Problem 1

3. It doesn't matter which is used

Problem 2

2. D_2

Problem 3

3. f(x) = 3

Problem 4

1. Model 1

M5-L1 Problem 1

In this problem, you will implement a function to calculate gini impurity on an arbitrary input vector.

For reference, the formula for Gini impurity is:

$$\text{Gini}(D) = 1 - \sum_{i=1}^{k} p_i^2$$

where D is the dataset containing samples from k classes and p_i is the probability of a data point belonging to class i.

Gini Impurity Function

Complete the function `gini(D)` below. It should take as input a 1-D array, where is the number of samples corresponding to each output class.

For example, consider the input array `D = np.array([4, 9, 7, 0, 3])` In this example, there are 5 input classes and 23 total samples. For this input, your function should return 0.707.

Your function should work regardless of the length of the input vector.

In [1]:
```
import numpy as np

def gini(D):
    # YOUR CODE GOES HERE
    sum = np.sum(D)
    gini_ = 1 - np.sum(np.square(D / sum))
    return gini_

D = np.array([4, 9, 7, 0, 3])
g = gini(D)
print(f"gini([4,9,7,0,3]) = {g:.3f} (should be about {0.707})")
```

```
gini([4,9,7,0,3]) = 0.707 (should be about 0.707)
```

More test cases

Compute and print the gini impurity for `D1`, `D2`, `D3`, and `D4`, defined below:

In [2]:
```
D1 = np.array([1,0,0])
D2 = np.array([0,0,4])
D3 = np.array([0, 20, 0, 0, 0, 3])
D4 = np.array([6, 6, 6, 6])

for D in [D1, D2, D3, D4]:
    # YOUR CODE GOES HERE
    g = gini(D)
    print(f"gini({D}) = {g:.3f}")
```

```
gini([1 0 0]) = 0.000
gini([0 0 4]) = 0.000
gini([ 0 20  0  0  0  3]) = 0.227
gini([6 6 6 6]) = 0.750
```

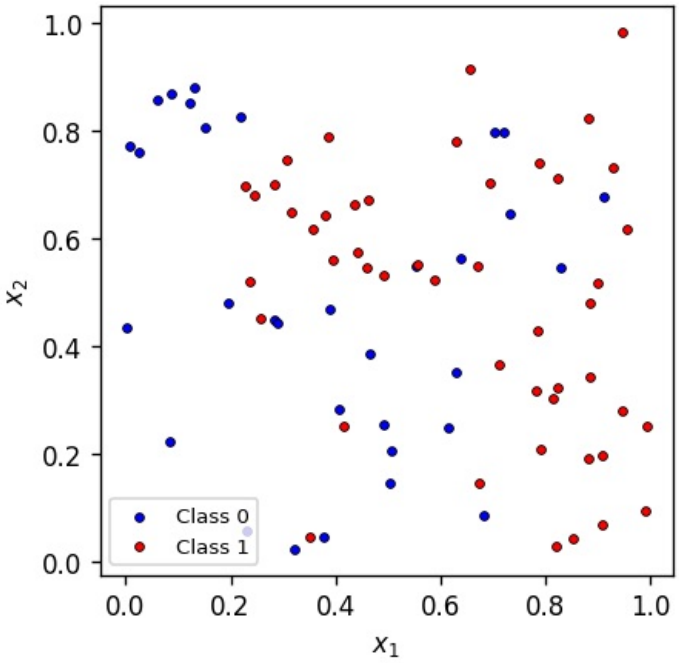
M5-L1 Problem 2

Now we will provide a 2D classification dataset and you will learn to use sklearn's decision tree classifier on the data.

First, run the following cell to load the data and import decision tree tools.

- Input: `X`, size 80×2
- Output: `y`, size 80

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier, plot_tree
from matplotlib.colors import ListedColormap

y = np.array([1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0
x1 = np.array([6.73834679e-01, 3.57095269e-01, 4.42510505e-01, 8.48412660e-02, 2.17890220e-01, 4.60241400e-01,
x2 = np.array([0.14784469, 0.61647661, 0.57595235, 0.2232836 , 0.82559199, 0.54569237, 0.73986085, 0.79782627,
X = np.vstack([x1, x2]).T

def plot_data(X,y):
    colors=["blue","red"]
    for i in range(2):
        plt.scatter(X[y==i,0],X[y==i,1],s=12,c=colors[i],edgecolors="black",linewidths=.5,label=f"Class {i}")
        plt.xlabel("$x_1$")
        plt.ylabel("$x_2$")
        plt.legend(loc="lower left",prop={'size':8})

plt.figure(figsize=(4,4),dpi=120)
plot_data(X,y)
plt.show()
```



Create and fit a decision tree classifier

Create an instance of a `DecisionTreeClassifier()` with `max_depth` of 5. Fit this to the data `X`, `y`.

For more details, consult: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

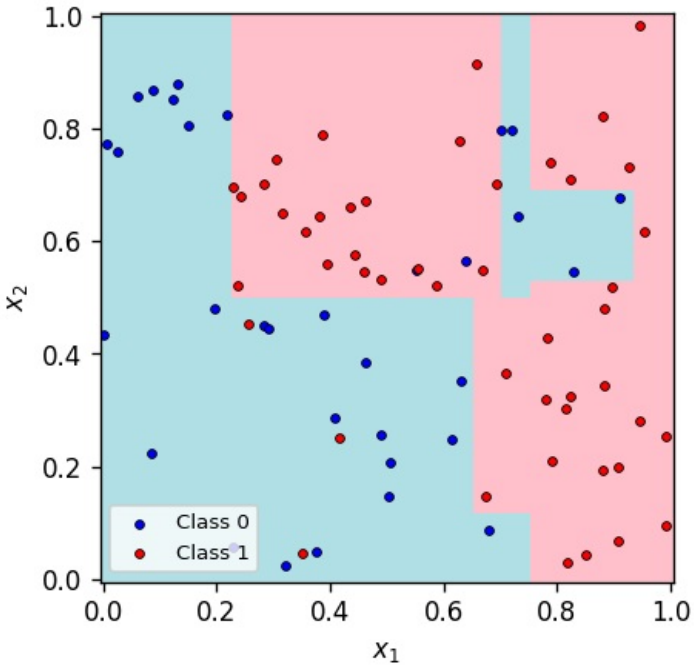
In [2]:
```
# YOUR CODE GOES HERE
model = DecisionTreeClassifier(max_depth = 5)
model.fit(X, y)
```

Out[2]:
```
DecisionTreeClassifier(max_depth=5)
```

Making new predictions using your model

Now use the decision tree you trained to evaluate on the meshgrid of points `X_test` as indicated below. The code here will generate a plot showing the decision boundaries created by the model.

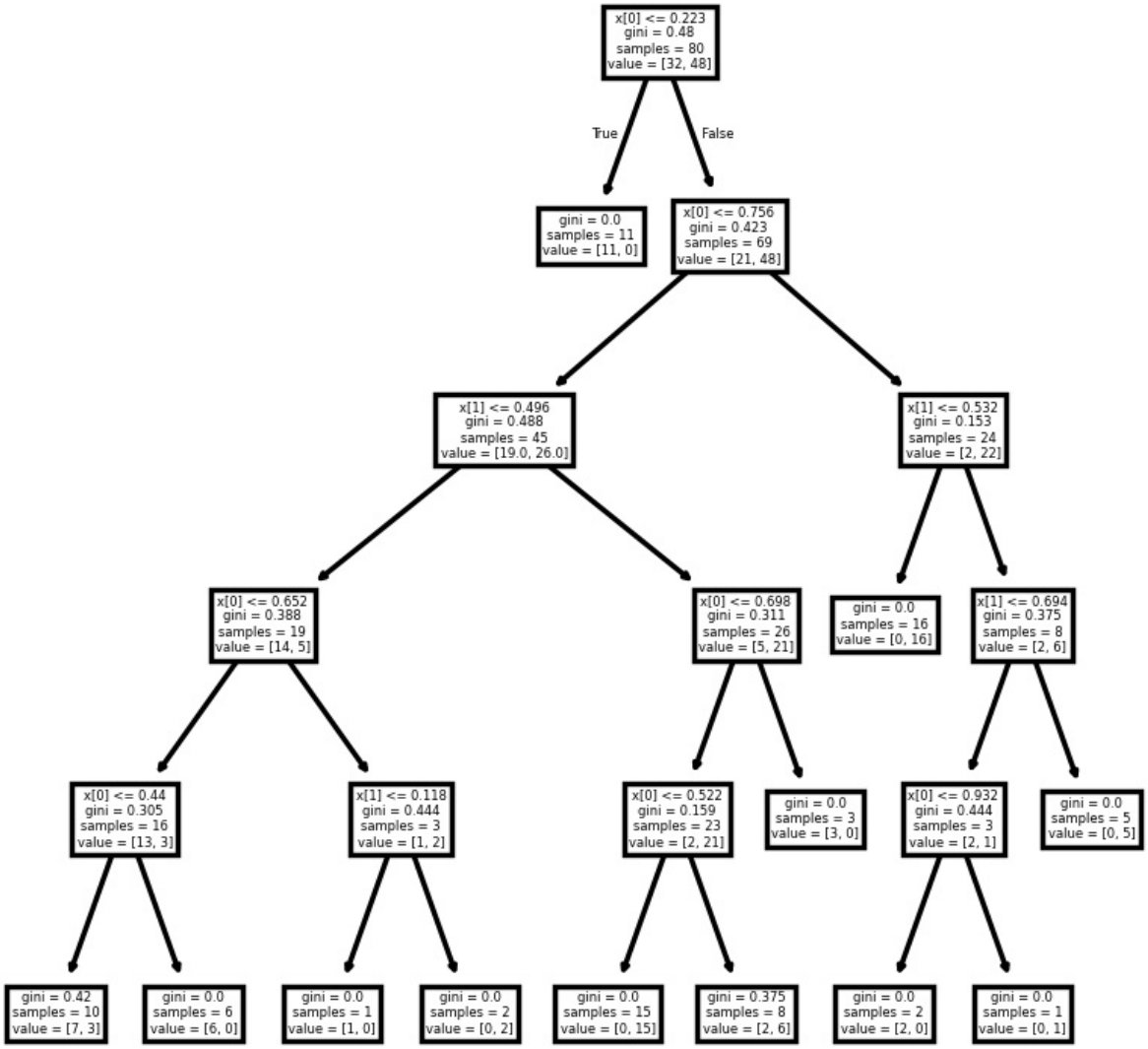
In [3]:
```
vals = np.linspace(0,1,100)
x1grid, x2grid = np.meshgrid(vals, vals)

X_test = np.vstack([x1grid.flatten(), x2grid.flatten()]).T


# YOUR CODE GOES HERE
# compute a prediction, `pred` for the input `X_test`
pred = model.predict(X_test)


plt.figure(figsize=(4,4),dpi=120)
bgcolors = ListedColormap(["powderblue","pink"])
plt.pcolormesh(x1grid, x2grid, pred.reshape(x1grid.shape), shading="nearest",cmap=bgcolors)
plot_data(X,y)
plt.show()
```



Visualizing the decision tree

The `plot_tree()` function (https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot_tree.html) can generate a simple visualization of your decision tree model. Try out this function below:

In [4]:
```
plt.figure(figsize=(4,4),dpi=250)

