Assignment 1

Mehreen Ali Gillani

2024-06-10

Question 1

Consider the following data set: The data set Auto, which is part of the ISLR package, contains information on various characteristics of different makes and models of automobiles.

```
import pandas as pd
import numpy as np
#!pip install nbformat
import nbformat
from nbclient import NotebookClient
#import ISLP
from ISLP import load_data
data = load_data("Auto")
data.head()
```

name	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin
chevrolet chevelle malibu	18.0	8	307.0	130	3504	12.0	70	1
buick skylark 320	15.0	8	350.0	165	3693	11.5	70	1
plymouth satellite	18.0	8	318.0	150	3436	11.0	70	1
amc rebel sst	16.0	8	304.0	150	3433	12.0	70	1
ford torino	17.0	8	302.0	140	3449	10.5	70	1

Remove missing values.

```
# Check for missing values
print("Missing values before removal:")
print(data.isnull().sum())
```

```python
# Remove rows with any missing values
data_clean = data.dropna()

# Check shape after removal
print(f"\nOriginal shape: {data.shape}")
print(f"Cleaned shape: {data_clean.shape}")
print(f"Rows removed: {len(data) - len(data_clean)}")
```

```
Missing values before removal:
mpg             0
cylinders       0
displacement    0
horsepower      0
weight          0
acceleration    0
year            0
origin          0
dtype: int64

Original shape: (392, 8)
Cleaned shape: (392, 8)
Rows removed: 0
```

a) Which of the predictors are quantitative, and which are qualitative?

```python
# Identify quantitative and qualitative predictors
quantitative_predictors = data_clean.select_dtypes(include=['number']).columns.tolist()
qualitative_predictors = data_clean.select_dtypes(include=['object', 'category']).columns.to
print("Quantitative Predictors:", quantitative_predictors)
print("Qualitative Predictors:", qualitative_predictors)
```

```
Quantitative Predictors: ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'accele
Qualitative Predictors: []
```

Find continous and categorical variables among the predictors.

find nunique for year and origin

```python
nunique_values = data_clean.nunique()
print(nunique_values)
```

```
mpg             127
cylinders         5
displacement     81
horsepower       93
weight          346
acceleration     95
year             13
origin            3
dtype: int64
```

Name is the only qualitative predictor, while the rest are quantitative predictors.

1. cylinders has 5 unique values, so it can be considered categorical.
2. Year has 13 unique values so it will be considered ordinal.
3. Origin has 3 unique values, making it categorical. Rest all are continuous variables.

b) What is the range of each quantitative predictor? You can answer this using the min() and max() methods in numpy.

```
for col in quantitative_predictors:
    col_min = data_clean[col].min()
    col_max = data_clean[col].max()
    print(f"{col}: Min = {col_min}, Max = {col_max}")
```

```
mpg: Min = 9.0, Max = 46.6
cylinders: Min = 3, Max = 8
displacement: Min = 68.0, Max = 455.0
horsepower: Min = 46, Max = 230
weight: Min = 1613, Max = 5140
acceleration: Min = 8.0, Max = 24.8
year: Min = 70, Max = 82
origin: Min = 1, Max = 3
```

c) What is the mean and standard deviation of each quantitative predictor?

```
for col in quantitative_predictors:
    col_mean = data_clean[col].mean()
    col_std = data_clean[col].std()
    print(f"{col}: Mean = {col_mean}, Std Dev = {col_std}")
```

```
mpg: Mean = 23.445918367346938, Std Dev = 7.8050074865717995
cylinders: Mean = 5.471938775510204, Std Dev = 1.7057832474527845
displacement: Mean = 194.41198979591837, Std Dev = 104.64400390890466
horsepower: Mean = 104.46938775510205, Std Dev = 38.49115993282849
weight: Mean = 2977.5841836734694, Std Dev = 849.4025600429492
acceleration: Mean = 15.541326530612244, Std Dev = 2.758864119188082
year: Mean = 75.9795918367347, Std Dev = 3.6837365435778295
origin: Mean = 1.5765306122448979, Std Dev = 0.8055181834183056
```

d) Now remove the 10th through 85th observations. What is the mean and standard deviation of each quantitative predictor in the subset of the data that remains?

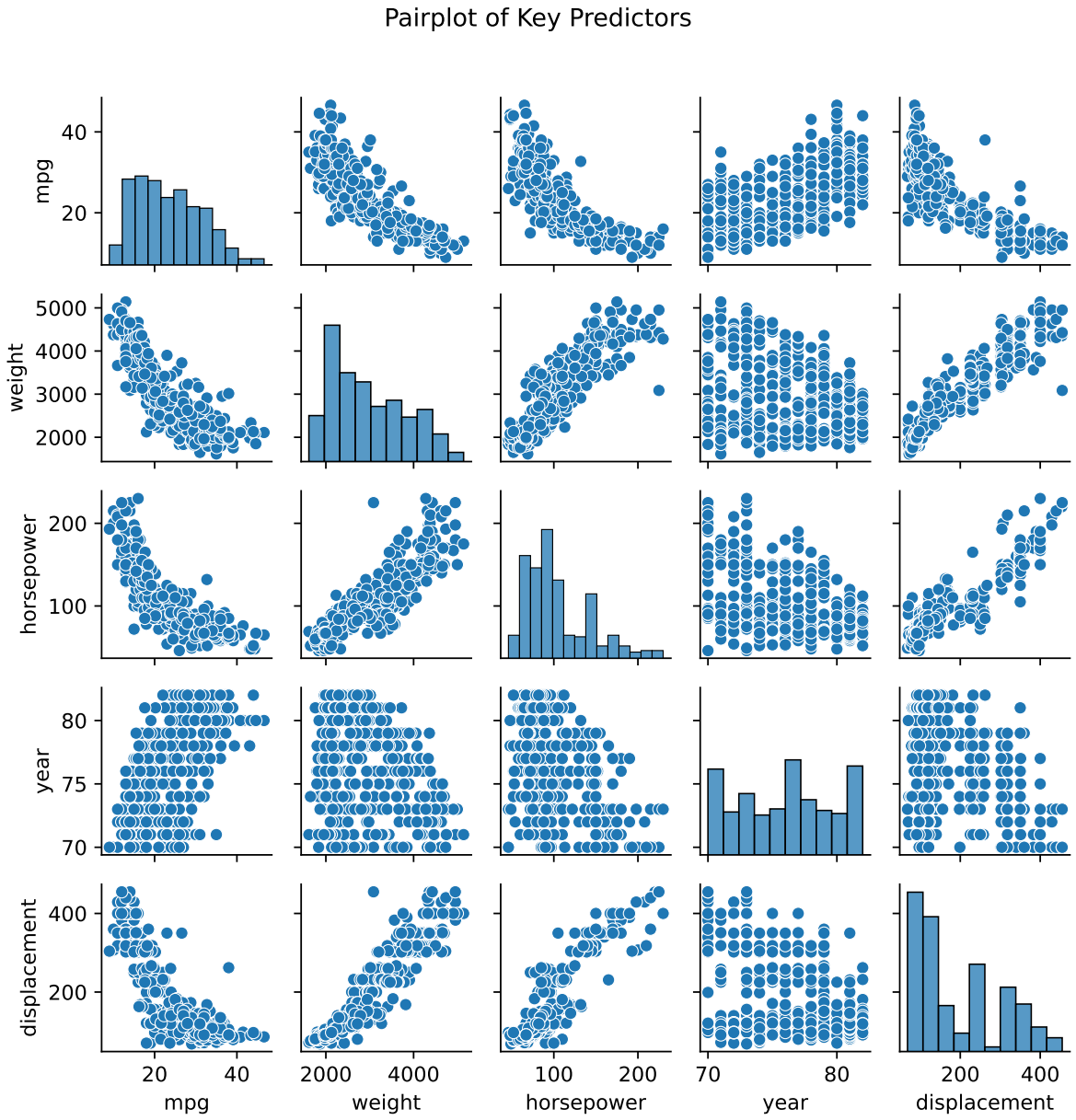
```
data_subset = data_clean.drop(data_clean.index[9:85])
for col in quantitative_predictors:
    col_mean = data_subset[col].mean()
    col_std = data_subset[col].std()
    print(f"{col} (Subset): Mean = {col_mean}, Std Dev = {col_std}")
```

