Linear Regression Implementation (Improved) Documentation

Author : Karan Sehgal

Overview

This implementation provides a comprehensive linear regression framework with enhanced features including regularization, mini-batch gradient descent, early stopping, cross-validation, and extensive model evaluation capabilities. The package consists of a main `LinearRegression` class and supporting functions for data preparation and hyperparameter tuning.

Key Features

- **Training Methods**: Batch, mini-batch, and stochastic gradient descent
- **Regularization**: L1 (Lasso) and L2 (Ridge) regularization options
- **Early Stopping**: Prevents overfitting by monitoring validation performance
- **Model Evaluation**: Multiple metrics including MSE, RMSE, MAE, R^2, and MAPE
- **Cross-Validation**: K-fold cross-validation for robust performance assessment
- **Visualization**: Learning curves, feature importance, correlation matrices, and residual analysis
- **Hyperparameter Tuning**: Parallel search over hyperparameter space
- **Data Preprocessing**: Handling missing values, categorical variables, and feature normalization
- **Model Persistence**: Save and load trained models

Core Components

1. LinearRegression Class

The main class implementing linear regression using gradient descent with various enhancements.

Initialization Parameters

- `learning_rate`: Step size for gradient descent (default: 0.01)
- `n_iterations`: Maximum number of training iterations (default: 1000)
- `batch_size`: Number of samples per batch; None for batch gradient descent (default: None)
- `early_stopping`: Whether to use early stopping (default: False)
- `patience`: Number of iterations to wait for improvement before stopping (default: 50)
- `tol`: Minimum improvement required (default: 1e-4)
- `l1_penalty`: L1 regularization strength (default: 0)
- `l2_penalty`: L2 regularization strength (default: 0)

Key Methods

Training & Prediction

- `fit(X, y, X_val=None, y_val=None, verbose=True, feature_names=None)`: Train the model
- `predict(X)`: Make predictions on new data

Evaluation

- `evaluate(X_test, y_test)`: Calculate performance metrics
- `cross_validate(X, y, k_folds=5, random_state=None)`: Perform k-fold cross-validation
- `get_feature_importance()`: Calculate importance of each feature
- `get_summary()`: Generate a summary of the model parameters

Visualization

- `plot_learning_curves()`: Plot training and validation loss curves
- `plot_feature_importance(top_n=None)`: Plot feature importance
- `plot_correlation_matrix(X, feature_names=None)`: Plot correlation matrix of features
- `plot_predictions(X_test, y_test, feature_index=0, feature_name=None)`: Plot actual vs predicted values
- `plot_residuals(X_test, y_test, feature_index=0, feature_name=None)`: Plot residual analysis

Model Persistence

- `save_model(filename)`: Save model to file
- `load_model(filename)`: Load model from file (class method)

Internal Methods

- `_initialize_weights(n_features)`: Initialize model weights using Xavier/Glorot initialization
- `_get_batch_indices(n_samples, batch_size, shuffle=True)`: Generate batch indices
- `_predict(X)`: Internal prediction function
- `_compute_cost(X, y)`: Compute cost function with regularization

2. Data Preparation Function

```
prepare_data(df, target_column, test_size=0.2, val_size=0.1,
random_state=42, normalize=True, handle_missing='mean',
categorical_encode='one-hot', drop_columns=None)
```

Prepares data for linear regression with various preprocessing options:

- Splits data into train, validation, and test sets
- Normalizes features if specified
- Handles missing values with various strategies

- Encodes categorical variables
- Drops specified columns

Parameters

- `df`: pandas DataFrame containing the dataset
- `target_column`: name of the target column
- `test_size`: proportion of data for testing
- `val_size`: proportion of training data for validation
- `random_state`: seed for reproducibility
- `normalize`: whether to normalize features
- `handle_missing`: strategy for missing values ('mean', 'median', 'mode', 'drop')
- `categorical_encode`: strategy for categorical variables ('one-hot', 'label')
- `drop_columns`: columns to exclude

Returns

- Training, validation, and test sets
- Feature names
- Scalers for features and target (if normalization is applied)

3. Hyperparameter Tuning Function

```
train_with_hyperparameter_search(X_train, y_train, X_val, y_val,
param_grid, verbose=True, n_jobs=-1)
```

Trains multiple models with different hyperparameter combinations to find the optimal configuration.

