Assignment 4 - Neural Networks

Daniela Jiménez Lara

Netid: dj216

Names of students you worked with on this assignment: Bárbara Flores

Note: this assignment falls under collaboration Mode 2: Individual Assignment – Collaboration Permitted. Please refer to the syllabus for additional information.

Instructions for all assignments can be found here.

Total points in the assignment add up to 90; an additional 10 points are allocated to presentation quality.

Learning objectives

Through completing this assignment you will be able to...

1. Identify key hyperparameters in neural networks and how they can impact model training and fit
2. Build, tune the parameters of, and apply feed-forward neural networks to data
3. Implement and explain each and every part of a standard fully-connected neural network and its operation including feed-forward propagation, backpropagation, and gradient descent.
4. Apply a standard neural network implementation and search the hyperparameter space to select optimized values.
5. Develop a detailed understanding of the math and practical implementation considerations of neural networks, one of the most widely used machine learning tools, so that it can be leveraged for learning about other neural networks of different model architectures.

1

[60 points] Exploring and optimizing neural network hyperparameters

Neural networks have become ubiquitous in the machine learning community, demonstrating exceptional performance over a wide range of supervised learning tasks. The benefits of these techniques come at a price of increased computational complexity and model designs with increased numbers of hyperparameters that need to be correctly set to make these techniques work. It is common that poor hyperparameter choices in neural networks result in significant decreases in model generalization performance. The goal of this exercise is to better understand some of the key hyperparameters you will encounter in practice using neural networks so that you can be better prepared to tune your model for a given application. Through this exercise, you will explore two common approaches to hyperparameter tuning a manual approach where we greedily select the best individual hyperparameter (often people will pick potentially sensible options, try them, and hope it works) as well as a random search of the hyperparameter space which as been shown to be an efficient way to achieve good hyperparameter values.

To explore this, we'll be using the example data created below throughout this exercise and the various training, validation, test splits. We will select each set of hyperparameters for our greedy/manual approach and the random search using a training/validation split, then retrain on the combined training and validation data before finally evaluating our generalization performance for both our final models on the test data.

In [ ]:
```
# Optional for clear plotting on Macs
# %config InlineBackend.figure_format='retina'

# Some of the network training leads to warnings. When we know and are OK with
#  what's causing the warning and simply don't want to see it, we can use the
#  following code. Run this block
#  to disable warnings
```

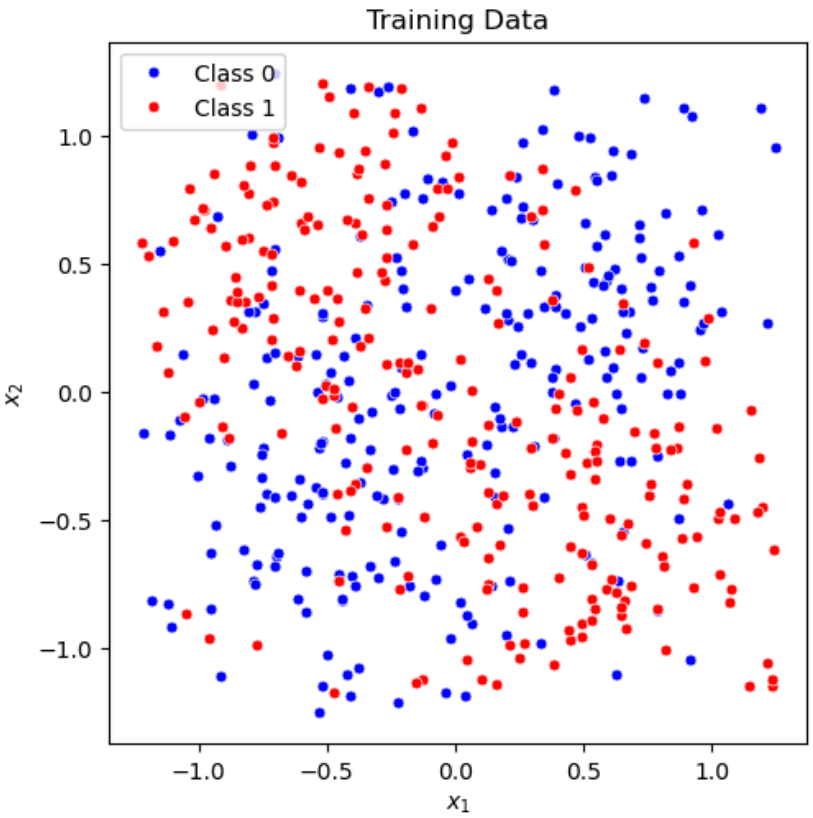
```
import sys
import os
import warnings

if not sys.warnoptions:
    warnings.simplefilter("ignore")
    os.environ["PYTHONWARNINGS"] = "ignore"
```

In [ ]:
```
import numpy as np
from sklearn.model_selection import PredefinedSplit


# ----------------------------------------------------------------------------
# Create the data
# ----------------------------------------------------------------------------
# Data generation function to create a checkerboard-patterned dataset
def make_data_normal_checkerboard(n, noise=0):
    n_samples = int(n / 4)
    shift = 0.5
    c1a = np.random.randn(n_samples, 2) * noise + [-shift, shift]
    c1b = np.random.randn(n_samples, 2) * noise + [shift, -shift]
    c0a = np.random.randn(n_samples, 2) * noise + [shift, shift]
    c0b = np.random.randn(n_samples, 2) * noise + [-shift, -shift]
    X = np.concatenate((c1a, c1b, c0a, c0b), axis=0)
    y = np.concatenate((np.ones(2 * n_samples), np.zeros(2 * n_samples)))

    # Set a cutoff to the data and fill in with random uniform data:
    cutoff = 1.25
    indices_to_replace = np.abs(X) > cutoff
    for index, value in enumerate(indices_to_replace.ravel()):
        if value:
            X.flat[index] = np.random.rand() * 2.5 - 1.25
    return (X, y)


# Training datasets
np.random.seed(42)
noise = 0.45
X_train, y_train = make_data_normal_checkerboard(500, noise=noise)

# Validation and test data
X_val, y_val = make_data_normal_checkerboard(500, noise=noise)
X_test, y_test = make_data_normal_checkerboard(500, noise=noise)

# For RandomSeachCV, we will need to combine training and validation sets then
#  specify which portion is training and which is validation
# Also, for the final performance evaluation, train on all of the training AND validation data
X_train_plus_val = np.concatenate((X_train, X_val), axis=0)
y_train_plus_val = np.concatenate((y_train, y_val), axis=0)

# Create a predefined train/test split for RandomSearchCV (to be used later)
validation_fold = np.concatenate((-1 * np.ones(len(y_train)), np.zeros(len(y_val))))
train_val_split = PredefinedSplit(validation_fold)
```

To help get you started we should always begin by visualizing our training data, here's some code that does that:

In [ ]:
```
import matplotlib.pyplot as plt


# Code to plot the sample data
def plot_data(ax, X, y, title, limits):
    # Select the colors to use in the plots
    color0 = "blue"  # Dark grey
    color1 = "red"  # Green
    color_boundary = "#858585"

    # Separate samples by class
    samples0 = X[y == 0]
    samples1 = X[y == 1]

    ax.plot(
        samples0[:, 0],
```

```
        samples0[:, 1],
        marker="o",
        markersize=5,
        linestyle="None",
        color=color0,
        markeredgecolor="w",
        markeredgewidth=0.5,
        label="Class 0",
    )
    ax.plot(
        samples1[:, 0],
        samples1[:, 1],
        marker="o",
        markersize=5,
        linestyle="None",
        color=color1,
        markeredgecolor="w",
        markeredgewidth=0.5,
        label="Class 1",
    )
    ax.set_title(title)
    ax.set_xlabel("$x_1$")
    ax.set_ylabel("$x_2$")
    ax.legend(loc="upper left")
    ax.set_aspect("equal")


fig, ax = plt.subplots(constrained_layout=True, figsize=(5, 5))
limits = [-1.25, 1.25, -1.25, 1.25]
plot_data(ax, X_train, y_train, "Training Data", limits)
```



The hyperparameters we want to explore control the architecture of our model and how our model is fit to our data. These hyperparameters include the (a) learning rate, (b) batch size, and the (c) regularization coefficient, as well as the (d) model architecture hyperparameters (the number of layers and the number of nodes per layer). We'll explore each of these and determine an optimized configuration of the network for this problem through this exercise. For all of the settings we'll explore and just, we'll assume the following default hyperparameters for the model (we'll use scikit learn's `MLPClassifier` as our neural network model):

- `learning_rate_init` = 0.03
- `hidden_layer_sizes` = (30,30) (two hidden layers, each with 30 nodes)

- `alpha` = 0 (regularization penalty)
- `solver` = 'sgd' (stochastic gradient descent optimizer)
- `tol` = 1e-5 (this sets the convergence tolerance)
- `early_stopping` = False (this prevents early stopping)
- `activation` = 'relu' (rectified linear unit)
- `n_iter_no_change` = 1000 (this prevents early stopping)
- `batch_size` = 50 (size of the minibatch for stochastic gradient descent)
- `max_iter` = 500 (maximum number of epochs, which is how many times each data point will be used, not the number of gradient steps)

This default setting is our initial guess of what good values may be. Notice there are many model hyperparameters in this list: any of these could potentially be options to search over. We constrain the search to those hyperparameters that are known to have a significant impact on model performance.