# YOUR CODE GOES HERE
plot_tree(model)

plt.show()
```

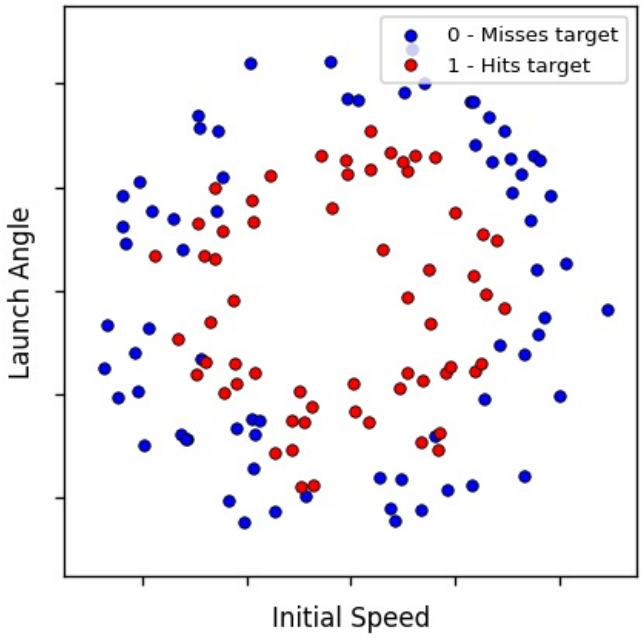


M5-L1 Problem 3

Let's revisit the initial speed vs. launch angle data from the logistic regression module. This time, you will train a decision tree classifier to predict whether a projectile launched with a given speed and angle will hit a target.

Run this cell to load the data and decision tree tools:

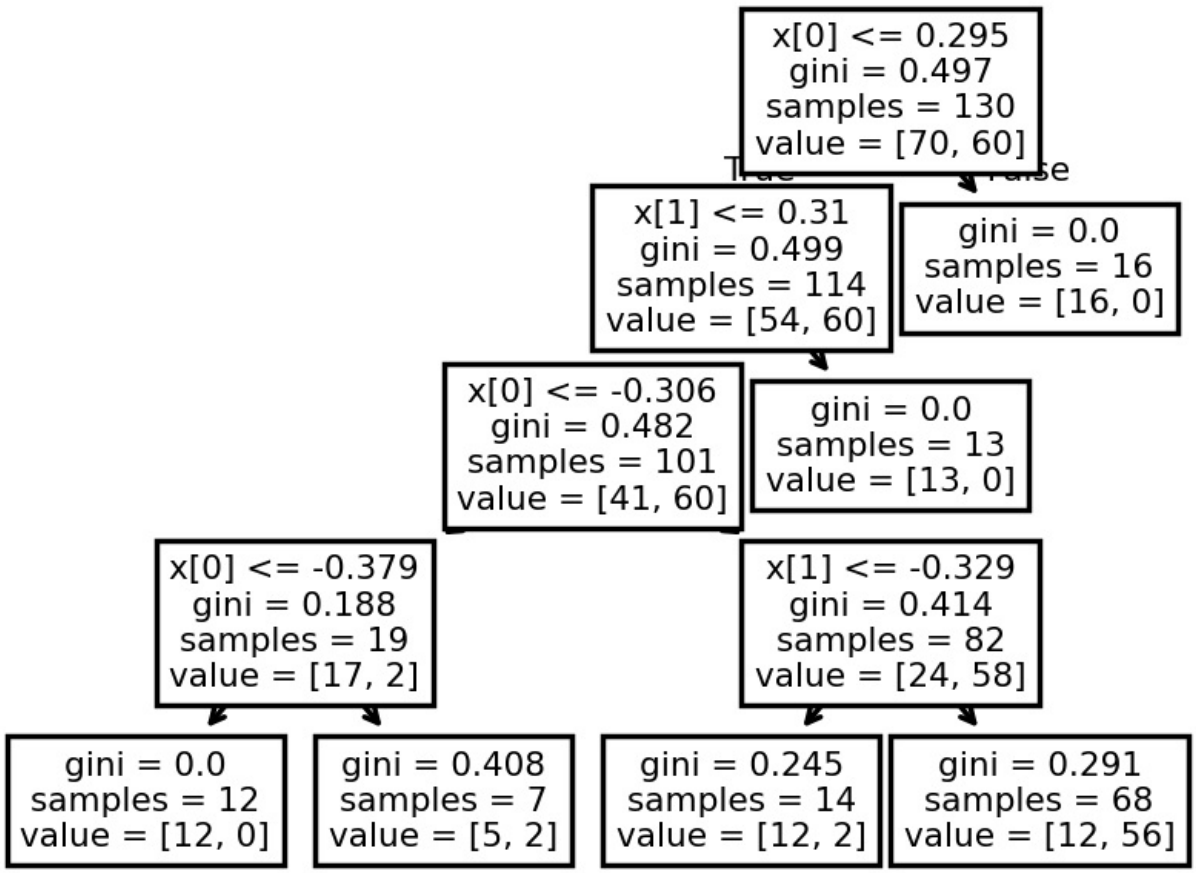
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier, plot_tree
from matplotlib.colors import ListedColormap

y = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
x1 = np.array([0.02693745, 0.41186575, 0.10363585, 0.08489663, 0.09512868, 0.31121109, 0.16015486, 0.75698706,
x2 = np.array([0.3501823 , 0.10349458, 0.20137442, 0.37973165, 0.71062143, 0.25377085, 0.64055034, 0.29218012,
X = np.vstack([x1, x2]).T

def plot_data(X,y):
    colors=["blue","red"]
    labels = ["0 - Misses target", "1 - Hits target"]
    for i in range(2):
        plt.scatter(X[y==i,0],X[y==i,1],s=20,c=colors[i],edgecolors="black",linewidths=.5,label=labels[i])
        plt.xlabel("Initial Speed")
        plt.ylabel("Launch Angle")
        plt.legend(loc="upper right",prop={'size':8})
        ax = plt.gca()
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        plt.xlim([-0.55,.55])
        plt.ylim([-0.55,.55])

plt.figure(figsize=(4,4),dpi=120)
plot_data(X,y)
plt.show()
```



Training a decision tree classifier.

Below, a decision tree of max depth 4 is trained, and the tree is visualized with `plot_tree()`.

In [2]:
```
dt = DecisionTreeClassifier(max_depth=4)
dt.fit(X,y)

plt.figure(figsize=(4,3),dpi=250)
plot_tree(dt)
plt.show()
```



Accuracy on training data

Compute the accuracy on the training data with the provided function `get_dt_accuracy(dt, X, y)`. Print the result.

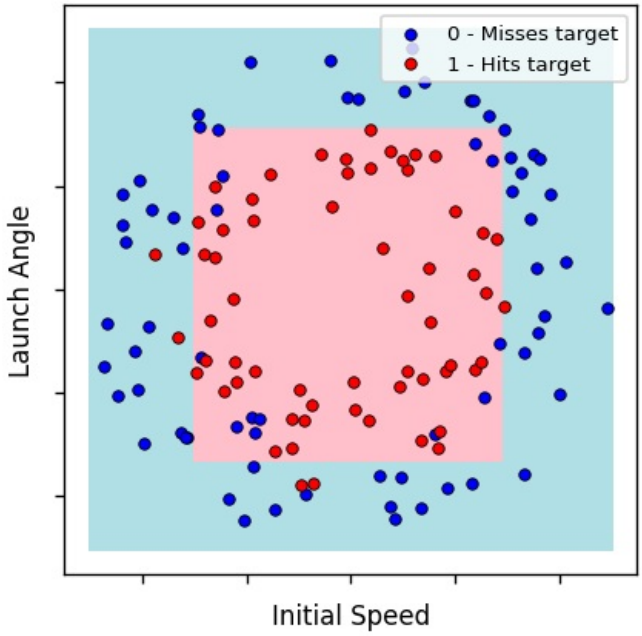
In [3]:
```
def get_dt_accuracy(dt, X, y):
    pred = dt.predict(X)
    return 100*np.sum(pred == y)/len(y)

# YOUR CODE GOES HERE
accuracy = get_dt_accuracy(dt, X, y)
print("Accuracy of prediction by decision tree: ", accuracy)
```

```
Accuracy of prediction by decision tree:  87.6923076923077
```

Visualizing tree predictions

By evaluating the model on a meshgrid of results, we can look at how our model performs on the input space:

In [4]:
```
vals = np.linspace(-.5,.5,100)
x1grid, x2grid = np.meshgrid(vals, vals)
X_test = np.vstack([x1grid.flatten(), x2grid.flatten()]).T

pred = dt.predict(X_test)

plt.figure(figsize=(4,4),dpi=120)
bgcolors = ListedColormap(["powderblue","pink"])
plt.pcolormesh(x1grid, x2grid, pred.reshape(x1grid.shape), shading="nearest",cmap=bgcolors)
plot_data(X,y)
plt.show()
```

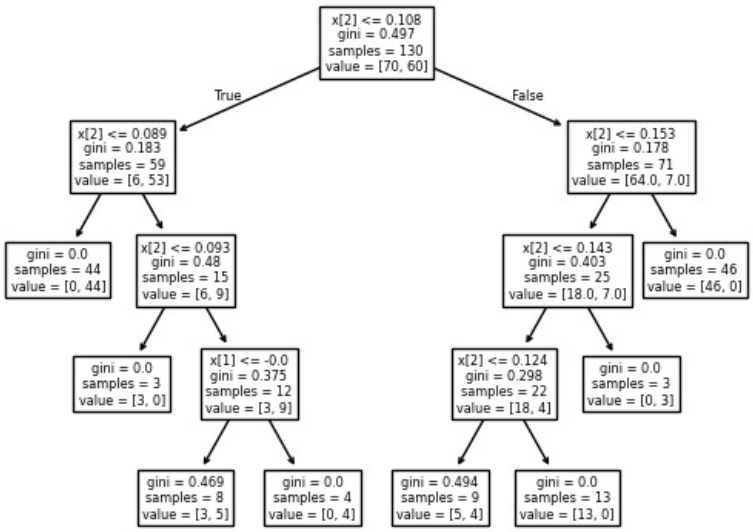


Expanded feature set

Now, we will add a third feature that (for this problem) happens to be very useful. That feature is $x_1^2 + x_2^2$. A new training input `X_ex` is generated below containing this additional feature.

Train a new decision tree, max depth 4, on this data. Then visualize the tree with `plot_tree()`.

In [10]:
```
def feature_expand(X):
    x1 = X[:,0].reshape(-1, 1)
    x2 = X[:,1].reshape(-1, 1)
    columns = [x1, x2, x1*x1 + x2*x2]
    return np.concatenate(columns, axis=1)

X_ex = feature_expand(X)

# YOUR CODE GOES HERE
# Train a new decision tree on X_ex, y
dt.fit(X_ex, y)
# Plot the tree
plot_tree(dt)
plt.show()
```



Accuracy on training data: expanded features

Compute the accuracy of this new model its training data. It should have increased. Note that the useful features to expand will vary significantly from problem to problem.

In [7]:
```
# YOUR CODE GOES HERE
accuracy = get_dt_accuracy(dt, X_ex, y)
```

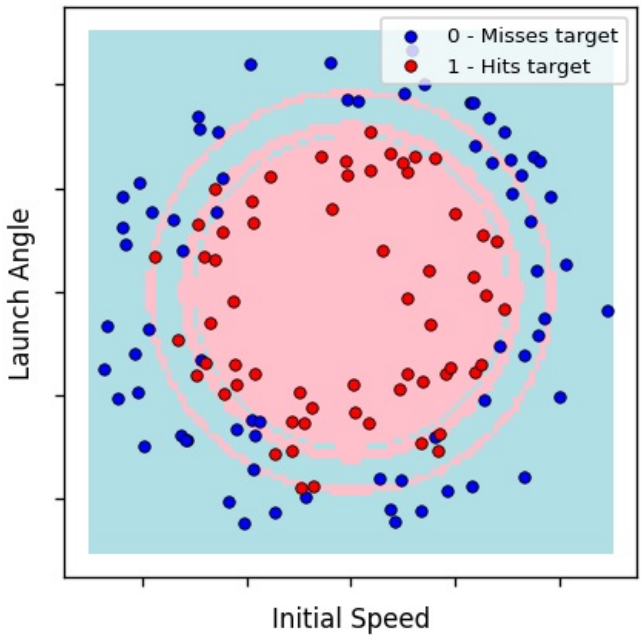
```
print("Accuracy after training on expanded features: ", accuracy)
```

```
Accuracy after training on expanded features:  94.61538461538461
```

Visualizing expanded feature results

Use your model to make a prediction called `pred` on the data `X_test_ex`, an expanded meshgrid of points, as indicated. This code will plot the class decisions. Note the difference between this and the previous model, which only had speed and angle as features.

In [13]:
```
X_test_ex = feature_expand(X_test)
# YOUR CODE GOES HERE
# Have your model make a prediction, `pred` on X_test_ex
pred = dt.predict(X_test_ex)

plt.figure(figsize=(4,4),dpi=120)
bgcolors = ListedColormap(["powderblue","pink"])
plt.pcolormesh(x1grid, x2grid, pred.reshape(x1grid.shape), shading="nearest",cmap=bgcolors)
plot_data(X,y)
plt.show()
```



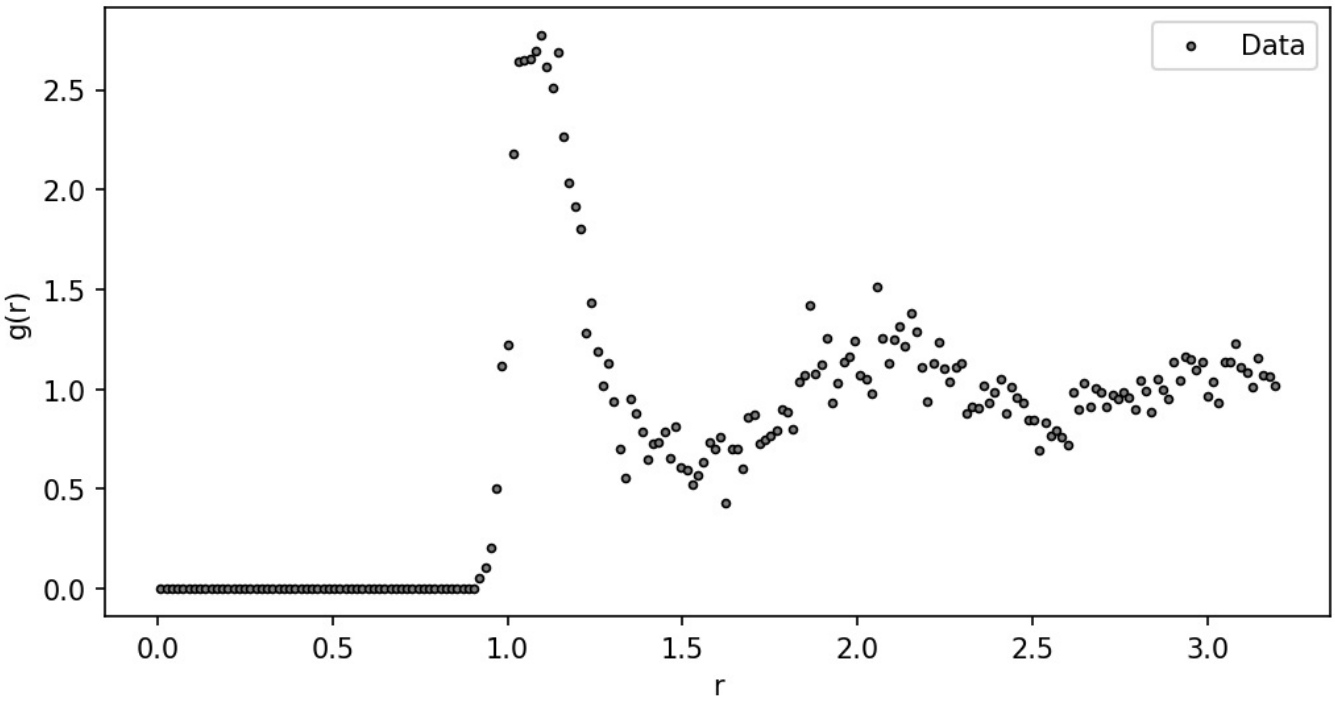
M5-L2 Problem 1

256 particles of liquid argon are simulated at 100K. A radial distribution function $g(r)$ describes the density of particles a distance of r from each particle in the system. When an $g(r)$ is computed in a simulation, it is done by creating a histogram of particle distances for a single simulation frame, resulting in a noisy function that is most often averaged over several frames.

Given g(r) vs. r data for a single frame, you will train a decision tree regressor to represent the underlying function.