Parameters

- `X_train`, `y_train`: Training data
- `X_val`, `y_val`: Validation data
- `param_grid`: Dictionary of hyperparameter combinations to try
- `verbose`: Whether to print progress
- `n_jobs`: Number of parallel jobs for training

Returns

- Best model based on validation performance
- Results for all hyperparameter combinations tested

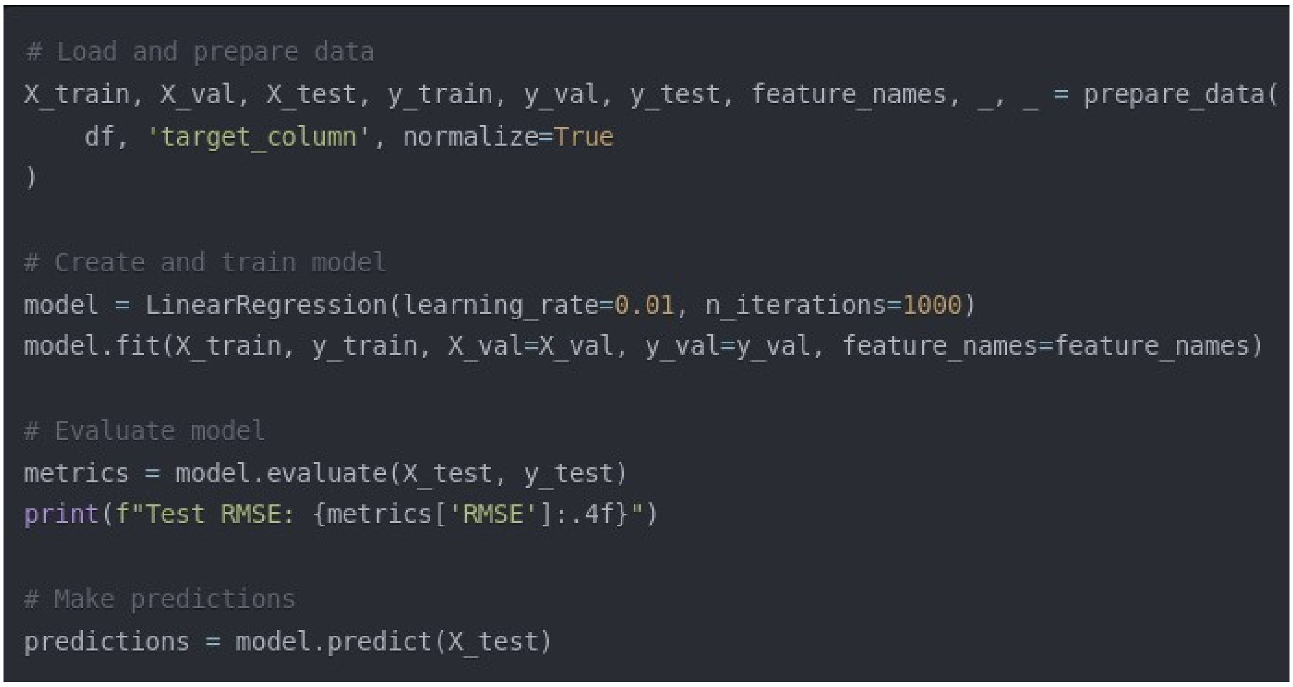
4. Main Function

The `main()` function provides a command-line interface for the entire workflow:

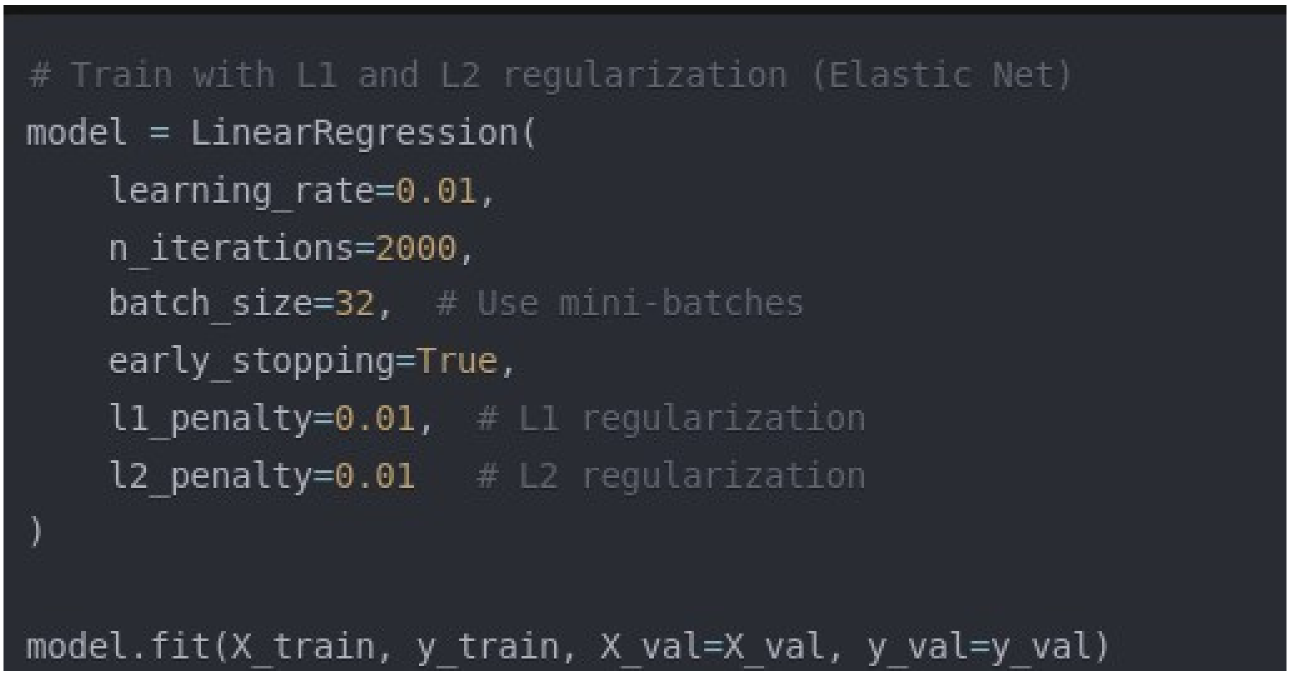
1. Data loading and inspection
2. Configuration selection (default or customized)
3. Data preparation
4. Model training (with or without hyperparameter tuning)
5. Model evaluation

