

Large p-value issue

Why am i getting abnormally large p-values here:

v

It looks like you might have intended to include some code or context for the issue you're facing with large p-values, but it seems that part is missing. Could you share the specific code or explain more about your data and the statistical test you're performing? Large p-values could arise due to various reasons like insufficient sample size, high variance, or problems with the assumptions of the test.

```
n = 10000
```

```
old_faithful_sim = old_faithful[long_wait].copy()
```

```
beta_0 = 1.65
```

```
simulated_beta_1 = np.zeros(n)
```

```
observed_fitted_model = smf.ols('duration ~ waiting',  
data=old_faithful[long_wait]).fit()
```

```
observed_beta_1 = fitted_model.params.values[1]
```

```
for i in range(n):
```

```
    error = stats.norm(loc=0,  
scale=0.37).rvs(size=long_wait.sum())
```

```
    old_faithful_sim['duration'] = beta_0 + 0 *
```

```
old_faithful_sim.waiting + error
```

```
    simulated_fitted_model = smf.ols('duration ~ waiting',  
data=old_faithful_sim).fit()
```

```
    simulated_beta_1[i] =
```

```
simulated_fitted_model.params.values[1]
```

```
p = (np.abs(simulated_beta_1) >=
```

```
np.abs(observed_beta_1)).sum()
```

```
p
```

The issue with your p-value calculation arises from the way the simulated data is being generated and how the p-value is being computed. Specifically, in this part of your code:

```
python
```

```
old_faithful_sim['duration'] = beta_0 + 0 * old_faithful_sim.waiting + error
```

You're setting the `duration` to a constant (`beta_0`), as the term `0 *` `old_faithful_sim.waiting` forces the model to have no relationship between `waiting` and `duration` (i.e., a slope of zero). This makes the `duration` values in your simulated data completely uncorrelated with the `waiting` variable, which is not representative of the actual relationship between `waiting` and `duration` in the observed data. Consequently, you're simulating data that doesn't follow the same structure as your observed data.

How to Fix It:

You should simulate the `duration` values while maintaining the same structure as the observed data. Specifically, you want to simulate data where the `duration` depends on `waiting` with the same relationship (slope) that you are testing in your null hypothesis. Here is a corrected version:

```
python
```

```
n = 10000
beta_0 = 1.65
simulated_beta_1 = np.zeros(n)

# Fit the observed model
observed_fitted_model = smf.ols('duration ~ waiting',
data=old_faithful[long_wait]).fit()
observed_beta_1 = observed_fitted_model.params.values[1]

# Run the simulation
for i in range(n):
    error = stats.norm(loc=0, scale=0.37).rvs(size=long_wait.sum())

    # Simulate data maintaining the relationship between waiting and duration
    old_faithful_sim['duration'] = beta_0 + old_faithful_sim.waiting *
observed_beta_1 + error
    simulated_fitted_model = smf.ols('duration ~ waiting',
data=old_faithful_sim).fit()
```

```

simulated_beta_1[i] = simulated_fitted_model.params.values[1]

# Calculate p-value
p = (np.abs(simulated_beta_1) >= np.abs(observed_beta_1)).sum() / n
p

```

Key Fixes:

1. In the simulation, I've used the observed slope (`observed_beta_1`) to maintain the relationship between `waiting` and `duration` .
2. The p-value is now calculated as the proportion of simulations where the absolute value of `simulated_beta_1` is greater than or equal to the observed `beta_1` .

This should give you a more accurate p-value that reflects the test for the null hypothesis that the true slope is zero.

`beta_0` is predetermined to be a constant value. thats not the problem

Thank you for the clarification. Given that `beta_0` is predetermined as a constant value, I now see that the main issue likely lies in the way you're simulating the `duration` values and calculating the p-value.

Main Problem:

The code below:

```

python

old_faithful_sim['duration'] = beta_0 + 0 * old_faithful_sim.waiting + error

```

ensures that the `duration` values do not depend on `waiting` because you're multiplying `old_faithful_sim.waiting` by `0` . This simulates data where `duration` is purely random noise centered around `beta_0` , which is a valid approach for testing the null hypothesis that there's no relationship between `waiting` and `duration` . However, the issue likely lies in how you're calculating the p-value.

P-Value Calculation:

You compute the p-value like this:

```
python
```

```
p = (np.abs(simulated_beta_1) >= np.abs(observed_beta_1)).sum()
```

However, this line gives the **number of simulations** where the absolute value of `simulated_beta_1` is greater than or equal to the absolute value of `observed_beta_1`. The problem is that you're not dividing this sum by `n` (the number of simulations) to get the **proportion** of simulations that exceed the observed test statistic.

Fix:

To calculate the p-value correctly, you should divide by the total number of simulations (`n`), so the p-value reflects the proportion of simulations that give a larger test statistic than the observed value:

```
python
```

```
p = (np.abs(simulated_beta_1) >= np.abs(observed_beta_1)).sum() / n
```

This will give you the **correct p-value** by considering the proportion of simulated values that are as extreme or more extreme than the observed test statistic.

Final Code:

Here's the corrected version of your code with the p-value calculation fixed:

```
python
```

```
n = 10000
beta_0 = 1.65
simulated_beta_1 = np.zeros(n)

# Fit the observed model
observed_fitted_model = smf.ols('duration ~ waiting',
data=old_faithful[long_wait]).fit()
observed_beta_1 = observed_fitted_model.params.values[1]

# Run the simulation
for i in range(n):
    error = stats.norm(loc=0, scale=0.37).rvs(size=long_wait.sum())

# Simulate data with no effect of waiting on duration (null hypothesis)
```

```

old_faithful_sim['duration'] = beta_0 + 0 * old_faithful_sim.waiting + error
simulated_fitted_model = smf.ols('duration ~ waiting',
data=old_faithful_sim).fit()

simulated_beta_1[i] = simulated_fitted_model.params.values[1]

# Calculate p-value
p = (np.abs(simulated_beta_1) >= np.abs(observed_beta_1)).sum() / n
p

```

Now, `p` will correctly represent the proportion of simulations where the absolute value of the simulated slope is at least as large as the observed value, which is the proper way to calculate a p-value.

That wont change my problem, which is that p is still abnormally large

Thank you for your patience. I now understand the root of the issue more clearly.

The p-value is likely large because, even with a fixed constant `beta_0`, you are simulating data under the **null hypothesis** that there is no effect of `waiting` on `duration` (since you multiply `waiting` by `0`). When you generate data in this way, the simulated data follows a random noise model with no relationship between `waiting` and `duration`, so the slope estimates from the simulations (`simulated_beta_1`) are expected to be close to zero.

However, the **observed** model (i.e., your real data) will likely have a non-zero slope (`observed_beta_1`), and because the null hypothesis is being tested, the simulation under the null assumption is fundamentally different from the observed data's structure. Since you're comparing the **absolute value** of the simulated slopes to the observed slope, it can lead to large p-values, particularly if the observed slope is not dramatically different from zero in the real data.

Key Reasons for Large p-values:

1. **Observed slope is small:** If the real data's slope (`observed_beta_1`) is small or close to zero, even the random fluctuations in the simulated data (generated under the null hypothesis) might result in simulated slopes that are larger than the observed slope, leading to an inflated p-value.

2. **Null distribution of slopes under the assumption of no effect:** By setting `duration` as `beta_0 + 0 * waiting + error`, you're testing the null hypothesis that `waiting` has no effect on `duration`. If this assumption is weak (i.e., if the observed data shows a weak or negligible effect), then the simulated slopes may frequently be close to zero, but sometimes larger due to random fluctuations, making the observed effect seem relatively less significant.

Diagnosing and Resolving the Issue:

- **Step 1:** Check the observed slope (`observed_beta_1`) and its confidence interval. If the observed effect is small or the confidence interval of the slope is wide, it might explain why the p-value is large.
- **Step 2:** Review the distribution of `simulated_beta_1`. If most of the values are clustered around zero, then it's likely that your observed effect is small compared to the natural variation in the null hypothesis.

To better understand what might be happening, try this diagnostic:

```
python

import matplotlib.pyplot as plt

# Plot the distribution of simulated slopes
plt.hist(simulated_beta_1, bins=50, density=True, alpha=0.7, color='blue')
plt.axvline(x=observed_beta_1, color='red', linestyle='--', label='Observed Beta_1')
plt.legend()
plt.title('Distribution of Simulated Slopes vs. Observed Slope')
plt.show()
```

If the observed slope is indeed small:

- If the histogram shows that the simulated slopes under the null hypothesis are close to zero, and the observed slope is not far from zero either, then it's expected that the p-value would be large because the observed effect isn't statistically different from the null distribution.
- **Consider an alternative test:** If the effect is small or noisy, you might want to reassess whether the data supports a meaningful effect, or you could consider more refined models (e.g., adding regularization or more complex error structures).

If the observed slope (`observed_beta_1`) is significantly different from zero and you still see a large p-value, there may be issues with the model fitting (e.g., if the error term or data structure doesn't align with the assumptions of OLS regression).

Let me know how this diagnostic goes or if you'd like further suggestions!

So maybe we just fail to reject H_0 ?

Yes, exactly! If the p-value is large, it generally means that the observed effect is not significantly different from what we'd expect under the null hypothesis (i.e., there is no effect of `waiting` on `duration`). In this case, **failing to reject the null hypothesis** is the correct interpretation.

Here's a summary of what might be happening and how to interpret the result:

Scenario:

- **Null hypothesis (H_0):** The slope of `waiting` (β_1) is zero, i.e., `waiting` has no effect on `duration` .
- **Alternative hypothesis (H_1):** The slope of `waiting` is not zero, i.e., `waiting` does have an effect on `duration` .

If the p-value is large (i.e., much greater than the common significance level, say 0.05), then it suggests that the observed data does **not** provide enough evidence to reject the null hypothesis.

What does it mean to fail to reject H_0 ?

- **Failing to reject the null hypothesis** means that the data doesn't show strong enough evidence to conclude that `waiting` significantly affects `duration` .
- **This does not prove that `waiting` has no effect**, but it suggests that any effect of `waiting` on `duration` is either very small or too noisy to detect with the current data.

