Lasso and Ridge Regression

Both **Lasso** and **Ridge** are types of regression that use regularization to improve model generalization by preventing overfitting. However, they differ in how they impose penalties on the model's coefficients. Here are the key takeaways from their comparison:

---

1. Effect of the Penalty Term

- **Lasso (L1 Regularization)** adds a penalty proportional to the **absolute values** of the coefficients ($|w_i|$). This leads to **some coefficients becoming exactly zero**, effectively performing **feature selection**.
- **Ridge (L2 Regularization)** penalizes the **squared values** of the coefficients (w_i^2), causing coefficients to shrink **towards zero** but never become exactly zero. Thus, it does not eliminate any features but reduces their impact.

Interpretation

- If you want to **select only the most relevant features**, Lasso is preferred.
- If you want to **retain all features while reducing overfitting**, Ridge is a better choice.

---

2. Impact on Model Complexity

- **Lasso** simplifies the model by eliminating less important features, making it useful when working with high-dimensional data.
- **Ridge** retains all features but controls their influence, making it effective in cases where all predictors contribute meaningfully.

Interpretation

- If the dataset has **many irrelevant or redundant features**, Lasso can simplify the model by removing them.
- If all features carry **some predictive power**, Ridge is more suitable, especially in high-dimensional problems.

---

3. Handling of Multicollinearity

- **Lasso** tends to randomly select one variable among a group of correlated variables while shrinking others to zero.
- **Ridge** distributes the weight among correlated variables, making it **better suited for handling multicollinearity**.

Interpretation

- If multiple variables are **highly correlated**, Ridge prevents instability by assigning **small but nonzero** values to each.
- Lasso, on the other hand, may arbitrarily select only one correlated variable and **ignore others**.

---

4. Bias-Variance Tradeoff

Regularization	Bias	Variance	Effect on Model
Lasso (L1)	Higher	Lower	More simplified, sparse model with feature selection.
Ridge (L2)	Lower	Higher	More stable model that retains all features but shrinks coefficients.

Interpretation

- If the goal is to **reduce variance** and handle overfitting, Ridge is preferred.
- If the dataset has **too many features**, leading to overfitting, Lasso helps by **eliminating irrelevant ones**.

---

5. When to Use Lasso vs. Ridge?

Scenario	Lasso (L1)	Ridge (L2)
Feature Selection	✅ Yes (eliminates irrelevant features)	❌ No (keeps all features)
Multicollinearity	❌ No (picks one variable at random)	✅ Yes (distributes importance among correlated variables)
Sparse Models	✅ Yes	❌ No
High-Dimensional Data	✅ Yes (selects only relevant features)	✅ Yes (shrinks all coefficients)
Interpretability	✅ More interpretable (fewer features)	❌ Less interpretable (all features retained)
Small Datasets	✅ Works well	❌ Can overfit if too many features exist
Large Datasets	❌ Can ignore useful features	✅ Works well

Interpretation

- If **feature selection** is important → Use **Lasso**.
- If **all features carry some predictive power** but need regularization → Use **Ridge**.
- If unsure, **Elastic Net** combines both Lasso and Ridge for a balanced approach.

---

6. The Elastic Net Alternative (L1 + L2 Regularization)

- **Elastic Net** combines both **Lasso (L1)** and **Ridge (L2)** penalties, making it useful when:
  - There are **many correlated features** (avoids Lasso's random feature selection).

- The dataset is **high-dimensional** and needs both **shrinkage and feature selection**.

Its penalty term is:

$$J(\mathbf{w}) = \sum(y_i - \hat{y}_i)^2 + \lambda_1 \sum |w_i| + \lambda_2 \sum w_i^2$$

- Here, λ_1 controls the Lasso effect, and λ_2 controls the Ridge effect.
- This balances **feature selection and coefficient shrinkage** for better model performance.

Final Thoughts

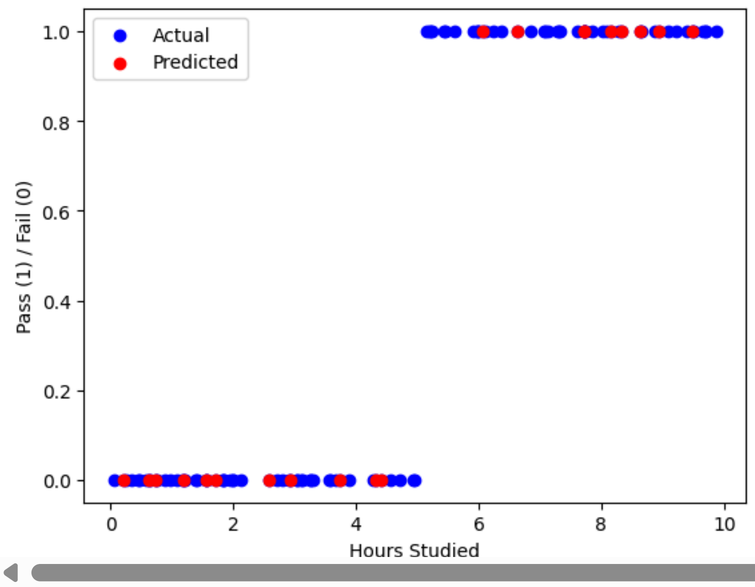
- **Lasso** is better when feature selection is needed.
- **Ridge** is better when **all features** contribute and multicollinearity is present.
- **Elastic Net** is useful when you want the benefits of both Lasso and Ridge.