(a) Visualize the impact of different hyperparameter choices on classifier decision boundaries. Visualize the impact of different hyperparameter settings. Starting with the default settings above make the following changes (only change one hyperparameter at a time). For each hyperparameter value, plot the decision boundary on the training data (you will need to train the model once for each parameter value):

1. Vary the architecture (`hidden_layer_sizes`) by changing the number of nodes per layer while keeping the number of layers constant at 2: (2,2), (5,5), (30,30). Here (X,X) means a 2-layer network with X nodes in each layer.
2. Vary the learning rate: 0.0001, 0.01, 1
3. Vary the regularization: 0, 1, 10
4. Vary the batch size: 5, 50, 500

This should produce 12 plots, altogether. For easier comparison, please plot nodes & layers combinations, learning rates, regularization strengths, and batch sizes in four separate rows (with three columns each representing a different value for each of those hyperparameters).

As you're exploring these settings, visit this website, the Neural Network Playground, which will give you the chance to interactively explore the impact of each of these parameters on a similar dataset to the one we use in this exercise. The tool also allows you to adjust the learning rate, batch size, regularization coefficient, and the architecture and to see the resulting decision boundary and learning curves. You can also visualize the model's hidden node output and its weights, and it allows you to add in transformed features as well. Experiment by adding or removing hidden layers and neurons per layer and vary the hyperparameters.

Answer

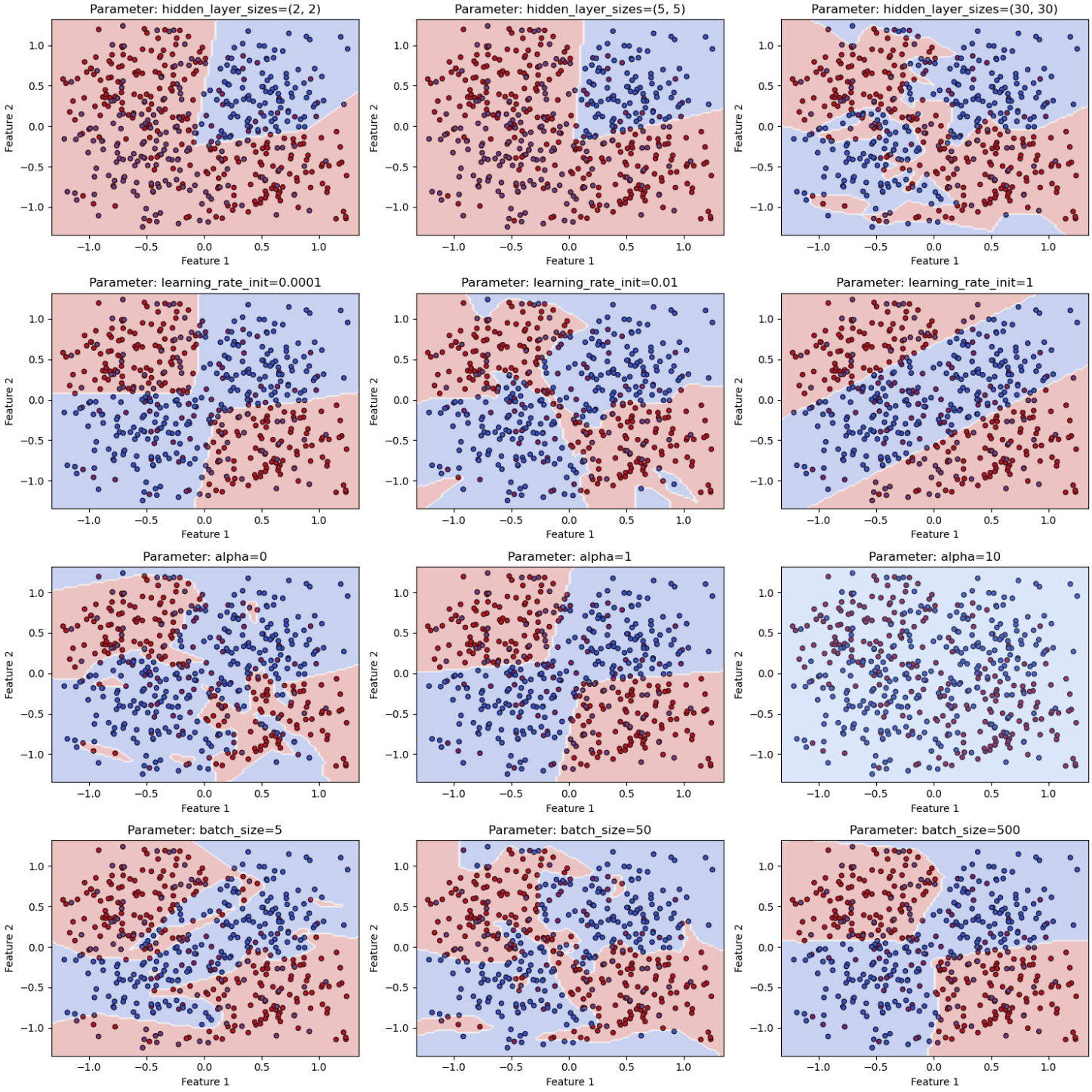
a)

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.neural_network import MLPClassifier


def plot_decision_boundary(X, y, model, title, ax=None):
    if ax is None:
        ax = plt.gca()

    ax.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm, s=20, edgecolors="k")

    x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
    y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.03), np.arange(y_min, y_max, 0.03))

    xx_flat = xx.ravel()
    yy_flat = yy.ravel()
    mesh_data = np.column_stack((xx_flat, yy_flat))

    Z = model.predict(mesh_data)

    Z = Z.reshape(xx.shape)
    ax.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.coolwarm, antialiased=True)
```

```
    ax.set_title(title)
    ax.set_xlabel("Feature 1")
    ax.set_ylabel("Feature 2")
```

In [ ]:

```
default_params = {
    "learning_rate_init": 0.03,
    "hidden_layer_sizes": (30, 30),
    "alpha": 0,
    "solver": "sgd",
    "tol": 1e-5,
    "early_stopping": False,
    "activation": "relu",
    "n_iter_no_change": 1000,
    "batch_size": 50,
    "max_iter": 5000,
}

architectures = [(2, 2), (5, 5), (30, 30)]
learning_rates = [0.0001, 0.01, 1]
alphas = [0, 1, 10]
batches = [5, 50, 500]

fig, axs = plt.subplots(4, 3, figsize=(15, 15))

for i, (param_name, values) in enumerate(
    zip(
        ["hidden_layer_sizes", "learning_rate_init", "alpha", "batch_size"],
        [architectures, learning_rates, alphas, batches],
    )
):
    for j, value in enumerate(values):
        updated_params = default_params.copy()
        updated_params[param_name] = value

        clf = MLPClassifier(**updated_params)
        clf.fit(X_train, y_train)

        plot_decision_boundary(
            X_train, y_train, clf, f"Parameter: {param_name}={value}", ax=axs[i, j]
        )

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



(b) Manual (greedy) hyperparameter tuning I: manually optimize hyperparameters that govern the learning process, one hyperparameter at a time. Now with some insight into which settings may work better than others, let's more fully explore the performance of these different settings in the context of our validation dataset through a manual optimization process. Holding all else constant (with the default settings mentioned above), vary each of the following parameters as specified below. Train your algorithm on the training data, and evaluate the performance of your trained algorithm on the validation dataset. Here, overall accuracy is a reasonable performance metric since the classes are balanced and we don't weight one type of error as more important than the other; therefore, use the `score` method of the `MLPClassifier` for this. Create plots of accuracy vs each parameter you vary (this will result in three plots).

1. Vary learning rate logarithmically from 10^{-5} to 10^{0} with 20 steps
2. Vary the regularization parameter logarithmically from 10^{-8} to 10^{2} with 20 steps
3. Vary the batch size over the following values: $[1, 3, 5, 10, 20, 50, 100, 250, 500]$

For each of these cases:

- Based on the results, report your optimal choices for each of these hyperparameters and why you selected them.