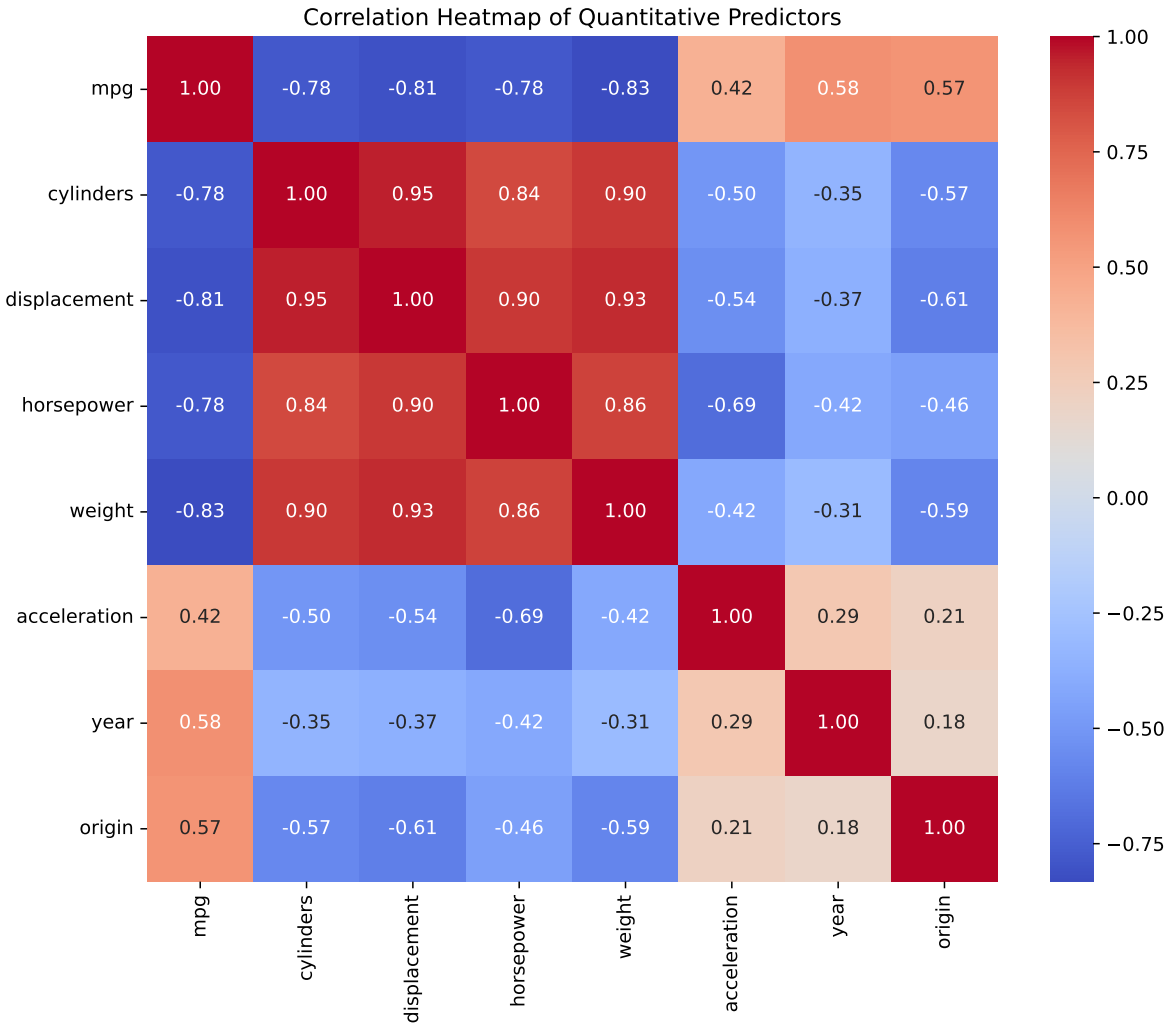
```
mpg (Subset): Mean = 25.04163701067616, Std Dev = 7.91287442060972
cylinders (Subset): Mean = 5.274021352313167, Std Dev = 1.6321549688677217
displacement (Subset): Mean = 179.37366548042704, Std Dev = 95.51289675679097
horsepower (Subset): Mean = 98.7153024911032, Std Dev = 33.8227113152057
weight (Subset): Mean = 2881.505338078292, Std Dev = 792.5484938790652
acceleration (Subset): Mean = 15.73879003558719, Std Dev = 2.5701912023177935
year (Subset): Mean = 77.50889679715303, Std Dev = 2.9894025669301008
origin (Subset): Mean = 1.6334519572953736, Std Dev = 0.8307603049561275
```

e) Using the full data set, investigate the predictors graphically, using scatterplots or other tools of your choice. Create some plots highlighting the relationships among the predictors. Comment on your findings

```
import matplotlib.pyplot as plt
import seaborn as sns
# Get the pairplot object
# Select only important predictors
key_predictors = ['mpg', 'weight', 'horsepower', 'year', 'displacement']
sns.pairplot(data_clean[key_predictors], height=1.5,  # Smaller height per subplot
             aspect=1)
plt.suptitle("Pairplot of Key Predictors", y=1.02)
plt.tight_layout()
plt.show()
```

```
# Correlation heatmap
plt.figure(figsize=(10, 8))
correlation_matrix = data_clean[quantitative_predictors].corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Correlation Heatmap of Quantitative Predictors")
plt.show()
```





