Homework 2

Zankhana Mehta:002320268

mehta.zan@northeastern.edu

QUESTION 1

In [3]:
```
import pandas as pd
from ISLP import load_data
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.metrics import accuracy_score,log_loss
```

In [4]:
```
df=load_data("Auto")
```

In [7]:
```
df.head()
```

Out[7]:

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

In [9]:
```
df.shape
```

Out[9]:
```
(392, 8)
```

In [11]:
```
df=df[df['horsepower'].notna()]
df.shape
```

Out[11]: (392, 8)

Part a

In [14]:
```
mpg_median=df['mpg'].median()
print(mpg_median)
```

```
22.75
```

In [16]:
```
df['mpg01'] = df['mpg'].apply(lambda x: 1 if x > mpg_median else 0)
df.head()
```

Out[16]:

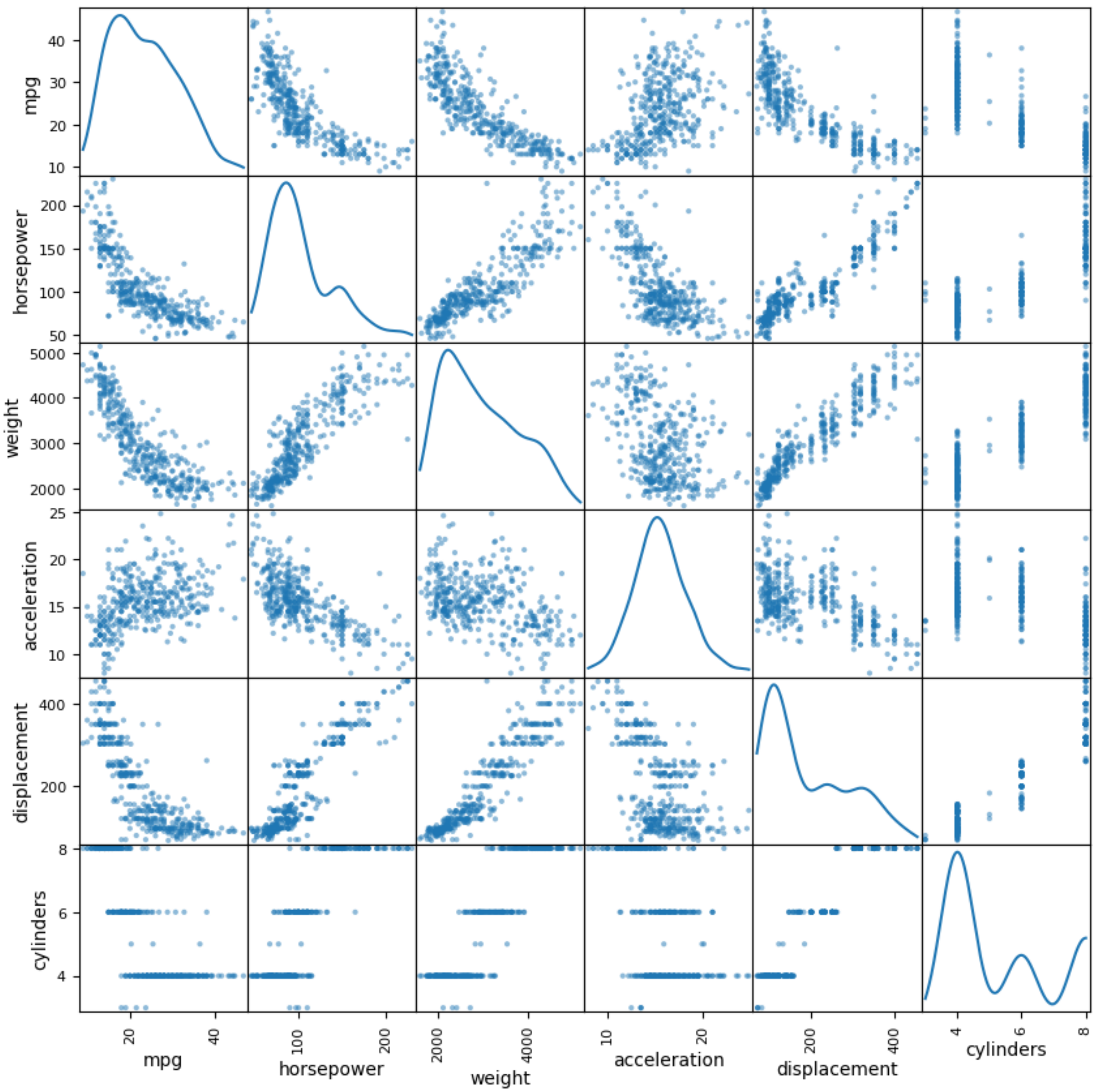
	mpg	cylinders	displacement	horsepower	weight	acceleration	year	orig
name								
chevrolet chevelle malibu	18.0	8	307.0	130	3504	12.0	70	
buick skylark 320	15.0	8	350.0	165	3693	11.5	70	
plymouth satellite	18.0	8	318.0	150	3436	11.0	70	
amc rebel sst	16.0	8	304.0	150	3433	12.0	70	
ford torino	17.0	8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

In [18]:
```
mpg01_counts = df['mpg01'].value_counts()
print(mpg01_counts)
```

```
mpg01
0    196
1    196
Name: count, dtype: int64
```

Part b

In [21]:
```
features = ['mpg', 'horsepower', 'weight', 'acceleration', 'displacement','c
pd.plotting.scatter_matrix(df[features], figsize=(10, 10), diagonal='kde')
plt.show()
```

