PFE 5

Dr Jacek Kałuży

Slajdy - Dr W Palubicki

Statistics

Why statistics?

- English believe 24% of their population are Muslim

Why statistics?

- English believe 24% of their population are Muslim → reality 5%

Why statistics?

- English believe 24% of their population are Muslim → reality 5%
- Saudis believe 28% of their population are overweight

Why statistics?

- English believe 24% of their population are Muslim → reality 5%
- Saudis believe 28% of their population are overweight → reality 71%

Why statistics?

- English believe 24% of their population are Muslim → reality 5%
- Saudis believe 28% of their population are overweight → reality 71%
- Japanese believe 56% of their population live in the countryside

Why statistics?

- English believe 24% of their population are Muslim → reality 5%
- Saudis believe 28% of their population are overweight → reality 71%
- Japanese believe 56% of their population live in the countryside → reality 7%

Source Ipsos MORI

Statistical inference

- Use data from a sampling measurement to infer information which is generally applicable

Example: drug testing

- 50 patients received new pain medication whereas a similar group of 50 patients were treated with older medication. Mean scores of perceived pain were 4.1 for the new medication and 4.5 for the old.

Example: drug testing

- Statistical inference addresses:
  - How can we estimate the difference of the effect of medication?
  - How can we quantify the precision of that estimate?

Statistical inference

- **Effect size**: the quantification of an effect, e.g. in the simplest case a single number
- **Confidence interval** and/or the **standard error**: the precision of the quantification (estimate)

numpy.mean

numpy.**mean**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*) [source]

Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. float64 intermediate and return values are used for integer inputs.

Examples

```
>>>
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([ 2.,  3.])
>>> np.mean(a, axis=1)
array([ 1.5,  3.5])
```

numpy.std Standard deviation

numpy.std(*a, axis=None, dtype=None, out=None, ddof=0, keepdims=False*) [source]

Compute the standard deviation along the specified axis.

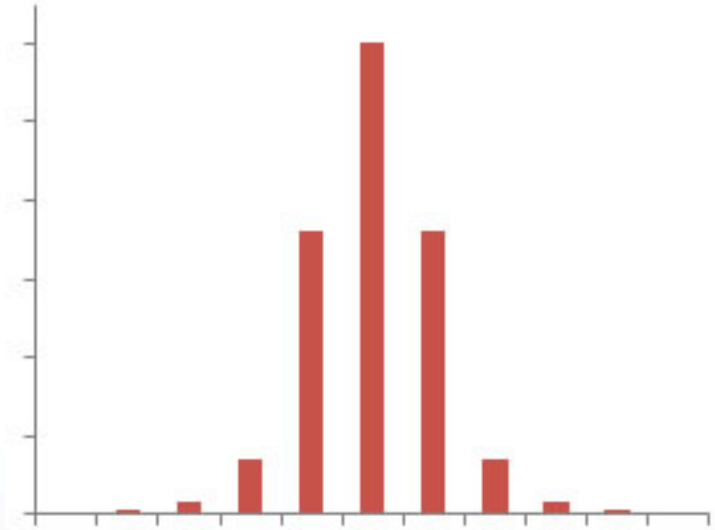
Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Examples

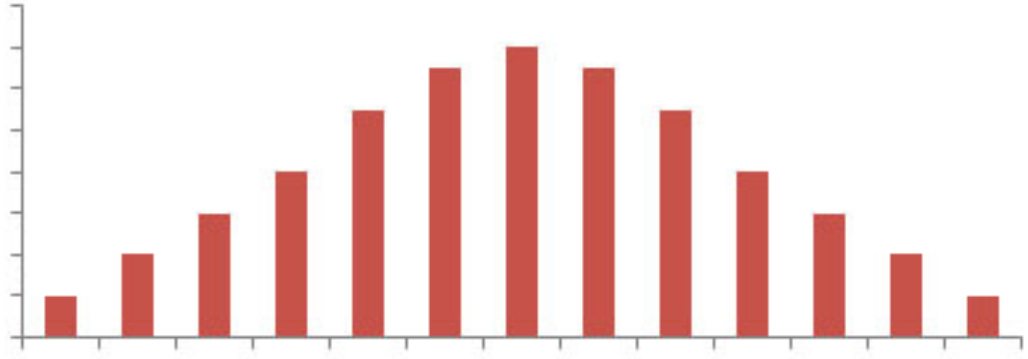
```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([ 1.,  1.])
>>> np.std(a, axis=1)
array([ 0.5,  0.5])
```

$$\sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2, \text{ std} = \sigma$$

Assessing variance with standard deviations



Histogram of distribution with small SD



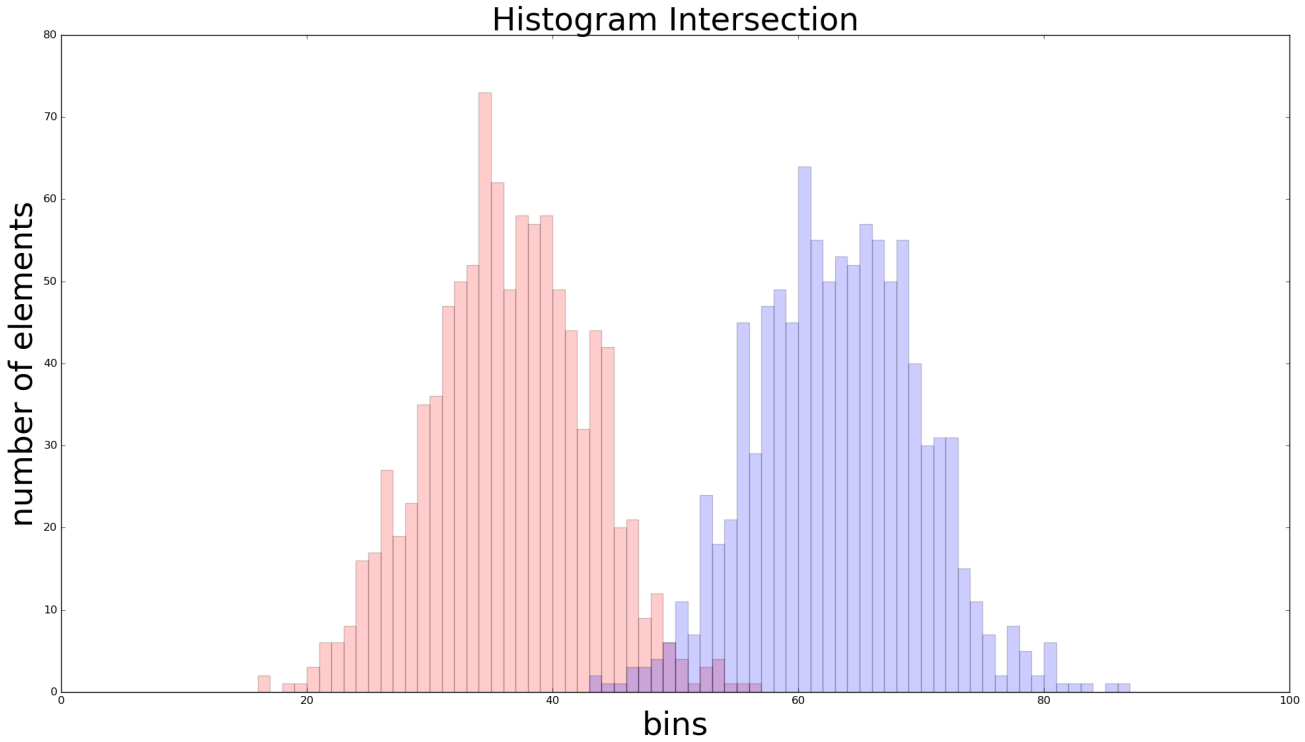
Histogram of distribution with large SD

Comparing distributions

- Difference of means: e.g. mu1 – mu2

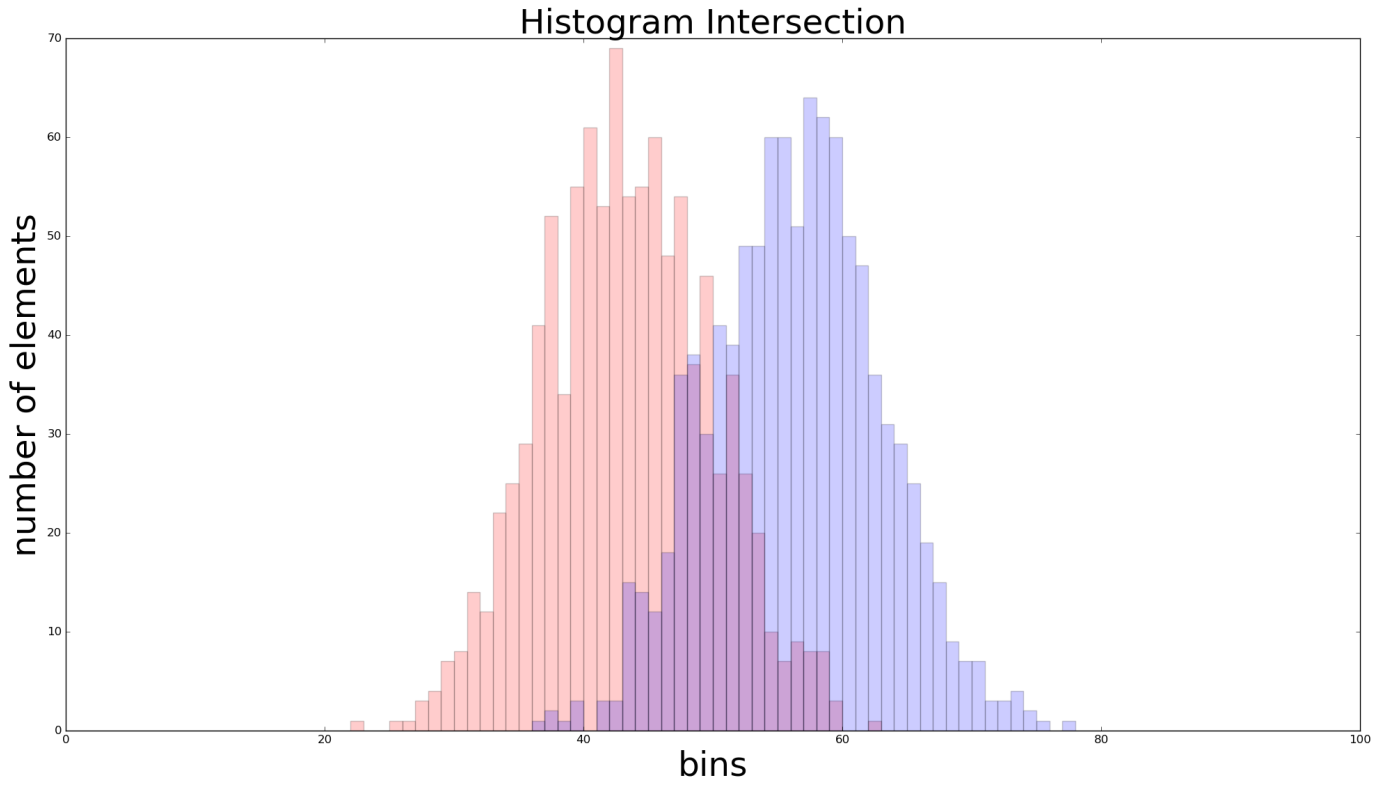
Comparing distributions

- Overlap of distributions



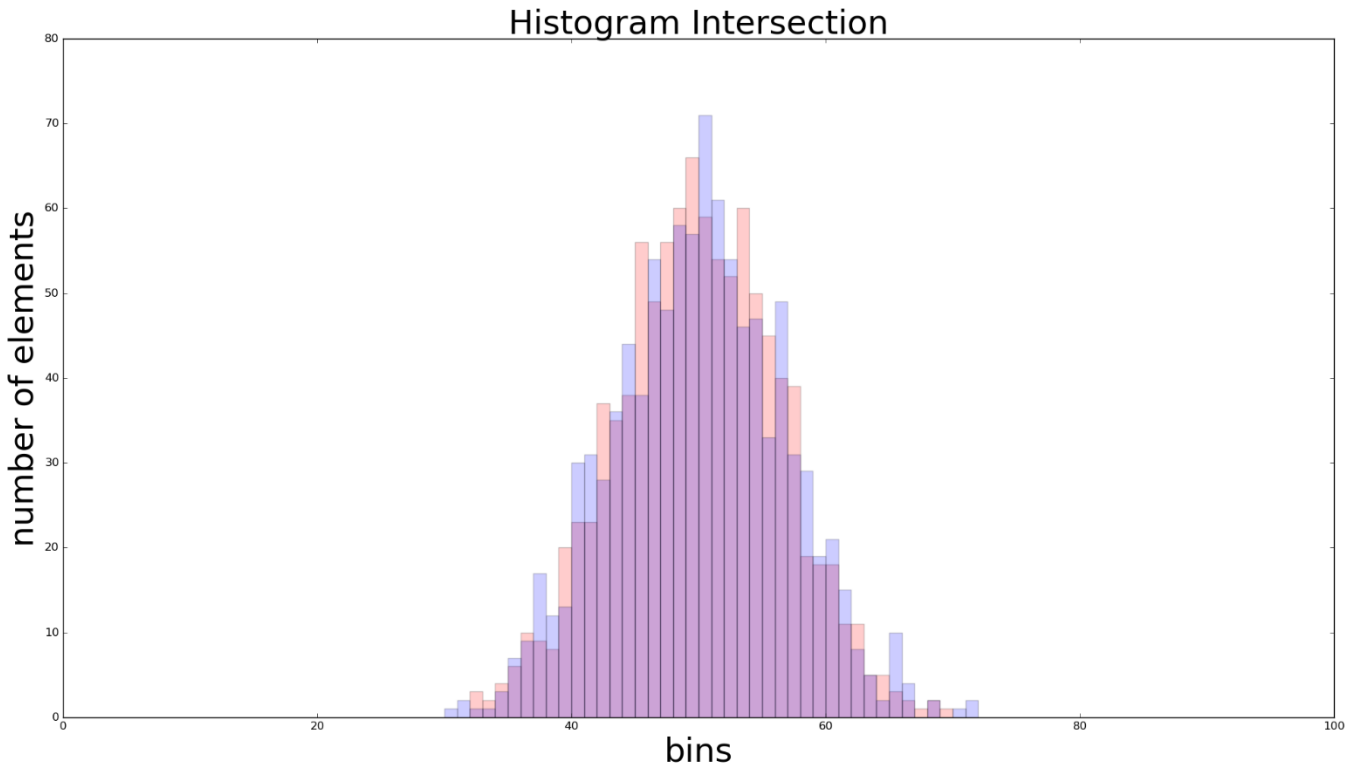
Comparing distributions

- Overlap of distributions



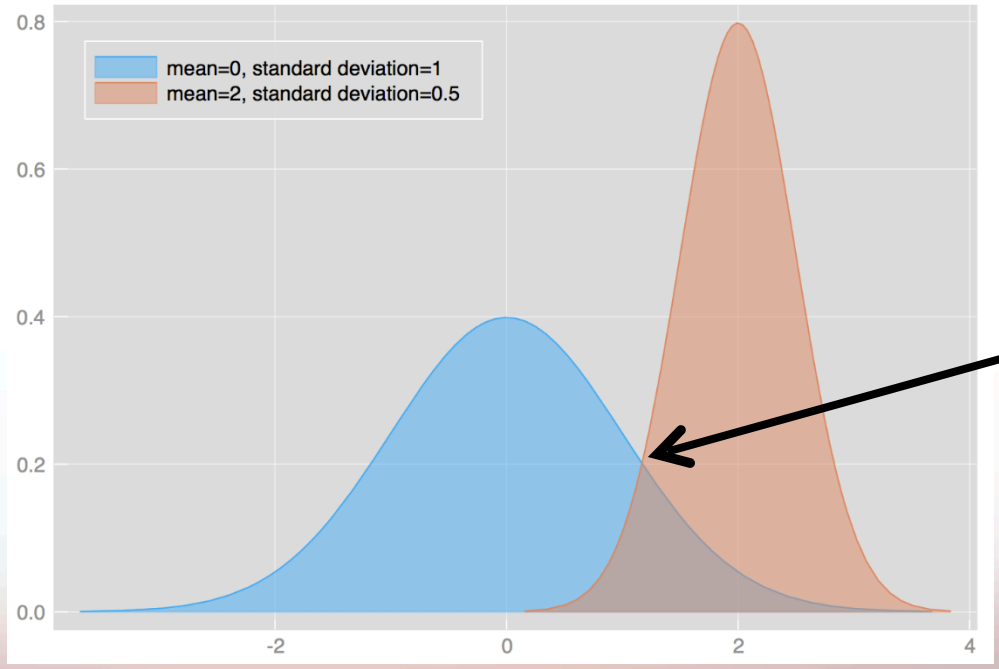
Comparing distributions

- Overlap of distributions



Computing overlap

- Define a threshold between the means of the distributions:
  - `thres = (std1 * mu2 + std2 * mu1) / (std1 + std2)`



threshold

Computing overlap

- Compute number of data points below threshold:

```
sample1_below_thres= sum(sample1 < thres)
sample2_above_thres= sum(sample2 > thres)
```

```
misclassified...                    ...ap +
sample2_overlap) / 2
```

Computing overlap

- Compute number of data points below threshold:

```
sample1_below_thres= sum(sample1 < thres) e.g. 100
sample2_above_thres= sum(sample2 > thres) e.g. 200
```

```
misclassified...                                    ...ap +
sample2_overlap) / 2
```

Computing overlap

- Compute number of data points below threshold:

```
sample1_below_thres= sum(sample1 < thres) e.g. 100
sample2_above_thres= sum(sample2 > thres) e.g. 200

sample1_overlap = sample1_below_thres / len(sample1)
sample2_overlap = sample2_above_thres / len(sample2)

misclassified = (sample1_overlap +
sample2_overlap) / 2
```

Computing overlap

- Compute number of data points below threshold:

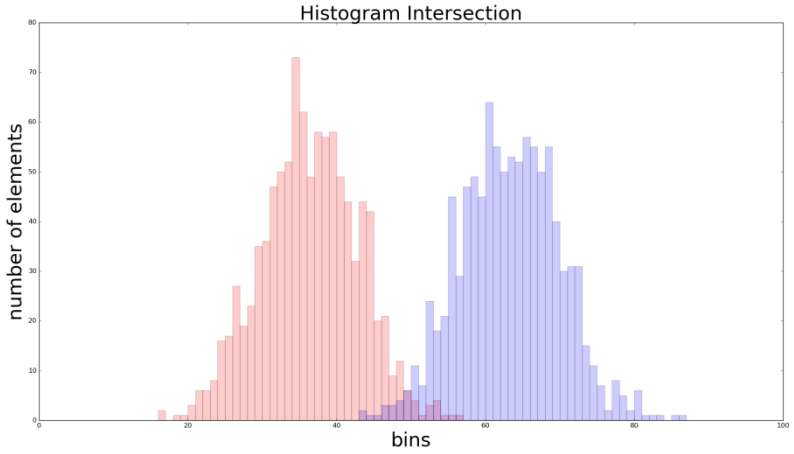
```
sample1_below_thres= sum(sample1 < thres) e.g. 100
sample2_above_thres= sum(sample2 > thres) e.g. 200

sample1_overlap = sample1_below_thres / len(sample1)
sample2_overlap = sample2_above_thres / len(sample2)
e.g. 0.2 and 0.3 (20% and 30%)
misclassification...                    ...ap +
sample2_overlap) / 2
```

Computing overlap

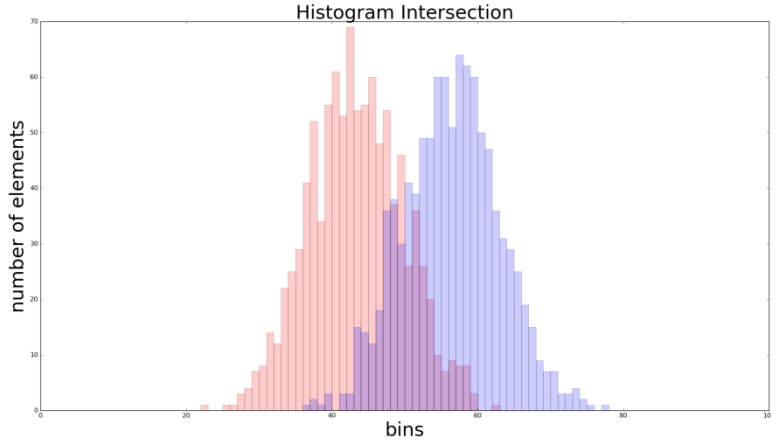
- Compute number of data points below threshold:

```
sample1_below_thres= sum(sample1 < thres)
sample2_above_thres= sum(sample2 > thres)

sample1_overlap = sample1_below_thres / len(sample1)
sample2_overlap = sample2_above_thres / len(sample2)

misclassification_rate = (sample1_overlap +
sample2_overlap) / 2
```

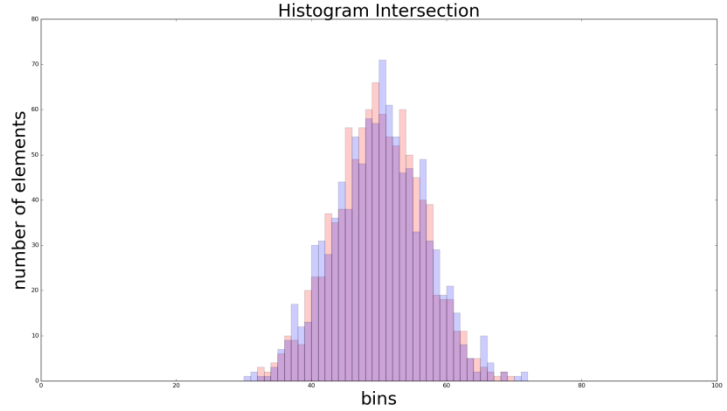
Example misclassification rates



~0.05



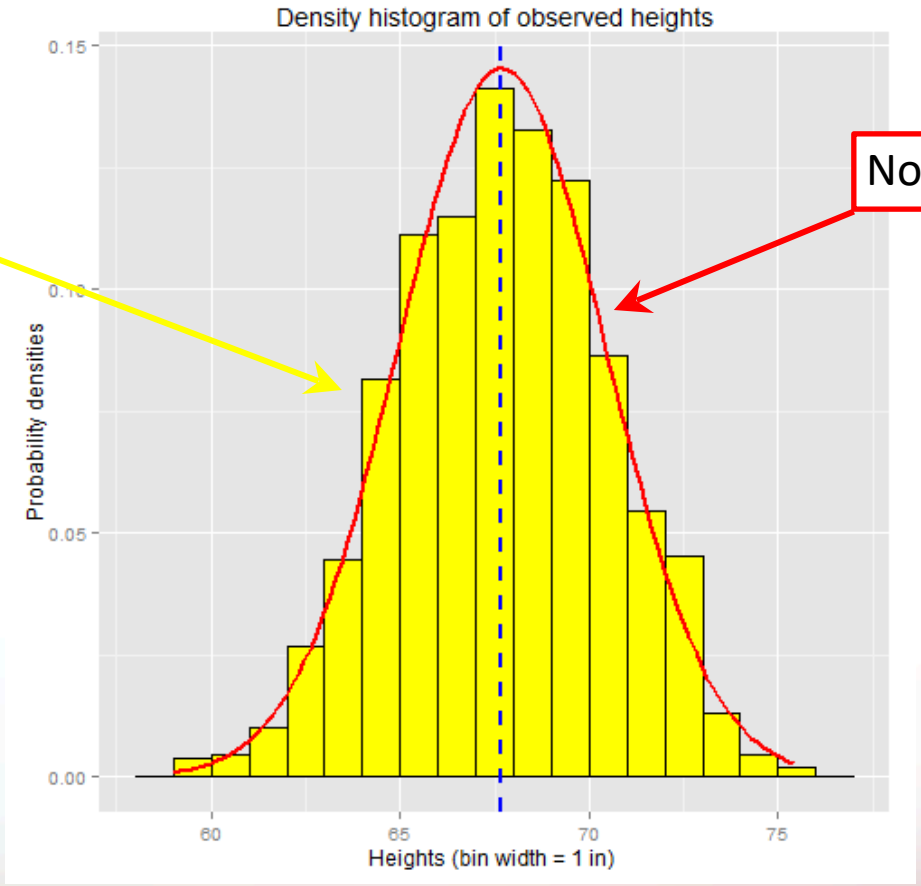
~0.2



~0.99

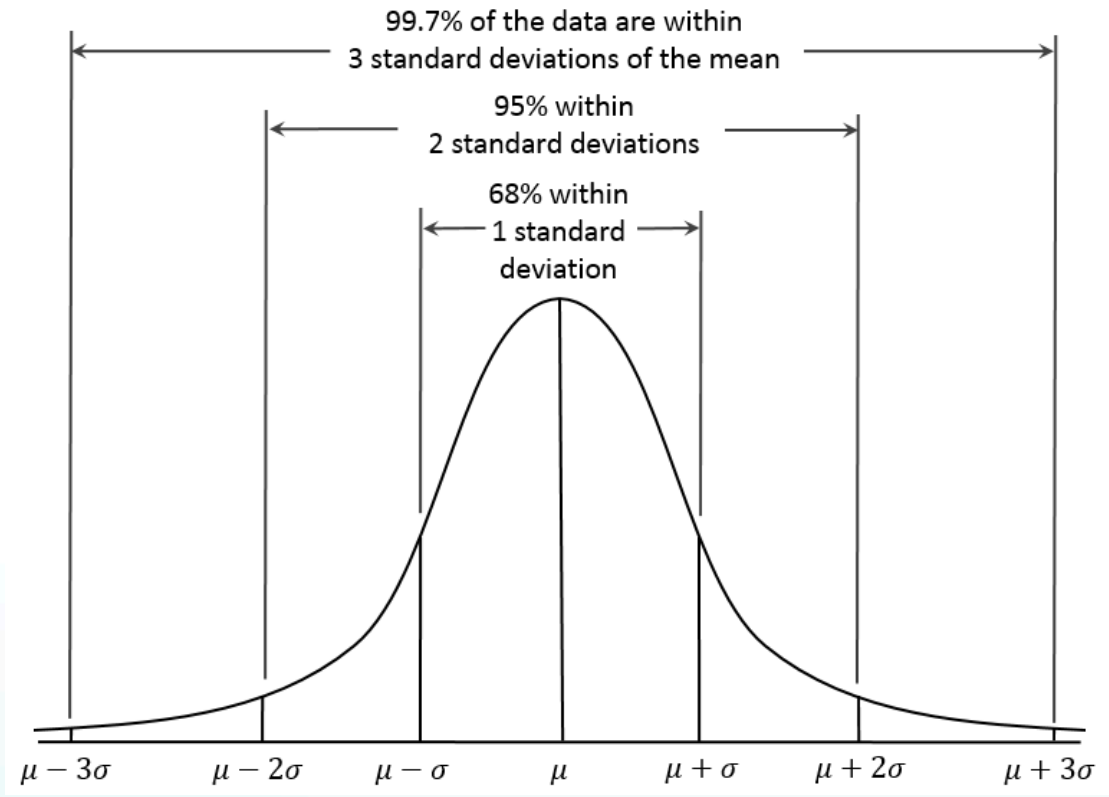
Histogram vs density distribution function

