Ensemble Pipeline Tutorial

A comprehensive machine learning pipeline for ensemble modeling with ONNX export capabilities and optimization features.

Overview

This pipeline provides a robust framework for training, evaluating, and deploying ensemble machine learning models. It includes features for:

- Model training and evaluation with cost-sensitive optimization
- Hyperparameter tuning using Bayesian optimization
- ONNX model export for deployment
- Interactive configuration through Streamlit UI
- Comprehensive model evaluation and visualization

Installation

1. Clone the repository:

```bash
git clone https://github.com/QuinnMazaris/ensamble_pipelineV2.git
cd ensemble_pipelineV2
```

2. Set up Python environment:

    - **Python Version**: Python 3.12 is recommended for best compatibility
    - **Windows Users**: If using Python < 3.12, you must enable long path support
    - Create and activate virtual environment:

    ```bash
    # Windows
    python -m venv venv
    .\venv\Scripts\activate

    # Linux/Mac
    python -m venv venv
    source venv/bin/activate
    ```

3. Install dependencies:

```bash
pip install -r requirements.txt
```

Usage

Running the Pipeline

1. Start the Streamlit interface:

```
streamlit run app/main.py

# Or if looking to host locally

streamlit run app/app.py --server.address 0.0.0.0 --server.port 8501
```

The Streamlit interface will be available at:

- `http://localhost:8501` (local access)
- `http://<your-ip-address>:8501` (network access)

2. Configure settings through the UI or `config.py`:

   - Adjust cost weights
   - Enable/disable k-fold cross-validation
   - Configure feature filtering
   - Set optimization parameters
   - Choose export format

3. Run the pipeline through the UI or `run.py`.

Model Evaluation

The pipeline provides comprehensive evaluation metrics and visualizations to assess model performance:

1. **Performance Metrics**:

   - **Precision**:
     - Ratio of true positives to all predicted positives (TP/(TP+FP))
     - Measures how many of the predicted defects are actual defects
     - Higher precision means fewer false alarms
   - **Recall**:
     - Ratio of true positives to all actual positives (TP/(TP+FN))
     - Measures how many of the actual defects are caught
     - Higher recall means fewer missed defects
   - **Accuracy**:
     - Overall correctness (TP+TN)/(TP+TN+FP+FN)
     - General measure of model performance
     - Can be misleading with imbalanced data
   - **Cost-weighted Performance**:
     - Combines precision and recall using cost weights (C_FP and C_FN)
     - C_FP (False Positive Cost): Cost of false alarms (default = 1)
     - C_FN (False Negative Cost): Cost of missed defects (default = 30)
     - Total Cost = (FP × C_FP) + (FN × C_FN)

        - Final cost is calculated from the test set's confusion matrix, where FP and FN are the actual counts of false positives and false negatives
    - **Confusion Matrix**:
        - Shows detailed breakdown of predictions:
            - True Negatives (TN): Correctly identified good parts
            - False Positives (FP): False alarms (good parts marked as defects)
            - False Negatives (FN): Missed defects
            - True Positives (TP): Correctly identified defects
    - **ROC Curve**:
        - Available in Streamlit UI's Model Analysis tab
        - Interactive Plotly visualization
        - Plots True Positive Rate (Recall) vs False Positive Rate
        - Shows trade-off between sensitivity and specificity
        - Area Under Curve (AUC) indicates overall model performance
    - **PR Curve** (Precision-Recall):
        - Available in Streamlit UI's Model Analysis tab
        - Interactive Plotly visualization
        - Plots Precision vs Recall
        - Better suited for imbalanced data than ROC
        - Shows trade-off between precision and recall at different thresholds

2. **Visualization**:

    - **Threshold Sweep Plots**:
        - Shows how metrics change across different decision thresholds
        - Identifies optimal thresholds for:
            - Cost minimization (using C_FP and C_FN)
            - Accuracy maximization
        - Helps balance false positives and false negatives
    - **Model Comparison Plots**:
        - Side-by-side comparison of all models
        - Shows performance at:
            - Cost-optimal threshold
            - Accuracy-optimal threshold
        - Includes key metrics for each model
    - **Class Balance Visualization**:
        - Shows distribution of classes in dataset
        - Displays effect of SMOTE resampling
        - Helps understand class imbalance

Project Structure

```
ensemble_pipelineV2/
├── app/                    # Streamlit application
│   ├── app.py             # Main Streamlit application
│   ├── sidebar.py         # Configuration sidebar
│   ├── tabs.py            # Interactive model configuration
```

```
│   ├── utils.py             # Streamlit-specific utilities
│   └── __init__.py          # Package initialization
├── helpers/                 # Core functionality
│   ├── data.py              # Data preparation and processing
│   ├── modeling.py          # Model training and optimization
│   ├── metrics.py           # Evaluation metrics
│   ├── model_export.py      # ONNX export functionality
│   ├── plotting.py          # Visualization utilities
│   ├── reporting.py         # Results reporting and analysis
│   ├── utils.py             # General utilities
│   ├── export_metrics_for_streamlit.py  # Streamlit metrics export
│   └── __init__.py          # Package initialization
├── data/                    # Data directory
├── output/                 # Output directory for models, plots, and predictions
├── config.py               # Configuration settings
├── run.py                  # Main pipeline execution
├── test_onnx.py            # ONNX export testing
├── requirements.txt         # Project dependencies
└── README.md               # Project documentation
```