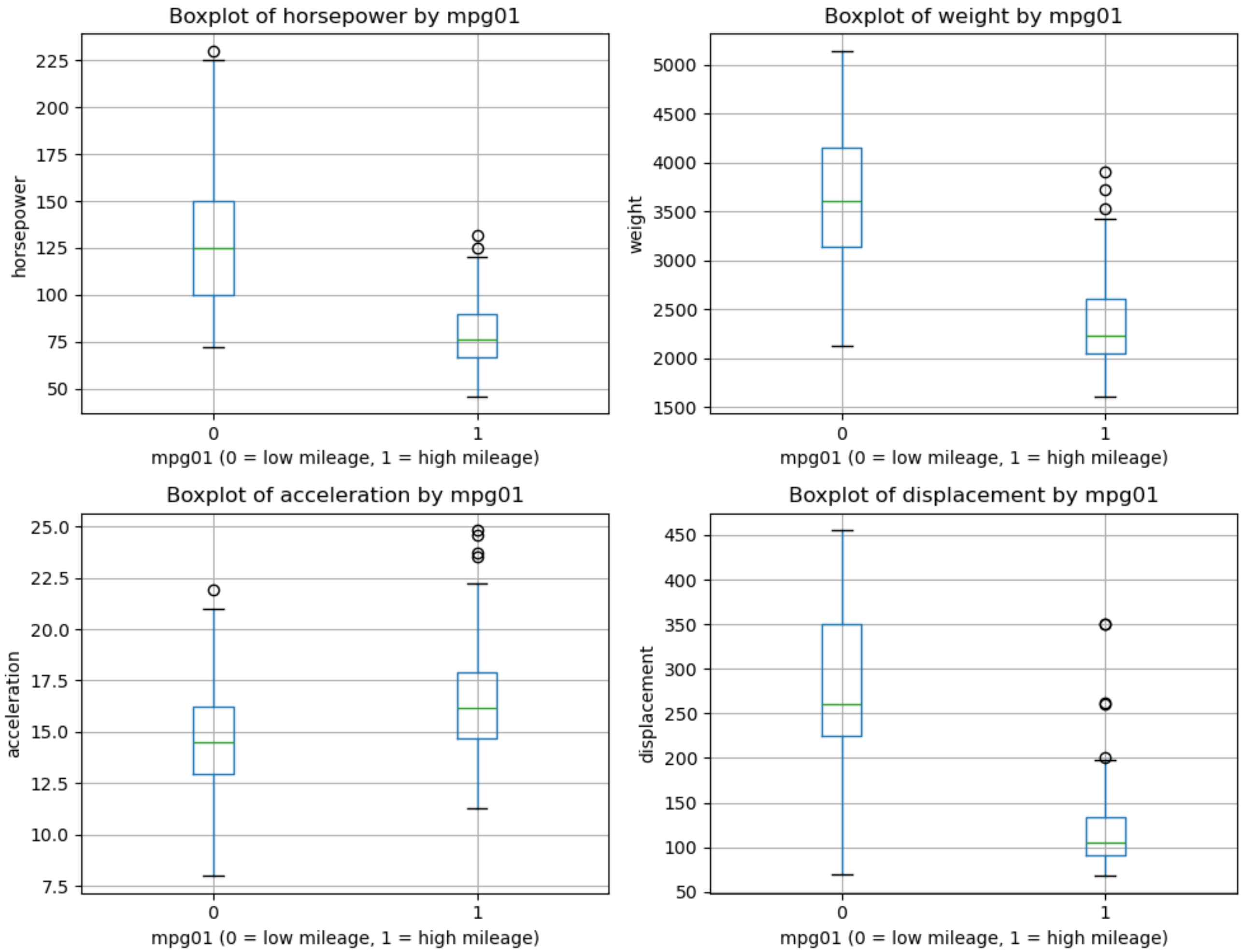


1. Horsepower: There's a clear negative correlation between mpg and horsepower. As horsepower increases, mpg (miles per gallon) decreases. This suggests that cars with higher horsepower tend to have lower gas mileage.
2. Weight: Similar to horsepower, there's a strong negative correlation between mpg and weight. Heavier cars tend to have lower gas mileage.
3. Acceleration: The relationship between mpg and acceleration is not as clear. There appears to be some mild correlation where cars with higher acceleration might have slightly higher mpg, but it's not as strong.
4. Displacement: There's also a negative correlation between mpg and displacement. Cars with larger engine displacements tend to have lower gas mileage.

= Based on the scatter plots:

Horsepower and weight seem to be the most useful features for predicting whether a car has high or low gas mileage (mpg01). Both features show a strong negative correlation with mpg, meaning that as these values increase, the gas mileage tends to decrease. Displacement is also a good predictor, as it shows a similar trend to weight and horsepower, though perhaps not as strong. Acceleration shows weaker correlation, making it less likely to be a strong predictor of mpg01.

