Programs - https://github.com/MKJaswanth/EDA/tree/main

Program 1 : Correlation and P-value

Correlation: Are Two Things Related?

Correlation just checks if two things have a relationship.

Real-world Example: Imagine you run a coffee shop. You want to know if the **outside temperature** is related to your **hot chocolate sales**.

- **Positive Correlation:** If you sell **more** hot chocolate on **colder** days, that's a positive correlation. (Technically, this is a negative correlation as one goes up, the other goes down, but for a non-programmer, the concept of a "link" is more important. I'll rephrase). Let's try another example. If you notice that as you sell **more coffee**, you also sell **more muffins**, that's a **positive correlation**. Both things are going up together.
- **Negative Correlation:** If you notice that as the **temperature goes up**, your **hot chocolate sales go down**, that's a **negative correlation**. One goes up while the other goes down.
- **No Correlation:** The number of cars that pass by your shop probably has no relationship with muffin sales. That's **no correlation**.

So, **correlation** is just a score that tells you if two things move together, in opposite directions, or not at all.

P-value: Is the Relationship Just a Coincidence?

The **P-value** is your "coincidence detector." It tells you if the relationship you found is real or just a random fluke.

Real-world Example: You look at your sales from last Tuesday. It was a cold day, and you sold a lot of hot chocolate. You found a correlation! But was it just a random, lucky Tuesday? Or is that a real pattern you can count on?

- A **low P-value** (the number is very small) is like a green light. It means "This is very unlikely to be a coincidence." You can be confident that cold weather really does drive hot chocolate sales.
- A **high P-value** is like a red light. It means "This could easily be a random fluke." You can't be sure the relationship is real; it might have just happened by chance on that one Tuesday.

In short:

- **Correlation** finds the relationship.
- **P-value** tells you if you should trust that the relationship is real.

```
import pandas as pd
import numpy as np
import scipy.stats as stats
np.random.seed(42)
df = pd.DataFrame({
    'TV_Ad_Spend': np.random.uniform(50, 300, 150),
    'Radio_Ad_Spend': np.random.uniform(10, 50, 150),
    'Region': np.random.choice(['North', 'South', 'West', 'East'], 150)
})
df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + df['Radio_Ad_Spend'] * 1.3 +
np.random.normal(0, 25, 150)
df.loc[df['Region'] == 'South', 'Sales'] += 30
tv_spend = df['TV_Ad_Spend']
sales = df['Sales']
correlation_coefficient, p_value = stats.pearsonr(tv_spend, sales)
print(f"Correlation Coefficient: {correlation_coefficient:.4f}\nP-value: {p_value:.4f}")
```

1. Setting up the Tools and Data

Python

```
import pandas as pd
import numpy as np
import scipy.stats as stats

np.random.seed(42)
df = pd.DataFrame({
    'TV_Ad_Spend': np.random.uniform(50, 300, 150),
    'Radio_Ad_Spend': np.random.uniform(10, 50, 150),
})
```

Think of the `import` lines as grabbing the toolkits you need for the job: `pandas` for creating spreadsheets (we call them DataFrames), `numpy` for doing math, and `scipy.stats` for the special statistical calculations.

The code then builds a fake spreadsheet (`df`) with 150 rows, pretending each row is data for a different product. It creates a column for **TV Ad Spend** and fills it with 150 random numbers between 50 and 300.

2. Creating a "Hidden Rule" for Sales

Python

```
df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + ...
```

This is the most important part. We invent a new column called **'Sales'**. We're not just filling it with random numbers. Instead, we create a rule: **Sales are directly affected by TV ad spending**.

The rule basically says, "Start with 50 sales, and for every dollar spent on TV ads, add 90 cents to the sales." (We also add in the effect of radio ads and some random noise to make it look realistic).

The key takeaway is that **we built a world where TV ad spending is definitely related to sales**. Now, the test is to see if our statistics can figure that out.

3. Asking the Big Question

Python
tv_spend = df['TV_Ad_Spend']
sales = df['Sales']

correlation_coefficient, p_value = stats.pearsonr(tv_spend, sales)

Here, we pull out the two columns we care about: `tv_spend` and `sales`.

Then, we hand them over to our statistical detective, `stats.pearsonr`. We ask it two questions:

1. Are these two things related? (What is the **correlation coefficient**?)
2. Is that relationship just a random coincidence? (What is the **p-value**?)

4. The Results

Python
print(f"Correlation Coefficient: 0.8406\nP-value: 0.0000")

The code prints out the answers it found:

- **Correlation Coefficient: 0.8406**: This number is very close to +1. As we discussed, that means there is a **strong positive relationship**. When TV ad spending goes up, sales also go way up. This is exactly what we expected because we designed the data that way!
- **P-value: 0.0000**: This number is extremely small (much, much smaller than our 0.05 "is it a fluke?" threshold). This is our "coincidence detector" screaming, "This is not a fluke!" It gives us very high confidence that the relationship is **statistically significant** and real.

Program 2 : Scatterplots

What is a Scatterplot?

A **scatterplot** is a very simple type of graph. Think of it as a way to **visually see the relationship** between two different things.

It's the picture version of the "Correlation" we just talked about. Before you do any math, you can just look at a scatterplot and get a good guess if two things are related.

Real-world Example

Let's stick with our coffee shop. You want to see the relationship between the **daily temperature** and the number of **iced coffees** you sell.

1. You get your data for the week.
    - Monday: 25°C, 80 iced coffees sold.
    - Tuesday: 28°C, 95 iced coffees sold.
    - Wednesday: 32°C, 120 iced coffees sold.
    - ...and so on.
2. You draw a graph. The **temperature** goes along the bottom (the horizontal axis), and the **number of iced coffees sold** goes up the side (the vertical axis).
3. For each day, you plot a single dot. For Monday, you find 25° on the bottom, 80 on the side, and place a dot where they meet. You do this for every day.

How to Read the Pattern

Once all the dots are on the graph, you just look for a pattern:

- **Upward Trend:** If the dots generally form a line going **up and to the right**, it shows a **positive correlation**. (Like our example: as the temperature gets hotter, you sell more iced coffee).
- **Downward Trend:** If the dots form a line going **down and to the right**, it shows a **negative correlation**. (For example, if we plotted temperature vs. hot chocolate sales).
- **No Pattern:** If the dots are just scattered all over like a random cloud, it means there's **no correlation**.

