

**Building Machine – Learning Models from Scratch**

**For All Novice Programmers**

Kentaro Suzuki

Department of Psychology, York University

PSYC 6273: Computer programming for experimental psychology

Dr. Richard Murray

Dec 17, 2024

## Introduction

This research examines whether beginners in Python can create machine learning (ML) models from scratch without using pre-built modules. To fully understand how ML modules are made, exploring the algorithms and calculations underlying them is essential. Harnessing the programming skills gained from the Psych6273 course, alongside insights from a Python programming book and free resources like YouTube, Kaggle, and GitHub, this study aims to not only deepen my understanding of Python but also build a strong foundation in research and data analysis for my master's thesis. Through hands-on projects, I explored and developed machine learning models such as k-Nearest Neighbors (k-NN), Linear Regression, and Logistic Regression, translating theoretical concepts into practical, real-world applications. These models have been carefully selected to align with my research design, which involves creating psychological items and questionnaires using large language models. Specifically, k-NN will facilitate classification tasks and cluster analysis for the newly generated psychometric items, while Linear Regression and Logistic Regression serve as essential components within the structural equation modelling framework used to evaluate these psychological items. Mastering the fundamentals of these models is critical to ensuring the success of my data analysis and advancing my research objectives.

By meticulously dissecting each step and coding block, I provide practical guidance for individuals with minimal Python experience (near zero) to comprehend and apply these machine-learning models in psychological studies. I inspire fellow psychological researchers to embrace advanced statistical tools, including machine learning and large language models, to push the boundaries of our field. This study serves as a candid journal, documenting the successes and

challenges of my learning and offering my insights and encouragement to those venturing into these transformative psychological methods.

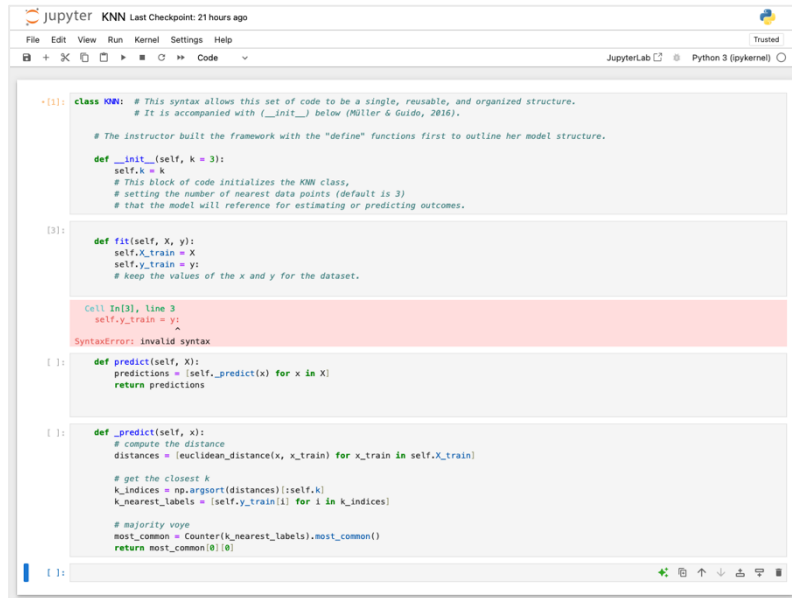
## **Methods**

I primarily relied on the free machine-learning tutorials available on YouTube, especially those offered by AssemblyAI Channel, a company specializing in speech AI models (AssemblyAI, 2022). These tutorials served as the foundation for building all my machine-learning models, including K-Nearest Neighbours (k-NN), Linear Regression, and Logistic Regression. Instead of using preexisting Python modules, I reviewed and applied the mathematical algorithms behind these models, created blocks of code that work as modules and ran real-life data used for psychological studies.

For executing the machine learning models, the instructors in the video presentations used Visual Studio Code as their coding platform, considering the anticipated and rather frequent coding errors due to lack of experience, I opted to use Jupyter Notebook, which allowed me to debug the code line by line or each block at a time (Figure 1). Furthermore, the instructor in each video provided verbal explanations of the coding choices and techniques used in the model. However, these explanations were often incomprehensible to a novice coder like me. Therefore, I recorded all the unknown coding language, jargon, statistical concepts, and techniques in my transcript as well as my coding file as I followed the video instructions and supplemented them with detailed explanations found in sources such as Python instructional books and literature per coding page (Figure 2).

**Figure 1**

*Jupyter Notebook – Identification of the Syntax Error Per a Code Block and Code Description.*



```
[1]: class KNN: # This syntax allows this set of code to be a single, reusable, and organized structure.
    # It is accompanied with (.__init__) below (Miller & Guido, 2016).
    # The instructor built the framework with the "define" functions first to outline her model structure.

    def __init__(self, k = 3):
        self.k = k
        # This block of code initializes the KNN class,
        # setting the number of nearest data points (default is 3)
        # that the model will reference for estimating or predicting outcomes.

[2]: def fit(self, X, y):
    self.X_train = X
    self.y_train = y:
    # Keep the values of the x and y for the dataset.

Cell In[3], line 3
self.y_train = y:
^
SyntaxError: invalid syntax

[ ]: def predict(self, X):
    predictions = [self._predict(x) for x in X]
    return predictions

[ ]: def _predict(self, x):
    # compute the distance
    distances = [euclidean_distance(x, x_train) for x_train in self.X_train]

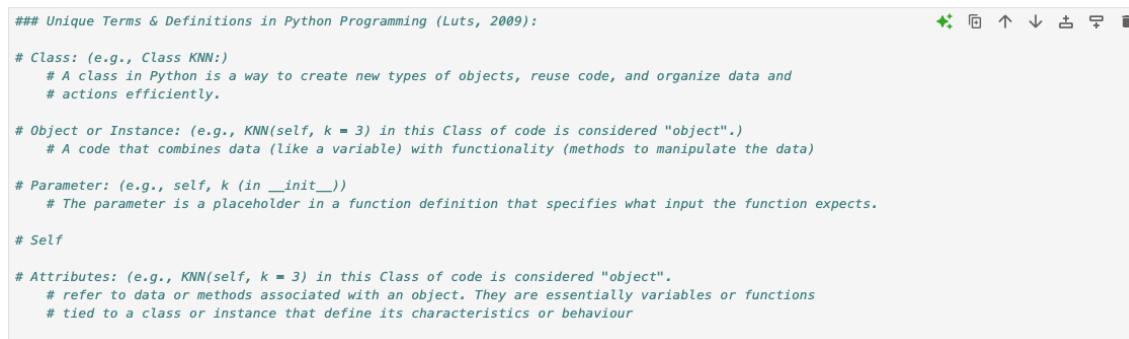
    # get the closest k
    k_indices = np.argsort(distances)[:self.k]
    k_nearest_labels = [self.y_train[i] for i in k_indices]

    # majority vote
    most_common = Counter(k_nearest_labels).most_common()
    return most_common[0][0]
```

*Note.* Jupyter Notebook allows users to verify their coding accuracy by first indicating the errors in orange, and then the platform provides the necessary correction methods when the error is detected.

**Figure 2**

*Sample Image of a Note for Unknown Terms and Concepts*



```
### Unique Terms & Definitions in Python Programming (Luts, 2009):

# Class: (e.g., Class KNN)
# A class in Python is a way to create new types of objects, reuse code, and organize data and
# actions efficiently.

# Object or Instance: (e.g., KNN(self, k = 3) in this Class of code is considered "object".)
# A code that combines data (like a variable) with functionality (methods to manipulate the data)

# Parameter: (e.g., self, k (in __init__))
# The parameter is a placeholder in a function definition that specifies what input the function expects.

# Self

# Attributes: (e.g., KNN(self, k = 3) in this Class of code is considered "object".)
# refer to data or methods associated with an object. They are essentially variables or functions
# tied to a class or instance that define its characteristics or behaviour
```

### ***Dataset – PISA 2018:***

The Programme for International Student Assessment (PISA), conducted triennially by the Organization for Economic Co-operation and Development, is an assessment that evaluates the academic performance of 15-year-olds in mathematics, science, and reading. I utilized the preprocessed dataset from Puah (2021), specifically the PISA2018 dataset, to analyze the academic performance of students in these subjects. The dataset comprises student performance scores, demographic details, and educational indicators, helping researchers to detect global educational trends and evaluate the effects of educational policies. In Puah's comparative ML study (Puah, 2021), machine learning models were compared to standard multiple linear regression (MLR) using PISA 2015 data. This study sets benchmarks for predictive performance and the processing of large-scale educational data. The dataset consisted of 198,712 students from 60 countries, with an average scientific literacy score of 492, a highest average score of 563, and a lowest average score of 371. Puah's dataset and article are accessible on the Open Science Framework (Puah, 2021). By employing a Variable Importance function from an existing random forest model, I narrowed down the variables to 12 (see Table 1) for my present study.

Additionally, given the current computing power, I decided to randomly select 300 observations from the master file. The significant reduction in data size aimed to decrease processing time and computational costs. Furthermore, variable names have been modified to improve clarity and ease of understanding of the statistical algorithms.

**Table 1.**

*Twelve Variables from PISA 2018*

STU_SCI_LITERACY	Scientific literacy performance of each student
AGE	Student's age
ECON_SOCIAL_CUL	Economic, social, and cultural status.
ED_RESOURCE	Availability of educational resources at home.
ENV_AWARENESS	Awareness of environmental matters (pollutions, climate, etc.).
HOUSE_POS	Home possessions
LANGUAGE	Language spoken at home.
PARENT_ED	Parents' education level
SCI_KNOWLEDGE	Knowledge about the nature, development, and justification of science.
SELF_EFFICACY	Student's self-efficacy about science
TECH_RESOURCE	Availability of technological resources at home.
WEALTH	Family wealth.

