

# Math for Machine Learning

## Machine Learning Workshop @ MPSTME

Instructor: Santosh Chapaneri

## Linear Algebra; Probability; Statistics; Optimization

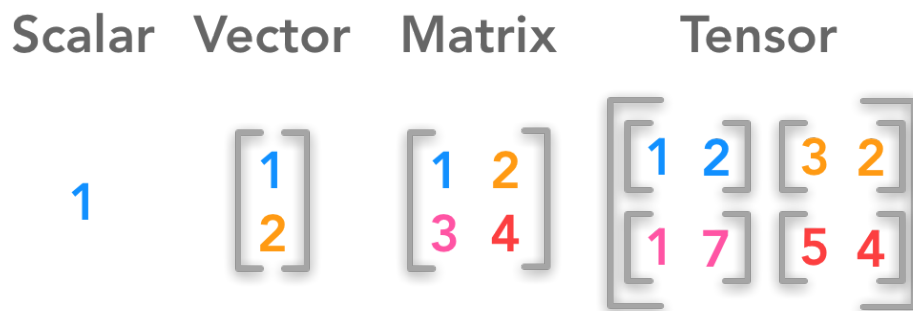
### Recommended Textbooks:

- G. Strang, *Linear Algebra and Its Applications*, Academic Press 1980
- I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIR Press 2016
- S. Boyd, *Convex Optimization*, Cambridge University Press 2004

## I. Linear Algebra

```
In [1]: from IPython.display import Image
        Image('Images/SVMT.png', width=600)
```

Out[1]:



## 1. Matrices Fundamentals

A matrix is a two-dimensional table. Here is an example of a  $3 \times 3$  matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

A vector is a  $n \times 1$  vector (there are **row** and **column** vectors).

## Distances and Norms

- Norm is a **qualitative measure of length of a vector** and is typically denoted as  $\|x\|$ .
- The norm should satisfy certain properties:
  - $\|\alpha x\| = |\alpha| \|x\|$ ,
  - $\|x + y\| \leq \|x\| + \|y\|$  (triangle inequality),
  - If  $\|x\| = 0$  then  $x = 0$ .
- The **distance** between two vectors is then defined as

$$d(x, y) = \|x - y\|$$

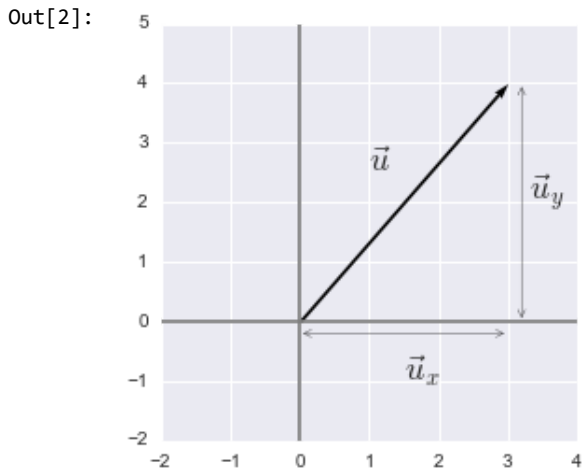
## Standard norms

The most well-known and widely used norm is **Euclidean norm**:

$$\|x\|_2 = \sqrt{\sum_{i=1}^n |x_i|^2},$$

which corresponds to the distance in our real life (the vectors might have complex elements, thus is the modulus here).

```
In [2]: from IPython.display import Image
Image('Images/L2Norm.png', width=300)
```



## $p$ -norm

Euclidean norm, or 2-norm, is a subclass of an important class of  $p$ -norms:

$$\|x\|_p = \left( \sum_{i=1}^n |x_i|^p \right)^{1/p}.$$

There are two very important special cases:

- Infinity norm, or Chebyshev norm which is defined as the maximal element:  $\|x\|_\infty = \max_i |x_i|$
- $L_1$  norm (or **Manhattan distance**) which is defined as the sum of modules of the elements of  $x$ :  $\|x\|_1 = \sum_i |x_i|$

## Computing Norms

The numpy package has all we need for computing norms (`np.linalg.norm` function)

[illegible]

# Matrix Norms

## How to measure distances between matrices?

**Frobenius** norm of the matrix:

$$\|A\|_F = \left( \sum_{i=1}^n \sum_{j=1}^m |a_{ij}|^2 \right)^{1/2}$$

Useful for computing objective function in machine learning for optimization

```
In [4]: n = 100
a = np.random.randn(n, n) # Random n x n matrix

norm_a = np.linalg.norm(a, 'fro') # Frobenius

print('Frobenius:', norm_a)

Frobenius: 98.98238763060469
```

## 2. Operations on Matrices

The **Inner Product** is defined as

$$\langle x, y \rangle = x^T y = \sum_{i=1}^n \bar{x}_i y_i,$$

where  $\bar{x}$  denotes the *complex conjugate* of  $x$ .

The Euclidean norm is then

$$\|x\|^2 = \langle x, x \rangle$$

=> the norm is **induced** by scalar product.

The **Outer Product** of vectors  $x$  and  $y$  is  $xy^T$  (matrix with rank 1).

**Remarks:** The **angle** between two vectors is defined as

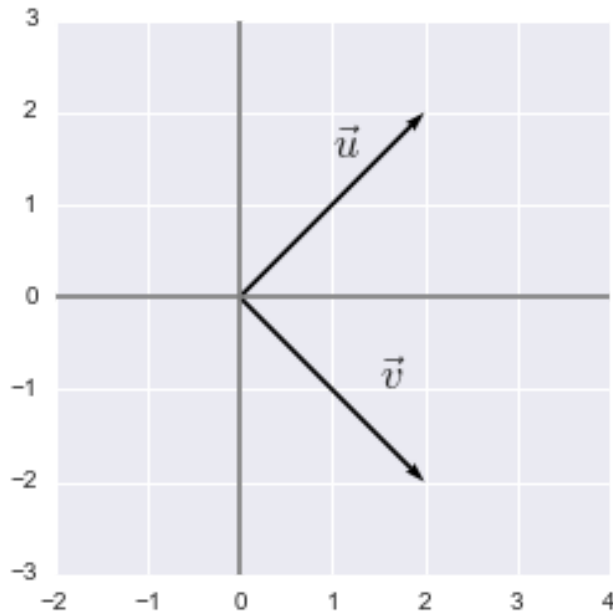
$$\cos \phi = \frac{\langle x, y \rangle}{\|x\| \|y\|}$$

Two vectors  $x$  and  $y$  are **Orthogonal** if  $\langle x, y \rangle = 0$ .

**Orthonormal vectors:** Vectors that satisfy following condition are orthonormal  $x_i^T x_j = 0$  when  $i \neq j$  and  $x_i^T x_i = 1$

In [5]: Image('Images/Orthogonal.png', width = 400)

Out[5]:



```
In [6]: x = np.array([1, 4, 0], float)
y = np.array([2, 2, 1], float)
print("x:", x)
print("y:", y)

print("Dot product of x and y:", np.dot(x, y))
print("Inner product of x and y:", np.inner(x, y))
print("Outer product of x and y:", np.outer(x, y))
print("Cross product of x and y:", np.cross(x, y))
# The Cross Product of two vectors is another vector that is
# at right angles to both
```

```
x: [1. 4. 0.]
y: [2. 2. 1.]
Dot product of x and y: 10.0
Inner product of x and y: 10.0
Outer product of x and y: [[2. 2. 1.]
 [8. 8. 4.]
 [0. 0. 0.]]
Cross product of x and y: [ 4. -1. -6.]
```

