STOR 235 — Lab 4

In this lab, you will use linear regression to create predictors and explore the effect of regularization on your predictors. We will continue to handle tabular data using the `pandas` package and data visualization using the `seaborn` package.

This lab is being released a few days before we discuss linear regression and regularization in class. I have tried to provide some basic intuitions here in the examples, but things will be clearer after the relevant lecture. (Sorry for going a bit out of order...)

Terminology note: linear regression (without regularization) is often called "ordinary least squares."

Instructions

There are three parts to this lab:

1. An exploration of basic plotting and linear regression commands;
2. An example application to real data;
3. A part asking you to analyze data using the methods of the previous parts.

The third part will be composed of a number of tasks for you to complete. The tasks are labeled in bold as **Problem 1**, **Problem 2**, and so on. Please make sure that you do not forget to complete any of the problems. Each problem will be worth 4 points.

There is a single code block in under this introductory material that imports various things you will use later. Please run it.

Important Information About Submitting Your Assignment

Please upload your assignment to Gradescope as a PDF file, and make sure to select to all relevant pages for the third part. Check the PDF file before you upload to make sure it contains all of your work, and nothing was cut off at the end.

```
#imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.datasets import fetch_california_housing
```

Part 1: Plotting and Linear Regression

We begin by simulating some data from a linear data generating process (with random noise), fitting a linear regression model, and plotting it.

```
# Simulate data
x = np.random.normal(0, 2, 30) #50 random numbers from a normal distribution
#with mean zero and variance 2
epsilon = np.random.normal(0, 0.5, 30)
#mean zero and variance 1/2
y = x/2 + epsilon
#create y values (using vector operations)
df = pd.DataFrame({'x-data': x, 'y-data': y})
#create data frame, with two column labels
```

```
print(x) #lots of random numbers
```

```
[ 0.57887521 -0.36484596  0.39799565 -1.24538848  0.35344621 -4.26241935
  1.16397541  0.95170772 -0.96147909 -0.66822626 -1.39371747  1.87960967
  1.00668789  0.22170227  2.88051503  2.55947491 -3.12449632 -3.05968539
 -0.3988752   1.04233646 -0.28467904 -1.03868637  1.00263165 -0.19588303
  0.10599425 -1.69727385 -0.16293661  0.52008854  0.43759187  0.28608012]
```

```
print(y) #more numbers
```

```
[ 0.26579343 -0.67783763 -0.0536603  -0.53854509  0.49777227 -2.16204993
  0.04820584  0.41263371 -1.26392933  0.14079833 -1.44185656  1.41427296
 -0.22692394  0.91041117  1.40923826  1.60883919 -1.33659061 -1.67425476
  0.04578035  0.73686079  0.63689781 -1.03963577  0.39871341 -0.05157962
  0.16624039 -1.21358344 -0.35730133  0.49957216 -0.30213971  0.24026994]
```

```
print(df)
```

```
       x-data    y-data
0   0.578875  0.265793
1  -0.364846 -0.677838
2   0.397996 -0.053660
3  -1.245388 -0.538545
4   0.353446  0.497772
5  -4.262419 -2.162050
6   1.163975  0.048206
7   0.951708  0.412634
8  -0.961479 -1.263929
9  -0.668226  0.140798
10 -1.393717 -1.441857
11  1.879610  1.414273
12  1.006688 -0.226924
13  0.221702  0.910411
14  2.880515  1.409238
15  2.559475  1.608839
16 -3.124496 -1.336591
17 -3.059685 -1.674255
18 -0.398875  0.045780
19  1.042336  0.736861
20 -0.284679  0.636898
21 -1.038686 -1.039636
22  1.002632  0.398713
23 -0.195883 -0.051580
24  0.105994  0.166240
25 -1.697274 -1.213583
26 -0.162937 -0.357301
27  0.520089  0.499572
28  0.437592 -0.302140
29  0.286080  0.240270
```

```
X = x.reshape(-1, 1)  # Reshape x for scikit-learn
Y = y.reshape(-1, 1)  # Reshape y for scikit-learn
# reshape makes 2D arrays from lists
# the arguments for reshape are (rows, columns).
#Putting -1 makes Python automatically infer the correct choice.
#this reshape command makes a column vector (1 in column argument)
#this is neeeded since scikit-learn wants column vectors as input
model_ols = LinearRegression()
model_ols.fit(X, Y)
y_pred_ols = model_ols.predict(X) #get the predictions from the model for each X
```

