2024

Machine Learning Assignment 3

RESULTS AND REPORT

EMMETT FITZHARRIS R00222357

Task 1

In Task 1 I begin by separating the labels and targets into separate data frames. Our targets, “Heating load” and “Cooling Load” are the last two columns in the data, so I use a slice to select the appropriate information.

```
def separate_labels_and_targets(df):  2 usages  Emmett
    # Separate the labels and targets
    labels = df.iloc[:, 0:8]
    targets = df.iloc[:, 8:10]
```

To verify my data is correct, I print a readable list of Labels and Targets to console.

```
################################################################################
                              Task 1: Input Data
################################################################################

Labels:
  - Relative compactness
  - Surface area
  - Wall area
  - Roof area
  - Overall height
  - Orientation
  - Glazing area
  - Glazing area distribution

Targets:
  - Heating load
  - Cooling load
```

The maximum and minimum of each target are calculated simply using the max() and min() methods.

```
# determine max and min values for each target
max_target = targets.max()
min_target = targets.min()

# create a DataFrame for max and min values
max_min_df = pd.DataFrame({'Max': max_target, 'Min': min_target})
```

One again I print the result, this time creating a merged data frame to make print formatting of the table simple.

```
Max and Min values for each target:

                Max    Min
Heating load  43.10   6.01
Cooling load  48.03  10.90
```

Task 2

Task 2 does not have immediate output. The result will be used in later tasks.

First, I create a function which calculates the value of a polynomial model. This accepts three parameters:

- "degree"
    - Complexity of the model.
- "feature_vectors"
    - Input data
- "coefficients"
    - The weights assigned to each term.

```
def polynomial_model(degree, feature_vectors, coefficients):  5 usages  new *
    result = np.zeros(feature_vectors.shape[0])
    k = 0
    for i in range(degree + 1):
        for j in range(degree + 1):
            if i + j <= degree:
                result += coefficients[k] * (feature_vectors[:, 0] ** i) * (feature_vectors[:, 1] ** j)
                k += 1
    return result
```

I then create a function which calculates the number of parameters which are required for a given degree. The degree decides the size of the coefficient vector, and therefore the complexity of the model.

I use a nested loop to iterate through all possible combinations of exponents for the polynomial terms, to find cases where the sum is less than the degree. The number of valid results is the number of parameters we should use.

```
def num_parameters(degree):  1 usage  new *
    t = 0
    for i in range(degree + 1):
        for j in range(degree + 1):
            if i + j <= degree:
                t += 1
    return t
```

Task 3

In task three I create a function which calculates the value of the model function and it's Jacobian at a given linearization point.

The "polynomial_model()" function from task 2 is called on line 88 to calculates the estimated target vector with initial coefficients, then again in the loop on line 103 with slightly modified coefficients to compute the partial derivatives for the Jacobian.

```
80  ########################################################################################################################
81  # Task 3: Linearization
82  ########################################################################################################################
83
84  def calculate_model_and_jacobian(degree, feature_vectors, coefficients):  1 usage  Emmett *
85
86      # Using the function from task 2, calculates the estimated target vector f0 using the initial coefficients at the
87      # linearization point.
88      f0 = polynomial_model(degree, feature_vectors, coefficients)
89
90      num_samples = feature_vectors.shape[0]
91      num_params = len(coefficients)
92
93      Jacobian = np.zeros((num_samples, num_params))
94      epsilon = 1e-6
95
96
97      for i in range(num_params):
98          coefficients_i = coefficients.copy()
99          coefficients_i[i] += epsilon
100         # Using the function from task 2, calculates the model function value fi with slightly modified coefficients to
101         # compute the partial derivatives for the Jacobian. The partial derivative is determined to be the difference
102         # between the model function value fi and the model function value f0 divided by epsilon.
103         fi = polynomial_model(degree, feature_vectors, coefficients_i)
104         Jacobian[:, i] = (fi - f0) / epsilon
105
106     return f0, Jacobian
```

Task 4

In task 4 I create the function “calculate_optimal_parameter_update()”.

This function creates a normal matrix, by performing matrix multiplication of the transposed Jacobian and a regular matrix.