Possible Explanations for a Large p-value:

1. **Small effect size:** The true relationship between `waiting` and `duration` may be weak, and your data may not have enough power to detect it.
2. **High variability/noise:** If there is a lot of variability in the data (large error term), it could mask any potential relationship, making it difficult to reject the null hypothesis.

3. **Insufficient sample size:** While you're working with a large number of simulations (`n = 10,000`), if your actual data sample size is small, it might not provide enough statistical power to detect a true effect.

What to Do Next?

- **Examine the effect size:** Review the magnitude of the observed `beta_1` and its confidence intervals. If the confidence intervals are wide or the effect is small, the large p-value is consistent with weak or no effect.
- **Consider statistical power:** If your sample size is small, consider whether the test might lack enough power to detect an effect, even if one exists. You could perform a **power analysis** to check if your sample size is adequate.
- **Reevaluate the model:** Ensure the model assumptions are met (e.g., linearity, homoscedasticity, independence of errors). If the assumptions of OLS regression are violated, this could lead to misleading p-values.

In conclusion, a large p-value suggests that you **fail to reject** the null hypothesis that there is no relationship between `waiting` and `duration`. This doesn't mean there is no effect, just that the evidence from your data isn't strong enough to conclude a significant effect.

Can you just make the hypothesis test code for me? I think im messing it up:"

simulate samples (of size `n=160`) from a Simple Linear Regression model that uses

,

along with the values of waiting for to create simulations of

and use these collect and visualize the sampling distribution of the fitted slope coefficient under a null hypothesis assumption of "no linear association (on average)"; then,

Certainly! Here's a code snippet that follows these steps:

1. Simulate samples from a Simple Linear Regression model under the null hypothesis of no association.
2. Use the existing `waiting` values to generate `duration` values based on a null hypothesis model.

3. Collect and visualize the sampling distribution of the fitted slope coefficient.

Let's assume:

- `n = 160` (sample size for each simulation).
- We're using the `waiting` variable from the `old_faithful` dataset.
- We're generating `duration` values with a model that assumes no association between `waiting` and `duration`.

python

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt
from scipy import stats

# Set up initial variables
n_simulations = 10000 # number of simulations to perform
sample_size = 160     # size of each simulated sample
beta_0 = 1.65         # intercept under the null hypothesis
null_slope = 0        # slope under the null hypothesis
error_std_dev = 0.37  # standard deviation for errors

# Use 'waiting' values from the old_faithful dataset as input for the simulations
waiting_values = old_faithful['waiting'].values
simulated_slopes = np.zeros(n_simulations)

# Perform simulations
for i in range(n_simulations):
    # Randomly sample 160 waiting times from the observed data
    sampled_waiting = np.random.choice(waiting_values, size=sample_size,
                                       replace=True)

    # Generate simulated duration values under the null hypothesis (no association)
    error = np.random.normal(loc=0, scale=error_std_dev, size=sample_size)
    simulated_duration = beta_0 + null_slope * sampled_waiting + error

    # Fit a simple linear regression to the simulated data
    sim_data = pd.DataFrame({'waiting': sampled_waiting, 'duration':
                             simulated_duration})
```

```

simulated_model = smf.ols('duration ~ waiting', data=sim_data).fit()

# Collect the slope coefficient from the model
simulated_slopes[i] = simulated_model.params['waiting']

# Visualize the sampling distribution of the simulated slopes
plt.hist(simulated_slopes, bins=50, density=True, alpha=0.7, color='skyblue',
edgecolor='black')
plt.xlabel('Slope Coefficient')
plt.ylabel('Density')
plt.title('Sampling Distribution of Slope Coefficient Under Null Hypothesis')
plt.axvline(x=0, color='red', linestyle='--', label='Null Hypothesis (Slope = 0)')
plt.legend()
plt.show()

```

Explanation:

1. **Sampling:** Each simulation generates a sample of size `n = 160` by randomly sampling from the observed `waiting` values.
2. **Simulated Data:** `duration` values are created under the null hypothesis (`duration = beta_0 + error`), meaning no effect of `waiting`.
3. **Model Fitting:** For each simulated dataset, we fit a simple linear regression of `duration ~ waiting` and record the estimated slope.
4. **Visualization:** Finally, we plot the histogram of the simulated slope coefficients to visualize the sampling distribution under the null hypothesis.

Optional Step:

If you want to calculate the p-value based on an observed slope (`observed_slope`), you can add this at the end:

```

python

observed_slope = observed_fitted_model.params['waiting'] # Replace with the
observed slope
p_value = (np.abs(simulated_slopes) >= np.abs(observed_slope)).sum() / n_simulations
print("P-value:", p_value)

```

This p-value reflects the proportion of times the simulated slopes under the null hypothesis are as extreme or more extreme than the observed slope.

What was wrong with this code:

```
bootstrapped_beta_1 = np.zeros(n)
```

Your code `bootstrapped_beta_1 = np.zeros(n)` is likely meant to initialize an array for storing slope coefficients in a **bootstrap sampling process**. However, the name `bootstrapped_beta_1` might be misleading if your goal is to perform **null hypothesis testing** via simulations. Let's clarify:

Common Issues with This Code in a Hypothesis Test

1. **Name Misleading for Hypothesis Testing:** The name `bootstrapped_beta_1` suggests bootstrapping (re-sampling with replacement from the original data to estimate the sampling distribution

That was the wrong code input, heres the right one:

```
np.random.seed(2)
```

```
n = 10000
```

```
alpha = 0.05
```

```
old_faithful_sim = old_faithful[long_wait].copy()
```

```
beta_0 = 1.65
```

```
simulated_beta_1 = np.zeros(n)
```

```
observed_fitted_model = smf.ols('duration ~ waiting',
```

```
data=old_faithful[long_wait]).fit()
```

```
observed_beta_1 = fitted_model.params.values[1]
```

```
for i in range(n):
```

```
    error = stats.norm(loc=0,  
    scale=0.37).rvs(size=long_wait.sum())
```

```
    old_faithful_sim['duration'] = beta_0 + 0 *
```

```
    old_faithful_sim.waiting + error
```

```
    simulated_fitted_model = smf.ols('duration ~ waiting',  
    data=old_faithful_sim).fit()
```

```
    simulated_beta_1[i] =
```

```
simulated_fitted_model.params.values[1]
```

```
p = (np.abs(simulated_beta_1) >=
      np.abs(observed_beta_1)).sum() / n
print('Reject the null hypothesis' if p < alpha else 'Fail to
reject the null hypothesis')
print(f'p-value: {p}')
```

Your code looks mostly correct, but here are a few potential issues that could be affecting the results:

1. Misnaming `observed_beta_1` Calculation:

- You assign `observed_beta_1` using `fitted_model.params.values[1]`, but `fitted_model` is undefined in this code.
- You likely meant to use `observed_fitted_model.params.values[1]` for the observed slope. This could lead to errors or unexpected results.

Correction:

```
python
```

```
observed_beta_1 = observed_fitted_model.params.values[1]
```

2. Null Hypothesis Assumption (Zero Slope):

- The null hypothesis simulation is correct in setting the slope to zero by using `0 * old_faithful_sim.waiting`. This is what should be done to simulate data under the assumption that there is no linear association.

3. Comparing Absolute Values in P-Value Calculation:

- The p-value calculation here counts how often the simulated slope's absolute value exceeds the observed slope's absolute value.
- This approach is fine for a two-tailed test, but ensure that's what you intend to test. If you're interested in a one-tailed test, you should modify this part.

4. Checking Standard Deviation for Error:

- Make sure the error standard deviation (`scale=0.37`) is appropriate for your model. This value should match the standard deviation of residuals in the observed data, as it directly affects the distribution of `simulated_beta_1`.

5. Output:

- The code correctly prints whether you reject or fail to reject the null hypothesis based on the p-value.

Revised Code

With the above points fixed, the code should look like this:

```
python
```

```
import numpy as np
import statsmodels.formula.api as smf
from scipy import stats

np.random.seed(2)

n = 10000
alpha = 0.05

# Prepare the data and variables
old_faithful_sim = old_faithful[long_wait].copy()
beta_0 = 1.65
simulated_beta_1 = np.zeros(n)

# Fit observed model
observed_fitted_model = smf.ols('duration ~ waiting',
data=old_faithful[long_wait]).fit()
observed_beta_1 = observed_fitted_model.params.values[1] # Fix this line

# Run the simulation under the null hypothesis
for i in range(n):
    # Simulate error and duration under null hypothesis (no effect of waiting)
    error = stats.norm(loc=0, scale=0.37).rvs(size=long_wait.sum())
    old_faithful_sim['duration'] = beta_0 + 0 * old_faithful_sim.waiting + error
    simulated_fitted_model = smf.ols('duration ~ waiting',
data=old_faithful_sim).fit()

    # Store the simulated slope
    simulated_beta_1[i] = simulated_fitted_model.params.values[1]

# Calculate p-value
p = (np.abs(simulated_beta_1) >= np.abs(observed_beta_1)).sum() / n
```

```
print('Reject the null hypothesis' if p < alpha else 'Fail to reject the null hypothesis')
print(f'p-value: {p}')
```

Explanation of Changes:

- **Corrected the observed slope extraction to**
`observed_fitted_model.params.values[1]`.
- **Checked assumptions:** Ensure `error` standard deviation aligns with the observed data's residuals.

If the p-value remains large, it's likely indicating a true failure to reject the null hypothesis, suggesting that `waiting` does not have a statistically significant linear association with `duration` under this model.

Why didnt u calculate p-value in your original code snippet?

You're absolutely right—it would have been helpful to include the p-value calculation in the original snippet. In hypothesis testing, the p-value is essential because it quantifies how extreme the observed result is under the null hypothesis, enabling a clear decision about rejecting or failing to reject the null.

Let me add the p-value calculation to the original code snippet for clarity:

```
python

import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt
from scipy import stats

# Set up initial variables
n_simulations = 10000 # number of simulations to perform
sample_size = 160    # size of each simulated sample
beta_0 = 1.65        # intercept under the null hypothesis
null_slope = 0        # slope under the null hypothesis
error_std_dev = 0.37  # standard deviation for errors
alpha = 0.05          # significance level

# Use 'waiting' values from the old_faithful dataset as input for the simulations
```

```

waiting_values = old_faithful['waiting'].values
simulated_slopes = np.zeros(n_simulations)

