A Data Science and Visualization Primer

Linear Regression – a Key Tool in a Scientist's Arsenal

assoc. Prof. Dr. Thomas Hofer
Department of Theoretical Chemistry
Email: T.Hofer@uibk.ac.at

Stefanie Kröll, MSc
Department of Theoretical Chemistry
Email: Stefanie.Kroell@uibk.ac.at

Welcome to the Third Exercise Block

In this example, we will go over different examples presenting variations of Linear Regression. This technique is a particularly important means of calibration/analysis relevant in every scientific discipline.

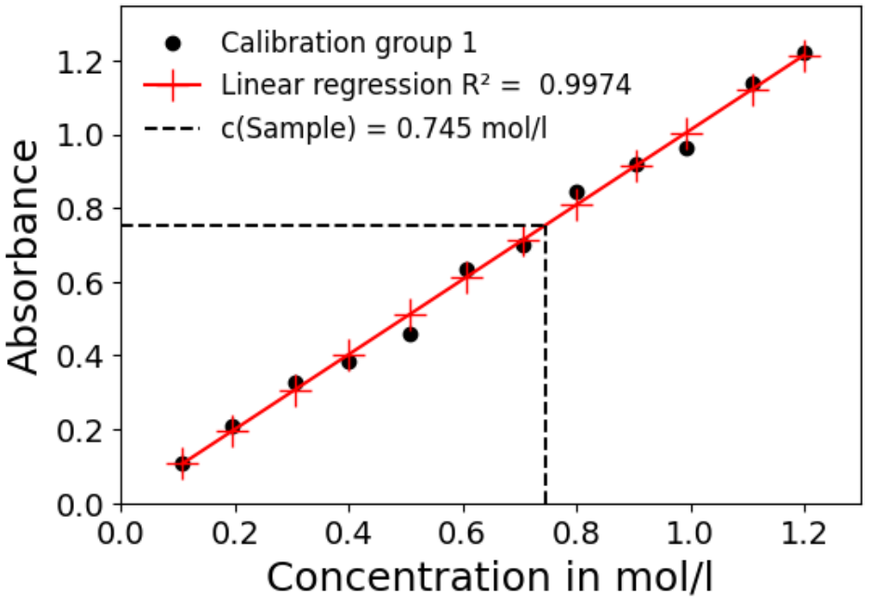
The goal is to identify a linear equation $\bar{y} = a \cdot x + b$ so that the sum of residuals R is minimized:

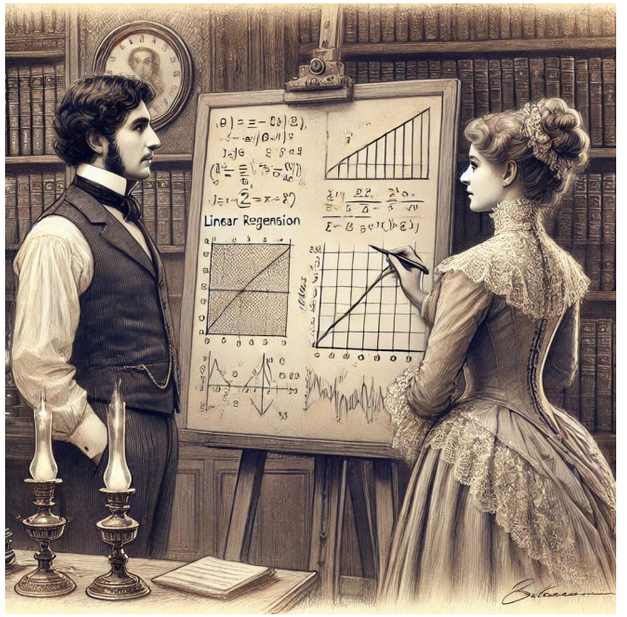
$$R = \sum_{i=1}^{n} (y_i - \bar{y}_i)^2$$

Although the calculation of a and b looks intricate, we only require the calculation of simple sum expression:

$$\textit{Slope} \qquad a = \frac{n \sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}$$

$$\textit{Intercept} \qquad b = \frac{\sum_{i=1}^{n} y_i - a \sum_{i=1}^{n} x_i}{n}$$





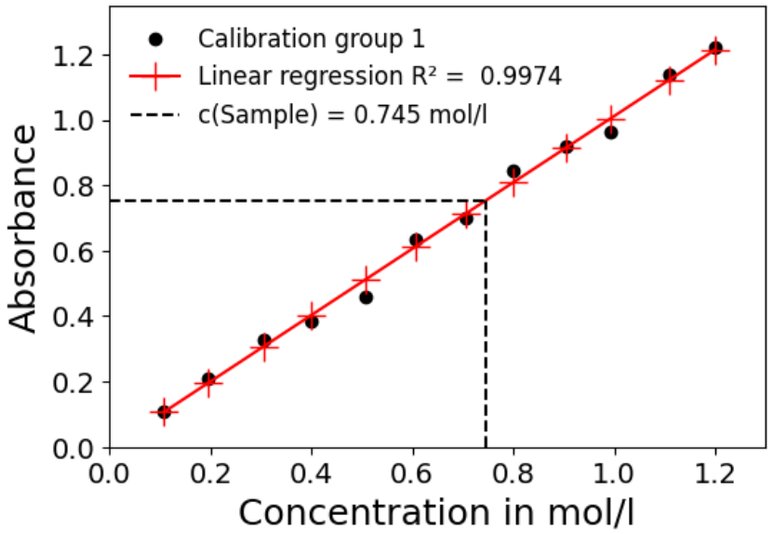
Exercise B.1 - Calibration of Infrared or UV/Vis Spectroscopy