Code Architecture

Directory Organization

The project follows a modular architecture with clear separation of concerns:

1. **Frontend (`app/` directory)**:
   - Built with Streamlit for interactive web interface
   - `app.py`: Main application entry point and tab management
   - `sidebar.py`: Configuration UI and settings management
   - `tabs.py`: Individual tab implementations for different features
   - `utils.py`: Frontend-specific utilities and helper functions
   - `__init__.py`: Package initialization
2. **Backend (`helpers/` directory)**:
   - Core ML pipeline functionality
   - `data.py`: Data loading, preprocessing, and feature engineering
   - `modeling.py`: Model training, optimization, and evaluation
   - `metrics.py`: Performance metrics and evaluation functions
   - `model_export.py`: Model export to ONNX and pickle formats
   - `plotting.py`: Visualization and plotting utilities
   - `reporting.py`: Results analysis and reporting
   - `utils.py`: General utility functions
   - `export_metrics_for_streamlit.py`: Metrics export for frontend display
   - `__init__.py`: Package initialization and exports
3. **Configuration (`config.py`)**:
   - Central configuration file

- Model definitions and hyperparameters
- Pipeline settings and parameters
- File paths and directory structure

4. **Data and Output**:

    - `data/`: Input data directory
    - `output/`: Generated files (models, plots, predictions)
        - `models/`: Trained model files
        - `plots/`: Generated visualizations
        - `predictions/`: Model predictions and results -`streamlit_data/`: Model predictions and results for front end

Frontend-Backend Interaction

The application uses a Python-based architecture where Streamlit serves as both frontend and backend:

1. **Frontend (Streamlit UI)**:

    - Provides interactive web interface
    - Manages user input and configuration
    - Displays results and visualizations
    - Organized into tabs:
        - Overview: Project summary and status
        - Data Management: Dataset handling
        - Preprocessing Config: Feature engineering settings
        - Model Zoo: Model selection and configuration
        - Model Analysis: Performance metrics and analysis
        - Plots Gallery: Visualization dashboard
        - Downloads: Export model files and results

2. **Backend (Python Pipeline)**:

    - Executes ML pipeline operations
    - Handles data processing and model training
    - Manages model optimization and evaluation
    - Exports models and results
    - Key components:
        - Data Pipeline: Loading → Preprocessing → Feature Engineering
        - Model Pipeline: Training → Optimization → Evaluation
        - Export Pipeline: Model Conversion → File Export

3. **Data Flow**:

```
User Input (Streamlit UI)
        ↓
Configuration (config.py)
        ↓
Backend Processing (helpers/)
```

```
        ↓
Results & Models (output/)
        ↓
Display (Streamlit UI)
```

4. **Configuration Management**:
   - Settings stored in `config.py`
   - UI updates config through `sidebar.py`
   - Backend reads config for pipeline execution
   - Changes can be saved permanently to config file

5. **Model Management**:
   - Models defined in `config.py`
   - Training handled by `helpers/modeling.py`
   - Export managed by `helpers/model_export.py`
   - UI configuration through Model Zoo tab

Key Features

1. **Modular Design**:
   - Separate frontend and backend components
   - Reusable helper functions
   - Clear separation of concerns
   - Easy to extend and modify

2. **Interactive Configuration**:
   - Real-time parameter adjustment
   - Immediate feedback on changes
   - Persistent configuration storage
   - Flexible model customization

3. **Comprehensive Pipeline**:
   - End-to-end ML workflow
   - Automated data processing
   - Model optimization and evaluation
   - Results visualization and export

4. **Extensible Architecture**:
   - Easy to add new models
   - Customizable preprocessing steps
   - Flexible export options
   - Configurable evaluation metrics

Configuration

The pipeline is highly configurable through `config.py` and the Streamlit UI. Each setting controls specific aspects of the model training and optimization process:

Model Settings

1. **Cost Weights**:
   - `C_FP` (False Positive Cost): Default = 1
   - `C_FN` (False Negative Cost): Default = 30
   - These weights are used in the cost-sensitive evaluation of models
   - Higher `C_FN` prioritizes reducing false negatives (missed defects)
   - Used in threshold optimization to find the optimal decision boundary
   - Implementation: `helpers/metrics.py` calculates weighted costs for model evaluation

2. **Cross-Validation**:
   - `USE_KFOLD`: Enable/disable k-fold cross-validation
   - `N_SPLITS`: Number of folds (default = 5)
   - When enabled:
     - Data is split into `N_SPLITS` stratified folds
     - Models are trained and evaluated on each fold
     - Results are averaged across folds for final metrics
     - Helps assess model stability and generalization
   - When disabled:
     - Uses a single 80/20 train-test split
     - Faster training but less robust evaluation
   - Implementation: `helpers/modeling.py` handles both split types

