockerfile:

FROM python:3.10-slim

WORKDIR /app

COPY . .

RUN pip install -r requirements.txt

CMD ["python", "main.py"]

**2. Build Docker Image**

docker build -t avm-app .

**3. Run Docker Container**

docker run -it avm-app