Demystifying Linear Regression and Introduction to Error Metrics

Minor In AI, IIT Ropar

3rd April, 2025

Welcome, future AI enthusiasts! This module is your friendly guide to understanding linear regression, a fundamental building block in the world of Artificial Intelligence. We'll explore this concept step-by-step, using simple examples and Python code, so you can build a solid foundation for your AI journey.

1 Predicting Land Prices - A Real-World Example

Imagine you're interested in buying a piece of land. You want to know how the size of the land affects its price. You start collecting data: looking at land plots and their corresponding prices.

Land Size (Square Feet)	Price (USD)
1000	50,000
1500	75,000
2000	100,000
2500	125,000

You notice a trend: larger plots of land generally cost more. But how can you *predict* the price of a plot you haven't seen before? That's where linear regression comes in. It helps us find a relationship between the land size (our input, often called 'X') and the price (our output, often called 'Y'). Once we have a formula, we can estimate the price of any plot of land, even if we don't have historical data on similar plots!

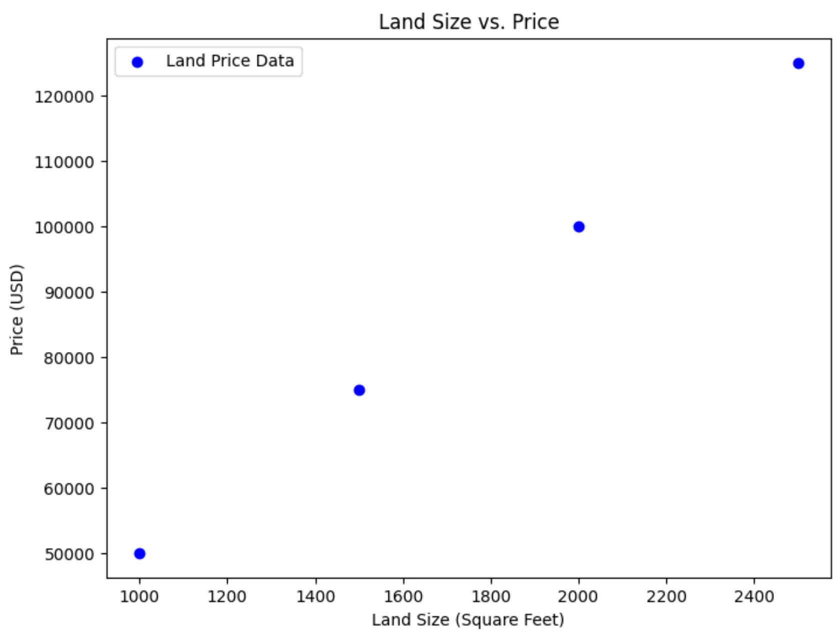

Figure 1: A scatter plot showing land size vs. price, illustrating a general upward trend.

In this module, we will learn how to find the relationship between "X" and "Y" and then how to use that to predict Y, given a new value of X.

2 Back to Basics: The Equation of a Line

Remember the equation of a straight line from high school math? It's the foundation of linear regression:

$$\mathbf{Y = MX + B}$$

Where:

- **Y** is the dependent variable (what we want to predict - the price in our example).
- **X** is the independent variable (what we use to make the prediction - the land size).
- **M** is the slope of the line (how much Y changes for every unit change in X).
- **B** is the y-intercept (the value of Y when X is zero).

Linear regression helps us find the best values for M (slope) and B (intercept) that represent the relationship between X and Y in our data.

