Linear Algebra Recap
==========

This is not a chapter where you can learn linear algebra from scratch. It is meant as a way to refresh your linear algebra knowledge. We will only consider the canonical finite dimensional vector space of vectors in $\setR^n$.

Vectors

.. sidebar:: **Real Vector Space**

   A *real vector space* is a set V of *vectors* with two operators (addition and scalar multiplication such that $\v x,\v y\in V: \v x+\v y\in V$ and $\v x\in V, a\in\setR:a\v x\in V$) and obeying the following requirements:

   1. Associativity of addition: $\v x + (\v y + \v z) = (\v x + \v y) + \v z$
   2. Commutativity of addition: $\v x + \v y = \v y + \v x$
   3. Identity element of addition (*zero vector*): $\exists \v 0 \in V: \forall \v x\in V: \v x + \v 0 = \v x$
   4. Inverse element of addition: $\forall \v x\in V: \exists -\v x\in V: \v x + (-\v x) = \v 0$
   5. Distributibity of scalar multiplication:

      * with respect to vector addition: $a(\v x+\v y)=a\v x + a\v y$

      * with respect to scalar addition: $(a+b)\v x = a\v x+b\v x$

   6. Compatibility of scalar and vector multiplication: $a(b)\v x= (ab)\v x$
   7. Identity element of scalar multiplication: $1 \v x = \v x$

   In this general setting (and in fact the scalars may come out of different *fields* like the complex numbers) vectors need not be the n-tuples with real numbers. We may also define the vector space of all *functions* and define the vector addition as $(f+g)(x)=f(x)+g(x)$ (note that the functions f and g are the vectors here!). The vector space of functions is stil a *real* vector space: the name

A vector $\v x$ in $\setR^n$ can be represented with an n-element vector

.. math::
   \v x = \matvec{c}{x_1\\x_2\\ \vdots\\ x_n}

each element (also called component or coordinate) x_i is a real value.

To sharpen your intuition for vectors consider a vector $\v x$ in two dimensions with elements a and b. This vector can be interpreted as either:

.. hier plaatje van de punt en pijl interpretatie

a point

  a and b are then the coordinates of a point (a,b) in the plane.

an arrow

  $\v x$ denotes the arrow starting in the origin (the point with coordinates $(0,0)$) and pointing to the point (a,b).

These interpretations will both be often used in practice.

Linear *algebra* is called an *algebra* because it defines way to make calculations with vectors and studies the properties of vectors and calculations with vectors.

The **addition** of two vectors $\v x$ and $\v y$ results in the vector $\v z$ whose elements are the sum of the corresponding elements of the vectors $\v x$ and $\v y$:

.. math::
  \v z &= \v x + \v y\\
  \matvec{c}{z_1\\z_2\\ \vdots\\ z_n}&=
  \matvec{c}{x_1\\x_2\\ \vdots\\ x_n}
  +
  \matvec{c}{y_1\\y_2\\ \vdots\\ y_n}

.. plaatje van vector additie in 2D

The addition of two vectors is easily interpreted as follows (we do a 2D example): take the arrow $\v x$ and draw it starting at the origin, next draw the arrow $\v y$ but start at the end point of $\v x$. These 'concatenated' arrows then end at the point $\v z$.

The **scalar multiplication** of a vector $\v x$ with a scalar α (i.e. a real value) is defined as the elementwise multiplication with α:

.. math::
  \alpha \v x = \alpha \matvec{c}{x_1\\x_2\\ \vdots\\ x_n} =
  \matvec{c}{\alpha x_1\\ \alpha x_2\\ \vdots\\ \alpha x_n}

Consider again the two dimensional example of the vector $\v x$ with elements a and b. The vector $\alpha \v x$, with $\alpha>0$ then is a vector *in the same direction* as $\v x$ but with a length that is α times the original lenght.

For $\alpha>1$ the length is increased, for $0<\alpha<1$ the length is

decreased. But what happens when we take a negative α? Consider the case:

.. math::
   -1 \v x = -\v x = -\matvec{c}{a\\b} = \matvec{c}{-a\\-b}

Indeed the direction is reversed, the vector now points in the opposite direction.

Note that **vector subtraction** $\v x - \v y$ simply is $\v x + (-\v y)$.

The **zero-vector** $\v 0$ is the vector of which all elements are zero. Remember from classical algebra that for any scalar a we have $a+0=a$. For the zero vector we have that for each vector $\v x$ it is true that: $\v x + \v 0=\v x$.

Vector addition is **commutative**, i.e. $\v x + \v y = \v y + \v x$.

Vector addition is **associative**, i.e. $(\v x + \v y) + \v z=\v x + (\v y + \v z)$.

Both these properties together imply that in a summation of vectors the order and sequence of the additions are irrelevant: you may leave out all brackets and change the order of vectors.

Basis Vectors

.. sidebar:: **Basis**

   A vector space V is an n-dimensional vector space in case n linear independent vectors $\v b_1,\ldots,\v b_n$ exist such that *any* element of V can uniquely be written as a linear combination of these *basis vectors*: $\v x = x_1 \v b_1 + \cdots + x_n \v b_n$.

