Pattern Recognition & Machine Learning
Lab-2 Assignment

Name: Tanish Pagaria
Roll No.: B21AI040

Question-1

Task-1

Preprocessing the data:

Basic preprocessing and data analysis were performed on the dataset. There were no NaN or NULL values.

```
df.isnull().sum()

X1    0
X2    0
X3    0
X4    0
X5    0
X6    0
X7    0
X8    0
Y1    0
dtype: int64
```

The features in the data frame had values that were in different ranges from each other, so the data needed to be normalized to make sure they were all spread out the same way.

We used `sklearn.preprocessing.StandardScaler` to normalize the data. It takes away the average of the values that have been measured and sets the standard deviation to 1.

Splitting the data:

To split the data in the ratio 70:10:20 that represents training:validation:testing, we used `sklearn.model_selection.train_test_split` twice, first dividing the data into a 70:30 ratio and then splitting the 30 into a 10:20 ratio.

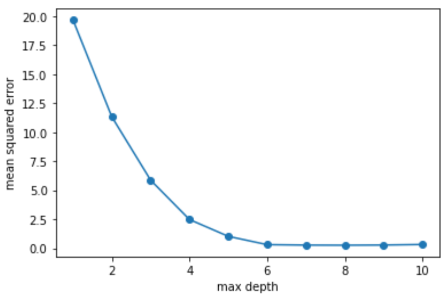
Task-2

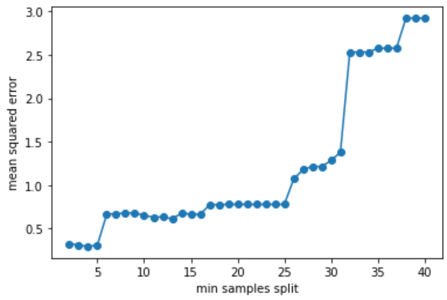
Function to train data using a regression decision tree and vary hyperparameters for best generalization:

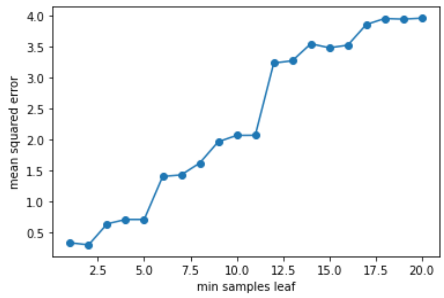
We used `sklearn.tree.DecisionTreeRegressor` to create a decision tree regressor model and train it. The hyper-parameters to be varied included `max_depth`, `min_samples_split`, and `min_samples_leaf`.

The function took the hyper-parameters as arguments to be passed to itself. It returned the mean squared error (MSE) value based on the arguments after checking the performance on the validation set (using `sklearn.metrics.mean_squared_error`).

We used the MSE values to plot graphs of the validation MSE.







(Variation of MSE with hyper-params)

By running nested for-loops, analyzing graphs, and keeping track of the minimum MSE in each case, `max_depth = 8`, `min_samples_split = 4`, and `min_samples_leaf = 1` were found to have the lowest MSE.

Task-3

Cross Validation

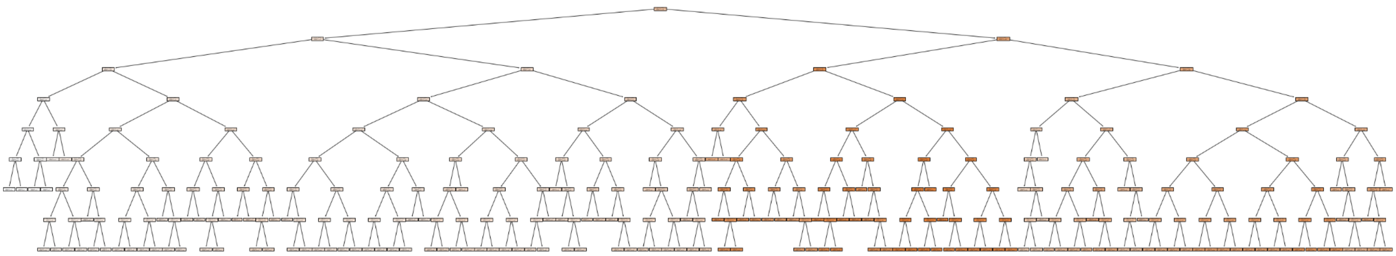
By using `sklearn.model_selection.cross_val_score`, `sklearn.model_selection.KFold`, `sklearn.model_selection.RepeatedKFold`, cross-validation was performed using the optimal hyper-parameters obtained previously.
The results of the cross-validation are shown below.

- Hold-out Cross Validation: *0.9967314632479397*
- 5-Fold Cross Validation: *0.964484164972087*
- Repeated 5-Fold Cross Validation: *0.9970735255151203*

The mean squared error between the predicted and ground-truth values in the test data of the best model was 0.24173506673554462 (calculated previously while hyper-parameter tuning).

Plotting the Decision Tree

The Decision Tree was plotted directly using the `sklearn.tree.plot_tree()` method.



