Calculus Project

Simon Holm
AI503: Calculus
Teacher: Shan Shan

Contents

1. Task 1: Create Example Data

1.1. Question:

To create a synthetic dataset, let's generate x-values and corresponding y-values with some added noise. Use the code below to create a set of points that should roughly lie on a line. You may choose different values for the true slope and intercept.

```
import numpy as np
import random

# Set a seed for reproducibility
np.random.seed(42)

# Generate x-values
x = np.linspace(0, 10, 50)
# Define the true slope and intercept
true_m = 2
true_b = 5
# Generate y-values with some noise
y = true_b + true_m * x + np.random.normal(0, 1, len(x))
```

Write Python code to plot the example data and plot the line given by $y = \text{true\_b} + \text{true\_m} \times x$.

1.2. Answer:

The following code below creates a set of points (x, y) which roughly lie on a line.

```
import numpy as np
import random

# set a seed for reproducibility
np.random.seed(42)

# define the number of points in the dataset
number_of_points = 100

# generate x values
x = np.linspace(0, 10, number_of_points)

# to generate corresponding y values with some noise
# so lets define the function values
slope = 2
intercept = 5
y = slope * x + intercept + np.random.normal(0, 1, size=x.shape)
```

The data has then been plotted using python libary “matplotlib”

```
import matplotlib.pyplot as plt

# *data implementation*

# Ploting with matplotlib
plt.figure(figsize=(10, 6))
plt.scatter(x, y, alpha=0.7, color='blue', s=20, label='Noisy data points')
plt.plot(x, slope * x + intercept, 'r--', linewidth=2, label=f'True line: y =
{slope}x + {intercept}')
plt.xlabel('x values')
plt.ylabel('y values')
plt.title('Linear Function with Random Noise')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

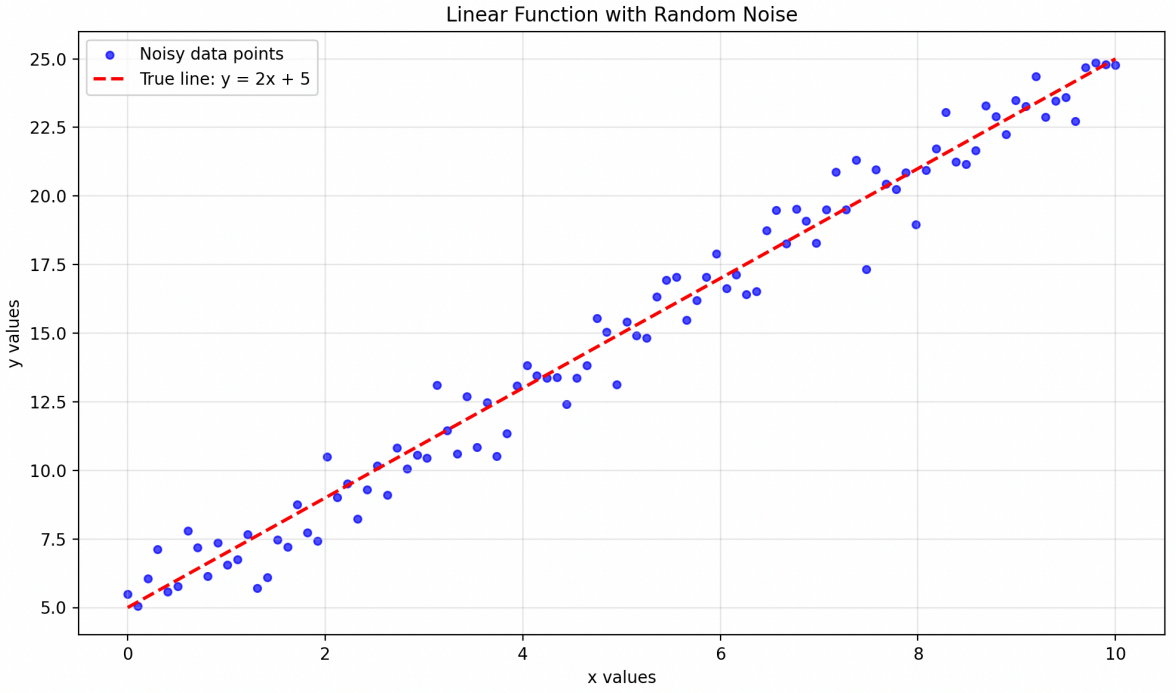


Figure 1: Noisy linear data generated with slope 2 and intercept 5; dashed red line shows the true model $y = 2x + 5$.

2. Task 2: Modeling the Least Squares Line

2.1. Question (a):

For each data point (x_i, y_i), show that the corresponding point on the least squares line has a y-coordinate of $b + mx_i$.

2.2. Answer (a):

The objective is to show that using the least squares line, it is possible to describe each point as a corresponding cordinate with a line $y = mx + b$

First calculate how much y changes per change in x

This is done by

$$\text{slope} = \frac{N\sum(xy) - \sum x \sum y}{N\left(\sum x^2 - (\sum x)^2\right)} \tag{1}$$

Where N is the number of datapoints. This can easily be done using python libary numpy

```
N = numOfPoints
sum_x = np.sum(x)
sum_y = np.sum(y)
sum_xy = np.sum(x * y)
sum_x_squared = np.sum(x**2)