Correlation Analysis Summary

Target Variable: MPG (Fuel Efficiency)

Strong Negative Correlations:

MPG decreases significantly with higher values of:

- Cylinders (-0.78)
- Displacement (-0.81)
- Horsepower (-0.78)
- Weight (-0.83)

Positive Correlations: MPG improves with:

- Model year (+0.58) - newer cars are more fuel efficient
- Acceleration (+0.42) - faster acceleration correlates with better MPG
- Origin (+0.57) - specific origin cars tend to be more fuel efficient

Engine Characteristics (Cylinders & Displacement) Strong Positive Relationships:

- Cylinders Displacement (+0.95)
- Cylinders Horsepower (+0.84)
- Displacement Horsepower (+0.90)
- All three show strong positive correlation with Weight (+0.90+)

Negative Trends Over Time:

- Cylinders negatively correlate with Year (-0.35) - newer cars have fewer cylinders
- Displacement negatively correlates with Year (-0.37) - engine sizes decreasing over time
- Both show negative correlation with Origin (-0.57) - specific origin cars favor smaller engines#Weight Relationships

Weight increases with:

- Engine size (Cylinders, Displacement, Horsepower) Weight decreases with:
- Model Year (-0.31) - newer cars are lighter
- Origin (-0.59) - non-American cars are lighter
- Acceleration (-0.42) - lighter cars accelerate faster

Acceleration shows moderate positive correlation with:

- Model Year (+0.29) - newer cars accelerate better
- Origin (+0.21) - non-American cars have better acceleration
- Weakest correlations observed with Acceleration, suggesting it's influenced by multiple factors beyond engine specs alone

f) Suppose that we wish to predict gas mileage (mpg) on the basis of the other variables. Do your plots suggest that any of the other variables might be useful in predicting mpg? Justify your answer. Yes, the plots and correlation analysis suggest that several variables could be useful in predicting gas mileage (mpg). Specifically:

1. Cylinders: Strong negative correlation with mpg (-0.78) indicates that cars with more cylinders tend to have lower fuel efficiency.
2. Displacement: Also shows a strong negative correlation with mpg (-0.81), suggesting that larger engine sizes are associated with lower gas mileage.
3. Horsepower: With a strong negative correlation (-0.78), higher horsepower engines tend to consume more fuel, leading to lower mpg.
4. Weight: The strongest negative correlation with mpg (-0.83) indicates that heavier cars generally have lower fuel efficiency.
5. Model Year: A positive correlation (+0.58) suggests that newer cars are more fuel efficient, likely due to advancements in technology and design.
6. Acceleration: A moderate positive correlation (+0.42) indicates that cars with better acceleration may also have better fuel efficiency.
7. Origin: The positive correlation (+0.57) suggests that cars from certain origins tend to be more fuel efficient. Overall, the strong correlations of cylinders, displacement, horsepower, and weight with mpg indicate that these variables are likely to be significant predictors of gas mileage. Additionally, model year, acceleration, and origin also show meaningful relationships with mpg that could enhance predictive models.

