Probabilistic Models

Contents

2.1. Linear Regression

In our previous chapter on probability, we delved into the fundamental concepts of probability and random variables. However, the distributions discussed were exclusively discrete, focusing on counts of letter or event occurrences. So, you might be wondering, how does this knowledge serve us in tackling broader analytical challenges? This question leads us into the realm of probabilistic modeling.
Let's begin with a problem that is likely familiar to everyone or has been encountered at least once. Envision a scenario where someone provides us with the data for the following figure:

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

# Configure Jupyter Notebook to display Matplotlib figures as SVG images
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

# Initialize a random number generator with a specific seed to ensure
reproducibility
rnd = np.random.RandomState(seed=42)  # Setting a specific seed value for
consistency

# Specify the number of data points
n_data = 20

# Define the true coefficients for a linear model
a_true = 1.73  # Modifying the slope coefficient
b_true = 4.53  # Adjusted the linear term coefficient

# Generate an array of random 'x' values uniformly distributed in the
range [0, 5]
```

```
x = rnd.uniform(0, 5, n_data)  # Creating random 'x' values within a
different range

# Calculate the corresponding 'y' values using the true linear model
y = a_true * x + b_true  # Calculating 'y' values

# Introduce random heteroscedastic Gaussian uncertainties in the 'y'
values
y_err = rnd.uniform(0.1, 0.8, size=n_data)  # Defining uncertainties with
different range

# Add Gaussian noise to the 'y' data
y = y + rnd.normal(0, y_err)  # Incorporating Gaussian noise into 'y' data

# Plot data
plt.errorbar(x, y, y_err, marker='o', linestyle='none')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.tight_layout()
plt.savefig('/content/drive/MyDrive/PSU/ColabNotebooks/Example2_1.png',
dpi=300)
```

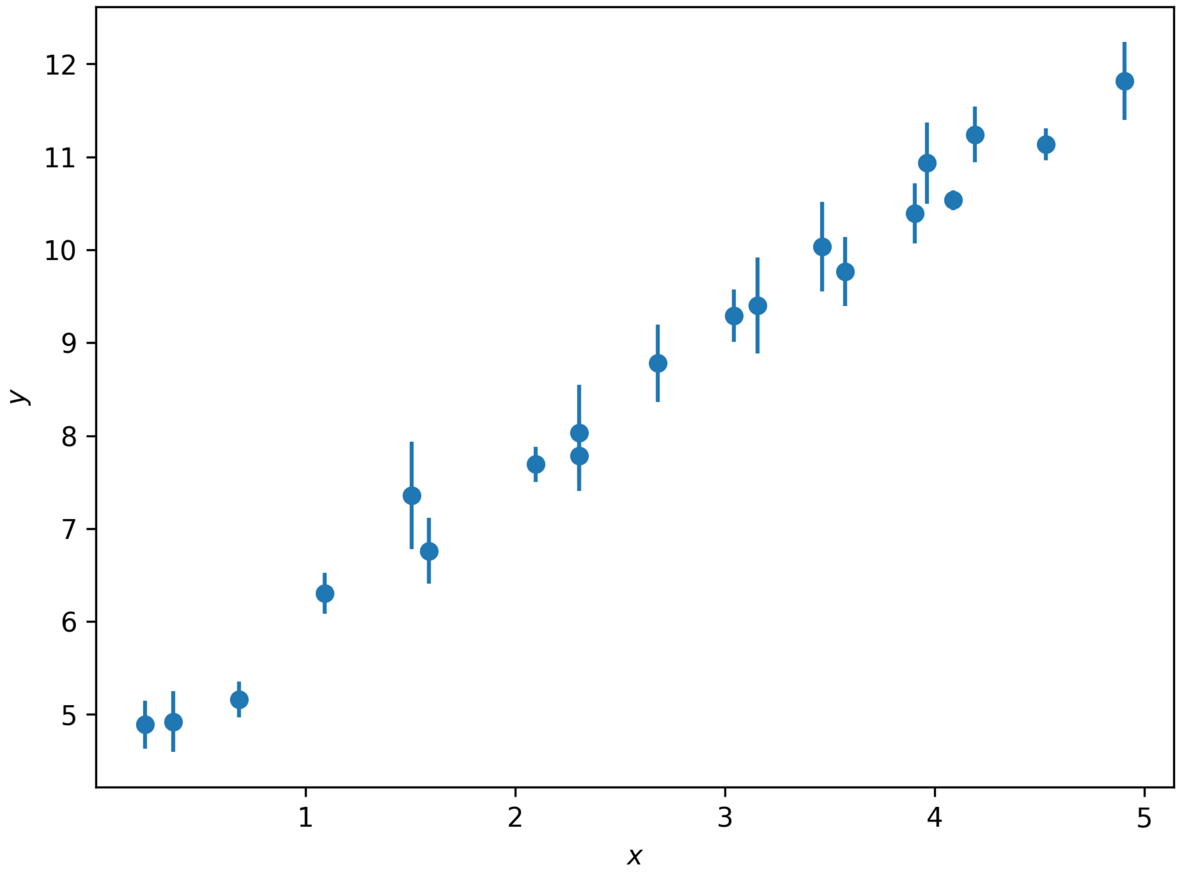

Figure 2.1

What can we conclude from it? We need to reason about the data-generating processes, i.e. the underlying models and their parameters. Remember: We usually need a signal model and a noise model.

Let's start with the obvious: we observe N data points $\mathbf{y}= \{y_1, y_2, \ldots y_N\}$ at locations $x = \{x_1, x_1, \ldots x_N\}$. It seems reasonable to assume that they are well-described by a linear relation:

$$y = f(x; a, b) = ax + b$$

We can also make a reasonable assumption that variations from the ideal linear relationship can be linked to the existence of additive noise. It is this noise factor, rather than the additive component, that gives the model its probabilistic characteristics. The inclusion of symmetric errors denoted as $\sigma = \{\sigma_1, \sigma_2, \ldots \sigma_N\}$, and exclusively these errors, suggests that they probably correspond to the standard deviation of a Gaussian distribution. This selection is justified because the Gaussian distribution is the most versatile and commonly used model for representing uncertainties in both position and magnitude.

However, our examination of the noise model isn't complete yet. The variability in the Gaussian's variance is variable; otherwise, a single σ value would have been provided. It's also noteworthy to observe that the x-values remain unaffected. There are no errors in the x-direction; they appear to be precise and independent measurements. What's more challenging to determine is whether

the y-errors themselves are unrelated. If errors were correlated, it would imply that, for instance, if one data point y_i deviates above the reference line, it might lead to a nearby data point y_j also deviating upwards. However, given the absence of information on this matter, we will assume that the uncertainties are independent and not influenced by each other.
So how can we write this in terms of signal and noise model:

$$y = f(x; a, b) + noise$$

Please note that this is an additive noise, and depending on the nature of the phenomena we may have other types of noise as well, such as multiplicative noise.

$$y_n = f(x_n; a, b) + \epsilon_n, \quad \epsilon \sim N(0, \sigma_n).$$

Now let's see how the model parameters vary in the solution space. This is for the above model where we have two model parameters.

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

# Define a function to compute the y-values for a straight line model at
given x-values.
def line_model(parameters, x_values):
    return parameters[0] * np.array(x_values) + parameters[1]

# Define a function to calculate the weighted absolute deviation between
data and model predictions.
def weighted_absolute_deviation(parameters, x, y, y_error):
    residuals = (y - line_model(parameters, x)) / y_error
    return np.sum(np.abs(residuals))

# Define a function to calculate the weighted squared deviation between
data and model predictions.
def weighted_squared_deviation(parameters, x, y, y_error):
    residuals = (y - line_model(parameters, x)) / y_error
    return np.sum(residuals**2)

# Create a grid of parameter values, centered on the true values.
a_grid = np.linspace(a_true - 5.0, a_true + 5.0, 256)
b_grid = np.linspace(b_true - 5.0, b_true + 5.0, 256)
a_grid, b_grid = np.meshgrid(a_grid, b_grid)
ab_grid = np.vstack((a_grid.ravel(), b_grid.ravel())).T

# Create a figure with two subplots for visualization.
```

```
fig, axes = plt.subplots(1, 2, figsize=(9, 5.1), sharex=True, sharey=True)

# Calculate and plot weighted absolute deviation and weighted squared
deviation.
for i, func in enumerate([weighted_absolute_deviation,
weighted_squared_deviation]):
    func_values = np.zeros(ab_grid.shape[0])
    for j, parameters in enumerate(ab_grid):
        func_values[j] = func(parameters, x, y, y_err)

    axes[i].pcolormesh(a_grid, b_grid, func_values.reshape(a_grid.shape),
shading='nearest',
                       cmap='Blues', vmin=func_values.min(),
vmax=func_values.min() + 256, rasterized=True)  # Arbitrary scale

    axes[i].set_xlabel('$a$')

    # Plot the true values as a reference point.
    axes[i].plot(a_true, b_true, marker='o', zorder=10, color='tab:red')
    axes[i].axis('tight')
    axes[i].set_title(func.__name__, fontsize=14)

axes[0].set_ylabel('$b$')

fig.tight_layout()

# Display the plot.
plt.show()
```

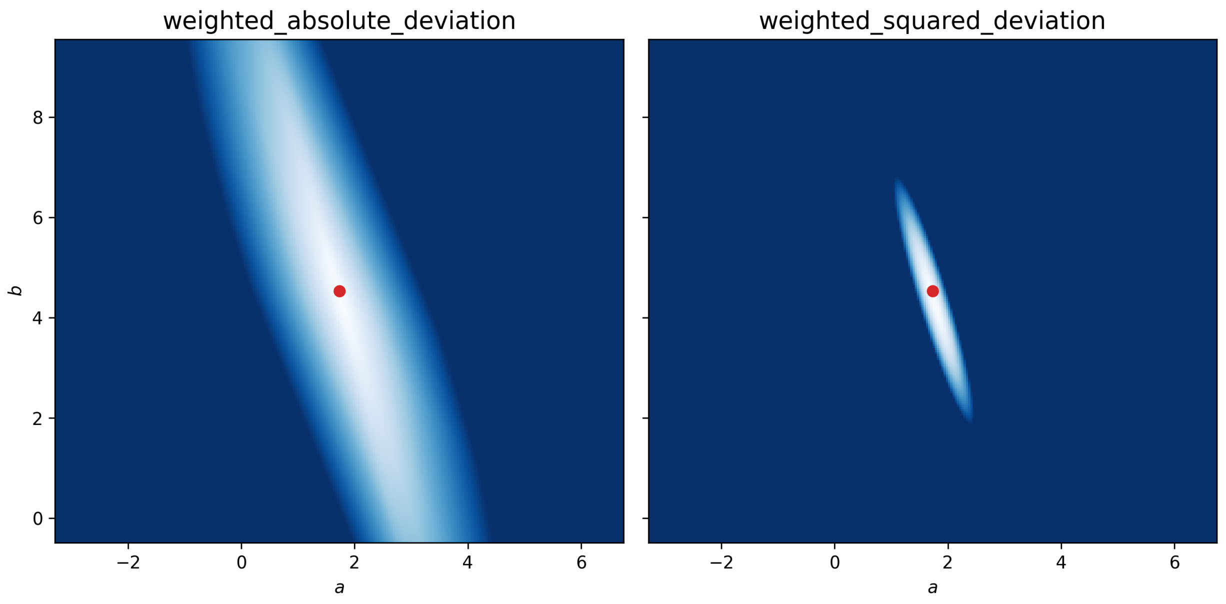

Figure 2.2

Let's create a numerical example for a simple linear regression problem and find the parameters 'a' and 'b' using the least squares approach. In this example, we'll generate some synthetic data and then use the least squares method to estimate the parameters.

In this example, we generate synthetic data with a linear relationship between x and y, with added noise. We then calculate the least squares estimates for 'a' and 'b' using the formulas for the slope and intercept based on the sample means and covariances. Finally, we plot the original data, the true regression line, and the least squares fit.

The estimated parameters 'a' and 'b' should be close to the true values 'a_t' and 'b_t', although they may not be exactly the same due to the added noise.
We calculate the least squares estimates for the slope 'a' and intercept 'b' using mathematical equations. Here are the equations for the least squares method:

Mean Values:
Calculate the mean of the independent variable x:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Calculate the mean of the independent variable y:

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

Calculate the mean of the product of 'x' and 'y':

$$\overline{xy} = \frac{1}{n}\sum_{i=1}^{n} x_i y_i$$

Calculate the mean of the squared values of 'x':

$$\overline{x^2} = \frac{1}{n}\sum_{i=1}^{n} {x_i}^2$$

Least Squares Estimates:
Calculate the slope 'a' using the formula:

$$a = \frac{\overline{xy} - \bar{x}.\bar{y}}{\overline{x^2} - \bar{x}^2}$$

Calculate the intercept 'b' using the formula:

$$b = \bar{y} - a.\bar{x}$$

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

# Generate synthetic data with a linear relationship: y = a * x + b +
noise
np.random.seed(0)
x = np.linspace(0, 10, 50)  # Independent variable
a_true = 2.5  # True slope
b_true = 1.0  # True intercept
noise = np.random.normal(0, 1, len(x))  # Additive noise
y = a_true * x + b_true + noise  # Dependent variable with noise

# Calculate the least squares estimates for 'a' and 'b'
x_mean = np.mean(x)
y_mean = np.mean(y)
xy_mean = np.mean(x * y)
x_squared_mean = np.mean(x**2)

a_est = (xy_mean - x_mean * y_mean) / (x_squared_mean - x_mean**2)
b_est = y_mean - a_est * x_mean

# Plot the original data and the regression line
plt.figure(figsize=(8, 6))
plt.scatter(x, y, label='Data with Noise')
plt.plot(x, a_true * x + b_true, color='red', label='True Regression
Line')
plt.plot(x, a_est * x + b_est, color='green', linestyle='--', label='Least
Squares Fit')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.title('Linear Regression using Least Squares')
plt.grid(True)

# Display the estimated parameters
print(f"Estimated a: {a_est:.3f}")
print(f"Estimated b: {b_est:.3f}")

plt.show()
```

```
Estimated a: 2.359
Estimated b: 1.846
```

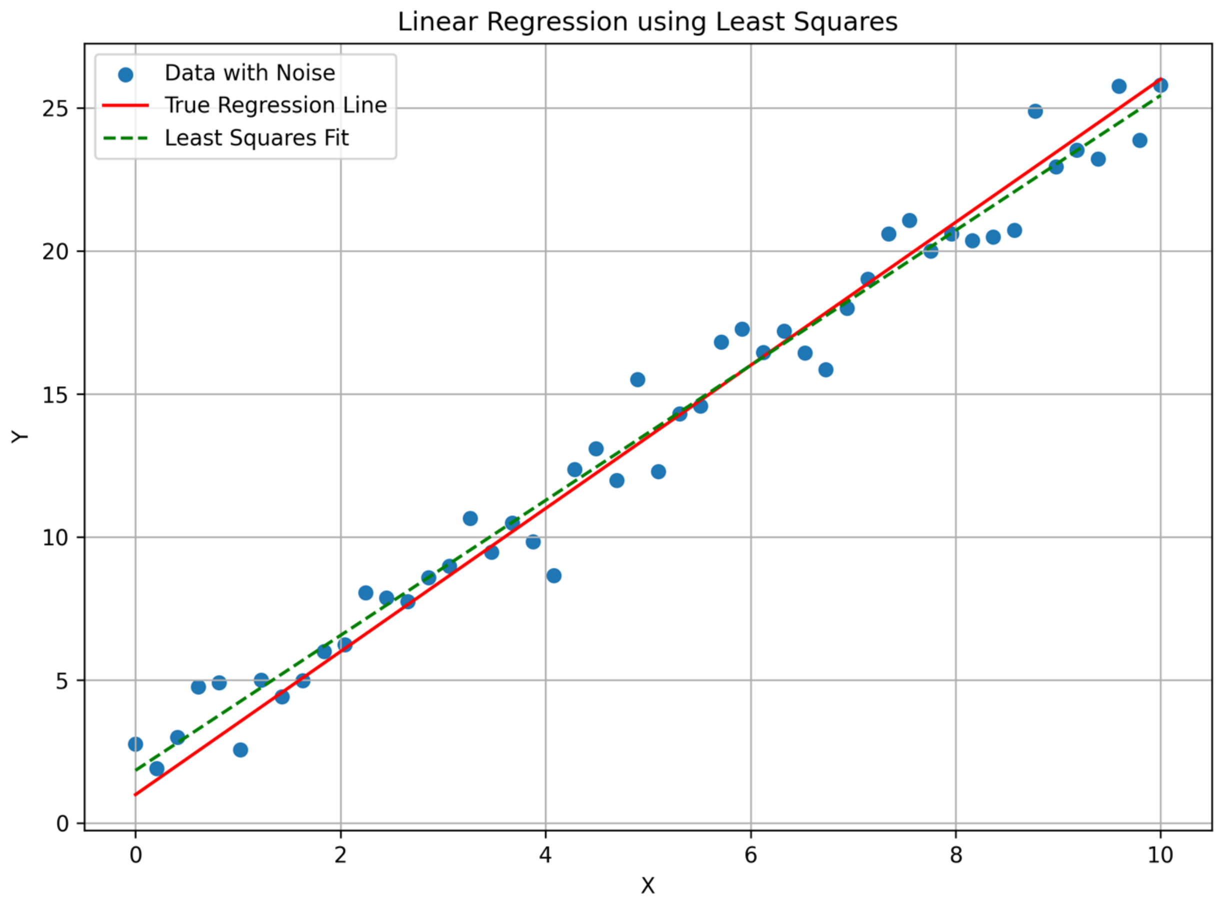

Figure 2.3

For the linear model in the form of $y = a.x + b$, where the a is the slope and b is the intercept, and x is the independent variable and y dependent variable, for each data point (x_i, y_i) the residual or error is calculated as $e_i = y_i - (a.x_i + b)$, and the goal of the linear regression is to minimize the sum of the squared residuals, which is often called the "objective function" or "cost function." It is defined as:

$$J(a, b) = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} \big(y_i - (a.x_i + b)\big)^2$$

To find the values of 'a' and 'b' that minimize $J(a, b)$, we take partial derivatives with respect to 'a' and 'b' and set them to zero:

Partial derivative with respect to 'a':

$$\frac{\partial J}{\partial a} = -2 \sum_{i=1}^{n} x_i \big(y_i - (a.x_i + b)\big) = 0$$

Partial derivative with respect to 'b':

$$\frac{\partial J}{\partial b} = -2\sum_{i=1}^{n}\big(y_i - (a.x_i + b)\big) = 0$$

Therefore, a and b are derived as follows:

$$a = \frac{\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \bar{y}}{\sum_{i=1}^{n} {x_i}^2 - \sum_{i=1}^{n} x_i \bar{x}} = \frac{\overline{xy} - \bar{x}.\bar{y}}{\overline{x^2} - \bar{x}^2}$$

$$b = \bar{y} - a.\bar{x}$$

```
import numpy as np

# Generate some sample data
np.random.seed(0)
x = np.linspace(0, 10, 50)  # Independent variable
a_true = 2.5  # True slope
b_true = 1.0  # True intercept
noise = np.random.normal(0, 1, len(x))  # Additive noise
y = a_true * x + b_true + noise  # Dependent variable with noise

# Define the cost function to be minimized
def cost_function(a, b):
    predictions = a * x + b
    residuals = y - predictions
    return np.sum(residuals**2)

# Use an optimization method to minimize the cost function
from scipy.optimize import minimize

initial_guess = [1.0, 1.0]  # Initial guess for a and b
result = minimize(lambda params: cost_function(params[0], params[1]),
initial_guess, method='L-BFGS-B')
a_est, b_est = result.x

# Display the estimated parameters
print(f"Estimated a: {a_est:.3f}")
print(f"Estimated b: {b_est:.3f}")
```

```
Estimated a: 2.359
Estimated b: 1.846
```

The example here demonstrates how to perform linear regression analysis on synthetic data representing the relationship between rainfall and river flow. The goal is to visualize the regression model and its associated uncertainty.
Data Generation:
  - Synthetic data for 30 days of rainfall and river flow is generated.
  - Rainfall values are generated randomly in the range [0, 50] mm.
  - The true linear relationship between rainfall and river flow, with some noise, is defined.
Regression Model Fitting:
  - A linear regression model is fitted to the data using scikit-learn's `LinearRegression` class.
  - The model is trained to find the best-fit line (regression line) that relates rainfall to river flow.
Uncertainty Visualization:
  - Uncertainty in the regression model is calculated based on the residuals (differences between observed data and predicted values).
  - A 95% confidence interval (uncertainty) is created, and the upper and lower confidence bands are shaded in pink to represent the uncertainty in the regression line.
Model Coefficients:
  - The coefficients of the linear regression model (slope and intercept) are displayed.
Outcome: The example provides a clear visualization of the linear regression model, showing how rainfall affects river flow. The pink shaded region illustrates the model's uncertainty, providing a 95% confidence interval around the regression line. This helps in understanding the reliability of the model's predictions and its sensitivity to the data.
The code can be adapted for real-world data, allowing you to analyze and visualize relationships between variables and quantify the uncertainty in the regression results.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Seed for reproducibility
np.random.seed(42)

# Generate synthetic data for 30 days of rainfall and river flow
n_days = 30
rainfall = np.random.uniform(0, 10, n_days)  # Rainfall data (in mm)
a_true = 1.5  # True slope
b_true = 3.75 # True intercept
river_flow = a_true * rainfall + b_true  # True relationship

# Introduce random Gaussian noise with varying standard deviation for
river flow
flow_err = np.random.uniform(0, 3, n_days)
```

```
river_flow = river_flow + np.random.normal(0, flow_err)

# Reshape the data for regression
rainfall = rainfall.reshape(-1, 1)
river_flow = river_flow.reshape(-1, 1)

# Fit a linear regression model
regression_model = LinearRegression()
regression_model.fit(rainfall, river_flow)

# Predict river flow values using the regression model
predicted_flow = regression_model.predict(rainfall)

# Plot the observed data and the regression line
plt.scatter(rainfall, river_flow, label='Observed Data', color='blue')
plt.plot(rainfall, predicted_flow, label='Regression Line', color='red')
plt.xlabel('Rainfall (mm)')
plt.ylabel('River Flow (cubic meters/second)')
plt.title('Linear Regression of Rainfall and River Flow')
plt.legend()

# Calculate and visualize the uncertainty in the regression model
x_test = np.linspace(0, 10, 100)  # Generate test points
x_test = x_test.reshape(-1, 1)  # Reshape to a 2D array
y_pred = regression_model.predict(x_test)  # Predictions for test points
sigma = np.std(river_flow - predicted_flow)  # Standard deviation of
residuals
y_upper = y_pred + 1.96 * sigma  # Upper confidence band (95% confidence
interval)
y_lower = y_pred - 1.96 * sigma  # Lower confidence band (95% confidence
interval)

plt.fill_between(x_test[:,0], y_lower[:,0], y_upper[:,0], color='pink',
alpha=0.5, label='Uncertainty (95% CI)')

# Display the model coefficients
print('Model Coefficients:')
print('Slope (a):', regression_model.coef_[0][0])
print('Intercept (b):', regression_model.intercept_[0])

plt.tight_layout()
plt.savefig('/content/drive/MyDrive/PSU/ColabNotebooks/Example2_3.png',
dpi=300)
plt.show()
```

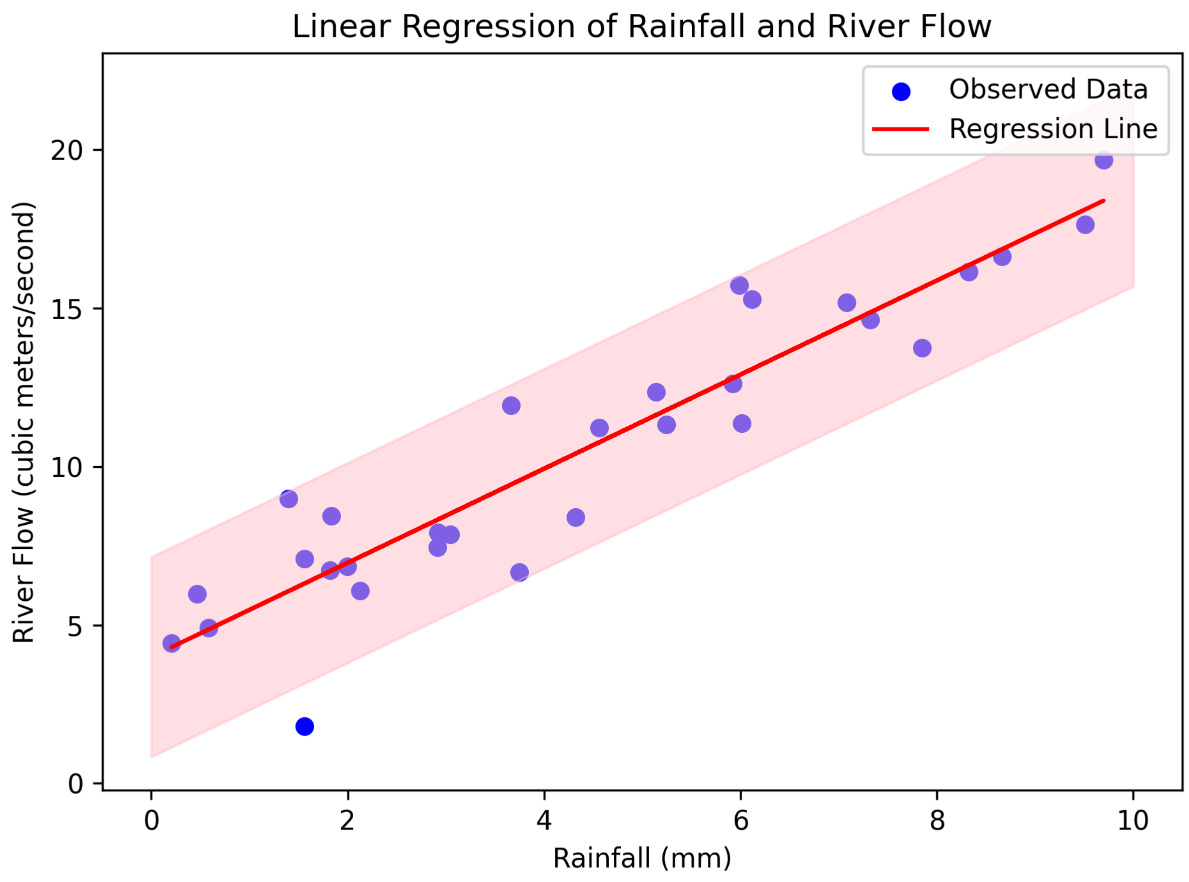

Figure 2.4

Please note that "1.96" is a constant multiplier representing the critical value for a 95% confidence interval in a standard normal (Gaussian) distribution. When dealing with a 95% confidence interval, 1.96 is often used to determine the margin of error for estimating the upper and lower bounds of the confidence interval. The upper confidence band represents the upper boundary of the confidence interval for the regression line. It indicates the range within which we are 95% confident that the true relationship between rainfall and river flow lies. This means that, statistically, we expect the actual data points to fall within this band with a 95% probability.

2.1. Least-squares fitting

The assumptions we made earlier suggest that for each data point (x_n, y_n), there exists a true underlying true y_n. However, due to limitations in our observation process, we cannot directly observe this truth. Instead, we acknowledge that the values we can observe will exhibit Gaussian (Normal) distribution characteristics centered around the true value. In mathematical terms:

$$p(y|y_{\mathrm{true}}) = N(y|y_{\mathrm{true}}, \sigma^2) = (2\pi\sigma^2)^{-1/2}\exp\left(-\frac{1}{2}\frac{(y-y_{\mathrm{true}})^2}{\sigma^2}\right)$$

This represents the likelihood of observing a specific y value, considering the true y. It provides the probabilistic description of the noise model.
So what if we have two data points y_1 and y_2? We need to calculate the joint probability.

$$p(y_1, y_2|y_{1,\mathrm{true}}, \sigma_1, y_{2,\mathrm{true}}, \sigma_2)$$

assuming that the data points are independent, that means the random point in one point does not affect the random point in the other point. Therefore, the joint probability is calculated using the product rule.

$$p(\{y_n\}|\{y_{n,\mathrm{true}}\}\{\sigma_n\}) = \prod_n^N p(y_n|y_{n,\mathrm{true}}, \sigma_n)$$

This is not usually archivable as we do not know the $y_{n,\mathrm{true}}$. Assuming that the $y_{n,\mathrm{true}}$ is represented by a linear regression model, $ax_n + b$, the above relationship can be expressed as:

$$p(\{y_n\}|a, b, \{x_n\}, \{\sigma_n\}) = \prod_n^N p(y_n|a, b, x_n, \sigma_n)$$

This probability distribution, which represents the data in relation to all pertinent parameters, is known as the likelihood. The product on the right-hand side of the likelihood involves products of exponentials (specifically Gaussians), which can be cumbersome to handle. However, maximizing the likelihood is essentially the same as maximizing the log-likelihood, enabling us to simplify the expression by taking the natural logarithm of both sides:

$$\begin{aligned}
\mathrm{Log}(\{y_n\}|a, b, \{x_n\}, \{\sigma_n\}) &= \sum_n^N \mathrm{Log}[p(y_n|a, b, x_n, \sigma_n)] \\
&= \sum_n^N \mathrm{Log}\left[(2\pi{\sigma_n}^2)^{-\frac{1}{2}}\exp\left(-\frac{1}{2}\frac{\left(y_n-(ax_n+b)\right)^2}{{\sigma_n}^2}\right)\right] \\
&= -\frac{N}{2}\mathrm{Log}(2\pi) - \frac{1}{2}\sum_n^N\left[\left(\frac{\left(y_n-(ax_n+b)\right)^2}{{\sigma_n}^2} + \mathrm{Log}({\sigma_n}^2)\right)\right] \\
&= -\frac{1}{2}\sum_n^N\left[\frac{\left(y_n-(ax_n+b)\right)^2}{{\sigma_n}^2}\right] + K == -\frac{1}{2}\sum_n^N \chi_n^2 + K
\end{aligned}$$

If we maximize this cost function, we can estimate the best values of a and b.

2.3 Homework #2 (10 points)

1- For a quadratic linear regression model $y = ax^2 + bx + c$, analytically find a, b and c by minimizing the residual cost function $J(a, b, c)$. ***(3 points).***

2- Let's assume $y = 1.5x^2 + 2x + 1 + \in_n$. where $\in_n \sim N(0,1)$. $\in$ is an additive noise. Write a Python script to calculate true values of y for x, which is an array of 50 equally spaced values between 0 and 10. Use np.random.seed(0). a) Define the cost function for the quadratic regression and minimize it to find the best estimate of a, b and c. ***(4 points)***. And then b) use the Negative Log-Likelihood function as the cost function and minimize it to estimate a, b and c. Compare the results of a) and b). ***(3 points)***

Bonus- Find/design an example with dataset in your area/research field that can be represented by Poisson regression model. Then derive the Likelihood Function and maximize it (or minimize Negative Log-Likelihood) to estimate the model parameters. ***(2 points)***