MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Regression & optimisation

Jesús Fernández López
4º Software Development
MACHINE LEARNING

·Index:

·Introduction:

Regression is a fundamental technique in statistical learning and machine learning used to model and analyze the relationship between a set of input variables (features) and a continuous output variable (target). It is a supervised learning method, meaning the model learns from labeled data, where the true output values are known. The goal is to find a mathematical function that maps inputs to outputs as accurately as possible, enabling predictions for unseen data. In this assignment, regression is applied to predict the heating and cooling energy loads of buildings based on their physical characteristics.

The task involves building a polynomial regression model, which extends the basic linear regression approach by considering polynomial relationships between features and the target. While linear regression assumes a direct proportionality, polynomial regression captures more complex, non-linear patterns. For instance, the energy efficiency of a building might not increase linearly with changes in its glazing area or height. By incorporating polynomial terms, we can model such intricate dependencies, providing more precise predictions.

Optimization, on the other hand, is a mathematical process used to adjust the model's parameters (coefficients) to minimize the error between predicted and actual outputs. In the context of regression, optimization aims to find the best-fitting coefficients for the polynomial function by minimizing a loss function, typically the sum of squared errors. This involves iterative procedures that refine the parameters to converge on an optimal solution.

The interplay of regression and optimization is central to this assignment. The regression model relies on optimization techniques to iteratively improve its accuracy through parameter updates. For example, linearization methods and Jacobian matrices help transform the problem into a solvable linear system, while the normal equation and regularization ensure numerical stability and prevent overfitting.

This project also introduces the concept of model selection and evaluation, which are critical for achieving robust predictions. Through cross-validation, the optimal polynomial degree for each energy load type (heating and cooling) is determined by balancing accuracy and complexity. This prevents the model from being too simplistic (underfitting) or overly complex (overfitting). Finally, the results are visualized and compared against actual data, demonstrating the model's performance and its alignment with real-world energy demands.

In summary, this assignment combines regression and optimization techniques to tackle a practical problem in building energy performance estimation. It highlights the importance of mathematical modeling, iterative refinement, and systematic evaluation in developing predictive tools for complex systems.

·Task 1: Input data

Load the data from the file and split it into features and targets. Determine and output the minimum and maximum heating and cooling loads of buildings present in the dataset.

The primary goal of Task 1 is to prepare the data for subsequent analysis and modeling. To achieve this, the dataset is first loaded from a CSV file using the pandas library. The dataset contains details about building characteristics alongside their heating and cooling energy requirements. Loading the dataset into a DataFrame allows us to easily manipulate and extract the required information for further steps (lines 8 - 19).

Once the data is loaded, it is split into two key components: features and targets. The features correspond to the first eight columns, which describe various physical characteristics of the buildings, such as surface area and height. These features act as input variables for the regression model. The targets, on the other hand, are the last two columns, labeled "Heating Load" and "Cooling Load." These represent the energy requirements for heating and cooling, respectively, and serve as the output variables our model aims to predict (lines 21 - 23).

Lastly, the minimum and maximum values for both heating and cooling loads are calculated. This step provides insight into the range and distribution of the target variables, which is crucial for understanding the data and for later evaluating the model's performance (lines 25 - 42).

```
17. def task_1_load_and_analyze_data():
18.     # Step 1: Load the dataset
19.     data = pd.read_csv("energy_performance.csv")
20. 
21.     # Step 2: Separate features and targets
22.     features = data.iloc[:, :-2]  # Extract all columns except the last two
23.      targets = data.iloc[:, -2:]  # Extract the last two columns (Heating and
    Cooling loads)
24. 
25.     # Step 3: Determine minimum and maximum values for heating and cooling loads
26.     heating_min = targets["Heating load"].min()
27.     heating_max = targets["Heating load"].max()
28.     cooling_min = targets["Cooling load"].min()
29.     cooling_max = targets["Cooling load"].max()
30. 
31.     # Prepare min-max results in a dictionary for better readability
32.     min_max_values = {
33.         "Heating Load": {"min": heating_min, "max": heating_max},
34.         "Cooling Load": {"min": cooling_min, "max": cooling_max}
35.     }
36. 
37.     # Output the results for inspection
38.     print("Minimum and Maximum Heating and Cooling Loads:")
39.     print(min_max_values)
40. 
41.     return features, targets, min_max_values
```

·Task 2: Model function

Create a polynomial model function that takes as input parameters the degree of the polynomial, a list of feature vectors as extracted in task 1, and a parameter vector of coefficients and calculates the estimated target vector using a multivariate polynomial of the specified degree . Create a second function that determines the correct size for the parameter vector from the degree of the multivariate polynomial.

Task 2 focuses on building the foundation of the regression model by implementing the logic for a polynomial regression. The task is broken into two clear objectives: calculating the predicted target values using a polynomial model and determining the correct size of the coefficient vector for a given degree of the polynomial and number of input features. Here's a detailed explanation of how these functions work and how they connect.