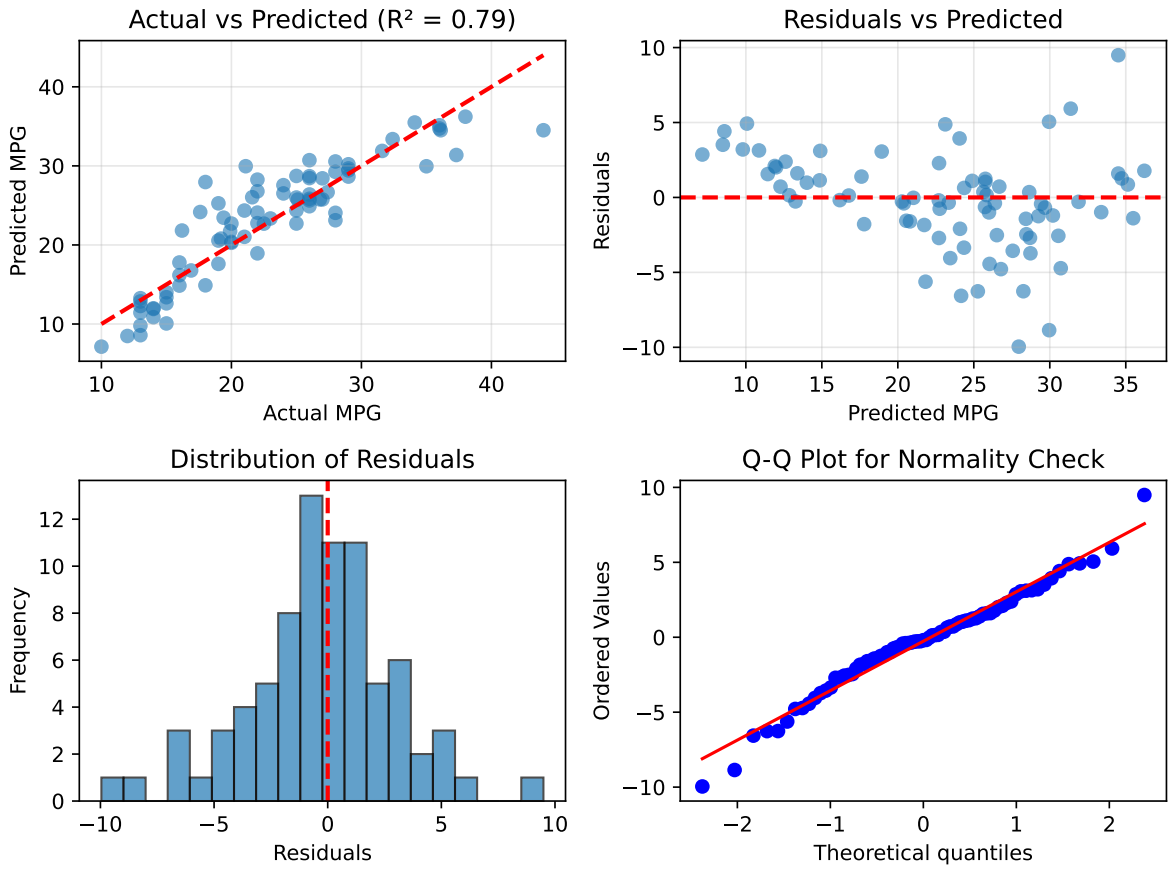
```
# Linear Regression Analysis for MPG Prediction

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
import scipy.stats as stats

# Select numeric columns
numeric_cols = data_clean.select_dtypes(include=['number']).columns.tolist()
print("Numeric columns:", numeric_cols)

# Remove 'mpg' from predictors
```

```python
if 'mpg' in numeric_cols:
    numeric_cols.remove('mpg')

X = data_clean[numeric_cols]
y = data_clean['mpg']

print(f"\nUsing {len(numeric_cols)} numeric predictors:")
print(numeric_cols)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, shuffle=True
)

# Train model
model = LinearRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Calculate residuals (ADD THIS LINE!)
residuals = y_test - y_pred

# Metrics
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("=" * 50)
print("MODEL PERFORMANCE EVALUATION")
print("=" * 50)
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f} MPG")
print(f"Mean Absolute Error (MAE): {mae:.2f} MPG")
print(f"R-squared (R²): {r2:.2f}")
print(f"Adjusted R²: {1 - (1-r2)*(len(y_test)-1)/(len(y_test)-X.shape[1]-1):.2f}")

# Compare to baseline
baseline_pred = np.full_like(y_test, y_train.mean())
baseline_mse = mean_squared_error(y_test, baseline_pred)
```

```
print(f"\nBaseline (predict mean): MSE = {baseline_mse:.2f}")
print(f"Improvement over baseline: {(1 - mse/baseline_mse)*100:.1f}%")

# Residual statistics
print(f"\nResidual Statistics:")
print(f"Mean residual: {residuals.mean():.2f}")
print(f"Std of residuals: {residuals.std():.2f}")
print(f"Max residual: {residuals.max():.2f}")
print(f"Min residual: {residuals.min():.2f}")

# Visualization
fig, axes = plt.subplots(2, 2, figsize=(8, 6))

# 1. Actual vs Predicted
axes[0, 0].scatter(y_test, y_pred, alpha=0.6)
axes[0, 0].plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],
                'r--', lw=2)
axes[0, 0].set_xlabel('Actual MPG')
axes[0, 0].set_ylabel('Predicted MPG')
axes[0, 0].set_title(f'Actual vs Predicted (R² = {r2:.2f})')
axes[0, 0].grid(True, alpha=0.3)

# 2. Residuals vs Predicted
axes[0, 1].scatter(y_pred, residuals, alpha=0.6)
axes[0, 1].axhline(y=0, color='r', linestyle='--', lw=2)
axes[0, 1].set_xlabel('Predicted MPG')
axes[0, 1].set_ylabel('Residuals')
axes[0, 1].set_title('Residuals vs Predicted')
axes[0, 1].grid(True, alpha=0.3)

# 3. Distribution of residuals
axes[1, 0].hist(residuals, bins=20, edgecolor='black', alpha=0.7)
axes[1, 0].axvline(x=0, color='r', linestyle='--', lw=2)
axes[1, 0].set_xlabel('Residuals')
axes[1, 0].set_ylabel('Frequency')
axes[1, 0].set_title('Distribution of Residuals')

# 4. Q-Q plot for normality
stats.probplot(residuals, dist="norm", plot=axes[1, 1])
axes[1, 1].set_title('Q-Q Plot for Normality Check')

plt.tight_layout()
```

```
plt.show()

# Additional: Show feature coefficients
print("\n" + "=" * 50)
print("FEATURE COEFFICIENTS")
print("=" * 50)
coefficients = pd.DataFrame({
    'Feature': X.columns,
    'Coefficient': model.coef_,
    'Abs_Effect': np.abs(model.coef_)
})
coefficients = coefficients.sort_values('Abs_Effect', ascending=False)
print(coefficients)
```

```
Numeric columns: ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration'

Using 7 numeric predictors:
['cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'year', 'origin']
==================================================
MODEL PERFORMANCE EVALUATION
==================================================
Mean Squared Error (MSE): 10.71
Root Mean Squared Error (RMSE): 3.27 MPG
Mean Absolute Error (MAE): 2.42 MPG
R-squared (R²): 0.79
Adjusted R²: 0.77

Baseline (predict mean): MSE = 51.62
Improvement over baseline: 79.3%

Residual Statistics:
Mean residual: -0.27
Std of residuals: 3.28
Max residual: 9.49
Min residual: -9.96
```




```
==================================================
FEATURE COEFFICIENTS
==================================================
        Feature  Coefficient  Abs_Effect
6        origin     1.613457    1.613457
5          year     0.767743    0.767743
0     cylinders    -0.345789    0.345789
4  acceleration     0.037950    0.037950
2    horsepower    -0.021302    0.021302
1  displacement     0.015109    0.015109
3        weight    -0.006142    0.006142
```

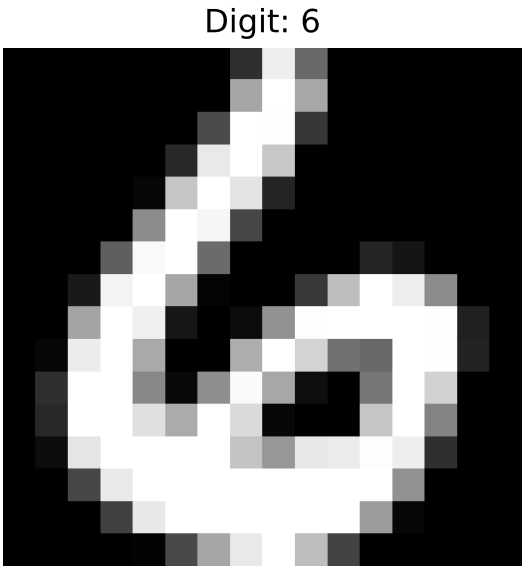
Findings: . The high R^2 of 0.79 confirms these variables collectively explain most MPG variation. The negative coefficients for weight/horsepower and positive for year align with our correlation analysis. . The MSE of 10.71 (RMSE 3.27) means our predictions are typically within 3-4 MPG of actual values, which is quite good for automotive fuel economy prediction.

Part 2 ## Question 2 a) Download and load the training and test data sets using pandas. Make sure to load all of the data (there is no header) The zeroth column corresponds to the class label, a digit from 0-9, and the columns 1 to 256 correspond to a grayscale value from -1 to 1. Select the first entry in the training set, resize it to 16x16, and plot the image (you can use plt.imshow()).

```
df_train = pd.read_csv("https://raw.githubusercontent.com/georgehagstrom/DATA622Spring2026/re
df_test = pd.read_csv("https://raw.githubusercontent.com/georgehagstrom/DATA622Spring2026/re

print(df_train.head())  # Show first 5 rows
print("\nColumns:", df_train.columns.tolist())

# Split space-separated strings into columns
df_train = df_train[0].str.split(expand=True)
df_test = df_test[0].str.split(expand=True)

# 2. CONVERT TO NUMERIC
df_train = df_train.astype(float)
df_test = df_test.astype(float)

# 3. EXTRACT FEATURES AND LABELS
y_train = df_train.iloc[:, 0].astype(int)
X_train = df_train.iloc[:, 1:257]

y_test = df_test.iloc[:, 0].astype(int)
X_test = df_test.iloc[:, 1:257]

print(f"Training: {X_train.shape[0]} samples, {X_train.shape[1]} features")
print(f"Test: {X_test.shape[0]} samples, {X_test.shape[1]} features")

# 4. VISUALIZE FIRST IMAGE
first_label = y_train.iloc[0]
first_image = X_train.iloc[0].values.reshape(16, 16)

plt.imshow(first_image, cmap='gray')
plt.title(f'Digit: {first_label}')
plt.axis('off')
plt.show()
```

```
                                                   0
0  6.0000 -1.0000 -1.0000 -1.0000 -1.0000 -1.0000...
1  5.0000 -1.0000 -1.0000 -1.0000 -0.8130 -0.6710...
```

```
2  4.0000 -1.0000 -1.0000 -1.0000 -1.0000 -1.0000...
3  7.0000 -1.0000 -1.0000 -1.0000 -1.0000 -1.0000...
4  3.0000 -1.0000 -1.0000 -1.0000 -1.0000 -1.0000...

Columns: [0]
Training: 7291 samples, 256 features
Test: 2007 samples, 256 features
```