6. Results visualization
7. Model saving

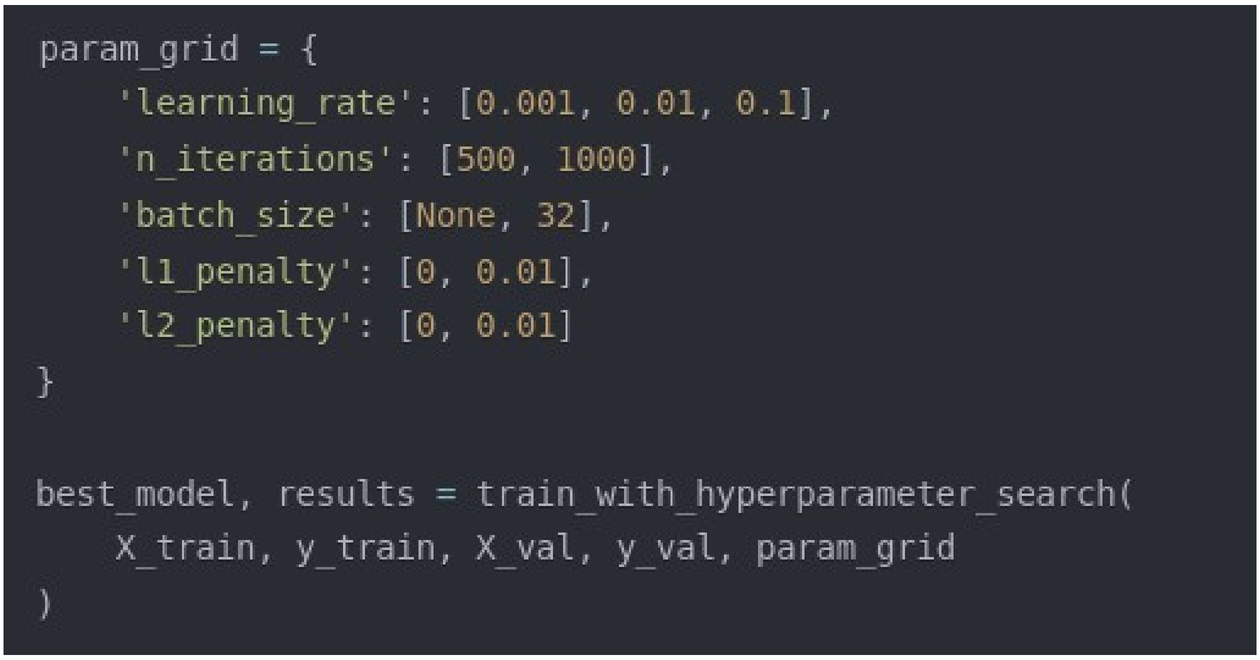
Basic Usage:



```python
# Load and prepare data
X_train, X_val, X_test, y_train, y_val, y_test, feature_names, _, _ = prepare_data(
    df, 'target_column', normalize=True
)

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

# Evaluate model
metrics = model.evaluate(X_test, y_test)
print(f"Test RMSE: {metrics['RMSE']:.4f}")

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



```python
# Train with L1 and L2 regularization (Elastic Net)
model = LinearRegression(
    learning_rate=0.01,
    n_iterations=2000,
    batch_size=32,  # Use mini-batches
    early_stopping=True,
    l1_penalty=0.01,  # L1 regularization
    l2_penalty=0.01   # L2 regularization
)

model.fit(X_train, y_train, X_val=X_val, y_val=y_val)
```



```python
param_grid = {
    'learning_rate': [0.001, 0.01, 0.1],
    'n_iterations': [500, 1000],
    'batch_size': [None, 32],
    'l1_penalty': [0, 0.01],
    'l2_penalty': [0, 0.01]
}