Histogram of data sample



Normal distribution function (mu, std)

Gaussian/Normal Distribution Function

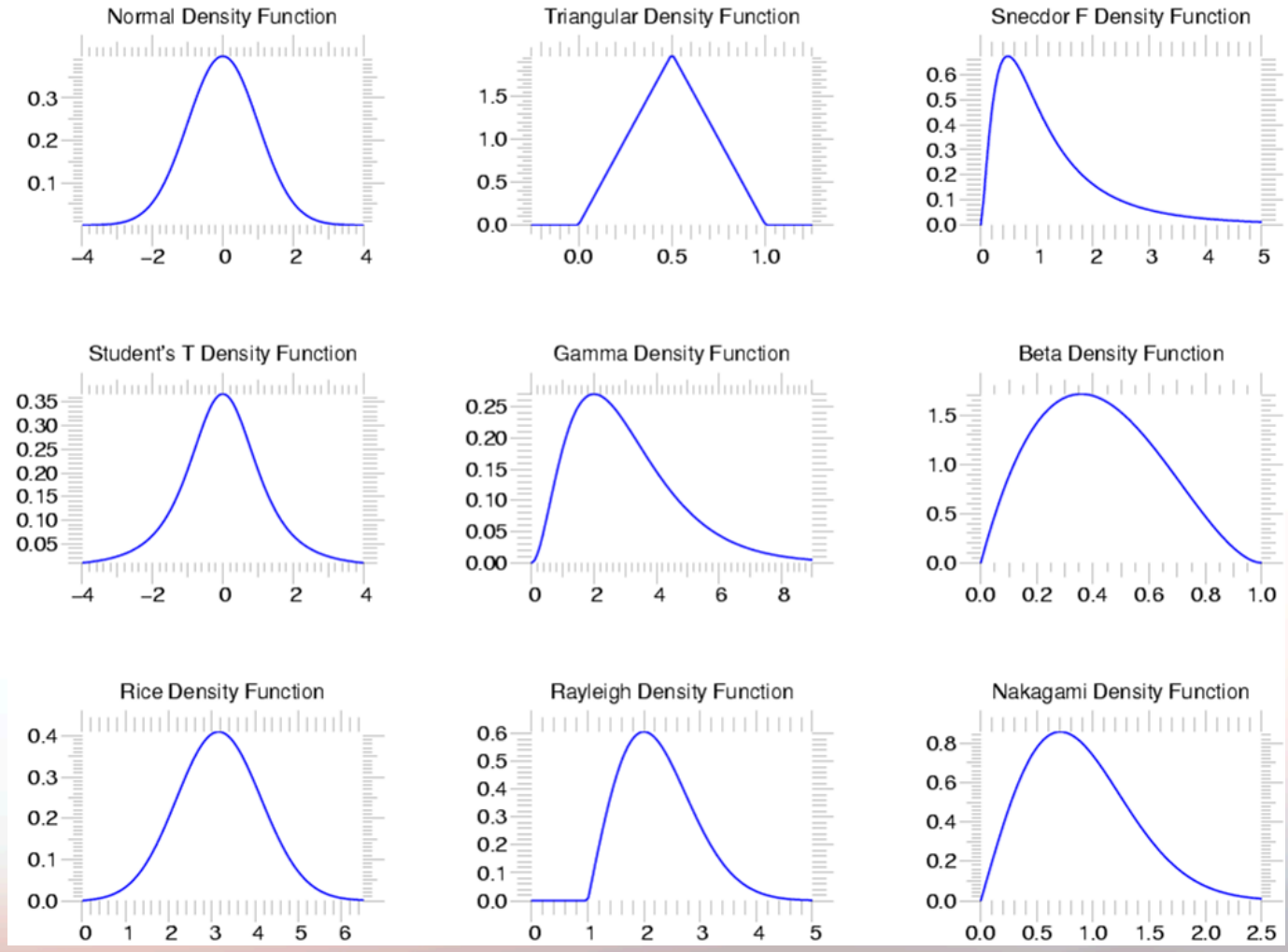


$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Statistics: scipy.stats

- Over 80 continuous distributions

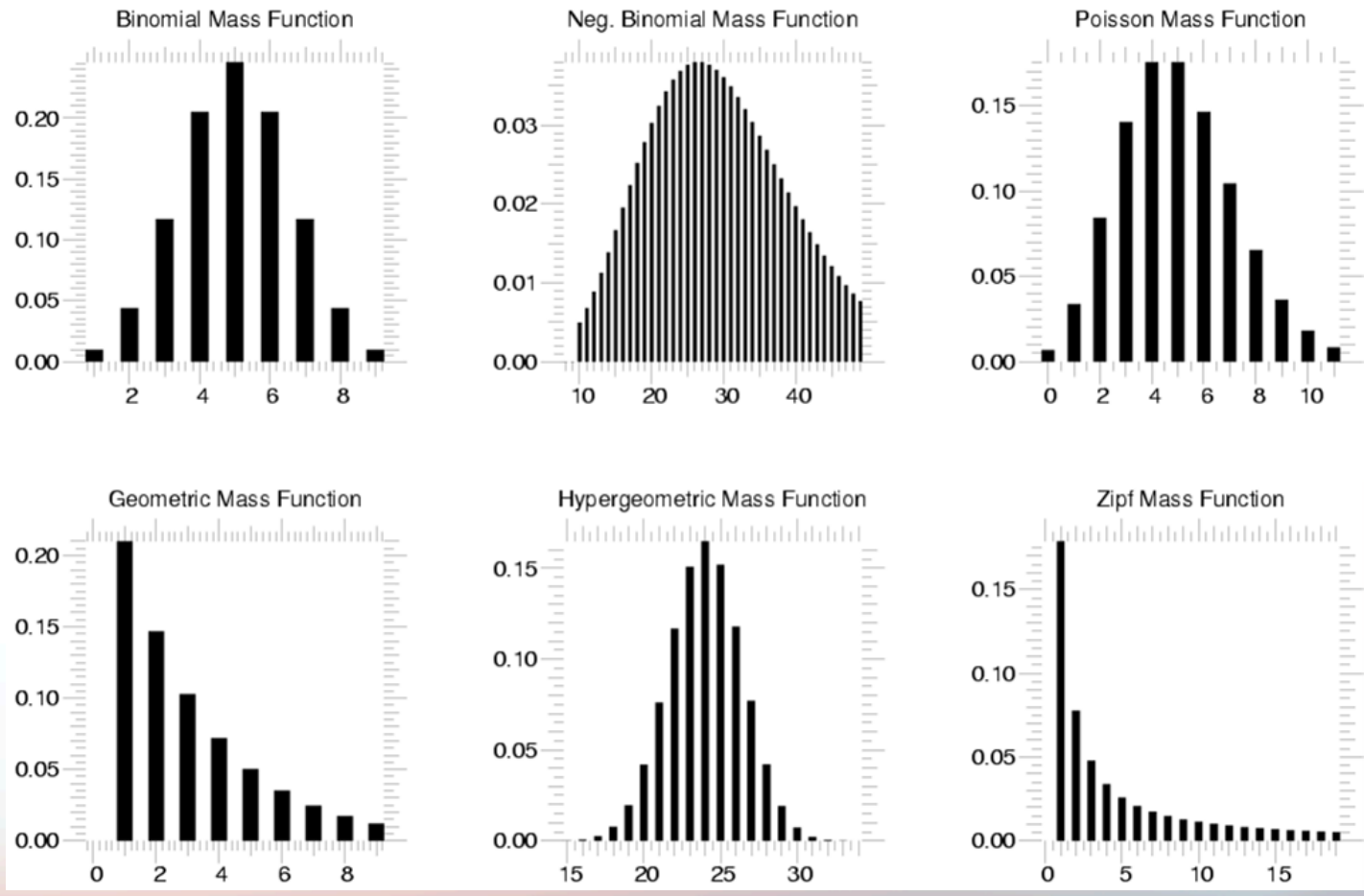
```
pdf
cdf
Rvs
ppf
fit
var
Mean
std
```



Statistics: scipy.stats

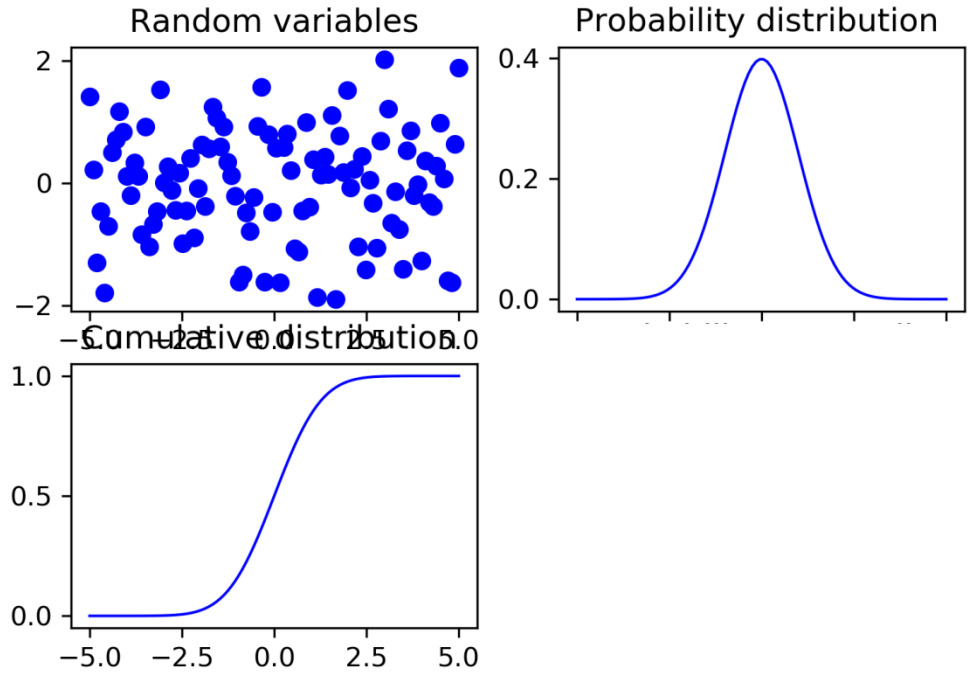
- 10 discrete distributions

```
pdf
cdf
Rvs
ppf
fit
var
Mean
std
```



Stats objects

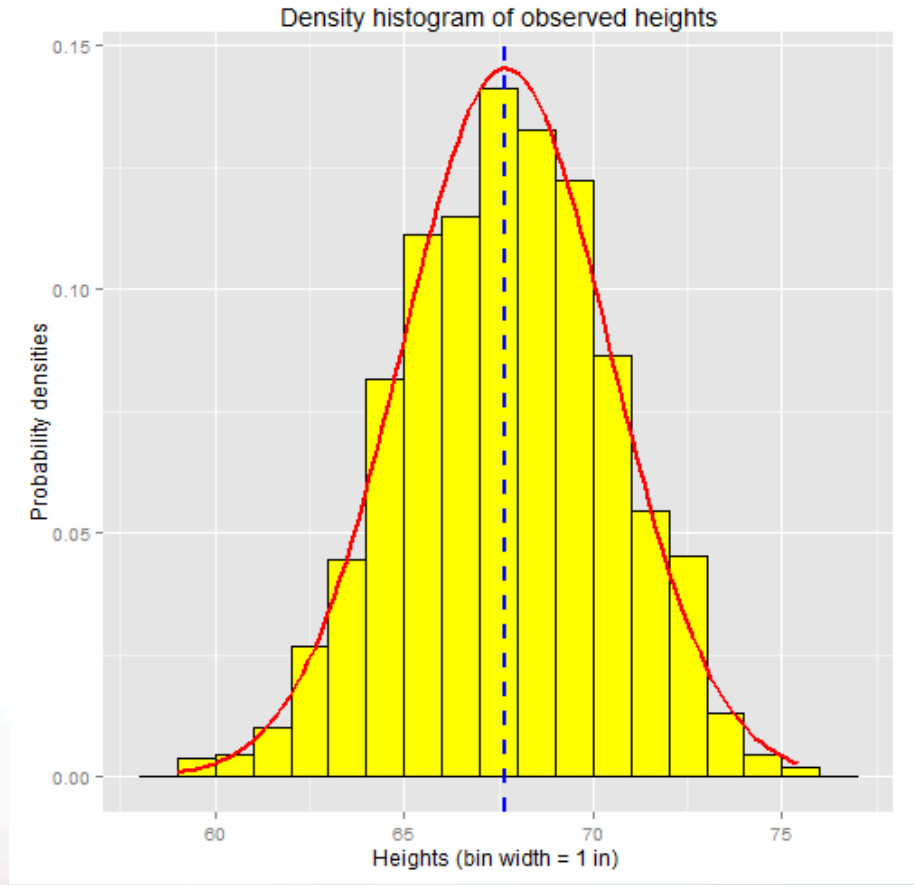
```
from scipy.stats import norm

samp = norm.rvs(size=100)

x = np.linspace(-5, 5, 100)

pdf = norm.pdf(x)

cdf = norm.cdf(x)
```



Stats objects

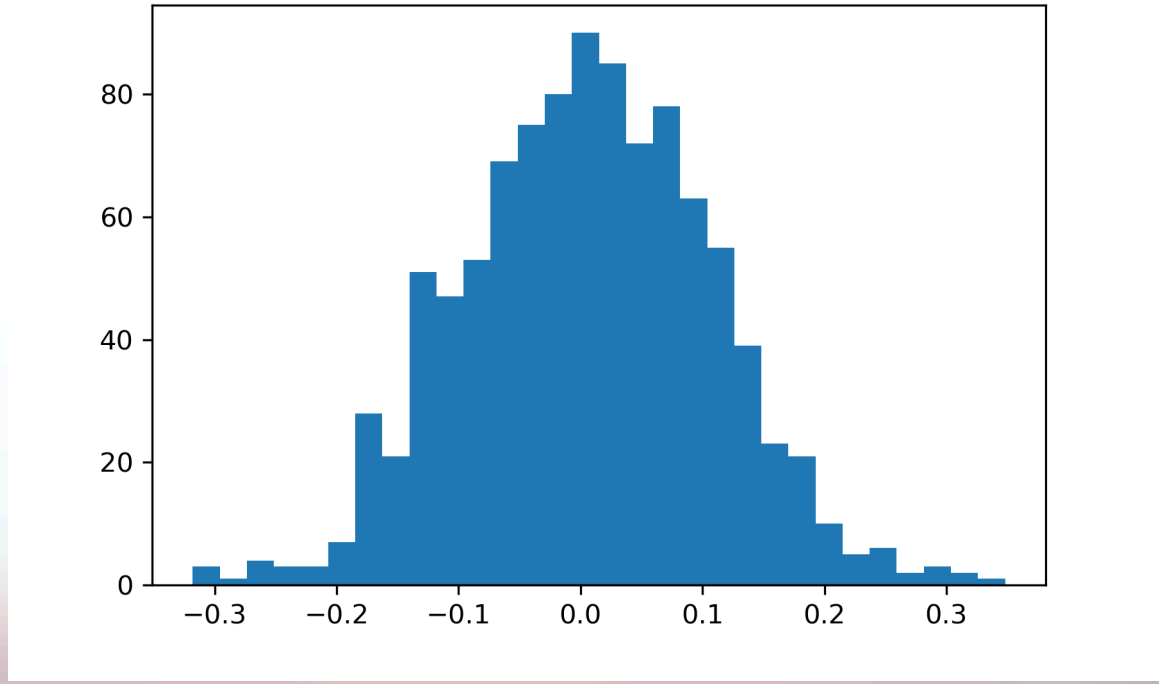
```
from scipy.stats import norm

samp = norm.rvs(size=100)

x = np.linspace(-5, 5, 100)

pdf = norm.pdf(x)

cdf = norm.cdf(x)
```

```
mu, sigma = norm.fit(samp)
-0.14, 1.01
```



Mean of sample means

Sample a 100 random variable array 1000 times and plot the **means** as a histogram

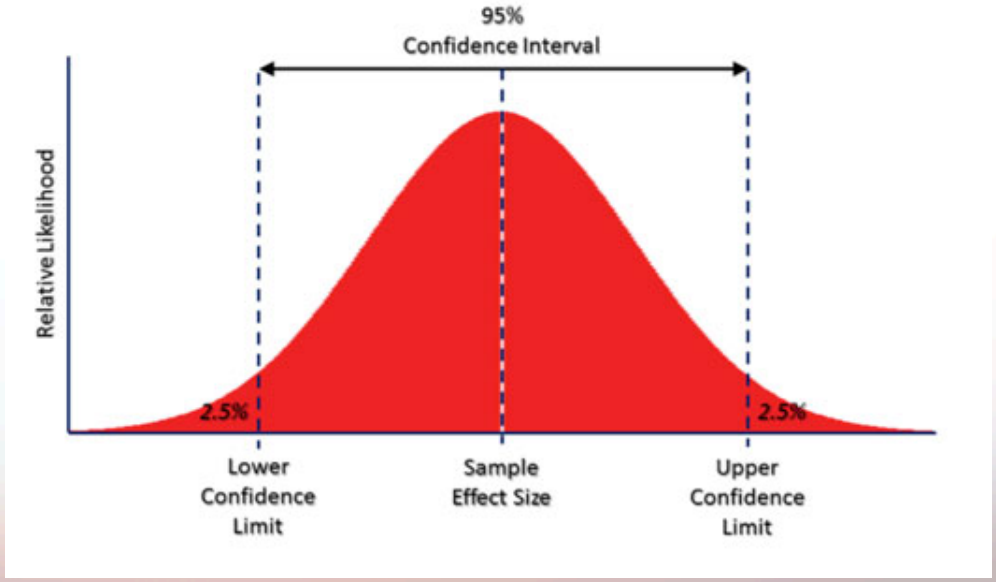


Confidence interval

np.percentile(sample, [2.5, 97.5])

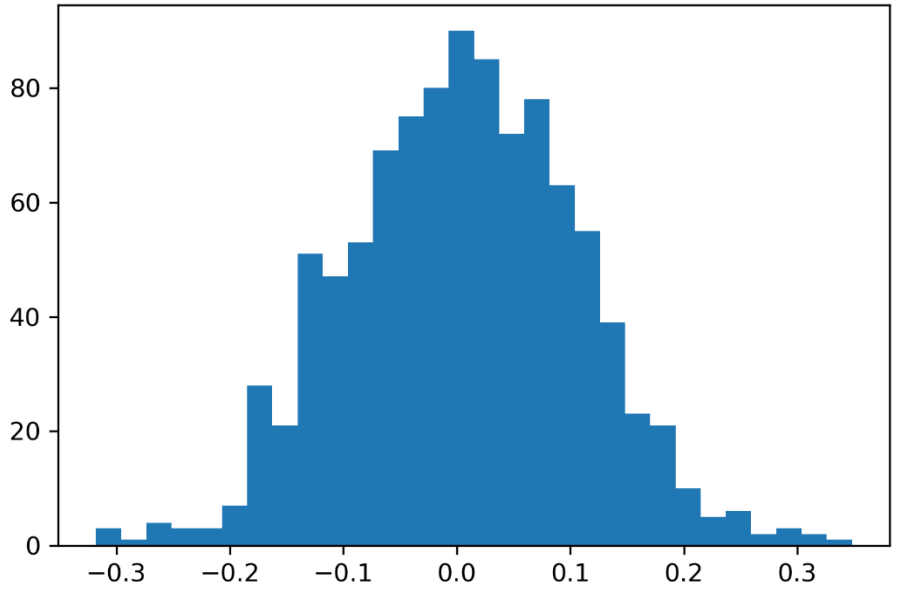
array([-0.0496105 , 0.05035856])

Lower Confidence Limit Upper Confidence Limit

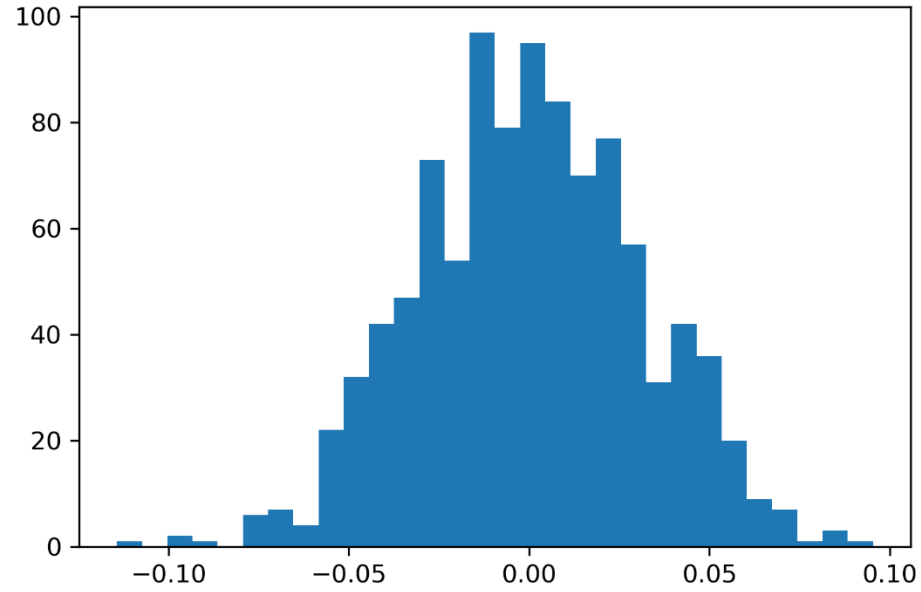


The sample mean std is the standard error

Sample a 1000 random variable array 1000 times and plot the means as a histogram: **standard deviation** ~0.03 (vs. ~0.1 for the 100 samples simulation)



100 samples

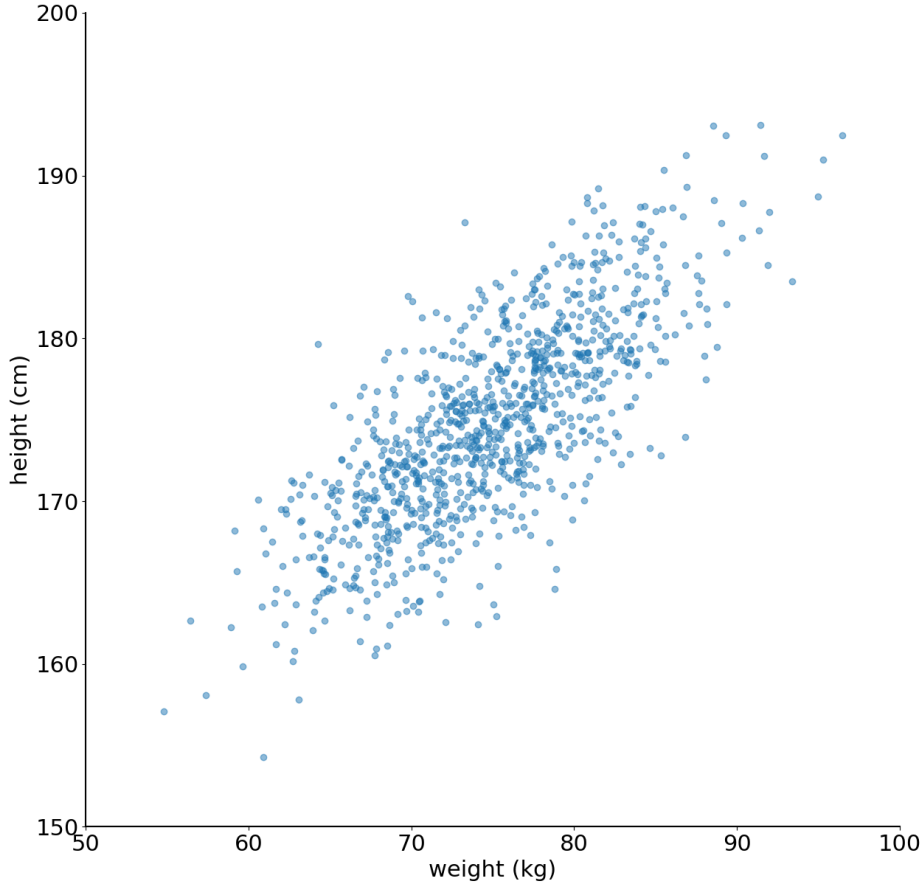


1000 samples

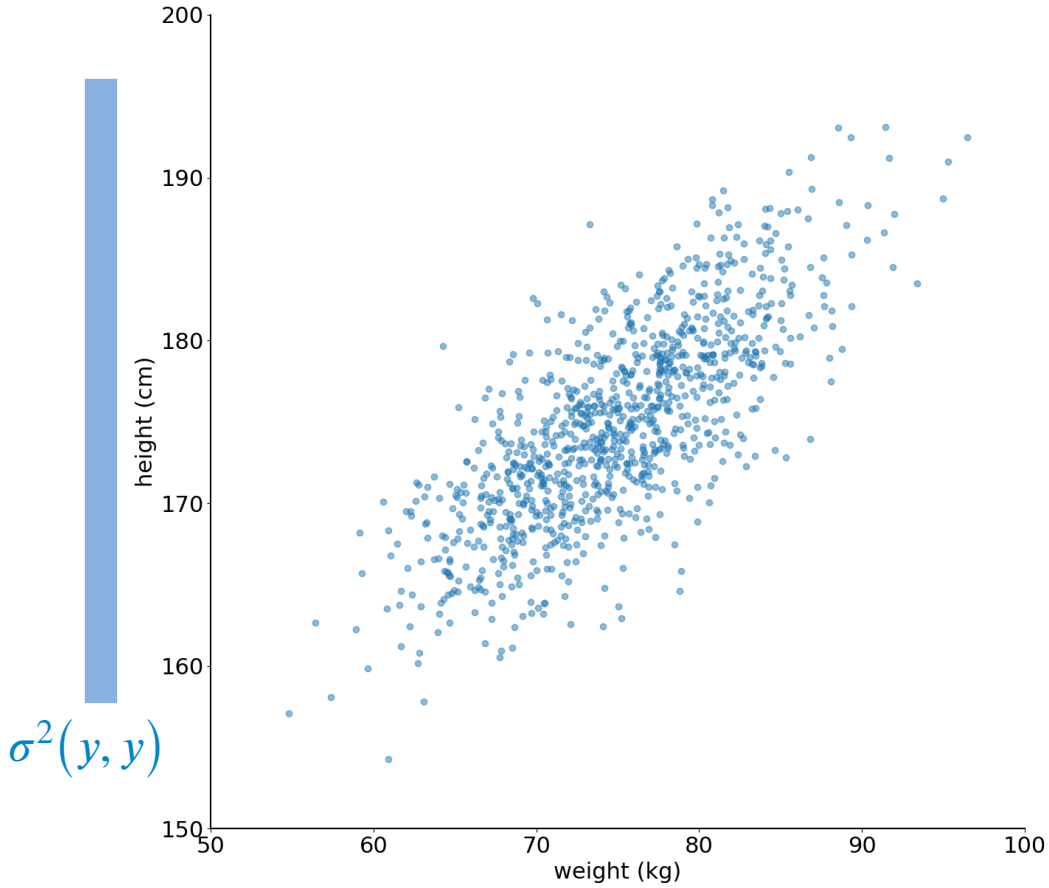
Statistical inference

- **Effect size**: means, standard deviations, overlaps
- **Precision:** Confidence interval and/or the standard error