## Matrix Inverse and Transpose

```
In [7]: # Determinant of Matrix
n = 6
M = np.random.randint(100,size=(n,n))
print(M)
print('Determinant:',np.linalg.det(M))
```

```
[[49 88 77 40 78 76]
 [35 80 40 85 13 29]
 [ 7 74 69 35 61  8]
 [96 80 42 16 87 65]
 [24 47  1 57 93 26]
 [36 14 49 35 85 42]]
Determinant: 107531492377.00043
```

```
In [8]: # Inverse
A = np.random.randint(100,size=(5,5))
print(A)
print()
Ainv = np.linalg.inv(A)
print(Ainv)
```

```
[[61  6  0 64 58]
 [78 20 81 95 35]
 [57 41 32 29 49]
 [ 3 92 23 69 63]
 [14 25 70 47 70]]

[[ 0.00103756  0.00243244  0.01537544 -0.00524171 -0.00812118]
 [-0.00892033  0.00038418  0.01056552  0.00888594 -0.00819417]
 [-0.01106113  0.00544282  0.00261387 -0.00381301  0.00804552]
 [ 0.00702165  0.00802861 -0.01805784  0.00672609 -0.00324523]
 [ 0.00932492 -0.01145715  0.00266219 -0.00282829  0.01296986]]
```

```
In [9]: from IPython.display import Image
Image('Images/Transpose.png', width=300)
```

Out[9]:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

```
In [10]: # Matrix Transpose
A = np.array([[1,2,0], [3,5,9]])
print(A)
print()
print(A.T)
```

```
[[1 2 0]
 [3 5 9]]
```

```
[[1 3]
 [2 5]
 [0 9]]
```

## Moore-Penrose Pseudo-Inverse of a Matrix

$$A^+ = (A^H A)^{-1} A^H$$

```
In [11]: # Moore-Penrose pseudo-inverse of a non-square matrix
A = np.random.randn(5, 3)
B = np.linalg.pinv(A)

print(A)
print()
print(B)

# Verify
np.allclose(A, np.dot(A, np.dot(B, A)))

[[-0.05657874  1.38117289  0.10198001]
 [ 0.30139228 -2.18349837 -0.13083042]
 [ 0.09115627  0.01090333 -1.96365142]
 [ 0.98339875  0.13906339  0.48230502]
 [ 0.90294366 -0.11074621 -1.18896585]]

[[ 0.04931736  0.04872002 -0.1058513  0.62305502  0.42643188]
 [ 0.21081634 -0.32217067  0.02423116  0.07010307  0.04195099]
 [ 0.00276491  0.01877736 -0.37351909  0.17234944 -0.15609228]]
```

Out[11]: True

Used extensively in Machine Learning, for example, Extreme Learning Machine

```
In [12]: from IPython.display import Image
Image('Images/ELM.png')
```

Out[12]: Given a training set  $\mathcal{N} = \{(\mathbf{x}_i, \mathbf{t}_i) | \mathbf{x}_i \in \mathbf{R}^n, \mathbf{t}_i \in \mathbf{R}^m, i = 1, \dots, N\}$ , activation function  $g$ , and the number of hidden nodes  $L$ ,

- ① Assign randomly input weight vectors or centers  $\mathbf{a}_i$  and hidden node bias or impact factor  $b_i, i = 1, \dots, L$ .
- ② Calculate the hidden layer output matrix  $\mathbf{H}$ .
- ③ Calculate the output weight  $\beta$ :  $\beta = \mathbf{H}^\dagger \mathbf{T}$ .

where  $\mathbf{H}^\dagger$  is the Moore-Penrose generalized inverse of hidden layer output matrix  $\mathbf{H}$ .

## Matrix Multiplication

Consider composition of two linear operators:

1.  $y = Bx$
2.  $z = Ay$

Then,  $z = Ay = ABx = Cx$ , where  $C$  is the **matrix-by-matrix product**.

A product of an  $n \times k$  matrix  $A$  and a  $k \times m$  matrix  $B$  is a  $n \times m$  matrix  $C$  with the elements

$$c_{ij} = \sum_{s=1}^k a_{is} b_{sj}, \quad i = 1, \dots, n, \quad j = 1, \dots, m$$

```
In [13]: import numpy as np
def matmul(a, b):
    n = a.shape[0]
    k = a.shape[1]
    m = b.shape[1]
    c = np.zeros((n, m))
    for i in range(n):
        for j in range(m):
            for s in range(k):
                c[i, j] += a[i, s] * b[s, j]
```

```
In [14]: n = 1000
a = np.random.randn(n, n)
b = np.random.randn(n, n)

%timeit c = matmul(a, b)

%timeit c = np.dot(a, b)
```

724 ms ± 27.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

77.3 µs ± 6.39 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

### 3. Special Matrices

```
In [15]: from IPython.display import Image
Image('Images/DiagSymm.png', width=400)
```

Out[15]: **Diagonal matrix**      **Symmetric matrix**



### Diagonal Matrix

- Widely useful, eg. SVD

```
In [16]: import numpy as np
v = np.array([2, 4, 3, 1])
np.diag(v)
```

```
Out[16]: array([[2, 0, 0, 0],
               [0, 4, 0, 0],
               [0, 0, 3, 0],
               [0, 0, 0, 1]])
```

### Identity Matrix

```
In [17]: np.identity(3)
```

```
Out[17]: array([[1., 0., 0.],
               [0., 1., 0.],
               [0., 0., 1.]])
```

```
In [18]: np.identity(3, dtype=int)
```

```
Out[18]: array([[1, 0, 0],
               [0, 1, 0],
               [0, 0, 1]])
```

Show that for any matrix  $A$ ,  $AI = IA = A$

```
In [19]: A = np.array([[4,2,1],[4,8,3],[1,1,0]])
I = np.identity(3, dtype=int)
np.dot(A,I)
```

```
Out[19]: array([[4, 2, 1],
               [4, 8, 3],
               [1, 1, 0]])
```

```
In [20]: np.dot(A,I) == np.dot(I,A)
```

```
Out[20]: array([[ True,  True,  True],
               [ True,  True,  True],
               [ True,  True,  True]])
```

## Unitary / Orthogonal Matrices

A **Unitary** matrix is a matrix that when multiplied by its complex conjugate transpose matrix, equals the identity matrix.

$$U^H U = U U^H = I$$

When  $U^H = U^\top$ , the matrix is called **Orthogonal**.

**Product of two unitary matrices is a unitary matrix:**

$$(UV)^H UV = V^H (U^H U) V = V^H V = I$$

```
In [21]: a = 0.7
b = (1-a**2)**0.5

U = np.array([[a,b], [-b,a]])
print(U)
print()
print(U.dot(U.conj().T))

[[ 0.7          0.71414284]
 [-0.71414284  0.7          ]]

[[1.00000000e+00 1.59237766e-18]
 [1.59237766e-18 1.00000000e+00]]
```

## 4. Matrix Rank

- The maximum number of linearly independent rows in a matrix  $A$  is called the **row rank** of  $A$ , and the maximum number of linearly independent columns in  $A$  is called **column rank** of  $A$ .

```
In [22]: # Computing matrix rank
import numpy as np

n = 50
a1 = np.ones((n, n))
a2 = np.array([[1, 0, -1], [0, 1, 0], [1, 0, 1]])

print('Rank of the matrix:', np.linalg.matrix_rank(a1))
print('Rank of the matrix:', np.linalg.matrix_rank(a2))

b = a1 + 1e-6 * np.random.randn(n, n) # adding very small Gaussian noise
print('Rank of the matrix:', np.linalg.matrix_rank(b, tol=1e-8) )
# Boom!

Rank of the matrix: 1
Rank of the matrix: 3
Rank of the matrix: 49
```



## Stability and Condition Number

Example:

$$\begin{pmatrix} 8 & 6 & 4 & 1 \\ 1 & 4 & 5 & 1 \\ 8 & 4 & 1 & 1 \\ 1 & 4 & 3 & 6 \end{pmatrix} x = \begin{pmatrix} 19 \\ 11 \\ 14 \\ 14 \end{pmatrix}$$

In [23]: `import numpy.linalg as LA`

```
A = np.array([[8,6,4,1],[1,4,5,1],[8,4,1,1],[1,4,3,6]])
b = np.array([19,11,14,14])
LA.solve(A,b)
```

Out[23]: `array([1., 1., 1., 1.])`

In [24]: `# Introduce tiny perturbations`  
`b = np.array([19.01,11.05,14.07,14.05])`  
`LA.solve(A,b)`

Out[24]: `array([-2.34 , 9.745, -4.85 , -1.34 ])`

Note that the *tiny* perturbations in the outcome vector  $b$  cause large differences in the solution! When this happens, we say that the matrix  $A$  is **ill-conditioned**.

