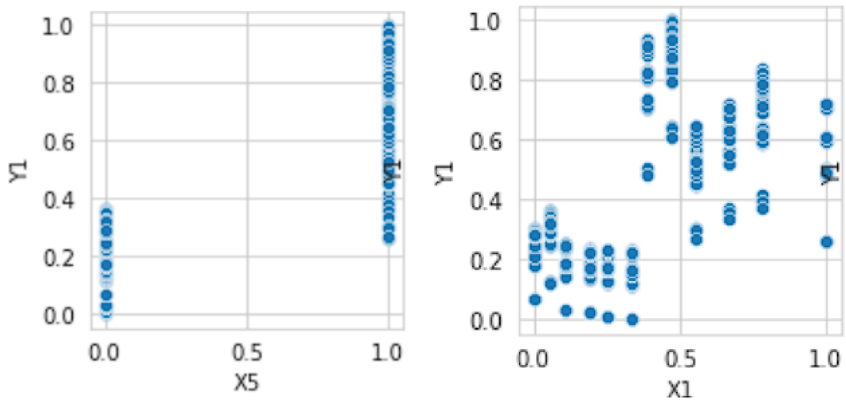
Lab Assignment 2
Prakhar Gupta
B21AI027

Question 1:

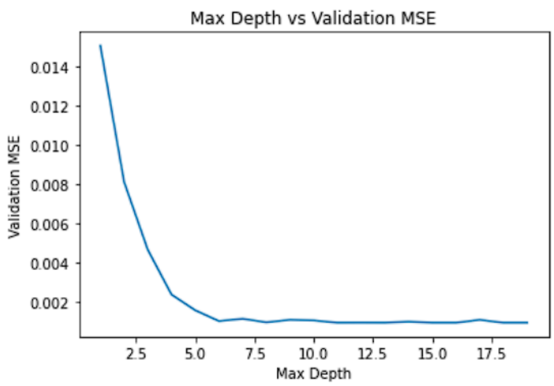
Part 1-

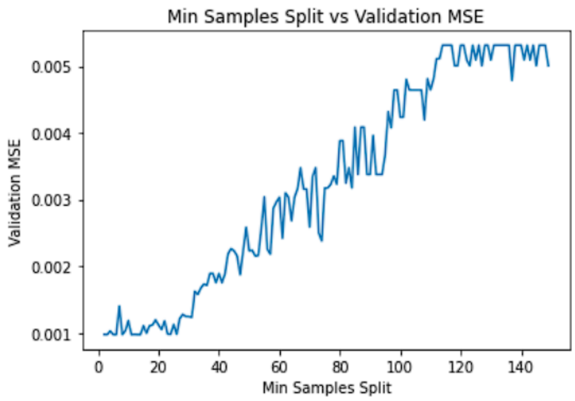
- Downloaded dataset using **wget** command called using **os.system**
- Loaded the **.csv** file into df using pd.read_csv
- Used **df.describe()** to get insights about the dataset
- Checked for not filled rows using **df.isnull().sum()**
- Applied **MinMaxScalar()** to every column to normalise data
- Converted **df to X,y**
- Using **seaborn.scatterplot** plotted **8 plots of X(i) vs Y1**
- Using **train_test_split** to split the X,y into **train:val:test - 70:10:20** ratio
- Relevant features are plotted below i.e. X5,X1 vs Y
- 

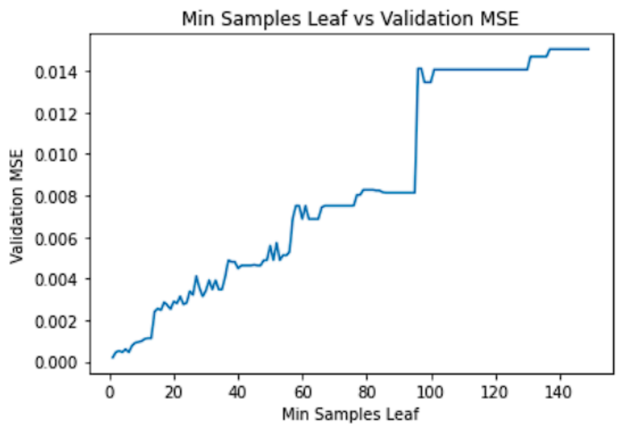
Part 2-

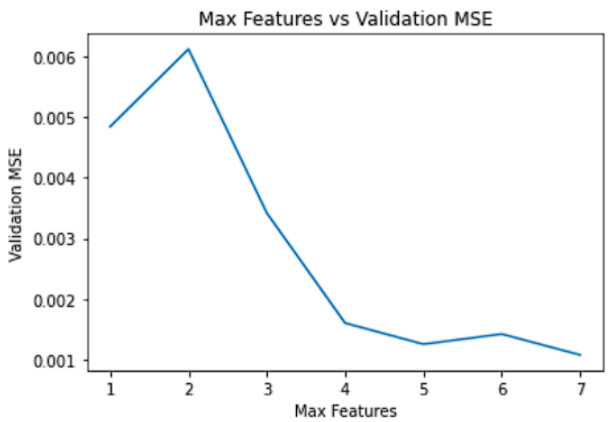
- Implemented **scratch built grid_search function**
- Trained the Decision Tree using scratch built grid_search function by varying **4 parameters max_depth, min_sample_split, max_features, min_samples_leaf**
-
```
Best parameters:  {'max_depth': 7, 'min_samples_split': 2,
'max_features': 5, 'min_samples_leaf': 1}
Best mean_squared_error on validation set:
0.00017774769617306808
Best accuracy on validation set: 99.10258166915922%
```

- **Hyperparameters Varying**:-
  1.)Max depth of the tree controls the complexity of the model. A deeper tree allows the model to capture more information from the data, but it also increases the risk of overfitting. A shallow tree has risk of underfitting
  2.)Minimum number of samples required to split an internal node controls the complexity of the model by setting a threshold for the amount of data needed to create a new split. Increasing the value of this will make model less overfit
  3.)Maximum number of features to consider when looking for the best split controls the complexity of the model by setting a threshold for the number of features to be considered at each split, thus making it high make it overfit
  4.)The minimum number of samples required to be at a leaf node controls the complexity of the model by setting a threshold for the amount of data needed to create a leaf. Increasing the value of this parameter will make the model to underfit
- **Effect of Varying the hyperparameters**:-









- The plots also **supports the arguments** of the hyperparameter

Part 3-

- Performed **Hold-out cross validation, 5-fold cross-validation and repeated-5-fold validation**
- `Hold-out cross test MSE: 0.0012021222805698557`
- `5-fold cross test MSE: 0.0036777312530699344`
- `Repeated 5-fold cross test MSE: 0.0008579605274120983`
- `Test data MSE: 0.00028758955749707665`
- `Accuracy: 99.02646237157461%`

Part 4-

- `Best L1/mean_average_error on validation set: 0.008621251858398281`
- `Best L2/mean_squared_error on validation set: 0.00017665595754066276`
- L2 is working better than L1. It is because L1 tends to perform better when there are a smaller number of important features , whereas L2 performs better when there are many features that are important. The same can also be seen in the decision boundary graphs.

<u>Question 2:</u>

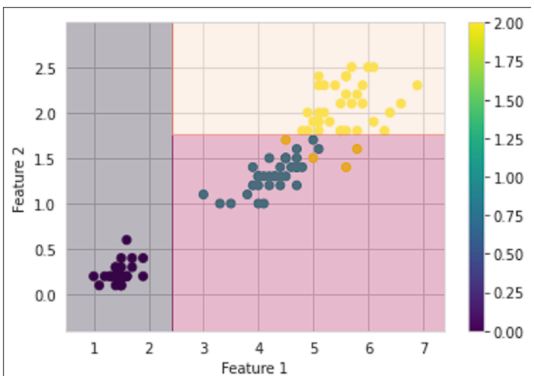
Classification:

Initials-

- Downloaded dataset using **wget** command called using **os.system**
- Load the **iris.csv** dataset in df variable using **pd.read_csv**
- We hardcoded the columns names as it was not given in the dataset itself with the name**['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species']**
- Used **df.describe()** && **df.drop** to drop **'sepal_length', 'sepal_width'**
- Converted **df to X,y**
- Using **train_test_split** splitted **X,y** into **train and test in 80:20** ratio

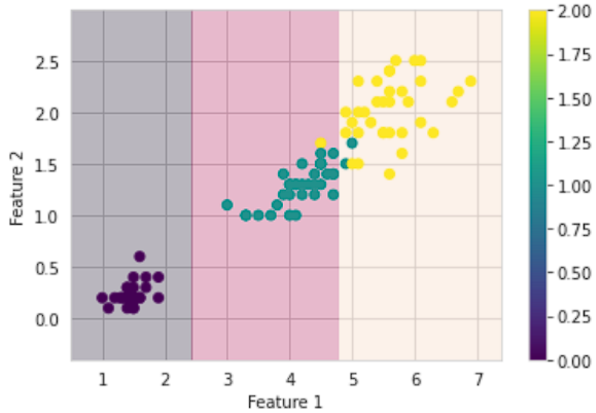
Part 1-

- Used **sklearn** library **DecisionTreeClassifier** to train dt
- Plotted the **decision tree**(which indicate the **depth** at which each **split was made**) as well as **decision boundary.**



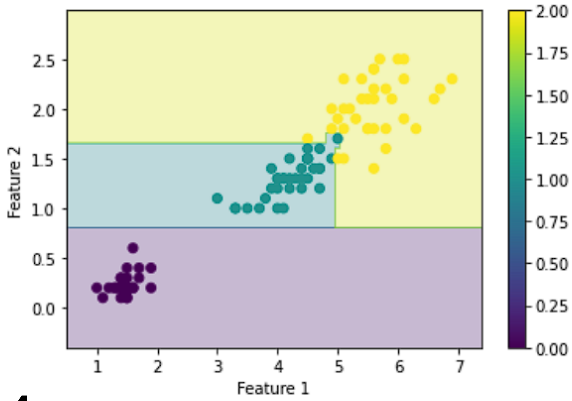
Part 2-

- Removed the widest Iris-Versicolor from the iris training set (the one with petals 4.8 cm long and 1.8 cm wide) using **np.delete**
- Used **sklearn** library **DecisionTreeClassifier** to train dt
- Plotted the **decision boundary.**



Part 3-

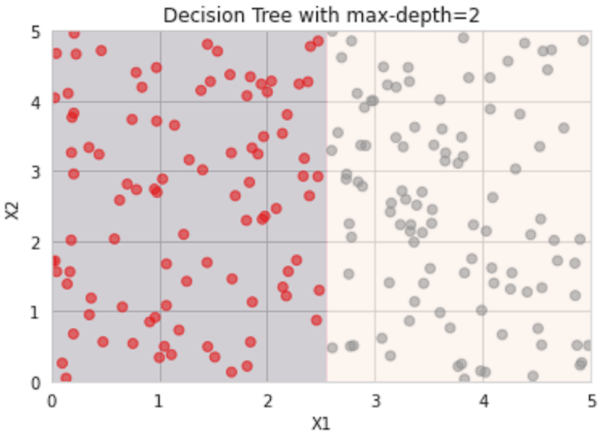
- Fitted the **Decision Tree Classifier** with (max-depth = None) and plotted the decision boundary.
- It is somewhat **overfitting** at the region where green point goes in yellow region
- The **accuracy of this will be greater** than one trained with (max_depth=2)



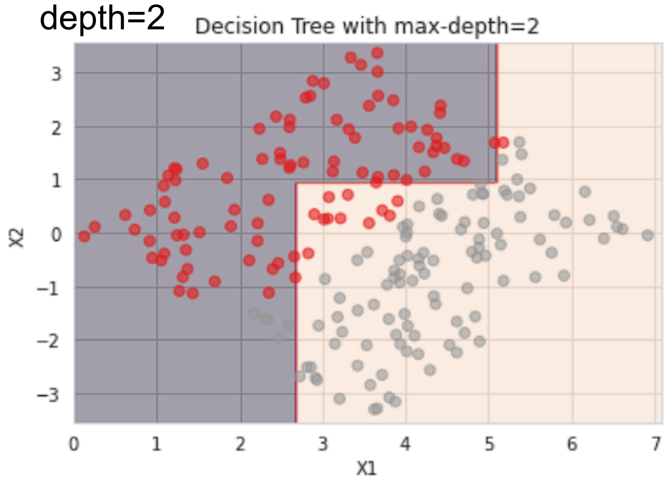
Part 4-

- Initialized **X** using **np.random.rand**
- Using **np.where** defined classes of X either 0,1 depending on the X[:, 0] < 2.5
- Trained a **DecisionTreeClassifier(max_depth=2)**
- Plotted the **decision boundary**
- Got **Accuracy=1**

- The accuracy is 1 because the points are **separable just using a single line** at x=2.5



- Rotated the points X in 45 degree in clockwise direction using **rot_matrix= ([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.cos(theta)]])**
- Then **X_rotated = X @ rot_matrix**
- Fitted the **DecisionTreeClassifier(max_depth=2)**
- Plotted the **decision boundary**
- Got **Accuracy=0.915**
- The accuracy is 0.915 because the points are not separable just using depth=2



Part 5-

- As we increased the depth of the model in part 3 we get **leaf node of the tree are pure** which significantly was a **major jump in performance compared to model in part 2**
- In part 4 we see that a **depth 2 model** can perform well when the data is **linearly separable** using just one line
- Whereas when the data is not linearly separable using just one line or as shown in decision boundary of part 4, it is **difficult to get high accuracy**.

- So if we **increase the depth** here like we do from problem 2 to problem 3, we will get a better model

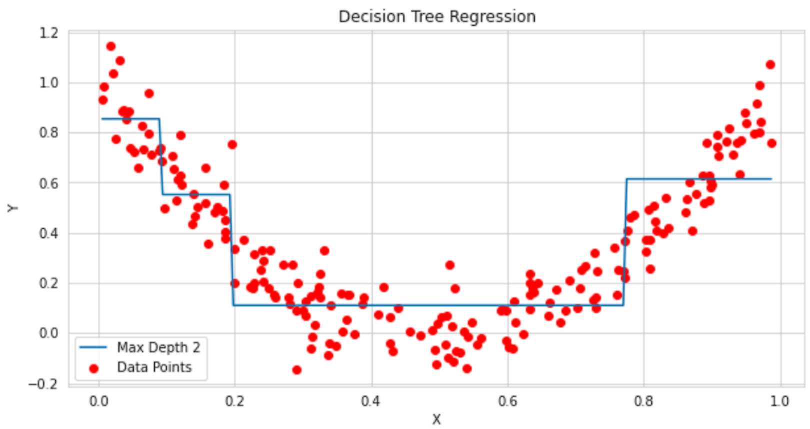
Regression:

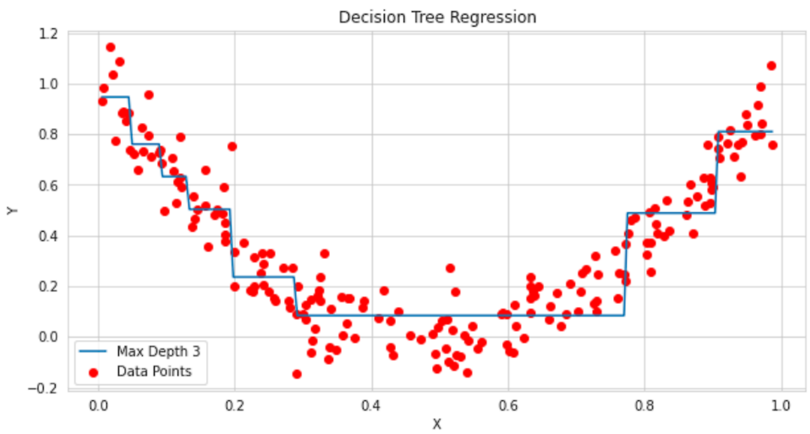
Initials-

- Downloaded dataset using **wget** command called using **os.system**
- Load the **task.csv** dataset in df variable using **pd.read_csv**
- Used **df.describe()** && **df.isnull().sum** to check any NaN value
- Converted **df to X,y**

Part 1-

- Used **sklearn** library **DecisionTreeRegressor** to train dt
- Plotted the **regression predictions** at **each depth** for **each max_depth**
- The most above line having max y=c value is the prediction at depth 0 and as we move down y=c we get into more depth

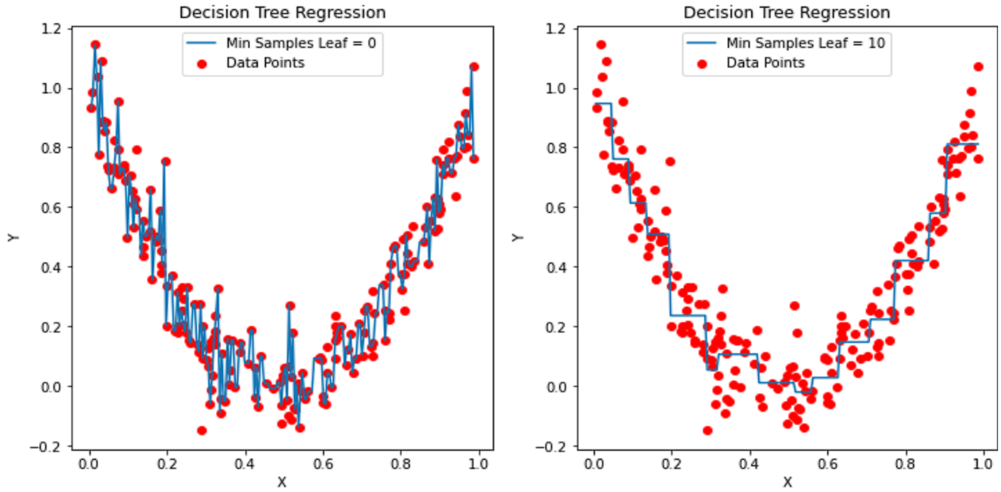




- We can clearly see that the Decision Tree makes a more **complex structure** as the **depth** of the tree is **increased**. So the model gets more complex as its depth increases

Part 2-

- Used sklearn library DecisionTreeRegressor to train 2 dt
- Plotted the **regression predictions** for the **two min_samples_leaf = 0,10**



- We can kind of say that DTR with min_sample_leaf=0 is kind of **overfitting** the data and min_sample_leaf=10 is **underfitting** the data

<u>Question 3:</u>

Part 1-

- Downloaded palmerpenguins using **!pip install** palmerpenguins
- Loaded the **dataset** file into **penguins**
- **Used** penguins.isnull().sum() to check for NaN value
- Using **dropna.()** we dropped the NaN rows which were total 11 rows dropped out of initial 344 rows
- Using categorical label encoder to **['island','sex','year'] columns**
- Converted penguins to X,y
- Used **penguins.describe()** to get insights about the dataset
- Used **MinMaxScalar()** to normalise **['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm', 'body_mass_g']**
- Converted **penguins to X,y**
- Using **seaborn.pairplot** plotted 36 plots
- Using **train_test_split** to split the X,y into **train:test - 80:20** ratio

Part 2-

- Roll number is B21AI027 so implemented gini_index

- Got gini_index(y) = `0.6383680978275572`

Part 3-

- Implemented cont_to_cat
- For every column in dataset, for every value in column splitted the column into two category
- Then find out the best gini for different splits and used that as a base to split the column into two categories
- Applied Col_to_categorised to **['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm', 'body_mass_g']**
- **Thresholds=**`[0.37090909090909085, 0.3928571428571428, 0.5762711864406782, 0.5]` **for ['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm', 'body_mass_g']**
- Assigning X=penguins to update the col_to_categorised

Part 4,5,6-

- Implemented scratch built class ‘**Node**’
- **Node** contains *X*,*y*,*depth*,*done_col*,*nodes*(its child nodes),gini,split_col(the column used to split at that node)

- Implemented scratch built class **‘DT’** (Decision Tree)
- **DT** contains following function
  __init__ , *fit* (training DT on dataset), *build_tree* (create DT), *best_split*(gives best split for a particular column), *most_common_pred*(returns max mode class at leaf node), *predict*(get predicts on test data), *accuracy*(find accuracy on test data), *class_wise_accuracy*(find class wise accuracy on test data)

  `build_tree(self,node)-`
  - ➔ check for node **not exceeding max_depth**
  - ➔ Check if **gini==0** then return most_common_pred
  - ➔ Find best split using best_split and then define ginis of the node from the return value of best_split
  - ➔ Append the best_col to done_col
  - ➔ Assign best_col to the split_col
  - ➔ Perform **DFS** by iterating over different categories in that column and calling build_tree on that child_node

  `best_split(self,X,y,done_col,gini)-`

- ➔ `Iterating over all col not in done_col and finding best gini and making it the best_col for split at that node`

`most_common_pred(self,y)-`

- ➔ `Returns the class value which has highest number of counts in the present data at that node`

`predict(self,X_test):`

- ➔ `For each row of test data it starts from the root node and traverse down till it reaches a node ,(it takes decision to which direction to traverse based on the split_col present in the particular node in which we are currently at) then at the leaf node it predicts the class using most_common_pred`

Part 7-

- We splitted the dataset before using col_to_categorised so resplitting the X,y into train and test into 80:20 ratio
- Trained DT using max_depth=5
- Found accuracy on test data equal to `1.0`
- Found class wise accuracy on test data equal to `{'Adelie': 1.0, 'Chinstrap': 1.0, 'Gentoo': 1.0}`