Object Oriented Programming - Assignment 1

Nynke Terpstra (s4574532) & Sara Huysmans (s4666089)
Group 76

6 december 2023

Introduction

The purpose of this report is to detail the design choices that we made while implementing the task in our code as per the specifications outlined in the assignment. The implementation adheres to object-oriented programming (OOP) principles.

1 multiple_linear_regression.py

This file defines a Python class *MultipleLinearRegression* that implements multiple linear regression. The class includes methods for training the model using the normal equation, making predictions, and retrieving a dataset (defaulting to the Wine dataset). The normal equation is used to calculate the coefficients for the linear regression model.

Class Initialization

```
\begin{verbatim}
class MultipleLinearRegression:
    def __init__(self, dataset_name:str='breast_cancer_data'):
        self._intercept = None
        self._dataset_name = dataset_name
```

Motivation

- **Dataset Name Parameter:** Introduced a `dataset_name` parameter to allow flexibility in choosing the dataset. The default value is set to 'breast_cancer_data'.
- **Intercept Attribute:** Made the `_intercept` attribute private to encapsulate the internal state of the class.

Train Method

```
def train(self, X:np.ndarray, y:np.ndarray) -> None:
    # ... (existing code)
```

Motivation

- **Input Parameters:** The method takes input data `X` and output data `y`, both as NumPy arrays. This design choice allows users to provide datasets in a familiar tabular/matrix form.
- **Exception Handling:** Implemented a try-except block to catch `LinAlgError` when attempting to invert the matrix. This ensures robustness by handling cases where the matrix is not invertible.

```
def predict(self, X:np.ndarray) -> np.ndarray:
    # ... (existing code)
```

Motivation

- **Input Parameter:** Similar to the `train` method, the `predict` method takes input data `X` as a NumPy array.
- **Predictions Calculation:** Predictions are calculated using the learned coefficients and the input data. The method adds a column of ones for the intercept term before making predictions.

Get Data Method

```
ddef get_data(self) -> Tuple[np.ndarray, np.ndarray]:
    # ... (existing code)
```

Motivation

- **Dataset Retrieval:** Introduced a `get_data` method to obtain input data `X` and output data `y` using the specified dataset name. This allows users to easily retrieve the data for training and testing.

2 compare_with_sklearn.py

This file is designed for comparing the performance and behavior of a custom multiple linear regression model against scikit-learn's model, focusing on coefficients and prediction accuracy.

Class Initialization

The `ModelComparer` class is initialized with the following attributes:

```
\begin{verbatim}
class ModelComparer:
    def __init__(self, our_model:MultipleLinearRegression, sklearn_model:object):
        self._our_model = our_model
        self._sklearn_model = sklearn_model
```

Motivation

1. **Model Instances:**
    - **Design Choice:** Instances of our model (`our_model`) and the sklearn model (`sklearn_model`) are stored as attributes.
    - **Motivation:** Keeping references to both models facilitates easy comparison of coefficients and predictions. It allows for a clear and direct evaluation of the performance of our implementation against the well-established sklearn model.

Method: `compare_coefficients`

This method compares the coefficients of our model and the sklearn model:

```
\begin{verbatim}
def compare_coefficients(self) -> None:
    print("Our model coefficients:", (self.our_model._intercept))
    print("Sklearn model coefficients:", [self.sklearn_model.intercept_,
    *self.sklearn_model.coef_])
```

Motivation

1. **Coefficient Printing:**
    - **Design Choice:** Coefficients of both models are printed to the console.
    - **Motivation:** This design choice allows users to visually inspect and compare the coefficients. It provides transparency into how well our model aligns with the coefficients produced by the sklearn model.

Method: `compare_predictions`

This method compares the predictions of our model and the sklearn model:

```
\begin{verbatim}
def compare_predictions(self, X:np.ndarray, y:np.ndarray) -> None:
    '''
    X is a numpy array with a shape (n, p) containing the input data (n units, p features)
    y is a numpy array with a 1d shape containing the output data
    '''
    # Make and print predictions using the trained models
    our_predictions = self._our_model.predict(X)
    sklearn_predictions = self._sklearn_model.predict(X)
    print("MSE our model:",  mean_squared_error(y, our_predictions))
    print("MSE sklearn model:", mean_squared_error(y, sklearn_predictions))
```

Motivation

[label=0.]**Mean Squared Error (MSE) Calculation:**

1.
    - **Design Choice:** Mean Squared Error is calculated for both our model and the sklearn model.
    - **Motivation:** Using MSE as a metric provides a quantitative measure of prediction accuracy. The comparison helps in understanding how well our model performs in terms of prediction error compared to the sklearn model.