First, run the cell below to load the data, etc.:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor,plot_tree

r = np.array([0.008,0.024,0.04,0.056,0.072,0.088,0.104,0.12,0.136,0.152,0.168,0.184,0.2,0.216,0.232,0.248,0.264
g = np.array([0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0

def plot(r, g, dt = None):
    if dt is not None:
        plt.figure(figsize=(12,3),dpi=150)
        plt.subplot(121)
        rs = np.linspace(0,4,1000)
        gs = dt.predict(rs.reshape(-1,1))
        plt.plot(rs,gs,color="red",label="Regression Tree",alpha=0.7)
    else:
        plt.figure(figsize=(8,4),dpi=150)

    plt.scatter(r,g,s=8,c="gray", label="Data", edgecolors="black",linewidths=.8)
    plt.legend(loc="upper right")
    plt.xlabel("r")
    plt.ylabel("g(r)")

    if dt is not None:
        plt.subplot(122)
        plot_tree(dt)
        plt.title(f"Tree max. depth: {dt.max_depth}",y=-.2)

    plt.show()

plot(r,g)
```



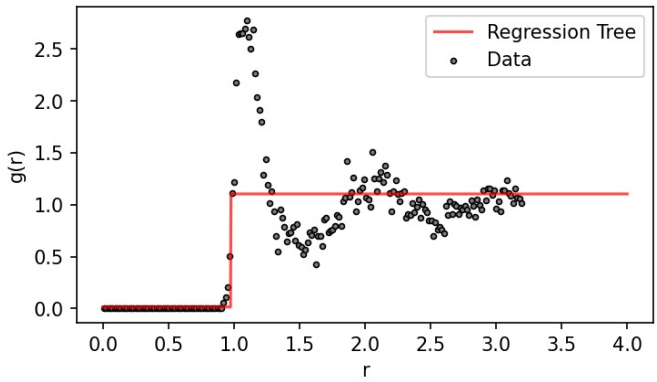
Training regression trees

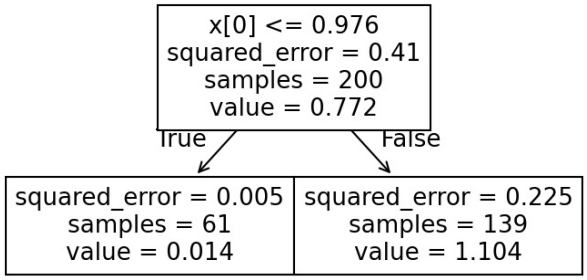
For input `r` and output `g`, train a `DecisionTreeRegressor()` to perform the regression with `max_depth` values of 1, 2, 6, 10.

Complete the code below, which will plot your decision tree results and visualize the tree. Name each decision tree within the loop `dt`.

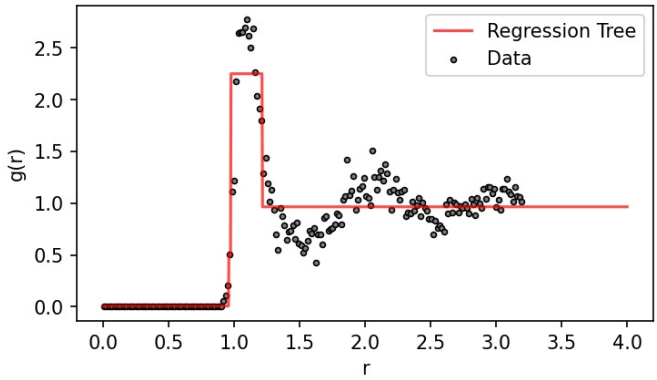
Note: you may need to resize the input `r` as `r.reshape(-1,1)` before passing it as input into the fitting function.

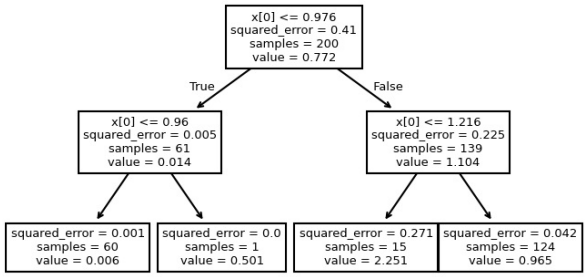
In [2]:
```
for max_depth in [1, 2, 6, 10]:
    # YOUR CODE GOES HERE
    # Create and fit `dt`
    dt = DecisionTreeRegressor(max_depth = max_depth)
    r_ = r.reshape(-1, 1)
    dt.fit(r_, g)
    plot(r,g,dt)
```



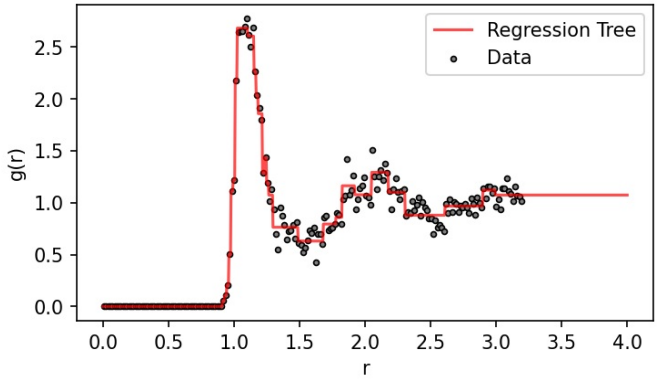


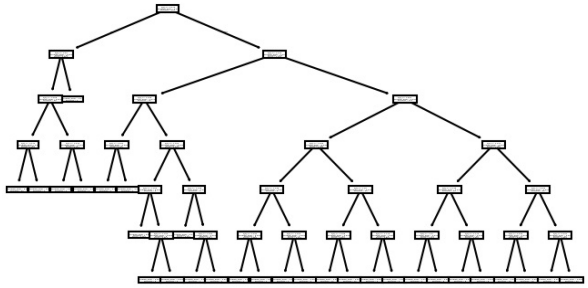
Tree max. depth: 1



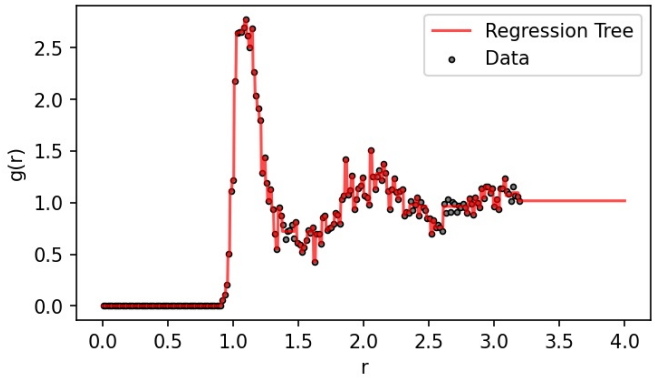


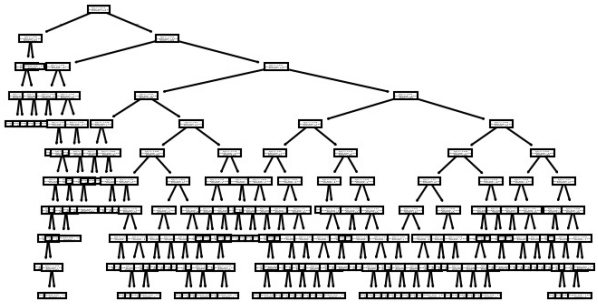
Tree max. depth: 2





Tree max. depth: 6





Tree max. depth: 10

M5-L2 Problem 2

Stress-strain measurements have been collected for many samples across many parts, resulting in much noisier data than would come from a tensile test, for example. Your job is to train an ensemble of decision trees that can predict stress for an input strain.

Scikit-Learn's `RandomForestRegressor()` has several parameters that you will experiment with below.

Run each cell; then, experiment with different settings of the `RandomForestRegressor()` to answer the questions at the end.

In [1]:
```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual, Layout, FloatSlider, Dropdown

# Load the data
y = np.array([133.18473289, 366.12422297, 453.70990214, 479.37136253, 238.16361712,  39.91719443, 282.21638562,
x = np.array([0.47358185, 0.80005535, 1.10968143, 1.85282726, 0.58177792, 0.24407275, 0.67817621, 0.59768343, 1
```

In [2]:
```
def plot(n_estimators, max_leaf_nodes, bootstrap):
    n_estimators = [1,10,20,30,40,50,60,70,80,90,100][int(n_estimators)]
    max_leaf_nodes = int(max_leaf_nodes)
    model = RandomForestRegressor(n_estimators=n_estimators,
                                  bootstrap=(True if "On" in bootstrap else False),
                                  max_leaf_nodes=max_leaf_nodes,
                                  random_state=0)
    model.fit(x.reshape(-1,1), y)

    xs = np.linspace(min(x),max(x),500)
    ys = model.predict(xs.reshape(-1,1))

    plt.figure(figsize=(5,3),dpi=150)
    plt.scatter(x,y,s=20,color="cornflowerblue",edgecolor="navy",label="Data")
    plt.plot(xs, ys, c="red",linewidth=2,label="Mean prediction")
    for i,dt in enumerate(model.estimators_):
        label = "Tree predictions" if i == 0 else None
        plt.plot(xs, dt.predict(xs.reshape(-1,1)), c="gray",linewidth=.5,zorder=-1, label = label)

    plt.legend(loc="lower right",prop={"size":8})
    plt.xlabel("Strain, %")
    plt.ylabel("Stress, MPa")
    plt.title(f"Num. estimators: {n_estimators}, Max leaves = {max_leaf_nodes}, Bootstrapping: {bootstrap}",fon
    plt.show()

slider1 = FloatSlider(
    value=2,
    min=0,
    max=10,
    step=1,
    description='# Estimators',
    disabled=False,
    continuous_update=True,
    orientation='horizontal',
    readout=False,
    layout = Layout(width='550px')
)

slider2 = FloatSlider(
    value=5,
    min=2,
    max=25,
    step=1,
    description='Max Leaves',
    disabled=False,
    continuous_update=True,
    orientation='horizontal',
    readout=False,
    layout = Layout(width='550px')
)

dropdown = Dropdown(
    options=["On (66% of data)", "Off"],
    value="On (66% of data)",
    description='Bootstrap',
    disabled=False,
)
```

```
interactive_plot = interactive(
    plot,
    bootstrap = dropdown,
    n_estimators = slider1,
    max_leaf_nodes = slider2
    )
output = interactive_plot.children[-1]
output.layout.height = '500px'

interactive_plot
```

Out[2]: interactive(children=(FloatSlider(value=2.0, description='# Estimators', layout=Layout(width='550px'), max=10.…

Questions

1. Keep bootstrapping on and set max leaf nodes constant at 3. Describe what happens to the mean prediction as the number of estimators increases.
2. Keep bootstrapping on and set number of estimators constant at 100. Describe what happens to the mean prediction as the leaf node maximum increases.
3. Now disable bootstrapping. Notice that all of the predictions are the same -- the gray lines are behind the red. Why is this? (Hint: Think about the number of features in this dataset.)

1. As the number of estimators increases while the number of max leaf nodes remain constant at 3, the mean prediction becomes more stable as the fluctuations in the individual trees average out. However, since each tree is constrained to 3 nodes, their complexity is still limited and they maintain a high bias. The mean prediction ends up being rather smooth and misses out on capturing finer details.
2. Increasing the maximum number of nodes in individual trees increases their complexities, and this helps in capturing more complex data. The mean prediciton will have reduced bias, and maintain a somewhat low variance due to averaging. As the number of nodes increases too much, individual tree may end up overfitting to their samples. This will cause it to susceptible to outliers and noise. The mean predicition still somewhat converges thanks to averaging, however, the change in performance if any, might not be significant.
3. Since the dataset has very few features, each tree has only limited options for splitting the data between its nodes. So it's highly likely that the algorithm makes the same split across all the trees. This means that the trees are essentially identical clones of each other. So, since all the predictions are the same, they appear to be hidden behind the mean prediction since they overlap perfectly.

Problem 6

Problem Description

In this problem you will train decision tree and random forest models using sklearn on a real world dataset. The dataset is the *Cylinder Bands Data Set* from the UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/Cylinder+Bands. The dataset is generated from rotogravure printers, with 39 unique features, and a binary classification label for each sample. The class is either 0, for 'band' or 1 for 'no band', where banding is an undesirable process delay that arises during the rotogravure printing process. By training ML models on this dataset, you could help identify or predict cases where these process delays are avoidable, thereby improving the efficiency of the printing. For the sake of this exercise, we only consider features 21-39 in the above link, and have removed any samples with missing values in that range. No further processing of the data is required on your behalf. The data has been partitioned into a training and testing set using an 80/20 split. Your models will be trained on just the train set, and accuracy results will be reported on both the training and testing sets.

Fill out the notebook as instructed, making the requested plots and printing necessary values.

You are welcome to use any of the code provided in the lecture activities.

Summary of deliverables:

- Accuracy function
- Report accuracy of the DT model on the training and testing set
- Report accuracy of the Random Forest model on the training and testing set

Imports and Utility Functions:

In [1]:
```
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
```

Load the data

Use the `np.load()` function to load "w5-hw1-train.npy" (training data) and "w5-hw1-test.npy" (testing data). The first 19 columns of each are the features. The last column is the label

In [12]:
```
# YOUR CODE GOES HERE
train_data = np.load("data/w5-hw1-train.npy")
test_data = np.load("data/w5-hw1-test.npy")

X_train = train_data[:, :19]
y_train = train_data[:, -1]

X_test = test_data[:, :19]
y_test = test_data[:, -1]
```

Write an accuracy function

Write a function `accuracy(pred,label)` that takes in the models prediction, and returns the percentage of predictions that match the corresponding labels.

In [9]:
```
# YOUR CODE GOES HERE
def accuracy(pred, label):
    acc = np.sum((pred == label)) * 100 / len(pred)
    return acc
```

Train a decision tree model

Train a decision tree using `DecisionTreeClassifier()` with a `max_depth` of 10 and using a `random_state` of 0 to ensure repeatable results. Print the accuracy of the model on both the training and testing sets.

In [19]:
```
# YOUR CODE GOES HERE
model = DecisionTreeClassifier(max_depth = 10, random_state = 0)
model.fit(X_train, y_train)

print("Model accuracy of DecisionTreeClassifier on training data: ", accuracy(model.predict(X_train), y_train))
print("Model accuracy of DecisionTreeClassifier on test data: ", accuracy(model.predict(X_test), y_test))
```

