Regularization in Machine Learning

Regularization is a technique used to prevent overfitting by adding a penalty term to the loss function, discouraging excessive model complexity. It helps improve generalization by reducing variance at the cost of introducing some bias. The most common types of regularization are **L1 (Lasso), L2 (Ridge), and Elastic Net**, but other forms exist.

Regularization Type	When to Use	When to Avoid	Algorithms That Use It	Algorithms That Typically Don't Use It
L1 (Lasso)	When feature selection is needed; When you expect many irrelevant features; When sparsity is beneficial	When important features might get removed; When high sparsity is not desired	Lasso Regression, Logistic Regression (L1), Some Neural Networks	Decision Trees, kNN, Naive Bayes
L2 (Ridge)	When dealing with multicollinearity; When you want all features to contribute but with smaller weights	When feature selection is needed; When strict sparsity is desired	Ridge Regression, Logistic Regression (L2), Support Vector Machines (SVM)	Decision Trees, kNN, Naive Bayes
Elastic Net	When features are correlated; When both feature selection and shrinkage are needed	When tuning two hyperparameters is complex; When either L1 or L2 alone is sufficient	Elastic Net Regression, Some deep learning models	Decision Trees, kNN, Naive Bayes
Dropout	When training deep learning models; When preventing co-adaptation of neurons	When training time is a concern; When network capacity is already small	Neural Networks (e.g., CNNs, RNNs, Transformers)	Non-neural network models
No Regularization	When data is large and overfitting is not an issue; When using simple models that generalize well	When model complexity is too high; When data is limited and prone to overfitting	Decision Trees, k-Nearest Neighbors (kNN), Random Forests, Naive Bayes	Most simple models without high-dimensional features

1. Intuitive Understanding of Regularization

Imagine you're fitting a model to data. If the model is too complex (e.g., too many parameters, high-degree polynomials), it will **overfit**—meaning it learns noise instead of the true underlying pattern. Regularization introduces a **penalty** to the model's parameters, forcing it to prefer simpler models.

- **L1 (Lasso) Regularization**: Encourages sparsity by driving some coefficients to zero.
- **L2 (Ridge) Regularization**: Shrinks coefficients but does not set them to zero.
- **Elastic Net**: Combines L1 and L2, useful when features are correlated.
- **Dropout** (for neural networks): Randomly drops connections to prevent overfitting.

2. When Regularization is Used and When It Should Be Avoided

When to Use Regularization

✓ When the model has **many features** (high-dimensional data).

✓ When the model is **overfitting** (i.e., low training error but high test error).

✓ When the dataset has **multicollinearity** (correlated features).

✓ When feature selection is needed (L1 helps remove irrelevant features).

When Not to Use Regularization

✘ If the model is **underfitting**, adding regularization may make things worse.

✘ If the dataset is **very small**, excessive regularization may prevent learning.

✘ If **interpretability** is key, L2 may be preferable over L1, as L1 removes some features.

3. When Regularization Fails and How to Avoid It

Regularization can **fail** if:

- **Too much regularization** (high λ in L1/L2) causes severe underfitting.
- **Not enough regularization** fails to prevent overfitting.
- **Improper feature scaling**—L1 and L2 are sensitive to feature magnitudes.
- **L1 regularization alone** in cases where correlated features exist (Elastic Net helps).

How to Avoid Issues

- **Tune λ (regularization strength) using cross-validation.**
- **Scale features (standardization)** before applying regularization.
- **Use Elastic Net** when features are correlated.

---

4. Advantages and Disadvantages

Regularization Type	Advantages	Disadvantages
L1 (Lasso)	Feature selection, sparse models (some weights become 0)	May discard important features if λ is too high
L2 (Ridge)	Prevents overfitting by keeping all features, good for collinearity	Does not perform feature selection (all weights shrink but stay nonzero)
Elastic Net	Balances feature selection and regularization, works well with correlated features	Requires tuning two hyperparameters (α and λ)
Dropout (NNs)	Prevents overfitting in deep learning	Can slow down training

5. Step-by-Step Pseudo Code

Here's how regularization is applied in a regression setting:

python

Copy Edit

```
Initialize model parameters (weights w)
Set regularization parameter λ

for each training iteration:
    Compute predictions ŷ = Xw
    Compute loss = MSE(y, ŷ) + λ * regularization_term(w)

    if L1 regularization (Lasso):
        regularization_term(w) = sum(abs(w))
    elif L2 regularization (Ridge):
        regularization_term(w) = sum(w^2)
    elif Elastic Net:
        regularization_term(w) = α * sum(abs(w)) + (1 - α) * sum(w^2)

    Compute gradients
    Update weights using gradient descent

Return optimized weights
```

