Exercise 11

Import notebook funcs

```
from notebookfuncs import *
```

Import user funcs

```
from userfuncs import *
```

In this problem we will investigate the t-statistic for the null hypothesis $H_0 : \beta = 0$ in simple linear regression without an intercept. To begin, we generate a predictor x and a response y as follows.

```
rng = np.random.default_rng (1)

x = rng.normal(size =100)

y = 2 * x + rng.normal(size =100)
```

```
import numpy as np
import pandas as pd

## [[https://stats.stackexchange.com/a/382059/270877]]
def generate_data(mean=0.0, sd=1.0, inverse=False):
    N = 100
    b = 2.0
    rng = np.random.default_rng(1)
```

```
    x = rng.normal(size=N)
    noise = rng.normal(loc=mean, scale=sd, size=N)
    y = b * x + noise
    if inverse:
        y = x
        x = (1 / b) * y - (1 / b**2) * noise

    df = pd.DataFrame({"x": x, "y": y})
    return df


df = generate_data()
std_x = np.std(df["x"])
std_y = np.std(df["y"])
df.head()
```

	x	y
0	0.345584	0.039887
1	0.821618	2.505681
2	0.330437	0.535282
3	-1.303157	-1.937161
4	0.905356	3.029555

(a) Perform a simple linear regression of y onto x, without an intercept. Report the coefficient estimate $\hat{\beta}$, the standard error of this coefficient estimate, and the t-statistic and p-value associated with the null hypothesis $H_0 : \beta = 0$. Comment on these results. (You can perform regression without an intercept using the keywords argument intercept=False to ModelSpec().)

```
formula = "y ~ x + 0"
model = smf.ols(f"{formula}", df)
results = model.fit()
result_df = get_results_df(results)
result_df
```

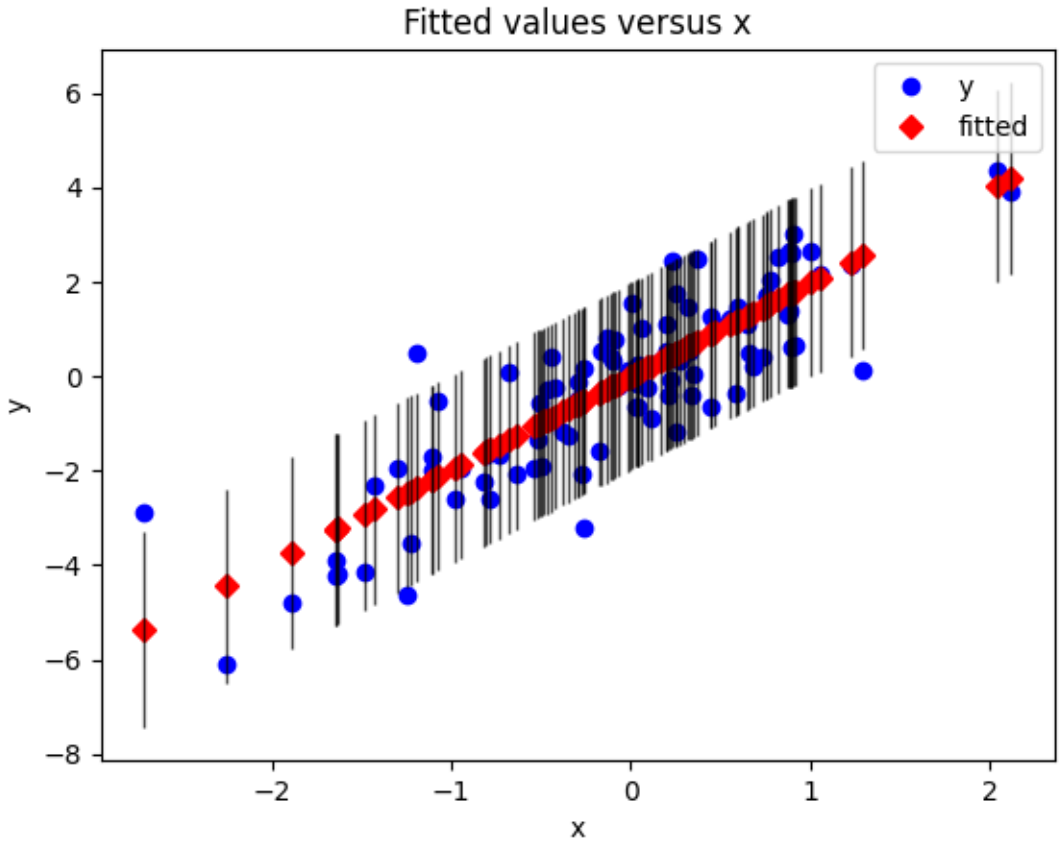
	coefficient	se	tstatistic	p-value	r-squared	pearson_coefficient	rss	sd_residua
x	1.976242	0.116948	16.898417	6.231546e-31	0.742561	0.86172	98.91997	0.999596