**Note.** The variable names in the sample dataset have been modified based on their descriptions to reflect the nature of the data and are different from the original PISA 2018 dataset.

### **Data Split for Training / Validation / and Test**

I split the data into three subsets following a 70/20/10 split for all three machine-learning models. The training set contains 70% of the observations used to train the models. These observations are randomly selected but made reproducible by setting *random\_state* = 42 to ensure consistency across runs and allow others to verify the results. The validation set consists of 20% of the data, which is used to fine-tune the model's hyperparameters and assess its performance on unseen data during the training phase. This helps prevent overfitting –Overfitting refers to a state in which a trained machine learning model becomes overly specialized to the training data, resulting in poor generalization performance on unseen data. Finally, the test set

contains the remaining 10% of the data, which is used to evaluate the model's final performance on entirely new data, providing an unbiased measure of its predictive accuracy.

### **Model Building 1: K-Nearest Neighbours: Basics & Framework**

K-Nearest Neighbors (k-NN) is a simple, non-parametric machine learning algorithm used for classification and regression tasks (Müller & Guido, 2016). In everyday language, k-NN is a simple way to make decisions based on what is common or average among similar data points. In practical psychological studies, the k-NN algorithm can be utilized in various contexts, such as predicting behaviours or preferences (Raschka et al., 2022).

For instance, Li et al. (2018) investigated how the number of electroencephalography (EEG) channels and frequency bands influence emotion recognition accuracy in the valence (pleasure) and arousal (activation) dimensions. The researchers discovered that employing more brain signal channels and concentrating on faster brain waves enhanced the accuracy of emotion identification in the data from 32 individuals. In this study, the researchers employed a k-NN algorithm to classify emotional states from EEG data.

In this study, I employed the k-NN model to predict the language spoken at home (English or Non-English) based on the student's parents' wealth. To achieve this, Euclidean distance calculation is a crucial component of the coding process.

#### ***Euclidean distance in k-NN***

Euclidean distance quantifies the shortest straight-line distance between two points, irrespective of the number of dimensions. It serves as the most prevalent method for determining the distance between two coordinates, representing the predicted value and the actual observation

(Pandey, 2021) . For two points (x, y) and (a, b), the Euclidean distance is calculated using the formula:

$$\text{distance} = \sqrt{(\text{SCI.KNOWLEDGE}_{\text{pred}_i} - \text{SCI.KNOWLEDGE}_i)^2}$$

Where:

- $\text{SCI.KNOWLEDGE}_{\text{pred}_i}$ : The predicted value of the target variable  $\text{SCI.KNOWLEDGE}$  for the i-th observation. This is the value produced by the model.
- $\text{SCI.KNOWLEDGE}_i$ : The actual (true) value of the target variable  $\text{SCI.KNOWLEDGE}$  for the i-th observation. This is the real value from the dataset.

## Model Building 2: Simple Linear Regression

Simple Linear Regression is a method used to model the linear relationship between one or more independent variables on the outcome of a linearity. This relationship can then be used to predict the values of the dependent variable based on the changes in the predictors. The equation used for this study is:

$$\text{STU.SCI.LITERACY} = \beta_0 + \beta_1 \cdot \text{HOUSE.POS} \quad (4)$$

Where:

- $\text{STU.SCI.LITERACY}$ : is the outcome variable, representing student's level of science literacy.
- $\beta_0$  : The intercept (value of Y when X = 0).
- $\beta_1$ : The regression coefficient (slope) for the predictor.
- $\text{HOUSE.POS}$  : The independent variable represents the composite score that indicates that they own their house).



To estimate the student's science literacy level, the linear regression model utilizes a function that optimizes estimation performance by minimizing error terms called Gradient Descent.

### ***Gradient Descent (MSE)***

To obtain the most accurate regression results, the regression model was computed using Gradient Descent. This optimization algorithm systematically reduces the loss function, which represents the error terms (MSE) in linear regression, by iteratively adjusting the model parameters, specifically the weight (w) and bias (b).

The objective is to minimize the Mean Squared Error (MSE) between the predicted and actual values in the model by gradually updating the weights and biases in a direction that reduces the loss (Prasad, 2020). This process is repeated in a calculation called the loss function. Consequently, the machine learning model and its programming code strictly adhere to a specific sequence when determining the most accurate predictive outcome:

- 1) Initialization: In this step, the weights (w) and biases (b) are set to zero.
- 2) Prediction: The prediction formula is:

$$\hat{y} = x \cdot w + b \quad (5)$$

- 3) Calculate MSE and obtain the loss.
- 4) Calculate the gradient descent and obtain the optimal balance between w and b. This process continues until w and b are in an optimal balance. (Prasad, 2020)

The Gradient Decent for weight and bias are calculated as:

For Weight:

$$\frac{\partial \text{Loss}}{\partial w} = \frac{2}{n} \sum_{i=1}^n (y_{\text{pred}_i} - y_i) x_i \quad (6)$$

Where:

- $\frac{\partial \text{Loss}}{\partial w}$ : represents the partial loss of weight which shows how much the loss will change if the w (weight) is adjusted.

For bias:

$$\frac{\partial \text{Loss}}{\partial b} = \frac{2}{n} \sum_{i=1}^n (y_{\text{pred}_i} - y_i) \quad (7)$$

- $\frac{\partial \text{Loss}}{\partial b}$ : It tells you how much the loss (error) will change if the bias (b) is adjusted.

Another critical component that makes the linear model unique is the learning rate. In short, this value allows users to prioritize between the speed of computational processing or accuracy of the estimation based on their research intent.

### ***Setting a Learning Rate:***

The course-provided programming code included a block of commands where I had to specify a learning rate. This learning rate determines how quickly the statistical model learns from the data and makes predictions. By adjusting this value, the model can strike a balance between training speed and accuracy.

More specifically, the provided code further illustrates the close relationship between the gradient descent and learning rate (controlled balance between the weight and the bias). This value essentially acts like a coefficient (or agent) that scales the influence of the model's weights and bias. The actual code is:

$$\text{self.weights} = \text{self.weights} - \text{self.lr} * dw \quad (8)$$

Where:

- `self.weights`: represents how much influence each predictor has on the prediction
- `self.lr`: represents how big or small the learning increment you desire
- `dw`: represents the direction and magnitude of change for reducing the error term.

Since this line of code is part of an iterative process, the learning rate plays a key role in determining the magnitude of weight adjustments during each iteration of gradient descent. This process continues until the optimal weights are reached, minimizing the error and improving the model's predictions.

A small learning rate results in slow progress, making the model converge gradually. In contrast, a large learning rate speeds up the learning process but risks overshooting the optimal solution or failing to converge altogether (Bhattbhatt, 2024). Choosing the learning rate is often a matter of trial and error because the optimal value is unknown beforehand (Bhattbhatt, 2024). The result of changing this parameter was recorded in the results section of this present study.

### **Model Building 3: Logistic Regression**

The equation looked like this if we use WEALTH as our predictor:

$$p(LANGUAGE) = \beta_0 + \beta_1 \cdot WEALTH \quad (9)$$

Where:

- $p$  = probability that Language being 1 (other than English is spoken at home).
- $\beta_1 \cdot WEALTH$  : coefficients showing the effect of WEALTH on the LANGUAGE spoken at home.

### ***Cross Entropy (Loss Function)***

Cross-entropy is a common measure used in classification problems to evaluate how well a model's predicted probabilities match the actual outcomes (Pykes, 2024). It penalizes incorrect predictions more heavily, ensuring the model improves its accuracy during training. If the predicted probability is close to the true class, the loss is small; if it is far off, the loss is large.

Cross-entropy is widely used in models like Logistic Regression and Neural Networks to optimize performance by minimizing error. Implementing it often involves experimentation, such as adjusting learning rates, to achieve the best results (Pykes, 2024). The cross-entropy loss function for binary classification is mathematically represented as:

$$J(w, b) = -\frac{1}{N} \sum_{i=1}^N \left[ \text{Language}^i \log \left( h_{\theta}(\text{WEALTH}^i) \right) + (1 - \text{Language}^i) \log \left( 1 - h_{\theta}(\text{WEALTH}^i) \right) \right] \quad (10)$$

Where:

- $J(w, b)$ : It measures how well the model's predicted probabilities match the actual outcomes
- $\text{Language}^i$  and  $(1 - \text{Language}^i)$ : takes into account the binary Language variable ( $1 - \text{Language} = \text{absence of speaking English at home}$ ;  $\text{Language}^i = \text{English is spoken at home}$ )
- $\log \left( h_{\theta}(\text{WEALTH}^i) \right)$ : Observation of the wealth of the people who speak English at home.
- $\log \left( 1 - h_{\theta}(\text{WEALTH}^i) \right)$ : Observation of wealth of the people who speak other than English at home.

## Models Evaluation Metrics:

### *R Squared ( $R^2$ ) – Coefficient of Determination*

$R^2$ , a measure of the goodness of fit between a model's predictions and actual data, ranges from 0 to 1. A value of 1 indicates perfect accuracy, while a value closer to 0 suggests poor accuracy. So, a higher  $R^2$  means better prediction accuracy.  $R^2$  is expressed as

$$R^2 = \frac{SS_{res}}{SS_{tot}} \quad (11)$$

Where:

- $SS_{res}$ , also known as Residual Sum of Squares, is the sum of the squared differences between the observed values ( $y_i$ ) and the predicted values ( $\hat{y}_i$ ) obtained from a model.

