Lecture 6: High dimensional data

Sources, linear algebra and plotting

Nils Olovsson
nils.olofsson@igp.uu.se

Introduction to data science and Python programming for medical research
Uppsala University

2025

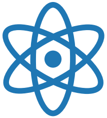
UPPSALA
UNIVERSITET

Content

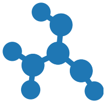
High dimensional data

High dimensional data

High dimensional data: Data

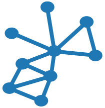


Atoms

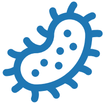


Molecules

Genes

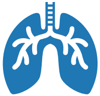


Interactions

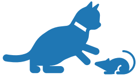


Cells

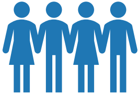
Tissue



Organs



Organisms



Populations

High dimensional data: Clinical setting

Anything that measures more than one single thing.
Data can be collected for studies or be part of patient medical record.

- Blood pressure
- Heart rate
- Skin conductivity
- Oxygen/gas exchange
- Blood
- Metabolites
- Genetics
- Family history
- ...

High dimensional data: (Example) Diabetes dataset

Ten measured baseline quantities and a quantitative description of disease (diabetes) progression.

TABLE 1

Diabetes study: 442 diabetes patients were measured on 10 baseline variables; a prediction model was desired for the response variable, a measure of disease progression one year after baseline

	AGE	SEX	BMI	BP	Serum measurements						Response
Patient	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	y
1	59	2	32.1	101	157	93.2	38	4	4.9	87	151
2	48	1	21.6	87	183	103.2	70	3	3.9	69	75
3	72	2	30.5	93	156	93.6	41	4	4.7	85	141
4	24	1	25.3	84	198	131.4	40	5	4.9	89	206
5	50	1	23.0	101	192	125.4	52	4	4.3	80	135
6	23	1	22.6	89	139	64.8	61	2	4.2	68	97
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
441	36	1	30.0	95	201	125.2	42	5	5.1	85	220
442	36	1	19.6	71	250	133.2	97	3	4.6	92	57

Adapted from "Least angle regression" Efron et al. (2004).

https://scikit-learn.org/stable/datasets/toy_dataset.html#diabetes-dataset

7.1.2. Diabetes dataset

Ten baseline variables, age, sex, body mass index, average blood pressure, and six blood serum measurements were obtained for each of n = 442 diabetes patients, as well as the response of interest, a quantitative measure of disease progression one year after baseline.

Data Set Characteristics:

Number of Instances: 442

Number of Attributes: First 10 columns are numeric predictive values

Target: Column 11 is a quantitative measure of disease progression one year after baseline

Attribute Information:

- age age in years
- sex
- bmi body mass index
- bp average blood pressure
- s1 tc, total serum cholesterol
- s2 ldl, low-density lipoproteins
- s3 hdl, high-density lipoproteins
- s4 tch, total cholesterol / HDL
- s5 ltg, possibly log of serum triglycerides level
- s6 glu, blood sugar level

Note: Each of these 10 feature variables have been mean centered and scaled by the standard deviation times the square root of `n_samples` (i.e. the sum of squares of each column totals 1).

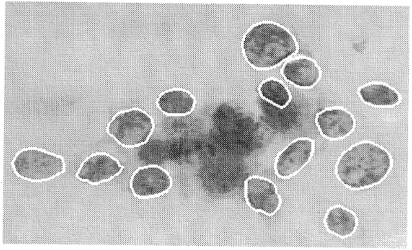
Source URL: https://www4.stat.ncsu.edu/~boos/var.select/diabetes.html

For more information see: Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani (2004) "Least Angle Regression," Annals of Statistics (with discussion), 407-499. (https://web.stanford.edu/~hastie/Papers/LARS/LeastAngle_2002.pdf)

High dimensional data: (Example) Breast cancer dataset

30 features and a classification label.

"Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image."



Cell nucleus boundaries. Street et al. (1993).

7.1.6. Breast cancer wisconsin (diagnostic) dataset

Data Set Characteristics:

Number of Instances: 569

Number of Attributes: 30 numeric, predictive attributes and the class

Attribute Information:

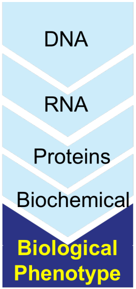
- radius (mean of distances from center to points on the perimeter)
- texture (standard deviation of gray-scale values)
- perimeter
- area
- smoothness (local variation in radius lengths)
- compactness (perimeter^2 / area - 1.0)
- concavity (severity of concave portions of the contour)
- concave points (number of concave portions of the contour)
- symmetry
- fractal dimension ("coastline approximation" - 1)

The mean, standard error, and "worst" or largest (mean of the three worst/largest values) of these features were computed for each image, resulting in 30 features. For instance, field 0 is Mean Radius, field 10 is Radius SE, field 20 is Worst Radius.

- **class:**
  - WDBC-Malignant
  - WDBC-Benign

doi:10.1117/12.148698
https://scikit-learn.org/stable/datasets/toy_dataset.html#breast-cancer-wisconsin-diagnostic-dataset
https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic

High dimensional data: Single cell analysis

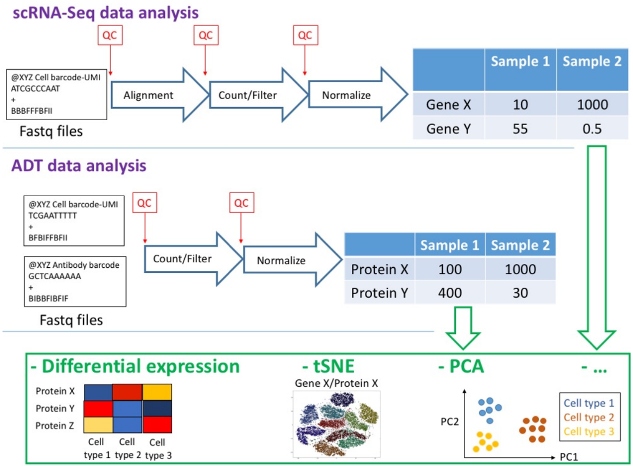


Genomics

Transcriptomics

Proteomics

Metabolomics

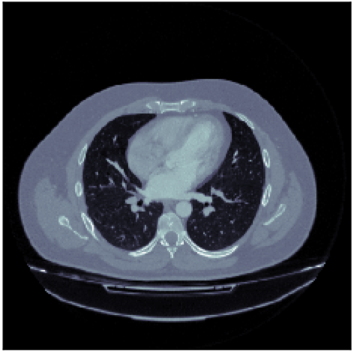


https://en.wikipedia.org/wiki/Metabolomics
https://en.wikipedia.org/wiki/CITE-Seq

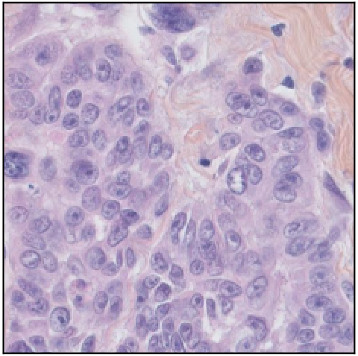
High dimensional data: Medical and biomedical images

Each pixel can be considered a dimension.
One CT scan of $512 \times 512 \times 100$ pixels has 26 million dimensions!
(How many of these that are actually useful is another question.)

Computed tomography (CT)



Histopathological image



High dimensional data: Toy datasets

In order to familiarize yourselves with the methods used to analyze data it is very convenient to be able to **find or create test data**.

sklearn contains a few **data sets** as well as methods to **generate** various forms of **test data**.

Test data sampled from **normal distributions** and a few others can be generated using the **random** module in **numpy**.

https://scikit-learn.org/stable/api/sklearn.datasets.html

Sample generators

make_biclusters	Generate a constant block diagonal structure array for biclustering.
make_blobs	Generate isotropic Gaussian blobs for clustering.
make_checkerboard	Generate an array with block checkerboard structure for biclustering.
make_circles	Make a large circle containing a smaller circle in 2d.
make_classification	Generate a random n-class classification problem.
make_friedman1	Generate the "Friedman #1" regression problem.
make_friedman2	Generate the "Friedman #2" regression problem.
make_friedman3	Generate the "Friedman #3" regression problem.
make_gaussian_quantiles	Generate isotropic Gaussian and label samples by quantile.
make_hastie_10_2	Generate data for binary classification used in Hastie et al. 2009, Example 10.2.
make_low_rank_matrix	Generate a mostly low rank matrix with bell-shaped singular values.
make_moons	Make two interleaving half circles.
make_multilabel_classification	Generate a random multilabel classification problem.
make_regression	Generate a random regression problem.
make_s_curve	Generate an S curve dataset.
make_sparse_coded_signal	Generate a signal as a sparse combination of dictionary elements.
make_sparse_spd_matrix	Generate a sparse symmetric definite positive matrix.
make_sparse_uncorrelated	Generate a random regression problem with sparse uncorrelated design.
make_spd_matrix	Generate a random symmetric, positive-definite matrix.
make_swiss_roll	Generate a swiss roll dataset.

