Linear regression

Contents

Mathematics Methods 1 Numerical Methods Data Science and Machine Learning for Geoscientists Excel and Statistics

Theory

Linearity refers to a linear relationship between two or more variables. Linear regression aims to predict the dependent variable value (y) based on a given independent variable (x). Therefore, linear regression finds out a linear relationship between x and y.

With noisy data or multiple different measurements of y at a given value of x, we may not be able to fit a function/curve that goes through all points exactly. Therefore, in linear regresssion the aim is to find a function that best approximates the data but does not necessarily go through all the points.

Simple linear regression

Plotting the independent variable x on the x-axis and dependent variable y on the y-axis, linear regression gives us a straight line with equation:

$$y = b_0 + b_1 x,$$

where b_0 is the intercept and b_1 is the slope of the line. The x and y variables remain the same as the data points cannot change, however, the intercept (b_0) and slope (b_1) can be modified to obtain the most optimal value for the intercept and the slope. The linear regression algorithm fits multiple lines on the data points and returns the line that results in the smallest error. This may be achieved by minimising the sum of the squares of the differences to the data, known as a least squares approximation.

Figure 1: Plot of scatter points in 2D space (blue) and line that results in the least error (red).

Multiple linear regression

This can be extended to multiple linear regression where there are more than two variables. In this scenario, the dependent variable is dependent upon several independent variables $x = (x_1, \dots, x_n)$ where n is the number of variables. You can assume a linear relationship between x and y with the regression equation:

$$y = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_3 + \dots b_n x_n + \epsilon,$$

where $b_0, b_1, \dots, b_n$ are the regression coefficients and ϵ is the random error.

Root-mean-square error

There are many methods to evaluate the performance of the linear regression algorithm. Two commonly used methods are the root-mean-square error (RMSE) and the coefficient of determination (R^2 score).

RMSE is the square root of the sum of all errors squared divided by the number of values. The equation for the RMSE is:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y_i} - y_i)^2},$$

where $\hat{y_1}, \hat{y_2}, \ldots, \hat{y_n}$ are the predicted values, $y_1, y_2, \ldots, y_n$ are the observed values and n is the number of observations.

Coefficient of determination

The coefficient of determinaion is a statistical measure of how close the data are to the linear regression line.

$$R^2 = \frac{\text{Explained variation}}{\text{Total variation}}.$$

R^2 is therefore always between 0 and 100%. The higher the R^2, the better the model fits the data.

R^2 is defined as follows:

$$R^2 = 1 - \frac{SS_r}{SS_t},$$

$$SS_r = \sum_{i=1}^{n}(y_i - \hat{y_i})^2,$$

$$SS_t = \sum_{i=1}^{n}(y_i - \bar{y_i})^2.$$

SS_r is the sum of squared regression and represents the variation explained by the linear regression model.

SS_t is the sum of squared yotal and represents the total variation in the data.

$y_1, y_2, \ldots, y_n$ are the observed values, $\hat{y_1}, \hat{y_2}, \ldots, \hat{y_n}$ are the predicted values of y, and $\bar{y_1}, \bar{y_2}, \ldots, \bar{y_n}$ are the mean values of y.

Based on the above equation the R^2 score usually ranges from 0 to 1, but can be negative if the model is completely wrong.

Least squares error calculation

Least squares fitting minimises the sum of the squares of the differences between the data provided and the polynomial approximation. In other words it minimises the folowing expression:

$$E = \sum_{i=0}^{N}(P(x_i) - y_i)^2,$$

where E is the squared error, $P(x_i)$ is the value of the polynomial function that has been fit to the data evaluated at point x_i, and y_i is the i^{th} data value.

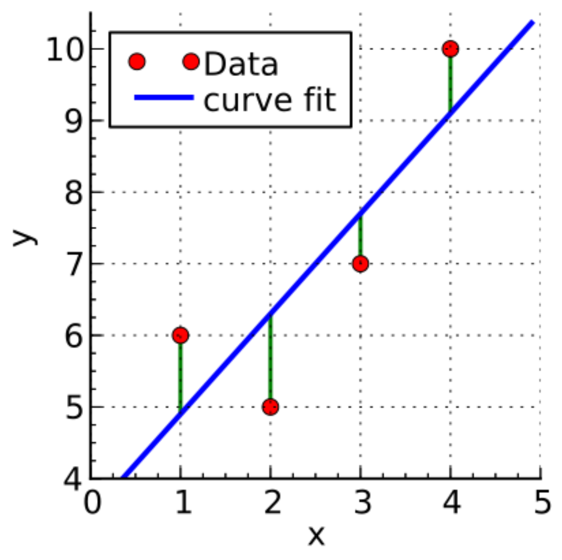


Figure 2: A plot of the data points (red), the least squares line of best fit (blue), and the residuals (green).

In this calulation we are computing the sum of the squares of the distances indicated in green in Figure 1.

Implementation of linear regression in Python

Simple example - submarine landslide size in the North Atlantic

```python
# Some imports needed for linear regression in python

%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import scipy.interpolate as si
import scipy.stats as ss

# Some default font sizes for plots
plt.rcParams['font.size'] = 12
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.sans-serif'] = ['Arial', 'Dejavu Sans']
```

In this example we are attempting to fit a linear best fit line to the data `length_width.dat` in log-log space. This file contains the lengths and widths of submarine landslides in the North Atlantic basin from Fig. 7 in Huhnerbach & Masson (2004).

Firstly, we will use `numpy.polyfit` in order to carry out the least squares error calculation to fit a linear polynomial. Next, we will use `scipy.stats.linregress` to perform linear regression using a SciPy implementation of linear regression. Then, we will compare the slope and the intercept (the two coefficients in the linear polynomial) between the two approaches.