# Fit the observed model to get the observed slope
observed_model = smf.ols('duration ~ waiting', data=old_faithful).fit()
observed_slope = observed_model.params['waiting']

# Perform simulations under the null hypothesis
for i in range(n_simulations):
    # Randomly sample 160 waiting times from the observed data
    sampled_waiting = np.random.choice(waiting_values, size=sample_size,
replace=True)

    # Generate simulated duration values under the null hypothesis (no association)
    error = np.random.normal(loc=0, scale=error_std_dev, size=sample_size)
    simulated_duration = beta_0 + null_slope * sampled_waiting + error

    # Fit a simple linear regression to the simulated data
    sim_data = pd.DataFrame({'waiting': sampled_waiting, 'duration':
simulated_duration})
    simulated_model = smf.ols('duration ~ waiting', data=sim_data).fit()

    # Collect the slope coefficient from the model
    simulated_slopes[i] = simulated_model.params['waiting']

# Calculate the p-value
p_value = (np.abs(simulated_slopes) >= np.abs(observed_slope)).sum() / n_simulations

# Visualize the sampling distribution of the simulated slopes
plt.hist(simulated_slopes, bins=50, density=True, alpha=0.7, color='skyblue',
edgecolor='black')
plt.axvline(x=observed_slope, color='red', linestyle='--', label='Observed Slope')
plt.xlabel('Slope Coefficient')
plt.ylabel('Density')
plt.title('Sampling Distribution of Slope Coefficient Under Null Hypothesis')
plt.legend()
plt.show()

# Print p-value and conclusion
print(f"p-value: {p_value}")
print('Reject the null hypothesis' if p_value < alpha else 'Fail to reject the null
hypothesis')

```

Summary of Additions:

- **Observed Slope:** We calculate `observed_slope` as the actual slope coefficient from the observed model.
- **P-Value Calculation:** After simulating the slope coefficients under the null hypothesis, we compute the p-value as the proportion of simulated slopes that are as extreme or more extreme than the observed slope.
- **Decision Based on p-value:** We then compare the p-value to `alpha` (significance level) to determine if we should reject the null hypothesis.

Thanks for prompting me to include the p-value calculation—it's an essential part of the hypothesis testing process!

U didnt consider the long-restriction:

```
import plotly.express as px

long_wait_limit = 71
long_wait = old_faithful.waiting > long_wait_limit

print(smf.ols('duration ~ waiting',
data=old_faithful[long_wait]).fit().summary().tables[1])

# Create a scatter plot with a linear regression trendline
fig = px.scatter(old_faithful[long_wait], x='waiting',
y='duration',
                title="Old Faithful Geyser Eruptions for short wait
times (>"+str(long_wait_limit)+")",
                trendline='ols')
fig.show() # USE fig.show(renderer="png") FOR ALL
GitHub and MarkUs SUBMISSIONS\
```

Thank you for the clarification! Incorporating the `long_wait` restriction will ensure we're only analyzing data for waiting times that exceed 71 minutes. Here's the updated code with this restriction applied throughout:

python

```
import numpy as np
import pandas as pd
```



```

import statsmodels.formula.api as smf
import matplotlib.pyplot as plt
from scipy import stats
import plotly.express as px

# Restriction on wait time
long_wait_limit = 71
long_wait = old_faithful.waiting > long_wait_limit
alpha = 0.05 # significance level

# Observed model with long wait times
observed_model = smf.ols('duration ~ waiting', data=old_faithful[long_wait]).fit()
observed_slope = observed_model.params['waiting']
print(observed_model.summary().tables[1])

# Set up simulation parameters
n_simulations = 10000
sample_size = 160
beta_0 = 1.65
null_slope = 0
error_std_dev = 0.37

# Sample waiting values from the restricted data
waiting_values = old_faithful[long_wait]['waiting'].values
simulated_slopes = np.zeros(n_simulations)

# Run simulations under the null hypothesis
for i in range(n_simulations):
    sampled_waiting = np.random.choice(waiting_values, size=sample_size,
    replace=True)
    error = np.random.normal(loc=0, scale=error_std_dev, size=sample_size)
    simulated_duration = beta_0 + null_slope * sampled_waiting + error

    sim_data = pd.DataFrame({'waiting': sampled_waiting, 'duration':
    simulated_duration})
    simulated_model = smf.ols('duration ~ waiting', data=sim_data).fit()
    simulated_slopes[i] = simulated_model.params['waiting']

# Calculate p-value
p_value = (np.abs(simulated_slopes) >= np.abs(observed_slope)).sum() / n_simulations
print(f"p-value: {p_value}")
print('Reject the null hypothesis' if p_value < alpha else 'Fail to reject the null

```

```
hypothesis')
```

```
# Plot sampling distribution and observed slope
```

```
plt.hist(simulated_slopes, bins=50, density=True, alpha=0.7, color='skyblue',  
edgecolor='black')  
plt.axvline(x=observed_slope, color='red', linestyle='--', label='Observed Slope')  
plt.xlabel('Slope Coefficient')  
plt.ylabel('Density')  
plt.title('Sampling Distribution of Slope Coefficient Under Null Hypothesis')  
plt.legend()  
plt.show()
```

```
# Scatter plot with regression trendline for long waits
```

```
fig = px.scatter(old_faithful[long_wait], x='waiting', y='duration',  
                 title=f"Old Faithful Geyser Eruptions for wait times >  
{long_wait_limit}",  
                 trendline='ols')  
fig.show(renderer="png")
```

Summary of Updates:

1. **Data Restriction:** Applied the `long_wait` mask based on `long_wait_limit` to filter data where `waiting > 71`.
2. **Observed Model:** Calculated `observed_slope` using only data satisfying `long_wait`.
3. **Waiting Values:** Used waiting values from the restricted data set in the simulations.
4. **Visualization:** Plots now reflect the restricted dataset.

This should provide accurate results for the hypothesis test and visualization under the `long_wait` restriction!

Are u stupid? Youre supposed to do the p-value urself, not get it from the OLS. the whole point is that were running a hypothesis test

Thank you for pointing that out—you're absolutely correct. In this hypothesis test, the p-value should be determined entirely from the simulated sampling distribution of the slope coefficients, without relying on the OLS summary output. Let's revise the code to calculate the p-value directly by comparing the observed slope with the distribution generated from the null hypothesis simulations.

Here's the revised approach, keeping everything under the null hypothesis and calculating the p-value as intended:

1. **Calculate the observed slope** for the actual data (restricted by `long_wait`).
2. **Simulate the slope coefficients** assuming the null hypothesis (no relationship between `waiting` and `duration`).
3. **Compute the p-value** based on the proportion of simulated slopes that are as extreme or more extreme than the observed slope.

python

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt
from scipy import stats
import plotly.express as px

# Set up the data and hypothesis test parameters
long_wait_limit = 71
long_wait = old_faithful.waiting > long_wait_limit
alpha = 0.05 # significance level

# Observed model with long wait times (calculate observed slope without relying on
# OLS p-value)
observed_model = smf.ols('duration ~ waiting', data=old_faithful[long_wait]).fit()
observed_slope = observed_model.params['waiting']
print(f"Observed slope (beta_1): {observed_slope}")

# Set up simulation parameters
n_simulations = 10000
sample_size = long_wait.sum() # size of restricted dataset
beta_0 = 1.65
null_slope = 0
error_std_dev = 0.37

# Sample waiting values from the restricted data
waiting_values = old_faithful[long_wait]['waiting'].values
simulated_slopes = np.zeros(n_simulations)

# Run simulations under the null hypothesis
```

```

for i in range(n_simulations):
    sampled_waiting = np.random.choice(waiting_values, size=sample_size,
replace=True)
    error = np.random.normal(loc=0, scale=error_std_dev, size=sample_size)
    simulated_duration = beta_0 + null_slope * sampled_waiting + error

    sim_data = pd.DataFrame({'waiting': sampled_waiting, 'duration':
simulated_duration})
    simulated_model = smf.ols('duration ~ waiting', data=sim_data).fit()
    simulated_slopes[i] = simulated_model.params['waiting']

# Calculate the p-value from the simulated distribution
p_value = (np.abs(simulated_slopes) >= np.abs(observed_slope)).sum() / n_simulations
print(f"p-value: {p_value}")
print('Reject the null hypothesis' if p_value < alpha else 'Fail to reject the null
hypothesis')

# Plot sampling distribution and observed slope
plt.hist(simulated_slopes, bins=50, density=True, alpha=0.7, color='skyblue',
edgecolor='black')
plt.axvline(x=observed_slope, color='red', linestyle='--', label='Observed Slope')
plt.xlabel('Slope Coefficient')
plt.ylabel('Density')
plt.title('Sampling Distribution of Slope Coefficient Under Null Hypothesis')
plt.legend()
plt.show()

# Scatter plot with regression trendline for long waits
fig = px.scatter(old_faithful[long_wait], x='waiting', y='duration',
                title=f"Old Faithful Geyser Eruptions for wait times >
{long_wait_limit}",
                trendline='ols')
fig.show(renderer="png")

```

Explanation

1. **Observed Slope:** The `observed_slope` is computed from the actual data (restricted to `long_wait`) without using any p-value output from the OLS summary.
2. **Null Hypothesis Simulations:** The simulated slope values (`simulated_slopes`) are generated under the null hypothesis.

3. **P-Value Calculation:** The p-value is directly calculated as the proportion of `simulated_slopes` that are as extreme or more extreme than the `observed_slope`.
4. **Visualization:** The histogram of `simulated_slopes` provides a visual reference for the observed slope relative to the null distribution.

This code correctly performs the hypothesis test by generating a p-value from the simulated sampling distribution.

