Assignment 1: Multiple Linear Regression using OOP

Deadline: see Brightspace

For this assignment, you are tasked with implementing a **Multiple Linear Regression** model in Python using OOP principles. In addition, you will be re-using the **K-Nearest Neighbors** implementation seen in Tutorial 1, and create a wrapper (a.k.a. façade) on the **Lasso** model of `scikit-learn`. Don't get scared by the length of this document, the first part serves as a proper introduction to the theme and summarizes what we cover in the first lectures.

Submission The submission must be operated **both** on **GitHub** (by pushing the definitive version) and on **Brightspace**, by pasting the link corresponding to your GitHub repo.

Introduction – Linear Regression

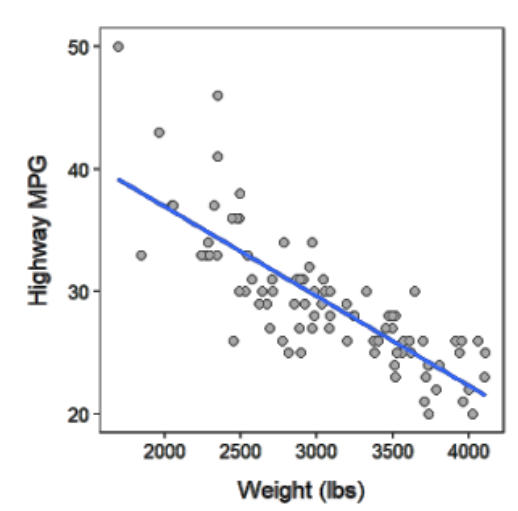
Linear regression is a Machine Learning (ML) model whose task is to take some data (**observations**) x and produce a **prediction** on a given variable y called **response**.

As an example, if we are managing a video streaming platform, x might be represented by the **characteristics of the user** (age, gender...) **and of the movies the user is watching and that they like** (title, genre, actors, directors...) and y, the quantity we want to predict, is the **time the user spends on our platform**. We might be interested, given a few information on a new user, to predict how much time they will spend on the platform. This will be carried out by gathering a limited number of samples $x_i, i \in \{1, \dots, n\}$ and the corresponding ground truths y_i. We will **train** the model on the data according to a given **loss/error/cost function** (which quantifies how *far* the prediction is from the ground truth value. We will then **deploy** the model on new data for which we have only the observations, but no ground truth.

Linear regression works on continuous data, i.e., all variables, both observations and response, are approximately represented by a real value. Examples of (approximately) continuous data are height and weight of a person, time spent watching a video, money in a bank account...

Simple Linear Regression

Simple Linear Regression is a special case of Linear Regression in which we have exactly 1 variable acting as observation and 1 response. During the lecture, we saw the problem of predicting a car fuel consumption (measured in miles per gallon—MPG) from its weight:



In simple linear regression, the idea is to fit a line $y = wx + b$ that passes through the points in order to capture the **linear trend** that connects x and y. w and b are the slope and intercept, respectively, of the line and are the parameters which we aim at training so that our line can fit the data. The problem of fitting a line through the data works perfectly (i.e., the line *interpolates* the points) when

- there are exactly two points, or
- there are more than two points, but they are all aligned.

These cases are non-existing in natural data: normally we have very large datasets and the there is a lot of noise (so points are not going to be aligned). Consequently, we relax the conditions by allowing the line to fit *optimally* the data, even with a minimal error.

The Mean Squared Error The optimality is determined by considering the loss function called Mean Squared Error (MSE). It measures the squared difference between a ground truth value and the corresponding prediction yielded by the line. For a generic observation $x^{(i)}, i \in \{1, \dots, n\}$, we have a

corresponding ground truth $y^{(i)}$, while the prediction $\hat{y}^{(i)} \doteq wx^{(i)} + b$. The MSE is calculated as

$$\text{MSE}(y^{(i)}, \hat{y}^{(i)}) \doteq \frac{\sum_i (y^{(i)} - \hat{y}^{(i)})^2}{n}. \tag{1}$$

You can immediately see how, if the prediction is perfect, than the error is 0; the further the prediction is from the ground truth value, the higher the error will be. It can be shown that the optimal parameters w^*, b^* that satisfy this condition are the following:

$$w^* = \frac{\sum^{(i)} \left[(x^{(i)} - \bar{x})(y^{(i)} - \bar{y})\right]}{(x^{(i)} - \bar{x})^2}; \quad b^* = \bar{y} - \bar{x}w^*, \tag{2}$$