-The Polynomial Model Function (line 50 - 81)

The polynomial_model function is the heart of this task. It implements the process of creating a design matrix, which encodes all polynomial terms of the input features up to the specified degree. This design matrix is a key component in polynomial regression, as it transforms the input features into a higher-dimensional space where the relationships between features and targets can be better captured.

For example, if the input has two features x_1 and x_2 and the polynomial degree is 2, the design matrix will include the terms:

$$1, x_1, x_2, x_1^2, x_2^2, x_1 x_2$$

The function iterates through all possible combinations of feature powers using Python's np.ndindex, ensuring that the sum of the powers in each term equals the current polynomial degree. This systematic approach guarantees that all valid terms are included, capturing the complexity of the relationships in the data.

Once the design matrix is built, the function multiplies it with the coefficient vector to compute the predicted target values. Before this step, the size of the coefficient vector is validated against the number of terms in the design matrix, ensuring consistency and avoiding runtime errors.

-The Coefficient Size Calculation Function (line 83- 96)

The calculate_coefficient_size function determines how many coefficients are required for the polynomial model. The number of terms in a polynomial of degree d with n features is given by the formula for combinations with repetition:

$$size = \binom{d+n}{n}$$

This ensures that the model includes all possible combinations of features and polynomial terms up to the specified degree. By separating this logic into its own function, we ensure modularity and clarity, allowing this calculation to be reused wherever needed in the project.

-Integrating with Task 1 (line 99 - 122)

The wrapper function task_2_polynomial_model demonstrates how these two core functions work together. It uses the feature data extracted in Task 1 and calculates the required size of the coefficient vector. A random coefficient vector is generated for demonstration purposes, and predictions are computed using the polynomial model.

By integrating the features directly from Task 1, this function shows how the polynomial model adapts to real data, ensuring compatibility across tasks. This design ensures a smooth workflow for future tasks, where optimized coefficients will replace the randomly generated ones.

Separating the logic into two functions aligns with the assignment's requirements and promotes clean code design. The polynomial_model function focuses solely on prediction, while the calculate_coefficient_size function isolates the mathematical complexity of determining the required number of coefficients. This modular approach improves readability, reusability, and maintainability of the code.

-Practical Use Case

Polynomial regression is particularly useful in capturing non-linear relationships between features and targets. In this assignment, the building's physical characteristics (features) and its heating or cooling load (targets) may not follow a simple linear relationship. By using a polynomial model, we account for these non-linearities, enabling more accurate predictions. For instance, the energy efficiency of a building might depend quadratically or interactively on its height and surface area, patterns which this model can capture effectively.

```
61. def polynomial_model(degree, features, coefficients):
62.     # Step 1: Create the design matrix for the polynomial terms
63.     n_samples, n_features = features.shape
64.     design_matrix = np.ones((n_samples, 1))  # Start with the bias term (intercept)
65.
66.     # Generate polynomial terms for all feature combinations
67.     for d in range(1, degree + 1):
68.         for combination in np.ndindex(*(d + 1,) * n_features):
69.             if sum(combination) == d:
70.                 term = np.prod(features ** np.array(combination), axis=1, keepdims=True)
71.                 design_matrix = np.hstack((design_matrix, term))
72.
73.     # Step 2: Validate coefficient vector size
74.     expected_size = design_matrix.shape[1]
75.     assert coefficients.shape[0] == expected_size, (
76.         f"Coefficient vector size mismatch. Expected {expected_size}, got {coefficients.shape[0]}"
77.     )
78.
```

```
    # Step 3: Calculate the predicted values
    predictions = np.dot(design_matrix, coefficients)
    return predictions
```

```
def calculate_coefficient_size(degree, n_features):
    from math import comb
    size = comb(degree + n_features, n_features)
    return size
```

```
def task_2_polynomial_model(features, degree):
    # Convert features from DataFrame to numpy array
    features_array = features.to_numpy()

    # Determine the size of the coefficient vector
    n_features = features_array.shape[1]
    coefficient_size = calculate_coefficient_size(degree, n_features)

    # Generate a random coefficient vector for demonstration
    coefficients = np.random.rand(coefficient_size)

    # Generate predictions using the polynomial model
    predictions = polynomial_model(degree, features_array, coefficients)
    return predictions
```

```
# Task 2 example with degree 2 polynomial
degree = 2
predictions = task_2_polynomial_model(features, degree) # Features given by task1
print("Example Predictions (Task 2):", predictions)
```

·Task 3: Linearization

Create a function that calculates the value of the model function implemented in task 2 and its Jacobian at a given linearization point using the numerical linearisation procedure discussed in the lectures/labs. The function should take the degree of the polynomial, a list of feature vectors as extracted in task 1, and the coefficients of the linearization point as input and calculate the estimated target vector and the Jacobian at the linearization point as output. In your code's comments, clearly indicate and explain where the model function implemented in task 2 is called and why, and where the partial derivatives for the Jacobian are calculated and how.