High dimensional data: Curse of dimensionality

As the dimensionality of our datasets get larger some **observations** can generally be made.

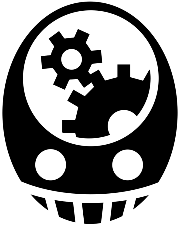
In order to cover as much relative *volume* of the data space the **sample size should increase**.

Some methods scale poorly as the dimensionality increases either in the data or in the number of parameters required to model it.

Etc ...

High dimensional data: Topics of this week

- Machine learning.
- Starting to work with data by applying clustering.
- Reducing the dimensionality of the data both to visualize it but also to find a **latent space**.
- Look briefly at machine learning methods for classification and regression.
- In order to make sense of these methods an introduction to some basic linear algebra is necessary.



Vectors

Vectors

Vectors: Data points, vectors in 2D, 3D and ND

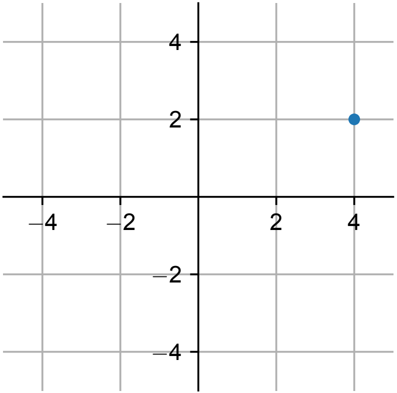
A **data point** can be thought of as a **vector**.

A vector is an **arrow** that points from the origin to our data point.

We then get a whole set of operations and techniques from **linear algebra** we can use to **operate on the data**.

Vectors are typically written with lower case letters in bold font or with some decorator indicating that they are vectors.

$$\mathbf{a} = \begin{pmatrix} 4 & 2 \end{pmatrix}$$



Vectors: Data points, vectors in 2D, 3D and ND

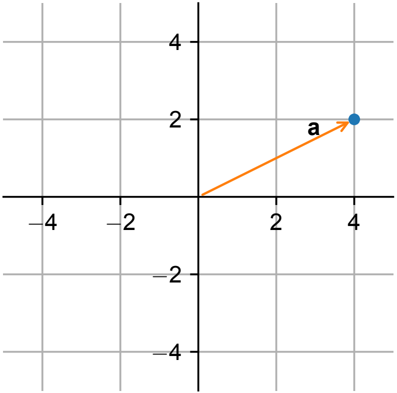
A **data point** can be thought of as a **vector**.

A vector is an **arrow** that points from the origin to our data point.

We then get a whole set of operations and techniques from **linear algebra** we can use to **operate on the data**.

Vectors are typically written with lower case letters in bold font or with some decorator indicating that they are vectors.

$$\mathbf{a} = \begin{pmatrix} 4 & 2 \end{pmatrix}$$



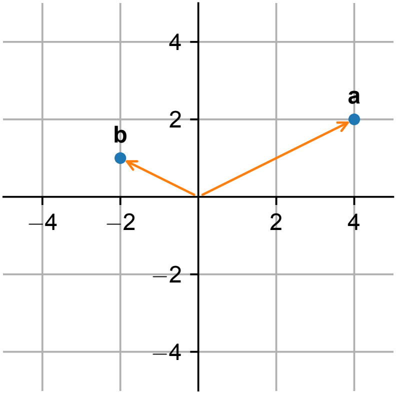
Vectors: Add and subtract vectors

Two vectors are added by placing the tail of one arrow at the head of the other.

This simply adds the elements of the vectors.

Subtraction of one vector from another works analogously.

```
import numpy as np
a = np.array([ 4, 2])
b = np.array([-2, 1])
c = a + b
```



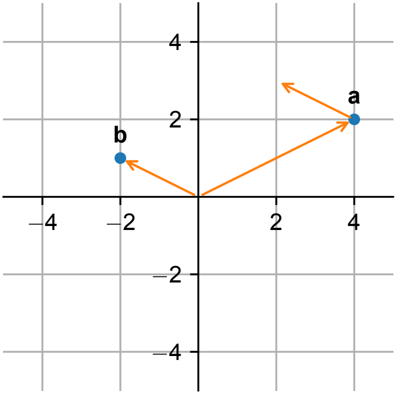
Vectors: Add and subtract vectors

Two vectors are added by placing the tail of one arrow at the head of the other.

This simply adds the elements of the vectors.

Subtraction of one vector from another works analogously.

```
import numpy as np
a = np.array([ 4, 2])
b = np.array([-2, 1])
c = a + b
```



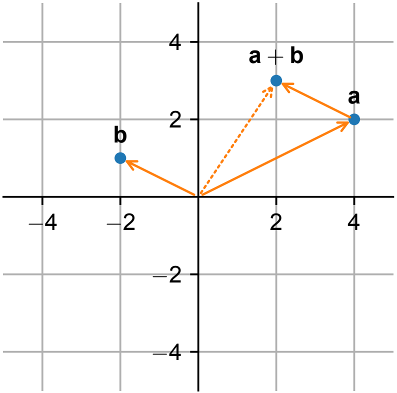
Vectors: Add and subtract vectors

Two vectors are added by placing the tail of one arrow at the head of the other.

This simply adds the elements of the vectors.

Subtraction of one vector from another works analogously.

```python
import numpy as np
a = np.array([ 4, 2])
b = np.array([-2, 1])
c = a + b
```

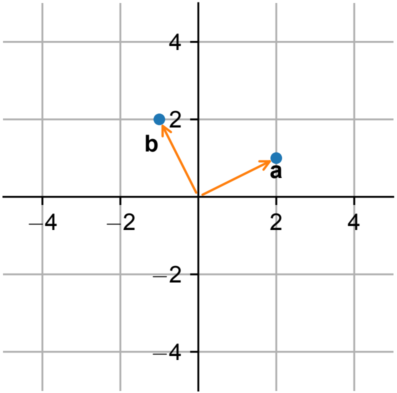


Vectors: Multiplying with or adding/subtracting a scalar

Multiplying with a scalar will scale the vector along its direction.

Adding/subtracting a scalar is mathematically a little weird!
If the vector is not uniform it will change the direction of the vector.

```
import numpy as np

# Initialize scalars (regular numbers)
s0 = 2.0
s1 = 3.0

# Initialize vectors
a = np.array([ 2, 1])
b = np.array([-1, 2])

# Multiply with or add scalars with vectors
c = s0 * a
d = b + s1
```

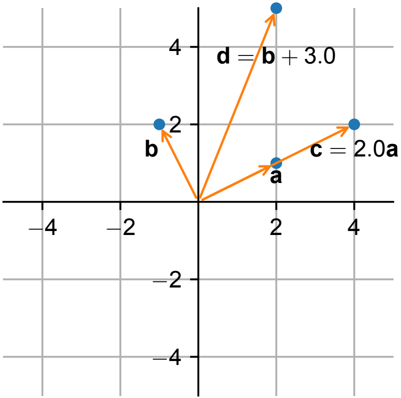


Vectors: Multiplying with or adding/subtracting a scalar

Multiplying with a scalar will scale the vector along its direction.

Adding/subtracting a scalar is mathematically a little weird!
If the vector is not uniform it will change the direction of the vector.

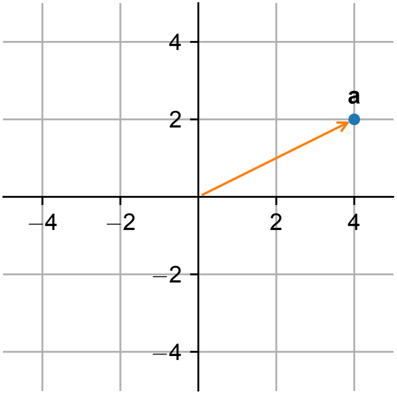
```python
import numpy as np

# Initialize scalars (regular numbers)
s0 = 2.0
s1 = 3.0

# Initialize vectors
a = np.array([ 2, 1])
b = np.array([-1, 2])

# Multiply with or add scalars with vectors
c = s0 * a
d = b + s1
```



Vectors: Negating

A vector can be inverted such that its direction is flipped by multiplying it by -1.

