PS2_solutions

April 7, 2021

1 Problem Set 2

Data Manipulation

CHEM 114

Shimon Weiss

Due Monday, Aril 12th

This PS is designed to have you practice the basics of Python data manipulation. We will try to walk you through most of the assignment, but you will have to enter and execute your own code.

By the end of this assignment you should be able to manipulate and probe datasets that you upload with Python.

1.1 Initialize your code

- Here, import the dependancies you will need to begin your analysis. > In computer programming, this is called the 'boilerplate'. The boilerplate is a section of code that must be included with little or no alteration. It is often used when referring to languages that are considered verbose, i.e. the programmer must write a lot of code to do minimal jobs. Python is considered a verbose language. In this case, our boilerplate is composed of the libraries that we import.
- If you are confused as to what to put in this section, look at other example code we have provided. Remember, `SHIFT+ENTER` to execute the code in a cell!

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.optimize import curve_fit
```

[2]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'  # use this for hi-dpi displays
sns.set_style('whitegrid')  # set the plotting style
```

1.2 Problem 1

Loading data

- Use the numpy `loadtxt` function to load the data file, "ProvidedData.csv". This will load your data from a .csv into a numpy array. When you load your data this way, you will need to specify the type of delimiter.

  HINT: csv stands for **c**omma **s**eparated **v**alue. Knowing this, can you guess the delimiter? If you can't remember how this is done, refer to the example done in class.

- Remember to give loaded data a name e.g. `ProvidedData` or `provided_data`.

- Preview your data so that you know it is loaded properly. Note that jupyter notebook will always print the last thing written in a cell when executed with `SHIFT + ENTER`. This is a very useful debugging technique. You can also use the builtin `print()` function.

- Check how many rows and column there are by finding the shape of your array, `provided_data`. Checking the shape and/or length of an array is often an important debugging step!

-

1.3 use `len()` to check the length of your array.

1.4 Solution 1

[3]:
```
provided_data = np.loadtxt('ProvidedData.csv',delimiter=',')
provided_data
```

[3]:
```
array([[ 3.5136e+01,  3.0000e+00,  9.7399e-01,  5.4378e+00,  1.0000e+00,
         2.7673e+00],
       [ 4.4878e+01,  4.0000e+00,  8.7947e-01,  3.3751e+00,  1.0099e+00,
         2.8620e+00],
       [ 2.8805e+01,  4.0000e+00,  5.3172e-01,  5.5909e+00,  1.0198e+00,
         2.6986e+00],
       …,
       [ 1.6703e+01,  4.0000e+00,  7.4682e-01,  2.8670e+00,  9.9980e+01,
        -1.3141e-01],
       [ 3.9595e+01,  0.0000e+00,  9.5972e-01,  5.1100e+00,  9.9990e+01,
         9.7506e-02],
       [ 2.0668e+01,  3.0000e+00,  9.5377e-01,  4.2306e+00,  1.0000e+02,
        -9.1760e-02]])
```

[4]:
```
provided_data.shape
```

[4]:
```
(10000, 6)
```

[5]:
```
len(provided_data)
```

[5]:
```
10000
```

Now that you have loaded the data, it is easier to manipulate the data if you reassign the data into X-variables, Y-variables, Z-varibles, etc.

1.5 Problem 2

Indexing your data

- From your loaded data, split the six columns into six separate 1D arrays. Be sure to use different array names so that you don't overwrite the original data
  - remember that indexing begins at 0 in Python! > A note on nameing, don't worry about being creative, a **descriptive** name is better. For example: > - `random_dist1` > - `random_dist2` > - `random_dist3` > - `random_dist4` > - `provided_x` > - `provided_y`

1.6 Solution 2

```
[6]: random_dist1 = provided_data[:,0]
     random_dist1
```

```
[6]: array([35.136, 44.878, 28.805, …, 16.703, 39.595, 20.668])
```

```
[7]: random_dist2 = provided_data[:,1]
     random_dist2
```

```
[7]: array([3., 4., 4., …, 4., 0., 3.])
```

```
[8]: random_dist3 = provided_data[:,2]
     random_dist3
```

```
[8]: array([0.97399, 0.87947, 0.53172, …, 0.74682, 0.95972, 0.95377])
```

```
[9]: random_dist4 = provided_data[:,3]
     random_dist4
```

```
[9]: array([5.4378, 3.3751, 5.5909, …, 2.867 , 5.11  , 4.2306])
```

```
[10]: provided_x = provided_data[:,4]
      provided_x
```

```
[10]: array([  1.    ,   1.0099,   1.0198, …,  99.98  ,  99.99  , 100.    ])
```

```
[11]: provided_y = provided_data[:,5]
      provided_y
```

```
[11]: array([ 2.7673  ,  2.862   ,  2.6986  , …, -0.13141 ,  0.097506,
              -0.09176 ])
```

1.7 Problem 3

Calculating statistics

- Now that the data is in a format that is easily manipulated, calculate the min, max, and standard deviation for `random_dist1`
- print each value clearly using f-strings. Round appropriately.
- calculate the variance as an operation within the curly bracket of your f-string

An example for the mean is shown below using an example array called `example_data`:

```
[12]: example_data = np.arange(1, 17).reshape(4,4)
      example_data
```

```
[12]: array([[ 1,  2,  3,  4],
             [ 5,  6,  7,  8],
             [ 9, 10, 11, 12],
             [13, 14, 15, 16]])
```

```
[13]: # calculate the mean for the indexed array using the .mean() method in numpy
      mean = example_data[0,:].mean()
      mean
```

```
[13]: 2.5
```

Here, the output is 2.5, but it is not obvious what the number 2.5 corresponds to without writing it explicitly.

In this case, it is useful to use the `print()` function so that the output is given a meaning and is *easy to read*. For example:

```
[14]: print('Mean =', mean)
```

```
Mean = 2.5
```

Another (better) way to print is with f-strings. In this case the variable is replaced inside the string at the location of the curly brackets, `{}`.

Implement the f-string by adding the letter 'f' to the begining of the string:

```
[15]: print(f'Mean = {mean}')
```

```
Mean = 2.5
```

Note that we can manipulate the value in the curly brackets. For example we can change the number of decimal places:

[16]:
```
print(f'Mean = {mean: .2f}')
```

```
Mean =  2.50
```

We can also perform an operation:

[17]:
```
print(f'squared mean = {mean ** 2}')
```

```
squared mean = 6.25
```

1.8 Solution 3

[18]:
```
rand1_min = random_dist1.min()
print(f'Min = {rand1_min: .1f}')
```

```
Min = -5.8
```

[19]:
```
rand1_max = random_dist1.max()
print(f'Max = {rand1_max: .1f}')
```

```
Max =  70.6
```

[20]:
```
rand1_std = random_dist1.std()
print(f'Standard deviation = {rand1_std: .1f}')
```

```
Standard deviation =  9.9
```

[21]:
```
print(f'variance = {rand1_std**2: .1f}')
```

```
variance =  97.5
```

1.9 Plotting data

Let's now try to get a little more information about the provided data by plotting the distribution with a histogram.

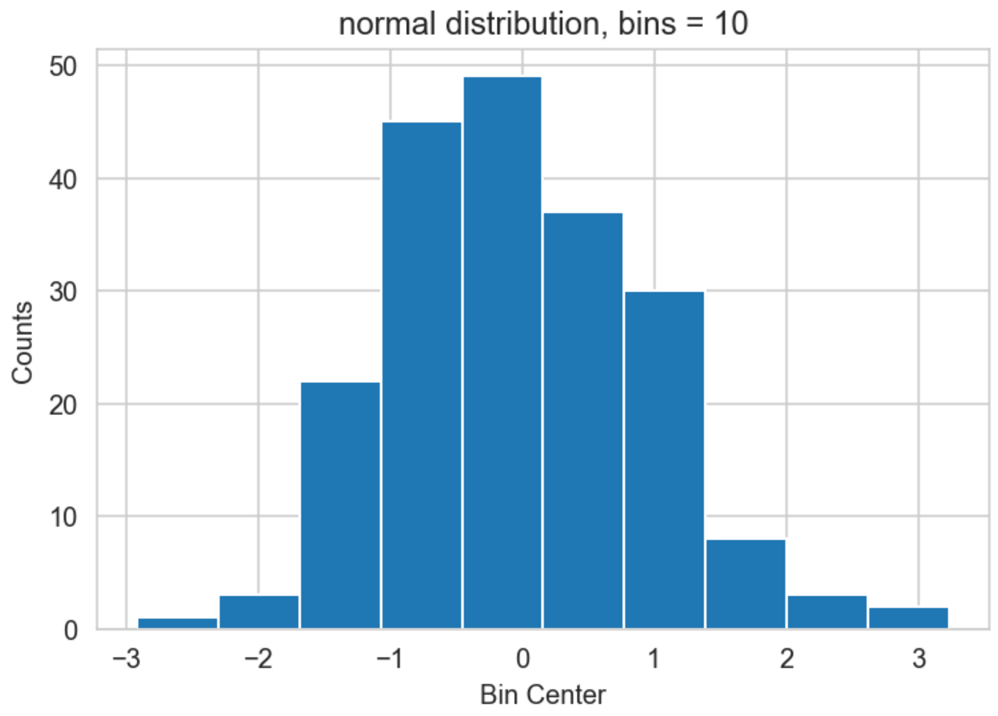
First, let's generate some random, normally distributed, data and plot a histogram:

[22]:
```
hist_data = np.random.randn(200)
hist_data.shape
```

[22]:
```
(200,)
```

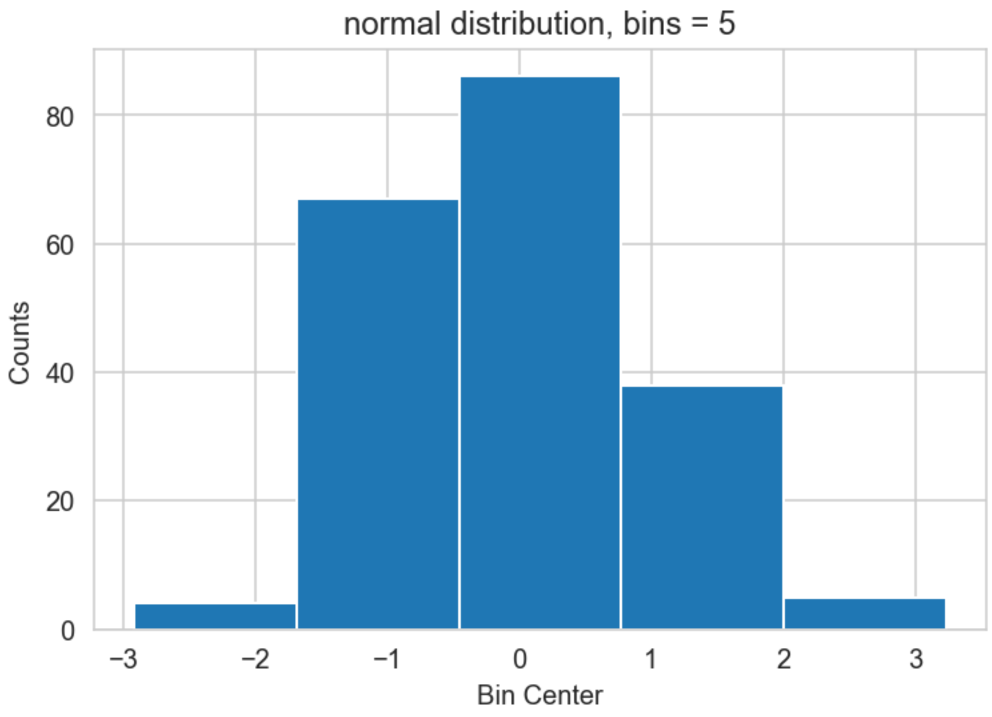
[23]:
```
bins = 10
plt.hist(hist_data, bins=bins)

plt.title(f'normal distribution, bins = {bins}')
plt.xlabel('Bin Center')
```

```python
plt.ylabel('Counts');
```

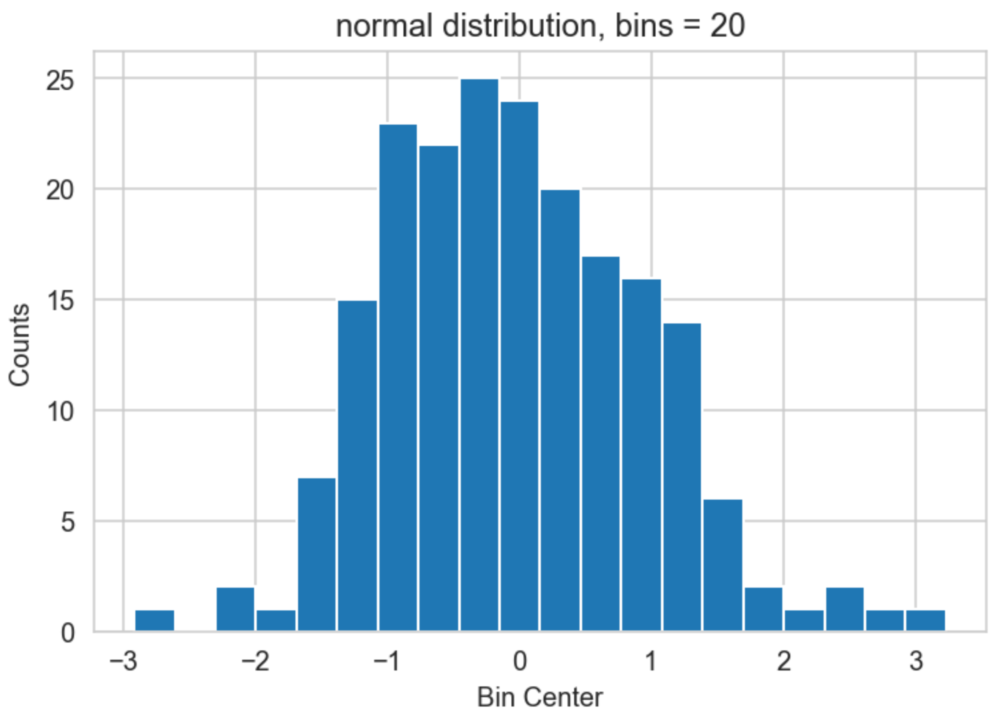


1.10 Solution 4

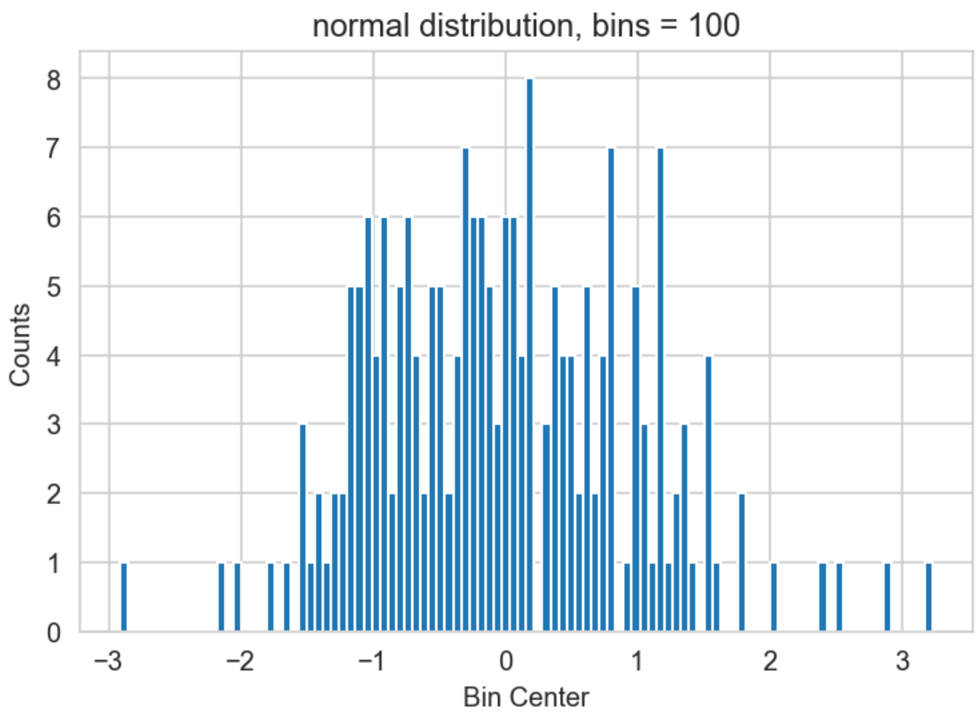
[24]:
```python
bins = 5
plt.hist(hist_data, bins=bins)

plt.title(f'normal distribution, bins = {bins}')
plt.xlabel('Bin Center')
plt.ylabel('Counts');
```



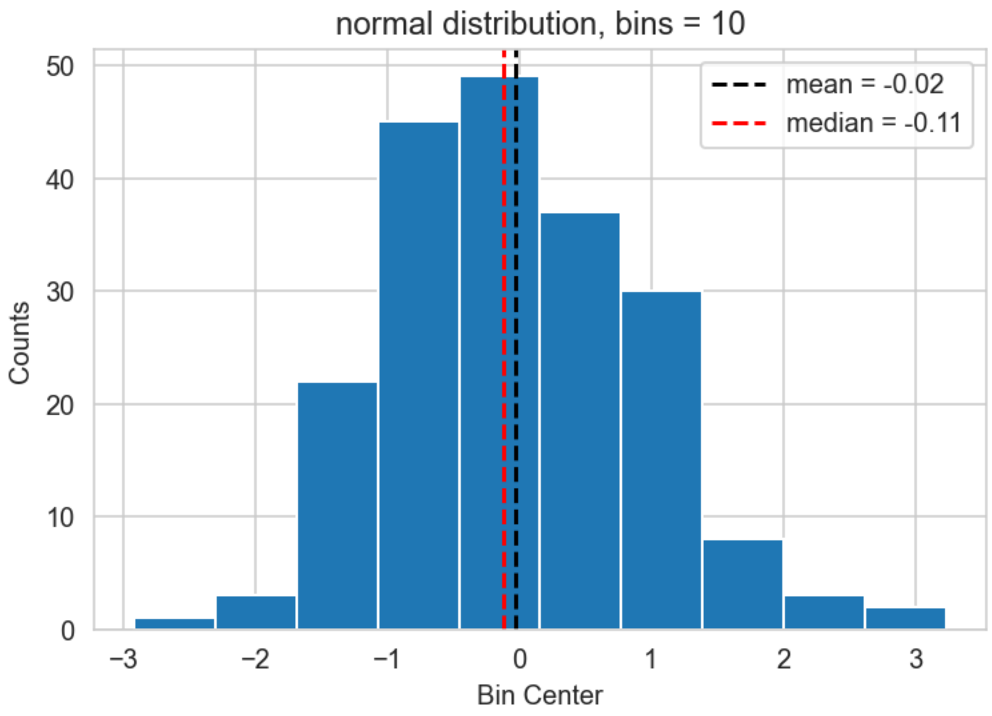
```
[25]: bins = 20
      plt.hist(hist_data, bins=bins)

      plt.title(f'normal distribution, bins = {bins}')
      plt.xlabel('Bin Center')
      plt.ylabel('Counts');
```



```
[26]: bins = 100
      plt.hist(hist_data, bins=bins)

      plt.title(f'normal distribution, bins = {bins}')
      plt.xlabel('Bin Center')
      plt.ylabel('Counts');
```



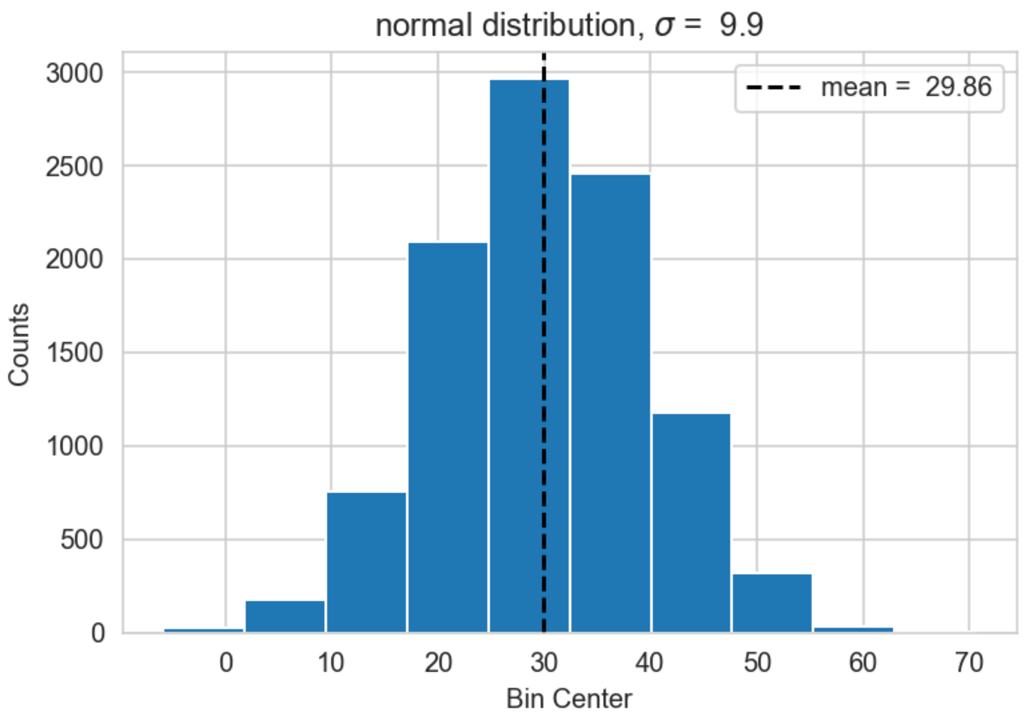
[27]:
```
bins = 10
plt.hist(hist_data, bins=bins)
plt.axvline(hist_data.mean(), color='k', linestyle='--', label=f'mean =␣
 ↪{hist_data.mean(): .2f}')
plt.axvline(np.median(hist_data), color='r', linestyle='--', label=f'median =␣
 ↪{np.median(hist_data): .2f}')

plt.title(f'normal distribution, bins = {bins}')
plt.xlabel('Bin Center')
plt.ylabel('Counts')
plt.legend();
```