# Numerator: N*sum(xy) - sum(x)*sum(y)
numerator = N * sum_xy - sum_x * sum_y

# Denominator: N*sum(x²) - (sum(x))²
denominator = N * sum_x_squared - (sum_x)**2

# Calculate slope
calculated_slope = numerator / denominator

# Then
print(f"Calculated slope: {calculated_slope:.4f}")

[OUTPUT] Calculated slope: 2.0138
```

The calculated slope ≈ 2.0138. And since the true slope of the datapoints are based on a slope $= 2$ the small difference of 0.0138 makes good sense, since the datapoints has been scattered from the true line of $y = 2x + b$.

Now to find the intercept, this is done by

$$\text{intercept} = \frac{\sum_y - \text{calculated\_slope} * \sum_x}{N} \tag{2}$$

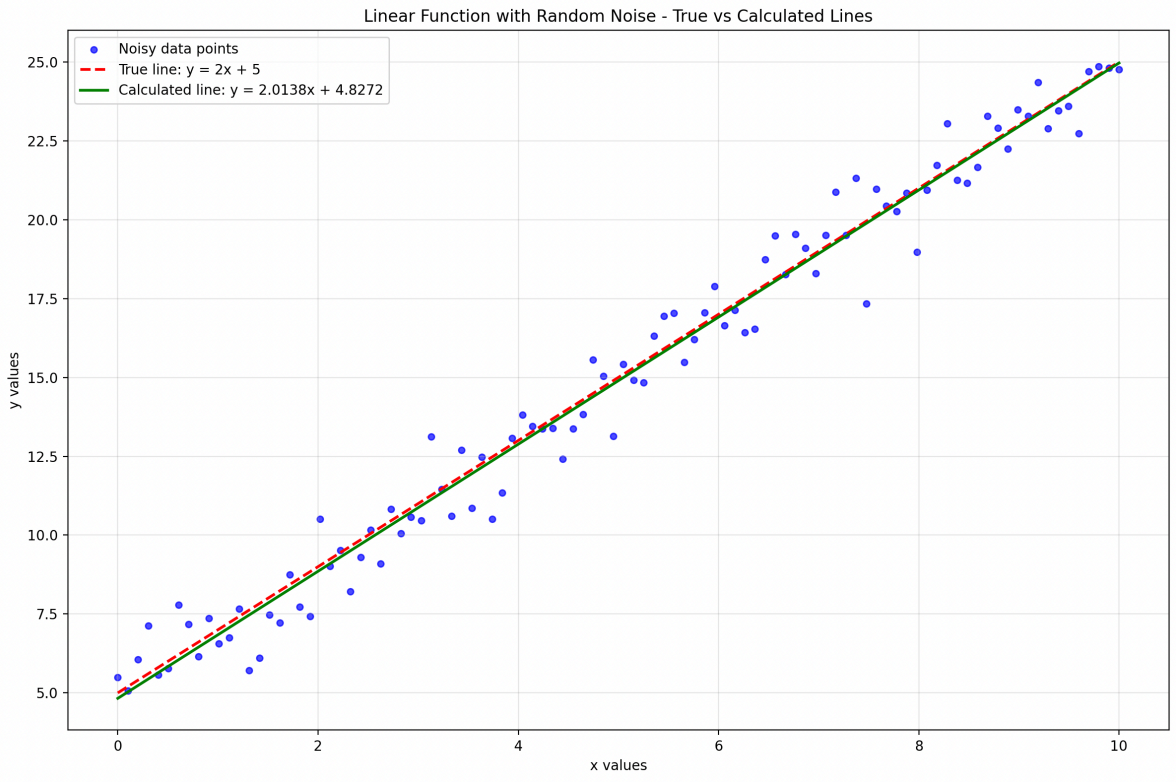
Where N is still the number of datapoints.

This is calculated using python libary numpy

```
# slope calculation from previous code block

calculated_intercept = (sum_y - calculated_slope * sum_x) / N

print(f"Calculated intercept: {calculated_intercept:.4f}")

[OUTPUT] Calculated intercept: 4.8272
```

So by this the calculated intercept is calculated to be ≈ 4.8272.

Finally by this the least squares line can be described as $y = 2.0138x + 4.8272$



The graph above clearly shows that the calculated line aligns roughly with both the data as well as the actual true line $y = 2x + 5$.

2.3. Question (b):

Implement a Python function, y_predicted(x, b, m), that calculates $b + mx$ for a given x, b, and m.

2.4. Answer (b):

For this task the following python function has been implemented:

```python
def y_predicted(x, b, m):
    """
    Calculate predicted y value for given x, intercept b, and slope m.
    """
    return b + m * x
```

3. Task 3: Calculating Squared Distances

3.1. Question (a):

Show that for each data point (x_i, y_i), the square of the vertical distance from it to the point on the line is $(y_i - (b + mx_i))^2$.

3.2. Answer (a):

The vertical distance from the point (x_i, y_i) to the line $y = b + mx$ is given by the difference in their y-coordinates:

$$\text{Distance} = y_i - (b + mx_i) \tag{3}$$

To find the square of this distance, we simply square the expression:

$$\text{Squared Distance} = (y_i - (b + mx_i))^2 \tag{4}$$

This shows that for each data point (x_i, y_i), the square of the vertical distance from it to the point on the line is indeed $(y_i - (b + mx_i))^2$.

3.3. Question (b):

Write a Python function, squared_distance(y, y_pred), to compute $\left(y - y_{\text{predicted}}\right)^2$ for values y and $y_{\text{predicted}}$.

3.4. Answer (b):

For this task the following python function has been implemented:

```python
def squared_distance(y, y_pred):
    """
    Compute squared distance between actual and predicted values.
    """
    return (y - y_pred) ** 2
```

4. Task 4: Defining and Minimizing the Cost Function

4.1. Question (a):

Define the cost function $f(b, m)$ as the sum of all n squared distances:

$$f(b,m) = \sum_{i=1}^{n} (y_i - (b + mx_i))^2 \tag{5}$$

Show that the partial derivatives $\frac{\partial f}{\partial b}$ and $\frac{\partial f}{\partial m}$ are:

$$\frac{\partial f}{\partial b} = -2 \sum_{i=1}^{n} (y_i - (b + mx_i)) \tag{6}$$

$$\frac{\partial f}{\partial m} = -2 \sum_{i=1}^{n} (y_i - (b + mx_i)) \cdot x_i \tag{7}$$

4.2. Answer (a):

Partial differentiation for both b and m with use of the chainrule

$$\frac{\partial}{\partial x} f(g(x)) = f'(g(x)) \cdot g(x) \tag{8}$$

This is done firstly with respect to b

$$\frac{\partial f}{\partial b} = \sum_{i=1}^{n} \frac{\partial}{\partial x} (y_i - (b + mx_i))^2 \cdot \frac{\partial}{\partial b} y_i - (b + mx_i) \tag{9}$$

$$\frac{\partial f}{\partial b} = \sum_{i=1}^{n} 2(y_i - (b + mx_i)) \cdot (-1) \tag{10}$$

$$\frac{\partial f}{\partial b} = -2 \sum_{i=1}^{n} (y_i - (b + mx_i)) \tag{11}$$

now for the derivative with respect to m

$$\frac{\partial f}{\partial m} = \sum_{i=1}^{n} \frac{\partial}{\partial m} (y_i - (b + mx_i))^2 \cdot \frac{\partial}{\partial m} y_i - (b + mx_i) \tag{12}$$

$$\frac{\partial f}{\partial m} = \sum_{i=1}^{n} 2(y_i - (b + mx_i)) \cdot (-x_i) \tag{13}$$

$$\frac{\partial f}{\partial m} = -2 \sum_{i=1}^{n} (y_i - (b + mx_i)) \cdot x_i \tag{14}$$

4.3. Question (b):

Write three Python functions:

(1) sum_of_squared_distances(x, y, b, m)
(2) partial_derivative_b
(3) partial_derivative_m

to compute $f(b, m)$, $\frac{\partial f}{\partial b}$, and $\frac{\partial f}{\partial m}$.

4.4. Answer (b):

For this task the function get_predicted_values have implemented

```
def get_predicted_values(x, b, m):
  """
  Returns a list of predicted y values for all x points.
  """
  predicted_values = []
  for xi in x:
      y_pred = y_predicted(xi, b, m)
      predicted_values.append(y_pred)
  return predicted_values
```

The function implemented above calculates all the predicted values for the dataset.

4.4.1. sum_of_squares()

```
def sum_of_squares(x, y, b, m) -> float:
    """
    Sums the square of distances between actual and predicted values.
    Formula: f(b, m) = sum((y_i - (b + m * x_i))^2)
    """
    predicted_values = get_predicted_values(x, b, m)
    total_sum = 0
    for i in range(len(y)):
        squared_diff = squared_distance(y[i], predicted_values[i])
        total_sum += squared_diff
    return total_sum
```

The implementation above, uses a help function (get_predicted_values) to firstly get all predicted values for the x-values in the dataset. Then it computes the sum of squares using the get_predicted_values function.

4.4.2. partial_derivative_b

```
def partial_derivative_b(x, y, b, m):
    """
    Compute partial derivative with respect to b.
    Formula: ∂f/∂b = -2 * sum(y_i - (b + m*x_i))
    """
    dfdb = 0
    for i in range(len(y)):
        dfdb += y[i] - y_predicted(x[i], b, m)
    return -2 * dfdb
```

4.4.3. partial_derivative_m

```
def partial_derivative_m(x, y, b, m):
    """
    Compute partial derivative with respect to m.
    Formula: ∂f/∂m = -2 * sum((y_i - (b + m*x_i)) * x_i)
    """
    dfdm = 0
    for i in range(len(y)):
        dfdm += (y[i] - y_predicted(x[i], b, m)) * x[i]
    return -2 * dfdm
```

4.5. Question (c):

Write Python code to compute $\frac{\partial f}{\partial b}$ and $\frac{\partial f}{\partial m}$ using Symbolic differentiation and Automatic differentiation. Check your answers and compare the computation time.

4.6. Answer (c):

4.6.1. Symbolic Differentiation with SymPy

```python
import sympy as sp
import time

# Define symbolic variables
b_sym, m_sym = sp.symbols('b m')
x_sym, y_sym = sp.symbols('x y')

# Define cost function symbolically
cost_function = (y_sym - (b_sym + m_sym * x_sym))**2

# Compute symbolic derivatives
df_db_sym = sp.diff(cost_function, b_sym)
df_dm_sym = sp.diff(cost_function, m_sym)


def symbolic_derivatives(x, y, b, m):
    """Compute derivatives using symbolic differentiation"""
    df_db_total = 0
    df_dm_total = 0

    for i in range(len(x)):
        # Substitute values into symbolic expressions
        df_db_val = df_db_sym.subs([(x_sym, x[i]), (y_sym, y[i]),
                                    (b_sym, b), (m_sym, m)])
        df_dm_val = df_dm_sym.subs([(x_sym, x[i]), (y_sym, y[i]),
                                    (b_sym, b), (m_sym, m)])
        df_db_total += df_db_val
        df_dm_total += df_dm_val

    return float(df_db_total), float(df_dm_sym)
```

4.6.2. Automatic Differentiation with JAX

```
import jax.numpy as jnp
from jax import autograd

def cost_function_jax(params, x, y):
    """Cost function for automatic differentiation"""
    b, m = params
    predictions = b + m * x
    return jnp.sum((y - predictions)**2)

# Create gradient function automatically
grad_function = grad(cost_function_jax)

def automatic_derivatives(x, y, b, m):
    """Compute derivatives using automatic differentiation"""
    params = jnp.array([b, m])
    x_jax = jnp.array(x)
    y_jax = jnp.array(y)

    gradients = grad_function(params, x_jax, y_jax)
    return float(gradients[0]), float(gradients[1])
```

4.6.3. Comparison and Timing

```
import time

# Manual implementation
start_time = time.time()
manual_db = partial_derivative_b(x, y, intercept, slope)
manual_dm = partial_derivative_m(x, y, intercept, slope)
manual_time = time.time() - start_time

# Symbolic differentiation
start_time = time.time()
sym_db, sym_dm = symbolic_derivatives(x, y, intercept, slope)
symbolic_time = time.time() - start_time

# Automatic differentiation
start_time = time.time()
auto_db, auto_dm = automatic_derivatives(x, y, intercept, slope)
auto_time = time.time() - start_time

print("Comparison Results:")
print(f"Manual:     ∂f/∂b = {manual_db:.6f}, ∂f/∂m = {manual_dm:.6f}")
print(f"Symbolic:   ∂f/∂b = {sym_db:.6f}, ∂f/∂m = {sym_dm:.6f}")
print(f"Automatic:  ∂f/∂b = {auto_db:.6f}, ∂f/∂m = {auto_dm:.6f}")

print(f"\nTiming Comparison:")
print(f"Manual: {manual_time:.6f} seconds")
print(f"Symbolic: {symbolic_time:.6f} seconds")
print(f"Automatic: {auto_time:.6f} seconds")

[OUTPUT]
Comparison Results:
Manual:     ∂f/∂b = 20.769303, ∂f/∂m = 80.393319
Symbolic:   ∂f/∂b = 20.769303, ∂f/∂m = 80.393319
Automatic:  ∂f/∂b = 20.769289, ∂f/∂m = 80.393196

Timing Comparison:
Manual: 0.000046 seconds
Symbolic: 0.146954 seconds
Automatic: 0.002684 seconds
```

Results show all methods give nealy identical results, with automatic and maunal differentiation being fastest.

5. Task 5: Setting Up and Solving the System of Equations

5.1. Question (a):

Show that setting $\frac{\partial f}{\partial b} = 0$ and $\frac{\partial f}{\partial m} = 0$ results in the linear system:

$$nb + \left(\sum_{i=1}^{n} x_i\right) m = \sum_{i=1}^{n} y_i \tag{15}$$

$$\left(\sum_{i=1}^{n} x_i\right) b + \left(\sum_{i=1}^{n} x_i^2\right) m = \sum_{i=1}^{n} x_i y_i \tag{16}$$

Explain why solving for b and m minimizes the cost function.

5.2. Answer (a):

When setting $\frac{\partial f}{\partial b} = 0$ and $\frac{\partial f}{\partial m} = 0$

this becomes

$$-2\sum_{i=1}^{n}(y_i - (b + mx_i)) = 0 \tag{17}$$

$$-2\sum_{i=1}^{n}(y_i - (b + mx_i)) \cdot x_i = 0 \tag{18}$$

Start by $\frac{\partial f}{\partial b}$

$$\begin{aligned}
-2\sum_{i=1}^{n}(y_i - (b + mx_i)) &= 0 \\
\sum_{i=1}^{n}(y_i - b - mx_i) &= 0 \\
\sum_{i=1}^{n} y_i - \sum_{i=1}^{n} b - \sum_{i=1}^{n} mx_i &= 0 \\
\sum_{i=1}^{n} y_i - nb - m\sum_{i=1}^{n} x_i &= 0 \\
nb + m\sum_{i=1}^{n} x_i &= \sum_{i=1}^{n} y_i
\end{aligned} \tag{19}$$

Then $\frac{\partial f}{\partial m}$

$$\begin{aligned} -2\sum_{i=1}^{n}(y_i - (b + mx_i)) \cdot x_i &= 0 \\ \sum_{i=1}^{n}(y_i - (b + mx_i)) \cdot x_i &= 0 \\ \sum_{i=1}^{n}(y_i x_i - (bx_1 + mx_i^2)) &= 0 \\ \sum_{i=1}^{n}(y_i x_i) - \left(b\sum_{i=1}^{n}x_1 + m\sum_{i=1}^{n}x_i^2\right) &= 0 \\ \sum_{i=1}^{n}(y_i x_i) &= \left(b\sum_{i=1}^{n}x_1 + m\sum_{i=1}^{n}x_i^2\right) \end{aligned} \tag{20}$$

This gives us the system of linear equations:

$$nb + \left(\sum_{i=1}^{n}x_i\right)m = \sum_{i=1}^{n}y_i \tag{21}$$

$$\left(\sum_{i=1}^{n}x_i\right)b + \left(\sum_{i=1}^{n}x_i^2\right)m = \sum_{i=1}^{n}x_i y_i \tag{22}$$

Why does solving for b and m minimize the cost function?

Setting the partial derivatives equal to zero finds the critical points of the cost function $f(b, m)$. Since our cost function is:

$$f(b, m) = \sum_{i=1}^{n}\left(y_i - (b + mx_i)\right)^2 \tag{23}$$

To confirm this critical point is indeed a minimum, we examine the **Hessian matrix** containing all second-order partial derivatives:

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial b^2} & \frac{\partial^2 f}{\partial b \partial m} \\ \frac{\partial^2 f}{\partial m \partial b} & \frac{\partial^2 f}{\partial m^2} \end{bmatrix} \tag{24}$$

Computing the second derivatives:

$$\frac{\partial^2 f}{\partial b^2} = \frac{\partial}{\partial b}\left(-2\sum_{i=1}^{n}(y_i - (b + mx_i))\right) = 2n \tag{25}$$

$$\frac{\partial^2 f}{\partial b \partial m} = \frac{\partial}{\partial m}\left(-2\sum_{i=1}^{n}(y_i - (b + mx_i))\right) = 2\sum_{i=1}^{n}x_i \tag{26}$$

$$\frac{\partial^2 f}{\partial m^2} = \frac{\partial}{\partial m}\left(-2\sum_{i=1}^{n}(y_i - (b + mx_i)) \cdot x_i\right) = 2\sum_{i=1}^{n}x_i^2 \tag{27}$$

This gives us:

$$H = \begin{bmatrix} 2n & 2\sum_{i=1}^{n} x_i \\ 2\sum_{i=1}^{n} x_i & 2\sum_{i=1}^{n} x_i^2 \end{bmatrix} \tag{28}$$

Then find the eigen values

$$\det(A - \lambda I) = \det\left(\begin{bmatrix} 2n - \lambda & 2\sum_{i=1}^{n} x_i \\ 2\sum_{i=1}^{n} x_i & 2\sum_{i=1}^{n} x_i^2 - \lambda \end{bmatrix} \right) = 0 \tag{29}$$

$$(2n - \lambda) \cdot \left(2\sum_{i=1}^{n} x_i^2 - \lambda \right) - \left(2\sum_{i=1}^{n} x_i \right)^2 = 0 \tag{30}$$

$$4n\widehat{x^2} - 2n\lambda - 2\widehat{x^2}\lambda + \lambda^2 - 4\hat{x}^2 = 0 \tag{31}$$

where $\hat{x} = \sum_{i=1}^{n} x_i$

$$\lambda^2 + \left(-2n - 2\widehat{x^2}\right)\lambda + 4n\widehat{x^2} - 4\hat{x}^2 = 0 \tag{32}$$

since $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

so..

$$\lambda = \frac{-\left(-2n - 2\widehat{x^2}\right) - \sqrt{\left(-2n - 2\widehat{x^2}\right)^2 - 4\left(4n\widehat{x^2} - 4\hat{x}^2\right)}}{2} \tag{33}$$

and

$$\lambda = \frac{-\left(-2n - 2\widehat{x^2}\right) + \sqrt{\left(-2n - 2\widehat{x^2}\right)^2 - 4\left(4n\widehat{x^2} - 4\hat{x}^2\right)}}{2} \tag{34}$$

```
coeff_a = 1.0
coeff_b = -(2.0*N + 2.0*sum_x_squared)
coeff_c = 4.0*N*sum_x_squared - 4.0*(sum_x**2)

discriminant = coeff_b**2 - 4.0*coeff_a*coeff_c
lambda1 = (-coeff_b - np.sqrt(discriminant)) / (2.0*coeff_a)
lambda2 = (-coeff_b + np.sqrt(discriminant)) / (2.0*coeff_a)

print(f"Eigenvalue 1: {lambda1}")
print(f"Eigenvalue 2: {lambda2}")
print(f"Both positive: {lambda1 > 0 and lambda2 > 0}")

[OUTPUT]:
Eigenvalue 1: 49.63981706705363
Eigenvalue 2: 6850.696883269647
Both positive: True
```

Since $\lambda_1 \approx 49.6$ and $\lambda_2 \approx 6850.7$ both are positive the critical point must be a local minimum

5.3. Question (b):

Write a Python function, solve_least_squares(x, y), to solve for b and m using this system of linear equations.

5.4. Answer (b):

```python
def solve_least_squares(x, y):
    """
    Solve for b and m using the specific linear system:
    n*b + (Σx_i)*m = Σy_i
    (Σx_i)*b + (Σx_i²)*m = Σx_i*y_i
    Returns: (intercept, slope)
    """
    n = len(x)
    sum_x = np.sum(x)
    sum_y = np.sum(y)
    sum_xy = np.sum(x * y)
    sum_x_squared = np.sum(x**2)

    # Set up the coefficient matrix A and right-hand side vector b
    # A * [b, m]^T = constants_vector
    A = np.array([[n, sum_x],
                  [sum_x, sum_x_squared]])

    constants_vector = np.array([sum_y, sum_xy])

    # Solve the linear system A * params = constants_vector
    result = np.linalg.solve(A, constants_vector)

    # Return intercept (b) and slope (m)
    return result[0], result[1]
```

6. Task 6: Deriving Explicit Formulas for b and m

6.1. Question (a):

Solve the equations above to derive explicit formulas for b and m:

$$b = \frac{\left(\sum_{i=1}^{n} x_i^2\right)\left(\sum_{i=1}^{n} y_i\right) - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} x_i y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \tag{35}$$

$$m = \frac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \tag{36}$$

6.2. Answer (a):

for simplicitys sake $\sum_{i=1}^{n} x_i = \sum x$

start with the system of equations:

$$nb + \left(\sum_{i=1}^{n} x_i\right) m = \sum_{i=1}^{n} y_i \tag{37}$$

$$\left(\sum_{i=1}^{n} x_i\right) b + \left(\sum_{i=1}^{n} x_i^2\right) m = \sum_{i=1}^{n} x_i y_i \tag{38}$$

then isolate b and m from the equations:

$$b = \frac{1}{n}\left(\sum y - m\sum x\right) \tag{39}$$

$$m = \frac{\sum xy - b\sum x}{\sum x^2} \tag{40}$$

now express b with the equation for m

$$b = \frac{1}{n}\left(\sum y - \left(\frac{\sum xy - b\sum x}{\sum x^2}\right)\sum x\right) \tag{41}$$

now simplify with common fraction rule $\frac{a}{b} - \frac{c}{d} = \frac{ad-bc}{bd}$

$$b = \frac{\sum y}{n} - \left(\frac{\sum xy - b\sum x}{n\sum x^2}\right)\sum x = \frac{n\sum y\sum x^2 - n\sum x\sum xy - nb(\sum x)^2}{n^2\sum x^2} \tag{42}$$

now simplify further

$$b = \frac{\sum y\sum x^2 - \sum x\sum xy - b(\sum x)^2}{n\sum x^2} \tag{43}$$

This can be rewritten in order to isolate b:

$$\begin{aligned} b &= \frac{\sum y \sum x^2 - \sum x \sum xy}{n \sum x^2} - \frac{b(\sum x)^2}{n \sum x^2} \\ b + \frac{b(\sum x)^2}{n \sum x^2} &= \frac{\sum y \sum x^2 - \sum x \sum xy}{n \sum x^2} \end{aligned} \tag{44}$$

Multiply both sides by $n \sum x^2$

$$\begin{aligned} b \cdot n \sum x^2 + b\Big(\sum x\Big)^2 &= \sum y \sum x^2 - \sum x \sum xy \\ b\Big(n \sum x^2 + \Big(\sum x\Big)^2\Big) &= \sum y \sum x^2 - \sum x \sum xy \\ b &= \frac{\sum y \sum x^2 - \sum x \sum xy}{n \sum x^2 - (\sum x)^2} \end{aligned} \tag{45}$$

Now to express m

First express m using the equation from b

$$m = \frac{\sum xy - b \sum x}{\sum x^2} = \frac{\sum xy - \left(\frac{1}{n}(\sum y - m \sum x)\right) \sum x}{\sum x^2} \tag{46}$$

Now simplify

$$\begin{aligned} m &= \frac{\sum xy - \left(\frac{1}{n}(\sum y - m \sum x)\right) \sum x}{\sum x^2} \\ &= \frac{\sum xy - \left(\frac{1}{n} \sum y - \frac{1}{n} m \sum x\right) \sum x}{\sum x^2} \\ &= \frac{\sum xy - \left(\frac{1}{n} \sum y \sum x - \frac{1}{n} m (\sum x)^2\right)}{\sum x^2} \\ &= \frac{\sum xy - \frac{\sum y \sum x}{n} + \frac{m(\sum x)^2}{n}}{\sum x^2} \end{aligned} \tag{47}$$

This can be rewritten in order to isolate b:

$$\begin{aligned} m &= \frac{\sum xy - \frac{\sum y \sum x}{n} + \frac{m(\sum x)^2}{n}}{\sum x^2} \\ m &= \frac{\sum xy}{\sum x^2} - \frac{\sum y \sum x}{n \sum x^2} + \frac{m(\sum x)^2}{n \sum x^2} \\ m - \frac{m(\sum x)^2}{n \sum x^2} &= \frac{\sum xy}{\sum x^2} - \frac{\sum y \sum x}{n \sum x^2} \\ m \cdot \frac{n \sum x^2 - (\sum x)^2}{n \sum x} &= \frac{n \sum xy \sum x^2 - \sum x^2 \sum x \sum y}{n(\sum x^2)^2} \end{aligned} \tag{48}$$

Now that m is isolated, just simplify

$$\begin{aligned} m &= \frac{n \sum x^2 (n \sum xy \sum x^2 - \sum x^2 \sum x \sum y)}{n(\sum x^2)^2 \left(n \sum x^2 - (\sum x)^2\right)} \\ m &= \frac{n^2 \sum xy (\sum x^2)^2 - n \sum x \sum y (\sum x^2)^2}{n(\sum x^2)^2 \left(n \sum x^2 - (\sum x)^2\right)} \\ m &= \frac{n \sum xy - \sum x \sum y}{n \sum x^2 - (\sum x)^2} \end{aligned} \tag{49}$$

6.3. Question (b):

Implement solve_least_squares_formula(x, y) in Python to calculate b and m directly from these formulas. Compare with solve_least_squares(x, y) on example data.

6.4. Answer (b):

```
def solve_least_squares_formula(x, y):
    """
    Calculate b and m directly from explicit formulas.
    Returns: (intercept, slope)
    """
    N = len(x) # number of points in the data
    sum_x = np.sum(x)
    sum_y = np.sum(y)
    sum_xy = np.sum(x * y)
    sum_x_squared = np.sum(x**2)

    denominator = N * sum_x_squared - sum_x**2
    b = (sum_x_squared * sum_y - sum_x * sum_xy) / denominator
    m = (N * sum_xy - sum_x * sum_y) / denominator
    return b, m

# Method 1: QR Decomposition
matrix_b, matrix_m = solve_least_squares(x, y)
print(f"Matrix Method:")
print(f"  Intercept (b): {matrix_b:.8f}")
print(f"  Slope (m):     {matrix_m:.8f}")

# Method 2: Explicit Formula
formula_b, formula_m = solve_least_squares_formula(x, y)
print(f"\n Formula Method:")
print(f"  Intercept (b): {formula_b:.8f}")
print(f"  Slope (m):     {formula_m:.8f}")

# Calculate differences
matrix_formula_diff_b = abs(matrix_b - formula_b)
matrix_formula_diff_m = abs(matrix_m - formula_m)

print(f"\nDifferences:")
print(f"  Matrix vs Formula - Intercept: {matrix_formula_diff_b:.2e}")
print(f"  Matrix vs Formula - Slope:     {matrix_formula_diff_m:.2e}")

[OUTPUT]
Matrix Method:
  Intercept (b): 4.82718715
  Slope (m):     2.01379327

Formula Method:
  Intercept (b): 4.82718715
  Slope (m):     2.01379327

Differences:
  Matrix vs Formula - Intercept: 1.78e-15
  Matrix vs Formula - Slope:     4.44e-16
```

Timing:

```
  # Timing comparison
  import time

  # iterations for calculating avg speed for methods
  iterations = 1000

  # Time  matrix method
  start = time.time()
  for _ in range(iterations):
      solve_least_squares(x, y)
  matix_time = (time.time() - start)/iterations

  # Time formula method
  start = time.time()
  for _ in range(iterations):
      solve_least_squares_formula(x, y)
  formula_time = (time.time() - start)/iterations

  print(f"\nTiming (avg iteration):")
  print(f"  Matrix method: {matix_time:.6f} seconds")
  print(f"  Formular method: {formula_time:.6f} seconds")
  print(f"  Speed ratio: {matix_time/formula_time:.2f}x")

[OUTPUT]
Timing (avg iteration):
  Matrix method: 0.000009 seconds
  Formular method: 0.000005 seconds
  Speed ratio: 1.79x
```

The two methods are nearly identical in computation (they roughly give the same answer). Notice that the formula method is faster, probably since it's just formula computing.

7. Task 7: Comparing with Gradient Descent

7.1. Question (a):

Implement gradient descent to minimize $f(b, m)$ by iteratively updating b and m using the partial derivatives.

7.2. Answer (a):

```python
def gradient_descent(x, y, b, m, learning_rate=0.01, max_iterations=1000):
    """
    Minimize f(b, m) using gradient descent.
    """
    epsilon = 1e-6

    for _ in range(max_iterations)
        dfdb = partial_derivative_b(x, y, b, m)
        dfdm = partial_derivative_m(x, y, b, m)

        # Check for too lage numbers instability (infiniy or div by 0)
        if not (np.isfinite(dfdb) and np.isfinite(dfdm)):
            return b, m

        if abs(dfdb) < epsilon and abs(dfdm) < epsilon:
            return b, m

        # Update parameters for next iteration
        b = b - learning_rate * dfdb
        m = m - learning_rate * dfdm

    # max iterations have been reached
    return b, m
```

7.3. Question (b):

Compare results from solve_least_squares, solve_least_squares_formula, and gradient descent on the example dataset. Discuss any differences in convergence and accuracy.

7.4. Answer (b):

```
# keep low
learning_rate = 0.00001

ls_b, ls_m = solve_least_squares(x, y)
formula_b, formula_m = solve_least_squares_formula(x, y)
gd_b, gd_m = gradient_descent(x, y, 0, 0, learning_rate=0.00001,
max_iterations=10000)

print("\nRegression Results:")
print(f"Least Squares:        b = {ls_b:.6f}, m = {ls_m:.6f}")
print(f"Explicit Formula:     b = {formula_b:.6f}, m = {formula_m:.6f}")
print(f"Gradient Descent:     b = {gd_b:.6f}, m = {gd_m:.6f}")

print("\nDifferences:")
print(f"LS vs Formula:        Δb = {abs(ls_b-formula_b):.2e}, Δm = {abs(ls_m-
formula_m):.2e}")
print(f"LS vs GD:             Δb = {abs(ls_b-gd_b):.2e}, Δm = {abs(ls_m-
gd_m):.2e}")
print(f"Formula vs GD:        Δb = {abs(formula_b-gd_b):.2e}, Δm =
{abs(formula_m-gd_m):.2e}")

[OUTPUT]
Regression Results:
Least Squares:     b = 4.827187, m = 2.013793
Explicit Formula:     b = 4.827187, m = 2.013793
Gradient Descent:     b = 4.796321, m = 2.018434

Differences:
LS vs Formula:        Δb = 1.78e-15, Δm = 4.44e-16
LS vs GD:             Δb = 3.09e-02, Δm = 4.64e-03
Formula vs GD:        Δb = 3.09e-02, Δm = 4.64e-03
```

To increase the accuracy, I should adjust the learning rate for the gradient descent, so that the differences are minimal.

```
# keep low
learning_rate = 0.0001

# calculation

[OUTPUT]
Regression Results:
Least Squares:     b = 4.827187, m = 2.013793
Explicit Formula:     b = 4.827187, m = 2.013793
Gradient Descent:     b = 4.827187, m = 2.013793

Differences:
LS vs Formula:        Δb = 1.78e-15, Δm = 4.44e-16
LS vs GD:             Δb = 2.01e-08, Δm = 3.03e-09
Formula vs GD:        Δb = 2.01e-08, Δm = 3.03e-09
```

The results show that by adjusting the step size (learning_rate) in gradient descent, we can achieve higher accuracy.

It is important to note that gradient descent only approaches the true values for b and m, and cannot surpass the exact solution (least squares method). Gradient descent iteratively moves closer to the minimum, but its accuracy is limited by the learning rate, the number of iterations, and numerical precision.

Also, any difference between the least squares (LS) and true formula methods should theoretically be zero, as both solve the same system. However, small discrepancies may occur due to internal rounding and/or floating-points in the computations.

8. Task 8: Differentiation Methods Analysis

8.1. Question (a):

Based on your study of PartI_differentiation.ipynb and PartII_autograd.ipynb, discuss the advantages and limitations of numerical differentiation, symbolic differentiation, and automatic differentiation.

8.2. Answer (a):

Note that there are not just one of these method which definitivly bette than the others. They all have different advatnges as well as limitations.

1. Numerical Differentiation:

The big advangtage of numerical differentiation, is that "it does not matter at all how the function was calculated - only the final values of it!" [1]. This is because of the `np.gradient(f)` function, which makes numerical differentiation very easy and intuitive to work with.

On the other hand, as explained in the [1], you might lose some accuracy with numerical differentiation. Also since numerical values cant handle "jumps" of the derivative. a good example is $f(x) = |x|$,

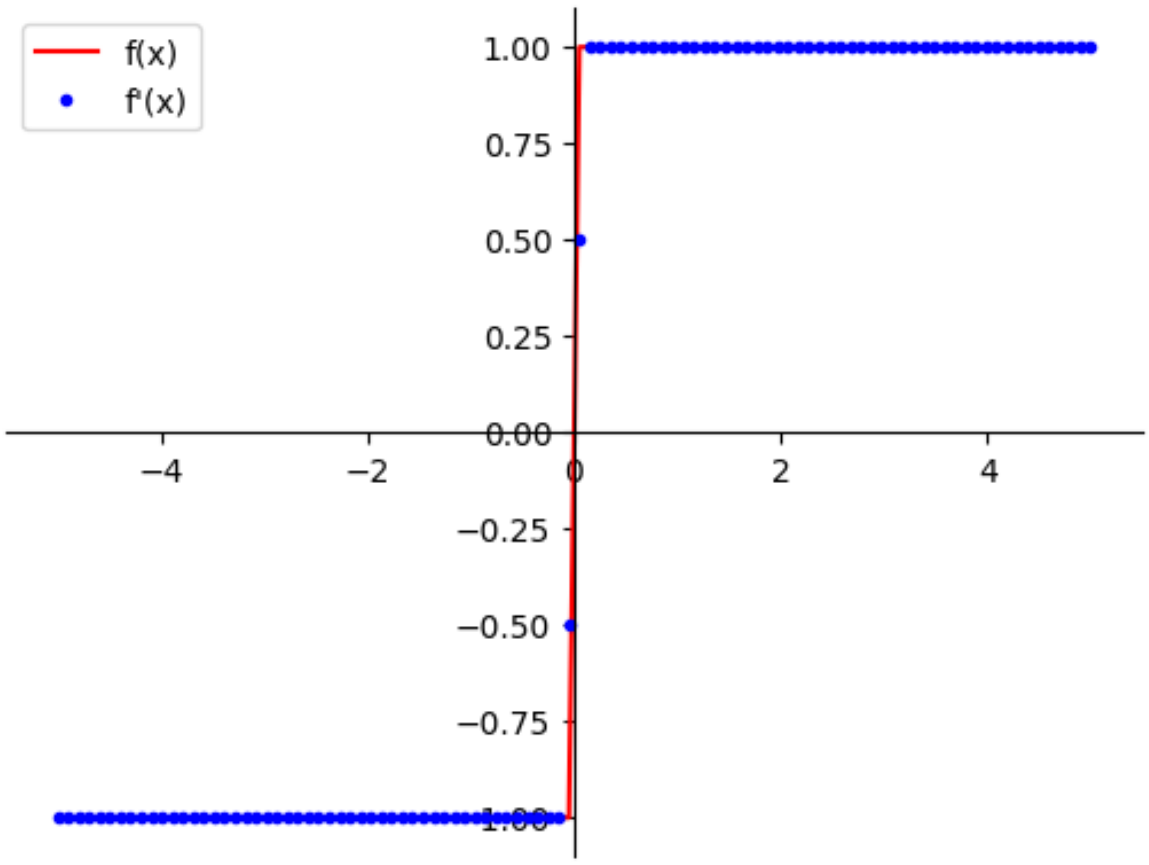


Figure 2: $f(x) = |x|$ shown as a graph [1],

1. Symbolic differentiation

   Symbolic differentiation uses exact representation of mathicatical objects [1]. Take the example `sp.sqrt(2)` will not be represented as 1.41421356237 but exactly as $\sqrt{2}$

   This way, calculations are more precise since no rounding is being done while calculating steps in the calculation of the result.

   Besides the jump in the derivative (e.g. $f(x) = |x|$) The main limitation of symbolic differentiation, is the computation time. Especially larger and more complex functions can get very slow. This is known as the **expression swell** [1].

2. Automatic differentiation

   Automatic differentiation offer a vastly differet and easier approch, with modern libaries like `Autograd` and `JAX` [2]. Take `JAX`, which is a libary full of usefull functions for differentiation, including the `jax.grad()` function which can differentiate whole python functions

   Automatic differentiation can be limiting in the sense that you cant really debug somthing like the `jax.grad()` or easily inspect the intermediate steps of the computation graph. Errors or unexpected results may be hard to trace, especially in complex models, because the differentiation happens behind the scenes.

8.3. Question (b):

Summarize the connection of multivariate chain rule and backpropagation, and discuss why backward differentiation is efficient for optimizing neural networks.

8.4. Answer (b):

Take the concept of a neural network as a vector (input) being transformed through layers until a loss is computed.

We can describe this as:

$$v_0 \to f_1(v_0) \to f_2(v_1) \to ... \to f_L(v_{L-1}) \to \text{Loss} \tag{50}$$

Where each layer might look something like this

$$f_L(v_L) = \sigma(W_L \cdot v_{L-1} + b_L) \tag{51}$$

This means that

$$\text{Loss} = f_L(f_{L-1}(f_{L-2}(...f_1(v_0)))) \tag{52}$$

To find the change of loss we now need to compute $\frac{\partial L}{\partial v_0}$

For this we use the multivariate chain rule

$$\frac{\partial L}{\partial v_0} = \frac{\partial L}{\partial f_L} \cdot \frac{\partial f_L}{\partial f_{L-1}} \cdot \frac{\partial f_{L-1}}{\partial f_{L-2}} \cdots \frac{\partial f_1}{\partial v_0} \tag{53}$$

Now with backprobagration we can go back though the layers and change the weights to reduce the loss

9. Conclusion

In this project, I explored several approaches to regression and differentiation in Python. Each method demonstrated distinct advantages and limitations in terms of speed, accuracy, and usability. I also examined how these mathematical concepts relate to machine learning and neural networks. Overall, this work provided valuable insights into the strengths of different techniques and deepened my understanding of their practical applications.

Bibliography

[1] *Differentiation in Python: Symbolic, Numerical and Automatic.*

[2] *Automatic Differentiation.*