Shrinkage Methods: Theory and Comparison

Criteria for Model Selection

As demonstrated in a previous project, the model containing all of the predictors will always have the smallest **RSS** and the largest **R^2**, since these quantities are related to the training error.

However, we wish to choose a model with **low test error**, not a model with low training error. Recall that **training error** is usually a poor estimate of **test error**.

Therefore, **RSS** and **R^2** are not suitable for selecting the best model among a collection of models with different numbers of predictors.

Two Approaches for Estimating Test Error:

1. **Indirect Estimation**:
   We can indirectly estimate test error by making an adjustment to the training error to account for the bias due to overfitting.

2. **Direct Estimation**:
   We can directly estimate the test error using either:

   - A **validation set approach**
   - A **cross-validation approach**, as discussed in the next lectures.

Shrinkage Methods and Project Focus

In this project, we will examine **model selection** through **shrinkage methods**: **Ridge** and **Lasso** regression.

Shrinkage Methods:

The subset selection methods use least squares to fit a linear model that contains a subset of the predictors.

As an alternative, we can fit a model containing all the predictors using a technique that **constrains** or **regularizes** the coefficient estimates, or equivalently, that shrinks the coefficient estimates towards zero.

It may not be immediately obvious why such a constraint should improve the fit, but shrinking the coefficient estimates can significantly reduce their variance. By doing so, shrinkage methods like Ridge and Lasso help prevent overfitting and improve model generalization.

- **Ridge Regression** shrinks coefficients uniformly.
- **Lasso Regression** can shrink some coefficients to exactly zero, effectively performing feature selection.

Through these methods, we aim to find a model that balances bias and variance, leading to better performance on test data.

In [ ]:
```
# Import the required libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import make_column_transformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import Ridge, Lasso
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
from matplotlib.ticker import FuncFormatter
```

In [ ]:
```
# The dataset contains information on various credit-related factors. We

# Set font sizes in plots for better visibility
sns.set(font_scale = 2)
# Display all columns when printing dataframes
pd.set_option('display.max_columns', None)

# Load the Credit dataset
Credit = pd.read_csv("Credit.csv")


# Summary of the dataset
print(Credit.describe())
```

```
           Income         Limit      Rating       Cards         Age  \
count  400.000000    400.000000  400.000000  400.000000  400.000000
mean    45.218885   4735.600000  354.940000    2.957500   55.667500
std     35.244273   2308.198848  154.724143    1.371275   17.249807
min     10.354000    855.000000   93.000000    1.000000   23.000000
25%     21.007250   3088.000000  247.250000    2.000000   41.750000
50%     33.115500   4622.500000  344.000000    3.000000   56.000000
75%     57.470750   5872.750000  437.250000    4.000000   70.000000
max    186.634000  13913.000000  982.000000    9.000000   98.000000

        Education      Balance
count  400.000000   400.000000
mean    13.450000   520.015000
std      3.125207   459.758877
min      5.000000     0.000000
25%     11.000000    68.750000
50%     14.000000   459.500000
75%     16.000000   863.000000
max     20.000000  1999.000000
```