The coefficient of determination is also determined by default from the linear regression calculation. To check these values agree we will also calculate the R^2 value using the `numpy.polyfit` data.

Let's define a function to evaluate squared error:

```python
# Function to evaluate the squared error
def sqr_error(p, xi, yi):

    """"Function to evaluate the sum of square of errors"""

    # Compute the square of the differences
    diff2 = (p(xi)-yi)**2

    # Return their sum
    return diff2.sum()
```

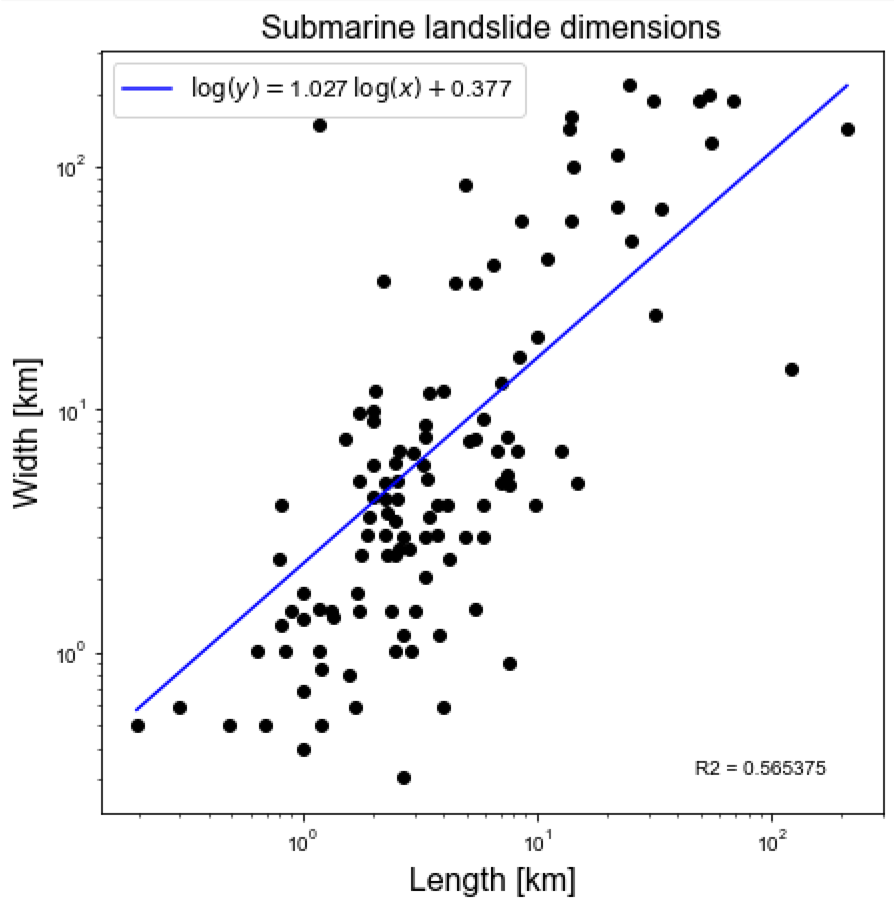
Open a file and store data in arrays:

```python
file = open("length_width.dat", 'r')

xi = []
yi = []

for line in file:
    xi.append(float(line.split()[0]))
    yi.append(float(line.split()[1]))

xi = np.array(xi)
yi = np.array(yi)
```

Perform linear regression and plot the results:

```python
# Set up figure
fig, ax1 = plt.subplots(1, 1, figsize=(7, 7))

# Plot the raw data
ax1.loglog(xi, yi, 'ko')

# Fit a linear line to the log of the data using numpy.polyfit
logxi = np.log(xi)
logyi = np.log(yi)
poly_coeffs = np.polyfit(logxi, logyi, 1)

# Construct the corresponding polynomial function from these coefficients
p1 = np.poly1d(poly_coeffs)
# print the polynomial coefficients to compare with regression
print('Lagrange polynomial coefficients = {}'.format(poly_coeffs))

#Calculate and print an R-squared value for this fit using the mathematical
# definition from https://en.wikipedia.org/wiki/Coefficient_of_determination
SS_res = sqr_error(p1, logxi, logyi)
SS_tot = np.sum((np.mean(logyi) - logyi)**2)
r2 = 1. - SS_res/SS_tot
print('R^2 value calculated from Lagrange polynomial fit to the data in log-log space =
{}\n'.format(r2))

# Only need two points to plot the regression
x = np.linspace(min(xi), max(xi), 2)
ax1.loglog(x, p1(x), 'b', label='$\log(y) = $%.3f$\,\log(x) + $%.3f' %
           (poly_coeffs[0], poly_coeffs[1]))
ax1.legend(loc='best', fontsize=12)

# Check values computed above against scipy's linear regression
slope, intercept, r_value, p_value, std_err = ss.linregress(logxi, logyi)
print('Linear regression: slope, intercept, r_value = {0:.8f}, {1:.8f}, {2:.8f}'\
      .format(slope, intercept, r_value))
print('R^2 = {:.8f}'.format(r_value**2))

ax1.set_title('Submarine landslide dimensions', fontsize=16)
ax1.set_xlabel('Length [km]', fontsize=16)
ax1.set_ylabel('Width [km]', fontsize=16)

ax1.text(0.76, 0.05, 'R2 = %.6f' % r2, transform=ax1.transAxes)

plt.show()
```

```
Lagrange polynomial coefficients = [1.0266104  0.37698383]
R^2 value calculated from Lagrange polynomial fit to the data in log-log space =
0.5653751967433511

Linear regression: slope, intercept, r_value = 1.02661040, 0.37698383, 0.75191435
R^2 = 0.56537520
```