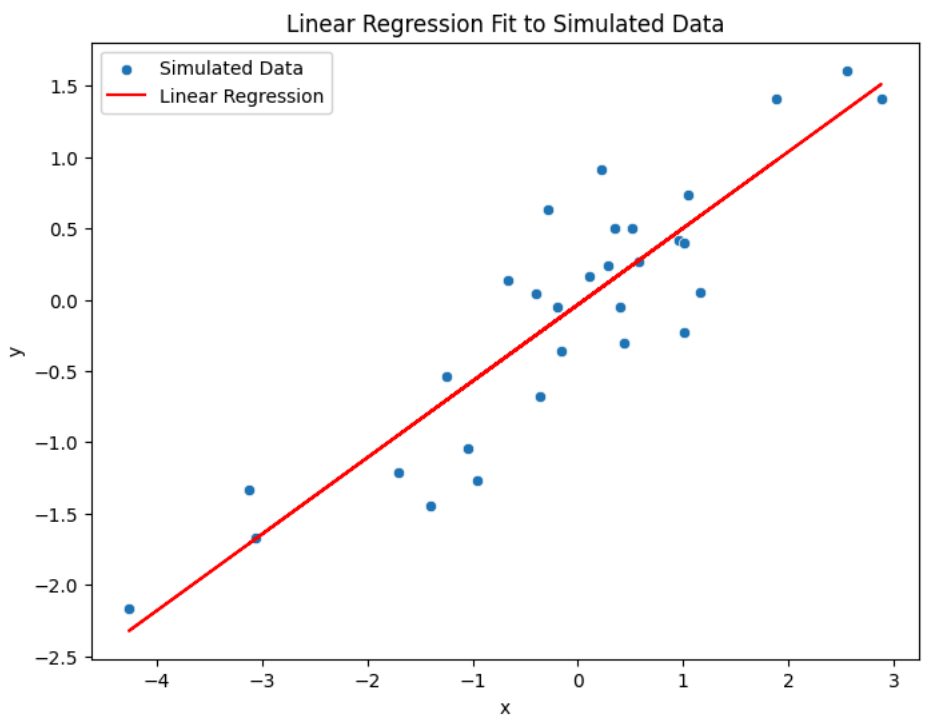
```
print(X) #same numbers as before, but in a column vector.
#You can print Y if you want, to see the same thing.
```

```
[[ 0.57887521]
 [-0.36484596]
 [ 0.39799565]
 [-1.24538848]
 [ 0.35344621]
 [-4.26241935]
 [ 1.16397541]
 [ 0.95170772]
 [-0.96147909]
 [-0.66822626]
 [-1.39371747]
 [ 1.87960967]
 [ 1.00668789]
 [ 0.22170227]
 [ 2.88051503]
 [ 2.55947491]
 [-3.12449632]
 [-3.05968539]
 [-0.3988752 ]
 [ 1.04233646]
 [-0.28467904]
 [-1.03868637]
 [ 1.00263165]
 [-0.19588303]
 [ 0.10599425]
 [-1.69727385]
 [-0.16293661]
 [ 0.52008854]
 [ 0.43759187]
 [ 0.28608012]]
```

```
print(y_pred_ols) #print predictions
```