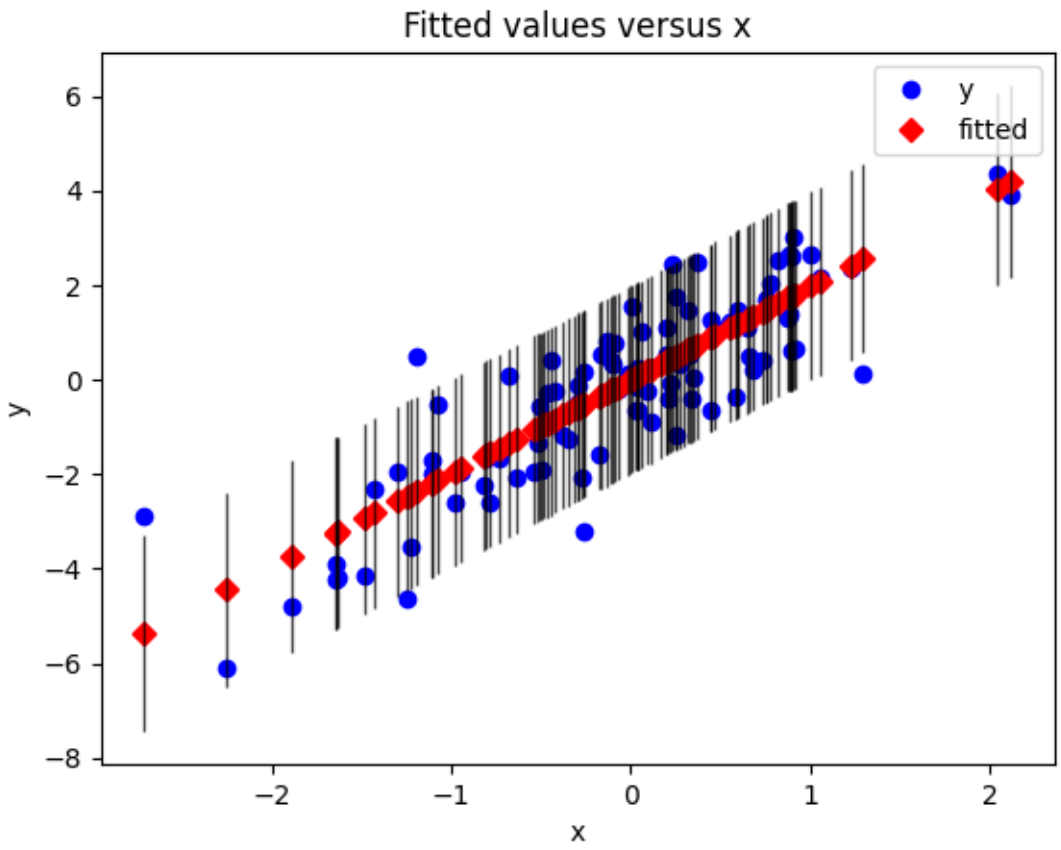
```
print("Calculated t-statistic:")
print(f"{results.params/results.bse}")
```

```
Calculated t-statistic:
x    16.898417
dtype: float64
```

- The β_1 estimate is 1.976242 which is close to the actual value of 2.0 used while generating the data.
- The standard error is 0.116948 which is low
- The p-value of $6.231546e^{-31}$ suggests a strong relationship between x and y of the form: $y = 1.976242 * x$

```
from statsmodels.graphics.regressionplots import plot_fit
plot_fit(results, "x")
```





(b) Now perform a simple linear regression of x onto y without an intercept, and report the coefficient estimate, its standard error, and the corresponding t-statistic and p-values associated with the null hypothesis $H_0 : \beta = 0$. Comment on these results.

```
formula = "x ~ y + 0"
model = smf.ols(f"{formula}", df)
results = model.fit()
result_df = get_results_df(results)
result_df
```

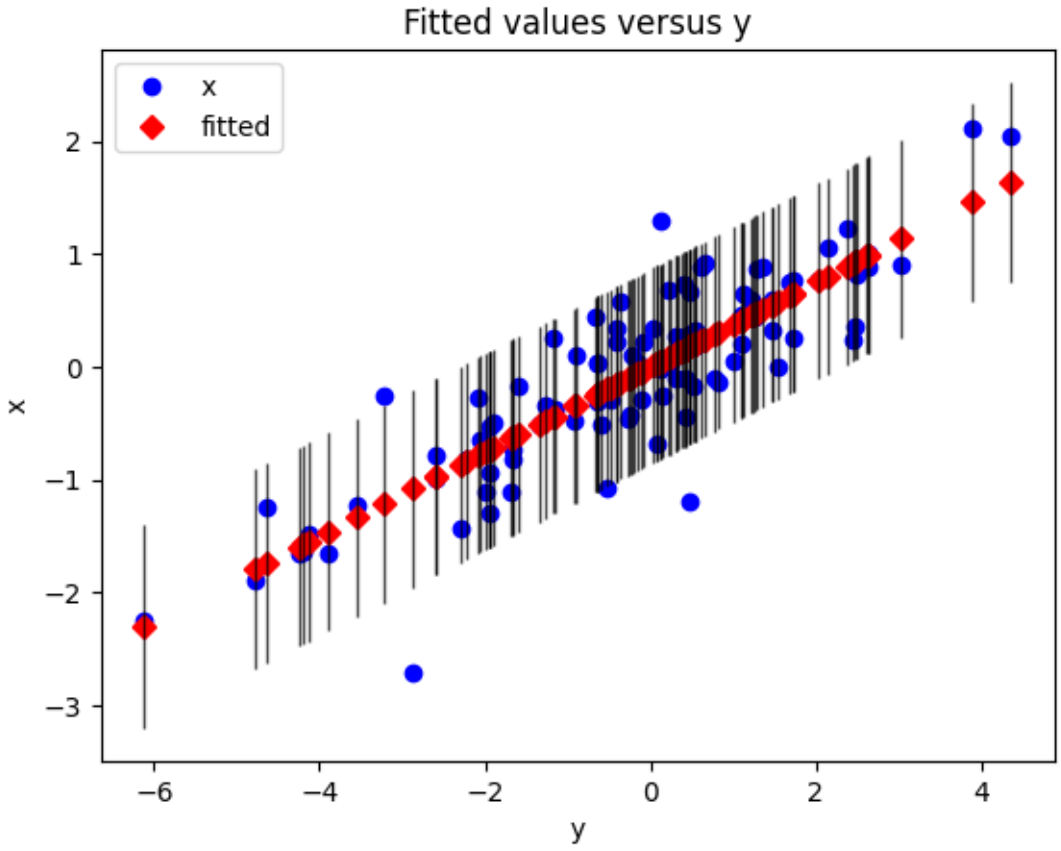
	coefficient	se	tstatistic	p-value	r-squared	pearson_coefficient	rss	sd_residua
y	0.375744	0.022235	16.898417	6.231546e-31	0.742561	0.86172	18.80769	0.435863