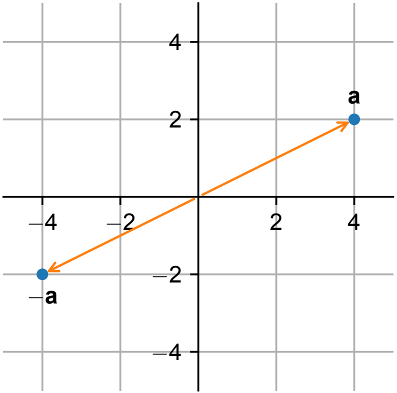
```
import numpy as np
a = np.array([4, 2])
inv_a = -a
```



Vectors: Negating

A vector can be inverted such that its direction is flipped by multiplying it by -1.

```
import numpy as np
a = np.array([4, 2])
inv_a = -a
```



Vectors: Length

The Euclidean **length of a vector v** is defined as

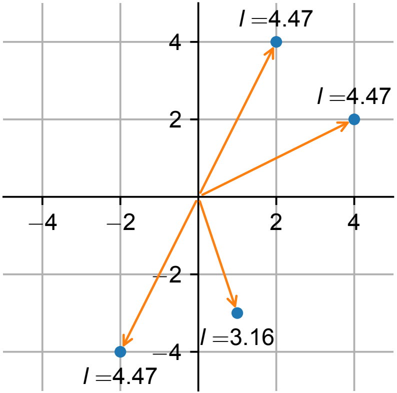
$$l = |\mathbf{v}| = \sqrt{\sum_{i=1}^{N} v_i^2}$$

In 2D this is just the Pythagorean formula for a right angled triangle

$$l = |\mathbf{v}| = \sqrt{v_1^2 + v_2^2}$$

Vectors with different directions can have the same length.

```python
import numpy as np
a = np.array([4, 2])
l = np.linalg.norm(a)
```

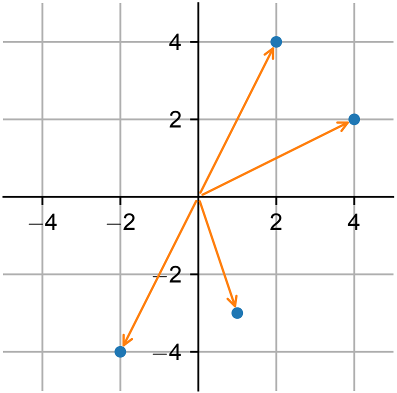


Vectors: Unit vectors and normalization

Unit vectors are **vectors with length 1**.

A vector can be **normalized** to have length 1 by **dividing it with** its original **length**.

```
import numpy as np
v = np.array([4, 2])
l = np.linalg.norm(v)
n = (1.0 / l) * v
```

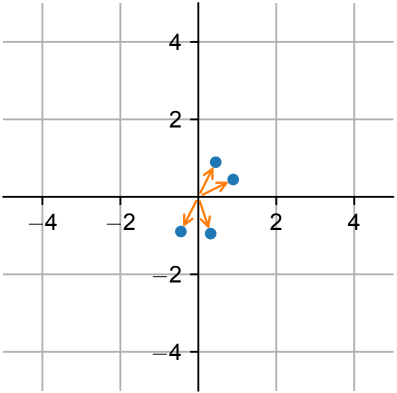


Vectors: Unit vectors and normalization

Unit vectors are **vectors with length 1**.

A vector can be **normalized** to have length 1 by **dividing it with** its original **length**.

```
import numpy as np
v = np.array([4, 2])
l = np.linalg.norm(v)
n = (1.0 / l) * v
```



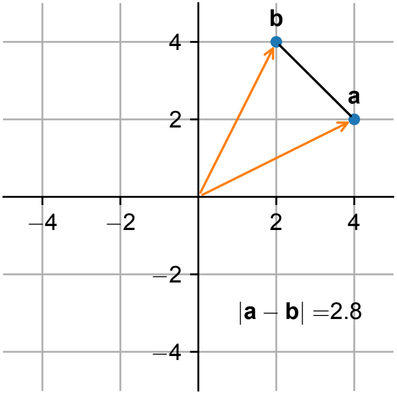
Vectors: Distance between vectors

The distance between two vectors, or data points, is the length of their difference.

$$d = |\mathbf{a} - \mathbf{b}|$$

Measures how dissimilar two data points are.

```
import numpy as np

# Initialize two vectors and compute
# distance between them
a = np.array([4, 2])
b = np.array([2, 4])
d = np.linalg.norm(b - a)
```



Vectors: Dot product

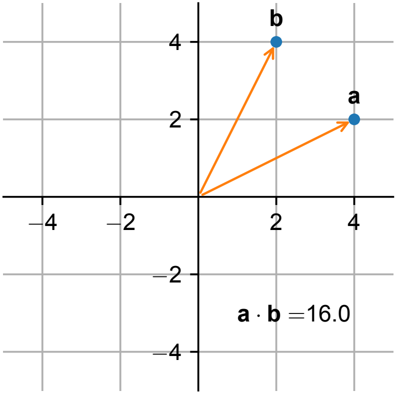
The dot product between two vectors, **a** and **b**, gives a scalar.

$$s = \mathbf{a} \cdot \mathbf{b} = \langle \mathbf{a}, \mathbf{b} \rangle$$

It is defined as the sum of the element wise products between entries in both vectors.

$$s = \mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{N} a_i b_i$$

```
import numpy as np

# Initialize two vectors and compute
# distance between them
a = np.array([4, 2])
b = np.array([2, 4])
s = np.dot(a, b)
```



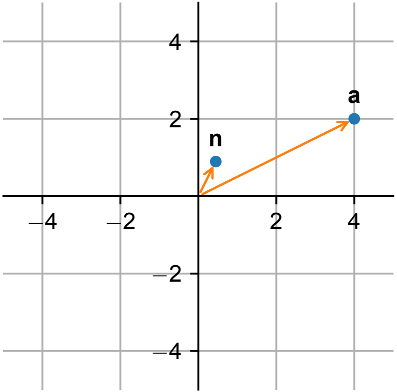
Vectors: Projection

Can compute the dot product between a vector **a** and a unit vector **n**.

Multiplying the scalar result with the unit vector will be the projection of **a** on **n**

$$\mathbf{p} = (\mathbf{a} \cdot \mathbf{n})\mathbf{n}$$

```python
import numpy as np
# Create two vectors
a = np.array([4, 2])
b = np.array([2, 4])
# Normalize the b vector
l = np.linalg.norm(b)
n = (1.0 / l) * b
# Compute the scalar product to project a on n
p = np.dot(a, n) * n
```



The dot product between two unit vectors can be used to measure how much two vectors are aligned. If their scalar product is 0, they are perpendicular to each other.

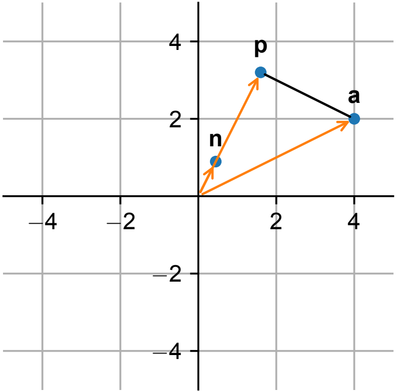
Vectors: Projection

Can compute the dot product between a vector **a** and a unit vector **n**.

Multiplying the scalar result with the unit vector will be the projection of **a** on **n**

$$\mathbf{p} = (\mathbf{a} \cdot \mathbf{n})\mathbf{n}$$

```python
import numpy as np
# Create two vectors
a = np.array([4, 2])
b = np.array([2, 4])
# Normalize the b vector
l = np.linalg.norm(b)
n = (1.0 / l) * b
# Compute the scalar product to project a on n
p = np.dot(a, n) * n
```



The dot product between two unit vectors can be used to measure how much two vectors are aligned. If their scalar product is 0, they are perpendicular to each other.

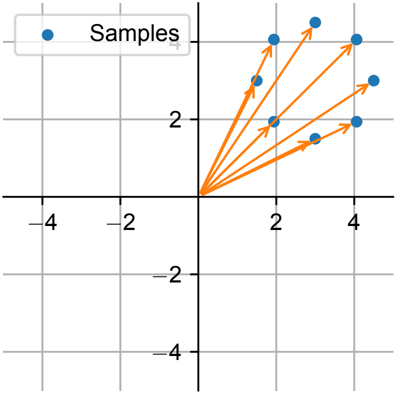
Vectors: Mean value of vectors

The mean value of a set of vectors is just the center of those vectors.

Can be calculated as the mean value along each dimension/feature.

E.g. in 2D it is the mean value along x and the mean value along y.

```
N = 8
c = np.array([3, 3])
p = np.zeros((N, 2))
r = 1.5
angles = np.linspace(0, 2*np.pi, N, endpoint=False)
for i, angle in enumerate(angles):
    x = r * np.cos(angle)
    y = r * np.sin(angle)
    p[i,:] = np.array([x, y]) + c
# Compute the mean value vector
m = np.mean(p, axis=0)
```



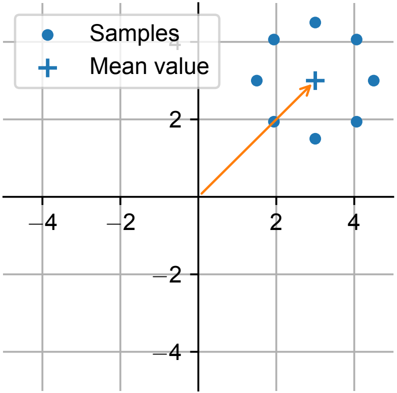
Vectors: Mean value of vectors

The mean value of a set of vectors is just the center of those vectors.

