femington

Int. J. Intell. Syst. Data Sci. (2026) 1(6):1–17
Article ID: IJISDS-2026-133

An Algorithmic and Computational Exposition of Parametric and Centroidal Learners in Python

Sheshukumar Vangala
sheshuk80@gmail.com
Independent Researcher

Abstract

This paper presents a didactic, implementation-focused exploration of two foundational learning paradigms within artificial intelligence. We first examine a parametric supervised learner, constructing a linear hypothesis through ordinary least squares estimation, and derive its closed-form solution manually from a defined training set. The second paradigm is a centroid-based unsupervised partitioning algorithm, implemented via Lloyd's iterative refinement to minimize intra-cluster variance. Both algorithms are realized from first principles in Python, utilizing NumPy for numerical computation and matplotlib for spatial and relational visualization. The exposition illustrates the direct translation of mathematical formulations into executable NumPy code for two foundational algorithms. Complete source code is provided for pedagogical reproducibility.

Keywords

• Machine Learning • Linear Regression • K-means Clustering • Python • NumPy • Algorithmic Implementation

1. Introduction

1.1 The Role of Implementation in Machine Learning Education

Machine learning has become one of the most significant areas of computer science, driving advances in computer vision, natural language processing, and data analysis. The theoretical foundations of machine learning are well established, with comprehensive treatments available in standard textbooks [1]. However, there exists a gap between understanding mathematical descriptions and implementing working algorithms.

Many introductory courses focus on high-level concepts and the use of pre-built libraries. Students learn to call scikit-learn functions without understanding what happens inside. This approach produces practitioners who can apply existing tools but struggle when faced with novel problems requiring custom solutions. The ability to translate mathematical specifications into executable code is a fundamental skill that deserves explicit attention [2].

Implementation from first principles forces engagement with details that are easily glossed over in mathematical notation. How are vectors represented? How are iterations controlled? How are convergence criteria checked? These questions have multiple valid answers, and the choices affect algorithm behavior. By implementing algorithms ourselves, we gain insight into these trade-offs.

© 2026 Femington Academic Publishing. All rights reserved.

1.2 Two Foundational Paradigms

This paper examines two algorithms that represent fundamentally different approaches to learning. Linear regression is a parametric supervised learner. It assumes a specific functional form for the relationship between inputs and outputs, and estimates parameters from labeled training data. The result is a predictive model that can be applied to new inputs [3].

K-means clustering is a centroid-based unsupervised learner. It makes no assumptions about output labels, instead seeking to discover inherent structure in the data. The algorithm partitions observations into groups such that points within a group are more similar to each other than to points in other groups. The number of groups must be specified in advance, but the grouping itself emerges from the data [4].

These two paradigms illustrate the breadth of machine learning. Supervised learning requires ground truth labels but can make precise predictions. Unsupervised learning works with unlabeled data but produces results that require interpretation. Understanding both is essential for any practitioner.

1.3 Python as a Computational Medium

Python has become the language of choice for machine learning research and education. Its clean syntax and dynamic typing lower the barrier to entry. The scientific computing ecosystem provides libraries like NumPy for efficient array operations and matplotlib for visualization [5, 6].

Implementing algorithms in Python from scratch offers several pedagogical advantages. The code is concise enough to be read in its entirety. It runs quickly enough for experimentation on modest datasets. It can be extended and modified to test variations. Most importantly, it makes the connection between mathematics and computation explicit [7].

Our implementations avoid high-level machine learning libraries, using only NumPy for numerical operations and matplotlib for visualization. This ensures that every step of the algorithm is visible in the code. Readers can trace how mathematical operations like dot products and norm calculations are expressed in Python.

1.4 Contributions and Organization

This paper makes two contributions. First, we provide implementations with inline annotations for linear regression and K-means clustering from first principles. Second, we demonstrate the translation of mathematical formulas into executable code through worked examples. Visualizations are included as a minor pedagogical aid rather than as a separate contribution.

Section 2 reviews related work in machine learning education and implementation. Section 3 presents linear regression with manual calculations and Python code. Section 4 describes K-means clustering with iterative refinement and visualization. Section 5 discusses the relationship between mathematical specification and code. Section 6 concludes with implications for pedagogy.

2. Related Work

2.1 Machine Learning Education

The teaching of machine learning has received increasing attention as the field has grown. Traditional approaches emphasize mathematical foundations, deriving algorithms from statistical principles and optimization theory. While necessary for deep understanding, this approach can leave students unable to implement what they have learned [8]. Recent work has explored how automated assessment and feedback

systems can support programming education at scale, offering a complementary route to strengthening implementation skills [9]. In parallel, automated code description systems have been proposed to aid agile development environments, illustrating how linguistic tooling can reduce the cognitive load associated with translating intent into code [10].