It can be calculated using the formula:

$$SS_{res} = \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (12)$$

- $SS_{tot}$  (Total Sum of Squares) quantifies the overall variability of the data. It's calculated by summing up the squared differences between each observed value ( $y_i$ ) and the average of all observed values ( $\bar{y}$ ).

The formula for  $SS_{tot}$  is:

$$SS_{tot} = \sum_{i=1}^n (y_i - \bar{y})^2 \quad (13)$$

### *Mean Squared Error (MSE)*

Mean Squared Error (MSE) is a commonly used loss function in regression analysis for evaluating the accuracy of a model's predictions. It is determined by calculating the average of

the squared differences between the actual and predicted outcomes. This approach guarantees that all errors are positive and places greater emphasis on larger discrepancies. (James et al., 2013) It quantifies the degree of deviation in predictions compared to actual results, offering a clear method for evaluating accuracy of predictions.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2 \quad (14)$$

Where:

- $Y_i$  is the true value for the i-th data point.
- $\hat{Y}_i$  is the predicted value for the i-th data point.
- $n$  is the number of observations

#### **Availability of Code files and Dataset:**

All code files are available at my GitHub repository  
(<https://github.com/KennedyTO/Psyc6273/tree/main>).

#### **Coding Environment:**

##### ***Software and Modules:***

I used Jupyter Notebook (Conda) on MacOS Sequoia 15.2. All necessary modules and functions are listed in the code files.

##### ***Hardware:***

M1 MacBook Air (2020 model), Memory 8 GB

## Results

### k-NN model (Appendix A1)

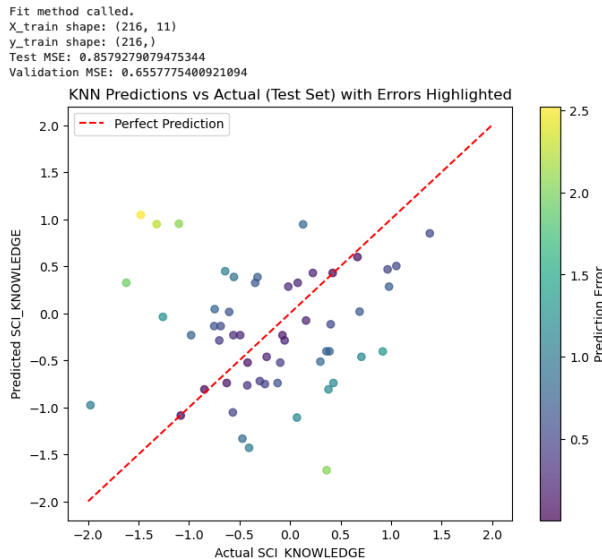
The Test Mean Squared Error (MSE) was 0.8579, which represented the average squared difference between the predicted and actual values on the test set (Figure 1). A lower MSE indicated better predictive accuracy. In this case, the value was reasonable but not exceptionally low, suggesting that there was some prediction error.

The Validation MSE was 0.6558, which was smaller than the test MSE. This indicated that the model performed slightly better on the validation set compared to the test set.

Overall, the model performed reasonably well with moderate prediction error. The small but noticeable difference between the test MSE and validation MSE suggested that the model generalized fairly well. However, it might have benefited from further fine-tuning, such as increasing the data size. If the validation error had remained significantly smaller than the test error on larger datasets, this could have indicated overfitting. However, in this case, the errors were relatively close, so overfitting did not appear to be a major concern.

**Figure 3**

*k*-NN Prediction vs. Actual Observation with Prediction Error Rate



**Note:** The darker the data point, the closer the proximity between predictions and actual values observations.

### Linear Regression (Appendix A2)

The summary provided by the Statsmodels function and the manual calculations showed both similarities and discrepancies. The  $R^2$  value calculated by the module was 0.022, indicating that the model explained only 2.2% of the variance in the student's science literacy. However, the manual calculation yielded a negative  $R^2$  value of -0.0156, suggesting inconsistency and a possible error in the manual computation.

Another discrepancy was the  $F$ -statistic in Statsmodels, which was 4.780 with a  $p$ -value of 0.0299. This indicated that the predictor variable was statistically significant ( $p < 0.05$ ). However, the manual result presented an invalid negative  $F$ -statistic of -0.3859, indicating further computational errors.

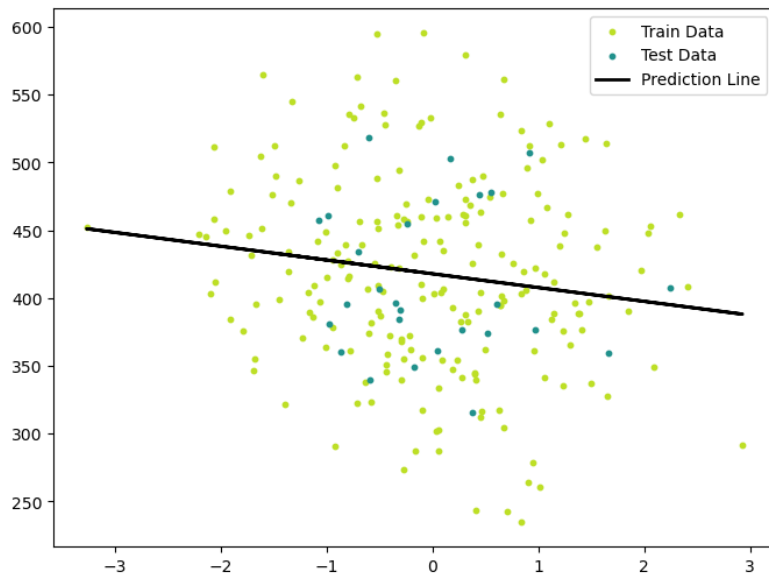


Despite these discrepancies, both approaches agreed on the coefficients: the intercept (417.9383) and slope (-10.1550), as well as the significance of house ownership ( $t = -2.186$ ,  $p$ -value: 0.030, degree of freedom: not available in this report). Moreover, the Residual Standard Error of 57.2407 remained consistent across both outputs. Additionally, the residual diagnostics in Statsmodels, such as skewness (0.023) and kurtosis (2.771), indicated that the residuals were well-behaved.

Overall, while the coefficients and error spread calculations were consistent, the discrepancies in R-squared and F-statistic values suggested that the results from Statsmodels were more reliable than those obtained through manual computations. The scatter plot (Figure 4) visually reinforced the earlier findings of the linear regression model. The prediction line showed a slight downward slope, reflecting the negative coefficient (-10.1550), but the data points were widely scattered around it, indicating a weak relationship between the predictor and the target variable. This aligned with the low R-squared value (0.022), confirming that the model explained very little of the variability in the data. The overlap between the training and testing data suggested the test set was representative, but the high variability around the line, reflected in the large residual standard error (57.2407), underscored the model's poor predictive accuracy.

**Figure 4**

*Prediction of Student's Science Literacy Based on Parent's Home Ownership.*



**Note.** Y axis is the literacy score, and the X axis is the parents' homeownership

### **Logistic Regression (Appendix A3)**

The logistic regression model was evaluated using both a manually implemented logistic regression algorithm and a Scikit-learn function. Both approaches yielded the same accuracy of 0.6667, indicating consistent performance across methods. The model utilized a single predictor variable, WEALTH, to classify the binary outcome variable, LANGUAGE.

### **Discussion**

This research investigated whether beginners like myself with less experience in Python could build machine learning models from scratch. The goal was to enhance my understanding of Python programming while laying a foundation for advanced statistical modeling in future research and data analysis for my master's thesis. Free YouTube tutorials, particularly those by

AssemblyAI, served as primary resources for developing k-Nearest Neighbors (k-NN), Linear Regression, and Logistic Regression models. Jupyter Notebook was chosen for its user-friendly interface, which allowed me to write code in small, manageable blocks and provided helpful suggestions when errors occurred. Throughout the project, debugging processes and unfamiliar coding concepts were carefully documented and explained using various additional resources.

Before evaluating the model's performance, it is worth reflecting on the key takeaways from this experience: First, I discovered that as long as the mathematical framework is available, it is possible to build machine-learning models from scratch. Initially, the process of creating such models seemed abstract, overwhelming, and quite daunting, and I had no clear starting point. However, this project allowed me to bridge the gap between my prior knowledge of statistics and basic statistical machine learning, providing valuable insights into model construction.

Secondly, delving into the statistical structures in detail provided me with a profound comprehension of their underlying principles. By meticulously analyzing each equation component, I was able to visually interpret and grasp complex statistical concepts like Euclidean distance, gradient descent, learning rate, and cross-entropy. This process not only elucidated the functions of statistical models but also deepened my understanding of the types of parameters involved, transcending the realm of programming. Moreover, it equipped me with the knowledge of how to manipulate these parameters as needed.

Finally, I significantly enhanced my foundational Python skills and learned how to efficiently locate and utilize Python resources, which proved invaluable during the project. However, this last point also directly links to areas for improvement.

In my linear regression model, the manually calculated F-statistic differed from the value obtained using a pre-existing module, highlighting the need for further verification and a deeper understanding of diverse validation methods. Online resources predominantly focus on building machine learning models, with limited emphasis on verification and problem-solving techniques. This underscores the critical importance of verification skills, which are arguably as essential as, if not more so than, model construction skills.

However, circling back to my positive gains, I understood the potential risks of relying on existing modules with unquestioning trust, using their functions as if their accuracy were guaranteed. This experience highlighted the importance of adopting a more critical and inquisitive mindset when using preexisting functions—a valuable and significant lesson I gained through this process. I also gained confidence that the mathematical framework behind the machine-learning framework will guide novice statisticians like myself out of structural and programming troubles.