Method: `compare`

This method calls both `compare_coefficients` and `compare_predictions`:

```
\begin{verbatim}
def compare(self, X:np.ndarray, y:np.ndarray) -> None:
    '''
    X is a numpy array with a shape (n, p) containing the input data
    (n units, p features)
    y is a numpy array with a 1d shape containing the output data
    '''
    self.compare_coefficients()
    self.compare_predictions(X, y)
```

Motivation

1. **Comprehensive Comparison:**
    - **Design Choice:** Both coefficient comparison and prediction comparison are bundled in a single method.
    - **Motivation:** This design choice enhances code organization and provides a convenient way to perform a holistic comparison between our model and the sklearn model. Users can efficiently evaluate various aspects of model performance in one call.

3 regression_plotter.py

This files defines a class *RegressionPlotter* for visualizing the results of linear regression. The class provides methods to plot the linear regression line, plane, or multiple lines based on the number of features in the dataset.

Class Initialization

```
\begin{verbatim}
class RegressionPlotter:
    def __init__(self):
        self._data = None
```

Motivation

- **Data Attribute:** Introduced a `data` attribute to store the input data for plotting. This attribute is set using the `set_data` method before performing any plotting operations.

Set Data Method

```
def set_data(self, X:np.ndarray, y:np.ndarray) -> None:
    # ... (existing code)
```

Motivation

- **Data Setup:** The `set_data` method allows users to set the input data for plotting. It takes input data `X` and output data `y` and stores them as a NumPy multidimensional array in the `data` attribute.

Plot Linear Regression Line Method

```
def plot_linear_regression_line(self, model:MultipleLinearRegression, feature:int,
    target:int) -> None:
    # ... (existing code)
```

Motivation

- **Single Feature Plotting:** This method plots a linear regression line when asked to plot a model with only one feature. It visualizes the relationship between the selected feature and the target variable.

Plot Linear Regression Plane Method

```
def plot_linear_regression_plane(self, model:MultipleLinearRegression, features:List[int],
    target:int) -> None:
    # ... (existing code)
```

Motivation

- **Two Feature Plotting:** This method plots a linear regression plane when asked to plot a model with two features. It visualizes the relationship between the selected features and the target variable in three dimensions. The scattering of the features have been specifically set to the color blue to enhance visibility.

Plot Multiple Linear Regression Method

```
def plot_multiple_linear_regression(self, model:MultipleLinearRegression,
    features:List[int], target:int) -> None:
    # ... (existing code)
```

Motivation

- **Multiple Feature Plotting:** This method produces a sequence of 2D plots when asked to plot a generic MultipleLinearRegression model with any number of features. Each plot presents one feature against the target variable.

Plot Linear Regression Method

```
def plot_linear_regression(self, model:MultipleLinearRegression, features:List[int],
    target:int) -> None:
    # ... (existing code)
```

Motivation

- **Behavior Handling:** This method determines the appropriate behavior for plotting based on the number of features provided. It calls the specific plotting methods accordingly.

Exception Handling

```
if self.data is None:
    raise ValueError("No data provided for plotting. Set the 'data' attribute first.")
```

Motivation

- **Data Check:** Added a check to ensure that data is set before attempting to plot. Raises a `ValueError` if no data is provided, guiding users to set the 'data' attribute first.

4 model_saver.py

This file defines a class *ModelSaver* that allows saving and loading model parameters in either JSON or CSV format. It is particularly useful when you want to persist the state of a machine learning model, making it easier to resume training or deploy the model in different environments.

Class Initialization

```
\begin{verbatim}
class ModelSaver:
    def __init__(self, format_type:str):
        self._format_type = format_type
```

Motivation

- **Format Type Parameter:** Introduced a `format_type` parameter to specify the desired format for saving and loading. This parameter is set at initialization and can be modified by the user.

Save Model Parameters Method

```
def save_model_parameters(self, model:any, file_path:str) -> None:
    # ... (existing code)
```

Motivation

- **JSON and CSV Support:** Implemented support for saving model parameters in both JSON and CSV formats. The method checks the specified format type and serializes the model parameters accordingly.

Load Model Parameters Method

```
def load_model_parameters(self, model:any, file_path:str) -> None:
    # ... (existing code)
```

Motivation

- **JSON and CSV Support:** Implemented support for loading model parameters from both JSON and CSV formats. The method checks the specified format type and deserializes the model parameters accordingly.

