Untitled

April 17, 2024

[106]:
```
# 1

# (a) If the Bayes decision boundary is linear, we would expect LDA to perform␣
↪better on the training set due to its simplicity and avoidance of␣
↪overfitting. However, on the test set, where generalization is crucial, QDA␣
↪might perform better if the true boundary is not strictly linear, as it has␣
↪more flexibility to capture complex relationships.

# (b) If the Bayes decision boundary is non-linear, QDA is likely to outperform␣
↪LDA on both the training and test sets because QDA can model non-linear␣
↪boundaries more accurately.

# (c) As the sample size  n increases, we anticipate the test prediction␣
↪accuracy of QDA relative to LDA to improve. With a larger sample size, the␣
↪variance of QDA is better managed, allowing it to exploit its flexibility␣
↪for better fitting the data while avoiding overfitting.

# (d) False. Even if the Bayes decision boundary for a given problem is linear,␣
↪using QDA rather than LDA might not lead to a superior test error rate.␣
↪QDA's increased flexibility could lead to overfitting, especially with a␣
↪smaller sample size, resulting in a higher test error rate compared to LDA.
```

[107]:
```
import math

# 2

def logistic_regression(hours_studied, GPA):
    # Estimated coefficients
    beta0 = -6
    beta1 = 0.05
    beta2 = 1

    linear_combination = beta0 + beta1 * hours_studied + beta2 * GPA

    probability = 1 / (1 + math.exp(-linear_combination))

    return probability
```

```
# (a) Predict the probability that a student who studies for 40 hours and has␣
 ↪an undergrad GPA of 3.5 gets an A
hours_studied_a = 40
GPA_a = 3.5
probability_a = logistic_regression(hours_studied_a, GPA_a)
print("Probability of getting an A (a):", probability_a)

# (b) How many hours would the student in the previous question need to study␣
 ↪to have a 50% chance of getting an A?
def find_hours_for_50_percent_chance(GPA):
    hours_studied_b = (0 - (-6) - 1 * GPA) / 0.05
    return hours_studied_b

hours_studied_b = find_hours_for_50_percent_chance(GPA_a)
print("Hours needed for 50% chance of getting an A (b):", hours_studied_b)
```

```
Probability of getting an A (a): 0.3775406687981454
Hours needed for 50% chance of getting an A (b): 50.0
```

[108]:
```
# 3

mean_dividend = 10
mean_no_dividend = 0
variance = 36
probability_dividend = 0.80
probability_no_dividend = 1 - probability_dividend
X = 4

def pdf(mean, variance, x):
    return (1 / (math.sqrt(2 * math.pi * variance))) * math.exp(-(x - mean)**2 /
 ↪ (2 * variance))

# Calculate P(X = 4 | D)
px_4_given_dividend = pdf(mean_dividend, variance, X)

# Calculate P(X = 4 | ~D)
px_4_given_no_dividend = pdf(mean_no_dividend, variance, X)

# Calculate P(D | X = 4) using Bayes' theorem
probability_dividend_given_x = (px_4_given_dividend * probability_dividend) /␣
 ↪(px_4_given_dividend * probability_dividend + px_4_given_no_dividend *␣
 ↪probability_no_dividend)

print("Probability of issuing a dividend given X = 4:",␣
 ↪probability_dividend_given_x)
```

```
Probability of issuing a dividend given X = 4: 0.7518524532975261
```

