```
# partial derivative of cost function with respect to jth m

def derivative_cost_wrt_mj(xtrain,ytrain,j,arrM,b):

    ans=0

    k=len(ytrain)

    n=len(xtrain[0])

    for i in range(k):

        temp=0

        for p in range(n):

            temp+=(arrM[p]*xtrain[i][p])

        temp+=b

        ans+=2*(ytrain[i]-temp)*(-xtrain[i][j])

    return ans/k
```

The code appears to be a Python function that calculates the partial derivative of the cost function with respect to a single weight coefficient (denoted as mj) in a linear regression model. Here is a step-by-step explanation of the code:

1. The function is defined with five input arguments: `xtrain`, `ytrain`, `j`, `arrM`, and `b`.
    - `xtrain` is a matrix of size (k x n) containing the training data features, where k is the number of training examples and n is the number of features.
    - `ytrain` is a vector of size (k,) containing the target values for each training example.

- `j` is an integer representing the index of the weight coefficient for which the derivative is calculated.
- `arrM` is a vector of size (n,) containing the current weight coefficients for the linear regression model.
- `b` is a scalar representing the bias term of the linear regression model.

2. The function initializes a variable `ans` to 0.
3. The function loops over each training example `i` in the range `0` to `k-1`.
    - For each training example, the function initializes a variable `temp` to 0.
    - The function loops over each feature `p` in the range `0` to `n-1`, and adds the product of the `p`-th weight coefficient and the `i`-th training example's `p`-th feature to `temp`.
    - The function adds the bias term `b` to `temp`.
    - The function computes the value of the inner sum in the derivative formula, which is `(yi - temp) * (-xtrain[i][j])`, and adds this value to `ans`.
4. After the loop over training examples, the function divides `ans` by `k` and returns the result as the output of the function.

The formula for the partial derivative of the cost function with respect to the `j`-th weight coefficient is `(2/k) * Σ[i=1,k] (-xij) * (yi - Σ[p=1,n] (xip * mp + b))`. The function implements this formula using a nested loop to compute the sum over training examples and features.

```
def predict(x,arrM,b):
    y=[]
    k=len(x)
    n=len(x[0])
    for i in range(k):
        ans=0
        for j in range(n):
            ans+=arrM[j]*x[i][j]
```

```
            ans+=b
            y.append(ans)
    return y
```

Python function that makes predictions using a linear regression model. Here is a step-by-step explanation of the code:

1. The function is defined with three input arguments: `x`, `arrM`, and `b`.
    - `x` is a matrix of size (k x n) containing the input data features, where k is the number of data points and n is the number of features.
    - `arrM` is a vector of size (n,) containing the weight coefficients for the linear regression model.
    - `b` is a scalar representing the bias term of the linear regression model.
2. The function initializes an empty list `y`.
3. The function loops over each data point `i` in the range `0` to `k-1`.
    - For each data point, the function initializes a variable `ans` to 0.
    - The function loops over each feature `j` in the range `0` to `n-1`, and adds the product of the `j`-th weight coefficient and the `i`-th data point's `j`-th feature to `ans`.
    - The function adds the bias term `b` to `ans`.
    - The function appends `ans` to the list `y`.
4. After the loop over data points, the function returns the list `y` as the output of the function.

The function computes the predicted target values using the formula `yi = Σ[j=1,n] (xij * mj) + b` for each data point `i`. This formula is implemented using a nested loop over data points and features to compute the sum in the formula for each data point. The predicted target values are stored in a list `y` and returned as the output of the function.

Code is a Python function that performs the gradient descent algorithm to optimize the weight coefficients of a linear regression model. Here is a step-by-step explanation of the code:

1. The function is defined with six input arguments: `xtrain`, `ytrain`, `learning_rate`, `num_iterations`, `print_cost`, and `initial_values`.

