**Regularization in Machine Learning:**

**Regularization** is an important technique in machine learning that helps to improve model accuracy by preventing overfitting which happens when a model learns the training data too well including noise and outliers and perform poor on new data. By adding a penalty for complexity it helps simpler models to perform better on new data. In this article, we will see main types of regularization i.e Lasso, Ridge and Elastic Net and see how they help to build more reliable models.

**Types of Regularization:**

* 1. **Lasso Regression**
  2. **Ridge Regression**
  3. **Elastic Net Regression**
  4. **Ridge regression:**

[**https://www.geeksforgeeks.org/what-is-ridge-regression/**](https://www.geeksforgeeks.org/what-is-ridge-regression/)

[**https://www.geeksforgeeks.org/implementation-of-ridge-regression-from-scratch-using-python/**](https://www.geeksforgeeks.org/implementation-of-ridge-regression-from-scratch-using-python/)

[**https://www.geeksforgeeks.org/ml-ridge-regressor-using-sklearn/**](https://www.geeksforgeeks.org/ml-ridge-regressor-using-sklearn/)

* 1. **Lasso regression:**

[**https://www.geeksforgeeks.org/what-is-lasso-regression/**](https://www.geeksforgeeks.org/what-is-lasso-regression/)

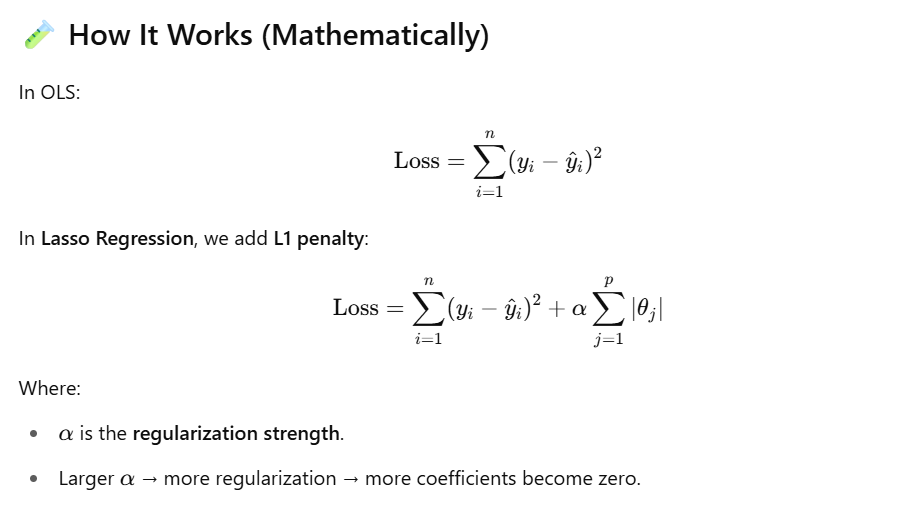
[**https://www.geeksforgeeks.org/implementation-of-lasso-regression-from-scratch-using-python/**](https://www.geeksforgeeks.org/implementation-of-lasso-regression-from-scratch-using-python/)

* 1. **Ridge Regression vs Lasso Regression:**

[**https://www.geeksforgeeks.org/ridge-regression-vs-lasso-regression/**](https://www.geeksforgeeks.org/ridge-regression-vs-lasso-regression/)

**1. Lasso Regression**

A regression model which uses the L1 Regularization technique is called [LASSO (Least Absolute Shrinkage and Selection Operator)](https://www.geeksforgeeks.org/what-is-lasso-regression/) regression. It adds the absolute value of magnitude of the coefficient as a penalty term to the loss function(L). This penalty can shrink some coefficients to zero which helps in selecting only the important features and ignoring the less important ones.



**When to Use Lasso Regression?**

* When you suspect **many features are irrelevant**.
* When you want to **reduce model complexity**.
* When you want **interpretability** by identifying important features.