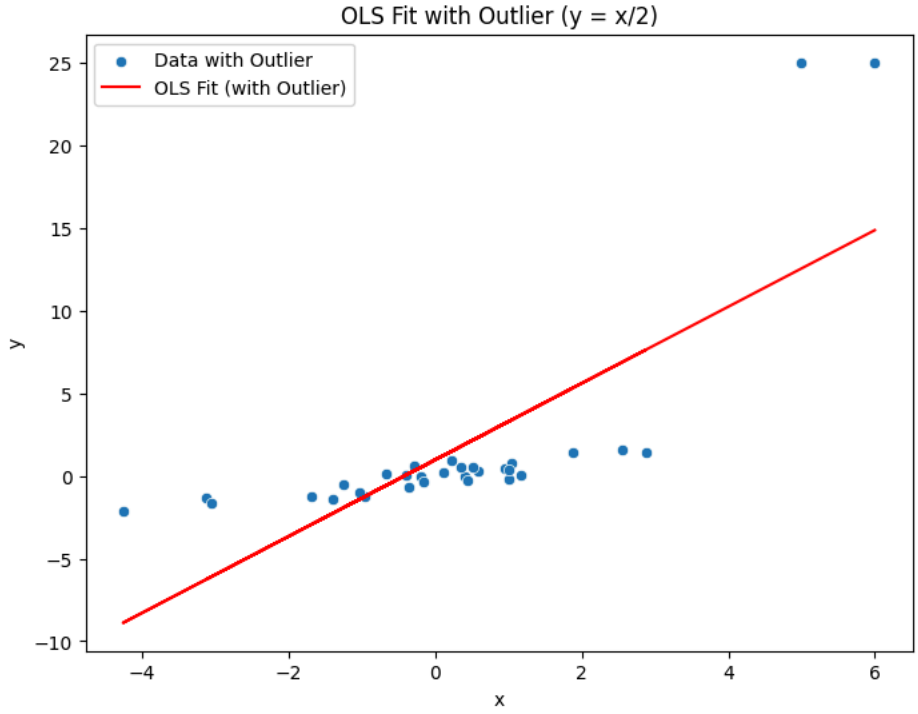
```
[[ 0.27595063]
 [-0.23069637]
 [ 0.17884346]
 [-0.70342523]
 [ 0.15492661]
 [-2.32315116]
 [ 0.59006805]
 [ 0.47610983]
 [-0.55100538]
 [-0.3935694 ]
 [-0.78305727]
 [ 0.97426411]
 [ 0.50562653]
 [ 0.08419844]
 [ 1.51161107]
 [ 1.33925718]
 [-1.71224477]
 [-1.67745032]
 [-0.24896535]
 [ 0.52476486]
 [-0.18765789]
 [-0.59245495]
 [ 0.50344889]
 [-0.13998679]
 [ 0.02207933]
 [-0.94602482]
 [-0.12229914]
 [ 0.24439036]
 [ 0.20010112]
 [ 0.11876039]]
```

```
plt.figure(figsize=(8, 6))
sns.scatterplot(data=df, x='x-data', y='y-data',  label='Simulated Data')
plt.plot(x, y_pred_ols, color='red', label='Linear Regression')
#plot command draws a smooth curve through the predicted points
#so we get a line, since we have a linear predictor
plt.xlabel('x')
plt.ylabel('y')
plt.title('Linear Regression Fit to Simulated Data')
plt.legend()
plt.show()
```