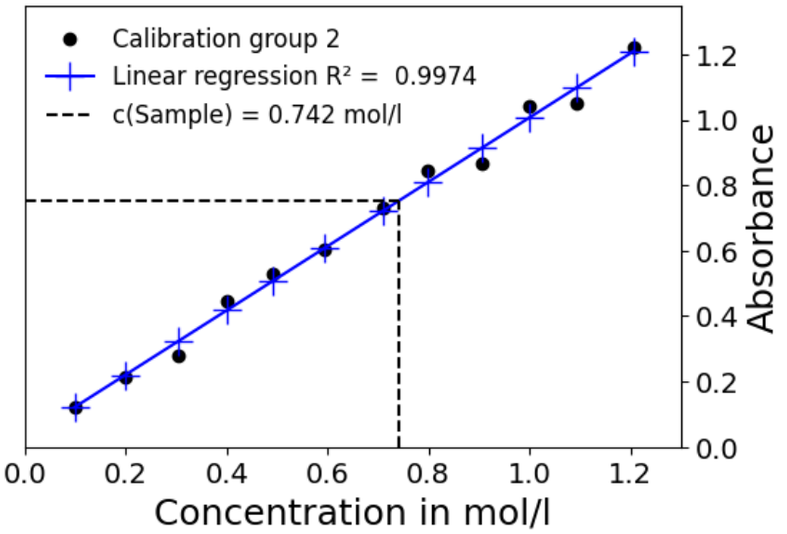
In quantitative spectroscopy we measure the (dimensionless) absorbance *A* of a beam of light at a selected frequency/wavelength to determine the concentration *c* of the sample.

To do this, we first calibrate the device using solutions with known concentration.

In our exercise we compare the calibrations of two groups of students available in the files *ir_spectroscopy1.dat* and *ir_spectroscopy2.dat*.

From that we have to predict the concentration of our unknown sample measured at the absorption of A_{Sample}= 0.753





Let's start by importing the respective files.

Loading the Files and Setting Up the Data (*low difficulty*)

Similar as in Excercise A we upload the data files to colab. Alternatively, providing access *via* google drive is possible.

```
from google.colab import files

uploaded = files.upload()
```

In the next cell we load the required python modules (again NumPy and Matplotlib).

We load the data from the textfiles using again `np.loadtext()`. into the arrays `data1` and `data2`.

Pro Tip: For many projects it is a good idea to extract the data into distinct arrays. In this case we use `c1` and `c2` to store the concentration data, the respective absorbance is stored in `A1` and `A2`.

```
import numpy as np
import matplotlib.pyplot as plt

data1 = np.loadtxt('ir_spectroscopy1.dat')
data2 = np.loadtxt('ir_spectroscopy2.dat')

c1 = data1[:, 0]
A1 = data1[:, 1]

c2 = data2[:, 0]
A2 = data2[:, 1]
```

Storing data in separate array does not require any additional memory but makes working with data sets that much easier.🤓

Always Double-Check Your Workflow (… or It Will Be Wrong)

Did we read the data correctly ... ?
Is the data even linear ... ?

```
plt.plot(c1, A1, color='k', marker = 'o', linestyle = ' ')
plt.plot(c2, A2, color='r', marker = 'o', linestyle = ' ')
```

It is always a good idea to have a look at the data before starting any analysis.

Let's make a quick plot(without title, axis label, *etc.*) for orientation.

As before we use the `plt.plot()` function of Matplotlib.

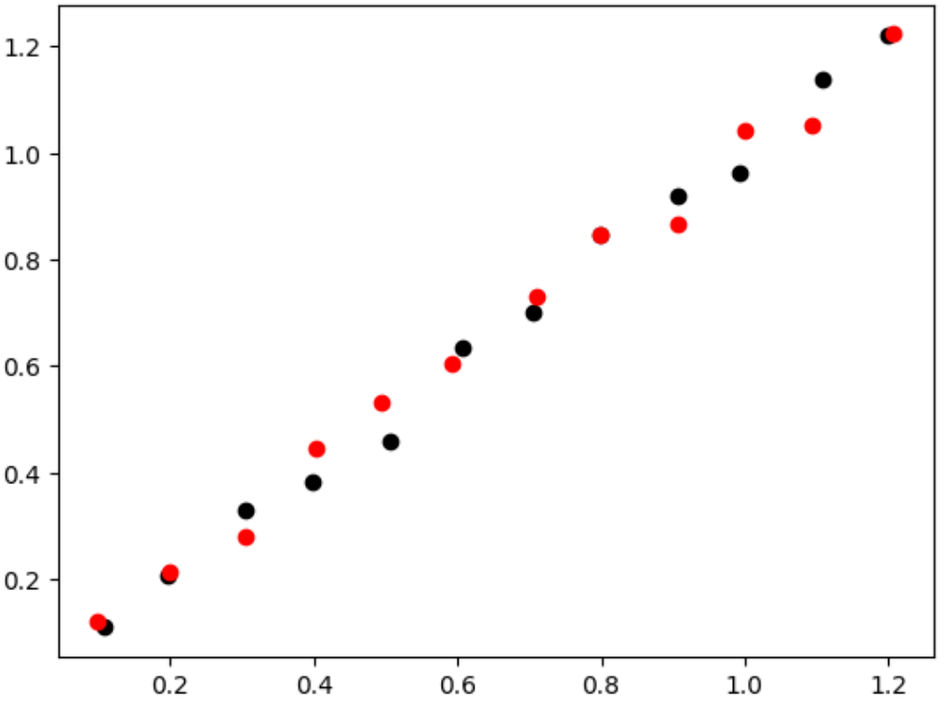
Using `marker = 'o'` and `linestyle = ' '` we don't show lines but the individual data points.

Alternative marker options include:

`'+','x', '.', '^', 'v', '*'`

Alternative linestyles include:

`'-', '--','-.', ':'`



Linear Regression – Three Popular Options, But only One Winner

Let's fit the data to our selection and plot the regression.

Method 1: The easiest methods is using NumPy's polynomial fit `np.polyift()`:

$$y = a + bx + cx^2 + dx^3 + ...$$

Choosing a degree of 1 yields a linear regression.

No R^2 (coefficient of determination) available!

```
reg_np_1 = np.polyfit(c1, A1, 1)

print(f"slope   : {reg_np_1[0]: 7.4f}")
print(f"interc. : {reg_np_1[1]: 8.4e} l/mol")
```

Method 2: Linear regression of SciKit-Learn.

Get's the job done, but is quite complicated.

```
from sklearn.linear_model import LinearRegression

x = c1.reshape(-1, 1)
y = A1

reg_sk_1 = LinearRegression()
reg_sk_1.fit(x, y)

print(f"slope   : {reg_sk_1.coef_[0]: 7.4f}")
print(f"interc. : {reg_sk_1.intercept_: 8.4e} l/mol")
print(f"R²      : {reg_sk_1.score(x, y): 7.4f}\n")
```

Method 3: Linear regression of SciPy

Needs to load one more module in addition to NumPy, but overall pretty straightforward.