Based on these experiences, future research should focus on:

1. Strengthening the understanding of the mathematical and statistical foundations underpinning advanced machine learning models.
2. Reverse-engineering existing machine learning functions by starting with a thorough analysis of real-life scientific studies to explore, both statistically and mathematically, why these models work, how they are programmed, and the theoretical frameworks that form their foundations.

## References

- AssemblyAI (Director). (2022, September 12). *Machine Learning From Scratch Full course* [Video recording]. [https://www.youtube.com/watch?v=p1hGz0w\\_OCo](https://www.youtube.com/watch?v=p1hGz0w_OCo)
- Bhattbhatt, V. (2024, February 16). Learning Rate and Its Strategies in Neural Network Training. *The Deep Hub*. <https://medium.com/thedeephub/learning-rate-and-its-strategies-in-neural-network-training-270a91ea0e5c>
- James, G., Witten, D., Hastie, T., Tibshirani, R., & Taylor, J. (2023). Introduction. In G. James, D. Witten, T. Hastie, R. Tibshirani, & J. Taylor, *An Introduction to Statistical Learning* (pp. 1–13). Springer International Publishing. [https://doi.org/10.1007/978-3-031-38747-0\\_1](https://doi.org/10.1007/978-3-031-38747-0_1)
- Li, M., Xu, H., Liu, X., & Lu, S. (2018). Emotion recognition from multichannel EEG signals using K-nearest neighbor classification. *Technology and Health Care*, 26(Suppl 1), 509–519. <https://doi.org/10.3233/THC-174836>
- Müller, A., & Guido, S. (2016). *Introduction to Machine Learning with Python: A Guide for Data Scientists* (1st edition). O'Reilly Media.
- Pandey, A. (2021, January 8). *The Math Behind KNN*. Medium. <https://ai.plainenglish.io/the-math-behind-knn-7883aa8e314c>
- Prasad, A. (2020, September 16). Linear Regression With Gradient Descent Derivation. *Analytics Vidhya*. <https://medium.com/analytics-vidhya/linear-regression-with-gradient-descent-derivation-c10685ddf0f4>

Puah, S. (2021). *Predicting Students' Academic Performance: A Comparison between Traditional MLR and Machine Learning Methods with PISA 2015* [Preprint]. PsyArXiv.

<https://doi.org/10.31234/osf.io/2yshm>

Pykes, K. (2024, August 10). *Cross-Entropy Loss Function in Machine Learning: Enhancing Model Accuracy*. Cross\_Entropy Loss Function in Machine Learning: Enhancing Model Accuracy.

[https://www.datacamp.com/tutorial/the-cross-entropy-loss-function-in-machine-learning?utm\\_source=chatgpt.com](https://www.datacamp.com/tutorial/the-cross-entropy-loss-function-in-machine-learning?utm_source=chatgpt.com)

## **Appendix A1 k-NN**

# KNN Predictive Model From Scratch

This model was designed to predict students between 15 and 16 years of age science knowledge based on predictors.

## Unique Terms & Definitions in Python Programming:

### Class: (e.g., Class KNN:)

A class in Python is a way to create new types of objects, reuse code, and organize data and # actions efficiently (Luts, 2009).

### Object or Instance: (e.g., KNN(self, k = 3) in this Class of code is considered "object".)

A code that combines data like a variable with functionality or methods to manipulate the data (Luts, 2009)

### Attributes:

(e.g., KNN(self, k = 3) in this class of code is considered "object" refer to data or methods associated with an object. They are essentially variables or functions tied to a class or instance that define its characteristics or behaviour

## Model Building

(Citations and references are provided in the Code Interpretation Sections).

```
In [4]: import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from collections import Counter
import matplotlib.pyplot as plt
```

```
In [5]: # Define the Euclidean distance function
```



```

def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))

# Define the KNN class
class KNN:
    def __init__(self, k=3):
        self.k = k # Number of neighbors

    def fit(self, X, y):
        self.X_train = X # Save training features
        self.y_train = y.reset_index(drop=True) # Reset index of y
        print("Fit method called.")
        print(f"X_train shape: {self.X_train.shape}")
        print(f"y_train shape: {self.y_train.shape}")

    def predict(self, X):
        predictions = [self._predict(x) for x in X]
        return predictions

    def _predict(self, x):
        distances = [euclidean_distance(x, x_train) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]

```

## Code Interpretation Notes (1. KNN Model Building):

### Define the Euclidean distance function

```

def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))

```

Euclidean distance, widely regarded as the most intuitive metric, determines the straight-line distance between two data points in a multi-dimensional space. Its formula, derived from the Pythagorean theorem, is remarkably simple: (Miesle, 2024)

The equation was  $D = \sqrt{\sum (x_i - x_j)^2}$  the square root of the sum of the squared differences between the corresponding data points across.

**"class KNN: def *init*(self, k = 3): self.k = k"**

This block of code initializes the KNN "class" setting the number of nearest data points that the model will reference for # estimating or predicting outcomes (the instructor set this value to 3 for now.). All parameters must be specified in the

(*init*) function (Müller & Guido, 2016).

```
"def fit(self, X, y)":

    self.X_train = X
    self.y_train = y.reset_index(drop=True)
    print("Fit method called.")
    print(f"X_train shape: {self.X_train.shape}")
    print(f"y_train shape: {self.y_train.shape}")
```

Video Instruction (AssemblyAI, 2022): This block of code is part of the fit method in the KNN class. It stores the training features (X\_train) and the target values (y\_train) as attributes of the class for later use during prediction. The reset\_index(drop=True) ensures that the index of y\_train is reset to a default sequential order (0, 1, 2, ...) to avoid any mismatch with the positional indices of X\_train. The method also prints the shapes of X\_train and y\_train to confirm that the data has been saved correctly and to provide a quick check on the dimensions of the input data.

**def predict(self, X):**

```
    predictions = [self._predict(x) for x in X]
    return predictions
```

Video Instruction (AssemblyAI, 2022) The predict method takes a group of data points (X) and uses the \_predict method to find the prediction for each point. It collects these predictions in a list and returns them, allowing the model to make predictions for multiple points at once.

**The concept of "SELF" - Analogy by Microsoft Copilot (2024): Relationship of Class, Instance, and Self.**

1. Coffee Shop (CoffeeShop class): This is the main place where everything happens.
2. Manager and Barista (Manager and Barista classes): These are the people working in the coffee shop.

Now, imagine each person in the coffee shop has their own notepad:

- Manager's Notepad: This is where the manager writes important notes.
- Barista's Notepad: This is where the baristas write down orders.

When the coffee shop opens, everyone gets their own notepad:

- Manager (Alice): Alice gets her own notepad.
- Baristas (Bob and Charlie): Bob and Charlie each get their own notepads.

When Alice writes a note, it goes in *her* notepad. When Bob or Charlie takes an order, it goes in their notepads. Here, "self" is like each person's notepad:

- ``self`` in ``Manager`` class: Refers to Alice's notepad.
- ``self`` in ``Barista`` class: Refers to Bob's or Charlie's notepad.

In programming terms, ``self`` ensures that each person (instance) keeps track of their own stuff (attributes and methods), just like Alice, Bob, and Charlie each have their own notepads.

## 70/20/10 Split - KNN Model Prediction

(Citations are provided in the Code Interpretation Note).

```
In [8]: # Load the dataset
df = pd.read_csv("PISA2018_data.csv")

# Filter numeric columns only
numeric_df = df.select_dtypes(include=['number'])

# Split data into 70/20/10
X = numeric_df.drop('SCI_KNOWLEDGE', axis=1)
y = numeric_df['SCI_KNOWLEDGE']

# First split: 90% train-test, 10% validation
X_temp, X_val, y_temp, y_val = train_test_split(X, y, test_size=0.1, r

# Second split: 70% training, 20% testing (from the 90% data)
X_train, X_test, y_train, y_test = train_test_split(X_temp, y_temp, te

# Make variables scaled so that different unit variables can be compar
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
X_val_scaled = scaler.transform(X_val)
```

```

# Initialize and train the KNN model
k = 3 # Number of neighbors
knn_model = KNN(k=k)
knn_model.fit(X_train_scaled, y_train)

# Test the model
y_test_pred = knn_model.predict(X_test_scaled)
test_mse = mean_squared_error(y_test, y_test_pred)
print(f"Test MSE: {test_mse}")

# Validate the model
y_val_pred = knn_model.predict(X_val_scaled)
val_mse = mean_squared_error(y_val, y_val_pred)
print(f"Validation MSE: {val_mse}")

# Calculate errors for colour differentiation
errors = np.abs(y_test - y_test_pred)

# Create the scatter plot
plt.figure(figsize=(8, 6))
scatter = plt.scatter(y_test, y_test_pred, c = errors, cmap='viridis',
plt.colorbar(scatter, label='Prediction Error') # Add color bar to in
plt.plot([-2, 2], [-2, 2], 'r--', label='Perfect Prediction') # Add p
plt.xlabel("Actual SCI_KNOWLEDGE")
plt.ylabel("Predicted SCI_KNOWLEDGE")
plt.title("KNN Predictions vs Actual (Test Set) with Errors Highlighte
plt.legend()
plt.show()