This happens when a matrix is close to being **singular** (i.e. non-invertible).

- The **Condition Number** is defined as:

$$\text{cond}(A) = \|A\| \cdot \|A^{-1}\|$$

- Also, defined as

$$\text{cond}(A) = \frac{\lambda_1}{\lambda_n}$$

where  $\lambda_1$  is the maximum singular value of  $A$  and  $\lambda_n$  is the smallest.

- The **higher the condition number, the more unstable the system**.

In [25]: `U, s, Vt = np.linalg.svd(A)`  
`print('Condition number of A: ', max(s)/min(s))`

Condition number of A: 3198.6725811994825

## 5. SVD (Singular Value Decomposition) of Matrix

In [26]: `Image('Images/SVDEq.png', width=400)`

Out[26]:

$$A = U D V^T$$

Left singular vectors

Singular values

Right singular vectors

In [27]: `Image('Images/SVDMatDim.png', width=400)`

Out[27]:

$$m \begin{bmatrix} & \\ & \end{bmatrix} = m \begin{bmatrix} & \\ & \end{bmatrix} m \begin{bmatrix} & \\ & \end{bmatrix} n \begin{bmatrix} & \\ & \end{bmatrix}$$

- The SVD decomposition is a factorization of a matrix, with many useful **applications in computer vision, signal processing and deep learning**.
- The SVD decomposition of a matrix  $A$  is of the form

$$A = U\Sigma V^T$$

- Since  $U$  and  $V$  are orthogonal (this means that  $U^T \times U = I$  and  $V^T \times V = I$ ) we can write the inverse of  $A$  as

$$A^{-1} = V\Sigma^{-1}U^T$$

```
In [28]: A = np.floor(np.random.rand(4,4)*20-10) # generating a random matrix
b = np.floor(np.random.rand(4,1)*20-10) # system Ax = b

print(A)
print(b)
```

```
[[ 2. -6. -4. -2.]
 [ 2. -8. -10.  4.]
 [ 2. -8. -8.  4.]
 [-9. -3.  4.  5.]]
[[-9.]
 [ 4.]
 [ 3.]
 [-3.]]
```

```
In [29]: U,s,Vt = np.linalg.svd(A) # SVD decomposition of A

# computing the inverse using pinv
pinv = np.linalg.pinv(A)

# computing the inverse using the SVD decomposition
pinv_svd = np.dot(np.dot(Vt.T, np.linalg.inv(np.diag(s))), U.T)

print("Inverse computed by lingal.pinv()\n",pinv)
print("Inverse computed using SVD\n",pinv_svd)
```

```
Inverse computed by lingal.pinv()
[[-7.44680851e-02 -4.46808511e-01  5.42553191e-01 -1.06382979e-01]
 [-1.22340426e-01  2.65957447e-01 -2.87234043e-01 -3.19148936e-02]
 [-2.33724540e-16 -5.00000000e-01  5.00000000e-01 -7.10485506e-17]
 [-2.07446809e-01 -2.44680851e-01  4.04255319e-01 -1.06382979e-02]]
Inverse computed using SVD
[[-7.44680851e-02 -4.46808511e-01  5.42553191e-01 -1.06382979e-01]
 [-1.22340426e-01  2.65957447e-01 -2.87234043e-01 -3.19148936e-02]
 [-2.34235608e-16 -5.00000000e-01  5.00000000e-01 -7.49112654e-17]
 [-2.07446809e-01 -2.44680851e-01  4.04255319e-01 -1.06382979e-02]]
```

Now, we can solve  $Ax = b$  using the inverse:

$$Ax = b \implies x = A^{-1}b$$

```
In [30]: x = np.linalg.solve(A, b) # solve Ax=b using Linalg.solve

xPinv = np.dot(pinv_svd, b) # solving Ax=b computing x = A^(-1)*b

print(x)
print(xPinv)

[[ 0.82978723]
 [ 1.39893617]
 [-0.5       ]
 [ 2.13297872]]
[[ 0.82978723]
 [ 1.39893617]
 [-0.5       ]
 [ 2.13297872]]
```

```
In [31]: # How much FAST is SVD for finding inverses?

n = 30

A = np.floor(np.random.rand(n,n)*20-10) # generating a random matrix
U, s, Vt = np.linalg.svd(A) # SVD decomposition of A

# computing the inverse using pinv
%timeit np.linalg.pinv(A)

# computing the inverse using the SVD decomposition
%timeit np.dot(np.dot(Vt.T, np.linalg.inv(np.diag(s))), U.T)

324 µs ± 3.12 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
187 µs ± 5.26 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

## Exercise

Write a function in Python to solve a system  $Ax = b$  using SVD decomposition.

- Your function should take  $A$  and  $b$  as input and return  $x$ .
- Your function should include the following:
  - First, check that  $A$  is invertible - return error message if it is not (*Hint*: product of singular values should be non-zero for invertibility)
  - Invert  $A$  using SVD and solve (Remember:  $A^{-1} = V\Sigma^{-1}U^T$ )
  - return  $x$

```
In [32]: def svdsolver(A, b):
          U, s, Vt = np.linalg.svd(A)
          if np.prod(s) == 0:
              print('Matrix is singular')
          else:
              return np.dot(np.dot((Vt.T).dot(np.diag(s**(-1))), U.T),b)

A = np.array([[1,1],[1,2]])
b = np.array([3,1])
print(np.linalg.solve(A,b))
print(svdsolver(A,b))

[ 5. -2.]
[ 5. -2.]
```

## 6. Eigen-things

### What is an eigenvector

An vector  $x \neq 0$  is called an **eigenvector** of a square matrix  $A$  if there exists a number  $\lambda$  such that

$$Ax = \lambda x.$$

The number  $\lambda$  is called an **eigenvalue**.

```
In [33]: from IPython.display import Image
Image('Images/Eigen.png', width=300)
```

```
Out[33]: 
$$\begin{bmatrix} 2 & 3 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 12 \\ 8 \end{bmatrix} = 4 \times \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

```

$A \cdot v = \lambda \cdot v$

```
In [1]: import numpy as np
import numpy.linalg as LA
A = np.diag((1, 2, 3))
print(A)

w, v = LA.eig(A)
print('Eigen Values:', w)
print('Eigen Vectors:')
print(v)
```

```
[[1 0 0]
 [0 2 0]
 [0 0 3]]
Eigen Values: [1. 2. 3.]
Eigen Vectors:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

```
In [2]: # Eigen-decomposition of Covariance matrix
mu = [0,0]
sigma = [[0.6,0.2], [0.2,0.2]]
n = 1000
x = np.random.multivariate_normal(mu, sigma, n).T

A = np.cov(x)
print(A)
```

```
[[0.63301579 0.22564912]
 [0.22564912 0.21385278]]
```

```
In [3]: w, v = LA.eig(A)
print(w)
print(v)
```

```
[0.73139846 0.11547011]
[[ 0.91666204 -0.39966324]
 [ 0.39966324  0.91666204]]
```