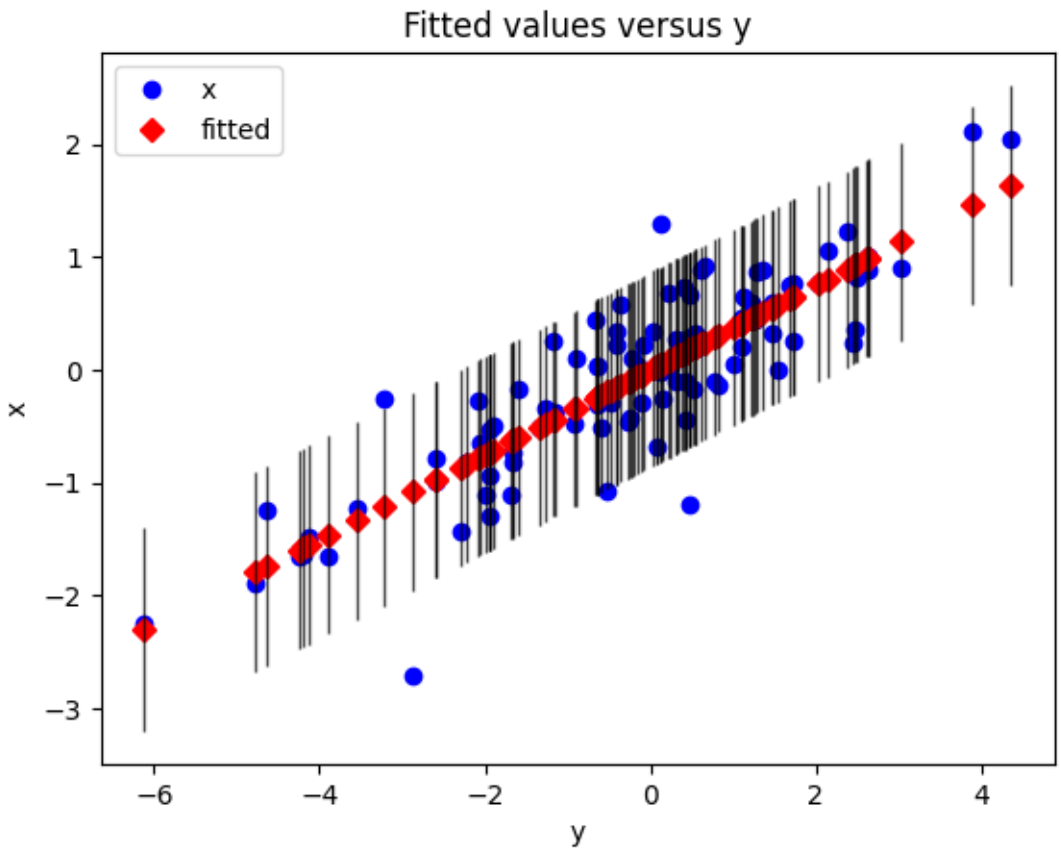
```
print("Calculated t-statistic:")
print(f"{results.params/results.bse}")
```

```
Calculated t-statistic:
y    16.898417
dtype: float64
```

- The β_1 estimate, in this case, is 0.375744 which is not as close as one would expect to 0.5
- The p-value of $6.231546e^{-31}$ signifies a strong relationship between x and y of the form $x = 0.375744 * y$
- The t-statistics and, hence, the p-values are identical in both regressions.
- We can also see that the R^2 and ρ (Pearson coefficient) are identical in both regressions.

```
plot_fit(results, "y")
```






```
pearson_coefficient = result_df["pearson_coefficient"]
```

```
y    0.86172
Name: pearson_coefficient, dtype: float64
```

(c) What is the relationship between the results obtained in (a) and (b)?

Why should the estimates of the coefficient of β_1 differ from the expected value of 0.5 in the regression of $x \sim y + 0$?

- The obvious answer is that the cause of the variation is because of the introduction of the error term (or residuals) that we introduced while generating y from x with a mean of 0 and a standard deviation of 1.
- For x and y to be perfectly symmetrical, the standard deviation of the error terms would have to be zero.

- We can check that with the regression below where we generate y with zero SD and regress y on x and then x on y.

```
df = generate_data(sd=0)
std_x_perfect = np.std(df["x"])
std_y_perfect = np.std(df["y"])
std_x_perfect, std_y_perfect
```

```
(0.8515567333390056, 1.7031134666780112)
```

```
formula = "y ~ x + 0"
model = smf.ols(f"{formula}", df)
results = model.fit()
result_df = get_results_df(results)
result_df
```

	coefficient	se	tstatistic	p-value	r-squared	pearson_coefficient	rss	sd_residuals
x	2.0	0.0	inf	0.0	1.0	1.0	0.0	0.0

```
formula = "x ~ y + 0"
model = smf.ols(f"{formula}", df)
results = model.fit()
result_df = get_results_df(results)
result_df
```

	coefficient	se	tstatistic	p-value	r-squared	pearson_coefficient	rss	sd_residuals
y	0.5	0.0	inf	0.0	1.0	1.0	0.0	0.0

Now, the formula for $\beta_{1(y\ on\ x)}$ is $\rho * \frac{S_y}{S_x}$ and for $\beta_{1(x\ on\ y)}$ is $\rho * \frac{S_x}{S_y}$.