```

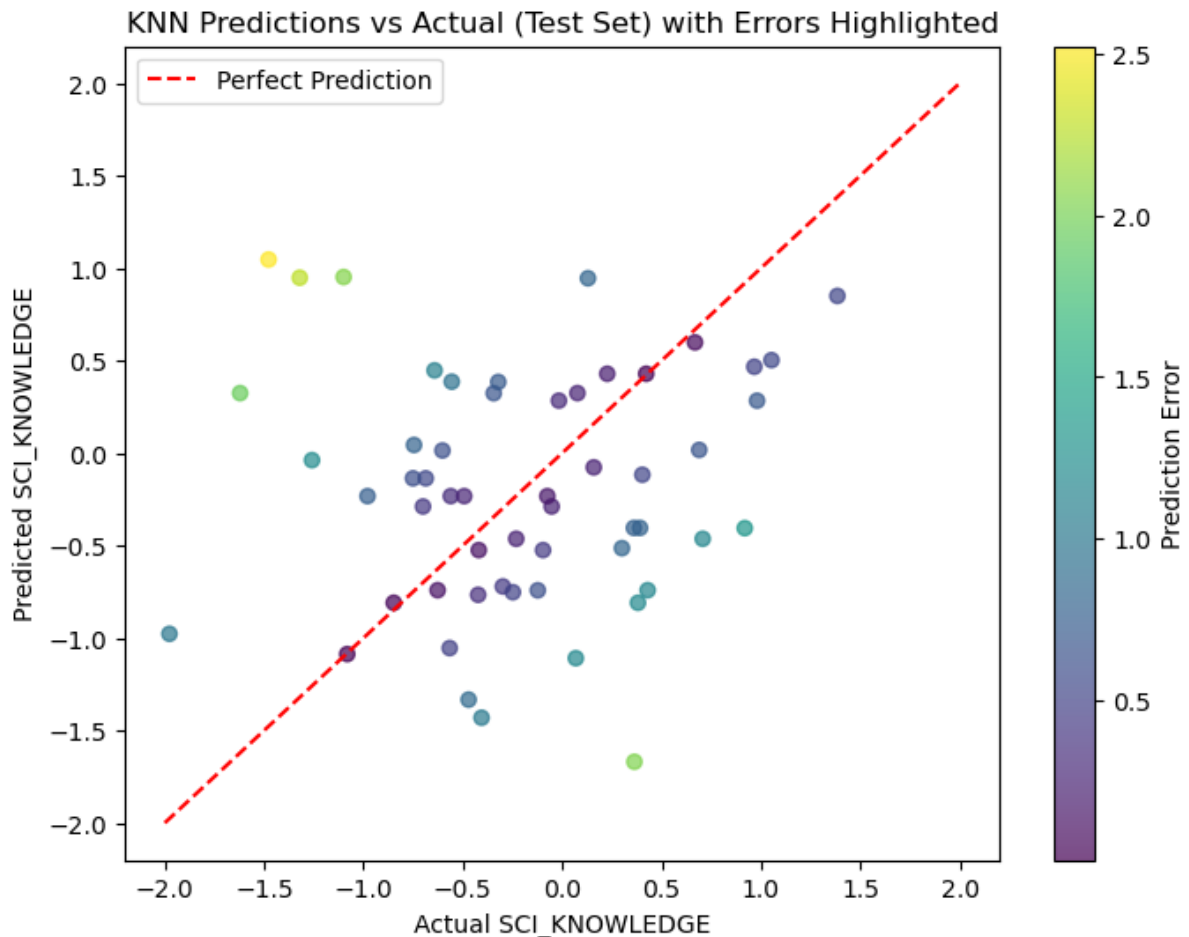
Fit method called.

X\_train shape: (216, 11)

y\_train shape: (216,)

Test MSE: 0.8579279079475344

Validation MSE: 0.6557775400921094



## Code Interpretation Note (2. KNN Model Training & Prediction):

`df.select_dtypes` (Pandas, 2024)

```
DataFrame.select_dtypes(include=None, exclude=None)
```

Return a subset of the DataFrame's columns based on the column types.

**Split data into 70/20/10 (scikit-learn, 2024a)**

```
X = numeric_df.drop('SCI_KNOWLEDGE', axis=1) # Features
y = numeric_df['SCI_KNOWLEDGE']             # Target
variable
```

This block splits the dataset into features (X) and the target variable (y). The `drop('SCI_KNOWLEDGE', axis=1)` removes the column `SCI_KNOWLEDGE` from the DataFrame `numeric_df`, leaving all other columns as features in X. The column `SCI_KNOWLEDGE` is then assigned to y as the target variable that the model will

predict.

### # First split: 90% train-test, 10% validation (AssemblyAI, 2022a)

```
X_temp, X_val, y_temp, y_val = train_test_split(X, y,  
test_size=0.1, random_state=42)
```

This line splits the data into two subsets: 90% for training and testing (X\_temp, y\_temp) and 10% for validation (X\_val, y\_val). The train\_test\_split function ensures the split is random, with test\_size=0.1 specifying that 10% of the data is allocated for validation. The random\_state=42 sets a seed to ensure the split is reproducible.

### # Make variables scaled so that different unit variables can be compared. (Scikit-learn, 2024a)

```
scaler = StandardScaler()  
X_train_scaled = scaler.fit_transform(X_train)  
X_test_scaled = scaler.transform(X_test)  
X_val_scaled = scaler.transform(X_val)
```

This block standardizes the variable units to have a mean of 0 and a variance of 1, which is important for distance-based models like KNN. The StandardScaler() is initialized, and the fit\_transform method is applied to X\_train to compute the scaling parameters (mean and standard deviation) and transform the training data. The transform method is then applied to X\_test and X\_val to scale them using the same parameters from X\_train, ensuring consistency across the datasets.

### # Initialize and train the KNN model

```
k = 3 # Number of neighbors  
knn_model = KNN(k=k)  
knn_model.fit(X_train_scaled, y_train)
```

This code block creates and trains a K-Nearest Neighbors (KNN) model. The variable k is set to 3, meaning the model will consider the 3 nearest neighbours when making predictions.

### Test the model (sklearn, 2024b) + Validate the model (sklearn, 2024b)

```

y_test_pred = knn_model.predict(X_test_scaled)
test_mse = mean_squared_error(y_test, y_test_pred)
print(f"Test MSE: {test_mse}")

y_val_pred = knn_model.predict(X_val_scaled)
val_mse = mean_squared_error(y_val, y_val_pred)
print(f"Validation MSE: {val_mse}")

```

These blocks test (with 20% of data) and validate (with 10% of data) the KNN model by predicting target values for the validation dataset (`X_val_scaled`) and calculating the Mean Squared Error (MSE) between the actual values (`y_val`) and the predicted values (`y_val_pred`). The MSE, a measure of prediction accuracy, is then printed, with lower values indicating better performance. This code references the "predict" function that I defined in the Model Building section.

### Showing the accuracy score(Stack overflow, 2015)

```

# Calculate errors for colour differentiation
errors = np.abs(y_test - y_test_pred)

```

Based on the visualization samples, I decided to add a colour differentiation per prediction based on the estimation accuracy compared to the read data points for the last 10% of the results. This value was stored as "errors" and was used in the plot as the 3rd dimension as the colour "c" in the plot.

### Create the scatter plot (Matplotlib, 2024)

```

plt.figure(figsize=(8, 6))
scatter = plt.scatter(y_test, y_test_pred, c = errors,
                      cmap='viridis', alpha=0.7)
plt.colorbar(scatter, label='Prediction Error')
plt.plot([-2, 2], [-2, 2], 'r--', label='Perfect
Prediction')
plt.xlabel("Actual SCI_KNOWLEDGE")
plt.ylabel("Predicted SCI_KNOWLEDGE")
plt.title("KNN Predictions vs Actual (Test Set) with
Errors Highlighted")
plt.legend()
plt.show()

```

Similarly to ggplot2 in R, there were helpful, quick reference PDFs on the Matplotlib website (they call them cheat sheets and handouts). You have to tinker around with the plot size and the overall look.

## References:

AssemblyAI (Director). (2022, September 11). How to implement KNN from scratch with Python (Vol. 1) [Youtube]. <https://www.youtube.com/watch?v=rTEtEy5o3X0>

codiearcher. (2019, September 10). Sklearn.accuracy\_score(y\_test, y\_predict) vs np.mean(y\_predict == y\_test) [Forum post]. Data Science Stack Exchange. <https://datascience.stackexchange.com/q/58961>

Lutz, M. (20). Learning Python: Powerful object-oriented programming; covers Python 2.6 and 3.x (4. ed., [Nachdr.]). O'Reilly.

Microsoft Copilot, L. L. M. (2024, December 13). Analogy of Self in Python Programming—Using a Coffee Shop. Microsoft Copilot: Your AI Companion. <https://copilot.microsoft.com/chats/md7pQ6w54MFWQX3GBmmWY>

Müller, A., & Guido, S. (2016). Introduction to Machine Learning with Python: A Guide for Data Scientists (1st edition). O'Reilly Media.