- `xtrain` is a matrix of size (k x n) containing the training data features, where k is the number of training examples and n is the number of features.
- `ytrain` is a vector of size (k,) containing the target values for each training example.
- `learning_rate` is a scalar representing the step size for gradient descent.
- `num_iterations` is an integer representing the number of iterations to run gradient descent.
- `print_cost` is a Boolean value indicating whether to print the cost function during optimization.
- `initial_values` is a vector of size (n+1,) containing the initial values for the weight coefficients and the bias term of the linear regression model.

2. The function initializes a variable `costs` to an empty list.
3. The function initializes the weight coefficients `arrM` and the bias term `b` to the values in `initial_values`.
4. The function loops over each iteration `i` in the range `0` to `num_iterations-1`.
    - The function computes the gradient of the cost function with respect to each weight coefficient using the `derivative_cost_wrt_mj()` function defined elsewhere.
    - The function updates the weight coefficients and the bias term using the gradient and the learning rate, according to the formula `mi_new = mi_old - learning_rate * dJ/dmi` for each weight coefficient `mi` and the bias term `b`.
    - If `print_cost` is `True`, the function computes the cost function using the `compute_cost()` function defined elsewhere and appends the cost to the `costs` list.
5. After the loop over iterations, the function returns the weight coefficients and the bias term as the output of the function, along with the `costs` list if `print_cost` is `True`.

The function implements the gradient descent algorithm to minimize the cost function for a linear regression model. The gradient descent algorithm is performed over a specified number of iterations, and the weight coefficients and the bias term are updated according to the gradient of the cost function and the learning rate. The cost function is computed at each iteration if `print_cost` is `True`, and the costs are stored in a list `costs`.

```
import pandas as pd

import numpy as np


# loading the training data through numpy


data=np.loadtxt("combined_cycle_powerPlant_train.csv",delimiter=",")

k=len(data)           # no. of rows

n=len(data[0])        # no. of columns

xtrain=data[:,0:n-1]

ytrain=data[:,n-1]

n=n-1
```

his code appears to be a Python script that loads a training dataset from a CSV file into a NumPy array and then separates the input features and target values into separate variables. Here is a step-by-step explanation of the code:

1. The code imports the `pandas` and `numpy` libraries.

2. The code loads the training data from a CSV file `"combined_cycle_powerPlant_train.csv"` using the `np.loadtxt()` function from NumPy. The data is stored in a NumPy array `data`.
3. The code initializes a variable `k` to the number of rows in the `data` array.
4. The code initializes a variable `n` to the number of columns in the `data` array.
5. The code extracts the input features into a NumPy array `xtrain` using NumPy's array slicing syntax `data[:,0:n-1]`. This creates a new array that includes all rows and all columns except the last one.
6. The code extracts the target values into a NumPy array `ytrain` using NumPy's array slicing syntax `data[:,n-1]`. This creates a new array that includes all rows and only the last column.
7. The code updates the variable `n` to be the number of input features in `xtrain` by subtracting 1 from the previous value of `n`.

The script loads the training data from a CSV file into a NumPy array, and then separates the input features and target values into separate variables. The input features are stored in a NumPy array `xtrain` and the target values are stored in a NumPy array `ytrain`. The variable `k` is set to the number of rows in the `data` array, and the variable `n` is set to the number of columns in the `data` array (which is updated later to be the number of input features).

```
from sklearn import preprocessing

scaler=preprocessing.StandardScaler()

scaler.fit(xtrain)

xtrain=scaler.transform(xtrain)
```

This code uses the `preprocessing` module from the `scikit-learn` library to standardize the input features in `xtrain` using the `StandardScaler` class. Here is a step-by-step explanation of the code:

1. The code imports the `preprocessing` module from the `scikit-learn` library.
2. The code creates an instance of the `StandardScaler` class called `scaler`.
3. The `fit()` method of the `scaler` object is called with the `xtrain` array as its argument. This method calculates the mean and standard deviation of each feature in `xtrain`.
4. The `transform()` method of the `scaler` object is called with the `xtrain` array as its argument. This method applies the standardization formula to each feature in `xtrain`, which transforms each feature to have a mean of 0 and a standard deviation of 1.
5. The standardized input features are then stored back in the `xtrain` variable.

By standardizing the input features, the code ensures that each feature is on the same scale, which can help improve the performance of some machine learning algorithms, such as linear regression. The `StandardScaler` class from the `scikit-learn` library standardizes the input features by subtracting the mean and dividing by the standard deviation of each feature.

```
alpha=float(input("Enter Learning Rate\n")) # 0.01

iterations=int(input("Enter Number of
iterations\n")) # 2000
```

```
arrM=[2 for i in range(n)]
b=0
for t in range(iterations):
    for i in range(n):

arrM[i]=arrM[i]-(alpha*derivative_cost_wrt_mj(xtrai
n,ytrain,i,arrM,b))

b=b-(alpha*derivative_cost_wrt_b(xtrain,ytrain,arrM
,b))
```

This code initializes the learning rate and number of iterations from user input, sets the initial values of the model parameters `arrM` and `b`, and then performs gradient descent to update the model parameters for the specified number of iterations. Here is a step-by-step explanation of the code:

1. The code prompts the user to enter a value for the learning rate `alpha` and reads in the input using the `input()` function. The value is converted to a float using the `float()` function.
2. The code prompts the user to enter a value for the number of iterations `iterations` and reads in the input using the `input()` function. The value is converted to an integer using the `int()` function.
3. The code initializes a list `arrM` of length `n` with initial values of 2. This represents the initial guesses for the slope coefficients of the linear regression model.
4. The code initializes the intercept `b` to 0. This represents the initial guess for the intercept of the linear regression model.
5. The code enters a loop that performs gradient descent for the specified number of iterations. The outer loop runs `iterations` times.
6. The inner loop updates each slope coefficient in `arrM` using gradient descent. The loop runs `n` times, once for each slope coefficient.
7. Within the inner loop, the code calls the `derivative_cost_wrt_mj()` function to calculate the gradient of the cost function with respect to the jth slope coefficient in `arrM`. The code updates the jth slope coefficient by subtracting `alpha` times the gradient from the current value.

8. After the inner loop has updated all the slope coefficients in `arrM`, the code updates the intercept `b` using gradient descent. The code calls the `derivative_cost_wrt_b()` function to calculate the gradient of the cost function with respect to the intercept `b`. The code updates `b` by subtracting `alpha` times the gradient from the current value.

The code performs gradient descent to optimize the linear regression model parameters `arrM` and `b` for the specified number of iterations using the learning rate `alpha`. The goal is to minimize the cost function, which measures the difference between the predicted values of the model and the actual values of the target variable `ytrain`.

```
xtest=np.loadtxt("combined_cycle_powerPlant_test.csv",delimiter=",")
# loading the testing data through numpy
xtest=scaler.transform(xtest)

# Prediction Part
y_pred1=predict(xtest,arrM,b)
y_pred2=predict(xtrain,arrM,b)
```

This code loads the testing data from a CSV file, scales the data using the same scaler object used to transform the training data, and then makes predictions using the trained linear regression model. Here is a step-by-step explanation of the code:

1. The code loads the testing data from a CSV file using the `np.loadtxt()` function. The function reads in the CSV file and returns a numpy array.
2. The code stores the features of the testing data in the `xtest` variable. This is done by selecting all rows and all but the last column of the `data` array.
3. The code applies the same scaler transformation that was applied to the training data to the testing data. This is done using the `scaler.transform()` method.
4. The code makes predictions for the testing data using the `predict()` function. The function takes in the scaled testing data `xtest`, the model parameters `arrM` and `b`, and returns a list of predicted target variable values `y_pred1`.

5. The code also makes predictions for the training data using the same `predict()` function. The function takes in the scaled training data `xtrain`, the model parameters `arrM` and `b`, and returns a list of predicted target variable values `y_pred2`.