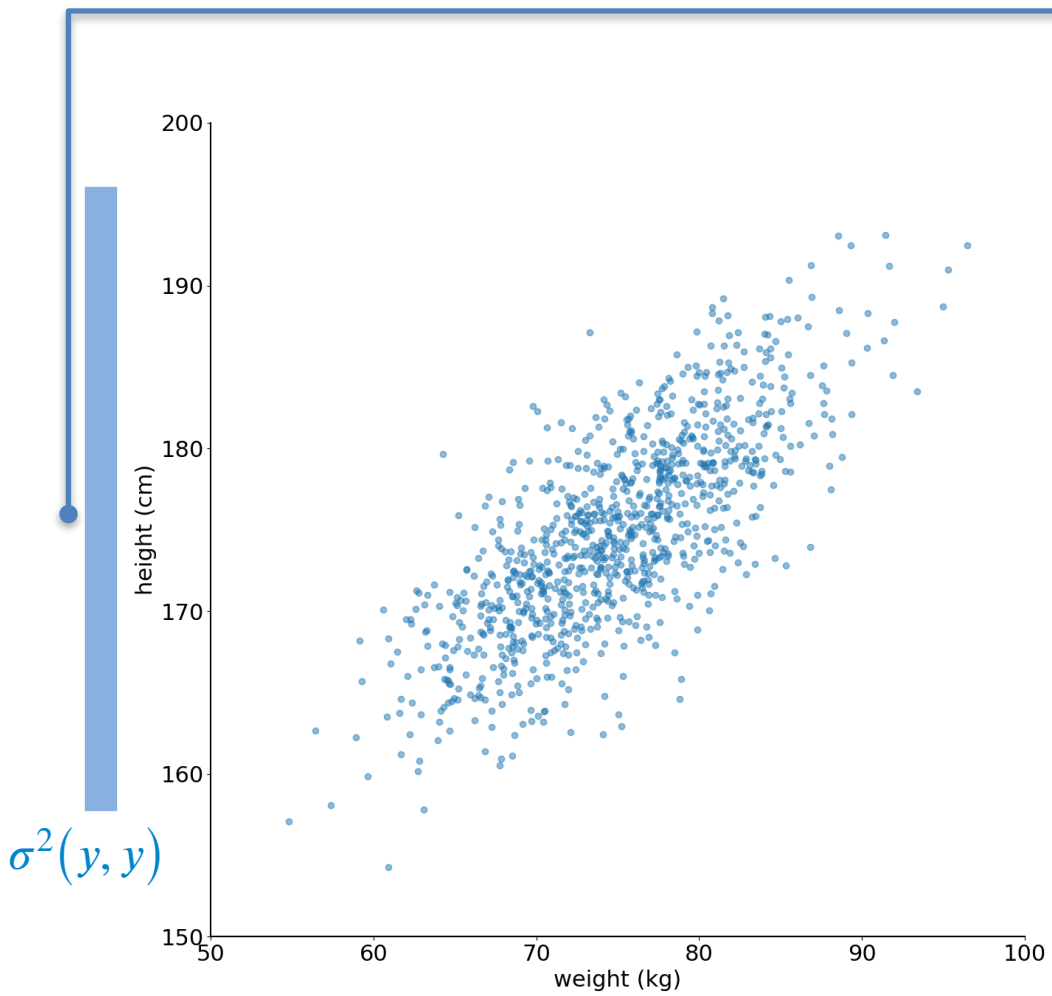
Multivariate data samples (covariance)?

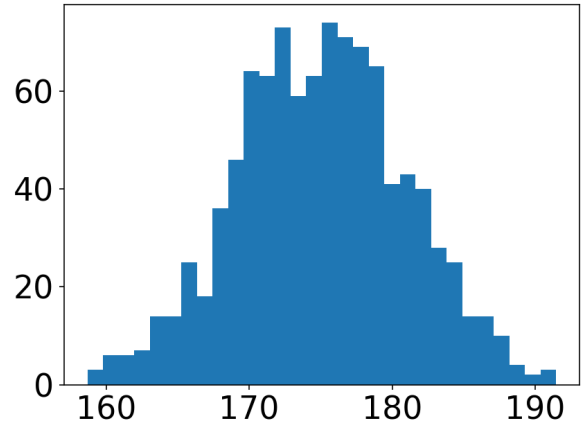


Multivariate data samples (covariance)?



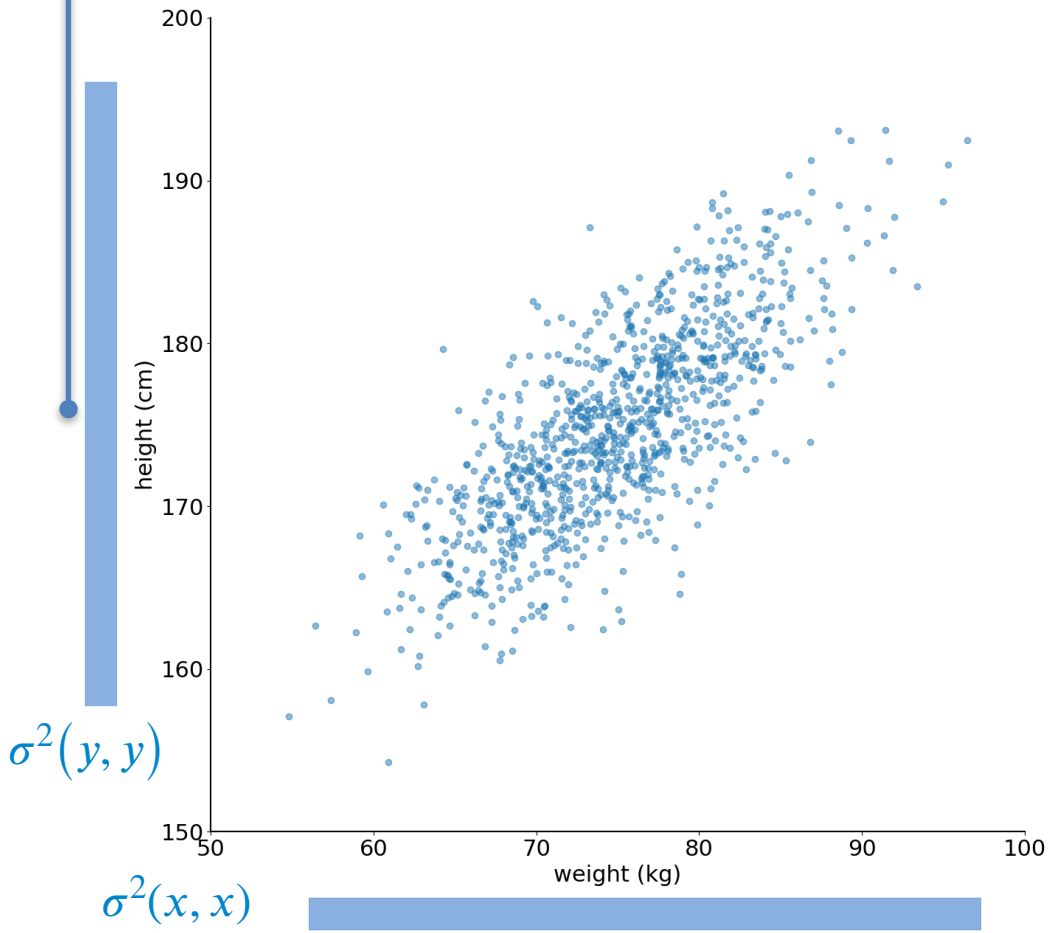
Multivariate data samples (covariance)?

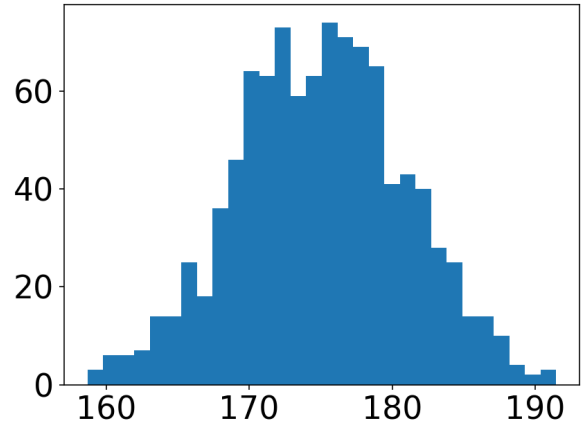




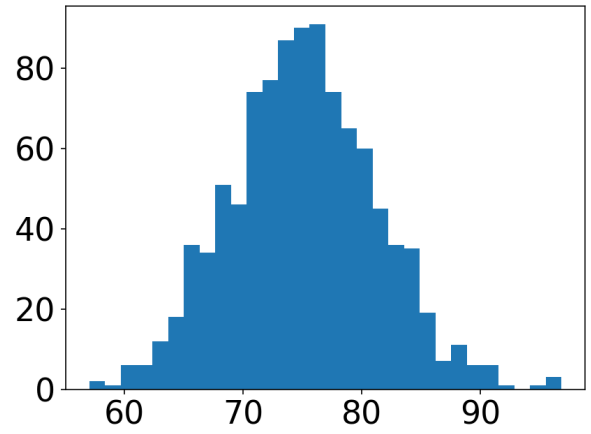
Variance in height = 36.9

Multivariate data samples (covariance)?



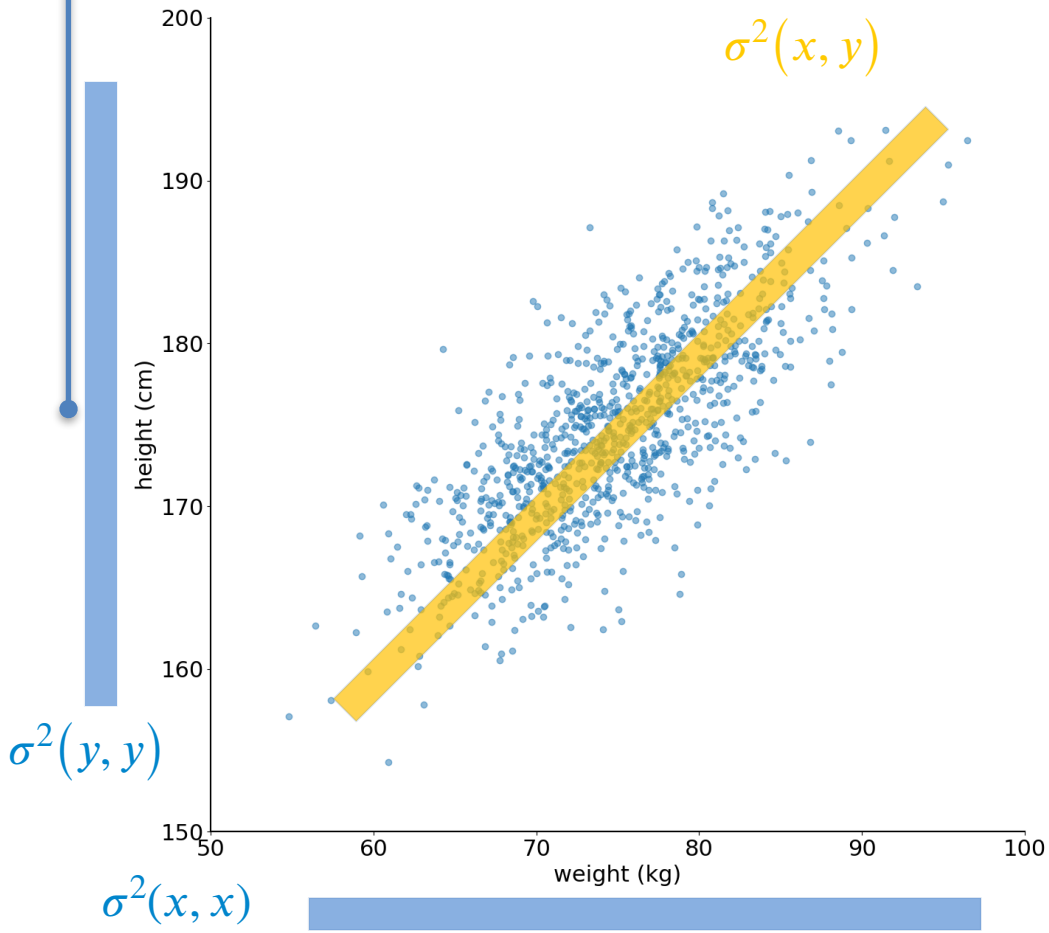


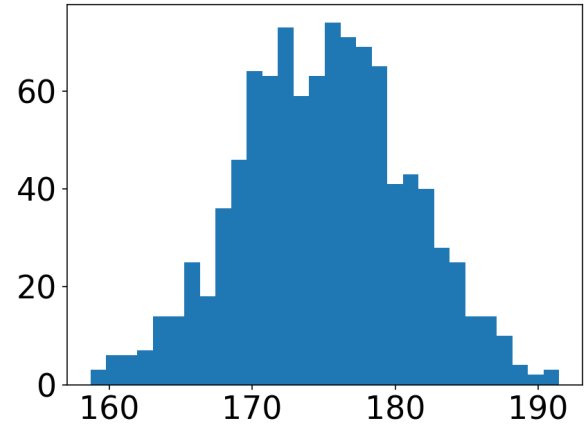
Variance in height = 36.9



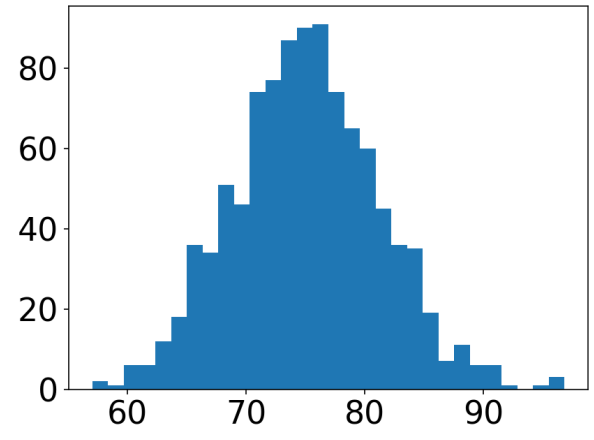
Variance in weight = 35.0

Multivariate data samples (covariance)?





Variance in height = 36.9



Variance in weight = 35.0

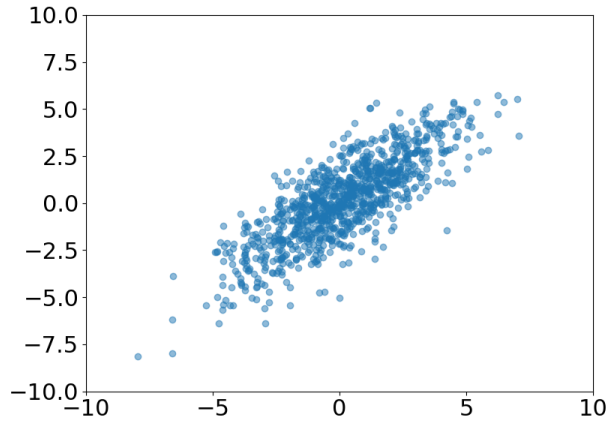
Covariance definition

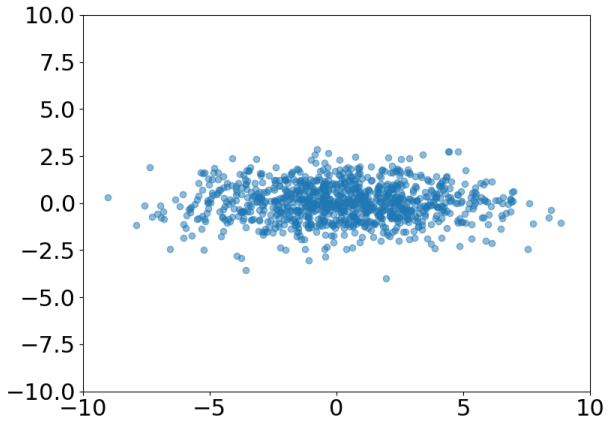
$$\sigma^2(x, y) = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu_x)(y_i - \mu_y)$$

$$\Sigma = \begin{bmatrix} \sigma^2(x, x) & \sigma^2(x, y) \\ \sigma^2(x, y) & \sigma^2(y, y) \end{bmatrix}$$

Example Covariance Matrices

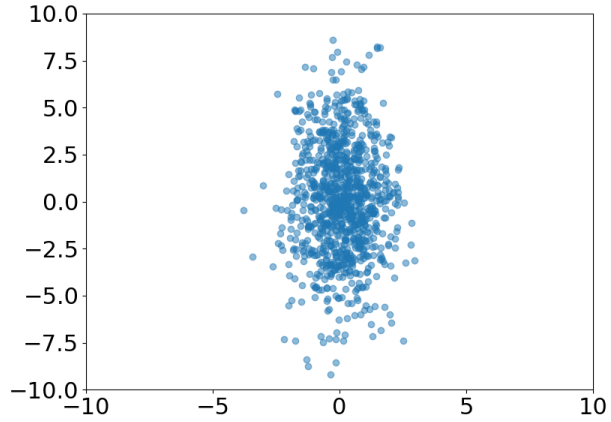
$$\Sigma = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}$$

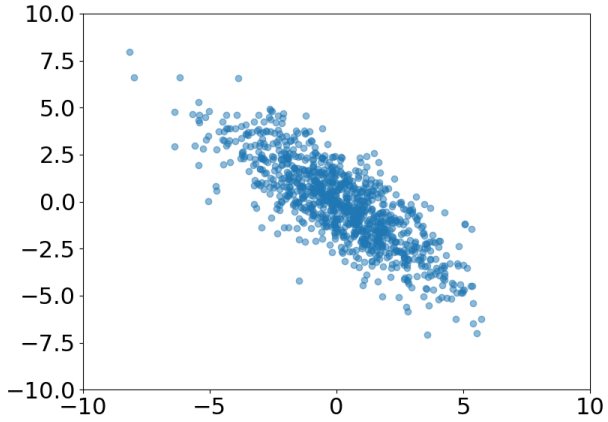




$$\Sigma = \begin{bmatrix} 9 & 0 \\ 0 & 1 \end{bmatrix}$$

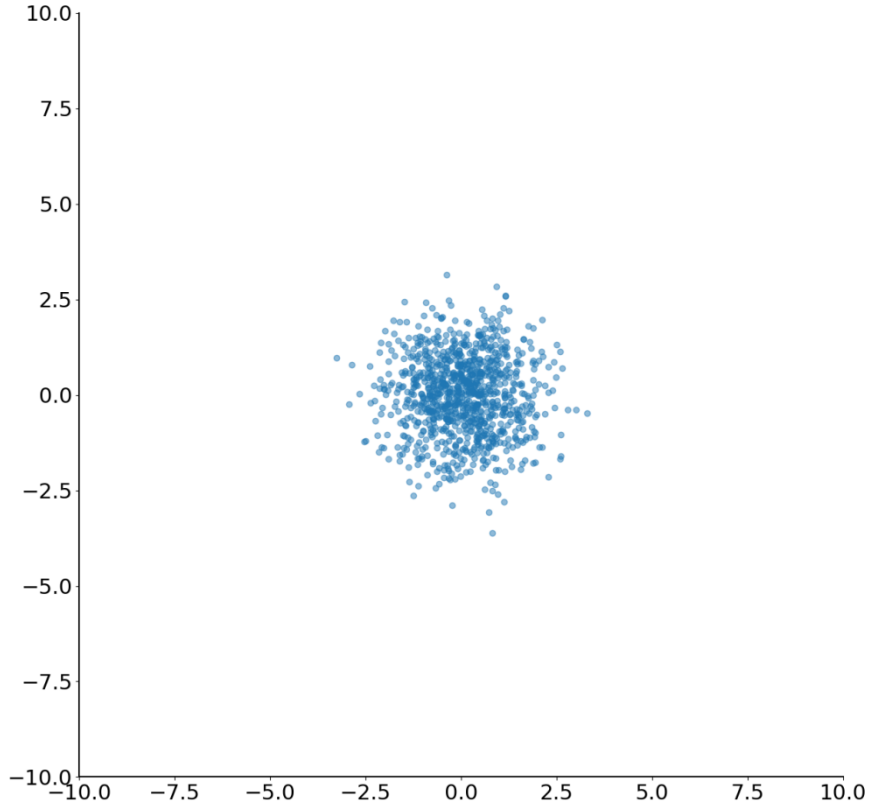
$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 9 \end{bmatrix}$$





$$\Sigma = \begin{bmatrix} 5 & -4 \\ -4 & 5 \end{bmatrix}$$

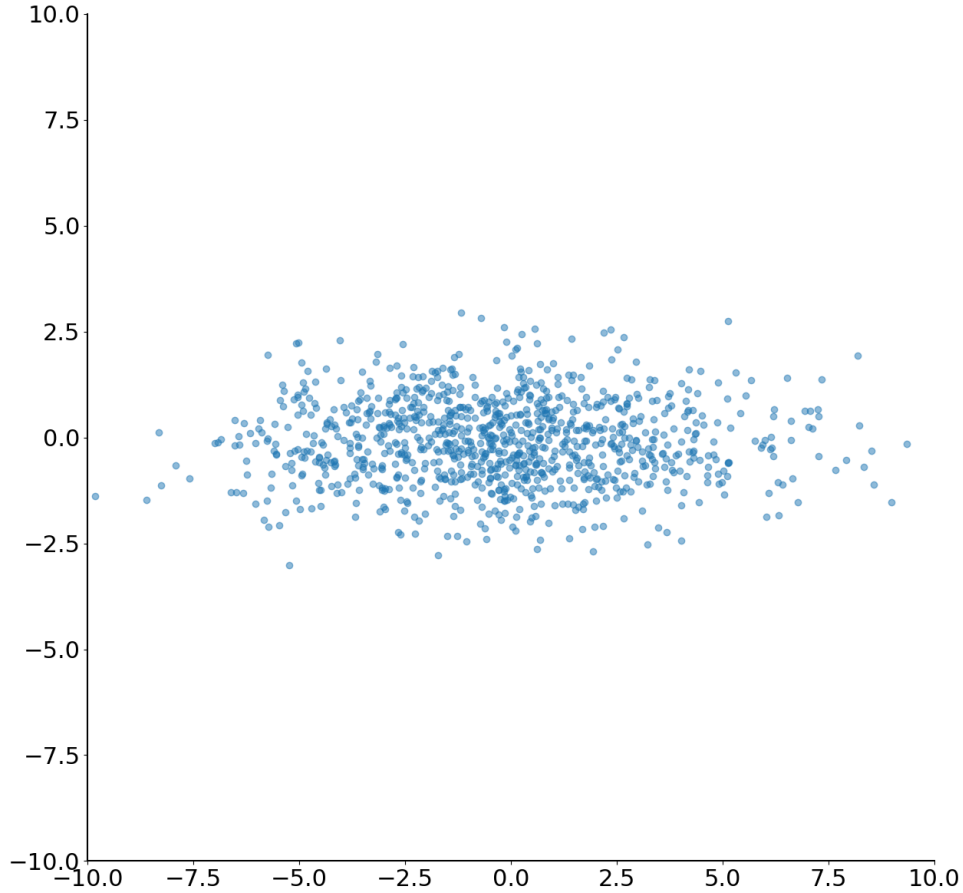
Normal distribution



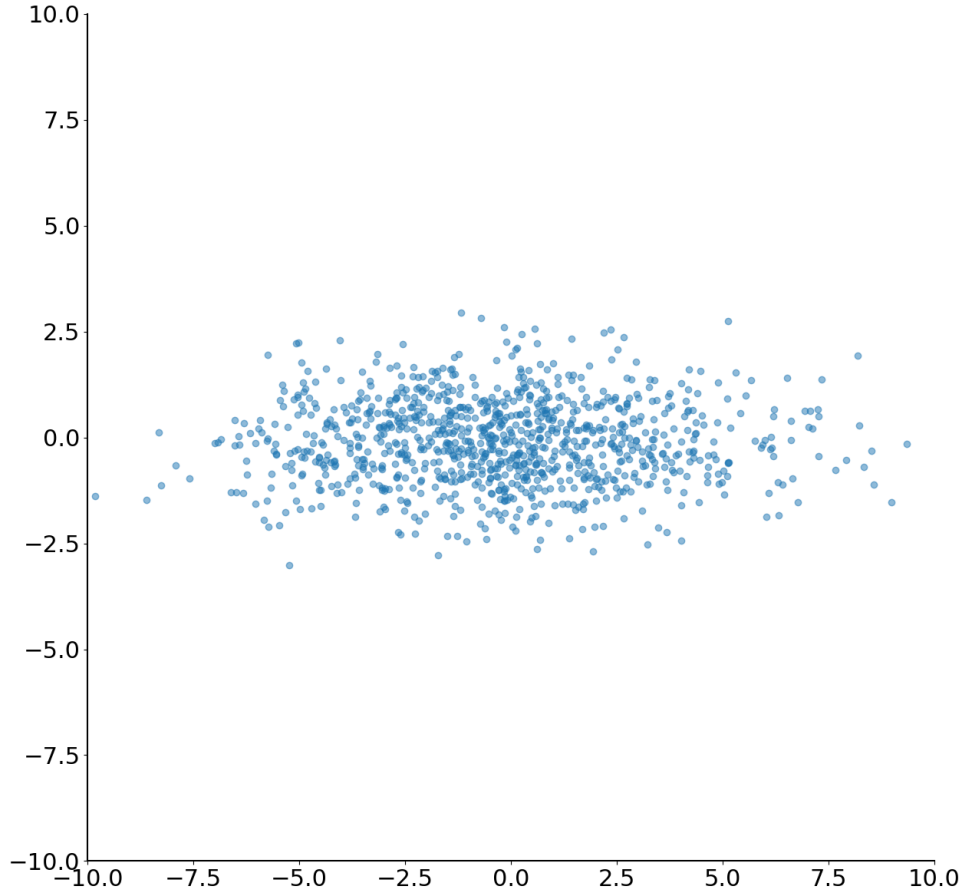
$$\begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\sigma^2(x, y) = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu_x)(y_i - \mu_y)$$

Scaling by 3 times in x-direction?