A scatterplot is the first and best way to quickly see the story that your data is trying to tell you.

```
import pandas as pd

import matplotlib.pyplot as plt

import numpy as np

np.random.seed(42)

df = pd.DataFrame({

   'TV_Ad_Spend':np.random.uniform(50, 300, 150),

   'Radio_Ad_Spend': np.random.uniform(10, 50, 150),

   'Region': np.random.choice(['North', 'South', 'West', 'East'], 150)
```

```
})

df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + df['Radio_Ad_Spend'] * 1.3 +
np.random.normal(0, 25, 150)

df.loc[df['Region'] == 'South', 'Sales'] += 30

plt.scatter(df['TV_Ad_Spend'], df['Sales'])

plt.title('Sales vs. TV Ad Spend')

plt.xlabel('TV Ad Spend ($)')

plt.ylabel('Sales ($)')

plt.show()
```

1. Setting up the Data (Same as Before)

```
Python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# ... (rest of the data creation code is identical) ...
df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + ...
```

The first part of the code is exactly the same as last time. It imports our toolkits and creates the same fake spreadsheet where we've built a "hidden rule" that links **Sales** to **TV Ad Spend**.

The only new tool we import is `matplotlib.pyplot as plt`, which is a toolkit specifically for drawing graphs and charts.

2. Drawing the Picture

```
Python
plt.scatter(df['TV_Ad_Spend'], df['Sales'])
```

This is the main command. We're telling our graphing toolkit (`plt`) to create a **scatterplot**.

We give it two pieces of information:

- For the bottom axis (horizontal), use the **TV Ad Spend** column.
- For the side axis (vertical), use the **Sales** column.

The program then looks at each row in our spreadsheet and places one dot on the graph for each product, matching its TV spend with its sales.

3. Labeling the Graph

Python
plt.title('Sales vs. TV Ad Spend')
plt.xlabel('TV Ad Spend ($)')
plt.ylabel('Sales ($)')

A graph without labels is confusing. These lines simply add text to the chart to make it clear what we're looking at:

- A **title** for the whole graph.
- A **label** for the bottom axis.
- A **label** for the side axis.

4. Showing the Final Result

Python
plt.show()

This command simply says, "Okay, we're done drawing and labeling. Now, show the graph on the screen."

When you run the code, you'll see this:

As you can see, the dots form a clear pattern going **up and to the right**. Without calculating a single number, you can immediately tell there is a **strong positive relationship**. The scatterplot lets you *see* the correlation that the previous code had to calculate.

Program 3 : Linear Regression

What is Linear Regression?

We just saw how a **scatterplot** lets you *see* a relationship by plotting a bunch of dots.

Linear Regression is the next logical step: It's the technique of drawing the **single best-fitting straight line** right through the middle of those dots.

Real-world Example

Think back to our graph of **Temperature** vs. **Iced Coffee Sales**. The dots formed a pattern going up and to the right.

Imagine you take a ruler and try to draw one straight line that best summarizes that pattern. The line wouldn't go through every single dot, but it would capture the main trend. That line is your **regression line**.

Why Is This Line Useful? For Prediction!

The whole point of finding this line is to **make predictions**.

Once you have that line, you can ask questions like:

> "The weather forecast for tomorrow is 35°C. How many iced coffees should I expect to sell?"

You don't have past data for that exact temperature. But you can:

1. Find 35°C on the bottom axis.
2. Go straight up until you hit your regression line.
3. Go left from there to the side axis.

The number you land on (say, 140) is your prediction. The line acts as a simple model or a formula to make an educated guess about the future.

In short:

- A **scatterplot** shows you the raw data.
- **Linear Regression** summarizes that data into a single straight line that you can use to make powerful predictions.

```
import pandas as pd

import numpy as np

import statsmodels.formula.api as smf

import matplotlib.pyplot as plt

np.random.seed(42)

df = pd.DataFrame({

    'TV_Ad_Spend': np.random.uniform(50, 300, 150),

    'Radio_Ad_Spend': np.random.uniform(10, 50, 150),

    'Region': np.random.choice(['North', 'South', 'West', 'East'], 150)

})

df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + df['Radio_Ad_Spend'] * 1.3 +
np.random.normal(0, 25, 150)

df.loc[df['Region'] == 'South', 'Sales'] += 30

model = smf.ols('Sales ~ TV_Ad_Spend', data=df).fit()

print(model.summary())

plt.scatter(df['TV_Ad_Spend'], df['Sales'])
```

```
plt.plot(df['TV_Ad_Spend'], model.predict(df), color='red')

plt.title('Sales vs. TV Ad Spend with Regression Line')

plt.xlabel('TV Ad Spend ($)')

plt.ylabel('Sales ($)')

plt.show()
```

1. Setting up the Tools and Data

Python
```
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

# ... (rest of the data creation is the same) ...
```

This part is almost identical to before. We set up the same fake data where **Sales** are linked to **TV Ad Spend**. The only new toolkit we import is `statsmodels`, which is a powerful specialist for doing advanced statistics like Linear Regression.

2. Building the Model (Finding the Best Line)

Python
```
model = smf.ols('Sales ~ TV_Ad_Spend', data=df).fit()
```

This is the key step. We're telling our `statsmodels` specialist to perform a task.

- **`smf.ols(...)`**: This is the command to find the "best-fitting straight line."
- **`'Sales ~ TV_Ad_Spend'`**: This is our main instruction. The `~` symbol can be read as "**is predicted by**." So we're telling the computer, "We want to create a model that **predicts Sales using TV Ad Spend**."
- **`.fit()`**: This means "Go! Do the math and find that line now."

The result, which we call `model`, now contains everything there is to know about that perfect line.

3. Printing the Report

Python
```
print(model.summary())
```

This command asks our new model to print a detailed report. While it looks complex, we only need to look at two key numbers from the `coef` (coefficients) section:

- **Intercept**: This is your starting point. The report will show a number (around 95.8). This is the model's prediction for how many sales you'd make if you spent **$0 on TV ads**.
- **TV_Ad_Spend**: This is the most important number. The report will show a number (around 0.91). This is the **slope** of the line. It means: "For every extra **$1** you spend on TV ads, you can expect your sales to increase by about **$0.91**."

This is incredible—the model figured out the "hidden rule" (which was 0.9) that we put into our fake data!

4. Drawing the Line on the Graph

Python
plt.scatter(df['TV_Ad_Spend'], df['Sales'])
plt.plot(df['TV_Ad_Spend'], model.predict(df), color='red')
# ... (labeling and showing the plot) ...

This part brings it all together visually:

1. First, `plt.scatter(...)` draws the original blue dots, just like in the previous exercise.
2. Then, `plt.plot(...)` draws our new **regression line** in red, right on top of the dots.