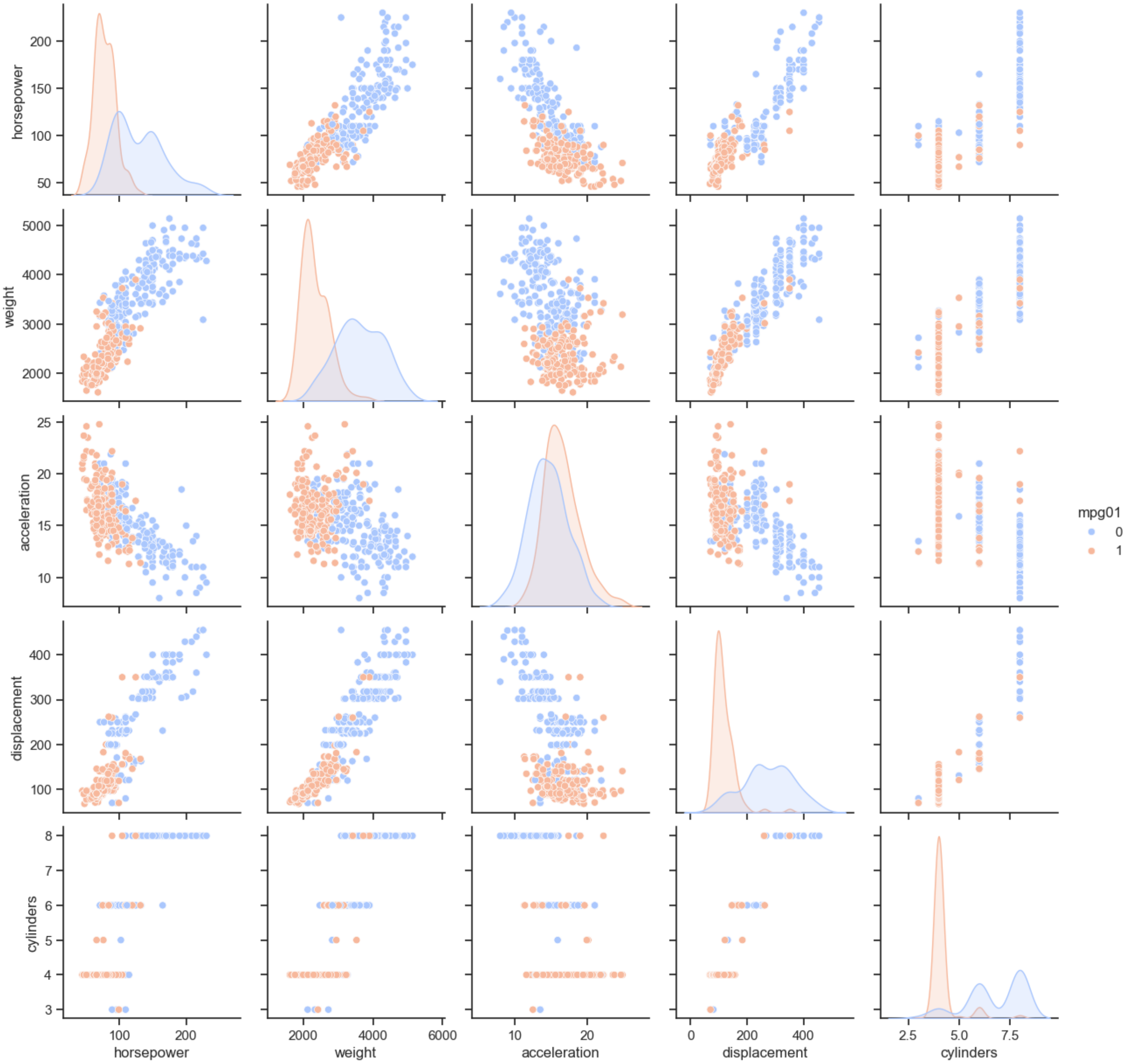
In [23]:
```
features = ['horsepower', 'weight', 'acceleration', 'displacement']
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

axes = axes.ravel()

for i, feature in enumerate(features):
    df.boxplot(column=feature, by='mpg01', ax=axes[i])
    axes[i].set_title(f'Boxplot of {feature} by mpg01')
    axes[i].set_xlabel('mpg01 (0 = low mileage, 1 = high mileage)')
    axes[i].set_ylabel(feature)

plt.tight_layout()
plt.suptitle('')
plt.show()
```



Horsepower, weight and displacement are the most significant features for predicting whether a car will have high or low gas mileage (mpg01). All three show strong separations between low and high mileage cars, with low mileage cars being heavier, more powerful, and having larger engines.

In [27]:

```
sns.set(style="ticks", color_codes=True)
features = ['horsepower', 'weight', 'acceleration', 'displacement','cylinder
sns.pairplot(df, vars=features, hue='mpg01', palette='coolwarm', diag_kind='
plt.show()
```



In [28]:
```
# we will also find the correlation between feature varibales to understand
correlation_matrix = df.corr()
print(correlation_matrix)
```

```
                   mpg  cylinders  displacement  horsepower    weight  \
mpg           1.000000  -0.777618     -0.805127   -0.778427 -0.832244
cylinders    -0.777618   1.000000      0.950823    0.842983  0.897527
displacement -0.805127   0.950823      1.000000    0.897257  0.932994
horsepower   -0.778427   0.842983      0.897257    1.000000  0.864538
weight       -0.832244   0.897527      0.932994    0.864538  1.000000
acceleration  0.423329  -0.504683     -0.543800   -0.689196 -0.416839
year          0.580541  -0.345647     -0.369855   -0.416361 -0.309120
origin        0.565209  -0.568932     -0.614535   -0.455171 -0.585005
mpg01         0.836939  -0.759194     -0.753477   -0.667053 -0.757757

              acceleration      year    origin     mpg01
mpg               0.423329  0.580541  0.565209  0.836939
cylinders        -0.504683 -0.345647 -0.568932 -0.759194
displacement     -0.543800 -0.369855 -0.614535 -0.753477
horsepower       -0.689196 -0.416361 -0.455171 -0.667053
weight           -0.416839 -0.309120 -0.585005 -0.757757
acceleration      1.000000  0.290316  0.212746  0.346822
year              0.290316  1.000000  0.181528  0.429904
origin            0.212746  0.181528  1.000000  0.513698
mpg01             0.346822  0.429904  0.513698  1.000000
```

In the above correlation matrix we see that for the mpg01 column the it is highly negatively correlated to cylinders, displacement, horsepower and weight. Whereas, acceleration, year and origin have lower correlation.