Lets see how to implement this using python:

* **X, y = make\_regression(n\_samples=100, n\_features=5, noise=0.1, random\_state=42)**: Generates a regression dataset with 100 samples, 5 features and some noise.
* **X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)** : Splits the data into 80% training and 20% testing sets.
* **lasso = Lasso(alpha=0.1)**: Creates a Lasso regression model with regularization strength alpha set to 0.1.

**Full Code with Explanations**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression, Lasso

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.preprocessing import PolynomialFeatures

# Step 1: Generate synthetic data (non-linear)

np.random.seed(42) # Ensures reproducibility

X = 2 \* np.random.rand(100, 1) - 1 # 100 points between [-1, 1]

y = 3 \* X\*\*2 + 2 \* X + 1 + np.random.randn(100, 1) \* 0.3 # y = 3x² + 2x + 1 + noise

# Print data before split

print("Sample Data (X, y):")

print(np.hstack((X[:5], y[:5])))

# Step 2: Polynomial features transformation (degree=2)

poly = PolynomialFeatures(degree=2, include\_bias=False)

X\_poly = poly.fit\_transform(X)

# Parameters:

# degree=2 → Create x and x² features

# include\_bias=False → Don't add column of 1s (for intercept)

# Step 3: Split into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_poly, y, test\_size=0.3, random\_state=42)

# Parameters:

# test\_size=0.3 → 30% data for testing

# random\_state=42 → Ensures consistent split across runs

# Step 4: Linear Regression model (baseline)

lin\_model = LinearRegression()

lin\_model.fit(X\_train, y\_train)

# Step 5: Lasso Regression model

lasso\_model = Lasso(alpha=0.1, max\_iter=10000)

lasso\_model.fit(X\_train, y\_train)

# Parameters:

# alpha=0.1 → Regularization strength (higher = more shrinkage)

# max\_iter=10000 → To ensure convergence

# Step 6: Predictions

y\_pred\_lin = lin\_model.predict(X\_test)

y\_pred\_lasso = lasso\_model.predict(X\_test)

# Step 7: Evaluation

mse\_lin = mean\_squared\_error(y\_test, y\_pred\_lin)

r2\_lin = r2\_score(y\_test, y\_pred\_lin)

mse\_lasso = mean\_squared\_error(y\_test, y\_pred\_lasso)

r2\_lasso = r2\_score(y\_test, y\_pred\_lasso)

# Print results

print("\n🔍 Linear Regression:")

print("Coefficients:", lin\_model.coef\_)

print("Intercept:", lin\_model.intercept\_)

print("MSE:", mse\_lin)

print("R²:", r2\_lin)

print("\n🔍 Lasso Regression:")

print("Coefficients:", lasso\_model.coef\_)

print("Intercept:", lasso\_model.intercept\_)

print("MSE:", mse\_lasso)

print("R²:", r2\_lasso)

# Step 8: Plotting

plt.figure(figsize=(10, 6))

plt.scatter(X, y, color='gray', alpha=0.5, label='Data')

# Sort X for plotting smooth curve

X\_plot = np.linspace(-1, 1, 100).reshape(-1, 1)

X\_plot\_poly = poly.transform(X\_plot)

plt.plot(X\_plot, lin\_model.predict(X\_plot\_poly), color='blue', label='Linear Regression')

plt.plot(X\_plot, lasso\_model.predict(X\_plot\_poly), color='red', linestyle='--', label='Lasso Regression (α=0.1)')

plt.title("Linear vs Lasso Regression")

plt.xlabel("X")

plt.ylabel("y")

plt.legend()

plt.grid(True)

plt.show()

**Result Analysis**

**Linear Regression:**

* Fits the data closely (no penalty).
* May **overfit** slightly when the noise is high or the model is too complex.