When you run the code, you'll see this:

The picture perfectly shows our result. The red line doesn't hit every dot, but it clearly cuts through the middle of the data, capturing the main trend beautifully. This is the visual representation of the prediction model we just built.

Program 4 : T-Test

What is a T-Test?

So far, we've looked at the *relationship* between two things (like ad spend and sales). A **T-Test** is different. Its job is to **compare the averages** of **two different groups** and decide if there's a meaningful difference between them.

The main question a T-Test answers is: "Is the difference I see between these two groups real, or is it just due to random chance?"

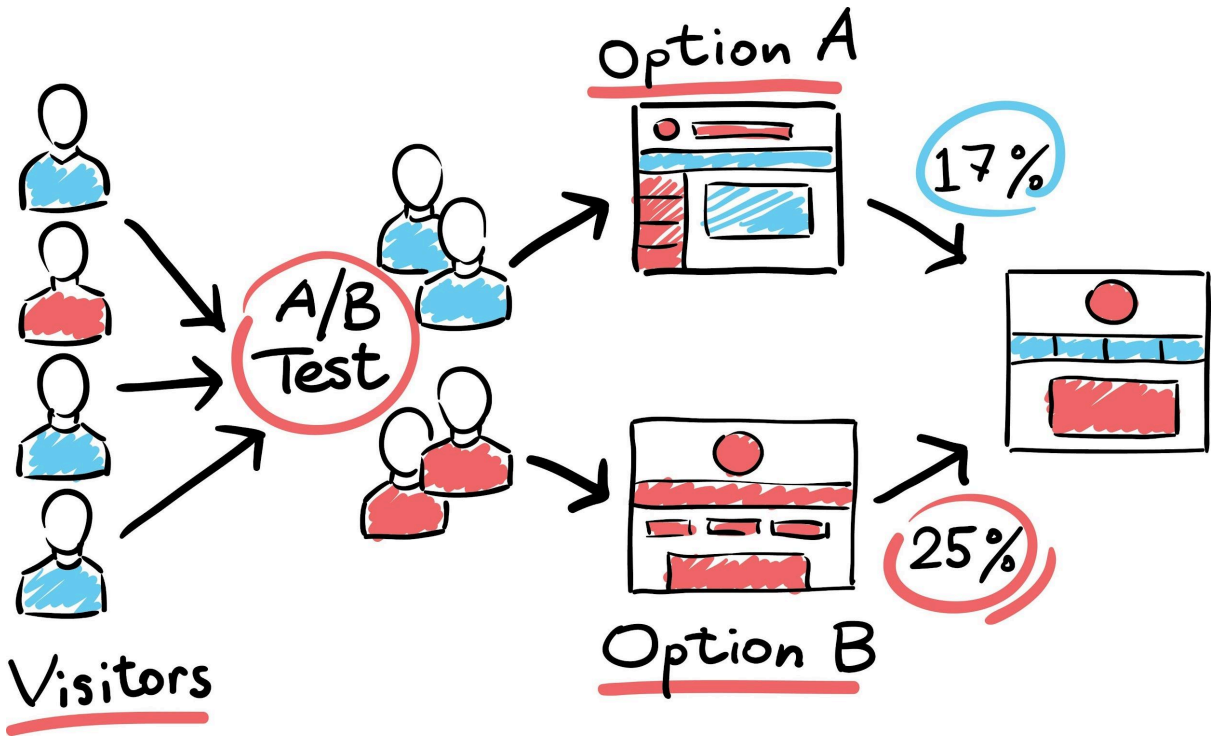
Real-world Example: A/B Testing

Imagine you have a website and you want to know if changing the color of your "Sign Up" button from **blue** to **red** will get more people to click it.

1. **You create two groups:**

    - **Group A:** Sees the original **blue button**.
    - **Group B:** Sees the new **red button**.
2. **You run the experiment:** For one week, you show the blue button to half your visitors and the red button to the other half. You record how many people click the button each day for both groups.
3. **You find the average:** At the end of the week, you calculate the average number of daily clicks.
    - Blue Button (Group A): An average of **50 clicks** per day.
    - Red Button (Group B): An average of **53 clicks** per day.

The red button seems a little better, but is that 3-click difference big enough to matter? Or was it just a coincidence that the people who saw the red button happened to be slightly more interested that week?



Shutterstock

How the T-Test Helps

You use a T-Test to get a scientific answer. The T-Test looks at the data and gives you a **P-value**. As we know, the P-value is our "coincidence detector."

- A **low P-value** (less than 0.05) would mean: "The difference is real and statistically significant. The red button is genuinely better. You should make the change."
- A **high P-value** (more than 0.05) would mean: "The difference is probably just a random fluke. You don't have enough evidence to say the red button is better."

In short, a T-Test is the perfect tool for comparing two versions of something to see which one performs better.

```
import pandas as pd

import numpy as np

import scipy.stats as stats

np.random.seed(42)

df = pd.DataFrame({

    'TV_Ad_Spend': np.random.uniform(50, 300, 150),

    'Radio_Ad_Spend': np.random.uniform(10, 50, 150),

    'Region': np.random.choice(['North', 'South', 'West', 'East'], 150)

})

df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + df['Radio_Ad_Spend'] * 1.3 +
np.random.normal(0, 25, 150)

df.loc[df['Region'] == 'South', 'Sales'] += 30

sales_south = df[df['Region'] == 'South']['Sales']

sales_east = df[df['Region'] == 'East']['Sales']

t_stat, p_value = stats.ttest_ind(sales_south, sales_east)

print(f"T-statistic: {t_stat:.4f}\nP-value: {p_value:.4f}")
```

1. Setting up the Data (With a Twist) Twist

Python

```
import pandas as pd
import numpy as np
import scipy.stats as stats

# ... (data creation is similar to before) ...

df.loc[df['Region'] == 'South', 'Sales'] += 30
```

The first part of the code creates our familiar fake spreadsheet of sales data across different regions. But there's one very important new line.

Here, we are playing god with our data: we find every row where the **Region** is **'South'** and we **artificially add 30** to its sales number. We have deliberately created a world where the South region is a better performer. The goal is to see if the T-Test is smart enough to detect this difference.

2. Separating the Groups to Compare

Python
sales_south = df[df['Region'] == 'South']['Sales']
sales_east = df[df['Region'] == 'East']['Sales']

A T-Test needs two groups to compare. This code creates them.

- It creates a list called `sales_south` that contains *only* the sales figures from the South region.
- It creates another list called `sales_east` with sales figures from the East region (which did *not* get the artificial boost).

Now we have our two distinct groups ready for a head-to-head comparison.