The objective of Task 3 is to compute the predicted target values and the Jacobian matrix for a polynomial regression model at a given linearization point. The linearization process involves approximating the relationship between the model's parameters (coefficients) and the predicted outputs using the Jacobian matrix, which measures how sensitive the predictions are to changes in the coefficients. This step is crucial for optimization procedures, as it enables iterative updates to the model's parameters based on how they influence the predicted outputs.

In the implementation, the polynomial model logic from Task 2 is reused to construct a design matrix. This matrix includes all polynomial terms derived from the input features up to the specified degree. For instance, with two features x_1 and x_2, and a polynomial degree of 2, the design matrix will contain terms such as 1 (the intercept), x_1, x_2, x_1^2, x_2^2 and $x_1 x_2$. Each row of the matrix represents the polynomial expansion of a single data sample. This design matrix forms the foundation for both the predicted target values and the Jacobian matrix.

The predicted target values are calculated by multiplying the design matrix with the coefficient vector. This process directly mirrors the model function implemented in Task 2, where the polynomial regression equation is used to map features to outputs. In Task 3, the function computes predictions at a specific linearization point, represented by the given coefficient vector. This linearization point is essential for optimization tasks, as it provides the starting point for parameter updates.

The Jacobian matrix, on the other hand, encapsulates the partial derivatives of the predicted values with respect to each coefficient. In the context of polynomial regression, these derivatives simplify to the terms in the design matrix. For example, the derivative of a term like $\beta_1 x_1$ with respect to β_1 is x_1. As such, the Jacobian matrix is mathematically equivalent to the design matrix itself. Each row of the Jacobian corresponds to a single data sample, and each column represents the sensitivity of the prediction to a specific coefficient (lines 131 - 167).

By combining the predicted target values and the Jacobian matrix, this function provides the tools necessary for subsequent optimization steps. These outputs allow the model to iteratively adjust its coefficients, improving prediction accuracy. The function's implementation adheres to the requirements by clearly incorporating the model logic from Task 2 and explicitly demonstrating how the Jacobian is derived. Furthermore, detailed comments are provided in the code to explain the purpose of each step, ensuring clarity and alignment with the task objectives (lines 170 - 193)

This step is integral to the overall regression process. The Jacobian matrix not only aids in optimization but also provides insight into the underlying structure of the model, enabling more informed adjustments to its parameters. By linking the polynomial model to its derivatives, Task 3 sets the stage for efficient and accurate optimization in the subsequent tasks.

```
144. def calculate_jacobian(degree, features, coefficients):
145.     # Step 1: Create the design matrix for the polynomial terms
146.     # The design matrix is built by generating all polynomial terms for the features
147.     # using the same procedure as in Task 2's model function.
148.     n_samples, n_features = features.shape
149.       design_matrix = np.ones((n_samples, 1))    # Start with the bias term (intercept)
150.
151.     for d in range(1, degree + 1):
152.         for combination in np.ndindex(*(d + 1,) * n_features):
153.             if sum(combination) == d:
154.                         term = np.prod(features ** np.array(combination), axis=1, keepdims=True)
155.                 design_matrix = np.hstack((design_matrix, term))
156.
157.     # Step 2: Calculate the predicted target values
158.     # The predicted values are obtained by multiplying the design matrix with the coefficients.
159.     # This directly calls the model logic from Task 2, where predictions are based on the polynomial terms.
160.     predictions = np.dot(design_matrix, coefficients)
161.
162.     # Step 3: Calculate the Jacobian matrix
163.       # The Jacobian matrix is equal to the design matrix in polynomial regression. This is because:
164.     # Partial derivatives of the polynomial terms w.r.t. each coefficient are constant and equal to the corresponding term in the design matrix.
165.     jacobian = design_matrix
166.
167.     return predictions, jacobian
```

```
def task_3_linearization_and_jacobian(features, degree):
    # Convert features to numpy array
    features_array = features.to_numpy()

    # Generate a random coefficient vector for demonstration
    n_features = features_array.shape[1]
    coefficient_size = calculate_coefficient_size(degree, n_features)
    coefficients = np.random.rand(coefficient_size)

        # Compute predictions and Jacobian using the linearization point (coefficients)
        predictions, jacobian = calculate_jacobian(degree, features_array, coefficients)
    return predictions, jacobian
```

```
# Task 3 example with degree 2 polynomial
degree = 2
predictions, jacobian = task_3_linearization_and_jacobian(features, degree)
if(False):
    print("Predictions (Task 3):", predictions)
    print("Jacobian Shape (Task 3):", jacobian.shape)
```

·Task 4: Parameter update

Create a function that calculates the optimal parameter update from the training target vector extracted in task 1 and the estimated target vector and Jacobian calculated in task 3 following the procedure discussed during the lectures/labs. To do that, start with calculating the normal equation matrix; make sure that you add a regularisation term to prevent the normal equation system from being singular. Now calculate the residual and build the normal equation system. Solve the normal equation system to obtain the optimal parameter update. The function should take the training target vector and the estimated target vector and Jacobian at the linearization point as input and calculate the optimal parameter update vector as output. In your code's comments, clearly indicate where the normal equation matrix is calculated and how it is regularised. Also indicate exactly where the residuals are calculated and explain how.