Careful: Provides *R*, so calculate R^2 yourself!

```
from scipy.stats import linregress

reg_sp_1 = linregress(c1, A1)

print(f"slope   : {reg_sp_1[0]: 7.4f} K")
print(f"interc. : {reg_sp_1[1]: 8.4e}")
print(f"R²      : {reg_sp_1[2]**2: 7.4f}"))
```

Plotting the Linear Regression and Highlight the Measured Sample

- It is a good idea to extract the slope, intercept and R^2 from the regression data (optional).
- To calculate the concentration of the sample from the absorbance we simply rearrange the linear regression equation to yield x from y:

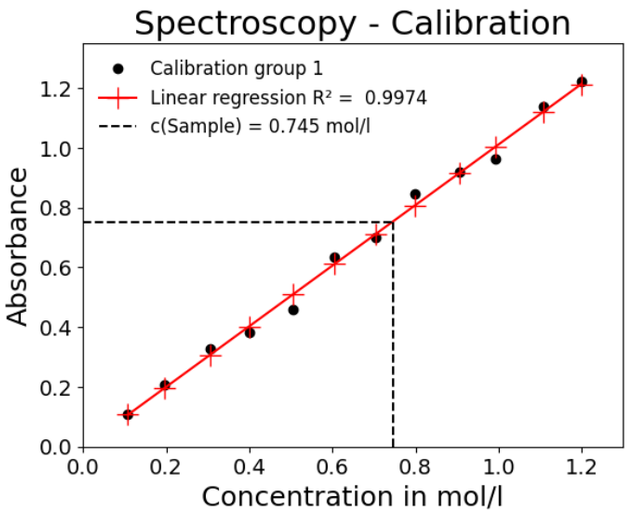
$$\bar{x} = \frac{y - b}{a}$$

- A good strategy to prepare regression plots is:
  - Plot the original data as points.
  - Plot the regression as line with different markers.
  - Add horizontal and vertical lines for a representative point by defining the start- and end points manually, *e.g.* for a horizontal line:

```
plt.plot([0, c_sample1], [abs_sample, abs_sample], color= ...)
```

x_{Start} x_{End} y_{Start} y_{End}

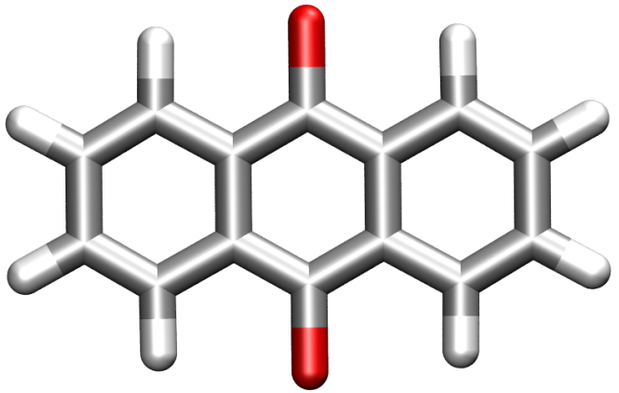
```
a1 = reg_sp_1[0]
b1 = reg_sp_1[1]
R1_squared = reg_sp_1[2]**2

abs_sample = 0.753
c_sample1 = (abs_sample - b1) / a1

plt.plot(c1, A1, color='k', marker='o', linestyle=' ',
         label ="Calibration group 1")

plt.plot(c1, a1*c_sample1+b1, color='r', marker='+',
         markersize = 15,
         label =f"Lin. Reg. R² = {R1_squared:7.4f} ")

plt.plot([0, c_sample1], [abs_sample, abs_sample],
         color='k', linestyle = '--', linewidth=1.5,
         label=f'c(Sample) = {c_sample1:5.3f} mol/l')

plt.plot([c_sample1, c_sample1], [0, abs_sample],
         color='k', linestyle = '--', linewidth=1.5)
```

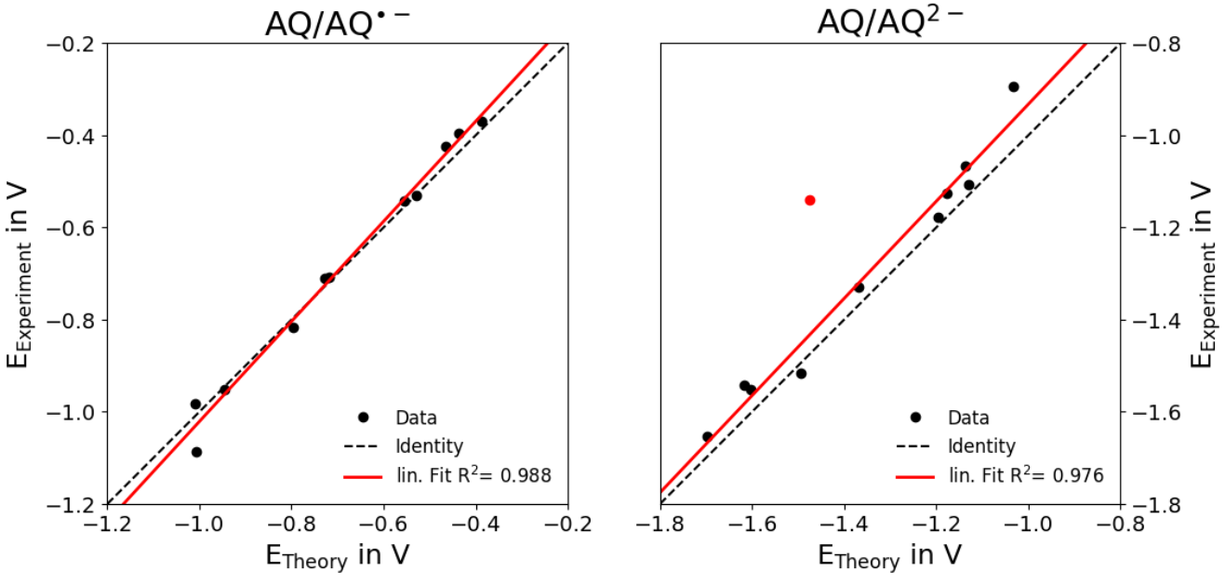


Exercise B.2 - Electrochemical Potential of Anthraquinone Derivatives

In this exercise we compare experimentally measured electrochemical potential of anthraquinone (AQ) derivatives with those calculated using theoretical methods. We want to find out how well the calculation can reproduce the experimental data.