Polynomial curve fitting

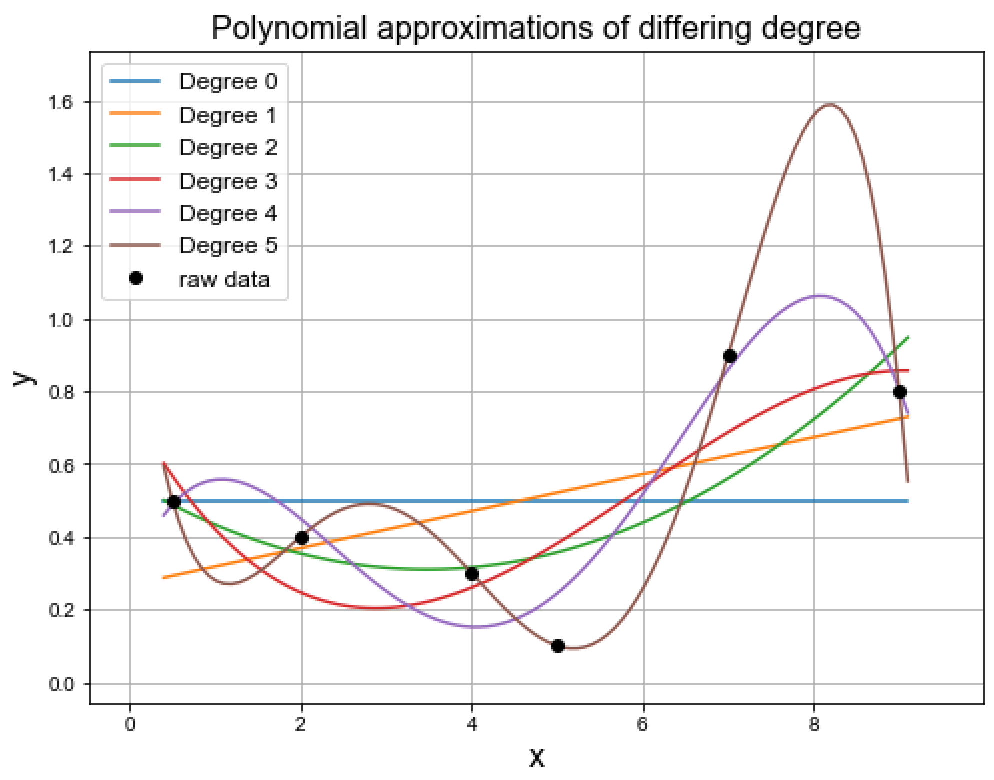
Curve fitting is popular to use for datasets containing noise. To fit these curves of varying polynomial degree we can again use the least squares error calculation.

Using `numpy.polyfit` we can fit curves of varying polynomial degree to the data points. This is demonstrated below.

```python
# Data points
xi = np.array([0.5, 2.0, 4.0, 5.0, 7.0, 9.0])
yi = np.array([0.5, 0.4, 0.3, 0.1, 0.9, 0.8])

# Let's set up some space to store all the polynomial coefficients
# there are some redundancies here, and we have assumed we will only
# consider polynomials up to degree N
N = 6
poly_coeffs = np.zeros((N, N))

for i in range(N):
    poly_coeffs[i, :(i+1)] = np.polyfit(xi, yi, i)

print('poly_coeffs = \n{}'.format(poly_coeffs))
```

```
poly_coeffs =
[[ 0.5         0.          0.          0.          0.          0.        ]
 [ 0.0508044   0.26714649  0.          0.          0.          0.        ]
 [ 0.02013603 -0.13983999  0.55279339  0.          0.          0.        ]
 [-0.00552147  0.09889271 -0.43193108  0.75909819  0.          0.        ]
 [-0.00420655  0.07403681 -0.38492428  0.59251888  0.27906056  0.        ]
 [-0.00301599  0.06536037 -0.49614427  1.59623195 -2.08266478  1.20030166]]
```

```python
fig = plt.figure(figsize=(8, 6))
ax1 = fig.add_subplot(111)
ax1.margins(0.1)

x = np.linspace(0.4, 9.1, 100)

for i in range(N):
    p = np.poly1d(poly_coeffs[i, :(i+1)])
    ax1.plot(x, p(x), label='Degree %i' % i)

ax1.plot(xi, yi, 'o', color="black", label="raw data")

plt.legend(loc='best', fontsize = 12)
plt.title('Polynomial approximations of differing degree', fontsize=16)
plt.grid(True)
plt.xlabel("x", fontsize=16)
plt.ylabel("y", fontsize=16)
plt.show()
```



Using the above function that evaluates the squared error, we can evaluate the error for each of the polynomials calculated above.

```python
for i in range(N):
    p = np.poly1d(poly_coeffs[i, :(i+1)])
    print('Square of the difference between the data and the '
          'polynomial of degree {0:1d} = {1:.8e}.'.format(i, sqr_error(p, xi, yi)))
```

```
Square of the difference between the data and the polynomial of degree 0 = 4.60000000e-
01.
Square of the difference between the data and the polynomial of degree 1 = 3.32988992e-
01.
Square of the difference between the data and the polynomial of degree 2 = 1.99478242e-
01.
Square of the difference between the data and the polynomial of degree 3 = 1.57303437e-
01.
Square of the difference between the data and the polynomial of degree 4 = 4.69232378e-
02.
Square of the difference between the data and the polynomial of degree 5 = 1.75525473e-
26.
```

As can be seen above the error drops as we approximate the data with higher degree polynomials.

