TorchML Foundations: Linear Models

Omkar Kottawar

August 2025

Overview

Linear models are a foundational family of machine learning algorithms that model relationships between input features and target variables using linear functions, possibly with transformations or regularization. They are widely used for regression and classification tasks due to their simplicity, interpretability, and computational efficiency. This document describes four linear models implemented in the *TorchML Foundations* project: Linear Regression, Logistic Regression, Ridge Regression, and Lasso Regression. Each model is analyzed through eight key questions to provide a comprehensive understanding of its purpose, mechanics, and implementation.

1 The Linear Model Family

The linear model family shares a common foundation of modeling relationships between independent variables $\mathbf{x} = [x_1, x_2, \dots, x_n]$ and a dependent variable y using a linear combination of features, but they differ in their objectives, loss functions, and regularization approaches. Below, we address the key aspects of the linear model family, with specific details for each model provided in subsections.

1.1 What is the Model's Goal?

The primary goal of models in the linear model family is to capture the relationship between independent variables and a dependent variable by fitting a linear function of the form $\mathbf{x}^T\beta + \beta_0$, where $\beta = [\beta_1, \dots, \beta_n]$ are coefficients and β_0 is the intercept. The models aim to estimate these coefficients to make accurate predictions or classifications while balancing model fit and complexity. The specific output depends on the model: Linear and Ridge/Lasso Regression predict continuous outcomes, while Logistic Regression predicts probabilities for binary or categorical outcomes.

- **Linear Regression:** Aims to predict a continuous dependent variable y by minimizing the difference between observed and predicted values: $y = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n + \epsilon$, where ϵ is the error term.
- **Logistic Regression:** Predicts the probability of a binary outcome $P(y = 1|\mathbf{x})$ using the logistic function: $P(y = 1|\mathbf{x}) = \sigma(\mathbf{x}^T\beta) = \frac{1}{1+e^{-\mathbf{x}^T\beta}}$, enabling classification based on a threshold.
- **Ridge Regression:** Extends linear regression by adding an L2 penalty to shrink coefficients, improving stability and reducing overfitting, especially in the presence of multicollinearity.

- **Lasso Regression:** Extends linear regression with an L1 penalty to shrink coefficients and perform feature selection by setting some coefficients to zero, enhancing model sparsity.

1.2 What is the Loss or Objective Function?

Linear models optimize a loss function that measures the discrepancy between predicted and actual values, often augmented with regularization for Ridge and Lasso. The choice of loss depends on the model's output: squared error for continuous outcomes (Linear, Ridge, Lasso) and log-loss for probabilistic outputs (Logistic). The general form of the loss function is:

$$J(\beta) = \text{Data Fit Term} + \lambda \cdot \text{Regularization Term},$$

where the data fit term quantifies prediction error, and the regularization term (if present) controls model complexity.

- **Linear Regression:** Uses Mean Squared Error (MSE):

$$J(\beta) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 = \frac{1}{m}(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta).$$

- **Logistic Regression:** Uses log-loss (binary cross-entropy):

$$J(\beta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log(\hat{p}_i) + (1 - y_i)\log(1 - \hat{p}_i)\right].$$

- **Ridge Regression:** Augments MSE with an L2 penalty:

$$J(\beta) = \frac{1}{m}(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\sum_{j=1}^{n}\beta_j^2.$$

- **Lasso Regression:** Augments MSE with an L1 penalty:

$$J(\beta) = \frac{1}{m}(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\sum_{j=1}^{n}|\beta_j|.$$

1.3 How is the Model Optimized?

Linear models are optimized by minimizing their respective loss functions. Optimization methods include closed-form solutions (when feasible) and iterative techniques like gradient descent.

