**Function 1**

1. **Data Loading:**
   * The dataset is loaded from a CSV file (**energy\_performance.csv** ) using Pandas.
2. **Feature-Target Split:**
   * The dataset is split into features and targets. Features are the input variables used for prediction, and targets are the output variables to be predicted.
3. **Statistical Analysis:**
   * Descriptive statistics, such as minimum and maximum values of heating and cooling loads, are calculated to gain insights into the data.

**Function 2**

**Explanation of the Code:**

* **Polynomial Model Function:**
  + **calculate\_polynomial\_model** function:
    - **feature\_matrix = np.column\_stack([np.power(features, d) for d in range(degree + 1)])**: Creates a feature matrix with polynomial terms up to the specified degree.
    - **return np.dot(feature\_matrix, coefficients)**: Calculates the dot product of the feature matrix and coefficients to obtain the estimated target vector.
* **Parameter Vector Size Determination:**
  + **determine\_parameter\_vector\_size** function:
    - **return (degree + 1) \* num\_features**: Determines the size of the parameter vector based on the degree of the polynomial and the number of features.
* **Example Usage:**
  + **degree\_of\_polynomial = 2**: Assumed degree of the polynomial for illustration.
  + **num\_features = features.shape[1]**: Retrieves the number of features from the dataset.
  + **coefficients = np.random.rand(determine\_parameter\_vector\_size(degree\_of\_polynomial, num\_features))**: Initializes random coefficients with the determined size.
  + **estimated\_target = calculate\_polynomial\_model(degree\_of\_polynomial, features.values, coefficients)**: Calculates the estimated target using the polynomial model function.

**Function 3**

**Explanation of the Code:**

* **Model Function:**
  + **calculate\_model\_function** function:
    - Calculates the value of the model function for a given set of features and coefficients.
* **Jacobian Calculation:**
  + **calculate\_jacobian** function:
    - Calculates the Jacobian matrix using the features, coefficients, and the degree of the polynomial.
* **Linearization Function:**
  + **linearize** function:
    - Calls **calculate\_jacobian** to obtain the estimated target and Jacobian at the linearization point.

**Function 4**

**Explanation of the Code:**

* **Normal Equation Matrix Calculation:**
  + **normal\_eq\_matrix = np.dot(jacobian.T, jacobian)**: Calculates the normal equation matrix by taking the dot product of the transposed Jacobian matrix and the original Jacobian matrix.
* **Regularization:**
  + **reg\_lambda = 0.01**: Sets the regularization parameter.
  + **reg\_matrix = reg\_lambda \* np.identity(normal\_eq\_matrix.shape[0])**: Creates a regularization matrix.
  + **normal\_eq\_matrix += reg\_matrix**: Adds the regularization term to the normal equation matrix.
* **Residuals Calculation:**
  + **residuals = training\_targets - estimated\_targets**: Computes the residuals by subtracting the estimated targets from the actual training targets.
* **Building the Normal Equation System:**
  + **right\_hand\_side = np.dot(jacobian.T, residuals)**: Builds the right-hand side of the normal equation system using the transposed Jacobian matrix and residuals.
* **Solving the Normal Equation System:**
  + **optimal\_parameter\_update = np.linalg.solve(normal\_eq\_matrix, right\_hand\_side)**: Solves the normal equation system to obtain the optimal parameter update vector.

**Function 5**

**Explanation of the Code:**

* **Initialization:**
  + **coefficients = np.zeros(determine\_parameter\_vector\_size(degree, num\_features))**: Initializes the parameter vector of coefficients with zeros.
* **Iterative Procedure:**
  + **for iteration in range(num\_iterations):**: Sets up a loop for the specified number of iterations.
  + **estimated\_targets, jacobian = linearize(features, coefficients, degree)**: Linearizes the model around the current coefficients.
  + **optimal\_parameter\_update = calculate\_optimal\_parameter\_update(training\_targets, estimated\_targets, jacobian)**: Calculates the optimal parameter update based on the linearized model.
  + **coefficients += optimal\_parameter\_update**: Updates the coefficients using the calculated optimal update.
* **Convergence Check:**
  + **if np.linalg.norm(optimal\_parameter\_update) < tol:**: Checks for convergence by comparing the norm of the optimal parameter update to a tolerance value (**tol**).
  + The loop breaks if convergence is achieved.

**Function 6**

**Explanation of the Code:**

* **Iterate Over Polynomial Degrees:**
  + **for degree in degrees:**: Iterate through the specified degrees.
  + **kf = KFold(n\_splits=5, shuffle=True, random\_state=42)**: Create a KFold object with 5 splits for cross-validation, shuffling the data, and setting a random seed for reproducibility.
  + **absolute\_diff = []**: Initialize an empty list to store absolute differences for each fold.
* **Cross-Validation Loop:**
  + **for train\_index, test\_index in kf.split(features):**: Loop over each fold, splitting the data into training and test sets.
  + **train\_features, test\_features = features.iloc[train\_index], features.iloc[test\_index]**: Extract features for training and test sets.
  + **train\_targets, test\_targets = targets.iloc[train\_index], targets.iloc[test\_index]**: Extract targets for training and test sets.
  + **coefficients = regression(degree, train\_features, train\_targets[load\_type + ' load'])**: Perform regression on the training data to obtain coefficients.
  + **predicted\_values = calculate\_model\_function(test\_features.values, coefficients)**: Predict values on the test set using the model function.
  + **absolute\_diff.append(mean\_absolute\_error(test\_targets[load\_type + ' load'], predicted\_values))**: Calculate and store mean absolute differences for each fold.
* **Results Storage:**
  + **mean\_absolute\_diff = np.mean(absolute\_diff)**: Calculate the mean of absolute differences across all folds.
  + **results.append({'Degree': degree, 'Mean Absolute Difference': mean\_absolute\_diff})**: Append a dictionary to the results list containing the degree and mean absolute difference.

**Function 7**

**Explanation of the Code:**

* **Optimal Degrees Selection:**
  + **optimal\_degree\_heating = model\_selection(degrees, features, targets[['Heating load']])**: Selects the optimal degree for heating load using cross-validation.
  + **optimal\_degree\_cooling = model\_selection(degrees, features, targets[['Cooling load']])**: Selects the optimal degree for cooling load using cross-validation.
* **Parameter Estimation:**
  + **coefficients\_heating = regression(optimal\_degree\_heating, features, targets['Heating load'])**: Estimates model parameters for heating load using the selected optimal degree.
  + **coefficients\_cooling = regression(optimal\_degree\_cooling, features, targets['Cooling load'])**: Estimates model parameters for cooling load using the selected optimal degree.
* **Predictions:**
  + **predicted\_heating = calculate\_model\_function(features.values, coefficients\_heating)**: Predicts heating loads for the entire dataset.
  + **predicted\_cooling = calculate\_model\_function(features.values, coefficients\_cooling)**: Predicts cooling loads for the entire dataset.
* **Visualization:**
  + The **plt.scatter** function is used to create a scatter plot with true loads on one axis and estimated loads on the other.