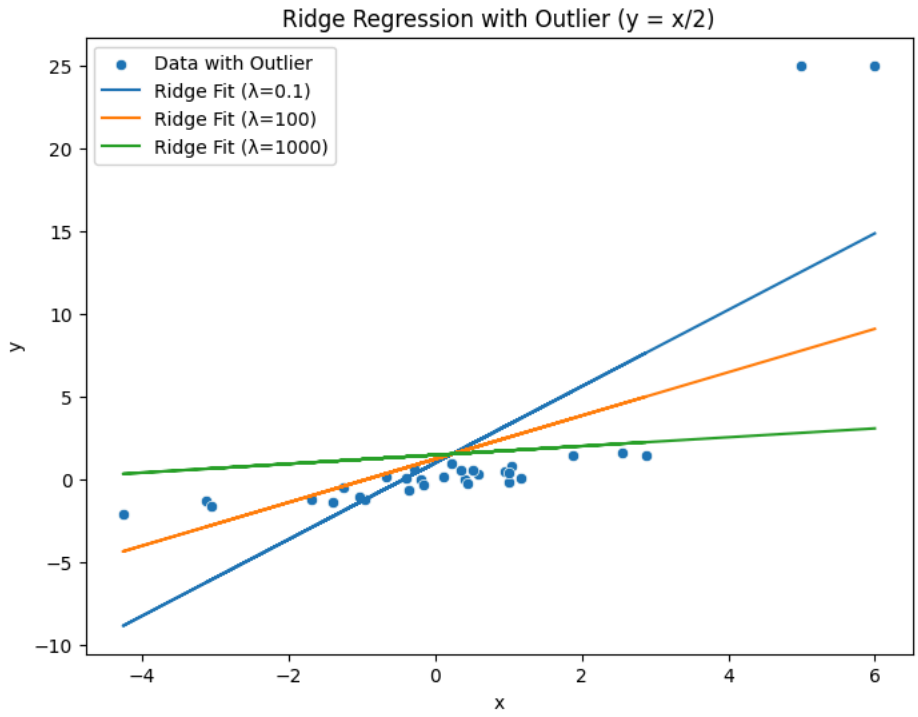
Now we add outlier data and see how it changes the linear regression line. (The exact code being used is not so important. You can ignore some of the more specialized commands.)

```
df_outlier = df.copy() #copy the old data
df_outlier.loc[len(df)] = {'x-data': 6, 'y-data': 25} #change the last data point
df_outlier.loc[len(df)-1] = {'x-data': 5, 'y-data': 25} #change the second-to-last data point
y_outlier = df_outlier['y-data'].values.reshape(-1,1) #extract the x and y values,
#and reshape to column vectors
X_outlier = df_outlier['x-data'].values.reshape(-1, 1)

model_ols_outlier = LinearRegression() #create a linear regression model
model_ols_outlier.fit(X_outlier, y_outlier)
y_pred_ols_outlier = model_ols_outlier.predict(X_outlier)

plt.figure(figsize=(8, 6))
sns.scatterplot(x='x-data', y='y-data', data=df_outlier, label='Data with Outlier')
plt.plot(df_outlier['x-data'], y_pred_ols_outlier, color='red', label='OLS Fit (with Outlier)')
plt.xlabel('x')
plt.ylabel('y')
plt.title('OLS Fit with Outlier (y = x/2)')  # Updated title
plt.legend()
plt.show()
```



By fitting linear regression with regularization, we reduce the impact of outliers in the training data, potentially improving generalization to other data points by reducing overfitting to extreme elements. (There are other reasons regularization is good, but this crude explanation will suffice for now...) To fit a regularized regression, we use the `Ridge` command. The `alpha` parameter controls the strength of the regularization, with a large alpha parameter leading to more regularization (and less fitting to the given data). Using `alpha=0` recovers the previous linear regression model.

```
lambdas = [0.1, 100, 1000]
plt.figure(figsize=(8, 6))
sns.scatterplot(x='x-data', y='y-data', data=df_outlier, label='Data with Outlier')

for lam in lambdas:
    model_ridge = Ridge(alpha=lam)
    model_ridge.fit(X_outlier, y_outlier)
    y_pred_ridge = model_ridge.predict(X_outlier)
    plt.plot(df_outlier['x-data'], y_pred_ridge, label=f'Ridge Fit (λ={lam})')

plt.xlabel('x')
plt.ylabel('y')
plt.title('Ridge Regression with Outlier (y = x/2)')  # Updated title
plt.legend()
plt.show()
```