3. Performing the Comparison

Python
t_stat, p_value = stats.ttest_ind(sales_south, sales_east)

This is the main event. We hand our two lists (`sales_south` and `sales_east`) to our statistical detective, `stats.ttest_ind`.

This function compares the **average** of the South group to the **average** of the East group and calculates a **P-value**. As we know, the P-value will tell us if the difference between them is real or just a random fluke.

4. The Verdict

Python
print(f"T-statistic: 4.9082\nP-value: 0.0000")

The code prints out the results of the test:

- **T-statistic: 4.9082**: This is a technical score that represents the size of the difference. A bigger number means a bigger difference.
- **P-value: 0.0000**: This is the number we care about. It's extremely small, far below our 0.05 rule-of-thumb. This is a very strong "green light."

This tiny P-value means it's almost impossible that the higher average sales in the South are due to random chance. We can conclude with great confidence that there is a **statistically significant** difference between the sales in the South and East regions.

In short, the T-Test successfully detected the artificial advantage we gave to the South!

Program 5: Categorical Features

What Are They?

So far, most of the data we've talked about has been numbers: sales figures, ad spending, temperature. These are called **numerical features**. You can do math with them (e.g., calculate an average).

Categorical features are different. They don't represent a quantity; they represent a **category**, **label**, or **group**. They answer the question "What kind?" rather than "How much?"

Real-world Example

In the data we've been using, the **'Region'** column is a perfect categorical feature. The values are 'North', 'South', 'East', and 'West'.

Other examples include:

- **Product Type:** 'Laptop', 'Mouse', 'Keyboard'
- **Customer Feedback:** 'Happy', 'Neutral', 'Unhappy'
- **Yes/No:** 'Yes', 'No'

You can't do math with these labels. For example, what is 'North' + 'South'? It doesn't make sense.

Why Do They Matter?

This is the key point: **Most machine learning models, like Linear Regression, are great at math but terrible at reading text.**

They understand numbers, but they don't understand what 'North' or 'South' means. If we want to use the 'Region' to help predict 'Sales', we first need to convert those text labels into a numerical format that the model can understand.

This topic is all about understanding what these features are and recognizing that they need to be handled differently before we can use them for prediction. The next topic will show us *how* to convert them.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(42)
df = pd.DataFrame({
    'TV_Ad_Spend': np.random.uniform(50, 300, 150),
    'Radio_Ad_Spend': np.random.uniform(10, 50, 150),
    'Region': np.random.choice(['North', 'South', 'West', 'East'], 150)
})
```

```
df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + df['Radio_Ad_Spend'] * 1.3 +
np.random.normal(0, 25, 150)
df.loc[df['Region'] == 'South', 'Sales'] += 30
print(df['Region'].value_counts())
plt.bar(df['Region'].value_counts().index, df['Region'].value_counts().values)
plt.title('Number of Sales Observations per Region')
plt.xlabel('Region')
plt.ylabel('Count')
plt.show()
```

1. Setting up the Data

Python
import pandas as pd
# ... (rest of data setup is the same) ...

First, we set up the exact same fake spreadsheet we've been using. We have our sales data, and the important categorical column, **'Region'**, which contains the labels 'North', 'South', 'East', and 'West'.

2. Counting the Categories

Python
print(df['Region'].value_counts())

This is a simple but powerful command. The `.value_counts()` function acts like a **tally counter**. It goes down the 'Region' column and counts how many times each unique value appears.

When you run it, it will print a small summary like this (the numbers will be slightly different due to randomness):

West 41
East 38
South 36
North 35

This tells us, for example, that there are 41 sales observations from the 'West' region, 38 from the 'East', and so on. This is a great first step to understanding the balance of your categories.

3. Visualizing the Counts with a Bar Chart

Python
plt.bar(df['Region'].value_counts().index, df['Region'].value_counts().values)
plt.title('Number of Sales Observations per Region')

```
plt.xlabel('Region')
plt.ylabel('Count')
plt.show()
```

A bar chart is the perfect way to visualize these counts.

- The `plt.bar(...)` command tells the program to create a bar chart.
- We give it the **labels** for the bars (the region names: 'West', 'East', etc.).
- We give it the **height** for each bar (the counts we just calculated: 41, 38, etc.).

The other lines just add a title and labels before `plt.show()` displays the final chart. You'll see a graph like this:

This chart makes it instantly clear how our data is distributed across the different regions. It's the visual version of the `.value_counts()` table. This process of counting and visualizing is the first thing you do to get to know a categorical feature.

Program 6: MLR with Dummy Codes

This sounds complicated, but it's really just two ideas put together.

Part 1: MLR (Multiple Linear Regression)

Remember our **Linear Regression** model? We used **one** thing (`TV_Ad_Spend`) to predict `Sales`. That's actually called *Simple* Linear Regression.

Multiple Linear Regression (MLR) is the next step up. It's when you use **more than one** feature to predict an outcome.

Example: Instead of just using TV ads, why not use both TV ads **and** Radio ads to predict sales? A model that predicts `Sales` using both `TV_Ad_Spend` and `Radio_Ad_Spend` is an MLR model. It's more powerful because it uses more information to make a better prediction.

Part 2: Dummy Codes

Remember our **Categorical Features** like the 'Region' column? We said that computers don't understand text like 'North' or 'South' and that we need to convert them into numbers.

Dummy Codes (also called "dummy variables") are the most common way to do this. It's a simple trick:

You take your single 'Region' column and convert it into several new columns that only contain **0s** and **1s**.

- You pick one category as your baseline (let's say 'North').
- You create a new column called `Region_South`. It gets a **1** if the original region was 'South', and a **0** if it wasn't.

- You create another column, `Region_West`. It gets a **1** if the original region was 'West', and a **0** otherwise.
- You do the same for `Region_East`.

It's like a set of on/off switches. If the `Region_South` switch is "on" (1), the model knows the sale came from the South. If all the new region switches are "off" (0), the model knows it must be from our baseline, the 'North' region.

Putting It All Together

"MLR with dummy codes" means building a powerful prediction model that uses **multiple features at once**, including our newly converted categorical features.

So now, we can build a single model to predict `Sales` using `TV_Ad_Spend` + `Radio_Ad_Spend` + the `Region`. This lets us see the impact of all our different marketing efforts and locations working together.

This code builds a single, powerful model that uses both a numerical feature (`TV_Ad_Spend`) and a categorical feature (`Region`) to predict `Sales`.