6. Full Mathematical Breakdown

Let's assume we have a linear regression model:

$$\hat{y} = w_1 x_1 + w_2 x_2 + ... + w_n x_n + b$$

The cost function (Mean Squared Error) is:

$$J(w) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

L1 Regularization (Lasso)

$$J(w) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} |w_j|$$

- **Effect:** Encourages sparsity by driving some weights to exactly zero.
- **Gradient update:**

$$w_i := w_i - \eta(\frac{\partial J}{\quad} + \lambda \cdot \text{sign}(w_i))$$

L2 Regularization (Ridge)

$$J(w) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} w_j^2$$

- **Effect:** Shrinks weights but does not remove them completely.
- **Gradient update:**

$$w_j := w_j - \eta(\frac{\partial J}{\partial w_j} + 2\lambda w_j)$$

Elastic Net Regularization

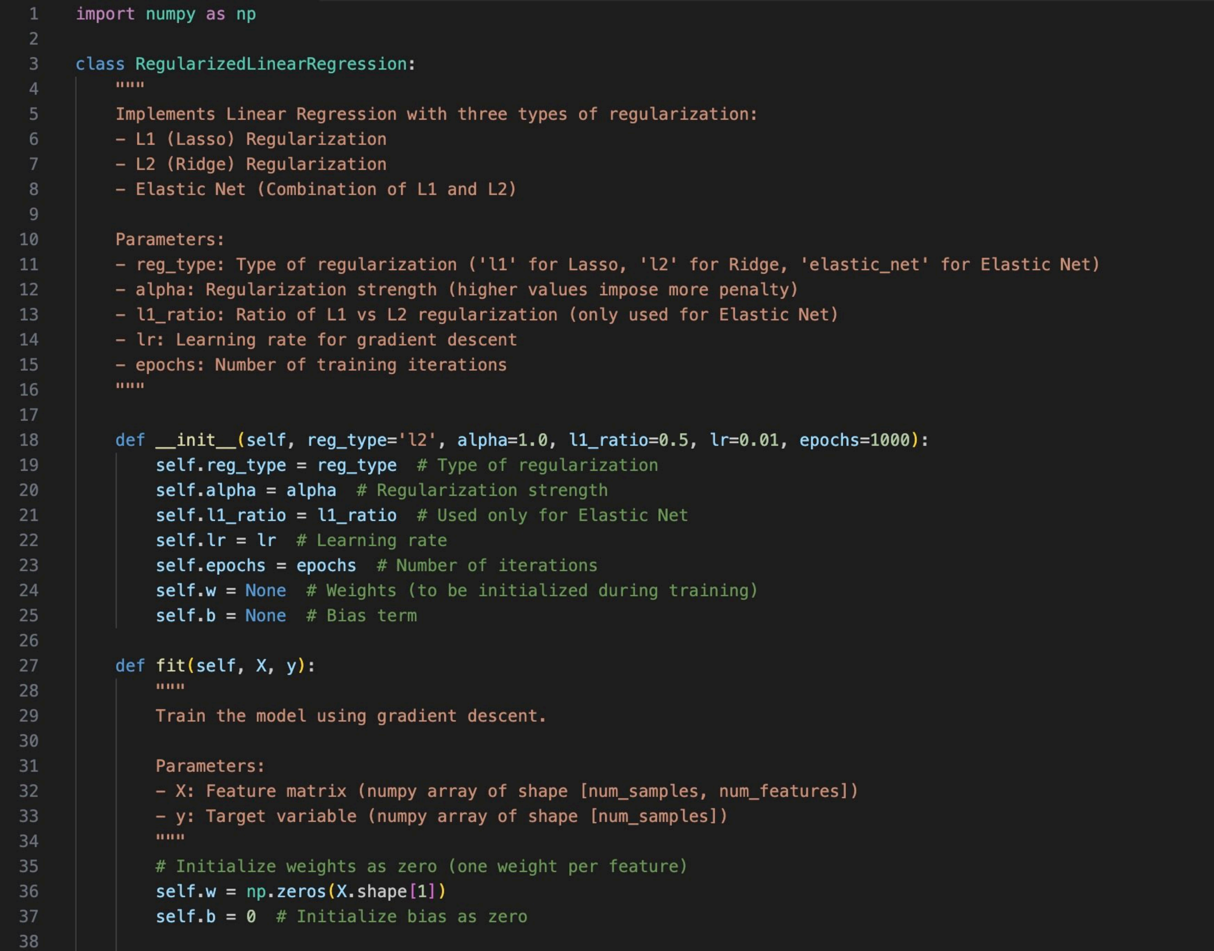
$$J(w) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 + \lambda \left[\alpha \sum_{j=1}^{p} |w_j| + (1-\alpha) \sum_{j=1}^{p} w_j^2 \right]$$