Part 2: Example Application to Real Data

We now consider an application to predicting California housing prices from census data. We begin by loading the California housing dataset.

```
california = fetch_california_housing()
df = pd.DataFrame(california.data, columns=california.feature_names)
#create dataframe from predictor variables (features)
df['MEDV'] = california.target  # Add the target variable (median house value)
```

Next, we print some basic information and summary statistics. We use the same commands as in the previous lab, when we printed similar information for the wine dataset.

```
print(california.DESCR)
```

```
.. _california_housing_dataset:

    California Housing dataset
    --------------------------

    **Data Set Characteristics:**

    :Number of Instances: 20640

    :Number of Attributes: 8 numeric, predictive attributes and the target

    :Attribute Information:
        - MedInc        median income in block group
        - HouseAge      median house age in block group
        - AveRooms      average number of rooms per household
        - AveBedrms     average number of bedrooms per household
        - Population    block group population
        - AveOccup      average number of household members
        - Latitude      block group latitude
        - Longitude     block group longitude

    :Missing Attribute Values: None

    This dataset was obtained from the StatLib repository.
    https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html

    The target variable is the median house value for California districts,
    expressed in hundreds of thousands of dollars ($100,000).

    This dataset was derived from the 1990 U.S. census, using one row per census
    block group. A block group is the smallest geographical unit for which the U.S.
```

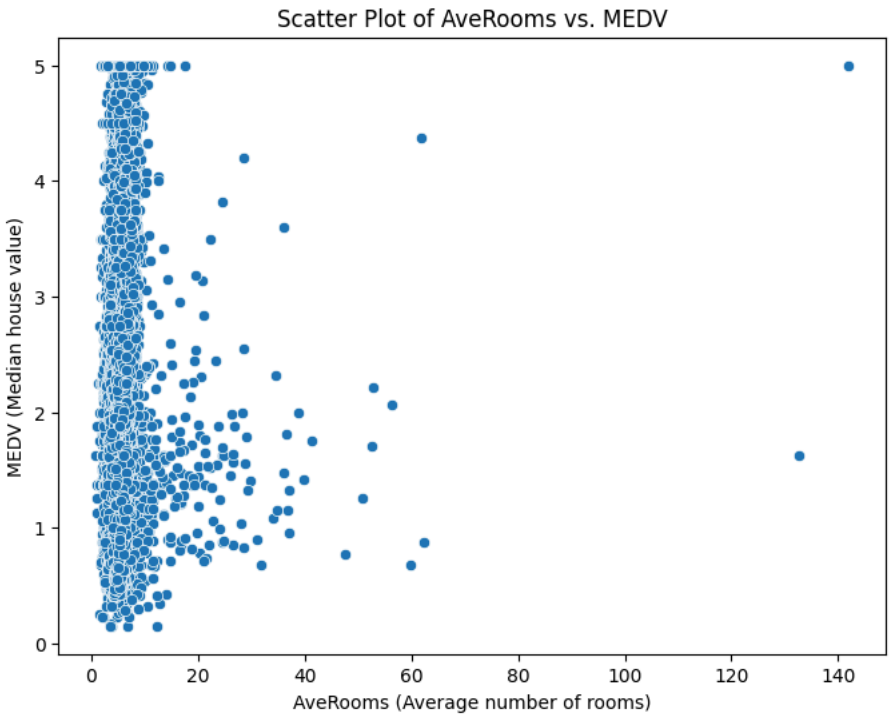
```
Census Bureau publishes sample data (a block group typically has a population
of 600 to 3,000 people).

A household is a group of people residing within a home. Since the average
number of rooms and bedrooms in this dataset are provided per household, these
columns may take surprisingly large values for block groups with few households
and many empty houses, such as vacation resorts.

It can be downloaded/loaded using the
:func:`sklearn.datasets.fetch_california_housing` function.

.. rubric:: References

- Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,
  Statistics and Probability Letters, 33 (1997) 291-297
```

```
print("Dataset shape:", df.shape)
print("\nFirst 5 rows:")
print(df.head())
print("\nSummary statistics:")
print(df.describe())
```

```
Dataset shape: (20640, 9)

First 5 rows:
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85

   Longitude   MEDV
0    -122.23  4.526
1    -122.22  3.585
2    -122.24  3.521
3    -122.25  3.413
4    -122.25  3.422

Summary statistics:
             MedInc      HouseAge      AveRooms     AveBedrms    Population  \
count  20640.000000  20640.000000  20640.000000  20640.000000  20640.000000
mean       3.870671     28.639486      5.429000      1.096675   1425.476744
std        1.899822     12.585558      2.474173      0.473911   1132.462122
min        0.499900      1.000000      0.846154      0.333333      3.000000
25%        2.563400     18.000000      4.440716      1.006079    787.000000
50%        3.534800     29.000000      5.229129      1.048780   1166.000000
75%        4.743250     37.000000      6.052381      1.099526   1725.000000
max       15.000100     52.000000    141.909091     34.066667  35682.000000

           AveOccup      Latitude     Longitude          MEDV
count  20640.000000  20640.000000  20640.000000  20640.000000
mean       3.070655     35.631861   -119.569704      2.068558
std       10.386050      2.135952      2.003532      1.153956
min        0.692308     32.540000   -124.350000      0.149990
25%        2.429741     33.930000   -121.800000      1.196000
50%        2.818116     34.260000   -118.490000      1.797000
75%        3.282261     37.710000   -118.010000      2.647250
max     1243.333333     41.950000   -114.310000      5.000010
```