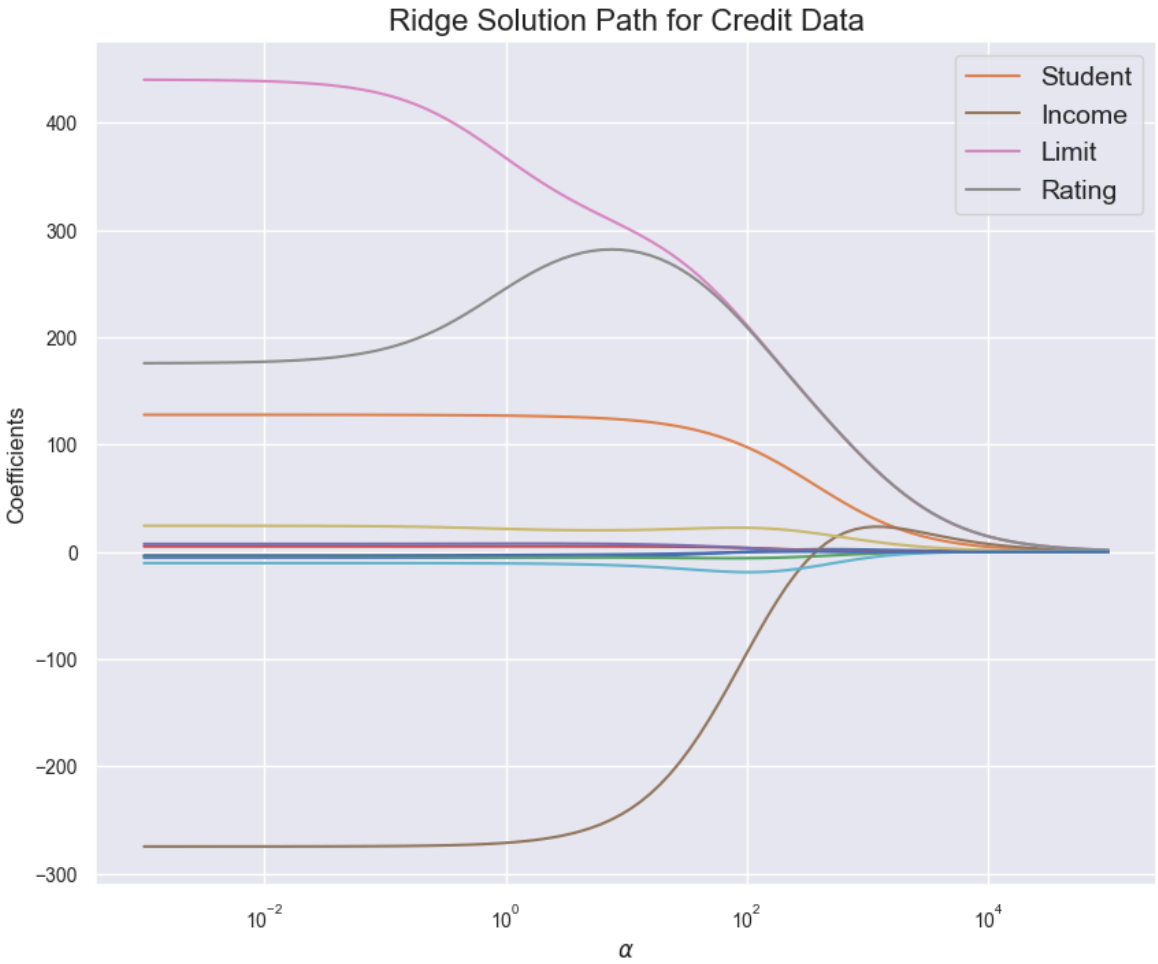
In [ ]:
```
# Response vector: Our target variable is 'Balance', and we will perform
y = Credit.Balance

# Preprocessing pipeline: Dummy encoding categorical variables and standa
```

```
cattr = make_column_transformer(
  (OneHotEncoder(drop = 'first'), ['Own', 'Student', 'Married', 'Region']
  remainder = 'passthrough'
)

# Standardize the data after encoding
pipe = Pipeline([('cat_tf', cattr), ('std_tf', StandardScaler())])
X = pipe.fit_transform(Credit.drop('Balance', axis = 1))
```

Ridge Regression Implementation

Theory: Ridge regression (L2 regularization) shrinks the coefficients by penalizing their magnitude. The penalty term is proportional to the square of the coefficients. As the regularization parameter alpha increases, the coefficients shrink more.

In [ ]:
```
# Ridge regression model
ridge_model = Ridge()

# Define a range of alpha values (regularization parameter)
alphas_ridge = np.logspace(-3, 5, 100)
ridge_coefs = []

# Train the model with different alpha values and store the coefficients
for a in alphas_ridge:
    ridge_model.set_params(alpha = a)
    ridge_model.fit(X, y)
    ridge_coefs.append(ridge_model.coef_)

# Visualize the Ridge solution path
plt.figure(figsize=(10, 8))
ax = plt.gca()
ax.plot(alphas_ridge, ridge_coefs)
ax.set_xscale("log")
ax.legend(labels=["_", "Student", "_", "_", "_", "Income", "Limit", "Rati
plt.xlabel(r"$\alpha$", fontsize = 12)
plt.ylabel("Coefficients", fontsize = 12)
plt.title("Ridge Solution Path for Credit Data", fontsize = 16)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.show()
```