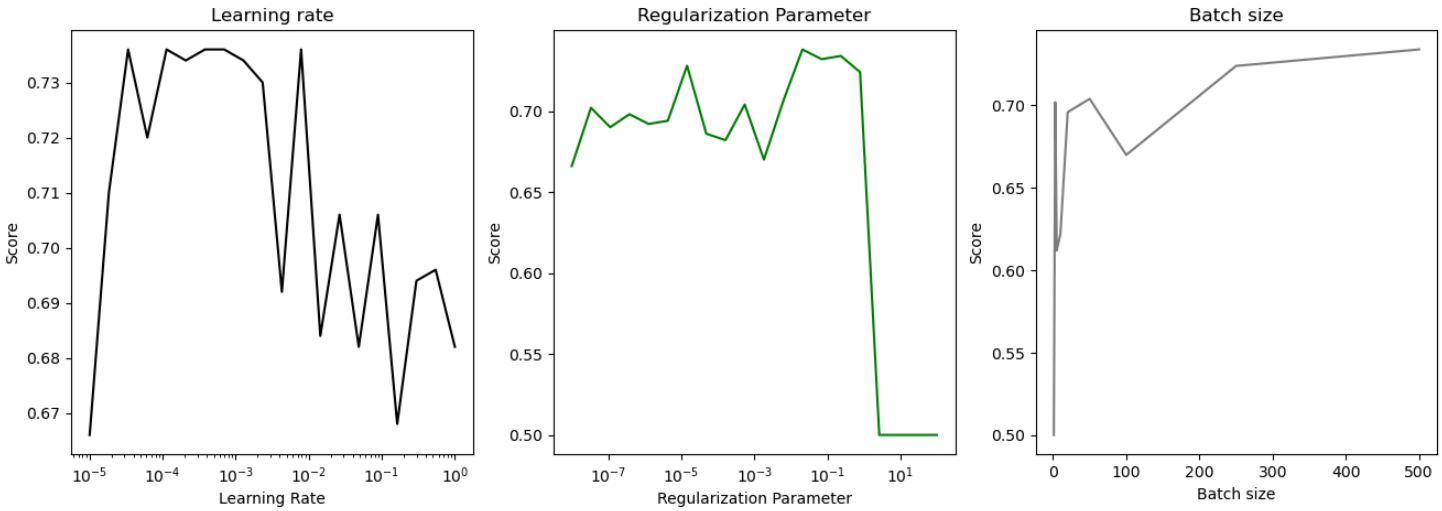
- Since neural networks can be sensitive to initialization values, you may notice these plots may be a bit noisy. Consider this when selecting the optimal values of the hyperparameters. If the noise seems significant, run the fit and score procedure multiple times (without fixing a random seed) and report the average. Rerunning the algorithm will change the initialization and therefore the output (assuming you do not set a random seed for that algorithm).
- Use the chosen hyperparameter values as the new default settings for section (c) and (d).

In [ ]:
```
lr_list = np.logspace(-5, 0, 20)
lr_score = []

# learning rate loop
o_model = MLPClassifier(
    learning_rate_init=0.03,
    hidden_layer_sizes=(30, 30),
    alpha=0,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="relu",
    n_iter_no_change=1000,
    batch_size=50,
    max_iter=5000,
)


for lr in lr_list:
    o_model.set_params(learning_rate_init=lr)
    # print(lr)
    o_model.fit(X_train, y_train)
    lr_s = o_model.score(X_val, y_val)
    # print(lr_s)
    lr_score.append(lr_s)

# batch size loop
o2_model = MLPClassifier(
    learning_rate_init=0.03,
    hidden_layer_sizes=(30, 30),
    alpha=0,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="relu",
    n_iter_no_change=1000,
    batch_size=50,
    max_iter=5000,
)


bz_list = [1, 3, 5, 10, 20, 50, 100, 250, 500]

bz_score = []

for bz in bz_list:
    o2_model.set_params(batch_size=bz)
    o2_model.fit(X_train, y_train)
    bz_s = o2_model.score(X_val, y_val)
    bz_score.append(bz_s)

# regularization loop
o1_model = MLPClassifier(
    learning_rate_init=0.03,
    hidden_layer_sizes=(30, 30),
    alpha=0,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="relu",
    n_iter_no_change=1000,
    batch_size=50,
    max_iter=5000,
)
```

```
rp_list = np.logspace(-8, 2, 20)

rp_score = []

for rp in rp_list:
    o1_model.set_params(alpha=rp)
    o1_model.fit(X_train, y_train)
    rp_s = o1_model.score(X_val, y_val)
    rp_score.append(rp_s)
```

In [ ]:
```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 3, figsize=(16, 5))

# learning rate
axs[0].plot(lr_list, lr_score, color="black")
axs[0].set_xscale("log")
axs[0].set_xlabel("Learning Rate")
axs[0].set_ylabel("Score")
axs[0].set_title("Learning rate")

# Regularization
axs[1].plot(rp_list, rp_score, color="green")
axs[1].set_xscale("log")
axs[1].set_xlabel("Regularization Parameter")
axs[1].set_ylabel("Score")
axs[1].set_title("Regularization Parameter")

# Batch size
axs[2].plot(bz_list, bz_score, color="grey")
# axs[2].set_xscale("log")
axs[2].set_xlabel("Batch size")
axs[2].set_ylabel("Score")
axs[2].set_title("Batch size")
```

Out[ ]:
```
Text(0.5, 1.0, 'Batch size')
```



In [ ]:
```
op_lr = lr_list[np.argmax(lr_score)]
op_bz = bz_list[np.argmax(bz_score)]
op_rp = rp_list[np.argmax(rp_score)]

print(f"The optimal value for learning rate is:{op_lr}")
print(f"The optimal value for batch size is: {op_bz}")
print(f"The optimal value for regularization parameter is: {op_rp}")
```

```
The optimal value for learning rate is:3.359818286283781e-05
The optimal value for batch size is: 500
The optimal value for regularization parameter is: 0.0206913808111479
```

> Repeating the fit and score procedure since the noise seems significant for all parameters:

In [ ]:
```
# lr loop 10 times
o_model = MLPClassifier(
    learning_rate_init=0.03,
    hidden_layer_sizes=(30, 30),
    alpha=0,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="relu",
    n_iter_no_change=1000,
    batch_size=50,
    max_iter=5000,
)

lr_list = np.logspace(-5, 0, 20)
list_av_lr_scores = []


for lr in lr_list:
    lr_score = []
    for _ in range(10):
        o_model.set_params(learning_rate_init=lr)
        o_model.fit(X_train, y_train)
        lr_s = o_model.score(X_val, y_val)
        lr_score.append(lr_s)
    av_lr_score = np.mean(lr_score)
    list_av_lr_scores.append(av_lr_score)
```

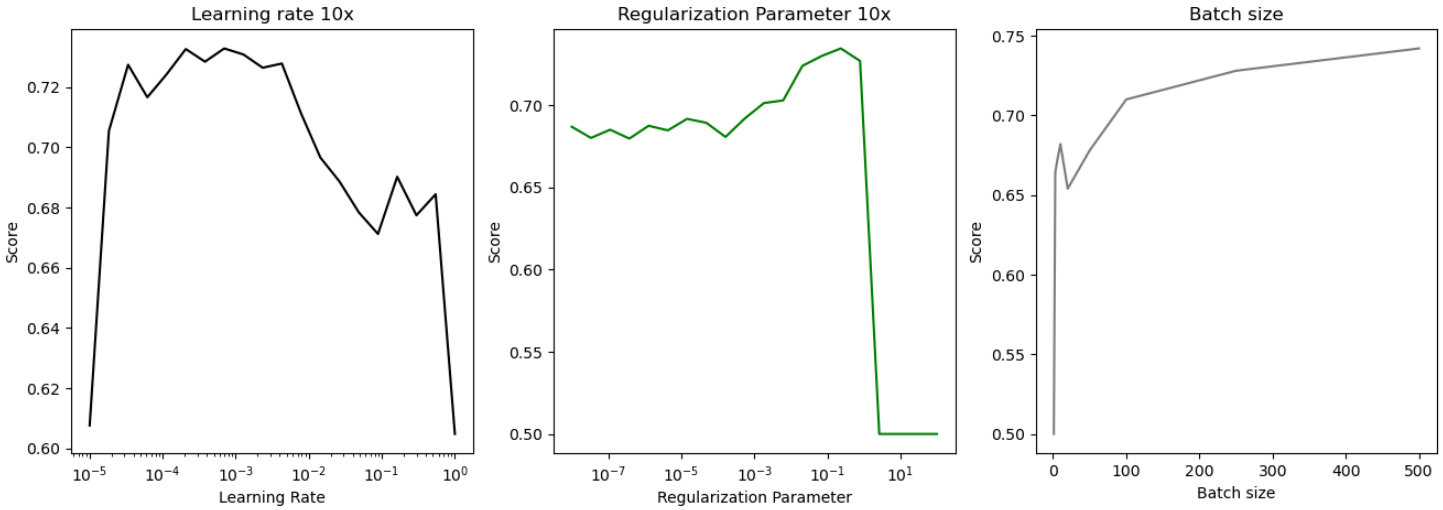
In [ ]:
```
# regularization loop 10 times
o1_model = MLPClassifier(
    learning_rate_init=0.03,
    hidden_layer_sizes=(30, 30),
    alpha=0,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="relu",
    n_iter_no_change=1000,
    batch_size=50,
    max_iter=5000,
)

rp_list = np.logspace(-8, 2, 20)
list_av_rp_scores = []

for rp in rp_list:
    rp_score = []
    for _ in range(10):
        o1_model.set_params(alpha=rp)
        o1_model.fit(X_train, y_train)
        rp_s = o1_model.score(X_val, y_val)
        rp_score.append(rp_s)
    av_rp_score = np.mean(rp_score)
    list_av_rp_scores.append(av_rp_score)
```