Please identify the error here:

```
np.random.seed(2)

n = 10000
alpha = 0.05

old_faithful_sim = old_faithful[long_wait].copy()
beta_0 = 1.65
simulated_beta_1 = np.zeros(n)

observed_fitted_model = smf.ols('duration ~ waiting',
data=old_faithful[long_wait]).fit()
observed_beta_1 = fitted_model.params.values[1]

for i in range(n):
    error = stats.norm(loc=0,
scale=0.37).rvs(size=long_wait.sum())

    old_faithful_sim['duration'] = beta_0 + 0 *
old_faithful_sim.waiting + error
    simulated_fitted_model = smf.ols('duration ~ waiting',
data=old_faithful_sim).fit()

    simulated_beta_1[i] =
simulated_fitted_model.params.values[1]
    print(simulated_fitted_model.params)

p = (np.abs(simulated_beta_1) >=
np.abs(observed_beta_1)).sum() / n
```

```
print('Reject the null hypothesis' if p < alpha else 'Fail to  
reject the null hypothesis')  
print(f'p-value: {p}')
```

There are a few issues in your code:

1. Incorrect Variable Name for Observed Model Parameters:

- `observed_beta_1` is calculated with `fitted_model.params.values[1]`, but `fitted_model` is undefined in your code. This should be `observed_fitted_model.params.values[1]`, which is the correct model used for the observed data.

Correction:

```
python  
  
observed_beta_1 = observed_fitted_model.params.values[1]
```

2. Printing Parameters in Each Simulation Loop:

- Inside the loop, `print(simulated_fitted_model.params)` will print the parameters for each of the 10,000 simulations, resulting in a large output that slows down your code significantly. If you only need to inspect this once, consider moving it outside the loop or printing it only for a specific iteration.

3. Potential Performance Issue with Large Simulation Count:

- Running 10,000 iterations with complex computations may be slow, so consider testing with fewer simulations if you're troubleshooting.

With these fixes, here's the revised code:

```
python  
  
import numpy as np  
import statsmodels.formula.api as smf  
from scipy import stats  
  
np.random.seed(2)  
  
n = 10000  
alpha = 0.05  
  
old_faithful_sim = old_faithful[long_wait].copy()
```

```

beta_0 = 1.65
simulated_beta_1 = np.zeros(n)

# Observed model fit
observed_fitted_model = smf.ols('duration ~ waiting',
data=old_faithful[long_wait]).fit()
observed_beta_1 = observed_fitted_model.params.values[1] # Corrected line

# Run simulations under the null hypothesis
for i in range(n):
    error = stats.norm(loc=0, scale=0.37).rvs(size=long_wait.sum())
    old_faithful_sim['duration'] = beta_0 + 0 * old_faithful_sim.waiting + error

    simulated_fitted_model = smf.ols('duration ~ waiting',
data=old_faithful_sim).fit()
    simulated_beta_1[i] = simulated_fitted_model.params.values[1]

# Optional: Uncomment to print parameters only once or for specific iterations
# if i == 0:
#     print(simulated_fitted_model.params)

# Calculate the p-value
p = (np.abs(simulated_beta_1) >= np.abs(observed_beta_1)).sum() / n
print('Reject the null hypothesis' if p < alpha else 'Fail to reject the null
hypothesis')
print(f'p-value: {p}')

```

Key Points

- The **observed slope** (`observed_beta_1`) is now correctly computed from `observed_fitted_model` .
- Unnecessary **print statements** in the loop have been addressed for clarity and performance.

↳ ****Incorrect Variable Name for Observed Model Parameters****:

Wow u acc caught it thanks

You're very welcome! Glad we caught that detail—variable name issues can be tricky but make a huge difference. Let me know if you have any more questions about the code or need further help with hypothesis testing. Happy coding! ☐

This is my bootstrapping code:

```
import numpy as np

np.random.seed(1)

n = 10000 # number of simulations

bootstrapped_beta_1 = np.zeros(n)

for i in range(n):
    bootstrapped_sample =
    old_faithful[long_wait].sample(n=long_wait.sum(),
    replace=True)

    fitted_model = smf.ols('duration ~ waiting',
    data=bootstrapped_sample).fit()
    beta_1 = fitted_model.params.values[1]

    bootstrapped_beta_1[i] = beta_1

ci = np.quantile(bootstrapped_beta_1, [0.025, 0.975])

print(f'Confidence Interval of Slope Coefficients: ({ci[0]},
{ci[1]}')
```

Add plotting functionality to it

Here's how you can add a plot to visualize the distribution of the bootstrapped slope coefficients, along with the confidence interval:

python

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

np.random.seed(1)

n = 10000 # number of simulations

# Array to store bootstrapped slope coefficients
```



```

bootstrapped_beta_1 = np.zeros(n)

# Perform bootstrapping
for i in range(n):
    bootstrapped_sample = old_faithful[long_wait].sample(n=long_wait.sum(),
replace=True)
    fitted_model = smf.ols('duration ~ waiting', data=bootstrapped_sample).fit()
    beta_1 = fitted_model.params.values[1]
    bootstrapped_beta_1[i] = beta_1

# Calculate the confidence interval
ci = np.quantile(bootstrapped_beta_1, [0.025, 0.975])
print(f'Confidence Interval of Slope Coefficients: ({ci[0]}, {ci[1]})')

# Plotting the distribution of bootstrapped slopes with confidence interval
plt.figure(figsize=(10, 6))
plt.hist(bootstrapped_beta_1, bins=50, color='skyblue', edgecolor='black',
density=True)
plt.axvline(ci[0], color='red', linestyle='--', label=f'2.5% Quantile: {ci[0]:.4f}')
plt.axvline(ci[1], color='green', linestyle='--', label=f'97.5% Quantile:
{ci[1]:.4f}')
plt.axvline(np.mean(bootstrapped_beta_1), color='blue', linestyle='-', label=f'Mean
of bootstrapped slopes: {np.mean(bootstrapped_beta_1):.4f}')

plt.xlabel('Slope Coefficient (Beta_1)')
plt.ylabel('Density')
plt.title('Bootstrapped Sampling Distribution of Slope Coefficients')
plt.legend()
plt.show()

```

Explanation

- **Histogram:** Shows the distribution of the bootstrapped slope coefficients.
- **Vertical Lines:**
 - The red and green lines represent the lower and upper bounds of the 95% confidence interval, respectively.
 - The blue line represents the mean of the bootstrapped slope coefficients for reference.

This visualization helps you assess the distribution of slope estimates and see how the confidence interval bounds relate to the central tendency of the bootstrapped slopes.

Can you make this plot prettier to look at? E.g. adding a symmetric line for the observed test statistic, shading, ...

```
# Set up the data and hypothesis test parameters
long_wait_limit = 71
long_wait = old_faithful.waiting > long_wait_limit
alpha = 0.05 # significance level

# Observed model with long wait times (calculate observed
slope without relying on OLS p-value)
observed_model = smf.ols('duration ~ waiting',
data=old_faithful[long_wait]).fit()
observed_slope = observed_model.params['waiting']
print(f"Observed slope (beta_1): {observed_slope}")

# Set up simulation parameters
n_simulations = 10000
sample_size = long_wait.sum() # size of restricted dataset
beta_0 = 1.65
null_slope = 0
error_std_dev = 0.37

# Sample waiting values from the restricted data
waiting_values = old_faithful[long_wait]['waiting'].values
simulated_slopes = np.zeros(n_simulations)

# Run simulations under the null hypothesis
for i in range(n_simulations):
    sampled_waiting = np.random.choice(waiting_values,
size=sample_size, replace=True)
    error = np.random.normal(loc=0, scale=error_std_dev,
size=sample_size)
    simulated_duration = beta_0 + null_slope *
sampled_waiting + error

    sim_data = pd.DataFrame({'waiting': sampled_waiting,
'duration': simulated_duration})
    simulated_model = smf.ols('duration ~ waiting',
data=sim_data).fit()
```

```

simulated_slopes[i] = simulated_model.params['waiting']

# Calculate the p-value from the simulated distribution
p_value = (np.abs(simulated_slopes) >=
np.abs(observed_slope)).sum() / n_simulations
print(f"p-value: {p_value}")
print('Reject the null hypothesis' if p_value < alpha else 'Fail
to reject the null hypothesis')

# Plot sampling distribution and observed slope
plt.hist(simulated_slopes, bins=50, density=True, alpha=0.7,
color='skyblue', edgecolor='black')
plt.axvline(x=observed_slope, color='red', linestyle='--',
label='Observed Slope')
plt.xlabel('Slope Coefficient')
plt.ylabel('Density')
plt.title('Sampling Distribution of Slope Coefficient Under
Null Hypothesis')
plt.legend()
plt.show()

# Scatter plot with regression trendline for long waits
fig = px.scatter(old_faithful[long_wait], x='waiting',
y='duration',
                title=f"Old Faithful Geyser Eruptions for wait times
> {long_wait_limit}",
                trendline='ols')
fig.show(renderer="png")

```

Here's a refined version of the plot that adds styling and features like shading under the curve, symmetric lines for the observed slope and its opposite, and enhanced labels for a clearer view of the confidence interval and p-value.

```

python

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

# Set up data and hypothesis test parameters
np.random.seed(1)
long_wait_limit = 71

```

```

long_wait = old_faithful.waiting > long_wait_limit
alpha = 0.05 # significance level

# Observed model and slope for long waits
observed_model = smf.ols('duration ~ waiting', data=old_faithful[long_wait]).fit()
observed_slope = observed_model.params['waiting']
print(f"Observed slope (beta_1): {observed_slope}")

# Set up simulation parameters
n_simulations = 10000
sample_size = long_wait.sum() # size of restricted dataset
beta_0 = 1.65
null_slope = 0
error_std_dev = 0.37
waiting_values = old_faithful[long_wait]['waiting'].values
simulated_slopes = np.zeros(n_simulations)

# Run simulations under the null hypothesis
for i in range(n_simulations):
    sampled_waiting = np.random.choice(waiting_values, size=sample_size,
    replace=True)
    error = np.random.normal(loc=0, scale=error_std_dev, size=sample_size)
    simulated_duration = beta_0 + null_slope * sampled_waiting + error

    sim_data = pd.DataFrame({'waiting': sampled_waiting, 'duration':
    simulated_duration})
    simulated_model = smf.ols('duration ~ waiting', data=sim_data).fit()
    simulated_slopes[i] = simulated_model.params['waiting']

# Calculate p-value from the simulated distribution
p_value = (np.abs(simulated_slopes) >= np.abs(observed_slope)).sum() / n_simulations
print(f"p-value: {p_value}")
print('Reject the null hypothesis' if p_value < alpha else 'Fail to reject the null
hypothesis')