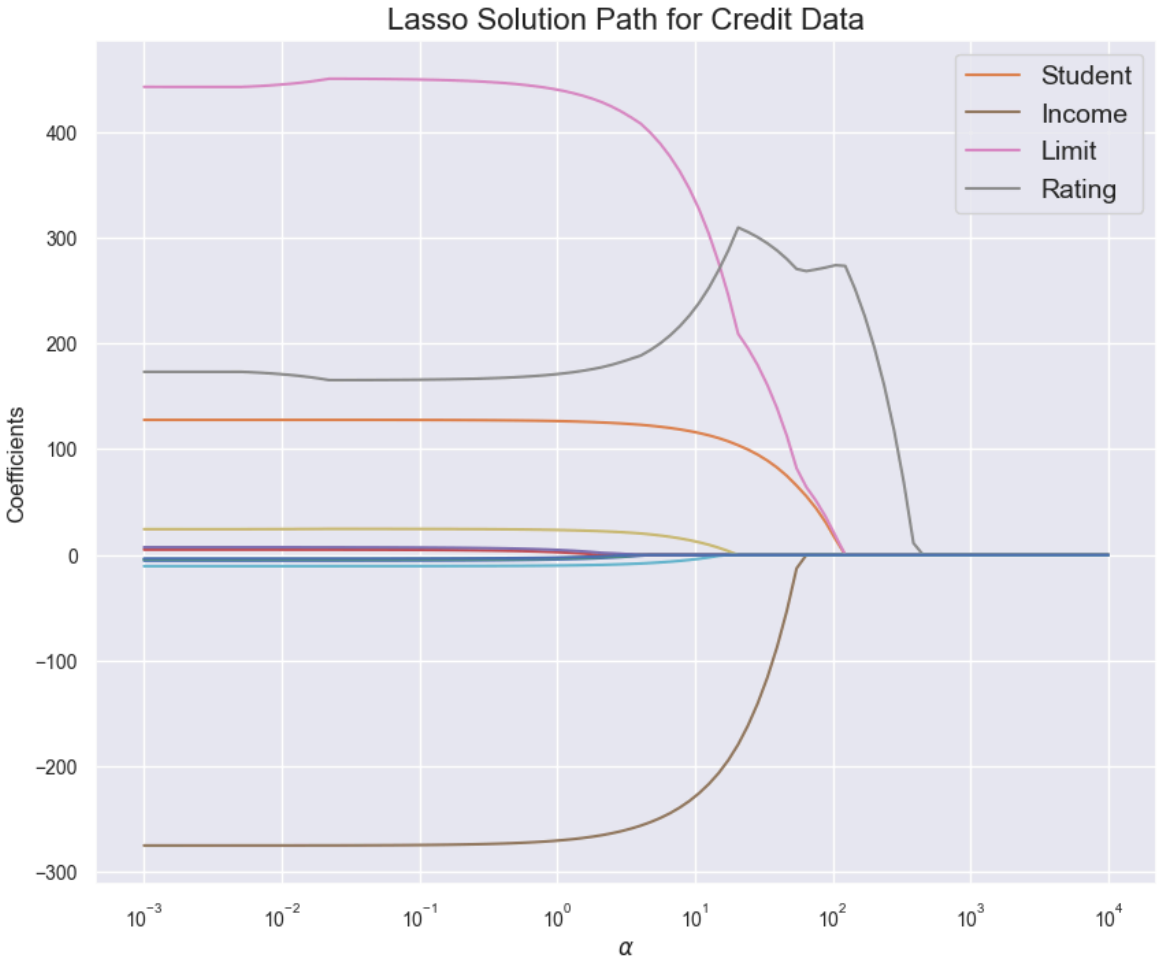


Lasso Regression Implementation

Theory: Lasso regression (L1 regularization) not only shrinks the coefficients but can also set some of them to zero, effectively **performing feature selection**. This can result in more interpretable models when there are many features.

In [ ]:
```
# Lasso regression model
lasso_model = Lasso()

# Define a range of alpha values for Lasso
alphas_lasso = np.logspace(-3, 4, 100)
lasso_coefs = []

# Train the model with different alpha values and store the coefficients
for a in alphas_lasso:
    lasso_model.set_params(alpha = a)
    lasso_model.fit(X, y)
    lasso_coefs.append(lasso_model.coef_)

# Visualize the Lasso solution path
plt.figure(figsize=(10, 8))
ax = plt.gca()
ax.plot(alphas_lasso, lasso_coefs)
ax.set_xscale("log")
ax.legend(labels=["_", "Student", "_", "_", "_", "Income", "Limit", "Rati
plt.xlabel(r"$\alpha$", fontsize = 12)
plt.ylabel("Coefficients", fontsize = 12)
plt.title("Lasso Solution Path for Credit Data", fontsize = 16)
```

```
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.show()
```