In [ ]:
```
# batch size loop 10 times

o2_model = MLPClassifier(
    learning_rate_init=0.03,
    hidden_layer_sizes=(30, 30),
    alpha=0,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="relu",
    n_iter_no_change=1000,
    batch_size=50,
    max_iter=5000,
)

list_av_bz_scores = []
```

```
bz_list = [1, 3, 5, 10, 20, 50, 100, 250, 500]
bz_score = []
for bz in bz_list:
    for _ in range(10):
        bz_score = []
        o2_model.set_params(batch_size=bz)
        o2_model.fit(X_train, y_train)
        bz_s = o2_model.score(X_val, y_val)
        bz_score.append(bz_s)
    av_bz_score = np.mean(bz_score)
    list_av_bz_scores.append(av_bz_score)
```

In [ ]:
```
fig, axs = plt.subplots(1, 3, figsize=(16, 5))

# learning rate
axs[0].plot(lr_list, list_av_lr_scores, color="black")
axs[0].set_xscale("log")
axs[0].set_xlabel("Learning Rate")
axs[0].set_ylabel("Score")
axs[0].set_title("Learning rate 10x")

# Regularization
axs[1].plot(rp_list, list_av_rp_scores, color="green")
axs[1].set_xscale("log")
axs[1].set_xlabel("Regularization Parameter")
axs[1].set_ylabel("Score")
axs[1].set_title("Regularization Parameter 10x")

# Batch size
axs[2].plot(bz_list, list_av_bz_scores, color="grey")
# axs[2].set_xscale("log")
axs[2].set_xlabel("Batch size")
axs[2].set_ylabel("Score")
axs[2].set_title("Batch size")
```

Out[ ]:
```
Text(0.5, 1.0, 'Batch size')
```



> The chosen hyperparameter values are the following:

In [ ]:
```
op_lr = lr_list[np.argmax(list_av_lr_scores)]
op_rp = rp_list[np.argmax(list_av_rp_scores)]
op_bz = bz_list[np.argmax(list_av_bz_scores)]

print(
    f"Since the learning rate graph was noisy , the fit-score procedure was repeared ten times,\n the new optim
)
print(
    f"Since the regularization parameter graph was noisy, the fit-score procedure was repeared ten times,\n the
)
print(
```

```
    f"Since the batch size graph was noisy, the fit-score procedure was repeated,\n the optimal value for batch
)
```

```
Since the learning rate graph was noisy , the fit-score procedure was repeared ten times,
 the new optimal value for learning rate is:0.000695
Since the regularization parameter graph was noisy, the fit-score procedure was repeared ten times,
 the new optimal value for batch size is: 0.233572
Since the batch size graph was noisy, the fit-score procedure was repeated,
 the optimal value for batch size is: 500
```

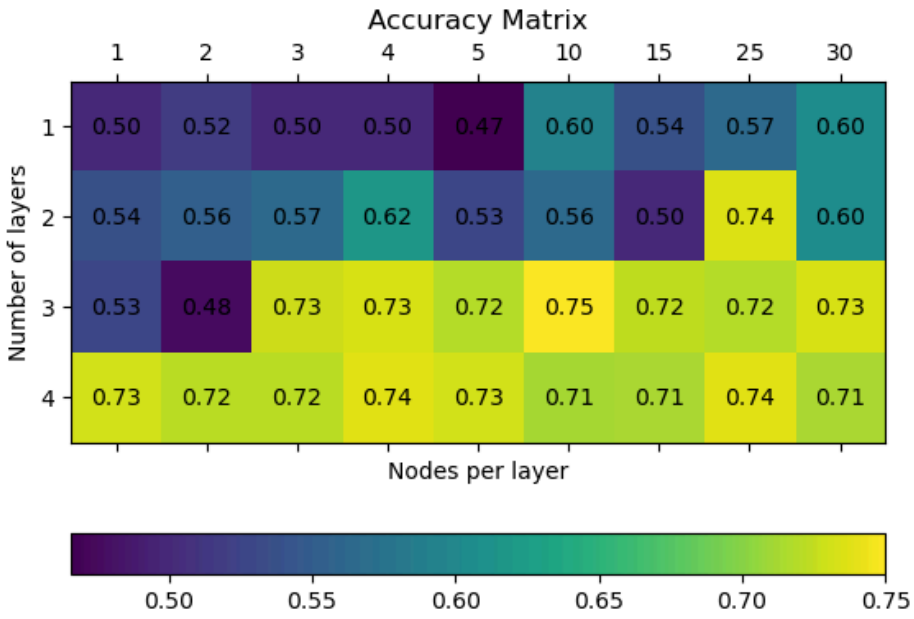
(c) Manual (greedy) hyperparameter tuning II: manually optimize hyperparameters that impact the model architecture. Next, we want to explore the impact of the model architecture on performance and optimize its selection. This means varying two parameters at a time instead of one as above. To do this, evaluate the validation accuracy resulting from training the model using each pair of possible numbers of nodes per layer and number of layers from the lists below. We will assume that for any given configuration the number of nodes in each layer is the same (e.g. (2,2,2), which would be a 3-layer network with 2 hidden node in each layer and (25,25) are valid, but (2,5,3) is not because the number of hidden nodes varies in each layer). Use the manually optimized values for learning rate, regularization, and batch size selected from section (b).

- Number of nodes per layer: $[1, 2, 3, 4, 5, 10, 15, 25, 30]$
- Number of layers = $[1, 2, 3, 4]$ Report the accuracy of your model on the validation data. For plotting these results, use heatmaps to plot the data in two dimensions. To make the heatmaps, you can use [this code for creating heatmaps] https://matplotlib.org/stable/gallery/images_contours_and_fields/image_annotated_heatmap.html). Be sure to include the numerical values of accuracy in each grid square as shown in the linked example and label your x, y, and color axes as always. For these numerical values, round them to **2 decimal places** (due to some randomness in the training process, any further precision is not typically meaningful).
- When you select your optimized parameters, be sure to keep in mind that these values may be sensitive to the data and may offer the potential to have high variance for larger models. Therefore, select the model with the highest accuracy but lowest number of total model weights (all else equal, the simpler model is preferred).
- What do the results show? Which parameters did you select and why?

In [ ]:
```
from sklearn.metrics import accuracy_score

nodes_l = [1, 2, 3, 4, 5, 10, 15, 25, 30]
layers_l = [1, 2, 3, 4]

ooo_model = MLPClassifier(
    learning_rate_init=0.03,
    hidden_layer_sizes=(30, 30),
    alpha=0,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="relu",
    n_iter_no_change=1000,
    batch_size=50,
    max_iter=5000,
)

accuracy_scores_layers = []


for nodes in nodes_l:
    for layer in layers_l:
        hidden_layer_s = tuple([nodes] * (layer))
        ooo_model.set_params(
            learning_rate_init=op_lr,
            alpha=op_rp,
            batch_size=op_bz,
            hidden_layer_sizes=hidden_layer_s,
        )
        ooo_model.fit(X_train, y_train)
        y_hat = ooo_model.predict(X_val)
        # accuracy_score = oo_model.score(X_val, y_val)
        acc_score = accuracy_score(y_val, y_hat)
        accuracy_scores_layers.append(acc_score)
```

```
accuracy_matrix = np.array(accuracy_scores_layers).reshape(len(layers_l), len(nodes_l))
```

In [ ]:
```
import matplotlib.pyplot as plt


fig, ax = plt.subplots()

cax = ax.matshow(accuracy_matrix, cmap="viridis")
ax.set_xticklabels([""] + nodes_l)
ax.set_yticklabels([""] + layers_l)

for i in range(len(layers_l)):
    for j in range(len(nodes_l)):
        plt.text(
            j,
            i,
            f"{accuracy_matrix[i, j]:.2f}",
            ha="center",
            va="center",
            color="black",
        )

plt.xlabel("Nodes per layer")
plt.ylabel("Number of layers")
plt.title("Accuracy Matrix")

# Position the color bar at the bottom
fig.colorbar(cax, ax=ax, location="bottom")

plt.show()
```



> The results show different scores for the combination of nodes and layers, where the highest accuracy can be reached with 3 layers and 10 nodes (which is computationally intensive). Together with the hyperparameters of the first tuning, these hyperparmeters and architecture will impact the model best.

(d) Manual (greedy) model selection and retraining. Based the optimal choice of hyperparameters, train your model with your optimized hyperparameters on all the training data AND the validation data (this is provided as `X_train_plus_val` and `y_train_plus_val`).