We can plot the target and predictor in a scatterplot to see how they relate.

```
plt.figure(figsize=(8, 6))
sns.scatterplot(x='AveRooms', y='MEDV', data=df)  # Example: Average number of rooms vs. median house value
plt.title('Scatter Plot of AveRooms vs. MEDV')
plt.xlabel('AveRooms (Average number of rooms)')
plt.ylabel('MEDV (Median house value)')
plt.show()
```



Now we separate into predictors (X) and the target (y).

```
X_cali = df.drop('MEDV', axis=1)
y_cali = df['MEDV']
```

We use a test–train split, as in the previous lab. This particular split uses 20% of the data as the test set.

```
X_train, X_test, y_train, y_test = train_test_split(X_cali, y_cali, test_size=0.2)
```

We now train and evaluate a linear regression model. The evaluation metric we use is called mean squared error (MSE), which is the sum of the squares of the differences between the true target values and the predicted values.

```
model_ols_cali = LinearRegression()
model_ols_cali.fit(X_train, y_train)
y_pred_ols_cali = model_ols_cali.predict(X_test)

mse_ols = mean_squared_error(y_test, y_pred_ols_cali)

print("\nOLS Performance:")
print(f"Mean Squared Error: {mse_ols:.4f}")
```

```
OLS Performance:
Mean Squared Error: 0.4934
```

Recalling that the units of the target are in multiples of $100,000, and MSE has units that are squares of these units, the average of the squared errors is about about $.55*100,000^2, so we can get a rough idea of the typical error by taking the square root to get $75,000. This is not horrible, but also not amazing. (Note that the summary statistics above indicate that the house values are all capped at $500,000.) A more sophisticated model with nonlinear features and interactions would likely do better. (Your numerical answers will vary slightly due to the randomness in the train–test split, but these should be approximately correct.)

Nonetheless, we can extract some interesting qualitative information from the coefficients. The code below prints the intercept and the coefficients from each of the features from the linear model that we fit. For example, there is a positive coefficient on the fourth feature, bedrooms, confirming that the number of bedrooms positively predicts housing price. There is a negative coefficient on the seventh feature, latitude, indicating that lower latitudes predict larger house values. (In other words, people seem to like SoCal...)