For some inspiration on multiple linear regression, you can look at “A beginner’s guide to linear regression in Python with Scikit-Learn” and “Linear regression case study”.

References

- Information in this notebook is compiled based on ACSE-3 (Numerical Methods), Lecture 1: Interpolation and Curve Fitting
- V. Huhnerbach, D.G. Masson, Landslides in the North Atlantic and its adjacent seas: an analysis of their morphology, setting and behaviour, Marine Geology 213 (2004) 343 – 362.
- Real Python - “Linear regression in Python”
- Towards Data Science - “A beginner’s guide to linear regression in Python with Scikit-Learn” and “What does RMSE really mean?”
- Acadgild - “Linear regression case study”

Introduction to statistics for Geoscientists

Contents

Excel and Statistics

Statistics is a study concerning the collection, organisation, analysis, interpretation and presentation of data. Statistics help with describing and analysing non-deterministic phenomena, i.e. ones that cannot be precisely predicted.

Population is a group/set of items/events of interest in the study.

A *sample* is a subset of population that we measure. Samples must be unbiased for reliable statistical analysis. Signor and Lipps (1982) for example showed that biased sampling in fossils can lead to wrong interpretations for extinctions due to discontinous fossil record in time and ranges of species in sediments.

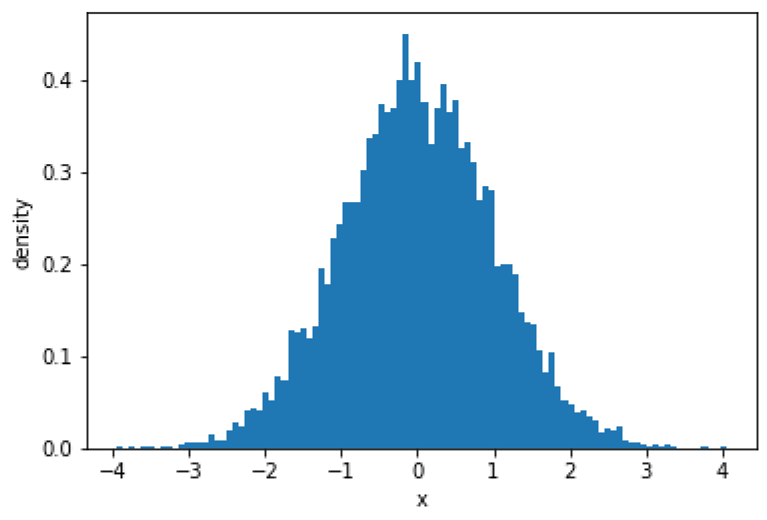
Distribution is the shape of a histogram. The most commonly known distribution is a normal distribution ('bell curve'). A normal (Gaussian) distribution is very powerful as it approximates many real-world distributions. Mathematically, normal distribution is described as:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where σ is the standard deviation and μ is the mean. It is often assumed that distributions will be normal in statistical methods. As an example, we can plot one using scipy.stats.

```python
from scipy import stats
import matplotlib.pyplot as plt
import numpy as np
```

```python
# Generate data with normal distribution
x = stats.norm.rvs(size=10000)

# Plot a histogram of the data
plt.hist(x, 100, density=1)

plt.xlabel("x")
plt.ylabel("density")

plt.show()
```



Averages

Average is a measure of central tendency, in colloquial language, it is a single number that represents a given population.

Mean (arithmetic mean) is calculated by summing up all values and dividing by the population size.

Median is the middle value for an ordered list of values.

Mode is most common value.

Median and mean are commonly used as averages. Median may be a better representation for skewed distributions as it reduces the effect of outliers. Mode isn't really used that often.

We can calculate this with numpy and scipy.stats in-built functions:

```
# Generate data
x = np.array([1, 1, 1, 2, 5, 3, 4, 9, 7, 7, 3])

print("Mean = %.2f." % np.mean(x))
print("Median = %.2f." % np.median(x))
mode, count = stats.mode(x)
print("Mode = %.2f." % mode)
```

```
Mean = 3.91.
Median = 3.00.
Mode = 1.00.
```

Measures of spread

Knowing populations by their means and medians can be misleading. Therefore, we can use spread with numpy function

```
np.ptp()
```

to test samples.

```
sample1 = np.array([-0.2, -0.1, -0.1, 0.0, 0.0, 0.0 , 0.1 , 0.1 , 0.2])
sample2 = np.array([-2000, -1000, 0, 1000, 2000])

print ("Sample 1 mean = %.2f, median = %.2f, spread = %.2f."
       % (np.mean(sample1), np.median(sample1), np.ptp(sample1)))

print ("Sample 2 mean = %.2f, median = %.2f, spread = %.2f."
       % (np.mean(sample2), np.median(sample2),np.ptp(sample2)))
```

```
Sample 1 mean = 0.00, median = 0.00, spread = 0.40.
Sample 2 mean = 0.00, median = 0.00, spread = 4000.00.
```