- **Effect:** Combines benefits of both L1 and L2.
- **Gradient update:**

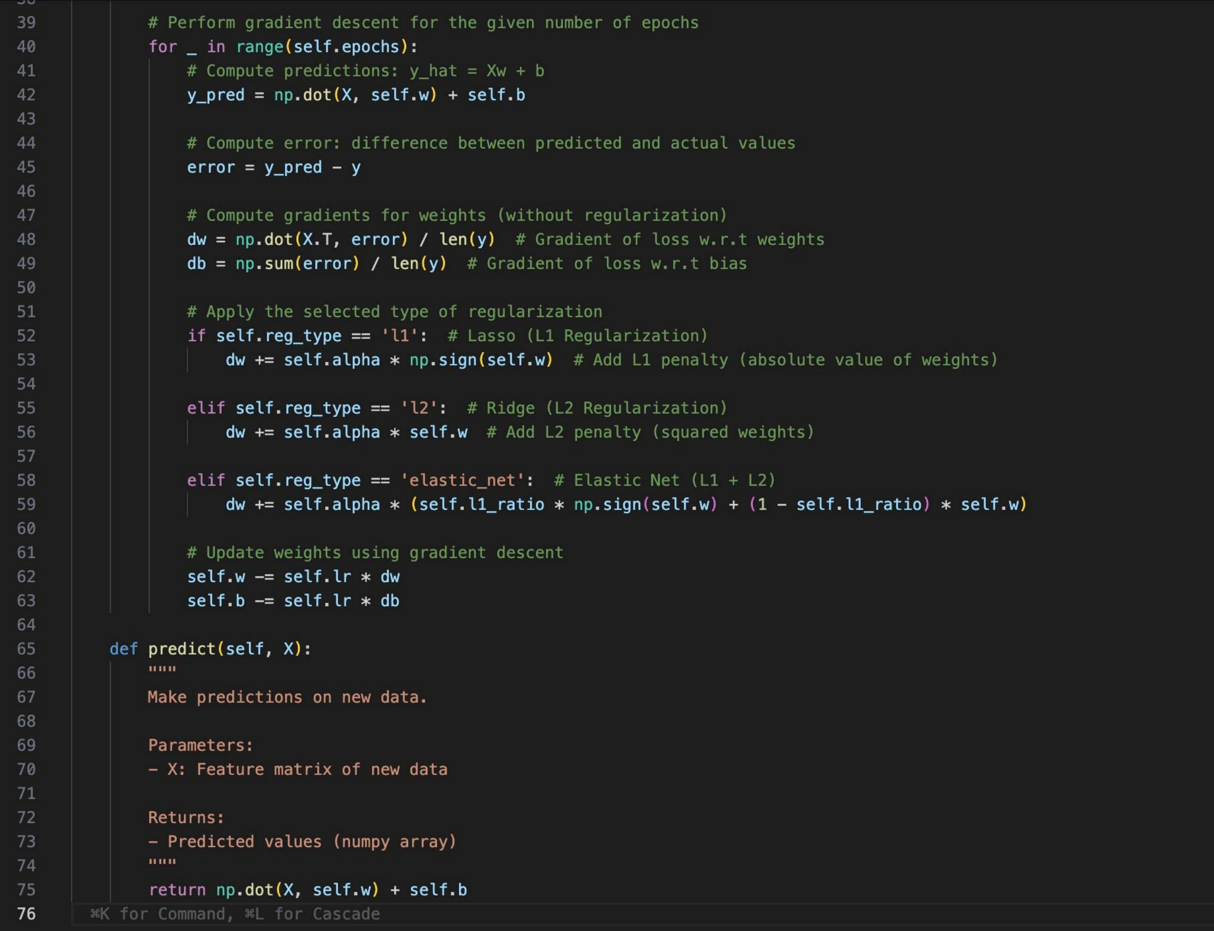
$$w_j := w_j - \eta(\frac{\partial J}{\partial w_j} + \lambda(\alpha \text{sign}(w_j) + 2(1-\alpha) w_j))$$