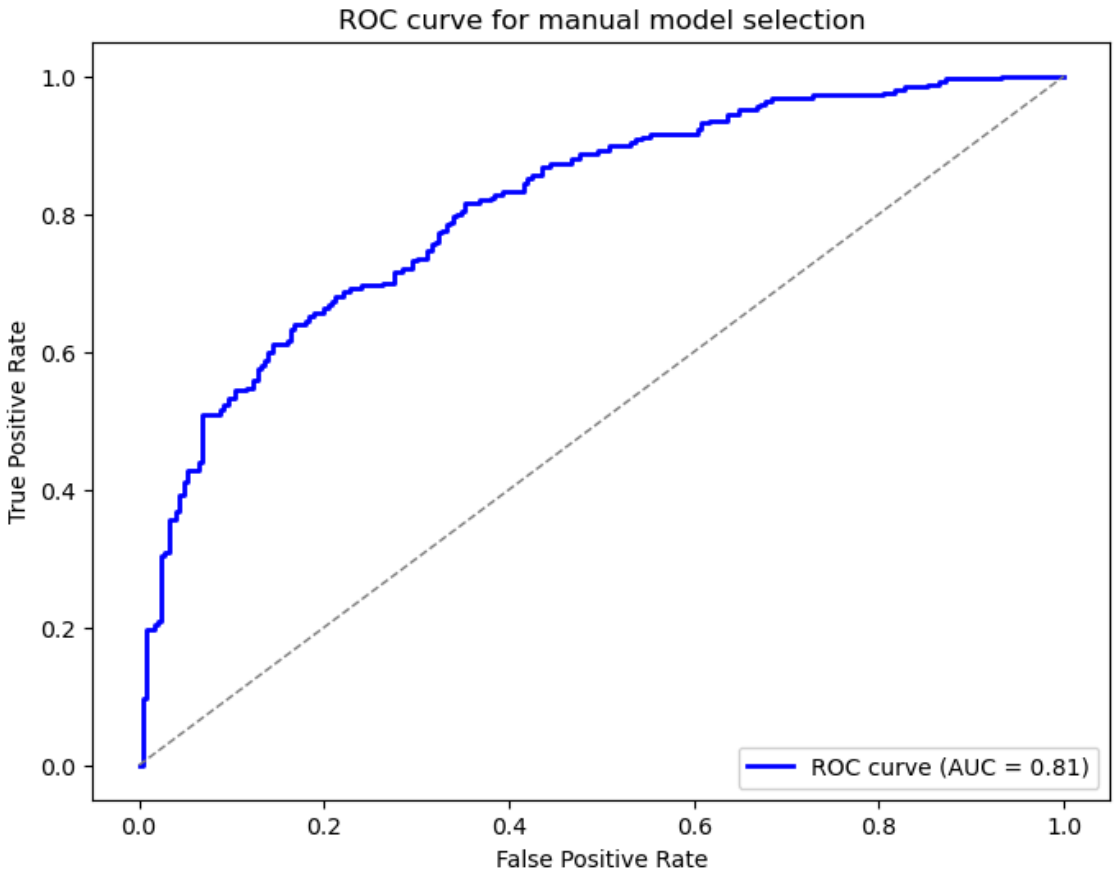
- Apply the trained model to the test data and report the accuracy of your final model on the test data.
- Plot an ROC curve of your performance (plot this with the curve in part (e) on the same set of axes you use for that question).

In [ ]:
```
from sklearn.metrics import roc_curve, roc_auc_score
```

```
gr1_model = MLPClassifier(
    learning_rate_init=op_lr,
    hidden_layer_sizes=(10, 10, 10),
    alpha=op_rp,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="relu",
    n_iter_no_change=1000,
    batch_size=op_bz,
    max_iter=5000,
)


gr1_model.fit(X_train_plus_val, y_train_plus_val)
ac_sc_test = gr1_model.score(X_test, y_test)
y_hat_test = gr1_model.predict_proba(X_test)[:, 1]

fpr, tpr, thres = roc_curve(y_test, y_hat_test)
auc_score = roc_auc_score(y_test, y_hat_test)
```

In [ ]:

```
plt.figure(figsize=(8, 6))
plt.plot(
    fpr,
    tpr,
    color="blue",
    lw=2,
    label="ROC curve (AUC = %0.2f)" % auc_score,
)
plt.plot([0, 1], [0, 1], color="gray", lw=1, linestyle="--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC curve for manual model selection")
plt.legend(loc="lower right")
plt.show()
```



(e) Automated hyperparameter search through random search. The manual (greedy) approach (setting one or two parameters at a time holding the rest constant), provides good insights into how the neural network hyperparameters impacts model fitting for this particular training process. However, it is limited in one very problematic way: it depends heavily on a good "default" setting of

the hyperparameters. Those were provided for you in this exercise, but are not generally know. Our manual optimization was somewhat greedy because we picked the hyperparameters one at a time rather than looking at different combinations of hyperparameters. Adopting such a pseudo-greedy approach to that manual optimization also limits our ability to more deeply search the hyperparameter space since we don't look at simultaneous changes to multiple parameters. Now we'll use a popular hyperparameter optimization tool to accomplish that: random search.

Random search is an excellent example of a hyperparameter optimization search strategy that has been shown to be more efficient (requiring fewer training runs) than another common approach: grid search. Grid search evaluates all possible combinations of hyperparameters from lists of possible hyperparameter settings - a very computationally expensive process. Yet another attractive alternative is Bayesian Optimization, which is an excellent hyperparameter optimization strategy but we will leave that to the interested reader.

Our particular random search tool will be Scikit-Learn's `RandomizedSearchCV`. This performs random search employing cross validation for performance evaluation (we will adjust this to ve a train/validation split).

Using `RandomizedSearchCV`, train on the training data while validating on the validation data (see instructions below on how to setup the train/validation split automatically). This tool will randomly pick combinations of parameter values and test them out, returning the best combination it finds as measured by performance on the validation set. You can use this example as a template for how to do this.

- To make this comparable to the training/validation setup used for the greedy optimization, we need to setup a training and validation split rather than use cross validation. To do this for `RandomSearchCV` we input the COMBINED training and validation dataset (`X_train_plus_val`, and `y_train_plus_val`) and we set the `cv` parameter to be the `train_val_split` variable we provided along with the dataset. This will setup the algorithm to make its assessments training just on the training data and evaluation on the validation data. Once `RandomSearchCV` completes its search, it will fit the model one more time to the combined training and validation data using the optimized parameters as we would want it to. *Note: The object returned by running fit (the random search) is NOT the best estimator. You can access the best estimator through the attribute* `.best_estimator_`, *assuming that you did not pass* `refit=False`.
- Set the number of iterations to at least 200 (you'll look at 200 random pairings of possible hyperparameters). You can go as high as you want, but it will take longer the larger the value.
- If you run this on Colab or any system with multiple cores, set the parameter `n_jobs` to -1 to use all available cores for more efficient training through parallelization
- You'll need to set the range or distribution of the parameters you want to sample from. Search over the same ranges as in previous problems. To tell the algorithm the ranges to search, use lists of values for candidate batch_size, since those need to be integers rather than a range; the `loguniform` `scipy` function for setting the range of the learning rate and regularization parameter, and a list of tuples for the `hidden_layer_sizes` parameter, as you used in the greedy optimization.
- Once the model is fit, use the `best_params_` property of the fit classifier attribute to extract the optimized values of the hyperparameters and report those and compare them to what was selected through the manual, greedy optimization.

For the final generalization performance assessment:

- State the accuracy of the optimized models on the test dataset
- Plot the ROC curve corresponding to your best model on the test dataset through greedy hyperparameter section vs the model identified through random search (these curves should be on the same set of axes for comparison). In the legend of the plot, report the AUC for each curve. This should be one single graph with 3 curves (one for greedy search, one for random search, and one representing random chance). Please also provide AUC score for greedy research and random search.
- Plot the final decision boundary for the greedy and random search-based classifiers along with the test dataset to demonstrate the shape of the final boundary
- How did the generalization performance compare between the hyperparameters selected through the manual (greedy) search and the random search?