Task-4

L1: Absolute Error Loss
L2: Squared Error Loss

The following accuracy scores were obtained when using L1 loss and L2 loss as two different criteria for the split in the Decision Tree Regressor model, as shown below.

```
model_using_criterion(train_X, train_y, test_X, test_y, criterion="absolute_error", depth=8, split=4, leaf=1)
```

```
0.9968784329885061
```

```
model_using_criterion(train_X, train_y, test_X, test_y, criterion="squared_error", depth=8, split=4, leaf=1)
```

```
0.9967910147197749
```

Question-2 (Classification)

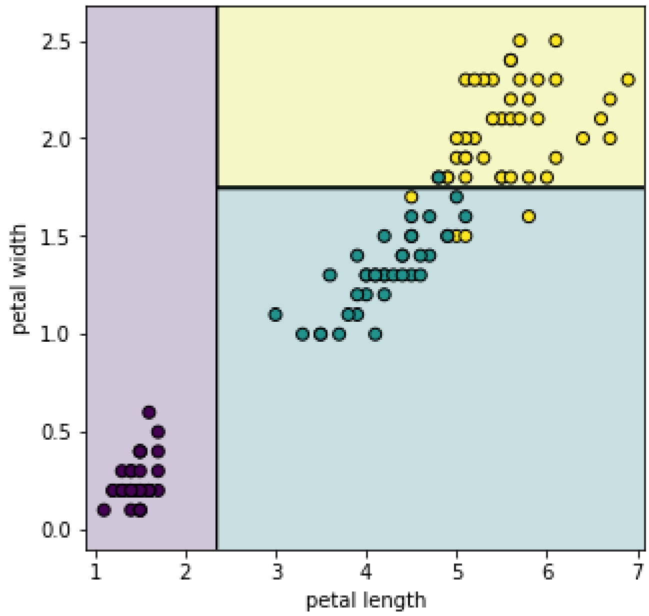
Task-1

Initially, basic data analysis of the Iris dataset was performed. It was noted that the dataset did not have any NaN or NULL values.

The dataset was preprocessed by encoding the categorical variables. The features used were "petal length" and "petal width" hence the other features were removed. The dataset was then split into training and testing in the ratio 80:20. A Decision Tree classifier was trained on the pre-processed dataset (taking the max_depth as 2).

The following function was implemented to plot the decision boundary, as shown below.

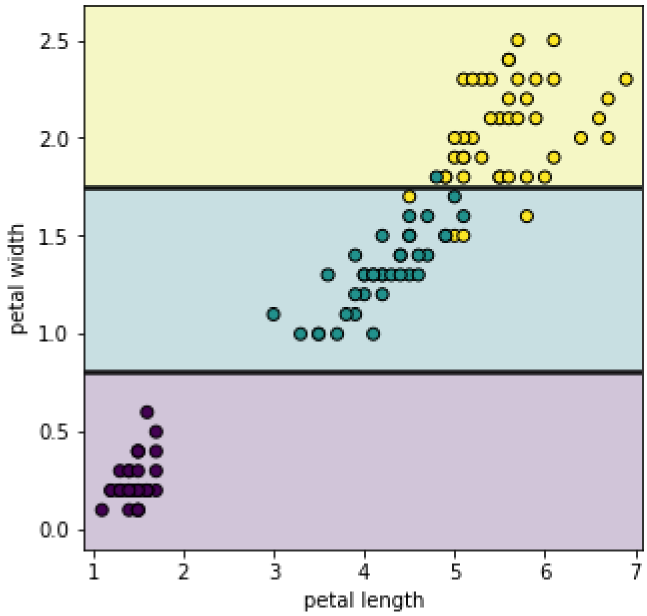
```
def plot_decision_boundary(classifier, train_X, train_y, feature1, feature2):
    h = 0.02

    xf1 = train_X[feature1].to_numpy()
    xf2 = train_X[feature2].to_numpy()
    train_y = train_y.to_numpy()

    x_min, x_max = xf1.min() - 10*h, xf1.max() + 10*h
    y_min, y_max = xf2.min() - 10*h, xf2.max() + 10*h
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = classifier.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.figure(figsize=(5,5))
    plt.contourf(xx, yy, Z, alpha=0.25)
    plt.contour(xx, yy, Z, colors='k', linewidths=0.7)

    plt.scatter(xf1, xf2, c=train_y, edgecolors='k')
    plt.xlabel(feature1)
    plt.ylabel(feature2)
    plt.show()
```



Decision boundary for the iris-dataset Decision Tree Classifier

Task-2

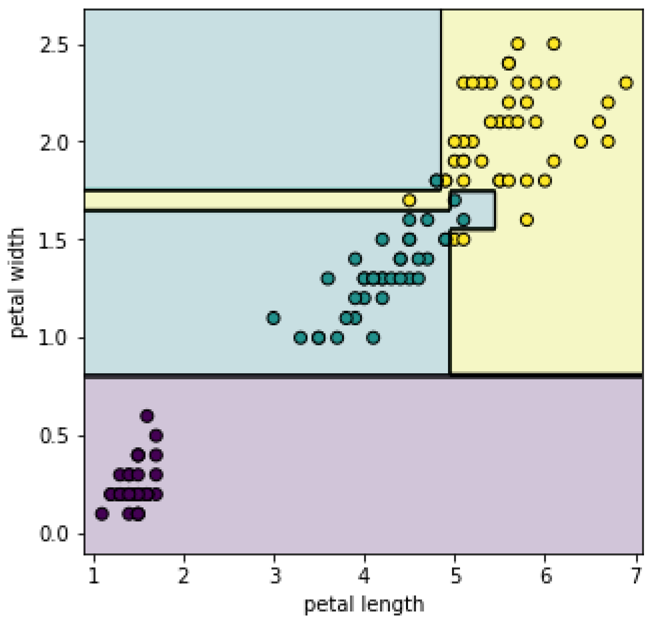
The widest Iris-Versicolor from the iris training set (the one with petals 4.8 cm long and 1.8 cm wide) was removed, and a new Decision Tree classifier model (with max_depth = 2) was trained. The decision boundary was plotted similarly.