best_model, results = train_with_hyperparameter_search(
    X_train, y_train, X_val, y_val, param_grid
)
```

Visualization Examples

The implementation provides various visualization functions:

- **Learning Curves**: `model.plot_learning_curves()`
- **Feature Importance**: `model.plot_feature_importance()`
- **Correlation Matrix**: `model.plot_correlation_matrix(X_train, feature_names)`
- **Predictions vs Actual**: `model.plot_predictions(X_test, y_test, feature_index=0)`
- **Residual Analysis**: `model.plot_residuals(X_test, y_test, feature_index=0)`

Dependencies

- NumPy: For numerical operations
- Pandas: For data manipulation
- Matplotlib: For visualization
- Seaborn: For enhanced visualizations
- Scikit-learn: For data splitting and preprocessing
- tqdm: For progress bars
- concurrent.futures: For parallel hyperparameter tuning

Performance Considerations

- For large datasets, consider using mini-batch gradient descent by setting `batch_size`
- Early stopping can significantly reduce training time while preventing overfitting

- Hyperparameter tuning uses parallel processing for efficiency but can be computationally intensive
- Feature normalization is recommended for better convergence and model stability

Technical Notes

1. **Xavier/Glorot Initialization**: Weights are initialized using a uniform distribution with variance scaled by input and output dimensions to improve convergence.
2. **Regularization**:
   - L1 (Lasso): Encourages sparse models by pushing weights to exactly zero
   - L2 (Ridge): Prevents overfitting by penalizing large weights
3. **Early Stopping**: Implementation monitors validation performance and stops training when no improvement is seen for a specified number of iterations.
4. **Gradient Descent Variants**:
   - Batch GD: Uses all training examples (batch_size=None)
   - Mini-batch GD: Uses subsets of data (typical batch_size: 32, 64, 128)
   - Stochastic GD: Uses single examples (batch_size=1)
5. **Evaluation Metrics**:
   - MSE (Mean Squared Error): Average squared difference between predictions and actual values
   - RMSE (Root Mean Squared Error): Square root of MSE, in the same units as the target
   - MAE (Mean Absolute Error): Average absolute difference between predictions and actual values
   - R^2 (Coefficient of Determination): Proportion of variance explained by the model
   - Adjusted R^2: R^2 adjusted for the number of predictors
   - MAPE (Mean Absolute Percentage Error): Percentage version of MAE

This implementation provides a comprehensive and flexible framework for linear regression that can be used for a wide variety of regression tasks with different data characteristics and requirements.

Concept Behind Improvements:

Linear regression is a fundamental machine learning algorithm used for predicting continuous values. However, standard linear regression can lead to overfitting when dealing with complex datasets. To address this issue, Ridge and Lasso regression introduce regularization techniques that help control the complexity of the model.

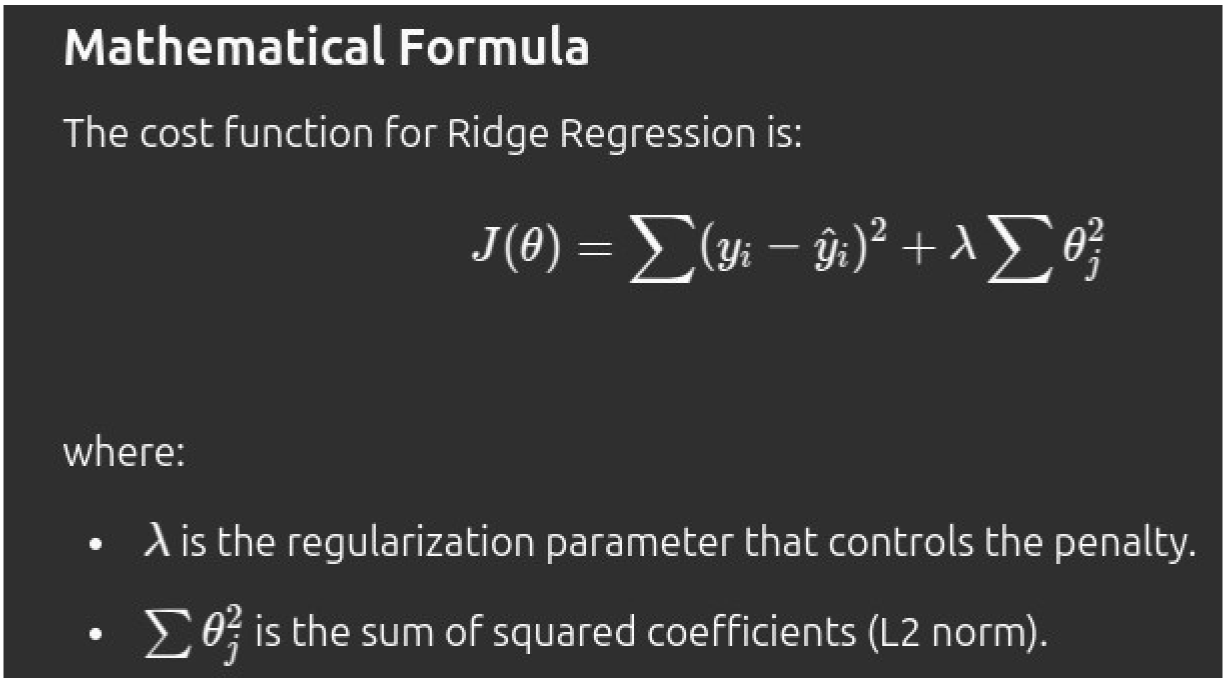
1. Ridge Regression (L2 Regularization)

Concept

Ridge regression applies an L2 penalty, which adds the sum of squared coefficients to the cost function. This reduces the magnitude of coefficients but does not set them to zero, ensuring that all features contribute to the model.

Mathematical Formula

The cost function for Ridge Regression is:



Mathematical Formula

The cost function for Ridge Regression is:

$$J(\theta) = \sum (y_i - \hat{y}_i)^2 + \lambda \sum \theta_j^2$$

where:

- λ is the regularization parameter that controls the penalty.
- $\sum \theta_j^2$ is the sum of squared coefficients (L2 norm).

Effects of Ridge Regression

- Prevents overfitting by shrinking large coefficients.
- Retains all features, but reduces their impact.
- Suitable for datasets with highly correlated features.

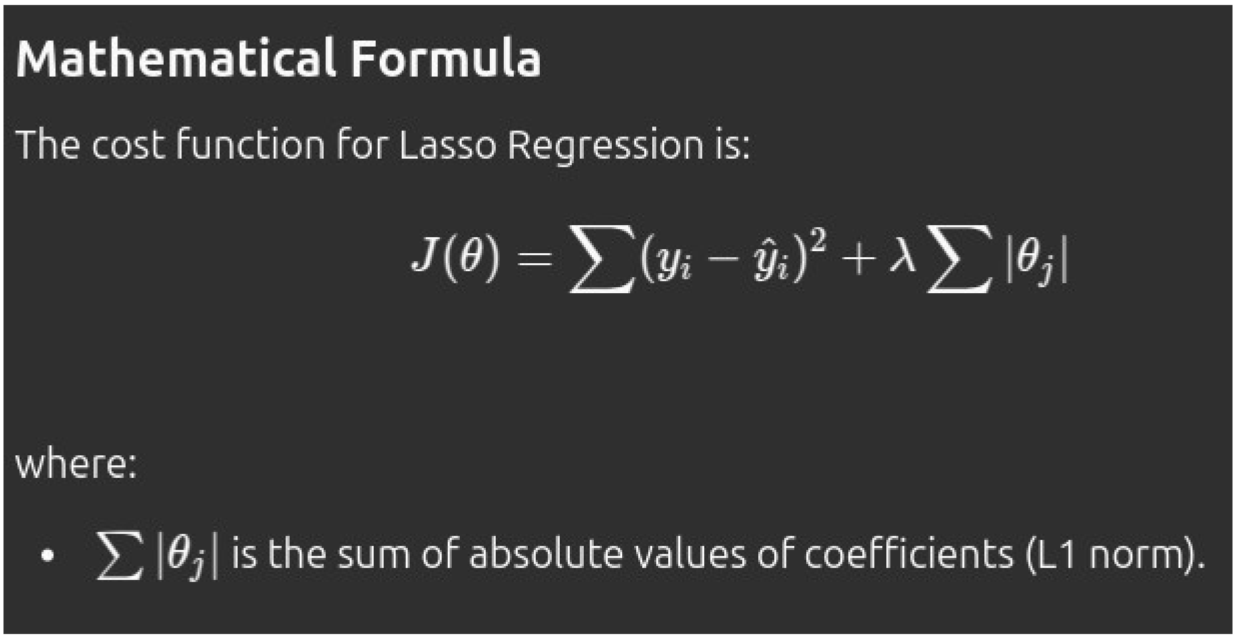
2. Lasso Regression (L1 Regularization)

Concept