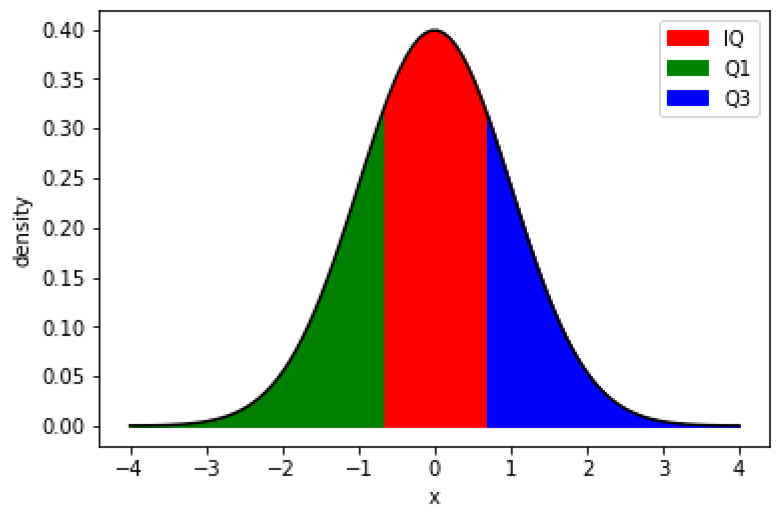
Percentiles

The nth percentile says that n% of the population are lower than given value.

The four percentiles with their own names are:

- 25th percentile - first quartile (Q1)
- 50th percentile - second quartile (Q2)
- 75th percentile - third quartile (Q3)
- Interquartile - range between Q1 and Q2.

```python
# Generate normal distribution
mu, std = 0, 1
xmin, xmax = [-4, 4]
x = np.linspace(xmin, xmax, 100)
p = stats.norm.pdf(x, mu, std)

plt.plot(x, p, 'k')

# Fill background
plt.fill_between(x, 0, p, interpolate=True,
                color="red", label="IQ")

# Fill between values that are less than 25th percentile
plt.fill_between(x, 0, p, interpolate=True,
                 where=(x<=stats.norm.ppf(0.25)),
                color="green", label="Q1")

# Fill between values that are more than 75th percentile
plt.fill_between(x, 0, p, interpolate=True,
                 where=(x>=stats.norm.ppf(0.75)),
                color="blue", label="Q3")
plt.legend(loc="best")

plt.xlabel("x")
plt.ylabel("density")

plt.show()
```



Variance and standard deviation

Variance and *standard deviation* are measures of spread. The sample variance is described as follows:

$$v = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2,$$

where N is the number of measurements, x_i is the individual measurement and $\bar{x}$ is the sample mean. For population variance we can use $\frac{1}{N}$, instead of $\frac{1}{N-1}$. This corrects for bias in the estimation (Bessel's correction).

The sample standard deviations is:

$$\sigma = \sqrt{v},$$

which is useful as it is in units of the measured samples.

Bivariant statistics

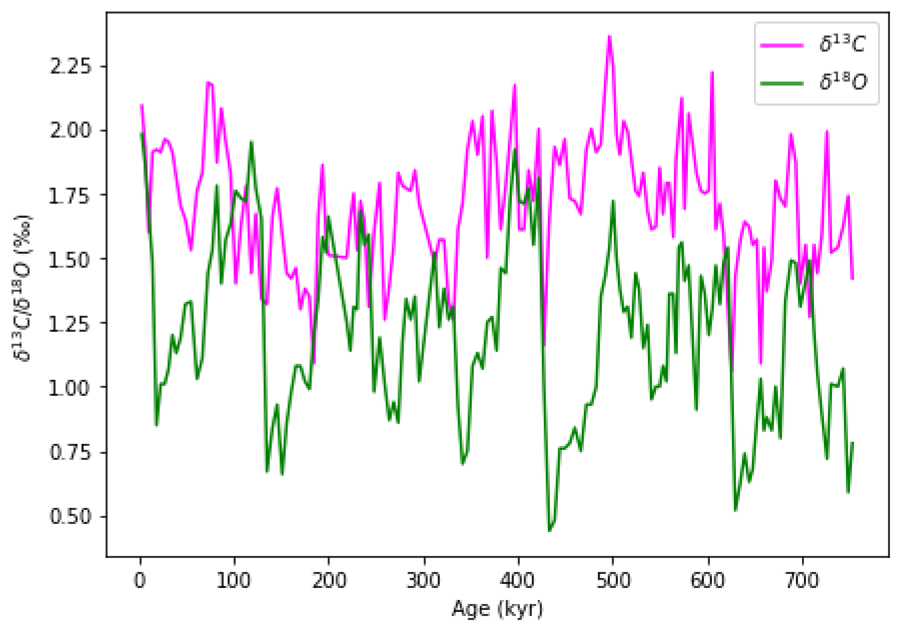
Bivariant statistics is when we analyse two variables to look for relationships.

For example, let's load isotope data from ODP 806 site in Java, downloaded from NOAA for a G.sacculifer foraminifera species.

```python
import pandas as pd

odp_data = pd.read_csv('data/odp_806_data.csv')

odp_data.head()
```