Decision boundary for the Iris-Dataset Decision Tree Classifier
(after removing the widest Iris-Versicolor from the training set)

There is a change in the decision boundary plots after removing the widest Iris-Versicolor from the training set.

Task-3
A new Decision Tree classifier (with max_depth = None) was trained on the preprocessed dataset, and the decision boundary was plotted similarly using the previously defined function.



Decision boundary for the Iris-Dataset Decision Tree Classifier
(for max_depth=None)

<u>Comparison and Analysis of the two decision boundaries</u>
From the two decision boundaries, we can clearly see that when the max_depth hyper-paramter is not defined, the model becomes overfit (as clearly evident from the plots).
We also observe that one of the classes in the dataset is linearly separable from the others while the rest of them are not.

Task-4

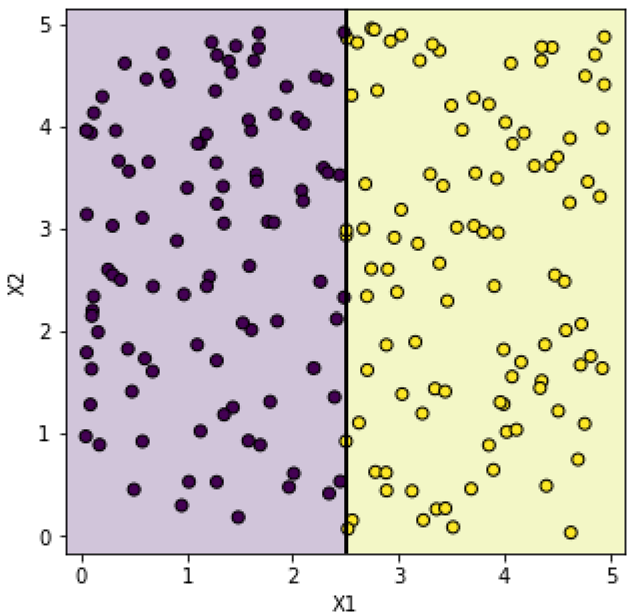
We created a random dataset having two attributes, X1 and X2, and two classes, y = 0 and y = 1. X1 and X2 are randomly sampled from the range (0, 5).

y = 0 when X1 < 2.5, and

y = 1 when X1 > 2.5.

The dataset had 100 data points for both classes. This was accomplished through the use of the random module and the creation of a Pandas dataframe.

A Decision Tree classifier model was trained on the given dataset, and the decision boundary was plotted using the previously defined function.



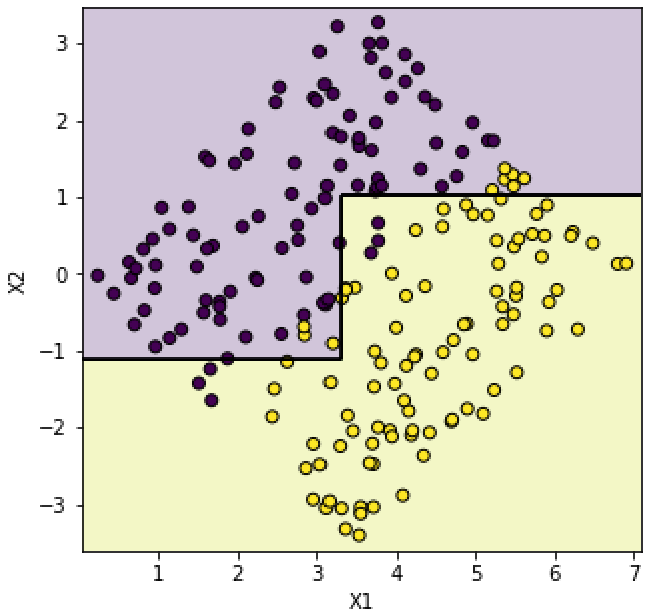
Decision Boundary for the Decision Tree Classifier for the random points dataset

To rotate the data points by 45°, we multiply them by rotation matrix and obtain the modified data points.

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Rotation Matrix

Another Decision Tree classifier was based on the rotated data points, and the decision boundary was plotted.



Decision Boundary for the Decision Tree Classifier for the random points dataset (after rotation of the data points by 45° about the origin)

We see that the decision boundary plots also get changed when the dataset points were rotated. Also, the data points were linearly separable initially. However, after the rotation, they are not (alsom evident from the decision boundary plots).

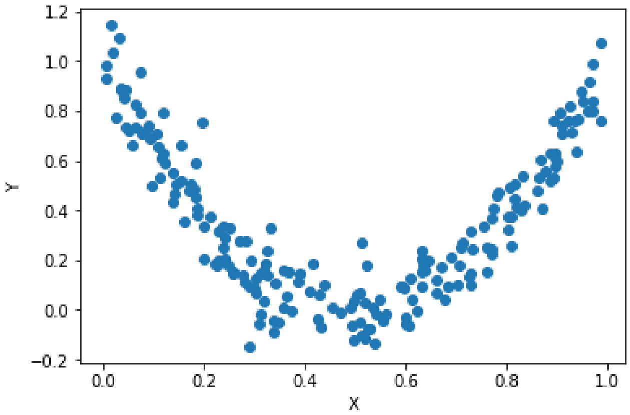
Task-5

The decision trees generally become overfitting when the max depth is increased significantly.