The features **horsepower, weight, cylinders and displacement** are clearly strong predictors of mpg01. Cars with lower values in these features (blue points) tend to have higher mileage, while cars with higher values (orange points) tend to have lower mileage.

Part c

In [33]:
```
X = df[['horsepower', 'weight', 'acceleration', 'displacement','cylinders']]
y = df['mpg01']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar

print("Training set size:", X_train.shape[0])
print("Test set size:", X_test.shape[0])
```

```
Training set size: 313
Test set size: 79
```

Part d

In [36]:
```
model = LogisticRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

```
y_prob=model.predict_proba(X_test)

accuracy = accuracy_score(y_test, y_pred)
test_err=1-accuracy
ll=log_loss(y_test,y_prob)
print("Accuracy score:",accuracy )
print("Test error:",test_err)
print("Test Loss:",ll)
```

```
Accuracy score: 0.8987341772151899
Test error: 0.10126582278481011
Test Loss: 0.2855911457875017
```

Part e

In [39]:
```
lda_model = LinearDiscriminantAnalysis()
lda_model.fit(X_train, y_train)
y_pred_lda = lda_model.predict(X_test)
y_prob_lda=lda_model.predict_proba(X_test)

acc =accuracy_score(y_test, y_pred_lda)
test_er=1-acc
ll2=log_loss(y_test,y_prob_lda)
print("Accuracy:", acc)
print("Test error:",test_er)
print("Test loss:", ll2)
```

```
Accuracy: 0.8987341772151899
Test error: 0.10126582278481011
Test loss: 0.37429222612767143
```

QUESTION 2

Bootstrapping is a resampling technique in machine learning that involves repeatedly drawing samples from a data set with replacement. So in a bootstrap sample, each observation is chosen randomly with replacement from the original sample of n observations.

Part a

Since the bootstrap process randomly samples with replacement, the probability of selecting any one observation (including the first one) in a single draw is 1/n.

Thus, the probability that the first bootstrap observation is not the first observation from the original sample is: **1-(1/n)**

The process is the same for every observation in the bootstrap sample. So, the probability that the last observation is not the first observation from the original sample is the same as the first observation: **1-(1/n)**

Part b

We want to find the probability that the first observation from the original sample does not appear at all in the bootstrap sample. Each time we sample, the probability of not selecting the first observation is 1-(1/n) We do this sampling n times for the bootstrap sample, so the probability that the first observation is never selected is **(1-1/n)^n**

When n=10 (1-1/10)^10 = (9/10)^10 = 0.348678 So, the probability that the first observation does not appear in the bootstrap sample is approximately **34.87%**

When n=100 (1-1/100)^100 = (99/100)^10 = 0.366 So, the probability that the first observation does not appear in the bootstrap sample when n=100 is approximately **36.60%**

QUESTION 3

In [43]:
```
from ISLP import load_data
import numpy as np
import statsmodels.api as sm
from sklearn.utils import resample
from sklearn.neighbors import KNeighborsRegressor
```

In [45]:
```
og_df=load_data('College')
og_df.head()
```

Out[45]:

	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad	P.Undergrad	C
0	Yes	1660	1232	721	23	52	2885	537	
1	Yes	2186	1924	512	16	29	2683	1227	
2	Yes	1428	1097	336	22	50	1036	99	
3	Yes	417	349	137	60	89	510	63	
4	Yes	193	146	55	16	44	249	869	

In [47]:
```
og_df.shape
```

Out[47]: (777, 18)

In [49]:
```
og_df['Accept_rate']=og_df['Accept']/og_df['Apps']
og_df.head()
```

Out[49]:

	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad	P.Undergrad	C
0	Yes	1660	1232	721	23	52	2885	537	
1	Yes	2186	1924	512	16	29	2683	1227	
2	Yes	1428	1097	336	22	50	1036	99	
3	Yes	417	349	137	60	89	510	63	
4	Yes	193	146	55	16	44	249	869	

In [51]:
```
test_df=og_df.drop(columns=['Accept', 'Apps'])
test_df.head()
```

Out[51]:

	Private	Enroll	Top10perc	Top25perc	F.Undergrad	P.Undergrad	Outstate	Room.
0	Yes	721	23	52	2885	537	7440	
1	Yes	512	16	29	2683	1227	12280	
2	Yes	336	22	50	1036	99	11250	
3	Yes	137	60	89	510	63	12960	
4	Yes	55	16	44	249	869	7560	

In [53]:
```
#mapping categorical variable
test_df['Private'] = test_df['Private'].map({'Yes': 1, 'No': 0})
test_df.head()
```

Out[53]:

	Private	Enroll	Top10perc	Top25perc	F.Undergrad	P.Undergrad	Outstate	Room.
0	1	721	23	52	2885	537	7440	
1	1	512	16	29	2683	1227	12280	
2	1	336	22	50	1036	99	11250	
3	1	137	60	89	510	63	12960	
4	1	55	16	44	249	869	7560	

Part a

In [56]:
```
mu_hat = test_df['Accept_rate'].mean()
print(f"Population mean: ",mu_hat)
```