Lasso regression applies an L1 penalty, which adds the sum of absolute values of coefficients to the cost function. Unlike Ridge regression, Lasso can shrink some coefficients to exactly zero, effectively performing feature selection.

Mathematical Formula

The cost function for Lasso Regression is:



Mathematical Formula

The cost function for Lasso Regression is:

$$J(\theta) = \sum (y_i - \hat{y}_i)^2 + \lambda \sum |\theta_j|$$

where:

- $\sum |\theta_j|$ is the sum of absolute values of coefficients (L1 norm).

Effects of Lasso Regression

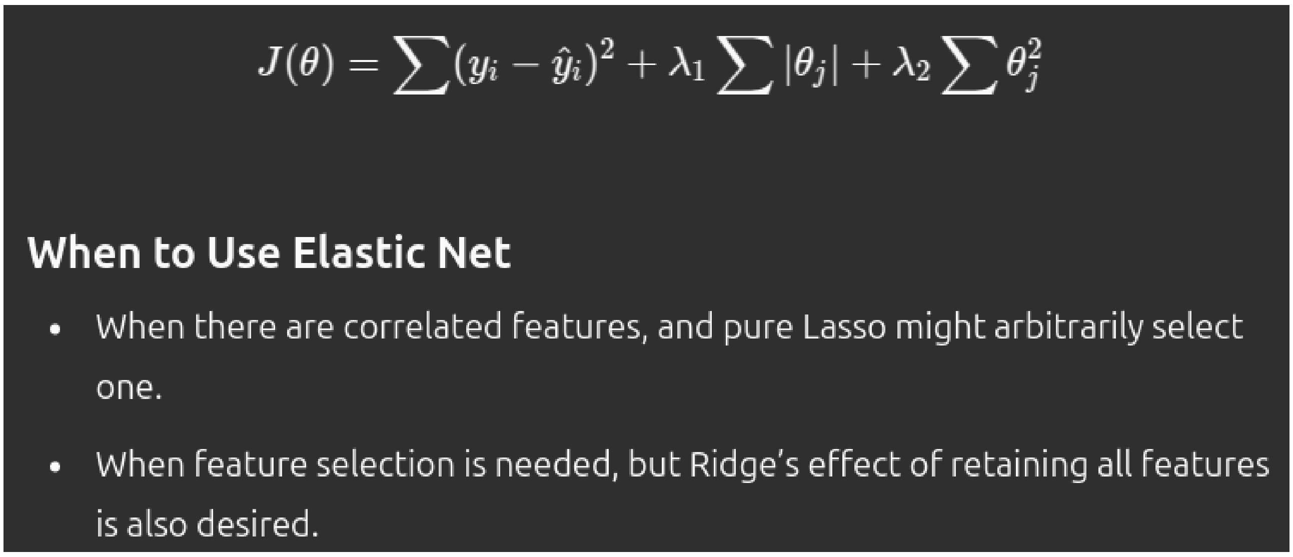
- Reduces the magnitude of coefficients.
- Some coefficients become exactly zero, effectively removing irrelevant features.
- Useful when feature selection is desired.

3. Key Differences Between Ridge and Lasso Regression

Feature	Ridge Regression	Lasso Regression
Regularization type	L2 (sum of squared coefficients)	L1 (sum of absolute coefficients)
Shrinks coefficients	Yes	Yes
Can set coefficients to 0?	No	Yes (performs feature selection)
Suitable for	Multicollinear data	Sparse data (many irrelevant features)

4. Elastic Net: Hybrid of Ridge & Lasso

Elastic Net combines both L1 (Lasso) and L2 (Ridge) penalties to balance shrinkage and feature selection:



$$J(\theta) = \sum (y_i - \hat{y}_i)^2 + \lambda_1 \sum |\theta_j| + \lambda_2 \sum \theta_j^2$$

When to Use Elastic Net

- When there are correlated features, and pure Lasso might arbitrarily select one.
- When feature selection is needed, but Ridge's effect of retaining all features is also desired.

Regularization techniques like Ridge and Lasso regression help improve the generalization of linear models by controlling complexity and reducing overfitting. Ridge is useful for maintaining all features with reduced impact, while Lasso is best suited for sparse models where feature selection is necessary. Elastic Net provides a balanced approach by combining both techniques.

By understanding these concepts, I built a more robust regression models that generalize well on unseen data.