These coefficients can be interpreted roughly as follows: a one-unit change in the corresponding independent variable produces, on average, a change in the target variable equal to the size of the coefficient. Since the independent variables on all on different scales, we cannot compare the sizes of the coefficients directly.

```
print(model_ols_cali.intercept_)
```

```
-37.387786280406075
```

```
print(model_ols_cali.coef_)
```

```
[ 4.27651001e-01  9.24636335e-03 -9.63243681e-02  5.91883951e-01
 -5.90388584e-06 -3.39518009e-03 -4.27722242e-01 -4.40487735e-01]
```

Part 3

In this part, you will fit a linear model to a dataset of cars. Your goal is to predict miles per gallon from various provided features, like horsepower and weight. The steps of this part are essentially identical to those in the previous part, but you will have to write the code yourself.

We begin by importing the data. I've also included some cleaning code (which you can ignore) to remove missing data, to make the rest of the lab go a bit smoother. The target variable (MPG) is already in the data frame. You do not need to add it.

```
# 1. Load the MPG dataset (from Seaborn)
mpg_data = sns.load_dataset('mpg')

# 2. Data Cleaning and Preprocessing
mpg_data = mpg_data.dropna()
mpg_data['horsepower'] = pd.to_numeric(mpg_data['horsepower'], errors='coerce')
mpg_data = mpg_data.dropna()
```

Problem 1. Print the shape, first five rows, and summary statistics for the MPG dataset. For summary statistics, use `describe`.

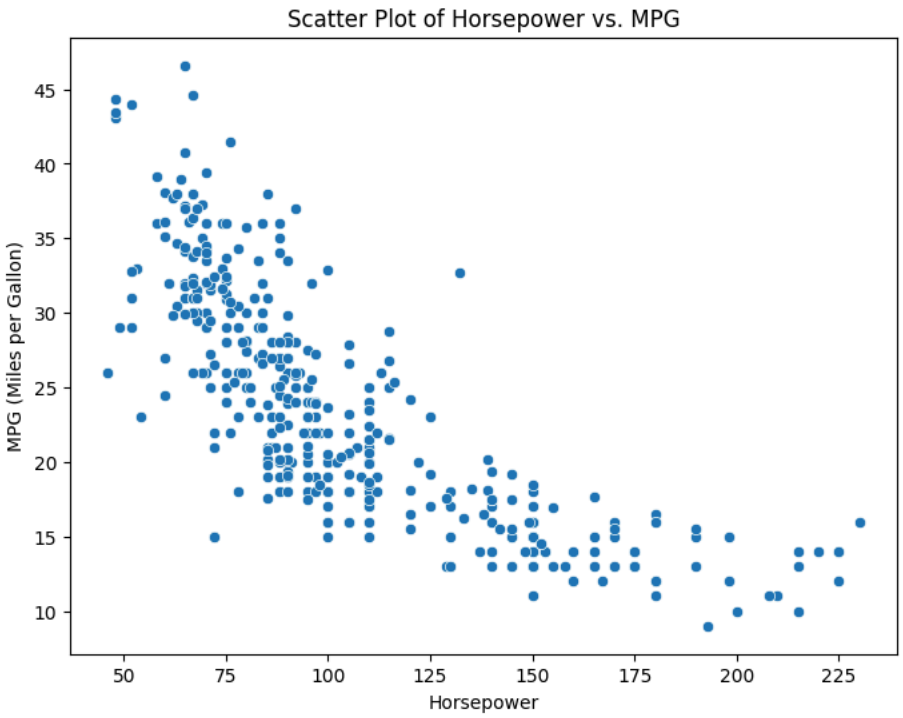
```
print("Dataset shape:", mpg_data.shape)


print("\nFirst 5 rows:")
print(mpg_data.head())

print("\nSummary statistics:")
print(mpg_data.describe())
```

```
Dataset shape: (392, 9)

First 5 rows:
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0    3504          12.0
1  15.0          8         350.0       165.0    3693          11.5
2  18.0          8         318.0       150.0    3436          11.0
3  16.0          8         304.0       150.0    3433          12.0
4  17.0          8         302.0       140.0    3449          10.5

   model_year origin                       name
0          70    usa  chevrolet chevelle malibu
1          70    usa          buick skylark 320
2          70    usa         plymouth satellite
3          70    usa              amc rebel sst
4          70    usa                ford torino

Summary statistics:
              mpg   cylinders  displacement  horsepower       weight  \
count  392.000000  392.000000    392.000000  392.000000   392.000000
mean    23.445918    5.471939    194.411990  104.469388  2977.584184
std      7.805007    1.705783    104.644004   38.491160   849.402560
min      9.000000    3.000000     68.000000   46.000000  1613.000000
25%     17.000000    4.000000    105.000000   75.000000  2225.250000
50%     22.750000    4.000000    151.000000   93.500000  2803.500000
75%     29.000000    8.000000    275.750000  126.000000  3614.750000
max     46.600000    8.000000    455.000000  230.000000  5140.000000

       acceleration  model_year
count    392.000000  392.000000
mean      15.541327   75.979592
std        2.758864    3.683737
min        8.000000   70.000000
25%       13.775000   73.000000
```

```
50%       15.500000   76.000000
75%       17.025000   79.000000
max       24.800000   82.000000
```