```
Population mean:  0.7469277072775414
```

Part b

In [59]:
```
std_dev = test_df['Accept_rate'].std()
n = len(test_df['Accept_rate'])
std_error = std_dev / np.sqrt(n)
print(f"Standard error: ",std_error)
```

```
Standard error:  0.005277323728707518
```

Part c

In [62]:
```
n = 1000
bootstrap_means = []
for i in range(n):
    bootstrap_sample = test_df['Accept_rate'].sample(frac=1, replace=True)
    bootstrap_means.append(bootstrap_sample.mean())
bootstrap_std_error = np.std(bootstrap_means)
print(f"Bootstrap standard error: ",bootstrap_std_error)
```

```
Bootstrap standard error:  0.0052420348658755835
```

The bootstrap estimated error is less than the standard error this is subject to the resampling of the dataset.

Part d

In [66]:
```
lower_bound = mu_hat - 2 * bootstrap_std_error
upper_bound = mu_hat + 2 * bootstrap_std_error
print(f"CI: [",lower_bound, upper_bound,"]")
```

```
CI: [ 0.7364436375457902 0.7574117770092926 ]
```

Part e

In [69]:
```
X = test_df['Top10perc']
y = test_df['Accept_rate']
X_with_const = sm.add_constant(X) #intercept

model = sm.OLS(y, X_with_const).fit()
initial_params = model.params
print(f"Initial Parameters: Intercept (B0) = {initial_params.iloc[0]}, Coeff

n = 1000
bootstrap_betas = np.zeros((n, 2))  # Store [B0, B1]

for i in range(n):
    bootstrap_sample = resample(og_df)
    X_boot = bootstrap_sample['Top10perc']
    y_boot = bootstrap_sample['Accept_rate']
```

```
    X_boot_with_const = sm.add_constant(X_boot)

    bootstrap_model = sm.OLS(y_boot, X_boot_with_const).fit()
    bootstrap_betas[i] = bootstrap_model.params

bootstrap_std_err = np.std(bootstrap_betas, axis=0)
statsmodels_std_err = model.bse

print(f"Bootstrap Standard Errors: Intercept (B0) = ", bootstrap_std_err[0],
print(f"Statsmodels Standard Errors: Intercept (B0) = ",statsmodels_std_err.
```

```
Initial Parameters: Intercept (B0) = 0.8569331798627102, Coefficient (B1) =
-0.003991699070596205
Bootstrap Standard Errors: Intercept (B0) =  0.009534588380208407  Coefficie
nt (B1) =  0.00033143848075090185
Statsmodels Standard Errors: Intercept (B0) =  0.00860399673826018 Coefficie
nt (B1) =  0.00026300038151207936
```

Part f

In [72]:

```
knn = KNeighborsRegressor(n_neighbors=10)

X = test_df[['Top10perc']].values
y = test_df['Accept_rate'].values

knn.fit(X, y)

top10perc_value = np.array([[76]])  # Value for prediction
predicted_accept_rate = knn.predict(top10perc_value)

print(f"Predicted Accept rate for Top10perc=76 (original model):", predicted

n = 1000
bootstrap_predictions = []

for i in range(n):
    bootstrap_sample = resample(test_df)

    X_bootstrap = bootstrap_sample[['Top10perc']].values
    y_bootstrap = bootstrap_sample['Accept_rate'].values

    knn.fit(X_bootstrap, y_bootstrap)

    bootstrap_pred = knn.predict(top10perc_value)

    bootstrap_predictions.append(bootstrap_pred[0])

bootstrap_predictions = np.array(bootstrap_predictions)
bootstrap_standard_error = np.std(bootstrap_predictions)
```

```
print(f"Bootstrap estimated standard error of the predicted Accept rate:", b
```

```
Predicted Accept rate for Top10perc=76 (original model): 0.41277623410287206
Bootstrap estimated standard error of the predicted Accept rate: 0.032532585
28267341
```

In [74]:

```
def bootstrap_knn(X, y, top10perc_value, K, n=1000):
    knn = KNeighborsRegressor(n_neighbors=K)
    bootstrap_predictions = []

    for i in range(n):
        # Resample the data with replacement
        bootstrap_sample = resample(test_df)
        X_bootstrap = bootstrap_sample[['Top10perc']].values
        y_bootstrap = bootstrap_sample['Accept_rate'].values

        knn.fit(X_bootstrap, y_bootstrap)

        bootstrap_pred = knn.predict(top10perc_value)
        bootstrap_predictions.append(bootstrap_pred[0])

    bootstrap_standard_error = np.std(bootstrap_predictions)
    return bootstrap_standard_error

X = test_df[['Top10perc']].values  # Features
y = test_df['Accept_rate'].values  # Target
top10perc_value = np.array([[76]])  # Value for prediction

se_knn_5 = bootstrap_knn(X, y, top10perc_value, K=5)
print(f"Bootstrap estimated standard error of the predicted Accept_rate (K=5

se_knn_50 = bootstrap_knn(X, y, top10perc_value, K=50)
print(f"Bootstrap estimated standard error of the predicted Accept_rate (K=5

