# 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 $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ 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 $\beta = [\beta_1, \ldots, \beta_n]$ are coefficients and $\beta_0$ 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 $\epsilon$ 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):

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

  where $\hat{y}_i = \mathbf{x}_i^T\beta + \beta_0$, $\mathbf{y}$ is the target vector, and $\mathbf{X}$ is the design matrix.

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

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

  where $\hat{p}_i = \sigma(\mathbf{x}_i^T\beta)$ is the predicted probability, and $y_i \in \{0, 1\}$.

- **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. Regularized models (Ridge, Lasso) require specialized approaches due to their penalty terms, and the choice of method depends on the loss function's properties and dataset size.

- **Linear Regression:** Uses the closed-form solution (normal equation):

$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y},$$

  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 or Newton-Raphson, as the log-loss is non-linear. The gradient is:

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

where $\hat{\mathbf{p}}$ is the vector of predicted probabilities.

- **Ridge Regression:** Has a closed-form solution:

$$\beta = (\mathbf{X}^T\mathbf{X} + m\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y},$$

where $\mathbf{I}$ excludes the intercept from regularization. Gradient descent is also used, with the gradient:

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

- **Lasso Regression:** Uses coordinate descent or proximal gradient methods (e.g., LARS) due to the non-differentiable L1 penalty. The subgradient includes:

$$\frac{\partial J(\beta)}{\partial \beta_j} = -\frac{2}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)x_{ij} + \lambda \cdot \text{sign}(\beta_j).$$

### 1.4 What Assumptions Does the Model Make About the Data?

Linear models generally assume a linear relationship between the features and the outcome (or log-odds for Logistic Regression), independence of observations, and minimal influence from outliers. Regularized models relax some assumptions, such as multicollinearity.

- **Linear Regression:** Assumes linearity, independence, homoscedasticity, normality of errors, no multicollinearity, and no significant outliers.
- **Logistic Regression:** Assumes linearity in the log-odds, independence, no multicollinearity, and a binary outcome. It does not require normality or homoscedasticity.
- **Ridge Regression:** Inherits linear regression assumptions but is robust to multicollinearity due to the L2 penalty.
- **Lasso Regression:** Similar to Ridge, but the L1 penalty further mitigates multicollinearity by selecting one feature among correlated ones.

### 1.5 How is the Model Evaluated?

Linear models are evaluated using metrics that assess predictive or classification performance, model fit, and generalization. Continuous outcome models (Linear, Ridge, Lasso) use error-based metrics, while Logistic Regression uses classification metrics. Cross-validation is common across all models to assess generalization.

- **Linear Regression:** Evaluated with MSE, RMSE, $R^2$, adjusted $R^2$, residual analysis, and cross-validation.
- **Logistic Regression:** Uses log-loss, accuracy, precision, recall, F1-score, ROC curve, AUC, and cross-validation.
- **Ridge Regression:** Same as linear regression, with emphasis on cross-validation to tune $\lambda$.
- **Lasso Regression:** Same as Ridge, with additional evaluation of sparsity (number of non-zero coefficients).

### 1.6 Does the Model Use Regularization or Constraints?

Regularization is a key feature of Ridge and Lasso Regression, but not standard Linear or Logistic Regression. Regularization adds a penalty to the loss function to control model complexity.

- **Linear Regression:** No regularization in its standard form.
- **Logistic Regression:** No regularization in its standard form, but Ridge or Lasso penalties can be added (e.g., $\lambda \sum \beta_j^2$ or $\lambda \sum |\beta_j|$).
- **Ridge Regression:** Uses L2 regularization: $\lambda \sum_{j=1}^{n} \beta_j^2$, shrinking coefficients to reduce variance.
- **Lasso Regression:** Uses L1 regularization: $\lambda \sum_{j=1}^{n} |\beta_j|$, promoting sparsity and feature selection.

### 1.7 What are the Model's Strengths and Weaknesses?

**Strengths (General):**

- **Interpretability:** Coefficients directly indicate feature importance or effect on the outcome.
- **Simplicity:** Easy to implement and computationally efficient for many datasets.
- **Foundation for Advanced Models:** Serve as building blocks for more complex algorithms.

**Weaknesses (General):**

- **Linearity Assumption:** Limited to linear relationships unless features are transformed.
- **Sensitivity to Violations:** Performance degrades if assumptions (e.g., independence, no multicollinearity) are violated.

**Linear Regression:**

- **Strengths:** Highly interpretable, efficient for small datasets, performs well when assumptions are met.
- **Weaknesses:** Sensitive to outliers, multicollinearity, and non-linear relationships.

**Logistic Regression:**

- **Strengths:** Provides probabilistic outputs, robust to small assumption violations, extensible to multiclass problems.
- **Weaknesses:** Limited to binary outcomes (in standard form), struggles with imbalanced data, assumes linearity in log-odds.

**Ridge Regression:**

- **Strengths:** Handles multicollinearity, prevents overfitting, stable coefficient estimates.
- **Weaknesses:** Does not perform feature selection, still sensitive to outliers, requires tuning $\lambda$.

**Lasso Regression:**

- **Strengths:** Performs feature selection, handles multicollinearity, improves interpretability through sparsity.

- **Weaknesses:** May arbitrarily select one feature among correlated ones, sensitive to outliers, requires tuning $\lambda$.

