Multivariate regression

What's the Deal with Multivariate Regression?

Imagine you're hosting a potluck dinner, and you want to predict how many guests will show up. You think it depends on:

1. **The number of dishes you're offering** 🍲
2. **How many text reminders you send out** 📲
3. **The weather that day** ☀️🌧️

Now, you don't just want to guess—you're a data scientist at heart! 🧠 So, you decide to analyze the relationship between these factors (**independent variables**) and the number of guests (**dependent variable**). This is where **multivariate regression** steps in.

The Magic Formula

Multivariate regression says:

$$\text{Guests} = \beta0 + \beta1 \times \text{Dishes} + \beta2 \times \text{Reminders} + \beta3 \times \text{Weather} + \epsilon$$

Here's the breakdown:

- $\beta0$: The base level, like if you did nothing, how many friends would still show up because they adore you. 🥰
- $\beta1, \beta2, \beta3$: The "weights" or influence of each factor. For example, if $\beta1$ is 2, then for every extra dish, 2 more people are likely to come.
- ϵ : The random "life happens" factor—someone cancels last minute, or your dog steals a dish. 🐶

Why Is This Cool?

It lets you go beyond gut feelings. It quantifies how much each factor matters, so you can plan better next time!

Example:
You find that adding more dishes has a big impact, but sending reminders doesn't matter much, and bad weather scares everyone away. So next time, focus on making extra food and pray for sunshine. 🌞

How Do You Do This in Real Life?

Let's say you're not just a potluck planner but a budding entrepreneur. You're launching a lemonade stand, and sales depend on:

- Temperature (the hotter it is, the more people want lemonade).
- Price per cup (higher prices might scare people off).
- Marketing efforts (flyers, Instagram posts, etc.).

Python Code

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Your dataset: past sales records
data = pd.DataFrame({
    'Temperature': [30, 25, 35, 20, 32],
    'Price': [10, 12, 8, 15, 9],
    'Marketing': [500, 300, 800, 200, 600],
    'Sales': [100, 80, 150, 50, 120]
})

# Independent variables (X) and dependent variable (y)
X = data[['Temperature', 'Price', 'Marketing']]
y = data['Sales']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Build the model
model = LinearRegression()
model.fit(X_train, y_train)