```
Model accuracy on training data:  93.12714776632302
Model accuracy on test data:  65.75342465753425
```

Train a random forest model

Train a random forest model using `RandomForestClassifier()` with a `max_depth` of 10, a `n_estimators` of 100, and using a random state of `0` to ensure repeatable results. Print the accuracy of the model on both the training and testing sets.

In [20]:
```
# YOUR CODE GOES HERE
model_ = RandomForestClassifier(max_depth = 10, n_estimators = 100, random_state = 0)
model_.fit(X_train, y_train)

print("Model accuracy of RandomForestClassifier on training data: ", accuracy(model_.predict(X_train), y_train))
print("Model accuracy of RandomForestClassifier on test data: ", accuracy(model_.predict(X_test), y_test))
```

```
Model accuracy of RandomForestClassifier on training data:  100.0
Model accuracy of RandomForestClassifier on test data:  82.1917808219178
```

Discuss the performance of the models

Compare the training and testing accuracy of the two models, and explain why the random forest model is advantageous compared to a standard decision tree model

A single decision tree might not be very accurate, especially if the depth is limited, there is noise in the data etc. Whereas in a random forest model, predictions of multiple decision trees are averaged out, resulting in irregularities being smoothened out and making the model less sensitive to noise and outliers. A decision might also be overfit to the data, which is avoided in random forest models, as the averaging and randomness generalizes the data better.

Problem 2

Problem Description

In this problem, you are given a dataset with two input features and one output. You will use a regression tree to make predictions for this data, evaluating each model on both training and testing data. Then, you will repeat this for multiple random forests.

Fill out the notebook as instructed, making the requested plots and printing necessary values.

You are welcome to use any of the code provided in the lecture activities.

Summary of deliverables:

- RMSE function
- Create 4 decision tree prediction surface plots
- Create 4 random forest prediction surface plots
- Print RMSE for train and test data for 4 decision tree models
- Print RMSE for train and test data for 4 random forest models
- Answer the 3 questions posed throughout

Imports and Utility Functions:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

def make_plot(X,y,model, title=""):
    res = 100
    xrange = np.linspace(min(X[:,0]),max(X[:,0]),res)
    yrange = np.linspace(min(X[:,1]),max(X[:,1]),res)
    x1,x2 = np.meshgrid(xrange,yrange)
    xmesh = np.vstack([x1.flatten(),x2.flatten()]).T
    z = model.predict(xmesh).reshape(res,res)

    fig = plt.figure(figsize=(12,10))
    plt.subplots_adjust(left=0.3,right=0.9,bottom=.3,top=.9)
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(x1,x2,z,cmap=cm.coolwarm,linewidth=0,alpha=0.9)
    ax.scatter(X[:,0],X[:,1],y,'o',c='black')
    ax.set_xlabel('$x_1$')
    ax.set_ylabel('$x_2$')
    ax.set_zlabel('y')
    plt.title(title)
    plt.show()
```

Load the data

Use the `np.load()` function to load "w5-hw2-train.npy" (training data) and "w5-hw2-test.npy" (testing data). The first two columns of each are the input features. The last column is the output. You should end up with 4 variables, input and output for each of the datasets.

In [4]:
```
# YOUR CODE GOES HERE
data_train = np.load("data/w5-hw2-train.npy")
data_test = np.load("data/w5-hw2-test.npy")

X_train = data_train[:, 0:-1]
y_train = data_train[:, -1]

X_test = data_test[:, 0:-1]
y_test = data_test[:, -1]
```

RMSE function

Complete a root-mean-squared-error function, `RMSE(y, pred)`, which takes in two arrays, and computes the RMSE between them:

In [6]:
```
def RMSE(y, pred):
    # YOUR CODE GOES HERE
    rms = np.sqrt(np.sum(np.square(pred - y)) / len(y))
    return rms
```

Regression trees

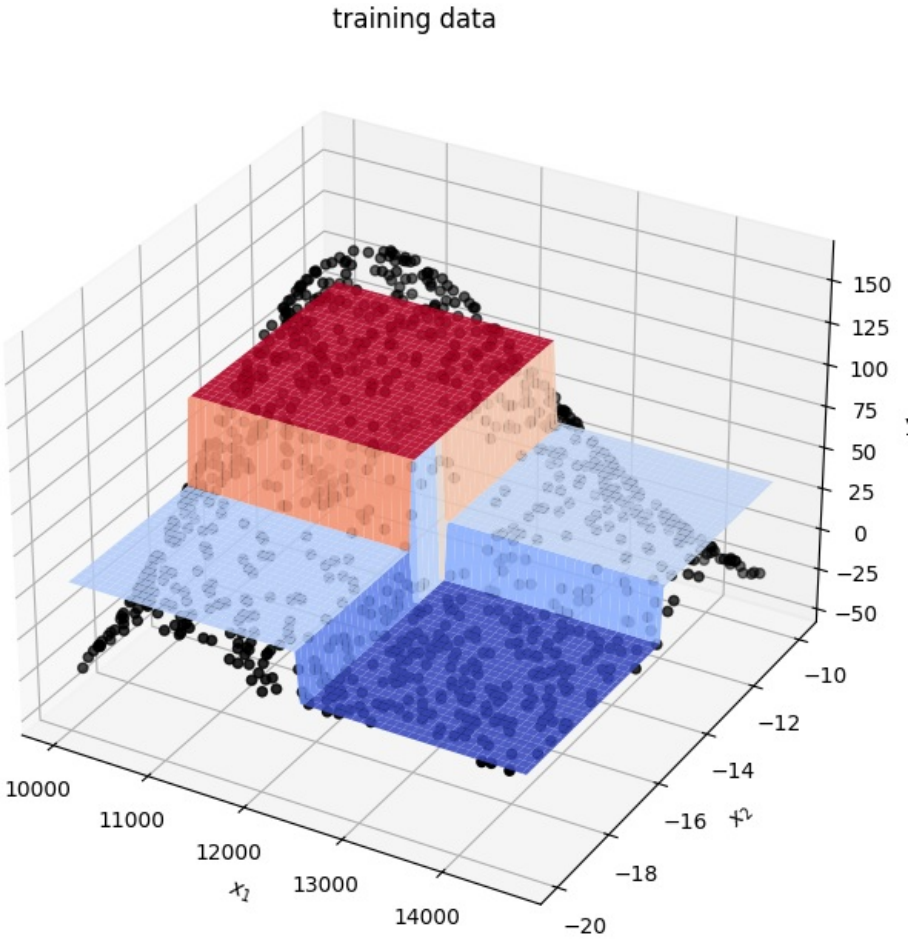
Train 4 regression trees in sklearn, with max depth values [2,5,10,25]. Train your models on the training data.

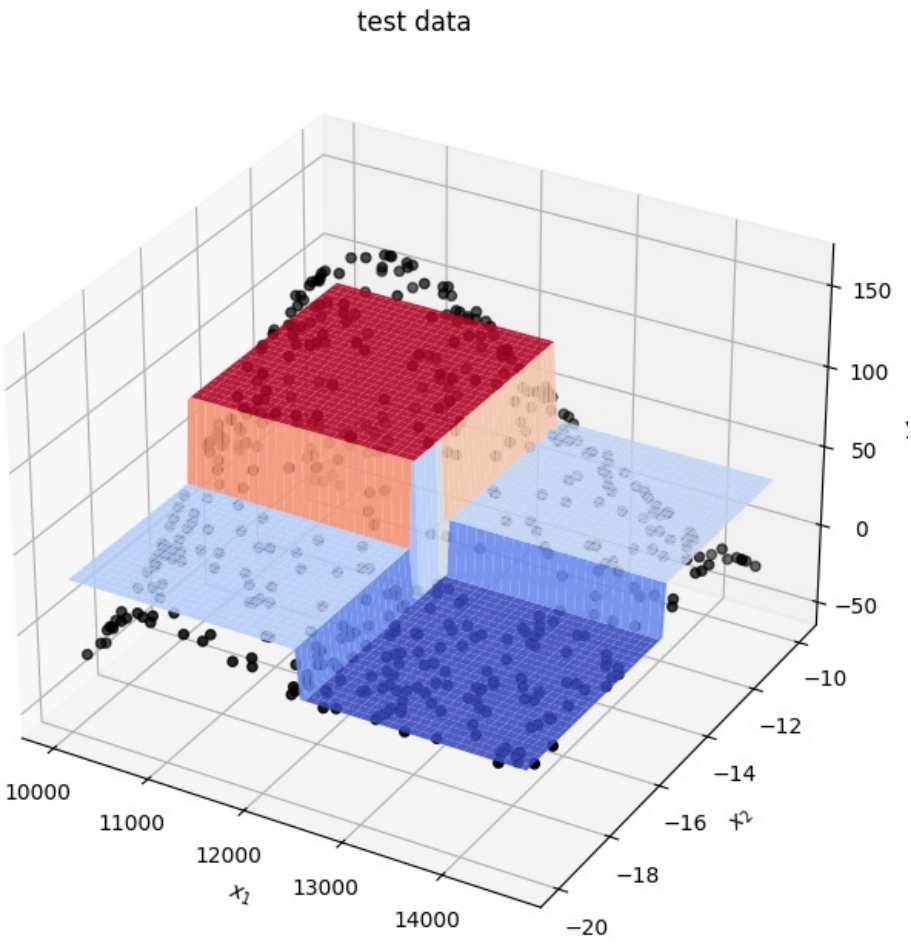
Plot the predictions as a surface plot along with test points -- you can use the provided function: `make_plot(X, y, model, title)`.

For each model, compute the train and test RMSE by calling your RMSE function. Print these results.

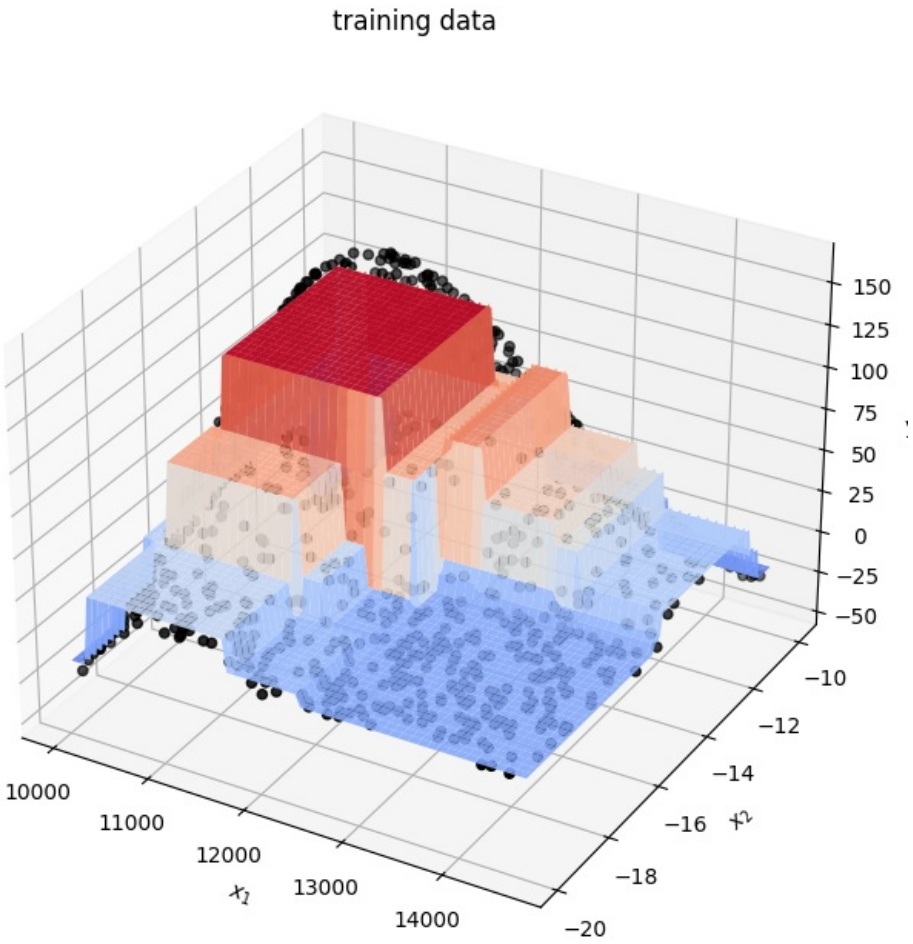
In [13]:
```
# YOUR CODE GOES HERE
depth = [2, 5, 10, 25]


for i in depth:
    dt = DecisionTreeRegressor(max_depth = i)
    dt.fit(X_train, y_train)

    pred_train = dt.predict(X_train)
    pred_test = dt.predict(X_test)
    rms_train = RMSE(y_train, pred_train)
    rms_test = RMSE(y_test, pred_test)

    print("\nMax Depth: ", i)
    print("RMS error for Decision Tree Regressor on training data: ", rms_train)
    print("RMS error for Decision Tree Regressor on test data: ", rms_test)
    make_plot(X_train, y_train, dt, title = "training data")
    make_plot(X_test, y_test, dt, title = "test data")
```