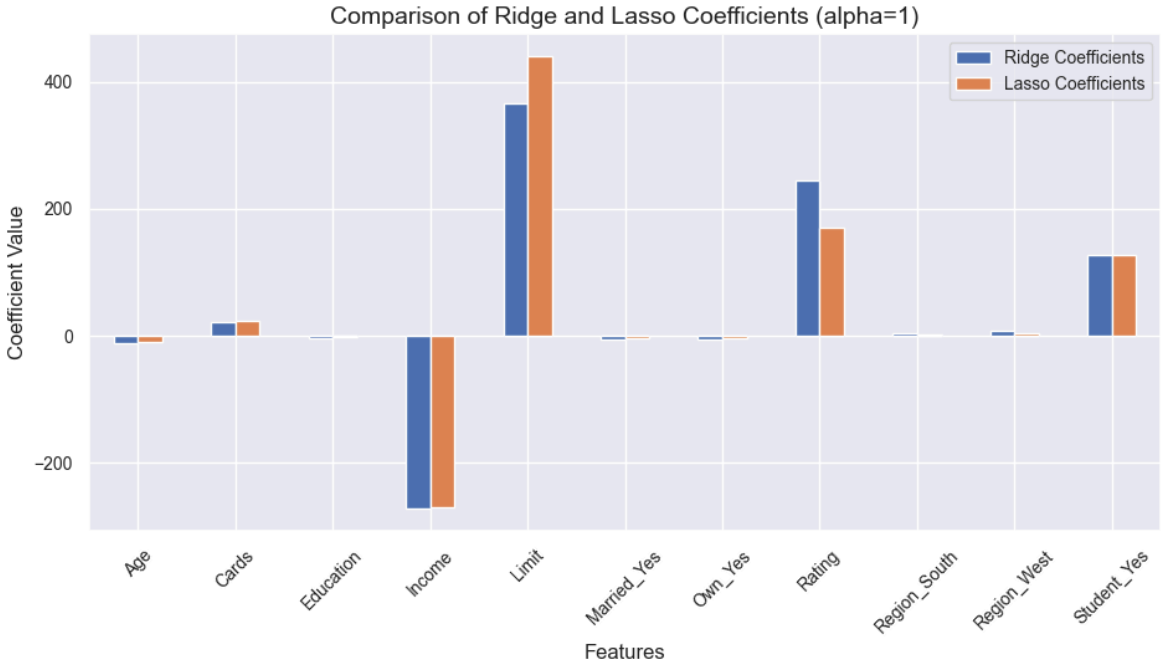


Comparison of Ridge and Lasso Regression

Theory: While Ridge shrinks coefficients toward zero without eliminating any, Lasso can set coefficients exactly to zero, making it more useful when feature selection is needed. Both methods add bias to reduce variance (i.e., overfitting), but they differ in how they handle feature selection and coefficient shrinkage.