---

7. Fully Commented Manual Implementation



```python
import numpy as np

class RegularizedLinearRegression:
    """
    Implements Linear Regression with three types of regularization:
    - L1 (Lasso) Regularization
    - L2 (Ridge) Regularization
    - Elastic Net (Combination of L1 and L2)

    Parameters:
    - reg_type: Type of regularization ('l1' for Lasso, 'l2' for Ridge, 'elastic_net' for Elastic Net)
    - alpha: Regularization strength (higher values impose more penalty)
    - l1_ratio: Ratio of L1 vs L2 regularization (only used for Elastic Net)
    - lr: Learning rate for gradient descent
    - epochs: Number of training iterations
    """

    def __init__(self, reg_type='l2', alpha=1.0, l1_ratio=0.5, lr=0.01, epochs=1000):
        self.reg_type = reg_type  # Type of regularization
        self.alpha = alpha  # Regularization strength
        self.l1_ratio = l1_ratio  # Used only for Elastic Net
        self.lr = lr  # Learning rate
        self.epochs = epochs  # Number of iterations
        self.w = None  # Weights (to be initialized during training)
        self.b = None  # Bias term

    def fit(self, X, y):
        """
        Train the model using gradient descent.

        Parameters:
        - X: Feature matrix (numpy array of shape [num_samples, num_features])
        - y: Target variable (numpy array of shape [num_samples])
        """
        # Initialize weights as zero (one weight per feature)
        self.w = np.zeros(X.shape[1])
        self.b = 0  # Initialize bias as zero

```



```python
        # Perform gradient descent for the given number of epochs
        for _ in range(self.epochs):
            # Compute predictions: y_hat = Xw + b
            y_pred = np.dot(X, self.w) + self.b

            # Compute error: difference between predicted and actual values
            error = y_pred - y

            # Compute gradients for weights (without regularization)
            dw = np.dot(X.T, error) / len(y)  # Gradient of loss w.r.t weights
            db = np.sum(error) / len(y)  # Gradient of loss w.r.t bias

            # Apply the selected type of regularization
            if self.reg_type == 'l1':  # Lasso (L1 Regularization)
                dw += self.alpha * np.sign(self.w)  # Add L1 penalty (absolute value of weights)

            elif self.reg_type == 'l2':  # Ridge (L2 Regularization)
                dw += self.alpha * self.w  # Add L2 penalty (squared weights)

            elif self.reg_type == 'elastic_net':  # Elastic Net (L1 + L2)
                dw += self.alpha * (self.l1_ratio * np.sign(self.w) + (1 - self.l1_ratio) * self.w)

            # Update weights using gradient descent
            self.w -= self.lr * dw
            self.b -= self.lr * db

    def predict(self, X):
        """
        Make predictions on new data.

        Parameters:
        - X: Feature matrix of new data

        Returns:
        - Predicted values (numpy array)
        """
        return np.dot(X, self.w) + self.b
```

Step-by-Step Breakdown

1. **Initialization (`__init__` method)**
    - Takes in parameters for regularization type (`l1`, `l2`, `elastic_net`), regularization strength (`alpha`), learning rate (`lr`), and number of iterations (`epochs`).
    - Initializes weights (`w`) and bias (`b`) to `None`.
2. **Training (`fit` method)**
    - Initializes all weights to `0` and bias to `0`.
    - Uses **gradient descent** to update weights and bias.
    - Computes predictions as:
    $$\hat{y} = Xw + b$$
    - Computes gradient (`dw`) and applies the appropriate regularization:
        - **L1 (Lasso)**: Adds `alpha * sign(w)` (penalizes large weights, encourages sparsity).
        - **L2 (Ridge)**: Adds `alpha * w` (shrinks weights towards zero but doesn't set them to zero).
        - **Elastic Net**: Combination of L1 and L2 regularization.
3. **Prediction (`predict` method)**
    - Simply applies the learned weights to new data:
    $$y_{pred} = Xw + b$$

Scikit-Learn Implementation

If you want to use **Lasso, Ridge, or Elastic Net** in `scikit-learn`, you can do:

python

Copy

Edit

```python
from sklearn.linear_model import Ridge, Lasso, ElasticNet

# Ridge Regression (L2 Regularization)
ridge = Ridge(alpha=1.0)
ridge.fit(X_train, y_train)
y_pred_ridge = ridge.predict(X_test)

# Lasso Regression (L1 Regularization)
lasso = Lasso(alpha=1.0)
lasso.fit(X_train, y_train)
y_pred_lasso = lasso.predict(X_test)

# Elastic Net (Combination of L1 & L2)
elastic_net = ElasticNet(alpha=1.0, l1_ratio=0.5)
elastic_net.fit(X_train, y_train)
y_pred_en = elastic_net.predict(X_test)
```