Task 4 focuses on calculating the optimal parameter update for the regression model. This process plays a crucial role in refining the model's coefficients to minimize prediction errors and improve its performance. The parameter update is determined using the residuals, the Jacobian matrix, and the normal equation, while incorporating regularization to ensure numerical stability.

The residuals are the starting point for this calculation. They represent the difference between the actual target values (training data) and the estimated target values (predicted by the current model). These residuals serve as a measure of the current error in the model's predictions and guide the adjustment of coefficients to reduce this error. Smaller residuals indicate that the model is aligning more closely with the target values, while larger residuals signal areas where the model needs improvement (lines 204 - 221).

To determine the parameter updates, the normal equation matrix is constructed. This matrix is derived as $J^T J$, where J is the Jacobian matrix representing the sensitivity of predictions to the coefficients. However, direct inversion of this matrix can be problematic due to numerical issues, such as singularity or instability, especially when the features are highly correlated or when the model is overparameterized. To address this, a regularization term, λI, is added to the matrix, where λ is a small positive constant and I is the identity matrix. This regularization ensures that the matrix is invertible and stable, allowing the parameter update to be computed reliably (lines 223 - 227)

The right-hand side of the normal equation is given by $J^T r$, where r is the vector of residuals. This step projects the residuals onto the parameter space, identifying the direction and magnitude of the required updates for each coefficient. The system of equations $\left(J^T J + \lambda I\right)\delta = J^T r$ is then solved for δ, the parameter update vector. This vector provides the adjustments to the coefficients that minimize the residuals, moving the model closer to an optimal solution (lines 229 - 235)

The incorporation of regularization is particularly important in this step. It prevents overfitting by controlling the magnitude of the parameter updates and stabilizes the inversion of the normal equation matrix. Without regularization, the model may become excessively sensitive to noise or computational artifacts, leading to instability in the optimization process.

The output of this task is the parameter update vector. Each value in this vector corresponds to the adjustment needed for a specific coefficient in the polynomial regression model. These updates reflect the influence of the associated polynomial term on reducing the residuals. Applying these updates iteratively, as done in subsequent tasks, leads to progressively smaller residuals and a better-fitting model (lines 241 - 268)

This task is fundamental to the overall regression process. It bridges the gap between the current state of the model and its optimal configuration by leveraging mathematical tools such as the normal equation and regularization. By ensuring stability and guiding the model towards minimizing errors, Task 4 sets the foundation for accurate and efficient regression modeling.

```
216. def calculate_parameter_update(target_vector, estimated_vector, jacobian, regularization_lambda=1e-5):
217.     # Step 1: Calculate residuals
218.     # The residuals are the difference between the actual target values and the estimated values.
219.     residuals = target_vector - estimated_vector
220.     # Explanation: Residuals measure the error of the model at the current linearization point.
221.     # Smaller residuals indicate closer alignment between predictions and true values.
222. 
223.     # Step 2: Build the normal equation matrix
224.     # The normal equation is J.T @ J, where J is the Jacobian.
225.     # A regularization term (lambda * I) is added to ensure numerical stability.
226.     # This prevents the matrix from being singular, especially in cases of overfitting or insufficient data.
227.     normal_matrix = jacobian.T @ jacobian + regularization_lambda * np.eye(jacobian.shape[1])
228. 
229.     # Step 3: Build the right-hand side of the normal equation
230.     # This is J.T @ residuals, representing the projection of residuals onto the parameter space.
231.     rhs = jacobian.T @ residuals
232. 
233.     # Step 4: Solve the normal equation to find the parameter update
234.     # Solving normal_matrix @ delta = rhs for delta, where delta is the parameter update.
235.     parameter_update = np.linalg.solve(normal_matrix, rhs)
236. 
237.     return parameter_update
```

```
def task_4_parameter_update(features, targets, degree, regularization_lambda=1e-5):
    # Convert features and targets to numpy arrays
    features_array = features.to_numpy()
     target_vector = targets["Heating load"].to_numpy()  # Example for Heating Load

    # Generate random coefficients and calculate predictions and Jacobian
    n_features = features_array.shape[1]
    coefficient_size = calculate_coefficient_size(degree, n_features)
     coefficients = np.random.rand(coefficient_size)  # Random initialization for demonstration
      estimated_vector, jacobian = calculate_jacobian(degree, features_array, coefficients)

    # Calculate the parameter update
    parameter_update = calculate_parameter_update(
        target_vector, estimated_vector, jacobian, regularization_lambda
    )
    return parameter_update
```

```
# Task 4 example with degree 2 polynomial
degree = 2
parameter_update = task_4_parameter_update(features, targets, degree)
print("Optimal Parameter Update (Task 4):", parameter_update)
```

