Eren Barış Bostancı
68770

ENGR 421 – HW5

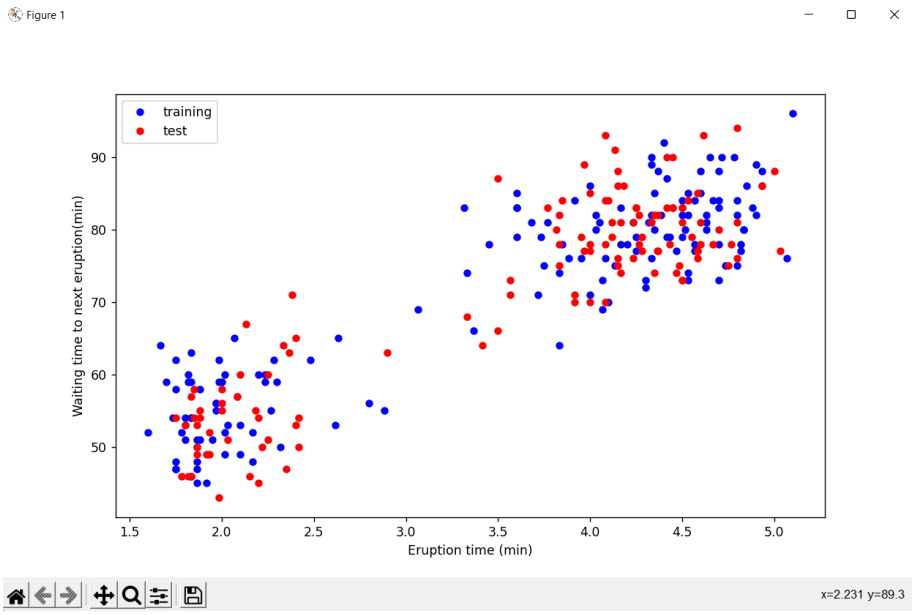
Data Importing:

This part was exactly same with the HW4 Data Importing.

```
data = np.genfromtxt("hw05_data_set.csv", delimiter=",")
data = data[1:]

x = data[:, 0]
y = data[:, 1].astype(int)

x_train = x[0:150]
y_train = y[0:150]

x_test = x[150:]
y_test = y[150:]

K = np.max(y)
N = x.shape[0]

P = 25

# get numbers of train and test samples
N_train = len(y_train)
N_test = len(y_test)

minimum_value = min(x_train)
maximum_value = max(x_train)
```

```
plt.figure(figsize=(10, 6))
plt.plot(x_train, y_train, "b.", markersize=10, label="training")
plt.plot(x_test, y_test, "r.", markersize=10, label="test")

plt.xlabel("Eruption time (min)")
plt.ylabel("Waiting time to next eruption(min)")
plt.legend(loc='upper left')
plt.show()
```

And plot the data.



Eren Barış Bostancı
68770

Decision Tree

I used the tree that we used on LAB7, but I modified it and put it in a function to call multiple times. I removed the parts that depends on D since our data set is one dimension and add the part to check if the depth is equal to the P if it is it won't split. Moreover, it counts the error and find the best split. These steps added for the Pre-Prune process. And instead of frequency I directly get node means.

```
def desicion_tree(P, x_train, y_train):
    # create necessary data structures
    node_indices = {}
    is_terminal = {}
    need_split = {}

    node_means = {}
    node_splits = {}

    # put all training instances into the root node
    node_indices[1] = np.array(range(len(x_train)))
    is_terminal[1] = False
    need_split[1] = True

    while True:

        # find nodes that need splitting
        split_nodes = [key for key, value in need_split.items() if value == True]
        if len(split_nodes) == 0:
            break

        # find best split positions for all nodes
        for split_node in split_nodes:
            data_indices = node_indices[split_node]
```

```
            need_split[split_node] = False
            node_mean = np.mean(y_train[data_indices])

            if x_train[data_indices].size <= P:
                node_means[split_node] = node_mean
                is_terminal[split_node] = True

            else:
                is_terminal[split_node] = False
                unique_values = np.sort(np.unique(x_train[data_indices]))
                split_positions = (unique_values[1:len(unique_values)] + unique_values[0:(len(unique_values) - 1)]) / 2
                split_scores = np.repeat(0.0, len(split_positions))
                for s in range(len(split_positions)):
                    left_indices = data_indices[x_train[data_indices] < split_positions[s]]
                    right_indices = data_indices[x_train[data_indices] >= split_positions[s]]
                    errors = 0
                    if len(left_indices) > 0:
                        errors += np.sum((y_train[left_indices] - np.mean(y_train[left_indices])) ** 2)
                    if len(right_indices) > 0:
                        errors += np.sum((y_train[right_indices] - np.mean(y_train[right_indices])) ** 2)
                    split_scores[s] = errors / (len(left_indices) + len(right_indices))

                # if len 1 is when we take unique values
```