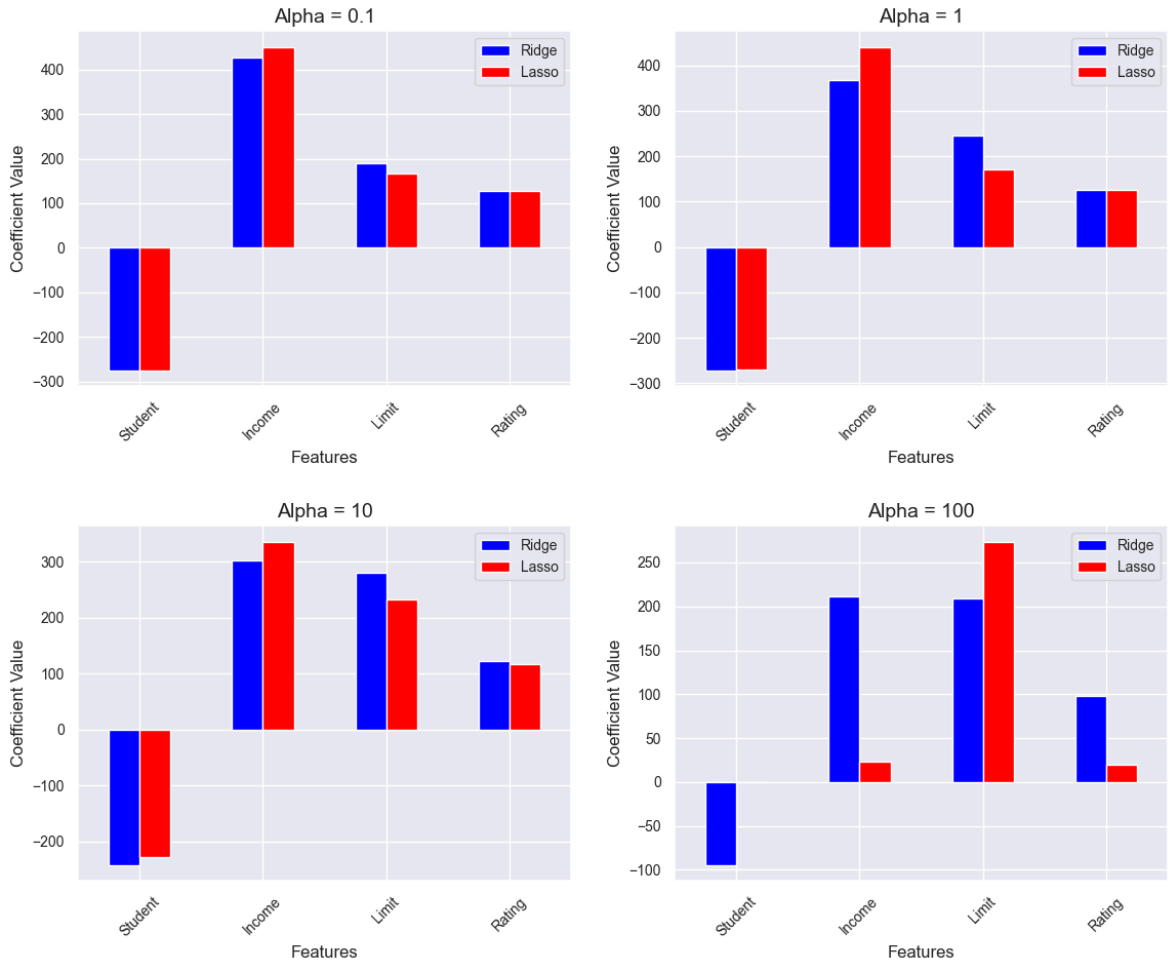
In [ ]:
```
# Comparison of Ridge and Lasso Regression Coefficients

# Set alpha for both Ridge and Lasso models
fixed_alpha = 1

# Train both models with the same alpha value
ridge_model.set_params(alpha=fixed_alpha)
ridge_model.fit(X, y)
lasso_model.set_params(alpha=fixed_alpha)
lasso_model.fit(X, y)

# Extract coefficients for all features (including categorical variables)
ridge_coeffs = ridge_model.coef_
lasso_coeffs = lasso_model.coef_

# Get feature names after preprocessing (OneHotEncoder + original feature
# Since we used OneHotEncoder, we need to recreate the correct feature na
```

```
categorical_features = pipe.named_steps['cat_tf'].transformers_[0][1].get
numerical_features = Credit.drop('Balance', axis=1).columns.drop(['Own',

# Combine categorical and numerical feature names
all_feature_names = np.concatenate([categorical_features, numerical_featu

# Create a DataFrame to hold coefficients
coefficients_df = pd.DataFrame({
    "Features": all_feature_names,
    "Ridge Coefficients": ridge_coeffs,
    "Lasso Coefficients": lasso_coeffs
})

# Sort by feature name for better comparison
coefficients_df = coefficients_df.sort_values(by="Features")

# Plot a comparison of Ridge and Lasso coefficients with smaller plot val
coefficients_df.plot(x="Features", y=["Ridge Coefficients", "Lasso Coeffi

# Add title and labels with smaller fonts
plt.title(f'Comparison of Ridge and Lasso Coefficients (alpha={fixed_alph
plt.xlabel('Features', fontsize=12)
plt.ylabel('Coefficient Value', fontsize=12)

# Rotate the x-axis labels for better readability, with smaller font size
plt.xticks(rotation=45, fontsize=10)
plt.yticks(fontsize=10)

# Adjust legend size
plt.legend(['Ridge Coefficients', 'Lasso Coefficients'], fontsize=10)

# Tight layout for better space management
plt.tight_layout()
plt.show()
```