# Check coefficients
coefficients = pd.DataFrame({'Feature': X.columns, 'Coefficient': model.coef_})
print(coefficients)

# Predict sales
predictions = model.predict(X_test)
print("Predicted Sales:", predictions)
```

Let's Get Even More Relatable

You're on a **fitness journey**, and your trainer says your weight loss depends on:

1. **Workout hours per week** 🏋️
2. **Calories consumed daily** 🍕
3. **Sleep quality** 🛌

Now, instead of just blaming that one cheat meal, you can mathematically figure out what's working and what's not. Maybe more sleep has a bigger impact than an extra hour at the gym—who knew?

Why It's a Big Deal

- **Real-world decisions:** Businesses use it to figure out what drives sales, customer satisfaction, or even employee performance.
- **Life hacks:** It's like having a crystal ball, but data-driven.

The Challenges (But Hey, We Got This)

- **Multicollinearity:** If two predictors are too similar (e.g., workout hours and step count), the model can get confused.
- **Garbage in, garbage out:** If your data isn't great (missing values, outliers), neither will your predictions be.
- **Overfitting:** Adding too many factors might make the model fit *too* well to your data, losing generality.

Final Fun Thought

Think of multivariate regression as hosting a party where every guest brings something to the table. Some guests (independent variables) are MVPs, while others barely contribute. Your job is to figure out who's who and plan your next party—or prediction—accordingly. 🎉

Logistic Regression

Let's talk **logistic regression**—the sassy cousin of linear regression who doesn't do numbers like 42.7 but instead likes to say, "Yes or no? True or false? Is this a cat or a dog?" 🐱🐶

What is Logistic Regression, Really?

Imagine you're organizing a weekend hike, and you want to predict who's going to show up. The factors:

- **Fitness level** 🏋️
- **Weather forecast** ☀️🌧️
- **Distance to the trail** 🗺️

But unlike before, this isn't about *how many* people will come—it's about whether a person **will show up or not** (yes or no). Logistic regression is perfect for this because it predicts **probabilities** and translates them into categories (binary: 0 or 1).

The Math-y Part (But Fun)

Logistic regression doesn't say, "Oh, you're 75% likely to come, so let's use 75." Instead, it uses a magical transformation called the **sigmoid function** to squish probabilities between 0 and 1.

The formula looks like this:

$$P(Y=1) = \frac{1}{1+e^{-z}}$$

Where:

- Z = β0 + β1.X1 + β2.X2 + ⋯
- P(Y=1) : Probability of an event happening (e.g., someone showing up).

If P > 0.5, logistic regression says, "Yep, they're coming!" Otherwise, it's a no.

Let's Make It Relatable

Scenario 1: Will Your Dog Bark at the Mailman? 🐕📪

Factors:

- Dog's energy level
- Time of day (mailman usually comes in the morning)
- Mailman's speed (a fast one might scare your pup).

You collect some data and train a logistic regression model. Now, every morning, you can predict whether your dog will start the "bark orchestra" or just chill.

How It Works in Python (with Pizza Delivery 🍕)

Let's predict whether a pizza will arrive on time based on:

- **Distance to your house** (in km)
- **Traffic conditions** (light, medium, heavy).
- **Weather** (sunny, rainy).

Python Code

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix

# Pizza delivery data
data = pd.DataFrame({
    'Distance': [2, 5, 1, 7, 3],
    'Traffic': [1, 3, 1, 2, 2],  # 1 = light, 2 = medium, 3 = heavy
    'Weather': [1, 0, 1, 0, 1],  # 1 = sunny, 0 = rainy
    'On_Time': [1, 0, 1, 0, 1]  # 1 = on time, 0 = late
})

# Independent variables (X) and target (y)
X = data[['Distance', 'Traffic', 'Weather']]
y = data['On_Time']

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Build the model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluation
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

The model will spit out something like:

- Probability = 0.85 → "Pizza is on time!"
- Probability = 0.35 → "It's late; grab a snack while you wait."

Why Is Logistic Regression Awesome?

1. **Yes-No Problems? It's Got You!**
    - Will your boss approve the budget? ✅ ❌
    - Is this email spam? 📧 🚫
    - Does this selfie have a dog or a cat? 🐶 🐱