b) The following code imports the kNN classification function from scikit-learn as well as an accuracy function, trains a kNN classifier, and tests its accuracy on hypothetical training and testing data:

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score


# Train KNN model
knn = KNeighborsClassifier(n_neighbors=4)
knn.fit(X_train, y_train)

# Predict on training data
y_pred_train = knn.predict(X_train)
train_accuracy = accuracy_score(y_train, y_pred_train)
print(f'Training Accuracy: {train_accuracy:.4f}')
```

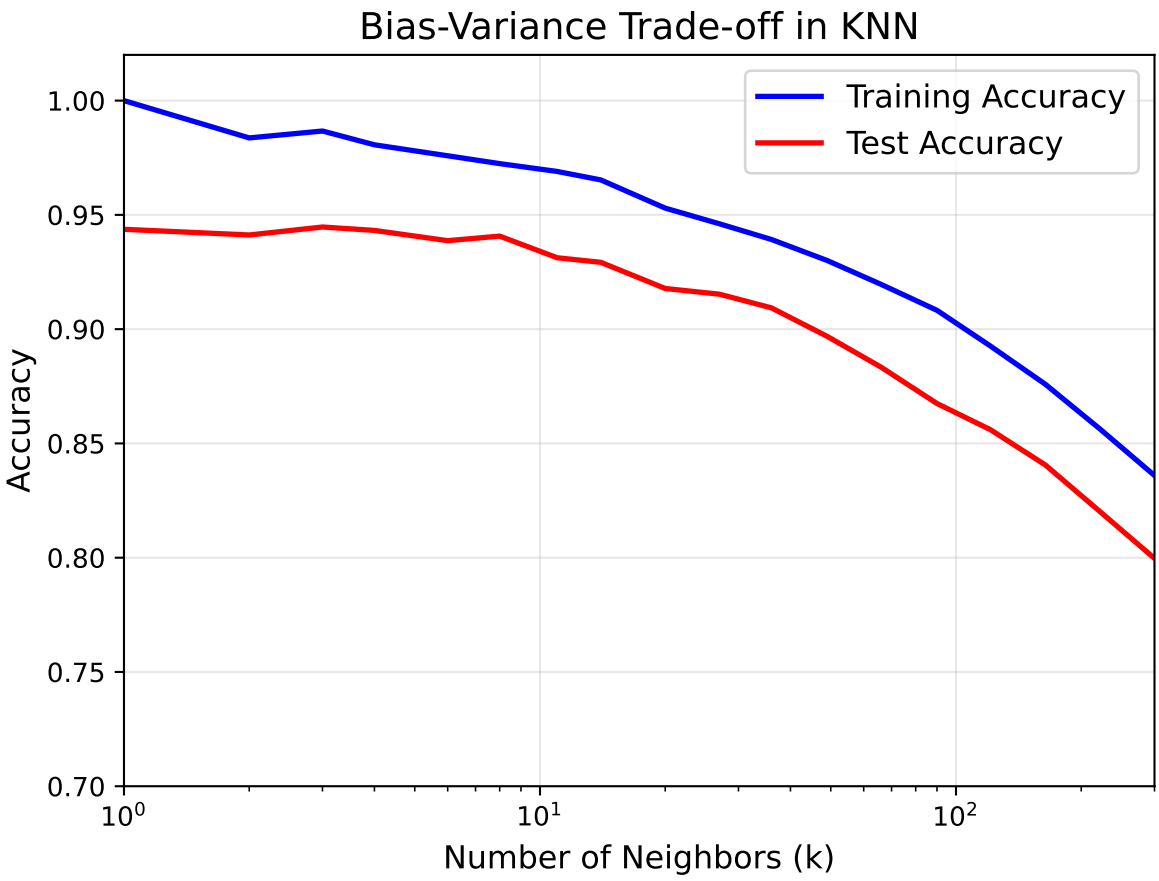
```
# Predict on test data (using your test variables)
y_pred_test = knn.predict(X_test)
test_accuracy = accuracy_score(y_test, y_pred_test)
print(f'Test Accuracy: {test_accuracy:.4f}')
```

```
Training Accuracy: 0.9807
Test Accuracy: 0.9432
```

```
# Generate k values (logarithmic scale like Figure 2.17)
k_values = np.unique(np.logspace(0, np.log10(300), 20).astype(int))
print(f"Testing k values: {k_values}")

train_accuracies, test_accuracies = [], []

for k in k_values:
    # Train KNN
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)

    # Training accuracy
    y_pred_train = knn.predict(X_train)
    train_acc = accuracy_score(y_train, y_pred_train)
    train_accuracies.append(train_acc)

    # Testing accuracy
    y_pred_test = knn.predict(X_test)
    test_acc = accuracy_score(y_test, y_pred_test)
    test_accuracies.append(test_acc)

    print(f"k={k:3d}: Train Acc={train_acc:.4f}, Test Acc={test_acc:.4f}")

# Plot results
plt.figure(figsize=(7, 5))
plt.semilogx(k_values, train_accuracies, 'b-', linewidth=2, label='Training Accuracy')
plt.semilogx(k_values, test_accuracies, 'r-', linewidth=2, label='Test Accuracy')
plt.xlabel('Number of Neighbors (k)', fontsize=12)
plt.ylabel('Accuracy', fontsize=12)
plt.title('Bias-Variance Trade-off in KNN', fontsize=14)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=12)
plt.xlim(1, 300)
```

```python
plt.ylim(0.7, 1.02)
plt.show()

# Find optimal k
optimal_idx = np.argmax(test_accuracies)
optimal_k = k_values[optimal_idx]
print(f"\nOptimal k: {optimal_k} with test accuracy: {test_accuracies[optimal_idx]:.4f}")