## 1.8 How is the Model Implemented in this Project?

In this project, Linear, Logistic, Ridge, and Lasso Regression are implemented from scratch using PyTorch, leveraging tensor operations and autograd for gradient-based optimization. Each model inherits from a base class (`BaseRegressor` or `BaseClassifier`), ensuring consistent interfaces for fitting, predicting, and retrieving parameters. The implementations use gradient descent (or proximal gradient descent for Lasso) with configurable hyperparameters (`learning_rate`, `max_iter`, `tol`, and `alpha` for Ridge and Lasso). Key features include:

- Input conversion to PyTorch tensors.
- Weight and bias initialization with small random values or zeros.
- Loss computation with numerical stability (e.g., clipping in Logistic Regression).
- Convergence checking based on loss difference.
- Sparsity tracking for Lasso.

### 1.8.1 Linear Regression

Implemented in `linear_regression.py` with gradient descent to minimize Mean Squared Error (MSE). Weights are initialized randomly, and predictions are computed as $\mathbf{X}\mathbf{w}+b$. The gradient descent updates use PyTorch's autograd to compute gradients of the MSE loss.

```
# From linear_regression.py
def fit(self, X, y):
    X = torch.FloatTensor(X)
    y = torch.FloatTensor(y)
    n_samples, n_features = X.shape
    self.weights = torch.randn(n_features, requires_grad=True)
    self.bias = torch.randn(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_()
        if epoch > 0 and abs(self.loss_history[-2] - self.loss_history[-1]) < self.tol:
            break
```

### 1.8.2 Logistic Regression

Implemented in `logistic_regression.py` with gradient descent to minimize log-loss (cross-entropy). It uses a sigmoid function with clipping ($\pm 250$) for numerical stability and an epsilon ($10^{-15}$) to prevent $\log(0)$ errors. Weights are initialized with small random values, and the bias is initialized to zero.

```
# From logistic_regression.py
def sigmoid(self, z):
    z = torch.clamp(z, -250, 250)
    return 1 / (1 + torch.exp(-z))

def fit(self, X, y):
    X = torch.FloatTensor(X)
    y = torch.FloatTensor(y)
    n_samples, n_features = X.shape
    self.weights = torch.randn(n_features, requires_grad=True) * 0.01
    self.bias = torch.zeros(1, requires_grad=True)
    for epoch in range(self.max_iter):
        z = X @ self.weights + self.bias
        y_pred = self.sigmoid(z)
        y_pred = torch.clamp(y_pred, 1e-15, 1 - 1e-15)
        loss = -torch.mean(y * torch.log(y_pred) + (1 - y) * torch.log(1 - y_pred))
        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_()
        if epoch > 0 and abs(self.loss_history[-2] - self.loss_history[-1]) < self.tol:
            break
```

### 1.8.3 Ridge Regression

Implemented in `ridge_regression.py` with gradient descent to minimize MSE plus an L2 penalty. The L2 term ($\alpha \sum w_j^2$) is included in the loss, and gradients are computed directly, incorporating the penalty term $2\alpha\mathbf{w}$. Weights are initialized with small random values, and the bias is initialized to zero.

```
# From ridge_regression.py
def fit(self, X, y):
    X = torch.FloatTensor(X)
    y = torch.FloatTensor(y).reshape(-1, 1)
    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
        mse_loss = torch.mean((y_pred - y) ** 2)
        l2_penalty = self.alpha * torch.sum(self.weights ** 2)
        total_loss = mse_loss + l2_penalty
        residual = y_pred - y
        grad_w = 2 / n_samples * X.T @ residual + 2 * self.alpha * self.weights
        grad_b = 2 / n_samples * torch.sum(residual)
        self.weights -= self.learning_rate * grad_w
        self.bias -= self.learning_rate * grad_b
        if epoch > 0 and abs(self.loss_history[-2] - self.loss_history[-1]) < self.tol:
            break
```

### 1.8.4 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 gradient descent on the MSE loss and applying soft-thresholding to enforce sparsity. Weights are initialized with small random values, and the bias is initialized to zero.

```
# From lasso_regression.py
def soft_thresholding(self, w, lmbd):
    return torch.sign(w) * torch.maximum(torch.abs(w) - lmbd, torch.zeros_like(w))

def fit(self, X, y):
    X = torch.FloatTensor(X)
    y = torch.FloatTensor(y).reshape(-1, 1)
    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
        mse_loss = torch.mean((y_pred - y) ** 2)
        l1_penalty = self.alpha * torch.sum(torch.abs(self.weights))
        total_loss = mse_loss + l1_penalty
        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)
        if epoch > 0 and abs(self.loss_history[-2] - self.loss_history[-1]) < self.tol:
            break
```

## 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 $i$ | $\hat{y}_i = x_i^T \beta + \beta_0$ |
| $\hat{p}_i$ | Predicted probability | $\hat{p}_i = \sigma(x_i^T \beta)$ |
| $\beta$ | Coefficient vector | $\beta = [\beta_1, \ldots, \beta_n]$ |
| $\beta_j$ | $j$-th coefficient | Weight for feature $j$ |
| $\beta_0$ | Intercept term | Bias parameter |
| $w$ | Weight vector | Alternative to $\beta$ |
| $b$ | Bias term | Alternative to $\beta_0$ |
| $m$ | 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 |
| $P(y = 1\|x)$ | Conditional probability | Logistic regression |
| $J(\beta)$ | Loss function | Objective to minimize |
| $\lambda, \alpha$ | Regularization parameter | Penalty strength |
| $\frac{\partial J}{\partial \beta}$ | Gradient | Optimization direction |
| $\epsilon$ | 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 |