Can be calculated as the mean value along each dimension/feature.

E.g. in 2D it is the mean value along x and the mean value along y.

```
N = 8
c = np.array([3, 3])
p = np.zeros((N, 2))
r = 1.5
angles = np.linspace(0, 2*np.pi, N, endpoint=False)
for i, angle in enumerate(angles):
    x = r * np.cos(angle)
    y = r * np.sin(angle)
    p[i,:] = np.array([x, y]) + c
# Compute the mean value vector
m = np.mean(p, axis=0)
```



Vectors: Row and column vectors

Vectors can be defined either as a row or a column.
Matters when performing certain mathematical operations.

In numpy these vectors are 2D arrays but with one of the shape components being 1.

$$\mathbf{a} = \begin{pmatrix} 2 & 5 & 1 \end{pmatrix}$$

$$\mathbf{b} = \begin{pmatrix} 2 \\ 5 \\ 1 \end{pmatrix}$$

```python
# Create a row vector and transpose it to get
# a column vector
import numpy as np
a = np.array([[2, 5, 1]])
b = a.T

print("## Vector a")
print(f"a.shape: {a.shape}")
print(a)
print("## Vector b")
print(f"b.shape: {b.shape}")
print(b)
```

```
>> ## Vector a
>> a.shape: (1, 3)
>> [[2 5 1]]
>> ## Vector b
>> b.shape: (3, 1)
>> [[2]
>>  [5]
>>  [1]]
```

Matrices

Matrices

Matrices: Table of values

Table or 2D array of values.

Important what the matrix represents.

- Data matrix
- Covariance matrix
- Distance matrix
- Adjacency matrix
- (Linear transforms) (Next week)

Mathematical operations, like with vectors, are a little special.

Write matrices with capital letter.

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$B = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

```python
import numpy as np
A = np.array([[1, 2],[3, 4]])
print(A)
```

```
>> [[1 2]
>>  [3 4]]
```

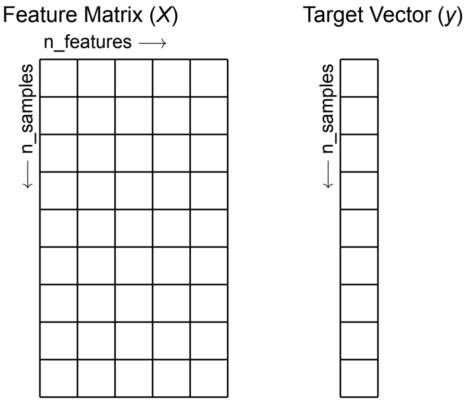
Matrices: Data matrix

Also called the **feature matrix** contains all our sampled feature vectors where **one sample** is represented by one **row**.

If the problem is a **regression or classification** problem we also have **target vector**.

```python
#import numpy as np
from sklearn.datasets import load_breast_cancer
X, y = load_breast_cancer(return_X_y=True)
print(f"Shape of X: {X.shape}")
print(f"Shape of y: {y.shape}")
```

```
>> Shape of X: (569, 30)
>> Shape of y: (569,)
```



Adapted from Python Data Science Handbook (VanderPlas)

Matrices: Covariance matrix

Normal distributions in 1D determined by two scalar values, mean and variance.

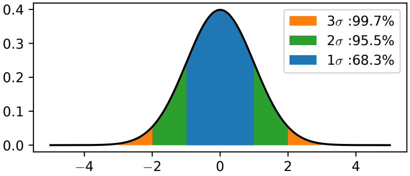
Multivariate normal distributions determined by mean, vector, and covariance matrix, C.

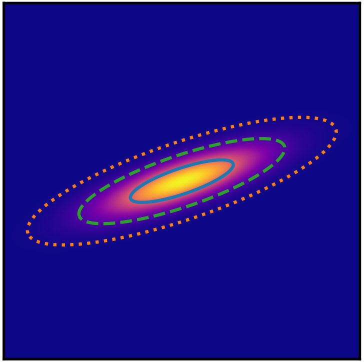
$N \times N$, where N is dimension of the space.

Covariance matrices are square and symmetric.

$$C_{2\mathrm{D}} = \begin{pmatrix} v_x & a \\ a & v_y \end{pmatrix} \quad C_{3\mathrm{D}} = \begin{pmatrix} v_x & a & b \\ a & v_y & c \\ b & c & v_z \end{pmatrix}$$

```python
import numpy as np
from sklearn.datasets import load_diabetes
X, y = load_diabetes(return_X_y=True)
C = np.cov(X)
```





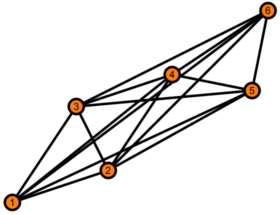
Matrices: Distance matrix

Pairwise distances between points, *e.g.* in matrix X. $M \times M$, where M is samples in the data. Distance matrices are square and, typically, symmetric.

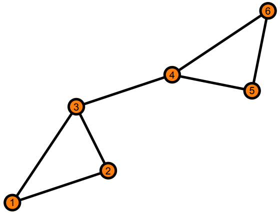
Can connect all points (dense) or subset of points (sparse).

$$D = \begin{pmatrix} 0.0 & 0.6 & 0.7 & 1.3 & 1.7 & 2.0 \\ 0.6 & 0.0 & 0.4 & 0.7 & 1.0 & 1.4 \\ 0.7 & 0.4 & 0.0 & 0.6 & 1.1 & 1.3 \\ 1.3 & 0.7 & 0.6 & 0.0 & 0.5 & 0.7 \\ 1.7 & 1.0 & 1.1 & 0.5 & 0.0 & 0.5 \\ 2.0 & 1.4 & 1.3 & 0.7 & 0.5 & 0.0 \end{pmatrix} \quad D = \begin{pmatrix} 0.0 & 0.6 & 0.7 & 0.0 & 0.0 & 0.0 \\ 0.6 & 0.0 & 0.4 & 0.0 & 0.0 & 0.0 \\ 0.7 & 0.4 & 0.0 & 0.6 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.6 & 0.0 & 0.5 & 0.7 \\ 0.0 & 0.0 & 0.0 & 0.5 & 0.0 & 0.5 \\ 0.0 & 0.0 & 0.0 & 0.7 & 0.5 & 0.0 \end{pmatrix}$$

```python
from sklearn.datasets import load_diabetes
import sklearn.metrics
X, y = load_diabetes(return_X_y=True)
D = sklearn.metrics.pairwise_distances(X)
```



Fully connected graph.

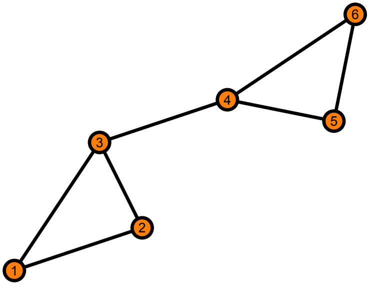


Connects only neighboring points.

Matrices: Adjacency matrix

Pairwise **connections** between points in data matrix, X.
$M \times M$, where M is samples in the data.
Adjacency matrices are square and, typically, symmetric.

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$



Connects only neighboring points.

They typically describe connections between neighboring points. Therefore they are typically very **sparse**, *i.e.* containing mostly zeroes.

Matrices: Addition and subtraction

Matrices of same shape can be added and subtracted.

The operations are performed element wise.

In numpy this is the same thing as adding two arrays.

```python
import numpy as np
rng = np.random.default_rng(seed=0)

# Create two matrices of same shape
A = rng.integers(0, 9, size=(2, 3))
B = rng.integers(0, 9, size=(2, 3))

# Add the matrices together
C = A + B
```

$$\begin{cases} A = \begin{pmatrix} 7 & 5 & 4 \\ 2 & 2 & 0 \end{pmatrix} \\ B = \begin{pmatrix} 0 & 0 & 1 \\ 7 & 5 & 8 \end{pmatrix} \end{cases}$$

$$\begin{aligned} C &= A + B \\ &= \begin{pmatrix} 7 & 5 & 4 \\ 2 & 2 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 1 \\ 7 & 5 & 8 \end{pmatrix} \\ &= \begin{pmatrix} 7 & 5 & 5 \\ 9 & 7 & 8 \end{pmatrix} \end{aligned}$$

(Addition or subtraction of a scalar works element wise on the matrix.)

Matrices: Multiplication

Matrix multiplication is special.