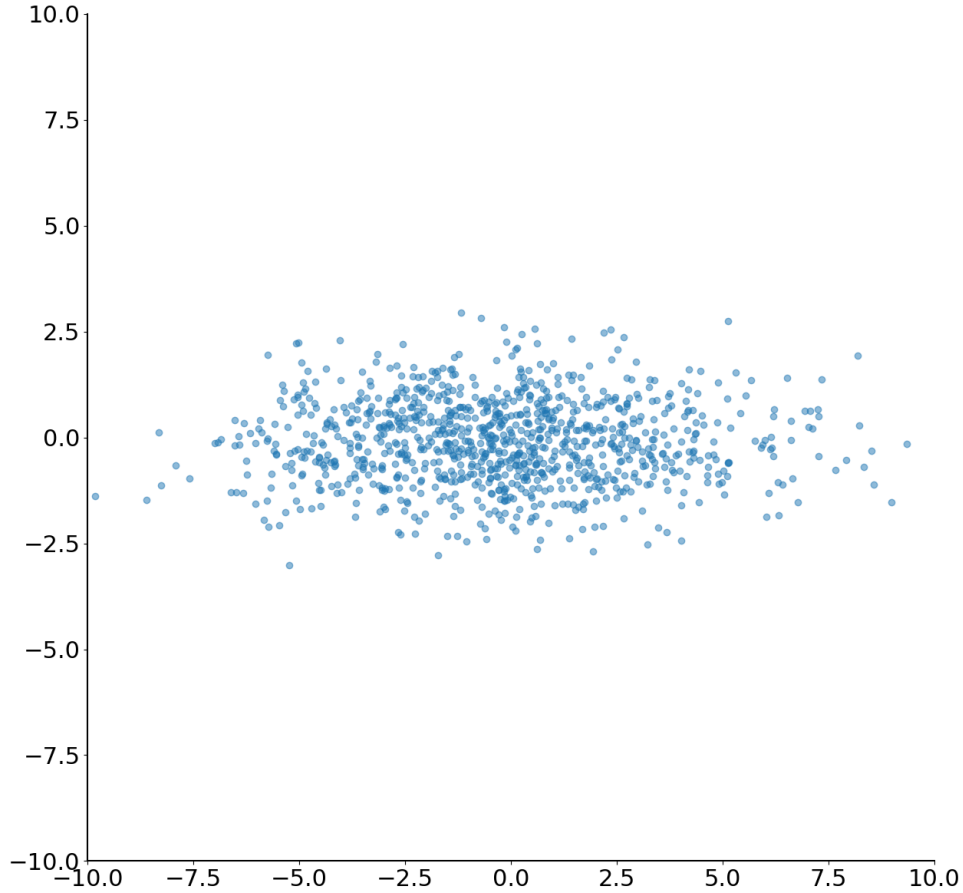


Scaling by 3 times in x-direction?



$$S = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}$$

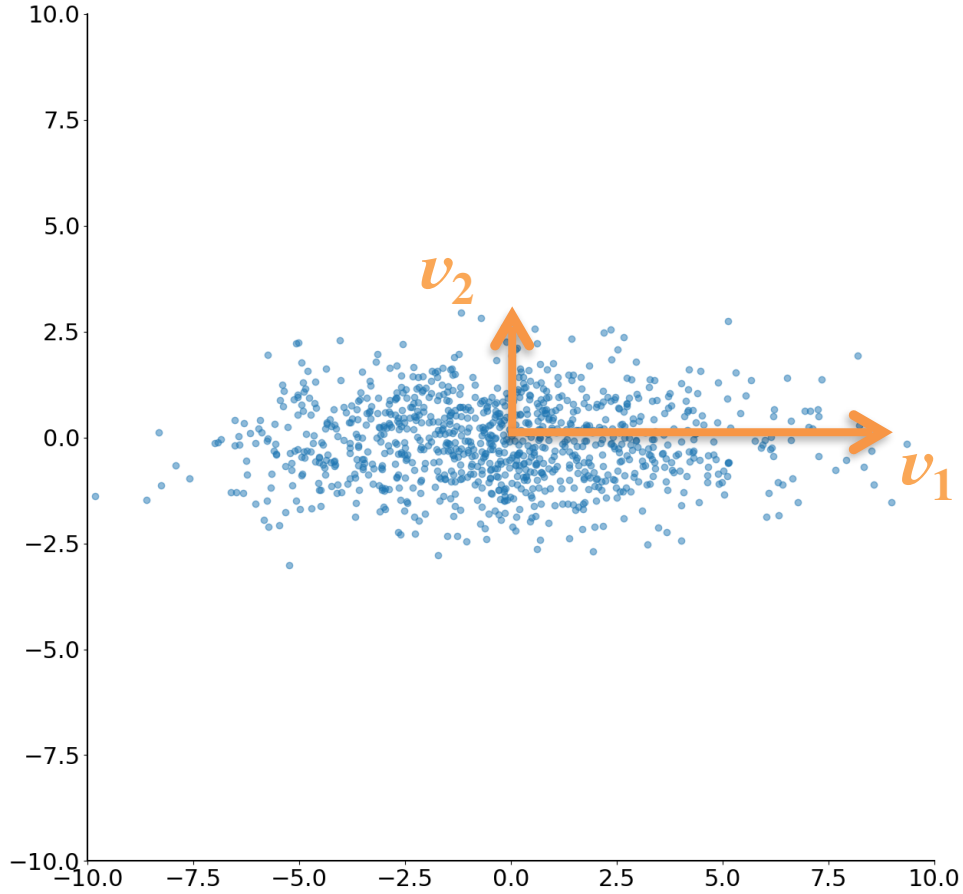
Scaling Transformation Matrix



$$S = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

Eigenvectors

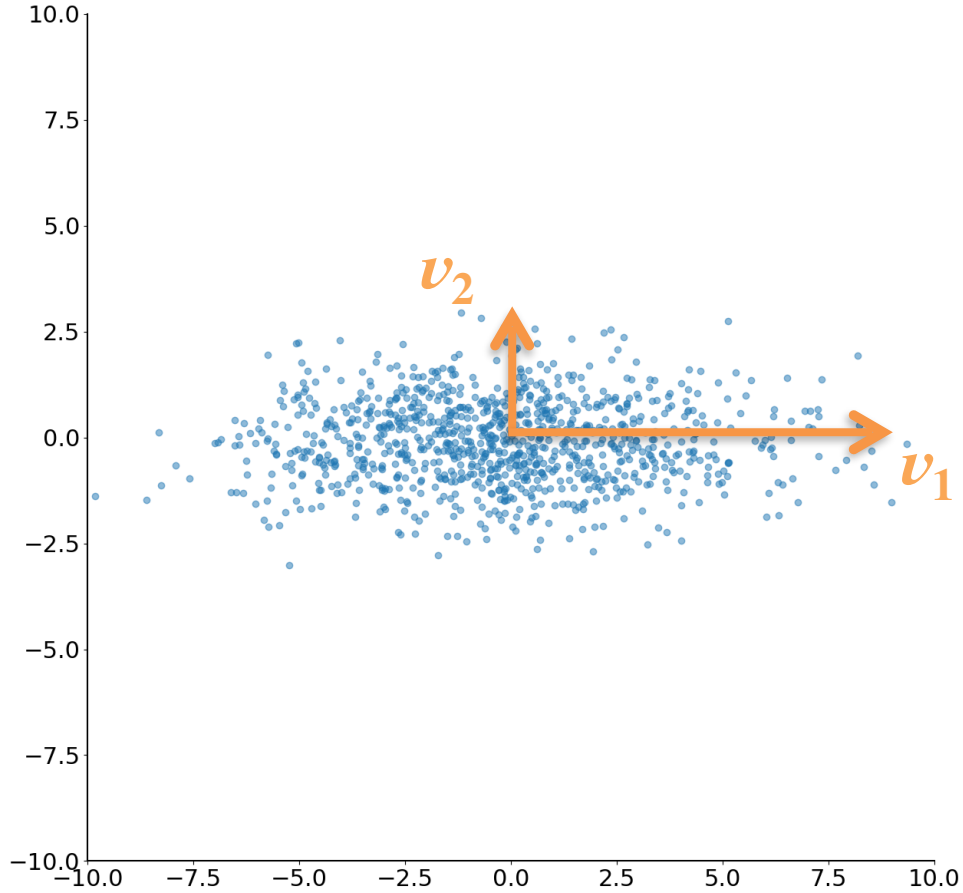


$$S = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

Covariance and Data Transformation

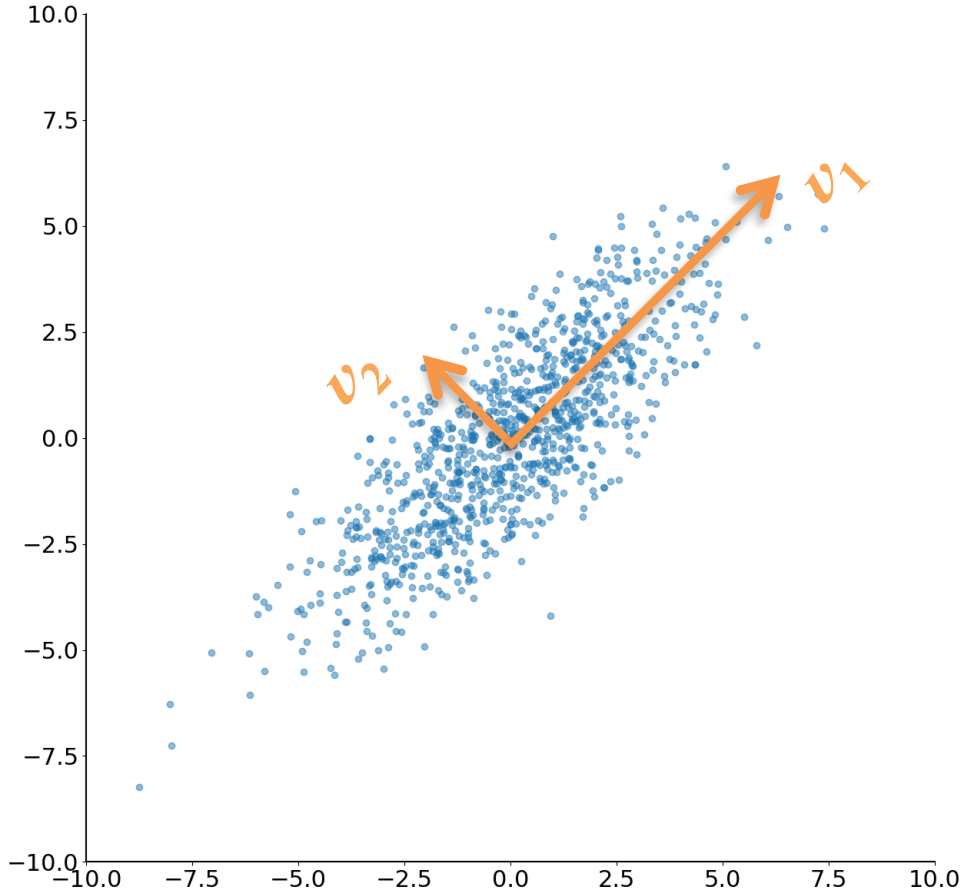
$$\Sigma = \begin{bmatrix} 9 & 0 \\ 0 & 1 \end{bmatrix} = S^2$$



$$S = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

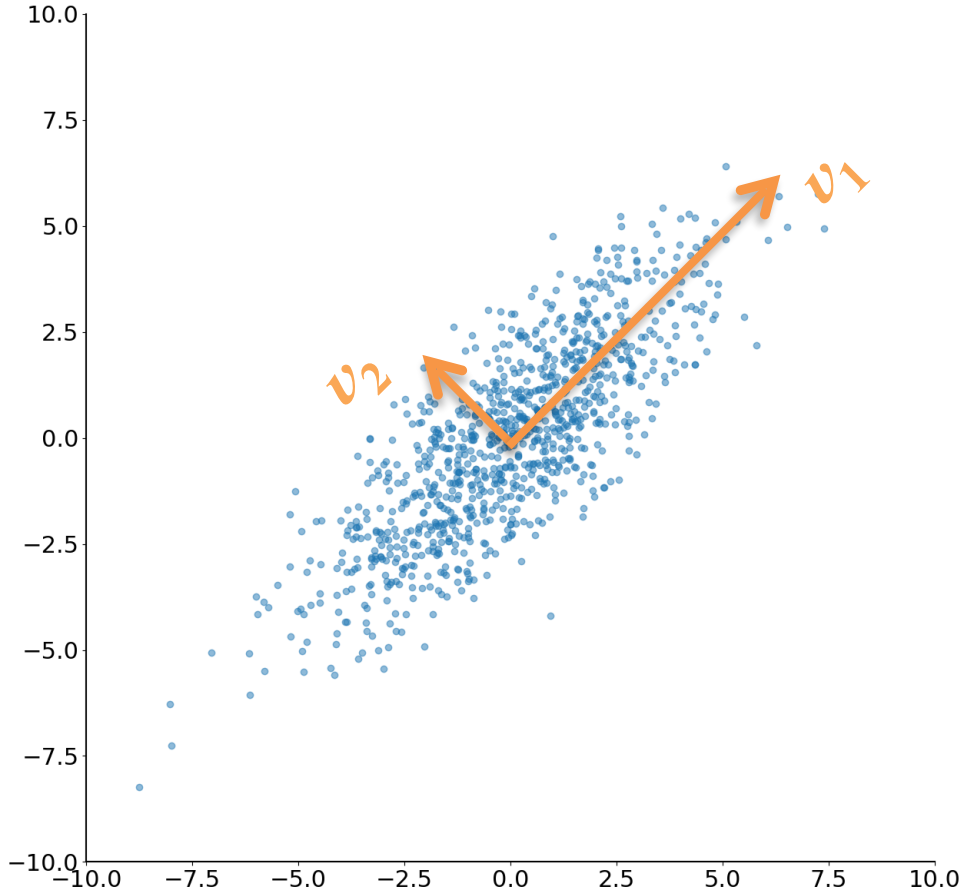
Rotation Transformation



$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta & \cos(\theta) \end{bmatrix}$$

Covariance as Linear Transformations

$$\Sigma v = \lambda v$$

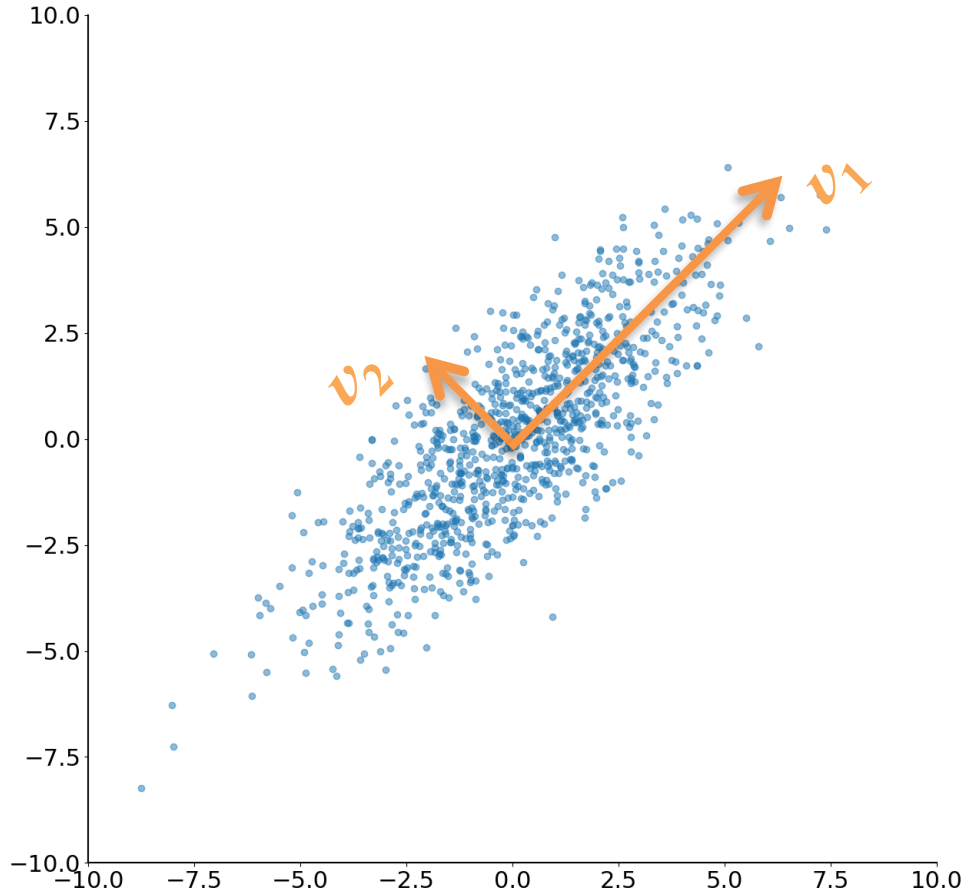


$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta & \cos(\theta) \end{bmatrix}$$

Covariance as Linear Transformations

$$\Sigma v = \lambda v$$

$$\Sigma V = LV$$



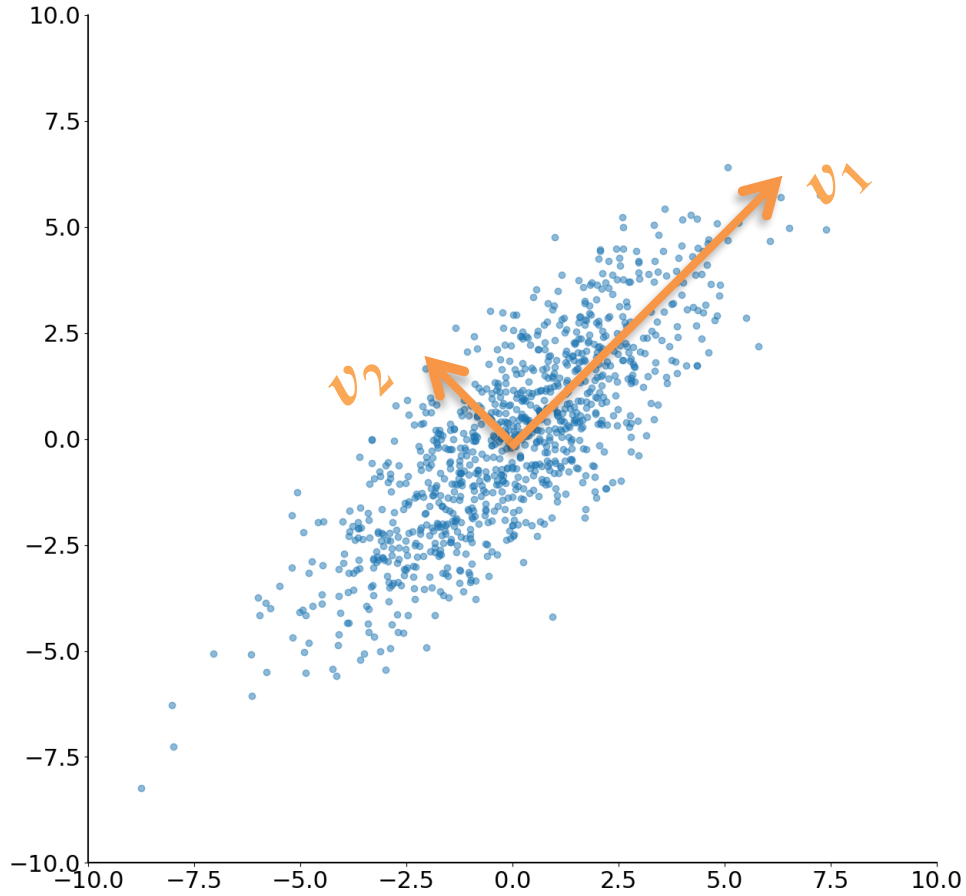
$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Covariance as Linear Transformations

$$\Sigma v = \lambda v$$

$$\Sigma V = LV$$

$$\Sigma = VLV^{-1}$$



$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta & \cos(\theta) \end{bmatrix}$$

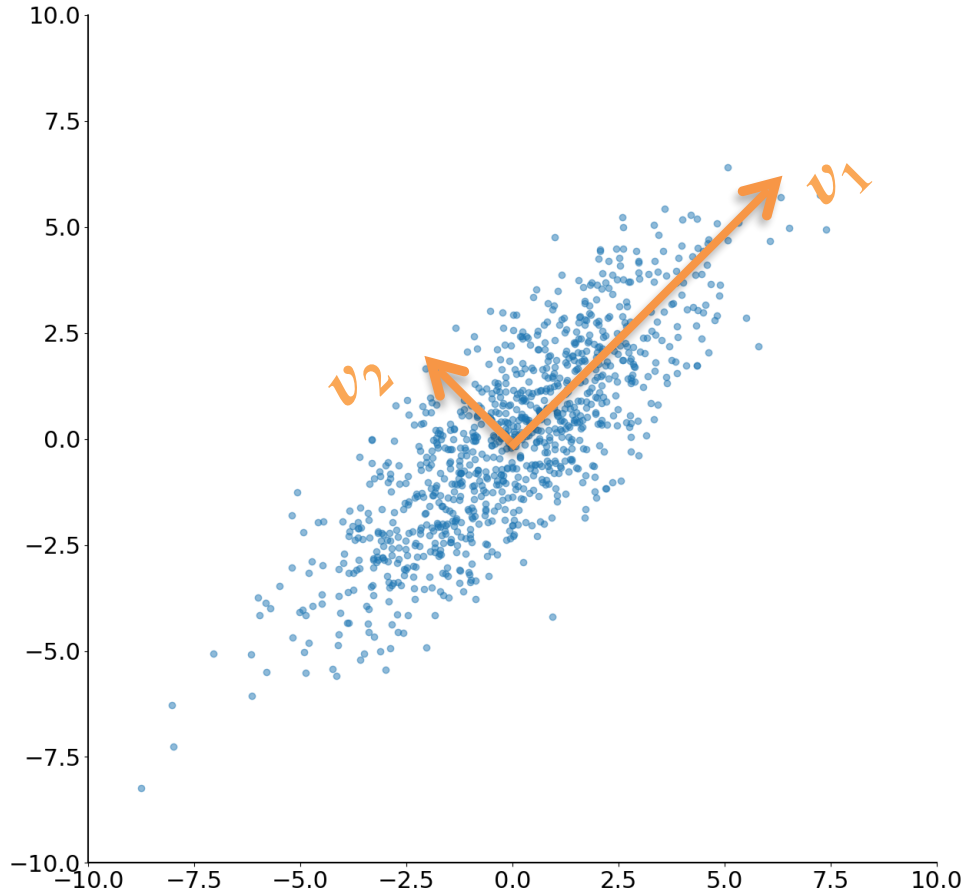
Covariance as Linear Transformations

$$\Sigma v = \lambda v$$

$$\Sigma V = LV$$

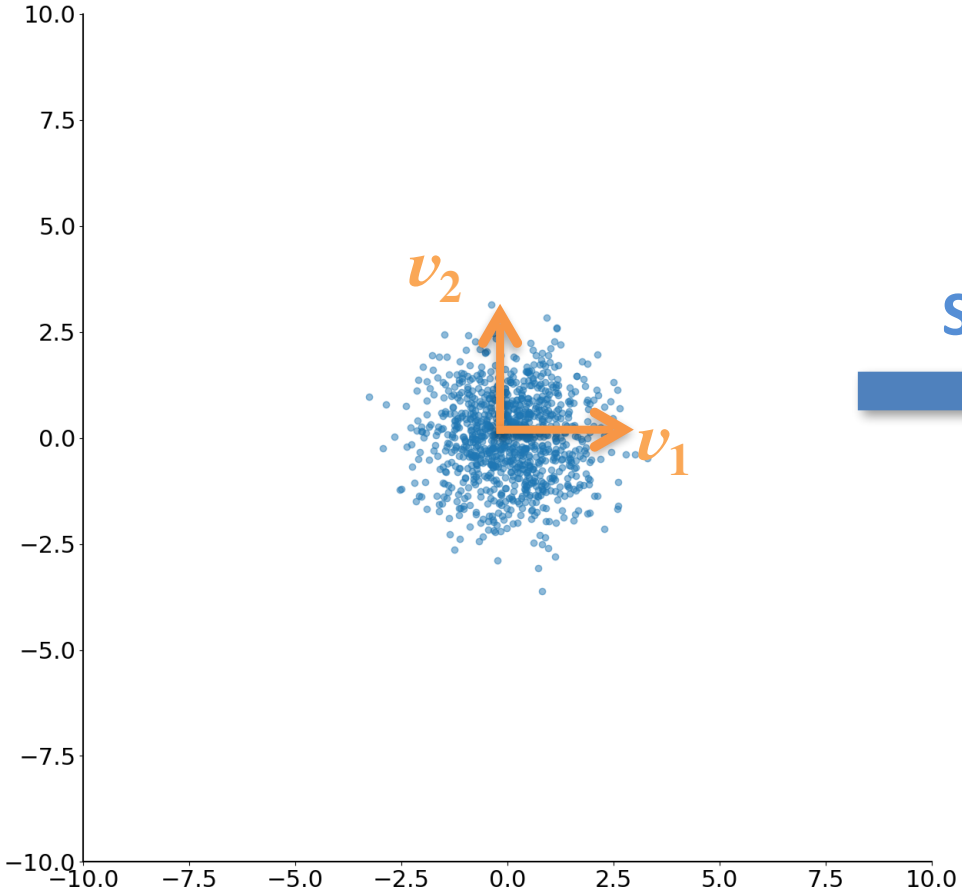
$$\Sigma = VLV^{-1}$$

$$\Sigma = RSSR^{-1}$$



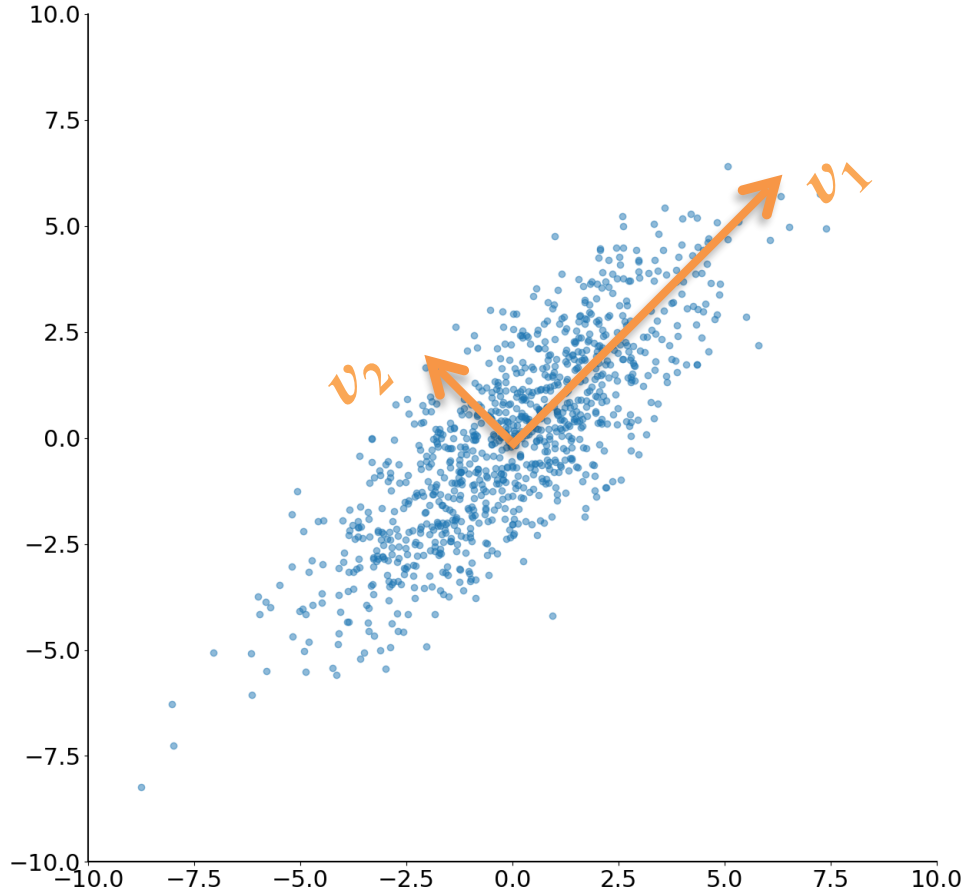
$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta & \cos(\theta) \end{bmatrix}$$