X_with_const = sm.add_constant(X)  # Add intercept (constant term) to the fe
linear_model = sm.OLS(y, X_with_const).fit()

top10perc_value_with_const = sm.add_constant(top10perc_value, has_constant='

predicted_accept_rate_linear = linear_model.predict(top10perc_value_with_con

linear_se = linear_model.bse[1]  # Standard error for the Top10perc coeffici
print(f"Standard error for the linear regression model:",linear_se)
```

```
Bootstrap estimated standard error of the predicted Accept_rate (K=5):  0.04
090085017450626
Bootstrap estimated standard error of the predicted Accept_rate (K=50):  0.0
27858210045884412
Standard error for the linear regression model: 0.00026300038151207936
```

Part g

KNN Regression with K=5: A smaller K (closer neighbors) tends to lead to predictions that are more sensitive to local variations in the data, which may result in higher variability and a higher standard error.

KNN Regression with K=50: A larger K (more neighbors) results in smoother predictions that are less sensitive to small variations in the data, and reduces the standard error due to averaging over a larger number of points. The decrease in error is due to decrease in the variance as the k increases.

QUESTION 4

Part a

In [79]:
```
from sklearn.linear_model import LinearRegression
import statsmodels.api as sm
```

In [81]:
```
test_df.head()
```

Out[81]:

	Private	Enroll	Top10perc	Top25perc	F.Undergrad	P.Undergrad	Outstate	Room.
0	1	721	23	52	2885	537	7440	
1	1	512	16	29	2683	1227	12280	
2	1	336	22	50	1036	99	11250	
3	1	137	60	89	510	63	12960	
4	1	55	16	44	249	869	7560	

In [83]:
```
test_df.shape
```

Out[83]: (777, 17)

In [85]:
```
# List of independent variables (predictors)
predictors = ['Private','Enroll', 'Top10perc', 'Top25perc', 'F.Undergrad',
              'P.Undergrad', 'Outstate', 'Room.Board', 'Books', 'Personal',
              'PhD', 'Terminal', 'S.F.Ratio', 'perc.alumni', 'Expend', 'Grac

# Target variable (dependent variable)
y = og_df['Accept_rate']

best_r2 = -1
best_predictor = None
```

```
best_model = None

for predictor in predictors:
    X = test_df[[predictor]]
    X_with_const = sm.add_constant(X)

    model = sm.OLS(y, X_with_const).fit()
    r2 = model.rsquared

    if r2 > best_r2:
        best_r2 = r2
        best_predictor = predictor
        best_model = model

print(f"The best one-predictor model is with ",best_predictor," with R-squar
print(best_model.summary())
```

```
The best one-predictor model is with  Top10perc  with R-squared:  0.22913012
842545943
                            OLS Regression Results
==============================================================================
==
Dep. Variable:            Accept_rate   R-squared:                       0.2
29
Model:                            OLS   Adj. R-squared:                  0.2
28
Method:                 Least Squares   F-statistic:                     23
0.4
Date:                Sun, 13 Oct 2024   Prob (F-statistic):           9.56e-
46
Time:                        22:28:35   Log-Likelihood:                 488.
30
No. Observations:                 777   AIC:                            -97
2.6
Df Residuals:                     775   BIC:                            -96
3.3
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
==
                 coef    std err          t      P>|t|      [0.025      0.97
5]
------------------------------------------------------------------------------
--
const          0.8569      0.009     99.597      0.000       0.840       0.8
74
Top10perc     -0.0040      0.000    -15.178      0.000      -0.005      -0.0
03
==============================================================================
==
Omnibus:                       40.930   Durbin-Watson:                   1.8
68
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               46.0
29
Skew:                          -0.576   Prob(JB):                     1.01e-
10
Kurtosis:                       3.307   Cond. No.                         6
0.8
==============================================================================
==

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is corre
ctly specified.
```

Part b

In [88]:
```
import itertools
```

In [90]:
```
best_r2_2 = -1
best_predictors_2 = None
best_model_2 = None

for predictor_pair in itertools.combinations(predictors, 2):
    X = test_df[list(predictor_pair)]
    X_with_const = sm.add_constant(X)

    model = sm.OLS(y, X_with_const).fit()
    r2 = model.rsquared

    #check for best r2
    if r2 > best_r2_2:
        best_r2_2 = r2
        best_predictors_2 = predictor_pair
        best_model_2 = model

print(f"The best two-predictor model is with predictors '{best_predictors_2[
print(best_model.summary())
```

```
The best two-predictor model is with predictors 'Private' and  Top10perc  wi
th R-squared:  0.25662146039174905
                            OLS Regression Results
==============================================================================
==
Dep. Variable:            Accept_rate   R-squared:                       0.2
29
Model:                            OLS   Adj. R-squared:                  0.2
28
Method:                 Least Squares   F-statistic:                     23
0.4
Date:                Sun, 13 Oct 2024   Prob (F-statistic):           9.56e-
46
Time:                        22:28:38   Log-Likelihood:                 488.
30
No. Observations:                 777   AIC:                            -97
2.6
Df Residuals:                     775   BIC:                            -96
3.3
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
==
                 coef    std err          t      P>|t|      [0.025      0.97
5]
------------------------------------------------------------------------------
--
const          0.8569      0.009     99.597      0.000       0.840       0.8
74
Top10perc     -0.0040      0.000    -15.178      0.000      -0.005      -0.0
03
==============================================================================
==
Omnibus:                       40.930   Durbin-Watson:                   1.8
68
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               46.0
29
Skew:                          -0.576   Prob(JB):                     1.01e-
10
Kurtosis:                       3.307   Cond. No.                         6
0.8
==============================================================================
==

