**STAT 4360 (Introduction to Statistical Learning, Spring 2023)**

**Mini Project 3  
Name: Ann Biju**

1. (a)

A screenshot of a computer

Description automatically generated

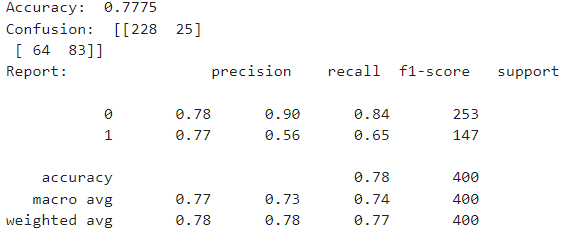
*Some summary statistics. Glucose seems to have the highest mean and mean BMI falls in the obese category.*

A screenshot of a graph

Description automatically generated

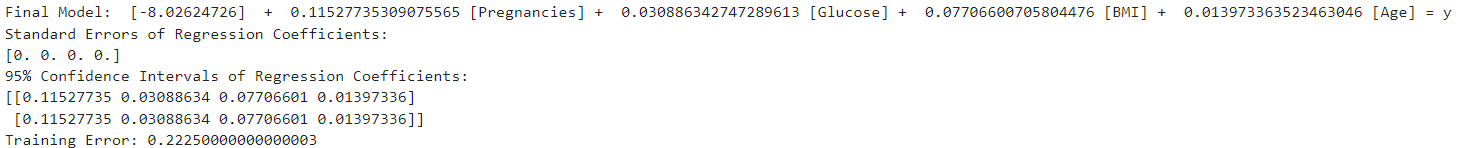
*Correlation heat map between the predictors shows that Outcome has the strongest relationship between Glucose. Pregnancy, Glucose, BMI, and Age are the most relevant features.*

1b.



*Model accuracy of 0.78 is not excellent but not weak either.*

1c.



Final model shows Pregnancies having the strongest or highest regression coefficient (0.115) out of those of Glucose (0.03), BMI (0.077), and Age (0.01).

2a.)

A screenshot of a computer

Description automatically generated

Error rate: 0.2149 – low error rate

Sensitivity: 0.5986 – True positive rate

Specificity: 0.8932 – True negative rate

2c.



The error rate went up slightly when using LOOCV.

2d.



2e.



2f.



2g.



2h. I would recommend the KNN classifier, since it has the lowest error rate.

3a.

A graph with a line and numbers

Description automatically generated with medium confidence

*Scatter plot of Method 2 (POS) v. Method 1 (OSM) with a superimposed 45 degree line.*

The scatterplot shows agreement between the 2 methods, but they are not in perfect agreement since the data points are not all exactly falling on the 45 degree line.

A graph with a line

Description automatically generated

*Boxplot of Absolute value of differences between the 2 methods (OSM and POS)*

The boxplot shows a median value close to 1.0, while only the minimum value is equal to 0. This shows that most of the differences between the 2 methods are 1.0 or higher. Hence, the methods are not in perfect agreement with each other.

3b.

A graph of a number of columns

Description automatically generated

*Histogram showing the absolute differences between the 2 methods and TDI estimate, given p = 0.90*

This histogram indicates that when theta (or the TDI) is small, most of the absolute differences are concentrated on the left side of the distribution, showing better agreement. If theta had been large, the differences would be centered mostly on the right side of the distribution.

3c. The point estimate of theta is 2.00. The code is attached below.

3d.

A number and numbers on a white background

Description automatically generated

Bias measures how much the bootstrap estimate deviates from the true population parameter, or theta. A small bias of 0.00639 indicates small deviation from theta. The standard error measures variability of bootstrap estimates. A small standard error of 0.13142 shows that the data is precise.

3f. The measure from 3e. had an upper confidence bound of 1.2444. I would conclude that the 2 methods cannot be used interchangeably in practice, since they have varying upper confidence interval bounds and also since from part 3a I determined that the 2 methods are not in perfect agreement with each other.

4a.

A math equations on a grid paper

Description automatically generated

4b.

A math equations on a grid

Description automatically generated

**Python Code (or R Code)**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

import sklearn as sk

from sklearn.model\_selection import train\_test\_split, LeaveOneOut

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

from sklearn.discriminant\_analysis import QuadraticDiscriminantAnalysis, LinearDiscriminantAnalysis

import statsmodels.api as sm

from sklearn.neighbors import KNeighborsClassifier

#Question 1a

# import data

data = pd.read\_csv("diabetes.csv")

# Summary statistics

print(data.describe())

# Calculate the correlation matrix

correlation\_matrix = data.corr()

correlation\_matrix

# Create a heatmap

plt.figure(figsize=(8, 6))

sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm', linewidths=0.5)

plt.show()

# 1b

# Split the data into training and testing sets

X = data[['Pregnancies \n', 'Glucose \n', 'BMI \n', 'Age \n']]

y = data['Outcome ']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize and train the logistic regression model

model1 = LogisticRegression(solver='lbfgs', max\_iter=1000)

model1 = model1.fit(X\_train, y\_train)

# Get the coefficients

coefficients = model1.coef\_

# Print the coefficients

print("Model Coefficients:")

coef\_p, coef\_b, coef\_a, coef\_g = "", "", "", ""

for feature, coef in zip(X\_train.columns, coefficients[0]):

print(f"{feature}: {coef}")

if feature == "Pregnancies \n":

coef\_p = coef

if feature =="Glucose \n":

coef\_g = coef

if feature == "BMI \n":

coef\_b = coef

if feature == "Age \n":

coef\_a = coef

# Assuming you've already trained your logistic regression model, 'model'

intercept = model1.intercept\_

print(f"Intercept (beta0): {intercept[0]}")

# Make predictions on the test set

y\_pred = model1.predict(X\_test)

# Get the indices of the training and testing data

train\_indices = X\_train.index

test\_indices **=** X\_test**.**index

# Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

confusion = confusion\_matrix(y\_test, y\_pred)

report = classification\_report(y\_test, y\_pred)

print("Accuracy: ", accuracy)

print("Confusion: ", confusion)

print("Report: ", report)

#1c

print("Final Model: ",intercept, " + ", coef\_p, "[Pregnancies] + ",

coef\_g, "[Glucose] + ", coef\_b,"[BMI] + ", coef\_a,"[Age] = y")

# Calculate the standard errors

standard\_errors = np.std(coefficients, axis=0)

# Print the standard errors of coefficients

print("Standard Errors of Regression Coefficients:")

print(standard\_errors)

# Calculate 95% confidence intervals

lower\_percentile = 2.5

upper\_percentile = 97.5

confidence\_intervals = np.percentile(coefficients, [lower\_percentile, upper\_percentile], axis=0)

# Print the 95% confidence intervals

print("95% Confidence Intervals of Regression Coefficients:")

print(confidence\_intervals)

# Calculate the training error using accuracy

training\_error = 1 - accuracy\_score(y\_test, y\_pred)

print("Training Error:", training\_error)

#QUESTION 2a

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize and train the logistic regression model

model2a = LogisticRegression(solver='lbfgs', max\_iter=1000)

model2a = model2a.fit(X\_test, y\_test)

# Make predictions on the test set

y\_pred = model2a.predict(X\_test)

# Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

confusion = confusion\_matrix(y\_test, y\_pred)

report = classification\_report(y\_test, y\_pred)

print(f'Accuracy: {accuracy}')

print(f'Confusion Matrix:\n{confusion}')

print(f'Classification Report:\n{report}')

# Calculate sensitivity and specificity

true\_negatives = confusion[0, 0]

false\_positives = confusion[0, 1]

false\_negatives = confusion[1, 0]

true\_positives = confusion[1, 1]

sensitivity = true\_positives / (true\_positives + false\_negatives)

specificity = true\_negatives / (true\_negatives + false\_positives)

error\_rate = 1 - accuracy

print(f'Error Rate: {error\_rate}')

print(f'Sensitivity: {sensitivity}')

print(f'Specificity: {specificity}')

#Question 2b

# Initialize an empty list to store the test error rates

test\_error\_rates = []

# Perform LOOCV

for i in range(len(X)):

# Split the data into training and testing sets for this iteration

X\_train = np.delete(X, i, axis=0)

y\_train = np.delete(y, i)

X\_test = np.array(X.iloc[i,:])

y\_test = y[i]

model2a = LogisticRegression(solver='lbfgs', max\_iter=1000)

model2a.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = model2a.predict(X\_test.reshape(1, -1))

# Calculate the test error rate for this iteration

test\_error = 1 - accuracy\_score([y\_test], y\_pred)

test\_error\_rates.append(test\_error)

# Calculate the average test error rate

average\_test\_error\_rate = np.mean(test\_error\_rates)

print(f'Average Test Error Rate (LOOCV): {average\_test\_error\_rate}')

#2c

model2c = LogisticRegression(solver='lbfgs', max\_iter=1000)

# Initialize the LeaveOneOut cross-validator

loocv1 = LeaveOneOut()

# Initialize a list to store test error rates

test\_error\_rates = []

# Perform LOOCV

for train\_index, test\_index in loocv1.split(X):

X\_train, X\_test = X.iloc[train\_index,:], X.iloc[test\_index,:]

y\_train, y\_test = y[train\_index], y[test\_index]

# Fit the logistic regression model on the training data

model2c.fit(X\_train, y\_train)

# Make predictions on the test data

y\_pred = model2c.predict(X\_test)

# Calculate the test error rate for this iteration

test\_error = 1 - accuracy\_score(y\_test, y\_pred)

test\_error\_rates.append(test\_error)

# Calculate the average test error rate

average\_test\_error\_rate = np.mean(test\_error\_rates)

print(f'Average Test Error Rate (LOOCV): {average\_test\_error\_rate}')

#2d

# Initialize the LeaveOneOut cross-validator

loo = LeaveOneOut()

# Initialize a list to store test error rates

test\_error\_rates = []

# Perform LOOCV

for train\_index, test\_index in loo.split(X):

X\_train, X\_test = X.iloc[train\_index], X.iloc[test\_index]

y\_train, y\_test = y.iloc[train\_index], y.iloc[test\_index]

# Fit the logistic regression model on the training data

model1.fit(X\_train, y\_train)

# Make predictions on the test data

y\_pred = model1.predict(X\_test)

# Calculate the test error rate for this iteration

test\_error = 1 - accuracy\_score(y\_test, y\_pred)

test\_error\_rates.append(test\_error)

# Calculate the average test error rate

average\_test\_error\_rate = np.mean(test\_error\_rates)

print(f'Average Test Error Rate (LOOCV): {average\_test\_error\_rate}')

#2e

lda = LinearDiscriminantAnalysis()

# Initialize the LeaveOneOut cross-validator

loo = LeaveOneOut()

# Initialize a list to store test error rates

test\_error\_rates = []

# Perform LOOCV

for train\_index, test\_index in loo.split(X):

X\_train, X\_test = X.iloc[train\_index], X.iloc[test\_index]

y\_train, y\_test = y.iloc[train\_index], y.iloc[test\_index]

# Fit the logistic regression model on the training data

lda.fit(X\_train, y\_train)

# Make predictions on the test data

y\_pred = lda.predict(X\_test)

# Calculate the test error rate for this iteration

test\_error = 1 - accuracy\_score(y\_test, y\_pred)

test\_error\_rates.append(test\_error)

# Calculate the average test error rate

average\_test\_error\_rate = np.mean(test\_error\_rates)

print(f'Average Test Error Rate (LOOCV): {average\_test\_error\_rate}')

#2f

qda = QuadraticDiscriminantAnalysis()

# Initialize the LeaveOneOut cross-validator

loo = LeaveOneOut()

# Initialize a list to store test error rates

test\_error\_rates = []

# Perform LOOCV

for train\_index, test\_index in loo.split(X):

X\_train, X\_test = X.iloc[train\_index], X.iloc[test\_index]

y\_train, y\_test = y.iloc[train\_index], y.iloc[test\_index]

# Fit the logistic regression model on the training data

qda.fit(X\_train, y\_train)

# Make predictions on the test data

y\_pred = qda.predict(X\_test)

# Calculate the test error rate for this iteration

test\_error = 1 - accuracy\_score(y\_test, y\_pred)

test\_error\_rates.append(test\_error)

# Calculate the average test error rate

average\_test\_error\_rate = np.mean(test\_error\_rates)

print(f'Average Test Error Rate (LOOCV): {average\_test\_error\_rate}')

# 2g

# Initialize the LeaveOneOut cross-validator

loo = LeaveOneOut()

test\_error\_rates = [] # Store test error rates for each LOOCV iteration

knn = KNeighborsClassifier(n\_neighbors=4)

knn.fit(X\_train, y\_train)

# Perform LOOCV

for train\_index, test\_index in loo.split(X):

X\_train, X\_test = X.iloc[train\_index], X.iloc[test\_index]

y\_train, y\_test = y.iloc[train\_index], y.iloc[test\_index]

y\_pred = knn.predict(X\_test)

test\_error = 1 - accuracy\_score(y\_test, y\_pred)

test\_error\_rates.append(test\_error)

avg\_test\_error\_rate = sum(test\_error\_rates) / len(test\_error\_rates)

print("Test Error Rate for Optimal KNN: ", avg\_test\_error\_rate)

# QUESTION 3

oxygen = pd.read\_csv('oxygen\_saturation.txt', sep = '\t')

oxygen.head()

import matplotlib.pyplot as plt

# 3a

#scatterplot

# Extract the data for each method

method1 = oxygen["osm"]

method2 = oxygen["pos"]

# Create a scatterplot

plt.scatter(method1, method2, alpha=0.5, color='blue', label="Data Points")

# Superimpose the 45° line

plt.plot([0, 100], [0, 100], color='red', linestyle='--', label="45° Line")

# Add labels and legend

plt.xlabel("Method 1")

plt.ylabel("Method 2")

plt.legend()

# Display the plot

plt.show()

# Calculate the absolute differences between the two methods

absolute\_differences = abs(method1 - method2)

# Create a boxplot

plt.boxplot(absolute\_differences, vert=False)

plt.xlabel("Absolute Differences")

plt.title("Boxplot of Absolute Differences between the 2 Methods")

# Display the plot

plt.show()

# 3b

# Creating random data for Y1 and Y2, you should replace this with your actual data

Y1 = method1

Y2 = method2

# Calculate the absolute differences (D)

D = np.abs(Y1 - Y2)

# Calculate the θ value for p = 0.90 (90th quantile of |D|)

p = 0.90

theta = np.quantile(D, p) #total derivation index

# Visualize the distribution of |D| and the θ value

plt.hist(D, bins=30, alpha=0.7, color='blue', label='|D|')

plt.axvline(theta, color='red', linestyle='--', label=f'θ = {theta:.2f}')

plt.xlabel('|D|')

plt.ylabel('Frequency')

plt.title('Distribution of Absolute Differences and θ')

plt.legend()

plt.show()

#3c

theta\_hat = np.percentile(D, 90)

print(f"The point estimate of θ (ˆθ) is: {theta\_hat:.2f}")

#3d

data = D

# Number of bootstrap samples to generate

num\_bootstrap\_samples = 1000

# Create an array to store the bootstrap estimates of θ

bootstrap\_theta\_hat = np.zeros(num\_bootstrap\_samples)

# Generate bootstrap samples and estimate θ

for i in range(num\_bootstrap\_samples):

# Resample with replacement from the original data

bootstrap\_sample = np.random.choice(data, size=len(data), replace=True)

# Calculate the 90th sample quantile of the bootstrap sample

bootstrap\_theta\_hat[i] = np.percentile(bootstrap\_sample, 90)

# Calculate the bias of θ

bias = np.mean(bootstrap\_theta\_hat) - np.percentile(data, 90)

# Calculate the standard error of θ

standard\_error = np.std(bootstrap\_theta\_hat)

# Calculate the 95% upper confidence bound using the percentile method

confidence\_level = 95

upper\_confidence\_bound = np.percentile(bootstrap\_theta\_hat, confidence\_level)

print(f"Bias of θ (ˆθ): {bias}")

print(f"Standard Error of θ (ˆθ): {standard\_error}")

print(f"95% Upper Confidence Bound for θ: {upper\_confidence\_bound}")

#3e

from scipy.stats import bootstrap

#Define your sample dataset

sample\_data = np.array([ab\_diff])

#Define a function to compute the statistic of interest (e.g., mean, median, etc.)

def estimate\_statistic(data): return np.mean(data) # Replace with the statistic you want to estimate

#Number of bootstrap resamples

n\_resamples = 1000

#Use the scipy.stats.bootstrap function for resampling

bootstrap\_estimates = bootstrap(sample\_data, estimate\_statistic, random\_state=333)

#Compute the confidence interval for the statistic

confidence\_interval = bootstrap\_estimates.confidence\_interval # You can also specify the confidence level

#Interpret the results

print("Bootstrap Estimates:", bootstrap\_estimates) print("Confidence Interval:", confidence\_interval)