In [ ]:
```
# Define four alpha values for comparison
alphas = [0.1, 1, 10, 100]

# Selected features for comparison (subset of features for readability)
```

```
selected_features = ["Student", "Income", "Limit", "Rating"]

# Prepare an empty dictionary to hold the coefficients for each alpha
coefficients = {'Ridge': {}, 'Lasso': {}}

# Train Ridge and Lasso models with the selected alphas and store the sel
for alpha in alphas:
    # Ridge
    ridge_model = Ridge(alpha=alpha)
    ridge_model.fit(X, y)
    ridge_coeffs = ridge_model.coef_

    # Lasso
    lasso_model = Lasso(alpha=alpha)
    lasso_model.fit(X, y)
    lasso_coeffs = lasso_model.coef_

    # Store the coefficients for selected features
    coefficients['Ridge'][alpha] = ridge_coeffs[[5, 6, 7, 1]]  # Extract
    coefficients['Lasso'][alpha] = lasso_coeffs[[5, 6, 7, 1]]

# Create a 2x2 subplot grid
fig, axs = plt.subplots(2, 2, figsize=(12, 10))

# Flatten the axes array to easily index the subplots
axs = axs.flatten()

# Plot for each alpha value
for i, alpha in enumerate(alphas):
    # Create a DataFrame with Ridge and Lasso coefficients for the select
    coef_df = pd.DataFrame({
        "Features": selected_features,
        "Ridge Coefficients": coefficients['Ridge'][alpha],
        "Lasso Coefficients": coefficients['Lasso'][alpha]
    })

    # Plot Ridge and Lasso coefficients as bars for each alpha
    coef_df.plot(x="Features", y=["Ridge Coefficients", "Lasso Coefficien

    # Set title and labels
    axs[i].set_title(f'Alpha = {alpha}', fontsize=14)
    axs[i].set_xlabel('Features', fontsize=12)
    axs[i].set_ylabel('Coefficient Value', fontsize=12)
    axs[i].tick_params(axis='x', labelrotation=45, labelsize=10)
    axs[i].tick_params(axis='y', labelsize=10)
    axs[i].legend(['Ridge', 'Lasso'], fontsize=10)

# Adjust layout
plt.tight_layout()
plt.show()
```