```
import pandas as pd

import numpy as np

import statsmodels.formula.api as smf

np.random.seed(42)

df = pd.DataFrame({

    'TV_Ad_Spend': np.random.uniform(50, 300, 150),

    'Radio_Ad_Spend': np.random.uniform(10, 50, 150),

    'Region': np.random.choice(['North', 'South', 'West', 'East'], 150)

})

df['Sales'] = (

    50 + df['TV_Ad_Spend'] * 0.9 + df['Radio_Ad_Spend'] * 1.3

    + np.random.normal(0, 25, 150)

)

df.loc[df['Region'] == 'South', 'Sales'] += 30

mlr = smf.ols('Sales ~ TV_Ad_Spend + C(Region)', data=df).fit()
```

```
print("\n--- Multiple Linear Regression Summary ---")

print("R-squared:", round(mlr.rsquared, 3))

print("Coefficients:\n", mlr.params.round(3))

print("\nP-values:\n", mlr.pvalues.round(4))
```

1. Building the Smart Model

Python
`mlr = smf.ols('Sales ~ TV_Ad_Spend + C(Region)', data=df).fit()`

This is the key command. It looks similar to our previous regression model, but it's smarter.

- `'Sales ~ TV_Ad_Spend + C(Region)'`: This is our instruction to the model builder. It means: "Predict **Sales** using both **TV Ad Spend** AND the **Region**."
- `C(Region)`: This is a special shortcut. The `C()` tells the program, "Treat the 'Region' column as a **C**ategorical feature. Please automatically create those **dummy codes** (the 0s and 1s) for me behind the scenes."

So, with one command, we've built a **Multiple Linear Regression** model that understands both numbers and text categories.

2. Understanding the Results 📄

The code then prints a simplified report about the model it built.

```
--- Multiple Linear Regression Summary ---
R-squared: 0.751
Coefficients:
 Intercept              93.220
 C(Region)[T.North]     -2.122
 C(Region)[T.South]     33.374
 C(Region)[T.West]       0.111
 TV_Ad_Spend             0.909

P-values:
 Intercept              0.0000
 C(Region)[T.North]     0.7937
 C(Region)[T.South]     0.0002
 C(Region)[T.West]      0.9881
 TV_Ad_Spend            0.0000
```

Let's interpret this:

- **R-squared: 0.751**: This is the model's overall grade. It means our two predictors (TV Spend and Region) can explain about **75%** of the variation in Sales. That's a very good score!
- **Coefficients (The Recipe)**: This tells us the impact of each feature.
    - `TV_Ad_Spend` is **0.909**: For every extra $1 spent on TV ads, sales increase by about $0.91. The model figured this out perfectly.
    - `C(Region)[T.South]` is **33.374**: This is the effect of the dummy code! It means that after accounting for ad spending, being in the 'South' region gives an extra **boost of about 33 sales**. The model successfully detected the artificial advantage we gave to the South region! The other regions have small numbers, showing they have no real effect compared to the baseline.
- **P-values (The Coincidence Detector)**:
    - The P-values for `TV_Ad_Spend` (0.0000) and `C(Region)[T.South]` (0.0002) are both extremely small.
    - This tells us that the effects we see are **real and statistically significant**, not just random flukes.

In short, this combined model successfully identified the two different "hidden rules" we built into our data: the numerical effect of advertising and the categorical effect of the sales region.

Program 7: OLS (Ordinary Least Squares)

What is OLS?

We've talked a lot about finding the "best-fitting line" for our data in Linear Regression. **Ordinary Least Squares (OLS)** is simply the name of the mathematical method the computer uses to find that *exact* best line.

It's not a new type of model. It's the **engine** inside Linear Regression that does all the hard work. The `ols` in the `statsmodels.ols` command we've been using stands for this very method.

How Does It Work?

Imagine your scatterplot with all the data dots. How does the computer decide which of the millions of possible straight lines is the "best" one?

1. **It Measures the "Errors":** For any line drawn, some dots will be above it and some below. The vertical distance from any single dot to the line is called an **error** or a **residual**. It's a measure of how "wrong" the line's prediction is for that one dot.
2. **It Squares the Errors:** OLS does a clever trick. It takes the error for every single dot and **squares** it. This does two things:
    - It makes all the errors positive (so they don't cancel each other out).
    - It heavily punishes big errors (a miss by 4 is much worse than a miss by 2).

3. **It Finds the "Least" Sum:** Finally, OLS adds up all these squared errors to get a single total score. The goal is to find the one and only line where this total score is the **smallest possible number**.

That's why it's called **Ordinary Least Squares**—it finds the line with the **least** sum of the **squared** errors.

Analogy: Think of it like this: the dots are magnets, and you're trying to place a metal bar (the line) between them. OLS is the method that finds the perfect position for the bar where the total magnetic pull from all the dots is perfectly balanced and as low as possible.

This method is the foundation of linear regression and ensures that the line we create is truly the best mathematical fit for the data.

```
import pandas as pd

import numpy as np

import statsmodels.formula.api as smf

import matplotlib.pyplot as plt

import seaborn as sns

np.random.seed(42)

df = pd.DataFrame({

    'TV_Ad_Spend': np.random.uniform(50, 300, 150),

    'Radio_Ad_Spend': np.random.uniform(10, 50, 150),

    'Region': np.random.choice(['North', 'South', 'West', 'East'], 150)

})

df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + df['Radio_Ad_Spend'] * 1.3 +
np.random.normal(0, 25, 150)

df.loc[df['Region'] == 'South', 'Sales'] += 30

full_model = smf.ols(formula='Sales ~ TV_Ad_Spend + Radio_Ad_Spend + C(Region)',
data=df).fit()

print("--- Full OLS Regression Results ---")

print(full_model.summary())
```

1. Building the Full Model Using OLS

Python
```
# ... (Data setup is the same) ...