Anthraquinone is an aromatic organic molecule, with manifold applications.





In nature, AQ and its derivatives are found in plants and microorganisms due to its key role in reversible redox reactions. For this reason AQ derivates are also key components in industrial processes and material development.

The data for this exercise is taken from a recent collaboration between the Theoretical Chemistry Department and the Institute for Physical chemistry. (Phys.Chem.Chem.Phys. 2022, 24, 16207 – 16219)

Let's start by loading the data files, but wait ...

Mixed Input Data – Objects and Numbers (*low difficulty*)

- The data files in this example contain text and number data. This requires special reading of the data as `dtype = object`.
  - Column 1: ID of AQ derivatives
  - Column 2: 1st reduction potential in mV
  - Column 3: 2nd reduction potential in mV
- This way the data is read in as text objects. To classify columns 2 and 3 as floating point numbers, we have to convert the data using add `.astype(float)`.
- This step can also be combined with mathematical operations such as dividing by 1000 to convert from mV to V.
- Now the names of the AQ derivatives are stored as text in the array `ID`, while the reduction potentials are stored as numbers in the `redox` arrays.

Contents of file `experimental.dat`:

```
1-OH          -530.0 -1178.5
2-OH          -707.5 -1139.8
1,2-OH        -542.9 -1125.5
1,4-OH        -423.9 -1066.4
1,5-OH        -369.1 -893.5
1,8-OH        -396.4 -1107.0
1-NH2         -816.4 -1516.9
1,2-NH2       -950.6 -1551.2
1,4-NH2       -983.4 -1541.8
2,6-NH2      -1086.0 -1652.3
1-NH2-4-OH    -710.4 -1328.9
```

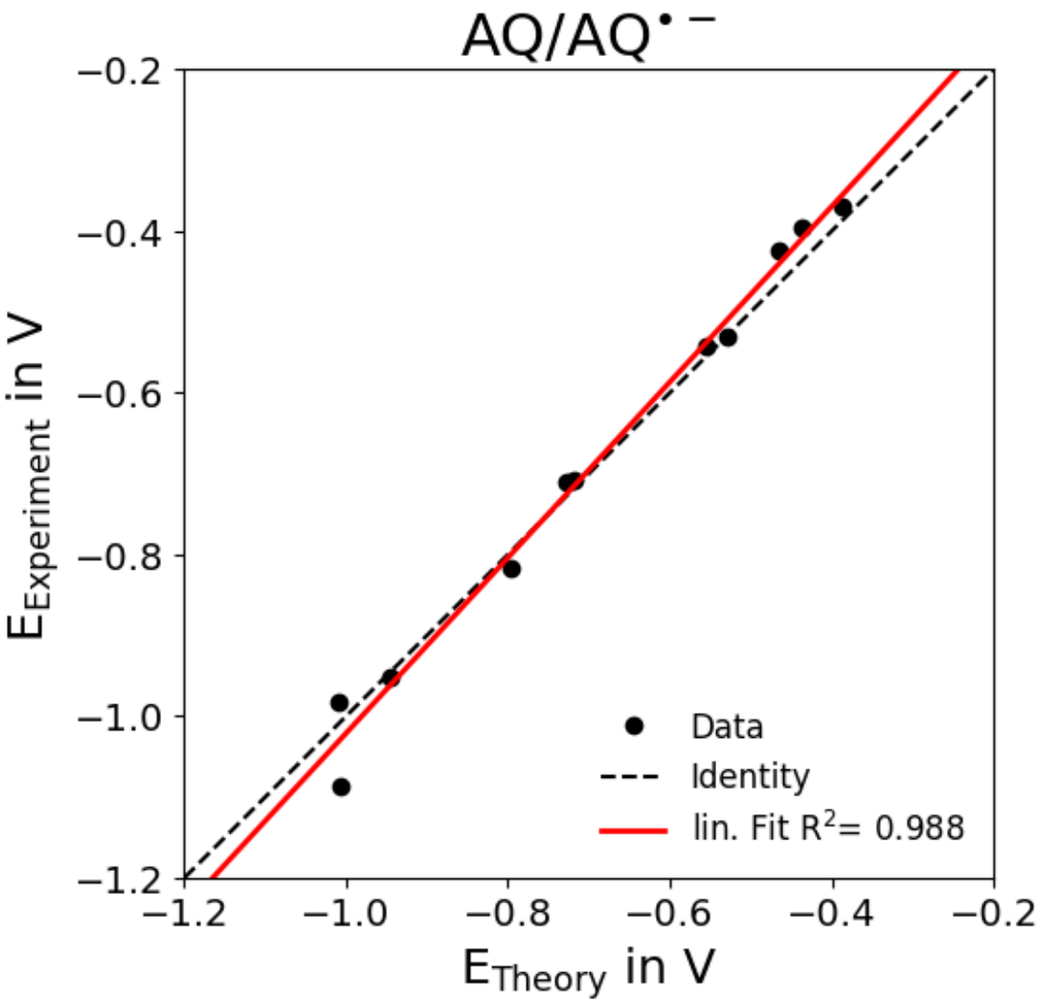
```
import numpy as np
import matplotlib.pyplot as plt

experim = np.loadtxt('experiment.dat', dtype=object)
theory = np.loadtxt('theory.dat', dtype=object)

ID = experim[:, 0]

redox1_exp = experim[:,1].astype(float)/1000
redox2_exp = experim[:,2].astype(float)/1000

redox1_theo = theory[:,1].astype(float)/1000
redox2_theo = theory[:,2].astype(float)/1000
```

Correlation Plots

- Correlation plot are a widely used, lucid way to compare data sets.
- In this case we compare the theoretical data on the x-axis with the experimental values on the y-axis.
- In the best case scenario, we end up with a linear regression having:
  - A slope of exactly 1.0.
  - An intercept of 0.0
  - An R^2-value of 1.0

  } $y = x$

- Always make correlation plots intuitive:
  - The x- and y-range should be identical.
  - The plot should be square (aspect ratio 1:1).
  - Include the ideal diagonal line, *i.e.* $y = x$.



