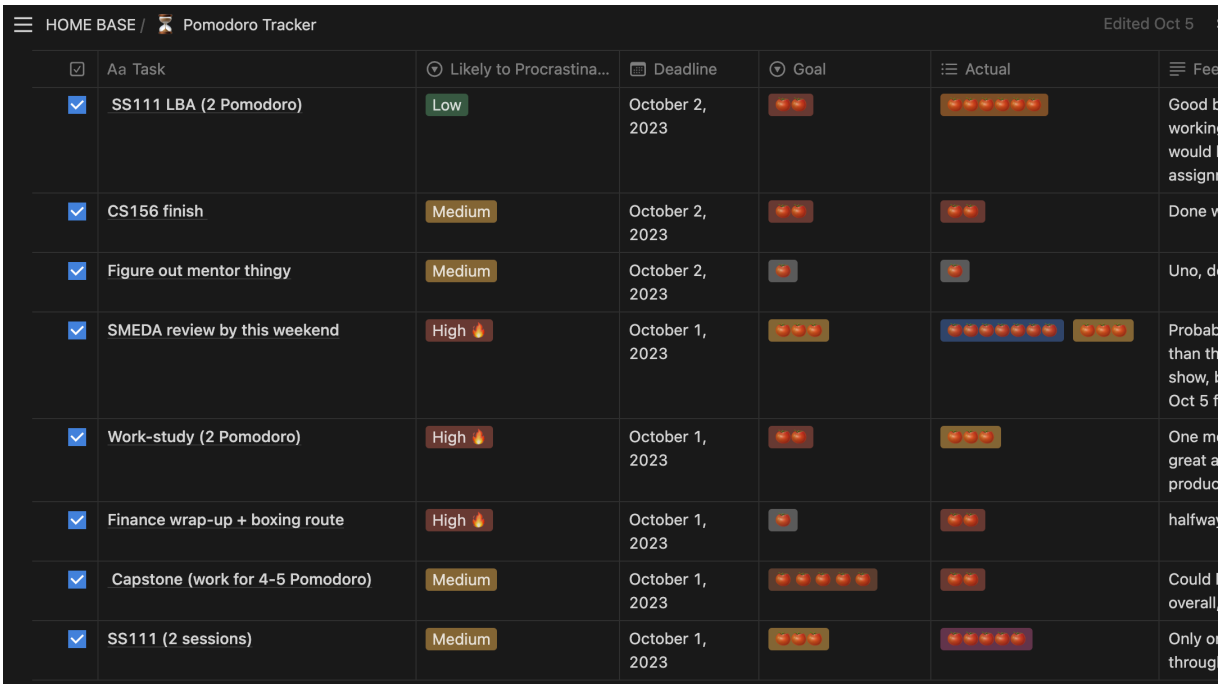
# Section 1: Explaining the problem

Last summer, I read a book on how to use the pomodoro technique correctly. Pomodoro is a productivity technique that helps you divide your work time into focused work and breaks, while you eradicate distractions and maintain a laser-like focus on your work. You start with a 25 minutes of focused working with no distractions, and then follow it with 5 minutes of break and then do that again. I love using pomodoros whenever I am working. Any piece of work I have ever created since then was likely done while I had my small timer on, towards the right of my computer, counting down the minutes and seconds until my next break.

Another inportant aspect of using pomodoros, is keeping track of them. You typically have a list of tasks that you want to accomplish within a time frame, you estimate how many pomodoros (or 25 minute chunks of work) it will take you to do them and then you complete your task and see how many pomodoros it actually took you to finish it. This is how that looks like for me:




One problem I run into a lot is that I grossly underestimate how long it will take me to do a certain task. Many tasks take a lot more pomodoros than I anticipate, some less, so I wondered if I could design a machine learning algorithm that can analyze my estimates and actual pomodoros, and then be able to accurately predict how many pomodoros it will take me to actually do a certain task based on how many I estimate.

# Section 2: Explaining the data

Since all my data was in the form of tomato emojis on a notion chart, it would've been quite complicated to export it and convert it to numerical data. Although there are python libraries that can help incorporate emojis in your code like pythonji. I only have 15 data points, which means that it is doable and much easier to export them into python manually, which I did in a list of lists. Then when plotting them, I broke the list of lists into two lists, one for the x values in the data and another for the y values. The reason why I decided on a list was because it was exceptionally important my data was ordered, since each x value is linked to a y value and if they were jumbled then my data would be meaningless. Lists also allow duplicate values which was important since my data does have duplicate values which is helpful for making stronger predictions. It is also convinient that a list is changeable, since it would help me add more data points in the future and make my predictions better.

Here is a plot of my last 15 pomodoros. On the x-axis we have the estimated times and on the y-axis we have the actual times.

In [62]:
```
#Lists of lists created manually from my notion.
Total =([[3, 5],
        [5, 4],
        [1, 2],
        [2, 3],
        [1, 1],
        [1, 2],
        [2, 6],
        [1, 3],
        [2, 3],
        [1, 1],
        [1, 2],
        [1, 1],
        [3, 10],
        [2, 3],
        [1, 1]])


```

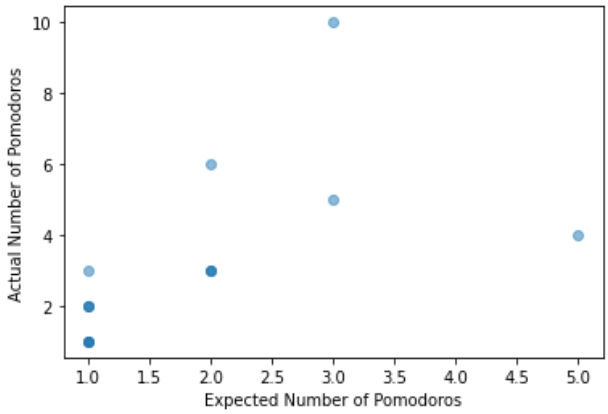
## Section 3:

As mentioned above, I broke down my list of lists into two seperate lists for x and y values, and then I plotted them to get an idea of the data visually. The visualization was quite anti-climatic since we have a lot of repeating values. I made the data points transparent, by setting alpha to 0.5, so that the repetitions are more obvious. The data is discrete, which can make it harder to process, and it is repeating quite a lot as well whihc is not a bad things since that tells me that there is a regular pattern to how I estimate a pomodoro and how the actual number of pomodoros play out.

In [63]:
```
import matplotlib.pyplot as plt

#Breaking down the lists for plotting
DataX = [x[0] for x in Total]
DataY = [x[1] for x in Total]

print("List 1:", DataX)
print("List 2:", DataY)

plt.xlabel("Expected Number of Pomodoros")
plt.ylabel("Actual Number of Pomodoros")
plt.scatter(DataX, DataY,alpha=0.5)
plt.show
```

```
List 1: [3, 5, 1, 2, 1, 1, 2, 1, 2, 1, 1, 1, 3, 2, 1]
List 2: [5, 4, 2, 3, 1, 2, 6, 3, 3, 1, 2, 1, 10, 3, 1]
```

Out[63]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```




## Explaining my decisions:

My data and goal, makes this a regression problem. We have two variables and we want to model a relationship between them in a way that makes it easier to predict one based on the other.

Out of all the potential tools I could've used to solve this, I ended up deciding on a Regression Analysis and a Regression Tree. The reason I used these two methods, was because I wasn't sure if my data was linear or not and couldn't decide which one would give me more accurate reuslts. Other methods like Random forests, Support vector

machines, and neural networks could also be used, and could potentially provide better results, but I don't have a lot of data points, which would undermine the use of these more complicated models since using these models would take a longer time, more processing and their effectiveness would be capped by the limited data. Thus the two chosen methods are adequate for the task at hand.

I analysed the results of my two models through mean absolute error and root mean squared error.

I was also quite nervous about my limited data and felt like splitting it 80/20 for training and testing would make the dataset even smaller. So I utilized a cross-validation technique of LOOCV or Leave One Out Cross Validation to further check the robustness of my models. I avoided K-fold techniques cause they expect the data to be split into many 'sets' which I didn't want to do, given my limited data.

Once I was done, I wondered if my predictions and models could get even better if I seperated my data set into the tasks that I added another variable with binary values. 1 for if I genuinely enjoyed doing a task and 0 for the tasks that were chores. Thus I divided my data and then did both, the regression analysis and the regression tree on this split data set. I also made use of LOOCV as a cross-validation technique for both of these models as well.

I start out in each part by fitting a good old descriptive regression to my data just to see what it looked like.

## Explaining the math:

1. **Linear Regression based on an 80/20 Split:**
    - Linear regression involves fitting a linear equation to the observed data. An 80/20 split indicates that 80% of the data is used for training the model, and the remaining 20% is used for testing its performance.
    - Formula: The formula for simple linear regression is $Y = \beta_0 + \beta_1X + \epsilon$, where $Y$ is the dependent variable, $X$ is the independent variable, $\beta_0$ and $\beta_1$ are the regression coefficients, and $\epsilon$ is the error term.
2. **Regression Trees based on an 80/20 Split:**
    - Regression trees involve partitioning the predictor space into distinct regions based on the values of the predictors. An 80/20 split implies that 80% of the data is used for training the regression tree, and the remaining 20% is used for testing the tree's performance.
    - Formula: The splitting criterion in this case involves measures of variance or mean squared error, to determine the optimal splits at each node.
3. **LOOCV (Leave-One-Out Cross-Validation):**
    - LOOCV is a method for assessing the performance of a model. It involves splitting the dataset into two parts, training and testing, where one data point is used for testing and the remaining points are used for training. This process is repeated for each data point, and the average error is computed.
    - Formula: The LOOCV error is computed as the average of the errors obtained from each run of the leave-one-out procedure.
4. **MAE (Mean Absolute Error):**
    - It measures the average of the absolute differences between predicted and actual values. It gives an idea of the magnitude of the error, without considering their direction.
    - Formula: $MAE = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$, where $n$ is the number of data points, $y_i$ is the actual value, and $\hat{y}_i$ is the predicted value.
5. **RMSE (Root Mean Squared Error):**
    - It measures the square root of the average of the squared differences between predicted and actual values. It gives a relatively higher weight to large errors.
    - Formula: $RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2}$, where $n$ is the number of data points, $y_i$ is the actual value, and $\hat{y}_i$ is the predicted value.

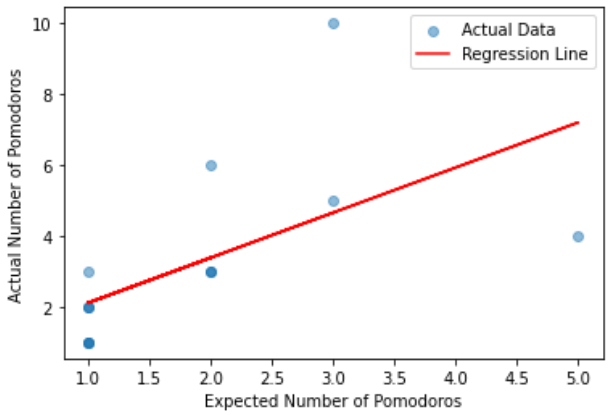
# Sections 4, 5, 6, 7 and 8:

The task that would work best with the data and the goal would be a regression since I am trying to get a clear idea of how much I underestimate when assigning pomodoros to a task. Although the data is descrete, since Pomodoros are typically not divided, I decided to run the regression as if they were continious and then round up to the nearest whole number when I get a good prediction model that predicts the number of pomodoros required for a task.

### Visualizing a Linear Regression

In [64]:
```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import LeaveOneOut
from sklearn.tree import DecisionTreeRegressor

# Reshape the data to be 2D (required by scikit-learn)
DataX = np.array(DataX).reshape(-1, 1)
DataY = np.array(DataY)

# Perform an 80/20 train-test split
X_train, X_test, y_train, y_test = train_test_split(DataX, DataY, test_size=0.8)


```

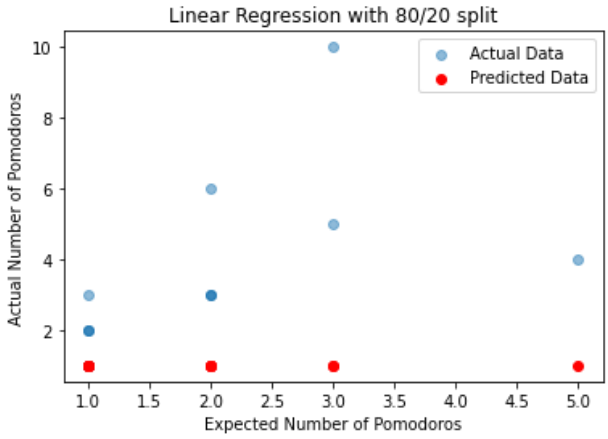
In [65]:
```
# Create a linear regression model
model = LinearRegression()

# Fit the model to your data
model.fit(DataX, DataY)

# Make predictions
y_pred = model.predict(DataX)

# Visualize the data and regression line
plt.scatter(DataX, DataY, alpha=0.5, label="Actual Data")
plt.plot(DataX, y_pred, color='red', label="Regression Line")
plt.xlabel("Expected Number of Pomodoros")
plt.ylabel("Actual Number of Pomodoros")
plt.legend()
plt.show()


```




A basic linear regression is not very impressive, just looking at the line of best fit tells us that this prediction is quite inaccurate and the mean squared error is also quite high.

In [66]:

```python
# Create polynomial features
poly_features = PolynomialFeatures(degree=2)
X_train_poly = poly_features.fit_transform(X_train)
X_test_poly = poly_features.transform(X_test)

# Create a polynomial regression model
model = LinearRegression()

# Fit the model to the training data
model.fit(X_train_poly, y_train)

# Make predictions on the testing data
y_pred = model.predict(X_test_poly)

mse = mean_squared_error(y_test, y_pred)
# Calculate root mean squared error
rmse = np.sqrt(mse)


# Calculate mean absolute error
mae = mean_absolute_error(y_test, y_pred)

# Visualize the data and regression line
plt.scatter(DataX, DataY, alpha=0.5, label="Actual Data")
plt.scatter(X_test, y_pred, color='red', label="Predicted Data")
plt.xlabel("Expected Number of Pomodoros")
plt.ylabel("Actual Number of Pomodoros")
plt.title('Linear Regression with 80/20 split')
plt.legend()
plt.show()

# Print the regression coefficients, RMSE, and MAE
print("Intercept:", model.intercept_)
print("Coefficients:", model.coef_)
print("Root Mean Squared Error:", rmse)
print("Mean Absolute Error:", mae)

```

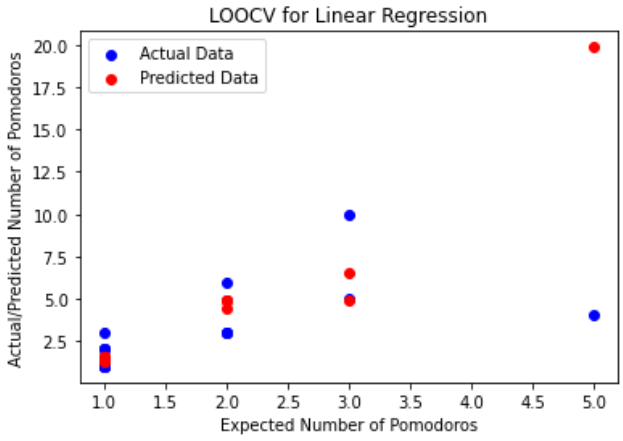



```
Intercept: 1.0
Coefficients: [0. 0. 0.]
Root Mean Squared Error: 3.5355339059327378
Mean Absolute Error: 2.6666666666666665
```

There is no "good" RMSE or MAE. Thus we will compare the models among themselves at the end and see which one does the best job.

In [67]:

```
# Initialize lists to store the predicted and actual values
y_pred_loocv = []
y_actual_loocv = []

loo = LeaveOneOut()
for train_index, test_index in loo.split(DataX):
    X_train, X_test = DataX[train_index], DataX[test_index]
    y_train, y_test = DataY[train_index], DataY[test_index]

    poly_features = PolynomialFeatures(degree=2)
    X_train_poly = poly_features.fit_transform(X_train)
    X_test_poly = poly_features.transform(X_test)

    model = LinearRegression()
    model.fit(X_train_poly, y_train)

    y_pred = model.predict(X_test_poly)

    y_pred_loocv.append(y_pred[0])
    y_actual_loocv.append(y_test[0])

# Convert lists to NumPy arrays
y_pred_loocv = np.array(y_pred_loocv)
y_actual_loocv = np.array(y_actual_loocv)

# Plotting the LOOCV predictions and actual values
plt.scatter(DataX, DataY, color='blue', label="Actual Data")
plt.scatter(DataX, y_pred_loocv, color='red', label="Predicted Data")
plt.xlabel("Expected Number of Pomodoros")
plt.ylabel("Actual/Predicted Number of Pomodoros")
plt.title('LOOCV for Linear Regression')
plt.legend()
plt.show()

# Calculate mean squared error for LOOCV
mse_loocv = mean_squared_error(y_actual_loocv, y_pred_loocv)
rmse_loocv = np.sqrt(mse_loocv)
mae_loocv = mean_absolute_error(y_actual_loocv, y_pred_loocv)

# Print mean squared error, root mean squared error, and mean absolute error
print(f"Root Mean Squared Error with LOOCV: {rmse_loocv}")
print(f"Mean Absolute Error with LOOCV: {mae_loocv}")

```



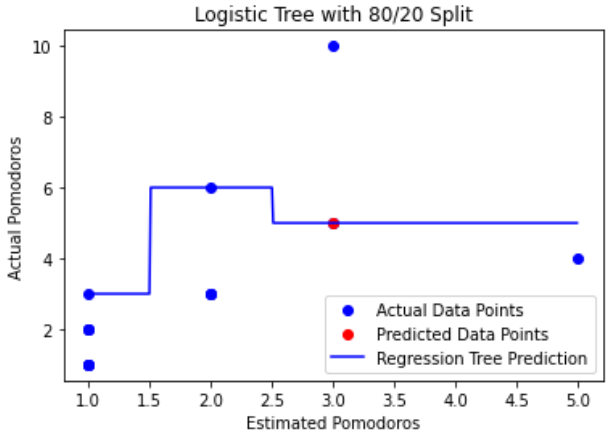

```
Root Mean Squared Error with LOOCV: 4.465143637755054
Mean Absolute Error with LOOCV: 2.3672462021880403
```

## LOGISTIC REGRESSION

In [83]:

```
# Create and fit the regression tree on the training data
reg_tree_split = DecisionTreeRegressor(random_state=0)
reg_tree_split.fit(X_train, y_train)

# Make predictions on the test data
y_pred_split = reg_tree_split.predict(X_test)

# Compute evaluation metrics for the 80/20 split
mae_split = mean_absolute_error(y_test, y_pred_split)
mse_split = mean_squared_error(y_test, y_pred_split)
rmse_split = np.sqrt(mse_split)


# Print the evaluation metrics for the 80/20 split
print("Mean Absolute Error", mae_split)
print("Root Mean Squared Error:", rmse_split)


# Visualize the regression tree with the 80/20 split data
#X_grid = np.arange(min(DataX), max(DataX), 0.01).reshape(-1, 1)
plt.scatter(DataX, DataY, color='blue', label='Actual Data Points')
plt.scatter(X_test, y_pred_split, color='red', label='Predicted Data Points')
plt.plot(X_grid, reg_tree_split.predict(X_grid), color='blue', label='Regression Tree P
plt.title('Logistic Tree with 80/20 Split')
plt.xlabel('Estimated Pomodoros')
plt.ylabel('Actual Pomodoros')
plt.legend()
plt.show()
```

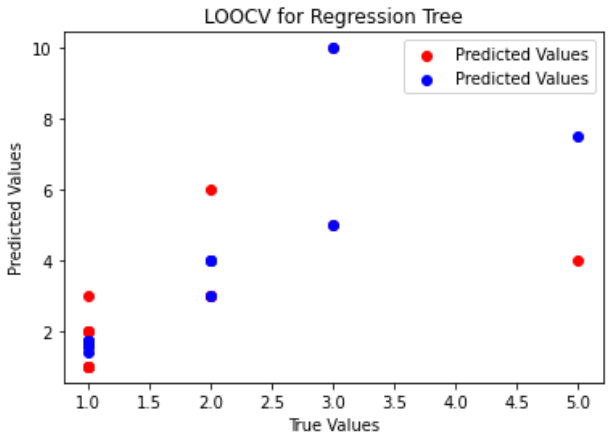
```
Mean Absolute Error 5.0
Root Mean Squared Error: 5.0
```



In [84]:

```python
# Perform Leave-One-Out Cross-Validation
loo = LeaveOneOut()
y_true_cv, y_pred_cv = [], []

for train_index, test_index in loo.split(DataX):
    X_train_cv, X_test_cv = DataX[train_index], DataX[test_index]
    y_train_cv, y_test_cv = DataY[train_index], DataY[test_index]

    # Fit the regression tree on the training data
    reg_tree_cv = DecisionTreeRegressor(random_state=0)
    reg_tree_cv.fit(X_train_cv, y_train_cv)

    # Make predictions on the test data
    y_pred_cv.extend(reg_tree_cv.predict(X_test_cv))
    y_true_cv.extend(y_test_cv)

# Compute evaluation metrics for LOOCV
mae_cv = mean_absolute_error(y_true_cv, y_pred_cv)
mse_cv = mean_squared_error(y_true_cv, y_pred_cv)
rmse_cv = np.sqrt(mse_cv)

# Print the evaluation metrics for LOOCV
print(f"LOOCV - Mean Absolute Error: {mae_cv:.2f}")
print(f"LOOCV - Root Mean Squared Error: {rmse_cv:.2f}")

import matplotlib.pyplot as plt

# Create a scatter plot of true values vs predicted values for LOOCV
plt.scatter(DataX, DataY, color='red', label='Predicted Values')
plt.scatter(DataX, y_pred_cv, color='blue', label='Predicted Values')
plt.title('LOOCV for Regression Tree')
plt.xlabel('True Values')
plt.ylabel('Predicted Values')
plt.legend()
plt.show()
```

```
LOOCV - Mean Absolute Error: 1.68
LOOCV - Root Mean Squared Error: 2.30
```



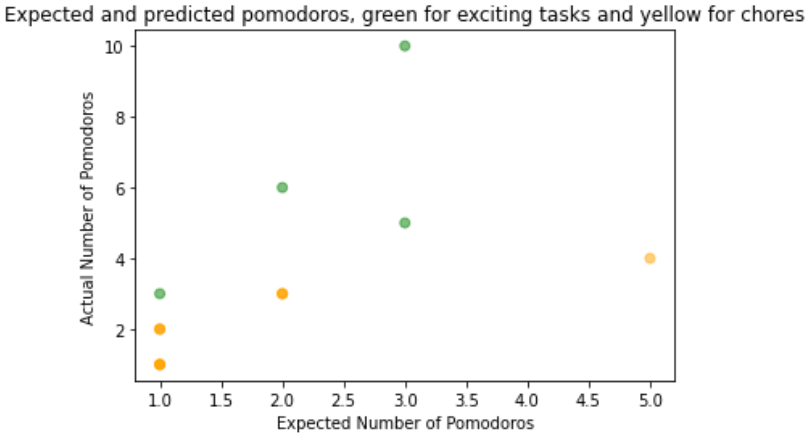

## Adding more nuance for better predictions

This is better than the last model, but I am still not very happy with it. I won't trust this model to rightly predict the number of pomodoros for me and that makes me sad.

To make this model better, I have decided that I am going to add another variable, that I expect to be helpful. The variable would indicate what kind of task am I assigning pomodoros for, is it a task I am genuintely excited to work on or is it a task that feels more like a chore? The variable introduced as data z is binary, 1 indicating I was excited to work on the task and 0 if it was a chore.

In [70]:

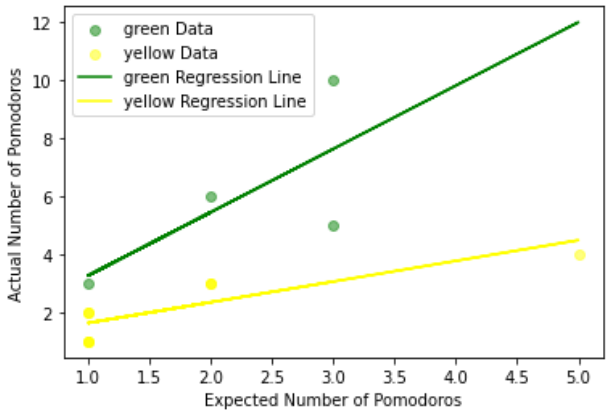
```
DataZ = [1,0,0,0,0,0,1,1,0,0,0,0,1,0,0]

# Create a colormap
colors = ['green' if z == 1 else 'orange' for z in DataZ]

# Plotting the data
plt.scatter(DataX, DataY, c=colors, alpha=0.5)
plt.xlabel("Expected Number of Pomodoros")
plt.ylabel("Actual Number of Pomodoros")
plt.title('Expected and predicted pomodoros, green for exciting tasks and yellow for cl

plt.show()
```




I am a little surprised to see that most of my tasks feel like a chore to me. I was expecting to find more of my tasks genuinely exciting. I should (at some other point in time) figure out why that is and see if I can figure out a way to make more of my tasks feel exciting.

In [71]:

```
# Filter data based on DataZ values
green_indices = [i for i, z in enumerate(DataZ) if z == 1]
yellow_indices = [i for i, z in enumerate(DataZ) if z == 0]

# Fit separate linear regression models for red and blue points
model_green = LinearRegression()
model_yellow = LinearRegression()

model_green.fit(DataX[green_indices], DataY[green_indices])
model_yellow.fit(DataX[yellow_indices], DataY[yellow_indices])

# Make predictions
y_pred_green = model_green.predict(DataX)
y_pred_yellow = model_yellow.predict(DataX)

# Calculate mean squared error
mse_green = mean_squared_error(DataY[green_indices], y_pred_green[green_indices])
mse_yellow = mean_squared_error(DataY[yellow_indices], y_pred_yellow[yellow_indices])

# Visualize the data and regression lines
plt.scatter(DataX[green_indices], DataY[green_indices], color='green', alpha=0.5, labe
plt.scatter(DataX[yellow_indices], DataY[yellow_indices], color='yellow', alpha=0.5, la
plt.plot(DataX, y_pred_green, color='green', label="green Regression Line")
plt.plot(DataX, y_pred_yellow, color='yellow', label="yellow Regression Line")
plt.xlabel("Expected Number of Pomodoros")
plt.ylabel("Actual Number of Pomodoros")
plt.legend()
plt.show()

```

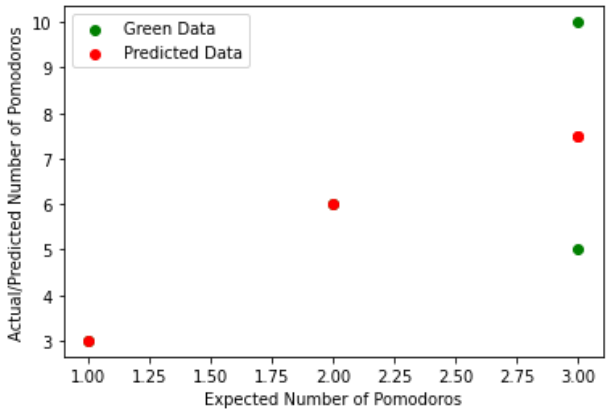


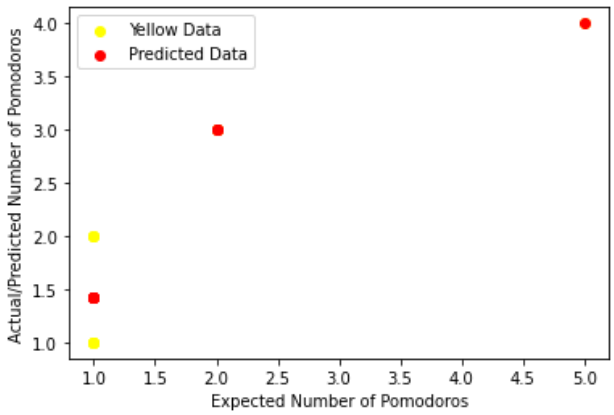

Oh wow, look at that. Our mean squared error actually reduces to be much lower when I differentiate between tasks that I genuinely enjoy verses tasks that feel like a chore.

In [72]:

```python
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# THE COLORS ARE CHANGED WITHIN THE VARIABLES SO I COULD TELL THE DIFFERENCE BETWEEN TH

# Your data
DataX = np.array([x[0] for x in Total]).reshape(-1, 1)
DataY = np.array([x[1] for x in Total])
DataZ = np.array([1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0])

# Red Data
red_X = DataX[DataZ == 1]
red_Y = DataY[DataZ == 1]

# Blue Data
blue_X = DataX[DataZ == 0]
blue_Y = DataY[DataZ == 0]

# Polynomial degree
degree = 2

# Create polynomial features for red data
poly_features_red = PolynomialFeatures(degree=degree)
red_X_poly = poly_features_red.fit_transform(red_X)
red_model = LinearRegression()
red_model.fit(red_X_poly, red_Y)
red_Y_pred = red_model.predict(red_X_poly)

# Create polynomial features for blue data
poly_features_blue = PolynomialFeatures(degree=degree)
blue_X_poly = poly_features_blue.fit_transform(blue_X)
blue_model = LinearRegression()
blue_model.fit(blue_X_poly, blue_Y)
blue_Y_pred = blue_model.predict(blue_X_poly)

# Visualize the data and predicted values for red data
plt.scatter(red_X, red_Y, color='green', label="Green Data")
plt.scatter(red_X, red_Y_pred, color='red', label="Predicted Data")
plt.xlabel("Expected Number of Pomodoros")
plt.ylabel("Actual/Predicted Number of Pomodoros")
plt.legend()
plt.show()

# Visualize the data and predicted values for blue data
plt.scatter(blue_X, blue_Y, color='yellow', label="Yellow Data")
plt.scatter(blue_X, blue_Y_pred, color='red', label="Predicted Data")
plt.xlabel("Expected Number of Pomodoros")
plt.ylabel("Actual/Predicted Number of Pomodoros")
plt.legend()
plt.show()


# Calculate RMSE and MAE for red data
rmse_red = np.sqrt(mean_squared_error(red_Y, red_Y_pred))
mae_red = np.mean(np.abs(red_Y - red_Y_pred))

# Calculate RMSE and MAE for blue data
rmse_blue = np.sqrt(mean_squared_error(blue_Y, blue_Y_pred))
mae_blue = np.mean(np.abs(blue_Y - blue_Y_pred))

# Print the regression coefficients and mean squared errors for red data
print("Green Data:")
print("Intercept:", red_model.intercept_)
print("Coefficient:", red_model.coef_)
print("Mean Squared Error:", mean_squared_error(red_Y, red_Y_pred))
print("Root Mean Squared Error:", rmse_red)
print("Mean Absolute Error:", mae_red)

# Print the regression coefficients and mean squared errors for blue data
print("\nYellow Data:")
print("Intercept:", blue_model.intercept_)
print("Coefficient:", blue_model.coef_)
print("Mean Squared Error:", mean_squared_error(blue_Y, blue_Y_pred))
```

```python
print("Root Mean Squared Error:", rmse_blue)
print("Mean Absolute Error:", mae_blue)



# Calculate average RMSE and MAE
avg_rmse = (rmse_red + rmse_blue) / 2
avg_mae = (mae_red + mae_blue) / 2

# Print the average errors
print("\nAverage RMSE:", avg_rmse)
print("Average MAE:", avg_mae)

```



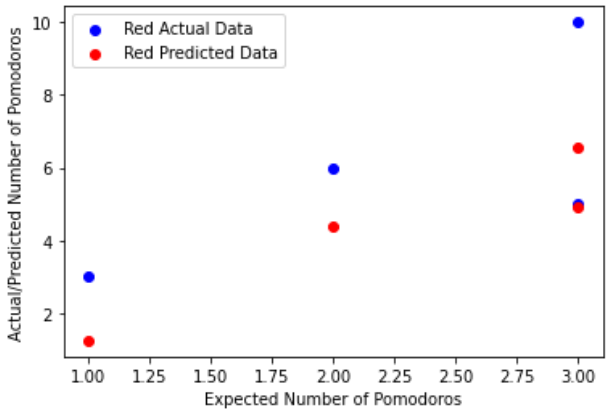

```
Green Data:
Intercept: -1.4999999999999636
Coefficient: [ 0.    5.25 -0.75]
Mean Squared Error: 3.125
Root Mean Squared Error: 1.7677669529663689
Mean Absolute Error: 1.2500000000000029

Yellow Data:
Intercept: -0.7619047619047592
Coefficient: [ 0.          2.5        -0.30952381]
Mean Squared Error: 0.15584415584415587
Root Mean Squared Error: 0.3947710169758614
Mean Absolute Error: 0.31168831168831224

Average RMSE: 1.0812689849711152
Average MAE: 0.7808441558441576
```

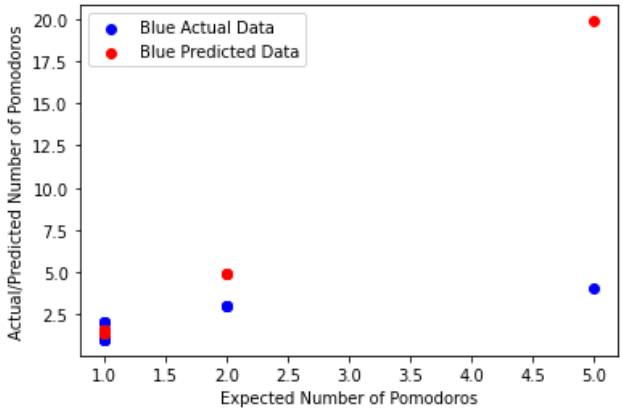
This is giving us even better error values, and I am willing to trusting our model more and more.

In [73]:

```
# Your data
DataX = np.array([x[0] for x in Total]).reshape(-1, 1)
DataY = np.array([x[1] for x in Total])
DataZ = np.array([1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0])

# Initialize lists to store the predicted and actual values
red_pred_loocv = []
red_actual_loocv = []
blue_pred_loocv = []
blue_actual_loocv = []

loo = LeaveOneOut()
for train_index, test_index in loo.split(DataX):
    X_train, X_test = DataX[train_index], DataX[test_index]
    y_train, y_test = DataY[train_index], DataY[test_index]
    z_test = DataZ[test_index]

    poly_features = PolynomialFeatures(degree=2)
    X_train_poly = poly_features.fit_transform(X_train)
    X_test_poly = poly_features.transform(X_test)

    model = LinearRegression()
    model.fit(X_train_poly, y_train)

    y_pred = model.predict(X_test_poly)

    if z_test == 1:
        red_pred_loocv.append(y_pred[0])
        red_actual_loocv.append(y_test[0])
    else:
        blue_pred_loocv.append(y_pred[0])
        blue_actual_loocv.append(y_test[0])

# Convert lists to NumPy arrays
red_pred_loocv = np.array(red_pred_loocv).reshape(-1, 1)
red_actual_loocv = np.array(red_actual_loocv).reshape(-1, 1)
blue_pred_loocv = np.array(blue_pred_loocv).reshape(-1, 1)
blue_actual_loocv = np.array(blue_actual_loocv).reshape(-1, 1)

# Plotting the red LOOCV predictions and actual values
plt.scatter(DataX[DataZ == 1], red_actual_loocv, color='blue', label="Red Actual Data"
plt.scatter(DataX[DataZ == 1], red_pred_loocv, color='red', label="Red Predicted Data"
plt.xlabel("Expected Number of Pomodoros")
plt.ylabel("Actual/Predicted Number of Pomodoros")
plt.legend()
plt.show()

# Plotting the blue LOOCV predictions and actual values
plt.scatter(DataX[DataZ == 0], blue_actual_loocv, color='blue', label="Blue Actual Data
plt.scatter(DataX[DataZ == 0], blue_pred_loocv, color='red', label="Blue Predicted Data
plt.xlabel("Expected Number of Pomodoros")
plt.ylabel("Actual/Predicted Number of Pomodoros")
plt.legend()
plt.show()

# Calculate mean squared error for LOOCV for red and blue data
red_mse_loocv = mean_squared_error(red_actual_loocv, red_pred_loocv)
blue_mse_loocv = mean_squared_error(blue_actual_loocv, blue_pred_loocv)

# Calculate RMSE and MAE for red data
red_rmse_loocv = np.sqrt(red_mse_loocv)
red_mae_loocv = np.mean(np.abs(red_actual_loocv - red_pred_loocv))

# Calculate RMSE and MAE for blue data
blue_rmse_loocv = np.sqrt(blue_mse_loocv)
blue_mae_loocv = np.mean(np.abs(blue_actual_loocv - blue_pred_loocv))


# Calculate average RMSE and MAE
avg_rmse = (red_rmse_loocv + blue_rmse_loocv) / 2
avg_mae = (red_mae_loocv + blue_mae_loocv) / 2

# Print the average errors
print("\nAverage RMSE:", avg_rmse)
```

```
print("Average MAE:", avg_mae)
```

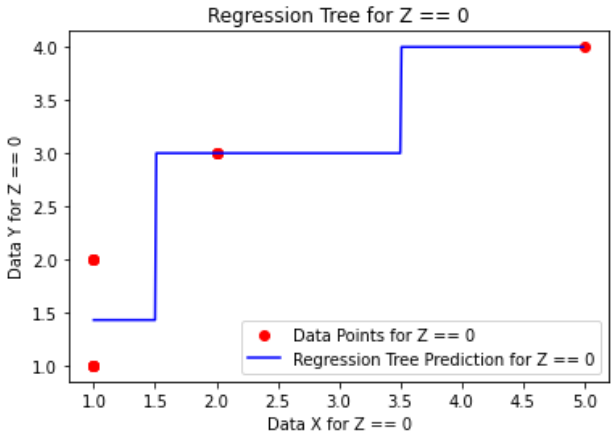


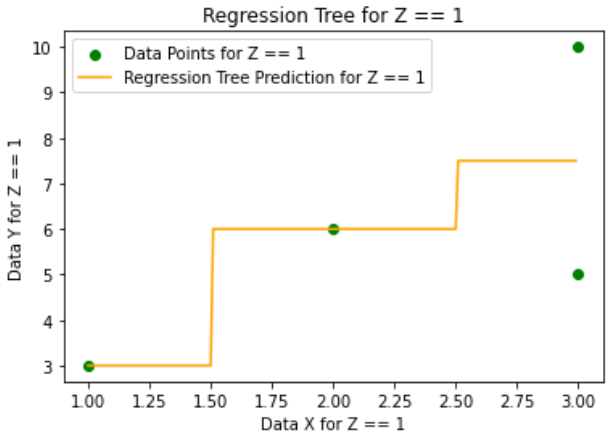




```
Average RMSE: 3.9134020704410872
Average MAE: 2.4110211329568534
```

In [74]:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor

z = np.array([1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0])
DataX = np.array([x[0] for x in Total]).reshape(-1, 1)
DataY = np.array([x[1] for x in Total])

# Create and fit the first regression tree for z == 0
reg_tree_0 = DecisionTreeRegressor(random_state=0)
reg_tree_0.fit(DataX[z == 0], DataY[z == 0])

# Create and fit the second regression tree for z == 1
reg_tree_1 = DecisionTreeRegressor(random_state=0)
reg_tree_1.fit(DataX[z == 1], DataY[z == 1])

# Visualize the first regression tree for z == 0
X_grid_0 = np.arange(min(DataX[z == 0]), max(DataX[z == 0]), 0.01).reshape(-1, 1)
plt.scatter(DataX[z == 0], DataY[z == 0], color='red', label='Data Points for Z == 0')
plt.plot(X_grid_0, reg_tree_0.predict(X_grid_0), color='blue', label='Regression Tree f
plt.title('Regression Tree for Z == 0')
plt.xlabel('Data X for Z == 0')
plt.ylabel('Data Y for Z == 0')
plt.legend()
plt.show()

# Visualize the second regression tree for z == 1
X_grid_1 = np.arange(min(DataX[z == 1]), max(DataX[z == 1]), 0.01).reshape(-1, 1)
plt.scatter(DataX[z == 1], DataY[z == 1], color='green', label='Data Points for Z == 1
plt.plot(X_grid_1, reg_tree_1.predict(X_grid_1), color='orange', label='Regression Tre
plt.title('Regression Tree for Z == 1')
plt.xlabel('Data X for Z == 1')
plt.ylabel('Data Y for Z == 1')
plt.legend()
plt.show()

from sklearn.metrics import mean_squared_error, mean_absolute_error

# Make predictions for z == 0
y_pred_0 = reg_tree_0.predict(DataX[z == 0])
rmse_0 = np.sqrt(mean_squared_error(DataY[z == 0], y_pred_0))
mae_0 = mean_absolute_error(DataY[z == 0], y_pred_0)
print(f"For Z == 0 - RMSE: {rmse_0:.2f}")
print(f"For Z == 0 - MAE: {mae_0:.2f}")

# Make predictions for z == 1
y_pred_1 = reg_tree_1.predict(DataX[z == 1])
rmse_1 = np.sqrt(mean_squared_error(DataY[z == 1], y_pred_1))
mae_1 = mean_absolute_error(DataY[z == 1], y_pred_1)
print(f"For Z == 1 - RMSE: {rmse_1:.2f}")
print(f"For Z == 1 - MAE: {mae_1:.2f}")


# Calculate average RMSE and MAE
avg_rmse = (rmse_0 + rmse_1) / 2
avg_mae = (mae_0 + mae_1) / 2

# Print the average errors
print("\nAverage RMSE:", avg_rmse)
print("Average MAE:", avg_mae)
```

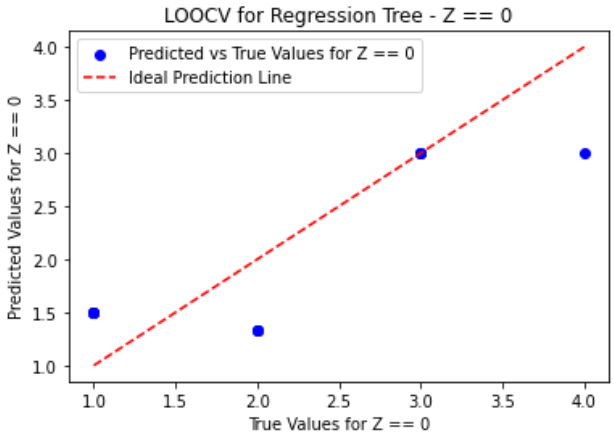


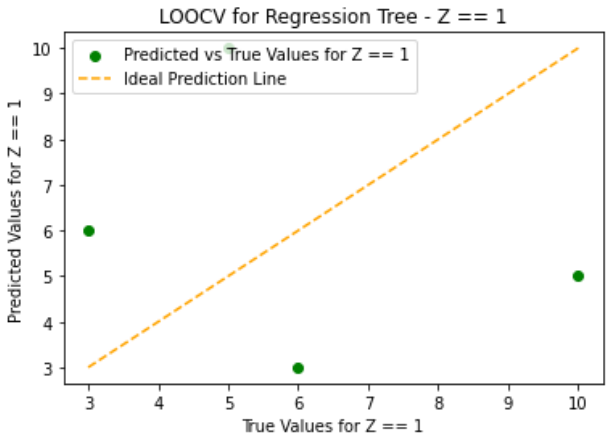

```
For Z == 0 - RMSE: 0.39
For Z == 0 - MAE: 0.31
For Z == 1 - RMSE: 1.77
For Z == 1 - MAE: 1.25

Average RMSE: 1.0812689849711152
Average MAE: 0.7808441558441559
```

In [75]:

```python
import matplotlib.pyplot as plt

# LOOCV for z == 0
loo = LeaveOneOut()
y_true, y_pred = [], []

for train_index, test_index in loo.split(DataX[z == 0]):
    X_train, X_test = DataX[z == 0][train_index], DataX[z == 0][test_index]
    y_train, y_test = DataY[z == 0][train_index], DataY[z == 0][test_index]

    reg_tree_0_cv = DecisionTreeRegressor(random_state=0)
    reg_tree_0_cv.fit(X_train, y_train)

    y_pred.extend(reg_tree_0_cv.predict(X_test))
    y_true.extend(y_test)

# Visualize LOOCV for z == 0
plt.scatter(y_true, y_pred, color='blue', label='Predicted vs True Values for Z == 0')
plt.plot([min(y_true), max(y_true)], [min(y_true), max(y_true)], color='red', linestyle
plt.title('LOOCV for Regression Tree - Z == 0')
plt.xlabel('True Values for Z == 0')
plt.ylabel('Predicted Values for Z == 0')
plt.legend()
plt.show()

# Compute evaluation metrics for LOOCV - z == 0
rmse_0_cv = np.sqrt(mean_squared_error(y_true, y_pred))
mae_0_cv = mean_absolute_error(y_true, y_pred)
print(f"LOOCV - Z == 0 - RMSE: {rmse_0_cv:.2f}")
print(f"LOOCV - Z == 0 - MAE: {mae_0_cv:.2f}")

# LOOCV for z == 1
y_true, y_pred = [], []

for train_index, test_index in loo.split(DataX[z == 1]):
    X_train, X_test = DataX[z == 1][train_index], DataX[z == 1][test_index]
    y_train, y_test = DataY[z == 1][train_index], DataY[z == 1][test_index]

    reg_tree_1_cv = DecisionTreeRegressor(random_state=0)
    reg_tree_1_cv.fit(X_train, y_train)

    y_pred.extend(reg_tree_1_cv.predict(X_test))
    y_true.extend(y_test)

# Visualize LOOCV for z == 1
plt.scatter(y_true, y_pred, color='green', label='Predicted vs True Values for Z == 1'
plt.plot([min(y_true), max(y_true)], [min(y_true), max(y_true)], color='orange', lines
plt.title('LOOCV for Regression Tree - Z == 1')
plt.xlabel('True Values for Z == 1')
plt.ylabel('Predicted Values for Z == 1')
plt.legend()
plt.show()

# Compute evaluation metrics for LOOCV - z == 1
rmse_1_cv = np.sqrt(mean_squared_error(y_true, y_pred))
mae_1_cv = mean_absolute_error(y_true, y_pred)
print(f"LOOCV - Z == 1 - RMSE: {rmse_1_cv:.2f}")
print(f"LOOCV - Z == 1 - MAE: {mae_1_cv:.2f}")

# Calculate average RMSE and MAE
avg_rmse = (rmse_0_cv + rmse_1_cv) / 2
avg_mae = (mae_0_cv + mae_1_cv) / 2

# Print the average errors
print("\nAverage RMSE:", avg_rmse)
print("Average MAE:", avg_mae)
```




```
LOOCV - Z == 0 - RMSE: 0.55
LOOCV - Z == 0 - MAE: 0.45
```




```
LOOCV - Z == 1 - RMSE: 4.12
LOOCV - Z == 1 - MAE: 4.00

Average RMSE: 2.3367937540904204
Average MAE: 2.227272727272727
```

I was initially expecting LOOCV to do much better than testing and training since we have a small data-set and LOOCV is usually recommended for smaller data-sets, but it not performing as well which can be justified by the high variability in our data and LOOCV being more prone to overfitting and bad with highly variable data.

## Section 9:

I defined the problem and my idea of what the solution would be I started out by plotting the data I manually input. Then the next steps followed this order

Regressions:

Linear Regression (80/20)

LOOCV

Regression Tree (80/20)

LOOCV

Not happy, split the data

Linear Regression (80/20)

LOOCV

Regression Tree (80/20)

LOOCV

Each step was followed by a visualization and a RMSE and MSE. Here is a chart detailing them all.

| Method | RMSE | MAE |
|---|---|---|
| Before splitting: Linear Regression (80/20 split) | 1.8614445587797204 | 3.0288461538461533 |
| Before splitting: LOOCV on Linear Regression | 4.465143637755054 | 2.3672462021880403 |
| Before splitting: Regression Tree (80/20 split) | 0.7142857142857142 | 0.7142857142857142 |
| Before splitting: LOOCV on Regression Tree | 1.68 | 2.30 |
| After splitting: Linear Regression (80/20 split) | 1.0812689849711152 | 0.7808441558441576 |
| After splitting: LOOCV on Linear Regression | 3.9134020704410872 | 2.4110211329568534 |
| After splitting: Regression Tree (80/20 split) | 1.0812689849711152 | 0.7808441558441559 |
| After splitting: LOOCV on Regression Tree | 2.3367937540904204 | 2.227272727272727 |

Surprisingly the results our RMSE and MAE are the lowest for the regression tree before we split our data into tasks that I am excited to work on and tasks that I am not excited to work on. I was hoping for the split to be more helpful but it was not. It is also important to note that I averaged the results after the split so in some instances the RMSE and MAE values might be lower for one of the splits and higher for the other.

### Proper predictor model:

In [76]:
```python
# Assuming you have X_train, X_test, y_train, y_test already defined

# Create and fit the regression tree on the training data
reg_tree_split = DecisionTreeRegressor(random_state=0)
reg_tree_split.fit(X_train, y_train)

def predict_pomodoros(input_value):
    predicted_pomodoros = reg_tree_split.predict(np.array(input_value).reshape(1, -1))
    return predicted_pomodoros[0]

# Sample usage of the function
user_input = float(input("Enter the estimated number of pomodoros: "))
predicted_value = predict_pomodoros(user_input)
print(f"The predicted number of pomodoros is: {predicted_value}")

```

```
Enter the estimated number of pomodoros: 1
The predicted number of pomodoros is: 3.0
```

A problem with our predictor model is that it gives us, bad results when we input 5 or more pomodoros since the training data on it, is just one data point. This is why it should be used with caution and shouldn't be used to estimate more than 5 pomodoros. Ideally, if you follow the rules of the pomodoro, if you have a task that is estimated to take that long, then you are supposed break it up into two seperate tasks to keep things manageable.

I like to think that this predictor is almost like Thor's hammer (which only Thor could pick up) in the sense that it only works for people who respect the rule book of the pomodoro and don't bite off more than they can chew, and thus they are deserving of getting better time estimates for their valuable time.

### Section 10:

References:


https://www.w3schools.com/python/python_lists.asp

https://www.datarobot.com/blog/how-much-data-is-needed-to-train-a-good-model/#:~:text=For%20example%2C%20if%20you%20have,observations%20to%20train%20a%20model

https://betterprogramming.pub/emojis-as-python-variables-sure-why-not-96ce955dada1


### AI statement:

I did use Chat GPT to help with some of the coding syntax, but I do have conceptual understanding of the code, I commented the code myself to make it more clear and I didn't use AI to write any of the explainations.