# Check for U-shaped curve
if test_accuracies[0] > test_accuracies[1]:
    print("  U-shaped curve observed: Test error decreases then increases with k")
else:
    print("  No clear U-shaped curve observed")
```

```
Testing k values: [  1   2   3   4   6   8  11  14  20  27  36  49  66  90 121 164 222 300]
k=  1: Train Acc=1.0000, Test Acc=0.9437
k=  2: Train Acc=0.9837, Test Acc=0.9412
k=  3: Train Acc=0.9867, Test Acc=0.9447
k=  4: Train Acc=0.9807, Test Acc=0.9432
k=  6: Train Acc=0.9759, Test Acc=0.9387
k=  8: Train Acc=0.9724, Test Acc=0.9407
k= 11: Train Acc=0.9690, Test Acc=0.9312
k= 14: Train Acc=0.9653, Test Acc=0.9292
k= 20: Train Acc=0.9530, Test Acc=0.9178
k= 27: Train Acc=0.9461, Test Acc=0.9153
k= 36: Train Acc=0.9392, Test Acc=0.9093
k= 49: Train Acc=0.9301, Test Acc=0.8969
k= 66: Train Acc=0.9196, Test Acc=0.8834
k= 90: Train Acc=0.9082, Test Acc=0.8675
k=121: Train Acc=0.8926, Test Acc=0.8560
k=164: Train Acc=0.8759, Test Acc=0.8406
k=222: Train Acc=0.8563, Test Acc=0.8201
k=300: Train Acc=0.8360, Test Acc=0.7997
```



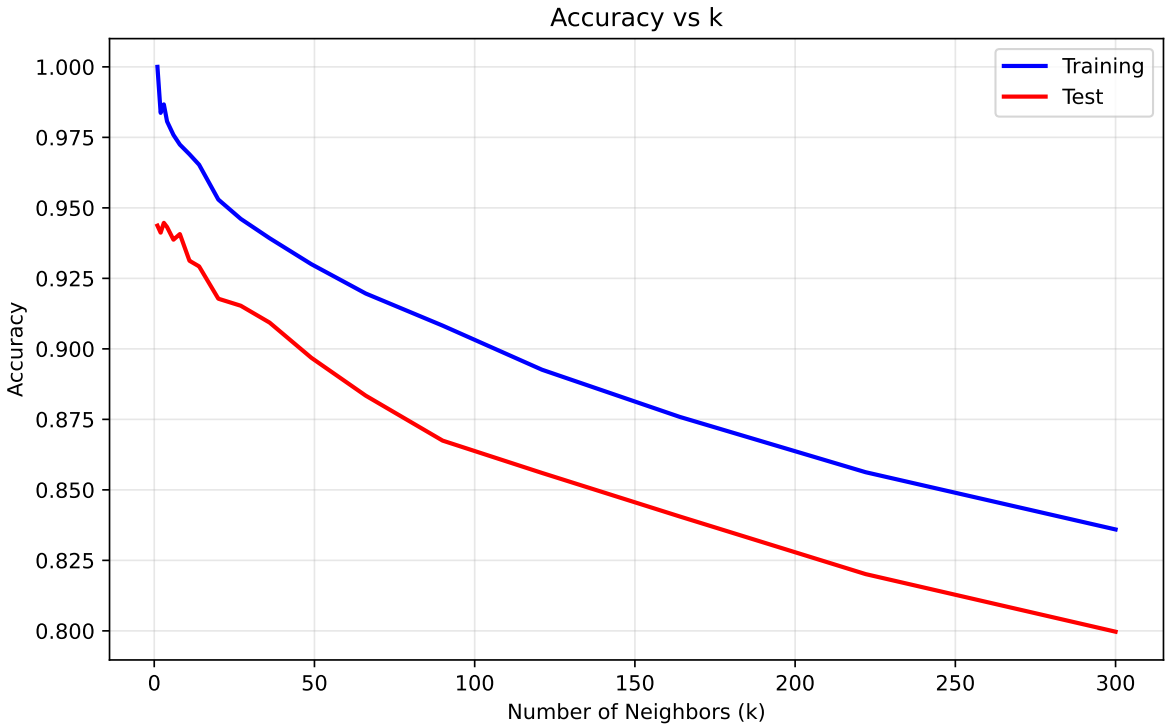

```
Optimal k: 3 with test accuracy: 0.9447
  U-shaped curve observed: Test error decreases then increases with k
```

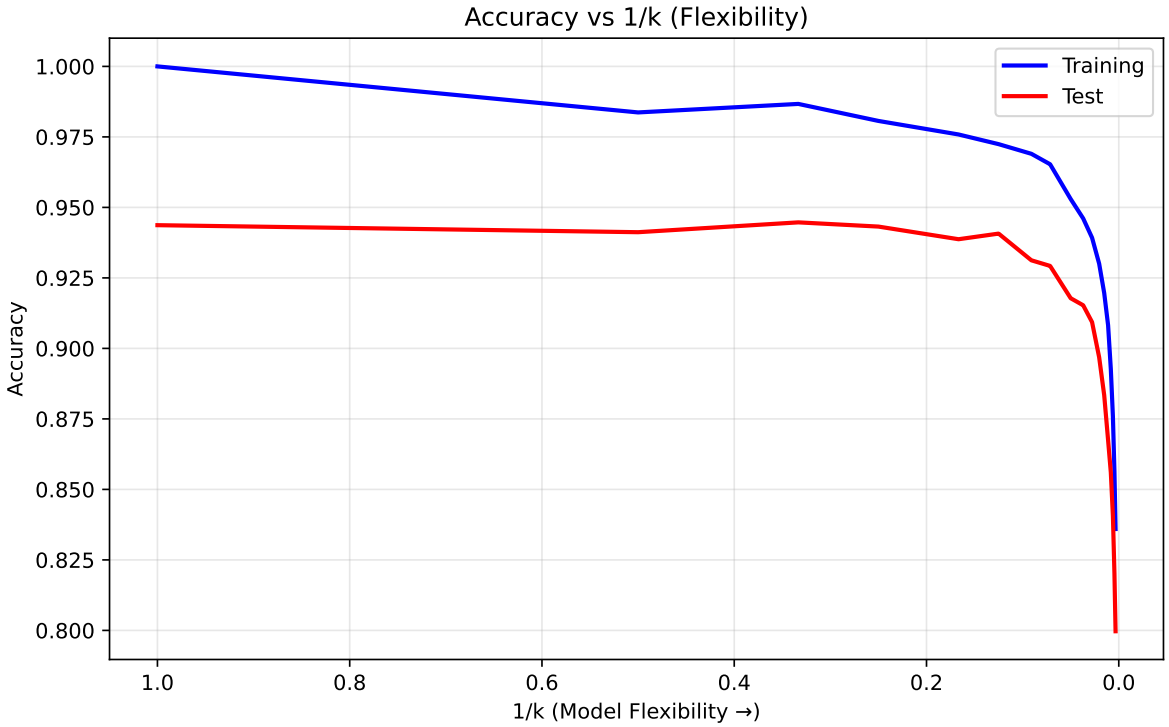
c) Modify the code above to plot the training and test accuracy as a function of 1/k (the inverse of k), similar to Figure 2.17 in the ISL book. Comment on the results, relating them to the bias-variance trade-off.