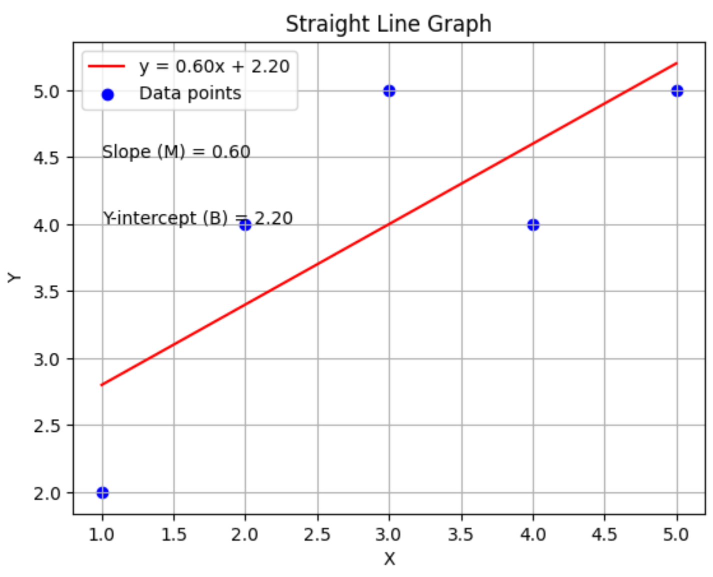

Figure 2: A graph showing a straight line with labels for X, Y, slope (M), and y-intercept (B).

3 Calculating the Slope (M) and Intercept (B)

There are multiple ways to calculate the slope (M) and intercept (B). Here, we'll explore a common method. First, we need to calculate the mean (average) of X and Y values in our dataset.

Method 1: Pen and Paper Method (High School Style)

In the old days in high school we used to establish the relationship between X and Y by drawing lines through the scatter plots of X and Y values.

Method 2: The Formulaic Approach

Here's the more statistical approach that you would have also seen in your high school and college days.

1. **Calculate the Means:** Find the average of all X values ($\bar{x}$) and all Y values ($\bar{y}$).
2. **Calculate the Slope (M):**

$$M = \frac{\sum[(X_i - \bar{x}) \cdot (Y_i - \bar{y})]}{\sum[(X_i - \bar{x})^2]}$$

   Where:

   - X_i is the i-th value of X.
   - Y_i is the i-th value of Y.
   - $\bar{x}$ is the mean of X.
   - $\bar{y}$ is the mean of Y.
   - $\sum$ means, Summation

3. **Calculate the Intercept (B):**

$$B = \bar{y} - M \cdot \bar{x}$$

4 Implementing Linear Regression with Python

Let's translate this into Python code. We will use the same formula to calculate the slope and intercept, and then use that to predict values of Y for our values of X.

```python
# Given data points
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 4, 6]

# Compute mean of x and y
x_mean = sum(x) / len(x)
y_mean = sum(y) / len(y)

# Compute slope (m)
numerator = sum((xi - x_mean) * (yi - y_mean) for xi, yi in zip(x, y))
denominator = sum((xi - x_mean) ** 2 for xi in x)
m = numerator / denominator

# Compute intercept (c)
c = y_mean - m * x_mean

# Print results
print(f"m = {m}")
print(f"c = {c}")
```

Listing 1: Manual implementation of linear regression

This code calculates the slope (`m`) and intercept (`c`) based on the formulas we discussed. It then prints these values, which define our linear regression model.

Using scikit-learn:

Let's now check out how to do that using scikit-learn.

```python
import numpy as np
from sklearn.linear_model import LinearRegression

# Sample data
X = np.array([1, 2, 3, 4, 5]).reshape((-1, 1))
y = np.array([2, 3, 5, 4, 6])

# Create and fit the model
model = LinearRegression()
model.fit(X, y)

# Get the coefficients and intercept
r_sq = model.score(X, y)
print(f"coefficient of determination: {r_sq}")
print(f"intercept: {model.intercept_}")
print(f"coefficients: {model.coef_}")
```

Listing 2: Linear regression using scikit-learn

5 Predicting Values and Visualizing the Results

Now that we have our slope and intercept, we can use the linear regression equation to predict the value of Y for any given X.