·Task 5: Regression

Create a function that calculates the coefficient vector that best fits the training data. To do that, initialise the parameter vector of coefficients with zeros. Then set up an iterative procedure that alternates linearization and parameter update following the approach discussed during the lectures/labs. The function should take the degree of the polynomial, the training data features, and the training data targets as input and return the best fitting polynomial coefficient vector as output. In your code's comments, clearly indicate the parameter vector and how it is updated. How do you expect the parameter update and the residuals calculated in the previous task to evolve in the iterations? How could you use this to determine the number of iterations required?

Task 5 focuses on fitting a polynomial regression model to the training data by iteratively refining the coefficients. The process begins with an initial coefficient vector set to zeros, which represents a baseline where none of the polynomial terms influence the predictions. This initialization is essential for ensuring the optimization process starts without any prior assumptions, allowing the coefficients to evolve entirely based on the data.

The iterative procedure alternates between two key steps: linearization and parameter updates. In the linearization step, the current coefficients are used to calculate the model's predicted target values and the Jacobian matrix. The predicted values represent the model's current output for each data sample, while the Jacobian matrix captures how sensitive these predictions are to changes in the coefficients. These calculations provide the foundation for assessing how well the model is fitting the data at the current step.

The residuals, which measure the difference between the actual target values and the predicted values, guide the parameter updates. Large residuals indicate areas where the model's predictions diverge significantly from the data, signaling the need for adjustment. Using the residuals and the Jacobian, the parameter update is calculated through the normal equation. This equation determines the optimal changes to the coefficients that would reduce the residuals most effectively. A regularization term is included in the normal equation to prevent instability and ensure numerical robustness, particularly when the model has a large number of coefficients or when the features are correlated.

As the coefficients are updated in each iteration, the residuals are expected to decrease progressively. This reflects the model's improving alignment with the data. Simultaneously, the magnitude of the parameter updates diminishes, as the coefficients converge towards their optimal values. These trends indicate that the model is approaching a state where further updates provide negligible improvement, marking the point of convergence.

The iterative procedure is governed by two stopping criteria: convergence tolerance and maximum iterations. The convergence tolerance ensures that the optimization halts once the parameter updates are sufficiently small, signifying that the coefficients have stabilized. The maximum number of iterations acts as a safeguard, preventing the optimization from running indefinitely in cases where convergence is slow or unreachable within the given constraints. Together, these criteria balance computational efficiency and model accuracy (lines 276 - 319).

By iteratively refining the coefficients, Task 5 achieves a polynomial regression model that closely fits the training data. The resulting coefficients encapsulate the contributions of all polynomial terms, providing a final model that minimizes residuals and predicts the target values with high accuracy. This task is a critical step in the regression process, laying the groundwork for evaluating the model's performance and applying it to unseen data in subsequent tasks (323 - 340).

```
291. def fit_polynomial_model(features, targets, degree, max_iterations=1000, tolerance=1e-6, regularization_lambda=1e-5):
292.     # Step 1: Initialize the coefficient vector with zeros
293. n_features = features.shape[1]
294. n_coefficients = calculate_coefficient_size(degree, n_features)
295. coefficients = np.zeros(n_coefficients)  # Initial coefficient vector
296. 
297. for iteration in range(max_iterations):
298.         # Step 3: Calculate predictions and Jacobian matrix at the current coefficients
299.         estimated_vector, jacobian = calculate_jacobian(degree, features, coefficients)
300. 
301.     # Step 4: Calculate residuals
302.     residuals = targets - estimated_vector
303. 
304.     # Step 5: Calculate the parameter update
305.         normal_matrix = jacobian.T @ jacobian + regularization_lambda * np.eye(jacobian.shape[1])
306.     rhs = jacobian.T @ residuals
307.     parameter_update = np.linalg.solve(normal_matrix, rhs)
308. 
309.     # Step 6: Update coefficients
310.     coefficients += parameter_update
311. 
312.     # Step 7: Check for convergence
313.      # If the magnitude of the parameter update is smaller than the tolerance, stop the iterations
314.     if np.linalg.norm(parameter_update) < tolerance:
315.         print(f"Converged in {iteration + 1} iterations.")
316.         break
317. else:
318.         print(f"Reached maximum iterations ({max_iterations}) without full convergence.")
319. 
320. return coefficients
```

```
def task_5_fit_model(features, targets, degree):
    # Convert features to numpy array
    features_array = features.to_numpy()

    # Fit the polynomial regression model
    coefficients = fit_polynomial_model(features_array, targets["Heating load"], degree)
    return coefficients
```

```
# Task 5 example with degree 2 polynomial
degree = 2
best_coefficients = task_5_fit_model(features, targets, degree)
print("Best Fitting Coefficients (Task 5):", best_coefficients)
```

·Task 6: Model selection

Setup two cross-validation procedures, one for the heat loads and one for cooling loads. Calculate the difference between the predicted target and the actual target for the test set in each cross-validation fold and output the mean of absolute differences across all folds for both the heating load estimation as well as the cooling load estimation. Using this as a quality metric, evaluate polynomial degrees ranging between 0 and 2 to determine the optimal degree for the model function for both the heating as well as the cooling loads.