Using the formulae above:

```
beta_1_x_on_y = pearson_coefficient * std_x / std_y
beta_1_y_on_x = pearson_coefficient * std_y / std_x
beta_1_y_on_x, beta_1_x_on_y
```

```
(y    1.970973
 Name: pearson_coefficient, dtype: float64,
 y    0.376748
 Name: pearson_coefficient, dtype: float64)
```

Now, if Pearson coeffcient were 1.0, i.e., there is perfect correlation between x and y, then we can calculate the betas and find them to be reciprocals of each other.

```
beta_1_x_on_y = 1.0 * std_x_perfect / std_y_perfect
beta_1_y_on_x = 1.0 * std_y_perfect / std_x_perfect
beta_1_y_on_x, beta_1_x_on_y
```

```
(2.0, 0.5)
```

If we were to standardize x and y, we would have the following regression results

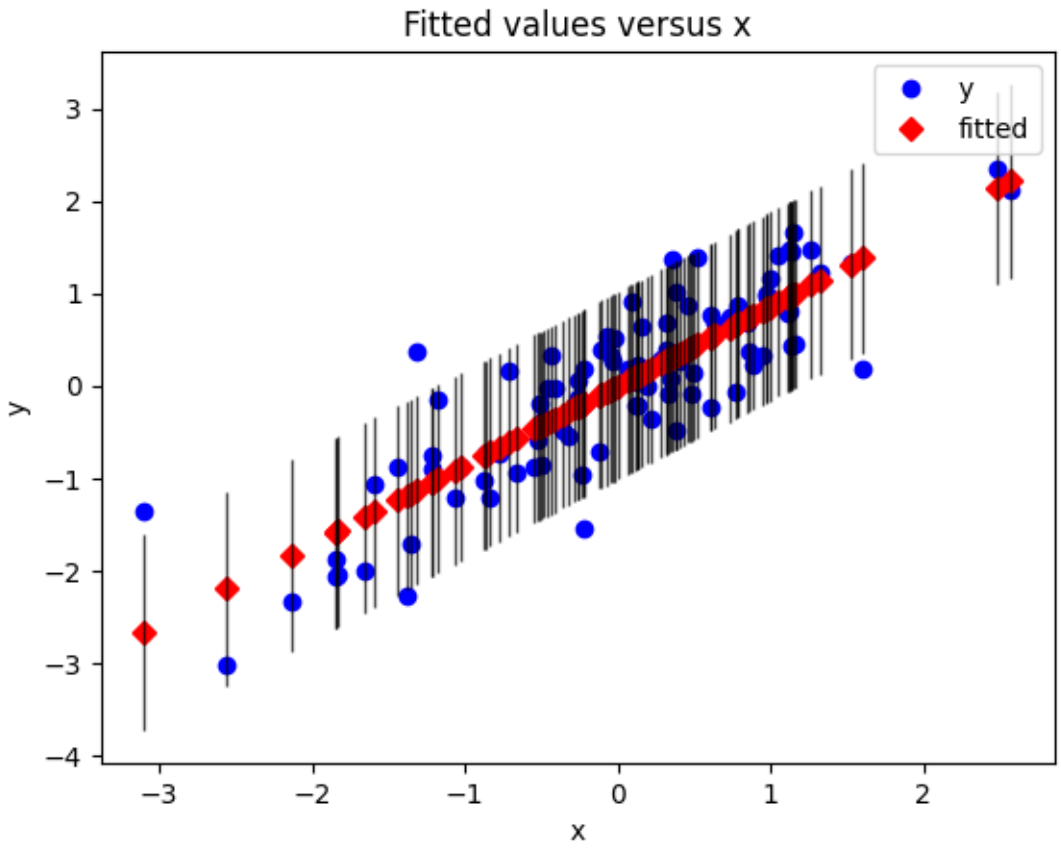
```
from scipy import stats

df = generate_data()
df["x"] = stats.zscore(df["x"])
df["y"] = stats.zscore(df["y"])
df.head()
```

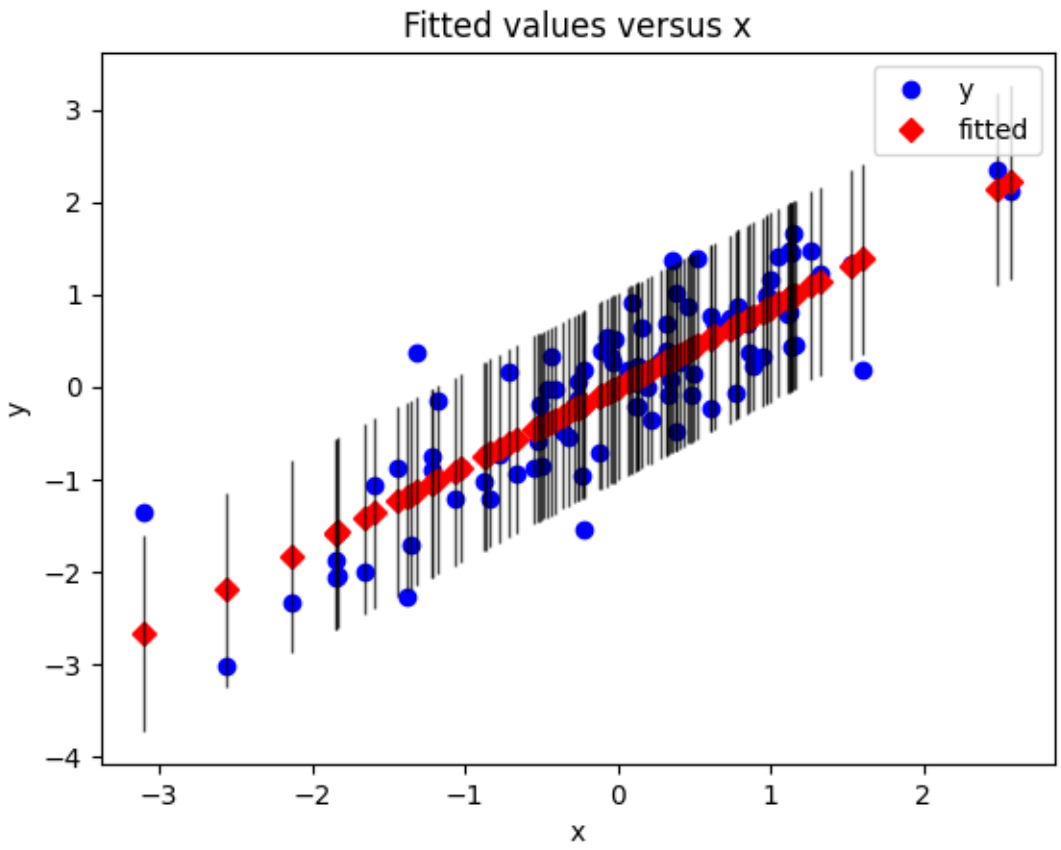
	x	y
0	0.492271	0.133912
1	1.051287	1.399897
2	0.474483	0.388257
3	-1.443879	-0.881142
4	1.149622	1.668864

```
formula = "y ~ x + 0"
model = smf.ols(f"{formula}", df)
results = model.fit()
result_df = get_results_df(results)
result_df
```

	coefficient	se	tstatistic	p-value	r-squared	pearson_coefficient	rss	sd_residu
x	0.860674	0.051172	16.819216	8.808756e-31	0.74076	0.860674	25.923964	0.511721

```
plot_fit(results, "x")
```