In [ ]:
```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import loguniform

# parameters:

hidden_layer_l = []
for n_ndes in nodes_l:
    for n_layer in layers_l:
```

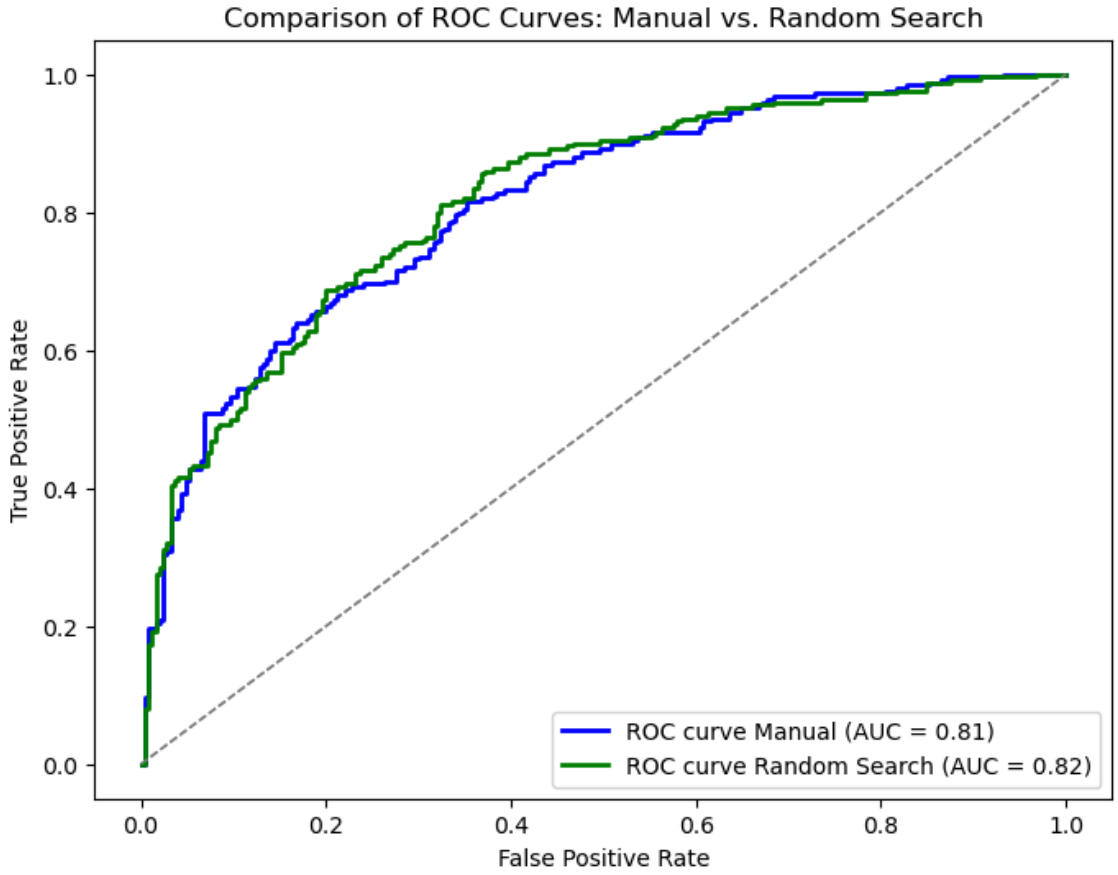
```
        hidden_layer_sizes = tuple([n_ndes] * (n_layer))
        hidden_layer_l.append(hidden_layer_sizes)
bz_list = [1, 3, 5, 10, 20, 50, 100, 250, 500]
lr_log_list = loguniform(1e-5, 1e0).rvs(20)
rp_log_list = loguniform(1e-8, 1e2).rvs(20)
```

In [ ]:

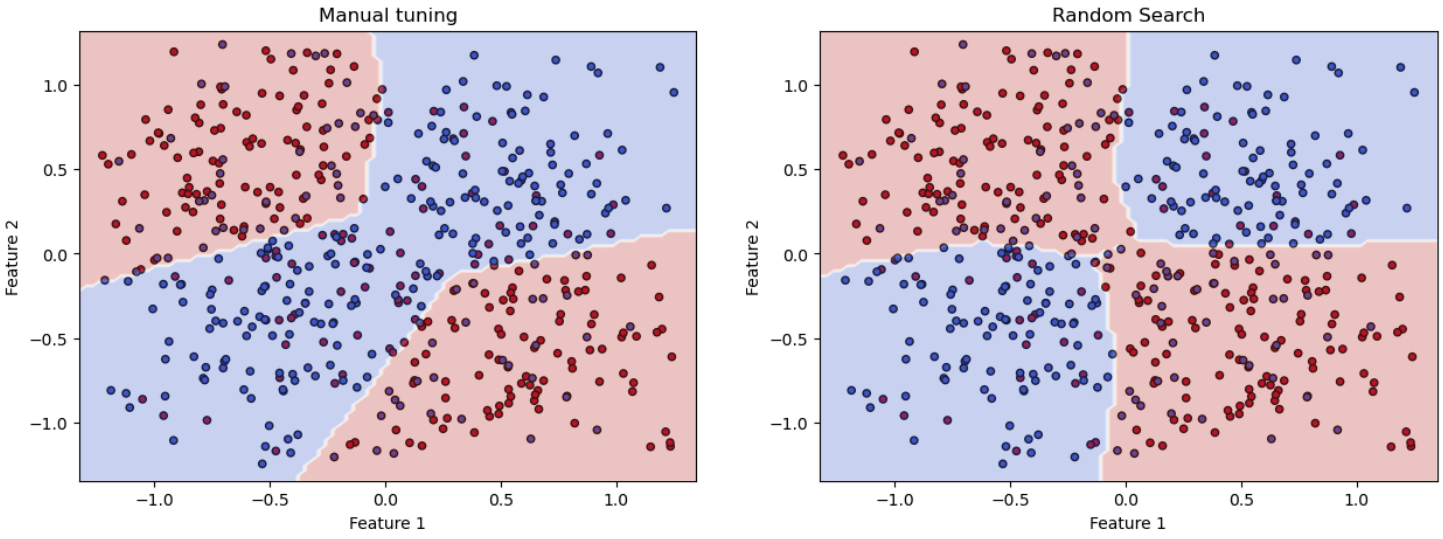
```
# generalization assesment

param_dict = {
    "hidden_layer_sizes": hidden_layer_l,
    "batch_size": bz_list,
    "learning_rate_init": lr_log_list,
    "alpha": rp_log_list,
}

mcv2 = MLPClassifier(
    random_state=12,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="relu",
    n_iter_no_change=1000,
    max_iter=500,
)

random_search = RandomizedSearchCV(
    estimator=mcv2,
    param_distributions=param_dict,
    n_iter=200,
    scoring="accuracy",
    cv=train_val_split,
    random_state=1,
    n_jobs=-1,
)

random_search.fit(X_train_plus_val, y_train_plus_val)
bp_rs = random_search.best_params_
ac_rs = random_search.score(X_test, y_test)

y_hat_rs = random_search.predict_proba(X_test)[:, 1]

fpr_rs, tpr_rs, th_random_search = roc_curve(y_test, y_hat_rs)

auc_score_rs = roc_auc_score(y_test, y_hat_rs)
```

In [ ]:

```
plt.figure(figsize=(8, 6))
plt.plot(
    fpr,
    tpr,
    color="blue",
    lw=2,
    label="ROC curve Manual (AUC = %0.2f)" % auc_score,
)

plt.plot([0, 1], [0, 1], color="gray", lw=1, linestyle="--")


plt.plot(
    fpr_rs,
    tpr_rs,
    color="green",
    lw=2,
    label="ROC curve Random Search (AUC = %0.2f)" % auc_score_rs,
)
plt.plot([0, 1], [0, 1], color="gray", lw=1, linestyle="--")

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Comparison of ROC Curves: Manual vs. Random Search")
plt.legend(loc="lower right")
plt.show()
```



In [ ]:
```
fig, axs = plt.subplots(1, 2, figsize=(15, 5))

plot_decision_boundary(X_train, y_train, gr1_model, "Manual tuning", axs[0])
plot_decision_boundary(X_train, y_train, random_search, "Random Search", axs[1])
```



In [ ]:
```
print("The best parameters for random search are the following:")
for key, value in bp_rs.items():
    print(f"{key}: {value}")
print(f"AUC:{auc_score:.2f}\naccuracy: {ac_sc_test:.2f}")


print(
    f"\nThe best parameters for manual (greedy) model are the following:\nlearning_rate_init{op_lr} \nhidden_la
)
print(f"AUC:{auc_score_rs:.2f}\naccuracy: {ac_rs:.2f}")
```

```
The best parameters for random search are the following:
learning_rate_init: 0.043758019378874864
hidden_layer_sizes: (30, 30, 30, 30)
batch_size: 500
alpha: 0.9407562153603201
AUC:0.81
accuracy: 0.73

The best parameters for manual (greedy) model are the following:
learning_rate_init0.0006951927961775605
hidden_layer_sizes: (10,10,10)
batch_size:500
alpha: 0.23357214690901212
AUC:0.82
accuracy: 0.73
```

> The generalization performance comparison of the two models allows us to see the impact of the different architectures of the model in the overall performance and accuracy. The random search model is more complex as it has 4 hidden layers with 30 nodes each, in contrast with the manual search model that has 3 layers with 10 nodes. This lack of complexity in the greedy model is beneficial in both computational times and can have lesser risk of overfitting than the random search one, making it a bit more generalizable. The AUC is slighlty greater on the random search, and the accuracy is the same in both. The greedy model proves to be more computationally efficient with almost similar accuracy scores.

2

[30 points] Build and test your own Neural Network for classification

There is no better way to understand how one of the core techniques of modern machine learning works than to build a simple version of it yourself. In this exercise you will construct and apply your own neural network classifier. You may use numpy if you wish but no other libraries.

(a) [10 points of the 30] Create a neural network class that follows the `scikit-learn` classifier convention by implementing `fit`, `predict`, and `predict_proba` methods. Your `fit` method should run backpropagation on your training data using stochastic gradient descent. Assume the activation function is a sigmoid. Choose your model architecture to have two input nodes, two hidden layers with five nodes each, and one output node.