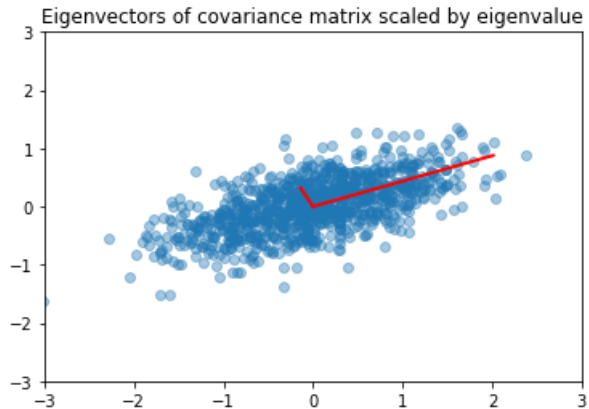
```
In [37]: ?zip
```

```
In [5]: import matplotlib.pyplot as plt

plt.scatter(x[0,:], x[1,:], alpha=0.4)

for evals, evects in zip(w, v.T):
    plt.plot([0, 3*evals*evects[0]], [0, 3*evals*evects[1]], 'r-', lw=2)

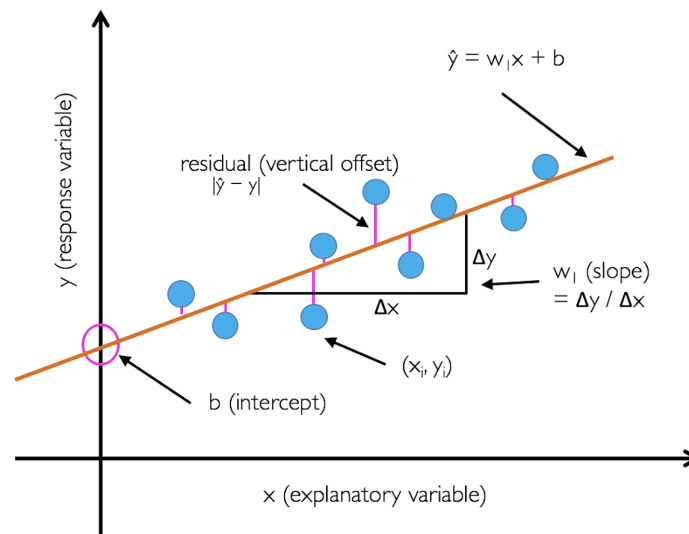
plt.axis([-3, 3, -3, 3])
plt.title('Eigenvectors of covariance matrix scaled by eigenvalue')
plt.show()
```



## 7. Application: Least-Squares Linear Regression

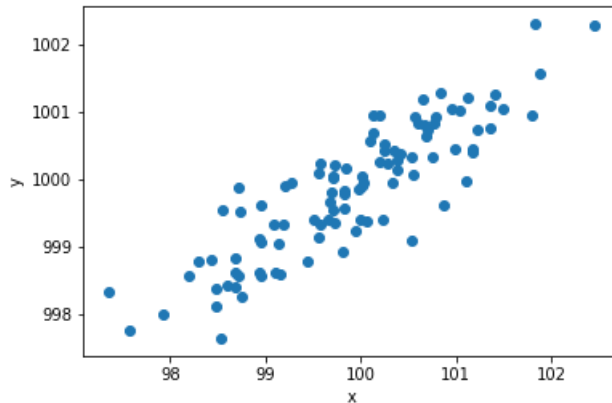
```
In [39]: import scipy.stats
import matplotlib.pyplot as plt
%matplotlib inline
```

- Fit slope and intercept so that the linear regression fit minimizes the sum of the residuals (vertical offsets or distances)



```
In [40]: rng = np.random.RandomState(123)
mean = [100, 1000]
cov = [[1, 0.9], [0.9, 1]]
sample = rng.multivariate_normal(mean, cov, size=100)
x, y = sample[:, 0], sample[:, 1]

plt.scatter(x, y)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



- Closed-form (analytical) solution:

$$L = \frac{1}{2} \sum_{i=1}^N (y_i - x_i^T w)^2 = \frac{1}{2} \|y - Xw\|^2 = \frac{1}{2} (y - Xw)^T (y - Xw)$$

$$\frac{\partial L}{\partial w} = -y^T X + w^T X^T X = 0$$

$$w = (X^T X)^{-1} X^T y$$

```
In [41]: # x.shape => (100,)
# newaxis: increase the dimension of existing array by one more dimension
X = x[:, np.newaxis]
# X.shape => (100,1)
print(X.shape[0])
```

100

```
In [42]: # adding a column vector of "ones"
# hstack: stack arrays in sequence horizontally (column wise)
Xb = np.hstack((np.ones((X.shape[0], 1)), X))
# Xb.shape => (100, 2); first column for bias, second column for X
w = np.zeros(Xb.shape[1])

# Closed-form solution
z = np.linalg.inv(np.dot(Xb.T, Xb))
w = np.dot(z, np.dot(Xb.T, y))

b, w1 = w[0], w[1]

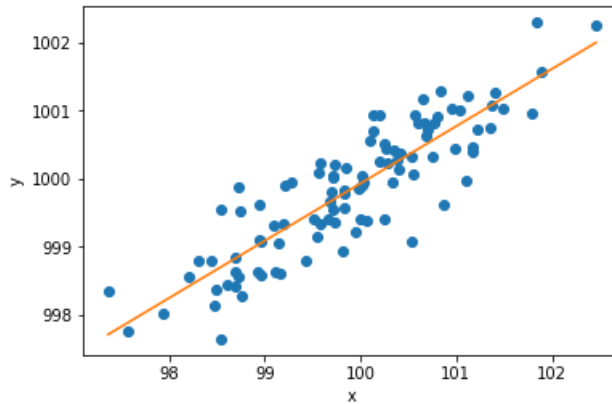
print('slope: %.2f' % w1)
print('y-intercept: %.2f' % b)
```

slope: 0.84  
y-intercept: 915.59

## Show line fit

```
In [43]: extremes = np.array([np.min(x), np.max(x)])
predict = extremes*w1 + b

plt.plot(x, y, marker='o', linestyle='')
plt.plot(extremes, predict)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



## Evaluate

### Mean squared error (MSE)

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

```
In [44]: y_predicted = x*w1 + b

mse = np.mean((y - y_predicted)**2)
mse
```

Out[44]: 0.21920128791623675

```
In [45]: rmse = np.sqrt(mse)
rmse
```

Out[45]: 0.46818937185313886

## II. Probability

```
In [46]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from scipy import stats
```

## Probability Mass Functions

- Goal of probability and uncertainty computations is to estimate population parameters from samples

## Bernoulli Trial

Bernoulli trial (or binomial trial): random experiment with 2 possible outcomes

```
In [47]: rng = np.random.RandomState(123)

coin_flips = rng.randint(0, 2, size=1000)
heads = np.sum(coin_flips)
heads
```

Out[47]: 520

```
In [48]: tails = coin_flips.shape[0] - heads
tails
```

Out[48]: 480

```
In [49]: rng = np.random.RandomState(123)

for i in range(7):
    num = 10**i
    coin_flips = rng.randint(0, 2, size=num)
    heads_proba = np.mean(coin_flips)
    print('Heads chance: %.2f' % (heads_proba*100))
```

```
Heads chance: 0.00
Heads chance: 40.00
Heads chance: 47.00
Heads chance: 53.70
Heads chance: 49.53
Heads chance: 49.80
Heads chance: 50.03
```

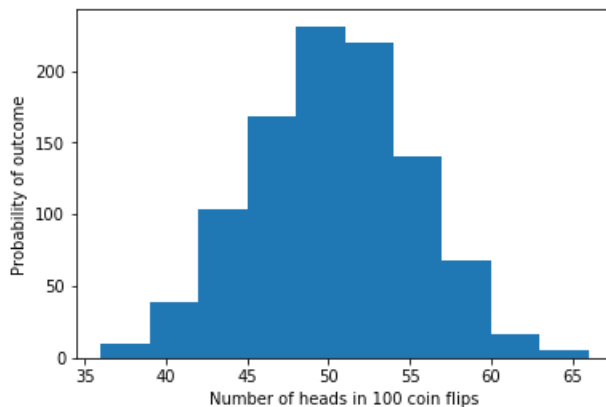
## Do 100 coin flips 1000 times:

```
In [50]: n_experiments = 1000
n_bernoulli_trials = 100

rng = np.random.RandomState(123)
outcomes = np.empty(n_experiments, dtype=np.float)

for i in range(n_experiments):
    coin_flips = rng.randint(0, 2, size=n_bernoulli_trials)
    head_counts = np.sum(coin_flips)
    outcomes[i] = head_counts

plt.hist(outcomes)
plt.xlabel('Number of heads in 100 coin flips')
plt.ylabel('Probability of outcome')
plt.show()
```