Numpy. (2024). NumPy user guide—NumPy v2.1 Manual [NumPy user guide]. Numpy. <https://numpy.org/doc/stable/user/index.html#user>

NumPy. (2024). numpy.argsort—NumPy v2.1 Manual. <https://numpy.org/doc/stable/reference/generated/numpy.argsort.html#numpy-argsort>

Pandas. (2024). pandas.DataFrame.select\_dtypes—Pandas 2.2.3 documentation. API Reference > DataFramepandas > pandasDataFrame (Pandas.DataFrame.Select\_dtypes). [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.select\\_dtypes.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.select_dtypes.html)

Python. (2024a). collections—Container datatypes. Python Documentation. <https://docs.python.org/3/library/collections.html>

Python, R. (2024b, Juy). Split Your Dataset With scikit-learn's train\_test\_split() – Real Python. <https://realpython.com/train-test-split-python-data/>

scikit-learn. (2024a). StandardScaler. Scikit-Learn. <https://scikit-learn/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>

scikit-learn. (2024b). Train\_test\_split. Train\_test\_split. [https://scikit-learn/stable/modules/generated/sklearn.model\\_selection.train\\_test\\_split.html](https://scikit-learn/stable/modules/generated/sklearn.model_selection.train_test_split.html)

sklearn. (2024). Mean\_squared\_error. Mean\_squared\_error. [https://scikit-learn/stable/modules/generated/sklearn.metrics.mean\\_squared\\_error.html](https://scikit-learn/stable/modules/generated/sklearn.metrics.mean_squared_error.html)



[learn/stable/modules/generated/sklearn.metrics.mean\\_squared\\_error.html](#)  
\_error.html

In [ ]:

In [ ]:

## **Appendix A2 Linear Regression**

# Linear Regression from Scratch!

## Model Building

```
In [17]: import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import statsmodels.api as sm
import matplotlib.pyplot as plt
```

```
In [19]: class LinearRegression:

    # Initialize the linear regression model with learning rate and number of iterations
    def __init__(self, lr=0.001, n_iters=1000):
        self.lr = lr
        self.n_iters = n_iters
        self.weights = None
        self.bias = None

    # Fit the linear regression model to the training data
    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        # Perform gradient descent in LOOP for the specified number of iterations
        for _ in range(self.n_iters):
            y_pred = np.dot(X, self.weights) + self.bias
            dw = (1 / n_samples) * np.dot(X.T, (y_pred - y))
            db = (1 / n_samples) * np.sum(y_pred - y)

            # Update weights and bias using the gradients and learning rate
            self.weights = self.weights - self.lr * dw
            self.bias = self.bias - self.lr * db

        # Predict target values for new input data
    def predict(self, X):
        return np.dot(X, self.weights) + self.bias
```

## Code Interpretation & Notes for Model Building

(AssemblyAI, 2022)

Defining the model parameters (lr stands for linear regression)

```
def __init__(self, lr = 0.001, n_iters=1000):  
    self.lr = lr  
    self.n_iters = n_iters  
    self.weights = None  
    self.bias = None
```

The description and calculation of each component is available in my transcript. The parameters `lr` (learning rate, default value 0.001) and `n_iters` (number of iterations, default 1000) are set to control the optimization process. The variables `self.weights` and `self.bias` are initialized as `None` and will later store the model's parameters (weights for input features and bias/intercept term). This setup prepares the class to learn these parameters during training (AssemblyAI, 2022).

## Fitting the model

```
def fit(self, X, y):  
    n_samples, n_features = X.shape  
    self.weights = np.zeros(n_features)  
    self.bias = 0  
  
    # This is the part where the weight and bias are  
    # optimized (Details can be found in my transcript)  
    for _ in range(self.n_iters):  
        y_pred = np.dot(X, self.weights) + self.bias  
  
        dw = (1/n_samples) * np.dot(X.T, (y_pred-y))  
        db = (1/n_samples) * np.sum(y_pred-y)  
  
        self.weights = self.weights - self.lr * dw  
        self.bias = self.bias - self.lr * db
```

This process is the optimization step in linear regression using gradient descent. The detail of the process can be found in my transcript. In short, it initializes the weights and bias as zero and iteratively adjusts them to minimize the Mean Squared Error (MSE) between predicted (`y_pred`) and actual values (`y`). In each iteration, the model calculates predictions (`y_pred`) using the current weights and bias, computes the gradients (`dw` and `db`) of the loss function with respect to the weights and bias, and updates these parameters using the learning rate (`lr`). This iterative process continues for `n_iters` iterations, gradually refining the weights and bias to improve the model's fit.

## Predicting using the multiple linear regression.

```
def predict(self, X):
    y_pred = np.dot(X, self.weights) + self.bias
    return y_pred
```

The predict function generates predictions for new data points using the trained linear regression model. It calculates the predicted values ( $y_{\text{pred}}$ ) by taking the dot product of the input features ( $X$ ) with the learned weights ( $\text{self.weights}$ ) and adding the learned bias ( $\text{self.bias}$ ). The result is returned as the output, representing the model's predictions based on the input data.

## Linear Model Prediction (70/20/10 Split)

```
In [23]: # Step 1: Load data and filter numeric columns
df = pd.read_csv("PISA2018_data.csv")
numeric_df = df.select_dtypes(include=[np.number])

# Step 2: Drop rows with NaN values
numeric_df = numeric_df.dropna()

# Step 3: Define Y (target) and X (features)
y = numeric_df['STU_SCI_LITERACY'].values # Target variable
X = numeric_df[['HOUSE_POS']].values # Replace with the desired feature

# Step 4: Scale the feature
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Step 5: Allocate 70/20/10 split
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.1)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.1)

# Step 6: Train the linear regression model
reg = LinearRegression(lr=0.01)
reg.fit(X_train, y_train)

# Step 7: Predict and calculate MSE for the test set
def mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

predictions = reg.predict(X_test)
mse_value = mse(y_test, predictions)
print(f"Mean Squared Error (Test Set): {mse_value}")

# Step 8: Manual calculations for R^2, F-statistic, Residual Standard Error
# Calculate residuals
residuals = y_test - predictions

# R^2 (R-squared)
```

```

def r_squared(y_true, y_pred):
    ss_total = np.sum((y_true - np.mean(y_true)) ** 2)
    ss_residual = np.sum((y_true - y_pred) ** 2)
    return 1 - (ss_residual / ss_total)

r2 = r_squared(y_test, predictions)
print(f"R-squared (Test Set): {r2}")

# F-statistic
def f_statistic(y_true, y_pred, num_predictors):
    n = len(y_true)
    ss_total = np.sum((y_true - np.mean(y_true)) ** 2)
    ss_residual = np.sum((y_true - y_pred) ** 2)
    ss_regression = ss_total - ss_residual
    ms_regression = ss_regression / num_predictors
    ms_residual = ss_residual / (n - num_predictors - 1)
    return ms_regression / ms_residual

num_predictors = X_train.shape[1]
f_stat = f_statistic(y_test, predictions, num_predictors)
print(f"F-statistic: {f_stat}")

# Residual Standard Error (RSE)
def residual_standard_error(y_true, y_pred, num_predictors):
    n = len(y_true)
    ss_residual = np.sum((y_true - y_pred) ** 2)
    return np.sqrt(ss_residual / (n - num_predictors - 1))

rse = residual_standard_error(y_test, predictions, num_predictors)
print(f"Residual Standard Error: {rse}")

# Step 9: Using statsmodels for comparisons between manual calculation
# Add constant (intercept) to the features (Gradient descent does not
X_train_sm = sm.add_constant(X_train)
X_test_sm = sm.add_constant(X_test)

# Fit the model using statsmodels
model = sm.OLS(y_train, X_train_sm)
results = model.fit()

# Print statsmodels summary
print("\nStatsmodels Summary:")
print(results.summary())

# Step 10: Optional Visualization (if there is only 1 feature)
if X.shape[1] == 1: # Only if there is 1 feature
    y_pred_line = reg.predict(X)
    cmap = plt.get_cmap('viridis')
    fig = plt.figure(figsize=(8, 6))
    plt.scatter(X_train, y_train, color=cmap(0.9), s=10, label="Train")
    plt.scatter(X_test, y_test, color=cmap(0.5), s=10, label="Test Data")
    plt.plot(X, y_pred_line, color='black', linewidth=2, label='Predictions')

```

```
plt.legend()  
plt.show()
```

Mean Squared Error (Test Set): 3033.7961698797367  
 R-squared (Test Set): -0.01568066630424436  
 F-statistic: -0.38596447743023277  
 Residual Standard Error: 57.24071857926066