To guide you in the right direction with this problem, please find a skeleton of a neural network class below. You absolutely MAY use additional methods beyond those suggested in this template, but the methods listed below are the minimum required to implement the model cleanly.

Strategies for debugging. One of the greatest challenges of this implementations is that there are many parts and a bug could be present in any of them. Here are some recommended tips:

- *Development environment*. Consider using an Integrated Development Environment (IDE). I strongly recommend the use of VS Code and the Python debugging tools in that development environment.
- *Unit tests*. You are strongly encouraged to create unit tests for most modules. Without doing this will make your code extremely difficult to bug. You can create simple examples to feed through the network to validate it is correctly computing activations and node values. Also, if you manually set the weights of the model, you can even calculate backpropagation by hand for some simple examples (admittedly, that unit test would be challenging and is optional, but a unit test is possible).
- *Compare against a similar architecture*. You can also verify the performance of your overall neural network by comparing it against the `scikit-learn` implementation and using the same architecture and parameters as your model (your model outputs will certainly not be identical, but they should be somewhat similar for similar parameter settings).

NOTE: Due to the depth this question requires, some students may choose not to complete this section (in lieu of receiving the 10 points from this question). If you choose not to build your own neural network, or if your neural network is not functional prior to submission, then use the `scikit-learn` implementation instead in the questions below; where it asks to compare to `scikit-learn`, compare against a random forest classifier instead.

(b) Apply your neural network.

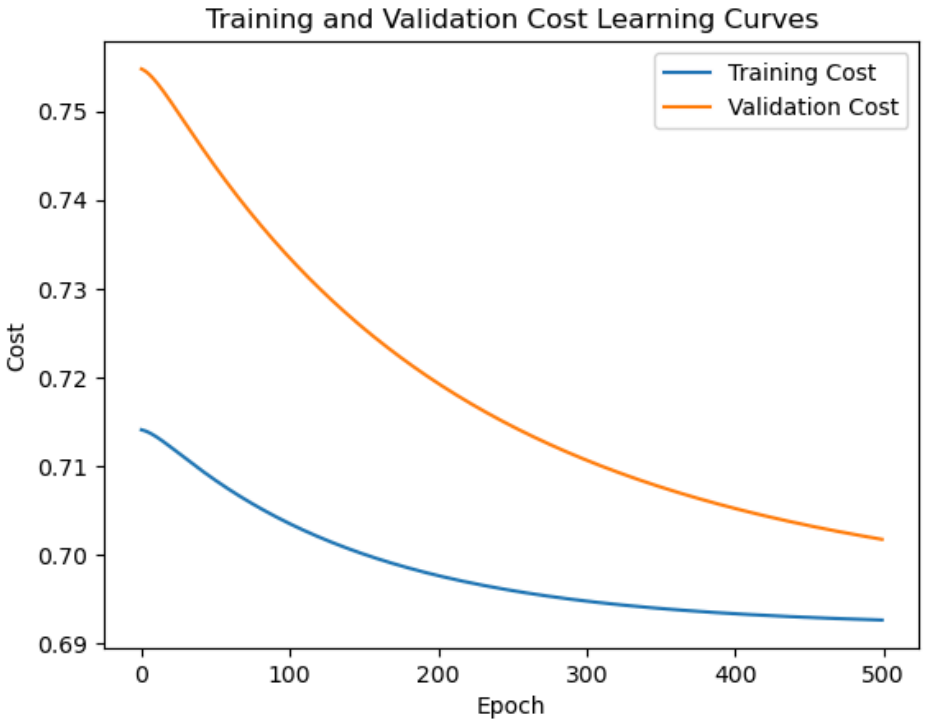
- Create training, validation, and test datasets using `sklearn.datasets.make_moons(N, noise=0.20)` data, where $N_{train} = 500$ and $N_{test} = 100$. The validation dataset should be a portion of your training dataset that you hold out for hyperparameter tuning.
- **Cost function plots**. Train and validate your model on this dataset plotting your training and validation cost learning curves on the same set of axes. This is the training and validation error for each epoch of stochastic gradient descent, where an epoch represents having trained on each of the training samples one time.
- Tune the learning rate and number of training epochs for your model to improve performance as needed. You're free to use any methods you deem fit to tune your hyperparameters like grid search, random search, Bayesian optimization etc.
- **Decision boundary plots**. In two subplots, plot the training data on one subplot and the validation data on the other subplot. On each plot, also plot the decision boundary from your neural network trained on the training data.
- **ROC Curve plots**. Report your performance on the test data with an ROC curve and the corresponding AUC score. Compare against the `scikit-learn` `MLPClassifier` trained with the same parameters on the same set of axes and include the chance diagonal. *Note: if you chose not to build your own neural network in part (a) above, or if your neural network is not functional prior to submission, then use the `scikit-learn` `MLPClassifier` class instead for the neural network and compare it against a random forest classifier instead. Be sure to set the hidden layer sizes, epochs, and learning rate for that model, if so.*
- **Remember to retrain your model.** After selecting your hyperparameters using the validation data set, when evaluating the final performance on the ROC curve, it's good practice to retrain your model with the selected hyperparameters on the train + validation dataset, before evaluating on the test data.

Note if you opted not to build your own neural network: in this case, for hyperparameter tuning, we recommend using the `partial_fit` method to train your model for every epoch. Partial fit allows you to incrementally fit on one sample at a time.

In [ ]:
```
from sklearn.datasets import make_moons
import sklearn
from sklearn.model_selection import train_test_split
from sklearn.metrics import log_loss

X, y = make_moons(600, noise=0.20, random_state=42)


Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=100, random_state=42)


Xtrain, Xval, ytrain, yval = train_test_split(
    Xtrain, ytrain, test_size=0.2, random_state=42
)
```

In [ ]:
```
q2_mlp = MLPClassifier(
    learning_rate_init=op_lr,
    hidden_layer_sizes=(5, 5),
    alpha=op_rp,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="logistic",
    # n_iter_no_change=1000,
    batch_size=op_bz,
    max_iter=5000,
    random_state=42,
)


train_costs = []
validation_costs = []

for epoch in range(500):
    q2_mlp.partial_fit(Xtrain, ytrain, classes=np.unique(ytrain))
    train_cost = log_loss(ytrain, q2_mlp.predict_proba(Xtrain))
    validation_cost = log_loss(yval, q2_mlp.predict_proba(Xval))

    train_costs.append(train_cost)
    validation_costs.append(validation_cost)
```

```
plt.plot(train_costs, label="Training Cost")
plt.plot(validation_costs, label="Validation Cost")
plt.xlabel("Epoch")
plt.ylabel("Cost")
plt.title("Training and Validation Cost Learning Curves")
plt.legend()
plt.show()
```



In [ ]:
```
# generalization assesment

m_moon = MLPClassifier(
    hidden_layer_sizes=(5, 5),
    alpha=0,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="logistic",
    n_iter_no_change=1000,
    batch_size=50,
    random_state=42,
)


lr_log_list = loguniform(1e-5, 1e0).rvs(20)

ep_list = list(range(1, 501))


param_dict = {"learning_rate_init": lr_log_list, "max_iter": ep_list}


X_train_plus_val = np.concatenate((Xtrain, Xval), axis=0)
y_train_plus_val = np.concatenate((ytrain, yval), axis=0)
# Create a predefined train/test split for RandomSearchCV (to be used later)
validation_fold = np.concatenate((-1 * np.ones(len(ytrain)), np.zeros(len(yval))))
train_val_split2 = PredefinedSplit(validation_fold)

random_search2 = RandomizedSearchCV(
    estimator=m_moon,
    param_distributions=param_dict,
    n_iter=200,
    scoring="accuracy",
    cv=train_val_split2,
    random_state=1,
```

```
    n_jobs=-1,
)

random_search2.fit(X_train_plus_val, y_train_plus_val)
bp_rs2 = random_search2.best_params_
ac_rs2 = random_search2.score(Xtest, ytest)

y_hat_rs2 = random_search2.predict_proba(Xtest)[:, 1]

fpr_rs2, tpr_rs2, th_rs2 = roc_curve(ytest, y_hat_rs2)

auc_score_rs_2 = roc_auc_score(ytest, y_hat_rs2)
```

> To improve the performance, the following hiperparameters will change:

In [ ]:
```
for key, value in bp_rs2.items():
    print(f"{key}: {value}")
```

```
max_iter: 218
learning_rate_init: 0.7563578877748215
```

In [ ]:
```
m_moon3 = MLPClassifier(
    learning_rate_init=bp_rs2["learning_rate_init"],
    hidden_layer_sizes=(5, 5),
    alpha=0,
    solver="sgd",
    tol=1e-5,
    early_stopping=False,
    activation="logistic",
    n_iter_no_change=1000,
    batch_size=50,
    max_iter=bp_rs2["max_iter"],
    random_state=490,
)

m_moon3.fit(Xtrain, ytrain)
```