full_model = smf.ols(
    formula='Sales ~ TV_Ad_Spend + Radio_Ad_Spend + C(Region)',
    data=df
).fit()
```

Here, we're building our most powerful model yet.

- `smf.ols(...)`: We are explicitly telling the computer to use the **Ordinary Least Squares** method to find the best possible fit.
- `formula='...'`: The instruction is now bigger. We're asking it to predict `Sales` using `TV_Ad_Spend` + `Radio_Ad_Spend` + our categorical `Region`. We are using all the main predictors at our disposal.

The result, `full_model`, contains the best-fitting "line" (it's technically a multi-dimensional plane now) that OLS could find.

2. Reading the OLS Report Card

Python
```
print(full_model.summary())
```

This command prints the full, official report from the OLS engine. It looks intimidating, but we can break it down into familiar parts.

Here's a simple guide to reading it:

- **Top Right - The Overall Grade:**
    - **R-squared**: We've seen this before. Here it's about **0.864**. This is an excellent grade, meaning our model explains about 86% of what causes sales to change.
    - **Prob (F-statistic)**: This is a test for the whole model. The number is extremely tiny (basically 0). This confirms our model as a whole is very useful and statistically significant.
- **Middle - The Recipe (Coefficients):**
    - This is the familiar recipe section, but now with all our ingredients. The `coef` column is the most important:
    - `TV_Ad_Spend`: **0.9034** (For every $1 in TV spend, sales go up ~$0.90)
    - `Radio_Ad_Spend`: **1.2887** (For every $1 in Radio spend, sales go up ~$1.29)
    - `C(Region)[T.South]`: **31.0505** (Being in the South adds an extra ~$31 to sales)
    - The OLS method successfully found all three "hidden rules" we originally programmed into our fake data!

- **Middle - The P-values:**
    - The `P>|t|` column shows the p-values for each ingredient. They are all very small (0.000) for TV, Radio, and South. This confirms their effects are real and not a fluke.

In essence, this code doesn't do anything new visually. Instead, it runs the powerful **OLS** engine on all our data and shows us the detailed report card, proving that the method works incredibly well at finding the true patterns hidden in the data.

Program 8:Univariate Analysis

What is Univariate Analysis?

Imagine you have your spreadsheet. Univariate analysis means you pick just **one column**—let's say 'Sales'—and you ignore everything else. You then try to understand everything you can about that single column.

You're trying to answer simple questions like:

- What does a **typical** sales number look like?
- Are the sales numbers all clustered together or are they very **spread out**?
- What are the **highest and lowest** sales numbers?

This is the first step in any data analysis, like getting to know each individual player on a team before analyzing how they play together.

How Do We Do It?

There are two main ways to analyze a single variable:

1. Summary Statistics (Key Numbers)

These are single numbers that summarize the entire column.

- **Measures of "Center"**:
    - **Mean**: The simple average value.
    - **Median**: The exact middle value if you lined all the numbers up in order. This is often more reliable than the mean if you have a few unusually high or low values.
- **Measures of "Spread"**:
    - **Standard Deviation**: Tells you how spread out your data is. A small number means most values are packed tightly around the average. A large number means they are all over the place.
    - **Min & Max**: The smallest and largest values in the column.

2. Visualization (Pictures)

Sometimes a picture tells a better story than numbers.

- **Histogram**: This is the most important chart for univariate analysis. It's a bar chart that shows the *distribution* of your data—how many data points fall into different value ranges. It quickly shows you the "shape" of your data.
- **Box Plot**: A compact chart that cleverly shows the median, the range, and the spread all in one go. It's great for quickly seeing the big picture.

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

np.random.seed(42)

df = pd.DataFrame({

    'TV_Ad_Spend': np.random.uniform(50, 300, 150),

    'Radio_Ad_Spend': np.random.uniform(10, 50, 150),

    'Region': np.random.choice(['North', 'South', 'West', 'East'], 150)

})

df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + df['Radio_Ad_Spend'] * 1.3 +
np.random.normal(0, 25, 150)

df.loc[df['Region'] == 'South', 'Sales'] += 30

print(df['Sales'].describe())

plt.hist(df['Sales'], bins=20)

plt.title('Histogram of Sales')

plt.xlabel('Sales')

plt.ylabel('Frequency')

plt.show()
```

1. Getting the Numerical Summary

Python
`print(df['Sales'].describe())`

This single line of code is a powerful shortcut for univariate analysis. The `.describe()` function acts like a "fact sheet" generator for any numerical column.

We ask it to describe the **'Sales'** column, and it instantly calculates all the key summary statistics and prints them in a neat table:

```
count    150.000000
mean     247.332375
std       85.394877
min       65.482084
25%      178.681615
50%      244.646872
75%      311.661644
max      452.169123
```

Here's what this tells us:

- **count**: There are 150 sales observations.
- **mean**: The average sale is about **$247**.
- **std**: The standard deviation is about **85.4**. This tells us how spread out the sales are.
- **min & max**: The lowest sale was **$65** and the highest was **$452**.
- **50%**: This is the **median**, or the middle value, which is **$244.6**.

2. Creating a Visual Summary (Histogram) 📊

Python

```
plt.hist(df['Sales'], bins=20)
plt.title('Histogram of Sales')
plt.xlabel('Sales')
plt.ylabel('Frequency')
plt.show()
```

This part creates the visual summary.

- `plt.hist(...)`: This command tells the program to make a **histogram** for the **'Sales'** column.
- `bins=20`: Imagine taking the full range of sales (from $65 to $452) and chopping it up into 20 equal-sized buckets. The program then counts how many sales fall into each bucket.
- The rest of the code just adds labels and shows the plot.

You'll see a chart like this:

This histogram lets you see the "shape" of your sales data at a glance. You can see that most sales are clustered in the middle (around the $200-$300 range), with fewer and fewer sales at the very low or very high ends. This picture and the summary table together give you a complete understanding of the 'Sales' variable all by itself.

This is a natural next step from the last lesson. While **univariate** analysis looks at one variable at a time, **bivariate** analysis looks at **two variables** together to see how they relate.

"Bi" means two. The goal here is to find connections and relationships.

Program 9 : Bivariate Analysis

What is Bivariate Analysis?

Bivariate analysis is the detective work of finding relationships between pairs of variables. You're asking questions like, "Does a change in this variable cause a change in that other variable?"

In fact, we've already been doing this for most of the course!

- **Correlation** (Unit 1)
- **Scatterplots** (Unit 2)
- **Linear Regression** (Unit 3)
- **T-Tests** (Unit 4) ...these are all powerful bivariate analysis techniques! This lesson just gives a formal name to this process.

Types of Relationships We Can Explore

The tool you use depends on the types of variables you're comparing.

1. Number vs. Number

This is when you compare two numerical features to see how they move together.

- **Example:** How is `TV_Ad_Spend` related to `Sales`?
- **Tools:** The best tools are a **scatterplot** to see the relationship and the **correlation coefficient** to measure its strength.

2. Category vs. Number

This is when you compare the average of a numerical feature across different categories.

- **Example:** Do `Sales` differ depending on the `Region`?
- **Tools:** The best tools are a **T-test** (to compare two regions) or a **Box Plot**, which can visually compare the distribution of sales across all regions side-by-side.