Question-2 (Regression)

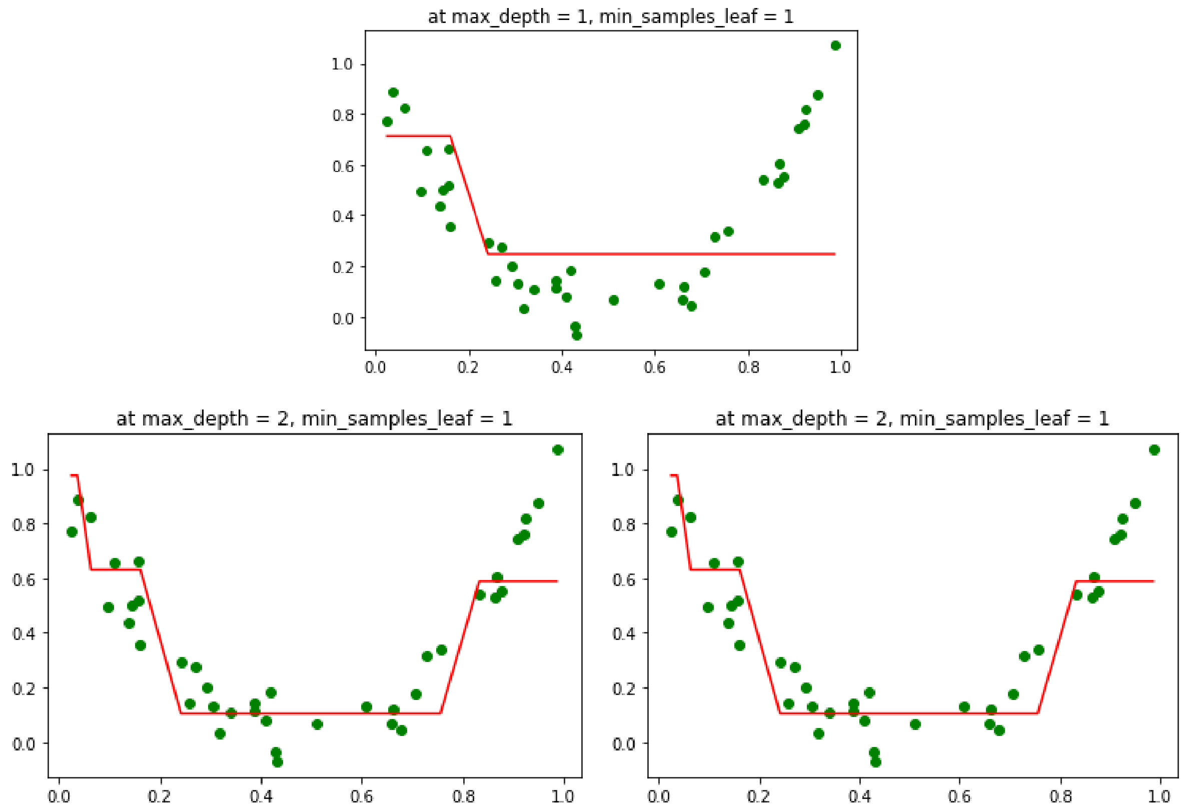
Task-1

We trained two decision tree models, one with max_depth = 2 and another with max_depth = 3.



Original distribution of the datapoints

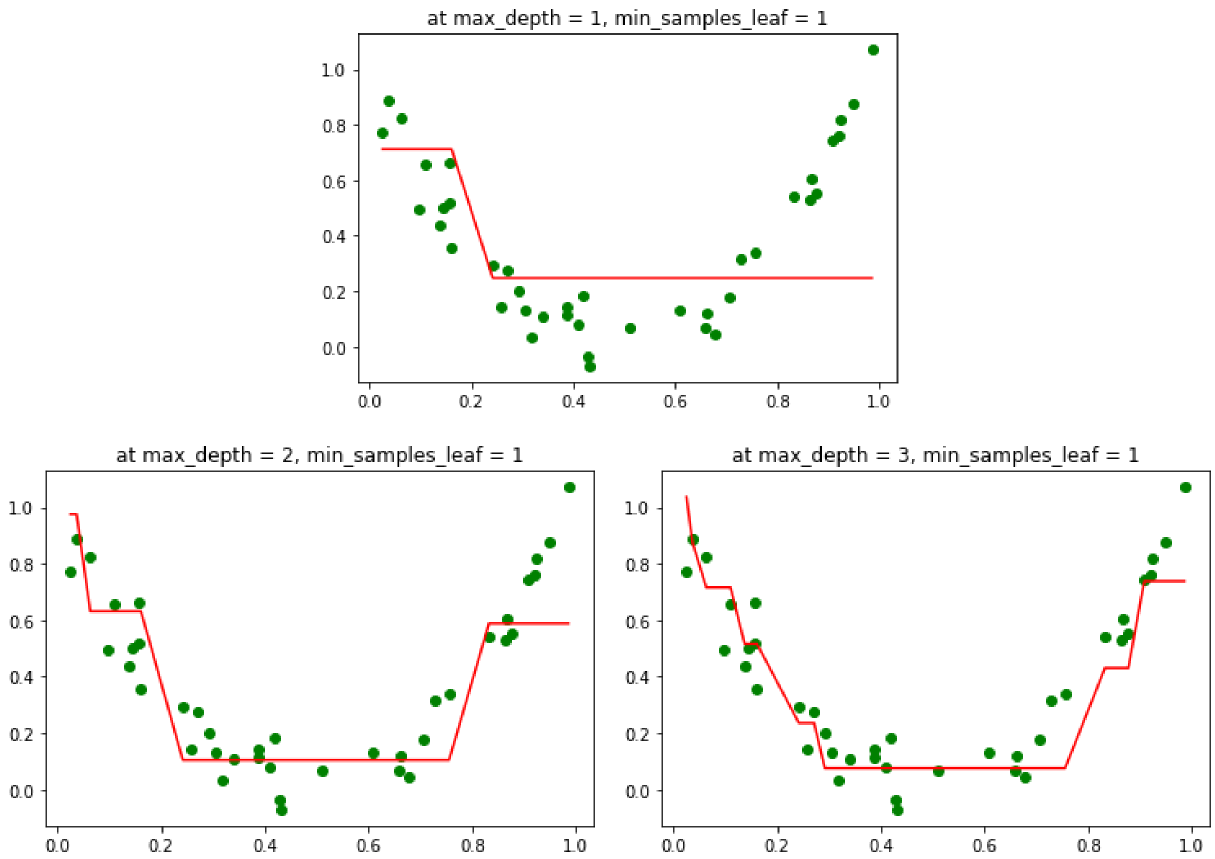
The regression predictions of the models were plotted below using a line plot along with the scatter plot of the data points of the original dataset, as shown below.