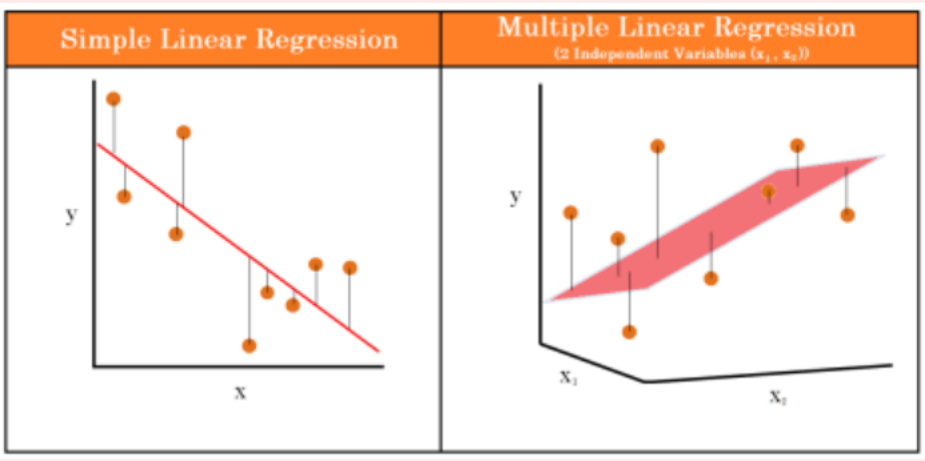
where $\bar{x}, \bar{y}$ are the mean of the $x^{(i)}$'s and $y^{(i)}$'s, respectively.

Multiple Linear Regression

Multiple linear regression is the multi-dimensional extension of simple linear regression. Instead of considering a relationship between a single observation variable and the response, **we expand the relationship to include p observation variables**:

$$\hat{y} = w_1 x_1 + \cdots + w_p x_p + b = \sum_{j=1}^{p} w_j x_j + b \tag{3}$$

By considering the previous example of the estimation of a car fuel consumption given the weight, we could extend it to multiple linear regression by considering an additional variable, such as maximum speed. From a geometric point of view, the **regression line** becomes a **regression plane**, with two observations (x_1, x_2) and one response:



By adding even more variables, the regression plane becomes a **regression hyperplane**. The geometric meaning of the intercept b is the same as in the line (value of y when the plane intercepts the y-axis—i.e., when all the x's are 0). The slope w_j instead represent the **individual contribution** of variable j in the prediction. Increasing the value of the variable j by 1 unit, while keeping all of the other variables constant, will increase the value of y by w_j.

Matrix formulation Next, we work out the extension of Equation (3) in matrix notation. Let us first consider the matrix $\mathbf{X}$ which represents our observations of n data points over p variables:

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & \dots & x_p^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(n)} & \dots & x_p^{(n)} \end{bmatrix}. \tag{4}$$

Analogously, we define the vector $\mathbf{y}$ containing our n responses as:

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix}. \tag{5}$$

And, we write the slope coefficients into a vector $\mathbf{w}$ of length p:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_p \end{bmatrix}. \tag{6}$$

We can thus write Equation (3) as:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b \tag{7}$$

We can furthermore incorporate the slope coefficient into $\mathbf{w}$:

$$\tilde{\mathbf{w}} \doteq \begin{bmatrix} w_1 \\ \vdots \\ w_p \\ b \end{bmatrix} \tag{8}$$

and we add a column of 1's to $\mathbf{X}$:

$$\tilde{\mathbf{X}} \doteq \begin{bmatrix} x_1^{(1)} & \dots & x_p^{(1)} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ x_1^{(n)} & \dots & x_p^{(n)} & 1 \end{bmatrix}. \tag{9}$$

In this way, we can express Equation (7) directly as:

$$\hat{\mathbf{y}} = \tilde{\mathbf{X}}\tilde{\mathbf{w}} \tag{10}$$

Quick dimension check: $\tilde{\mathbf{X}}$ is $n \times (p+1)$, while $\tilde{\mathbf{w}}$ is $(p+1) \times 1$, so, their multiplication is $n \times 1$.

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

$$\tilde{\mathbf{w}}^* = \left(\tilde{\mathbf{X}}^\top \tilde{\mathbf{X}}\right)^{-1} \tilde{\mathbf{X}}^\top \mathbf{y} \tag{11}$$

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` unless otherwise stated.

Remember to write your full names, student codes, and group number (as per Brightspace) in the repository. Add this information to the bottom of the README.md file.

Finally, remember to submit your code in time. Late submissions will not be accepted.

1 Multiple Linear Regression class

Using the code implemented in lecture 2, 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`.

- Create a `fit` method that calculates the parameters as from Equation (11) based on a given training dataset composed of observations and ground truth.
  - The observations and ground truth need to be passed as arguments of the `fit` method, not set as attributes in the constructor.
  - The observations and ground truth should be `np.ndarray`s. Make sure the number of samples is in the row dimension, while the variables are in the column dimension.
  - The parameters should be stored as a dict with `parameters` key, and the corresponding value as a single `np.ndarray`. Provide a **read-only** view of the parameters.
- Implement the `predict` method (described in Equation (10)). *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).
  Notes for the more ML-savvy people:
  1. The dataset shouldn't have categorical features (need to be treated differently than continuous features)

2. No data normalization is necessary for this task, although nobody will tell you anything if you do it correctly
3. Also, no train/test split is required

2 K-Nearest Neighbors

In a file called `k_nearest_neighbors.py`, put the code for the K-Nearest Neighbors model implemented in Tutorial 1. Be sure to keep the name of the class consistent to the code produced during the tutorial.