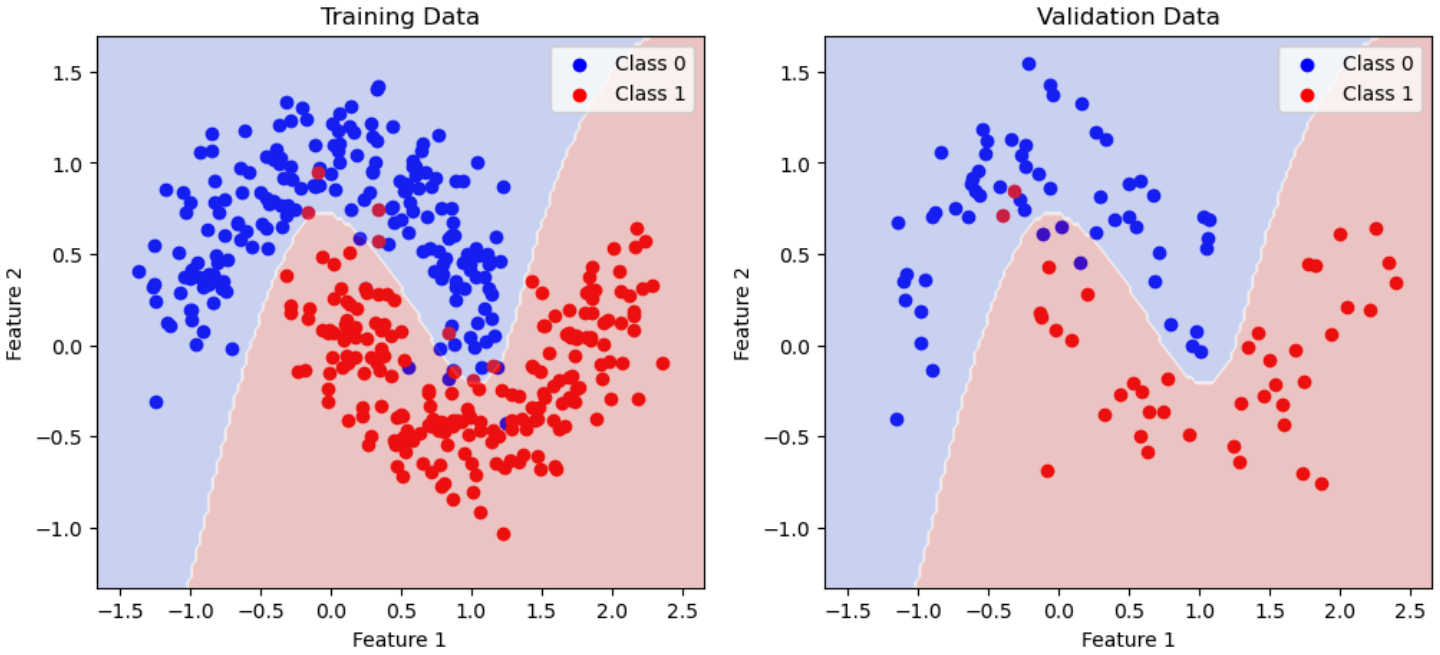
Out[ ]:

MLPClassifier

```
MLPClassifier(activation='logistic', alpha=0, batch_size=50,
              hidden_layer_sizes=(5, 5), learning_rate_init=0.7563578877748215,
              max_iter=218, n_iter_no_change=1000, random_state=490,
              solver='sgd', tol=1e-05)
```

In [ ]:
```
def plot_boundary(X, model):
    x_min, x_max = X[:, 0].min() - 0.3, X[:, 0].max() + 0.3
    y_min, y_max = X[:, 1].min() - 0.3, X[:, 1].max() + 0.3

    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.03), np.arange(y_min, y_max, 0.03))
    mesh_data = np.column_stack((xx.ravel(), yy.ravel()))
    Z = model.predict(mesh_data)
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.coolwarm, antialiased=True)


plt.figure(figsize=(12, 5))

# training plot
plt.subplot(1, 2, 1)
plt.scatter(
    Xtrain[ytrain == 0][:, 0],
    Xtrain[ytrain == 0][:, 1],
    color="blue",
    label="Class 0",
)
plt.scatter(
    Xtrain[ytrain == 1][:, 0],
    Xtrain[ytrain == 1][:, 1],
    color="red",
```

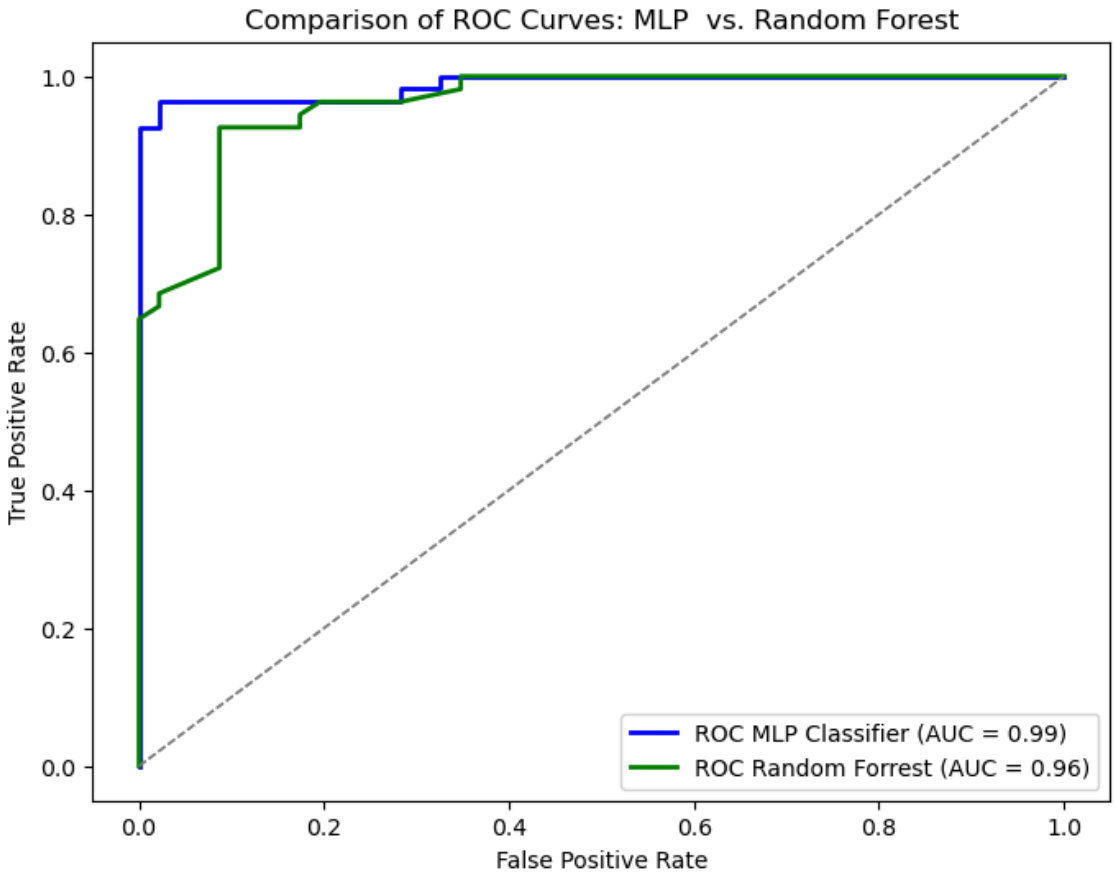
```
    label="Class 1",
)

plot_boundary(Xtrain, m_moon3)
plt.title("Training Data")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()


# validation plot
plt.subplot(1, 2, 2)
plt.scatter(Xval[yval == 0][:, 0], Xval[yval == 0][:, 1], color="blue", label="Class 0")
plt.scatter(Xval[yval == 1][:, 0], Xval[yval == 1][:, 1], color="red", label="Class 1")

plot_boundary(Xtrain, m_moon3)

plt.title("Validation Data")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()


plt.show()
```



In [ ]:
```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(max_depth=2, random_state=0)
rf.fit(Xtrain, ytrain)
y_hat_rf = rf.predict_proba(Xtest)[:, 1]


fpr_rf, tpr_rf, th_rf = roc_curve(ytest, y_hat_rf)

auc_score_rf = roc_auc_score(ytest, y_hat_rf)
```

In [ ]:
```
plt.figure(figsize=(8, 6))
plt.plot(
    fpr_rs2,
    tpr_rs2,
    color="blue",
    lw=2,
    label="ROC MLP Classifier (AUC = %0.2f)" % auc_score_rs_2,
)

plt.plot([0, 1], [0, 1], color="gray", lw=1, linestyle="--")
```

```
plt.plot(
    fpr_rf,
    tpr_rf,
    color="green",
    lw=2,
    label="ROC Random Forrest (AUC = %0.2f)" % auc_score_rf,
)
plt.plot([0, 1], [0, 1], color="gray", lw=1, linestyle="--")

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Comparison of ROC Curves: MLP  vs. Random Forest")
plt.legend(loc="lower right")
plt.show()
```



(c) Suggest two ways in which you neural network implementation could be improved: are there any options we discussed in class that were not included in your implementation that could improve performance?

In [ ]: