Object-Oriented Programming for AI A.Y. 2023-24

Assignment 1: Multiple Linear Regression using OOP

Deadline: December 6th, 2023

For this assignment, you are tasked with implementing a **Multiple Linear Regression** model in Python using OOP principles.

Introduction A multiple linear regression fits a hyperplane to a multi-dimensional dataset composed of features X and targets y. An example of application would be to calculate Simple linear regression, which we have seen during the examples in the previous lecture, relates y to a one-dimensional x; multiple linear regression extends it by allowing for multi-dimensional features: instead of observing one single variable x, we observe p variables instead, and we want to study how they relate to the targets y. The equation of the hyperplane is the following:

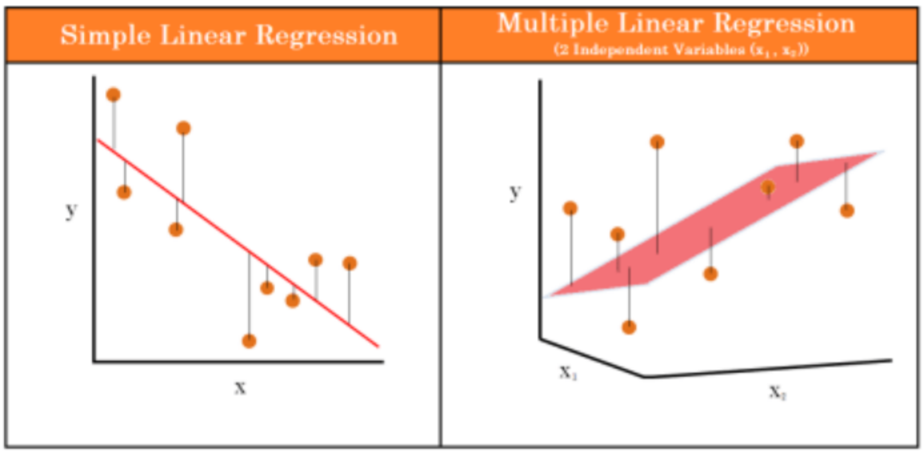
$$\hat{y} = b + w_1 x_1 + \cdots + w_p x_p \tag{1}$$

Notice that this is just a simple extension of the regression line from the simple linear regression case:

$$\hat{y} = b + wx \tag{2}$$

b identifies the *intercept* of the line and w is the *slope*. The hyperplane also has an intercept b, but adds individual slope coefficients for each of the variables $x_1, \dots, x_p$. Also, notice the difference in notation for the targets: y is the actual/**ground truth** target, while $\hat{y}$ is the prediction, which we want to be *as close as possible* to y.

We can visualize the extension from the simple linear regression to a multiple linear regression by plotting a regression line (as in Equation (2)) to a regression plane (as in Equation (1) with 2 input features):



In the simple linear regression case, we had a simple way for analytically calculate the best coefficients (according to the Mean Squared Error loss $(\hat{y}-y)^2$) given a set of observations on features x and targets y. Similarly, we can update the weights of the multiple linear regression by finding an optimal analytical solution. Instead of considering the line/hyperplane equation for a single data point as in Equation (1), we consider **multiple data points** $1, \ldots, n$ at the same time. The ground truth can be represented as such (notice the shift between y—a scalar—and $\mathbf{y}$, a vector):

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{3}$$

Similarly, the observations can be put into a $n \times p$ matrix $\mathbf{X}$:

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & \ldots & x_{1,p} \\ x_{2,1} & \ldots & x_{2,p} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \ldots & x_{n,p} \end{bmatrix} \tag{4}$$

Also, we can group the coefficients w in a vector $\mathbf{w}$:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_p \end{bmatrix} \tag{5}$$

Notice the difference in length of the vectors $\mathbf{w}$ and $\mathbf{y}$.

Now, we can formulate the hyperplane from Equation (1) with a notation that calculates simultaneously the predictions for the n datapoints of $\mathbf{X}$:

$$\hat{\mathbf{y}} = b + \mathbf{X}\mathbf{w} \tag{6}$$

Quick dimension check: $\mathbf{X}$ is $n \times p$, while $\mathbf{w}$ is $p \times 1$, so, their multiplication is $n \times 1$. b is a scalar (so we can add it to a vector without changing its dimension), thus $\hat{\mathbf{y}}$ is $n \times 1$, the same length as $\mathbf{y}$.

We can do more than that to render the notation more compact: we can put b within the coefficients vector $\mathbf{w}$:

$$\mathbf{w} = \begin{bmatrix} b \\ w_1 \\ \vdots \\ w_p \end{bmatrix} \tag{7}$$

and we can add a column of ones in the matrix $\mathbf{X}$ to account for the new formulation of $\mathbf{w}$:

$$\mathbf{X} = \begin{bmatrix} 1 & x_{1,1} & \dots & x_{1,p} \\ 1 & x_{2,1} & \dots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & \dots & x_{n,p} \end{bmatrix} \tag{8}$$

Now, the compact hyperplane equation Equation (6) becomes

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} \tag{9}$$

Using this notation, we can recover the optimal parameters configuration $\mathbf{w}^*$:

$$\mathbf{w}^* = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y} \tag{10}$$

If you're interested in viewing how this equation is obtained, you can go through this blog post: https://towardsdatascience.com/analytical-solution-of-linear-regression-a0e870b038d5.