Numpy Array Serialization

```
# Convert numpy arrays to lists to make them JSON serializable
for key, value in model_dict.items():
    if isinstance(value, np.ndarray):
        model_dict[key] = value.tolist()
```

Motivation

- **JSON Serialization of Numpy Arrays:** Added a conversion step for numpy arrays to lists to ensure they are JSON serializable. This is necessary when saving model parameters in JSON format.

CSV Reader for Loading

```
with open(file_path, 'r', newline='') as file:
    # Load the model parameters from a file
    reader = csv.reader(file)
    model_parameters = {rows[0]: float(rows[1]) for rows in reader}
    model.__dict__.update(model_parameters)
```

Motivation

- **CSV Loading with Reader:** Implemented CSV loading using the `csv.reader` to handle reading rows from the CSV file. The loaded parameters are then updated in the model's dictionary.

Exception Handling

```
else:
    raise ValueError("Unsupported format type. Choose 'json' or 'csv'.")
```

Motivation

- **Unsupported Format Handling:** Added a check to raise a `ValueError` if an unsupported format type is specified. This ensures that users choose between 'json' and 'csv' to maintain compatibility.

5 main.py

The *main.py* file is an implementation of Multiple Linear Regression using Object-Oriented Programming principles. It uses various functionalities that are implemented in the previous files, including model training, prediction, comparison with the sklearn implementation, plotting regression lines and planes, and saving/loading model parameters.

Importing Modules

```
from multiple_linear_regression import MultipleLinearRegression
from sklearn.linear_model import LinearRegression
from regression_plotter import RegressionPlotter
from compare_with_sklearn import ModelComparer
from model_saver import ModelSaver
```

Motivation

- **Modular Design:** Imported the necessary classes and modules to keep the code modular and organized. This allows for easy access to the functionalities of the linear regression model, plotter, model comparer, and model saver.

Instantiating Objects

```
model = MultipleLinearRegression(dataset_name='breast_cancer')
plotter = RegressionPlotter()
saver = ModelSaver(format_type='json')
```

Motivation

- **Object Instantiation:** Created instances of the `MultipleLinearRegression`, `RegressionPlotter`, and `ModelSaver` classes to use their functionalities. The `dataset_name` parameter is set for the multiple linear regression model, and the `format_type` parameter is set for the model saver.

Training the Model

```
X, y = model.get_data()
model.train(X, y)
```

Motivation

- **Model Training:** Used the `get_data` method to obtain input data and ground truth for training the linear regression model. The `train` method is then called to calculate the model parameters.

Comparing with sklearn

```
# compare our model with sklearn
sklearn_model = LinearRegression()
sklearn_model.fit(X, y)
compare_model = ModelComparer(model, sklearn_model)
compare_model.compare(X, y)
```

Motivation

- **Model Comparison:** Created an instance of `ModelComparer` to compare the custom linear regression model with the one from `sklearn`. The `compare` method is used to showcase the differences in coefficients and predictions.

Making Predictions and Printing Results

```
# Get the intercept
print(f"Intercept: {model._intercept}")

# Make predictions using the trained model
```

```
predictions = model.predict(X)

# Print the ground truth and predicted values
print("Ground truth:", y)
print("Predicted value:", predictions)
```

Motivation

- **Result Display:** Printed the intercept, ground truth, and predicted values to showcase the performance of the trained linear regression model.

Plotting Regression Lines and Planes

```
# Plot the regression line
plotter.set_data(X, y)
plotter.plot_linear_regression(model, [1], 1)

# Plot the regression plane
plotter.set_data(X, y)
plotter.plot_linear_regression_plane(model, [2,3], 1)

# Plot multiple regression lines
plotter.set_data(X, y)
plotter.plot_linear_regression(model, [3,4,5,6,7,8], 1)
```

Motivation

- **Visualization:** Utilized the `RegressionPlotter` to visualize the linear regression results. Different functionalities were demonstrated, including plotting a regression line, a regression plane, and multiple regression lines against the dataset.

Saving and Loading Model Parameters

```
# Save the model parameters
saver.save_model_parameters(model, 'model_parameters.json')

# Load the model parameters
saver.load_model_parameters(model, 'model_parameters.json')
```

Motivation

- **Model Persistence:** Utilized the `ModelSaver` to save the model parameters in JSON format. The saved parameters were then loaded back into the model to showcase the persistence of model state.