- We already have established that SciPy provides the easiest method.
- Just don't forget that we have to calculate the square of the R-value manually!

Correlation Plots in Python (*low difficulty*)

- The x- and y-range should be identical, so we set suitable x- and y-ranges using again `plt.set_xlim()` and `plt.set_ylim()`.

  Usually requires some trial and error ...

- The plot should be square (aspect ratio 1:1).

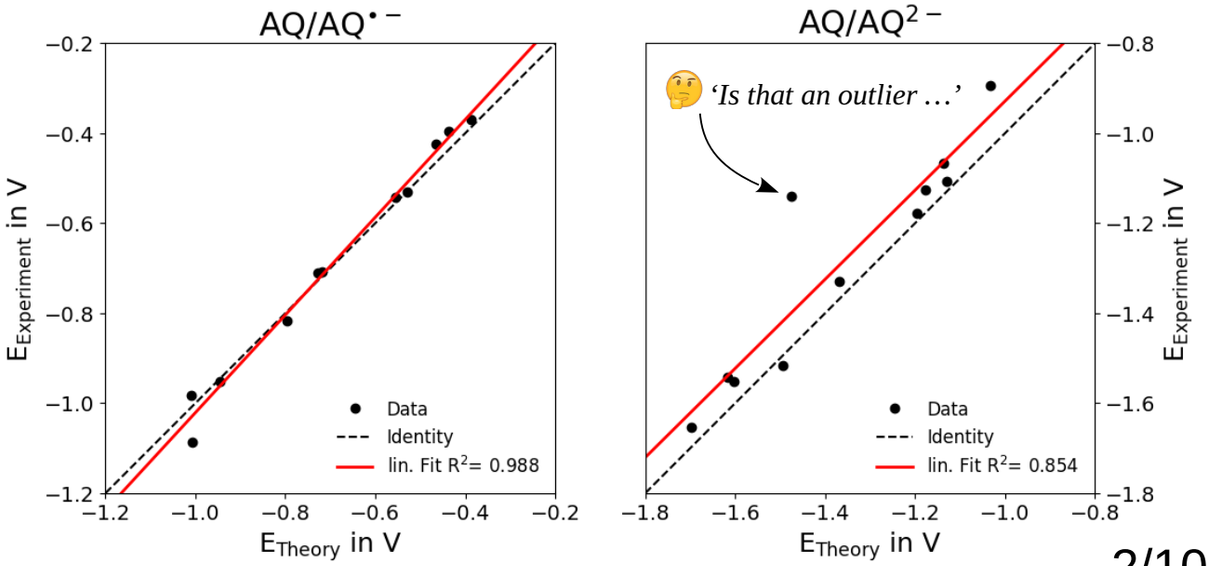
  Use `ax1.set_aspect('equal')` to make the plot appear as a square.

- Include the ideal diagonal line, *i.e.* $y = x$.

  To draw the ideal line over the entire range define new x-points via `np.linspace()`

- A good strategy to prepare correlation plots is:
  - Draw the ideal diagonal over the entire range.
  - Draw the regression over the entire range.
  - Plot the original data only as points.

```
reg_redox_1 = linregress(redox1_theo, redox1_exp)

ax1.set_xlim(-1.2,-0.2)
ax1.set_ylim(-1.2,-0.2)

ax1.set_aspect('equal')

x1=np.linspace(-1.2, -0.2, 2)

ax1.plot(redox1_theo, redox1_exp, marker = 'o',
         color = 'k', linestyle = ' ',
         label = 'Data')

ax1.plot([-1.2, -0.2], [-1.2, -0.2], color = 'k',
         linestyle = '--', label = "Identity")

ax1.plot(x1, reg_redox_1[0]*x1+reg_redox_1[1],
         color = 'r', linewidth=2,
         label=f"lin. Fit R$^2$={reg_redox_1[2]**2: 6.3f}")
```



Finding Outliers – z-scores

- A simple framework to detect outliers are z-scores. For a normal data set, they are computed as:

$$z_i = \frac{y_i - y_{\mathrm{av}}}{\sigma_y}$$

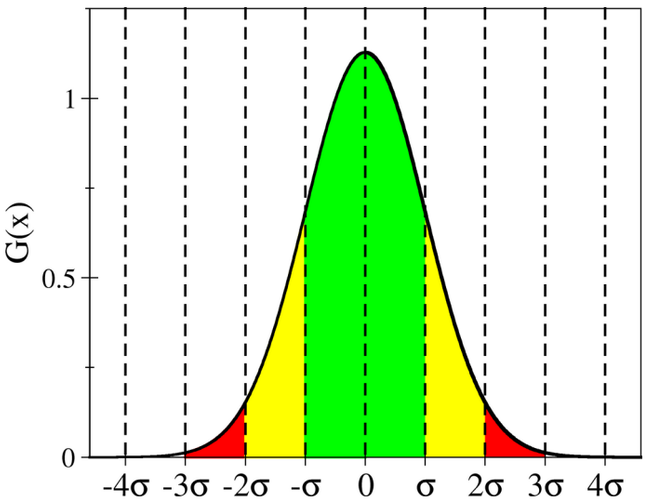
  If the $|z|$-value of data point i is above a value of 2 or 3, it is considered as an outlier.