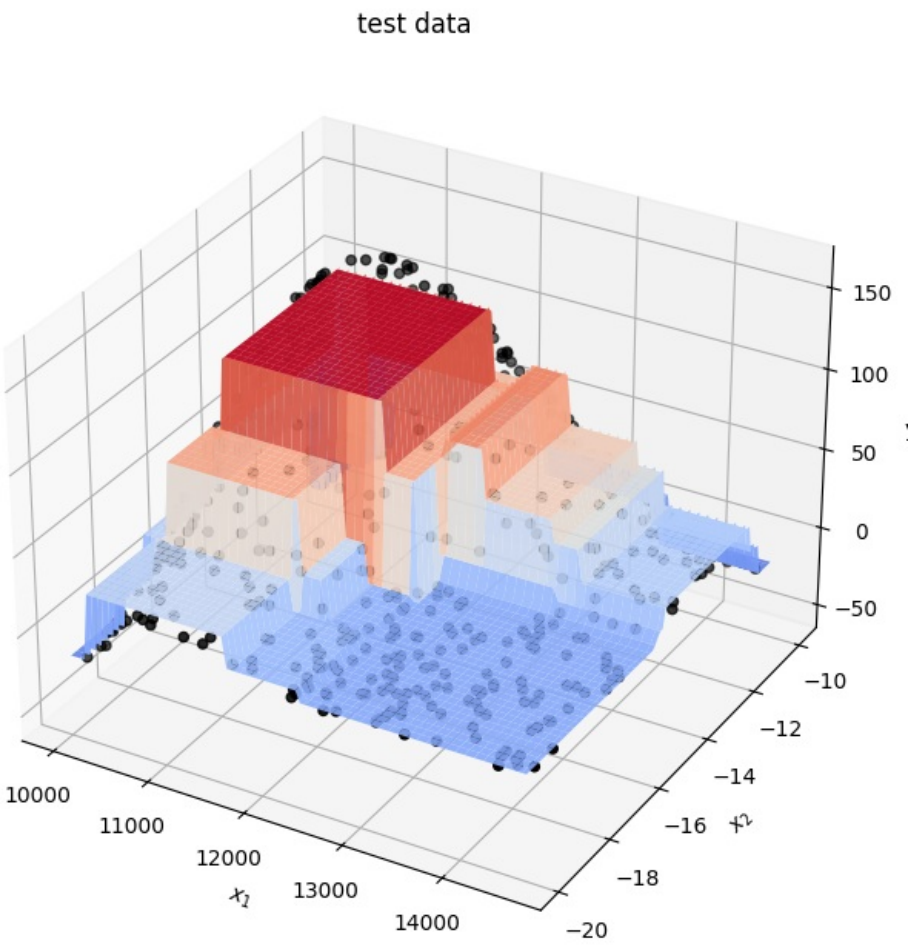
```
Max Depth:  2
RMS error for Decision Tree Regressor on training data:  35.47184989095342
RMS error for Decision Tree Regressor on test data:  37.54886839401237
```



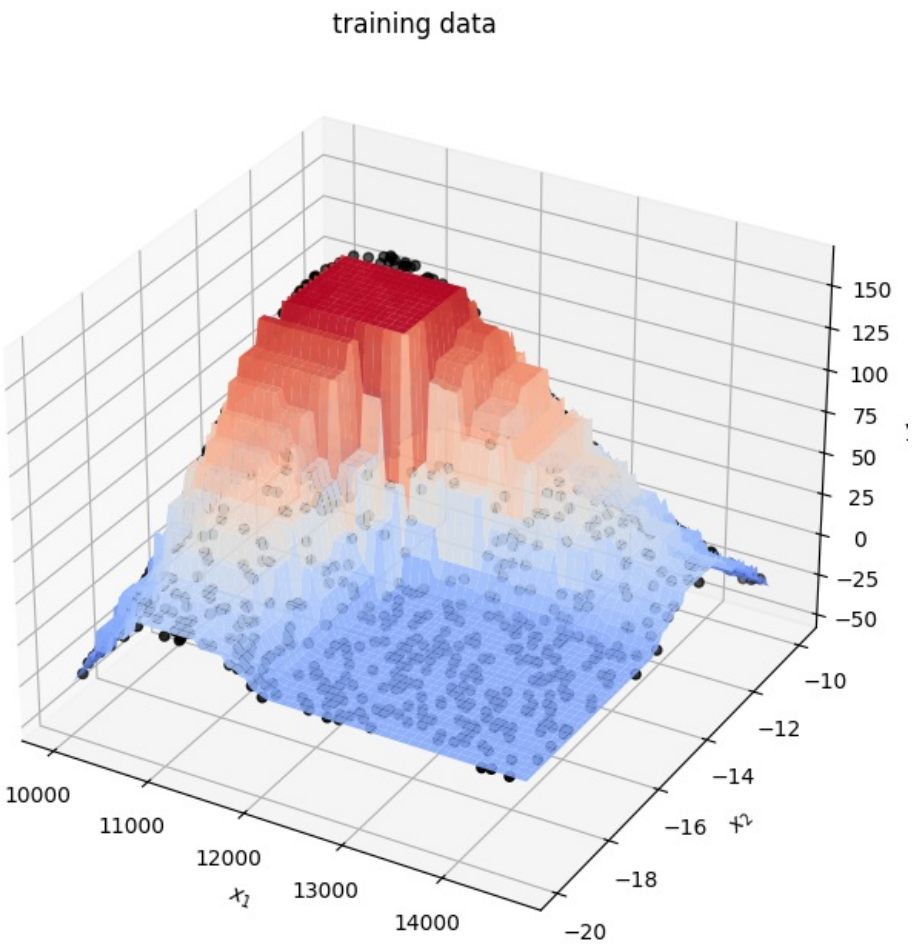


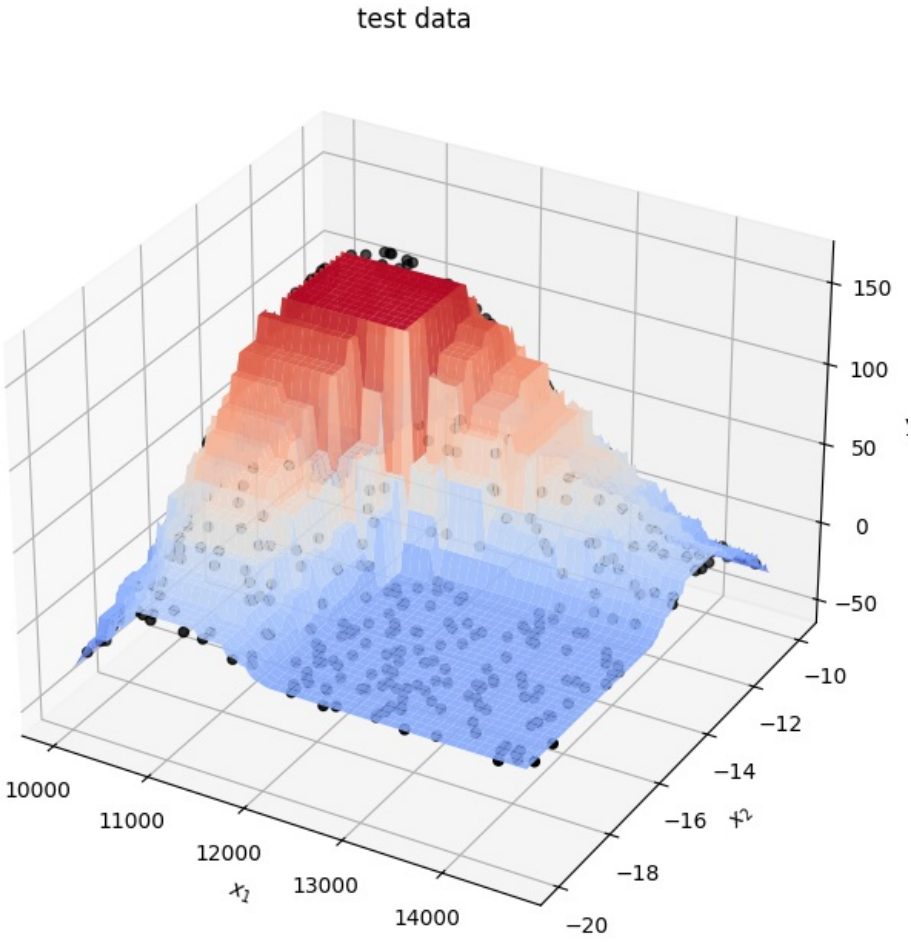

```
Max Depth:  5
RMS error for Decision Tree Regressor on training data:  17.932673237502154
RMS error for Decision Tree Regressor on test data:  19.02935744931633
```


training data
150
125
100
75
50
25
0
−25
−50
10000
11000
12000
13000
14000
x_1
−10
−12
−14
−16
−18
−20
x_2

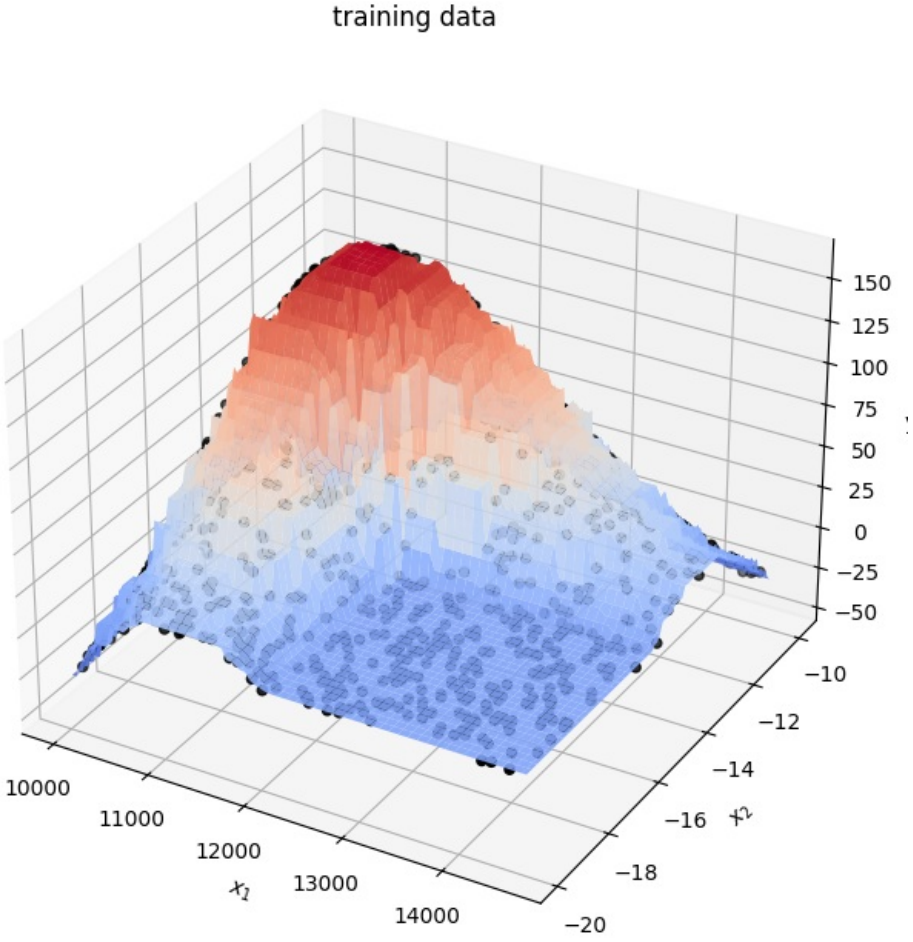


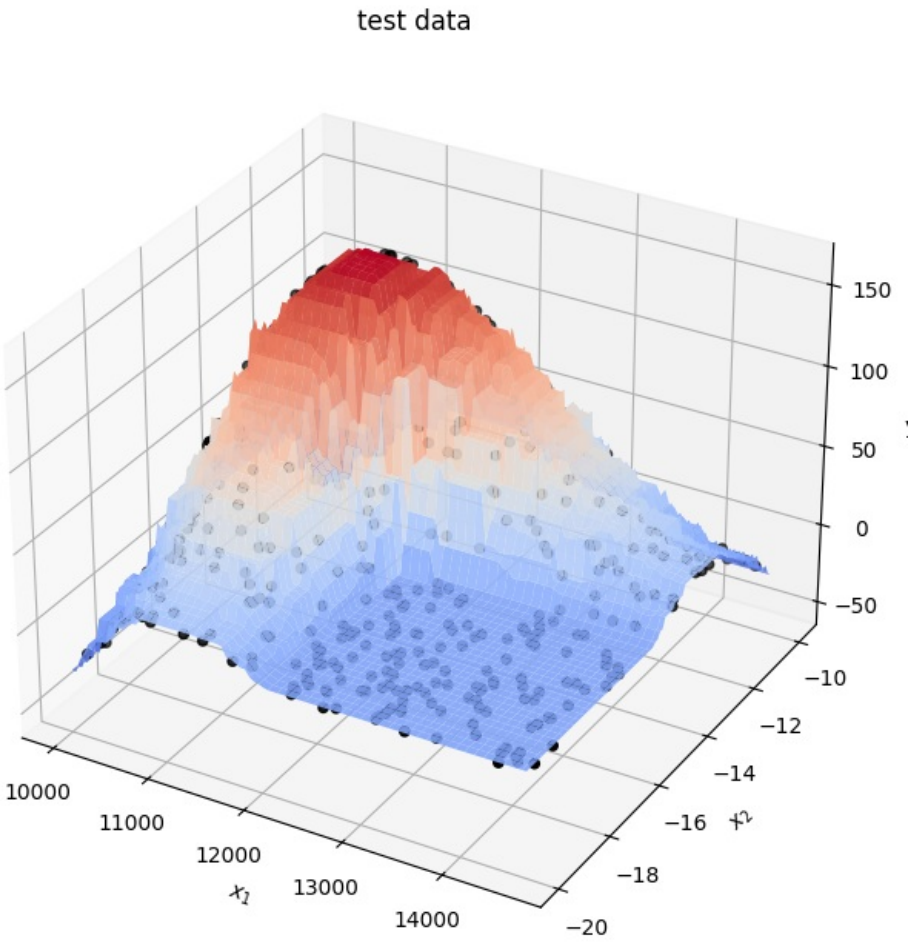

```
Max Depth:  10
RMS error for Decision Tree Regressor on training data:  4.417134916147934
RMS error for Decision Tree Regressor on test data:  7.811633178930412
```


training data
150
125
100
75
50
25
0
−25
−50
10000
11000
12000
13000
14000
x_1
−10
−12
−14
−16
−18
−20
x_2




```
Max Depth:  25
RMS error for Decision Tree Regressor on training data:  0.0
RMS error for Decision Tree Regressor on test data:  5.597397150274175
```


training data
150
125
100
75
50
25
0
−25
−50
10000
11000
12000
13000
14000
x_1
−10
−12
−14
−16
−18
−20
x_2



Question

- Which of your regression trees performed the best on testing data?
  The model with the maximum depth of 25 performed best on the testing data.