The number of columns in A must be equal to the number of rows in B.
C has the number of rows of A and the number of columns of B.

In numpy two matrices represented by arrays can be multiplied using `@`.

```python
import numpy as np
rng = np.random.default_rng(seed=1)

# Create two matrices of compatible shape
A = rng.integers(0, 9, size=(3, 2))
B = rng.integers(0, 9, size=(2, 3))

# Multiply the matrices together
C = A @ B
```

$$\begin{cases} A = \begin{pmatrix} 4 & 4 \\ 6 & 8 \\ 0 & 1 \end{pmatrix} \\ B = \begin{pmatrix} 7 & 8 & 2 \\ 2 & 7 & 3 \end{pmatrix} \end{cases}$$

$$\begin{aligned} C &= AB \\ &= \begin{pmatrix} 4 & 4 \\ 6 & 8 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 7 & 8 & 2 \\ 2 & 7 & 3 \end{pmatrix} \\ &= \begin{pmatrix} 36 & 60 & 20 \\ 58 & 104 & 36 \\ 2 & 7 & 3 \end{pmatrix} \end{aligned}$$

(Multiplication or division with a scalar works element wise on the matrix.)

Matrices: Identity matrix

The identity matrix, I, is a square matrix with ones along the diagonal and zeros everywhere else.

When a square matrix of same size is multiplied by I the result is the same matrix.
Analogous to multiplying an ordinary number by 1.

$$\begin{cases} A = \begin{pmatrix} 4 & 4 & 6 & 8 \\ 0 & 1 & 7 & 8 \\ 2 & 2 & 7 & 3 \\ 2 & 7 & 2 & 3 \end{pmatrix} \\ I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{cases}$$

```python
import numpy as np
rng = np.random.default_rng(seed=1)

# Create one square matrix
A = rng.integers(0, 9, size=(4, 4))
# Create identity matrix of same size
I = np.eye(4)

# Multiply the matrices together
C = I @ A
```

$$\begin{aligned} C &= IA \\ &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 4 & 4 & 6 & 8 \\ 0 & 1 & 7 & 8 \\ 2 & 2 & 7 & 3 \\ 2 & 7 & 2 & 3 \end{pmatrix} \\ &= \begin{pmatrix} 4 & 4 & 6 & 8 \\ 0 & 1 & 7 & 8 \\ 2 & 2 & 7 & 3 \\ 2 & 7 & 2 & 3 \end{pmatrix} \end{aligned}$$

Matrices: Inverse

There is **no division** between matrices.

However the matrix inverse, A^{-1}, is defined such that

$$I = A^{-1}A = AA^{-1}$$

Analogous to how

$$1 = a^{-1}a = aa^{-1}$$

```python
import numpy as np
rng = np.random.default_rng(seed=1)
# Create one square matrix
A = rng.integers(0, 9, size=(3, 3))
# Create identity matrix of same size
Ainv = np.linalg.inv(A)
# Multiply the matrices together
I = Ainv @ A
```

$$\begin{cases} A = \begin{pmatrix} 4 & 4 & 6 \\ 8 & 0 & 1 \\ 7 & 8 & 2 \end{pmatrix} \\ A^{-1} = \begin{pmatrix} -0.03 & 0.13 & 0.01 \\ -0.03 & -0.11 & 0.14 \\ 0.20 & -0.01 & -0.10 \end{pmatrix} \end{cases}$$

$$\begin{aligned} I &= A^{-1}A \\ &= \begin{pmatrix} -0.03 & 0.13 & 0.01 \\ -0.03 & -0.11 & 0.14 \\ 0.20 & -0.01 & -0.10 \end{pmatrix} \begin{pmatrix} 4 & 4 & 6 \\ 8 & 0 & 1 \\ 7 & 8 & 2 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

Computing the matrix inverse is expensive! You almost never compute it explicitly!

Matrices: Transpose

Transposing **flips a matrix on its side**.

If the matrix is **square**, the shape remains the same.
Otherwise the **shape is flipped** as well.

A special case are matrices that are **symmetric** around the diagonal.
These matrices are unaffected by transposing.

Both **distance, adjacency and covariance matrices** are **symmetric**.

```python
import numpy as np
rng = np.random.default_rng(seed=0)
# Create a 3x3 matrix filled with random integers
A = rng.integers(0, 9, size=(3,3))
# Create the transpose of this matrix
At = A.T
```

$$A = \begin{pmatrix} 7 & 5 & 4 \\ 2 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \rightarrow A^T = \begin{pmatrix} 7 & 2 & 0 \\ 5 & 2 & 0 \\ 4 & 0 & 1 \end{pmatrix}$$

$$B = \begin{pmatrix} 7 & 5 & 8 \\ 4 & 5 & 8 \end{pmatrix} \rightarrow B^T = \begin{pmatrix} 7 & 4 \\ 5 & 5 \\ 8 & 8 \end{pmatrix}$$

$$C = \begin{pmatrix} 8 & 1 & -3 \\ 1 & 2 & -1 \\ -3 & -1 & 3 \end{pmatrix} \rightarrow C^T = \begin{pmatrix} 8 & 1 & -3 \\ 1 & 2 & -1 \\ -3 & -1 & 3 \end{pmatrix}$$

$$D = \begin{pmatrix} 6 \\ 5 \\ 4 \end{pmatrix} \rightarrow D^T = \begin{pmatrix} 6 & 5 & 4 \end{pmatrix}$$

Matrices: Eigen decomposition (*Spectral analysis*)

Square, symmetric (positive definite) matrices A can be **factorized** as

$$A = QVQ^{-1}$$

Where the **columns of** Q are its **eigenvectors**, a set of **normalized, linearly independent vectors**. The diagonal of V contains their respective **eigenvalues**.

```python
import numpy as np
from sklearn.datasets import make_spd_matrix
rng = np.random.default_rng(seed=1)
# Create symmetric, positive definite matrix
A = make_spd_matrix(4)
# Eigen decomposition
v, Q = np.linalg.eig(A)
V = np.diag(v)
```

$$A = \begin{pmatrix} 1.33 & -0.14 & -1.01 & -1.32 \\ -0.14 & 0.70 & -0.10 & 0.08 \\ -1.01 & -0.10 & 1.88 & 1.60 \\ -1.32 & 0.08 & 1.60 & 2.33 \end{pmatrix}$$

$$v = \begin{pmatrix} 4.57 & 0.82 & 0.39 & 0.45 \end{pmatrix}$$

$$V = \begin{pmatrix} 4.57 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.82 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.39 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.45 \end{pmatrix}$$

$$Q = \begin{pmatrix} 0.46 & -0.36 & -0.67 & -0.46 \\ -0.02 & 0.81 & -0.04 & -0.58 \\ -0.57 & -0.45 & 0.28 & -0.63 \\ -0.68 & 0.12 & -0.69 & 0.23 \end{pmatrix}$$

Dot product between two **eigenvectors** is 0, *i.e.* they are all perpendicular to each other.

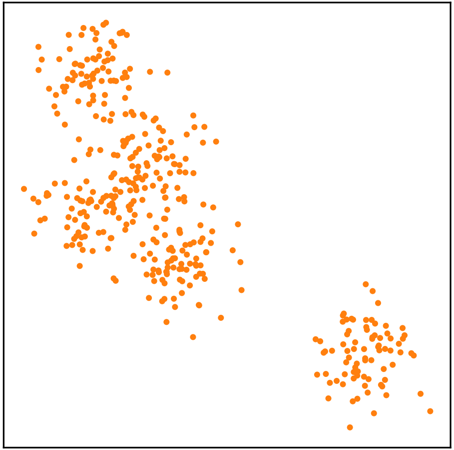
Transformed coordinate system!