Your tasks For carrying out these tasks, you are allowed to use the standard Python libraries + `matplotlib`, `numpy`, and `pandas`. You are **not** allowed to use `scikit-learn`. The code is to be submitted as a fork of the following GitHub repository:
https://github.com/rug-oop-2023/assignment_1.
Submit your code along with a concise report specifying your design choices (e.g., why an attribute is public/private, *etc.*). The report is to consist of one markdown (`.md`) file to be appended to the `README.md` file already present in the repo.

1. (3 pts.) Using the code implemented in lecture 2 as a blueprint, implement a `MultipleLinearRegression` class using OOP principles described up to now. The class is to be implemented in a file called `multiple_linear_regression.py`.
   - This class should have a `train` method which calculates the parameters as from Equation (10) based on a given training dataset of observations and ground truth.
     - You can pick the format for the dataset you prefer; however, we suggest it to be encoded in a tabular/matrix form with the rows representing the n units and the columns representing the p features. Similarly, the ground truth should be a single columnm table/vector of size n.
   - Also, there should be a `predict` method that formulates a prediction for some given datapoints. *Tip: Be careful about the dimension of the incoming data.*
   - You can check the correct functioning of these features by finding any regression dataset and checking the values of the coefficients & predictions against the ones produced from any multiple linear regression package (`sklearn.linear_model.LinearRegression` is a good choice, but also using `lm` from `R` is fine). Note:
     (a) The dataset shouldn't have categorical features (need to be treated differently than continuous features)
     (b) No data normalization is necessary for this task, although nobody will tell you anything if you do it correctly
     (c) Also, no train/test split is required
2. (3 pts.) In a file called `regression_plotter.py`, implement one additional class, called `RegressionPlotter`, disjoint from `MultipleLinearRegression`, which handles the drawing of a linear regression line or plane on top of a scatterplot of data. This class should allow for some behaviors:

(a) When asked to plot a model with only one feature, it shall draw the regression line, with the single feature in the x axis and the target in the y axis.

(b) When asked to plot a model with two features, it shall draw the corresponding regression plane.

(c) There should be an option to plot a generic `MultipleLinearRegression` model with any number of features by producing a sequence of p 2D plots (NB: p here is the number of features used in the model), each plot presenting one feature in the x axis vs the target in the y axis.

Tips:

- *The behavior (b) and (c) may be conflicting. Sort out a way to produce the desired behavior (e.g., by letting the user choose the requested behavior)*
- *You can use `matplotlib` for producing the plots. The library has extensive support for 2D and 3D plotting and is able to produce vectors/matrices of plots (e.g., see `subplots`).*
- *It's suggested that the data be part of the argument of the aforementioned methods, or an attribute of the `RegressionPlotter` class. The user should be able to select data which might not necessarily coincide from the training data of the selected `MultipleLinearRegression` class.*

3. (3 pts.) In a file called `model_saver.py`, implement a `ModelSaver` class, disjoint from `MultipleLinearRegression`. This is a class which is able to handle the saving and loading of the parameters of a generic machine learning model using different formats (e.g., `csv`, `json`, `pickle`, *etc.*). Pick at least two formats. The format is set at initialization and should be modifiable by the user. The class works by getting and passing the parameters to `MultipleLinearRegression`. **You should implement `ModelSaver` such that it works independently of `MultipleLinearRegression`**. `ModelSaver` has two main methods:

   - A method that gets the parameters from a generic model and saves them.
   - A method that loads the parameters (saved on disk) for a generic model and sets them on the given model.

Tip: to implement this class independently of ***`MultipleLinearRegression`****, it is probable that you will need to make some modifications to the* ***`MultipleLinearRegression`*** *in order to conform to the request of* ***`ModelSaver`****.*

4. (Wrap-up) Create a `main.py` file where you put together all of the three pieces constructed up to now.

   - Create or download a (toy) dataset for linear regression and train a multiple linear regression model on it. Showcase the prediction method by operating the predictions, then comparing them w.r.t. the ground truth labels.
   - Given the model trained in the previous point, showcase the regression plotter by trying out the various functionalities on the dataset.
   -

5. (after November 29th—1 pt.) Update your code by adding support for Exceptions:

   (a) Type checks on the `MultipleLinearRegression`, `ModelSaver`, `RegressionPlotter`.

   (b) (*for extra 0.5 pt.*) Catch exception for singular $\mathbf{X}^\top\mathbf{X}$ matrix from Equation (10). This would cause the matrix to be noninvertible, which would raise an exception from whichever tool you are using to invert the matrix. The goal is to catch this exception and to re-raise it with a more fitting error message.

Checklist

- ☐ Docstrings. Pick the preferred style from this list. There is no need to convert the docstrings to HTML documentation.
- ☐ Type hints for arguments of methods
- ☐ Return type hints for methods
- ☐ Private & public attributes/methods
- ☐ Exceptions implemented
- ☐ Motivate decisions in report