Overall, this code allows us to evaluate the performance of the trained linear regression model on new, unseen data. The scaled testing data is used to make predictions, and the performance of the model is evaluated based on how well these predictions match the true target variable values. The code also generates predictions for the training data to compare how well the model performs on data that it was trained on versus new data that it has not seen before.

```
y_pred1=np.array(y_pred1)
y_pred2=np.array(y_pred2)

y_pred1=np.round(y_pred1,2)
y_pred2=np.round(y_pred2,2)
```

This code converts the predicted target variable values `y_pred1` and `y_pred2` to numpy arrays using the `np.array()` function. Then, it rounds the values to two decimal places using the `np.round()` function. Here is a step-by-step explanation of the code:

1. The code converts the `y_pred1` list of predicted target variable values for the testing data to a numpy array using the `np.array()` function.
2. The code converts the `y_pred2` list of predicted target variable values for the training data to a numpy array using the `np.array()` function.
3. The code rounds the values in `y_pred1` to two decimal places using the `np.round()` function.
4. The code rounds the values in `y_pred2` to two decimal places using the `np.round()` function.

Overall, this code is useful for formatting the predicted target variable values into a numpy array and rounding the values to two decimal places. This can be helpful for comparing the predicted values to the true target variable values and evaluating the performance of the model.

```
np.savetxt("ccpp_result.csv",y_pred1,delimiter=",")
```

This code saves the predicted target variable values for the testing data to a CSV file named "ccpp_result.csv" using the `np.savetxt()` function. Here is a step-by-step explanation of the code:

1. The code specifies the name of the CSV file to be saved as "ccpp_result.csv".
2. The code passes the `y_pred1` numpy array of predicted target variable values for the testing data as the second argument to the `np.savetxt()` function.
3. The code sets the `delimiter` parameter to "," to specify that the values in the CSV file should be separated by commas.

Overall, this code is useful for saving the predicted target variable values to a CSV file so that they can be easily loaded into other programs or used for further analysis.

`y_pred2`

`'y_pred2'` is a numpy array containing the predicted target variable values for the training data. The predicted values are generated by the `predict()` function using the trained coefficients (`arrM` and `b`) and the scaled training input data (`xtrain`).

The predicted values in `y_pred2` can be used to evaluate the performance of the model on the training data. One way to evaluate the performance is to calculate the mean squared error between the

predicted values and the true target variable values (`ytrain`). A lower mean squared error indicates better performance.

Note that the model should not be evaluated based solely on its performance on the training data. It is also important to evaluate the performance on unseen testing data to ensure that the model is able to generalize well to new data.

```
ytr=np.array(ytrain)
```

This code converts the `ytrain` list of true target variable values for the training data to a numpy array using the `np.array()` function and assigns it to the variable `ytr`. Here is a step-by-step explanation of the code:

1. The code passes the `ytrain` list of true target variable values for the training data as the argument to the `np.array()` function.
2. The `np.array()` function converts the `ytrain` list to a numpy array.
3. The numpy array is assigned to the variable `ytr`.

Overall, this code is useful for converting the true target variable values for the training data to a numpy array, which can be useful for various types of data analysis and manipulation.

```
score=1-((np.sum((ytr-y_pred2)**2))/(np.sum((ytr-np.mean(ytr))**2)))
```

This code calculates the coefficient of determination (R-squared) of the linear regression model using the predicted target variable values for the training data (`y_pred2`) and the true target variable values for the training data (`ytr`). The R-squared value is a statistical measure that represents the proportion of variance in the target variable that is explained by the independent variables in the model. Here is a step-by-step explanation of the code:

1. The code calculates the sum of squares of residuals (SSres) by subtracting each predicted value in `y_pred2` from its corresponding true value in `ytr`, squaring the difference, and then summing the squares.
2. The code calculates the total sum of squares (SStot) by subtracting each true value in `ytr` from the mean of `ytr`, squaring the difference, and then summing the squares.
3. The code calculates the R-squared value by subtracting the ratio of SSres to SStot from 1.
4. The R-squared value is assigned to the variable `score`.

Overall, this code is useful for evaluating the performance of the linear regression model on the training data using the R-squared value. A higher R-squared value indicates that the model is able to explain more of the variance in the target variable using the independent variables.