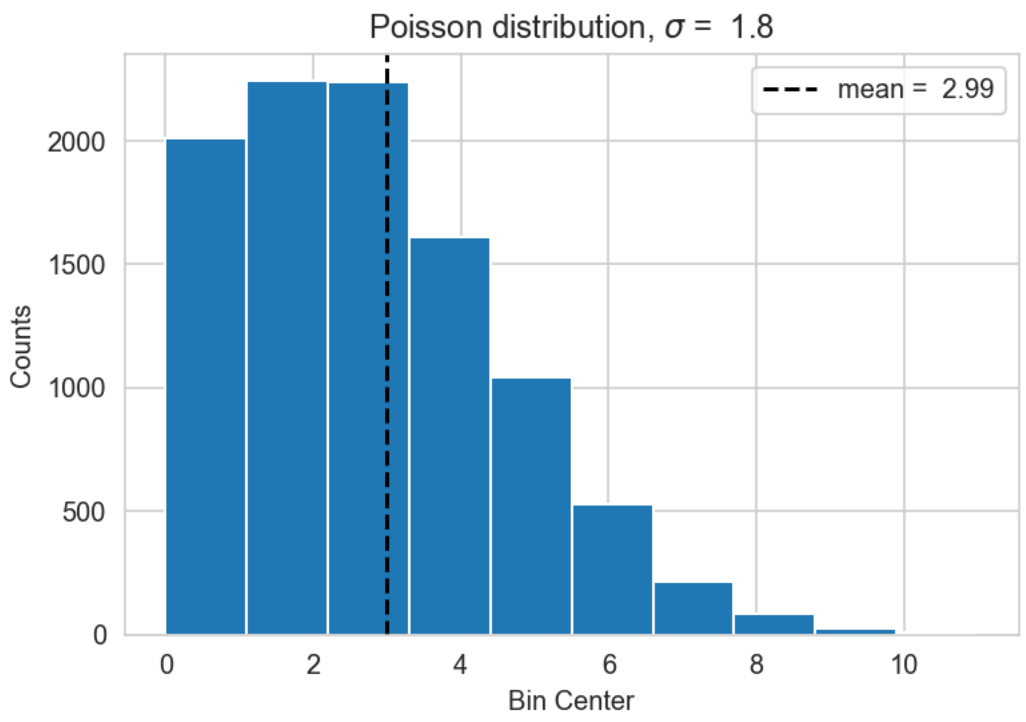
How does the number of bins affect the noise of your distribution? > a high number of bins will show the noise of your distribution > > A low number of bins will smooth the distribution and cause information loss

1.11 Solution 5

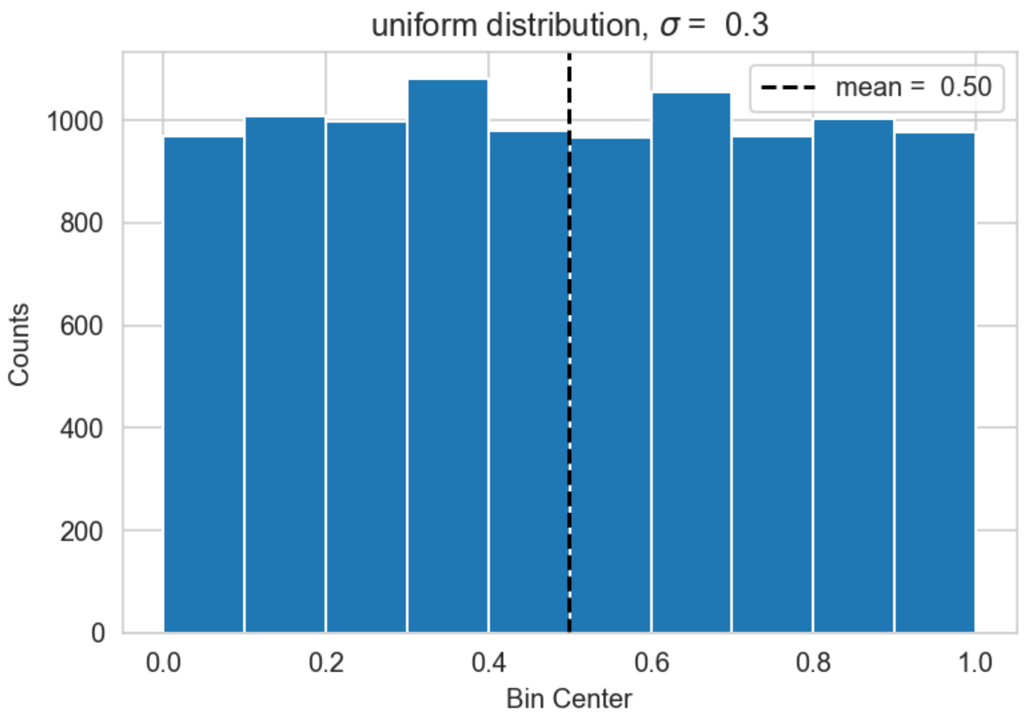
[28]:
```
plt.hist(random_dist1)
plt.axvline(random_dist1.mean(), color='k', linestyle='--', label=f'mean =␣
↪{random_dist1.mean(): .2f}')
plt.xlabel('Bin Center')
plt.ylabel('Counts')
plt.title(f'normal distribution, $\sigma$ = {random_dist1.std(): .1f}')
plt.legend();
```



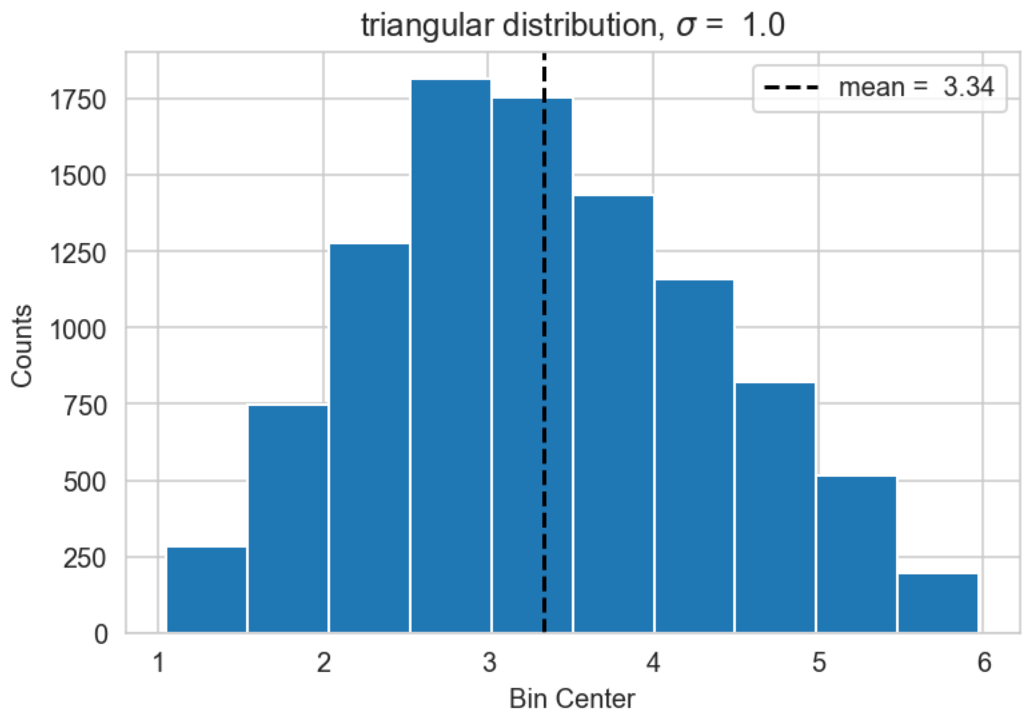
```
[29]: plt.hist(random_dist2)
      plt.axvline(random_dist2.mean(), color='k', linestyle='--', label=f'mean =␣
       ↪{random_dist2.mean(): .2f}')
      plt.xlabel('Bin Center')
      plt.ylabel('Counts')
      plt.title(f'Poisson distribution, $\sigma$ = {random_dist2.std(): .1f}')
      plt.legend();
```



```
[30]: plt.hist(random_dist3)
      plt.axvline(random_dist3.mean(), color='k', linestyle='--', label=f'mean =␣
       ↪{random_dist3.mean(): .2f}')
      plt.xlabel('Bin Center')
      plt.ylabel('Counts')
      plt.title(f'uniform distribution, $\sigma$ = {random_dist3.std(): .1f}')
      plt.legend();
```



```
[31]: plt.hist(random_dist4,)
      plt.axvline(random_dist4.mean(), color='k', linestyle='--', label=f'mean =␣
       ↪{random_dist4.mean(): .2f}')
      plt.xlabel('Bin Center')
      plt.ylabel('Counts')
      plt.title(f'triangular distribution, $\sigma$ = {random_dist4.std(): .1f}')
      plt.legend();
```



1.12 Fitting with `curve_fit`

The following is an example of fitting using the `curve_fit` function from the `scipy` module.

[32]:
```python
# define some x data
example_x = example_data[0,:]
example_x
```

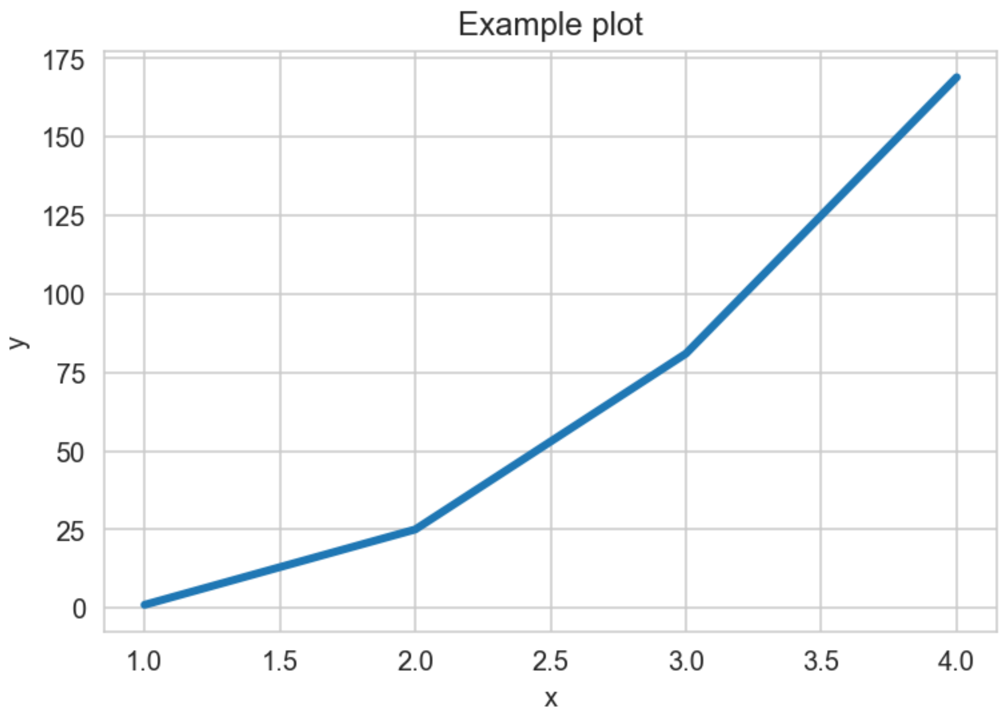
[32]:
```
array([1, 2, 3, 4])
```

[33]:
```python
# define some y data
example_y = example_data[:,0]**2
example_y
```

[33]:
```
array([  1,  25,  81, 169])
```

After defining our data, the first thing to do is to plot it so that we can see how it bahaves in order to choose a fit function.

[34]:
```python
plt.plot(example_x, example_y, linewidth = 3)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Example plot');
```



Now we will fit the data by defining a function.

In the above case, we will define a function with parameters `a` and `b`, of the form $Y = X^a + b$ that we will use to fit our data.

[35]:
```
def fit_func(x, a=1, b=1):
    return x**a + b
```

> Note that syntax matters here. The colon is required, as well as the indent for `return`. Any deviations from the accepted syntax will give errors. the function should take all necessary parameters i.e. `fit_func(x, a, b)`, and you can also choose to pass default values. For example, `a=1` and `b=1` when you create the function.

The line below will use the function wedeinfed above, called `fit_func`, to fit our example data. It returns a tuple of arrays, where the first array, `example_results`, contains the optimized parameters `a` and `b`.
Note that we will not be using the covariance matrix in this problem set.

[36]:
```
example_results, cov = curve_fit(fit_func, example_x, example_y)
example_results, cov
```

[36]:
```
(array([ 3.66882804, 11.05803497]),
 array([[ 5.00134926e-03, -3.68725585e-01],
        [-3.68725585e-01,  6.77501653e+01]]))
```

Now that we have the best fit parameters `a` and `b` for the equation $Y = X^a + b$ we can create the best fit line.

First we need to define the range for x. In this case we us `np.linespace()` to get 50 evenly spaced x-Values from 1.0 to 4.0 > numpy uses the convention (start, stop, number of steps)

```
[37]: example_xfit = np.linspace(1.0, 4.0, 50)
      example_xfit
```

```
[37]: array([1.        , 1.06122449, 1.12244898, 1.18367347, 1.24489796,
             1.30612245, 1.36734694, 1.42857143, 1.48979592, 1.55102041,
             1.6122449 , 1.67346939, 1.73469388, 1.79591837, 1.85714286,
             1.91836735, 1.97959184, 2.04081633, 2.10204082, 2.16326531,
             2.2244898 , 2.28571429, 2.34693878, 2.40816327, 2.46938776,
             2.53061224, 2.59183673, 2.65306122, 2.71428571, 2.7755102 ,
             2.83673469, 2.89795918, 2.95918367, 3.02040816, 3.08163265,
             3.14285714, 3.20408163, 3.26530612, 3.32653061, 3.3877551 ,
             3.44897959, 3.51020408, 3.57142857, 3.63265306, 3.69387755,
             3.75510204, 3.81632653, 3.87755102, 3.93877551, 4.        ])
```

Now generate the y-values for your fit by passing the `x` array to `fit_func`.

```
[38]: example_yfit = fit_func(example_xfit, *example_results)
      example_yfit
```

```
[38]: array([ 12.05803497,  12.30163983,  12.58578627,  12.91445095,
              13.29175834,  13.72197818,  14.20952304,  14.75894608,
              15.37493891,  16.06232963,  16.8260809 ,  17.6712882 ,
              18.60317811,  19.62710671,  20.74855803,  21.97314263,
              23.30659614,  24.754778  ,  26.32367009,  28.01937559,
              29.84811772,  31.81623867,  33.93019844,  36.19657385,
              38.62205746,  41.21345664,  43.97769259,  46.92179942,
              50.05292328,  53.37832146,  56.90536159,  60.64152082,
              64.59438501,  68.77164798,  73.18111078,  77.83068094,
              82.72837175,  87.88230163,  93.30069341,  98.99187371,
             104.96427225, 111.22642131, 117.78695506, 124.65460898,
             131.83821931, 139.34672244, 147.18915439, 155.37465027,
             163.91244374, 172.81186647])
```

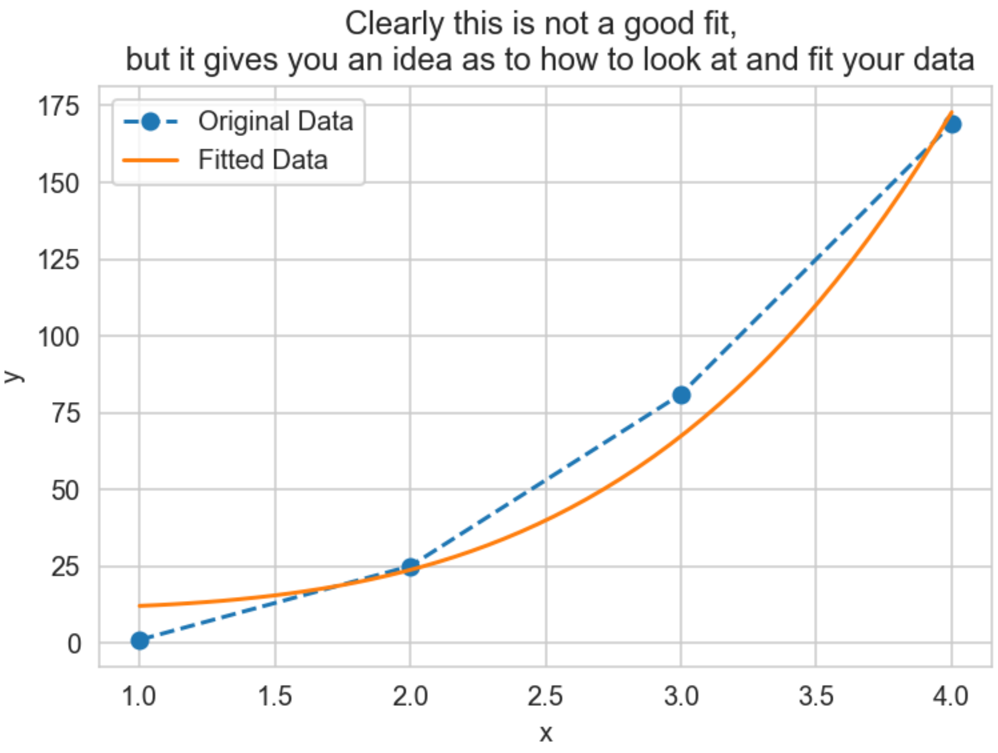
In the previous cell, we calculated the fit for each element in the array `x_fit` and passed `*results` which contains the optimized `a` and `b` parameters.

> Note: the `*` is a special Python syntax that unwraps the elements of the object. Without it you would need to specify the elements in `results` by slicing, e.g. `results[0]` and `results[1]`

Now, plot your data and print the values for `a` and `b`.

```
[39]: plt.plot(example_x, example_y, '--o', label='Original Data') # plot raw data
      plt.plot(example_xfit, example_yfit,  label='Fitted Data') # plot fit data
      plt.title('Clearly this is not a good fit, \n but it gives you an idea as to␣
      ↪how to look at and fit your data')
      plt.xlabel('x')
      plt.ylabel('y')
      plt.legend() #this prints the legend on the graph with the labels you specified␣
      ↪in the `plt.plot` function

      print(f'Variable a = {example_results[0]: .2f}')
      print(f'Variable b = {example_results[1]: .2f}')
```

```
Variable a =  3.67
Variable b =  11.06
```



1.13 Fitting with `lmfit`

LMFit provides a high-level interface to non-linear optimization and curve fitting problems for Python. It builds on and extends many of the optimization methods of `scipy.optimize`. Initially inspired by (and named for) extending the Levenberg-Marquardt method from `scipy.optimize.leastsq`, `lmfit` now provides a number of useful enhancements to optimization and data fitting problems.

One useful enhancement is the parameter and results classes. We will now perform the same example fit using `lmfit` to demonstrate the difference.

See the `lmfit` documentation for help using this package.

[40]:
```
import lmfit
print('lmfit version:', lmfit.__version__) #it is good to know exactly what␣
↪version of a package you are using
```

```
lmfit version: 1.0.2
```

[41]:
```
# define the model
example_model = lmfit.Model(fit_func)
```

`lmfit.Model` uses default values as initial values for fitting. Parameters can have constraints (i.e. boundaries: `min` and/or `max`), for example,

[42]:
```
example_model.set_param_hint('sigma', min=0)
```

[43]:
```
# perform fit
example_params = example_model.make_params()
# print parameters' initial values and constraints
example_params
```

[43]:
```
Parameters([('a', <Parameter 'a', value=1, bounds=[-inf:inf]>),
            ('b', <Parameter 'b', value=1, bounds=[-inf:inf]>),
            ('sigma', <Parameter 'sigma', value=-inf, bounds=[0:inf]>)])
```

Now, we can fit the model to the data,

[44]:
```
# perform the fit
example_results = example_model.fit(example_y, x=example_x)
# print statistics and  parameters' optimized fit values and constraints
example_results
```

[44]:
```
<lmfit.model.ModelResult at 0x7fc67039dcd0>
```

> **NOTE** You can optionally pass initial values for the fitted parameters (e.g. `model.fit(y, x=x, mean=2)`). When no inital values are speciefied they are taken from the defaults values in the function definition (e.g. `fit_func`).