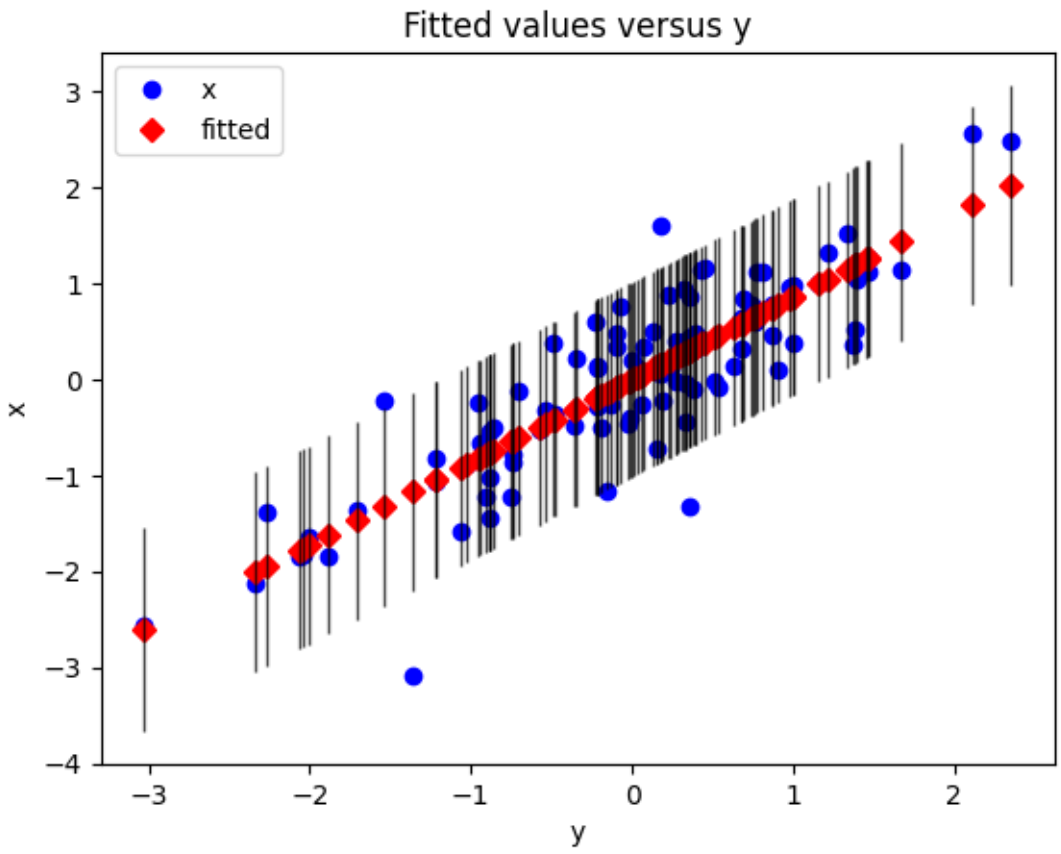


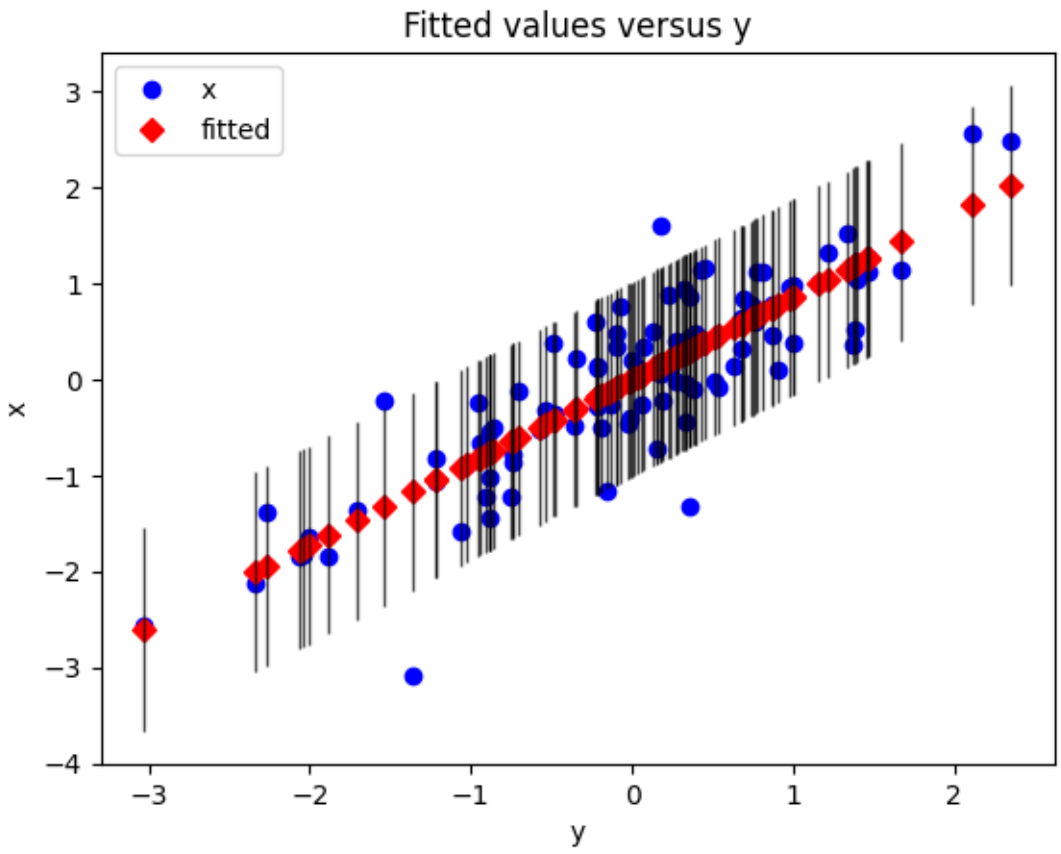



```
formula = "x ~ y + 0"
model = smf.ols(f"{formula}", df)
results = model.fit()
result_df = get_results_df(results)
result_df
```

