



Linear Regression

1. In-Depth and Specific Intuitive Understanding

Linear regression is the most fundamental supervised learning algorithm used for regression tasks. The goal is to model the relationship between one or more independent variables (X) and a dependent variable (y) using a linear function.

Intuition

Think of linear regression as fitting a straight line through data points in such a way that the total error (difference between actual and predicted values) is minimized. Mathematically, the linear regression model is given by:

$$y = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

where:

- y is the predicted output (dependent variable).
- x_1, x_2, \dots, x_n are the input features (independent variables).
- w_0 (bias) and w_1, w_2, \dots, w_n (weights) are the model parameters that we optimize.

Using the Normal Equation

In matrix form, the equation becomes:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \epsilon$$

where:

- \mathbf{X} is the design matrix (includes all feature values).
- \mathbf{w} is the vector of weights.
- ϵ is the error term.

The **Normal Equation** provides an explicit formula to directly compute \mathbf{w} :

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

Unlike gradient descent, the normal equation does **not** require iterations; it finds the best weights in one computation, provided $\mathbf{X}^T \mathbf{X}$ is invertible.

2. When Linear Regression is Used and When It Should Be Avoided

When to Use It

- ✓ When the relationship between independent and dependent variables is **approximately linear**.
- ✓ When the dataset is **not too large**, making matrix inversion computationally feasible.
- ✓ When interpretability is crucial—weights directly indicate feature importance.
- ✓ When multicollinearity is not severe.

When to Avoid It

- ✗ If the relationship is **highly non-linear**, a straight line won't capture patterns.
- ✗ If **multicollinearity exists**, weights may be unstable (variance inflation).
- ✗ If the dataset has **significant outliers**, as they can distort the line.
- ✗ If the dataset has **too many features relative to samples**, it may lead to overfitting.

3. When It Fails to Converge and How to Avoid That & When It Always Converges and Why

When Linear Regression Fails to Converge

- Using Gradient Descent with a bad learning rate (α):
 - If α is too large, weights oscillate and never settle.
 - If α is too small, convergence is painfully slow.
- Severe multicollinearity (high correlation between features):
 - Causes numerical instability, leading to diverging or unpredictable weights.
- Feature scaling issues:
 - If features have vastly different magnitudes, gradient descent updates can be inefficient.
- Overfitting due to too many features:
 - If $n > m$ (more features than data points), the system becomes underdetermined.

How to Ensure Convergence

- ✓ Choose a reasonable learning rate using grid search or adaptive methods like Adam.
- ✓ Scale features (standardization: mean = 0, variance = 1) to stabilize gradient updates.
- ✓ Use Ridge Regression (L2 regularization) to prevent instability when features are correlated.
- ✓ Use batch or mini-batch gradient descent instead of stochastic if data is noisy.

When It Always Converges

- For convex loss functions (MSE), gradient descent always converges with a small enough learning rate.
- The Normal Equation always provides a solution if $X^T X$ is invertible.
 - If $X^T X$ is not invertible (i.e., determinant is 0), use pseudo-inverse or regularization.



4. Advantages and Disadvantages

Advantages

- ✓ Easy to implement and computationally efficient for small datasets.
- ✓ Interpretable—coefficients show the effect of each feature.
- ✓ Converges reliably when the dataset meets assumptions.

Disadvantages

- ✗ Sensitive to outliers—a single bad data point can distort results.
 - ✗ Assumes linearity—fails when relationships are complex.
 - ✗ Requires feature engineering—irrelevant variables can harm performance.
-

5. Intuitive Algorithm / Pseudo Code

Gradient Descent

1. Initialize weights w and bias w_0 to small random values.
2. Compute predictions using:

$$\hat{y} = w_0 + \sum_{j=1}^n w_j x_j$$

3. Compute loss (Mean Squared Error):

$$MSE = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$$

4. Compute gradients:

$$w_j := w_j - \alpha \frac{\partial MSE}{\partial w_j}$$

5. Update weights until convergence.

Normal Equation




1. Compute:

$$w = (X^T X)^{-1} X^T y$$

2. Done! No iterations required.



6. When to Use the Normal Equation

-  If the dataset is small ($n < 10,000$), Normal Equation is preferable since it finds the exact solution without iterations.
-  If $X^T X$ is not invertible, Normal Equation fails. Use regularization (Ridge Regression) or pseudo-inverse.
-  If there are too many features ($n \gg 10,000$), computing the inverse is too slow; use Gradient Descent.