Problem 2. Create a 2d scatterplot with horsepower as the independent variable and MPG as the dependent variable.

```
plt.figure(figsize=(8, 6))
sns.scatterplot(x='horsepower', y='mpg', data=mpg_data)
plt.title('Scatter Plot of Horsepower vs. MPG')
plt.xlabel('Horsepower')
plt.ylabel('MPG (Miles per Gallon)')
plt.show()
```



Problem 3. Prepare the data for modeling by separating the features (predictors) from the target variable using the `drop` command, as in the previous part. You will also need to take out the non-numerical features (`origin`, `name`). You can do this with the `drop` command, too, by using it multiple times in succession.

Then run create variables `X_mpg_train, X_mpg_test, y_mpg_train, y_mpg_test` using `test_train_split` with a test set that is 25% of the total data.

```
X_mpg = mpg_data.drop(['mpg', 'origin', 'name'], axis=1)
y_mpg = mpg_data['mpg']
```

```
X_mpg_train, X_mpg_test, y_mpg_train, y_mpg_test = train_test_split(X_mpg, y_mpg, test_size=0.25)
```

Problem 4. Fit a linear regression model to your training data and measure the MSE on the test data.

Remember that MSE in this case has units MPG^2, so you need to take a square root to get an estimate of the typical error, but you don't need to provide this as an answer here. Nevertheless, I enourage you to compute it, because it should indicate that you have a reasonably good predictor of MPG from the other variables.

```
model_ols_mpg = LinearRegression()
model_ols_mpg.fit(X_mpg_train, y_mpg_train)
y_pred_mpg = model_ols_mpg.predict(X_mpg_test)
mse_mpg = mean_squared_error(y_mpg_test, y_pred_mpg)
print(f"Mean Squared Error (MSE): {mse_mpg:.4f}")
```

```
Mean Squared Error (MSE): 11.4742
```

Problem 5. Print the coefficients from your model and use the output to answer the following questions.

Does horsepower positively or negatively predict MPG? Does weight negatively or positively predict MPG? Do the these findings match your intuition? Write your answers after the prompt below.

Answers:

```
print("Coefficients:", model_ols_mpg.coef_)
```

```
Coefficients: [-1.26585083e-01 -1.06362582e-03 -5.15694984e-04 -6.12321704e-03
  5.34224401e-02  7.87742046e-01]
```

Both horsepower and weight negatively predict the MPG. This is because the coefficient of horsepower and weight is a negative number. These findings do match my intuition. I would assume a more powerful car would consume more fuel so more horsepower leads to a decrease in MPG. I also expect heavier covers to have a lower MPG so as weight goes up, the MPG goes down. Both ideas of logic are supported by the coefficients as there is a negative relationship with horsepower/weight and MPG.

Start coding or generate with AI.