```
# Generate k values (1 to 300, with more points at small k for detail)
k_values = k_values = np.unique(np.logspace(0, np.log10(300), 20).astype(int))
print(f"Testing k values: {k_values}")

train_acc, test_acc = [], []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k).fit(X_train, y_train)
```

```
    train_acc.append(knn.score(X_train, y_train))
    test_acc.append(knn.score(X_test, y_test))

# Calculate 1/k (inverse relationship)
inverse_k = 1 / k_values

# Plot with 1/k on x-axis (as in ISL book)
plt.figure(figsize=(8, 10))

# Plot 1: Traditional k on x-axis
plt.subplot(2, 1, 1)
plt.plot(k_values, train_acc, 'b-', linewidth=2, label='Training')
plt.plot(k_values, test_acc, 'r-', linewidth=2, label='Test')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Accuracy')
plt.title('Accuracy vs k')
plt.grid(True, alpha=0.3)
plt.legend()
plt.tight_layout(rect=[0, 0, 1, 0.96])

# Plot 2: 1/k on x-axis (like ISL Figure 2.17)
plt.subplot(2, 1, 2)
plt.plot(inverse_k, train_acc, 'b-', linewidth=2, label='Training')
plt.plot(inverse_k, test_acc, 'r-', linewidth=2, label='Test')
plt.xlabel('1/k (Model Flexibility →)')
plt.ylabel('Accuracy')
plt.title('Accuracy vs 1/k (Flexibility)')
plt.gca().invert_xaxis()  # Reverse so 1/k increases left to right
plt.grid(True, alpha=0.3)
plt.legend()

plt.tight_layout()
plt.show()
```

```
Testing k values: [  1   2   3   4   6   8  11  14  20  27  36  49  66  90 121 164 222 300]
```





This code shows you two views of the same phenomenon:

Left plot: How accuracy changes as we add more neighbors (smaller k = more flexible, larger k = less flexible) Right plot: How accuracy changes as model flexibility changes (1/k is a proxy for flexibility - higher 1/k = more flexible model) The right plot mimics the ISL book's Figure 2.17 by using 1/k on the x-axis to represent model flexibility, illustrating the bias-variance trade-off in KNN classification.

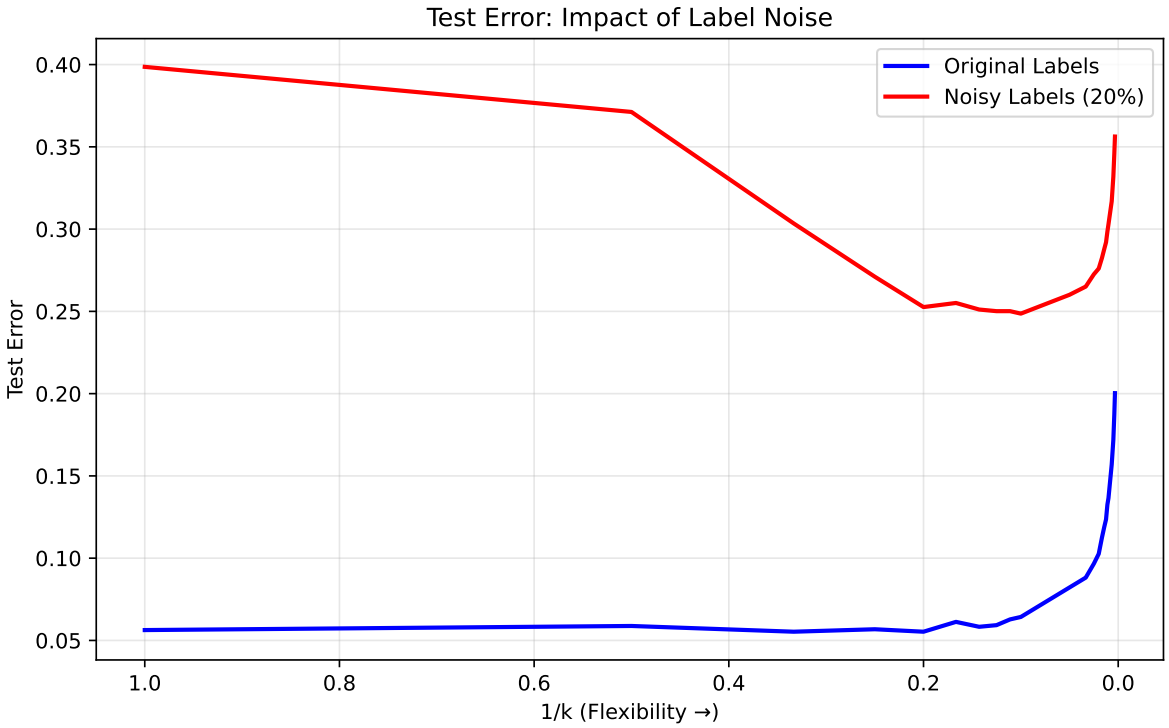
As k decreases (more flexible model), training accuracy increases, but test accuracy initially increases then decreases, showing overfitting. As k increases (less flexible), both accuracies converge, indicating underfitting. The optimal k balances bias and variance for best generalization.

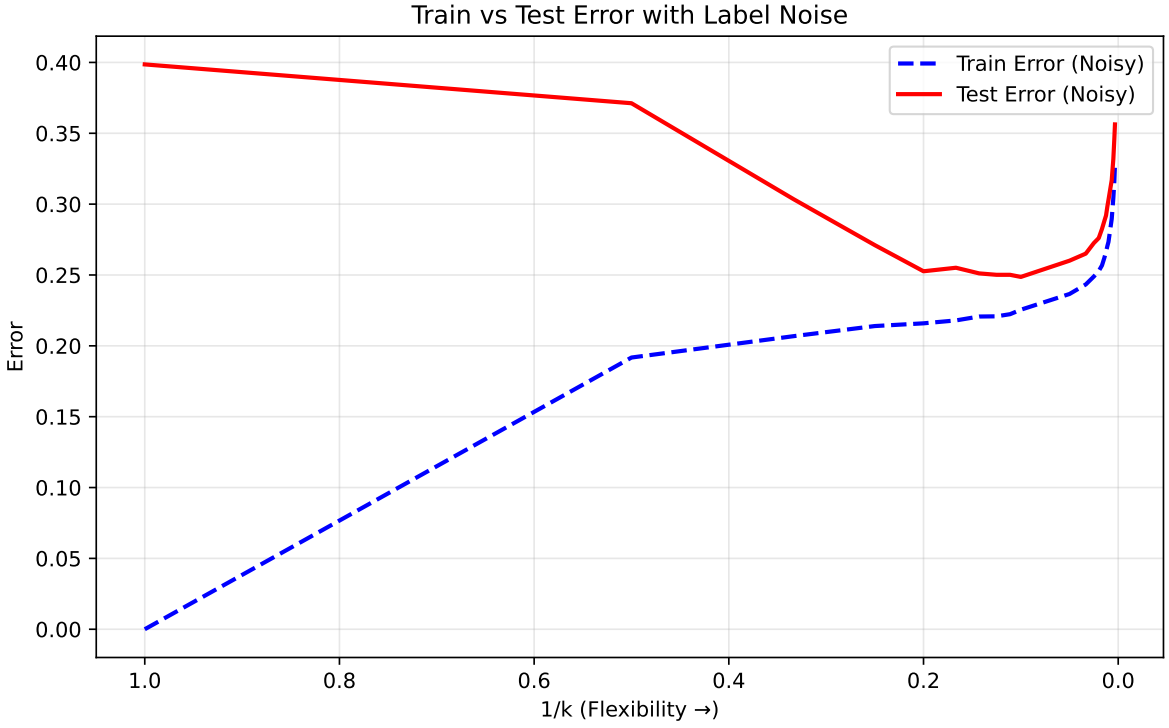
c) Introduce some noise in the training and testing labels for both the training and testing data. You can do this by using np.random.choice to sample from the range of indices of each of the training and test set to determine which labels will be changed, and np.random.choice again to pick the new label. After making this modification, repeat problem (b). How did adding label noise impact the shape of the testing and training error versus 1/k curves?