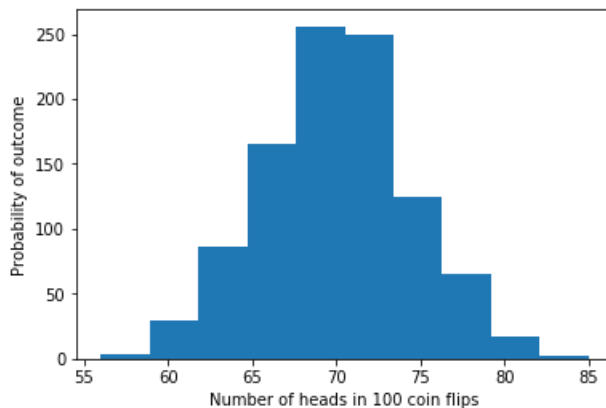
## Repeat with biased coin

```
In [51]: p = 0.7
n_experiments = 1000
n_bernoulli_trials = 100

rng = np.random.RandomState(123)
outcomes = np.empty(n_experiments, dtype=np.float)

for i in range(n_experiments):
    coin_flips = rng.rand(n_bernoulli_trials)
    head_counts = np.sum(coin_flips < p)
    outcomes[i] = head_counts

plt.hist(outcomes)
plt.xlabel('Number of heads in 100 coin flips')
plt.ylabel('Probability of outcome')
plt.show()
```



## Binomial Distribution

- Bernoulli trial (or binomial trial): random experiment with 2 possible outcomes
- a binomial distribution describes a binomial variable  $B(n, p)$  of  $n$  of Bernoulli trials (which are statistically independent);  $p$  is the probability of success (and  $q$  is the probability of failure,  $1-p$ )
- Probability of  $k$  successes:

$$P(k) = \binom{n}{k} p^k q^{n-k}$$

where  $\binom{n}{k}$  (" $n$  choose  $k$ ") is the binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

## Compute probability of 50 heads in 100 Bernoulli trials flipping a fair coin:

```
In [52]: def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

def combin(n, k): # "n choose k"
    return factorial(n) / factorial(k) / factorial(n - k)
```

```
In [53]: p = 0.5 # probability of success
n = 100 # n_trials
k = 50 # k_successes

proba = combin(n, k) * p**k * (1 - p)**(n - k)
proba
```

```
Out[53]: 0.07958923738717877
```

```
In [54]: # Direct method
rv = stats.binom(n, p)
rv.pmf(50)
```

```
Out[54]: 0.07958923738717888
```

## Probability Density Functions (PDFs)

- for working with continuous variables (vs. probability mass functions for discrete variables)
- here, the area under the curve give the probability (in contrast to probability mass functions where we have probabilities for every single value)
- the area under the whole curve is 1

## Normal Distribution (Gaussian Distribution)

### Probability Density Function of the Normal Distribution

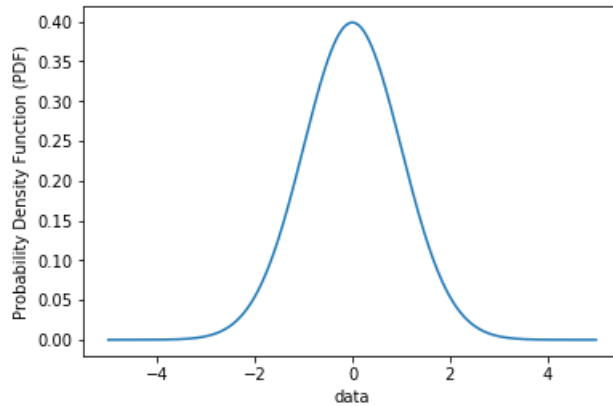
- unimodal and symmetric
- many algorithms in machine learning & statistics have normality assumptions
- two parameters: mean (center of the peak) and standard deviation (spread);  $\mu, \sigma$
- we can estimate parameters of  $\mathcal{N}(\mu, \sigma^2)$  by sample mean ( $\bar{x}$ ) and sample variance ( $s^2$ )
- univariate Normal distribution:

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

- standard normal distribution with zero mean and unit variance,  $\mathcal{N}(0, 1)$

```
In [55]: def univariate_gaussian_pdf(x, mean, variance):
          return (1. / np.sqrt(2*np.pi*variance) *
                  np.exp(- ((x - mean)**2 / 2.*variance)))

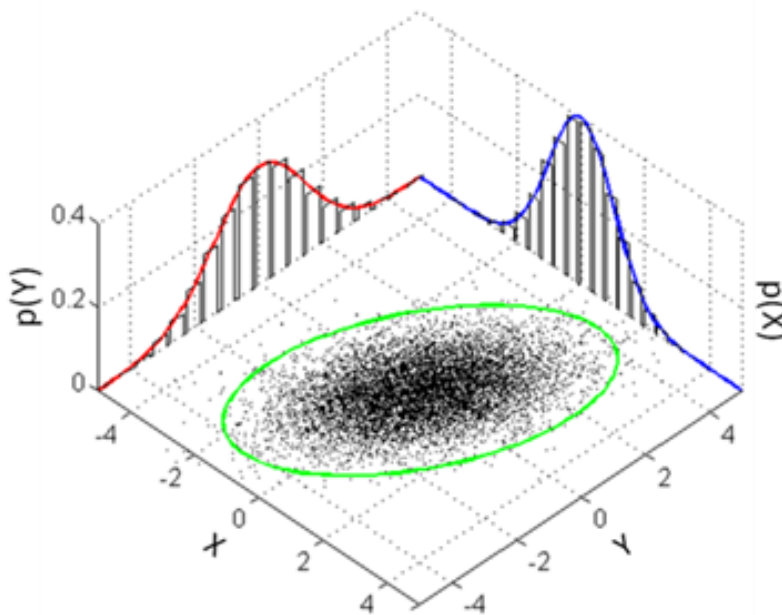
          mean = 0
          stddev = 1
          x = np.arange(-5, 5, 0.01)
          y = univariate_gaussian_pdf(x, mean, stddev**2)
          plt.plot(x, y)
          plt.xlabel('data')
          plt.ylabel('Probability Density Function (PDF)')
          plt.show()
```



## Application: Anomaly Detection

```
In [6]: from IPython.display import Image
        Image('Images/MVN.png', width=500)
```

Out[6]:

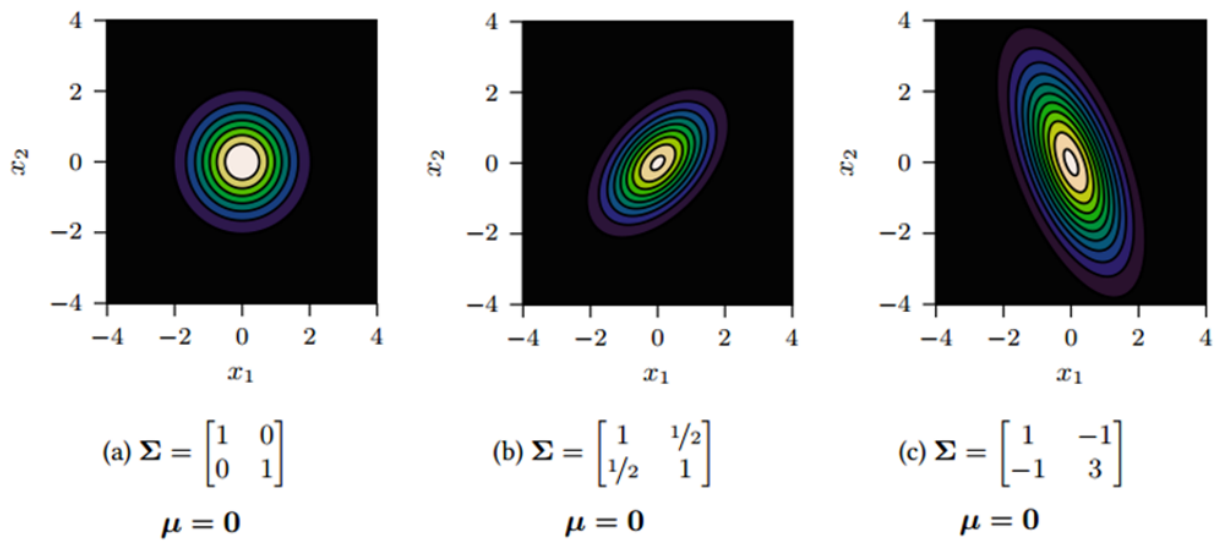


### Multi-variate Normal (MVN)

$$\mathcal{N}(x|\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^D}} \frac{1}{\sqrt{|\Sigma|}} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu)\right)$$