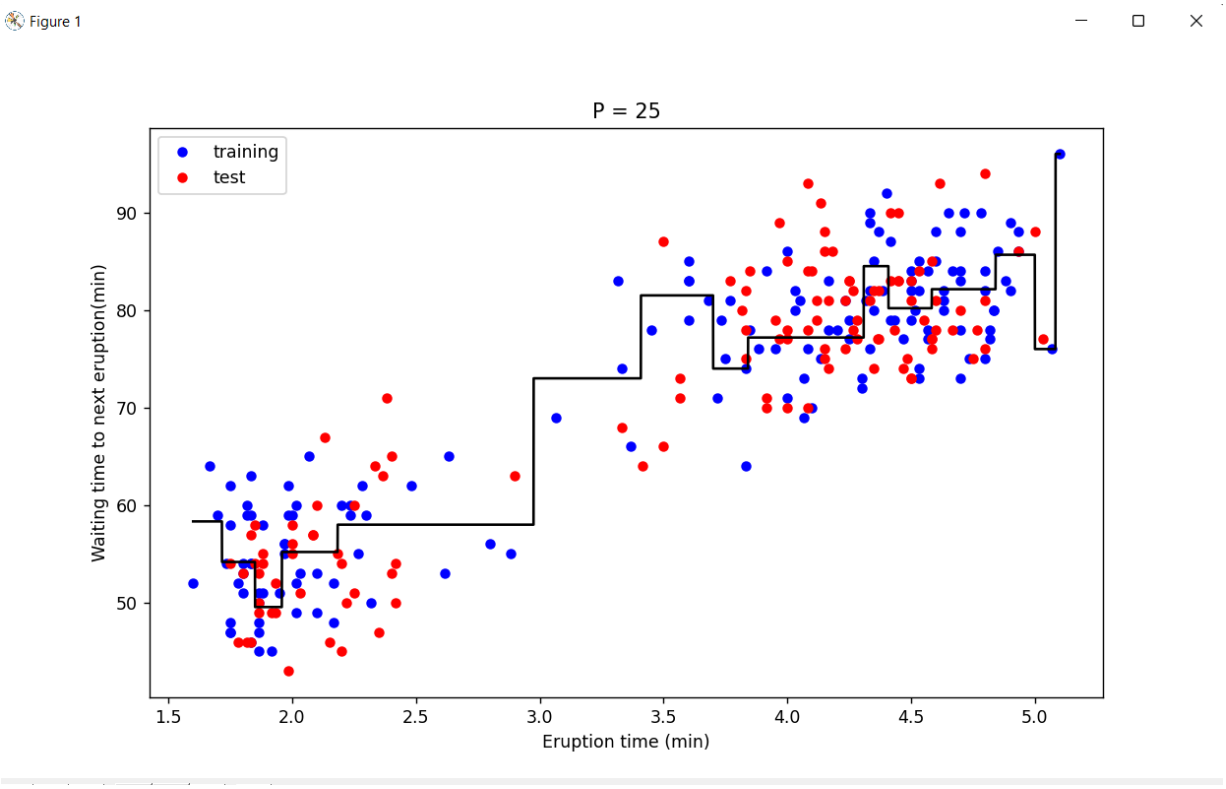
```
            # if len 1 is when we take unique values
            if len(unique_values) == 1:
                is_terminal[split_node] = True
                node_means[split_node] = node_mean
                continue
            best_split = split_positions[np.argmin(split_scores)]
            node_splits[split_node] = best_split

            # create left node using the selected split
            left_indices = data_indices[(x_train[data_indices] < best_split)]
            node_indices[2 * split_node] = left_indices
            is_terminal[2 * split_node] = False
            need_split[2 * split_node] = True

            # create right node using the selected split
            right_indices = data_indices[(x_train[data_indices] >= best_split)]
            node_indices[2 * split_node + 1] = right_indices
            is_terminal[2 * split_node + 1] = False
            need_split[2 * split_node + 1] = True
    return node_splits, node_means, is_terminal
```

It returns the noed_splits, node_means and is_terminal for checking. Then calculate the p_hat using the data interval.

```
node_splits, node_means, is_terminal = desicion_tree(P, x_train, y_train)
data_interval = np.linspace(minimum_value, maximum_value, 6001)

p_hat = []
for interval in data_interval:
    index = 1
    while True:
        if is_terminal[index]:
            p_hat.append(node_means[index])
            break
        if interval < node_splits[index]:
            index = index * 2
        else:
            index = index * 2 + 1
```

And the plot.

```
plt.figure(figsize=(10, 6))
plt.plot(x_train, y_train, "b.", markersize=10, label="training")
plt.plot(x_test, y_test, "r.", markersize=10, label="test")

plt.plot(data_interval, p_hat, color="black")

plt.title("P = " + str(P))

plt.xlabel("Eruption time (min)")
plt.ylabel("Waiting time to next eruption(min)")
plt.legend(loc='upper left')
plt.show()
```



RMSE

I used the RMSE function that I used for HW4.

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{N_{train}} (y_i - \hat{y}_i)^2}{N_{train}}} \qquad \text{RMSE} = \sqrt{\frac{\sum_{i=1}^{N_{test}} (y_i - \hat{y}_i)^2}{N_{test}}}.$$

```
def RMSE(x_test, y_test, y_hat, data_interval):
    rmse = 0
    for i in range(len(y_hat) - 1):
        for j in range(len(x_test)):
            test = x_test[j]
            if (data_interval[i] < test) & (test <= data_interval[i + 1]):
                rmse = rmse + (y_test[j] - y_hat[i]) * (y_test[j] - y_hat[i])

    return math.sqrt(rmse / len(y_test))
```

```
rmse_training = RMSE(x_train, y_train, p_hat, data_interval)
print("RMSE on training set is " + str(rmse_training) + " when P is " + str(P))

rmse_test = RMSE(x_test, y_test, p_hat, data_interval)
print("RMSE on test set is " + str(rmse_test) + " when P is " + str(P))
```

```
D:\Users\erenb\PycharmProjects\ML-HW4\venv\Scripts\python.exe
RMSE on training set is 4.511675842160385 when P is 25
RMSE on test set is 6.454083413352087 when P is 25
```

For different P values I created P list and recall tree with these values than calculate the RMSE for each of them and plot it.

```
Ps = np.arange(5, 55, 5)
rmse_p_train = []
rmse_p_test = []
for p in Ps:
    node_splits, node_means, is_terminal = desicion_tree(p, x_train, y_train)
    p_hat = []
    for interval in data_interval:
        index = 1
        while True:
            if is_terminal[index]:
                p_hat.append(node_means[index])
                break
            if interval < node_splits[index]:
                index = index * 2
            else:
                index = index * 2 + 1
    rmse_p_train.append(RMSE(x_train, y_train, p_hat, data_interval))
    rmse_p_test.append(RMSE(x_test, y_test, p_hat, data_interval))
```

```
plt.figure(figsize=(10, 6))
plt.plot(Ps, rmse_p_train, "bo-", linewidth=2, markersize=10, label="training")
plt.plot(Ps, rmse_p_test, "ro-", linewidth=2, markersize=10, label="test")
plt.xlabel("Pre-pruning size (P)")
plt.ylabel("RMSE")
plt.legend(loc='upper right')
plt.show()
```