Covariance as Linear Transformations



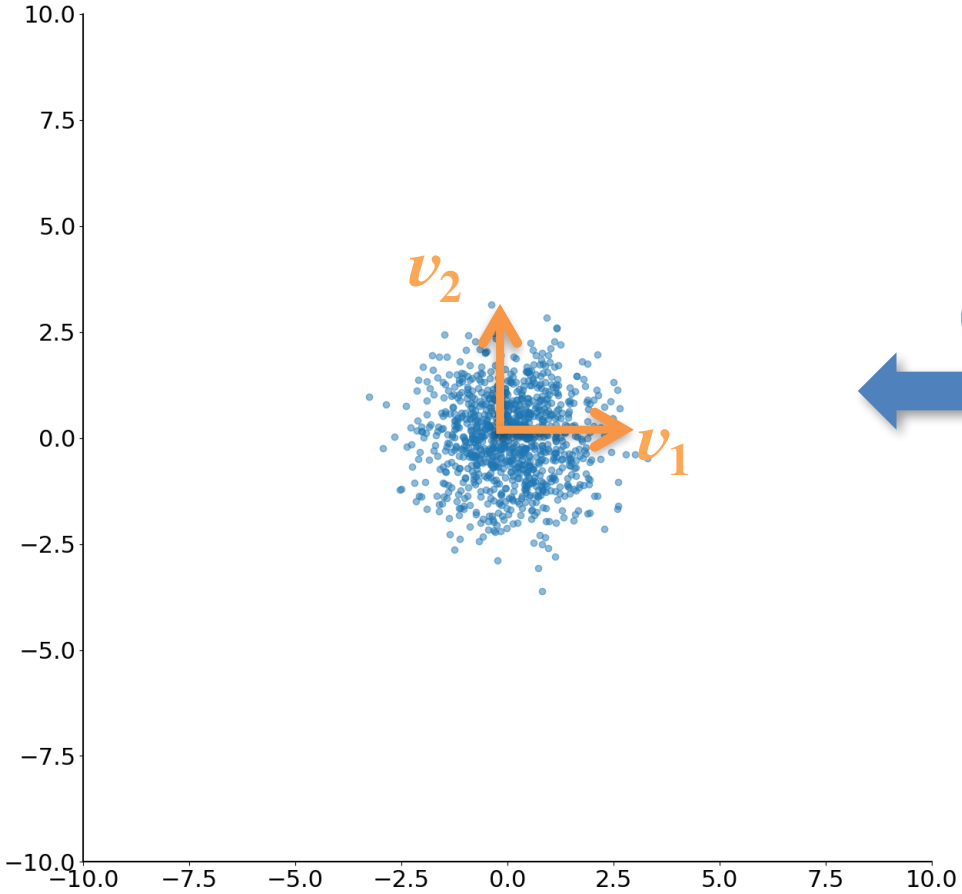
Σ

SR



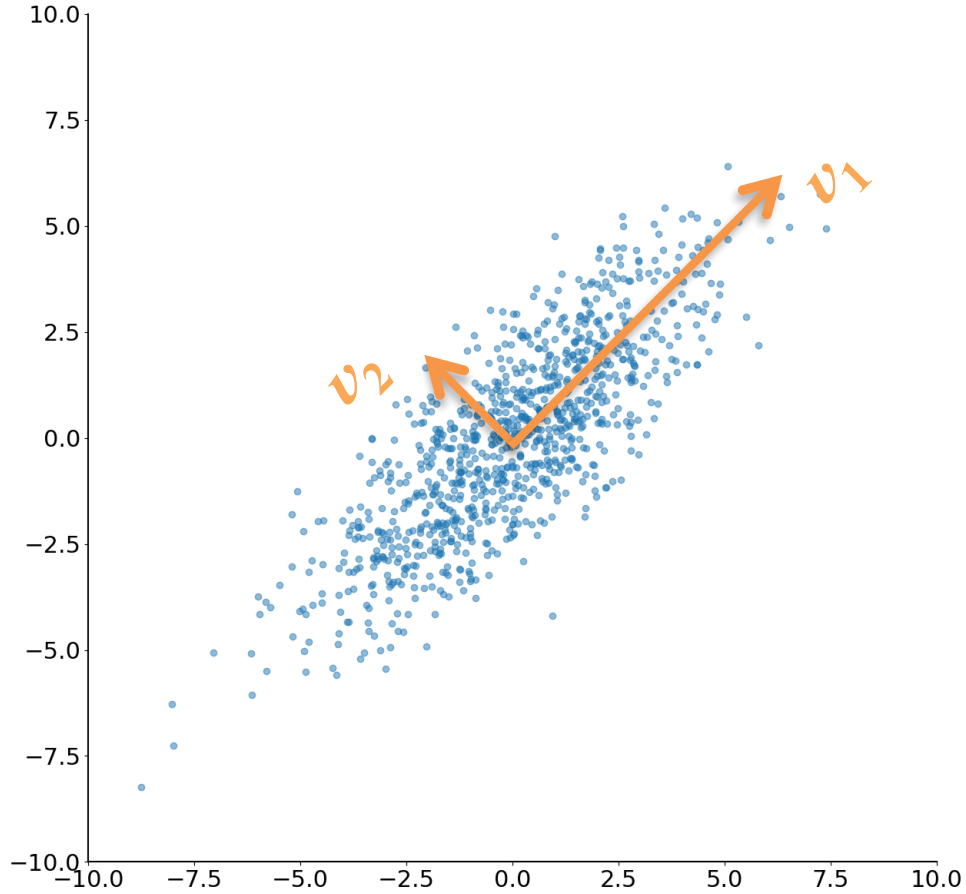
$\Sigma' = RSSR^{-1}$

Covariance as Linear Transformations



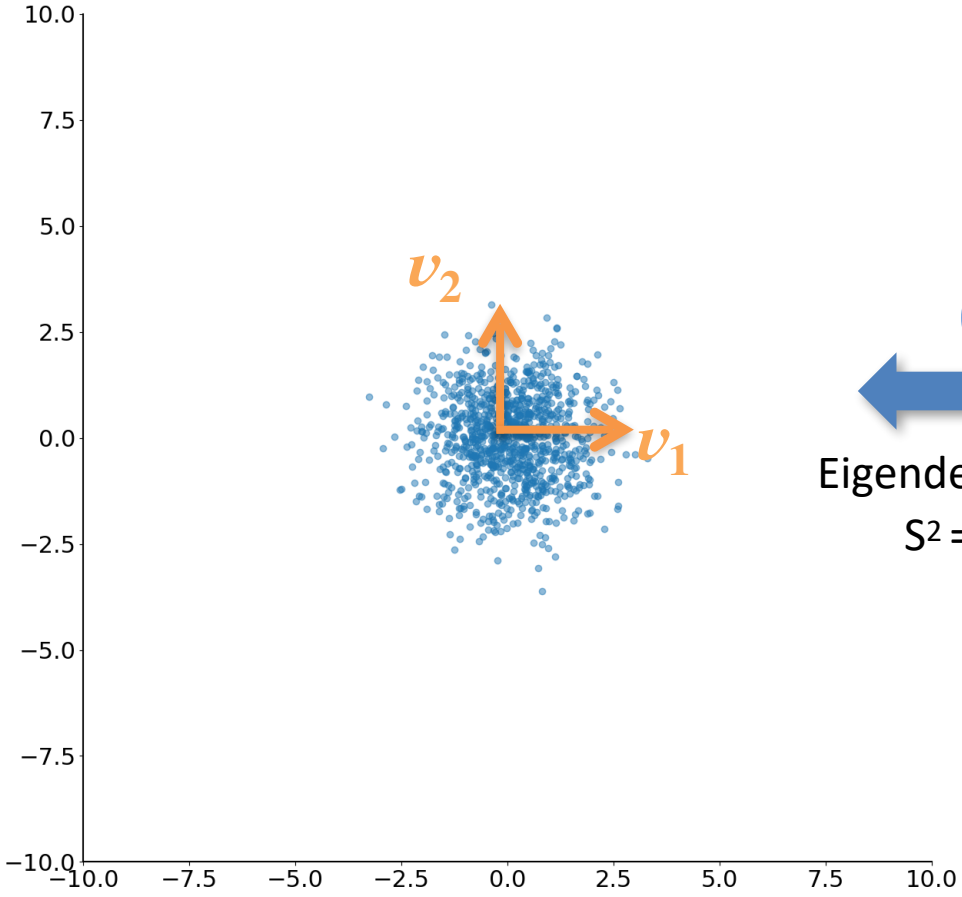
Σ

$(SR)^{-1}$



$\Sigma' = RSSR^{-1}$

Covariance as Linear Transformations

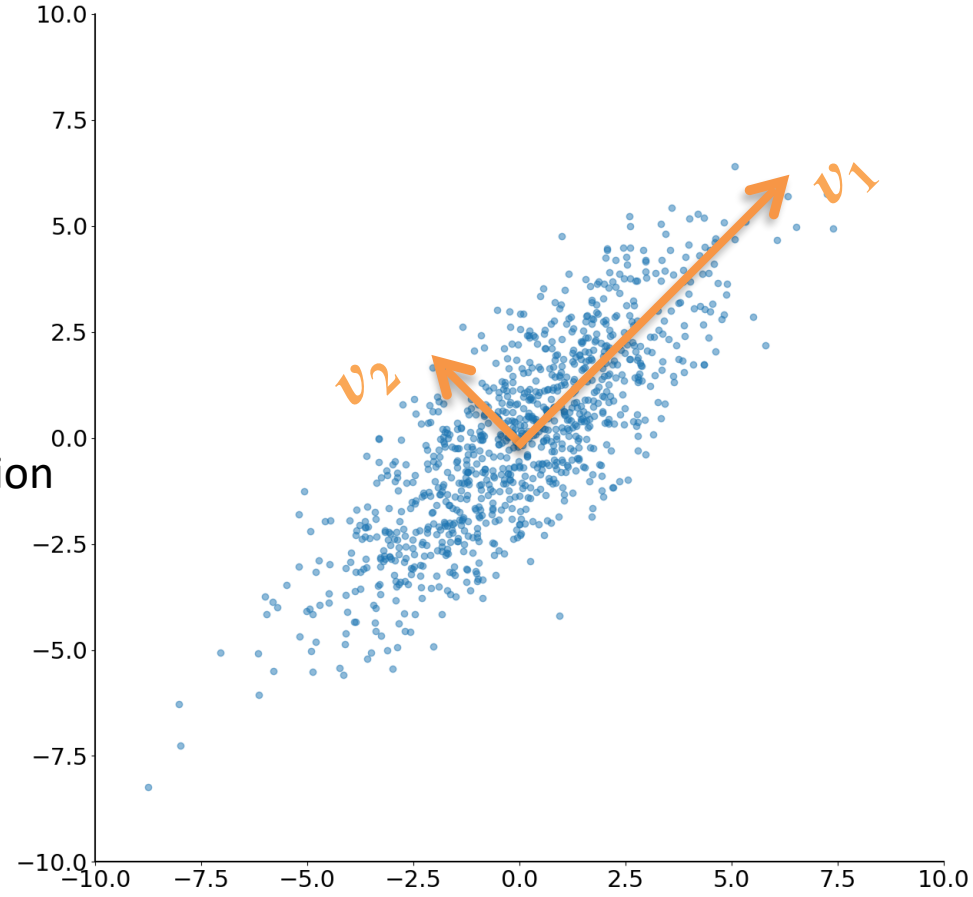


Σ

$(SR)^{-1}$

Eigendecomposition

$S^2 = L$, $R = V$



$\Sigma' = RSSR^{-1}$

numpy.corrcoef

numpy.**corrcoef**(*x*, *y=None*, *rowvar=1*, *bias=<class numpy._NoValue at 0x40a7274c>*, *ddof=<class numpy._NoValue at 0x40a7274c>*) [source]

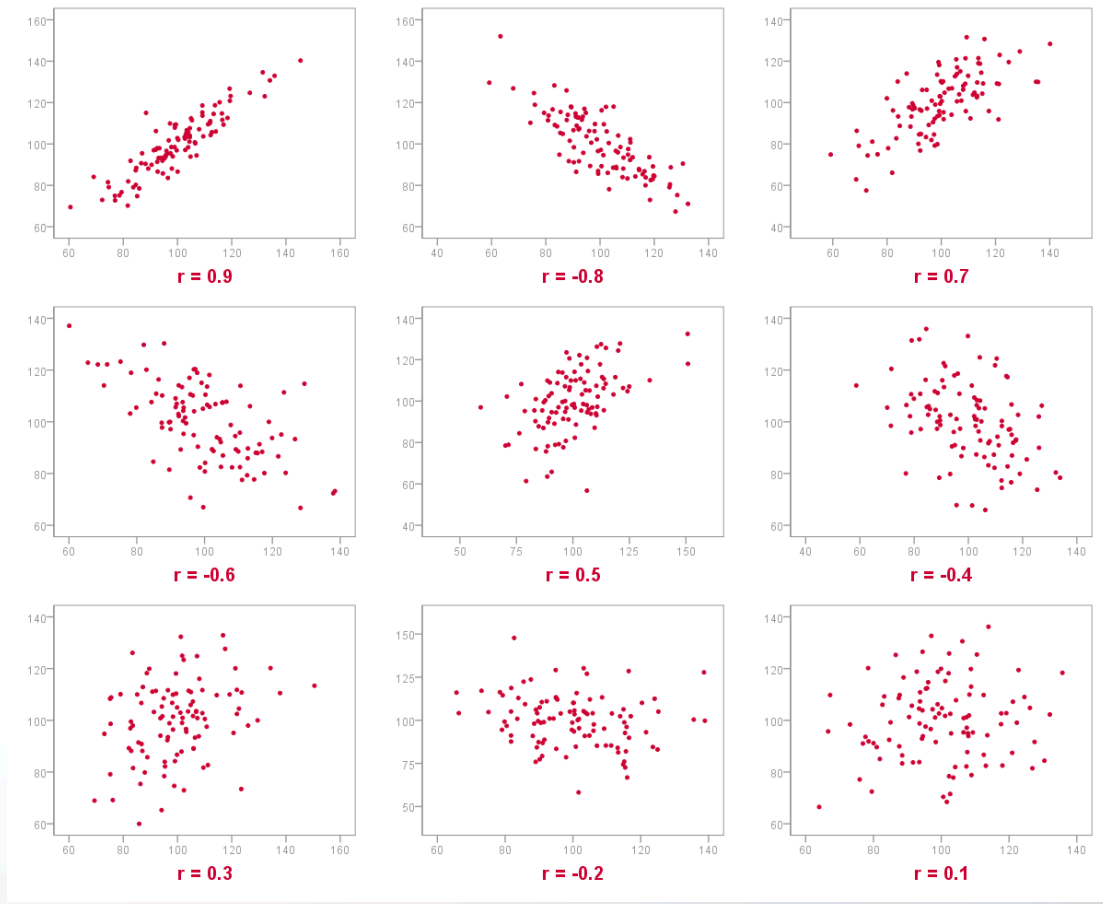
Return Pearson product-moment correlation coefficients.

Please refer to the documentation for cov for more detail. The relationship between the correlation coefficient matrix, *R*, and the covariance matrix, *C*, is

$$R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$$

The values of *R* are between -1 and 1, inclusive.

r value	Strength
0.0 – 0.2	Weak correlation
0.3 – 0.6	Moderate correlation
0.7 – 1.0	Strong correlation



numpy.cov

numpy.cov(*m*, *y=None*, *rowvar=1*, *bias=0*, *ddof=None*, *fweights=None*, *aweights=None*) [source]

Estimate a covariance matrix, given data and weights.

Covariance indicates the level to which two variables vary together. If we examine N-dimensional samples, $X = [x_1, x_2, ...x_N]^T$, then the covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i.

```
>>> x
array([[0, 1, 2],
       [2, 1, 0]])
>>> np.cov(x)
array([[ 1., -1.],
       [-1.,  1.]])
```

numpy.linalg.eig

numpy.linalg.eig(*a*) [source]

Compute the eigenvalues and right eigenvectors of a square array.

Parameters: **a : *(..., M, M) array***

Matrices for which the eigenvalues and right eigenvectors will be computed

Returns: **w : *(..., M) array***

The eigenvalues, each repeated according to its multiplicity. The eigenvalues are not necessarily ordered. The resulting array will be of complex type, unless the imaginary part is zero in which case it will be cast to a real type. When *a* is real the resulting eigenvalues will be real (0 imaginary part) or occur in conjugate pairs

v : *(..., M, M) array*

The normalized (unit “length”) eigenvectors, such that the column `v[:,i]` is the eigenvector corresponding to the eigenvalue `w[i]`.

numpy.dot

numpy.dot(*a*, *b*, *out=None*)

Dot product of two arrays. Specifically,

- If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both *a* and *b* are 2-D arrays, it is matrix multiplication, but using **matmul** or `a @ b` is preferred.
- If either *a* or *b* is 0-D (scalar), it is equivalent to **multiply** and using `numpy.multiply(a, b)` or `a * b` is preferred.
- If *a* is an N-D array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*.
- If *a* is an N-D array and *b* is an M-D array (where `M>=2`), it is a sum product over the last axis of *a* and the second-to-last axis of *b*:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

Parameters:

a : *array_like*

First argument.

b : *array_like*

Second argument.

out : *ndarray, optional*

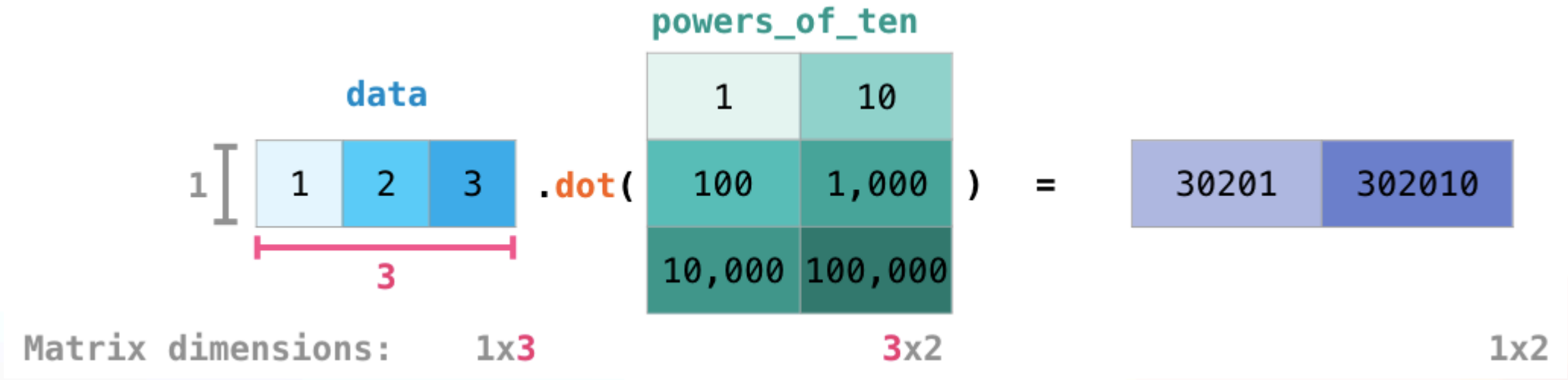
Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for *dot(a,b)*. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

Returns:

output : *ndarray*

Returns the dot product of *a* and *b*. If *a* and *b* are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If *out* is given, then it is returned.

Dot Broadcasting



Exercise

- Generate 100 random points sample from a normal distribution (mu=3.0, sigma=1.0) and visualize them as a histogram.
- Generate 100 random points sample from a normal distribution (mu=1.0, sigma=2.0) and visualize them together with the previous histogram as a histogram.
- Calculate the misclassification rate for these two distributions.
- Fit a normal distribution function to the first data set.
- Generate 100 times 100 random points sample from a normal distribution (mu=3.0, sigma=1.0). Create a histogram of the means and calculate the standard error.
- Repeat the previous point 1000 times instead of 100. How does the standard error differ?

Exercise

- Create a bivariate distribution of 1000 points (normal with mu=1, sigma=1) and visualize it as a scatter plot. You should see a point cloud centered around the origin.
- Apply a transformation along the x-axis via a scaling matrix that stretches the point cloud of data by a factor 3 and visualize the points again.
- Rotate the point cloud by 45 degrees “up” using a rotation matrix.
- How can you apply both transformations (scaling and rotating) in a single transformation?
- Compute the correlation coefficient for the transformed data set.

Exercise

- Calculate the covariance matrix of your transformed bivariate data sample. Compute the eigendecomposition of the covariance matrix. Compute the inverse of the transformation matrix by multiplying eigenvalues with the matrix composed of both eigenvectors (don’t forget to take the square root of the eigenvalues) and transform your data set with it. Plot the transformed data as a scatter plot, you should see a similar point cloud to the original one again (without the stretch and rotation).

numpy.linalg.svd

numpy.linalg.svd(*a*, *full_matrices=True*, *compute_uv=True*, *hermitian=False*) [source]

Singular Value Decomposition.

When *a* is a 2D array, it is factorized as `u @ np.diag(s) @ vh = (u * s) @ vh`, where *u* and *vh* are 2D unitary arrays and *s* is a 1D array of *a*'s singular values. When *a* is higher-dimensional, SVD is applied in stacked mode as explained below.

Parameters:

a : *(..., M, N) array_like*

A real or complex array with `a.ndim >= 2`.

full_matrices : *bool, optional*

If True (default), *u* and *vh* have the shapes `(..., M, M)` and `(..., N, N)`, respectively. Otherwise, the shapes are `(..., M, K)` and `(..., K, N)`, respectively, where `K = min(M, N)`.

compute_uv : *bool, optional*

Whether or not to compute *u* and *vh* in addition to *s*. True by default.

Returns:

u : *{ (..., M, M), (..., M, K) } array*

Unitary array(s). The first `a.ndim - 2` dimensions have the same size as those of the input *a*. The size of the last two dimensions depends on the value of *full_matrices*. Only returned when *compute_uv* is True.

s : *(..., K) array*

Vector(s) with the singular values, within each vector sorted in descending order. The first `a.ndim - 2` dimensions have the same size as those of the input *a*.

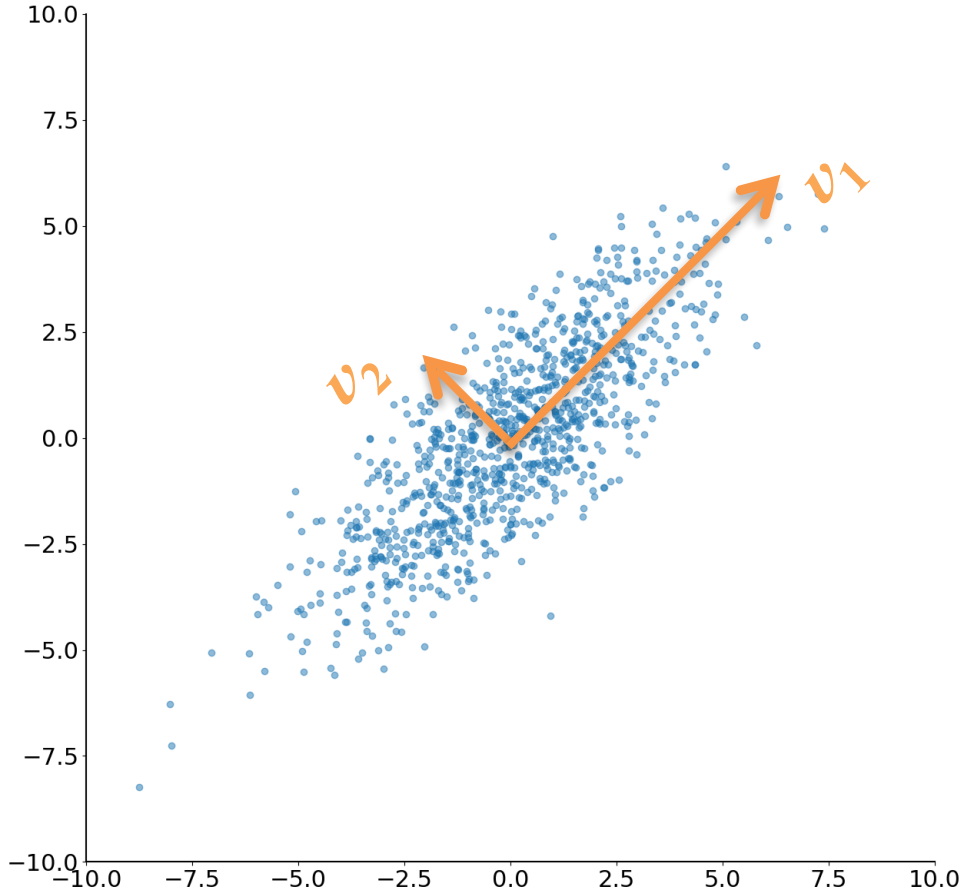
vh : *{ (..., N, N), (..., K, N) } array*

Unitary array(s). The first `a.ndim - 2` dimensions have the same size as those of the input *a*. The size of the last two dimensions depends on the value of *full_matrices*. Only returned when *compute_uv* is True.