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is corre
ctly specified.
```

Part c

In [93]:
```
#similar to part a
best_r2_one = -1
best_one_predictor = None
best_one_model = None

for predictor in predictors:
    X = test_df[[predictor]]
    X_with_const = sm.add_constant(X)

    model = sm.OLS(y, X_with_const).fit()
    r2 = model.rsquared

    if r2 > best_r2_one:
        best_r2_one = r2
        best_one_predictor = predictor
        best_one_model = model

print(f"The best one-predictor model is with ",best_one_predictor," with R-s

#finding the second best
remaining_predictors = [p for p in predictors if p != best_one_predictor]
best_r2_two = -1
best_two_predictors = None
best_two_model = None

for predictor in remaining_predictors:
    X_two = test_df[[best_one_predictor, predictor]]
    X_two_with_const = sm.add_constant(X_two)

    model = sm.OLS(y, X_two_with_const).fit()
    r2 = model.rsquared

    if r2 > best_r2_two:
        best_r2_two = r2
        best_two_predictors = [best_one_predictor, predictor]
        best_two_model = model

print(f"The best two-predictor model is with predictors",best_two_predictors
print(best_two_model.summary())
```

```
The best one-predictor model is with  Top10perc  with R-squared:  0.22913012
842545943
The best two-predictor model is with predictors Top10perc  and  Private  wit
h R-squared:  0.2566214603917484
                            OLS Regression Results
==============================================================================
==
Dep. Variable:            Accept_rate   R-squared:                       0.2
57
Model:                            OLS   Adj. R-squared:                  0.2
```

```
55
Method:                 Least Squares   F-statistic:                    13
3.6
Date:                Sun, 13 Oct 2024   Prob (F-statistic):          1.44e-
50
Time:                        22:28:39   Log-Likelihood:                502.
41
No. Observations:                 777   AIC:                           -99
8.8
Df Residuals:                     774   BIC:                           -98
4.8
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
==
                 coef    std err          t      P>|t|      [0.025      0.97
5]
------------------------------------------------------------------------------
--
const          0.8229      0.011     77.807      0.000       0.802       0.8
44
Top10perc     -0.0042      0.000    -16.114      0.000      -0.005      -0.0
04
Private        0.0555      0.010      5.350      0.000       0.035       0.0
76
==============================================================================
==
Omnibus:                       34.044   Durbin-Watson:                   1.9
18
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               37.6
78
Skew:                          -0.503   Prob(JB):                     6.58e-
09
Kurtosis:                       3.389   Cond. No.                          9
5.2
==============================================================================
==

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is corre
ctly specified.
```

Part d

a. For the one predictor model there is one model for each predictor so total **16 models**.

b. For the two-predictor model all possible combinations of 2 predictors are considered. The number of ways to select 2 predictors from 16 is C(n,k), where n is the total number of predictors and k is the number of predictors. Hence C(16,2)= **120 models**

c. In the c part we implement forward stepwise selection. We find the predictor 1 which gives us 16 models then we use that predictor and get the second best predictor where another 15 times modelling is done for the remaining predictors. So in total we implement 16+15 models = **31 models**

QUESTION 5

In [101…

```
from sklearn.model_selection import train_test_split,cross_val_score
from sklearn.linear_model import Ridge, Lasso
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
```

Part a

In [104…

```
X=test_df.drop('Accept_rate',axis=1)
y=test_df['Accept_rate']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar
X_train.head()
#y_train.head()
```

Out[104…

	Private	Enroll	Top10perc	Top25perc	F.Undergrad	P.Undergrad	Outstate	Roo
739	0	478	12	25	2138	227	4470	
133	1	249	23	57	1698	894	9990	
234	1	198	7	20	545	42	11750	
55	1	156	25	55	421	27	6500	
639	0	588	56	86	1846	154	9843	

Part b

In [107…

```
lamda = np.arange(0.001, 100, 10)
best_lambda = None
lowest_mse = float('inf')

# Loop over each lambda for cross validation
for lam in lamda:
    ridge = Ridge(alpha=lam)

    mse = -cross_val_score(ridge, X_train, y_train, cv=10, scoring='neg_mear

    if mse < lowest_mse:
        lowest_mse = mse
```

```
        best_lambda = lam

print(f"Best lambda (alpha):",best_lambda)
print(f"Lowest cross-validated MSE:", lowest_mse)

final_ridge = Ridge(alpha=best_lambda)
final_ridge.fit(X_train, y_train)

y_pred = final_ridge.predict(X_test)

test_mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on test set: {test_mse}")

non_zero_coeff = np.sum(final_ridge.coef_ != 0)
print(f"Number of non-zero coefficients",non_zero_coeff)
```

```
Best lambda (alpha): 0.001
Lowest cross-validated MSE: 0.01353330874547807
Mean Squared Error on test set: 0.019014581265742247
Number of non-zero coefficients 16
```

Part c

In [110…

```
lamda = np.arange(0.001, 100, 10)

best_lambda = None
lowest_mse = float('inf')

for lam in lamda:
    lasso = Lasso(alpha=lam, max_iter=10000)

    mse = -cross_val_score(lasso, X_train, y_train, cv=10, scoring='neg_mean

    if mse < lowest_mse:
        lowest_mse = mse
        best_lambda = lam

print(f"Best lambda:",best_lambda)
print(f"Lowest cross-validated MSE:",lowest_mse)

final_lasso = Lasso(alpha=best_lambda, max_iter=10000)
final_lasso.fit(X_train, y_train)

y_pred = final_lasso.predict(X_test)

test_mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on test set: {test_mse}")

non_zero_coefficients = np.sum(final_lasso.coef_ != 0)
print(f"Number of non-zero coefficients",non_zero_coefficients)
```