	coefficient	se	tstatistic	p-value	r-squared	pearson_coefficient	rss	sd_residu
y	0.860674	0.051172	16.819216	8.808756e-31	0.74076	0.860674	25.923964	0.511721

```
plot_fit(results, "y")
```


Fitted values versus y
x
fitted
3
2
1
0
−1
−2
−3
−4
x
−3
−2
−1
0
1
2
y



This actually matches our intuition that the slopes of both lines converge and the coefficient is the same in both regressions. This is because the standardized x and y variables have standard deviation of 1 and hence the slope in either case is simply ρ.

If we try to mimimize the residuals using the formula $\frac{1}{b^2} * E(Y - bX)^2$

Using some data manipulation in the generate_data method, we can come close to the expected estimate of 0.5 in the X ~ Y + 0 regression.

```
df = generate_data(inverse=True);
```

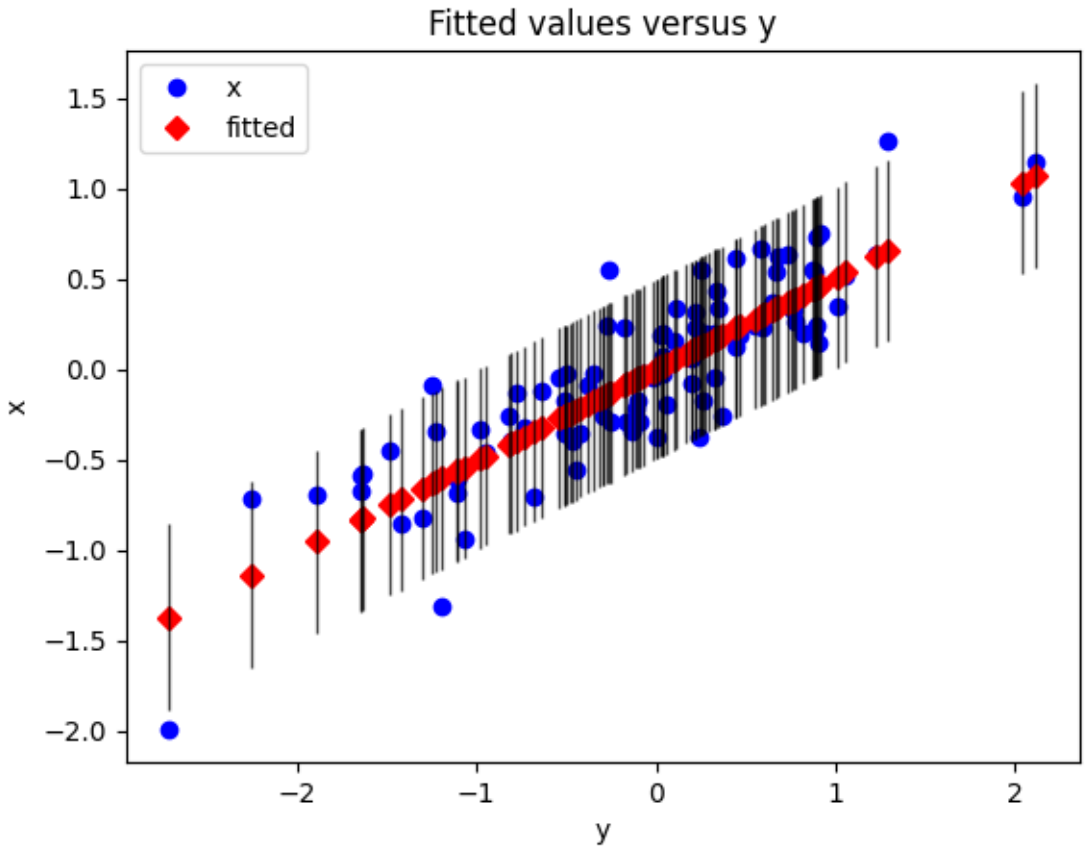
```
formula = "x ~ y + 0"
model = smf.ols(f"{formula}", df)
results = model.fit()
```