## Statsmodels Summary:

## OLS Regression Results

```

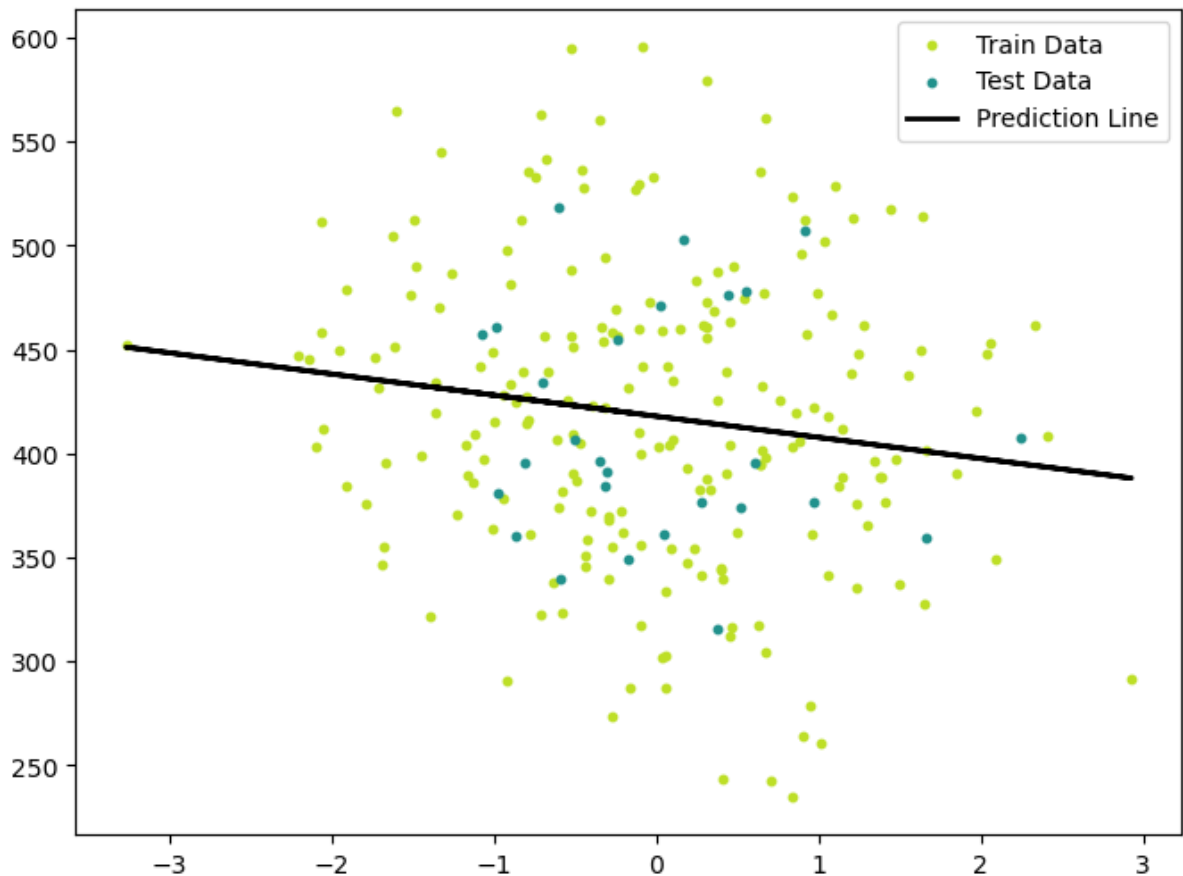
=====
=====
Dep. Variable:                y    R-squared:
0.022
Model:                        OLS    Adj. R-squared:
0.018
Method:                        Least Squares    F-statistic:
4.780
Date:                          Mon, 16 Dec 2024    Prob (F-statistic):
0.0299
Time:                          20:20:59    Log-Likelihood:
-1192.5
No. Observations:              210    AIC:
2389.
Df Residuals:                  208    BIC:
2396.
Df Model:                      1
Covariance Type:               nonrobust
=====
=====
                                coef    std err          t      P>|t|      [0.025
0.975]
-----
const                417.9383      4.912      85.087      0.000      408.255
427.622
x1                   -10.1550      4.645      -2.186      0.030     -19.312
-0.998
=====
=====
Omnibus:                0.329    Durbin-Watson:
1.758
Prob(Omnibus):          0.849    Jarque-Bera (JB):
0.479
Skew:                   0.023    Prob(JB):
0.787
Kurtosis:               2.771    Cond. No.
1.07
=====
=====

```

## Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.





## Prediction Model Interpretation and Notes

### Step 1: Load data and filter numeric columns

```
df = pd.read_csv("PISA2018_data.csv")  
numeric_df = df.select_dtypes(include=[np.number])
```

I removed all the categorical variables from the dataset because multiple linear regression only works with numeric variables. I named the dataset containing only numeric variables "numeric\_df" to distinguish it from the original data ("df") (Numpy, 2024; Pandas, 2024).

### Step 2: Drop rows with NaN values

```
numeric_df = numeric_df.dropna()
```

The model returned NaN for MAE value, even though the original dataset had no missing values. Therefore, I removed the rows containing any NaN values for unknown reasons. Please note that using this correction method to handle NaN values is not ideal and not recommended, and I wouldn't have used it if it weren't

part of a practice exercise for building a machine-learning model. I'd recommend properly handling missing values to anyone building their custom machine-learning models for research.

### Step 3: Define Y (Outcome) and X (Predictors)

```
y = numeric_df['STU_SCI_LITERACY'].values
X = numeric_df[['HOUSE_POS']].values # Replace with the
desired feature
```

This code sets up the variables X and y for your study. X can include several items, but it must exclude the y variable.

### Step 4: Scale the feature

```
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

This code standardizes variables to allow comparisons by bringing values with different units to the same scale. StandardScaler (scikit-learn, 2024a) is a function under sklearn.preprocessing (import StandardScaler).

### Step 5: Allocate 70/20/10 split

```
X_train, X_temp, y_train, y_temp = train_test_split(X,
y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp,
y_temp, test_size=0.3, random_state=42)
```

This code divides the dataset roughly into training 70% / validation 20% / testing 10%. These settings ensure we use new subsets of samples within the numeric\_df dataset to verify and test the linear regression model. (scikit-learn, 2024b)

### Step 6: Train the linear regression model

```
reg = LinearRegression(lr=0.01)
reg.fit(X_train, y_train)
```

This code calls for the Linear Regression model that I previously built based on the video instruction by AssemblyAI (2022). The learning rate (details are

available in my transcript) is set to 0.01. In short, a smaller value (e.g., 0.01) ensures slower but more stable learning, while a larger value could lead to faster convergence but lacks accuracy (Bhattbhatt, 2024).

### Step 7: Predict and calculate MSE for the test set

```
def mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

predictions = reg.predict(X_test)
mse_value = mse(y_test, predictions)
print(f"Mean Squared Error (Test Set): {mse_value}")
```

This code calculates the Mean Squared Error (MSE), which measures how far the model's predictions are from the actual values in the test set (the equation is available in my transcript). ( $y_{\text{true}}$ ) = the data point in the dataset and ( $y_{\text{pred}}$ ) = the model's predicted  $y$  value. "reg" is predetermined on step 6, and predicts is predetermined in the model building ( $\text{predict} = \text{return np.dot}(X, \text{self.weights}) + \text{self.bias}$ ) (sklearn, 2024).

### Step 8: Manual calculations for $R^2$ , F-statistic, Residual Standard Error

```
# Calculate residuals
residuals = y_test - predictions
```

This calculates the residuals, which are the differences between the actual values ( $y_{\text{test}}$ ) and the predicted values (predictions) from the model.

### $R^2$ (R-squared)

```
def r_squared(y_true, y_pred):
    ss_total = np.sum((y_true - np.mean(y_true)) ** 2)
    ss_residual = np.sum((y_true - y_pred) ** 2)
    return 1 - (ss_residual / ss_total)

r2 = r_squared(y_test, predictions)
print(f"R-squared (Test Set): {r2}")
```

The equation for calculating the R squared is available in my transcript. I used  $\text{np.mean}$  to calculate the mean values and  $\text{np.sum}$  to add the squared residual values.

## F-statistic

```
def f_statistic(y_true, y_pred, num_predictors):
    n = len(y_true)
    ss_total = np.sum((y_true - np.mean(y_true)) ** 2)
    ss_residual = np.sum((y_true - y_pred) ** 2)
    ss_regression = ss_total - ss_residual
    ms_regression = ss_regression / num_predictors
    ms_residual = ss_residual / (n - num_predictors - 1)
    return ms_regression / ms_residual

num_predictors = X_train.shape[1]
f_stat = f_statistic(y_test, predictions,
num_predictors)
print(f"F-statistic: {f_stat}")
```

The mathematical equation for F-statistic is available in my transcript.

## Residual Standard Error (RSE)

```
def residual_standard_error(y_true, y_pred,
num_predictors):
    n = len(y_true)
    ss_residual = np.sum((y_true - y_pred) ** 2)
    return np.sqrt(ss_residual / (n - num_predictors -
1))

rse = residual_standard_error(y_test, predictions,
num_predictors)
print(f"Residual Standard Error: {rse}")
```

The mathematical equations are given in my transcripts. The following metrics helped me determine if my model was functioning correctly. I also wanted to use the conventional Python module "statsmodels (Josef Perktold et al., 2024)" to compare with just to varify the outcome.

## Step 9: Using statsmodels for comparisons between manual calculation and the existing model (Josef Perktold et al., 2024)

Add constant (intercept) to the features (Gradient descent does not need this process)

```
X_train_sm = sm.add_constant(X_train)
X_test_sm = sm.add_constant(X_test)
```

### Fit the model using statsmodels

```
model = sm.OLS(y_train, X_train_sm)
results = model.fit()
```

### Print statsmodels summary

```
print("\nStatsmodels Summary:")
print(results.summary())
```

The module required the intercept values for training and testing (Josef Perktold et al., 2024). This block code is only required when the statsmodels module is used.

### Step 10: Optional Visualization (if there is only 1 feature)