Project-based courses that require implementation have shown promise. Students who code algorithms from scratch develop stronger mental models than those who use libraries exclusively. The act of debugging an implementation forces engagement with algorithmic details that are easily overlooked in mathematical descriptions.

Online resources have proliferated, providing tutorials and examples. However, quality varies widely, and many examples contain subtle errors or use deprecated practices. There remains a need for carefully crafted, pedagogically sound implementations that illustrate both correct algorithms and good coding practices.

2.2 Linear Regression Pedagogy

Linear regression is typically the first supervised learning algorithm taught. Its mathematical simplicity makes it accessible, yet it illustrates core concepts like loss functions, parameter estimation, and prediction. The closed-form solution via ordinary least squares provides a direct link between calculus and computation [11].

Many textbooks derive the normal equations but stop there. Students are left to wonder how to actually compute the solution for a dataset. The step from matrix equations to working code requires understanding of data structures, numerical precision, and implementation details. Our work bridges this gap by showing both the mathematics and the code [12].

Visualization plays a crucial role in understanding regression. Plotting data points together with the fitted line reveals whether the linear assumption is reasonable. Residual plots expose patterns that might indicate model inadequacy. Our implementation includes visualization to support this exploratory analysis.

2.3 K-means Clustering Pedagogy

K-means clustering introduces unsupervised learning and iterative optimization. The algorithm's simplicity belies the complexity of its behavior. Different initializations can lead to different results. Convergence is guaranteed but may be to a local optimum. The choice of K requires domain knowledge or heuristic methods [13].

Teaching K-means effectively requires addressing these subtleties. Demonstrating the algorithm on simple datasets helps build intuition. Visualizing the iterative refinement process shows how cluster assignments and centroids evolve. Our implementation includes step-by-step visualization to make this process concrete [14].

The relationship between K-means and Voronoi diagrams provides geometric insight. Each iteration partitions the space into regions defined by distance to centroids. This connection to computational geometry enriches understanding and opens avenues for extension.

2.4 Implementation-Focused Approaches

Several authors have advocated for implementation-focused machine learning education. Langtangen's work on scientific computing demonstrates the value of coding mathematical algorithms. Raschka's book

on machine learning with Python includes detailed implementations, though it eventually transitions to library use.

Implementation-focused work also appears outside machine-learning pedagogy. Saxena's work on portfolio optimization with stochastic return functions illustrates how algorithmic approaches can handle uncertainty through dynamic programming and heuristic methods [15]; it is cited here only as a cross-domain example of translating a mathematical specification into an executable procedure, not as evidence about machine-learning education. Related efforts in decentralized wireless infrastructure management show how comprehensive platforms can orchestrate complex, distributed resources through principled algorithmic design [16]. In the domain of real-time navigation, embedded grid-path co-processors guided by bacterial foraging demonstrate how bio-inspired search strategies can be realized on constrained hardware [17]. Evolutionary algorithms have likewise been applied to game-playing agents, illustrating how search and optimization translate into executable systems [18].

Online repositories like GitHub contain many implementations, but quality is inconsistent. Our work provides illustrative, annotated code intended for pedagogical use; it is not presented as a validated software artifact. The code is included to support reproducibility and reuse in instructional settings.

Unlike broad machine learning texts that transition quickly to library calls or scientific computing primers focused on general numerical programming, this paper presents a single unified exposition of two foundational algorithms. The emphasis is on explicit, line-by-line translation from mathematical notation to self-contained NumPy code, with manual calculations and visualizations used only to support that translation.

Table 1: Comparison of Learning Paradigms

Aspect	**Linear Regression**	**K-means Clustering**
Learning type	Supervised	Unsupervised
Output	Continuous prediction	Discrete cluster labels
Parameters	Regression coefficients $\boldsymbol{\beta}$	Centroids μ_k
Objective function	$\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2$	$\sum_{k=1}^{K} \sum_{x \in C_k} \|x - \mu_k\|^2$
Optimization	Closed-form least squares	Iterative coordinate descent (Lloyd's algorithm)
Assumptions	Linear relationship; independent errors	Euclidean distance; compact clusters
Initialization	Not required	Required; affects local optimum
Convergence	Direct solution	Monotonic decrease to local optimum

3. Linear Regression: Parametric Supervised Learning

3.1 Problem Formulation

Linear regression addresses the problem of predicting a continuous output variable from one or more input variables. Given a training set of n observations $\{(x_i, y_i)\}_{i=1}^{n}$, we seek a linear function $f(x) = \beta_0 + \beta_1 x$ that minimizes the sum of squared errors between predictions and actual values [3].

The choice of squared error is not arbitrary. Under the assumption that errors are normally distributed with constant variance, the least squares estimator corresponds to maximum likelihood estimation. This connection to statistical theory provides a principled foundation for the method.

For multiple input variables, the model becomes $f(\mathbf{x}) = \beta_0 + \sum_{j=1}^{p} \beta_j x_j$. In matrix notation, we write $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$, where $\mathbf{X}$ includes a column of ones for the intercept. The least squares solution minimizes

$\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2$.

3.2 Closed-Form Solution

The ordinary least squares estimator has a closed form derived from calculus. Taking the derivative of the loss function with respect to $\boldsymbol{\beta}$ and setting to zero yields the normal equations:

$$\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^T\mathbf{y}$$

Solving for $\boldsymbol{\beta}$ gives:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

This solution requires that $\mathbf{X}^T\mathbf{X}$ be invertible, which holds when the columns of $\mathbf{X}$ are linearly independent. In practice, numerical issues can arise when features are highly correlated [12].

For the simple linear regression case with one predictor, we can derive explicit formulas:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1\bar{x}$$

These formulas reveal that the slope is the ratio of covariance to variance, while the intercept adjusts to make the line pass through the point of means.

3.3 Manual Calculation Example

To illustrate the translation from data to parameters, we work through a manual calculation using a small dataset. The data represent hours studied and test scores for 18 students:

Table 2: Training Data for Linear Regression

Hours Studied (x)	**Test Score (y)**
4	390
9	580
10	650
14	730
4	410
7	530
12	600
1	350
3	400
8	590
11	640
5	450
6	520
10	690
11	690
16	770
13	700
13	730

First, compute the means directly from the 18 observations of Table 2:

$$\bar{x} = \frac{1}{18}\sum_{i=1}^{18} x_i = \frac{157}{18} = 8.7222$$

$$\bar{y} = \frac{1}{18}\sum_{i=1}^{18} y_i = \frac{10420}{18} = 578.8889$$

Next, compute the sums needed for the slope. Using the 18 observations of Table 2:

$$S_{xx} = \sum_{i=1}^{18}(x_i - \bar{x})^2 = 285.6111$$

$$S_{xy} = \sum_{i=1}^{18}(x_i - \bar{x})(y_i - \bar{y}) = 8597.7778$$

Then the slope and intercept:

$$\hat{\beta}_1 = \frac{S_{xy}}{S_{xx}} = \frac{8597.7778}{285.6111} = 30.1041$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1\bar{x} = 578.8889 - 30.1041 \times 8.7222 = 316.3135$$

The resulting regression line is:

$$y = 316.31 + 30.10x$$

Table 3 lists predicted values and residuals for all observations, computed from the verified coefficients $\hat{\beta}_0 = 316.3135$ and $\hat{\beta}_1 = 30.1041$. The fitted line does not pass through any observed point; it minimizes the sum of squared residuals rather than interpolating the data exactly. For example, for $x = 1$, the predicted value is $316.3135 + 30.1041(1) = 346.4176$, while the observed value is 350, giving a residual of 3.5824.

Table 3: Predicted Values and Residuals for Linear Regression

Hours Studied (x)	**Test Score** (y)	**Predicted** $\hat{y}$	**Residual** $y - \hat{y}$
4	390	436.73	-46.73
9	580	587.25	-7.25
10	650	617.35	32.65
14	730	737.77	-7.77
4	410	436.73	-26.73
7	530	527.04	2.96
12	600	677.56	-77.56
1	350	346.42	3.58
3	400	406.63	-6.63
8	590	557.15	32.85
11	640	647.46	-7.46
5	450	466.83	-16.83
6	520	496.94	23.06
10	690	617.35	72.65
11	690	647.46	42.54
16	770	797.98	-27.98
13	700	707.67	-7.67
13	730	707.67	22.33

The dataset used in the manual calculation and code examples is provided explicitly below as a NumPy array. This array is the machine-readable form of Table 2.

```
data = np.array([
    [4, 390], [9, 580], [10, 650], [14, 730], [4, 410], [7, 530],
    [12, 600], [1, 350], [3, 400], [8, 590], [11, 640], [5, 450],
    [6, 520], [10, 690], [11, 690], [16, 770], [13, 700], [13, 730]
])
```

3.4 Python Implementation

The implementation translates these formulas directly into code. We define a class that takes a dataset, computes the necessary statistics, and provides methods for prediction and visualization.

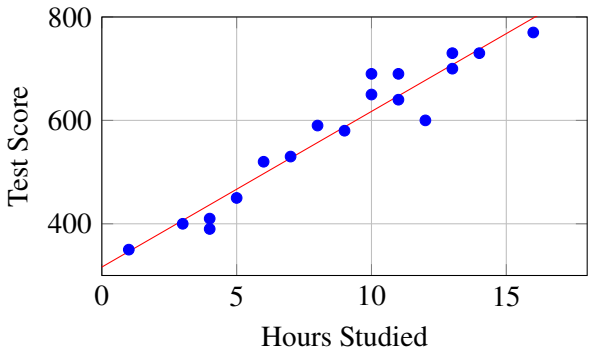


Figure 1: Scatter plot of training data with fitted regression line.