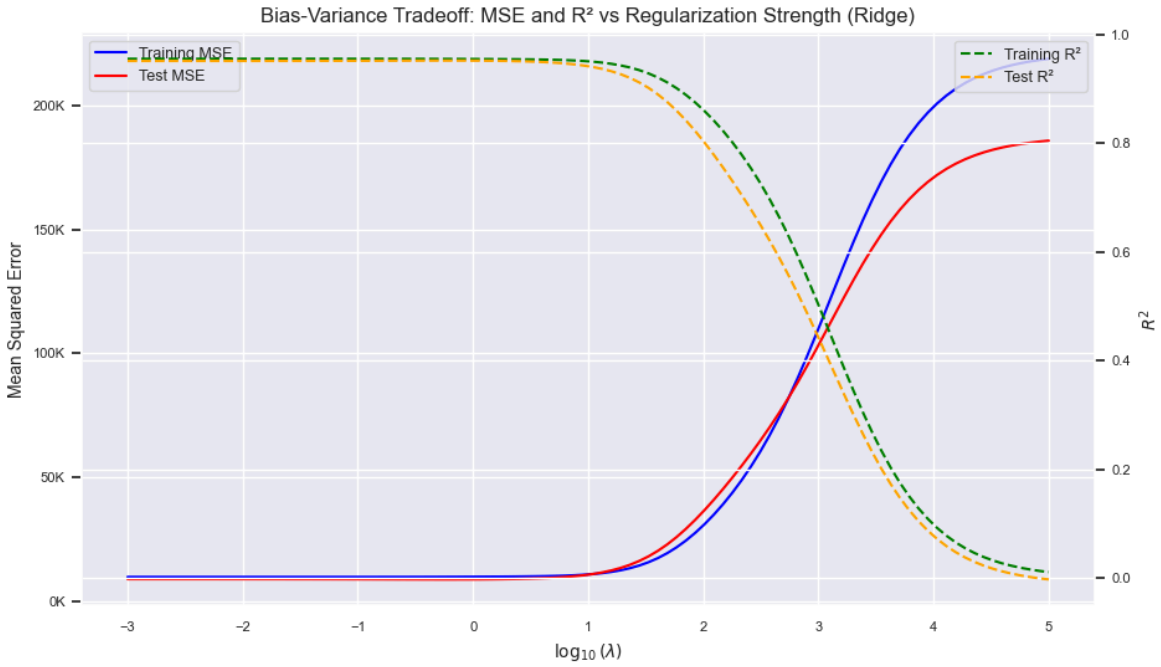
Bias - Variance Tradeoff

Theory: The bias-variance tradeoff is clearly demonstrated in both Ridge and Lasso regression. As we increase regularization strength, the variance of the model decreases, but this comes at the cost of increased bias. The key is finding the right balance, often through techniques like cross-validation, to ensure that the model generalizes well to unseen data. By visualizing the MSE and R^2 as functions of λ, you can see the interplay between bias and variance and how each method (Ridge or Lasso) reacts to increasing regularization.

In [ ]:
```
# Function to format MSE in thousands
def thousands(x, pos):
    return '%1.0fK' % (x * 1e-3)
```

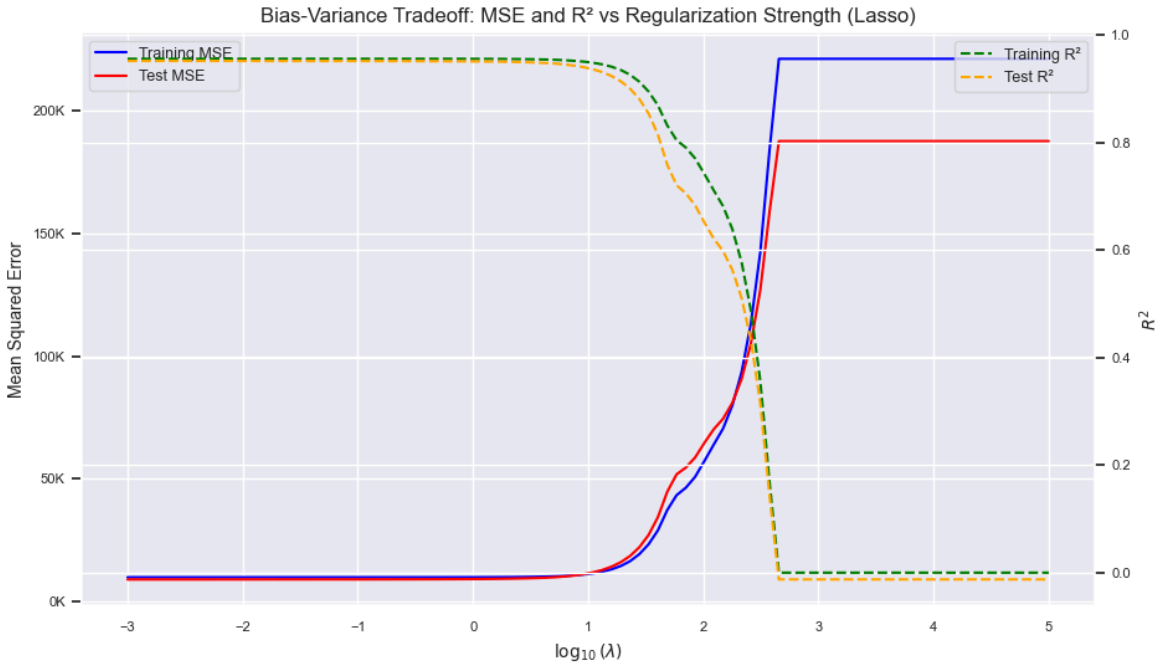
In [ ]:
```
# Split the Credit data into training and test sets (using the X and y fr
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,

# Define a range of alpha values (regularization strength)
alphas = np.logspace(-3, 5, 100)

# Store MSE and R² values for training and test sets
ridge_train_mse, ridge_test_mse = [], []
ridge_train_r2, ridge_test_r2 = [], []

# Train Ridge regression with different alpha values and calculate metric
for alpha in alphas:
```

```
    ridge_model = Ridge(alpha=alpha)
    ridge_model.fit(X_train, y_train)

    # Predict on training and test sets
    y_train_pred = ridge_model.predict(X_train)
    y_test_pred = ridge_model.predict(X_test)

    # Calculate Mean Squared Error (MSE)
    ridge_train_mse.append(mean_squared_error(y_train, y_train_pred))
    ridge_test_mse.append(mean_squared_error(y_test, y_test_pred))

    # Calculate R²
    ridge_train_r2.append(r2_score(y_train, y_train_pred))
    ridge_test_r2.append(r2_score(y_test, y_test_pred))

# Create the figure and axis objects
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot MSE on the left y-axis
ax1.plot(np.log10(alphas), ridge_train_mse, label='Training MSE', color='
ax1.plot(np.log10(alphas), ridge_test_mse, label='Test MSE', color='red')
ax1.set_xlabel(r'$\log_{10}(\lambda)$', fontsize=10)
ax1.set_ylabel('Mean Squared Error', fontsize=10)
ax1.legend(loc='upper left', fontsize=9)
ax1.yaxis.set_major_formatter(FuncFormatter(thousands))

# Create a second y-axis for R²
ax2 = ax1.twinx()
ax2.plot(np.log10(alphas), ridge_train_r2, label='Training R²', color='gr
ax2.plot(np.log10(alphas), ridge_test_r2, label='Test R²', color='orange'
ax2.set_ylabel(r'$R^2$', fontsize=10)
ax2.legend(loc='upper right', fontsize=9)

# Adjust tick sizes
ax1.tick_params(axis='x', labelsize=8)
ax1.tick_params(axis='y', labelsize=8)
ax2.tick_params(axis='y', labelsize=8)

# Add a title and adjust layout
plt.title('Bias-Variance Tradeoff: MSE and R² vs Regularization Strength
plt.tight_layout()
plt.show()
```