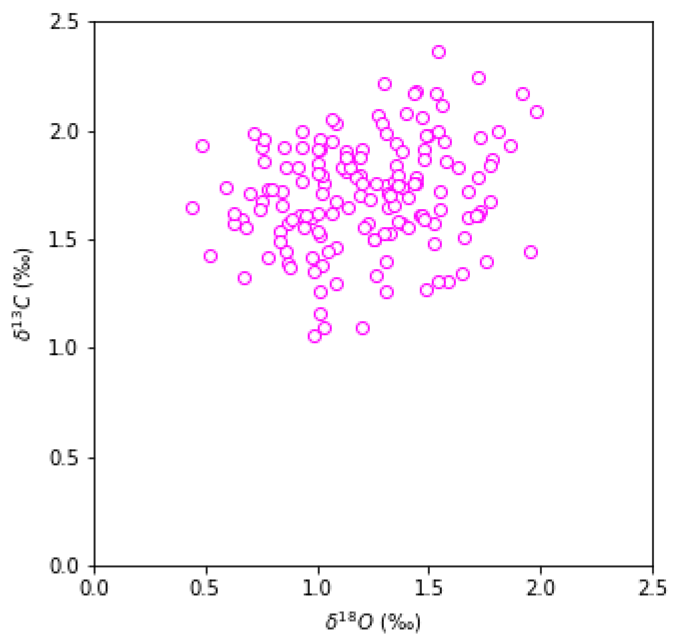
	depth_1	depth_2	age	oxygen	carbon
0	0.1	0.09	3.42	1.98	2.09
1	0.2	0.19	6.94	1.86	1.93
2	0.3	0.28	10.61	1.68	1.60
3	0.4	0.38	14.63	1.48	1.91
4	0.5	0.47	18.90	0.85	1.92

```
plt.figure(figsize=(7,5))

plt.plot(odp_data.age, odp_data.carbon, label="$\delta^{13}C$",
        c="magenta")
plt.plot(odp_data.age, odp_data.oxygen, label="$\delta^{18}O$",
        c="green")

plt.ylabel("$\delta^{13}C$/$\delta^{18}O$ (‰)")
plt.xlabel("Age (kyr)")

plt.legend(loc="best")
plt.show()
```



We can see that throughout past 800,000 years the oxygen isotope data and carbon isotope data is correlated. Let's plot them against each other:

```
plt.figure(figsize=(5,5))

plt.scatter(odp_data.oxygen, odp_data.carbon, label="$\delta^{13}C$",
           c="white", edgecolor="magenta")

plt.ylabel("$\delta^{13}C$ (‰)")
plt.xlabel("$\delta^{18}O$ (‰)")

plt.xlim(0,2.5)
plt.ylim(0,2.5)

plt.show()
```



The data is quite scattered but the general trend is that with increased carbon isotope content we observe increased oxygen content. Linear regression can help to mathematically find, whether these to are correlated at all.

Linear regression

Linear regression can help us determine whether there is relationship between two parameters. Most commonly used technique to fit a line into data is least squares regression that minimises the sum of squares of vertical errors. In Python, we can use:

```
scipy.stats.linregress(x, y)
```

that returns slope m and intercept c for line equation $y = mx + c$. It also returns correlation coefficient *rvalue*, two-sided *p-value* for hypothesis if slope is zero and *stderrr*, the standard error of estimated gradient. The meaning *p-value* is explained more in section on Inferential Statistics.

Correlation coefficients

The correlation coefficient can be used to determine how well two samples are correlated.

r-value	Interpretation
-1	Perfect negative correlation
-0.7	Strong negative correlation
-0.5	Moderate negative correlation
-0.3	Weak negative correlation
0	No correlation
0.3	Weak correlation
0.5	Moderate correlation
0.7	Strong correlation
1	Perfect correlation

Scipy has a range of correlation coefficients. Most common is **Pearson r-value**:

```
scipy.stats.pearsonr(x, y)
```

This coefficient is calculated according to:

$$r = \frac{\sum (x - m_x)(y - m_y)}{\sqrt{\sum (x - m_x)^2 \sum (y - m_y)^2}},$$

where m_x and m_y are mean values for x and y vectors. You can read more about it here.

Pearson correlation coefficient assumes that two datasets are **normally distributed**. That is not always the case, therefore, **Spearman's rank correlation coefficient** can be used for *non-parametric* correlation (assuming no specific distribution). You can call it using:

```
scipy.stats.spearmanr(x, y)
```