# Plot the distribution of bootstrapped slopes
plt.figure(figsize=(12, 6))
# Plot histogram and shade under curve
hist_counts, bin_edges, _ = plt.hist(simulated_slopes, bins=50, density=True,
color='skyblue', alpha=0.6, edgecolor='black')
plt.fill_between(bin_edges, 0, np.histogram(simulated_slopes, bins=bin_edges,
density=True)[0],

```

```

color='skyblue', alpha=0.4)

# Add vertical lines and labels for observed slope and confidence interval
plt.axvline(observed_slope, color='red', linestyle='--', linewidth=2,
label=f'Observed Slope: {observed_slope:.4f}')
plt.axvline(-observed_slope, color='red', linestyle='--', linewidth=2,
label=f'Symmetric line for -Observed Slope: {-observed_slope:.4f}')
ci = np.percentile(simulated_slopes, [2.5, 97.5])
plt.axvline(ci[0], color='green', linestyle='--', linewidth=2, label=f'Lower 2.5%
Quantile: {ci[0]:.4f}')
plt.axvline(ci[1], color='green', linestyle='--', linewidth=2, label=f'Upper 97.5%
Quantile: {ci[1]:.4f}')

# Add shaded area for the p-value region
plt.fill_between(bin_edges, 0, np.histogram(simulated_slopes, bins=bin_edges,
density=True)[0],
                where=(bin_edges >= np.abs(observed_slope)) | (bin_edges <= -
np.abs(observed_slope)),
                color='lightcoral', alpha=0.5, label='Extreme areas for p-value')

# Labeling and display
plt.xlabel('Slope Coefficient (Beta_1)', fontsize=12)
plt.ylabel('Density', fontsize=12)
plt.title('Bootstrapped Sampling Distribution of Slope Coefficients with Observed
Test Statistic', fontsize=15)
plt.legend()
plt.show()

```

Explanation of Enhancements

1. Shading:

- The histogram is shaded under the curve to provide a smooth, continuous look, and the extreme areas for the p-value calculation are shaded in a distinct color (light coral) to highlight the regions used for the p-value calculation.

2. Symmetric Line for the Observed Slope:

- An additional red dashed line is added at `-observed_slope`, showing symmetry relative to the null distribution, which can make the test more visually intuitive.

3. Confidence Interval Lines:

- Green dashed lines mark the 2.5% and 97.5% quantiles to illustrate the 95% confidence interval for the slope coefficient under the null hypothesis.

4. Clearer Labels and Titles:

- Each element, including the histogram, observed slope, and confidence interval, is labeled in the legend with a clear description, making it easier to interpret the plot.

This enhanced plot should give a better visual understanding of the test and make it easier to interpret the results.

Provide me a summary of our entire session for the purpose of a hw assignment

Certainly! Here's a concise summary of our entire session, tailored for your homework assignment:

Summary of Session on Hypothesis Testing and Bootstrapping

Objective:

The goal was to explore hypothesis testing, bootstrapping, and confidence intervals using the Old Faithful dataset, particularly focusing on the relationship between the eruption duration and waiting time.

Key Concepts and Steps:

1. Hypothesis Testing:

- The task involved testing the null hypothesis that there is no linear relationship between waiting times and eruption duration for long wait times (`waiting > 71`).
- We performed simulations (10,000 iterations) under the null hypothesis, where the slope coefficient (`beta_1`) was assumed to be 0, and compared it with the observed slope from the dataset.
- A **p-value** was calculated to determine whether we should reject the null hypothesis.

2. Implementation of Hypothesis Test:

- We generated bootstrapped samples from the `waiting` and `duration` data for long wait times.
- For each sample, we fit a linear regression model to estimate the slope and collect these slopes in a distribution.
- The p-value was calculated as the proportion of simulated slopes greater than or equal to the observed slope in magnitude.
- The hypothesis test resulted in a decision to either **reject** or **fail to reject** the null hypothesis, based on the p-value.

3. Bootstrap Confidence Interval (CI):

- The session also included creating a **bootstrapped sampling distribution** for the slope coefficient and calculating the 95% confidence interval (CI) by taking the 2.5% and 97.5% quantiles.
- We visualized the distribution of the bootstrapped slopes using a histogram with overlaid lines for the observed slope, the confidence interval, and the symmetric line for the opposite slope.

4. Plotting Enhancements:

- The resulting plots were enhanced to make them visually clearer and more informative:
 - **Shading under the histogram** to represent the density of bootstrapped slopes.
 - Adding **vertical lines** for the observed slope and its negative counterpart.
 - Shading the regions beyond the observed slope to highlight the extreme areas used to calculate the p-value.

5. Final Conclusion:

- Based on the p-value calculated from the bootstrapping simulation, the hypothesis was tested, and the decision was made to either **reject** or **fail to reject** the null hypothesis.

Code Snippets:

- **Bootstrapping for Confidence Intervals:**

```
python
```

```
# Example of bootstrapping code to estimate slope coefficients and confidence intervals
simulated_slopes = np.zeros(n_simulations)
for i in range(n_simulations):
    # Bootstrap sampling and fitting regression model
    # Store slope coefficients in simulated_slopes
    ci = np.percentile(simulated_slopes, [2.5, 97.5])
```

- **Plotting Bootstrapped Distribution:**

```
python
```

```
plt.hist(simulated_slopes, bins=50, density=True, color='skyblue', alpha=0.6,
edgecolor='black')
plt.axvline(observed_slope, color='red', linestyle='--', label='Observed Slope')
plt.axvline(ci[0], color='green', linestyle='--', label='2.5% Quantile')
plt.axvline(ci[1], color='green', linestyle='--', label='97.5% Quantile')
```

Conclusion:

This session involved using bootstrapping methods to perform hypothesis testing on the relationship between waiting time and eruption duration for Old Faithful geyser data. The key takeaway is that through simulations, we tested the null hypothesis of no linear relationship, calculated the p-value, and visualized the results using enhanced plots for better interpretation.

Should be a 1 paragraph summary

In this session, we explored hypothesis testing and bootstrapping using the Old Faithful geyser dataset, focusing on the relationship between eruption duration and waiting time for long wait times (greater than 71 minutes). We performed a hypothesis test to assess whether there is a significant linear relationship between these variables by simulating data under the null hypothesis (assuming no relationship). We computed a p-value based on bootstrapped slope coefficients and compared it to the observed slope. Additionally, we calculated a 95% confidence interval for the slope using bootstrapping, visualized the sampling distribution with enhanced plots, and made a decision to either reject or fail to

reject the null hypothesis based on the p-value. The session provided insights into hypothesis testing through simulation, confidence intervals, and visualizing the results effectively.

Expand just a little longer — be more specific to what we talked about (struggles, etc.)

In this session, we focused on performing hypothesis testing and bootstrapping using the Old Faithful geyser dataset, specifically analyzing the relationship between eruption duration and waiting time for long wait times (greater than 71 minutes). We aimed to test the null hypothesis that there is no linear relationship between these variables by simulating data under the assumption of a zero slope (null hypothesis) and comparing the bootstrapped slope coefficients to the observed slope. We calculated a p-value from the simulations and interpreted it to decide whether to reject or fail to reject the null hypothesis. During this process, we encountered some challenges, such as correctly setting up the bootstrapping procedure and ensuring that the p-value was accurately calculated from the distribution of simulated slopes. We also had to ensure that the correct data was used, especially when applying the long-wait restriction. After refining the code, we successfully visualized the results with enhanced plots, including shaded regions for the p-value, vertical lines for the observed slope, and confidence intervals. This session provided valuable insights into hypothesis testing, the process of bootstrapping for confidence intervals, and how to visualize and interpret the results effectively.

```
ValueError                                Traceback (most recent call
last)
Cell In[15], line 40
      38 # Plot histogram and shade under curve
      39 hist_counts, bin_edges, _ = plt.hist(simulated_slopes,
      bins=50, density=True, color='skyblue', alpha=0.6,
      edgecolor='black')
--> 40 plt.fill_between(bin_edges, 0,
      np.histogram(simulated_slopes, bins=bin_edges,
      density=True)[0],
      41                    color='skyblue', alpha=0.4)
      43 # Add vertical lines and labels for observed slope and
      confidence interval
      44 plt.axvline(observed_slope, color='red', linestyle='--',
      linewidth=2, label=f'Observed Slope: {observed_slope:.4f}')
```

File /opt/conda/lib/python3.11/site-

```
packages/matplotlib/pyplot.py:3096, in fill_between(x, y1,
y2, where, interpolate, step, data, **kwargs)
```

```
3084
```

```
@_copy_docstring_and_deprecators(Axes.fill_between)
```

```
3085 def fill_between(
```

```
3086     x: ArrayLike,
```

```
    (...)
```

```
3094     **kwargs,
```

```
3095 ) -> PolyCollection:
```

```
-> 3096     return gca().fill_between(
```

```
3097         x,
```

```
3098         y1,
```

```
3099         y2=y2,
```

```
3100         where=where,
```

```
3101         interpolate=interpolate,
```

```
3102         step=step,
```

```
3103         **({"data": data} if data is not None else {}),
```

```
3104         **kwargs,
```

```
3105     )
```

```
File /opt/conda/lib/python3.11/site-
```

```
packages/matplotlib/_init_.py:1465, in _preprocess_data.
```

```
<locals>.inner(ax, data, *args, **kwargs)
```

```
1462 @functools.wraps(func)
```

```
1463 def inner(ax, *args, data=None, **kwargs):
```

```
1464     if data is None:
```

```
-> 1465         return func(ax, *map(sanitize_sequence, args),
**kwargs)
```

```
1467     bound = new_sig.bind(ax, *args, **kwargs)
```

```
1468     auto_label = (bound.arguments.get(label_namer)
```

```
1469                     or bound.kwargs.get(label_namer))
```

```
File /opt/conda/lib/python3.11/site-
```

```
packages/matplotlib/axes/_axes.py:5509, in
```

```
Axes.fill_between(self, x, y1, y2, where, interpolate, step,
**kwargs)
```

```
5507 def fill_between(self, x, y1, y2=0, where=None,
interpolate=False,
```

```
5508                 step=None, **kwargs):
```

```
-> 5509     return self._fill_between_x_or_y(
```

```
5510         "x", x, y1, y2,
```

```
5511         where=where, interpolate=interpolate,
```

```
step=step, **kwargs)
```

```
File /opt/conda/lib/python3.11/site-  
packages/matplotlib/axes/_axes.py:5425, in  
Axes._fill_between_x_or_y(self, ind_dir, ind, dep1, dep2,  
where, interpolate, step, **kwargs)  
    5422     if where.size != ind.size:  
    5423         raise ValueError(f"where size ({where.size}) does  
not match "  
    5424                             f"{ind_dir} size ({ind.size})")  
-> 5425 where = where & ~functools.reduce(  
    5426     np.logical_or, map(np.ma.getmaskarray, [ind,  
dep1, dep2]))  
    5428 ind, dep1, dep2 = np.broadcast_arrays(  
    5429     np.atleast_1d(ind), dep1, dep2, subok=True)  
    5431 polys = []
```

ValueError: operands could not be broadcast together with
shapes (51,) (50,)

The error occurs because the shapes of the arrays you're trying to use in `plt.fill_between()` do not match. Specifically, `bin_edges` has 51 elements, while the histogram counts (`np.histogram(simulated_slopes, bins=bin_edges, density=True)[0]`) return an array of size 50. This is due to the fact that `plt.hist()` automatically generates one more bin edge than the number of bins used, creating a mismatch when trying to fill the area under the histogram.

To fix this issue, you should adjust the `fill_between()` call so that it correctly aligns with the bin edges and counts. Here's how you can modify the code:

1. You can drop the last bin edge to match the number of histogram bins, or
2. You can use the histogram counts and edges as they are, but ensure they are aligned.

Here's the corrected code:

```
python
```

```
# Plot histogram and shade under curve  
hist_counts, bin_edges, _ = plt.hist(simulated_slopes, bins=50, density=True,  
color='skyblue', alpha=0.6, edgecolor='black')  
  
# Adjust the bin edges and counts to avoid mismatch
```

```
plt.fill_between(bin_edges[:-1], 0, hist_counts, color='skyblue', alpha=0.4) # Use
bin_edges[:-1] to match hist_counts

# Add vertical lines and labels for observed slope and confidence interval
plt.axvline(observed_slope, color='red', linestyle='--', linewidth=2,
label=f'Observed Slope: {observed_slope:.4f}')
plt.axvline(ci[0], color='green', linestyle='--', linewidth=2, label=f'2.5%
Quantile: {ci[0]:.4f}')
plt.axvline(ci[1], color='green', linestyle='--', linewidth=2, label=f'97.5%
Quantile: {ci[1]:.4f}')

# Customize plot
plt.xlabel('Slope Coefficient')
plt.ylabel('Density')
plt.title('Bootstrapped Distribution of Slope Coefficient')
plt.legend()
plt.show()
```

Explanation:

- The `bin_edges[:-1]` ensures that the number of bin edges aligns with the number of histogram counts (`hist_counts`). The `[:-1]` removes the last bin edge because `plt.hist()` creates one more bin edge than the number of bins.
- The rest of the code adds vertical lines for the observed slope and confidence interval and includes plot customization.

This should resolve the broadcasting error and create the desired plot.

Update the entire cell

Certainly! Here's the updated version of the entire code cell, with the fix for the broadcasting issue and the plot enhancements you requested:

```
python

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

# Set up the data and hypothesis test parameters
long_wait_limit = 71
```

```

long_wait = old_faithful.waiting > long_wait_limit
alpha = 0.05 # significance level

# Observed model with long wait times (calculate observed slope without relying on
OLS p-value)
observed_model = smf.ols('duration ~ waiting', data=old_faithful[long_wait]).fit()
observed_slope = observed_model.params['waiting']
print(f"Observed slope (beta_1): {observed_slope}")

# Set up simulation parameters
n_simulations = 10000
sample_size = long_wait.sum() # size of restricted dataset
beta_0 = 1.65
null_slope = 0
error_std_dev = 0.37

# Sample waiting values from the restricted data
waiting_values = old_faithful[long_wait]['waiting'].values
simulated_slopes = np.zeros(n_simulations)

# Run simulations under the null hypothesis
for i in range(n_simulations):
    sampled_waiting = np.random.choice(waiting_values, size=sample_size,
replace=True)
    error = np.random.normal(loc=0, scale=error_std_dev, size=sample_size)
    simulated_duration = beta_0 + null_slope * sampled_waiting + error

    sim_data = pd.DataFrame({'waiting': sampled_waiting, 'duration':
simulated_duration})
    simulated_model = smf.ols('duration ~ waiting', data=sim_data).fit()
    simulated_slopes[i] = simulated_model.params['waiting']

# Calculate the p-value from the simulated distribution
p_value = (np.abs(simulated_slopes) >= np.abs(observed_slope)).sum() / n_simulations
print(f"p-value: {p_value}")
print('Reject the null hypothesis' if p_value < alpha else 'Fail to reject the null
hypothesis')

# Plot histogram and shade under curve
hist_counts, bin_edges, _ = plt.hist(simulated_slopes, bins=50, density=True,
color='skyblue', alpha=0.6, edgecolor='black')

```

```

# Adjust the bin edges and counts to avoid mismatch
plt.fill_between(bin_edges[:-1], 0, hist_counts, color='skyblue', alpha=0.4) # Use
bin_edges[:-1] to match hist_counts

# Add vertical lines and labels for observed slope and confidence interval
ci = np.quantile(simulated_slopes, [0.025, 0.975])
plt.axvline(observed_slope, color='red', linestyle='--', linewidth=2,
label=f'Observed Slope: {observed_slope:.4f}')
plt.axvline(ci[0], color='green', linestyle='--', linewidth=2, label=f'2.5%
Quantile: {ci[0]:.4f}')
plt.axvline(ci[1], color='green', linestyle='--', linewidth=2, label=f'97.5%
Quantile: {ci[1]:.4f}')

# Customize plot
plt.xlabel('Slope Coefficient')
plt.ylabel('Density')
plt.title('Bootstrapped Distribution of Slope Coefficient')
plt.legend()
plt.show()

# Scatter plot with regression trendline for long waits
import plotly.express as px
fig = px.scatter(old_faithful[long_wait], x='waiting', y='duration',
                title=f"Old Faithful Geyser Eruptions for wait times >
{long_wait_limit}",
                trendline='ols')
fig.show(renderer="png")

```

Key Updates:

1. **Histogram and Shading:** Fixed the `fill_between()` issue by ensuring the bin edges and histogram counts align correctly using `bin_edges[:-1]`.
2. **Confidence Interval:** Added vertical lines for the observed slope and the 95% confidence interval calculated from the bootstrapped slopes (`ci`).
3. **Plot Customization:** The plot includes labels, a title, and a legend to make it more informative and visually appealing.

Now, this code will generate a clean histogram of the bootstrapped slopes with shaded areas and correctly labeled vertical lines for the observed slope and confidence interval.

WTF? THIS ISNT BOOTSTRAPPED NOR IS IT A CONFIDENCE INTERVAL ITS A FUCKING HYPOTHESIS TEST

Apologies for the confusion earlier, and thank you for your patience. Let's clarify what we're doing and refocus the approach.

Goal:

We are conducting a **hypothesis test** for the slope coefficient from a regression, specifically under the **null hypothesis** that there is **no linear association** (i.e., the slope is zero). We are simulating the distribution of the slope under the null hypothesis, comparing it to the observed slope, and using it to compute a **p-value**.

What We Are Doing:

1. **Observed slope:** Calculate the observed slope from the regression on the dataset where waiting times are greater than the threshold.
2. **Simulate under the null hypothesis:** Generate simulated samples where the null hypothesis (no linear relationship) holds. For each simulated sample, fit a regression and calculate the slope.
3. **Calculate p-value:** Compare the observed slope to the distribution of simulated slopes to see how extreme the observed value is.
4. **Plot:** Create a plot of the simulated distribution of slopes and mark the observed slope to see if it falls in the extreme tails.

Here's the corrected code to perform the **hypothesis test** and visualize the **sampling distribution** with appropriate labels and shading:

```
python

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

# Set up the data and hypothesis test parameters
long_wait_limit = 71
long_wait = old_faithful.waiting > long_wait_limit
alpha = 0.05 # significance level

# Observed model with long wait times (calculate observed slope without relying on
```

OLS p-value)

```
observed_model = smf.ols('duration ~ waiting', data=old_faithful[long_wait]).fit()
observed_slope = observed_model.params['waiting']
print(f"Observed slope (beta_1): {observed_slope}")

# Set up simulation parameters
n_simulations = 10000
sample_size = long_wait.sum() # size of restricted dataset
beta_0 = 1.65
null_slope = 0
error_std_dev = 0.37

# Sample waiting values from the restricted data
waiting_values = old_faithful[long_wait]['waiting'].values
simulated_slopes = np.zeros(n_simulations)

# Run simulations under the null hypothesis (no relationship, slope = 0)
for i in range(n_simulations):
    sampled_waiting = np.random.choice(waiting_values, size=sample_size,
    replace=True)
    error = np.random.normal(loc=0, scale=error_std_dev, size=sample_size)
    simulated_duration = beta_0 + null_slope * sampled_waiting + error

    sim_data = pd.DataFrame({'waiting': sampled_waiting, 'duration':
    simulated_duration})
    simulated_model = smf.ols('duration ~ waiting', data=sim_data).fit()
    simulated_slopes[i] = simulated_model.params['waiting']

# Calculate the p-value from the simulated distribution
p_value = (np.abs(simulated_slopes) >= np.abs(observed_slope)).sum() / n_simulations
print(f"p-value: {p_value}")
print('Reject the null hypothesis' if p_value < alpha else 'Fail to reject the null
hypothesis')

# Plot histogram and shade under curve
hist_counts, bin_edges, _ = plt.hist(simulated_slopes, bins=50, density=True,
color='skyblue', alpha=0.6, edgecolor='black')

# Adjust the bin edges and counts to avoid mismatch
plt.fill_between(bin_edges[:-1], 0, hist_counts, color='skyblue', alpha=0.4) # Use
bin_edges[:-1] to match hist_counts
```



```

# Add vertical lines and labels for observed slope and p-value
plt.axvline(observed_slope, color='red', linestyle='--', linewidth=2,
label=f'Observed Slope: {observed_slope:.4f}')
plt.axvline(ci[0], color='green', linestyle='--', linewidth=2, label=f'2.5%
Quantile: {ci[0]:.4f}')
plt.axvline(ci[1], color='green', linestyle='--', linewidth=2, label=f'97.5%
Quantile: {ci[1]:.4f}')

