CS 6140: Assignment 4

Luke Davidson

December 3 2022

Exercise 1

The Python script I wrote to solve this problem can be seen below:

```
import numpy as np
import random
import tensorflow as tf
from tensorflow.keras import metrics as Metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

random.seed(1234)
np.random.seed(1234)
class Prob1():
    def __init__(self, info):
        self.info = info
        # self.n = info['n']
        self.n_finegrid = info['n_finegrid']
        self.part = info['part']
        if self.part == "A":
            self.bounds = np.array([[-4, -1, 3, 0],
                                    [-2, 1, -1, -4],
                                    [2, 5, 1, -2]])
        elif self.part == "B":
            self.bounds = np.array([[-4, -3, 3, 2],
                                    [-1, 0, -2, -3],
                                    [2, 3, 0, -1]])
        self.num_epochs = info['num_epochs']
        self.batch_size = info['batch_size']
        self.final_results = {}

    def generate_data(self):
        x1 = np.random.uniform(-6, 6 + 1e-10, self.n).reshape(self.n, 1)
        x2 = np.random.uniform(-4, 4 + 1e-10, self.n).reshape(self.n, 1)
        self.x_data = np.concatenate((x1, x2), axis=1)
        y = []
        for i in range(self.n):
```

```
            inSquare = False
            for pt in range(self.bounds.shape[0]):
                if self.bounds[pt, 0] < x1[i, 0] < self.bounds[pt, 1]
                and self.bounds[pt, 3] < x2[i, 0] < self.bounds[pt, 2]:
                    # Within a square
                    inSquare = True
                    if random.random() > 0.97:
                        # Incorrectly labeles as -
                        y.append(0)
                    else:
                        # Correctly labeled as +
                        y.append(1)
            if inSquare is False:
                # Not within a square
                if random.random() > 0.01:
                    # Correctly labeled as -
                    y.append(0)
                else:
                    # Incorrectly labeled as +
                    y.append(1)
        self.y_data = np.array(y).reshape(self.n, 1)
        self.all_data = np.concatenate((self.x_data, self.y_data),
            axis=1)
        # self.regularize()
        return self.x_data, self.y_data

    def generate_finegrid_data(self):
        x1 = np.random.uniform(-6, 6 + 1e-10,
            self.n_finegrid).reshape(self.n_finegrid, 1)
        x2 = np.random.uniform(-4, 4 + 1e-10,
            self.n_finegrid).reshape(self.n_finegrid, 1)
        self.x_data_finegrid = np.concatenate((x1, x2), axis=1)
        y = []
        for i in range(self.n_finegrid):
            inSquare = False
            for pt in range(self.bounds.shape[0]):
                if self.bounds[pt, 0] < x1[i, 0] < self.bounds[pt, 1]
                and self.bounds[pt, 3] < x2[i, 0] < self.bounds[pt, 2]:
                    # Within a square
                    inSquare = True
                    y.append(1)
            if inSquare is False:
                y.append(0)
        self.y_data_finegrid = np.array(y).reshape(self.n_finegrid, 1)
        self.all_data_finegrid = np.concatenate((self.x_data_finegrid,
            self.y_data_finegrid), axis=1)
        # self.regularize()
        return self.x_data_finegrid, self.y_data_finegrid
```

```
def split_data(self):
    indexes = np.arange(self.all_data.shape[0])
    np.random.shuffle(indexes)
    self.num_train = int(0.7*self.x_data.shape[0])
    self.train_data = self.all_data[indexes[:self.num_train], :]
    self.test_data = self.all_data[indexes[self.num_train:], :]
    self.x_train = self.train_data[:, [0, 1]]
    self.y_train = self.train_data[:, 2]
    self.x_test = self.test_data[:, [0, 1]]
    self.y_test = self.test_data[:, 2]

def regularize(self):
    self.x_data[:, 0] /= 6
    self.x_data[:, 1] /= 6

def calculate_balanced_acc(self, FN, FP, TN, TP):
    sensitivity = TP/(TP+FN)
    specificity = TN/(TN+FP)
    return (sensitivity + specificity)/2

def NN(self, h1, h2):
    if h2 == 0:
        model = Sequential()
        model.add(Dense(h1, input_shape=(2,),
            activation=tf.keras.activations.tanh))
        model.add(Dense(1, activation=tf.keras.activations.tanh))
        model.compile(loss='binary_crossentropy', optimizer='adam',
            metrics=['accuracy', tf.keras.metrics.AUC(),
                     tf.keras.metrics.FalseNegatives(),
                     tf.keras.metrics.FalsePositives(),
                     tf.keras.metrics.TrueNegatives(),
                     tf.keras.metrics.TruePositives()])
        model.fit(self.x_train, self.y_train,
            epochs=self.num_epochs, batch_size=self.batch_size)
        _, class_accuracy, auc, FN, FP, TN, TP =
            model.evaluate(self.x_test, self.y_test)
        balanced_acc = self.calculate_balanced_acc(FN, FP, TN, TP)
        _, true_performance, _, _, _, _, _ =
            model.evaluate(self.x_data_finegrid,
                self.y_data_finegrid,
                batch_size=self.x_data_finegrid.shape[0])
    else:
        model = Sequential()
        model.add(Dense(h1, input_shape=(2,),
            activation=tf.keras.activations.tanh))
        model.add(Dense(h2, activation=tf.keras.activations.tanh))
        model.add(Dense(1, activation=tf.keras.activations.tanh))
```

```
            model.compile(loss='binary_crossentropy', optimizer='adam',
                metrics=['accuracy', tf.keras.metrics.AUC(),
                                tf.keras.metrics.FalseNegatives(),
                                tf.keras.metrics.FalsePositives(),
                                tf.keras.metrics.TrueNegatives(),
                                tf.keras.metrics.TruePositives()])
            model.fit(self.x_train, self.y_train,
                epochs=self.num_epochs, batch_size=self.batch_size)
            _, class_accuracy, auc, FN, FP, TN, TP =
                model.evaluate(self.x_test, self.y_test)
            balanced_acc = self.calculate_balanced_acc(FN, FP, TN, TP)
            _, true_performance, _, _, _, _, _ =
                model.evaluate(self.x_data_finegrid,
                    self.y_data_finegrid,
                    batch_size=self.x_data_finegrid.shape[0])
        return (class_accuracy, balanced_acc, auc, true_performance)

    def save_data(self, results, h1, h2):
        class_accuracy, balanced_acc, auc, true_performance = results
        name = str(self.n) + "_" + str(h1) + "_" + str(h2)
        self.final_results.update({name: [class_accuracy, balanced_acc,
            auc, true_performance]})

    def display_data(self):
        print("\n\n========================")
        print("===== FINAL RESULTS =====")
        print("========================")
        for k, v in self.final_results.items():
            n = k.split('_')[0]
            h1 = k.split('_')[1]
            h2 = k.split('_')[2]
            print(f"N: {n}, H1: {h1}, H2: {h2}")
            print(f"\tClassification Accuracy: {round(v[0]*100, 2)}%")
            print(f"\tBalanced Accuracy: {round(v[1]*100, 2)}%")
            print(f"\tROC AUC Estimate: {round(v[2], 3)}")
            print(f"\tTrue Performance: {round(v[3]*100, 2)}%\n")

    def run(self):
        x_fg, y_fg, = self.generate_finegrid_data()
        for n in [250, 1_000, 10_000]:
            self.n = n
            x, y = self.generate_data()
            self.split_data()
            for h1 in [1, 4, 12]:
                for h2 in [0, 3]:
                    results = self.NN(h1, h2)
                    self.save_data(results, h1, h2)
                    print("\n\n========================================")
```

```
                    print(f"===== DONE WITH N: {n} H1: {h1} H2: {h2}")
                    print("=========================================\n\n")
        self.display_data()

# Run Part A
params = {"n_finegrid": 1_000_000,
          "part": "A",
          "num_epochs": 150,
          "batch_size": 32}
prob = Prob1(params)
prob.run()

# Run Part B
params = {"n_finegrid": 1_000_000,
          "part": "B",
          "num_epochs": 150,
          "batch_size": 32}
prob = Prob1(params)
prob.run()
```