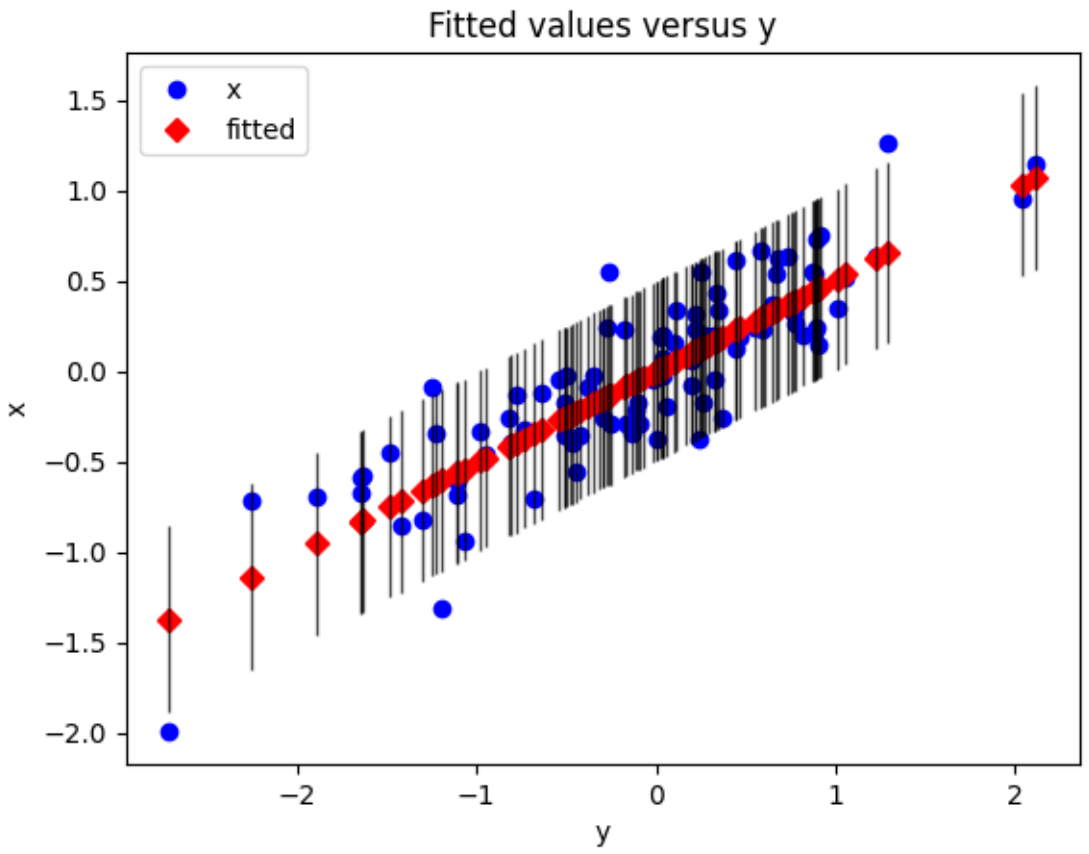
```
result_df = get_results_df(results)
result_df
```

	coefficient	se	tstatistic	p-value	r-squared	pearson_coefficient	rss	sd_residuals
y	0.505939	0.029237	17.30471	1.068850e-31	0.751539	0.866914	6.182498	0.249899

Note:- The standard deviation of the residuals is $\frac{1}{b}^2$ times thr SD of the residuals for Y ~ X + 0.

```
plot_fit(results, "y")
```





If we again invert the regression, regression y on x, we have:

```
formula = "y ~ x + 0"
model = smf.ols(f"{formula}", df)
results = model.fit()
result_df = get_results_df(results)
result_df
```

	coefficient	se	tstatistic	p-value	r-squared	pearson_coefficient	rss	sd_residuals
x	1.485433	0.08584	17.30471	1.068850e-31	0.751539	0.866914	18.151752	0.428195

- *Again, we see that since we're minimizing the distance between x and its fitted values in the first regression and y and its fitted values in the next, the estimatesof the first regression are close to the actual value but not so in its inverse. This only works if the residuals are manipulated to have SD of $\frac{-1}{b^2}$ in the second.*

- *That's the most I can explain.*
- *There is an ettempt to explain the above in this StackOverflow post but I'm not as comfortable or well-versed with linear algebra as yet to expound on it further.*

```
formula = "x ~ y + 0"
model = smf.ols(f"{formula}", df)
results = model.fit()
result_df = get_results_df(results)
result_df
```

	coefficient	se	tstatistic	p-value	r-squared	pearson_coefficient	rss	sd_residuals
y	0.505939	0.029237	17.30471	1.068850e-31	0.751539	0.866914	6.182498	0.249899

We can check that the smaller the noise or residuals, the more likely that the regression of y on x and x on y are more or less reciprocals of each other in terms of the coefficient of the regressor.

```
from sklearn.linear_model import LinearRegression

# Generate data
np.random.seed(0)
x = np.random.rand(100)
denominator = 1.0
noise = np.random.randn(100)

while True:

    print(f"SD of noise: {np.std(noise)}")
    y = 2 * x + noise
    # Fit regressions
    x_on_y = LinearRegression(fit_intercept=False).fit(y[:, np.newaxis], x)
    y_on_x = LinearRegression(fit_intercept=False).fit(x[:, np.newaxis], y)

    # Coefficient values
    beta1 = y_on_x.coef_[0]
    alpha1 = x_on_y.coef_[0]
    print("beta1, alpha1")
    print(beta1, alpha1)
    # Verify inverse relationship
```

```
    print("Are beta and alpha reciprocally close?")
    print(np.isclose(beta1 * alpha1, 1))
    if np.isclose(beta1 * alpha1, 1):
        break
    denominator *= 10
    noise = noise / denominator
```

```
SD of noise: 0.99637805284736
beta1, alpha1
2.2794461383640625 0.2689337259018727
Are beta and alpha reciprocally close?
False
SD of noise: 0.09963780528473601
beta1, alpha1
2.027944613836406 0.4892084095475252
Are beta and alpha reciprocally close?
False
SD of noise: 0.00099637805284736
beta1, alpha1
2.0002794461383635 0.499929738398587
Are beta and alpha reciprocally close?
True
```

- *Reference: https://stats.stackexchange.com/questions/22718/what-is-the-difference-between-linear-regression-on-y-with-x-and-x-with-y*

(d) For the regression of Y onto X without an intercept, the t-statistic for $H_0 : \beta = 0$ takes the form $\hat{\beta}/SE(\hat{\beta})$, where $\hat{\beta}$ is given by (3.38), and where

$$SE(\hat{\beta}) = \sqrt{\frac{\sum_{i=1}^{n}\left(y_i - x_i\hat{\beta}\right)^2}{(n-1)\sum_{i'=1}^{n} x_{i'}^2}}$$