7. Manual Implementation in Python

Batch Gradient Descent

python

Copy

Edit

```
import numpy as np

class LinearRegressionGD:
    def __init__(self, learning_rate=0.01, epochs=1000):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        m, n = X.shape
        self.weights = np.zeros(n)
        self.bias = 0

        for _ in range(self.epochs):
            y_pred = np.dot(X, self.weights) + self.bias
            error = y_pred - y

            dw = (2/m) * np.dot(X.T, error)
            db = (2/m) * np.sum(error)

            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict(self, X):
        return np.dot(X, self.weights) + self.bias
```

Stochastic Gradient Descent

python

 Copy

 Edit

```
class LinearRegressionSGD:
    def __init__(self, learning_rate=0.01, epochs=1000):
        self.learning_rate = learning_rate
        self.epochs = epochs

    def fit(self, X, y):
        m, n = X.shape
        self.weights = np.zeros(n)
        self.bias = 0

        for _ in range(self.epochs):
            for i in range(m):
                xi = X[i, :]
                yi = y[i]
                y_pred = np.dot(xi, self.weights) + self.bias
                error = y_pred - yi

                dw = 2 * xi * error
                db = 2 * error

                self.weights -= self.learning_rate * dw
                self.bias -= self.learning_rate * db

    def predict(self, X):
        return np.dot(X, self.weights) + self.bias
```

```
1 from sklearn.linear_model import LinearRegression # Import linear regression model
2 from sklearn.model_selection import train_test_split # Import function for splitting dataset
3 from sklearn.datasets import make_regression # Generate synthetic regression dataset
```

```
4
5 # Generate a synthetic dataset with:
```

```
6 # - 100 samples (n_samples=100)
```

```
7 # - 1 feature (n_features=1)
```

```
8 # - Gaussian noise added (noise=10)
```

```
9 # - Reproducibility with random_state=42
```

```
10 X, y = make_regression(n_samples=100, n_features=1, noise=10, random_state=42)
```

```
11
12 # Split dataset into training (80%) and testing (20%)
```

```
13 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
14
15 # Initialize the Linear Regression model
```

```
16 model = LinearRegression()
```

```
17
18 # Train (fit) the model using the training data
```

```
19 model.fit(X_train, y_train)
```

```
20
21 # Make predictions on the test set
```

```
22 y_pred = model.predict(X_test)
```

```
23
24 # Print model parameters (weights and bias)
```

```
25 print(f"Coefficients: {model.coef_}") # Slope (w)
```

```
26 print(f"Intercept: {model.intercept_}") # Bias (w_0)
```

```
27
```

Subit implement

Sk learn

Final Notes & Summary

Linear regression is a **fundamental supervised learning algorithm** that models a linear relationship between independent variables (X) and a dependent variable (y). There are two main ways to solve it:

1. Gradient Descent (GD)

- **Batch Gradient Descent (BGD)**: Uses the entire dataset for each update step.
- **Stochastic Gradient Descent (SGD)**: Uses only one data point per update step.

2.

Normal Equation

- Directly computes the optimal weights **without iterations** but requires matrix inversion, which is slow for large datasets.

When to Use Each Approach

Method	When to Use It	When to Avoid It
Batch Gradient Descent (BGD)	When dataset is small or medium (fitting entire dataset in memory is feasible).	If dataset is too large , computation is slow.
Stochastic Gradient Descent (SGD)	When dataset is large (avoids storing all data in memory at once).	If dataset is small , can be unstable due to noisy updates.
Normal Equation	If dataset is small and we want an exact solution.	If dataset has many features , matrix inversion is expensive or impossible.

Key Takeaways

- Use Batch Gradient Descent (BGD) if dataset is small/medium and fits into memory.
- Use Stochastic Gradient Descent (SGD) if dataset is large, as it allows training in batches or one sample at a time.
- Use the Normal Equation if the number of features is low (avoids iteration-based optimization).
- Feature scaling is essential for gradient descent to ensure convergence.
- If $X^T X$ is not invertible, use Ridge Regression or pseudo-inverse.