To briefly explain the script, the data is generated according to a given number of data points "n", in our case {250, 1000, 10000}. The x_1 and x_2 coordinates are drawn uniformly from the ranges [-6, 6] and [-4, 4], respectively. Each data point is then checked to see if it falls within one of the positively labeled squares for the respective part and is given a class label according to the posteriors given in the problem description. 1,000,000 points of fine grid data (no label noise) is also generated to check for the "true performance" of each model. The data is then split in to training and test data for cross validation. For every (h_1, h_2) combination, a neural network is created using TensorFlow. Each network is trained on the training data and testing results are found by evaluating the test data. The following results are printed after each part is executed:

Part A:

```
=========================
===== FINAL RESULTS =====
=========================
N: 250, H1: 1, H2: 0
        Classification Accuracy: 74.67%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.458
        True Performance: 71.88%

N: 250, H1: 1, H2: 3
        Classification Accuracy: 74.67%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.588
        True Performance: 71.88%

N: 250, H1: 4, H2: 0
```

```
        Classification Accuracy: 80.0%
        Balanced Accuracy: 70.96%
        ROC AUC Estimate: 0.781
        True Performance: 77.41%

N: 250, H1: 4, H2: 3
        Classification Accuracy: 74.67%
        Balanced Accuracy: 67.39%
        ROC AUC Estimate: 0.797
        True Performance: 70.52%

N: 250, H1: 12, H2: 0
        Classification Accuracy: 84.0%
        Balanced Accuracy: 73.64%
        ROC AUC Estimate: 0.82
        True Performance: 82.5%

N: 250, H1: 12, H2: 3
        Classification Accuracy: 81.33%
        Balanced Accuracy: 77.07%
        ROC AUC Estimate: 0.86
        True Performance: 73.29%

N: 1000, H1: 1, H2: 0
        Classification Accuracy: 71.67%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.619
        True Performance: 71.88%

N: 1000, H1: 1, H2: 3
        Classification Accuracy: 71.67%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.57
        True Performance: 71.88%

N: 1000, H1: 4, H2: 0
        Classification Accuracy: 79.0%
        Balanced Accuracy: 63.3%
        ROC AUC Estimate: 0.641
        True Performance: 77.85%

N: 1000, H1: 4, H2: 3
        Classification Accuracy: 72.67%
        Balanced Accuracy: 56.39%
        ROC AUC Estimate: 0.833
        True Performance: 72.13%

N: 1000, H1: 12, H2: 0
```

```
        Classification Accuracy: 83.0%
        Balanced Accuracy: 75.69%
        ROC AUC Estimate: 0.929
        True Performance: 81.8%

N: 1000, H1: 12, H2: 3
        Classification Accuracy: 90.67%
        Balanced Accuracy: 88.15%
        ROC AUC Estimate: 0.96
        True Performance: 88.07%

N: 10000, H1: 1, H2: 0
        Classification Accuracy: 72.27%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.622
        True Performance: 71.88%

N: 10000, H1: 1, H2: 3
        Classification Accuracy: 81.37%
        Balanced Accuracy: 71.37%
        ROC AUC Estimate: 0.837
        True Performance: 79.87%

N: 10000, H1: 4, H2: 0
        Classification Accuracy: 84.57%
        Balanced Accuracy: 78.17%
        ROC AUC Estimate: 0.916
        True Performance: 84.52%

N: 10000, H1: 4, H2: 3
        Classification Accuracy: 85.23%
        Balanced Accuracy: 81.82%
        ROC AUC Estimate: 0.931
        True Performance: 85.82%

N: 10000, H1: 12, H2: 0
        Classification Accuracy: 88.03%
        Balanced Accuracy: 84.46%
        ROC AUC Estimate: 0.941
        True Performance: 88.84%

N: 10000, H1: 12, H2: 3
        Classification Accuracy: 83.67%
        Balanced Accuracy: 86.11%
        ROC AUC Estimate: 0.931
        True Performance: 84.33%
```