(These formulas are slightly different from those given in Sections 3.1.1 and 3.1.2, since here we are performing regression without an intercept.) Show algebraically, and confirm numerically in Python, that the t-statistic can be written as

$$\frac{(n-1)\sum_{i=1}^{n} x_i y_i}{\sqrt{\left(\sum_{i=1}^{n} x_i^2\right)\left(\sum_{i=1}^{n} y_i^2\right) - \left(\sum_{i'=1}^{n} x_{i'} y_{i'}\right)^2}}$$

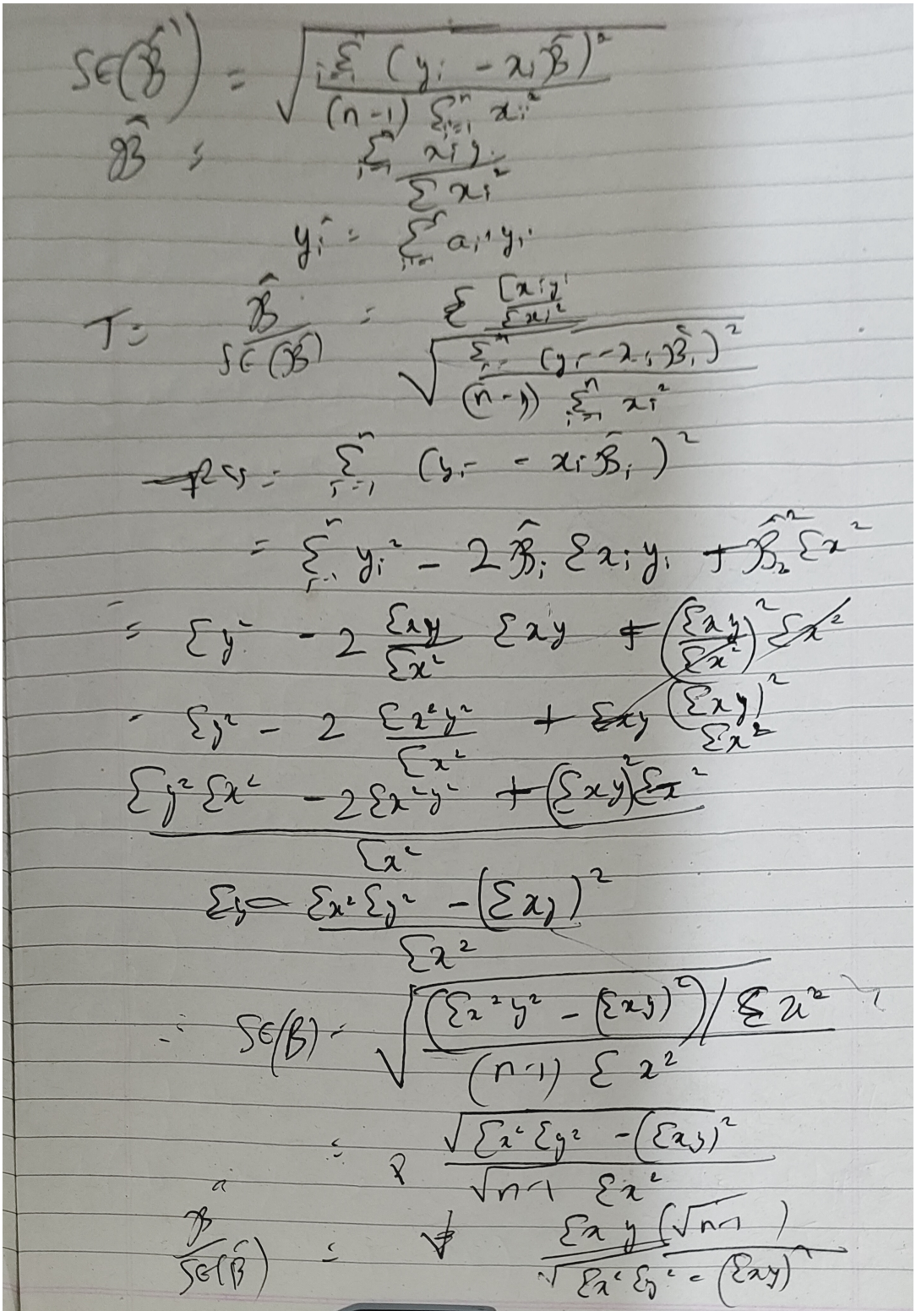


Figure 1: Solution

```
df = generate_data()
formula = "y ~ x + 0"
model = smf.ols(f"{formula}", df)
results = model.fit()
result_df = get_results_df(results)
t = result_df["tstatistic"]
```

```
x    16.898417
Name: tstatistic, dtype: float64
```

The calculated t-statistic from the formula above is:

```
x = df["x"]
y = df["y"]
numerator = np.sqrt(results.nobs - 1) * np.sum(x * y)
denominator = np.sqrt(np.sum(x**2) * np.sum(y**2) - np.sum(x * y) ** 2)
tstat = numerator / denominator
```

```
16.898417063035094
```

```
np.isclose(t, tstat)
```

```
array([ True])
```

(e) Using the results from (d), argue that the t-statistic for the regression of y onto x is the same as the t-statistic for the regression of x onto y.

- This is evident when we swap y and x in the derived formula that the result is the same.
- This is also what we observed when we regressed y on x and x on y (without intercept)
- Hence, the T-statistic is the same in both cases

(f) In Python, show that when regression is performed with an intercept, the t-statistic for $H_0 : \beta_1 = 0$ is the same for the regression of y onto x as it is for the regression of x onto y.

```
formula = "y ~ x"
model = smf.ols(f"{formula}", df)
results = model.fit()
result_df = get_results_df(results)
tx = result_df["tstatistic"].iloc[1]
```

16.734055202403045

```
formula = "x ~ y"
model = smf.ols(f"{formula}", df)
results = model.fit()
result_df = get_results_df(results)
ty = result_df["tstatistic"].iloc[1]
```

16.734055202403038

```
np.isclose(tx, ty)
```

True

```
allDone();
```

<IPython.lib.display.Audio object>