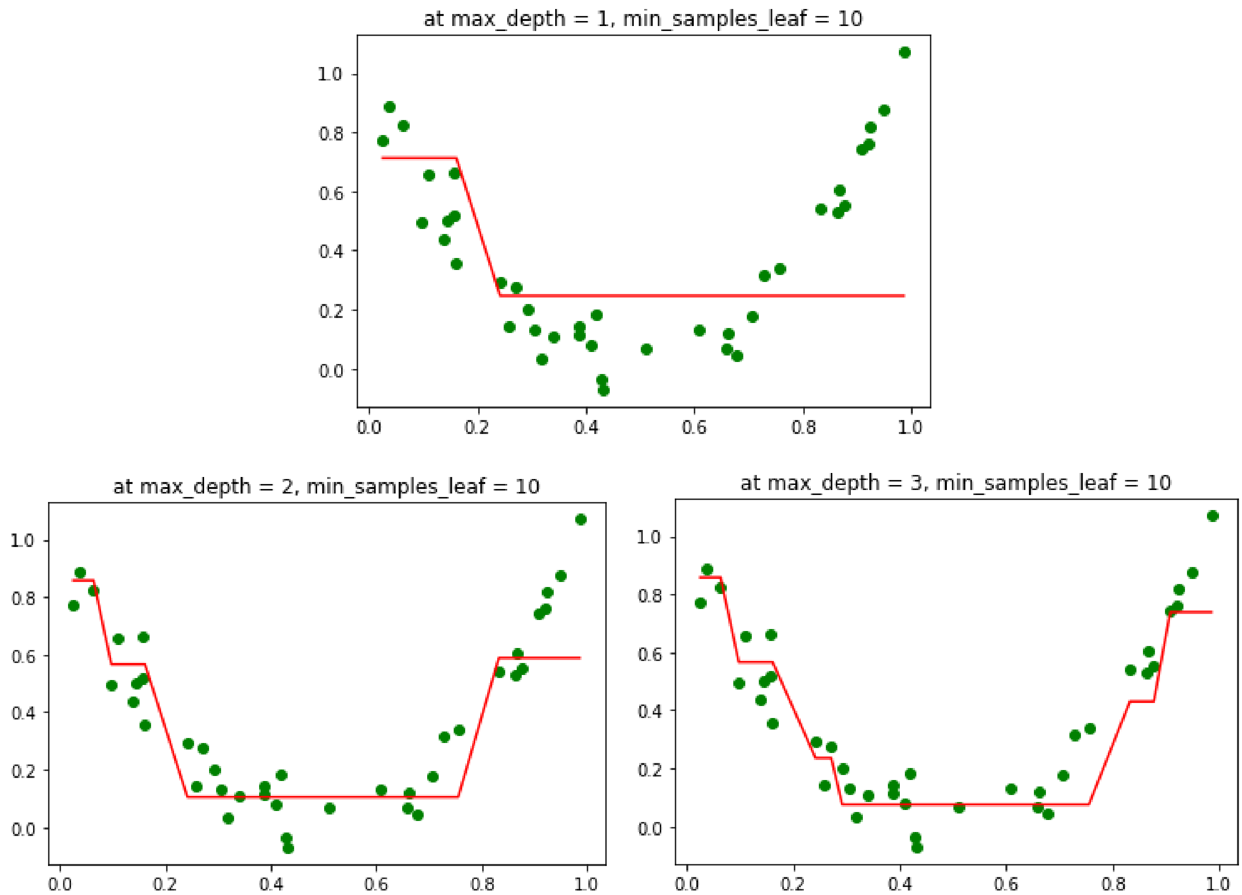
Regression plots (line graph) along with the original data points (scatter plot)

Task-2

The regression plots to show the decision tree fits on the dataset in two cases: min_samples_leaf = 0 and min_samples_leaf = 10 were plotted, as shown below.



(for min_split_leaf = 1)



(for min_split_leaf = 10)

We observe that the decision tree models become overfitting on increasing the maximum depth and also on decreasing the minimum sample leafs.