In [7]: `Image('Images/MVNCov.png', width=800)`

Out[7]:



```
In [56]: import matplotlib.pyplot as plt
import numpy as np
from numpy import genfromtxt
from scipy.stats import multivariate_normal
```

```
In [57]: def read_dataset(filePath,delimiter=','):
    return genfromtxt(filePath, delimiter=delimiter)

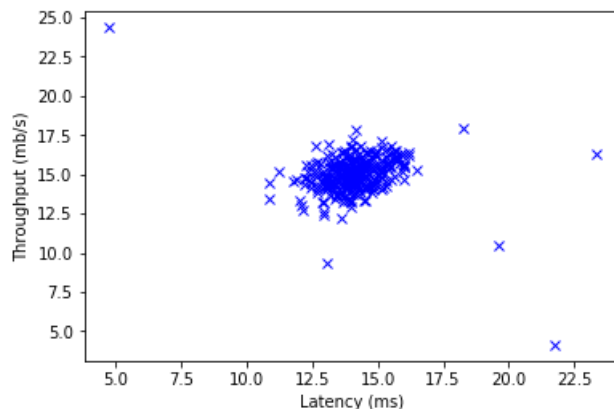
tr_data = read_dataset('Data/anomaly_detect_data.csv')
```

```
In [58]: n_training_samples = tr_data.shape[0]
n_dim = tr_data.shape[1]

print('Number of datapoints in training set: %d' % n_training_samples)
print('Number of dimensions/features: %d' % n_dim)
print(tr_data[1:5,:])
```

```
Number of datapoints in training set: 307
Number of dimensions/features: 2
[[13.409 13.763]
 [14.196 15.853]
 [14.915 16.174]
 [13.577 14.043]]
```

```
In [59]: plt.xlabel('Latency (ms)')
plt.ylabel('Throughput (mb/s)')
plt.plot(tr_data[:,0],tr_data[:,1],'bx')
plt.show()
```



```
In [60]: def estimateGaussian(dataset):
mu = np.mean(dataset, axis=0) # mean along each dimension / column
sigma = np.cov(dataset.T)
return mu, sigma

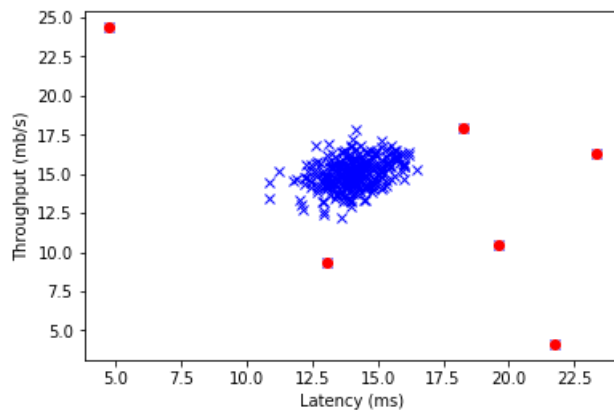
def multivariateGaussian(dataset,mu,sigma):
p = multivariate_normal(mean=mu, cov=sigma)
return p.pdf(dataset)
```

```
In [61]: mu, sigma = estimateGaussian(tr_data)
p = multivariateGaussian(tr_data,mu,sigma)
```

```
In [62]: thresh = 9e-05
# determining outliers/anomalies
outliers = np.asarray(np.where(p < thresh))
outliers
```

```
Out[62]: array([[300, 301, 303, 304, 305, 306]], dtype=int64)
```

```
In [63]: plt.figure()
plt.xlabel('Latency (ms)')
plt.ylabel('Throughput (mb/s)')
plt.plot(tr_data[:,0],tr_data[:,1], 'bx')
plt.plot(tr_data[outliers,0],tr_data[outliers,1], 'ro')
plt.show()
```



### III. Statistics

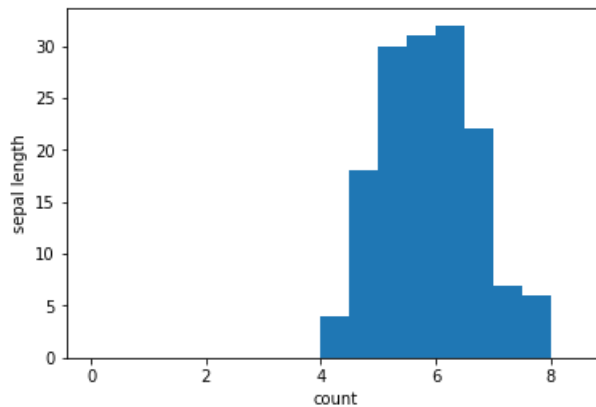
```
In [64]: import pandas as pd
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
%matplotlib inline

# read dataset
df = pd.read_csv('Data/iris.csv')

def histo():
    # create histogram
    bin_edges = np.arange(0, df['sepal_length'].max() + 1, 0.5)
    fig = plt.hist(df['sepal_length'], bins=bin_edges)

    # add plot labels
    plt.xlabel('count')
    plt.ylabel('sepal length')

histo()
plt.show()
```



### Sample Mean:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

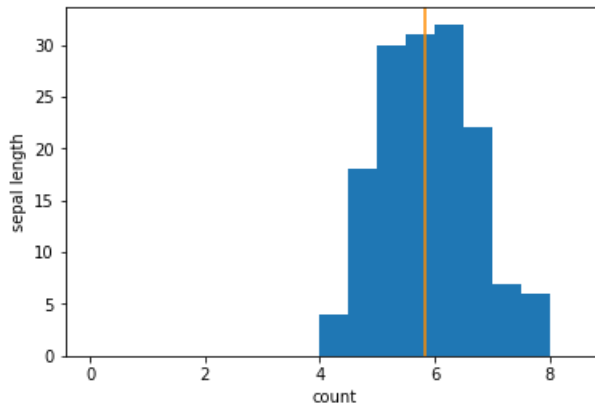
```
In [65]: x = df['sepal_length'].values
sum(i for i in x) / len(x)
```

```
Out[65]: 5.843333333333335
```

```
In [66]: x_mean = np.mean(x)
x_mean
```

```
Out[66]: 5.843333333333334
```

```
In [67]: histo()
plt.axvline(x_mean, color='darkorange')
plt.show()
```



### Sample Variance:

$$Var_x = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

- Bessel's correction to correct the bias of the population variance estimate
- Note the *unit* of the variable is now *unit*<sup>2</sup>

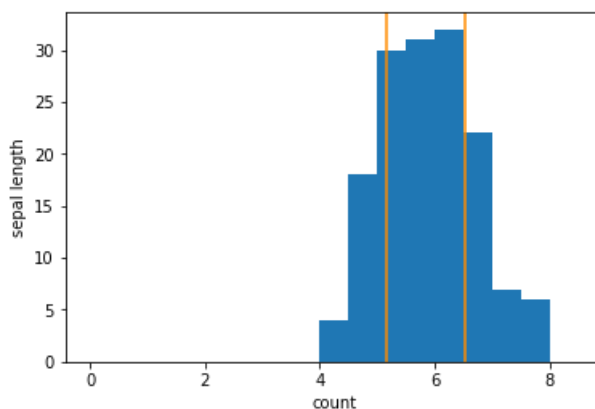
```
In [68]: sum([(i - x_mean)**2 for i in x]) / (len(x) - 1)
```

```
Out[68]: 0.6856935123042504
```

```
In [69]: var = np.var(x, ddof=1)
var
```

```
Out[69]: 0.6856935123042507
```

```
In [70]: histo()
plt.axvline(x_mean + var, color='darkorange')
plt.axvline(x_mean - var, color='darkorange')
plt.show()
```