Plotting in 2D

Plotting in 2D

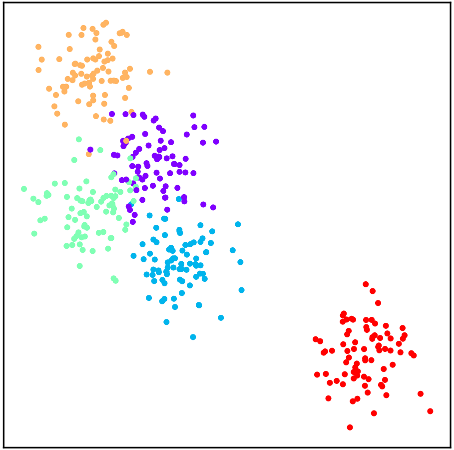
Plotting in 2D: Scatter

```python
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
# Create blobby 2D data
n_clusters = 5
X, y = make_blobs(n_samples=400,
                  n_features=2,
                  centers=n_clusters,
                  cluster_std=1.0,
                  random_state=0)
# Create figure and draw blobby data
fig, ax = plt.subplots(figsize=(4, 4))
ax.scatter(X[:, 0], X[:, 1],
           s=2**2, marker='o',
           facecolor='tab:orange')
```



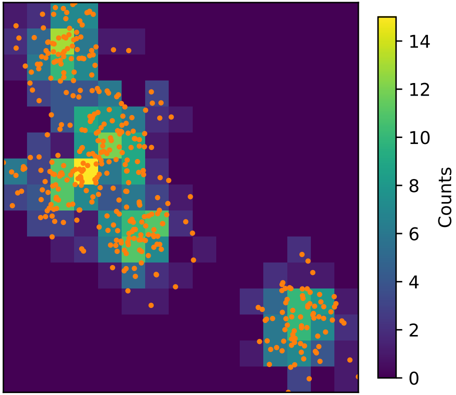
Plotting in 2D: Scatter (Labeled)

```python
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
# Create blobby 2D data
n_clusters = 5
X, y = make_blobs(n_samples=400,
                  n_features=2,
                  centers=n_clusters,
                  cluster_std=1.0,
                  random_state=0)
# Create figure and draw blobby data
fig, ax = plt.subplots(figsize=(4, 4))
cmap = plt.get_cmap('rainbow', n_clusters)
ax.scatter(X[:, 0], X[:, 1],
           s=2**2, marker='o',
           c=y,
           cmap=cmap)
```



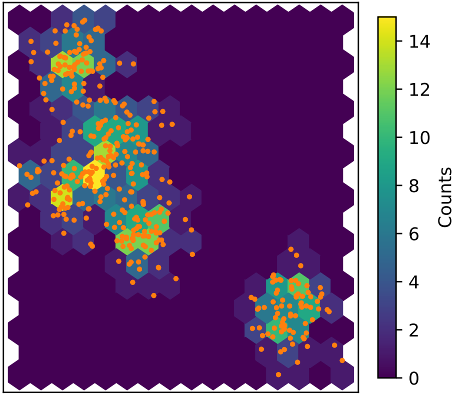
Plotting in 2D: 2D histogram

```python
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
# Create blobby 2D data
n_clusters = 5
X, y = make_blobs(n_samples=400,
                  n_features=2,
                  centers=n_clusters,
                  cluster_std=1.0,
                  random_state=0)
# Create figure and draw blobby data
fig, ax = plt.subplots(figsize=(4, 4))
hs = ax.hist2d(X[:,0], X[:,1], bins=15)
ax.scatter(X[:, 0], X[:, 1],
           s=2**2, marker='o',
           facecolor='tab:orange')
bar = fig.colorbar(ax=ax,
                   mappable=hs[3],
                   label='Counts',
                   fraction=0.046)
#ax.set_aspect('equal')
```



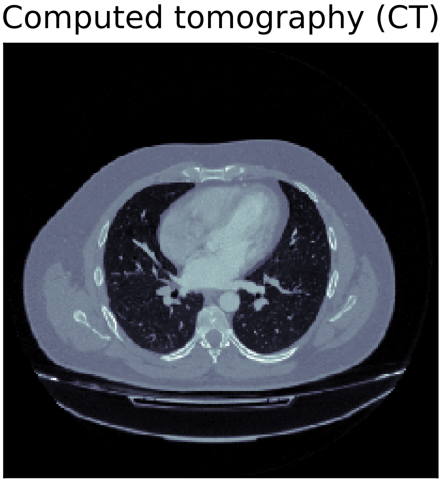
Plotting in 2D: Hexagonal bins

```python
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
# Create blobby 2D data
n_clusters = 5
X, y = make_blobs(n_samples=400,
                  n_features=2,
                  centers=n_clusters,
                  cluster_std=1.0,
                  random_state=0)
# Create figure and draw blobby data
fig, ax = plt.subplots(figsize=(4, 4))
hx = ax.hexbin(X[:,0], X[:,1], gridsize=15)
ax.scatter(X[:, 0], X[:, 1],
           s=2**2, marker='o',
           facecolor='tab:orange')
bar = fig.colorbar(ax=ax,
                   mappable=hx,
                   label='Counts',
                   fraction=0.046)
```



Plotting in 2D: Image (CT)

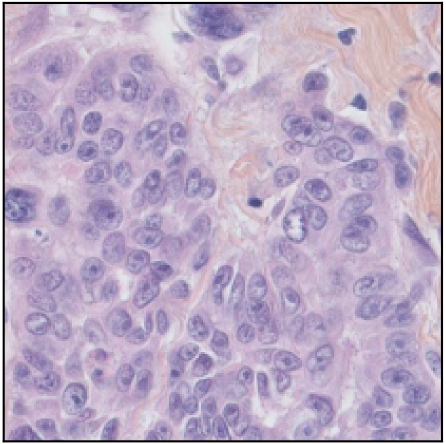
```python
import os
import numpy as np
import matplotlib.pyplot as plt

# Use the pydicom package to load DICOM image
import pydicom

filename = 'image_160.dcm'
directory = os.path.join('..', '..',
                         'data', '3Dircadb2_1')
filepath = os.path.join(directory, filename)
dcm = pydicom.dcmread(filepath)
img = dcm.pixel_array

# Create figure and plot image
ax = plt.figure(figsize=(3,3)).gca()
ax.imshow(img, cmap='bone', interpolation='nearest')
ax.set_title('Computed tomography (CT)')
```



Plotting in 2D: Image (color)

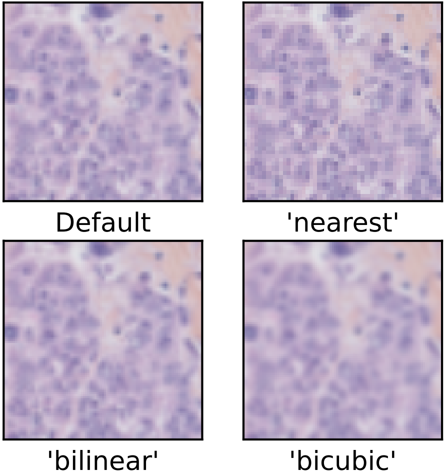
```python
import os
import numpy as np
import matplotlib.pyplot as plt

# Use the skimage package to load jpg image
import skimage.io

filename = 'cell-nuclei.jpg'
directory = os.path.join('..', '..',
                         'data', 'histo')
filepath = os.path.join(directory, filename)

img = skimage.io.imread(filepath)

ax = plt.figure(figsize=(3,3)).gca()

ax.imshow(img)
ax.set_title('Histopathological image')
```

Histopathological image

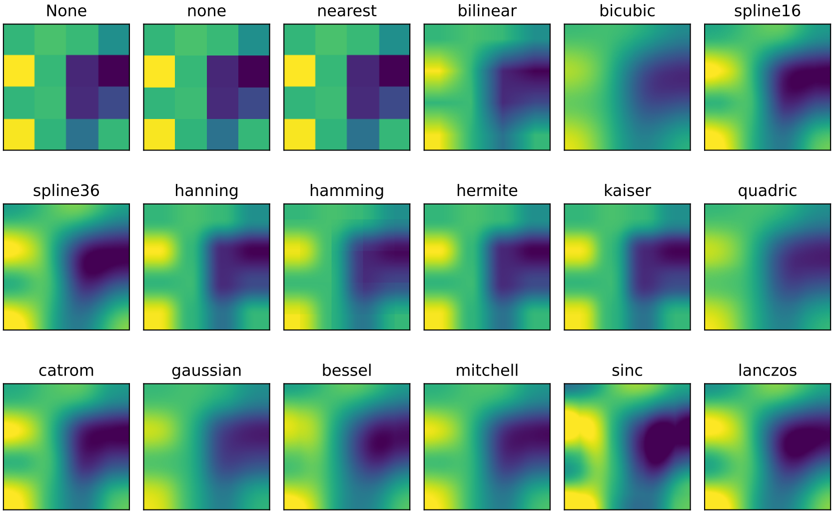


Plotting in 2D: Image interpolation

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

# Use the skimage package to load jpg image
import skimage.io
from skimage.transform import rescale

filename = 'cell-nuclei.jpg'
directory = os.path.join('..', '..',
                         'data', 'histo')
filepath = os.path.join(directory, filename)

img = skimage.io.imread(filepath)
img = rescale(img, 0.1, anti_aliasing=True,
     channel_axis=2)

fig, axs = plt.subplots(2, 2, figsize=(3,3))
axs[0, 0].imshow(img)
axs[0, 1].imshow(img, interpolation='nearest')
axs[1, 0].imshow(img, interpolation='bilinear')
axs[1, 1].imshow(img, interpolation='bicubic')