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

```python

# Given data points
x = np.array([1, 2, 3, 4, 5])
y_actual = np.array([2, 3, 5, 4, 6])

# Regression equation: y = 0.9x + 1.3
m = 0.9
c = 1.3
y_predicted = m * x + c  # Calculate predicted y values

# Calculate errors (absolute differences)
errors = abs(y_actual - y_predicted)

# Plot actual points
plt.scatter(x, y_actual, color='blue', label="Actual Points")

# Plot regression line
plt.plot(x, y_predicted, color='red', label="Regression Line (y=0.9x
    +1.3)")

# Draw vertical lines for errors
for i in range(len(x)):
    plt.plot([x[i], x[i]], [y_actual[i], y_predicted[i]], 'g--', alpha
    =0.6)

# Labels and legend
plt.xlabel("x")
plt.ylabel("y")
plt.title("Linear Regression with Error Visualization")
plt.legend()
plt.grid(True)
plt.show()
```

Listing 3: Visualizing linear regression results

This code calculates the predicted Y values (`y_predicted`) using the calculated slope (`m`) and intercept (`c`). It then plots the original data points, the regression line, and visualizes the error (the difference between the actual and predicted values).

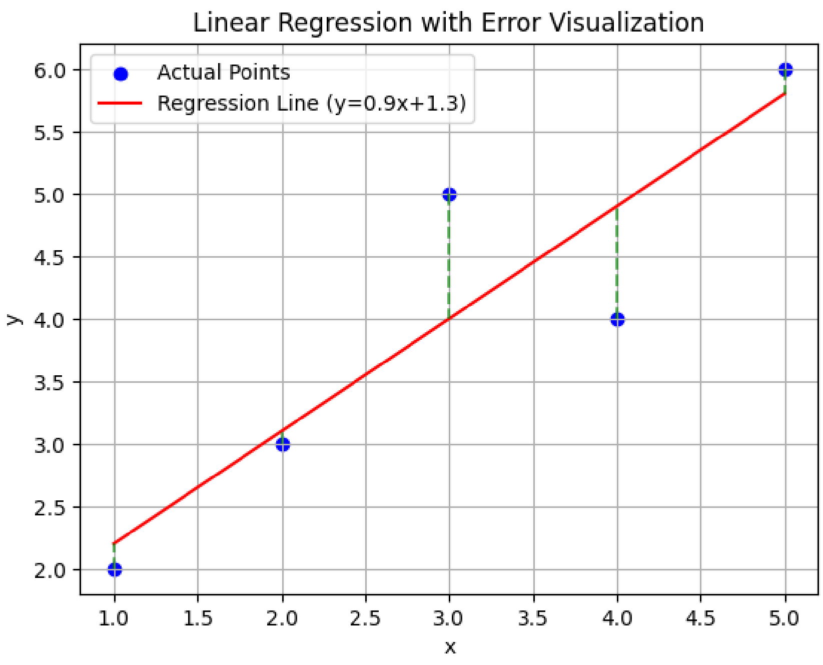

Figure 3: A scatter plot showing actual data points, the regression line, and error lines.

6 Evaluating the Model: Error Metrics

How do we know if our linear regression model is doing a good job? We need to measure the error, the difference between the predicted values and the actual values. There are many ways to do that:

1. **Mean Absolute Error (MAE):** This calculates the average of the absolute differences between predicted and actual values. This tells us on average, how far off our predictions were with our actual values. This is calculated by finding the average of the summation of all absolute differences between predicted values and actual values.

```python
from sklearn.metrics import mean_absolute_error, mean_squared_error
import numpy as np

# Example data
y_actual = np.array([2, 4, 6, 8])
y_predicted = np.array([2.2, 3.8, 5.9, 8.1])

# Calculate MAE and MSE
mae = mean_absolute_error(y_actual, y_predicted)

print(f"Mean Absolute Error (MAE): {mae:.2f}")
```

