Custom Linear Regression Implementation – Documentation

Author: Karan Sehgal

Overview

This project implements a custom linear regression model in Python without using machine learning libraries like scikit-learn. It provides a complete workflow from data import and preprocessing through model training to evaluation and visualization.

Requirements

The implementation requires the following Python libraries:

- NumPy: For numerical computations
- Pandas: For data manipulation and analysis
- Matplotlib: For visualization
- scikit-learn: Only for the StandardScaler and train_test_split utility functions

Implementation Details

Mathematical Foundation

The linear regression model is based on the equation:

```
y = Xw + b
```

Where:

- y is the target variable
- X is the feature matrix
- w is the weight vector
- b is the bias term

The model is trained using batch gradient descent to minimize the Mean Squared Error (MSE) cost function:

```
MSE = (1/2m) * Σ(y_pred - y)²
```

The weight update rules are:

```
w = w - α * (1/m) * X^T * (y_pred - y)
b = b - α * (1/m) * Σ(y_pred - y)
```

Where:

- α is the learning rate
- m is the number of samples

Class: LinearRegression

Constructor

`def __init__(self, learning_rate=0.01, n_iterations=1000)`

Parameters:

- `learning_rate`: Step size for gradient descent (default: 0.01)
- `n_iterations`: Maximum number of iterations (default: 1000)

Methods

`fit(X, y, verbose=True)`

Trains the model using gradient descent.

Parameters:

- `X`: Training features (numpy array)
- `y`: Target values (numpy array)
- `verbose`: Whether to print progress during training (default: True)

`predict(X)`

Makes predictions using the trained model.

Parameters:

- `X`: Input features (numpy array)

Returns:

- Predicted values (numpy array)

`evaluate(X_test, y_test)`

Evaluates the model using multiple metrics.

Parameters:

- `X_test`: Test features (numpy array)
- `y_test`: True target values (numpy array)

Returns:

- Dictionary containing the following evaluation metrics:
    - MSE: Mean Squared Error

- RMSE: Root Mean Squared Error
- MAE: Mean Absolute Error
- R^2: Coefficient of determination
- Adjusted R^2: R^2 adjusted for the number of features
- MAPE: Mean Absolute Percentage Error

`plot_cost_history()`

Visualizes the cost function over iterations.

`plot_predictions(X_test, y_test, feature_index=0, feature_name="Feature")`

Plots actual vs. predicted values against a specified feature.

Parameters:

- `X_test`: Test features (numpy array)
- `y_test`: True target values (numpy array)
- `feature_index`: Index of the feature to plot against (default: 0)
- `feature_name`: Name of the feature for the x-axis label (default: "Feature")

`plot_residuals(X_test, y_test, feature_index=0, feature_name="Feature")`

Generates residual plots for model diagnostics.

Parameters:

- `X_test`: Test features (numpy array)
- `y_test`: True target values (numpy array)
- `feature_index`: Index of the feature to plot against (default: 0)
- `feature_name`: Name of the feature for the x-axis label (default: "Feature")

Data Preparation Function

`prepare_data(df, target_column, test_size=0.2, random_state=42, normalize=True)`

Prepares data for linear regression training.

Parameters:

- `df`: Pandas DataFrame containing the dataset
- `target_column`: Name of the target column
- `test_size`: Proportion of data to use for testing (default: 0.2)
- `random_state`: Random seed for reproducibility (default: 42)

- `normalize`: Whether to normalize the features (default: True)

Returns:

- `X_train`: Training features
- `X_test`: Testing features
- `y_train`: Training targets
- `y_test`: Testing targets
- `scaler_X`: Feature scaler object (None if normalize=False)
- `scaler_y`: Target scaler object (None if normalize=False)

Main Execution Flow

The `main()` function provides a complete workflow:

1. Load dataset from CSV
2. Display dataset information and check for missing values
3. Prepare data for training
4. Create and train the linear regression model
5. Evaluate the model using multiple metrics
6. Print model parameters (weights and bias)
7. Generate visualizations (cost history, predictions, residuals)
8. Show sample predictions in the original scale (if normalized)

Usage Examples

Basic Usage

```
# Load dataset
df = pd.read_csv('housing_data.csv')

# Prepare data
X_train, X_test, y_train, y_test, scaler_X, scaler_y =
prepare_data(
    df, target_column='price', test_size=0.2,
normalize=True
)

# Create and train model
model = LinearRegression(learning_rate=0.01,
n_iterations=1000)
model.fit(X_train, y_train)

# Evaluate model
metrics = model.evaluate(X_test, y_test)
```

```
print(metrics)

# Make predictions
predictions = model.predict(X_test)
```

Custom Learning Parameters

```
# Create model with custom parameters
model = LinearRegression(learning_rate=0.05, n_iterations=2000)
model.fit(X_train, y_train, verbose=False)  # Turn off verbose output
```

Visualization Guide

The implementation provides three types of visualizations:

1. **Cost History Plot**: Shows how the cost function decreases over iterations. A steadily decreasing curve indicates proper convergence.
2. **Predictions Plot**: Compares actual vs. predicted values against a specific feature. Ideally, the predicted values (red) should closely follow the actual values (blue).
3. **Residual Analysis**:
    - **Residual Scatter Plot**: Shows the error distribution across a feature. Ideally, residuals should be randomly distributed around zero with no clear pattern.
    - **Residual Histogram**: Shows the error distribution. Ideally, this should be approximately normally distributed around zero.

Performance Considerations

- **Feature Scaling**: Normalize features for faster convergence
- **Learning Rate**: Choose carefully to prevent divergence (too high) or slow convergence (too low)
- **Iterations**: Ensure enough iterations for convergence
- **Performance on Large Datasets**: Consider using minibatch gradient descent for very large datasets
- **Multicollinearity**: Be cautious with highly correlated features