```
# 1. Add label noise (20% of labels)
def add_noise(y, noise_frac=0.2):
    y_noisy = y.copy()
    n_noise = int(len(y) * noise_frac)
    noise_indices = np.random.choice(len(y), n_noise, replace=False)

    for idx in noise_indices:
        current_label = y[idx]
        # Choose new label from all labels except current one
        possible_labels = [l for l in np.unique(y) if l != current_label]
        y_noisy[idx] = np.random.choice(possible_labels)

    return y_noisy

# 2. Create noisy labels
np.random.seed(42)
y_train_noisy = add_noise(y_train, 0.2)
y_test_noisy = add_noise(y_test, 0.2)

# 3. Test k values
k_values = np.concatenate([np.arange(1, 10), np.arange(10, 101, 10), np.arange(150, 301, 50)]

# 4. Train with original and noisy labels
train_acc_orig, test_acc_orig = [], []
train_acc_noisy, test_acc_noisy = [], []
```

```
for k in k_values:
    # Original labels
    knn_orig = KNeighborsClassifier(n_neighbors=k).fit(X_train, y_train)
    train_acc_orig.append(knn_orig.score(X_train, y_train))
    test_acc_orig.append(knn_orig.score(X_test, y_test))

    # Noisy labels
    knn_noisy = KNeighborsClassifier(n_neighbors=k).fit(X_train, y_train_noisy)
    train_acc_noisy.append(knn_noisy.score(X_train, y_train_noisy))
    test_acc_noisy.append(knn_noisy.score(X_test, y_test_noisy))

# 5. Calculate errors
train_err_orig = [1 - acc for acc in train_acc_orig]
test_err_orig = [1 - acc for acc in test_acc_orig]
train_err_noisy = [1 - acc for acc in train_acc_noisy]
test_err_noisy = [1 - acc for acc in test_acc_noisy]

# 6. Plot comparison
plt.figure(figsize=(8, 10))

# Plot 1: Test error comparison
plt.subplot(2, 1, 1)
inverse_k = 1/k_values
plt.plot(inverse_k, test_err_orig, 'b-', label='Original Labels', linewidth=2)
plt.plot(inverse_k, test_err_noisy, 'r-', label='Noisy Labels (20%)', linewidth=2)
plt.gca().invert_xaxis()
plt.xlabel('1/k (Flexibility →)')
plt.ylabel('Test Error')
plt.title('Test Error: Impact of Label Noise')
plt.legend()
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.grid(True, alpha=0.3)

# Plot 2: Train vs Test error with noise
plt.subplot(2, 1, 2)
plt.plot(inverse_k, train_err_noisy, 'b--', label='Train Error (Noisy)', linewidth=2)
plt.plot(inverse_k, test_err_noisy, 'r-', label='Test Error (Noisy)', linewidth=2)
plt.gca().invert_xaxis()
plt.xlabel('1/k (Flexibility →)')
plt.ylabel('Error')
plt.title('Train vs Test Error with Label Noise')
plt.legend()
```

```
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# 7. Analyze impact
print("IMPACT OF LABEL NOISE:")
print("="*40)

# Find optimal k for each
optimal_k_orig = k_values[np.argmin(test_err_orig)]
optimal_k_noisy = k_values[np.argmin(test_err_noisy)]

print(f"Without noise: Optimal k = {optimal_k_orig}, Min test error = {min(test_err_orig):.4f
print(f"With noise:    Optimal k = {optimal_k_noisy}, Min test error = {min(test_err_noisy):

# Error increase
error_increase = min(test_err_noisy) - min(test_err_orig)
print(f"\nTest error increased by: {error_increase:.4f} (+{error_increase/min(test_err_orig)*

# Check U-shape preservation
print(f"\nU-shape preserved with noise: {'Yes' if test_err_noisy[0] > min(test_err_noisy) and
```


Test Error: Impact of Label Noise
Original Labels
Noisy Labels (20%)
Test Error
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
1.0
0.8
0.6
0.4
0.2
0.0
1/k (Flexibility →)


Train vs Test Error with Label Noise
Train Error (Noisy)
Test Error (Noisy)
Error
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
1.0
0.8
0.6
0.4
0.2
0.0
1/k (Flexibility →)

```
IMPACT OF LABEL NOISE:
========================================
Without noise: Optimal k = 3, Min test error = 0.0553
With noise:    Optimal k = 10, Min test error = 0.2486

Test error increased by: 0.1933 (+349.5%)

U-shape preserved with noise: Yes
```

Label noise acts like adding variance to the model, making it harder to fit the true underlying patterns. As a result, we observe:

1. Increased Test Error: The minimum test error increases significantly with label noise, indicating worse generalization performance.
2. Shifted Optimal k: The optimal k value may shift, often favoring larger k values to smooth out the noise.
3. U-Shape Preservation: The U-shaped curve of test error vs. 1/k is generally preserved, but the curve may be less pronounced, indicating that the model struggles more with noisy labels across all k values.

Overall, label noise degrades model performance and highlights the importance of clean data for effective learning.