Regression trees

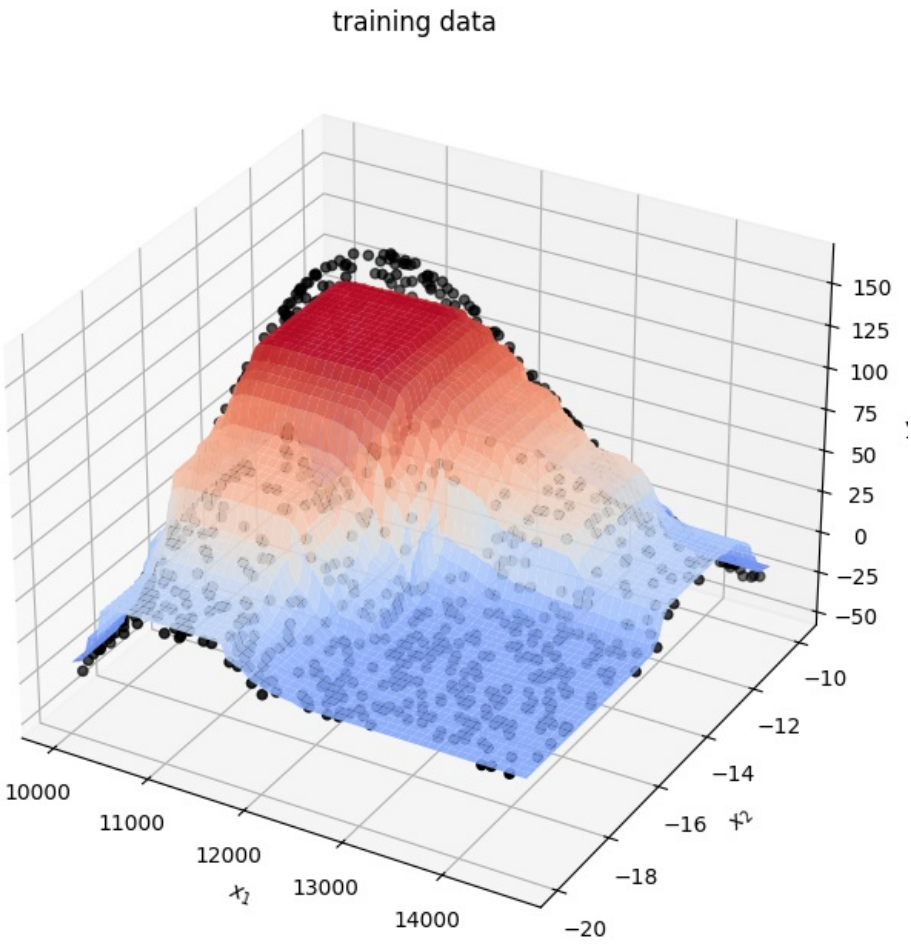
Train 4 random forests in sklearn. For all of them, use the max depth values from your best-performing regression tree. The number of estimators should vary, with values [5, 10, 25, 100].

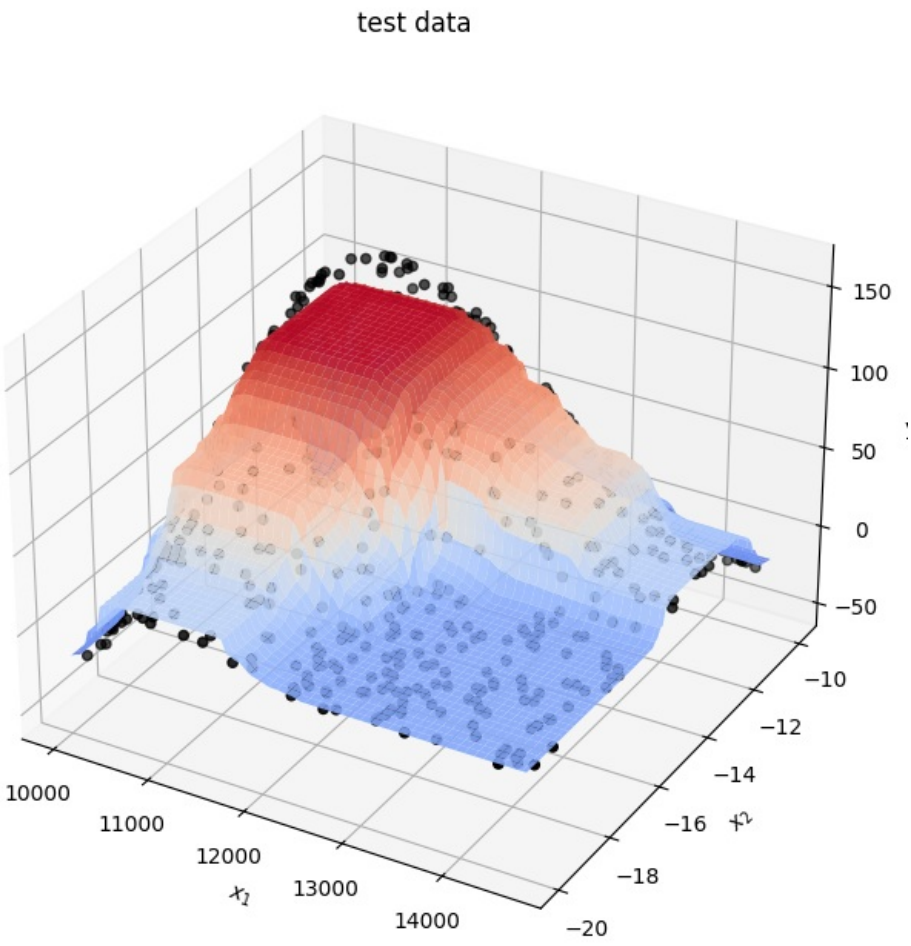
Plot the predictions as a surface plot along with test points. Once again, for each model, compute the train and test RMSE by calling your RMSE function. Print these results.

In [14]:
```
# YOUR CODE GOES HERE
max_depth = [5, 10, 25, 100]

for i in max_depth:
```

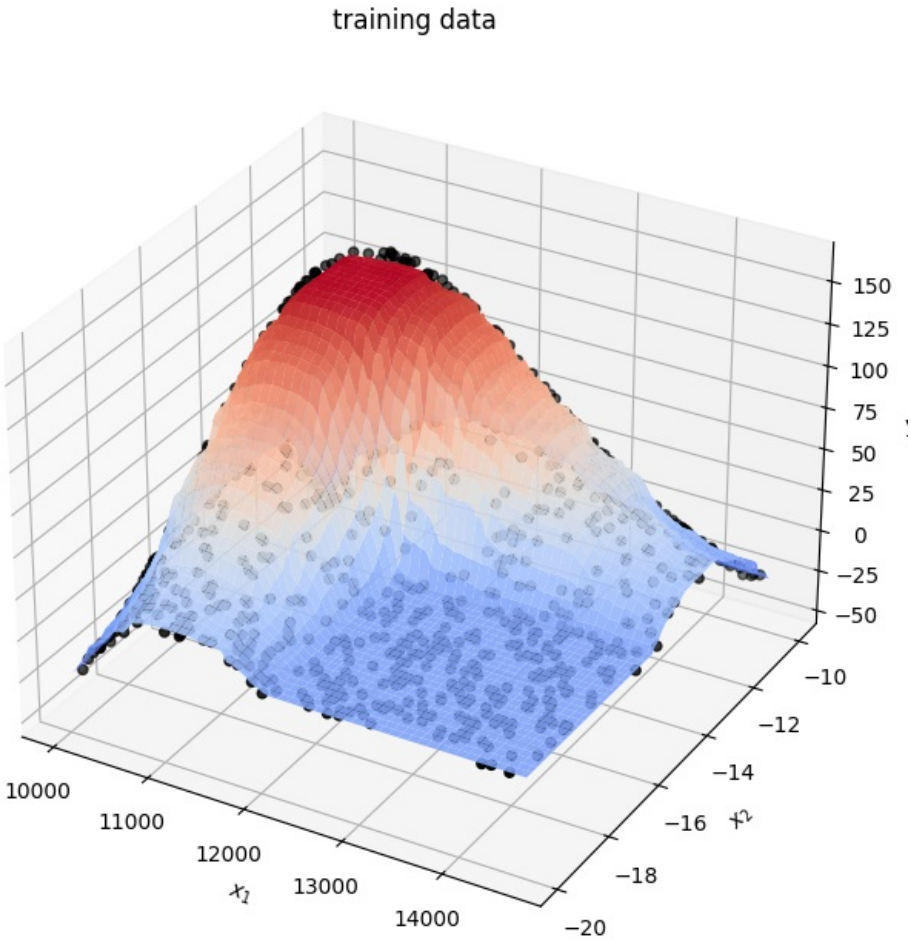
```
    dt = RandomForestRegressor(max_depth = i)
    dt.fit(X_train, y_train)

    pred_train = dt.predict(X_train)
    pred_test = dt.predict(X_test)
    rms_train = RMSE(y_train, pred_train)
    rms_test = RMSE(y_test, pred_test)

    print("\nMax Depth: ", i)
    print("RMS error for Random Forest Regressor on training data: ", rms_train)
    print("RMS error for Random Forest Regressor on test data: ", rms_test)
    make_plot(X_train, y_train, dt, title = "training data")
    make_plot(X_test, y_test, dt, title = "test data")
```

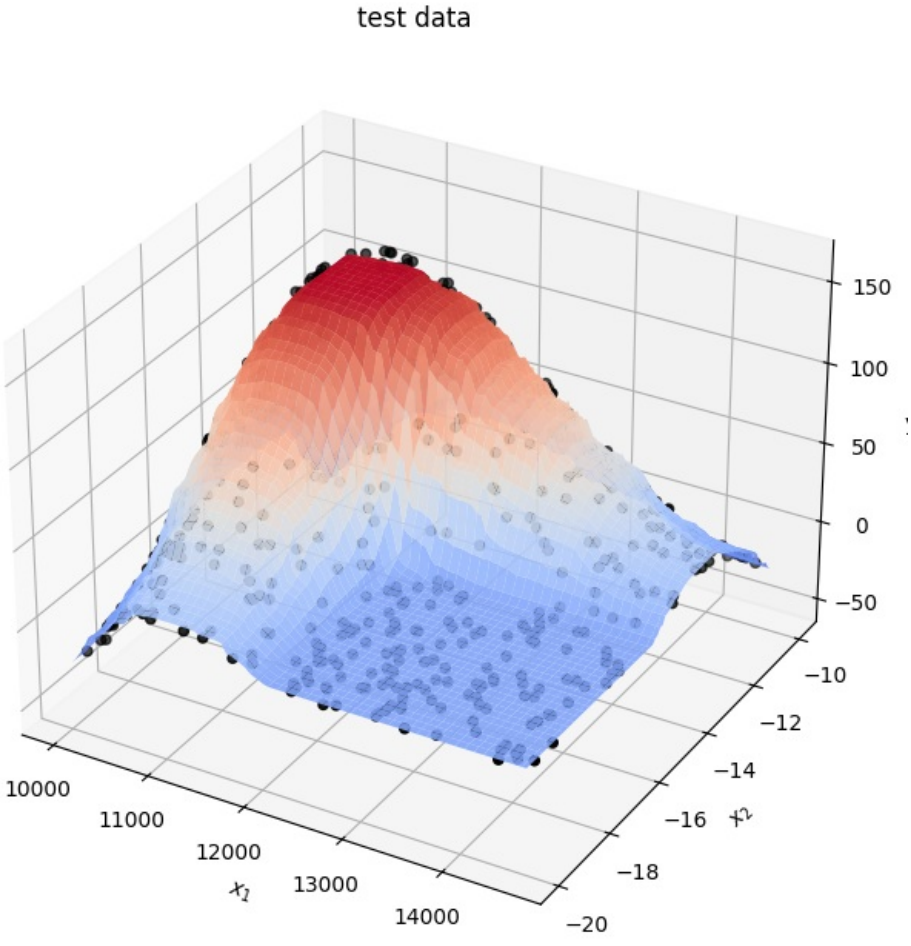
```
Max Depth:  5
RMS error for Random Forest Regressor on training data:  14.870059581561991
RMS error for Random Forest Regressor on test data:  14.751953582837048
```