1. Basic Logistic Regression (Binary Classification)

A simple logistic regression model on a small dataset (e.g., predicting if a student passes based on study hours).

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Sample dataset
np.random.seed(42)
hours_studied = np.random.rand(100, 1) * 10  # Random study hours (0 to 10)
passed_exam = (hours_studied > 5).astype(int).ravel()  # Label: 1 if hours > 5, else 0
print('Dimension=',passed_exam.shape)
# Split data
X_train, X_test, y_train, y_test = train_test_split(hours_studied, passed_exam, test_size=0.2, random_state=42)

# Train Logistic Regression Model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predict and evaluate
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))

# Plot decision boundary
plt.scatter(hours_studied, passed_exam, color='blue', label='Actual')
plt.scatter(X_test, y_pred, color='red', label='Predicted')
plt.xlabel("Hours Studied")
plt.ylabel("Pass (1) / Fail (0)")
plt.legend()
plt.show()
```

```
Dimension= (100,)
Accuracy: 1.0
```



2. Multi-Class Logistic Regression (One-vs-Rest)

Using Iris dataset to classify species into three classes.

```
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# Load dataset
iris = datasets.load_iris()
X = iris.data  # Features
y = iris.target  # Labels (0, 1, 2)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train Logistic Regression
model = LogisticRegression(multi_class='ovr', solver='liblinear')
# OneVsRestClassifier(LogisticRegression(multi_class='ovr', solver='liblinear'))
model.fit(X_train, y_train)

# Predict and evaluate
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      0.89      0.94         9
           2       0.92      1.00      0.96        11

    accuracy                           0.97        30
   macro avg       0.97      0.96      0.97        30
weighted avg       0.97      0.97      0.97        30

/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:1256: FutureWarning: 'multi_class' was deprecated in versi
  warnings.warn(
```

3. Logistic Regression with Feature Engineering

Adding polynomial features for a complex dataset.

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline

# Generate a synthetic dataset
np.random.seed(42)
X = 2 * np.random.rand(100, 1) - 1  # Random values between -1 and 1
y = (X[:, 0]**2 + np.random.randn(100) * 0.1 > 0.5).astype(int)  # Non-linear decision boundary

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a pipeline with polynomial features
model = Pipeline([
    ('poly', PolynomialFeatures(degree=2)),
    ('log_reg', LogisticRegression())
])

# Train and evaluate
model.fit(X_train, y_train)
print("Accuracy:", model.score(X_test, y_test))
```

```
Accuracy: 0.9
```

4. Logistic Regression on Imbalanced Data

Handling an imbalanced dataset using class weights.

```
from sklearn.utils import resample
from sklearn.metrics import confusion_matrix
```

```
# Generate an imbalanced dataset
X = np.random.randn(1000, 2)
y = np.hstack((np.zeros(950), np.ones(50)))  # 95% class 0, 5% class 1

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train logistic regression with class weights
model = LogisticRegression(class_weight='balanced')
model.fit(X_train, y_train)

# Predict and evaluate
y_pred = model.predict(X_test)
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
Confusion Matrix:
 [[107  83]
 [  5   5]]
              precision    recall  f1-score   support

         0.0       0.96      0.56      0.71       190
         1.0       0.06      0.50      0.10        10

    accuracy                           0.56       200
   macro avg       0.51      0.53      0.41       200
weighted avg       0.91      0.56      0.68       200
```

```
Start coding or generate with AI.
```

5. Logistic Regression with Stochastic Gradient Descent (SGD)

Scaling up logistic regression for large datasets using SGD.

```
from sklearn.linear_model import SGDClassifier
from sklearn.datasets import make_classification

# Generate a large dataset
X, y = make_classification(n_samples=100000, n_features=20, random_state=42)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train using SGD
model = SGDClassifier(loss='log_loss', max_iter=1000, tol=1e-3)
model.fit(X_train, y_train)

# Evaluate
print("Accuracy:", model.score(X_test, y_test))
```

```
Accuracy: 0.869
```

```
Start coding or generate with AI.
```