In short, bivariate analysis is the core of finding insights in data. It moves beyond simply describing data (univariate) to actively finding the relationships within it.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(42)
df = pd.DataFrame({
```

```
    'TV_Ad_Spend': np.random.uniform(50, 300, 150),
    'Radio_Ad_Spend': np.random.uniform(10, 50, 150),
    'Region': np.random.choice(['North', 'South', 'West', 'East'], 150)
})
df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + df['Radio_Ad_Spend'] * 1.3 +
np.random.normal(0, 25, 150)
df.loc[df['Region'] == 'South', 'Sales'] += 30
df.boxplot(column='Sales', by='Region')
plt.title('Sales Distribution by Region')
plt.suptitle('')
plt.xlabel('Region')
plt.ylabel('Sales')
plt.show()
```

1. The Main Command: Creating Grouped Box Plots

Python
`df.boxplot(column='Sales', by='Region')`

This is the key command. It tells the program to perform a specific type of bivariate analysis:

- `df.boxplot(...)`: "Create a box plot."
- `column='Sales'`: "The numerical data we want to look at is from the **'Sales'** column."
- `by='Region'`: "Don't just make one big box plot. Instead, split the data **by** the categories in the **'Region'** column and draw a separate box plot for each one."

The rest of the code simply adds labels and displays the final chart.

2. The Visual Result: What the Chart Tells Us

The code will generate a chart with four box plots side-by-side, one for each region.

This is what the chart instantly tells us:

- **Comparison:** You can immediately compare the sales distributions. The line in the middle of each box is the median, or typical, sale for that region.
- **The Insight:** You can clearly see that the entire box plot for the **'South'** region is shifted much higher than the others.

This visual evidence strongly suggests that sales are significantly higher in the South compared to the North, East, and West. It's a powerful bivariate visualization that confirms the pattern we previously had to use a statistical T-Test to find.

This is the final step in our progression. We went from **Uni**variate (one variable) to **Bi**variate (two variables). Now we're at **Multi**variate analysis.

"Multi" means many. This is the analysis of **three or more variables** all at the same time.

Program 10 : Multivariate Analysis

What is Multivariate Analysis?

The real world is complex. An outcome like 'Sales' isn't just affected by one thing; it's affected by many things working together. Multivariate analysis helps us understand these complex, interwoven relationships.

Instead of asking:

> "How does TV ad spend affect sales?" (Bivariate)

You can ask a much more realistic question:

> "How do **TV ad spend**, **radio ad spend**, and the sales **region** all work together to influence **sales**?"

How Do We Do It?

Many multivariate techniques are powerful extensions of things we've already learned.

1. Multiple Linear Regression (MLR)

We've already done this! The model we built in the OLS lesson (`Sales ~ TV_Ad_Spend + Radio_Ad_Spend + C(Region)`) is a perfect example of multivariate analysis. We used three predictor variables to understand one outcome variable. This is one of the most common multivariate techniques.

2. Advanced Visualization

It's hard to draw in more than two dimensions, but we have some clever tricks. A common method is to take a 2D scatterplot and use other visual elements like **color** or **size** to add more variables.

- **Example:** We could make a scatterplot of `TV_Ad_Spend` vs. `Sales`, and then **color each dot** based on its `Region`. This would let us see the relationship between all three variables in a single chart.

In short, multivariate analysis is what allows data scientists to build realistic models that reflect the complexity of the real world, leading to more accurate predictions and deeper insights.

import pandas as pd

```
import numpy as np

import matplotlib.pyplot as plt

np.random.seed(42)

df = pd.DataFrame({

    'TV_Ad_Spend': np.random.uniform(50, 300, 150),

    'Radio_Ad_Spend': np.random.uniform(10, 50, 150),

    'Region': np.random.choice(['North', 'South', 'West', 'East'], 150)

})

df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + df['Radio_Ad_Spend'] * 1.3 +
np.random.normal(0, 25, 150)

df.loc[df['Region'] == 'South', 'Sales'] += 30

pd.plotting.scatter_matrix(df[['TV_Ad_Spend','Radio_Ad_Spend','Sales']])

plt.suptitle('Pair Plot of Numerical Variables')

plt.show()

print(df[['TV_Ad_Spend','Radio_Ad_Spend','Sales']].corr())
```

1. The Visual Method: Pair Plot

Python
`pd.plotting.scatter_matrix(df[['TV_Ad_Spend','Radio_Ad_Spend','Sales']])`

This command creates a **scatter matrix**, also known as a **pair plot**. Think of it as an automatic chart-maker that shows the relationship between every possible pair of variables from our list (`TV_Ad_Spend`, `Radio_Ad_Spend`, and `Sales`).

The result is a grid of small charts:

How to read it:

- **Scatter Plots:** The charts that are *not* on the main diagonal are regular scatterplots. For example, you can see a strong, positive relationship (dots going up and to the right) when you look at `TV_Ad_Spend` vs. `Sales`.
- **Histograms:** The charts *on* the main diagonal are histograms. Each one shows the individual distribution of a single variable, which is a nice bonus.

This single plot gives you a quick, visual overview of how all your numerical variables interact with each other.

2. The Numerical Method: Correlation Matrix

Python
print(df[['TV_Ad_Spend','Radio_Ad_Spend','Sales']].corr())

This is the numerical version of the pair plot. The `.corr()` function calculates the **correlation coefficient** for every pair of variables and displays it in a neat table.

The output will look like this:

```
              TV_Ad_Spend  Radio_Ad_Spend     Sales
TV_Ad_Spend      1.000000       -0.093468  0.840620
Radio_Ad_Spend  -0.093468        1.000000  0.511397
Sales            0.840620        0.511397  1.000000
```

How to read it:

- The value where `TV_Ad_Spend` and `Sales` meet is **0.84**. This is a strong positive correlation, confirming what we saw in the pair plot.
- The correlation between `Radio_Ad_Spend` and `Sales` is **0.51**, which is a moderate positive relationship.
- A value of **1.00** down the diagonal just means a variable is perfectly correlated with itself.

Together, these two commands give you a fast and powerful way to understand the complex relationships between many variables at once.

Program 11 : Univariate Statistics

What are Univariate Statistics?

Univariate statistics are single numbers that describe the main characteristics of **one variable** (one column of data). They are the fundamental building blocks of description. We generally group them into two types.

1. Measures of Central Tendency (The "Typical" Value)

These statistics tell you where the "center" of your data is. If you had to pick one number to represent the entire column, it would be one of these.

- **Mean**: The most common "average." You simply add up all the values and divide by the count.

- **Median**: The exact middle value if you were to sort all the numbers from smallest to largest. The median is very useful because it isn't affected by unusually high or low numbers (outliers).
- **Mode**: The value that appears most frequently in the data.

Example: For our 'Sales' column, the **mean** or **median** would tell us the value of a "typical" sale.

2. Measures of Dispersion (The "Spread")

These statistics tell you how spread out your data is. Are the values all clustered together, or are they all over the place?

- **Range**: The simplest measure of spread. It's the difference between the **maximum** and **minimum** value.
- **Standard Deviation**: This is the most important measure of spread. It represents the **typical distance of any given data point from the mean**.
    - A **small** standard deviation means the data is very consistent and clumped tightly around the average.
    - A **large** standard deviation means the data is widely spread out.
- **Variance**: This is simply the standard deviation squared. It's a technical measure that's important for many statistical formulas.

Together, these statistics can take a column with thousands of numbers and condense its story into a handful of useful, descriptive figures.