3. **Feature Filtering**:
   - `FilterData`: Enable/disable feature filtering
   - When enabled, applies two filtering steps:
     1. **Variance Filter** (`VARIANCE_THRESH` = 0.01):
        - Removes features with variance below threshold
        - Eliminates near-constant features
        - Implementation: `helpers/data.py` → `apply_variance_filter()`
     2. **Correlation Filter** (`CORRELATION_THRESH` = 0.95):
        - Removes highly correlated features
        - Keeps one feature from each correlated group
        - Implementation: `helpers/data.py` → `apply_correlation_filter()`
   - Both filters are applied sequentially during data preprocessing

Optimization Settings

1. **Hyperparameter Optimization**:
   - `OPTIMIZE_HYPERPARAMS`: Enable/disable Bayesian optimization
   - `HYPERPARAM_ITER`: Number of optimization trials (default = 50)
   - Uses Optuna for Bayesian optimization:

        - Efficiently searches the hyperparameter space
        - Optimizes based on cross-validation performance
        - Supports early stopping for faster optimization
    - Parameter spaces defined in `config.py`:

    ```
    HYPERPARAM_SPACE = {
        'XGBoost': {
            'max_depth': [3,4,5,6],
            'learning_rate': [0.01,0.05,0.1],
            'subsample': [0.6,0.8,1.0],
            'colsample_bytree': [0.6,0.8,1.0],
            'n_estimators': [100,200,400],
            'gamma': [0,0.1,0.2],
        },
        'RandomForest': {
            'n_estimators': [100,200,300],
            'max_depth': [None,5,10],
            'min_samples_split': [2,5,10],
            'min_samples_leaf': [1,2,5],
        }
    }
    ```

    - If the user chooses to not use this feature, they are able to adjust the parameters manually via the UI or in `config.py`
    - Implementation: `helpers/modeling.py` → `optimize_hyperparams()`

2. **Final Model Optimization**:

    - `OPTIMIZE_FINAL_MODEL`: Enable/disable production model training
    - When enabled:
        - If `OPTIMIZE_HYPERPARAMS` is also enabled:
            1. Performs hyperparameter optimization on the full dataset
            2. Uses `HYPERPARAM_ITER` trials (default = 50) to find best parameters
            3. Trains final model with optimized parameters
        - If `OPTIMIZE_HYPERPARAMS` is disabled:
            1. Uses default model parameters from `config.py`
            2. No hyperparameter optimization is performed
            3. Trains final model with default parameters
        - In both cases:
            1. Trains on the entire dataset
            2. Performs cost-sensitive threshold optimization
            3. Saves the production-ready model
    - When disabled:
        - Skips final model training completely
        - No production model is created or saved
        - Only cross-validation or single-split models are generated
    - Implementation: `helpers/modeling.py` → `FinalModelCreateAndAnalyize()`

3. **Class Imbalance Handling**:

   - `USE_SMOTE`: Enable/disable SMOTE (Synthetic Minority Over-sampling Technique)
   - `SMOTE_RATIO`: Ratio of minority to majority class (default = 0.5)
   - When enabled:
     - Generates synthetic samples for the minority class
     - Helps prevent model bias towards majority class
     - Applied only to training data, not validation/test
   - Implementation: `run.py` applies SMOTE during data preparation
   - Note: XGBoost models also use `scale_pos_weight` for additional imbalance handling

Model Export Settings

1. **ONNX Export**:
   - `EXPORT_ONNX`: Enable/disable ONNX model export
   - `ONNX_OPSET_VERSION`: ONNX operator set version (default = 12)
   - When enabled:
     - Converts models to ONNX format for deployment
     - Preserves feature names and model metadata
     - Supports deployment in production environments
   - When disabled:
     - Falls back to pickle format
     - Still maintains all model functionality
   - Implementation: `helpers/model_export.py` → `export_model()`

Additional Settings

1. **Data Splitting**:

   - `TEST_SIZE`: Proportion of data for testing (default = 0.2)
   - `RANDOM_STATE`: Random seed for reproducibility (default = 42)

2. **Output Control**:

   - `SAVE_MODEL`: Save trained models
   - `SAVE_PLOTS`: Generate and save visualizations
   - `SAVE_PREDICTIONS`: Save model predictions
   - `SUMMARY`: Print detailed progress information

Settings can be modified through the Streamlit UI or directly in `config.py`. Changes made in the UI can be saved permanently to `config.py` using the "Save & Reload Config" button in the sidebar.

Model Export and Deployment

ONNX Export

The pipeline automatically handles model conversion to ONNX format:

1. **Export Process**:

- Models are converted using `skl2onnx`
- Feature names are preserved for input mapping
- Opset version 12 is used by default
- Automatic fallback to pickle if ONNX unavailable

2. **ONNX to Halcon**:

   - ONNX models can be imported into Halcon
   - Input/output tensor names are preserved
   - Model metadata includes feature names and thresholds

Export Settings

- Enable ONNX export in the UI or `config.py`
- Adjust opset version if needed (9-15 supported)
- Models are saved in `output/models/`