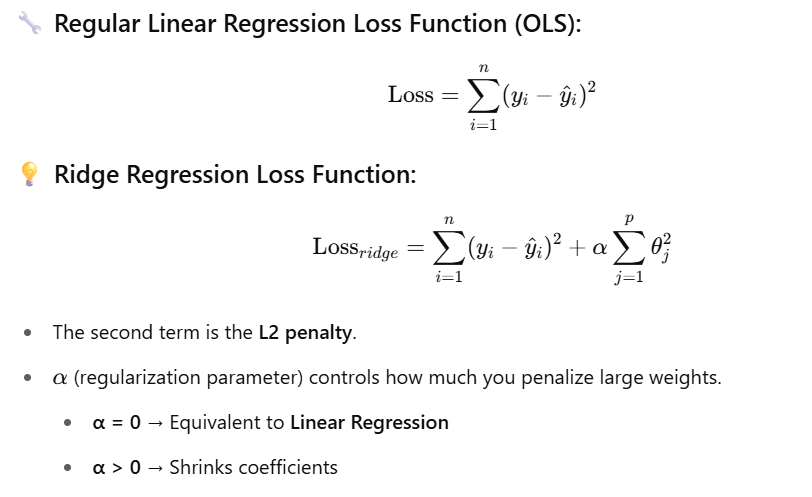
**Lasso Regression:**

* Applies L1 penalty to shrink coefficients.
* If alpha is high, it can **zero out coefficients** → Feature selection.
* Helps in **preventing overfitting**.

**2. Ridge Regression:**

A regression model that uses the **L2 regularization** technique is called [**Ridge regression**](https://www.geeksforgeeks.org/what-is-ridge-regression/). It adds the **squared magnitude** of the coefficient as a penalty term to the loss function(L).

**Ridge Regression** (also called **L2 regularization**) is a variant of **Linear Regression** that **adds a penalty on the size of coefficients** to **reduce overfitting** and **improve generalization**.



**When to Use Ridge Regression?**

* When you have **multicollinearity** (correlated features)
* When you want to **prevent overfitting**
* When **number of features ≥ number of samples**

**Full Ridge Regression Example in Python (with Analysis)**

**🎯 Goal: Compare Linear Regression and Ridge Regression on polynomial data**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression, Ridge

from sklearn.preprocessing import PolynomialFeatures

from sklearn.metrics import mean\_squared\_error, r2\_score

# Step 1: Generate synthetic data (non-linear)

np.random.seed(42)

X = 2 \* np.random.rand(100, 1) - 1 # X in [-1, 1]

y = 3 \* X\*\*2 + 2 \* X + 1 + np.random.randn(100, 1) \* 0.3 # Quadratic + noise

print("Sample data (first 5 rows):")

print(np.hstack((X[:5], y[:5])))

# Step 2: Polynomial features (degree=2)

poly = PolynomialFeatures(degree=2, include\_bias=False)

X\_poly = poly.fit\_transform(X) # Creates [x, x^2]

# Step 3: Train/test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_poly, y, test\_size=0.3, random\_state=42)

# Step 4: Linear Regression (baseline)

lin\_reg = LinearRegression()

lin\_reg.fit(X\_train, y\_train)

y\_pred\_lin = lin\_reg.predict(X\_test)

# Step 5: Ridge Regression

ridge\_reg = Ridge(alpha=1.0) # α = regularization strength

ridge\_reg.fit(X\_train, y\_train)

y\_pred\_ridge = ridge\_reg.predict(X\_test)

# Step 6: Evaluation

print("\n🔍 Linear Regression:")

print("Coefficients:", lin\_reg.coef\_)

print("Intercept:", lin\_reg.intercept\_)

print("MSE:", mean\_squared\_error(y\_test, y\_pred\_lin))

print("R² Score:", r2\_score(y\_test, y\_pred\_lin))

print("\n🔍 Ridge Regression:")

print("Coefficients:", ridge\_reg.coef\_)

print("Intercept:", ridge\_reg.intercept\_)

print("MSE:", mean\_squared\_error(y\_test, y\_pred\_ridge))

print("R² Score:", r2\_score(y\_test, y\_pred\_ridge))

# Step 7: Visualization

plt.figure(figsize=(10, 6))

plt.scatter(X, y, color='gray', alpha=0.5, label='Data')

X\_plot = np.linspace(-1, 1, 100).reshape(-1, 1)