- **Linear Regression:** Uses the closed-form solution (normal equation) or gradient descent with the gradient:

$$\frac{\partial J(\beta)}{\partial \beta} = -\frac{2}{m}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta).$$

- **Logistic Regression:** Relies on iterative methods like gradient descent. The gradient is:

$$\frac{\partial J(\beta)}{\partial \beta} = \frac{1}{m}\mathbf{X}^T(\hat{\mathbf{p}} - \mathbf{y}).$$

- **Ridge Regression:** Has a closed-form solution and can also use gradient descent. The gradient is:

$$\frac{\partial J(\beta)}{\partial \beta} = -\frac{2}{m}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) + 2\lambda\beta.$$

- **Lasso Regression:** Uses iterative methods like coordinate descent or proximal gradient descent due to the non-differentiable L1 penalty. The latter involves a standard gradient descent step on the MSE term followed by a soft-thresholding step.

1.4 How is the Model Implemented in this Project?

In this project, Linear, Logistic, Ridge, and Lasso Regression are implemented from scratch using PyTorch. The implementations primarily leverage tensor operations and PyTorch's autograd for gradient-based optimization. A notable exception is Lasso Regression, which requires a specialized proximal gradient descent algorithm to handle the non-differentiable L1 penalty. Each model inherits from a base class (`BaseRegressor` or `BaseClassifier`), ensuring consistent interfaces.

1.4.1 Linear Regression

Implemented in `linear_regression.py` with gradient descent to minimize MSE. The gradient descent updates use PyTorch's autograd to compute gradients of the MSE loss.

```python
# From linear_regression.py (Standardized with autograd)
def fit(self, X, y):
    X = self._to_tensor(X)
    y = self._to_tensor(y, reshape_y=True)
    n_samples, n_features = X.shape

    self.weights = torch.randn(n_features, 1, requires_grad=True)
    self.bias = torch.zeros(1, requires_grad=True)

    for epoch in range(self.max_iter):
        y_pred = X @ self.weights + self.bias
        loss = torch.mean((y_pred - y) ** 2)
        loss.backward()

        with torch.no_grad():
            self.weights -= self.learning_rate * self.weights.grad
            self.bias -= self.learning_rate * self.bias.grad
            self.weights.grad.zero_()
            self.bias.grad.zero_()
```

1.4.2 Logistic Regression

Implemented in `logistic_regression.py` with gradient descent and autograd to minimize log-loss (cross-entropy). It uses a sigmoid function with clipping for numerical stability.

```python
# From logistic_regression.py (Standardized with autograd)
def fit(self, X, y):
    X = self._to_tensor(X)
    y = self._to_tensor(y, reshape_y=True)

    self.weights = torch.randn(n_features, 1, requires_grad=True)
    self.bias = torch.zeros(1, requires_grad=True)

    for epoch in range(self.max_iter):
        z = X @ self.weights + self.bias
        y_pred_proba = self._sigmoid(z)

        epsilon = 1e-15
        y_pred_clipped = torch.clamp(y_pred_proba, epsilon, 1 - epsilon)
        loss = -torch.mean(y * torch.log(y_pred_clipped) + (1 - y) * torch.log(1 - y_pred_clipped))
        loss.backward()

        with torch.no_grad():
            self.weights -= self.learning_rate * self.weights.grad
            self.bias -= self.learning_rate * self.bias.grad
            self.weights.grad.zero_()
            self.bias.grad.zero_()
```

1.4.3 Ridge Regression

Implemented in `ridge_regression.py` with gradient descent to minimize MSE plus an L2 penalty. The implementation uses autograd to automatically compute the gradients for the combined loss term.

```python
# From ridge_regression.py (Standardized with autograd)
def fit(self, X, y):
    X = self._to_tensor(X)
    y = self._to_tensor(y, reshape_y=True)
    n_samples, n_features = X.shape

    self.weights = torch.randn(n_features, 1, requires_grad=True)
    self.bias = torch.zeros(1, requires_grad=True)

    for epoch in range(self.max_iter):
        y_pred = X @ self.weights + self.bias
        mse_loss = torch.mean((y_pred - y) ** 2)
        l2_penalty = self.alpha * torch.sum(self.weights ** 2)
        total_loss = mse_loss + l2_penalty
        total_loss.backward()

        with torch.no_grad():
            self.weights -= self.learning_rate * self.weights.grad
            self.bias -= self.learning_rate * self.bias.grad
            self.weights.grad.zero_()
            self.bias.grad.zero_()
```