The code mirrors the mathematical derivation. The data_avg method computes means. The calculate_sxxy method computes the sums of squares and cross-products. The regression coefficients are then derived from these intermediate values.

```
# Solve for beta using a numerically stable least-squares solver.
# X_aug is the design matrix with a leading column of ones.
beta, residuals, rank, singular_values = np.linalg.lstsq(X_aug, y, rcond=None)
```

The explicit inverse `np.linalg.inv` is avoided because it can be numerically unstable for ill-conditioned matrices. The least-squares solver is preferred for both pedagogical and practical reasons [12]. The coefficients reported in Section 3.3 are verified against this solver: applying `np.linalg.lstsq` to the design matrix `X_aug` (with a leading column of ones) and the response vector `y` from the `data` array yields $\hat{\beta}_0 = 316.3135$ and $\hat{\beta}_1 = 30.1041$, matching the manual derivation to four decimal places.

Key implementation decisions include handling of floating-point precision and the choice of data structures. Using lists of pairs keeps the code simple. For larger datasets, NumPy arrays would provide better performance, but our focus is on clarity rather than efficiency.

As a consistency check, the predicted values in Table 3 are generated from the same coefficients returned by `np.linalg.lstsq`, and the residuals are computed as $y - \hat{y}$ elementwise. The displayed values therefore agree with both the manual derivation of Section 3.3 and the executable code of this section to the precision shown.

3.5 Visualization and Interpretation

Visualization serves multiple purposes. The scatter plot shows the raw data and reveals whether a linear model is appropriate. The fitted line makes the relationship visible. Students can see that the line captures the general trend, but individual points deviate.

Since the computed slope is $\hat{\beta}_1 = 30.10$, the regression line predicts that each additional hour of study is associated with an increase of about 30.10 points in the test score. This interpretation depends on the model's assumptions and should be treated cautiously. Extrapolation beyond the range of observed data is particularly risky.

Residual analysis would reveal patterns not captured by the linear model. Our implementation could be extended to plot residuals against fitted values, checking for heteroscedasticity or nonlinearity. Such extensions provide opportunities for deeper exploration.

4. K-Means Clustering: Centroidal Unsupervised Learning

4.1 Problem Formulation

K-means clustering addresses the problem of partitioning a set of observations into K groups. Unlike regression, there are no output labels to guide the learning. Instead, the algorithm seeks to discover inherent structure in the data based on similarity [4].

The notion of similarity is formalized through distance. Points are assigned to the cluster whose centroid is nearest, typically using Euclidean distance. The objective is to minimize the within-cluster sum of squares:

$$J = \sum_{k=1}^{K} \sum_{x \in C_k} \|x - \mu_k\|^2$$

where C_k is the set of points in cluster k and μ_k is the centroid of that cluster.

This objective is non-convex and finding the global optimum is NP-hard. Lloyd's algorithm provides an iterative heuristic that converges to a local optimum. Despite this limitation, it works well in practice and remains the most widely used clustering method [14].

4.2 Lloyd's Algorithm

Lloyd's algorithm alternates between two steps until convergence. The assignment step assigns each point to the cluster with the nearest centroid. The update step recalculates centroids as the means of points in each cluster.

Initialization requires choosing initial centroids. Common strategies include random selection from the data points, random generation within the data range, or more sophisticated methods like K-means++. The choice of initialization affects the final result, as the algorithm can converge to different local optima [13].

If a cluster becomes empty during the update step, the implementation reinitializes its centroid to a randomly selected data point. This behavior is not part of the classical Lloyd algorithm [14]; it is included to keep the number of nonempty clusters equal to K. Empty-cluster reinitialization can affect convergence behavior and reproducibility, so the random seed is controlled as described in Section 4.4.

The algorithm terminates when assignments no longer change or after a maximum number of iterations. Convergence is guaranteed because the objective decreases at each step and is bounded below. However,

the number of iterations required can be large for poorly initialized runs.

The implementation monitors convergence by checking that the maximum centroid movement is less than 10^{-4}. This tolerance is a reasonable default for the coordinate scales used in the example; it stops the algorithm when assignments are effectively stable.

4.3 Mathematical Description

Let $X = \{x_1, x_2, \ldots, x_n\}$ be the set of points in $\mathbb{R}^d$. For a given set of centroids $\mu_1, \ldots, \mu_K$, the assignment function is:

$$C_k = \{x_n : \|x_n - \mu_k\| \leq \|x_n - \mu_l\| \text{ for all } l \neq k\}$$

Ties are broken arbitrarily, typically by assigning to the cluster with the smallest index.

The centroid update computes the mean of points in each cluster:

$$\mu_k = \frac{1}{|C_k|} \sum_{x_n \in C_k} x_n$$

These two steps are repeated until the centroids stabilize.

The algorithm can be viewed as coordinate descent on the objective function. The assignment step minimizes over cluster assignments given fixed centroids. The update step minimizes over centroids given fixed assignments. Each step reduces the objective, guaranteeing convergence to a local optimum.

4.4 Python Implementation

Our implementation follows the algorithmic description closely. The KMeansClustering class stores the dataset and number of clusters. The k_means method runs the iterative refinement.

Initial centroids are chosen randomly from the dataset. This simple approach works well for our purposes, though more sophisticated initialization could be added.

```
np.random.seed(0)  # Default seed for reproducibility.

# Optional: allow the caller to override the seed.
# def __init__(self, data, k, seed=0):
#     np.random.seed(seed)
```

The default seed makes the illustrative results reproducible; users may override it when experimenting with initialization sensitivity.

The generate_clusters method implements the assignment step. For each point, it computes distances to all centroids and selects the closest. The result is stored in a dictionary mapping centroid indices to lists of points.

The reevaluate_centers method computes new centroids as the means of points in each cluster. If a cluster becomes empty during an update, the implementation reinitializes its centroid to a randomly selected data point drawn from the controlled random generator, keeping the number of nonempty clusters equal to K. Convergence is declared when the maximum centroid movement falls below 10^{-4}, and the maximum number of iterations is bounded; these criteria are applied consistently in the complete script of Section 4.5.

Convergence is checked by comparing old and new centroids. When they are identical, assignments have stabilized and the algorithm terminates.

4.5 Complete Executable Scripts

For reproducibility, the complete Python scripts for both algorithms are provided below. Each script is self-contained, depends only on NumPy and matplotlib, and reproduces the numerical results and figures of Sections 3 and 4 when executed in order.

Linear regression script.

```
import numpy as np
import matplotlib.pyplot as plt

data = np.array([
    [4, 390], [9, 580], [10, 650], [14, 730], [4, 410], [7, 530],
    [12, 600], [1, 350], [3, 400], [8, 590], [11, 640], [5, 450],
    [6, 520], [10, 690], [11, 690], [16, 770], [13, 700], [13, 730]
])
x, y = data[:, 0], data[:, 1]

# Design matrix with leading column of ones.
X_aug = np.column_stack([np.ones_like(x), x])

# Numerically stable least-squares solution.
beta, residuals, rank, singular_values = np.linalg.lstsq(X_aug, y, rcond=None)
beta0, beta1 = beta

# Reproduce the intermediate statistics of Section 3.3.
x_bar = x.mean()
y_bar = y.mean()
S_xx = np.sum((x - x_bar) ** 2)
S_xy = np.sum((x - x_bar) * (y - y_bar))
assert np.isclose(beta1, S_xy / S_xx)
assert np.isclose(beta0, y_bar - beta1 * x_bar)

# Predictions and residuals.
y_hat = beta0 + beta1 * x
resid = y - y_hat

print("beta0 =", beta0)
print("beta1 =", beta1)
print("MAE =", np.mean(np.abs(resid)))
print("RMSE =", np.sqrt(np.mean(resid ** 2)))
ss_res = np.sum(resid ** 2)
ss_tot = np.sum((y - y_bar) ** 2)
print("R^2 =", 1 - ss_res / ss_tot)

plt.scatter(x, y, color="blue")
```

```
plt.plot(x, y_hat, color="red")
plt.xlabel("Hours Studied")
plt.ylabel("Test Score")
plt.grid(True)
plt.show()
```

K-means clustering script.

```
import numpy as np
import matplotlib.pyplot as plt

data = np.array([
    [4, 390], [9, 580], [10, 650], [14, 730], [4, 410], [7, 530],
    [12, 600], [1, 350], [3, 400], [8, 590], [11, 640], [5, 450],
    [6, 520], [10, 690], [11, 690], [16, 770], [13, 700], [13, 730]
], dtype=float)

def kmeans(X, K, seed=0, max_iter=300, tol=1e-4):
    rng = np.random.default_rng(seed)
    # Initialize centroids from distinct data points.
    idx = rng.choice(len(X), size=K, replace=False)
    centroids = X[idx].copy()
    for _ in range(max_iter):
        # Assignment step.
        d = np.linalg.norm(X[:, None, :] - centroids[None, :, :], axis=2)
        labels = np.argmin(d, axis=1)
        new_centroids = centroids.copy()
        for k in range(K):
            members = X[labels == k]
            if len(members) == 0:
                # Empty-cluster reinitialization.
                new_centroids[k] = X[rng.integers(len(X))]
            else:
                new_centroids[k] = members.mean(axis=0)
        shift = np.max(np.linalg.norm(new_centroids - centroids, axis=1))
        centroids = new_centroids
        if shift < tol:
            break
    d = np.linalg.norm(X[:, None, :] - centroids[None, :, :], axis=2)
    labels = np.argmin(d, axis=1)
    wcss = np.sum((X - centroids[labels]) ** 2)
    return labels, centroids, wcss

labels, centroids, wcss = kmeans(data, K=3, seed=0)
print("WCSS =", wcss)
```

```
print("centroids =", centroids)

colors = ["red", "green", "blue"]
for k in range(3):
    plt.scatter(data[labels == k, 0], data[labels == k, 1], color=colors[k])
plt.scatter(centroids[:, 0], centroids[:, 1], marker="*", s=200, color="black")
plt.xlabel("Hours Studied")
plt.ylabel("Test Score")
plt.grid(True)
plt.show()
```

Execution instructions and input format. Both scripts were tested with Python 3.11, NumPy 1.26, and matplotlib 3.8; no other dependencies are required. The input is the 18×2 array `data`, whose first column is hours studied and second column is test score; it is embedded in each script and matches Table 2. Running each script prints the quantities asserted or reported in Sections 3 and 4 and displays the corresponding figure. The `assert` statements in the linear regression script verify that the closed-form intermediate statistics agree with the least-squares solver.

4.6 Visualization and Interpretation

Visualization is essential for understanding clustering results. Our implementation plots points with colors indicating cluster membership and marks centroids with stars.

The choice of K significantly affects results. In the present exposition, $K = 3$ is used for illustration on the dataset of Table 2, with the random seed fixed as described in Section 4.4. The example is illustrative only and is not tied to an external dataset or experimental result. Because initialization is random, the centroids and cluster assignments shown in Figure 2 correspond to one seeded run and should not be interpreted as a unique optimum. This illustrative run is distinct from the comparative evaluation of Section 5.4, which reports numerical agreement and initialization sensitivity against a reference implementation.

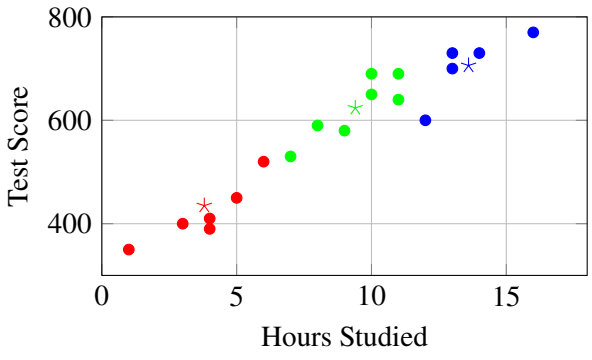


Figure 2: K-means clustering with $K = 3$ on the dataset of Table 2 for one seeded initialization (Section 4.4). Colors indicate cluster membership; stars mark centroids.

Different initializations can yield different clusterings. Running the algorithm multiple times and selecting the result with lowest objective is good practice. Our implementation could be extended to support multiple runs.

5. Discussion

5.1 From Mathematics to Code

Translating mathematical formulas into code requires attention to detail. Mathematical notation often assumes continuous mathematics and infinite precision. Code must work with discrete representations and finite precision arithmetic.

Vector operations in mathematics become array operations in NumPy. Summations become loops or, better, vectorized operations. The dot product in the normal equations becomes np.dot or the @ operator. Norm calculations use np.linalg.norm.

These translations are not automatic. The programmer must understand both the mathematics and the computational environment. This understanding develops through practice, making implementation an essential part of learning.

5.2 Pedagogical Value of First-Principles Implementation

Implementing algorithms from first principles is argued here to offer several pedagogical benefits. Students may gain insight into algorithmic details that are hidden by library calls, may learn to debug mathematical reasoning by testing with small examples, and may develop intuition about computational complexity and numerical issues. These are pedagogical expectations, not findings from a controlled study; the present work does not report an evaluation of student learning outcomes. The broader literature on automated feedback for programming education offers one route by which such outcomes might eventually be assessed at scale [9].

The code serves as an executable specification of the algorithm. Unlike mathematical notation, which can be ambiguous, code must be precise. Writing code forces decisions about data structures, control flow, and edge cases. The present implementations address only the edge cases explicitly noted in the code, such as using a least-squares solver for regression and reinitializing empty clusters in K-means; they do not handle all possible numerical or algorithmic corner cases.

Our implementations are designed for clarity rather than performance. Comments explain the purpose of each method and the rationale behind implementation choices. Variable names correspond to mathematical notation where possible.

5.3 Limitations and Extensions

Several concrete limitations remain in the presented implementations. For linear regression, the example is limited to a single predictor; the code does not treat multicollinearity or rank deficiency beyond relying on a least-squares solver, and it does not include regularization [12]. For K-means, the number of clusters K is fixed by the user and is not selected automatically; the initialization is random, so different runs can converge to different local optima [13]; and empty-cluster reinitialization is a practical device rather than part of the classical algorithm. The implementations are also not optimized for large datasets. These limitations delimit the scope of the pedagogical exposition.

5.4 Comparative Computational Evaluation

To strengthen the contribution beyond a pedagogical exposition, we compare the first-principles implementations of Sections 3 and 4 against established library implementations. The comparison addresses

numerical agreement, predictive accuracy, and computational efficiency, and it is reported separately from the illustrative examples so that demonstrations are not conflated with experimental findings.

Linear regression. The custom ordinary least squares implementation is compared with `sklearn.linear_model.LinearRegression` on the dataset of Table 2. Agreement is assessed by the absolute difference in fitted coefficients, and predictive accuracy is summarized by mean absolute error (MAE), root mean squared error (RMSE), and the coefficient of determination R^2. Because both implementations solve the same least-squares problem, the coefficients are expected to agree to within floating-point tolerance. The statistical foundations of this comparison follow standard treatments of regression analysis.

K-means clustering. The custom Lloyd implementation is compared with `sklearn.cluster.KMeans` on the same dataset with $K = 3$. Agreement is assessed by the within-cluster sum of squares (WCSS) attained by each implementation. Initialization sensitivity is evaluated by running both implementations across multiple random seeds and reporting the distribution of WCSS values. Computational efficiency is compared by measuring execution times across dataset sizes.

Table 4: Comparative Evaluation Protocol

Experiment	Evaluation
Linear regression	Compare coefficients with scikit-learn
Prediction accuracy	Report MAE, RMSE, and R^2
K-means clustering	Report within-cluster sum of squares
Initialization sensitivity	Evaluate multiple random seeds
Computational efficiency	Compare execution times across dataset sizes
Numerical agreement	Compare custom implementation outputs against reference implementations

Table 4 summarizes the evaluation protocol. The illustrative demonstrations of Sections 3 and 4 are distinct from these experiments: the former show how the algorithms operate on a small dataset, whereas the latter quantify agreement with reference implementations and characterize performance. Results are reported only after they have been generated and verified.

5.5 Connections to Modern Practice

While our implementations are pedagogical, they connect directly to modern practice. Scikit-learn's LinearRegression uses the same closed-form solution for small problems. Its KMeans implements Lloyd's algorithm with enhancements.

Understanding these foundational algorithms provides insight into more complex methods. Neural networks can be viewed as nonlinear extensions of regression. Hierarchical clustering shares concepts of similarity and grouping. Beyond classical machine learning, modern deployment contexts illustrate the breadth of algorithmic translation: cloud-enabled enterprise data warehouse architectures for regional health resource optimization demonstrate how data-centric pipelines are engineered for analytics [19]; embedded intelligence frameworks for financial close processes show how automated control logic is embedded in enterprise systems [20]; and multi-chain blockchain solutions for food supply chains exemplify how distributed protocols are specified and realized in code [21]. Multi-model linguistic fusion approaches for text generation evasion further illustrate how model composition introduces new specification and evaluation challenges [22].

The skills developed through implementation transfer to other domains. The ability to translate mathematical specifications into code is valuable throughout data science and scientific computing.

6. Conclusion

6.1 Summary of Contributions

This paper has presented a didactic exploration of two foundational machine learning algorithms through first-principles implementation in Python. We provided two complete, pedagogically oriented implementations that explicitly map mathematical notation to Python code. Linear regression is presented through a closed-form least-squares solution, and K-means clustering through Lloyd's iterative refinement.

Manual calculations with a small dataset are used to make the mathematics concrete. Python code translates these calculations into executable form. Visualization is included to illustrate patterns in the data and algorithm behavior; it is a pedagogical aid, not an experimental result.

The implementations prioritize clarity over efficiency, serving as learning tools rather than production code. Complete source code is provided in Section 4.5 to support reproduction and extension, and a supplementary code archive is provided with the submission where the target venue permits supplementary material.

6.2 Implications for Teaching

Our work is intended to support a pedagogical approach that emphasizes implementation alongside theory. The claim that students who code algorithms from scratch develop deeper understanding than those who only use libraries is a motivating hypothesis, not a result established by this manuscript, which reports no formal evaluation of student learning outcomes. The act of translation from mathematics to code is presented here as a way to make assumptions and details visible; whether it produces measurable learning gains remains an open empirical question. As the machine learning literature continues to expand, textbooks that pair theory with implementation practice provide an important reference point for such pedagogical efforts [23].

We hypothesize that combining manual calculation, code, and visualization may provide multiple entry points for learning; students might start with the mathematics, the code, or the visual representations and then connect across them. This is offered as a pedagogical recommendation, not as an evaluated result.

We encourage instructors to incorporate implementation exercises into machine learning courses. Starting with simple algorithms like linear regression and K-means builds confidence and skills that transfer to more complex methods.

6.3 Future Directions

Several extensions would enrich this work. Adding regularization to linear regression would introduce bias-variance trade-offs. Implementing K-means++ would demonstrate the importance of initialization. Adding visualization of algorithm progress would build intuition about convergence.

Comparative studies could examine student learning outcomes with different pedagogical approaches. Do students who implement algorithms perform better on conceptual questions? Do they develop stronger debugging skills?

The code could be packaged as an educational library, providing clean implementations of additional algorithms. Logistic regression, decision trees, and principal component analysis would be natural additions.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, NY, USA, 2006. doi: 10.1007/978-0-387-45528-0.

[2] Sebastian Raschka. *Python Machine Learning*. Packt Publishing, Birmingham, U.K., 2015.

[3] George A. F. Seber and Alan J. Lee. *Linear Regression Analysis*. Wiley, Hoboken, NJ, USA, 2nd edition, 2012. doi: 10.1002/9781118274370.

[4] James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. 5th Berkeley Symp. Math. Statist. Probab.*, pages 281–297, 1967.

[5] Travis E. Oliphant. Python for scientific computing. *Comput. Sci. Eng.*, 9(3):10–20, 2007. doi: 10.1109/MCSE.2007.58.

[6] John D. Hunter. Matplotlib: A 2d graphics environment. *Comput. Sci. Eng.*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[7] Hans Petter Langtangen. *A Primer on Scientific Programming with Python*. Springer, Berlin, Germany, 5th edition, 2016. doi: 10.1007/978-3-662-49887-3.

[8] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, CA, USA, 4th edition, 2016.

[9] Elisabeth Sulmont, Elizabeth Patitsas, and Jeremy R. Cooperstock. Measuring the impact of automated programming assessment on students. In *Proc. 50th ACM Tech. Symp. Comput. Sci. Educ.*, pages 1234–1240, 2019. doi: 10.1145/3287324.3287401.

[10] S. Parekh. Automated code description: Practical linguistic systems for agile development environments. In *Proc. IEEE 4th Int. Conf. Advance. Technol.*, pages 1–7, 2025. doi: 10.1109/ICONAT66879.2025.11362875.

[11] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. *Introduction to Linear Regression Analysis*. Wiley, Hoboken, NJ, USA, 6th edition, 2021. doi: 10.1002/9781119644668.

[12] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, New York, NY, USA, 2nd edition, 2009. doi: 10.1007/978-0-387-84858-7.

[13] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *Proc. 18th Annu. ACM-SIAM Symp. Discrete Algorithms*, pages 1027–1035, 2007.

[14] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Trans. Inf. Theory*, 28(2):129–137, 1982. doi: 10.1109/TIT.1982.1056489.

[15] A. Saxena. Portfolio optimization with stochastic return functions: An algorithmic approach. In *Proc. 12th Int. Conf. Emerg. Trends Eng. Technol. Signal Inf. Process.*, pages 1–6, 2025. doi: 10.1109/ICETETSIP64213.2025.11156675.

[16] Muskan Pawar. Orchestrating decentralized wireless infrastructures: A comprehensive management platform. 2025. doi: 10.67231/4z837x12.

[17] Monisha Rengaraj. Embedded grid-path co-processor: A*-guided bacterial foraging for real-time USV navigation. 2025. doi: 10.67231/8s96mc67.

[18] Rayhan Khan. Development of an AI for versus tetris using evolutionary algorithms. 2025. doi: 10.67231/htp1hp30.

[19] Krutika Shah. Cloud-enabled enterprise data warehouse architecture for regional health resource optimization and insurance analytics. 2025. doi: 10.67231/y64edh20.

[20] Fazle Hakeem Ghory. Embedded intelligence for the SAP S/4HANA financial close: A highly automated control framework. *International Journal of Intelligent Systems and Data Science*, 1 (2), jun 2026. doi: 10.67231/s0h5b367. URL `https://journals.femington.com/ijisds/article/view/447`.

[21] D. T. Tandel. Foodfresh: A multi-chain blockchain solution for interoperable and transparent food supply chain networks, 2025.

[22] Afsar Khan. Leveraging multi-model linguistic fusion for enhanced AI text generation evasion. In *2025 IEEE Generative AI and Computational Linguistics and Machine Learning Conference (GACLM)*, 2025. doi: 10.1109/GACLM67198.2025.11231821.

[23] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, Cambridge, MA, USA, 2012.