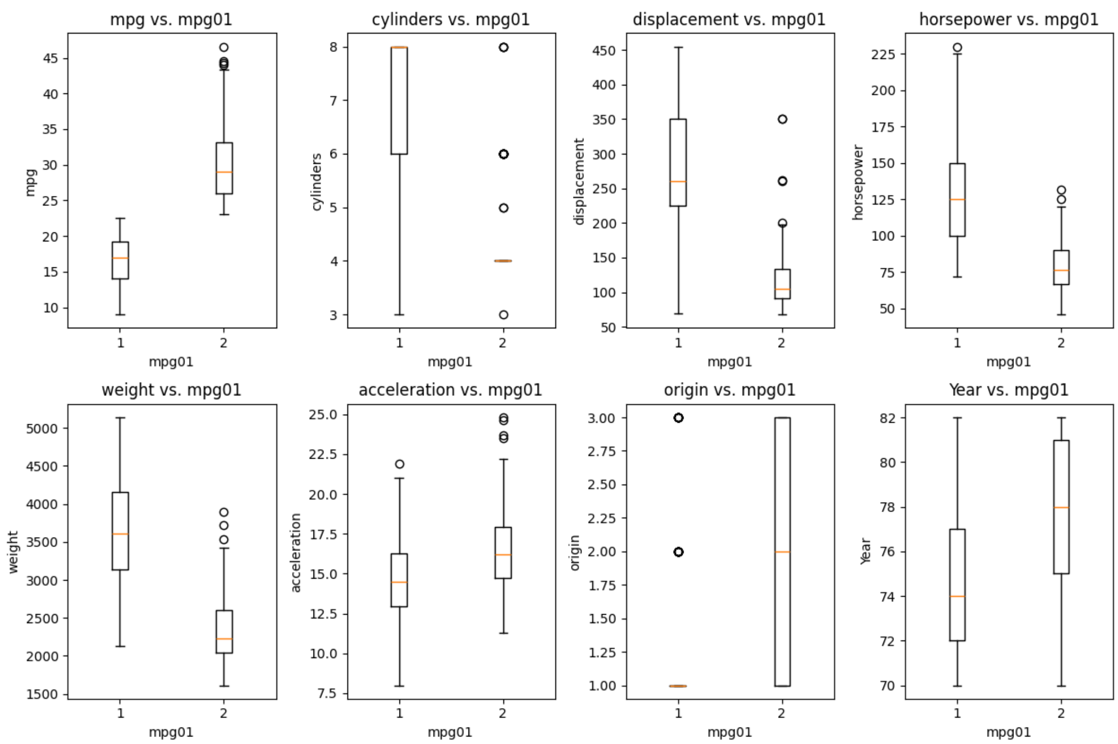
[109]:
```python
# 4

# (a)

import pandas as pd

df_auto = pd.read_csv('/Users/rouren/Desktop/24S ML/HW/hw3/Data-Auto.csv')

mpg_median = df_auto['mpg'].median()

df_auto['mpg01'] = (df_auto['mpg'] > mpg_median).astype(int)

print(df_auto.head())
```

```
   Unnamed: 0   mpg  cylinders  displacement  horsepower  weight  \
0           1  18.0          8         307.0         130    3504
1           2  15.0          8         350.0         165    3693
2           3  18.0          8         318.0         150    3436
3           4  16.0          8         304.0         150    3433
4           5  17.0          8         302.0         140    3449

   acceleration  year  origin                       name  mpg01
0          12.0    70       1  chevrolet chevelle malibu      0
1          11.5    70       1          buick skylark 320      0
2          11.0    70       1         plymouth satellite      0
3          12.0    70       1              amc rebel sst      0
4          10.5    70       1                ford torino      0
```

[110]:
```python
# (b)

import seaborn as sns
import matplotlib.pyplot as plt

import matplotlib.pyplot as plt

plt.subplots(2, 4, figsize=(12, 8))

# Boxplot for mpg vs. mpg01
plt.subplot(2, 4, 1)
plt.boxplot([df_auto['mpg'][df_auto['mpg01'] == 0],␣
 ↪df_auto['mpg'][df_auto['mpg01'] == 1]])
plt.xlabel('mpg01')
plt.ylabel('mpg')
plt.title('mpg vs. mpg01')

# Boxplot for cylinders vs. mpg01
plt.subplot(2, 4, 2)
```

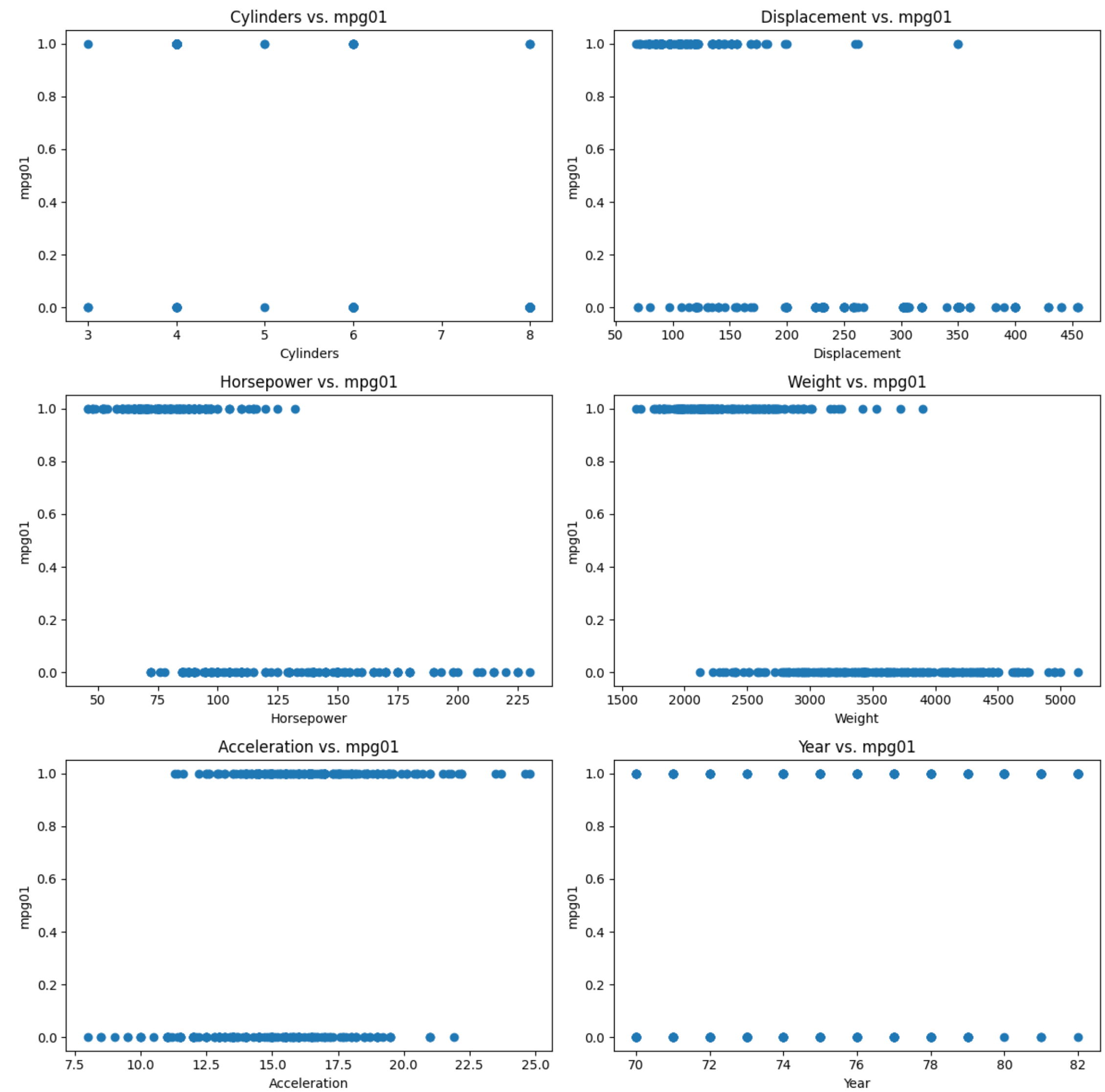
```python
plt.boxplot([df_auto['cylinders'][df_auto['mpg01'] == 0], df_auto['cylinders'][df_auto['mpg01'] == 1]])
plt.xlabel('mpg01')
plt.ylabel('cylinders')
plt.title('cylinders vs. mpg01')

# Boxplot for displacement vs. mpg01
plt.subplot(2, 4, 3)
plt.boxplot([df_auto['displacement'][df_auto['mpg01'] == 0], df_auto['displacement'][df_auto['mpg01'] == 1]])
plt.xlabel('mpg01')
plt.ylabel('displacement')
plt.title('displacement vs. mpg01')

# Boxplot for horsepower vs. mpg01
plt.subplot(2, 4, 4)
plt.boxplot([df_auto['horsepower'][df_auto['mpg01'] == 0], df_auto['horsepower'][df_auto['mpg01'] == 1]])
plt.xlabel('mpg01')
plt.ylabel('horsepower')
plt.title('horsepower vs. mpg01')

# Boxplot for weight vs. mpg01
plt.subplot(2, 4, 5)
plt.boxplot([df_auto['weight'][df_auto['mpg01'] == 0], df_auto['weight'][df_auto['mpg01'] == 1]])
plt.xlabel('mpg01')
plt.ylabel('weight')
plt.title('weight vs. mpg01')

# Boxplot for acceleration vs. mpg01
plt.subplot(2, 4, 6)
plt.boxplot([df_auto['acceleration'][df_auto['mpg01'] == 0], df_auto['acceleration'][df_auto['mpg01'] == 1]])
plt.xlabel('mpg01')
plt.ylabel('acceleration')
plt.title('acceleration vs. mpg01')

# Boxplot for origin vs. mpg01
plt.subplot(2, 4, 7)
plt.boxplot([df_auto['origin'][df_auto['mpg01'] == 0], df_auto['origin'][df_auto['mpg01'] == 1]])
plt.xlabel('mpg01')
plt.ylabel('origin')
plt.title('origin vs. mpg01')
```

```
# Boxplot for year vs. mpg01
plt.subplot(2, 4, 8)
plt.boxplot([df_auto['year'][df_auto['mpg01'] == 0],␣
↪df_auto['year'][df_auto['mpg01'] == 1]])
plt.xlabel('mpg01')
plt.ylabel('Year')
plt.title('Year vs. mpg01')

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



[111]:
```
plt.subplots(3, 2, figsize=(12, 12))