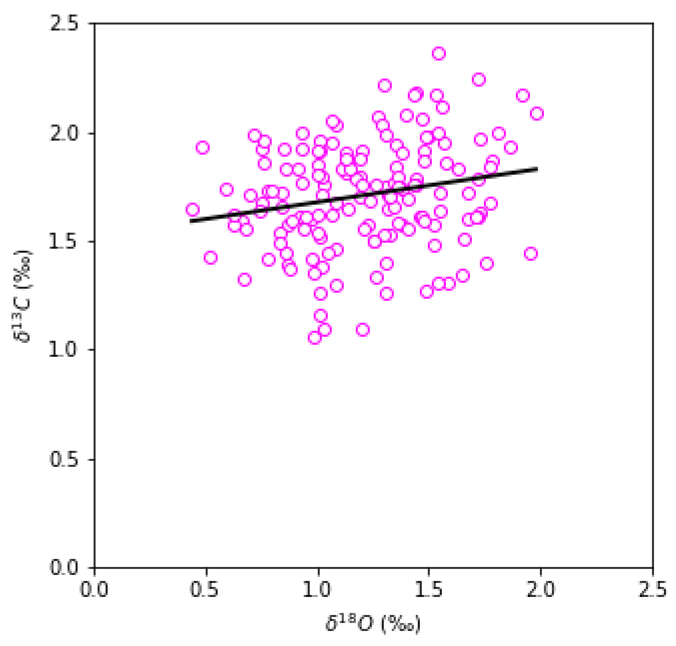
Exercise

Let's fit a line into the oxygen and carbon isotope data and find how well these two correlate:

```python
# Create x values for line plotting
x = np.linspace(np.min(odp_data.oxygen), np.max(odp_data.oxygen), 100)

# Fit a line
slope, intercept, r_value, p_value, stderr = stats.linregress(odp_data.oxygen,
odp_data.carbon)

print("Slope = %.2f." % slope)
print("Intercept = %.2f." % intercept)
print("Correlation coefficient = %.2f." % r_value)
print("P-value = %.2f." % p_value)
print("Standard error = %.2f." % stderr)

pearson_r, pearson_p = stats.pearsonr(odp_data.oxygen, odp_data.carbon)
spearman_r, spearman_p = stats.spearmanr(odp_data.oxygen, odp_data.carbon)

print("Pearson r = %.2f." % pearson_r)
print("Spearman r = %.2f." % spearman_r)
```

```
Slope = 0.15.
Intercept = 1.52.
Correlation coefficient = 0.21.
P-value = 0.01.
Standard error = 0.06.
Pearson r = 0.21.
Spearman r = 0.20.
```

```python
# Create a plot
plt.figure(figsize=(5,5))

plt.scatter(odp_data.oxygen, odp_data.carbon, label="$\delta^{13}C$",
           c="white", edgecolor="magenta")

plt.plot(x, slope*x+intercept, color="black", lw=2)

plt.ylabel("$\delta^{13}C$ (‰)")
plt.xlabel("$\delta^{18}O$ (‰)")

plt.xlim(0,2.5)
plt.ylim(0,2.5)

plt.show()
```

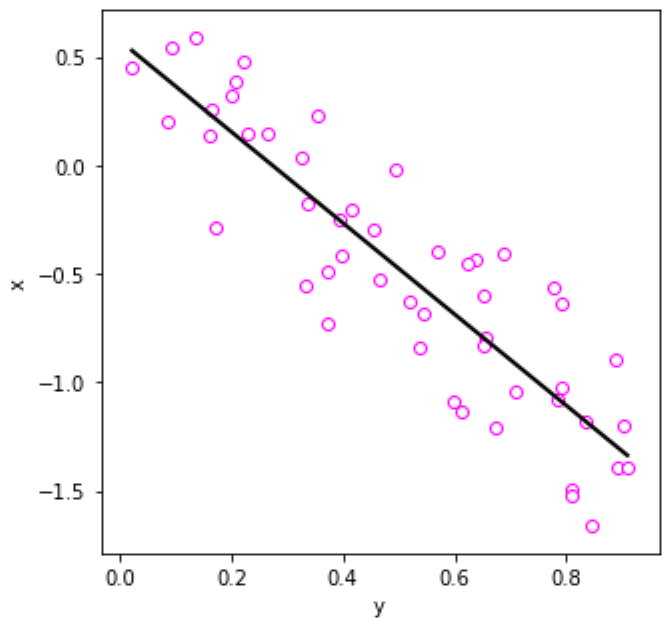


As expected, our data is quite scattered and correlation coefficient is < 0.3, meaning there is very weak correlation. Let's generate some random data and fit a line:

```
# Create random data
np.random.seed(20202020)
x = np.random.random(50)
y = -2*x + np.random.random(50)

# Create equally space data for fitted line
xx = np.linspace(np.min(x), np.max(x), 100)

# Fit a line
slope, intercept, r_value, p_value, stderr = stats.linregress(x, y)

print("Slope = %.2f." % slope)
print("Intercept = %.2f." % intercept)
print("Correlation coefficient = %.2f." % r_value)
print("P-value = %.2f." % p_value)
print("Standard error = %.2f." % stderr)

pearson_r, pearson_p = stats.pearsonr(x, y)
spearman_r, spearman_p = stats.spearmanr(x, y)

print("Pearson r = %.2f." % pearson_r)
print("Spearman r = %.2f." % spearman_r)
```

```
Slope = -2.10.
Intercept = 0.58.
Correlation coefficient = -0.89.
P-value = 0.00.
Standard error = 0.16.
Pearson r = -0.89.
Spearman r = -0.88.
```

```
# Create a plot
plt.figure(figsize=(5,5))

plt.scatter(x, y, label="$\delta^{13}C$",
           c="white", edgecolor="magenta")

plt.plot(xx, slope*xx+intercept, color="black", lw=2)

plt.ylabel("x")
plt.xlabel("y")

plt.show()
```



This time data shows strong negative correlation.

Inferential statistics

Inferential statistics aims to infer properties of data based on hypotheses testing. Hypotheses will test probability.

Probability p describes how likely an event is to occur. It ranges between $(0, 1)$, where $p = 0$ implies that event will never occur and $p = 1$ means that the event will always occur.

Hypotheses

We can formulate two hypotheses - the *null hypothesis* (H0) and an *alternative hypothesis* (H1). We will only accept H1, if it is very unlikely for H0 to explain our results.

For example, we have grain sizes for samples A and B, where sample A has higher mean:

- H0 - There is no difference in mean grain size between A and B.
- H1 - Mean grain size is higher in sample A than B.

Student’s T-test

Different test types are available. For example, we can use Student’s T-test to check the probability for two samples to have the same underlying mean. Scipy provides:

```
scipy.stats.ttest_ind(a, b)
```

This test will return *t-statistic* which describes how the means are different from one another and *p-value*, the probability of null hypothesis being true.

We can use this test on some randomly generated data with normal distributions.

```
# Generate data that will have the same mean
a = stats.norm.rvs(size=100000, random_state=12345)
b = stats.norm.rvs(size=100000, random_state=67890)

# Use Student's T-test
t_statistic, p_value = stats.ttest_ind(a, b)
print ("t-statistic = %.2f." % t_statistic)
print ("p-value = %.2f." % p_value)
```

```
t-statistic = 0.77.
p-value = 0.44.
```

If observed p-value is > 0.05 or 0.10, we cannot reject the null hypothesis. The threshold is arbitrary, $p = 0.05$ is just most commonly used.

T-test is commonly used but it is not always appropriate to use:

- Assumes normally distributed samples
- Needs a large sample size to give sensible results

Tails

Instead of testing how different the means are, we could test whether the mean of A is greater than the mean of B. For that we can use *one-tailed* test.