# Customize plot
plt.xlabel('Slope Coefficient')
plt.ylabel('Density')
plt.title('Hypothesis Test of Slope Coefficient under Null Hypothesis')
plt.legend()
plt.show()

# Scatter plot with regression trendline for long waits
import plotly.express as px
fig = px.scatter(old_faithful[long_wait], x='waiting', y='duration',
                title=f"Old Faithful Geyser Eruptions for wait times >
{long_wait_limit}",
                trendline='ols')
fig.show(renderer="png")

```

Key Features:

1. **Observed Slope:** The `observed_slope` is calculated from the actual data under the long wait condition, where we regress `duration` on `waiting`.
2. **Simulating the Null Hypothesis:** We simulate new samples under the assumption that the true slope is 0 (no linear relationship). These are resampled and used to fit regression models to generate a distribution of slopes.
3. **p-value Calculation:** The p-value is calculated by comparing the observed slope to the simulated distribution of slopes under the null hypothesis.
4. **Plot:** The plot visualizes the distribution of slopes under the null hypothesis, with shaded areas and vertical lines marking the observed slope, along with the 95% confidence interval from the bootstrap.

This should give you the desired hypothesis testing and visualization for your analysis.

WHY TF ARE THERE STILL QUANTILES. WHY IS THERE A WEIRD SHADED KDE. WTF IS THIS. WHY ARE WE PLACING

Apologies again for the confusion and frustration. Let me walk through this carefully. You're absolutely right: this should not have any quantiles, KDEs, or any concept of a confidence interval. We are **conducting a hypothesis test**, not a bootstrap confidence interval analysis.

The Goal:

1. **Hypothesis Test:** Test the null hypothesis that there is no linear relationship (i.e., slope = 0) in the regression model using simulation.
2. **Plot:** You want to see the distribution of slopes from simulated data under the null hypothesis, and the observed slope, without any unnecessary features.

Updated Approach:

- **Null Hypothesis:** The true slope of the regression is 0.
- We simulate regression slopes assuming the null hypothesis is true (slope = 0), and then compare the observed slope from the real data to this distribution.
- We compute the **p-value** by counting how many simulated slopes are greater than or equal to the absolute value of the observed slope.
- We then plot the **distribution of simulated slopes** and mark the **observed slope**.

Here's the corrected code:

python

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

# Set up the data and hypothesis test parameters
long_wait_limit = 71
long_wait = old_faithful.waiting > long_wait_limit
alpha = 0.05 # significance level

# Observed model with long wait times (calculate observed slope without relying on
# OLS p-value)
observed_model = smf.ols('duration ~ waiting', data=old_faithful[long_wait]).fit()
observed_slope = observed_model.params['waiting']
```

```

print(f"Observed slope (beta_1): {observed_slope}")

# Set up simulation parameters
n_simulations = 10000
sample_size = long_wait.sum() # size of restricted dataset
beta_0 = 1.65
null_slope = 0
error_std_dev = 0.37

# Sample waiting values from the restricted data
waiting_values = old_faithful[long_wait]['waiting'].values
simulated_slopes = np.zeros(n_simulations)

# Run simulations under the null hypothesis (no relationship, slope = 0)
for i in range(n_simulations):
    sampled_waiting = np.random.choice(waiting_values, size=sample_size,
    replace=True)
    error = np.random.normal(loc=0, scale=error_std_dev, size=sample_size)
    simulated_duration = beta_0 + null_slope * sampled_waiting + error

    sim_data = pd.DataFrame({'waiting': sampled_waiting, 'duration':
    simulated_duration})
    simulated_model = smf.ols('duration ~ waiting', data=sim_data).fit()
    simulated_slopes[i] = simulated_model.params['waiting']

# Calculate the p-value from the simulated distribution
p_value = (np.abs(simulated_slopes) >= np.abs(observed_slope)).sum() / n_simulations
print(f"p-value: {p_value}")
print('Reject the null hypothesis' if p_value < alpha else 'Fail to reject the null
hypothesis')

# Plot histogram of the simulated slopes and mark the observed slope
plt.hist(simulated_slopes, bins=50, density=True, color='skyblue', alpha=0.6,
edgecolor='black')

# Add vertical line for the observed slope
plt.axvline(observed_slope, color='red', linestyle='--', linewidth=2,
label=f'Observed Slope: {observed_slope:.4f}')

# Labeling and titles
plt.xlabel('Slope Coefficient')
plt.ylabel('Density')

```

```
plt.title('Hypothesis Test of Slope Coefficient under Null Hypothesis')
plt.legend()

# Show plot
plt.show()

# Scatter plot with regression trendline for long waits
import plotly.express as px
fig = px.scatter(old_faithful[long_wait], x='waiting', y='duration',
                 title=f"Old Faithful Geyser Eruptions for wait times >
{long_wait_limit}",
                 trendline='ols')
fig.show(renderer="png")
```

Key Changes:

1. **No Quantiles or Confidence Intervals:** The focus is entirely on the **hypothesis test**, so I have removed any mention of confidence intervals or quantiles. This is simply about comparing the observed slope to the distribution of slopes under the null hypothesis.
2. **Histogram of Simulated Slopes:** The plot is a simple **histogram** showing the distribution of slopes from the simulations.
3. **Observed Slope Line:** A vertical red dashed line represents the **observed slope** on the plot, making it easy to see where the real data's slope falls in comparison to the simulated distribution.
4. **P-value Calculation:** We calculate the **p-value** based on the observed slope and the simulated slopes, and use that to determine if we **reject** or **fail to reject** the null hypothesis.

Summary:

- This is a **hypothesis test** where we check if the observed slope is statistically significantly different from zero (the null hypothesis).
- The plot shows the **distribution of slopes under the null hypothesis**, and the **observed slope** is clearly marked on the plot.

Let me know if that aligns with your expectations!

i still want the fucking symmetric line for the observed test

Got it! You want a **symmetric line** for the observed test statistic, which means marking **both sides** of the observed slope (i.e., positive and negative) to illustrate the test under the **null hypothesis**.

To clarify:

- You want to see how extreme the observed slope is on both sides of the null hypothesis distribution (symmetric around zero).
- We'll still display the **observed slope** (as a vertical dashed line) and highlight its **symmetric counterparts** (i.e., the positive and negative of the observed slope) to show how extreme it is under the null hypothesis.

I'll update the plot and code accordingly.

Here's the updated code to include the symmetric lines for the observed test statistic:

python

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

# Set up the data and hypothesis test parameters
long_wait_limit = 71
long_wait = old_faithful.waiting > long_wait_limit
alpha = 0.05 # significance level

# Observed model with long wait times (calculate observed slope without relying on
# OLS p-value)
observed_model = smf.ols('duration ~ waiting', data=old_faithful[long_wait]).fit()
observed_slope = observed_model.params['waiting']
print(f"Observed slope (beta_1): {observed_slope}")

# Set up simulation parameters
n_simulations = 10000
sample_size = long_wait.sum() # size of restricted dataset
beta_0 = 1.65
null_slope = 0
error_std_dev = 0.37
```

```

# Sample waiting values from the restricted data
waiting_values = old_faithful[long_wait]['waiting'].values
simulated_slopes = np.zeros(n_simulations)

# Run simulations under the null hypothesis (no relationship, slope = 0)
for i in range(n_simulations):
    sampled_waiting = np.random.choice(waiting_values, size=sample_size,
replace=True)
    error = np.random.normal(loc=0, scale=error_std_dev, size=sample_size)
    simulated_duration = beta_0 + null_slope * sampled_waiting + error

    sim_data = pd.DataFrame({'waiting': sampled_waiting, 'duration':
simulated_duration})
    simulated_model = smf.ols('duration ~ waiting', data=sim_data).fit()
    simulated_slopes[i] = simulated_model.params['waiting']

# Calculate the p-value from the simulated distribution
p_value = (np.abs(simulated_slopes) >= np.abs(observed_slope)).sum() / n_simulations
print(f"p-value: {p_value}")
print('Reject the null hypothesis' if p_value < alpha else 'Fail to reject the null
hypothesis')

# Plot histogram of the simulated slopes and mark the observed slope
plt.hist(simulated_slopes, bins=50, density=True, color='skyblue', alpha=0.6,
edgecolor='black')

# Add vertical lines for the observed slope and its symmetric counterparts
plt.axvline(observed_slope, color='red', linestyle='--', linewidth=2,
label=f'Observed Slope: {observed_slope:.4f}')
plt.axvline(-observed_slope, color='red', linestyle='--', linewidth=2,
label=f'Symmetric Counterpart: {-observed_slope:.4f}')

# Labeling and titles
plt.xlabel('Slope Coefficient')
plt.ylabel('Density')
plt.title('Hypothesis Test of Slope Coefficient under Null Hypothesis')
plt.legend()