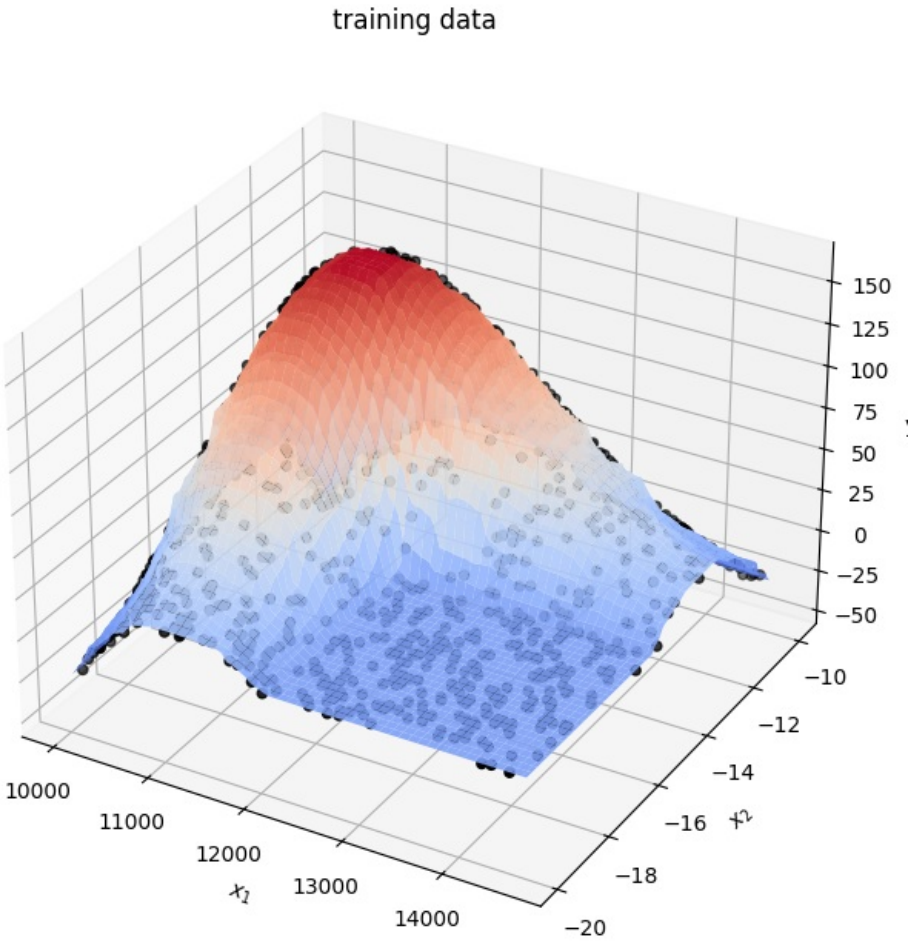


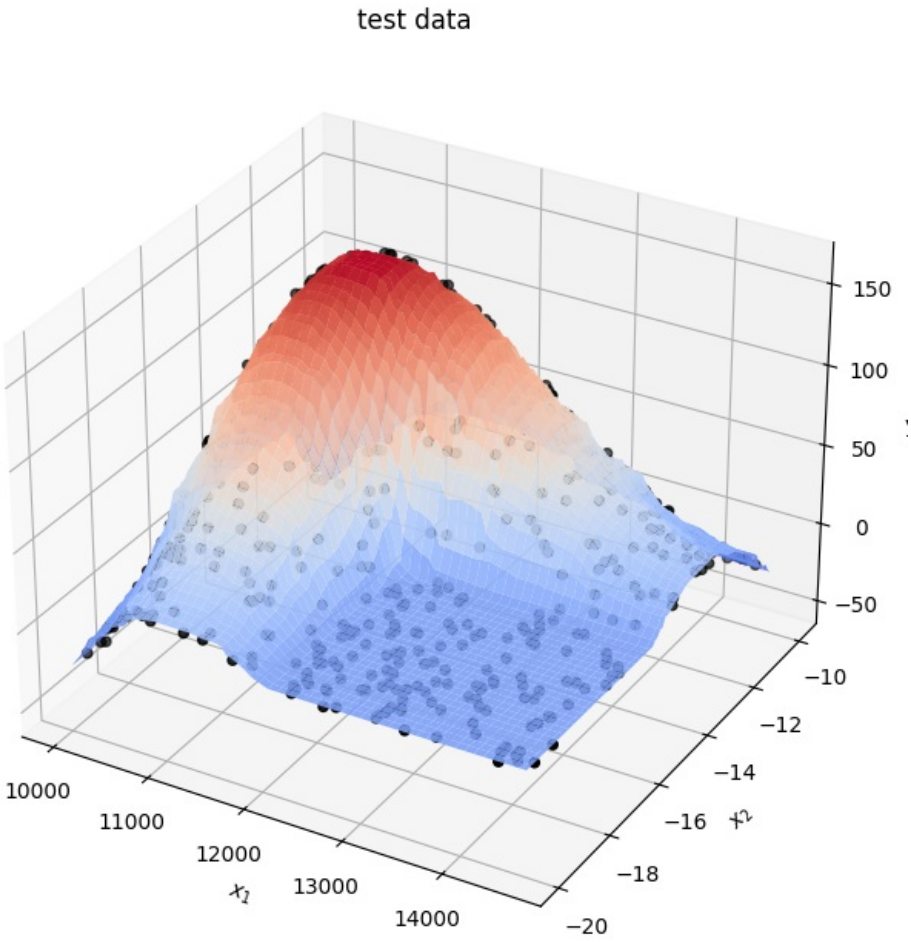

```
Max Depth:  10
RMS error for Random Forest Regressor on training data:  2.946007563532823
RMS error for Random Forest Regressor on test data:  3.980096503464295
```


training data
150
125
100
75
50
25
0
−25
−50
10000
11000
12000
13000
14000
x_1
−10
−12
−14
−16
−18
−20
x_2

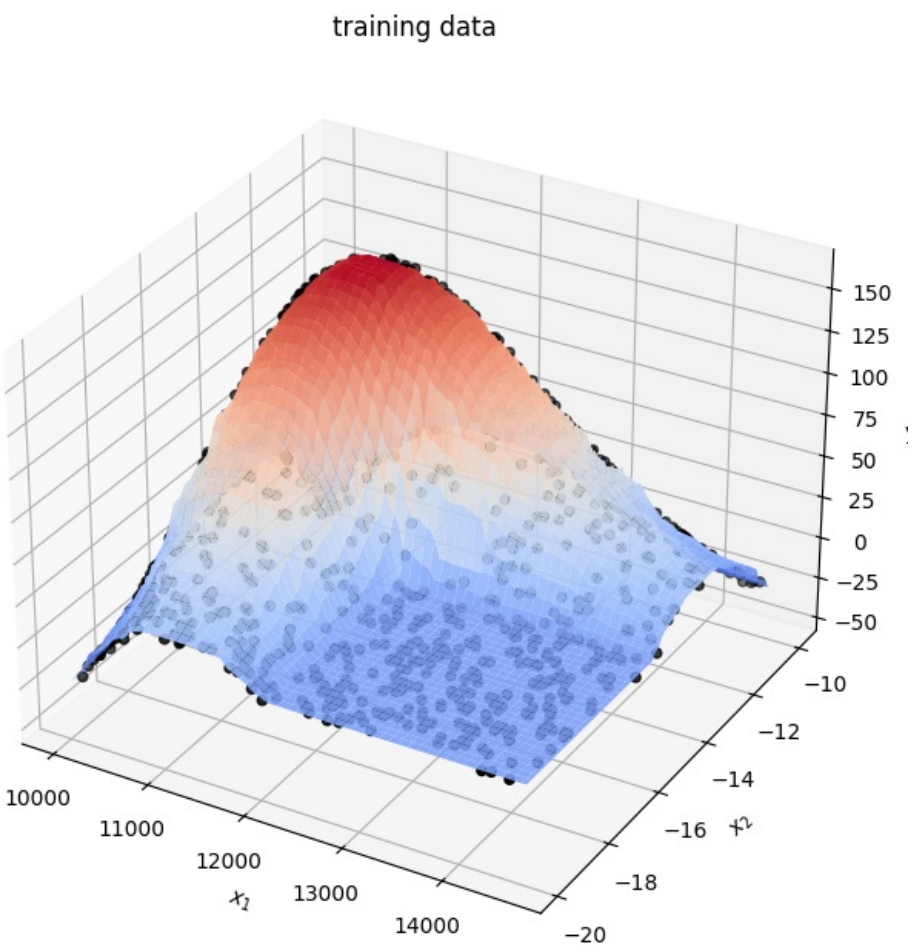


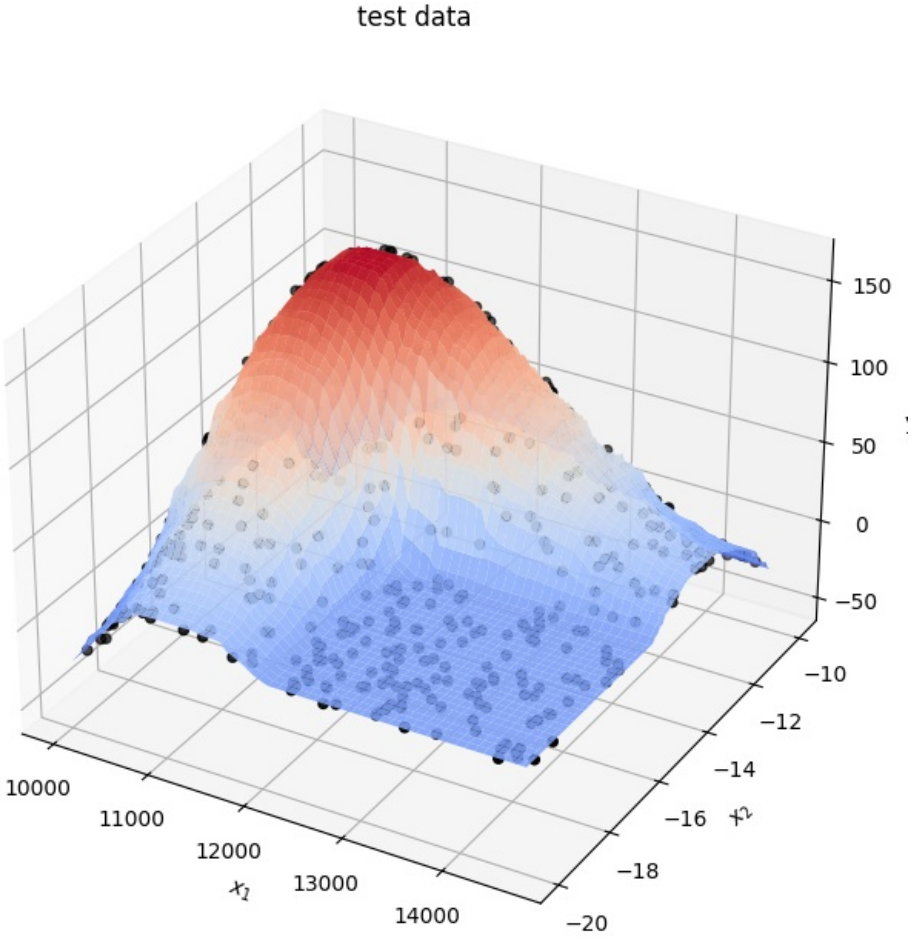

```
Max Depth:  25
RMS error for Random Forest Regressor on training data:  1.3509754829019005
RMS error for Random Forest Regressor on test data:  3.0555287923736825
```


training data
150
125
100
75
50
25
0
−25
−50
10000
11000
12000
13000
14000
x_1
−10
−12
−14
−16
−18
−20
x_2




```
Max Depth:  100
RMS error for Random Forest Regressor on training data:  1.3685013000875361
RMS error for Random Forest Regressor on test data:  2.976915452615326
```


training data
150
125
100
75
50
25
0
−25
−50
10000
11000
12000
13000
14000
x_1
−10
−12
−14
−16
−18
−20
x_2


test data
150
100
50
0
−50
10000
11000
12000
13000
14000
x_1
−10
−12
−14
−16
−18
−20
x_2

Questions

- Which of your random forests performed the best on testing data?
  Again, the model with the highest maximum depth performed the best on the testing data. This was slightly surprising, since I expected that the model with the highest maximum depth might would be overfit to the training data, resulting in sub-optimal performance in the test data.
- How does the random forest prediction surface differ qualitatively from that of the decision tree? The random forest prediction surface is much smoother and without irregularities when compared to that of the decision tree, which is much more discontinuous and jagged. Thanks to the averaging and randomization in the random forest model, leading to it being less sensitive to noise and outliers.

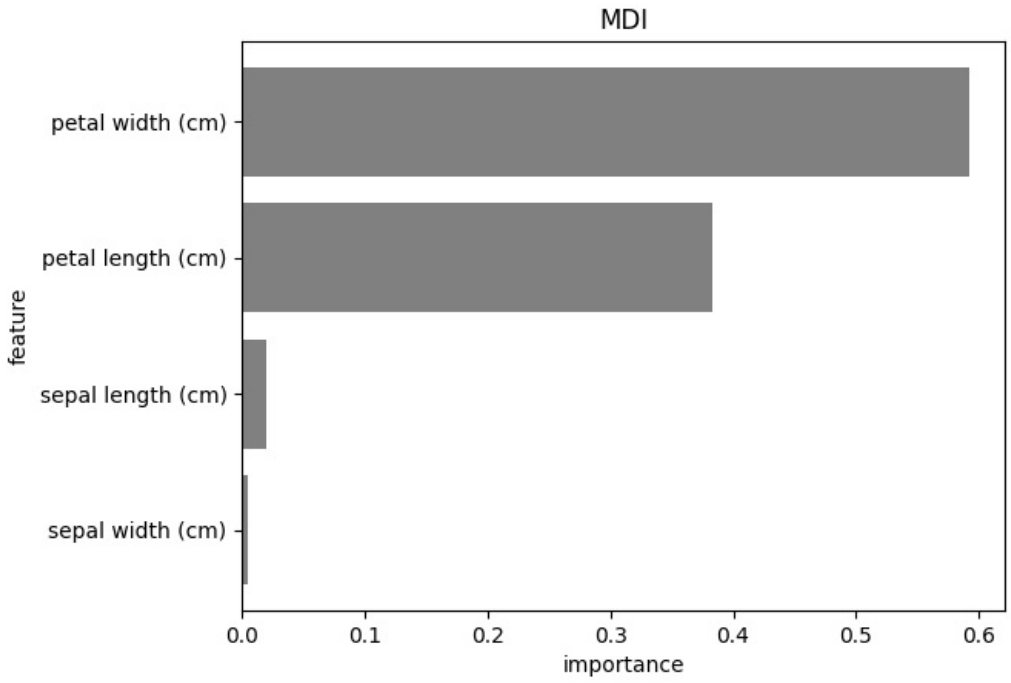
In [51]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.inspection import permutation_importance
import numpy as np
from sklearn.ensemble import RandomForestRegressor
import matplotlib.pyplot as plt

data = load_iris()
X = data.data
y = data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify = y, random_state = 10, test_size = 0.2)
```

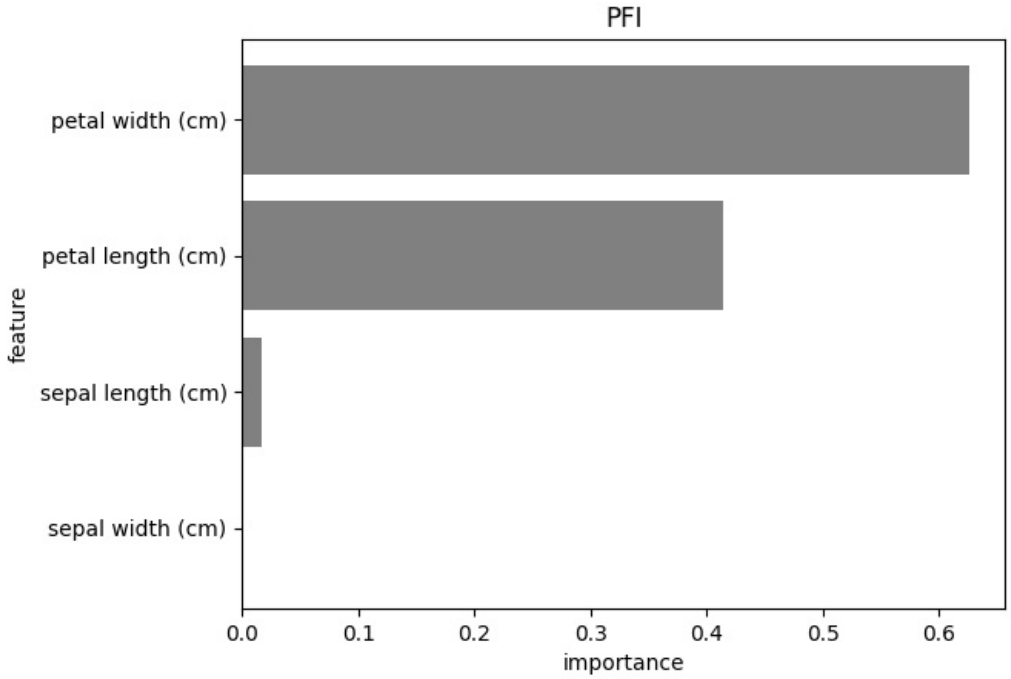
In [52]:
```
model = RandomForestRegressor(n_estimators = 5)
model.fit(X_train, y_train)
```

Out[52]:
```
RandomForestRegressor
RandomForestRegressor(n_estimators=5)
```

In [53]:
```
mdi = model.feature_importances_
indices_mdi = np.argsort(mdi)