# Scatterplot for cylinders vs. mpg01
plt.subplot(3, 2, 1)
plt.scatter(df_auto['cylinders'], df_auto['mpg01'])
plt.xlabel('Cylinders')
plt.ylabel('mpg01')
plt.title('Cylinders vs. mpg01')

# Scatterplot for displacement vs. mpg01
plt.subplot(3, 2, 2)
plt.scatter(df_auto['displacement'], df_auto['mpg01'])
```

```
plt.xlabel('Displacement')
plt.ylabel('mpg01')
plt.title('Displacement vs. mpg01')

# Scatterplot for horsepower vs. mpg01
plt.subplot(3, 2, 3)
plt.scatter(df_auto['horsepower'], df_auto['mpg01'])
plt.xlabel('Horsepower')
plt.ylabel('mpg01')
plt.title('Horsepower vs. mpg01')

# Scatterplot for weight vs. mpg01
plt.subplot(3, 2, 4)
plt.scatter(df_auto['weight'], df_auto['mpg01'])
plt.xlabel('Weight')
plt.ylabel('mpg01')
plt.title('Weight vs. mpg01')

# Scatterplot for acceleration vs. mpg01
plt.subplot(3, 2, 5)
plt.scatter(df_auto['acceleration'], df_auto['mpg01'])
plt.xlabel('Acceleration')
plt.ylabel('mpg01')
plt.title('Acceleration vs. mpg01')