```
if X.shape[1] == 1: # Only if there is 1 feature
    y_pred_line = reg.predict(X)
    cmap = plt.get_cmap('viridis')
    fig = plt.figure(figsize=(8, 6))
    plt.scatter(X_train, y_train, color=cmap(0.9), s=10,
label="Train Data")
    plt.scatter(X_test, y_test, color=cmap(0.5), s=10,
label="Test Data")
    plt.plot(X, y_pred_line, color='black', linewidth=2,
label='Prediction Line')
    plt.legend()
    plt.show()
```

Matplotlib is a simple and straightforward tool for creating visualizations, and I could easily spend hours learning about it. I picked the easiest sample based on online cheatsheet and handouts (Matplotlib, 2024). Visualizing data is so important in analysis that I would have spent more time on it if I had the chance.

## References

AssemblyAI (Director). (2022, September 12). Machine Learning From Scratch Full course [Video recording]. [https://www.youtube.com/watch?v=p1hGz0w\\_OCo](https://www.youtube.com/watch?v=p1hGz0w_OCo)

Bhattbhatt, V. (2024, February 16). Learning Rate and Its Strategies in Neural Network Training. The Deep Hub. <https://medium.com/thedeephub/learning-rate-and-its-strategies-in-neural-network-training-270a91ea0e5c>

Josef Perktold, Skipper Seabold, Kevin Sheppard, ChadFulton, Kerby Shedden, jbrockmendel, j-grana6, Peter Quackenbush, Vincent Arel-Bundock, Wes McKinney, Ian Langmore, Bart Baker, Ralf Gommers, yogabonito, s-scherrer, Yauhen Zhurko, Matthew Brett, Enrico Giampieri, yl565, ... Yaroslav Halchenko. (2024). statsmodels/statsmodels: Release 0.14.2 (Version v0.14.2) [Computer software]. Zenodo. <https://doi.org/10.5281/ZENODO.593847>

Matplotlib. (2024). Matplotlib cheatsheets—Visualization with Python. Matplotlib Cheatsheets and Handouts. <https://matplotlib.org/cheatsheets/>

Numpy. (2024). NumPy user guide—NumPy v2.1 Manual [NumPy user guide]. Numpy. <https://numpy.org/doc/stable/user/index.html#user>

Pandas. (2024). pandas.DataFrame.select\_dtypes—Pandas 2.2.3 documentation. API Reference > DataFramepandas > pandasDataFrame (Pandas.DataFrame.Select\_dtypes). [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.select\\_dtypes.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.select_dtypes.html)

scikit-learn. (2024a). StandardScaler. Scikit-Learn. <https://scikit-learn/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>

scikit-learn. (2024b). Train\_test\_split. Train\_test\_split. [https://scikit-learn/stable/modules/generated/sklearn.model\\_selection.train\\_test\\_split.html](https://scikit-learn/stable/modules/generated/sklearn.model_selection.train_test_split.html)

sklearn. (2024). Mean\_squared\_error. Mean\_squared\_error. [https://scikit-learn/stable/modules/generated/sklearn.metrics.mean\\_squared\\_error.html](https://scikit-learn/stable/modules/generated/sklearn.metrics.mean_squared_error.html)

## **Appendix A3 Logistic Regression**

# Logistic Regression From Scratch!

## Model Building

```
In [24]: import numpy as np
```

```
In [26]: class LogisticRegression:

    # Initializes the Logistic Regression model with learning rate and
    def __init__(self, lr=0.1, n_iters=5000):
        self.lr = lr
        self.n_iters = n_iters
        self.weights = None
        self.bias = None

    # Sigmoid function: Converts linear model output to probabilities
    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    # Fits the Logistic Regression model to the training data
    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        # Performs gradient descent for the specified number of iterations
        for _ in range(self.n_iters):
            linear_model = np.dot(X, self.weights) + self.bias
            y_pred = self.sigmoid(linear_model)

            # Computes gradients for weights and bias
            dw = (1 / n_samples) * np.dot(X.T, (y_pred - y))
            db = (1 / n_samples) * np.sum(y_pred - y)

            # Updates weights and bias using the gradients and learning rate
            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    # Predicts class labels (0 or 1) for input data
    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_pred = self.sigmoid(linear_model)
        return [1 if i > 0.5 else 0 for i in y_pred]
```

## Model Interpretation and Memo for



# Logistic Regression (AssemblyAI, 2022)

Initializes the Logistic Regression model with learning rate and number of iterations

```
def __init__(self, lr=0.01, n_iters=5000):  
    self.lr = lr  
    self.n_iters = n_iters  
    self.weights = None  
    self.bias = None
```

The **init** method initializes the logistic regression model. It sets the learning rate (lr), which controls the step size during optimization, and the number of iterations (n\_iters) for gradient descent. It also initializes weights and bias to None, which will later store the model parameters. The learning rate and iteration frequencies (n\_iters) can be changed based on your research requirement. The lower the learning rate and higher the iteration, the more precise your model will be. However, the training process may slow down. For a small sample dataset like in the present study did not change the outcome at all.

## sigmoid $[1 / (1 + e^{-\text{value}})]$ (AssemblyAI, 2022)

```
# Sigmoid function: Converts linear model output to  
probabilities between 0 and 1  
def sigmoid(self, z):  
    return 1 / (1 + np.exp(-z))
```

The sigmoid function is a mathematical function used in logistic regression to ensure the output falls between 0 and 1, representing a probability. It belongs to a group of functions known as logistic functions, which share this property (Brownlee, 2016). The sigmoid method applies the sigmoid activation function to a given input z. It maps any real number into a range between 0 and 1, making it useful for converting linear model outputs into probabilities (AssemblyAI, 2022).

## fitting (AssemblyAI, 2022)

```
# Fits the Logistic Regression model to the training  
data  
def fit(self, X, y):  
    n_samples, n_features = X.shape  
    self.weights = np.zeros(n_features)  
    self.bias = 0
```

Similarly to the Linear Regression model, this is where the weight and bias are initially set to zero for the gradient descent loop that starts at the next code block. This process is repeated until the optimal balance is maintained between the two. Detailed description of how the gradient descent process work can be found in my transcript under the Linear Regression model. Based the coefficient estimates here allows to move onto the prediction.

### predicting (AssemblyAI, 2022)

```
# Predicts class labels (0 or 1) for input data
def predict(self, X):
    linear_model = np.dot(X, self.weights) + self.bias
    y_pred = self.sigmoid(linear_model)
    return [1 if i > 0.5 else 0 for i in y_pred]
```

This code defines the prediction set up in the "LogisticRegression" class and tells the model to use the X variable to predict. np.dot() function provides the dot function, where it takes into account the increase or decrease of Y value based on the change in X and add the bias term. The concept of balance between the weights and bias is well explained by James et al, (2023). Highly recommended to read through.

## Logistic Model Prediction (70/20/10)

```
In [30]: import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classifi
```

```
In [39]: # Load data
df = pd.read_csv("PISA2018_data.csv")

# Use one predictor variable (e.g., 'WEALTH')
X = df['WEALTH'].values.reshape(-1, 1)
y = df['LANGUAGE'].values # Binary outcome variable ONLY!!

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.

# Train logistic regression model
clf = LogisticRegression(lr=0.001, n_iters=5000)
clf.fit(X_train, y_train)

# Make predictions
y_pred = clf.predict(X_test)
```

```
# Manual accuracy
def accuracy(y_pred, y_test):
    return np.sum(y_pred == y_test) / len(y_test)

manual_acc = accuracy(y_pred, y_test)
print(f"Manual Accuracy: {manual_acc:.4f}")

# Using sklearn metrics
print(f"Sklearn Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

Manual Accuracy: 0.6667

Sklearn Accuracy: 0.6667

## Prediction Model Interpretation and Note

Use one predictor variable (e.g., 'WEALTH')

```
df = pd.read_csv("PISA2018_data.csv")
X = df['WEALTH'].values.reshape(-1, 1)
y = df['LANGUAGE'].values # Binary outcome variable
ONLY!!
```

The dataset PISA2018\_data.csv was loaded into a Pandas DataFrame, with 'WEALTH' as the predictor feature and 'LANGUAGE' as the binary outcome. 'WEALTH' was reshaped into a 2D (rows and columns) array as required by scikit-learn for input variables, which works just like transposing in the MS Excel. This solution was facilitated by the ChatGPT-4o model. I modified the original code by AssemblyAI (2022) to utilize scikit-learn metrics for accuracy comparison against a manual calculation, ensuring the correct equation.

### Train-Test Split

```
X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size=0.2, random_state=42)
```

Just like in the linear regression model, the data was split into training and test sets using an 80-20 ratio. X\_train and y\_train are used to train the model, while X\_test and y\_test are reserved for evaluating the model's performance. A fixed random\_state = 42 works as though setting a seed in R, ensuring reproducibility in randomness.

### Train Logistic Regression Model (Scikit-learn, 2022)

```
clf = LogisticRegression(lr=0.001, n_iters=5000)
clf.fit(X_tran, y_train)
```

A custom logistic regression model ( `LogisticRegression` ) was initialized with a learning rate of 0.001 and set to run for 5000 iterations. Trained on the dataset (X\_train and y\_train), the model learns the relationship between WEALTH and the binary outcome LANGUAGE. In contrast to a previous Random Forest model that failed due to a lack of my machine specifications, this smaller dataset indicates that variable adjustments do not significantly affect accuracy or processing speed. For larger datasets, modifying these parameters can optimize accuracy and processing speed according to your research needs.

## Make Predictions

```
y_pred = clf.predict(X_test)
```

The trained logistic regression model predicts the outcomes for the test data (X\_test). These predictions (y\_pred) are later used to calculate accuracy.

## Manual Accuracy

```
def accuracy(y_pred, y_test):
    return np.sum(y_pred == y_test) / len(y_test)

manual_acc = accuracy(y_pred, y_test)
print(f"Manual Accuracy: {manual_acc:.4f}")
```

A manual accuracy function is defined to compare predictions (y\_pred) with actual test outcomes (y\_test). I differentiated the outcome between the manual and the pre-existing function for an easy comparison. The accuracy is calculated as the proportion of correctly predicted values over the total number of observations (AssemblyAI, 2022).

## Using Sklearn Metrics

```
print(f"Scikit-learn Accuracy: {accuracy_score(y_test,
y_pred):.4f}")
```

The `accuracy_score` function from `scikit-learn` metrics is used to validate the manual accuracy calculation. Both results are printed and compared to ensure

consistency.

## References:

AssemblyAI (Director). (2022). (2) How to implement Logistic Regression from scratch with Python—YouTube [YouTube]. [https://www.youtube.com/watch?v=YYEJ\\_GUguHw](https://www.youtube.com/watch?v=YYEJ_GUguHw)

Scikit-learn. (2024). 1.1. Linear Models. Scikit-Learn (User Guide). [https://scikit-learn/stable/modules/linear\\_model.html](https://scikit-learn/stable/modules/linear_model.html)

In [ ]:

In [ ]: