Quentin JONNEAUX
Student Number : R00274704
COMP 8043 - MACHINE LEARNING

ASSIGNMENT 3 - REGRESSION & OPTIMISATION
PREDICTING EXPECTED THERMAL LOADS OF A BUILDING

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Table of content

Summary

The goal is to train and evaluate a regression function that can be used to predict the expected thermal loads of a building based on a selection of input variables.

The full dataset contains 768 instances of example buildings that have been generated by the authors of the above paper using a building energy performance simulation tool.

The input variables for the regression function are the following basic building characteristics:
- Relative compactness
- Surface area
- Wall area
- Roof area
- Overall height
- Orientation
- Glazing area
- Glazing area distribution

The goal is to predict how much energy is required to either heat or cool the building as indicated by the two output variables:
- Heating load
- Cooling load

We created the process of implementing two polynomial regression functions that take the basic eight building parameters as input and calculate either the expected heating load or the expected cooling load of the building as output.

From the targets data frame, we can derive the minimum and maximum of heating and cooling loads with the following results:
- Minimum Heating Load: **6.01**
- Maximum Heating Load: **43.1**
- Minimum Cooling Load: **10.9**
- Maximum Cooling Load: **48.03**

Our study returned predictions also with low mean absolute error deviation for both Heating Loads (0.78) and Cooling Loads (1.45) and strong positive linear patterns for both targets with correlation coefficients of 0.99 and 0.98 for heating and cooling loads respectively, reflecting the accuracy of the regression.

Tsanas and Xifara (2012) can predict Heating Loads and Cooling loads with low mean absolute error deviations from the ground truth which is established using Ecotect (0.51 and 1.42, respectively), according to their study on quantitative estimation of energy performance of residential buildings using statistical machine learning tools.

Per the study Abstract, Tsanas and Xifara result are deviating less from ground truth, this may be explained by the choice of their machine learning algorithm (Random Forest) and the use of statistical testing to identify principal components.

Required libraries and functions

```
# Importing necessary libraries
import pandas as pd # Pandas for data reading and manipulation
import numpy as np # Numpy for mathematical calculation and manipulating arrays

import matplotlib.pyplot as plt # Matplotlib pyplot to provide visuals
import seaborn as sns # Seaborn to provide advanced plots

from sklearn import model_selection # Scikit Learn model_selection to apply K-Fold cross validation
```

Several libraries are needed to deliver this project.

We are using pandas for reading and manipulating the dataset but also, as additional step, create other data frames summarizing each differences between predictions and actual loads to compute the metric (mean absolute difference).

Numpy is used for calculation and computing arrays to train the classifier and evaluate it.

Also matplotlib is used mainly to create lineplots to visualise the relationships between predictions and actual values. Seaborn will be used to support the relationship of such data by computing a linear trendline.

Finally, we import the model_selection module of Scikit Learn to create a K-Fold cross validation procedure to perform hyperparameter tuning to determine the best polynomial degree of the for the regression.

Task 1 - Input data

```
################################################################################
# Task 1 - Input data
# Create a function that import and preprocess data
def data_input():
    # Store file path of Dataset
    path = '/Users/Quentin/Desktop/Hdip Data Science and Analytics/Year 1/Semester 3/COMP8043 - Machine Learning/Assignment 3/energy_performance.csv'

    # Store the data in a data frame
    data = pd.read_csv(path)

    # Split features and targets
    features = data.drop(['Heating load', 'Cooling load'], axis=1)
    targets = data[['Heating load', 'Cooling load']]

    # Get and Output the minimum and maximum heating loads
    min_heating_load = min(targets['Heating load'])
    print('Minimum Heating Load: ',min_heating_load)
    max_heating_load = max(targets['Heating load'])
    print('Maximum Heating Load: ',max_heating_load)

    # Get and Output the minimum and maximum cooling loads
    min_cooling_load = min(targets['Cooling load'])
    print('Minimum Cooling Load: ',min_cooling_load)
    max_cooling_load = max(targets['Cooling load'])
    print('Maximum Cooling Load: ',max_cooling_load)

    # Convert features and each target into numpy arrays for algorithm training
    features=np.array(features)
    heating_targets=np.array(targets['Heating load'])
    cooling_targets=np.array(targets['Cooling load'])

    # Return arrays for training
    return features,heating_targets,cooling_targets

################################################################################
```

We are creating the data_input function to read the dataset and preprocess the data for training the algorithm.

Since the targets are heating and cooling loads, we are dropping the 2 last columns to get the features dataset. The targets would be those columns.

From the targets data frame, we can derive the minimum and maximum of heating and cooling loads with the following results:

- Minimum Heating Load: **6.01**
- Maximum Heating Load: **43.1**
- Minimum Cooling Load: **10.9**
- Maximum Cooling Load: **48.03**

Task 2 - Model Function and parameter size

```
# Task 2 - model function
# Create a function that compute a polynomial
def calculate_model_function(degree, data, p0):
    # We initialize the result vector of zeros with a lengths of the data points
    result = np.zeros(data.shape[0])
    # We initialize the index at 0 for the coefficient access for computation
    a = 0
    # We iterate through 8 loops for 8 variables to calculate the function result
    for i in range(degree+1): # For each loop, we iterate with the degree number
        for j in range(degree+1):
            for k in range(degree+1):
                for l in range(degree+1):
                    for m in range(degree+1):
                        for n in range(degree+1):
                            for o in range(degree+1):
                                for p in range(degree+1):
                                    if i+j+k+l+m+n+o+p<=degree:
                                        # We compute result as long as we haven't reach the degree
                                        result += p0[a] * (data[:, 0] ** i) * (data[:, 1] ** j) * (data[:, 2] ** k) * (data[:, 3] ** l) * (data[:, 4] ** m) * (data[:, 5] ** n) * (data[:, 6] ** o) * (data[:, 7] ** p)
                                        a += 1
    # We return the result of the model function
    return result
```

Firstly, we are creating the calculate_model function to compute 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 multi-variate polynomial of the specified degree

We first initialize the result vector with zeros and shape it with the number of rows of the data array to allow matrix calculations. Since our features contains 8 input variables, we create 8 loop nested in each other and iterate the number of times of the polynomial degree and compute the result based on the coefficient vector.

```
# Create a function that compute the vector size from the degree
def num_parameters(degree):
    # initialize size at zero
    t = 0
    # We iterate through 8 loops for 8 variables to calculate the vector size
    for i in range(degree+1):
        for j in range(degree+1):
            for k in range(degree+1):
                for l in range(degree+1):
                    for m in range(degree+1):
                        for n in range(degree+1):
                            for o in range(degree+1):
                                for p in range(degree+1):
                                    if i+j+k+l+m+n+o+p<=degree:
                                        # We add 1 to size as long as we haven't reach degree
                                        t = t + 1
    # We return the size
    return t
```

Secondly, we are num_parameters creating function that determines the correct size for the parameter vector from the degree of the multi-variate polynomial. Similarly, we increment the size by one within the 8 nested loops until the sum of coefficient equals the degree parameters.

Those functions will allow us to linearize as we need to make calculations using coefficients a lot and a perturbation to calculate the derivatives and return the Jacobians. Also calculating the vector size for a degree will facilitate the parameter update calculation to estimate the best coefficients.

Task 3 - Linearization

```python
##############################################################################
# Task 3 - linearization
# Create a function that compute the target vector and the Jacobian at the linearization point
def linearize(deg, data, p0):
    # Calculating the model function at p0 (to then calculate the Jacobian)
    f0 = calculate_model(deg, data, p0)
    # Initialize the Jacobian vector with the model function vector length)
    J = np.zeros((len(f0), len(p0)))
    # Defining small incrementation value (to calculate the partial derivative)
    epsilon = 1e-6
    # Iterate for each value in the p0 vector
    for i in range(len(p0)):
        # Add the perturbation value to the coefficient
        p0[i] += epsilon
        # Calculate the model function at new point
        fi = calculate_model(deg, data, p0)
        # Revert perturbation
        p0[i] -= epsilon
        # Calculate the partial derivative (difference divided by perturbation)
        di = (fi - f0) / epsilon
        # Store the partial derivative in the Jacobian matrix
        J[:, i] = di
    # We return the target vector and the Jacobian matrix
    return f0, J
##############################################################################
```

We are creating the linearize function to compute the estimated target vector and the Jacobian matrix at the linearization point. The idea of the algorithm is:

- choose an initial parameter vector
- iterate until convergence, for each training sample:
  - Calculate the model function with parameters
  - Calculate the model function Jacobian with parameters
  - Calculate the parameter update
  - Update the parameter vector

Using the polynomial degree, the features vector and the coefficient vector, we first calculate the polynomial function and initialize the Jacobian matrix at 0s with the shape of the function and the coefficient.

We define a very small perturbation to use and iterate through each coefficient. At each iteration, we add the small perturbation the coefficient and calculate the model function for it (and revert the perturbation to not affect the calculation at the next coefficient) and we calculate the partial derivative by dividing the model functions difference by the perturbation.

By repeating this step at each coefficient we are able to provide the estimated target vector and the Jacobian Matrix.

Task 4 - Parameter update

```python
################################################################################
# Task 4 - parameter update
# Create a function that calculates the optimal parameter update
def calculate_update(y, f0, J):
    # We define a step
    l = 1e-2
    # We compute the normal equation matrix, augmented with regularization term by adding a diagonal matrix of size of the Jacobian
    N = J.T @ J + l * np.eye(J.shape[1])
    # We calculate the residual by substracting the estimation to the actual value at data point
    r = y - f0
    # We create the right hand-side of the normal equation system
    n = J.T @ r
    # We solve the parameter update
    dp = np.linalg.solve(N, n)
    # We return the update
    return dp
################################################################################
```

We are creating the calculate_update function to compute the optimal parameter update from the training target vector extracted in task 1 and the estimated target vector and Jacobian calculated in task 3.

We start with calculating the normal equation matrix and add a regularisation term (here: 1e-2) to prevent the normal equation system from being singular. We then calculate the residual by substracting the estimation to the actual value at data point and built the normal equation system. Finally, we solve the normal equation system to obtain the optimal parameter update.

Task 5 - Regression

```
################################################################################
# Task 5 - regression
# Create a function that calculates the coefficient vector that best fits the training data
def regression(degree,features,targets):
    # Setting up the maximum number of iteration to prevent converging endlessly
    max_iter = 100
    # initialise the parameter vector of coefficients with zeros (adjusting the size to degrees)
    p0 = np.zeros(num_parameters(degree))
    # Setting up an iterative procedure
    for i in range(max_iter):
        # Linearize
        f0, J = linearize(degree, features, p0)
        # Update parameter
        dp = calculate_update(targets, f0, J)
        # Update paramater vector by adding the update to each coefficient
        p0 += dp
        print(i)
        print(dp)
    return p0
################################################################################
```

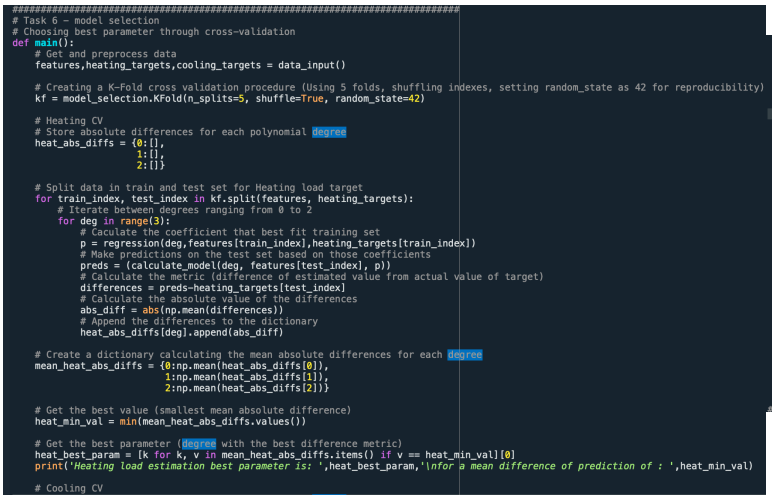
```
97
[-1.62545570e-01 -4.42260085e-02 -5.04777305e-06 -1.04323193e-01
  1.24386700e-04 -1.73576227e-03 -3.92847269e-02  5.02093431e-06
  3.01635863e-04  9.75983051e-06  4.80307884e-01  5.34805431e-04
  5.64467741e-03  5.74592166e-04 -4.93483957e-02  1.18169789e+00
 -1.91795374e-01  9.63919192e-02  6.54070892e-01  2.20256364e-01
  3.56327750e-02  6.00650906e-01 -9.59069600e-02  4.80944844e-02
  3.27025513e-01  1.10259574e-01  6.92818697e-01  3.37500665e-01
 -5.97669142e-01  9.59363897e-02 -4.80617094e-02 -3.27000169e-01
 -1.10164300e-01  2.45464581e-02 -3.16327744e-01 -2.11762480e-02
  1.08720728e-03  2.40181266e-02  2.10756791e-02  2.03889611e-02
  3.11482527e-02  2.47945010e-03 -5.26152181e-03  1.78531886e-03
 -2.88101965e-01]
98
[-1.50933103e-01 -9.05822710e-02  6.98541099e-05 -5.46358568e-02
  5.07990669e-04 -2.95809439e-02 -6.57256890e-02  1.01452059e-05
 -2.05113063e-04  1.73365427e-05 -1.29414737e+00  1.19252376e-03
  2.18430690e-04  8.47459866e-04  1.69573291e-01 -3.42268528e-01
 -5.35095657e-01 -4.06130480e-01  1.67238354e-01  5.77130365e-02
  6.62577189e-01 -1.89052046e-01 -2.67568595e-01 -2.03064766e-01
  8.36041515e-02  3.03054662e-02  6.23997264e-01  1.46354753e-01
  1.85664674e-01  2.67627637e-01  2.03118983e-01 -8.35611512e-02
 -3.02002456e-02 -4.93363069e-02 -5.37243099e-03 -1.40979693e-01
  2.33591100e-02  4.80387158e-02  4.03774908e-02  3.52968076e-02
  1.08031039e-01 -2.02742277e-03  3.85786522e-03 -1.64880325e-03
 -2.55645565e-01]
99
[ 7.61271710e-02  7.79520879e-02 -1.18657224e-04  1.62968132e-01
 -9.17446355e-04  5.58554858e-02  7.91197393e-02 -3.56824706e-06
 -6.25914530e-05 -1.32204692e-05  4.68603070e-01 -9.78010357e-04
 -4.38207887e-03 -1.06871607e-03 -5.76004620e-02  8.97824151e-02
  5.82044409e-01  2.04345430e-01  3.63643940e-01  2.13677216e-02
  1.57598157e-01  4.92009860e-02  2.91039238e-01  1.02246298e-01
  1.81840906e-01  1.03951978e-02  1.65439588e-01  4.33229616e-02
 -5.07956528e-02 -2.91089985e-01 -1.02357765e-01 -1.81892343e-01
 -1.03685521e-02  4.93280455e-01  2.42713239e-01 -2.86035423e-01
  6.96290722e-03 -4.14643876e-02 -8.27382161e-02 -4.20458885e-02
  2.57435604e-02  7.04761112e-04 -1.33652512e-04  6.01795093e-04
 -2.72588940e-01]
```

We are creating the regression function that computes the coefficient vector that best fits the training data. We initialise the parameter vector of coefficients with zeros (with the appropriate shape based on polynomial degree). Then we set up an iterative procedure that alternates linearization and parameter update, updating the coefficients resulting from the parameter update calculation.

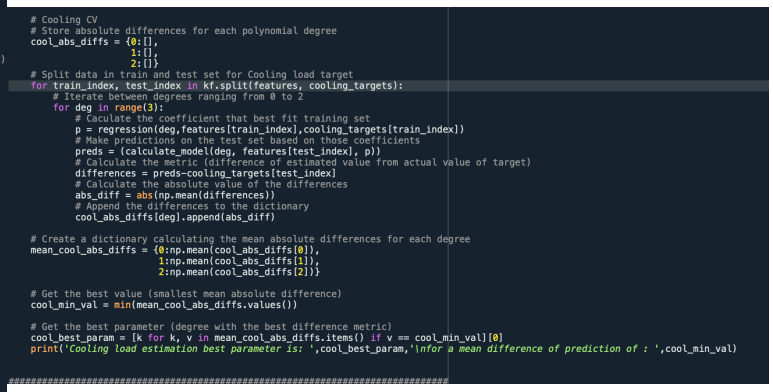
We expect the the parameter update and the residuals to be smaller and smaller as we go through iterations. The first updates are big as coefficients are starting at 0 but as coefficients are update closer to the ones that best fits, there is very little update and this update will be close to 0. Knowing this, we can set up a maximum iteration number (here 100) to be close to convergence.

Per sample output on the right, we can see that updates for a bi-variate polynomial are alternating the sign (positive and negative through iterations) and are on a scale e-01 to sometime e-06, meaning we are very close to 0.

Task 6 - Model Selection



```
######################################################################
# Task 6 - model selection
# Choosing best parameter through cross-validation
def main():
    # Get and preprocess data
    features,heating_targets,cooling_targets = data_input()

    # Creating a K-Fold cross validation procedure (Using 5 folds, shuffling indexes, setting random_state as 42 for reproducibility)
    kf = model_selection.KFold(n_splits=5, shuffle=True, random_state=42)

    # Heating CV
    # Store absolute differences for each polynomial degree
    heat_abs_diffs = {0:[],
                      1:[],
                      2:[]}

    # Split data in train and test set for Heating load target
    for train_index, test_index in kf.split(features, heating_targets):
        # Iterate between degrees ranging from 0 to 2
        for deg in range(3):
            # Caculate the coefficient that best fit training set
            p = regression(deg,features[train_index],heating_targets[train_index])
            # Make predictions on the test set based on those coefficients
            preds = [calculate_model(deg, features[test_index], p)]
            # Calculate the metric (difference of estimated value from actual value of target)
            differences = preds-heating_targets[test_index]
            # Calculate the absolute value of the differences
            abs_diff = abs(np.mean(differences))
            # Append the differences to the dictionary
            heat_abs_diffs[deg].append(abs_diff)

    # Create a dictionary calculating the mean absolute differences for each degree
    mean_heat_abs_diffs = {0:np.mean(heat_abs_diffs[0]),
                           1:np.mean(heat_abs_diffs[1]),
                           2:np.mean(heat_abs_diffs[2])}

    # Get the best value (smallest mean absolute difference)
    heat_min_val = min(mean_heat_abs_diffs.values())

    # Get the best parameter (degree with the best difference metric)
    heat_best_param = [k for k, v in mean_heat_abs_diffs.items() if v == heat_min_val][0]
    print('Heating load estimation best parameter is: ',heat_best_param,'\nfor a mean difference of prediction of : ',heat_min_val)

    # Cooling CV
```



```
    # Cooling CV
    # Store absolute differences for each polynomial degree
    cool_abs_diffs = {0:[],
                      1:[],
                      2:[]}
    # Split data in train and test set for Cooling load target
    for train_index, test_index in kf.split(features, cooling_targets):
        # Iterate between degrees ranging from 0 to 2
        for deg in range(3):
            # Caculate the coefficient that best fit training set
            p = regression(deg,features[train_index],cooling_targets[train_index])
            # Make predictions on the test set based on those coefficients
            preds = [calculate_model(deg, features[test_index], p)]
            # Calculate the metric (difference of estimated value from actual value of target)
            differences = preds-cooling_targets[test_index]
            # Calculate the absolute value of the differences
            abs_diff = abs(np.mean(differences))
            # Append the differences to the dictionary
            cool_abs_diffs[deg].append(abs_diff)

    # Create a dictionary calculating the mean absolute differences for each degree
    mean_cool_abs_diffs = {0:np.mean(cool_abs_diffs[0]),
                           1:np.mean(cool_abs_diffs[1]),
                           2:np.mean(cool_abs_diffs[2])}

    # Get the best value (smallest mean absolute difference)
    cool_min_val = min(mean_cool_abs_diffs.values())

    # Get the best parameter (degree with the best difference metric)
    cool_best_param = [k for k, v in mean_cool_abs_diffs.items() if v == cool_min_val][0]
    print('Cooling load estimation best parameter is: ',cool_best_param,'\nfor a mean difference of prediction of : ',cool_min_val)

######################################################################
```

We then setup two cross-validation procedures, one for the heat loads and one for cooling loads (with 5 folds). 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, we 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.

When outputting the best parameters, we obtain the following results:

- Heating load estimation best parameter is: **2** (for a mean difference of prediction of : **0.08619903003348724**)
- Cooling load estimation best parameter is: **2** (for a mean difference of prediction of : **0.12968929948354954**)

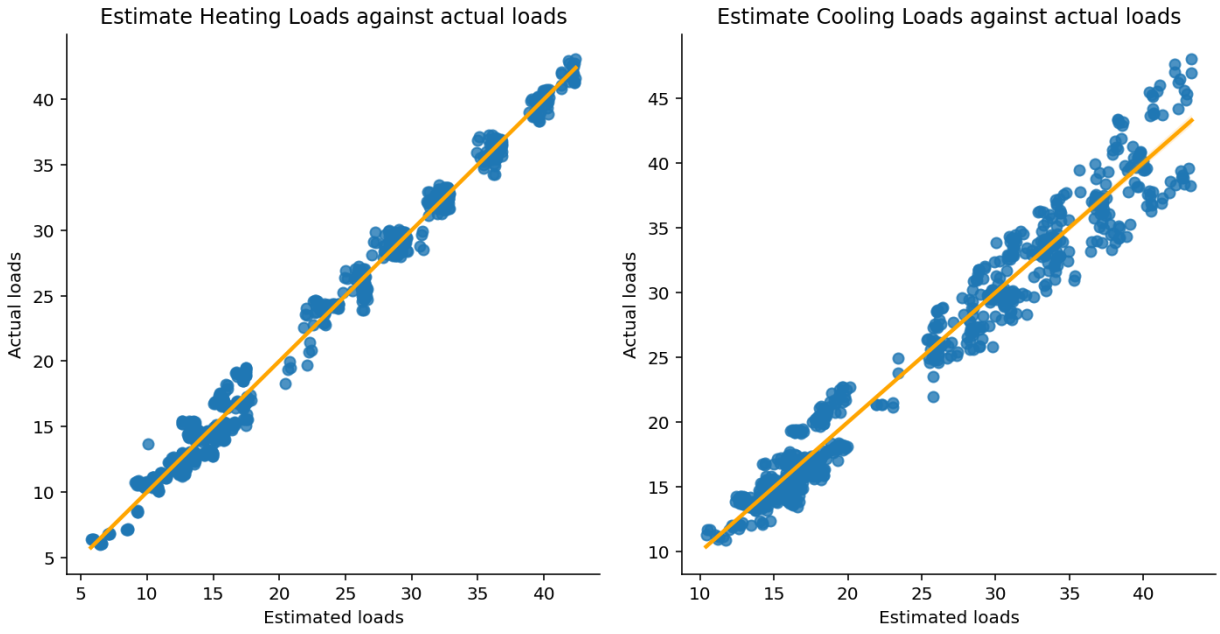
```
In [1]: %runfile '/Users/Quentin/Desktop/Hdip Data Science and A
R00274704_Quentin.Jonneaux_Assignement.3.py' --wdir
Minimum Heating Load:  6.01
Maximum Heating Load:  43.1
Minimum Cooling Load:  10.9
Maximum Cooling Load:  48.03
Heating load estimation best parameter is:  2
for a mean difference of prediction of :  0.08619903003348724
Cooling load estimation best parameter is:  2
for a mean difference of prediction of :  0.12968929948354954
```

We would there for set the degree hyperparameter as 2 for estimating both targets

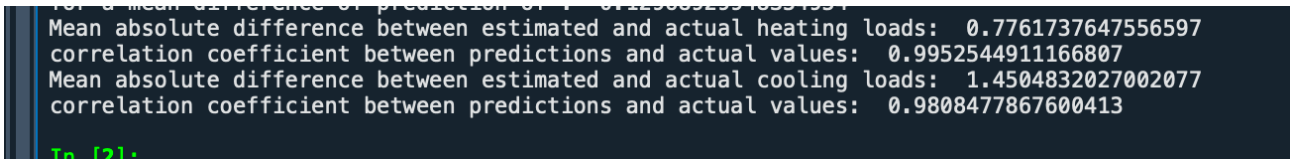
Task 7 - evaluation and visualisation

```
# Task 7 - evaluation and visualisation of results

    # Final Evaluation of Heating Loads predictions
    # Train the model by calculating the best parameter vectors using the best degree parameter from CV procedure
    heat_p = regression(heat_best_param,features,heating_targets)
    # Make predictions for all features in the dataset
    heat_preds = (calculate_model(heat_best_param, features, heat_p))
    # Make a dataframe storing predictions, actual values and absolute differences for each observations
    heat_data = pd.DataFrame({'Predictions':heat_preds,'Actual':heating_targets,'Absolute Differences':abs(heat_preds-heating_targets)})

    # Clear the plots
    plt.close("all")
    # Create a linear model plot to illustrate relationship between predictions and actual values
    sns.lmplot(x='Predictions',y='Actual',data = heat_data,line_kws={'color': 'orange'})
    # Name visual and axis
    plt.title('Estimate Heating Loads against actual loads')
    plt.xlabel('Estimated loads')
    plt.ylabel('Actual loads')
    # Display plot
    plt.show()

    # Compute final mean absolute difference for the target
    final_mean_heat_abs_diff = np.mean(heat_data['Absolute Differences'])
    print('Mean absolute difference between estimated and actual heating loads: ',final_mean_heat_abs_diff)

    # Final Evaluation of Cooling Loads predictions

    # Train the model by calculating the best parameter vectors using the best degree parameter from CV procedure
    cool_p = regression(cool_best_param,features,cooling_targets)
    # Make predictions for all features in the dataset
    cool_preds = (calculate_model(cool_best_param, features, cool_p))

    # Make a dataframe storing predictions, actual values and absolute differences for each observations
    cool_data = pd.DataFrame({'Predictions':cool_preds,'Actual':cooling_targets,'Absolute Differences':abs(cool_preds-cooling_targets)})

    # Clear the plots
    plt.close("all")
    # Create a linear model plot to illustrate relationship between predictions and actual values
    sns.lmplot(x='Predictions',y='Actual',data = cool_data,line_kws={'color': 'orange'})
    # Name visual and axis
    plt.title('Estimate Cooling Loads against actual loads')
    plt.xlabel('Estimated loads')
    plt.ylabel('Actual loads')
    # Display plot
    plt.show()

    # Compute final mean absolute difference for the target
    final_mean_cool_abs_diff = np.mean(cool_data['Absolute Differences'])
    print('Mean absolute difference between estimated and actual cooling loads: ',final_mean_cool_abs_diff)


main()
```

We now use the full dataset, estimate the model parameters for both the heating loads as well as the cooling loads using the selected optimal model function and predict heating and cooling loads using the estimated model parameters for the entire dataset (we are using degree 2 per previous task).



We then plot the estimated loads against the true loads for both the heating and the cooling case. We can see both observe a linear positive relationship, which is expected as we are trying to make predictions close the actual values and mean absolute errors resulting from cross validation procedure are close to 0.1. However, we observe difference variance pattern as the the heating variance seem to have an elliptical pattern while the cooling variance seem to increase with load.



Finally, Calculate and output the mean absolute difference between estimated heating/cooling loads and actual heating/cooling loads. We also calculate the correlation coefficient to quantify their relationships. For the final evaluation, we have the following output:

- Mean absolute difference between estimated and actual heating loads**: 0.7761737647556597**
- Correlation coefficient between predictions and actual heating values: **0.9952544911166807**
- Mean absolute difference between estimated and actual cooling loads: **1.4504832027002077**
- Correlation coefficient between predictions and actual cooling values: **0.9808477867600413**

We can see that the mean absolute differences metrics differ significantly from the one outputted as the cross validation stage (close to 0.1).

Task 8 - Conclusion

Tsanas and Xifara (2012) can predict Heating Loads and Cooling loads with low mean absolute error deviations from the ground truth which is established using Ecotect (0.51 and 1.42, respectively), according to their study on quantitative estimation of energy performance of residential buildings using statistical machine learning tools.

Our study returned predictions also with low mean absolute error deviation for both Heating Loads (0.78) and Cooling Loads (1.45).

Per the study Abstract, Tsanas and Xifara result are deviating less from ground truth, this may be explained by the choice of their machine learning algorithm (Random Forest) and the use of statistical testing to identify principal components.

References

Athanasios Tsanas, Angeliki Xifara. Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools. Energy and Buildings, Volume 49, 2012, Pages 560-567, ISSN 0378-7788, https://doi.org/10.1016/j.enbuild.2012.03.003