Listing 4: Calculating Mean Absolute Error

2. **Mean Squared Error (MSE):** This calculates the average of the *squared* differences between predicted and actual values. Squaring the differences penalizes larger errors more heavily than smaller errors. This means that if we are getting large deviations, the mean squared error would explain the errors more accurately.

```python
from sklearn.metrics import mean_absolute_error , mean_squared_error
import numpy as np

# Example data
y_actual = np.array([2, 4, 6, 8])
y_predicted = np.array([2.2, 3.8, 5.9, 8.1])

mse = mean_squared_error(y_actual , y_predicted)

print(f"Mean Squared Error (MSE): {mse:.2f}")
```

Listing 5: Calculating Mean Squared Error

3. **R-squared (R^2):** The R-squared (R^2) metric is used when we want to bring correlation into the picture. R-squared is based on the variance of the data, and gives us an idea about how accurately our line is fitting the data. We will talk more about this in the next chapter.

7 Limitations of Linear Regression: When a Line Isn't Enough

Linear regression works well when there is a linear relationship between X and Y. But what if the relationship is not linear? What if the relationship between age and how slow a person's reaction time is not linear, but rather has a U-shaped form? In those cases, linear regression isn't the best fit.

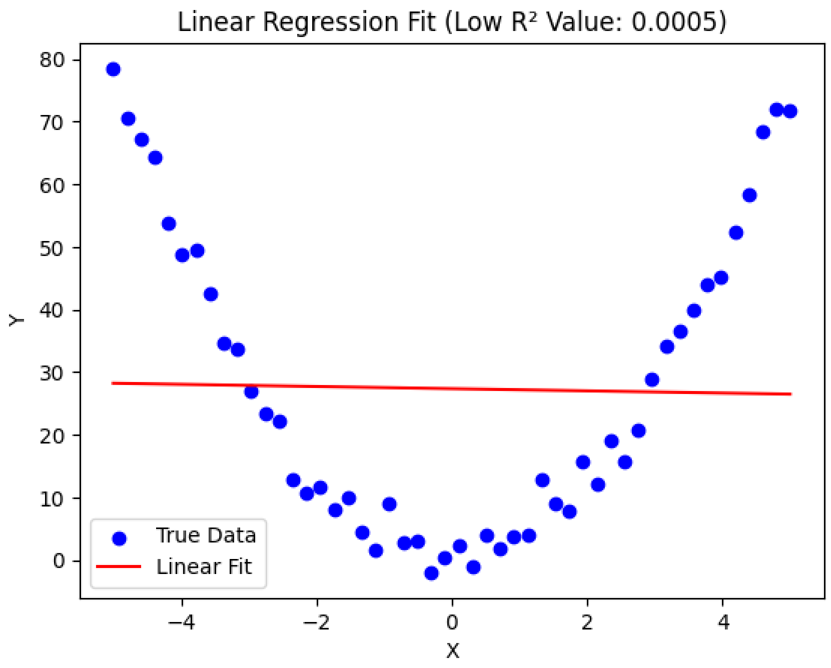

Figure 4: A scatter plot showing data points with a non-linear relationship, with a linear regression line that doesn't fit well.

In such scenarios, we need to consider other models, such as polynomial regression, which can capture more complex relationships.

8 Beyond Linear: An Introduction to Polynomial Regression

Polynomial regression allows us to fit curves to our data by using polynomial equations:

$$\mathbf{Y} = \mathbf{A}_0 + \mathbf{A}_1\mathbf{X} + \mathbf{A}_2\mathbf{X}^2 + \mathbf{A}_3\mathbf{X}^3 + \ldots$$

The degree of the polynomial determines the complexity of the curve. A degree of 2 (quadratic) creates a parabola, a degree of 3 (cubic) creates an S-shaped curve, and so on. This helps us fit curves to the data.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