2. **Probability Magic:**
   Logistic regression doesn't just say "yes" or "no"; it also gives you **confidence**. Like, "I'm 90% sure this email is spam."
3. **Straightforward:**
   It's simple to understand and interpret. No need to be a math wizard. 🧙

Let's Add a Fun Twist

Scenario 2: Should You Ask Your Crush Out? 💌

Factors:

- How many times they've smiled at you this week 😊
- Whether they laughed at your jokes 😂
- Their zodiac sign compatibility with yours ♌

You train a logistic regression model and get a probability:

- 0.92 → "Go for it! They're into you."
- 0.18 → "Maybe next time, champ."

Challenges (But Keep It Cool)

1. **Non-linearity:** If the relationship between factors and outcomes is super curvy, logistic regression might struggle.
2. **Overfitting:** Adding too many irrelevant factors can make it messy.
3. **Multicollinearity:** If your factors are super similar (like smile counts and eye contact), the model gets confused.

The Bottom Line

Logistic regression is like a decision-making buddy for life's yes-or-no questions. It's perfect when you're not looking for a number but a category, like "Late or On Time?" or "Love or Just Friends?"

Functions

1. Features (X): The Ingredients

Think of features as the raw materials of a prediction recipe.
For example:

- In a cookie recipe: Flour, sugar, chocolate chips 🍫.
- In logistic regression: Hours studied, sleep hours, caffeine intake ☕.

Each feature adds its own unique flavor to the outcome, but not all ingredients are equally important!

2. Weights (W): The Importance

Weights are like the importance you give to each feature:

- **High weight:** Chocolate chips—because cookies without them are sad.
- **Low weight:** The shape of the sugar crystals—nobody notices.
- **Zero weight:** Raisins—because some of us pretend they don't exist. 😅

Logistic regression adjusts these weights dynamically to figure out what really matters.

3. Net Input (z): The Mixing Bowl

The net input is where everything comes together. It's like throwing your ingredients into a bowl and calculating the total "cookie potential." 🍪

The formula:

$$Z = W1.X1 + W2.X2 + \cdots + Wn.Xn + b$$

Where:

- W: Weights
- X: Features
- b: Bias (like a secret pinch of salt that brings everything together).

Example: Predicting if you'll pass an exam:

- Hours studied ($X1=10$, $W1 = 0.5$).
- Sleep hours ($X2 = 6$, $W2 = 0.2$).
- Bias ($b = 1$).

$$Z = (0.5 \times 10) + (0.2 \times 6) + 1 = 6.2$$

4. Activation Function: The Magic Oven (Sigmoid)

Now we take z and bake it with the sigmoid function to turn it into a probability:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

The sigmoid squishes any number into a range between **0 and 1**, making it digestible as a probability.

Example:

- If $z=6.2$, then $\sigma(6.2) \approx 0.998$

  This means a 99.8% chance of passing.

- If $z=-1.5$, then $\sigma(-1.5) \approx 0.18$
  This means an 18% chance—probably not happening.

5. Predicted Label: The Taste Test

Now it's time to decide—pass or fail? Yes or no?

- If $\sigma(z) \geq 0.5$: **Class 1 (e.g., Pass, Cookie is Delicious)**.
- If $\sigma(z) < 0.5$: **Class 0 (e.g., Fail, Cookie Tastes Meh)**.

Think of this like a taste test:

- **Good taste?** 🍪 Keep the recipe.
- **Bad taste?** 🤢 Fix it.

6. Cost Function (J): The Kitchen Critic

The cost function measures how bad your predictions are. It's like a food critic rating your cookies.

The formula for logistic regression uses log loss:

$$J = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(y_i^{\wedge}) + (1-y_i)\log(1-y_i^{\wedge})\right]$$

- y : Actual label (e.g., Pass/Fail).
- y^ : Predicted probability (e.g., 0.9).

Example:

- If you predict a 90% chance someone will pass but they fail, the cost is high—your cookies flopped.

7. Error (e): The Difference

Error is the difference between what you thought would happen and what actually happened.

$$e=y-y\wedge$$

Example:

- If you thought your cookie was a 10/10 but the critic says it's a 6/10, your error is **4**.

8. Update Function: Fix the Recipe

The weights (W) get updated using gradient descent:

$$W_{\text{new}} = W_{\text{old}} - \eta \frac{\partial J}{\partial W}$$

Where:

- η: Learning rate (how quickly we adjust the recipe).
- $\partial W/\partial J$: Gradient (the direction to improve).

If the cookies are too salty, you reduce salt. Too sweet? Cut back on sugar. Logistic regression tweaks the weights until predictions are just right.

Relatable Analogy: Baking Cookies 🍪

Here's how logistic regression maps to baking cookies:

1. **Features (X):** Flour, sugar, and chocolate chips.
2. **Weights (W):** How much of each ingredient to use.
3. **Net Input (z):** Mixing everything in the bowl.
4. **Sigmoid:** Baking—turning raw dough into actual cookies.
5. **Predicted Label:** Taste test—delicious or not?
6. **Error:** If it's bad, figure out what went wrong.
7. **Cost Function:** The critic's score—higher if your cookies are bad.
8. **Update Function:** Tweak the recipe and bake again.

Conclusion

Logistic regression is like perfecting a recipe. You start with raw ingredients, mix them, bake them, and refine the recipe based on feedback. With every iteration, you get closer to the perfect cookie—or in this case, the perfect prediction! 🍪✨

Gradient Descent

Gradient Descent is an optimization algorithm used to minimize the cost function in machine learning and deep learning. Its goal is to iteratively adjust the parameters (weights and biases) of a model to minimize the difference between predicted and actual values.

1. The Core Idea

Gradient Descent finds the minimum of a function (e.g., the cost function) by taking steps in the direction of the steepest descent (negative gradient).

2. Key Components

- **Cost Function (J):** Measures how far off the model's predictions are.
- **Parameters (W):** Weights and biases that need to be optimized.
- **Learning Rate (η):** Determines the size of each step in the descent.
- **Gradient ($\partial J/\partial W$):** Direction and slope of the cost function at a given point.

3. The Gradient Descent Formula

For each parameter W, update it as:

$$W_{\text{new}} = W_{\text{old}} - \eta \cdot \frac{\partial J}{\partial W}$$

4. Step-by-Step Process

1. **Initialize Parameters:** Start with random values for W.
2. **Compute Predictions:** Use the current W to make predictions (y^)
3. **Calculate the Cost Function:** Find the error between predictions (y^) and actual values (y).

4. **Find the Gradient:** Compute the partial derivative (∂J/∂W) of the cost function w.r.t. each parameter.
5. **Update Parameters:** Adjust W using the gradient and learning rate.
6. **Repeat Until Convergence:** Iterate until the cost function is minimized.

5. Example: Minimizing the Cost Function

Imagine we're trying to fit a line to data points ($y = mx + c$).

- **Data:**

$$x = [1, 2, 3], \quad y = [2, 4, 6]$$

- **Model:**

$$\hat{y} = W \cdot x + b$$

Here W is the weight (slope), and b is the bias (intercept).

Initial Parameters:

$$W = 0, \quad b = 0, \quad \eta = 0.1$$

Iteration 1:

1. **Predictions:**

$$\hat{y} = W \cdot x + b = 0 \cdot [1, 2, 3] + 0 = [0, 0, 0]$$

2. **Cost Function (Mean Squared Error):**

$$J = \frac{1}{n} \sum (y - \hat{y})^2 = \frac{1}{3}[(2-0)^2 + (4-0)^2 + (6-0)^2] = 18.67$$

3. **Gradient**: Partial derivatives w.r.t W and b:

$$\frac{\partial J}{\partial W} = -\frac{2}{n}\sum x \cdot (y - \hat{y}) = -\frac{2}{3}[1 \cdot (2-0) + 2 \cdot (4-0) + 3 \cdot (6-0)] = -28$$

$$\frac{\partial J}{\partial b} = -\frac{2}{n}\sum (y - \hat{y}) = -\frac{2}{3}[(2-0) + (4-0) + (6-0)] = -8$$

4. **Update Parameters:**

$$W_{\text{new}} = W_{\text{old}} - \eta \cdot \frac{\partial J}{\partial W} = 0 - 0.1 \cdot (-28) = 2.8$$

$$b_{\text{new}} = b_{\text{old}} - \eta \cdot \frac{\partial J}{\partial b} = 0 - 0.1 \cdot (-8) = 0.8$$

Iteration 2:

Repeat the process with $W = 2.8$, $b = 0.8$.

6. Visualization

Imagine you're hiking down a hill:

- The cost function is the hill's surface.
- The gradient tells you which direction to go to descend fastest.
- The learning rate decides how big each step is.

7. Relatable Analogy

Think of gradient descent as adjusting a recipe:

1. **Initial Recipe:** You randomly mix ingredients (random parameters).
2. **Taste Test:** You evaluate the dish (cost function).
3. **Adjust Ingredients:** Add or reduce based on taste (gradient).
4. **Repeat:** Keep refining until it's perfect (minimum cost).