axs[0, 0].set_xlabel('Default')
axs[0, 1].set_xlabel("'nearest'")
axs[1, 0].set_xlabel("'bilinear'")
axs[1, 1].set_xlabel("'bicubic'")
```

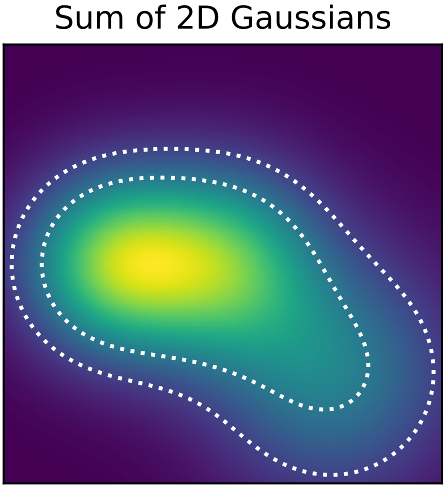


Plotting in 2D: Image interpolation settings



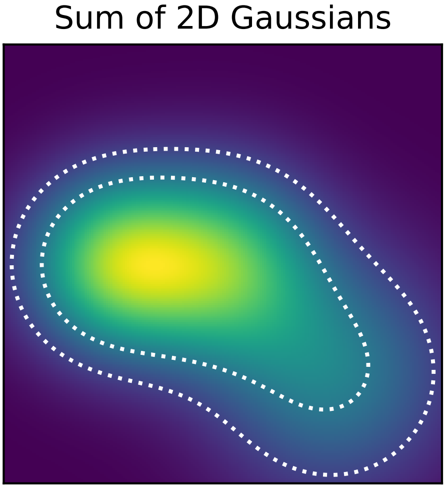
Plotting in 2D: Contour

```
import numpy as np
import matplotlib.pyplot as plt
# Generate 2D Gaussian data
x = np.linspace(-4, 4, 400, endpoint=True)
X = x[:, np.newaxis] + np.zeros((1, 400))
Y = X.T
z0 = common.gauss_2d(X, Y, np.array([0, 0]), 1.5 *
     np.array([[1,0],[0,1]]) )
z1 = common.gauss_2d(X, Y, np.array([2, 2]), 2.3 *
     np.array([[1,0],[0,1]]) )
z2 = common.gauss_2d(X, Y, np.array([0,-2]), 1.3 *
     np.array([[1,0],[0,1]]) )
z = z1 + z0 + z2
# Set two levels
zmin = np.amin(z)
zmax = np.amax(z)
r = zmax - zmin
levels = [zmin + 0.2 * r, zmin + 0.4 * r]
# Create figure and plot data
ax = plt.figure(figsize=(3,3)).gca()
ax.imshow(z)
ax.contour(z, levels=levels, colors='white',
     linestyles=':')
ax.set_title('Sum of 2D Gaussians')
```



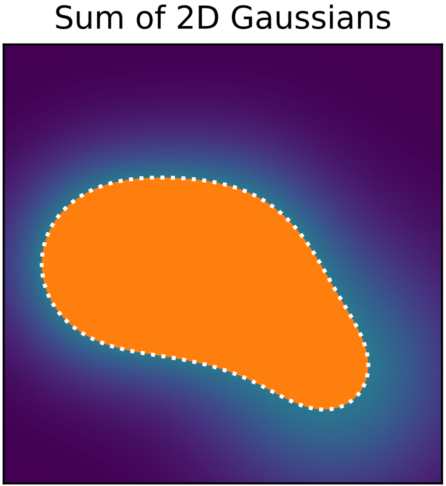
Plotting in 2D: Contour

```python
import numpy as np
import matplotlib.pyplot as plt
# Generate 2D Gaussian data
x = np.linspace(-4, 4, 400, endpoint=True)
X = x[:, np.newaxis] + np.zeros((1, 400))
Y = X.T
z0 = common.gauss_2d(X, Y, np.array([0, 0]), 1.5 *
     np.array([[1,0],[0,1]]) )
z1 = common.gauss_2d(X, Y, np.array([2, 2]), 2.3 *
     np.array([[1,0],[0,1]]) )
z2 = common.gauss_2d(X, Y, np.array([0,-2]), 1.3 *
     np.array([[1,0],[0,1]]) )
z = z1 + z0 + z2
# Set two levels
zmin = np.amin(z)
zmax = np.amax(z)
r = zmax - zmin
levels = [zmin + 0.2 * r, zmin + 0.4 * r]
# Create figure and plot data
ax = plt.figure(figsize=(3,3)).gca()
ax.imshow(z)
ax.contour(z, levels=levels, colors='white',
     linestyles=':')
ax.set_title('Sum of 2D Gaussians')
```



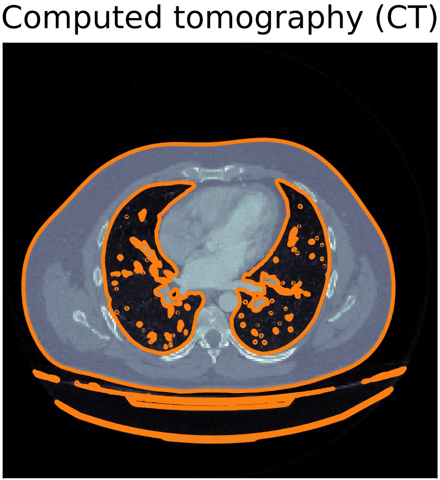
Plotting in 2D: Contour (filled)

```python
import numpy as np
import matplotlib.pyplot as plt
# Generate 2D Gaussian data
x = np.linspace(-4, 4, 400, endpoint=True)
X = x[:, np.newaxis] + np.zeros((1, 400))
Y = X.T
z0 = common.gauss_2d(X, Y, np.array([0, 0]), 1.5 *
     np.array([[1,0],[0,1]]) )
z1 = common.gauss_2d(X, Y, np.array([2, 2]), 2.3 *
     np.array([[1,0],[0,1]]) )
z2 = common.gauss_2d(X, Y, np.array([0,-2]), 1.3 *
     np.array([[1,0],[0,1]]) )
z = z1 + z0 + z2
# Set two levels (fill between these)
zmin = np.amin(z)
zmax = np.amax(z)
r = zmax - zmin
levels = [zmin + 0.4 * r, zmax]
# Create figure and plot data
ax = plt.figure(figsize=(3,3)).gca()
ax.imshow(z, cmap='viridis')
ax.contourf(z, levels=levels, colors = ['tab:orange'
     ])
ax.contour(z, levels=[levels[0]], colors='white',
     linestyles=':')
ax.set_title('Sum of 2D Gaussians')
```



Plotting in 2D: Contour (image)

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

# Use the pydicom package to load DICOM image
import pydicom

filename = 'image_160.dcm'
directory = os.path.join('..', '..',
                         'data', '3Dircadb2_1')
filepath = os.path.join(directory, filename)
dcm = pydicom.dcmread(filepath)
img = dcm.pixel_array

# Create figure and plot image
ax = plt.figure(figsize=(3,3)).gca()
ax.imshow(img, cmap='bone', interpolation='nearest')
ax.contour(img, levels=[-400], colors=['tab:orange'], linestyles=['-'])
ax.set_title('Computed tomography (CT)')
```

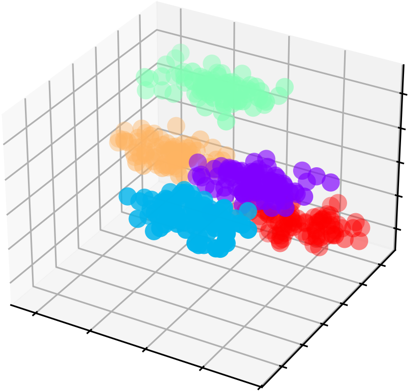


Plotting in 3D

Plotting in 3D

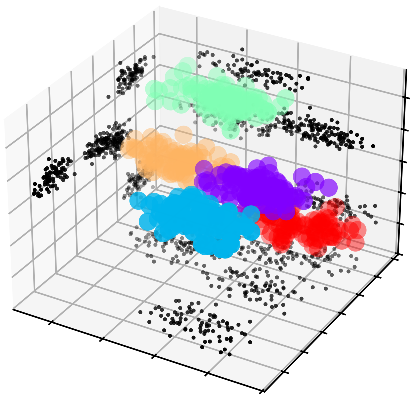
Plotting in 3D: Scatter 3D

```python
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
# Create blobby data
n_clusters = 5
X, y = make_blobs(n_samples=400,
                  n_features=3,
                  centers=n_clusters,
                  cluster_std=1.0,
                  random_state=0)
# Create figure and draw blobby data
fig, ax = plt.subplots(figsize=(4, 4),
                       subplot_kw={"projection": "3d"})
cmap = cmap=plt.get_cmap('rainbow', n_clusters)
ax.scatter3D(X[:, 0],
             X[:, 1],
             X[:, 2],
             s=8**2,
             marker='o',
             c = y,
             cmap = cmap)
#plt.show()
```