Task 6 focuses on evaluating and selecting the optimal polynomial degree for predicting both heating and cooling loads using cross-validation. The primary goal is to balance model complexity and prediction accuracy by systematically assessing polynomial degrees ranging from 0 to 2. The process involves splitting the dataset into training and testing sets multiple times to ensure the evaluation is robust and captures the model's ability to generalize to unseen data.

For each fold of the cross-validation, the dataset is divided into a training set used to fit the model and a testing set used to evaluate it. The model is trained using the iterative procedure defined in Task 5, starting with zero-initialized coefficients and refining them until convergence or until the maximum number of iterations is reached. Once the coefficients are fitted, predictions are made on the test set using the design matrix corresponding to the specified polynomial degree. The mean absolute error (MAE) between the predicted and actual target values is calculated for each fold, providing a measure of the model's performance for that specific split.

This procedure is repeated for all folds, and the average MAE across folds is computed as the final evaluation metric for the given polynomial degree. By repeating this process for polynomial degrees 0, 1, and 2, the task identifies how the model's complexity affects its accuracy. Simpler models, such as those with degree 0, may struggle to capture meaningful patterns, leading to higher MAEs. In contrast, degree 2 models can account for quadratic relationships, potentially achieving lower errors. However, overly complex models could risk overfitting, which would be reflected in inconsistencies between training and testing performance.

The results of the cross-validation are aggregated for both heating and cooling loads. The degree with the lowest MAE is selected as the optimal degree for each target variable. This ensures the model is neither too simplistic nor unnecessarily complex, striking the right balance to maximize predictive performance.

The output of this task includes the cross-validation results for each degree and the optimal polynomial degree for heating and cooling loads, along with their corresponding MAEs. By systematically evaluating the polynomial degrees, Task 6 ensures that the model selection process is data-driven and rigorous, setting the stage for final model evaluation and visualization in Task 7.

```
def cross_validate_model(features, targets, degree, n_splits=5, regularization_lambda=1e-5):
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
    mean_absolute_errors = []

    for train_index, test_index in kf.split(features):
        # Split the data into training and testing sets for the current fold
        X_train, X_test = features[train_index], features[test_index]
        y_train, y_test = targets[train_index], targets[test_index]

        # Fit the model on the training set
                coefficients = fit_polynomial_model(X_train, y_train, degree, regularization_lambda=regularization_lambda)

        # Predict on the test set
        design_matrix_test = create_design_matrix(X_test, degree)
        y_pred = np.dot(design_matrix_test, coefficients)

        # Calculate the mean absolute error for this fold
        mae = np.mean(np.abs(y_test - y_pred))
        mean_absolute_errors.append(mae)

    # Return the mean of the mean absolute errors across all folds
    return np.mean(mean_absolute_errors)
```

```
def create_design_matrix(features, degree):
    n_samples, n_features = features.shape
    design_matrix = np.ones((n_samples, 1))  # Start with bias term (intercept)

    for d in range(1, degree + 1):
        for combination in np.ndindex(*(d + 1,) * n_features):
            if sum(combination) == d:
                    term = np.prod(features ** np.array(combination), axis=1, keepdims=True)
                design_matrix = np.hstack((design_matrix, term))
    return design_matrix
```

```
def task_6_cross_validation(features, targets):
    features_array = features.to_numpy()
    heating_targets = targets["Heating load"].to_numpy()
    cooling_targets = targets["Cooling load"].to_numpy()

    degrees = range(0, 3)  # Polynomial degrees to evaluate: 0, 1, 2
    results = {"Heating load": {}, "Cooling load": {}}
```

```
    # Evaluate each degree for both heating and cooling loads
    for degree in degrees:
            heating_mae = cross_validate_model(features_array, heating_targets, degree)
            cooling_mae = cross_validate_model(features_array, cooling_targets, degree)
        results["Heating load"][degree] = heating_mae
        results["Cooling load"][degree] = cooling_mae

    # Find the optimal degree for each load
     optimal_heating_degree = min(results["Heating load"], key=results["Heating load"].get)
     optimal_cooling_degree = min(results["Cooling load"], key=results["Cooling load"].get)

    print("Cross-Validation Results:")
    print(results)
    print(f"Optimal Degree for Heating load: {optimal_heating_degree}")
    print(f"Optimal Degree for Cooling load: {optimal_cooling_degree}")

                    return      optimal_heating_degree,      results["Heating load"][optimal_heating_degree], \
        optimal_cooling_degree, results["Cooling load"][optimal_cooling_degree]
```

```
# Task 6 cross-validation
optimal_degrees = task_6_cross_validation(features, targets)
print("Optimal Degrees and MAEs (Task 6):", optimal_degrees)
```