```
Best lambda: 0.001
Lowest cross-validated MSE: 0.01353790620465565
Mean Squared Error on test set: 0.018875975839989893
Number of non-zero coefficients 16
```

Part_d

Both Ridge and Lasso selected the same lambda value (0.001).Since the data was not standardized, this could significantly affect both Ridge and Lasso regression results. Ridge penalizes large coefficients but does not inherently differentiate between variables with different scales like Top10perc vs Outstate. Lasso applies absolute shrinkage, but it is more sensitive to feature scaling. As a result, predictors with larger scales may dominate the model, preventing some coefficients from being driven to zero.

The test set Mean Squared Error (MSE) for both models is very close, with Lasso slightly outperforming Ridge. The difference between Ridge (0.0190) and Lasso (0.0189) is very small, indicating both models performed similarly on the test set.

Both models retained all 16 coefficients as non-zero. Ridge typically shrinks coefficients towards zero but rarely eliminates any of them, which aligns with the result here. Lasso, however, is expected to perform feature selection by setting some coefficients exactly to zero. The fact that it did not eliminate any predictors suggests that variables with larger scales might have dominated, making it harder for Lasso to identify less relevant features.

Part a(Standardized)

In [114…

```
X=test_df.drop('Accept_rate',axis=1)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(test_df[predictors])
df_X_scaled = pd.DataFrame(X_scaled, columns = predictors)
y=test_df['Accept_rate']
X_train, X_test, y_train, y_test = train_test_split(df_X_scaled, y, test_siz
df_X_scaled.head()
```

Out[114…

	Private	Enroll	Top10perc	Top25perc	F.Undergrad	P.Undergrad	Outstate
0	0.612553	-0.063509	-0.258583	-0.191827	-0.168116	-0.209207	-0.746356
1	0.612553	-0.288584	-0.655656	-1.353911	-0.209788	0.244307	0.457496
2	0.612553	-0.478121	-0.315307	-0.292878	-0.549565	-0.497090	0.201305
3	0.612553	-0.692427	1.840231	1.677612	-0.658079	-0.520752	0.626633
4	0.612553	-0.780735	-0.655656	-0.596031	-0.711924	0.009005	-0.716508

Part b(Standardized)

In [117…

```
lamda = np.arange(0.001, 100, 10)
best_lambda = None
lowest_mse = float('inf')

# Loop over each lambda for cross validation
for lam in lamda:
    ridge = Ridge(alpha=lam)

    mse = -cross_val_score(ridge, X_train, y_train, cv=10, scoring='neg_mear

    if mse < lowest_mse:
        lowest_mse = mse
        best_lambda = lam

print(f"Best lambda (alpha):",best_lambda)
print(f"Lowest cross-validated MSE:", lowest_mse)

final_ridge = Ridge(alpha=best_lambda)
final_ridge.fit(X_train, y_train)

y_pred = final_ridge.predict(X_test)

test_mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on test set:", test_mse)

non_zero_coeff = np.sum(final_ridge.coef_ != 0)
print(f"Number of non-zero coefficients",non_zero_coeff)
```

```
Best lambda (alpha): 30.001
Lowest cross-validated MSE: 0.013470438843064642
Mean Squared Error on test set: 0.01902206576269187
Number of non-zero coefficients 16
```

Part c(Standardized)

In [120…

```
lamda = np.arange(0.001, 100, 10)

best_lambda = None
lowest_mse = float('inf')

for lam in lamda:
    lasso = Lasso(alpha=lam, max_iter=10000)

    mse = -cross_val_score(lasso, X_train, y_train, cv=10, scoring='neg_mear

    if mse < lowest_mse:
        lowest_mse = mse
        best_lambda = lam

print(f"Best lambda:",best_lambda)
print(f"Lowest cross-validated MSE:",lowest_mse)

final_lasso = Lasso(alpha=best_lambda, max_iter=10000)
final_lasso.fit(X_train, y_train)

y_pred = final_lasso.predict(X_test)

test_mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on test set:",test_mse)

non_zero_coefficients = np.sum(final_lasso.coef_ != 0)
print(f"Number of non-zero coefficients",non_zero_coefficients)
```

```
Best lambda: 0.001
Lowest cross-validated MSE: 0.013508366611471104
Mean Squared Error on test set: 0.01906846446038031
Number of non-zero coefficients 15
```

Part d(Standardized)

The lambda for ridge regression is 30.001, higher than the previous optimal value of 0.001 (before standardization). After standardization, all features are on the same scale, so the model can apply a larger penalty across all coefficients without favoring any particular feature. For lasso lambda is same as before. This indicates that the regularization strength needed didn't change much, even after standardization.

Test MSE for ridge is 0.01902. This is very close to the earlier test MSE (~0.01901), meaning the model's predictive ability has remained consistent even after scaling. In lasso The test error remains close to that from Ridge and Lasso before standardization, showing that scaling the data did not drastically affect model performance.

For ridge, the same number of non-zero coefficients suggests that all features in the model are still useful predictors, even after standardization. Wheras in lasso 1 feature was dropped compared to the earlier output (where all 16 coefficients were non zero). Standardization helped Lasso perform better feature selection, by identifying and shrinking an unnecessary feature to zero.

In [ ]:

```

```