# Scatterplot for year vs. mpg01
plt.subplot(3, 2, 6)
plt.scatter(df_auto['year'], df_auto['mpg01'])
plt.xlabel('Year')
plt.ylabel('mpg01')
plt.title('Year vs. mpg01')

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



[112]:
```
# Comparing distributions between features with above-median and below-median
mpg, boxplots reveal outliers in the former, particularly with values
exceeding those of four-cylinder engines. Cars with above-median mpg
typically feature smaller engine capacities compared to those with
below-median mpg, which holds true for factors like horsepower and weight.
However, there isn't a significant difference in acceleration and year.
Notably, cylinders, displacement, horsepower, and weight contribute
significantly to predicting mpg01.
```

```
# In contrast, scatterplots demonstrate clearer distinctions between features
↪with above-median and below-median mpg. Key influencing factors include
↪horsepower, acceleration, and weight, where values show less overlap
↪compared to other predictors.
```

[113]:
```
# (c)

from sklearn.model_selection import train_test_split
import numpy as np
np.random.seed(1)

train, test = train_test_split(df_auto, test_size=0.5, random_state=22)

print(test.head())
```

```
     Unnamed: 0   mpg  cylinders  displacement  horsepower  weight  \
280         283  22.3          4         140.0          88    2890
57           59  25.0          4          97.5          80    2126
46           48  19.0          6         250.0         100    3282
223         226  17.5          6         250.0         110    3520
303         306  28.4          4         151.0          90    2670

     acceleration  year  origin                  name  mpg01
280          17.3    79       1        ford fairmont 4      0
57           17.0    72       1      dodge colt hardtop      1
46           15.0    71       1        pontiac firebird      0
223          16.4    77       1      chevrolet concours      0
303          16.0    79       1  buick skylark limited      1
```

[114]:
```
# (d)

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.metrics import confusion_matrix, accuracy_score

predictors = ['cylinders', 'weight', 'displacement', 'horsepower']

# Fit LDA model
lda_model = LinearDiscriminantAnalysis()
lda_model.fit(train[predictors], train['mpg01'])

lda_pred = lda_model.predict(test[predictors])

conf_matrix_d = confusion_matrix(test['mpg01'], lda_pred)

test_error_d = 1 - accuracy_score(test['mpg01'], lda_pred)

print("Confusion Matrix:")
print(conf_matrix_d)
```

```
print("Test Error:", test_error_d)
```

```
Confusion Matrix:
[[87 14]
 [ 7 88]]
Test Error: 0.1071428571428571
```

[115]:
```
# (e)

from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

# Fit QDA model
qda_model = QuadraticDiscriminantAnalysis()
qda_model.fit(train[predictors], train['mpg01'])

qda_pred = qda_model.predict(test[predictors])

conf_matrix_e = confusion_matrix(test['mpg01'], qda_pred)

test_error_e = 1 - accuracy_score(test['mpg01'], qda_pred)

print("Confusion Matrix:")
print(conf_matrix_e)
print("Test Error:", test_error_e)
```

```
Confusion Matrix:
[[89 12]
 [ 9 86]]
Test Error: 0.1071428571428571
```

[116]:
```
# (f)

from sklearn.linear_model import LogisticRegression

# Fit logistic regression model
glm_model = LogisticRegression(max_iter=1000)
glm_model.fit(train[predictors], train['mpg01'])

# Predict probabilities for test data
probs = glm_model.predict_proba(test[predictors])[:, 1]

# Assign class labels based on probability threshold of 0.5
pred_glm = (probs > 0.5).astype(int)

# Compute confusion matrix
conf_matrix_f = confusion_matrix(test['mpg01'], pred_glm)

# Compute test error
```

```
test_error_f = 1 - accuracy_score(test['mpg01'], pred_glm)

print("Confusion Matrix:")
print(conf_matrix_f)
print("Test Error:", test_error_f)
```

```
Confusion Matrix:
[[92  9]
 [ 8 87]]
Test Error: 0.08673469387755106
```

[117]:
```
# (g)

from sklearn.naive_bayes import GaussianNB

# Fit Naive Bayes model
nb_model = GaussianNB()
nb_model.fit(train[predictors], train['mpg01'])

nb_pred = nb_model.predict(test[predictors])

conf_matrix_g = confusion_matrix(test['mpg01'], nb_pred)

test_error_g = 1 - accuracy_score(test['mpg01'], nb_pred)

print("Confusion Matrix:")
print(conf_matrix_g)
print("Test Error:", test_error_g)
```

```
Confusion Matrix:
[[88 13]
 [ 8 87]]
Test Error: 0.10714285714285711
```

[118]:
```
# 5

# (a)

import statsmodels.api as sm

df_de = pd.read_csv('/Users/rouren/Desktop/24S ML/HW/hw3/Data-Default.csv')

np.random.seed(1)
X = df_de[['income', 'balance']]
y = df_de['default']

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

```
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)

print("Coefficients:", log_reg.coef_)
print("Intercept:", log_reg.intercept_)
```

```
Coefficients: [[-0.0001229   0.00040355]]
Intercept: [-1.12688016e-06]
```

[119]:
```
# (b)

X_train, X_val, y_train, y_val = train_test_split(df_de[['income', 'balance']],␣
↪df_de['default'], test_size=0.5)

log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)

y_pred = log_reg.predict(X_val)

validation_error = np.mean(y_pred != y_val)
print("Validation Error:", validation_error)
```

```
Validation Error: 0.0244
```

[154]:
```
# (c)

num_repetitions = 3
validation_errors_without_student = []

for i in range(num_repetitions):
    X_train, X_val, y_train, y_val = train_test_split(df_de[['income',␣
↪'balance']], df_de['default'], test_size=0.5)
    log_reg = LogisticRegression()
    log_reg.fit(X_train, y_train)
    y_pred = log_reg.predict(X_val)
    validation_error = np.mean(y_pred != y_val)
    validation_errors_without_student.append(validation_error)

    print(f"Validation error for repetition {i+1}: {validation_error}")

print("Validation errors for all repetitions:",␣
↪validation_errors_without_student)

# Although there is some variability, all errors are relatively low, indicating␣
↪consistent model performance across different data splits. This suggests␣
↪that the model generalizes well and is robust.
```

```
Validation error for repetition 1: 0.0314
Validation error for repetition 2: 0.0326
```

```
Validation error for repetition 3: 0.0348
Validation errors for all repetitions: [0.0314, 0.0326, 0.0348]
```

[121]:
```python
# (d)

X = df_de[['income', 'balance', 'student']]
y = df_de['default']

encoder = OneHotEncoder(drop='first')
X_encoded = pd.DataFrame(encoder.fit_transform(X[['student']]).toarray(),
 ↪columns=['student'])

X = X.drop(columns=['student'])
X = pd.concat([X, X_encoded], axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)

log_reg_with_student = LogisticRegression()
log_reg_with_student.fit(X_train, y_train)

y_pred_with_student = log_reg_with_student.predict(X_test)

validation_error_with_student = np.mean(y_pred_with_student != y_test)
print("Validation Error with student variable:", validation_error_with_student)

avg_validation_errors = np.mean(validation_errors_without_student)
print("Validation Error without student variable:", avg_validation_errors)

if validation_error_with_student < avg_validation_errors:
    print("Including the dummy variable for student leads to a reduction in the
 ↪test error rate.")
else:
    print("Including the dummy variable for student does not lead to a
 ↪reduction in the test error rate.")
```

```
Validation Error with student variable: 0.0346
Validation Error without student variable: 0.032
Including the dummy variable for student does not lead to a reduction in the
test error rate.
```