·Task 7: Evaluation and visualisation of results

Now using the full dataset, estimate the model parameters for both the heating loads as well as the cooling loads using the selected optimal model function as determined in task 6. Calculate the predicted heating and cooling loads using the estimated model parameters for the entire dataset. Plot the estimated loads against the true loads for both the heating and the cooling case. Calculate and output the mean absolute difference between estimated heating/cooling loads and actual heating/cooling loads.

Task 7 involves leveraging the entire dataset to estimate the model parameters for both heating and cooling loads using the optimal polynomial degree identified in Task 6. The process begins with determining the model parameters that best fit the data. For this, we use the selected polynomial degree (degree 2 for both heating and cooling loads) to construct the basis functions and estimate the coefficients. These coefficients are calculated using the iterative linearization and parameter update procedure developed in previous tasks, ensuring convergence to the best-fitting values. With the model parameters in hand, we calculate the predicted heating and cooling loads for the entire dataset (lines 452 - 479).

To evaluate the model's performance, the predicted loads are compared to the true loads from the dataset. The mean absolute difference (MAD) between the predicted and true loads is computed for both heating and cooling cases. This metric provides a quantitative measure of the model's accuracy, reflecting the average error magnitude between the predictions and the actual values (lines 481 - 483).

Visual evaluation of the model is done by plotting the predicted heating loads against the true heating loads, as well as the predicted cooling loads against the true cooling loads. Each plot includes a red dashed line representing the ideal fit, where predicted values exactly equal the true values. This visual representation allows for a qualitative assessment of the model's performance. A strong alignment of points along the ideal fit line indicates a well-fitting model (lines 485 - 511).

From the generated output, the MAD values for heating and cooling loads were found to be approximately 0.5807 and 1.1407, respectively, which are both small values. This suggests that the model is accurate in predicting the heating and cooling loads. Additionally, the plots confirm this accuracy, as the majority of the data points closely follow the ideal fit line, with only minor deviations. The convergence behavior during parameter estimation shows that the iterative optimization process was effective for most cases, although some iterations reached the maximum limit without full convergence. However, these instances did not significantly affect the overall model accuracy.

In summary, Task 7 effectively applies the optimal polynomial models to predict heating and cooling loads for the entire dataset, evaluates the predictions both quantitatively (using MAD) and visually (through scatter plots), and demonstrates strong performance with minimal prediction errors.

```
def task_7_final_model_evaluation(features, targets, optimal_heating_degree, optimal_cooling_degree):
    # Convert features and targets to numpy arrays
    features_array = features.to_numpy()
    heating_targets = targets["Heating load"].to_numpy()
    cooling_targets = targets["Cooling load"].to_numpy()

    # Estimate coefficients for heating load
    heating_coefficients = fit_polynomia212109
    211_model(features_array, heating_targets, optimal_heating_degree)
            heating_design_matrix = create_design_matrix(features_array, optimal_heating_degree)
    heating_predictions = np.dot(heating_design_matrix, heating_coefficients)

    # Estimate coefficients for cooling load
     cooling_coefficients = fit_polynomial_model(features_array, cooling_targets, optimal_cooling_degree)
            cooling_design_matrix = create_design_matrix(features_array, optimal_cooling_degree)
    cooling_predictions = np.dot(cooling_design_matrix, cooling_coefficients)

    # Calculate mean absolute differences
    heating_mae = np.mean(np.abs(heating_targets - heating_predictions))
    cooling_mae = np.mean(np.abs(cooling_targets - cooling_predictions))

    # Plot results
    plt.figure(figsize=(14, 6))

    # Heating Load
    plt.subplot(1, 2, 1)
            plt.scatter(heating_targets, heating_predictions, alpha=0.7, label="Predicted vs. True")
    plt.plot([heating_targets.min(), heating_targets.max()],
            [heating_targets.min(), heating_targets.max()],
            color='red', linestyle='--', label="Ideal Fit")
    plt.title("Heating load: Predicted vs. True")
    plt.xlabel("True Heating load")
    plt.ylabel("Predicted Heating load")
    plt.legend()

    # Cooling Load
    plt.subplot(1, 2, 2)
            plt.scatter(cooling_targets, cooling_predictions, alpha=0.7, label="Predicted vs. True")
    plt.plot([cooling_targets.min(), cooling_targets.max()],
            [cooling_targets.min(), cooling_targets.max()],
            color='red', linestyle='--', label="Ideal Fit")
    plt.title("Cooling load: Predicted vs. True")
    plt.xlabel("True Cooling load")
    plt.ylabel("Predicted Cooling load")
    plt.legend()

    plt.tight_layout()
    plt.show()
```

```

    # Output results
    print(f"Mean Absolute Difference (Heating load): {heating_mae}")
    print(f"Mean Absolute Difference (Cooling load): {cooling_mae}")

    return heating_mae, cooling_mae
```

```
# Using optimal degrees from Task 6
optimal_heating_degree = 2
optimal_cooling_degree = 2

# Task 7 evaluation
heating_mae, cooling_mae = task_7_final_model_evaluation(
    features, targets, optimal_heating_degree, optimal_cooling_degree
)
```