Alternatively, we can take a two-tailed test to find whether mean of A is higher or lower than mean of B.

When using scipy.stats tests, make sure you check whether it is a one-tailed or two-tailed test!

Non-parametric tests

Tests that don’t require data to be normally distributed are for example, the Mann-Whitney rank test and Kolmogorow-Smirnov test for goodness of fit. Two-sample K-S test can be used as an alternative to the t-test. It answers the question whether samples come from the same underlying population.

Testing for normal distribution

We can test data with hypotheses:

- H0 - underlying distribution is normal
- H1 - underlying distibution is not normal

Scipy provides:

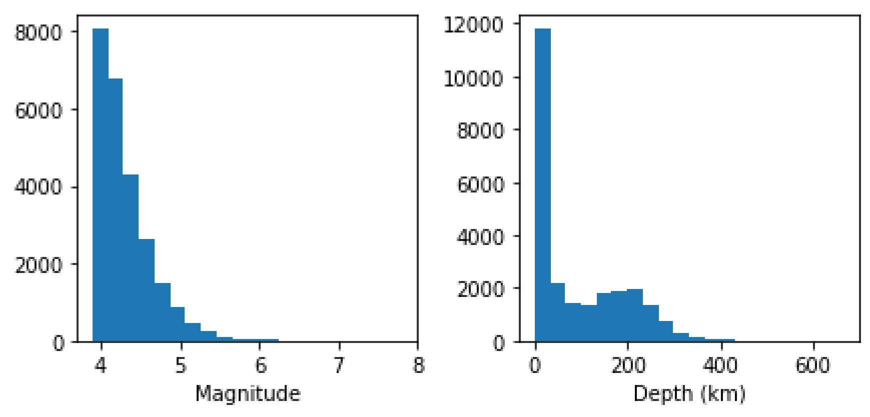
```
scipy.stats.normaltest()
```

It tests the null hypothesis if a sample comes from a normal distribution. We can test earthquake depth and magnitude data from New Zealand whether they come from normal distribution:

```
nz_eqs = pd.read_csv("data/nz_largest_eq_since_1970.csv")
nz_eqs.head()
```

	year	month	day	utc_time	mag	lat	lon	depth_km	region	iris_id	timestamp
0	2009	7	15	09:22:31	7.8	-45.8339	166.6363	20.9	OFF W. COAST OF S. ISLAND, N.Z.	2871698	1247649751
1	2016	11	13	11:02:59	7.8	-42.7245	173.0647	22.0	SOUTH ISLAND, NEW ZEALAND	5197722	1479034979
2	2003	8	21	12:12:47	7.2	-45.0875	167.0892	6.8	SOUTH ISLAND, NEW ZEALAND	1628007	1061467967
3	2001	8	21	06:52:06	7.1	-36.8010	-179.7230	33.5	EAST OF NORTH ISLAND, N.Z.	1169374	998376726
4	2004	11	22	20:26:23	7.1	-46.4964	164.8273	7.6	OFF W. COAST OF S. ISLAND, N.Z.	1888053	1101155183

```
fig, axes = plt.subplots(1,2, figsize=(7,3))

ax1 = axes[0]
ax1.hist(nz_eqs.mag,bins=20)
ax1.set_xlabel("Magnitude")

ax2 = axes[1]
ax2.hist(nz_eqs.depth_km,bins=20)
ax2.set_xlabel("Depth (km)")

plt.subplots_adjust(wspace=0.3)
plt.show()
```



The distributions do not look normal, however, we can test them to get a more robust evidence:

```
statistic, p_value = stats.normaltest(nz_eqs.mag)

print("P-value = %.9f." % p_value)

statistic, p_value = stats.normaltest(nz_eqs.depth_km)
print("P-value = %.9f." % p_value)
```

```
P-value = 0.000000000.
P-value = 0.000000000.
```

In both cases, the p-value is so small, the null hypothesis is rejected, and therefore, the distributions are not normal.

Chi-squared test (or χ^2 test)

Chi-squared test is used for discrete (categorised data). For example, discrete data may be rock type (granite, sandstone, limestone, etc.) or fault type (normal, strike-slip, etc.).

This test assesses how likely it is that counts of discrete data fit some expected pattern. In Python we can use a chi-squared test with:

```
scipy.stats.chisquare()
```

that tests the null hypothesis that the categorical data has the given frequency. It returns *s-statistic* that is the value of Chi-squared and *p-value* which is the two-tailed probability of the result occuring by chance.

Exercise

Let's say that we analyse 2000 grades and they fall into fail, 2rd, 2ii, 2i and 1st categories. We find that the marks breakdown is

Fail: 3.2% 3rd: 11.5% 2ii: 15.9% 2i: 41.3% 1st: 28.1

We will consider now a group of students and their marks:

Grade	Group 1	Group 2
Fail	1	0
3rd	12	8
2ii	23	8
2i	29	24
1st	21	40

Chi-squared test will help us determine whether these grades are atypical.

```
observed1 = np.array([1, 12, 23, 29, 21])
observed2 = np.array([0, 8, 8, 24, 40])

predicted = np.array([3.2, 11.5, 15.9, 41.3, 28.1])

s_statistic, p_value = stats.chisquare(observed1, predicted)
print("Group 1 p-value = %.5f." % p_value)

s_statistic, p_value = stats.chisquare(observed2, predicted)
print("Group 2 p-value = %.5f." % p_value)
```

```
Group 1 p-value = 0.03779.
Group 2 p-value = 0.00040.
```

The p-value indicates that these groups have quite typical mark distribtion.