Question-3

Task-1

Preprocessing the data included the following steps:-

→ Checking the number of NaN values in each feature (performed using pandas).

```
species               0
island                0
bill_length_mm        2
bill_depth_mm         2
flipper_length_mm     2
body_mass_g           2
sex                  11
year                  0
```

Since the number of these values was significantly less, the rows containing the NaN values were dropped.

→ Categorising the features into categorical and non-categorical.

```
Number of unique entries in species : 3
Number of unique entries in island : 3
Number of unique entries in bill_length_mm : 163
Number of unique entries in bill_depth_mm : 79
Number of unique entries in flipper_length_mm : 54
Number of unique entries in body_mass_g : 93
Number of unique entries in sex : 2
Number of unique entries in year : 3
```

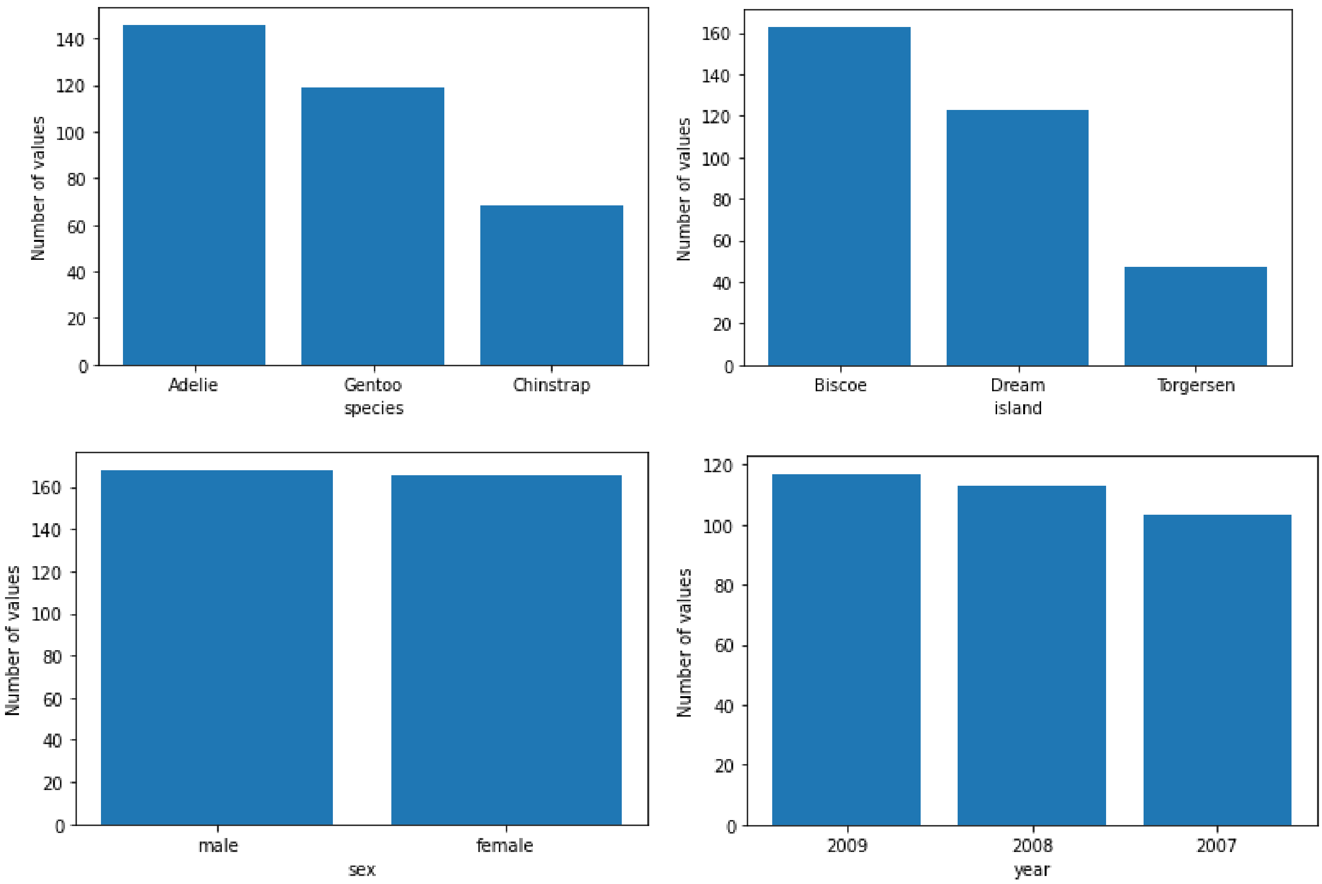
```python
categorical_features = ['species', 'island', 'sex', 'year']
non_categorical_features = ['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm', 'body_mass_g']
```

→ Encoding the categorical features.