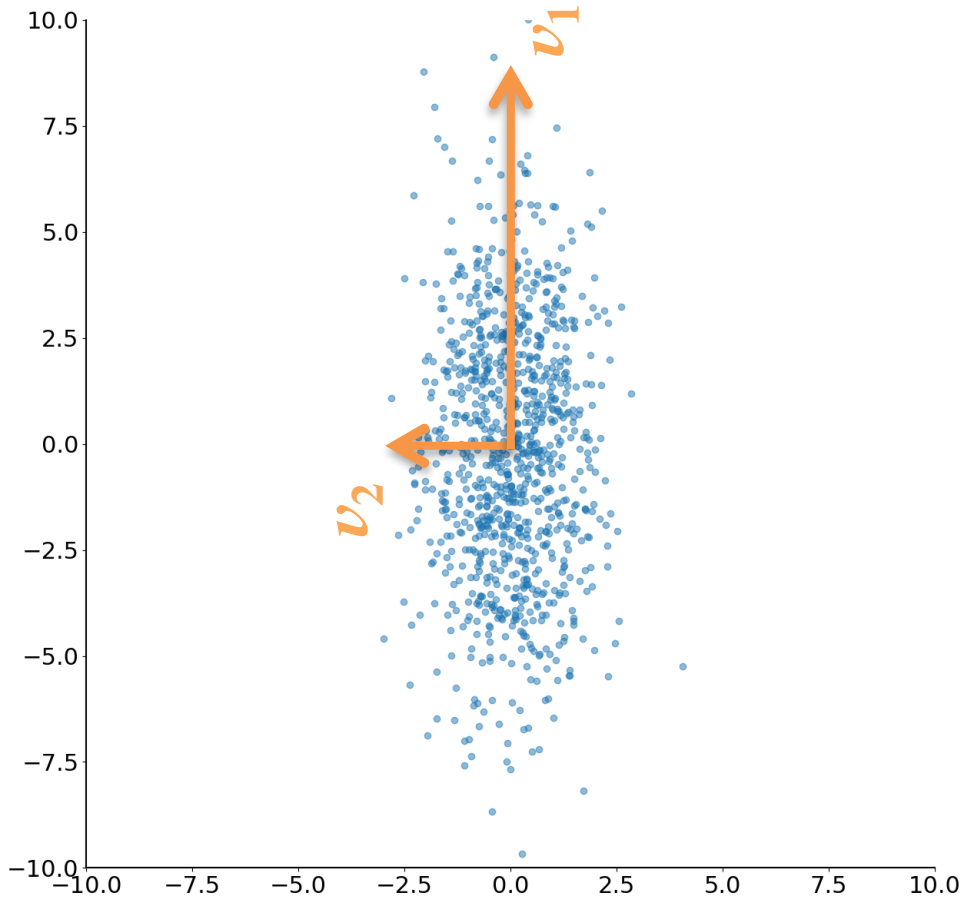
hermitian : *bool, optional*

If True, *a* is assumed to be Hermitian (symmetric if real-valued), enabling a more efficient method for finding singular values. Defaults to False.
New in version 1.17.0.

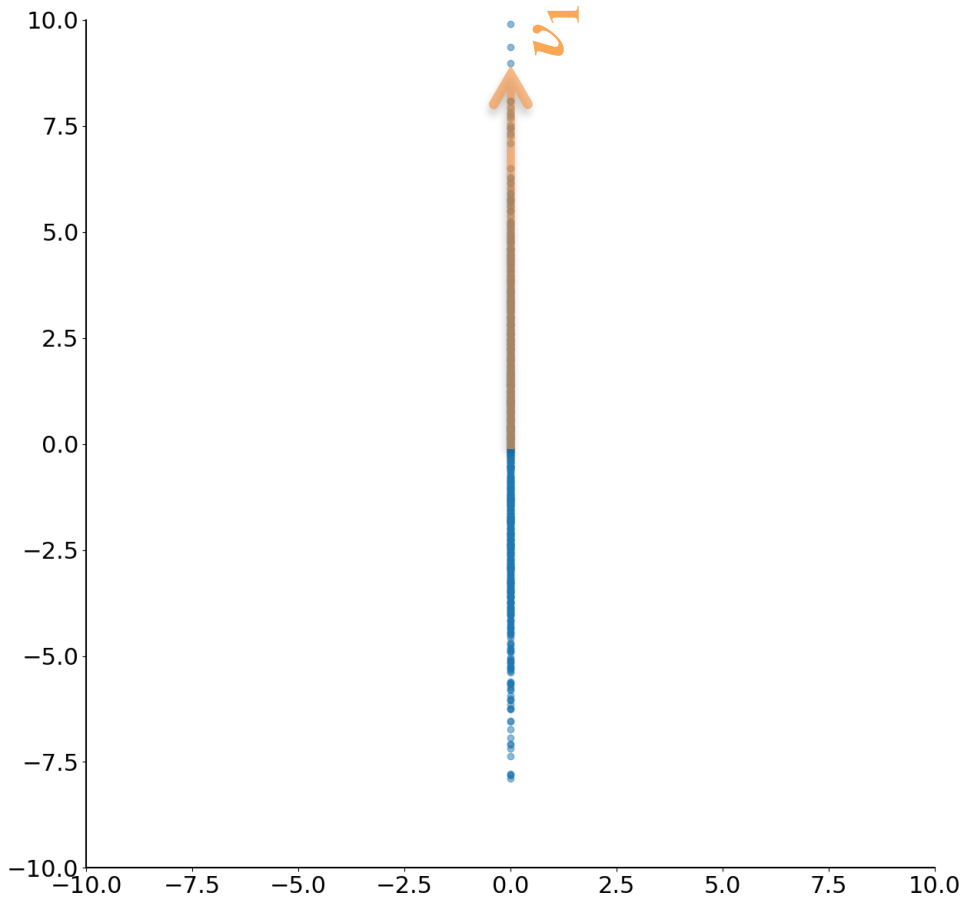
Dimensionality Reduction



Dimensionality Reduction

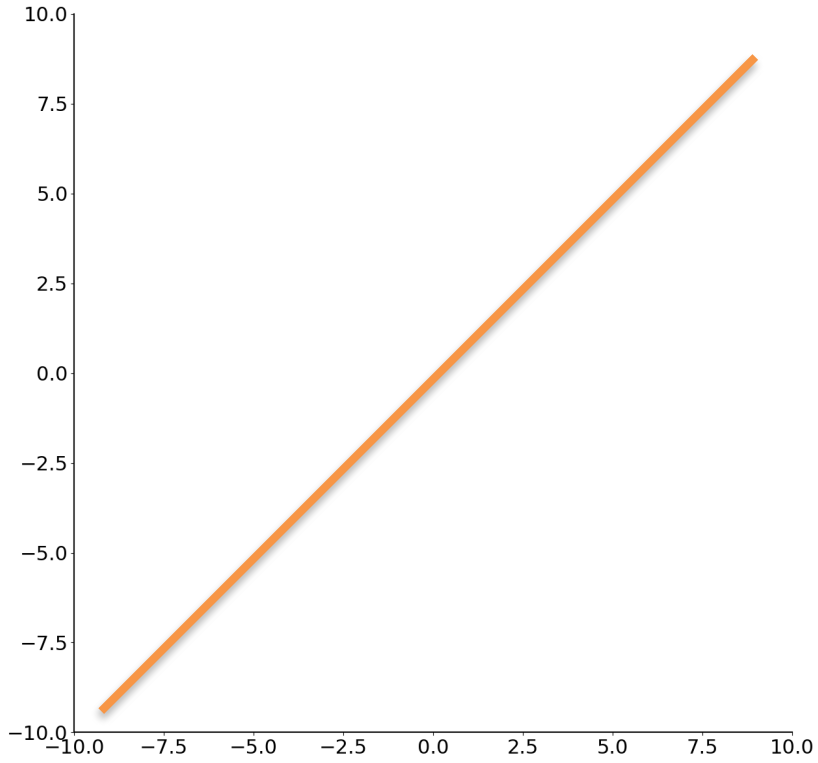


Dimensionality Reduction



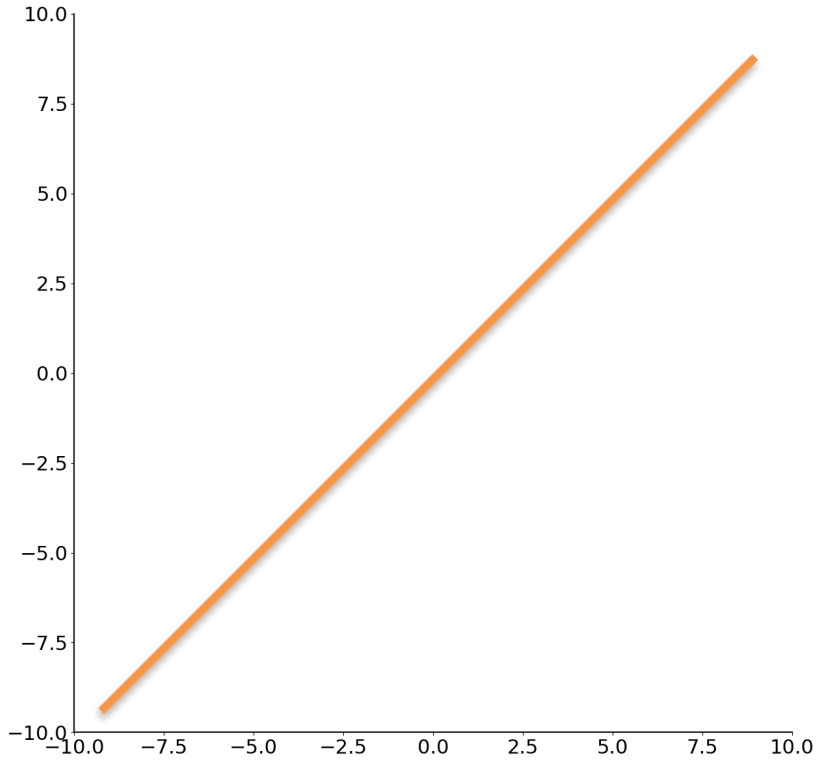
Projection of data on largest eigenvector

Data Model

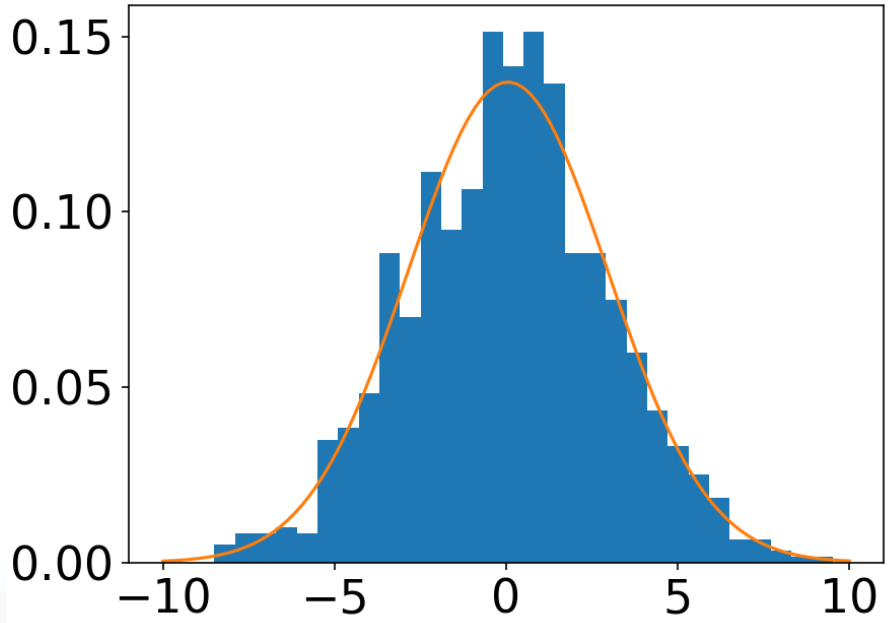


gradient = 1, offset = 0

Data Model

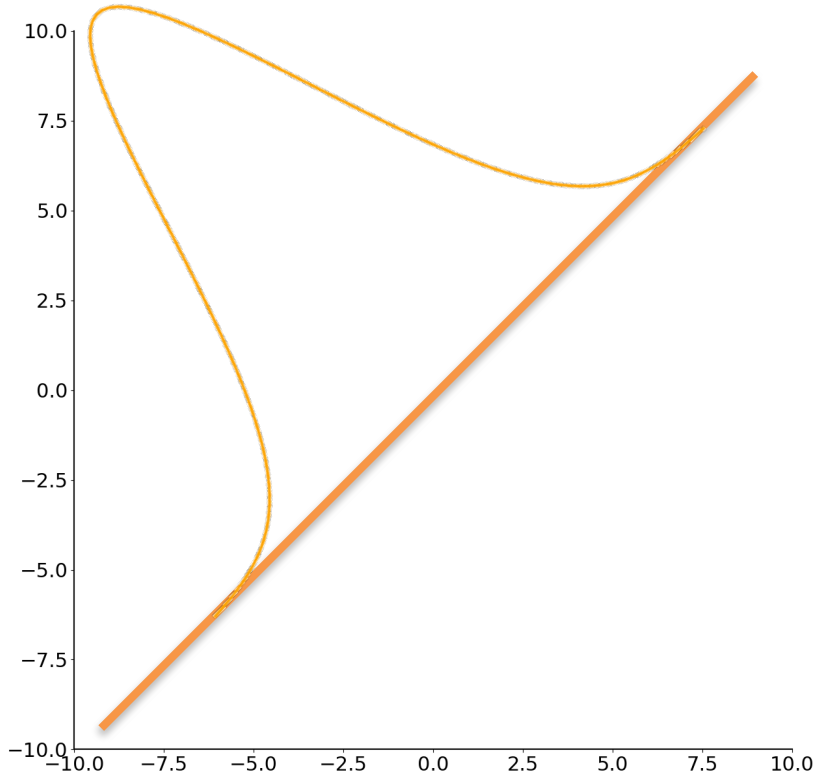


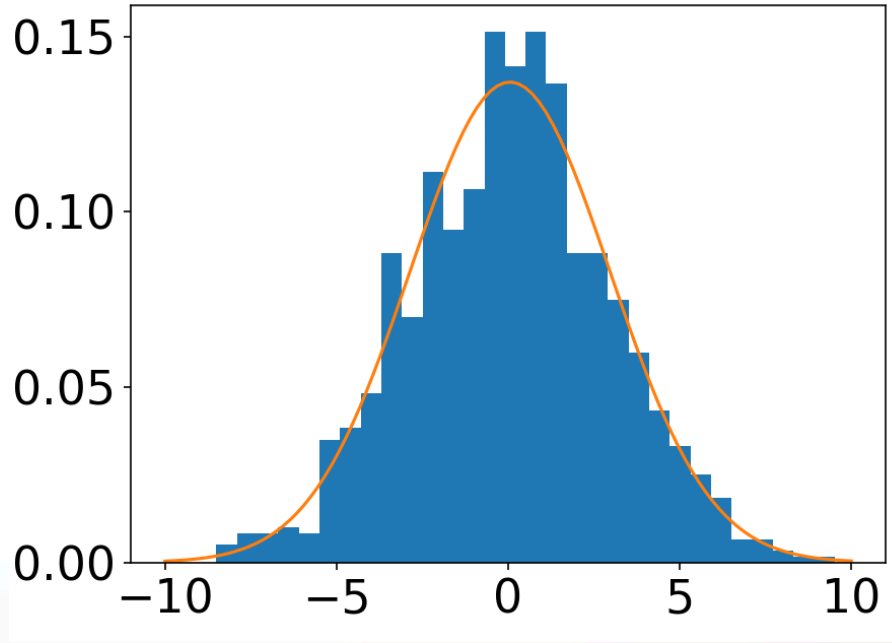
gradient = 1, offset = 0



mu = 0.05, sigma = 2.91

Data Model





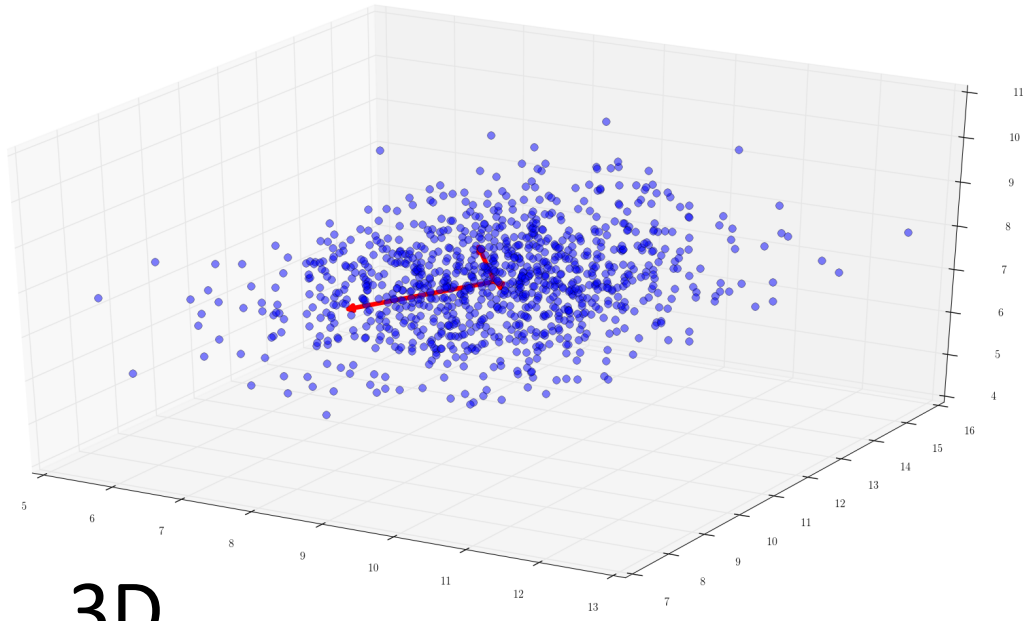
gradient = 1, offset = 0

mu = 0.05, sigma = 2.91

Describe data set of 2000 points with just 4 numbers!

Principal component analysis (PCA)

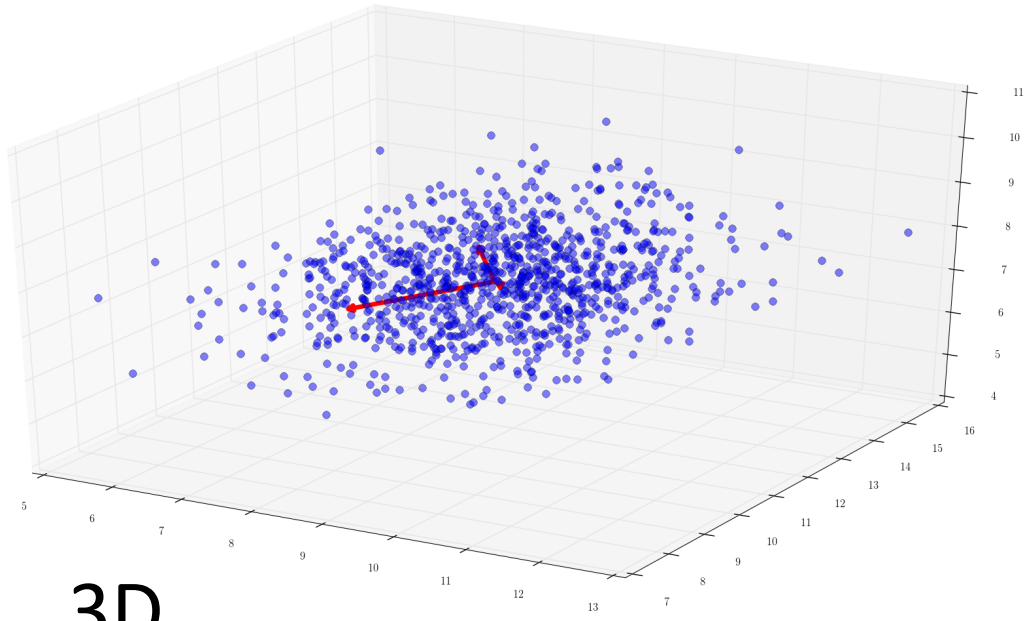
- Reduce dimensions of data sample by projecting onto a lower dimension (while preserving the high variance information)



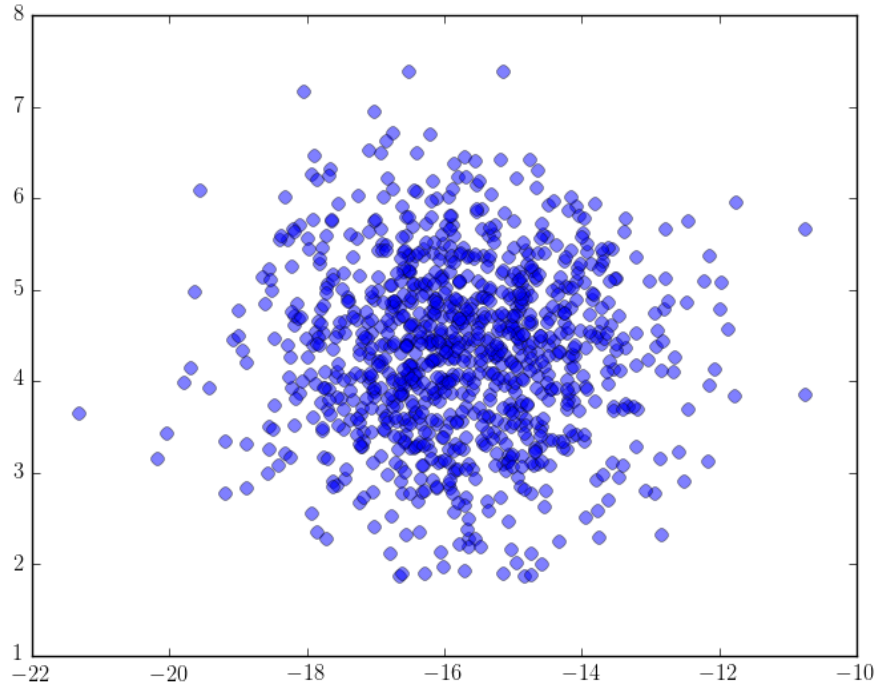
3D

Principal component analysis (PCA)

- Reduce dimensions of data sample by projecting onto a lower dimension (while preserving the high variance information)

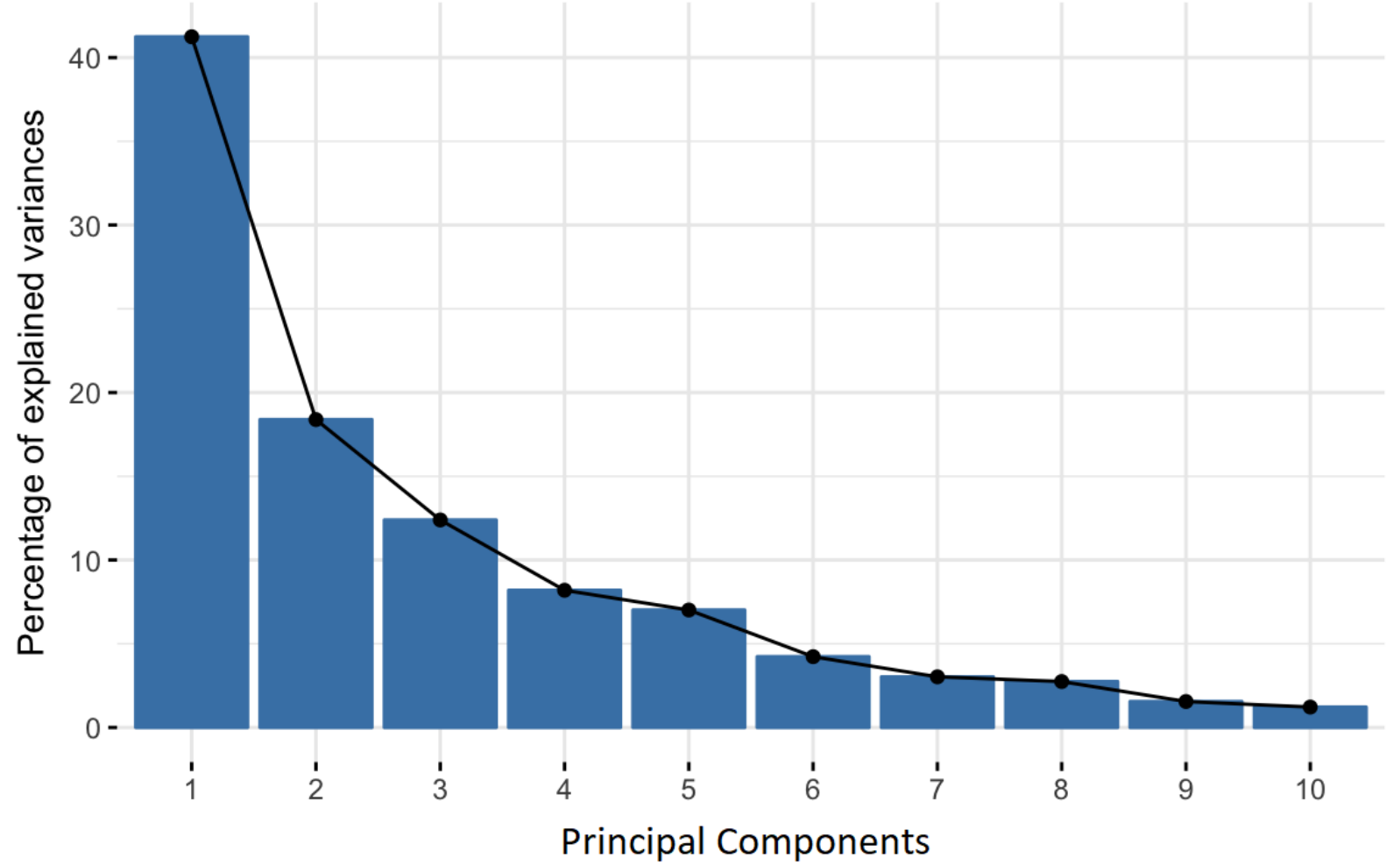


3D



2D

10-Dimensional case



PCA Algorithm

- **Input**: Data X of sample size N
- **Output**: k principal components
- **Centering**: Subtract mean from data
- **Scaling**: Scale each dimension by its variance
- Compute **covariance matrix** C
- Compute k largest **eigenvectors** of C (alternatively calculate **SVD**)

PCA Algorithm

- **Input**: Data X of sample size N
- **Output**: k principal components
- **Centering**: Subtract mean from data
- **Scaling**: Scale each dimension by its variance
- Compute **covariance matrix** C
- Compute k largest **eigenvectors** of C (alternatively calculate **SVD**)

Describes only linear relations in data set!

sklearn.decomposition.PCA

class `sklearn.decomposition.` **`PCA`** (*n_components=None*, *copy=True*, *whiten=False*, *svd_solver='auto'*, *tol=0.0*, *iterated_power='auto'*, *random_state=None*) [source]

Principal component analysis (PCA)

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.

It uses the LAPACK implementation of the full SVD or a randomized truncated SVD by the method of Halko et al. 2009, depending on the shape of the input data and the number of components to extract.

It can also use the scipy.sparse.linalg ARPACK implementation of the truncated SVD.

Notice that this class does not support sparse input. See `TruncatedSVD` for an alternative with sparse data.

Read more in the User Guide.

Parameters:

n_components : *int, float, None or string*

Number of components to keep. if n_components is not set all components are kept:

```
n_components == min(n_samples, n_features)
```

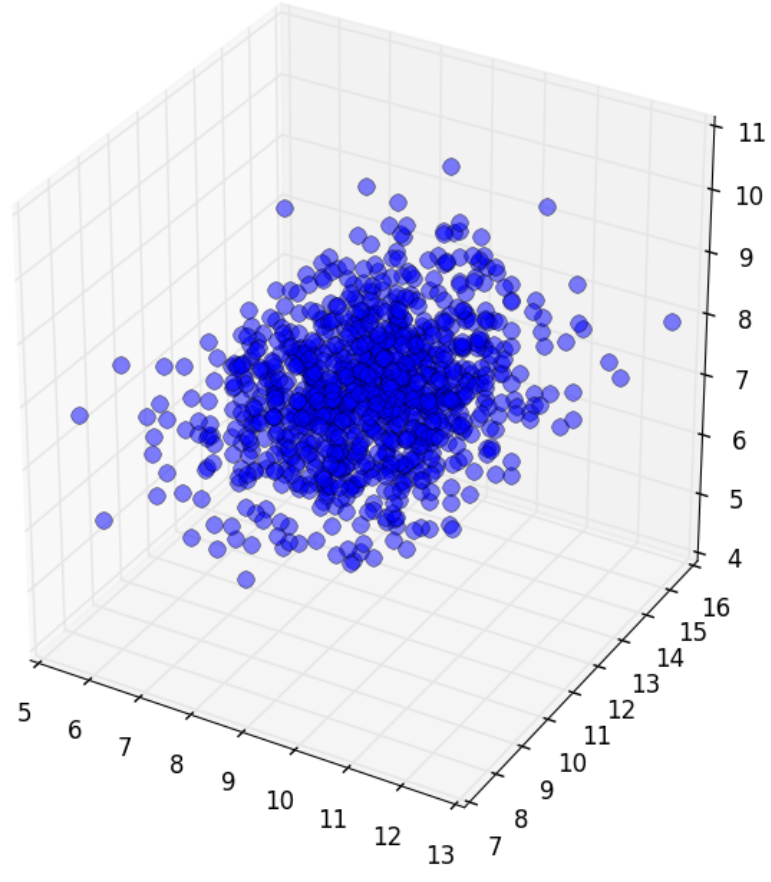
If `n_components == 'mle'` and `svd_solver == 'full'`, Minka's MLE is used to guess the dimension. Use of `n_components == 'mle'` will interpret `svd_solver == 'auto'` as `svd_solver == 'full'`.

If `0 < n_components < 1` and `svd_solver == 'full'`, select the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by n_components.

If `svd_solver == 'arpack'`, the number of components must be strictly less than the minimum

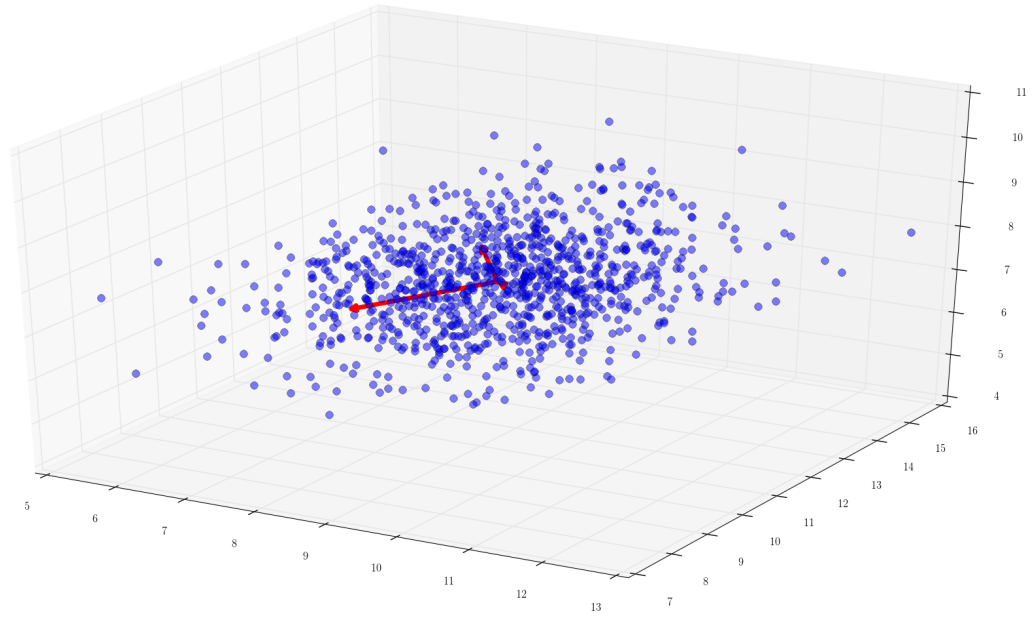
Exercise

- Visualize measurements of the flowers of *ludwiga octovalvis* on a 3D plot. Compute the means and the covariance matrix as well as the correlation matrix – what can you say about the relations of the 3 parameters: sepal length, petal length and sepal width?



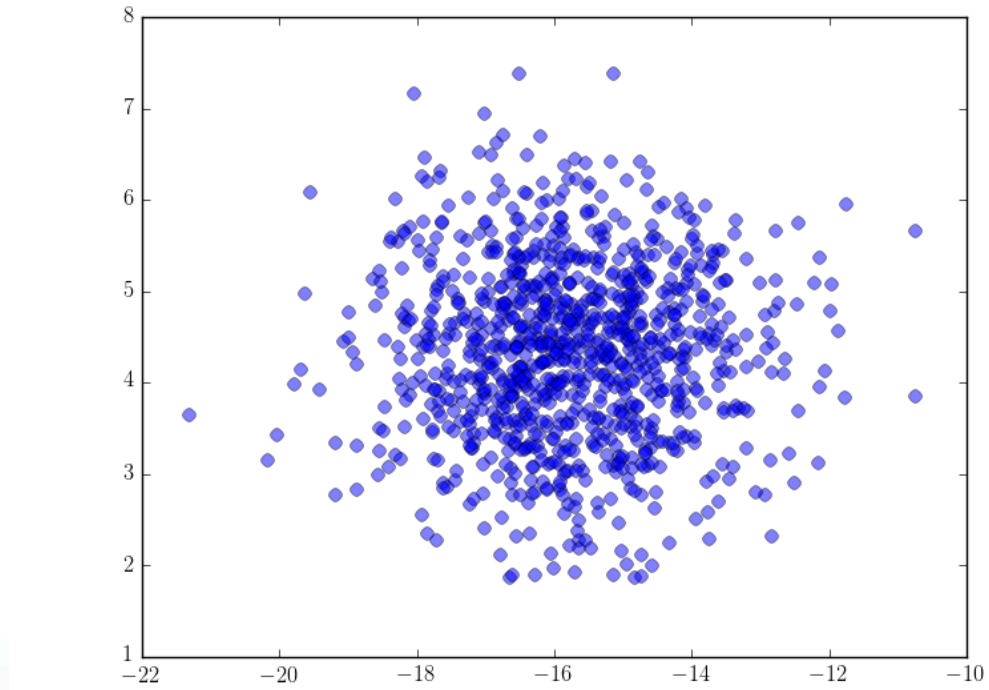
Exercise

- Calculate eigenvectors and eigenvalues of the covariance matrix. Visualize the vectors originating at the data mean (code). To reduce the dimensionality of the data from three to two dimensions what subspace will preserve most information?



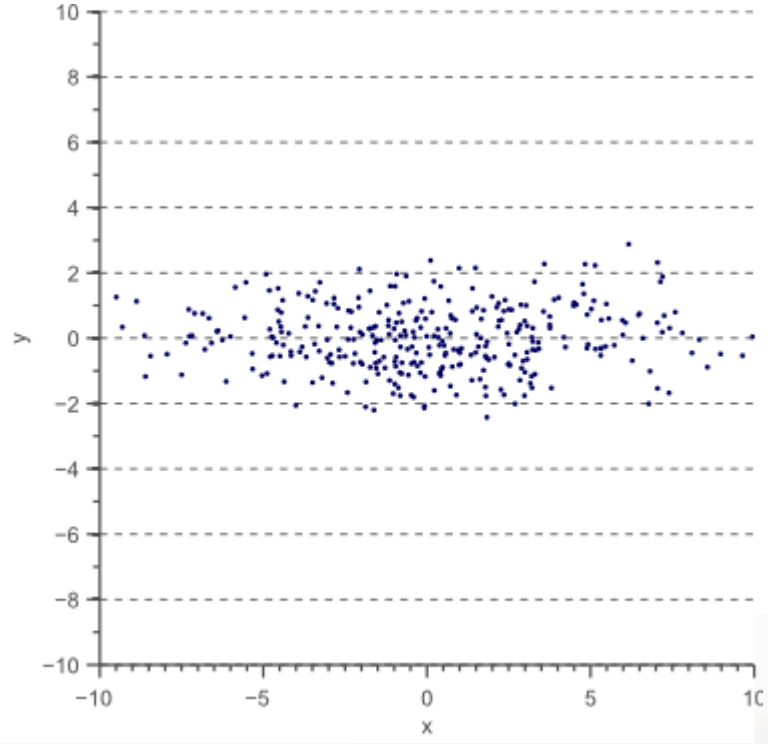
Exercise

- Project the data from 3D to 2D and visualize the result on another plot (multiply the data with a 3x2 matrix constructed from the two largest eigenvectors – alternatively use SVD). Compare your results to the PCA method from the module matplotlib.mlab or sklearn.



Exercise

- Which principal components explain summarily more than 90% of the variance of this multivariate data set?


10
8
6
4
2
0
−2
−4
−6
−8
−10
y
−10
−5
0
5
10
x

numpy.any

numpy.any(*a*, *axis=None*, *out=None*, *keepdims=False*) [source]

Test whether any array element along a given axis evaluates to True.

Returns single boolean unless *axis* is not `None`

Parameters: **a** : *array_like*
Input array or object that can be converted to an array.

axis : *None or int or tuple of ints, optional*
Axis or axes along which a logical OR reduction is performed. The default (*axis* = *None*) is to perform a logical OR over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.
New in version 1.7.0.
If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

Examples

```
>>> np.any([[True, False], [True, True]])
True
```

```
>>> np.any([[True, False], [False, False]], axis=0)
array([ True, False], dtype=bool)
```

numpy.argmax

numpy.argmax(*a*, *axis=None*, *out=None*)¶ [source]

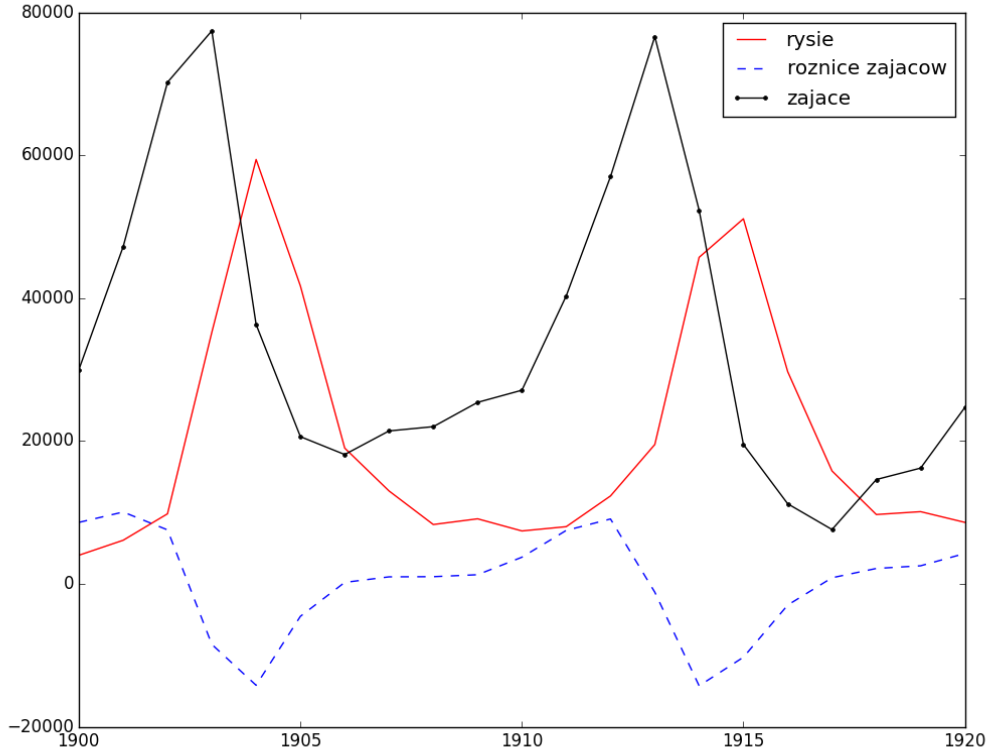
Returns the indices of the maximum values along an axis.

Examples

```
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.argmax(a)
5
>>> np.argmax(a, axis=0)
array([1, 1, 1])
>>> np.argmax(a, axis=1)
array([2, 2])
```

Exercise

- Download the file populacje.txt which contains population numbers of hares, lynxes and carrots in a given time range. Calculate the population means as well as standard deviations of the populations; print out the year in which each species had its greatest population size; for each year which species had the greatest population size; in which years which species had a population size of more than 50,000; create a plot of the differences in the populations of the hares, as well as the populations of hares and lynxes – calculate the correlation coefficients between the differences of the population numbers and the population graph of the lynxes.



scipy.stats.gaussian_kde¶

class **scipy.stats.gaussian_kde**(*dataset*, *bw_method=None*) [source]

Representation of a kernel-density estimate using Gaussian kernels.

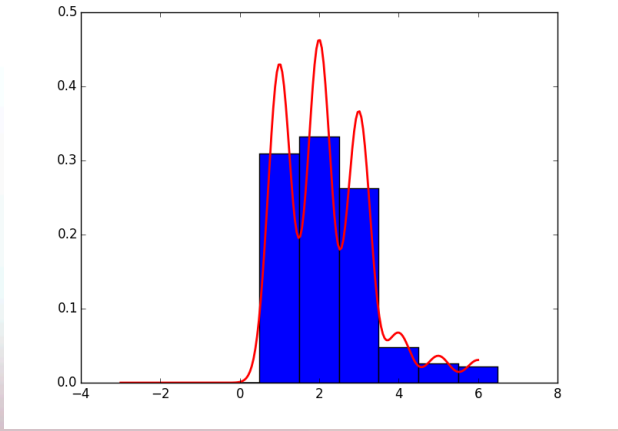
Kernel density estimation is a way to estimate the probability density function (PDF) of a random variable in a non-parametric way. gaussian_kde works for both uni-variate and multi-variate data. It includes automatic bandwidth determination. The estimation works best for a unimodal distribution; bimodal or multi-modal distributions tend to be oversmoothed.

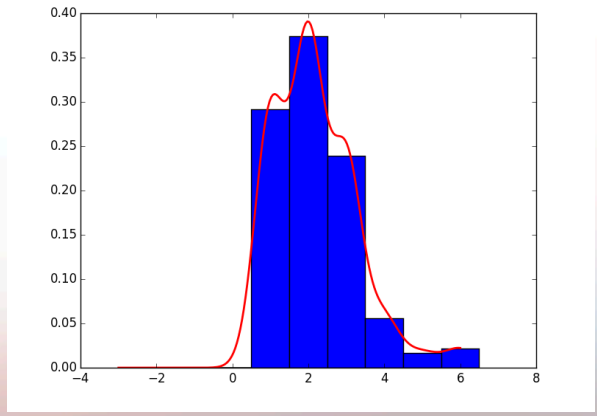
Parameters: **dataset** : *array_like*

Datapoints to estimate from. In case of univariate data this is a 1-D array, otherwise a 2-D array with shape (# of dims, # of data).

bw_method : *str, scalar or callable, optional*

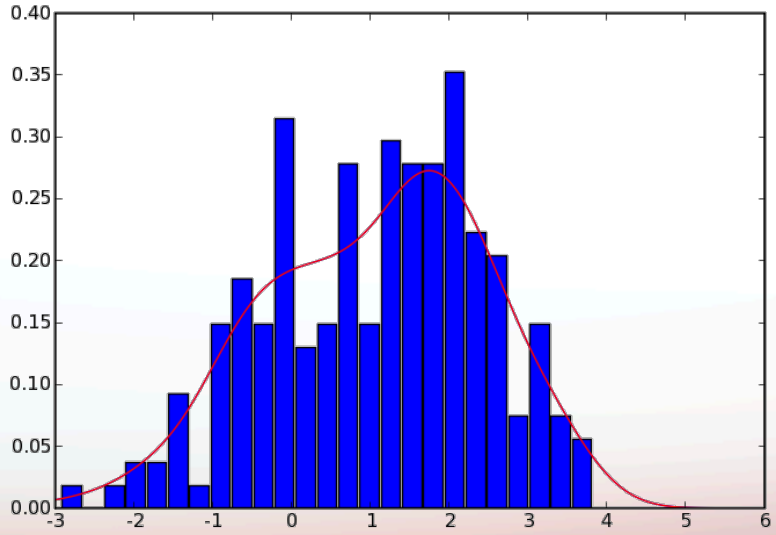
The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If a scalar, this will be used directly as *kde.factor*. If a callable, it should take a gaussian_kde instance as only parameter and return a scalar. If None (default), 'scott' is used. See Notes for more details.





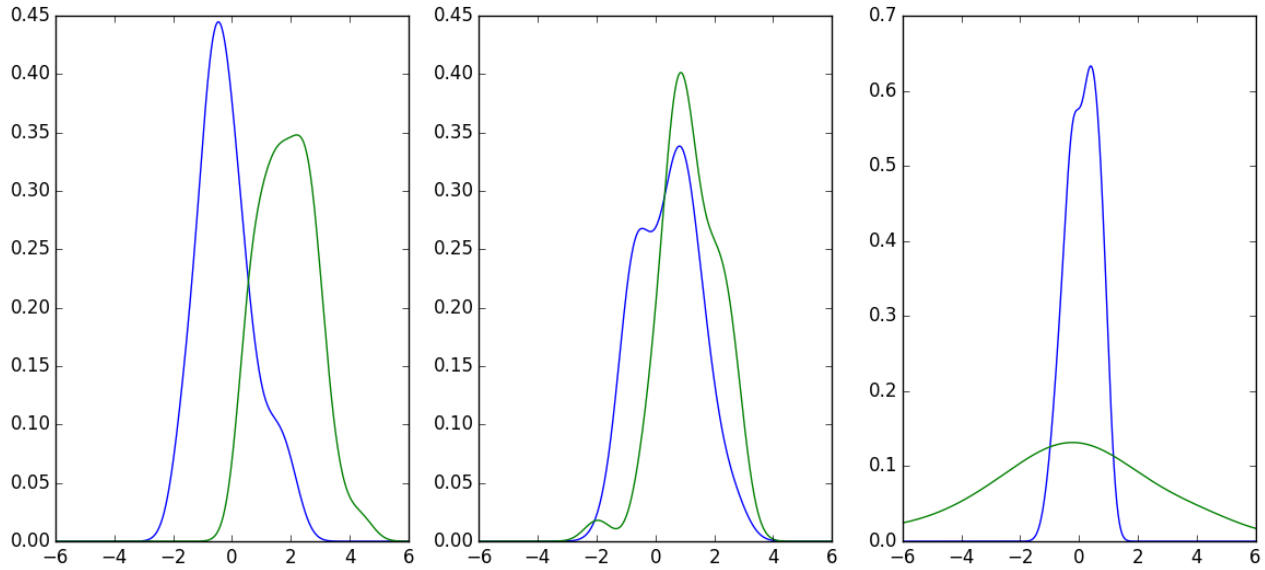
Exercise

- Create a random data sample using two normal distribution functions (mu, sigma = (0.0, 1.0) and (2.0, 0.8)). Calculate the kernel density estimation of this distribution. Draw the histogram of the data sample and the KDE.



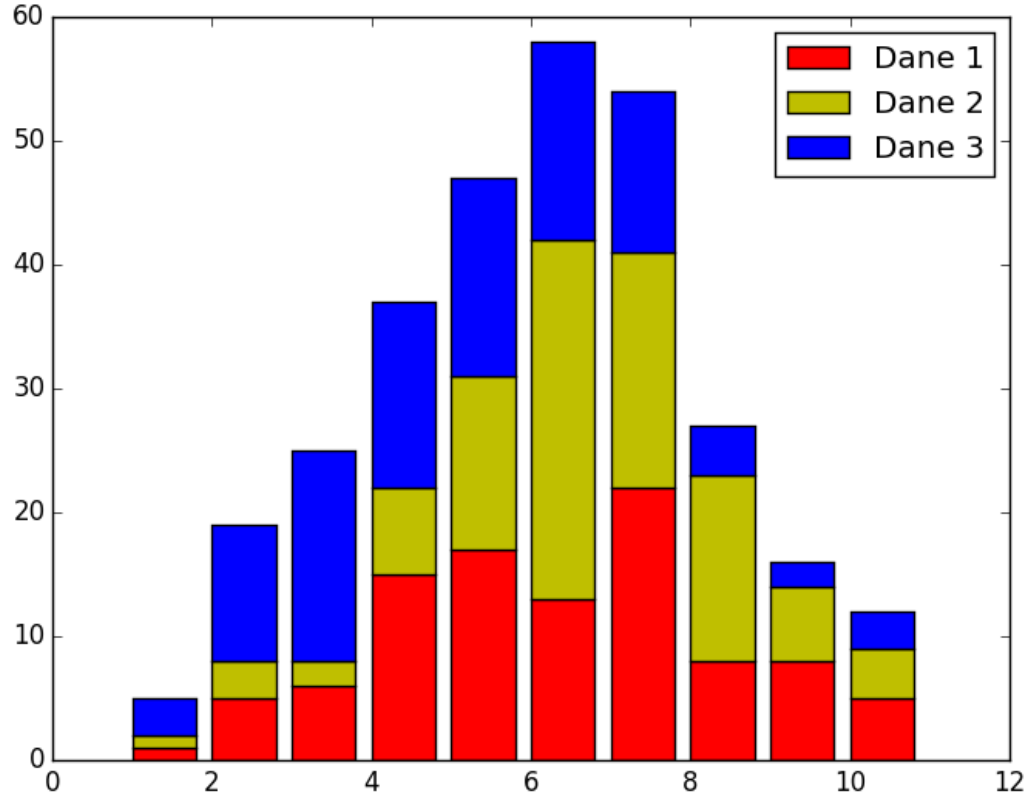
Exercise

- For different pairs of data samples (of two independent variables) calculate and draw the KDEs on a graph as well as test whether the means are significantly different. Test the zero hypothesis for each distribution: the mean is equal to 0 (stats.ttest_1samp).



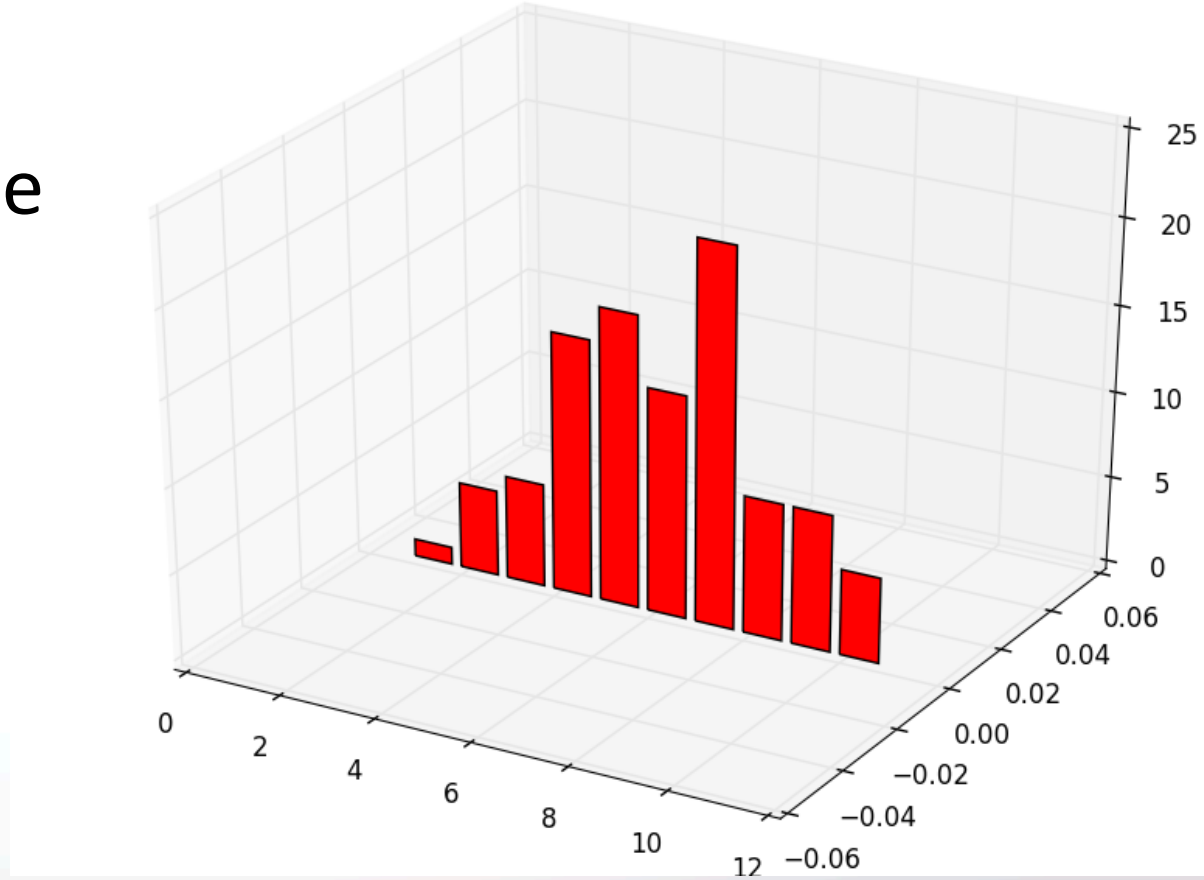
Exercise

- Download the file containing 3 data samples (in separate rows). Create a box plot just like in the image to the right. Use the function **plt.bar()** and the parameter **bottom**.



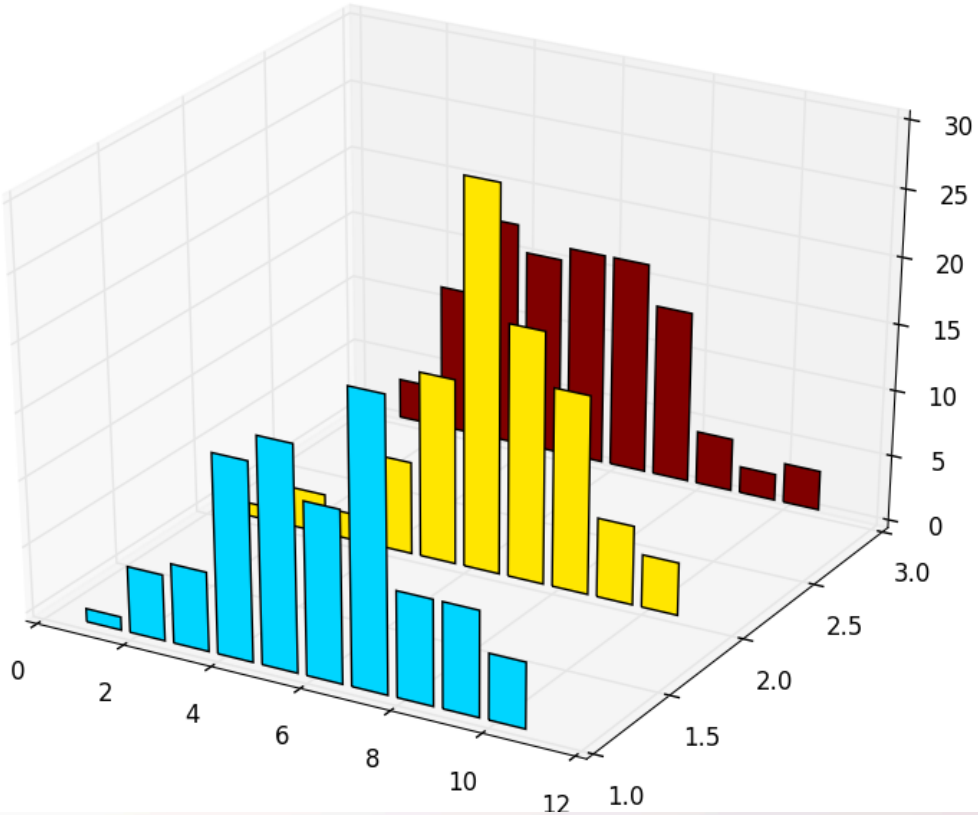
Exercise

- Create a 3D box plot using the data from the previous exercise (visualize only one data sample). Use the function **Axes3D.bar(x, y)** and the parameter **zdir**.



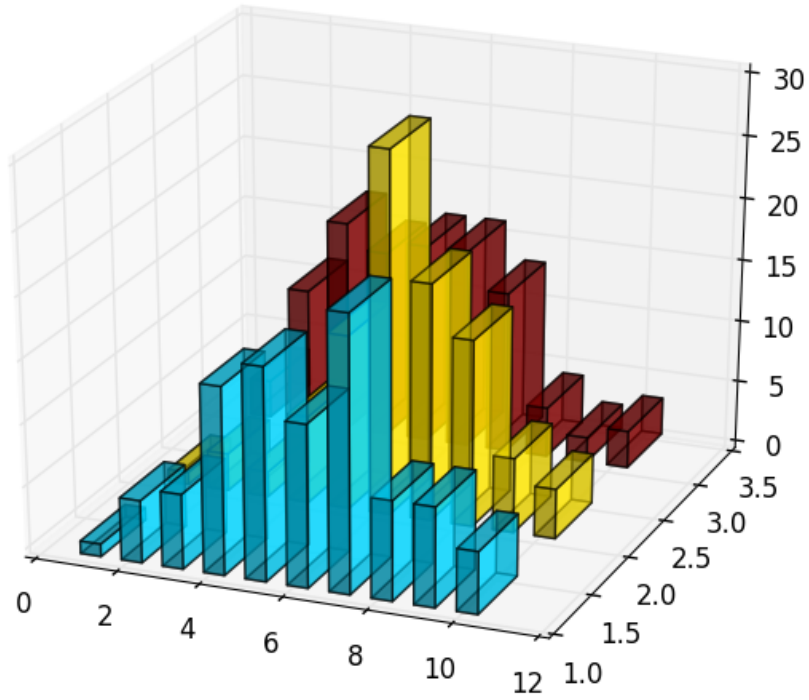
Exercise

- Use the functions **Axes3D.bar(x, y, z)** and **flatten()** to change the box plot to the one on the right. Use the color map **jet** and apply it direction of the y-axis.

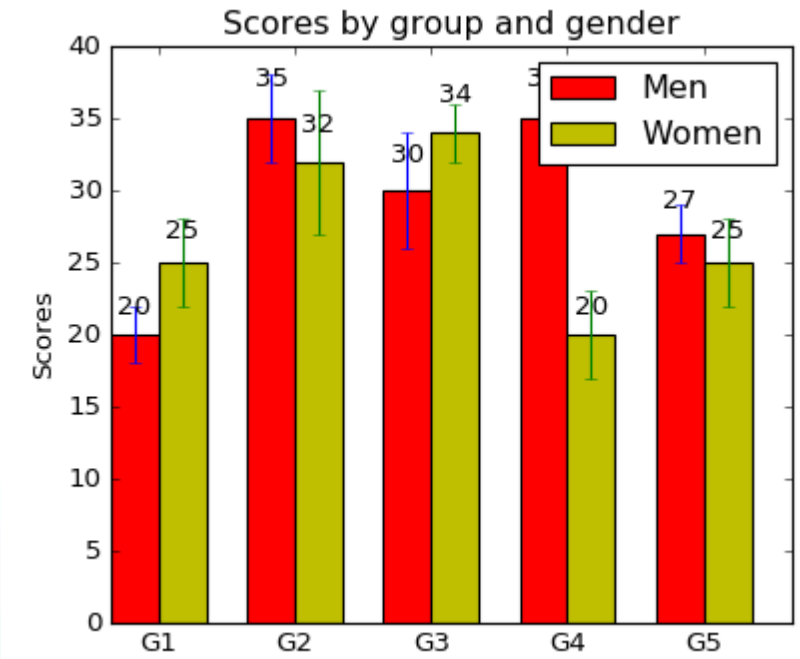


Exercise

- Use the functions **Axes3D.bar3d(), flatten()** and the parameter **alpha** to change the box plot to the one on the right.



plt.bar()



`matplotlib.pyplot.bar(left, height, width=0.8, bottom=None, hold=None, data=None, **kwargs)`

Make a bar plot.

Make a bar plot with rectangles bounded by:

`left, left + width, bottom, bottom + height`

(left, right, bottom and top edges)

Parameters:

left : sequence of scalars

the x coordinates of the left sides of the bars

height : sequence of scalars

the heights of the bars

width : scalar or array-like, optional

the width(s) of the bars default: 0.8

bottom : scalar or array-like, optional

the y coordinate(s) of the bars default: None

color : scalar or array-like, optional

the colors of the bar faces

edgecolor : scalar or array-like, optional

the colors of the bar edges

linewidth : scalar or array-like, optional

width of bar edge(s). If None, use default linewidth; If 0, don't draw edges. default: None

tick_label : string or array-like, optional

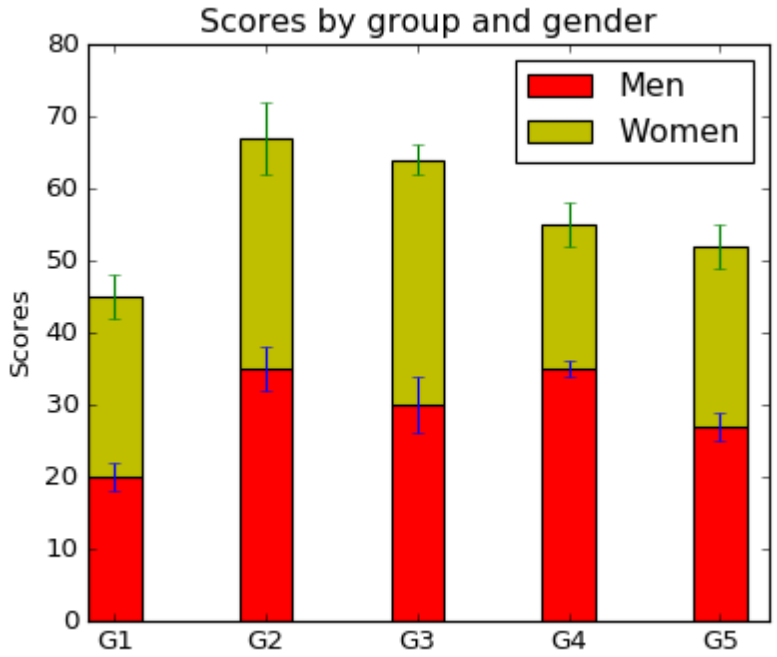
the tick labels of the bars default: None

xerr: scalar or array-like, optional

if not None, will be used to generate errorbar(s) on the bar chart default: None

yerr: scalar or array-like, optional

plt.bar()



matplotlib.pyplot.bar(*left*, *height*, *width=0.8*, *bottom=None*, *hold=None*, *data=None*, ***kwargs*)

Make a bar plot.

Make a bar plot with rectangles bounded by:

`left`, `left + width`, `bottom`, `bottom + height`

(left, right, bottom and top edges)

Parameters:

left : sequence of scalars

the x coordinates of the left sides of the bars

height : sequence of scalars

the heights of the bars

width : scalar or array-like, optional

the width(s) of the bars default: 0.8

bottom : scalar or array-like, optional

the y coordinate(s) of the bars default: None

color : scalar or array-like, optional

the colors of the bar faces

edgecolor : scalar or array-like, optional

the colors of the bar edges

linewidth : scalar or array-like, optional

width of bar edge(s). If None, use default linewidth; If 0, don't draw edges. default: None

tick_label : string or array-like, optional

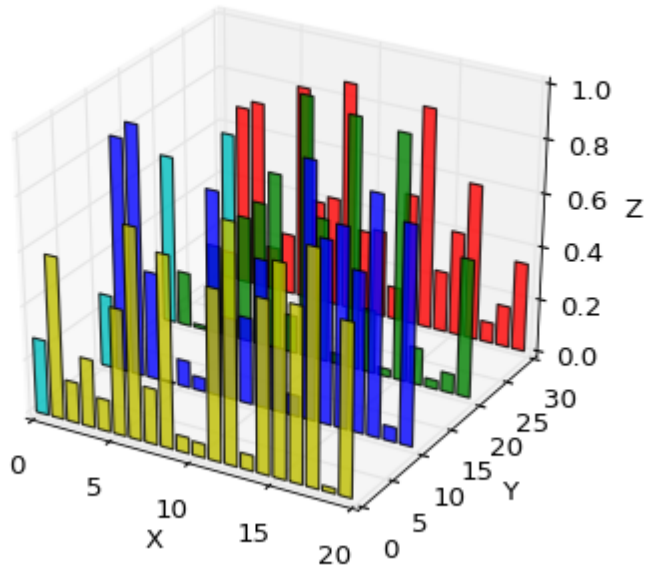
the tick labels of the bars default: None

xerr : scalar or array-like, optional

if not None, will be used to generate errorbar(s) on the bar chart default: None

yerr : scalar or array-like, optional

Axes3D.bar()



`bar(left, height, zs=0, zdir='z', *args, **kwargs)`

Add 2D bar(s).

Argument	Description
left	The x coordinates of the left sides of the bars.
height	The height of the bars.
zs	Z coordinate of bars, if one value is specified they will all be placed at the same z.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Keyword arguments are passed onto `bar()`.

Returns a `Patch3DCollection`

Axes3D.bar()

- Axes3D(plt.gcf()).bar(x, y, z)

`bar(left, height, zs=0, zdir='z', *args, **kwargs)`

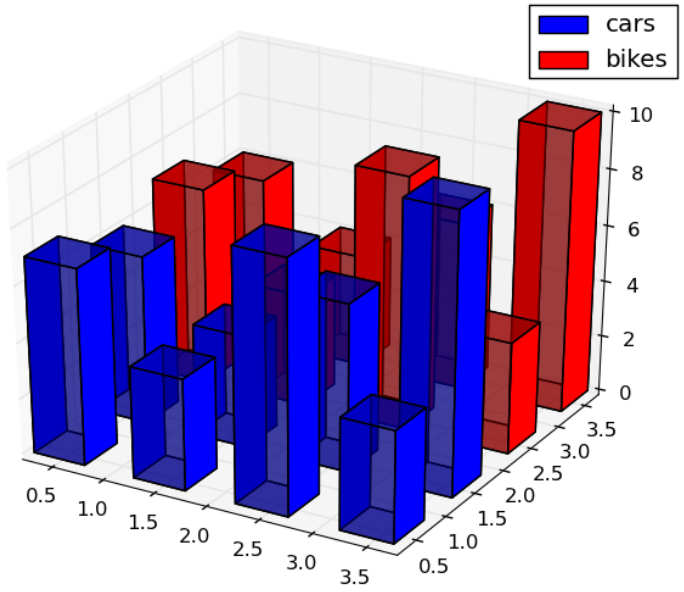
Add 2D bar(s).

Argument	Description
left	The x coordinates of the left sides of the bars.
height	The height of the bars.
zs	Z coordinate of bars, if one value is specified they will all be placed at the same z.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Keyword arguments are passed onto `bar()`.

Returns a `Patch3DCollection`

Axes3D.bar3d()



`bar3d(x, y, z, dx, dy, dz, color='b', zsort='average', *args, **kwargs)`

Generate a 3D bar, or multiple bars.

When generating multiple bars, x, y, z have to be arrays. dx, dy, dz can be arrays or scalars.

color can be:

- A single color value, to color all bars the same color.
- An array of colors of length N bars, to color each bar independently.
- An array of colors of length 6, to color the faces of the bars similarly.
- An array of colors of length 6 * N bars, to color each face independently.

When coloring the faces of the boxes specifically, this is the order of the coloring:

1. -Z (bottom of box)
2. +Z (top of box)
3. -Y
4. +Y
5. -X
6. +X

Keyword arguments are passed onto `Poly3DCollection()`

Axes3D.bar3d()

- Axes3D(plt.gcf()).bar3d(x, y, z, dx, dy, dz)
- All arrays have to be 1D!

`bar3d(x, y, z, dx, dy, dz, color='b', zsort='average', *args, **kwargs)`

Generate a 3D bar, or multiple bars.

When generating multiple bars, x, y, z have to be arrays. dx, dy, dz can be arrays or scalars.

color can be:

- A single color value, to color all bars the same color.
- An array of colors of length N bars, to color each bar independently.
- An array of colors of length 6, to color the faces of the bars similarly.
- An array of colors of length 6 * N bars, to color each face independently.

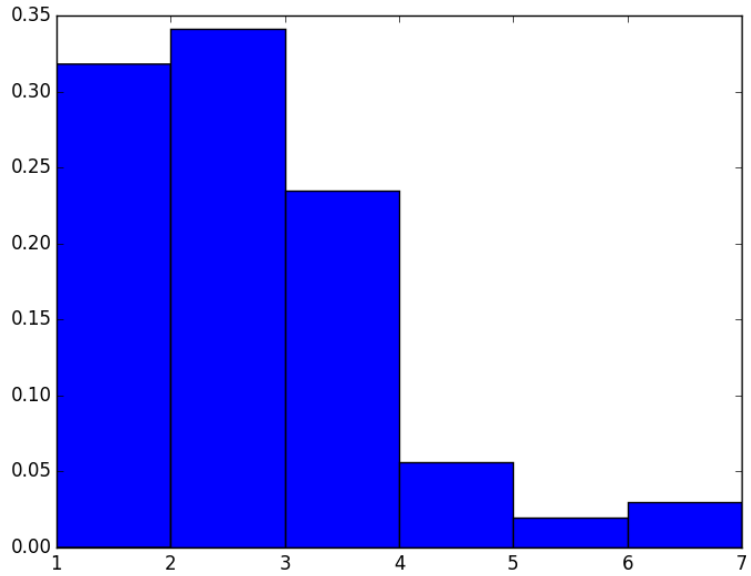
When coloring the faces of the boxes specifically, this is the order of the coloring:

1. -Z (bottom of box)
2. +Z (top of box)
3. -Y
4. +Y
5. -X
6. +X

Keyword arguments are passed onto `Poly3DCollection()`

Custom distribution functions

```
from scipy.stats import rv_discrete

xk = [1,2,3,4,5,6]
pk = [0.3,0.35,0.25,0.05,0.025,0.025]
new = rv_discrete(values=(xk,pk))
samples = new.rvs(size=1000)
```



Testing statistical hypotheses

scipy.stats.ttest_ind

scipy.stats.ttest_ind(*a*, *b*, *axis=0*, *equal_var=True*)¶ [source]

Calculates the T-test for the means of TWO INDEPENDENT samples of scores.

This is a two-sided test for the null hypothesis that 2 independent samples have identical average (expected) values. This test assumes that the populations have identical variances.

Parameters:

a, b : *array_like*
The arrays must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

axis : *int, optional*
Axis can equal None (ravel array first), or an integer (the axis over which to operate on a and b).

equal_var : *bool, optional*
If True (default), perform a standard independent 2 sample test that assumes equal population variances [R315]. If False, perform Welch's t-test, which does not assume equal population variance [R316].
New in version 0.11.0.

Returns:

t : *float or array*
The calculated t-statistic.

prob : *float or array*
The two-tailed p-value.

Statistical testing example

```
from scipy.stats import ttest_ind

data1, data2 = ...

stat, p = ttest_ind(data1, data2)
```

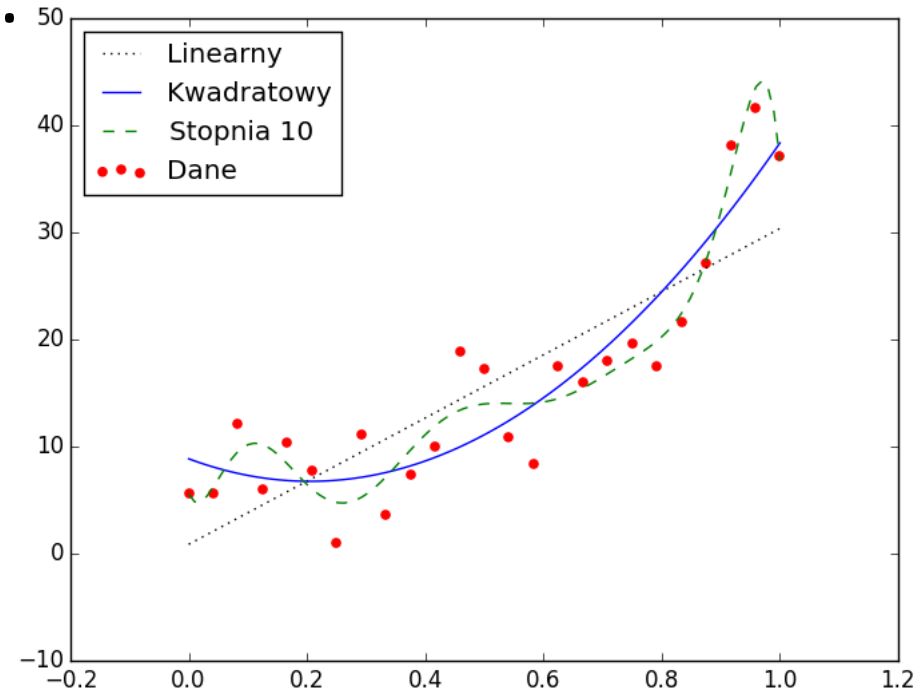
If we observe a large p-value, for example larger than 0.05 or 0.1, then we cannot reject the null hypothesis

Summary

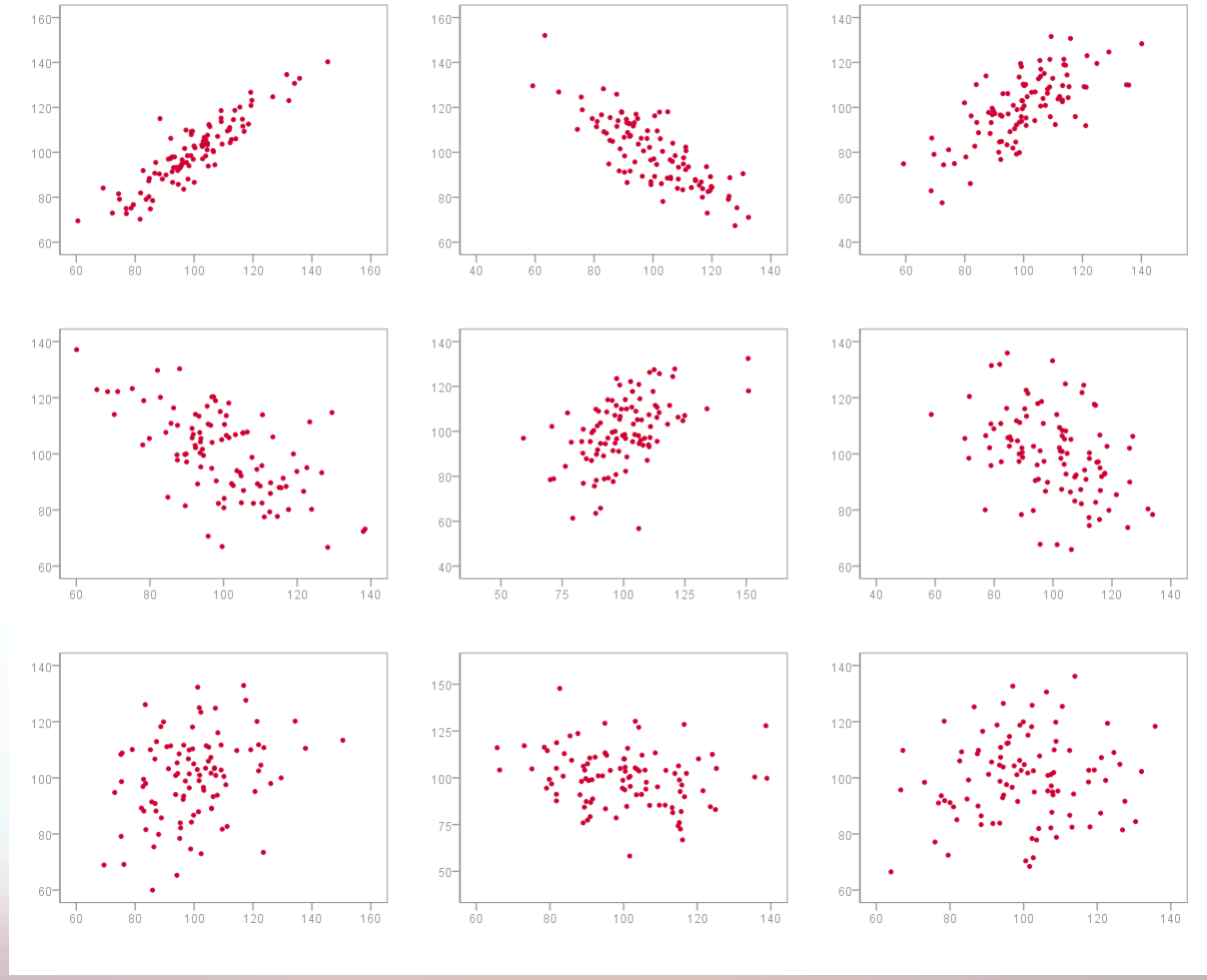
- Reduce dimensions of data
- Statistical inference (assess event size, quantify error, compare to random process)
- Create (regression) models of statistical data

Exercise

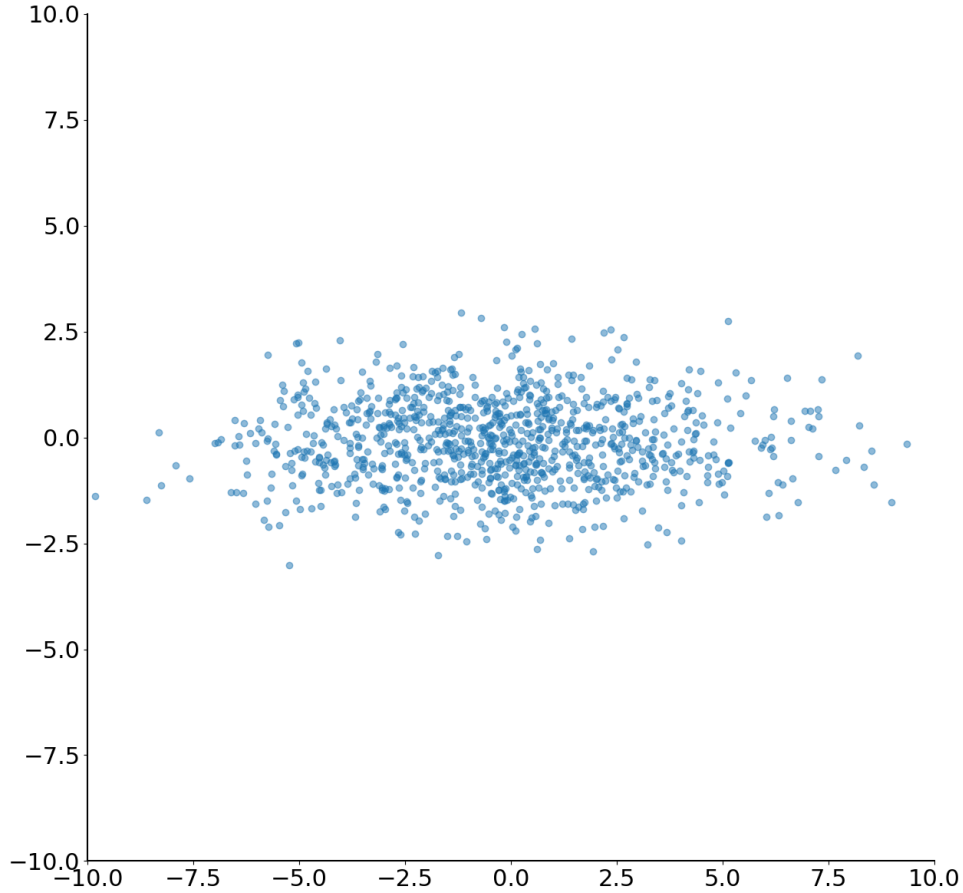
- Approximate the data points x and y with a polynomial (minimizing the least squares error): linear, quadratic, 10th order. Visualize the data points on a graph and draw the approximations.



Covariance



Scaling in x-direction?



$$\Sigma' = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix} = \begin{bmatrix} 16 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 5 & 0 \\ 0 & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

$$T = \sqrt{\Sigma'} = \begin{bmatrix} 4 & 0 \\ 0 & 1 \end{bmatrix}.$$