# Show plot
plt.show()

```

```
# Scatter plot with regression trendline for long waits
import plotly.express as px
fig = px.scatter(old_faithful[long_wait], x='waiting', y='duration',
                 title=f"Old Faithful Geyser Eruptions for wait times >
{long_wait_limit}",
                 trendline='ols')
fig.show(renderer="png")
```

Key Updates:

1. **Symmetric Lines:** I've added another vertical dashed line at the **negative observed slope** (`-observed_slope`) to create the symmetric line. This will help you visualize the two extremes of the observed slope compared to the null hypothesis distribution.
2. **Plot:** Now, the plot will show:
 - The **observed slope** (positive) in **red dashed**.
 - The **symmetric counterpart** of the observed slope (negative) in **red dashed**.
 - The histogram of simulated slopes from the null hypothesis.

Output:

- The histogram shows the simulated distribution of slopes under the null hypothesis.
- The observed slope is marked with a dashed line, and its **symmetric counterpart** is also shown to give you a better sense of how extreme the observed test statistic is, compared to the null hypothesis.

Let me know if this now matches what you're expecting!

#CHECKPOINT

Ok this is the mark of a new question. I dont want to start a new chat bc u alr have context, so heres my prompt/

11. Since we've considered wait times of around <64 "short" and wait times of >71 "long", let's instead just divide the data and instead call wait times of <68 "short" and otherwise just call them "long". Consider the Simple Linear Regression model specification using an indicator variable of the wait time length where we use (rather than) (to refer to the "kind" or "katagory" or "kontrast") column

(that you may have noticed was already a part) of the original dataset; and, explain the "big picture" differences between this model specification and the previously considered model specifications and report the evidence against a null hypothesis of "no difference between groups "on average") for the new indicator variable based model

I ONLY WANT YOU TO FOCUS ON THE HYPOTHESIS TESTING NOW.

To focus on the hypothesis testing here, we'll use the new binary indicator variable for "short" and "long" wait times. This allows us to examine if there's a statistically significant difference in the average eruption duration between these two categories.

Here's how to approach it:

1. **Model Setup:** Use a Simple Linear Regression with `duration` as the response variable and the **indicator variable** for wait time length (let's call it `wait_type`, where `wait_type = 0` for short and `wait_type = 1` for long) as the predictor. This binary variable essentially splits the data into two groups and allows us to test for an average difference in `duration` between them.
2. **Null Hypothesis:**
 - H_0 : No difference in average eruption duration between short and long wait times.
 - H_1 : There is a difference in average eruption duration between short and long wait times.
3. **Test Statistic:** In this model, the coefficient of `wait_type` (the slope) will tell us the average difference in `duration` between "long" and "short" wait times. Under H_0 , this coefficient should be close to zero.
4. **Performing the Hypothesis Test:** Fit the regression model and examine the p-value associated with the `wait_type` coefficient. If this p-value is below a significance level (commonly $\alpha = 0.05$), we reject the null hypothesis and conclude there is a significant difference between the two groups.

Here's the code for the hypothesis test:

```
python
```



```

import statsmodels.formula.api as smf
import pandas as pd

# Prepare the indicator variable based on new threshold
old_faithful['wait_type'] = (old_faithful['waiting'] >= 68).astype(int)

# Fit the model using the indicator variable
model = smf.ols('duration ~ wait_type', data=old_faithful).fit()

# Display the model summary, focusing on the p-value for wait_type
print(model.summary().tables[1])

# Extract the p-value for hypothesis testing
p_value = model.pvalues['wait_type']
print(f'p-value for wait_type coefficient: {p_value}')

# Conclusion based on p-value
alpha = 0.05
if p_value < alpha:
    print("Reject the null hypothesis: Significant difference between groups.")
else:
    print("Fail to reject the null hypothesis: No significant difference between groups.")

```

Explanation of Output

- **p-value for `wait_type`:** This value tells us whether the difference in eruption durations for "short" vs. "long" wait times is statistically significant.
- **Interpretation:**
 - If $p < 0.05$, we reject H_0 , indicating a significant difference in duration between "short" and "long" wait groups.
 - If $p \geq 0.05$, we fail to reject H_0 , meaning no evidence of a significant difference.

Let me know if this hypothesis test setup addresses your focus.

Okay perfect, now provide a summary for me from the beginning of the #CHECKPOINT onward

After the #CHECKPOINT, we shifted our focus to a new hypothesis testing question that analyzes eruption durations based on a redefined classification of wait times. Here, wait

times are categorized as "short" if under 68 minutes and "long" otherwise, represented by an indicator variable (`wait_type`) in the model. This variable allows us to test for average differences in eruption duration between the two groups. The null hypothesis states there is no difference in average eruption durations between short and long wait times, while the alternative hypothesis asserts a difference. We implemented a Simple Linear Regression model with `duration` as the response and `wait_type` as the predictor. The test focuses on the p-value of the `wait_type` coefficient, with a significance level of 0.05. If the p-value is below 0.05, we reject the null hypothesis, indicating a significant difference in eruption duration between the two groups; if not, we conclude there is no significant difference. This streamlined setup clarified how the indicator variable impacts eruption durations between the classified wait groups.

Dont mention the word checkpoint

We shifted focus to analyzing eruption durations based on a redefined classification of wait times. Wait times are categorized as "short" if under 68 minutes and "long" otherwise, represented by an indicator variable (`wait_type`). This setup allows us to test for average differences in eruption duration between the two groups. The null hypothesis states there is no difference in average durations between short and long wait times, while the alternative asserts a difference. We implemented a Simple Linear Regression model with `duration` as the response and `wait_type` as the predictor. The hypothesis test centers on the p-value of the `wait_type` coefficient at a significance level of 0.05. If the p-value is below 0.05, we reject the null hypothesis, indicating a significant difference in eruption duration between groups; if not, we conclude there is no significant difference. This approach clarified the effect of the indicator variable on eruption duration across wait categories.

#CHECKPOINT 2

Understood!



Uploaded image

Thank you for sharing the image! This figure displays histograms of residuals from four different models based on wait time classifications for eruption data, with each model representing a different approach:

1. **Model 1:** Uses a slope across all data (blue histogram).
2. **Model 2:** Applies only to short wait data (green histogram).
3. **Model 3:** Applies only to long wait data (orange histogram).
4. **Model 4:** Uses an indicator variable for short/long waits on all data (red histogram).

Each plot includes a fitted normal distribution curve for visual comparison and labels the sample size (n) for each model. This visualization allows for assessment of residual normality and model fit differences across different wait-time-based model specifications. Let me know if you'd like further analysis or modifications based on this comparison!

Which one of these passes the normality assumption

Based on the visual inspection of the histograms and the overlaid normal distribution curves, **Model 1 (All Data using slope)** and **Model 4 (All Data using indicator)** appear to pass the normality assumption more closely than the others. They show a relatively symmetric, bell-shaped distribution of residuals centered around zero, with fewer extreme deviations compared to the other models.

Analysis:

- **Model 1:** The residuals are centered around zero, with a shape closely following the normal curve, suggesting this model reasonably meets the normality assumption.
- **Model 4:** Similarly, the residuals here also follow a roughly normal shape, indicating that the indicator-based model across all data maintains normality.

Models with Potential Issues:

- **Model 2 (Short Wait Data) and Model 3 (Long Wait Data):** Both these models exhibit some skewness and deviations from the normal curve, especially in the tails, indicating that the residuals are less normally distributed. This may suggest that separate models for short and long waits don't capture the data as effectively under the normality assumption.

Thus, **Model 1** and **Model 4** would be more reliable for analyses that assume normally distributed residuals.

Based on closer inspection, **Model 1 (All Data using slope)** is the only model that appears to pass the normality assumption.

Explanation:

- **Model 1:** The residuals for this model follow a symmetric, bell-shaped distribution closely aligned with the overlaid normal curve, indicating that this model satisfies the normality assumption.
- **Model 4 (All Data using indicator),** although somewhat close, shows heavier tails and slight deviations from the normal distribution, which suggest that its residuals are not perfectly normal. This could indicate that the indicator variable introduces some non-normality in the residuals, possibly due to differences between short and long wait groups that are not entirely captured by this simple split.

In summary, **Model 1** is the only model that fully meets the normality assumption based on the residual histogram comparison.

Identify which of the histograms suggests the plausibility of the assumption that the distribution of error terms is normal for each of the models, and explain why the other three do not support this assumption.

The histogram for **Model 1 (All Data using slope)** is the only one that suggests the plausibility of the assumption that the distribution of error terms is normal. Here's why this histogram supports normality and why the others do not:

Model 1: All Data using Slope

- The histogram for Model 1 is symmetric, centered around zero, and closely follows the overlaid normal curve. This shape indicates that the residuals are likely drawn from a normal distribution, satisfying the normality assumption for the error terms. The relatively balanced distribution without heavy tails or skewness suggests this model aligns well with the assumption of normally distributed errors.

Why the Other Models Do Not Support Normality

1. Model 2: Short Wait Data

- The histogram for Model 2 shows a skewed distribution with a high peak around zero and a longer right tail, deviating from the shape of the overlaid normal curve. This suggests that the residuals for this model are not normally distributed, as they lack symmetry and have an uneven spread.

2. Model 3: Long Wait Data

- The histogram for Model 3 has a similar issue, with a pronounced peak and a longer left tail, creating an asymmetrical distribution. This skewness indicates non-normality in the residuals, likely due to the limited data range in the "long wait" subset. This imbalance prevents the model from meeting the normality assumption.

3. Model 4: All Data using Indicator

- While Model 4's histogram is somewhat symmetric, it has heavier tails than a normal distribution and shows slight deviations from the normal curve. This distribution suggests that residuals may not perfectly adhere to normality, likely due to differences between the "short" and "long" wait categories that are not fully captured by the indicator variable.

In summary, **Model 1** is the only model that adequately supports the normality assumption, while the other three models show signs of skewness or heavier tails, suggesting that the residuals do not follow a normal distribution in those cases.

give me a session summary for this checkpoint

We began this session by shifting our approach to defining wait times, categorizing waits under 68 seconds as "short" and those 68 seconds or longer as "long." This adjustment led to a new Simple Linear Regression model incorporating an indicator variable to differentiate between these categories. We focused on conducting hypothesis testing to evaluate if there was a statistically significant difference in eruption durations between the "short" and "long" wait categories, under the null hypothesis of no difference.

To assess model fit and the normality assumption of residuals, we generated histograms of residuals for four models: (1) All Data using Slope, (2) Short Wait Data, (3) Long Wait Data, and (4) All Data using the Indicator variable. By examining these plots, we determined that only Model 1's residuals aligned closely with a normal distribution, indicating it met the normality assumption. The other three models displayed skewness or heavy tails, which suggested violations of this assumption due to factors like the limited range in wait time subsets and the indicator variable's inability to fully capture the distribution nuances.

Recall, this checkpoint was purely focused on the normality assumption

In this session, we focused on assessing the normality assumption for different regression models by examining histograms of their residuals. Specifically, we looked at four models: (1) All Data using Slope, (2) Short Wait Data, (3) Long Wait Data, and (4) All Data using an Indicator variable for wait time. Our objective was to identify which model's residuals aligned with the assumption of normally distributed errors.

Among these models, only Model 1 (All Data using Slope) met the normality assumption, as its residuals followed a bell-shaped distribution closely resembling a normal distribution. The other three models—Short Wait Data, Long Wait Data, and All Data using the Indicator—displayed deviations from normality. These deviations suggested that the models did not fully satisfy the normality assumption, likely due to the limitations imposed by splitting the data into subsets or by the simplicity of the indicator variable model.