1.4.4 Lasso Regression

Implemented in `lasso_regression.py` using proximal gradient descent with a soft-thresholding operator to handle the L1 penalty. The algorithm alternates between a gradient descent step on the MSE loss and applying soft-thresholding to enforce sparsity.

```python
# From lasso_regression.py (Refined)
def fit(self, X, y):
    X = self._to_tensor(X)
    y = self._to_tensor(y, reshape_y=True)
    n_samples, n_features = X.shape

    self.weights = torch.randn(n_features, 1) * 0.01
    self.bias = torch.zeros(1)

    for epoch in range(self.max_iter):
        y_pred = X @ self.weights + self.bias

        residual = y_pred - y
        grad_w = (2 / n_samples) * X.T @ residual
        grad_b = (2 / n_samples) * torch.sum(residual)

        self.weights -= self.learning_rate * grad_w
        self.bias -= self.learning_rate * grad_b

        threshold = self.alpha * self.learning_rate
        self.weights = self._soft_thresholding(self.weights, threshold)
```

Comparison of Linear Models

Model	Loss Function	Regularization	Strengths	Weaknesses
Linear	MSE	None	Simple, interpretable	Assumes linearity
Logistic	Cross-Entropy	Optional	Probabilistic outputs	Limited to binary classes
Ridge	MSE + L2	L2	Reduces overfitting	Less interpretable weights
Lasso	MSE + L1	L1	Feature selection	Subgradient descent complexity

Table 1: Comparison of linear models based on key characteristics.

A Mathematical Notation

Symbol	Description	Context
x	Input feature vector	$x = [x_1, x_2, \ldots, x_n]$
x_i	Feature vector for sample i	Training data
x_{ij}	j-th feature of sample i	Individual feature value
y	Target variable/vector	Dependent variable
y_i	Target value for sample i	Ground truth
$\hat{y}$	Predicted value	Model output
$\hat{y}_i$	Prediction for sample i	$\hat{y}_i = x_i^T \beta + \beta_0$

Symbol	**Description**	**Context**
$\hat{p}_i$	Predicted probability	$\hat{p}_i = \sigma(x_i^T \beta)$
β	Coefficient vector	$\beta = [\beta_1, \ldots, \beta_n]$
β_j	j-th coefficient	Weight for feature j
β_0	Intercept term	Bias parameter
w	Weight vector	Alternative to β
b	Bias term	Alternative to β_0
m	Number of samples	Training set size
n	Number of features	Dimensionality
X	Design matrix	Size $m \times n$
X^T	Matrix transpose	Size $n \times m$
$(X^T X)^{-1}$	Matrix inverse	Normal equation
I	Identity matrix	Regularization
$\sigma(z)$	Sigmoid function	$\frac{1}{1+e^{-z}}$
$\log(\cdot)$	Natural logarithm	Log-loss
$\text{sign}(\cdot)$	Sign function	Soft thresholding
$\lvert \cdot \rvert$	Absolute value	L1 penalty
$P(y = 1 \mid x)$	Conditional probability	Logistic regression
$J(\beta)$	Loss function	Objective to minimize
λ, α	Regularization parameter	Penalty strength
$\frac{\partial J}{\partial \beta}$	Gradient	Optimization direction
ϵ	Error term	$y = X\beta + \epsilon$
$\sum_{j=1}^{n} \lvert \beta_j \rvert$	L1 penalty	Lasso regularization
$\sum_{j=1}^{n} \beta_j^2$	L2 penalty	Ridge regularization