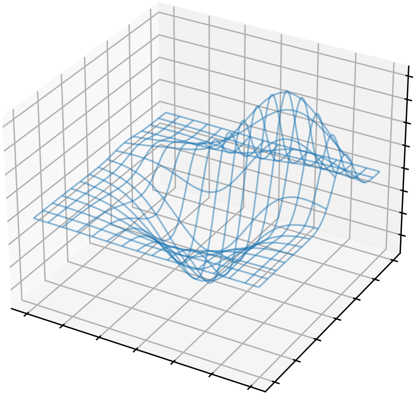
Plotting in 3D: Scatter 3D (Drop shadows)

```python
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
# Create blobby data
n_clusters = 5
X, y = make_blobs(n_samples=400,
                  n_features=3,
                  centers=n_clusters,
                  cluster_std=1.0,
                  random_state=0)
# Create figure and draw blobby data
fig, ax = plt.subplots(figsize=(4, 4),
                       subplot_kw={"projection": "3d"})
cmap = cmap=plt.get_cmap('rainbow', n_clusters)
ax.scatter3D(X[:, 0], X[:, 1], X[:, 2],
             s=8**2, marker='o',
             c = y, cmap = cmap)
# Draw drop shadows
xlim = ax.get_xlim()
ylim = ax.get_ylim()
zlim = ax.get_zlim()
s = 1**2
ax.scatter3D(xlim[0], X[:, 1], X[:, 2],
             s=s, marker='o', facecolor='k')
ax.scatter3D(X[:, 0], ylim[1], X[:, 2],
             s=s, marker='o', facecolor='k')
ax.scatter3D(X[:, 0], X[:, 1], zlim[0],
             s=s, marker='o', facecolor='k')
```



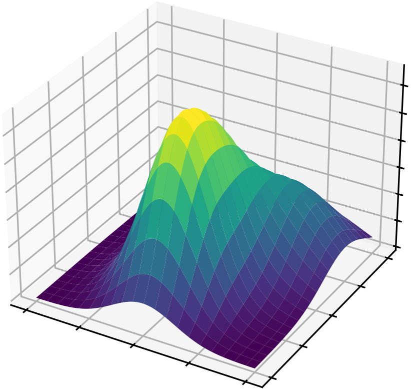
Plotting in 3D: Wireframe

```
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import axes3d

ax = plt.figure().add_subplot(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)

# Plot the 3D surface
ax.plot_wireframe(X, Y, Z,
                  edgecolor='tab:blue',
                  lw=1.0,
                  rstride=8,
                  cstride=8,
                  alpha=0.5)
```



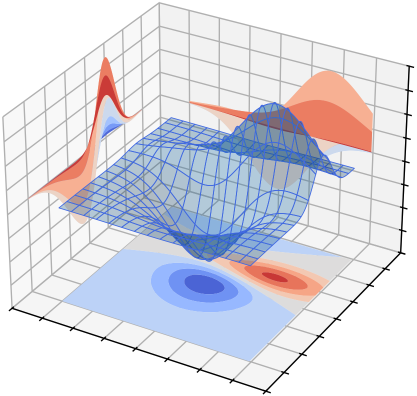
Plotting in 3D: Surface

```python
import numpy as np
import matplotlib.pyplot as plt
N = 100
stride = N//20
x = np.linspace(-4, 4, 100, endpoint=True)
X = x[:, np.newaxis] + np.zeros((1, 100))
Y = X.T
z0 = common.gauss_2d(X, Y, np.array([0, 0]), 1.5 *
     np.array([[1,0],[0,1]]) )
z1 = common.gauss_2d(X, Y, np.array([2, 2]), 2.3 *
     np.array([[1,0],[0,1]]) )
z2 = common.gauss_2d(X, Y, np.array([0,-2]), 1.3 *
     np.array([[1,0],[0,1]]) )
z = z1 + z0 + z2
zmin = np.amin(z)
zmax = np.amax(z)
r = zmax - zmin
levels = [zmin + 0.4 * r, zmax]
# Create figure and plot data
fig, ax = plt.subplots(figsize=(4, 4),
                       subplot_kw={"projection": "3d
     "})
ax.plot_surface(X, Y, z, cmap='viridis',
                edgecolor=None, lw=0.0,
                rstride=stride, cstride=stride)
```



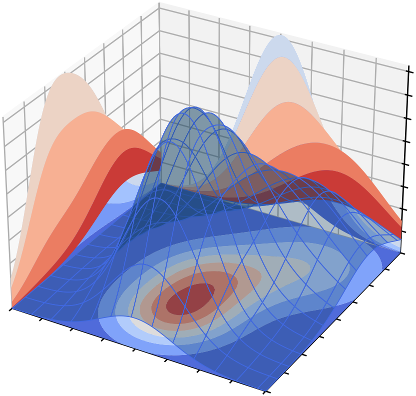
Plotting in 3D: Combined (1)

```
# https://matplotlib.org/stable/gallery/mplot3d/contourf3d_2.html
import matplotlib.pyplot as plt
ax = plt.figure().add_subplot(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
# Plot the 3D surface
ax.plot_surface(X, Y, Z,
                edgecolor='royalblue', lw=0.5,
                rstride=8, cstride=8, alpha=0.3)
# Plot projections of the contours for each dimension.
# Offsets to match axes limits -> contours sit on walls
ax.contourf(X, Y, Z, zdir='z', offset=-100, cmap='coolwarm')
ax.contourf(X, Y, Z, zdir='x', offset=-40, cmap='coolwarm')
ax.contourf(X, Y, Z, zdir='y', offset=40, cmap='coolwarm')
ax.set(xlim=(-40, 40),
       ylim=(-40, 40),
       zlim=(-100, 100))
```



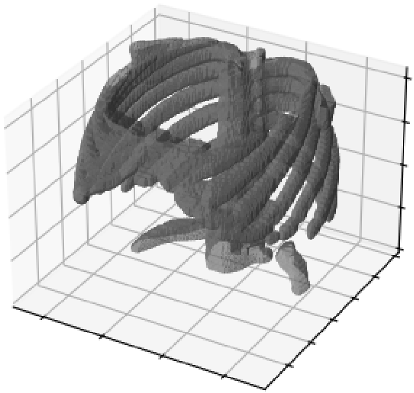
Plotting in 3D: Combined (2)

```python
# https://matplotlib.org/stable/gallery/mplot3d/contourf3d_2.html
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-4, 4, 100, endpoint=True)
X = x[:, np.newaxis] + np.zeros((1, 100))
Y = X.T
z0 = common.gauss_2d(X, Y, np.array([0, 0]), 1.5 * np.array([[1,0],[0,1]]) )
z1 = common.gauss_2d(X, Y, np.array([2, 2]), 2.3 * np.array([[1,0],[0,1]]) )
z2 = common.gauss_2d(X, Y, np.array([0,-2]), 1.3 * np.array([[1,0],[0,1]]) )
Z = z1 + z0 + z2
# Plot the 3D surface
ax = plt.figure().add_subplot(projection='3d')
ax.plot_surface(X, Y, Z,
                edgecolor='royalblue', lw=0.5,
                rstride=8, cstride=8, alpha=0.3)
# Plot projections of the contours for each dimension.
# Offsets to match axes limits -> contours sit on walls
ax.contour(X, Y, Z, zdir='z', offset=np.amin(Z), cmap='coolwarm')
ax.contourf(X, Y, Z, zdir='x', offset=-4, cmap='coolwarm')
ax.contourf(X, Y, Z, zdir='y', offset=4, cmap='coolwarm')
ax.set(xlim=(-4, 4),
       ylim=(-4, 4),
       zlim=(np.amin(Z), np.amax(Z)))
```



Plotting in 3D: Triangle meshes

```python
import os
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
import SimpleITK as sitk
from skimage import measure
from skimage.transform import rescale
# Bone mask DICOM images, rescale
directory = os.path.join('..', '..', 'data',
                         '3Dircadb2_1',
                         'bone-mask')
reader = sitk.ImageSeriesReader()
dicom_names = reader.GetGDCMSeriesFileNames(directory)
reader.SetFileNames(dicom_names)
img = reader.Execute()
arr = sitk.GetArrayFromImage(img)
arr = rescale(arr, 0.50, anti_aliasing=True)
# Marching cubes triangle mesh, flip x<->z
v, f, n, u = measure.marching_cubes(arr, 0)
v[:, [2, 0]] = v[:, [0, 2]]
# Plot, fancy indexing: 'verts[faces]' -> triangles
ax = plt.figure().add_subplot(projection='3d')
mesh = Poly3DCollection(v[f], shade=True, facecolors='tab:gray')
ax.add_collection3d(mesh)
```



Conclusion

Conclusion

Conclusion: Linear algebra and plotting

Data

Data can come from different sources and in different forms.

Vectors

Can represent data points as vectors.

Matrices

Matrices can be used to represent collections of data points or connections between them.

Plotting

Matplotlib has good functionality for 2D plotting and some for 3D plotting as well.