Residuals are then calculated as the difference between training and estimated target values, and are used to find the product of the transpose of the Jacobian and the residuals, as the rhs (“Right hand side”).

```
########################################################################################################################
# Task 4: Parameter Update
########################################################################################################################

def calculate_optimal_parameter_update(training_target, estimated_target, jacobian):
    regularization_lambda = 1e-6

    # Normal matrix is calculated as the product of the Jacobian transposed and the Jacobian, plus a regularization term.
    # The "@" operator is used to perform matrix multiplication.
    # np.eye creates an identity matrix with the same number of columns as the Jacobian matrix.
    normal_matrix = jacobian.T @ jacobian + regularization_lambda * np.eye(jacobian.shape[1])

    # The residual is the difference between the training_target vector and the estimated_target vector.
    # This represents the error between the actual target values and the values predicted by the model.
    residual = training_target - estimated_target

    rhs = jacobian.T @ residual

    optimal_update = np.linalg.solve(normal_matrix, rhs)

    return optimal_update
```

The rhs is used to solve the linear matrix equation, which provides the optimal parameter update.

Task 5

In task 5 I create the function “fit_polynomial_model()”.

In this function I start by Initializing an array of zeros matching the number of parameters.

I then create a loop, which repeatedly calculates coefficient parameter updates using the function defined in task 5, to fit the model to the data.

When the change is small enough that it is not significant, I break out of the loop.

```python
########################################################################################################################
# Task 5: Regression
########################################################################################################################

def fit_polynomial_model(degree, training_features, training_targets, max_iterations=100, tolerance=1e-6):
    num_params = num_parameters(degree)
    coefficients = np.zeros(num_params) # parameter vector

    for iteration in range(max_iterations):
        estimated_target, jacobian = calculate_model_and_jacobian(degree, training_features, coefficients)

        optimal_update = calculate_optimal_parameter_update(training_targets, estimated_target, jacobian)

        coefficients += optimal_update # update the parameter vector

        # Initially these parameter updates are likely to be large, as the model starts from all zeros which are
        # unlikely to be the optimal values. As the model converges, the parameter updates will become smaller. The
        # residuals will also decrease as the model converges and the model becomes better fit to the data.
        #
        # As such we can use the norm of the optimal update vector as a measure of convergence and determine the optimal
        # number of iterations. Convergence is determined by the norm of the optimal update vector being less than the
        # tolerance value. For the tolerance value a small value of 1e-6 is used. This is arbitrary and can be adjusted
        # as needed. It could be tested as a parameter itself to determine the optimal value.
        if np.linalg.norm(optimal_update) < tolerance:
            # # Uncomment to print the number of iterations to convergence
            # print(f'Converged in {iteration + 1} iterations.')
            iterations_to_convergence = iteration + 1
            break

    return coefficients
```

Task 6

In task 6 I create 2 primary functions, "cross_validate_polynomial_model()" and "evaluate_polynomial_degrees()", then call them from the "task6()" function which itself is called in main().

```python
##########################################################################################################################
# Task 6: Model Selection
##########################################################################################################################

def task6(labels, targets):  1 usage  Emmett *
    OutputFormat.print_header(section: 'h1', text: 'Task 6: Model Selection')
    features = labels.values
    heating_targets = targets.iloc[:, 0].values
    cooling_targets = targets.iloc[:, 1].values

    optimal_heating_degree, optimal_cooling_degree = evaluate_polynomial_degrees(
        features, heating_targets, cooling_targets
    )
    return optimal_heating_degree, optimal_cooling_degree
```

cross_validate_polynomial_model() accepts degree, features and targets, and performs a cross fold validation. I first split the data into 5 folds, and arbitrary number which can be controlled by the parameter for further tuning.

Each fold is split into test and training data, then the model is fit to the training data and evaluated based on the test data. They then return the mean differences.

```python
def cross_validate_polynomial_model(degree, features, targets, n_splits=5):  2 usages  Emmett
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
    absolute_differences = []

    for train_index, test_index in kf.split(features):
        # Split the data into training and testing sets
        train_features, test_features = features[train_index], features[test_index]
        train_targets, test_targets = targets[train_index], targets[test_index]

        # Fit the polynomial model to the training data
        coefficients = fit_polynomial_model(degree, train_features, train_targets)

        # Predict the targets for the test data
        predicted_targets = polynomial_model(degree, test_features, coefficients)

        # Calculate the mean absolute difference between the predicted and actual targets
        absolute_differences.append(mean_absolute_error(test_targets, predicted_targets))

    # Return the mean of the mean absolute differences
    return np.mean(absolute_differences)
```

evaluate_polynomial_degrees() calls the above for each target set. Then outputs the mean difference between expected values for each target (Heating and Cooling).

```python
def evaluate_polynomial_degrees(features, heating_targets, cooling_targets, max_degree=2):  1 usage  Emmett
    heating_errors = []
    cooling_errors = []

    for degree in range(max_degree + 1):
        heating_error = cross_validate_polynomial_model(degree, features, heating_targets)
        cooling_error = cross_validate_polynomial_model(degree, features, cooling_targets)

        heating_errors.append(heating_error)
        cooling_errors.append(cooling_error)

        # Output the mean absolute difference
        print(f'Degree {degree}: Heating Load Error = {heating_error}, Cooling Load Error = {cooling_error}')

    OutputFormat.print_divider('h2')

    optimal_heating_degree = np.argmin(heating_errors)
    optimal_cooling_degree = np.argmin(cooling_errors)

    print(f'Optimal Degree for Heating Load: {optimal_heating_degree}')
    print(f'Optimal Degree for Cooling Load: {optimal_cooling_degree}')

    return optimal_heating_degree, optimal_cooling_degree
```