·Task 8: Optional comparison

Compare your results to the results reported by [Tsanas and Xifara, 2012].

Task 8 requires comparing the results obtained in our analysis to the results reported by Tsanas and Xifara (2012). Their study emphasizes the use of advanced methodologies, including Random Forests (RF) and Iteratively Reweighted Least Squares (IRLS), to predict heating load (HL) and cooling load (CL) using eight building characteristics as inputs. Here is a structured analysis of the comparison:

-Key Metrics Comparison

Tsanas and Xifara provided error metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), and Mean Relative Error (MRE) to evaluate prediction accuracy for HL and CL. These results were obtained through 10-fold cross-validation repeated 100 times, ensuring robustness.

Our Model's Results

- Mean Absolute Difference (MAD) (computed as part of Task 7):
    - Heating Load: 0.5807
    - Cooling Load: 1.1407

Tsanas and Xifara's Results

- Using IRLS:
    - HL MAE: 2.14 ± 0.24
    - CL MAE: 2.21 ± 0.28
- Using RF:
    - HL MAE: 0.51 ± 0.11
    - CL MAE: 1.42 ± 0.25

Comparison:

- Our model's MAD for HL (0.5807) is close to the RF MAE of 0.51 ± 0.11, indicating comparable performance for heating load prediction.
- For CL, our model's MAD (1.1407) outperforms RF's MAE (1.42 ± 0.25) and is significantly better than IRLS.

-Model Complexity

- **Tsanas and Xifara** utilized two learners:
    - **IRLS**: A linear regression approach.
    - **RF**: A complex non-linear model capable of capturing intricate relationships and interactions among variables. It outperformed IRLS, particularly for non-linear relationships like those between input variables and CL/HL.

- **Our Model**:
    - Based on polynomial regression with a degree of 2 for both HL and CL.
    - While polynomial regression is less sophisticated than RF, it demonstrates strong predictive performance. This supports the argument that second-degree polynomial models can effectively approximate the relationships in this dataset.

-Variable Importance

- Tsanas and Xifara found that glazing area (X7) was the most important predictor for both HL and CL, as identified by RF. This aligns with engineering intuition since glazing significantly impacts heat absorption and leakage.
- In our polynomial model:
    - Variable importance was not explicitly computed, but the quadratic terms in the model inherently account for potential non-linear relationships.

-Interpretation of Errors

- Tsanas and Xifara noted that HL was consistently predicted more accurately than CL, regardless of the learner used. They attributed this to interactions among variables that were more effective for estimating HL.
- Similarly, in our results, the MAD for HL (0.5807) is smaller than that for CL (1.1407), aligning with their observation.

-Strengths of Our Model vs. Tsanas and Xifara

- **Our Model**:
    - Achieves comparable accuracy to RF while using a simpler polynomial regression framework. This highlights the potential of polynomial models for datasets where relationships are not overly complex.
    - Demonstrates strong results even without advanced ensemble techniques like RF.
- **Tsanas and Xifara**:
    - Showcased the robustness of RF for complex data by incorporating input variable importance and non-linear interactions.
    - Their RF model produced consistently lower MAE and MSE values compared to IRLS, especially for CL.

-Discussion

Our model performs remarkably well compared to RF, particularly for CL prediction, where it achieves lower MAD. However, RF remains superior for datasets with significant complexity and variable interactions due to its non-parametric nature. Tsanas and Xifara's emphasis on using RF also highlights its resilience to multicollinearity, a challenge in traditional regression models.

Given our simpler approach, the comparable accuracy suggests that polynomial regression, especially with careful degree selection, is a viable alternative for similar prediction tasks. However, extending this framework with RF or another advanced learner could further improve performance, particularly in datasets with more complex relationships or larger feature sets.

-Conclusion

In summary, our results align closely with those of Tsanas and Xifara, achieving comparable prediction accuracy for HL and slightly better for CL. While their study demonstrates the power of RF for building energy prediction, our polynomial regression approach proves effective with lower computational complexity. Both methods underline the importance of understanding input-output relationships and selecting appropriate learners for accurate predictions.