3 Lasso `scikit-learn` wrapper

In an additional file called `sklearn_wrap.py`, create a wrapper (or, in jargon, a façade) around the Lasso from `scikit-learn.linear_model.Lasso`. The Lasso should implement the same train/predict structure as the multiple linear regression and the K-Nearest Neighbors models. Lasso is a modified (in jargon, *regularized*) version of linear regression, in which the formula for the regression hyperplane is the same as in Equation (7), but the coefficients for slopes and intercept are obtained using a different loss function than the MSE contained in Equation (1).

- For what concerns the parameters, you should store them in a dict, using the same keys and values used by scikit-learn for this model. The parameters are the same that are stored by the scikit-learn Lasso class.
- For what concerns the constructor of the scikit-learn Lasso version, you should not let the user choose any of the constructor arguments—i.e., always use the default ones.

4 Main and other files

Add two datasets to the project:

- Download or craft a dataset for regression where all of the variables (observations + ground truth) are **continuous** (no categorical variables allowed) and another dataset for classification, where only the ground truth is categorical, while the observations are continuous. Be careful that both datasets are max 1MB in size.

- Put the datasets in the `data` folder of the repo.

Add a `main.py` file for showcasing your implementation.

- For each of the three models (multiple linear regression, k-nearest neighbors, lasso) showcase instantiation, fit, and predicting on the multiple linear regression; then, print the parameters.
- The multiple linear regression and lasso should be trained on the dataset for regression, while the k-nearest neighbors on the dataset for classification. Feel free to choose with any value for k.

In order for the `main.py` to run, please indicate the libraries you used in the `requirements.txt` file using the `pip` format.

Folder structure

Be careful to abide to this folder structure:

```
(root)
  |
  - models/            # your files for Parts 1-3
    |
    - __init__.py      # empty Python script for allowing imports
    - multiple_linear_regression.py
    - k_nearest_neighbors.py
    - sklearn_wrap.py
  |
  - data/              # put here your datasets for testing the models
    |
    ...
  - .github/workflows/ # automatic tests - DON'T MODIFY
  - main.py
  - requirements.txt
```

Checklist

- ☐ Use docstrings for documenting your code
- ☐ Remember type hints for all the methods and arguments (excluding `self` in methods). Help yourself with `ty` or `mypy`.

- ☐ Ensure that your code runs not only on your local machine (tips: **always avoid** absolute file paths, ensure `main.py` is running correctly before the final push, abide to the folder structure illustrated above, include the correct libraries on `requirements.txt`).
- ☐ PUSH EVERYTHING AND OPERATE SUBMISSION ON BRIGHTSPACE

Automatic tests

A series of automatic tests will be executed after each successful push to your GitHub repositories. These tests will serve to both you and us for checking the correctness of some properties of your code. The tests we will run are the following:

- `flake8` style check
- reproducibility check (i.e., the libraries indicated in `requirements.txt` can be correctly installed and the main runs without exceptions)
- structural check (i.e., the basic structure of the repo and classes is abided to)—check the results manually (test details→*Run submission check* task)

Grading information

While not releasing an official rubric before the submission deadline, here is some info on how the grading will be computed.

1. A grade on the 0→10 scale is computed by considering the following distribution:
    - Part 1: up to 5 pts.
    - Part 2: up to 1 pt.
    - Part 3: up to 4 pts.
2. The code **must** run in order to consider the project as eligible for a grade:
    - A code for which the main does not run because of import errors or wrong folder structure will be automatically given a minimum grade. An empty main will also be treated as such.

- If the execution succeeds partially, the code will be graded up to the point in which the exception occurs. *Example: Multiple Linear Regression can fit, but not predict → only the fit method will be graded.*
- *Tip: ensure that the reproducibility automatic test passes to be sure you did everything correctly!*

3. Penalties:

   - Each leakage of a private attribute will be penalized up to 1 pt.
   - Each unmotivated/wrongly motivated public attribute or method will be penalized up to 0.5 pts.
   - Each `flake8` style violation (according to automatic tests) will be penalized by 0.2 pt.
   - Each violation of the Python naming conventions (e.g., attributes and methods named using camelCase) will be penalized by 0.2 pts. Make reference to the slides for style for additional info.
   - Each missing docstring (for classes and methods) will be penalized by 0.25 pts.
   - Excessively long methods (that can be split into multiple sub-methods) will be penalized by 0.25 pts.
   - Meaningless/uninformative/misleading variable names will be penalized by up to 0.25 pts.
   - Each missing typing will be penalized by 0.5 pts.

4. **Bonus: consistent usage of meaningful and exhaustive git commit messages will be granted up to a 0.5 pts. bonus.** The grade cannot anyway be higher than 10 in this case.

After computing the grade in the 0→10 scale, this will be adjusted in the 1→10 scale by applying the following formula:

$$[1\rightarrow 10 \text{ grade}] = [0\rightarrow 10 \text{ grade}] \cdot 0.9 + 1$$

Finally, rounding is applied to the 2^{nd} decimal number.