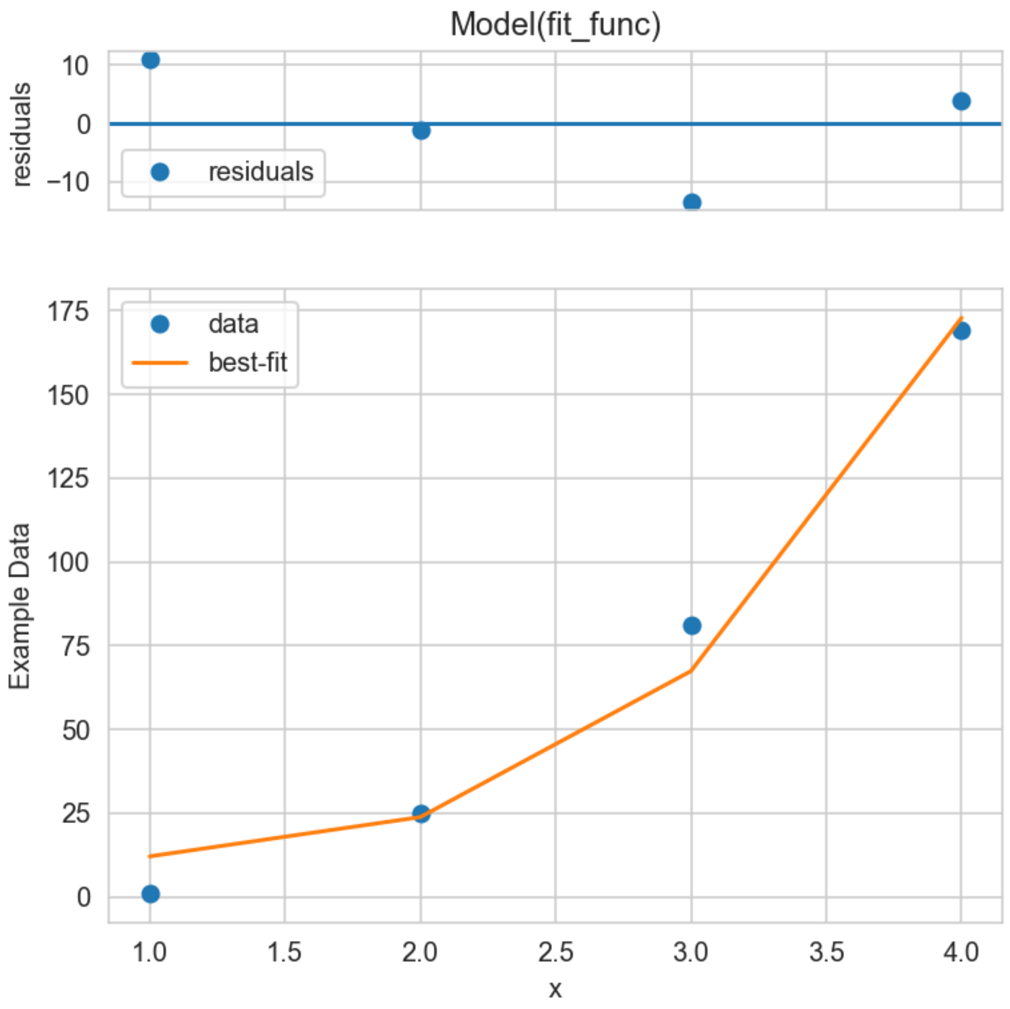
Results can be accessed from `example_results`. For example,

[45]:
```
example_results.chisqr
```

[45]:
```
324.5263269344105
```

The `lmfit` object `example_results` supports plotting, and will generate a plot of the data and fit as well as the residuals.

[46]:
```
fig = example_results.plot()
plt.ylabel('Example Data');
```



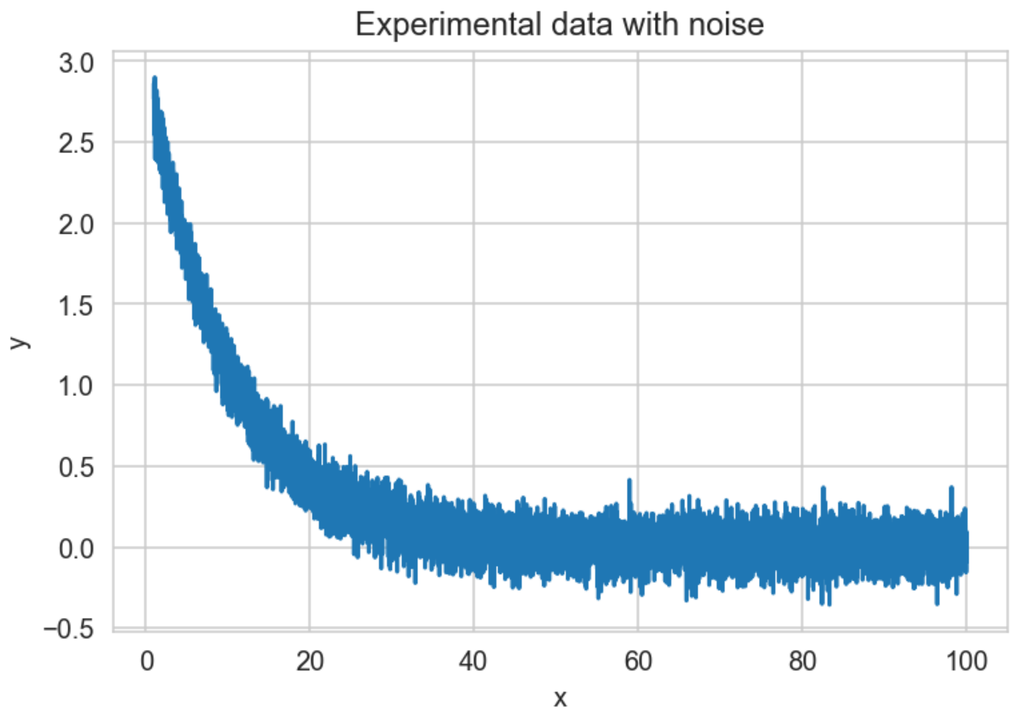
Note for advanced users

In the command above, we used `example_results.plot`, meaning that the function `plot` comes from the results. These kinds of functions contained in an object (and doing some operation on the object) are called 'methods.' In other words, `results` has a plot method so it knows how to plot itself.

1.14 Solution 6

The `curve_fit` function calculates the optimal parameters by minimizing the sum of the squared residuals.

[47]:
```
plt.plot(provided_x, provided_y);
plt.xlabel('x')
plt.ylabel('y')
plt.title('Experimental data with noise');
```



[48]:
```python
def exponential_fit(x, a=1, b=1, c=1):
    return a*np.exp(-b*x) + c
```

[49]:
```python
# test your function
exponential_fit(0)
```

[49]:
```
2.0
```

[50]:
```python
# calculate optimized fit parameters and covariance matrix
results, cov = curve_fit(exponential_fit, provided_x, provided_y)

print(f'Fitted variables a = {results[0]: .3f} \n\t\t b = {results[1]: .4f}␣
↪\n\t\t c = {results[2]: .4f}')
```

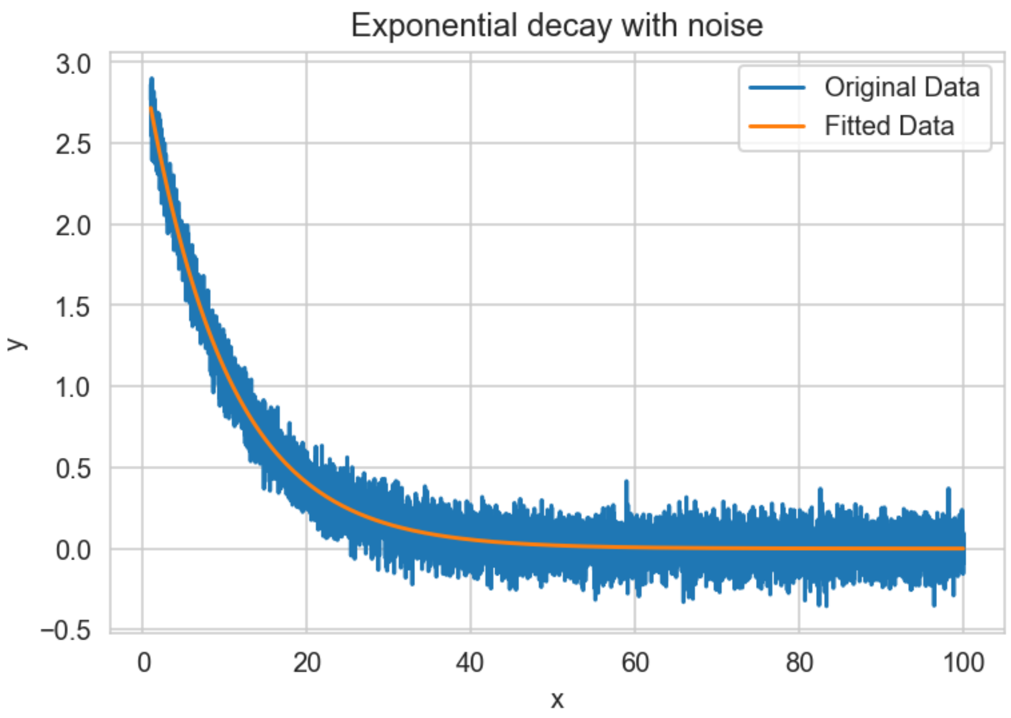
```
Fitted variables a =  3.003
                 b =  0.0999
                 c = -0.0005

<ipython-input-48-107f6914f1c5>:2: RuntimeWarning: overflow encountered in exp
  return a*np.exp(-b*x) + c
<ipython-input-48-107f6914f1c5>:2: RuntimeWarning: overflow encountered in
multiply
  return a*np.exp(-b*x) + c
```

Create x and y arrays for the fit results:

```
[51]: x_fit = np.linspace(1.0, 100.0, 200)
      y_fit = exponential_fit(x_fit, *results)
```

```
[52]: plt.plot(provided_x, provided_y, label='Original Data')
      plt.plot(x_fit, y_fit, label='Fitted Data')
      plt.xlabel('x')
      plt.ylabel('y')
      plt.title('Exponential decay with noise')
      plt.legend()

      print(f'Fitted variables a = {results[0]} \n\t\t b = {results[1]} \n\t\t c =␣
       ↪{results[2]}')
```

```
Fitted variables a = 3.0032972278233325
                 b = 0.09993498517292466
                 c = -0.0004977137634771056
```



1.15 Solution 7

```
[53]: model = lmfit.Model(exponential_fit)
```

```
[54]: model.set_param_hint('sigma', min=0)
```

```
[55]: params = model.make_params()
      params
```

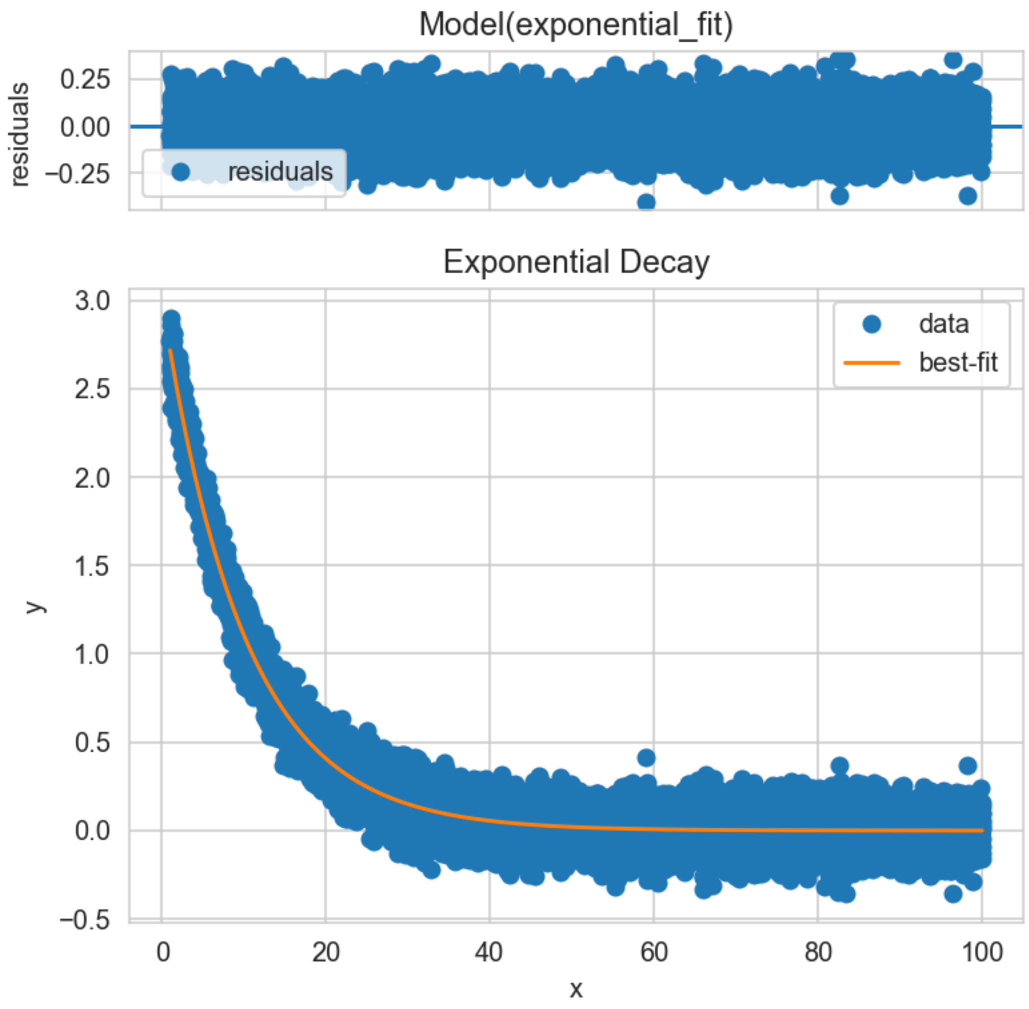
```
[55]: Parameters([('a', <Parameter 'a', value=1, bounds=[-inf:inf]>),
                  ('b', <Parameter 'b', value=1, bounds=[-inf:inf]>),
                  ('c', <Parameter 'c', value=1, bounds=[-inf:inf]>),
                  ('sigma', <Parameter 'sigma', value=-inf, bounds=[0:inf]>)])
```

```
[56]: results = model.fit(provided_y, x=provided_x, nan_policy='propagate')
      results
```

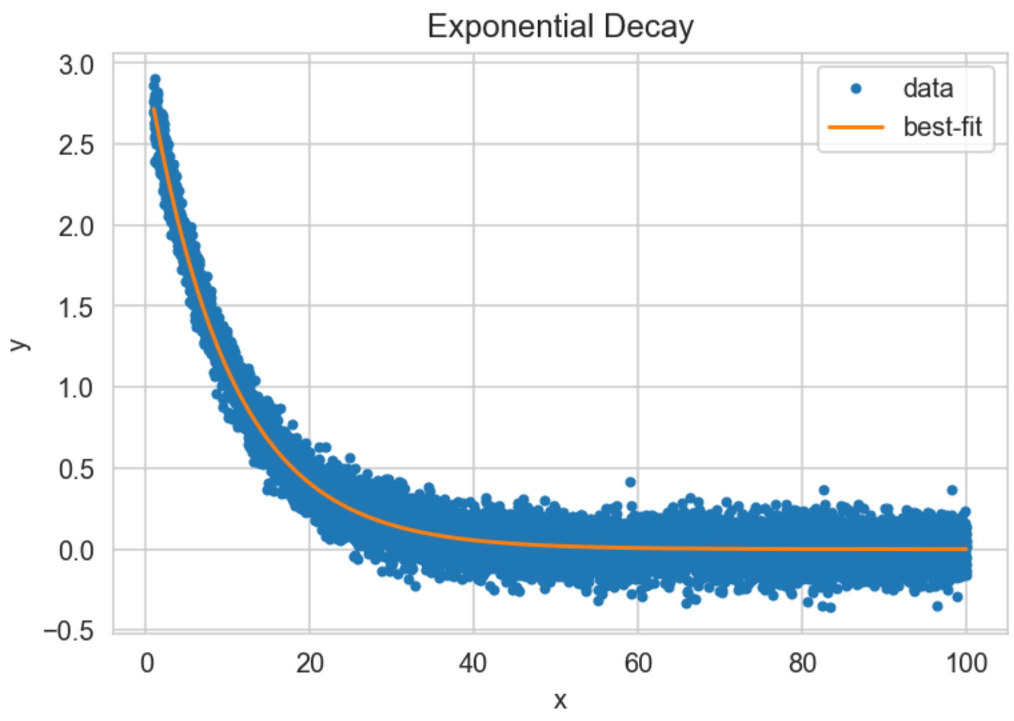
```
[56]: <lmfit.model.ModelResult at 0x7fc620ac86a0>
```

Now, we can fit the model to the data,

```
[57]: fig = results.plot()
      plt.ylabel('y')
      plt.xlabel('x')
      plt.title('Exponential Decay');
```



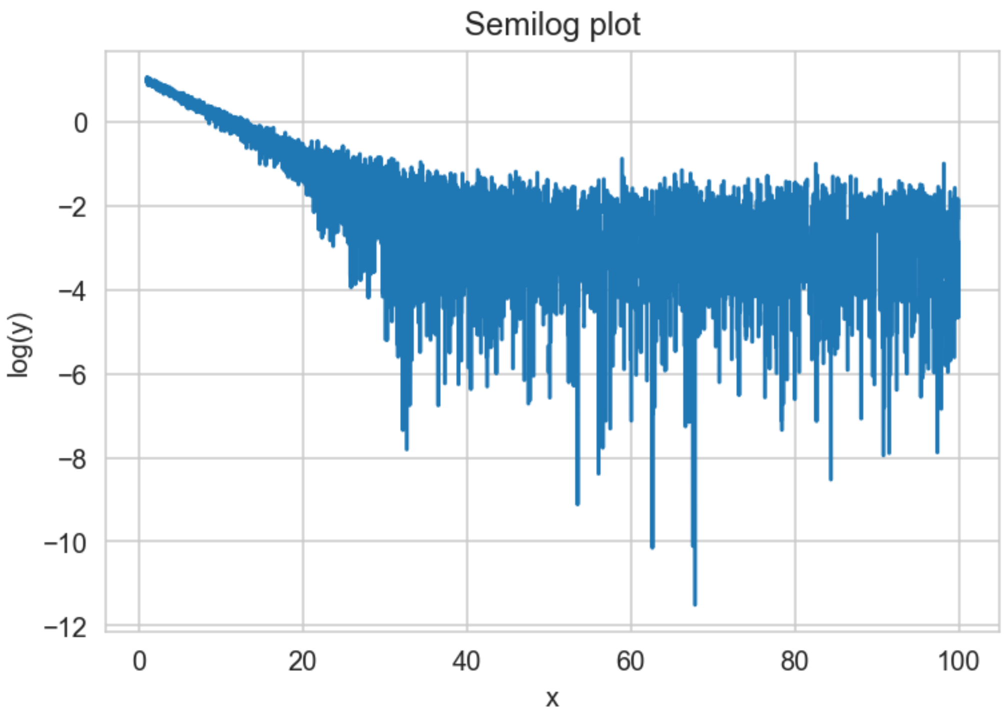
```
[58]: plt.plot(provided_x, provided_y, '.', label='data');
      plt.plot(provided_x, results.best_fit, label='best-fit');
      plt.title('Exponential Decay');
      plt.xlabel('x');
      plt.ylabel('y');
      plt.legend();
```



An exponential function will be linear on the semilog scale. A bi-exponential function will appear as a sum of two lines, where each line has a different slope. This is useful for identifing the number of species in a mixture.

[59]:
```
plt.plot(provided_x, np.log(provided_y))
plt.xlabel('x')
plt.ylabel('log(y)')
plt.title('Semilog plot');
```

```
<ipython-input-59-2e58c45d8e2b>:1: RuntimeWarning: invalid value encountered in
log
  plt.plot(provided_x, np.log(provided_y))
```



Note that the data appear to have two slopes, suggesting a sum of exponentials.

1.16 Fitting real data

In this final question, you will load and perfome a fit of real single-molecule spectroscopy data from the Shimon Weiss lab.

> This experimental dataset has been acquired by **Yazan Alhadid** (yalhadid@ucla.edu). For more information on the science behind this kind of measurement see: - A Novel Initiation Pathway in Escherichia Coli Transcription, (2016), Lerner/Chung et al. doi:10.1101/042432

The dataset below contains two experimental kinetic curves representing DNA transcription by RNA polymerase starting from two different initial states ITC2 and ITC7 (meaning that the RNAP has transcribed 2 or 7 ribonucleotides respectively). It is interesting to note that the kinetics of the ITC7 complex are slower than for the ITC2 complex.

Each number in the table is the result of a smFRET measurement and represents a transcription efficiency, E, measured after a certain amount of time. Note that, for some time points the measurement has been performed only in one configuration (either ITC2 or ITC7) so there are missing data points (NaN, i.e. not-a-number).

When fitting this dataset we need to make sure we correctly handle the NaN elements.

1.16.1 Data format

The data has been saved in CSV (comma separated values) in Excel. We will load this data in a pandas DataFrame, a special python object to hold tabular data. See the documentation for pandas to get started, and also checkout this paper, for a short and well-written explaination of the Tidy-Data concept.

Difference between DataFrame and array A numpy array is a container for uniform types of data (for example integers, floats, etc.). A numpy array can be 1-D (a vector), 2-D, or N-D. Conversely, a pandas DataFrame is a table (similar to a 2-D array) in which each column can be of a different type.

Taking a 2-D array, you can access the columns (and rows) with a with a numerical index (0, 1, …). So you need to know the meaning of each row/column.

Conversely, in pandas DataFrame each column has a name, and you can select a column by name. The first column of a DataFrame is a special column called the Index. The index is used to select rows in the table. In our example the index is the time axis that is common to the two columns representing the two set of measurements.

Essentially, it works like this:

- Each variable you measure should be in one column.
- Each different observation of that variable should be in a different row.
- There should be one table for each “kind” of variable.
- If you have multiple tables, they should include a column in the table that allows them to be linked (for example an index).

[60]:
```
import pandas as pd
```

[61]:
```
datafile = 'ITC2 ITC7 exit kinetics.csv'
```

1.17 Solution 8

[62]:
```
df = pd.read_csv(datafile) #open .csv as a DataFrame
df
```

[62]:
```
   Time(s)  ITC2  ITC7
0       60  13.0   NaN
1      300  41.0  32.0
2      420  43.0   NaN
3      600  75.0  28.0
4      900  81.0  37.0
5     1200  89.0  62.0
6     1500  91.0   NaN
7     1800   NaN  82.0
8     2700   NaN  91.0
```

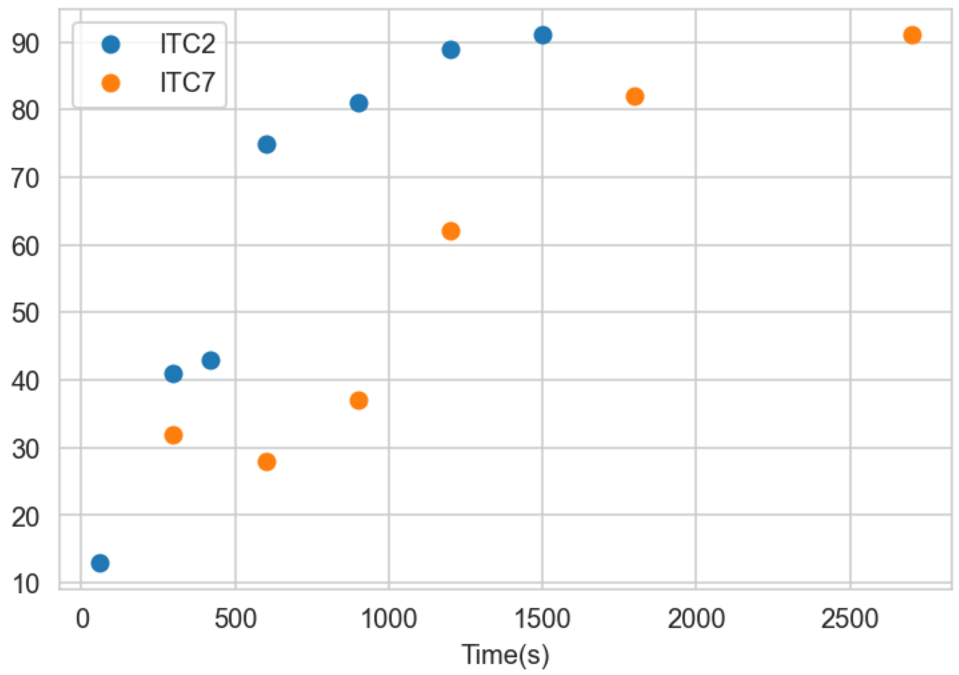
Note that the default setting is to create an index column, denoted in bold. However, is it often useful to set the index column to the zeroth column corresponding to your independent variable.

```
[63]: df = pd.read_csv(datafile, index_col=0) #set index column to be the zeroth row
      df
```

```
[63]:         ITC2  ITC7
      Time(s)
      60      13.0   NaN
      300     41.0  32.0
      420     43.0   NaN
      600     75.0  28.0
      900     81.0  37.0
      1200    89.0  62.0
      1500    91.0   NaN
      1800     NaN  82.0
      2700     NaN  91.0
```

A nice feature of DataFrames is that they can be quickly plotted, using the `df.plot` method:

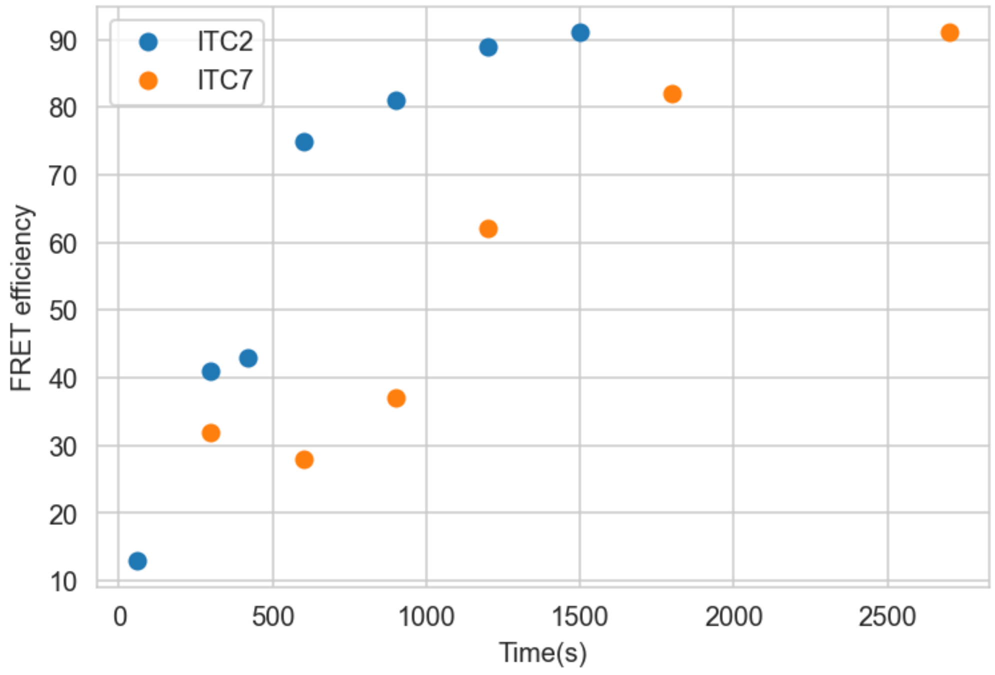
```
[64]: df.plot(marker='o', lw=0);
```



Note that the data of the two columns are plotted against the time axis (the index) and the two columns are automatically labeled in the legend. Also the x-axis has been labeled with the name associated with the index column.

Note that we're still missing the y-label, so let's add it:

[65]:
```python
ylabel = 'FRET efficiency'

ax = df.plot(marker='o', lw=0)
ax.set_ylabel(f'{ylabel}');
```



1.18 Solution 9

1.18.1 Defining the model

We want to fit the data with an exponential curve that starts at 0 for $t = 0$ and goes asymptotically to a value > 0 for $t = \infty$.

First we define a python function representing the mathematical function:

$$y = A\left(1 - e^{-\frac{t}{\tau}}\right)$$

[66]:
```python
def increasing_exp(x, tau=1, amplitude=1):
    return (1 - np.exp(-x/tau))*amplitude
```

[67]:
```python
kinetics_model = lmfit.Model(increasing_exp)
```

[68]:
```python
kinetics_model.set_param_hint('amplitude', min=0)
kinetics_model.set_param_hint('tau', value=100, min=0)
```

As a check, we print the resulting initial parameters:

```
[69]: kinetics_params = kinetics_model.make_params()
      kinetics_params
```

```
[69]: Parameters([('tau', <Parameter 'tau', value=100, bounds=[0:inf]>),
                  ('amplitude', <Parameter 'amplitude', value=1, bounds=[0:inf]>)])
```

Finally we fit the model to the ITC2 data and print the fit results. In order to specify a column in the DataFrame you can use the following syntax:

`df.ITC2` or `df.index` etc.

Here we specify a method of fitting, i.e. `method=nelder` and we pass `nan_policy='omit'` to deal with NaN elements.

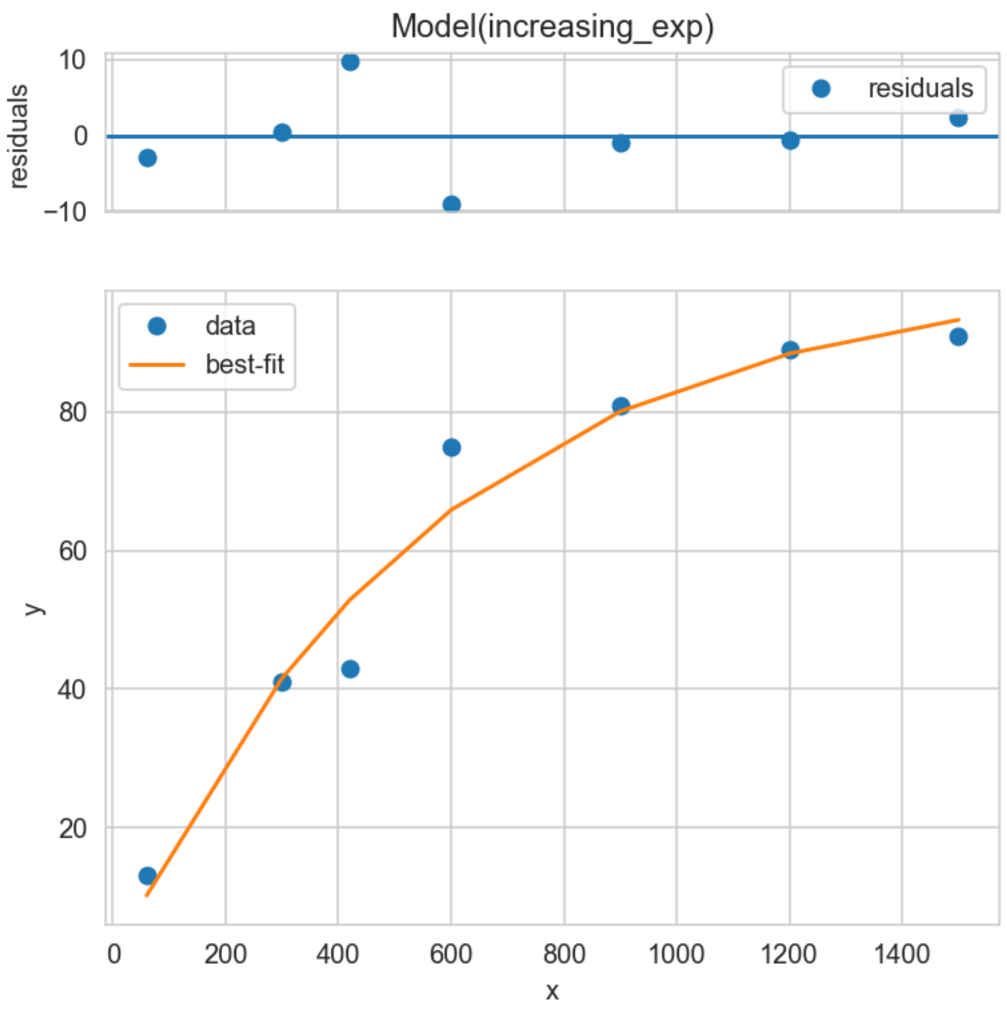
```
[70]: results_itc2 = kinetics_model.fit(df.ITC2, x=df.index, method='nelder',␣
       ↪nan_policy='omit')
      results_itc2
```

```
[70]: <lmfit.model.ModelResult at 0x7fc6702cc880>
```

Now plot your results for the ITC2 fit:

```
[71]: results_itc2.params.pretty_print()
      fig2 = results_itc2.plot()
```

```
Name          Value      Min      Max   Stderr     Vary     Expr Brute_Step
amplitude     100.2        0      inf     None     True     None       None
tau           560.5        0      inf     None     True     None       None
```



1.19 Solution 10

We fit the same model to the ITC7 data:

```
[72]: results_itc7 = kinetics_model.fit(df.ITC7, x=df.index, method='nelder',␣
       ↪nan_policy='omit')
      results_itc7
```

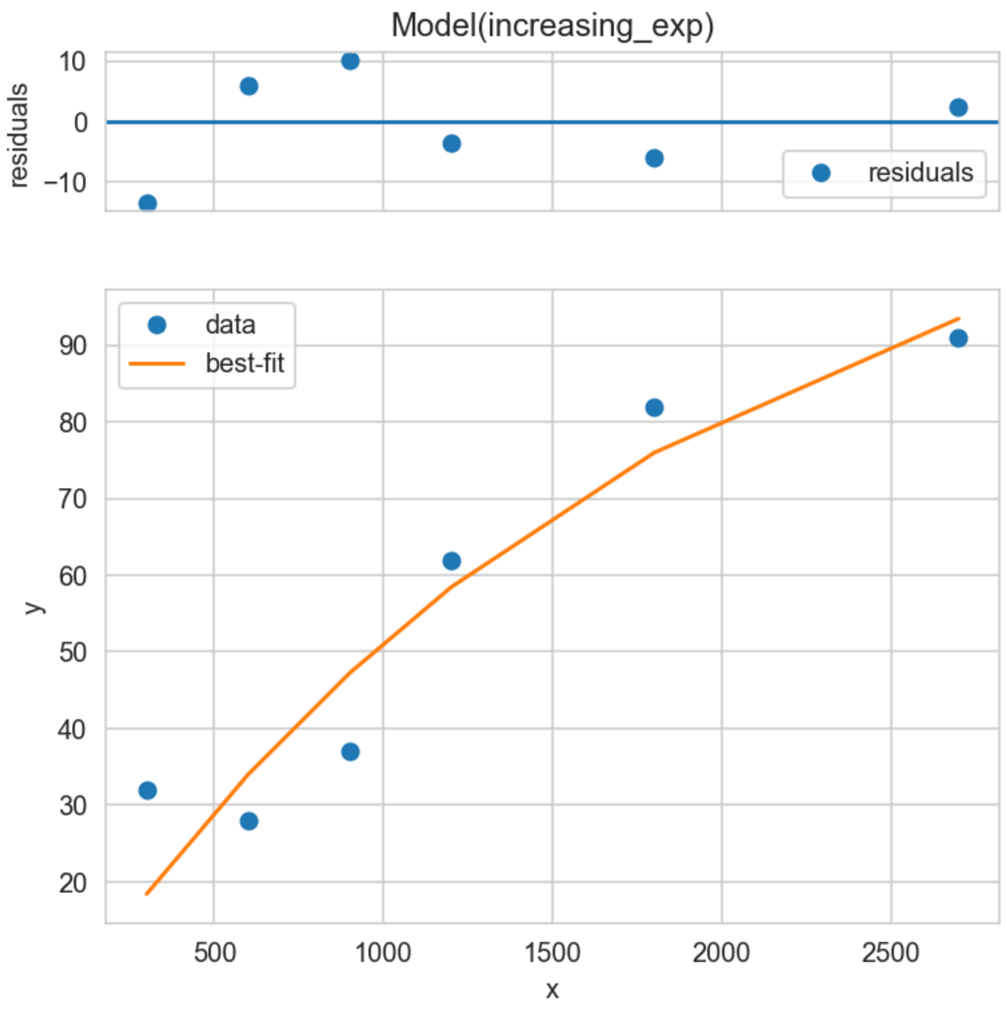
```
[72]: <lmfit.model.ModelResult at 0x7fc670271310>
```

Plot your results for the ITC7 fit:

```
[73]: results_itc7.params.pretty_print()
      fig7 = results_itc7.plot()
```

```
Name           Value       Min      Max   Stderr      Vary      Expr Brute_Step
amplitude      120.6         0      inf     None      True      None       None
```

```
tau            1812      0      inf     None     True     None     None
```



1.20 Fit comparison

As a last step we create a single plot which shows the two datasets and the fitted curves.

First, we need to define an array for the time axis called `t`,

[74]:
```
t = np.arange(0, 3000)
```

We can then save the fiting results from the two data sets into a dictionary using the following syntax,

[75]:
```
# create dictionary of parameters using the syntax: 'keys', values
results_itc2.params.valuesdict()
```

[75]:
```
OrderedDict([('tau', 560.4829564943957), ('amplitude', 100.19607850202087)])
```

Now, we create our y data,

```
[76]: y2 = increasing_exp(t, **results_itc2.params.valuesdict())
      y2
```

```
[76]: array([ 0.        ,  0.17860801,  0.35689764, …, 99.71898612,
             99.71983658, 99.72068552])
```

The `**` is a special syntax for unpacking a dictionary.

1.21 Problem 11

- create the y data for the ITC7 data set and name it `y7`
- use `matplotlib` to make a plot with both data sets and both fits
    - be sure to include a title, axes labels, and a legend
- print the values for τ using the general synthax `tau = results.params['tau'].value`

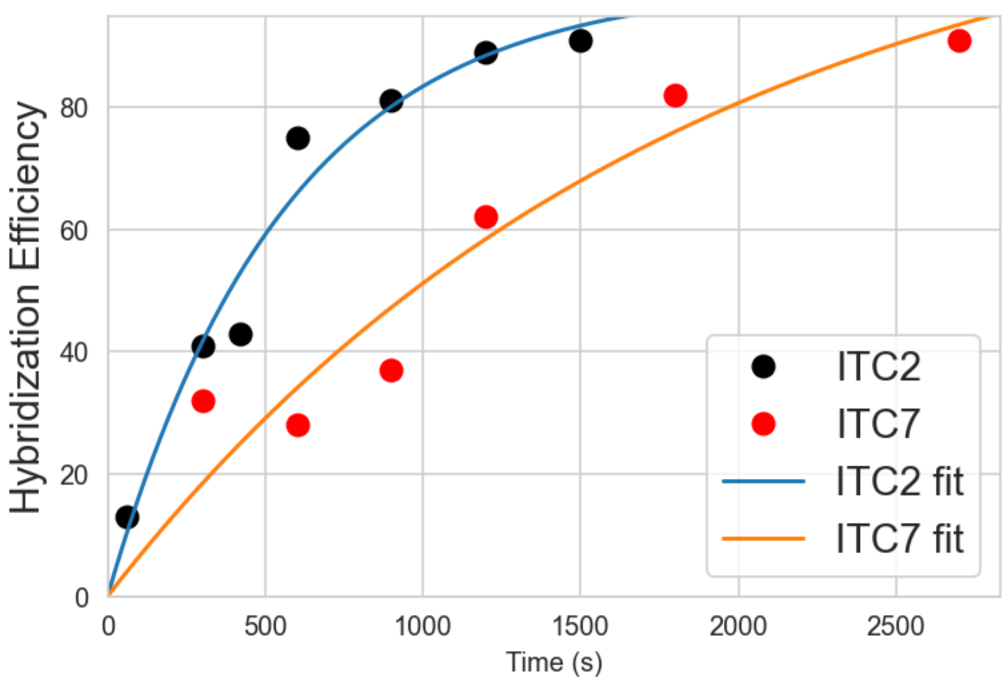
1.22 Solution 11

```
[77]: y7 = increasing_exp(t, **results_itc7.params.valuesdict())
      y7
```

```
[77]: array([0.00000000e+00, 6.65617007e-02, 1.33086680e-01, …,
             9.75688169e+01, 9.75815506e+01, 9.75942772e+01])
```

```
[78]:  2 = results_itc2.params['tau'].value
       7 = results_itc7.params['tau'].value

      plt.plot('ITC2', data=df, lw=0, marker='o', ms=8, color='black')
      plt.plot('ITC7', data=df, lw=0, marker='o', ms=8, color='red')
      plt.ylim(0)
      plt.xlim(0)
      plt.ylabel('Hybridization Efficiency', fontsize =15)
      plt.xlabel('Time (s)')

      plt.plot(t, y2, label = 'ITC2 fit')
      plt.plot(t, y7, label = 'ITC7 fit')
      plt.legend(loc='best', fontsize = 15)
      print(f'ITC2 fit, tau = { 2}')
      print(f'ITC7 fit, tau = { 7}')
```

```
ITC2 fit, tau = 560.4829564943957
ITC7 fit, tau = 1812.1020451320815
```


Hybridization Efficiency
Time (s)
0
20
40
60
80
0
500
1000
1500
2000
2500
ITC2
ITC7
ITC2 fit
ITC7 fit

```
[ ]:
```

```
[ ]:
```