### Sample Standard Deviation:

$$Std_x = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2}$$

```
In [71]: np.sqrt(np.var(x, ddof=1))
```

```
Out[71]: 0.828066127977863
```

```
In [72]: std = np.std(x, ddof=1)
std
```

```
Out[72]: 0.828066127977863
```

### Min/Max:

```
In [73]: print(np.min(x))
print(np.max(x))
```

```
4.3
7.9
```

### 25th and 75th Percentile:

```
In [74]: np.percentile(x, q=[25, 75], interpolation='lower')
```

```
Out[74]: array([5.1, 6.4])
```

### Median (50th Percentile):

```
In [75]: np.median(x)
```

```
Out[75]: 5.8
```

## Covariance and Correlation

```
In [76]: # read dataset
df = pd.read_csv('Data\iris.csv')
X = df[df.columns[:-1]].values
X.shape
```

```
Out[76]: (150, 4)
```

## Sample Covariance

- Measures how two variables differ from their mean
- Positive Covariance: that the two variables are both above or both below their respective means
  - variables are positively "correlated" -- they go up or down together
- Negative Covariance: valuables from one variable tends to be above the mean and the other below their mean
  - negative covariance means that if one variable goes up, the other variable goes down

$$\sigma_{x,y} = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$$

- note that similar to variance, the dimension of the covariance is  $unit^2$

```
In [77]: # Compute covariance between 2nd and 3rd feature:
```

```
x_mean, y_mean = np.mean(X[:, 2:4], axis=0)

sum([(x - x_mean) * (y - y_mean)
      for x, y in zip(X[:, 2], X[:, 3])]) / (X.shape[0] - 1)
```

```
Out[77]: 1.2956093959731545
```



Covariance matrix for the 4-feature dataset:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{1,2} & \sigma_{1,3} & \sigma_{1,4} \\ \sigma_{2,1} & \sigma_2^2 & \sigma_{2,3} & \sigma_{2,4} \\ \sigma_{3,1} & \sigma_{3,2} & \sigma_3^2 & \sigma_{4,3} \\ \sigma_{4,1} & \sigma_{4,2} & \sigma_{4,3} & \sigma_4^2 \end{bmatrix}$$

- Notice the variance along the diagonal
- Remember, the sample variance is computed as follows:

$$\sigma_x^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

```
In [78]: np.cov(X.T)
```

```
Out[78]: array([[ 0.68569351, -0.042434 ,  1.27431544,  0.51627069],
                [-0.042434 ,  0.18997942, -0.32965638, -0.12163937],
                [ 1.27431544, -0.32965638,  3.11627785,  1.2956094 ],
                [ 0.51627069, -0.12163937,  1.2956094 ,  0.58100626]])
```

## Pearson Correlation Coefficient

- Pearson correlation is "dimensionless" version of the covariance, achieved by dividing by the standard deviation
- Pearson correlation coefficient:

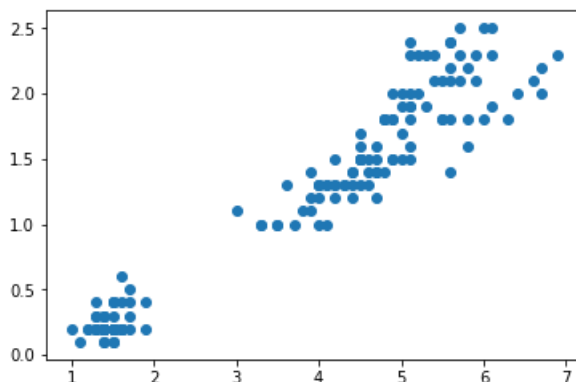
$$\rho_{x,y} = \frac{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\frac{1}{n-1} \sum_{i=1}^n (y_i - \bar{y})^2}}$$

$$= \frac{\sigma_{x,y}}{\sigma_x \sigma_y}$$

- Measures degree of a linear relationship between variables, assuming the variables follow a normal distribution
  - $\rho = 1$ : perfect positive correlation
  - $\rho = -1$ : perfect negative correlation
  - $\rho = 0$ : no correlation

```
In [79]: plt.scatter(X[:, 2], X[:, 3])
```

```
Out[79]: <matplotlib.collections.PathCollection at 0x1c77ec6f390>
```



```
In [80]: (np.cov(X[:, 2:4].T)[0, 1] /
          (np.std(X[:, 2], ddof=1) * np.std(X[:, 3], ddof=1)))
```

```
Out[80]: 0.9628654314027963
```

```
In [81]: np.corrcoef(X[:, 2:4].T)
```

```
Out[81]: array([[1.          ,  0.96286543],
                [0.96286543,  1.          ]])
```

```
In [82]: stats.pearsonr(X[:, 2], X[:, 3])
```

```
Out[82]: (0.9628654314027961, 4.675003907327543e-86)
```

The p-value roughly indicates the probability of an uncorrelated system producing datasets that have a Pearson correlation at least as extreme as the one computed from these datasets. The p-values are not entirely reliable but are probably reasonable for datasets larger than 500 or so.

(<https://docs.scipy.org/doc/scipy-0.19.0/reference/generated/scipy.stats.pearsonr.html> (<https://docs.scipy.org/doc/scipy-0.19.0/reference/generated/scipy.stats.pearsonr.html>))

## Scaled Variables

### Standardization

$$Z = \frac{X - \mu}{\sigma}$$

```
In [83]: from sklearn import preprocessing
import numpy as np
X_train = np.array([[ 1., -1.,  2.],[ 2.,  0.,  0.],[ 0.,  1., -1.]])
print(X_train)
X_scaled = preprocessing.scale(X_train)
print(X_scaled)

[[ 1. -1.  2.]
 [ 2.  0.  0.]
 [ 0.  1. -1.]]
[[ 0.          -1.22474487  1.33630621]
 [ 1.22474487  0.          -0.26726124]
 [-1.22474487  1.22474487 -1.06904497]]
```

- Scaled data has zero mean and unit variance:

```
In [84]: X_scaled.mean(axis=0)
```

```
Out[84]: array([0., 0., 0.])
```

```
In [85]: X_scaled.std(axis=0)
```

```
Out[85]: array([1., 1., 1.])
```

### Min-Max Scaler aka Normalization

$$Z = \frac{X - \min(X)}{\max(X) - \min(X)}$$

```
In [86]: X_train = np.array([[ 1., -1.,  2.],
...                          [ 2.,  0.,  0.],
...                          [ 0.,  1., -1.]])
...
min_max_scaler = preprocessing.MinMaxScaler()
X_train_minmax = min_max_scaler.fit_transform(X_train)
X_train_minmax
```

```
Out[86]: array([[0.5       , 0.        , 1.         ],
               [1.        , 0.5       , 0.33333333],
               [0.        , 1.        , 0.         ]])
```

The same instance of the transformer can then be applied to some new test data unseen during the fit call: the same scaling and shifting operations will be applied to be consistent with the transformation performed on the train data:

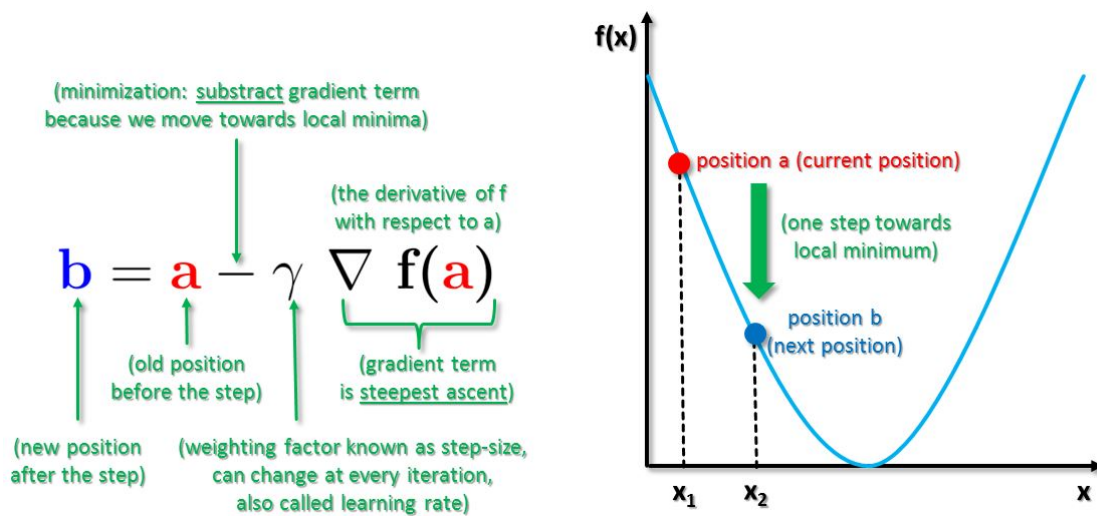
```
In [87]: X_test = np.array([[ -3., -1.,  4.]])
X_test_minmax = min_max_scaler.transform(X_test)
X_test_minmax

Out[87]: array([[ -1.5          ,  0.          ,  1.66666667]])
```

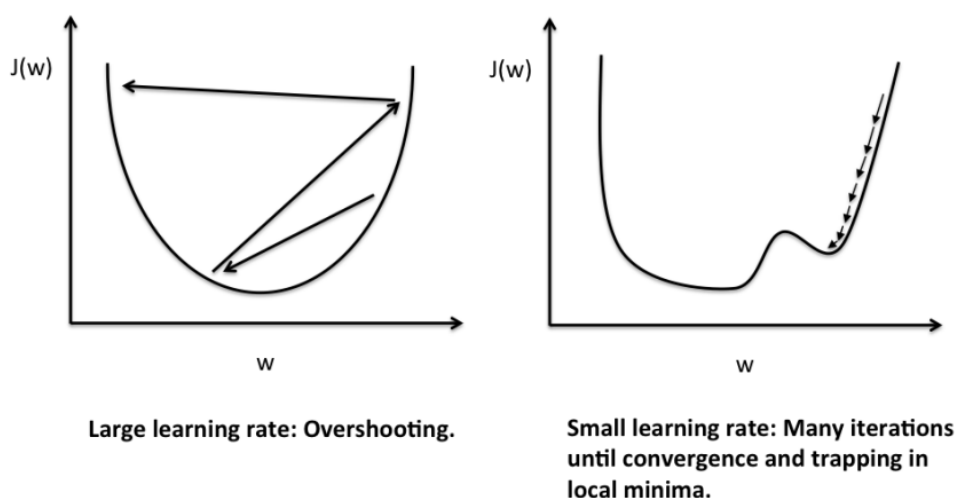
## IV. Optimization

### Gradient Descent

```
In [89]: from IPython.display import Image, display
display(Image(filename='Images/GradDescent.jpg', width=700))
```



```
In [90]: from IPython.display import Image, display
display(Image(filename='Images/learningrate.png', width=600))
```



## Batch Gradient Descent

Assume that we have a vector of parameters  $\theta$  and a cost function  $J(\theta)$  which is the variable we want to minimize (our objective function). Typically, the objective function has the form:

$$J(\theta) = \sum_{i=1}^m J_i(\theta)$$

where  $J_i$  is associated with the  $i$ -th observation in our data set.

- The batch gradient descent algorithm, starts with some initial feasible  $\theta$  (which we can either fix or assign randomly) and then repeatedly performs the update:

$$\theta := \theta - \eta \nabla_{\theta} J(\theta) = \theta - \eta \sum_{i=1}^m \nabla J_i(\theta)$$

where  $\eta$  is a constant controlling step-size and is called the learning rate.

- Note that in order to make a single update, we need to calculate the gradient using the **entire dataset**. This can be very **inefficient for large datasets**.
- In code, batch gradient descent looks like this:

```
for i in range(n_epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

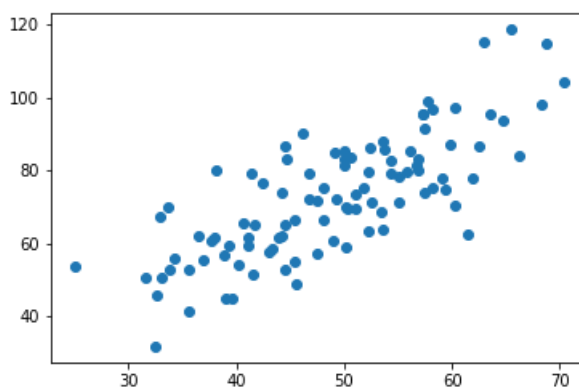
- For a given number of epochs  $n_{epochs}$ , we first evaluate the gradient vector of the loss function using **ALL** examples in the data set, and then we update the parameters with a given learning rate.
- Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

```
In [91]: import numpy as np
points = np.genfromtxt("Data/gd_data.csv", delimiter=",")
```

```
In [92]: import matplotlib.pyplot as plt
%matplotlib inline

plt.scatter(points[:,0],points[:,1])
```

```
Out[92]: <matplotlib.collections.PathCollection at 0x1c70792b6a0>
```



- Let's suppose we want to model the above set of points with a line.
- To do this we'll use the standard  $y = mx + b$  line equation where  $m$  is the line's slope and  $b$  is the line's  $y$ -intercept.
- To find the best line for our data, we need to find the best set of slope  $m$  and  $y$ -intercept  $b$  values.
- The error function is given by:

$$E = \frac{1}{N} \sum_{i=1}^N (y_i - (mx_i + b))^2$$

- The partial derivatives are given by:

$$\frac{\partial E}{\partial m} = \frac{2}{N} \sum_{i=1}^N -x_i (y_i - (mx_i + b))$$

$$\frac{\partial E}{\partial b} = \frac{2}{N} \sum_{i=1}^N -(y_i - (mx_i + b))$$

In [93]: `%run gradient_descent.py`

```
Starting gradient descent at b = 0, m = 0, error = 5565.107834483211
Running...
After 1000 iterations b = 0.08893651993741346, m = 1.4777440851894448, error = 112.61481011613473
<Figure size 432x288 with 0 Axes>
```

## Stochastic Gradient Descent (SGD)

- When we have very large data sets, the calculation of  $\nabla(J(\theta))$  can be costly as we must process every data point before making a single step (hence the name "batch").
- An alternative approach, the stochastic gradient descent method, is to update  $\theta$  sequentially with every observation. The updates then take the form:

$$\theta := \theta - \alpha \nabla_{\theta} J_i(\theta)$$

- This allows us to start making progress on the minimization problem right away. It is **computationally cheaper**, but it results in a **larger variance** of the loss function in comparison with GD.
- In code, the algorithm should look something like this:

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for example in data:
        params_grad = evaluate_gradient(loss_function, example, params)
        params = params - learning_rate * params_grad
```

- For a given epoch, we first reshuffle the data (to avoid bias from a particular order), and then for a single example, we evaluate the gradient of the loss function and then update the params with the chosen learning rate.

## Mini-batch SGD

- What if instead of single example from the dataset, we use a batch of data examples with a given size every time we calculate the gradient:

$$\theta = \theta - \eta \nabla_{\theta} J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

- Using mini-batches has the advantage that the **variance in the loss function is reduced**, while the **computational burden is still reasonable**, since we do not use the full dataset.
- The size of the mini-batches becomes another hyper-parameter of the problem. In standard implementations it ranges from 50 to 256.
- In code, mini-batch gradient descent looks like this:

```
for i in range(nb_epochs):  
    np.random.shuffle(data)  
    for batch in get_batches(data, batch_size=50):  
        params_grad = evaluate_gradient(loss_function, batch, params)  
        params = params - learning_rate * params_grad
```

- The difference with SGD is that for each update we use a batch of few examples (eg. 100) to estimate the gradient.