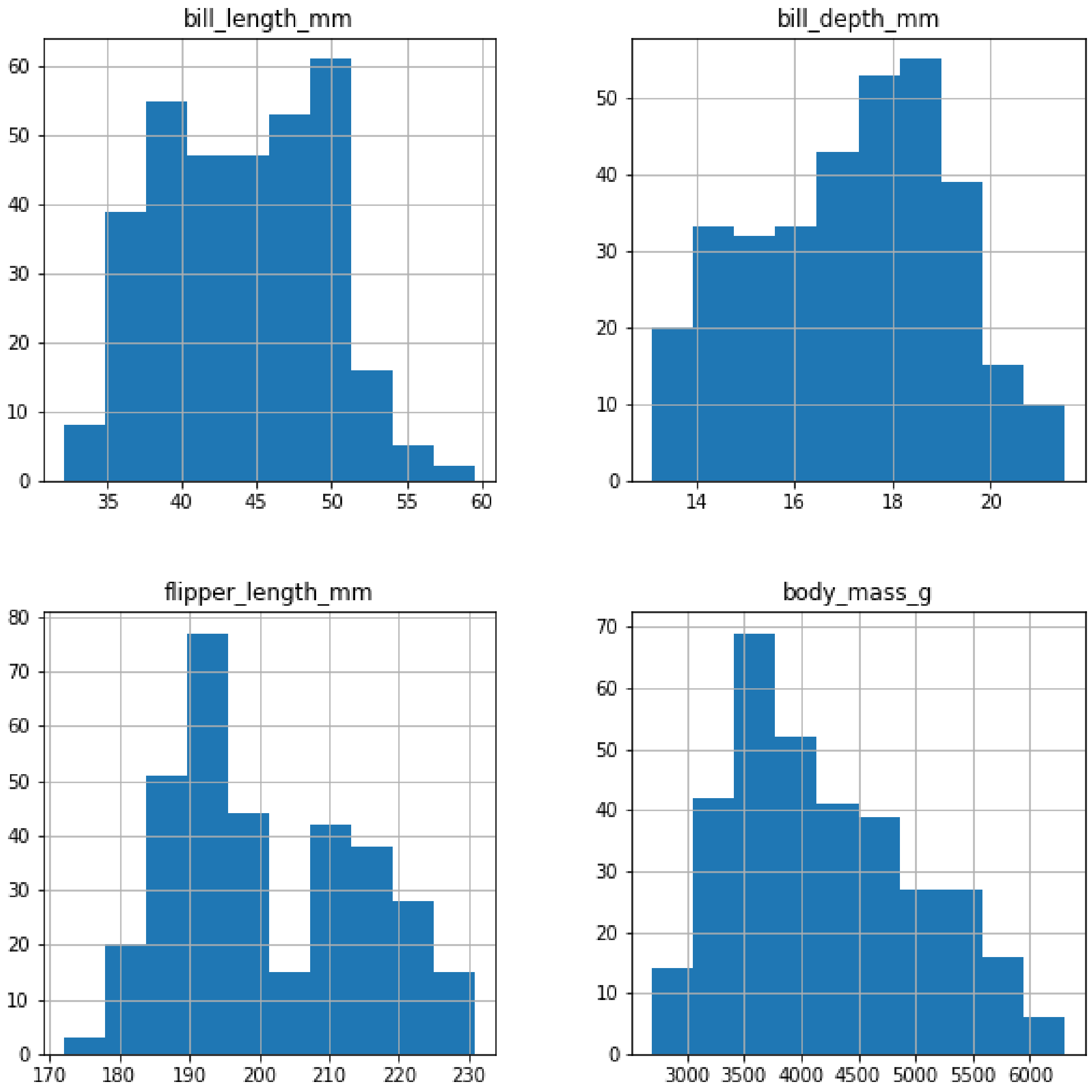
→ Since all the non-categorical features had varied ranges, they were normalized by a custom normalization function (which worked the same as `sklearn`'s `StandardScaler`).

```python
def normalize(df, feature):
    f_mean = df[feature].mean()
    f_std = df[feature].std()
    df[feature] = (df[feature]-f_mean)/f_std
    return df
```

Visualization (using bar plots and histograms)



Categorical Features



Non-categorical Features

Task-2

Implementing the entropy loss function

$$Entropy(S) = \sum_{i=1}^{c} -p_i \log_2 p_i$$

Using the above formula, we implemented the entropy cost function to be used in finding the splits. We also defined Information Gain function to be used in the Decision Tree Classifier implementation.

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

Task-3 & Task-4

<u>Implementing a decision function to make the most optimal split</u>

The following function cont_to_cat() was implemented in order to make the most optimal split of the dataset while Decision Tree implementation and training.

```python
def cont_to_cat(train_X, train_y):
    '''
    Decision Function that returns the best possible split
    (Continuous variables to Continuous variables conversion)
    '''
    split = {}
    info_gain = -1

    features = train_X.columns.tolist()
    for Feature in features:
        possible_thresholds = train_X[Feature].unique()

        for Threshold in possible_thresholds:
            # left branch
            left_branch_X = train_X[train_X[Feature] <= Threshold]
            left_branch_y = train_y[train_X[Feature] <= Threshold]

            # right branch
            right_branch_X = train_X[train_X[Feature] > Threshold]
            right_branch_y = train_y[train_X[Feature] > Threshold]

            if len(left_branch_X > 0) and len(right_branch_X > 0):
                current_info_gain = information_gain(train_y, left_branch_y, right_branch_y)

                if current_info_gain > info_gain:
                    split['feature'] = Feature
                    split['threshold'] = Threshold
                    split['left_branch_X'] = left_branch_X
                    split['left_branch_y'] = left_branch_y
                    split['right_branch_X'] = right_branch_X
                    split['right_branch_y'] = right_branch_y
                    split['info_gain'] = current_info_gain
                    info_gain = current_info_gain

    return split
```

cont_to_cat() function implementation

The function also deals with getting the attribute that leads to the best split, and making the split.