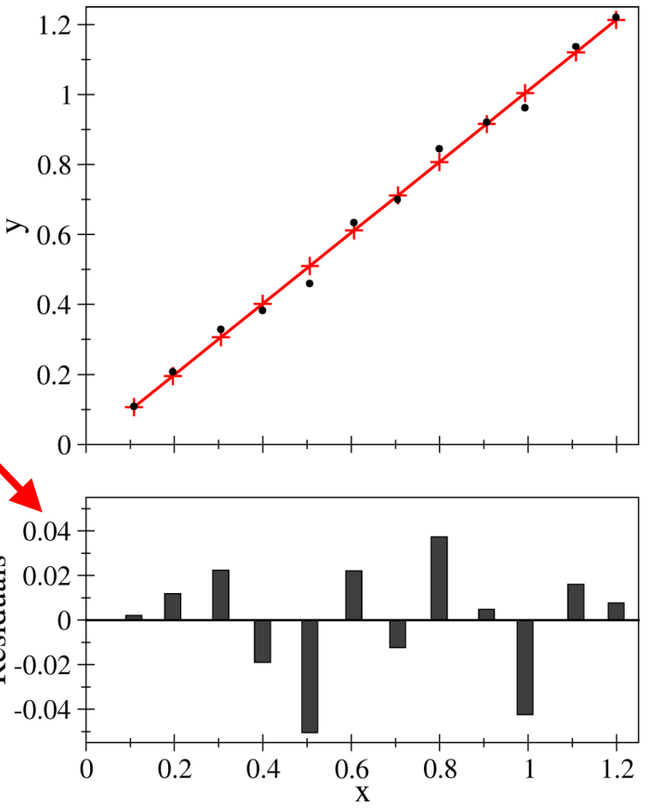
- But: In case of a linear regression, the z-score has to be determined for the residuals r_i and not for the data points y_i along the y-axis!!

$$r_i = y_i - \bar{y}_i$$

$$z_i = \frac{r_i - r_{\mathrm{av}}}{\sigma_r}$$

- Careful: Most online tutorials and AI systems will give you the WRONG advice in the is context, namely to calculate the z-score of the y-axis only.





if (regression):
z-score goes here

z-scores in Python (*low/intermediate difficulty*)

- To setup the outlier testing the residuals r_i have to be computed first:

$$r_i = y_i - \bar{y}_i$$

- Calculating the z-scores manually via `np.mean()` and `np.std()` is fairly easy.
- Alternatively, `scipy.stats` provides a `zscore()` module giving the same results.
- A clever way to look at the data is to print the residuals and z-scores line by line using a loop.
- Here we can make use of conditional printing. If the z-score is larger than a threshold we print “OUTLIER”, else we print nothing (*i.e.* “”)

  (Note: the else statement is mandatory.)

```
y_pred = reg_redox_2[0] * redox2_theo + reg_redox_2[1]
residuals = redox2_exp - y_pred

z_scores = (residuals - np.mean(residuals)) / np.std(residuals)
```

```
from scipy.stats import zscore

y_pred = reg_redox_2[0] * redox2_theo + reg_redox_2[1]
residuals = redox2_exp - y_pred

z_scores_scipy = zscore(residuals)
```

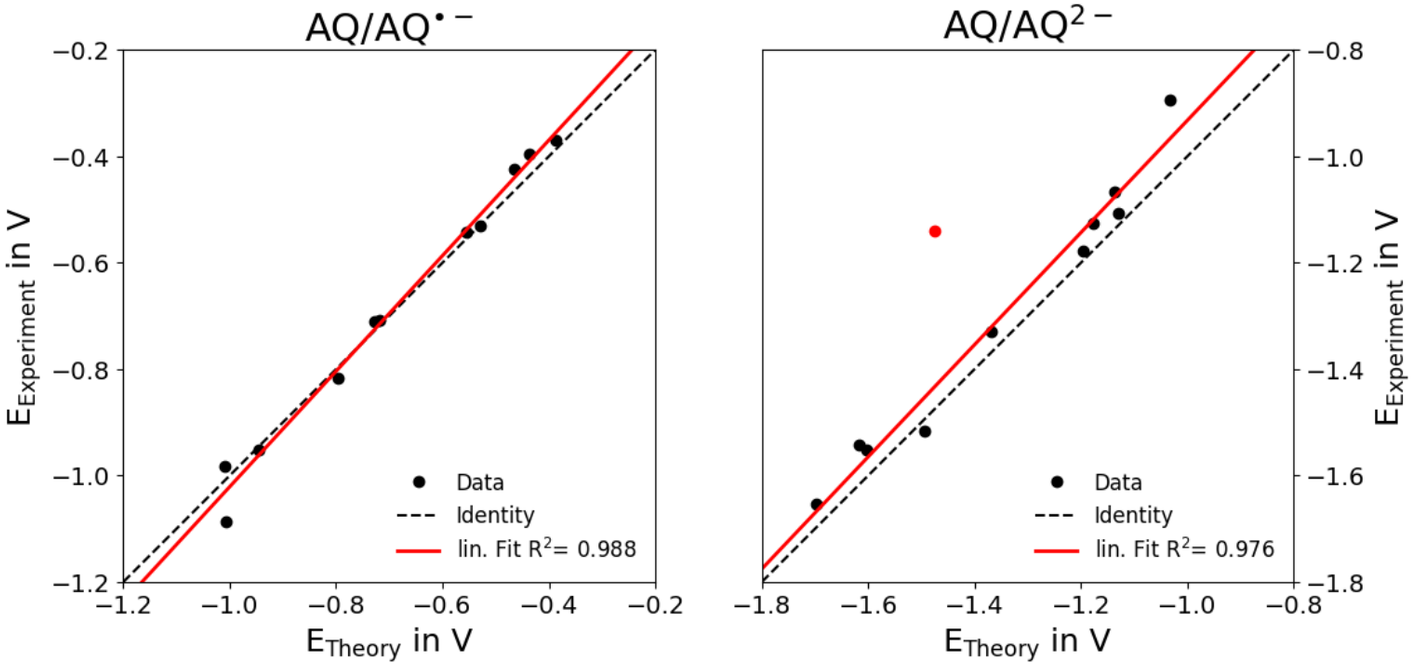
```
print("-------------------------------------------------")
print(f'idx  Compound        ΔE          z-score')
print("-------------------------------------------------")

for i in np.arange(0,len(redox2_exp)):
  print(f'{i:3d}  {ID[i]:10s}
          {residuals[i]:8.4f} V
          {z_scores_scipy[i]:8.4f}
          {"OUTLIER" if abs(z_scores_scipy[i]) > 2 else ""}')

print("-------------------------------------------------")
```

Excluding the Outlier and Re-plot (*low difficulty*)

- From the z-scores we now know that data point 2 (index 1) is the outlier.
- The easiest way is to store the data in a new array and exclude that data point using `np.delete(array, index)`.
- Calculate the regression as before using the new data set.
- We can then create the figures the way we did before.
- To highlight the outlier we plot the data set showing only a single data point, *e.g.* in this case:

```
ax2.plot(redox2_theo[1], redox2_exp[1],
         marker = 'o', color = 'r',
         linestyle = ' ')
```

```
redox2_theo_filtered = np.delete(redox2_theo,1)
redox2_exp_filtered  = np.delete(redox2_exp,1)

reg_redox_2_filtered = linregress(redox2_theo_filtered,
                                  redox2_exp_filtered)
```



Exercise B.3 - Arrhenius Curve Fitting (*low difficulty*)

In this exercise we have a look at diffusion data (either from experiment or theoretical calculations) and want to obtain the activation energy E_A. Similar to many dynamical properties in chemistry (such as rate constants, diffusion, etc.) diffusion follows an Arrhenian temperature dependency

$$y(T) = y_0 e^{-\frac{E_A}{RT}}$$

with y_0 being the pre-exponential factor, R and T are the molar gas constant (8.3145 J mol^{-1} K^{-1}) and temperature, respectively.

One potential option to obtain E_A is a non-linear exponential fit, but this is known to be less reliable than its linear counterpart! Consequently, Mr. Svante Arrhenius used a linearization of the equation, which in case of the diffusion coefficient D looks like this:

$$ln(D) = ln(D_0) - \frac{E_A}{R}\frac{1}{T} \longrightarrow y = a \cdot x + b \quad \begin{cases} y = ln(D) & b = ln(D_0) \\ x = \frac{1}{T} & a = -\frac{E_A}{R} \end{cases}$$

From this we can directly access E_A *via*: $E_A = -a \cdot R$

Loading the Files and Having a First Look at the Data (*zero difficulty*)

This time the files are in csv-format (comma-separated values), *i.e.* the different data columns are separated by comma symbols.

Luckily, we can again use the command `np.loadtext()`, but we have to indicate the comma by adding `delimiter = ','`.

Most programs such as Excel, Origin and scientific software can export data sets in this format. If you want to use python in your research, this is most likely the most common file format to input you data sets.

Let's quickly read and plot our data sets.

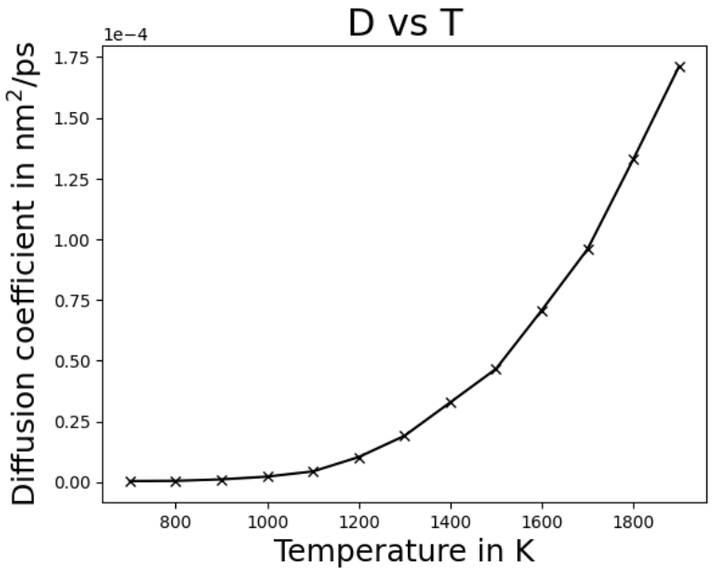
Since our y-values are really small, we want to change the style of the y-tick labels using `plt.ticklabel_format()` from the module `matplotlib.ticker`.

Rinse & repeat for the other two data sets. 🤓

```
from google.colab import files

uploaded = files.upload()
```

```
import numpy as np
import matplotlib.pyplot as plt

dataset_a = np.loadtxt('D_vs_T_v1.csv', delimiter=',')
```

```
import matplotlib.ticker as ticker

plt.plot(dataset_a[:, 0], dataset_a[:, 1],
         color='k',  marker = 'x')

plt.ticklabel_format(style='sci', axis='y',
scilimits=(0, 0))
```



Data Conversion Made Easy … with a Twist (*low difficulty*)

Converting the data so we get an Arrhenius plot is straightforward.

ProTip: Choosing effective array names can make your life so much easier.

But wait:

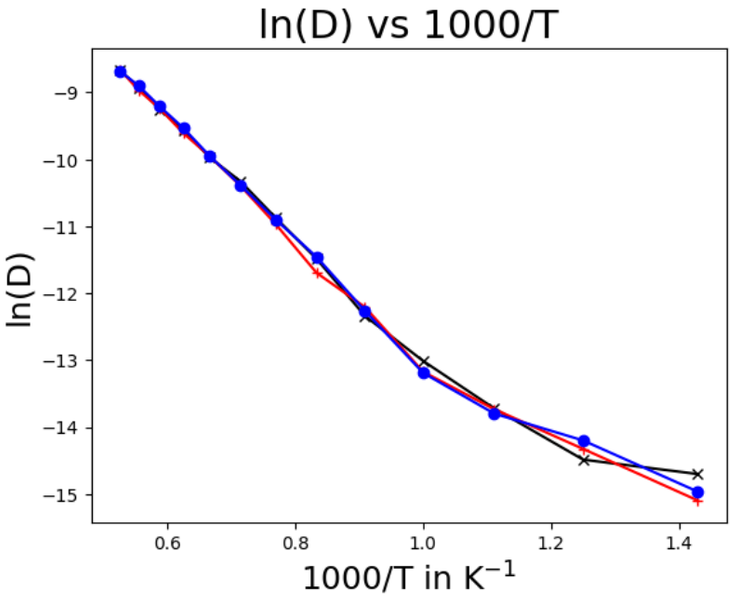
Does `log()` compute log(x) or ln(x)?

```
# Euler's number e

euler=2.7182818284590452

print(np.log(10),   np.log(euler),
      np.log10(10), np.log10(euler) )
```

Careful: In NumPy, log(x) calculates the natural logarithm ln(x), whereas log10(x) is the decadic log.

It is very common to use 1000/T to obtain managable numbers . Now, the value of 1 on the x-axis corresponds to 1000K (only for the plot).

```
inv_Ta = 1.0/dataset_a[:,0]
ln_Da  = np.log(dataset_a[:,1])

inv_Tb = 1.0/dataset_b[:,0]
ln_Db  = np.log(dataset_b[:,1])

inv_Tc = 1.0/dataset_c[:,0]
ln_Dc  = np.log(dataset_c[:,1])

plt.plot(1000 * inv_Ta, ln_Da, color='k', marker = 'x')
plt.plot(1000 * inv_Tb, ln_Db, color='r', marker = '+')
plt.plot(1000 * inv_Tc, ln_Dc, color='b', marker = 'o')

plt.xlabel('1000/T in K$^{-1}$', fontsize=18)
plt.ylabel('ln(D)', fontsize=18)
plt.title('ln(D) vs 1000/T', fontsize=22)
```



Selecting a Data Range in Python (*low difficulty*)

In all three cases the plot is only linear above 1000K (to left of 1.0 in the plot).

When carrying out the linear regression, only the data points above 1000K should be used!

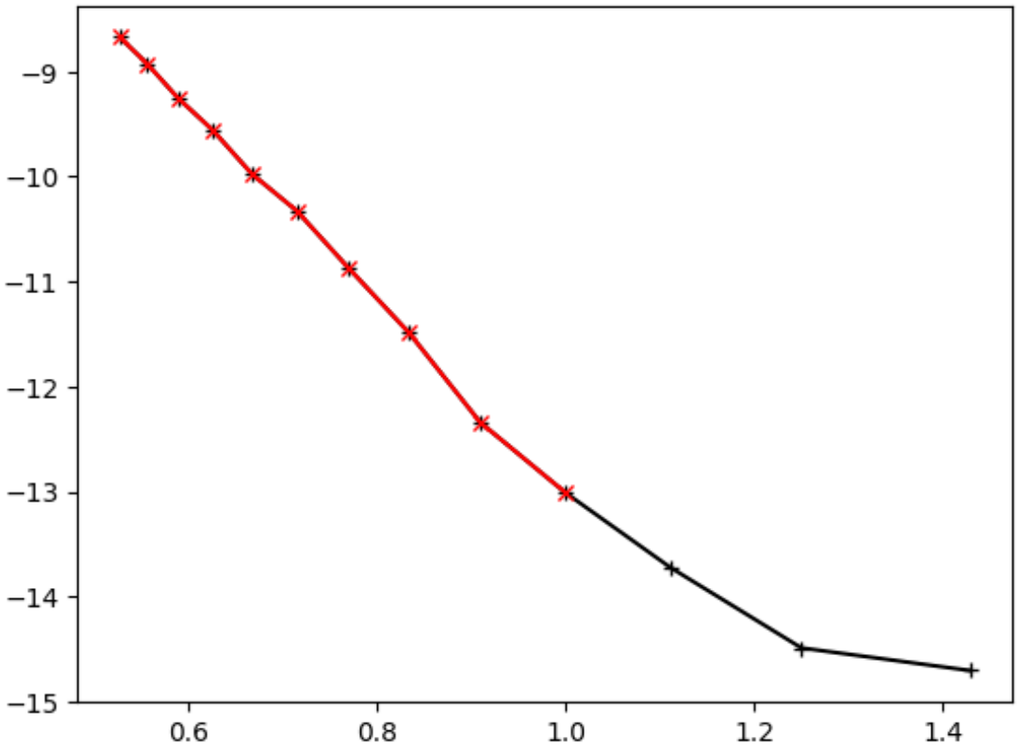
Let's test this only for data set 1 first, by plotting only the last data points:

Careful:

- Python counts from zero, so our 13 data points are indexed from 0 to 12.
- The last entry is not included, so `invT1[3:12]` excludes data point 12.
- To access the correct data range, we need `invT1[3:13]` and `ln_D[3:13]`.

First, let's make and overlay of the full and selected data range for data set 1.

```
# Full data range
plt.plot(1000 * inv_Ta, ln_Da,
         color='k', marker = '+')

# Selected data range
plt.plot(1000 * Inv_Ta[3:13], ln_Da[3:13],
         color='r', marker = 'x')
```



Linear Regression Using the Selected Data Range (*low difficulty*)

We can directly use selection `[3:13]` in the `linregress()` module of SciPy.

Make sure to use the same range for both the x- and y-input.

Using the slope from the linear regression parameters we can easily calculate the activation energy *via*:

$$E_A = -a \cdot R$$

$$R = 8.3145 \frac{\mathrm{J}}{\mathrm{mol\ K}}$$

In this case most literature values are given in electron volts (eV). The respective conversion factor is:

$$1 \frac{\mathrm{kJ}}{\mathrm{mol}} = 0.01036427 \mathrm{eV}$$

```
from scipy.stats import linregress

reg_a = linregress(inv_Ta[3:13], ln_Da[3:13])
reg_b = linregress(inv_Tb[3:13], ln_Db[3:13])
reg_c = linregress(inv_Tc[3:13], ln_Dc[3:13])

act_energy_a = -reg_a[0] * 8.31415 / 1000.0 * 0.01036427
act_energy_b = -reg_b[0] * 8.31415 / 1000.0 * 0.01036427
act_energy_c = -reg_c[0] * 8.31415 / 1000.0 * 0.01036427
```

	Slope	Intercept	R^2	act. Energy
A:	-9300.12K	-3.7566	0.9986	-0.801 eV
B:	-9490.08K	-3.6647	0.9987	-0.818 eV
C:	-9503.99K	-3.6152	0.9991	-0.819 eV

The Final Plots

Here, we show the regression as overlaid on the original data.

Again, we make use of the selection `[3:13]`.

```
ax1.plot(1000 * inv_Ta, ln_Da, color='k', marker = '+',
         markersize = 15, label = 'Data Set A')

ax1.plot(1000 * inv_Ta[3:13],
         reg_a[0] * inv_Ta[3:13] + reg_a[1], color='r',
         marker = 'o', label = 'lin. Regression')
```

This is another perfect example to exercise using `plt.subplot()` showing three individual plots next to each other.