X\_plot\_poly = poly.transform(X\_plot)

plt.plot(X\_plot, lin\_reg.predict(X\_plot\_poly), color='blue', label='Linear Regression')

plt.plot(X\_plot, ridge\_reg.predict(X\_plot\_poly), color='green', linestyle='--', label='Ridge Regression (α=1.0)')

plt.title("Linear vs Ridge Regression")

plt.xlabel("X")

plt.ylabel("y")

plt.grid(True)

plt.legend()

plt.show()

**Results Interpretation**

| **Metric** | **Linear Regression** | **Ridge Regression** |
| --- | --- | --- |
| Coefficients | May be large | Smaller (shrunk) |
| MSE | Slightly lower | Slightly higher (but better generalization) |
| R² Score | High (overfit risk) | High, but more stable |

**Parameters of Ridge()**

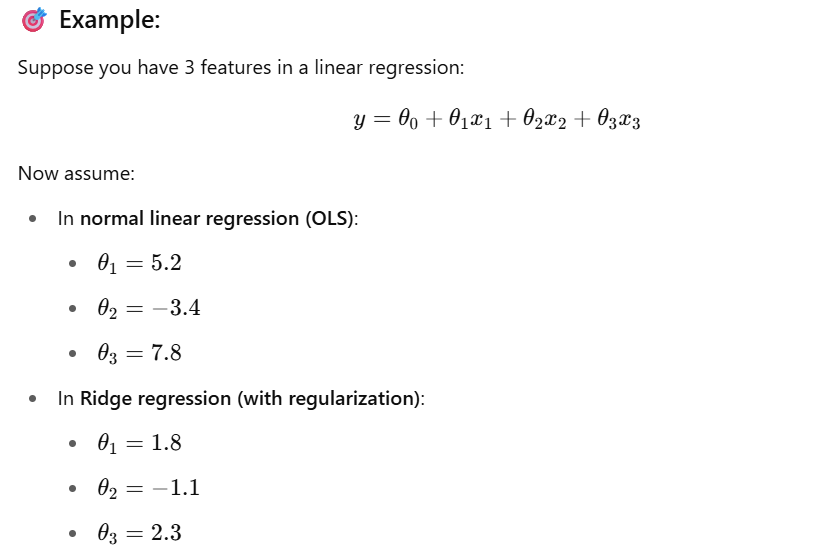
| **Parameter** | **Meaning** |
| --- | --- |
| alpha | Regularization strength (default = 1.0) |
| fit\_intercept | Whether to fit the intercept (default = True) |
| max\_iter | Max iterations (important for convergence with large alpha) |
| solver | Algorithm used in optimization (auto, svd, cholesky, etc.) |

**Summary**

* ✅ Ridge adds **L2 penalty** to reduce model complexity
* ✅ Keeps **all features** but **shrinks coefficients**
* ✅ Ideal when **features are correlated**
* ✅ Use when you want to **balance bias-variance tradeoff**

**“Keeps all features but shrinks coefficients” — What does it mean?**

In **Ridge Regression**, the algorithm does **not eliminate** any input features (independent variables), but it **reduces the impact** of some by making their corresponding **coefficients smaller (closer to zero)**.

****

Notice:  
✅ All features x1, x2, x3 are still **used**.  
📉 But their **influence is reduced** — the coefficients are **smaller**.

That's **what “shrinking” means**.

**🤔 Why shrink?**

Because **large coefficients** may cause the model to **overfit** — they make the model too sensitive to small changes in the input data.

By shrinking the weights:

* We prevent the model from being too "certain" about any one feature.
* We help the model **generalize better to unseen data**.

**🔄 In contrast: Lasso Regression**

* **Lasso** can **shrink some coefficients exactly to zero** → effectively **removing features**
* That’s called **feature selection**

But Ridge:

* Shrinks all coefficients toward zero
* **Never makes them exactly zero**

**🔁 Summary**

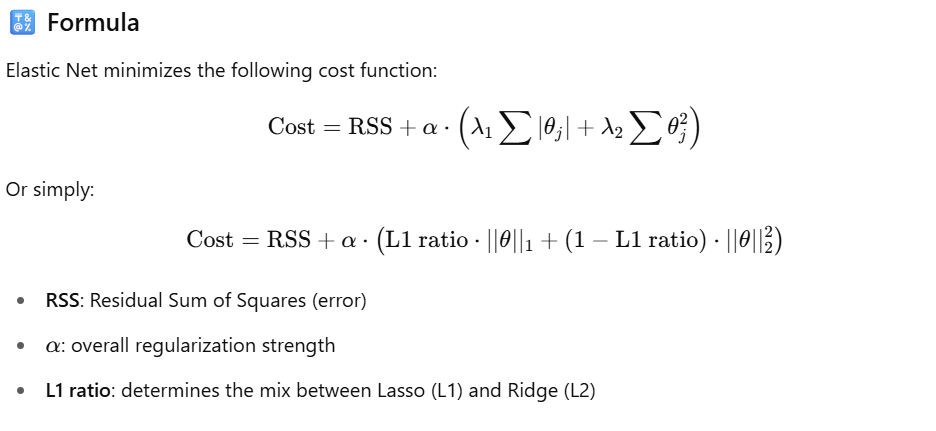
| **Term** | **Meaning** |
| --- | --- |
| **"Keep all features"** | Ridge doesn't eliminate any feature (doesn't set any weight exactly to 0) |
| **"Shrink coefficients"** | Ridge makes the weights smaller, reducing their influence to avoid overfitting |

**Elastic Net Regression**

[**Elastic Net Regression**](https://www.geeksforgeeks.org/implementation-of-elastic-net-regression-from-scratch/) is a combination of both**L1 as well as L2 regularization.** That shows that we add the **absolute norm of the weights** as well as the **squared measure of the weights**. With the help of an extra hyperparameter that controls the ratio of the L1 and L2 regularization.

**Elastic Net** is a **regularization technique** that combines both:

* **Lasso Regression (L1 penalty)**: encourages **sparsity** — can shrink some coefficients to **zero** (feature selection).
* **Ridge Regression (L2 penalty)**: **shrinks** all coefficients — prevents overfitting by reducing their size.



**When to Use Elastic Net**

* When you suspect **correlated features** (Ridge handles this better than Lasso).
* When **feature selection is needed** but Lasso alone is unstable.
* When you want **benefits of both L1 and L2 regularization**.

**📌 Key Points**

| **Method** | **Can remove features?** | **Handles multicollinearity?** | **Shrinks coefficients** |
| --- | --- | --- | --- |
| Lasso (L1) | ✅ Yes | ❌ Poor | ✅ Yes |
| Ridge (L2) | ❌ No | ✅ Good | ✅ Yes |
| Elastic Net | ✅ Yes | ✅ Good | ✅ Yes |

**🧪 Code Example: Elastic Net Regression with Explanation**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import ElasticNet, LinearRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error, r2\_score

# 1. Generate synthetic data

np.random.seed(42)

X = 2 \* np.random.rand(100, 1)

y = 4 + 3 \* X[:, 0] + np.random.randn(100)

# Add multicollinearity (second feature is correlated with first)

X = np.c\_[X, X[:, 0] + 0.01 \* np.random.randn(100)]

# 2. Split train/test

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 3. Fit Linear Regression

lr = LinearRegression()

lr.fit(X\_train, y\_train)

y\_pred\_lr = lr.predict(X\_test)

# 4. Fit Elastic Net

en = ElasticNet(alpha=0.1, l1\_ratio=0.5, random\_state=42) # 50% L1, 50% L2

en.fit(X\_train, y\_train)

y\_pred\_en = en.predict(X\_test)

# 5. Evaluate

print("🔷 Linear Regression:")

print(" Coefficients:", lr.coef\_)

print(" Intercept:", lr.intercept\_)

print(" MSE:", mean\_squared\_error(y\_test, y\_pred\_lr))

print(" R² Score:", r2\_score(y\_test, y\_pred\_lr))

print("\n🔷 Elastic Net Regression:")

print(" Coefficients:", en.coef\_)

print(" Intercept:", en.intercept\_)

print(" MSE:", mean\_squared\_error(y\_test, y\_pred\_en))

print(" R² Score:", r2\_score(y\_test, y\_pred\_en))

# 6. Plot

plt.figure(figsize=(10, 6))

plt.scatter(X\_test[:, 0], y\_test, color='blue', label='True Data')

plt.scatter(X\_test[:, 0], y\_pred\_lr, color='green', label='Linear Predicted')

plt.scatter(X\_test[:, 0], y\_pred\_en, color='red', label='Elastic Net Predicted', marker='x')

plt.xlabel("Feature X")

plt.ylabel("Target y")

plt.title("Linear vs Elastic Net Regression")

plt.legend()

plt.grid(True)

plt.show()

**Analysis of Results**

**Linear Regression**

* May overfit if features are highly correlated.
* Coefficients can be unstable in presence of multicollinearity.

**Elastic Net**

* Coefficients are **shrunk**, improving generalization.
* Can **zero out** some coefficients (if l1\_ratio is high).
* Helps handle **collinearity** better than Lasso alone.

**🔧 Parameter Explanation**

| **Parameter** | **Description** |
| --- | --- |
| alpha | Regularization strength (higher = more penalty) |
| l1\_ratio | Mix between Lasso (1.0) and Ridge (0.0) |
| random\_state | For reproducibility |
| fit\_intercept | Whether to estimate intercept (default True) |

**✅ Summary**

**Elastic Net** is the best of both worlds:

* Lasso's ability to remove irrelevant features
* Ridge's robustness to correlated predictors

It’s often used when:

* You have **many features**
* You want to **reduce overfitting**
* You want **some feature selection** but with **stability**

**What are Correlated Features?**

**Correlation** between features means that **one feature can be linearly predicted from another** with some degree of accuracy.

* If Feature A increases and Feature B also increases, they are **positively correlated**.
* If Feature A increases and Feature B decreases, they are **negatively correlated**.
* If there is **no linear relationship**, they are **uncorrelated**.

📌 Correlation is measured by **Pearson's correlation coefficient (r)**, which ranges from -1 to +1.

**🔷 What is Multicollinearity?**

**Multicollinearity** is a condition where **two or more independent variables (features) in a regression model are highly correlated**.

This is a problem because:

* It becomes hard to determine the effect of each feature independently.
* Coefficients become unstable (large or flip signs).
* It can lead to **overfitting** and **poor generalization**.

✅ Ridge and ElasticNet handle multicollinearity well by **shrinking coefficients**.

**🧪 Sample Data to Illustrate Correlation & Multicollinearity**

**🔹 Step 1: Generate Example**

python

CopyEdit

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

# Seed for reproducibility

np.random.seed(0)

# Create a feature X1

X1 = np.random.rand(100)

# Create another feature X2 highly correlated with X1

X2 = X1 + np.random.normal(0, 0.01, 100) # Add small noise

# Create a target variable y based on X1

y = 3 \* X1 + np.random.normal(0, 0.1, 100)

# Combine into DataFrame

df = pd.DataFrame({'X1': X1, 'X2': X2, 'y': y})

print(df.head())

**🔹 Step 2: Visualize Correlation Matrix**

python

CopyEdit

# Compute and plot correlation matrix

corr = df.corr()

sns.heatmap(corr, annot=True, cmap='coolwarm')

plt.title("Correlation Matrix")

plt.show()

**💡 Expected Output:**

markdown

CopyEdit

X1 X2 y

0 0.548814 0.549401 1.753551

1 0.715189 0.716988 2.189402

2 0.602763 0.603061 1.832431

3 0.544883 0.546506 1.566114

4 0.423655 0.425522 1.325846

**🔸 Correlation Matrix Output (example):**

markdown

CopyEdit

X1 X2 y

X1 1.00 0.999 0.97

X2 0.999 1.00 0.97

y 0.97 0.97 1.00

🔥 Here, **X1 and X2 have a correlation of 0.999**, meaning they're almost **linearly dependent**.

**🔍 Why Is This a Problem?**

Imagine using both X1 and X2 in linear regression:

python

CopyEdit

from sklearn.linear\_model import LinearRegression

model = LinearRegression()

model.fit(df[['X1', 'X2']], df['y'])

print("Coefficients:", model.coef\_)

print("Intercept:", model.intercept\_)

You might get weird or unstable coefficients like:

makefile

CopyEdit

Coefficients: [ 8.3 -5.2 ]

This happens because the model cannot distinguish the separate effects of X1 and X2 since they are almost the same.

**🛠 How to Detect Multicollinearity?**

* **Correlation matrix** (as we did above)
* **Variance Inflation Factor (VIF)**: VIF > 5 or 10 is a red flag

python

CopyEdit

from statsmodels.stats.outliers\_influence import variance\_inflation\_factor

# Calculate VIF for each feature

X = df[['X1', 'X2']]

vif\_data = pd.DataFrame()

vif\_data["Feature"] = X.columns

vif\_data["VIF"] = [variance\_inflation\_factor(X.values, i) for i in range(X.shape[1])]

print(vif\_data)

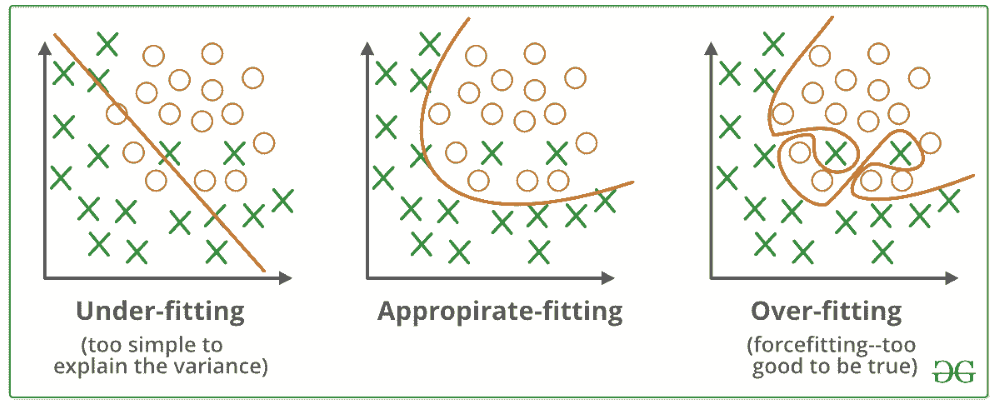
This might show very high VIF values (~100+), confirming **multicollinearity**.

**✅ Summary**

| **Term** | **Meaning** |
| --- | --- |
| Correlated Feature | Two or more features that move together (have a high correlation) |
| Multicollinearity | Multiple features are correlated, affecting model stability |
| Problem? | Yes. Makes coefficients unreliable in linear models like OLS regression |
| Solution | Use **Ridge** or **Elastic Net** to handle it |

**What are Overfitting and Underfitting?**

**Overfitting** and **underfitting**are terms used to describe the performance of machine learning models in relation to their ability to generalize from the training data to unseen data.



[**Overfitting**](https://www.geeksforgeeks.org/how-to-handle-overfitting-in-tensorflow-models/) happens when a machine learning model learns the training data too well including the noise and random details. This makes the model to perform poorly on new, unseen data because it memorizes the training data instead of understanding the general patterns.

For example, if we only study last week’s weather to predict tomorrow’s i.e our model might focus on one-time events like a sudden rainstorm which won’t help for future predictions.

[**Underfitting**](https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/) is the opposite problem which happens when the model is too simple to learn even the basic patterns in the data. An underfitted model performs poorly on both training and new data. To fix this we need to make the model more complex or add more features.

For example if we use only the average temperature of the year to predict tomorrow’s weather hence the model misses important details like seasonal changes which results in bad predictions.

**What are Bias and Variance?**

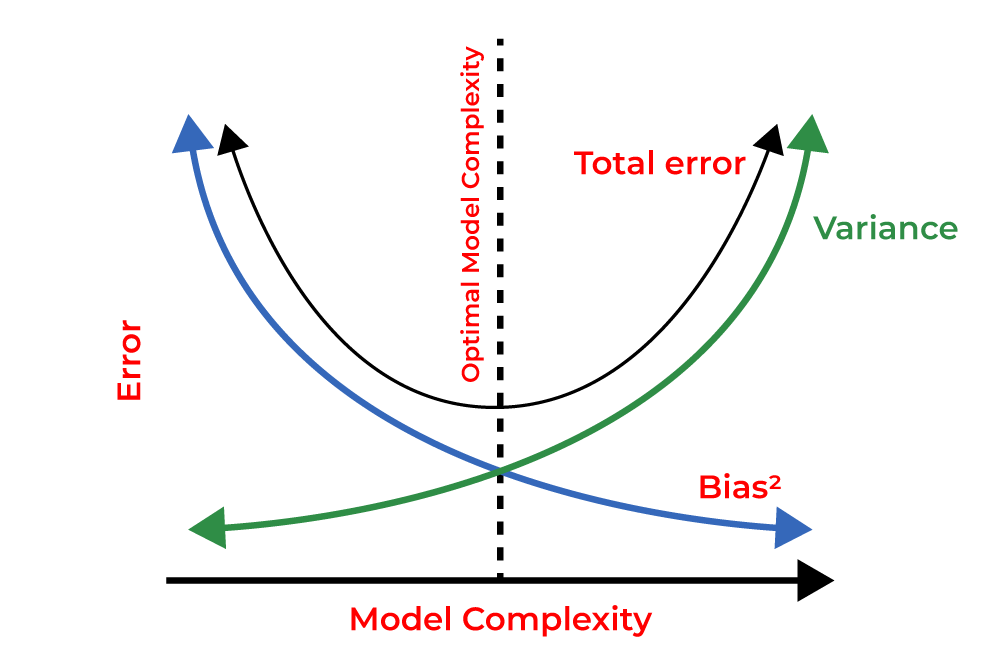
* **Bias** refers to the errors which occur when we try to fit a statistical model on real-world data which does not fit perfectly well on some mathematical model. If we use a way too simplistic a model to fit the data then we are more probably face the situation of **High Bias** (underfitting) refers to the case when the model is unable to learn the patterns in the data at hand and perform poorly.
* **Variance** shows the error value that occurs when we try to make predictions by using data that is not previously seen by the model. There is a situation known as **high variance** (overfitting) that occurs when the model learns noise that is present in the data.

Finding a proper balance between the two is also known as the **Bias-Variance Tradeoff** which helps us to design an accurate model.

**Bias Variance tradeoff**

The [**Bias-Variance Tradeoff**](https://www.geeksforgeeks.org/ml-bias-variance-trade-off/)refers to the balance between bias and variance which affect predictive model performance. Finding the right tradeoff is important for creating models that generalize well to new data.

* The **bias-variance tradeoff**shows the inverse relationship between bias and variance. When one decreases, the other tends to increase and vice versa.
* Finding the right balance is important. An overly simple model with high bias won't capture the underlying patterns while an overly complex model with high variance will fit the noise in the data.



**Benefits of Regularization**

Now, let’s see various benefits of regularization which are as follows:

1. **Prevents Overfitting:** Regularization helps models focus on underlying patterns instead of memorizing noise in the training data.
2. **Improves Interpretability:** L1 (Lasso) regularization simplifies models by reducing less important feature coefficients to zero.
3. **Enhances Performance:** Prevents excessive weighting of outliers or irrelevant features helps in improving overall model accuracy.
4. **Stabilizes Models:** Reduces sensitivity to minor data changes which ensures consistency across different data subsets.
5. **Prevents Complexity:** Keeps model from becoming too complex which is important for limited or noisy data.
6. **Handles Multicollinearity:** Reduces the magnitudes of correlated coefficients helps in improving model stability.
7. **Allows Fine-Tuning:** Hyperparameters like alpha and lambda control regularization strength helps in balancing bias and variance.
8. **Promotes Consistency:** Ensures reliable performance across different datasets which reduces the risk of large performance shifts.