Task-5

Decision Tree Classifier Implementation

First, we defined a constructor function for each node of the Decision Tree. Then, a recursive function was implemented for the creation of the tree using the function we defined above ensuring optimal split at each point.

```python
def Node(feature=None, left=None, right=None, threshold=None, value=None):
    '''
    Constructor for each node of the Decision Tree
    '''
    return {'feature': feature, 'left': left, 'right': right, 'threshold': threshold, 'value': value}
```

```python
def tree(train_X, train_y, current_depth=0, max_depth=2, min_sample_split=2):
    '''
    Recursive function that creates the Decision Tree
    '''
    num_datapoints = len(train_X)

    if (num_datapoints >= min_sample_split) and (current_depth <= max_depth):
        split = cont_to_cat(train_X, train_y)

        if split['info_gain'] > 0:
            left_branch = tree(split['left_branch_X'], split['left_branch_y'], current_depth+1)
            right_branch = tree(split['right_branch_X'], split['right_branch_y'], current_depth+1)

            return Node(feature=split['feature'], left=left_branch, right=right_branch, threshold=split['threshold'])

    # in case of leaf node
    leaf_value = max(train_y)
    return Node(value=leaf_value)
```

Functions for Decision Tree Creation

Further, we implemented the function for training the decision tree based on the training dataset. The constructor functions for the tree as well as the train function had the parameters to ensure that the algorithm would self-identify when there is no information gain being done, i.e. the
model has plateaued in its training and would not grow any further.

The `max_depth` parameter was defined, which dealt with the depth after which the tree would not be allowed to grow. The `min_sample_split` parameter was defined, which specified the minimum number of samples required to split an internal node.

```
def train(train_X, train_y, max_depth=2, min_sample_split=2):
    '''
    Function for training the Decision Tree using the training dataset
    Returns root node of the Decision Tree
    '''
    return tree(train_X, train_y, max_depth=max_depth, min_sample_split=min_sample_split)
```

Training function

Task-6 & Task-7

The following functions were defined for making the classification predictions and testing the classifier to ensure its proper working and checking the accuracy of the trained model.

```
def predict(test_X, node):
        '''
        Predicts the class for a single data point
        (Recursive Function)
        '''
        if node['value'] != None:
            return node['value']

        node_feature = test_X[node['feature']]
        if node_feature <= node['threshold']:
            return predict(test_X, node['left'])
        else:
            return predict(test_X, node['right'])
```

```
def test(test_X, test_y, root_node):
        '''
        Returns the predictions and accuracy scored for the test dataset
        '''
        pred_y = []
        Test_y = test_y.tolist()
        for x in range(len(test_X)):
            pred_y.append(predict(test_X.iloc[x], root_node))

        print('True Values:', Test_y)
        print()
        print('Predictions:', pred_y)
        print()

        accuracy = 0
        for each in range(len(pred_y)):
            if pred_y[each] == Test_y[each]:
                accuracy += 1
        accuracy /= len(test_X)
        print('Accuracy:', accuracy)

        classwise_accuracy = 0
        classes = test_y.unique()
        classwise_true_pos_count = {}
        for Class in classes:
            classwise_true_pos_count[Class] = 0

        classwise_count = test_y.value_counts().to_dict()

        for each in range(len(pred_y)):
            if pred_y[each] == Test_y[each]:
                Class = Test_y[each]
                classwise_true_pos_count[Class] += 1

        for Class in classes:
            classwise_accuracy += classwise_true_pos_count[Class]/classwise_count[Class]

        classwise_accuracy /= len(classes)
        print('Classwise Accuracy:', classwise_accuracy)
```

Functions for prediction & testing

The test function also returned the accuracy values for the implemented Decision Tree Classifier.

On performing testing on the test split of the dataset, the following accuracy scores were obtained.

Creation & Training of the Decision Tree Classifier

```
root_node = train(train_X, train_y, max_depth=5)
```

Testing the Decision Tree Classifier

```
test(test_X, test_y, root_node)
```

```
True Values: [0, 0, 1, 0, 0, 0, 2, 1, 2, 2, 1, 0, 0, 2, 0, 0, 2, 0, 2, 0, 0, 0, 1, 2, 1, 2, 0, 0, 0, 0, 0, 2, 0, 2, 0, 1, 2, 0,
2, 0, 1, 2, 0, 0, 0, 0, 0, 0, 2, 0, 2, 0, 2, 1, 0, 0, 0, 0, 0, 1, 0, 2, 0, 1, 0, 0, 2, 2, 2, 1, 2, 2, 1, 2, 0, 1, 0, 1, 0, 2,
2, 2, 2, 1, 2, 2, 2, 0, 0, 1, 1, 0, 2, 0, 1, 0, 2, 0, 2, 2]

Predictions: [0, 0, 1, 0, 1, 1, 2, 1, 2, 2, 1, 0, 0, 2, 0, 0, 2, 0, 2, 0, 0, 0, 1, 2, 1, 2, 1, 0, 0, 0, 0, 2, 0, 2, 0, 1, 2, 0,
2, 1, 1, 2, 0, 1, 0, 0, 0, 0, 2, 2, 2, 0, 2, 1, 0, 0, 0, 1, 0, 1, 0, 2, 0, 2, 0, 1, 2, 2, 2, 1, 2, 2, 1, 2, 0, 1, 0, 2, 0, 2,
2, 2, 2, 1, 2, 2, 2, 0, 0, 1, 1, 0, 2, 1, 1, 0, 2, 0, 2, 2]

Accuracy: 0.89
Classwise Accuracy: 0.9004629629629629
```

Training, Testing and Accuracy Measurement of the implemented Decision Tree Classifier