Finally using the numpy argmin() function, we find the index of the minimum value of the error arrays. This corresponds to the degree of the polynomial which showed the lowest error.

This value is output for each target.

```
################################################################################
                            Task 6: Model Selection
################################################################################

Degree 0: Heating Load Error = 9.158466128809945, Cooling Load Error = 8.594410479985314
Degree 1: Heating Load Error = 6.041895407143999, Cooling Load Error = 5.566484812314997
Degree 2: Heating Load Error = 4.7778833688459015, Cooling Load Error = 4.444413616781516

********************************************************************************

Optimal Degree for Heating Load: 2
Optimal Degree for Cooling Load: 2
```

The optimal value for degree in both cases is 2, as the error is at its lowest.

Task 7

In task 7 I create the functions “estimate_and_plot()” and “plot_estimated_vs_true_loads()”/

estimate_and_plot() starts by separating the heating targets from the cooling targets, using all data. A model is fit for each target using the previously calculated optimal degrees, and then tested against predictions.

From here it calls plot_estimated_vs_true_loads to plot the data.

Finally I calculate and output the mean absolute error using the method from _regression.

```python
def estimate_and_plot(labels, targets, optimal_heating_degree, optimal_cooling_degree):
    features = labels.values
    heating_targets = targets.iloc[:, 0].values
    cooling_targets = targets.iloc[:, 1].values

    # Estimate model parameters for heating loads
    heating_coefficients = fit_polynomial_model(optimal_heating_degree, features, heating_targets)
    predicted_heating_loads = polynomial_model(optimal_heating_degree, features, heating_coefficients)

    # Estimate model parameters for cooling loads
    cooling_coefficients = fit_polynomial_model(optimal_cooling_degree, features, cooling_targets)
    predicted_cooling_loads = polynomial_model(optimal_cooling_degree, features, cooling_coefficients)

    plot_estimated_vs_true_loads(heating_targets, predicted_heating_loads, cooling_targets, predicted_cooling_loads)

    # Calculate and output the mean absolute difference
    heating_mad = mean_absolute_error(heating_targets, predicted_heating_loads)
    cooling_mad = mean_absolute_error(cooling_targets, predicted_cooling_loads)

    print(f'Mean Absolute Difference for Heating Loads: {heating_mad}')
    print(f'Mean Absolute Difference for Cooling Loads: {cooling_mad}')
```

As we can see, cooling predictions are more accurate than heating, with a lower absolute difference.

```
###############################################################################
                 Task 7:  Evaluation and Visualisation of Results
###############################################################################

Mean Absolute Difference for Heating Loads: 4.742537496129844
Mean Absolute Difference for Cooling Loads: 4.406287602138309
```

To plot the data, I scatter Heating data in Orange, and Cooling data in Blue. I then plot a reference line, from the min to max of each target.

```python
def plot_estimated_vs_true_loads(heating_targets, predicted_heating_loads, cooling_targets, predicted_cooling_loads):
    plt.figure()

    # Plot estimated vs true loads for heating
    plt.scatter(heating_targets, predicted_heating_loads, color='orange', label='Heating Loads', alpha = 0.5)

    # Plot estimated vs true loads for cooling
    plt.scatter(cooling_targets, predicted_cooling_loads, color='blue', label='Cooling Loads', alpha = 0.5)

    # Plot the reference lines
    min_heating = heating_targets.min()
    max_heating = heating_targets.max()
    plt.plot([min_heating, max_heating], [min_heating, max_heating], 'orange', linestyle='dotted')

    min_cooling = cooling_targets.min()
    max_cooling = cooling_targets.max()
    plt.plot([min_cooling, max_cooling], [min_cooling, max_cooling], 'blue', linestyle='dotted')

    plt.xlabel('True Loads')
    plt.ylabel('Estimated Loads')
    plt.title('True vs Estimated Loads')
    plt.legend()
    plt.tight_layout()
    plt.show()
```

We can our observations are consistent with the visualization. In most cases cooling trends closer to the reference line than Heating.