# Real-time example: Age vs. Reaction Time
age = np.array([10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70]).
    reshape(-1, 1)
reaction_time = np.array([500, 480, 450, 430, 420, 415, 420, 430, 450,
    480, 520, 570, 630])  # Reaction time in ms

# Fit a Linear Regression Model
linear_model = LinearRegression()
linear_model.fit(age, reaction_time)
reaction_time_pred_linear = linear_model.predict(age)
linear_r2 = r2_score(reaction_time, reaction_time_pred_linear)

# Transform features for Polynomial Regression (Degree = 2)
poly = PolynomialFeatures(degree=2)
age_poly = poly.fit_transform(age)
polynomial_model = LinearRegression()
polynomial_model.fit(age_poly, reaction_time)
reaction_time_pred_poly = polynomial_model.predict(age_poly)
poly_r2 = r2_score(reaction_time, reaction_time_pred_poly)

# Plot Linear and Polynomial Regression
plt.scatter(age, reaction_time, label='True Data', color='blue')
plt.plot(age, reaction_time_pred_linear, label=f'Linear Fit (R^2: {
    linear_r2:.4f})', color='red')
plt.plot(age, reaction_time_pred_poly, label=f'Polynomial Fit (R^2: {
    poly_r2:.4f})', color='green')
plt.title('Linear vs. Polynomial Regression')
plt.xlabel('Age (years)')
plt.ylabel('Reaction Time (ms)')
plt.legend()
plt.show()

print(f"Linear R^2 Value: {linear_r2:.4f}")
print(f"Polynomial R^2 Value: {poly_r2:.4f}")
```

Listing 6: Polynomial regression implementation

As we improve the model, we also improve our model's accuracy when predicting values for data that are out of the range of data that it was trained on.

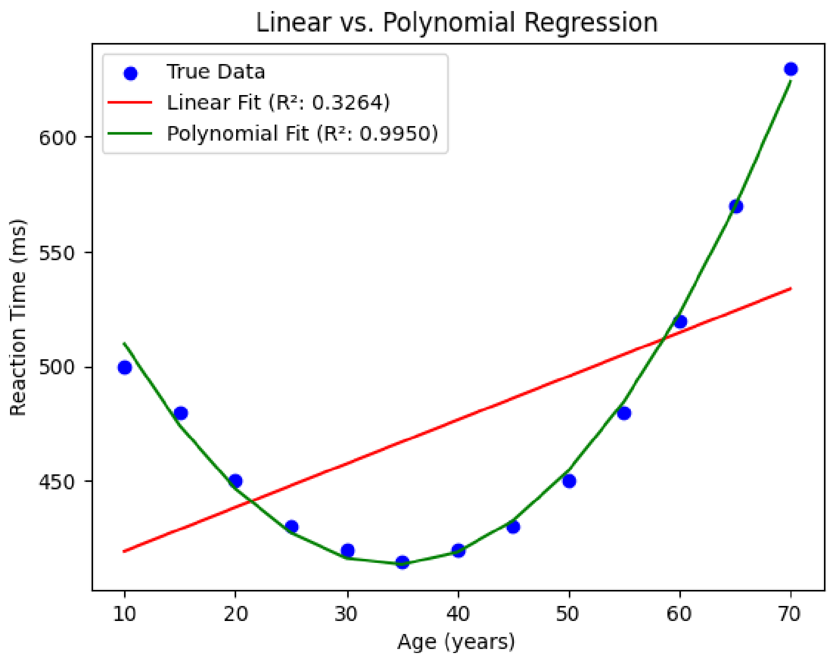

Figure 5: A scatter plot showing non-linear data, with a polynomial regression curve that fits the data much better.

Congratulations! You've taken your first steps into the world of linear regression. You've learned how to calculate the slope and intercept, implement the model in Python, evaluate its performance, and understand its limitations. Remember, this is just the beginning. As you delve deeper into AI, you'll encounter more sophisticated models and techniques, but the foundational knowledge you've gained here will serve you well. Happy learning!