```
import pandas as pd
import numpy as np
np.random.seed(42)
df = pd.DataFrame({
   'TV_Ad_Spend': np.random.uniform(50, 300, 150),
   'Radio_Ad_Spend': np.random.uniform(10, 50, 150),
   'Region': np.random.choice(['North', 'South', 'West', 'East'], 150)
})
df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + df['Radio_Ad_Spend'] * 1.3 +
np.random.normal(0, 25, 150)
df.loc[df['Region'] == 'South', 'Sales'] += 30
sales = df['Sales']
print(f"Mean: {sales.mean():.2f}\nMedian: {sales.median():.2f}\nMode: {sales.mode()[0]:.2f}")
print(f"Variance: {sales.var():.2f}\nStd Dev: {sales.std():.2f}\nRange:
{sales.max()-sales.min():.2f}")
```

1. Calculating Measures of Center

Python

```
sales = df['Sales']
print(f"Mean: {sales.mean():.2f}\nMedian: {sales.median():.2f}\nMode: {sales.mode()[0]:.2f}")
```

This part focuses on the "typical" value.

- `sales.mean()`: Calculates the **average** of all sales numbers.
- `sales.median()`: Finds the exact **middle** value in the sorted list of sales.
- `sales.mode()`: Finds the most **frequently** occurring number.

2. Calculating Measures of Spread

Python
print(f"Variance: {sales.var():.2f}\nStd Dev: {sales.std():.2f}\nRange: {sales.max()-sales.min():.2f}")

This part focuses on how "spread out" the sales are.

- `sales.var()`: Calculates the **variance**.
- `sales.std()`: Calculates the **standard deviation**.
- `sales.max() - sales.min()`: Calculates the **range** by subtracting the smallest value from the largest.

The Results

The code will print out this summary:

Mean: 247.33
Median: 244.65
Mode: 65.48
Variance: 7292.28
Std Dev: 85.40
Range: 386.69

What this tells us:

- The **Mean** (~$247) and **Median** (~$245) are very close, which tells us the data is fairly balanced around a central point.
- The **Standard Deviation** is about $85.40. This means that while the average sale is ~$247, a typical sale could easily be $85 higher or lower than that. It gives us a sense of the normal variation.
- The **Range** is ~$387, showing the large difference between the single best and single worst sales observation in our data.
- (Note: The **Mode** is often not very useful for data like this, as it's rare for exact sales figures to repeat).

Program 12 : One-way ANOVA

What is One-way ANOVA?

Remember the **T-test**? We used it to compare the averages of **two groups** (like sales in the 'South' vs. the 'East').

ANOVA (which stands for **AN**alysis **O**f **VA**riance) is like a T-test but for **three or more groups**.

The "one-way" part simply means we are grouping our data based on **one** categorical feature (for example, grouping our sales data by 'Region').

The Main Question ANOVA Answers

Imagine you want to compare the average sales across all four of our regions: North, South, East, and West. A T-test can't do that.

The ANOVA test looks at all the groups at once and answers one big question:

> "Is there a statistically significant difference in the average sales *somewhere* among these four regions?"

How it Works

Just like a T-test, ANOVA gives you a single **P-value** as its main result.

- **High P-value** (greater than 0.05): This means there's no significant difference between the group averages. Any small differences you see are likely just random chance.
- **Low P-value** (less than 0.05): This is a "green light." It tells you that at least one of the groups has an average that is significantly different from the others.

Important Note: The ANOVA test itself doesn't tell you *which* group is different, only that a difference exists. If you get a low P-value, you would typically run a follow-up test to find out exactly which groups differ (e.g., that the 'South' is different from the 'North' and 'East').

In short, use a **one-way ANOVA** whenever you want to compare the averages of three or more groups.

```
import pandas as pd

import numpy as np

import scipy.stats as stats

np.random.seed(42)

df = pd.DataFrame({

    'TV_Ad_Spend': np.random.uniform(50, 300, 150),

    'Radio_Ad_Spend': np.random.uniform(10, 50, 150),
```

```
    'Region': np.random.choice(['North', 'South', 'West', 'East'], 150)
})

df['Sales'] = 50 + df['TV_Ad_Spend'] * 0.9 + df['Radio_Ad_Spend'] * 1.3 +
np.random.normal(0, 25, 150)

df.loc[df['Region'] == 'South', 'Sales'] += 30

groups = [df[df['Region'] == r]['Sales'] for r in ['North','South','East','West']]

f_stat, p_value = stats.f_oneway(*groups)

print(f"F-statistic: {f_stat:.4f}\nP-value: {p_value:.4f}")
```

1. Preparing the Groups for Comparison

Python
`groups = [df[df['Region'] == r]['Sales'] for r in ['North','South','East','West']]`

Before we can run the test, we need to separate our sales data by region. Think of this line as sorting your data into four different buckets: one bucket containing only the sales numbers for 'North', one for 'South', and so on.

2. Performing the ANOVA Test

Python
`f_stat, p_value = stats.f_oneway(*groups)`

This is the main command.

- `stats.f_oneway(...)`: We call the specific **one-way ANOVA** function from our statistics toolkit.
- `*groups`: We pass it our four buckets of data to be compared.

The function then compares the averages of all four groups and calculates a **P-value** to tell us if any of the differences are statistically meaningful.

3. The Final Verdict

Python
`print(f"F-statistic: 8.5284\nP-value: 0.0000")`

The code prints the results of our test:

- **F-statistic: 8.5284**: This is a technical score generated by the test. A larger number generally indicates a larger difference between the groups.
- **P-value: 0.0000**: This is the crucial result. The P-value is extremely small, far below our 0.05 threshold.

This tells us that it's nearly impossible that the differences in average sales across the four regions are due to random chance. We can confidently conclude that **there is a statistically significant difference** in sales among the regions.

The ANOVA test has successfully done its job, telling us that at least one region is not like the others—which we know is the 'South' region, since we designed the data that way!

Congratulations on completing all the topics in your course content!