In [ ]:
```
# Split the Credit data into training and test sets (using the X and y fr
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,

# Define a range of alpha values (regularization strength)
alphas = np.logspace(-3, 5, 100)

# Store MSE and R² values for training and test sets
lasso_train_mse, lasso_test_mse = [], []
lasso_train_r2, lasso_test_r2 = [], []

# Train Lasso regression with different alpha values and calculate metric
for alpha in alphas:
    lasso_model = Lasso(alpha=alpha)
    lasso_model.fit(X_train, y_train)

    # Predict on training and test sets
    y_train_pred = lasso_model.predict(X_train)
    y_test_pred = lasso_model.predict(X_test)

    # Calculate Mean Squared Error (MSE)
    lasso_train_mse.append(mean_squared_error(y_train, y_train_pred))
    lasso_test_mse.append(mean_squared_error(y_test, y_test_pred))

    # Calculate R²
    lasso_train_r2.append(r2_score(y_train, y_train_pred))
    lasso_test_r2.append(r2_score(y_test, y_test_pred))

# Create the figure and axis objects
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot MSE on the left y-axis
ax1.plot(np.log10(alphas), lasso_train_mse, label='Training MSE', color='
ax1.plot(np.log10(alphas), lasso_test_mse, label='Test MSE', color='red')
ax1.set_xlabel(r'$\log_{10}(\lambda)$', fontsize=10)
ax1.set_ylabel('Mean Squared Error', fontsize=10)
ax1.legend(loc='upper left', fontsize=9)
ax1.yaxis.set_major_formatter(FuncFormatter(thousands))

# Create a second y-axis for R²
```

```
ax2 = ax1.twinx()
ax2.plot(np.log10(alphas), lasso_train_r2, label='Training R²', color='gr
ax2.plot(np.log10(alphas), lasso_test_r2, label='Test R²', color='orange'
ax2.set_ylabel(r'$R^2$', fontsize=10)
ax2.legend(loc='upper right', fontsize=9)

# Adjust tick sizes
ax1.tick_params(axis='x', labelsize=8)
ax1.tick_params(axis='y', labelsize=8)
ax2.tick_params(axis='y', labelsize=8)

# Add a title and adjust layout
plt.title('Bias-Variance Tradeoff: MSE and R² vs Regularization Strength
plt.tight_layout()
plt.show()
```



Observations and Conclusions

- Ridge Regression shrinks the coefficients gradually, with all features retaining non-zero values, even at high alpha values.
- Lasso Regression sets some coefficients to zero, effectively performing feature selection. This is evident for features like 'Student', where the coefficient becomes zero for larger alpha values.
- At a fixed alpha value, Ridge tends to shrink all coefficients uniformly, while Lasso pushes certain coefficients to zero.
- In practice, Lasso may be preferred when feature selection is important, while Ridge is more suitable when the goal is to shrink coefficients without eliminating features.