pfi_result = permutation_importance(model, X_test, y_test, n_repeats = 10, random_state = 50)
pfi = pfi_result.importances_mean
indices_pfi = np.argsort(pfi)
```

In [54]:
```
labels = data.target_names
features = data.feature_names

plt.barh(np.array(features)[indices_mdi], np.array(mdi)[indices_mdi], color='gray')
plt.title("MDI")
plt.ylabel("feature")
plt.xlabel("importance")
plt.show()

plt.barh(np.array(features)[indices_pfi], np.array(pfi)[indices_pfi], color='gray')
plt.title("PFI")
plt.ylabel("feature")
plt.xlabel("importance")
plt.show()
```



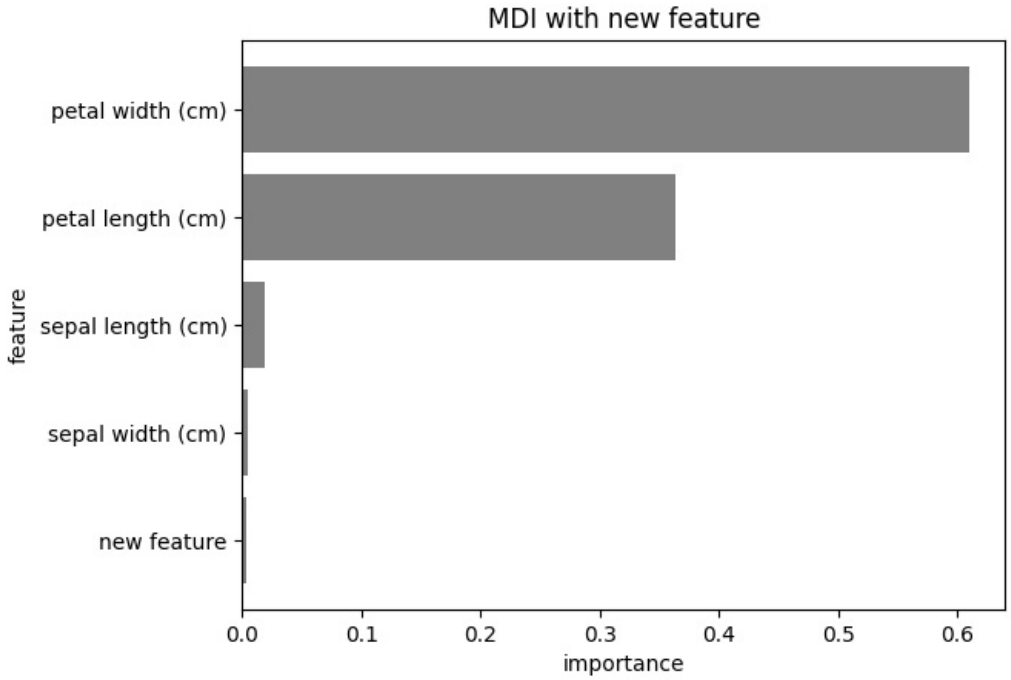


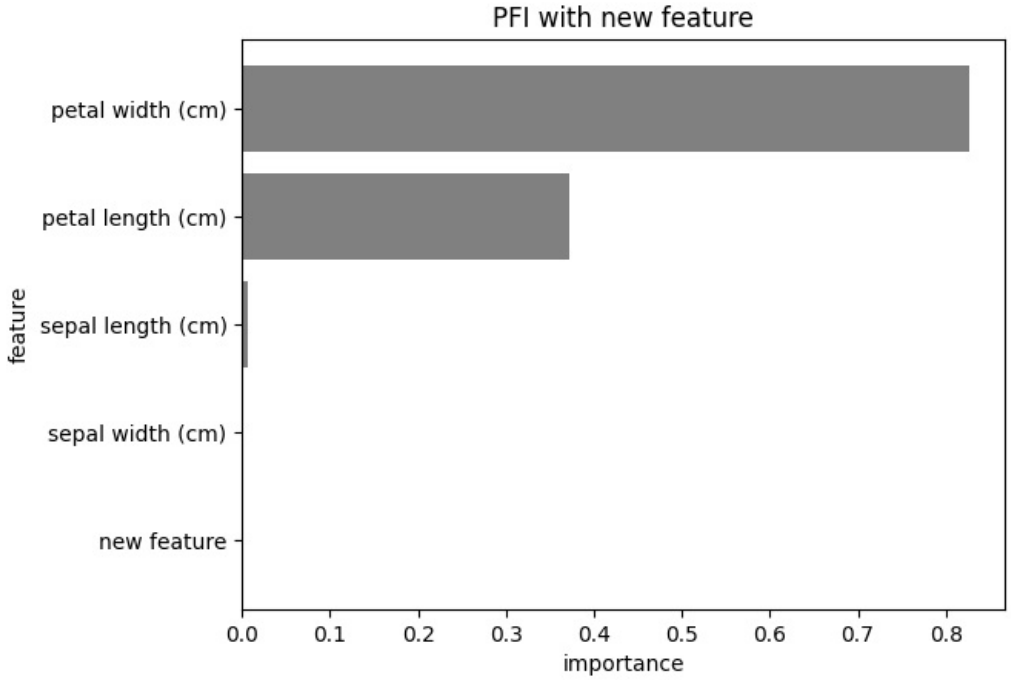
In [55]:
```
rand_feature = np.random.normal(loc = 2, scale = 1, size = (X.shape[0], 1))
X_ex = np.hstack([X, rand_feature])

X_train_ex, X_test_ex, y_train_ex, y_test_ex = train_test_split(X_ex, y, stratify = y, random_state = 10, test_s

model_ = RandomForestRegressor(n_estimators = 5)
model_.fit(X_train_ex, y_train_ex)
```

Out[55]:
```
RandomForestRegressor
RandomForestRegressor(n_estimators=5)
```

In [56]:
```
mdi_ = model_.feature_importances_
indices_mdi_ = np.argsort(mdi_)

pfi_result_ = permutation_importance(model_, X_test_ex, y_test_ex, n_repeats = 10, random_state = 50)
pfi_ = pfi_result_.importances_mean
indices_pfi_ = np.argsort(pfi_)
```

In [57]:
```
features_ = features
features_.append("new feature")

plt.barh(np.array(features_)[indices_mdi_], np.array(mdi_)[indices_mdi_], color='gray')
plt.title("MDI with new feature")
plt.ylabel("feature")
plt.xlabel("importance")
plt.show()

plt.barh(np.array(features_)[indices_pfi_], np.array(pfi_)[indices_pfi_], color='gray')
plt.title("PFI with new feature")
plt.ylabel("feature")
plt.xlabel("importance")
plt.show()
```





In [58]:
```
X_exx = np.hstack([X_ex, np.ones([X.shape[0], 1])])

X_train_exx, X_test_exx, y_train_exx, y_test_exx = train_test_split(X_exx, y, stratify = y, random_state = 10,

model_1 = RandomForestRegressor(n_estimators = 5)
model_1.fit(X_train_exx, y_train_exx)
```

Out[58]:

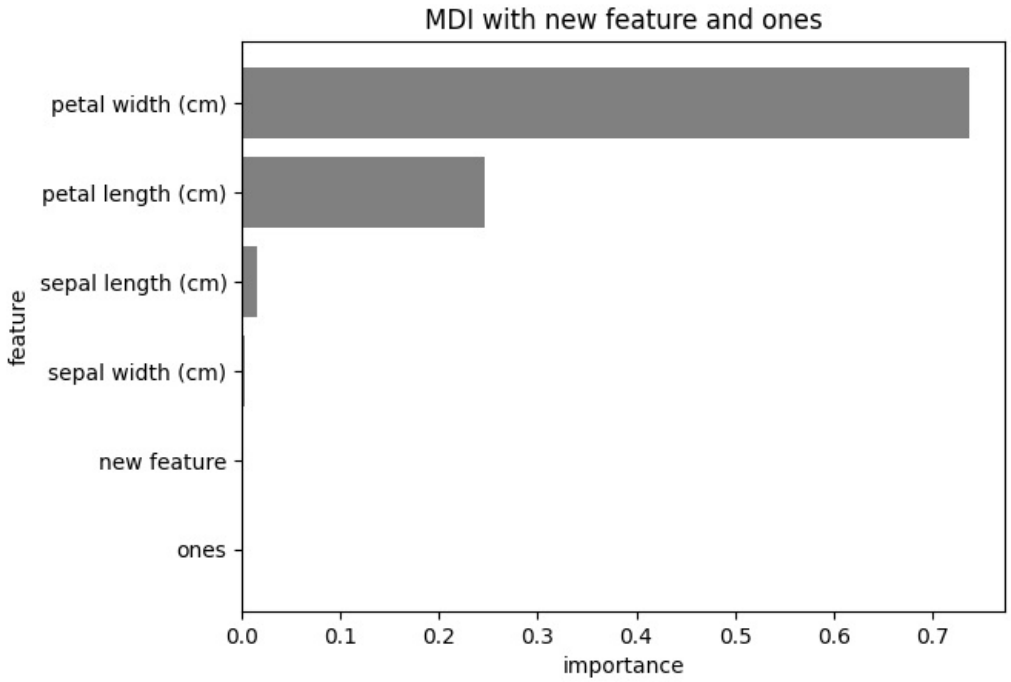
RandomForestRegressor

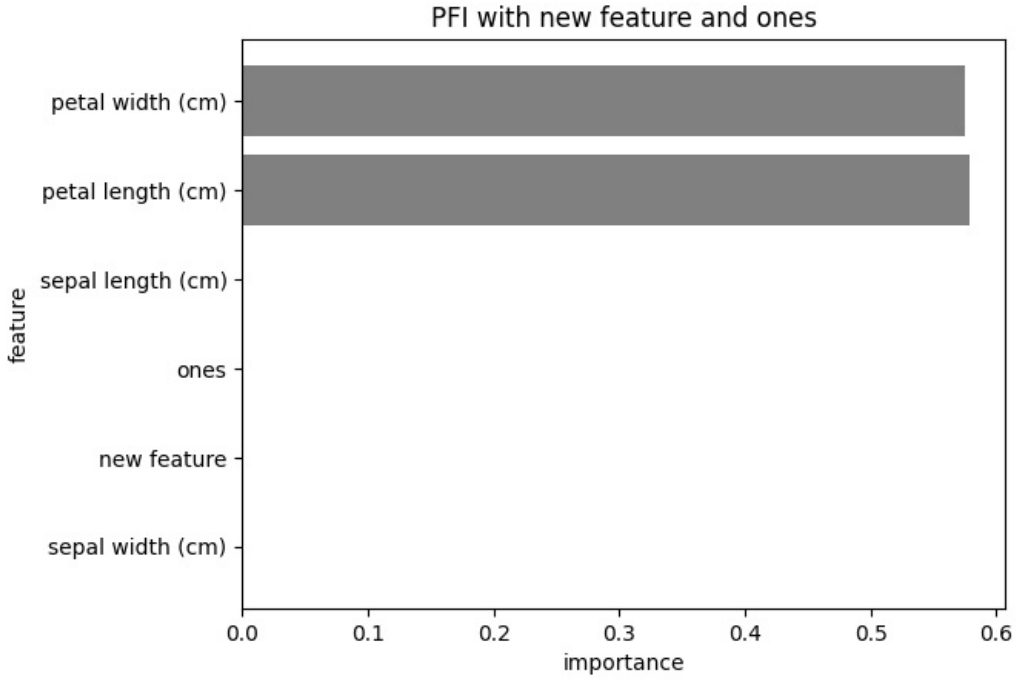
RandomForestRegressor(n_estimators=5)

In [59]:
```
mdi_1 = model_1.feature_importances_
indices_mdi_1 = np.argsort(mdi_1)

pfi_result_1 = permutation_importance(model_1, X_test_exx, y_test_exx, n_repeats = 10, random_state = 50)
pfi_1 = pfi_result_1.importances_mean
indices_pfi_1 = np.argsort(pfi_1)
```

In [60]:
```
features_1 = features_
features_1.append("ones")

plt.barh(np.array(features_1)[indices_mdi_1], np.array(mdi_1)[indices_mdi_1], color = 'gray')
plt.title("MDI with new feature and ones")
plt.ylabel("feature")
plt.xlabel("importance")
plt.show()
```

```
plt.barh(np.array(features_1)[indices_pfi_1], np.array(pfi__)[indices_pfi_1], color = 'gray')
plt.title("PFI with new feature and ones")
plt.ylabel("feature")
plt.xlabel("importance")
plt.show()
```





Conclusions

1. MDI might rank random or irrelevant features as more important than they actually are because it depends on how frequently a feature is used to split the data, not whether it improves the model's predictive performance.
2. PFI is a more robust measure for identifying features that really contribute to the model's performance, as it evaluates how sensitive the model's accuracy is to shuffling the feature values.
3. The constant feature is correctly identified by both MDI and PFI as unimportant, but MDI can be tricked into ranking random features higher than they deserve.