Part B:

```
=========================
===== FINAL RESULTS =====
=========================
N: 250, H1: 1, H2: 0
        Classification Accuracy: 96.0%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.616
        True Performance: 96.9%

N: 250, H1: 1, H2: 3
        Classification Accuracy: 96.0%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.463
        True Performance: 96.9%

N: 250, H1: 4, H2: 0
        Classification Accuracy: 90.67%
        Balanced Accuracy: 47.22%
        ROC AUC Estimate: 0.88
        True Performance: 93.93%

N: 250, H1: 4, H2: 3
        Classification Accuracy: 96.0%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.644
        True Performance: 96.9%

N: 250, H1: 12, H2: 0
        Classification Accuracy: 96.0%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.979
        True Performance: 96.7%

N: 250, H1: 12, H2: 3
        Classification Accuracy: 96.0%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.743
        True Performance: 96.9%

N: 1000, H1: 1, H2: 0
        Classification Accuracy: 95.33%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.539
        True Performance: 96.9%

N: 1000, H1: 1, H2: 3
        Classification Accuracy: 95.33%
        Balanced Accuracy: 50.0%
```

```
        ROC AUC Estimate: 0.589
        True Performance: 96.9%

N: 1000, H1: 4, H2: 0
        Classification Accuracy: 95.33%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.644
        True Performance: 96.9%

N: 1000, H1: 4, H2: 3
        Classification Accuracy: 95.33%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.764
        True Performance: 96.9%

N: 1000, H1: 12, H2: 0
        Classification Accuracy: 95.33%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.891
        True Performance: 96.9%

N: 1000, H1: 12, H2: 3
        Classification Accuracy: 95.33%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.781
        True Performance: 96.9%

N: 10000, H1: 1, H2: 0
        Classification Accuracy: 96.0%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.577
        True Performance: 96.9%

N: 10000, H1: 1, H2: 3
        Classification Accuracy: 96.0%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.613
        True Performance: 96.9%

N: 10000, H1: 4, H2: 0
        Classification Accuracy: 96.0%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.819
        True Performance: 96.9%

N: 10000, H1: 4, H2: 3
        Classification Accuracy: 96.0%
        Balanced Accuracy: 50.0%
```

```
        ROC AUC Estimate: 0.786
        True Performance: 96.9%

N: 10000, H1: 12, H2: 0
        Classification Accuracy: 96.0%
        Balanced Accuracy: 50.0%
        ROC AUC Estimate: 0.5
        True Performance: 96.9%

N: 10000, H1: 12, H2: 3
        Classification Accuracy: 96.3%
        Balanced Accuracy: 54.15%
        ROC AUC Estimate: 0.729
        True Performance: 97.27%
```

There are a few interesting observations when analyzing the results. Starting with Part A, it is clear that the "classification accuracy" for each network is typically slightly better than the "true performance" (the classification accuracy for the non-noisy data labels). This is likely due to the fact that the models were trained and tested on the noisy data, leading to a slightly better fit for the noisy labels than for the non-noisy data when comparing a lot of data points. Next, classification accuracies of greater than 84% were achieved for the (n, h_1, h_2) combinations of ((250, 12, 0), (1000, 12, 3), (10000, 12, 0), (10000, 4, 3), (10000, 12, 3), (10000, 4, 0)). A general pattern can be seen in the data that higher accuracies were reached as n increased, likely mainly due to more data points being used during the training process. Because there is noise, better results will be acquired when more outliers are seen in the training data. It is also clear that the neural networks that with a higher h_1 (12) tend to perform better due to their ability to handle more complex approximations, like data with noise. This is also seen in the ROC AUC calculations. The ROC AUC calculations are much lower for the networks with small h_1 and h_2 values (ex. (1, 0), (1, 3)), showing worse performance and an increase in false labels for those models.

In looking at the results from Part B, both classification accuracies and true performances are much higher compared to part A. This is likely due to the fact that since the positively labeled areas are so much smaller, there is a much smaller chance a uniformly random point lies within one of these regions, and thus a much smaller percentage of mislabeled points. There is also a clear trend in the balanced accuracy calculations: it is exactly 50% for almost all (n, h_1, h_2) combinations, and is consistently lower than all of those from part A. This is because balanced accuracy best relates the accuracies of each class. There will be many more negatively labeled data points in this set compared to part A due to the decreased area of the positively labeled regions. Therefore there will be a lot more incorrectly labeled negative points compared to positive points, and that is clearly seen in the low balanced accuracy score, and not so much in the classification accuracy.

Exercise 2

a.) To solve for the ALS equation for each u_i, we will want to minimize

$$\| Vu_i - x_i \|^2 + \gamma \| u_i \|^2$$

with respect to u_i. Deriving each term and setting equal to 0 (to find min) leads to:

$$2(u_i^T V^T - x_i)V + 2\gamma u_i^T = 0$$

Expanding...

$$-x_i V + u_i^T V^T V + \gamma u_i^T = 0$$

Separating u_i^T...

$$u_i^T (V^T V + \gamma I) = x_i V$$

Leading to our derivation of formula #1:

$$u_i^T = x_i V (V^T V + \gamma I)^{-1} \tag{2.1}$$

for every i = 1 to n.

The derivation for formula #2 is very similar, just now assuming we have a constant U and varying j 1 to d. The equation we want to minimize is then

$$\| Uv_j - x_j \|^2 + \gamma \| v_j \|^2$$

We follow the same steps as above. Starting by deriving each term and expanding...

$$2(v_j^T U^T - x_j)U + 2\gamma v_j^T = 0$$

$$-x_j U + v_j^T U^T U + \gamma v_j^T = 0$$

Separating v_j^T...

$$v_j^T (U^T U + \gamma I) = x_j U$$

And finally leading to our derivation of formula #2:

$$v_j^T = x_j U (U^T U + \gamma I)^{-1} \tag{2.2}$$

for every j = 1 to d.

b.) If some values in X are missing (which is the practical application), the main error equation we will want to minimize becomes

$$min_{U,V} \sum_{i,j} (w_{ij}(x_{ij} - u_i^T v_j)^2) + \gamma (\sum_i \| u_i \|^2 + \sum_j \| v_j \|^2) \tag{2.3}$$

where w represents a 1-hot feature matrix where $w_{ij} = 1$ where X_{ij} is known, and 0 otherwise. We now follow the same derivation process with the added w feature matrix. Expansion and simplifcation (factor out 2s) leads to...

$$-w_i x_i V + u_i^T w_i V \| V \|^2 + \gamma u_i^T = 0$$

Separating u_i^T...

$$u_i^T (w_i V \| V \|^2 + \gamma I) = w_i x_i V$$

and finally our derivation with the 1-hot feature weight matrix included:

$$u_i^T = (w_i x_i) V (w_i V \| V \|^2 + \gamma I)^{-1} \tag{2.4}$$

The same exact process can be followed for finding v_j^T, similar to above. This will lead to the update equation of:

$$v_j^T = (w_j x_j) U (w_j U \| U \|^2 + \gamma I)^{-1} \tag{2.5}$$

Exercise 3

Visuals of 2-D input, single output 3 layer and single layer neural networks are shown below:

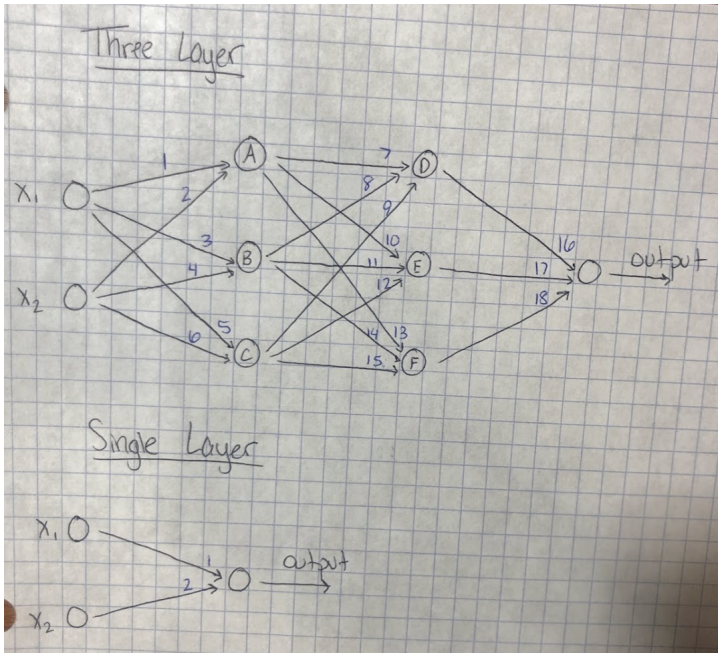

Figure 1: 3-Layer and Single Layer Neural Network Architectures

In the above figure, the blue numbers represent each individual weight in the network. Each node is the of a sum of the previous nodes multiplied by the respective weight. The output of each node will be the result of an activation. In this case, the activation function is linear, so each output is the same as the sum calculation (f(x) = x).

The output for the single layer is trivial and is calculated as

$$output_{1layer} = x_1 w_1 + x_2 w_2$$

Now we solve for the output of the three layer network. We start by calculating nodes A, B, and C:

$$A = x_1 w_1 + x_2 w_2$$

$$B = x_1 w_3 + x_2 w_4$$

$$C = x_1 w_5 + x_2 w_6$$

We can now solve for D, E, and F:

$$D = Aw_7 + Bw_8 + Cw_9 = (x_1 w_1 + x_2 w_2)w_7 + (x_1 w_3 + x_2 w_4)w_8 + (x_1 w_5 + x_2 w_6)w_9$$

$$E = Aw_{10} + Bw_{11} + Cw_{12} = (x_1 w_1 + x_2 w_2)w_{10} + (x_1 w_3 + x_2 w_4)w_{11} + (x_1 w_5 + x_2 w_6)w_{12}$$

$$F = Aw_1 3 + Bw_1 4 + Cw_1 5 = (x_1 w_1 + x_2 w_2)w_{13} + (x_1 w_3 + x_2 w_4)w_{14} + (x_1 w_5 + x_2 w_6)w_{15}$$

And now for the output:

$$output_{3layer} = Dw_{16} + Ew_{17} + Fw_{18}$$

where D, E and F are above.

It is clearly seen that the expansion of the above equations will lead to a linear combination of x_1, x_2 and a combination of weights. That is:

$$output_{3layer} = x_1 W_1 + x_2 W_2$$

where W_1 and W_2 are combinations of all of the above weights 1-18. All weights in this problem are nothing but numbers, so equivalence of outputs is proved between the single layer and 3 layer networks for any weight values where:

$$w_1 = W_1$$

$$w_2 = W_2$$

with w_1 and w_2 from the single layer, and W_1 and W_2 from the 3 layer.

Exercise 4

The Python script I wrote to solve this problem can be seen below:

```
import numpy as np
import random
import tensorflow as tf
from tensorflow.keras import metrics as Metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt
from sklearn.metrics import roc_auc_score
# from sklearn.linear_model import LogisticRegression
import sklearn.linear_model as lm

class q4():
    def __init__(self, info):
        # P(Y=0), P(Y=1)
        self.p_Y0 = info['p_Y0']
        self.p_Y1 = 1-self.p_Y0
        self.p_Y0_b = info['p_Y0_b']
        self.p_Y1_b = 1-self.p_Y0_b

        # d = num training pts, n = num test points
        self.d = info['d']
        self.n = self.d/10

        # Number of each class to create
        self.num_zero = int(self.d*self.p_Y0)
        self.num_one = int(self.d - self.num_zero)
        self.num_zero_a = int(self.n*self.p_Y0)
        self.num_one_a = int(self.n-self.num_zero_a)
        self.num_zero_b = int(self.n*self.p_Y0_b)
        self.num_one_b = int(self.n-self.num_zero_b)

        # Normal distributions
        self.mu_0 = [info['mu_0'], info['mu_0']]
        self.mu_1 = [info['mu_1'], info['mu_1']]
        self.sig_00 = info['sig_0'][0]
        self.sig_01 = info['sig_0'][1]
        self.sig_10 = info['sig_1'][0]
        self.sig_11 = info['sig_1'][1]
        self.sig_00_c = info['sig_0_c'][0]
        self.sig_01_c = info['sig_0_c'][1]
        self.sig_10_c = info['sig_1_c'][0]
        self.sig_11_c = info['sig_1_c'][1]
        self.sig_0 = [[self.sig_00, 0],
                      [0, self.sig_01]]
        self.sig_1 = [[self.sig_10, 0],
```

```
                [0, self.sig_11]]
        self.sig_0_c = [[self.sig_10_c, 0],
                [0, self.sig_11_c]]
        self.sig_1_c = [[self.sig_10_c, 0],
                [0, self.sig_11_c]]

        # NN
        self.num_epochs = info['num_epochs']
        self.batch_size = info['batch_size']

    def reshape(self, x00, x01, x10, x11, y0, y1):
        x00 = x00.reshape(x00.shape[0], 1)
        x01 = x01.reshape(x01.shape[0], 1)
        x10 = x10.reshape(x10.shape[0], 1)
        x11 = x11.reshape(x11.shape[0], 1)
        y0 = y0.reshape(y0.shape[0], 1)
        y1 = y1.reshape(y1.shape[0], 1)
        return x00, x01, x10, x11, y0, y1

    def generate_training_data(self):
        x00, x01 = np.random.multivariate_normal(self.mu_0,
            self.sig_0, self.num_zero).T
        x10, x11 = np.random.multivariate_normal(self.mu_1,
            self.sig_1, self.num_one).T
        y0 = np.zeros((self.num_zero, 1))
        y1 = np.ones((self.num_one, 1))
        x00, x01, x10, x11, y0, y1 = self.reshape(x00, x01, x10, x11,
            y0, y1)
        self.zero_train_data = np.concatenate((x00, x01, y0), axis=1)
        self.one_train_data = np.concatenate((x10, x11, y1), axis=1)
        self.all_train_data = np.concatenate((self.zero_train_data,
            self.one_train_data), axis=0)
        np.random.shuffle(self.all_train_data)

    def generate_test_data_a(self):
        x00, x01 = np.random.multivariate_normal(self.mu_0,
            self.sig_0, self.num_zero_a).T
        x10, x11 = np.random.multivariate_normal(self.mu_1,
            self.sig_1, self.num_one_a).T
        y0 = np.zeros((self.num_zero_a, 1))
        y1 = np.ones((self.num_one_a, 1))
        x00, x01, x10, x11, y0, y1 = self.reshape(x00, x01, x10, x11,
            y0, y1)
        self.zero_test_data = np.concatenate((x00, x01, y0), axis=1)
        self.one_test_data = np.concatenate((x10, x11, y1), axis=1)
        self.all_test_data = np.concatenate((self.zero_test_data,
            self.one_test_data), axis=0)
        np.random.shuffle(self.all_test_data)
```

```
def generate_test_data_b(self):
    x00, x01 = np.random.multivariate_normal(self.mu_0,
        self.sig_0, self.num_zero_b).T
    x10, x11 = np.random.multivariate_normal(self.mu_1,
        self.sig_1, self.num_one_b).T
    y0 = np.zeros((self.num_zero_b, 1))
    y1 = np.ones((self.num_one_b, 1))
    x00, x01, x10, x11, y0, y1 = self.reshape(x00, x01, x10, x11,
        y0, y1)
    self.zero_test_data = np.concatenate((x00, x01, y0), axis=1)
    self.one_test_data = np.concatenate((x10, x11, y1), axis=1)
    self.all_test_data = np.concatenate((self.zero_test_data,
        self.one_test_data), axis=0)
    np.random.shuffle(self.all_test_data)

def generate_test_data_c(self):
    x00, x01 = np.random.multivariate_normal(self.mu_0,
        self.sig_0_c, self.num_zero_a).T
    x10, x11 = np.random.multivariate_normal(self.mu_1,
        self.sig_1_c, self.num_one_a).T
    y0 = np.zeros((self.num_zero_a, 1))
    y1 = np.ones((self.num_one_a, 1))
    x00, x01, x10, x11, y0, y1 = self.reshape(x00, x01, x10, x11,
        y0, y1)
    self.zero_test_data = np.concatenate((x00, x01, y0), axis=1)
    self.one_test_data = np.concatenate((x10, x11, y1), axis=1)
    self.all_test_data = np.concatenate((self.zero_test_data,
        self.one_test_data), axis=0)
    np.random.shuffle(self.all_test_data)

def logistic_regression(self):
    x_train = self.all_train_data[:, [0, 1]]
    y_train = self.all_train_data[:, 2]
    x_test = self.all_test_data[:, [0, 1]]
    y_true = self.all_test_data[:, 2]
    model = lm.LogisticRegression()
    model_self = model.fit(x_train, y_train)
    y_est = model.predict(x_test)
    auc = roc_auc_score(y_true, y_est)
    return auc

def multivariate_normal_pdf(self, x, num):
    mu_0 = np.array(self.mu_0).reshape(2, 1)
    mu_1 = np.array(self.mu_1).reshape(2, 1)
    sig_0 = np.array(self.sig_0).reshape(2, 2)
    sig_1 = np.array(self.sig_1).reshape(2, 2)
    if num == 0:
```

```
            mult = 1/(np.sqrt(np.linalg.det(sig_0)*(2*np.pi)**2))
            exp = np.dot((np.dot(np.transpose(x - mu_0),
                np.linalg.inv(sig_0))), x - mu_0)
            return mult*np.exp(-0.5*exp)
        elif num == 1:
            mult = 1/(np.sqrt(np.linalg.det(sig_1)*(2*np.pi)**2))
            exp = np.dot((np.dot(np.transpose(x - mu_1),
                np.linalg.inv(sig_1))), x - mu_1)
            return mult*np.exp(-0.5*exp)

    def naive_bayes(self):
        Ps = np.empty((self.all_test_data.shape[0], 2))
        for i in range(self.all_test_data.shape[0]):
            x = self.all_test_data[i, [0, 1]].reshape(2, 1)
            y_true = self.all_test_data[i, 2]
            WasOne = False
            WasZero = False
            if y_true == 0:
                # Y = 0
                Ps[i, 0] = self.p_Y0*self.multivariate_normal_pdf(x, 0)
                WasZero = True
            else:
                # Y = 1
                Ps[i, 1] = self.p_Y1*self.multivariate_normal_pdf(x, 1)
                WasOne = True
            # Replace other entry with other class calculation
            if WasZero:
                Ps[i, 1] = self.p_Y1*self.multivariate_normal_pdf(x, 1)
            elif WasOne:
                Ps[i, 0] = self.p_Y0*self.multivariate_normal_pdf(x, 0)
        y_est = np.argmax(Ps, axis=1)
        y_true = self.all_test_data[:, 2]
        auc = roc_auc_score(y_true, y_est)
        return auc

    def NN(self):
        model = Sequential()
        model.add(Dense(12, input_shape=(2,),
            activation=tf.keras.activations.tanh))
        model.add(Dense(4, activation=tf.keras.activations.tanh))
        model.add(Dense(1, activation=tf.keras.activations.tanh))
        model.compile(loss='binary_crossentropy', optimizer='adam',
            metrics=['accuracy', tf.keras.metrics.AUC(),
                        tf.keras.metrics.FalseNegatives(),
                        tf.keras.metrics.FalsePositives(),
                        tf.keras.metrics.TrueNegatives(),
                        tf.keras.metrics.TruePositives()])
        model.fit(self.all_train_data[:, [0, 1]],
```

```
            self.all_train_data[:, 2], epochs=self.num_epochs,
            batch_size=self.batch_size)
        _, class_accuracy, auc, FN, FP, TN, TP =
            model.evaluate(self.all_test_data[:, [0, 1]],
            self.all_test_data[:, 2])
        # print(f"AUC Neural Net: {round(auc, 4)}")
        return auc

    def render(self):
        plt.scatter(self.zero_train_data[:, 0],
            self.zero_train_data[:, 1], s=1)
        plt.scatter(self.one_train_data[:, 0], self.one_train_data[:,
            1], s=1)
        plt.scatter(self.zero_test_data[:, 0], self.zero_test_data[:,
            1], s=3)
        plt.scatter(self.one_test_data[:, 0], self.one_test_data[:,
            1], s=3)
        plt.legend(["Train: Y = 0", "Train: Y = 1", "Test: Y = 0",
            "Test: Y = 1"])
        plt.show()

    def run(self, part):
        self.generate_training_data()
        if part == "a":
            self.generate_test_data_a()
        elif part == "b":
            self.generate_test_data_b()
        elif part == "c":
            self.generate_test_data_c()
        auc_nb = self.naive_bayes()
        auc_lr = self.logistic_regression()
        auc_nn = self.NN()
        print(f"\n==================")
        print(f"===== Part {part} =====")
        print("==================")
        print(f"AUC Naive Bayes: {round(auc_nb, 4)}")
        print(f"AUC Logistic Regression: {round(auc_lr, 4)}")
        print(f"AUC Neural Net: {round(auc_nn, 4)}")
        self.render()

params = {"p_Y0": 0.5,
        "p_Y0_b": 0.1,
        "d": 10000,
        "mu_0": 1,
        "mu_1": 4,
        "sig_0": [10, 3],
        "sig_1": [3, 4],
        "sig_0_c": [3, 15],
```

```
        "sig_1_c": [8, 6],
        "num_epochs": 200,
        "batch_size": 32}
q_4 = q4(params)
q_4.run("b")
```

To briefly describe the above code, "d" number of training data points are drawn from two 2D normal distributions with the params mu_0, mu_1, sig_0, sig_1 seen in the "params" library. The number of each label is decided by "p_Y0" (and subsequently 1-p_Y0). Then, depending on the "part" input, test data is created via 10-fold cross validation. For part a, the method generate_test_data_a() draws from the same distribution as the training data. For part b, test data is drawn from the same distributions, although with a different class prior distribution ($\bar{P}(Y = 0) \neq P(Y = 0)$). In part c, the test data is drawn from the same class priors, but different posterior distributions. More specifically, the 2x2 covariance matrices defined in the "params" library are slightly shifted to represent bias in the test data. The script then creates 3 models: A Naive Bayes classifier, logistic regression model, and a 3-layer Neural network with 2 hidden layers of dimension 12 and 4, respectively, and tanh activation function. Each model is trained on the training data and then tested on the respective part's test data. To run the script for each part, simply change the letter in the last line, "q_4.run("a")", between "a", "b", and "c".

Visuals of the training and test data can be seen below for each part:

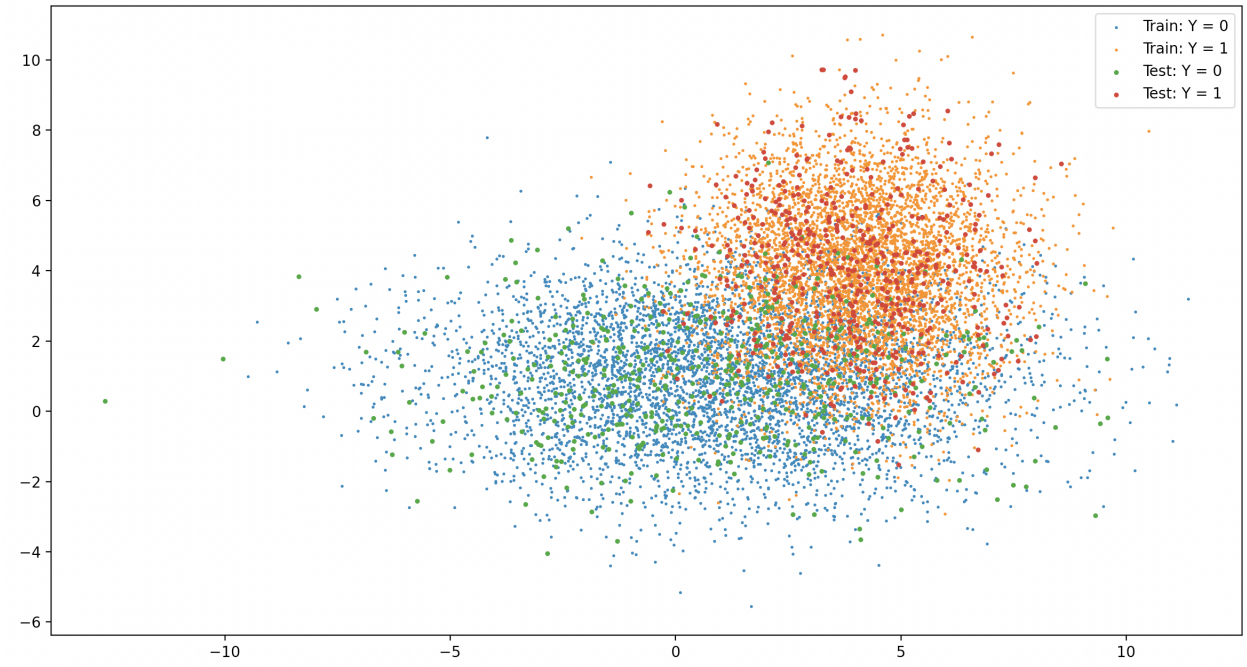

Figure 2: Q4 Part A - No Bias

The test data Bias can be well seen in these figures. In part a, it is clearly seen that the test data (larger green and red dots) are well representative of the training data (smaller blue and orange dots).

In part b, the class prior bias is well observed as there are clearly many more red dots (P(Y=1)) representing the first distribution, than green dots representing the second distribution.

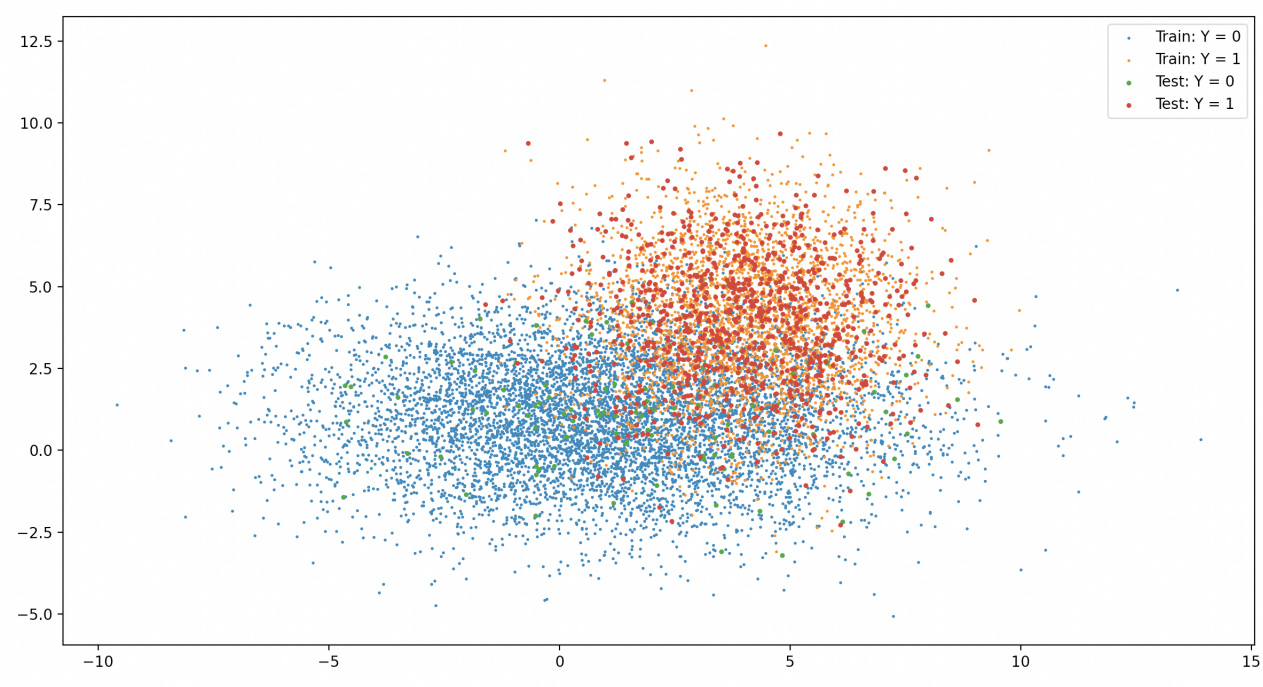

Figure 3: Q4 Part B - Class Prior Bias

Finally, in part c, the shifted covariance is also well observed. The red and green test data points are clearly spread out much further apart than the bounds of the training distributions.

The script prints out the resulting ROC AUC values for each model after the respective part. The outputs I received for each part are seen below:

```
==================
===== Part a =====
==================
AUC Naive Bayes: 0.839
AUC Logistic Regression: 0.839
AUC Neural Net: 0.9308

==================
===== Part b =====
==================
AUC Naive Bayes: 0.8022
AUC Logistic Regression: 0.8294
AUC Neural Net: 0.9133

==================
===== Part c =====
==================
AUC Naive Bayes: 0.725
AUC Logistic Regression: 0.737
AUC Neural Net: 0.8281
```

Models were ran multiple times to ensure consistent results.

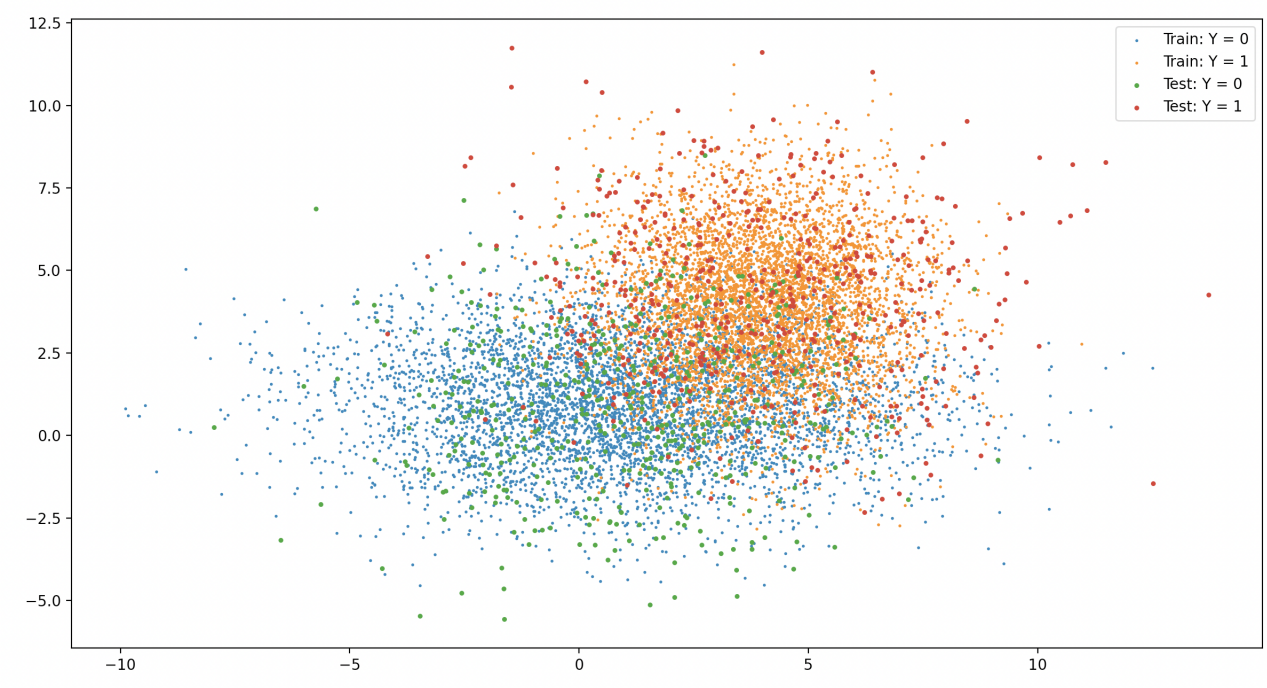

Figure 4: Q4 Part C - Class Posterior Bias

There are a few initial observations when analyzing results. First, the ROC AUC calculation for the neural network is highest in all 3 situations, displaying the ability for a deep network to learn more complex functions with higher success rate than the Naive Bayes and Logistic Regression models. In comparing part a (baseline) and part b, it can be seen that all models dropped slightly in performance due to the bias in the b test data. Again, the bias in the part b test data was due to drawing from different class prior distributions. This is expected to drop the overall results due to this change between the training data and test data, although it is important to note that it did not drop accuracies by that much. This is expected since the models are not affected as negatively as they would be with other types of bias. During training, I did notice that I could greatly exaggerate the results only if I greatly exaggerated the difference in class priors, for example P(Y=0) from 0.1 to 0.9. It is also important to note that the AUC calculation dropped the most for the Naive Bayes model. This is because the class priors carry the heaviest weight in effecting results in this model type compared to the others, since it is directly involved in the calculation.

Comparing part c results to the baseline results in part a, it is clear that performance worsened more so than part b. This is due to the covariance bias having a much greater effect on the resulting models than the class prior bias. With covariance class posterior bias, the test data points have a much higher chance of misrepresenting the training data compared to simply changing the class prior. When tuning hyperparameters, I made sure to not change it too much as to get very poor results, but made sure to change the bias enough to notice the difference. I also ensured that there was enough overlap between the two distributions in order for accurate results to be obtained.