   We say that the set of vectors $\v b_1,\ldots,\v b_n$ *span* the entire vector space V.

   For any n dimensional real vector space each vector in V can be represented with n real numbers: its *coordinates*. Carefully note that the coordinate representation of a vector thus depends on the basis that is, implicitly, used.

   And yes... infinite dimensional vector spaces do exist!

Consider the 2D vector $\v x=\matvec{cc}{3&2}\T$ [#transpose]_. When we draw it in a standard coordinate frame we mark the point that is 3 standard units from the right of the origin and 2 units above the origin.

Note that the vector $\v e_1 = \matvec{c}{1&0}\T$ is pointing one unit in the horizontal direction and the vector $\v e_2=\matvec{c}{0&1}\T$ is pointing one unit in the vertical direction. The vector $\v x$ may thus be written as:

.. math::
   \matvec{cc}{3\\2} &= 3\matvec{cc}{1\\0} + 2\matvec{cc}{0\\1}\\
   &= 3 \v e_1 + 2 \v e_2

In general a vector $\matvec{cc}{a&b}\T$ can be written as $a\v e_1+b\v e_2$. The vectors $\v e_1$ and $\v e_2$ are said to form a **basis** of $\setR^2$: any vector in $\setR^2$ can be written as the linear combination of the basis vectors.

But are the vectors $\v e_1$ and $\v e_2$ unique in that sense? Aren't there other vectors say $\v b_1$ and $\v b_2$ with the property that any vector can be written as a linear combination of these vectors? Yes there are! Any two vectors $\v b_1$ and $\v b_2$ that do not point in the same (or opposite) direction can be used as a basis.

For instance take $\v b_1=\matvec{cc}{1&1}$ and $\v b_2 = \matvec{cc}{-1&1}$ then we have that:

.. math::
   \matvec{c}{3\\2} = X \matvec{c}{1\\1} + Y \matvec{c}{-1\\1}

as can be checked by working out the vector expression in the right hand side of the above equality.

.. coordinates versus vectors

   from 2D to nD
     - linear independence

   coordinate transforms

Linear Mappings

.. sidebar:: **Linear Mappings in Vector Space**

   Note that although this section on linear mappings uses examples of $V=\setR^2$ the definitions and results are generic. They apply to any finite dimensional vector space.

   A linear mapping from a finite dimensional real vector space V to another finite dimensional real vector space V' can be represented with an $n\times n$ matrix. Again carefully note that

the matrix representation depends on the bases that are chosen for both V and V'.

A linear mapping is a mapping $\op L$ that takes a vector from $\setR^n$ and maps it on a vector from $\setR^m$ such that:

.. math::
  &\op L (\v x + \v y) = \op L\v x + \op L\v y\\
  &\op L (\alpha \v x) = \alpha \op L \v x

Consider a mapping $\op L$ that takes vectors from $\setR^2$ and maps these on vectors from $\setR^2$ as well. Remember that any vector in $\setR^2$ can be written as $\v x = x_1 \v e_1 + x_2 \v e_2$, then:

.. math::
  \op L \v x &= \op L (x_1 \v e_1 + x_2 \v e_2)\\
  &= x_1 \op L \v e_1 + x_2 \op L \v e_2

Note that $\op L \v e_1$ is the result of applying the mapping on vector $\v e_1$, by choice this is a vector in $\setR^2$ as well and therefore there must be two numbers L_{11} and L_{21} such that:

.. math::
  \op L \v e_1 = L_{11} \v e_1 + L_{21} \v e_2

Equivalently:

.. math::
  \op L \v e_2 = L_{12} \v e_1 + L_{22} \v e_2

So we get:

.. math::
  \op L \v x &= x_1 \op L \v e_1 + x_2 \op L \v e_2\\
  &= x_1 (L_{11} \v e_1 + L_{21} \v e_2) + x_2 (L_{12} \v e_1 + L_{22} \v e_2)\\
  &= (L_{11} x_1 + L_{12} x_2) \v e_1 + (L_{21} x_1 + L_{22} x_2) \v e_2

Let $\op L\v x = \matvec{cc}{x'_1&x'_2}\T$ be the coordinates with respect to the elementary basis, then we have:

.. math::
  \matvec{c}{x'_1\\x'_2} = \matvec{c}{L_{11} x_1 + L_{12} x_2\\L_{21} x_1 + L_{22} x_2}

The four scalars L_{ij} for $i,j=1,2$ thus completely define this linear mapping. It is customary to write these scalars in a **matrix**:

.. math::
  L = \matvec{cc}{L_{11}&L_{12}\\L_{21}&L_{22}}

For now a matrix is a symbolic notation, we will turn matrices and vectors into a real algebra in the next subsection when we consider calculations with matrices and vectors. With the matrix L we can define the matrix-vector product that describes the relation between the original vector $\v x$ and the resulting vector $\v x'$:

.. math::
   \matvec{c}{x'_1\\x'_2} =
   \matvec{cc}{L_{11}&L_{12}\\L_{21}&L_{22}}\matvec{c}{x_1\\x_2}

or equivalently:

.. math::
   \v x' = L \v x

Note that we may identify a linear mapping $\op L$ with its matrix L
(as is often done) but then carefully note that the basis for the
input vectors and the basis for the output vectors are implicitly
defined.

Matrices

A matrix A is a rectangular array of numbers. The elements A_{ij}
in the matrix are indexed with the row number i and the column
number j. The size of a matrix is denoted as $m\times n$ where m
is the number of rows and n is the number of columns.

Matrix Addition. Two matrices A and B can only be added
together in case both are of size $m\times n$. Then we have:

.. math::
   C &= A + B\\
   C_{ij} &= A_{ij} + B_{ij}

.. sidebar:: **Matrix Product**

Matrix Multiplication Two matrices A and B can be multiplied
to give $C=AB$ in case A is of size $m\times n$ and B is of size
$n\times p$. The resulting matrix C is of size $m\times p$. The
matrix multiplication is defined as:

.. math::
   C &= AB\\
   C_{ij} &= \sum_{k=1}^{n} A_{ik} B_{kj}

Matrix-Vector Multiplication. We may consider a vector $\v x$ in
$\setR^n$ as a matrix of size $n\times 1$. In case A is a matrix of
size $m\times n$ the matrix-vector multiplication $A\v x$ results in a
vector in m dimensional space (or a matrix of size $m\times 1$.

In a previous section we have already seen that any linear mapping
from n dimensional vector space to n dimensional vector space can
be represented with a $m\times n$ matrix.

Dot Product. Consider two vectors $\v x$ and $\v y$ both in
$\setR^n$, then the dot product is defined as the sum of the
multiplications of the elements of both vectors:

.. math::
   \v x \cdot \v y = \sum_{i=1}^n x_i y_i

note that we can also write:

.. math::
   \v x \cdot \v y = \v x\T\v y = \v y\T \v x

Matrix Transpose. Consider a matrix A of size $m\times n$, then the *transpose* $A\T$ is the matrix where the rows and columns of A are interchanged:

.. math::
   (A\T)_{ij} = A_{ji}

note that $A\T$ has size $n\times m$. The transpose is the operator that turns a column vector into a row vector (and vice versa).

Symmetric Matrices. In case $A\T = A$ the matrix A is called symmetric.

Square Matrices. A matrix is square if $m=n$.

Diagonal Matrix. A square matrix with only non zero elements on the main diagonal (i.e. the elements A_{ii}).

Identity Matrix. The diagonal matrix with all diagonal elements equal to one. The diagonal matrix of size $m\times m$ is denoted as I_m (the size subscript is omitted in case this is clear from the context).

Interpreted as a linear mapping, the identity matrix corresponds with the identity mapping (i.e. $I\v x = \v x$ for all $\v x$).

Linear Equations, Determinants and Inverse Matrices

The matrix-vector equation $A\v x= \v b$ where A is a known matrix of size $m\times n$, $\v x$ is the unknown column vector of size $n\times 1$ and $\v b$ is a known vector of size $m\times 1$, defines a set of m equations in n unknowns:

.. math::
   A_{11} x_1 + A_{12} x_2 + \cdots + A_{1n} x_n &= b_1\\
   A_{21} x_1 + A_{22} x_2 + \cdots + A_{2n} x_n &= b_2\\
   \vdots\\
   A_{m1} x_1 + A_{12} x_2 + \cdots + A_{mn} x_n &= b_m

A necessary condition for such a set of equations to have a unique solution is that $m=n$, i.e. is A is a **square matrix**.

In case the **determinant** $|A|$ is not equal to zero the matrix is called **non-singular** and an **inverse matrix** $A\inv$ exists. For the inverse of a matrix A (if it exists) we have that $A A\inv = A\inv A = I$.

A set of linear equations and represented with $A\v x = \v b$ thus has a solution $\v x = A\inv \v b$. Please note that solving a set of linear equation by calculating the inverse matrix is not a wise thing to do. Better ways to solve such a system are known and available in most linear algebra packages.

Vector Norm, Inner Products and Orthogonality

Let $\v x$ be a vector in 2D space with coordinate representation $\v x = \matvec{cc}{x_1&x_2}\T$ with respect to orthogonal basis vectors of unit length (i.e. an **orthonormal basis**), then we may use Pythogoras' theorem to calculate the length of the vector, which is denoted as $\|\v x\|$ and is called the **vector norm**:

.. math::
   \|\v x\| = \sqrt{x_1^2+x_2^2}

The **inner product** $\langle \v x, \v y \rangle$ of two vector $\v x$ and $\v y$ relates the length of the vectors with the angle between the vectors:

.. math::
   \langle \v x, \v y \rangle = \|\v x\| \|\v y\| \cos\theta

In a vector space where the coordinates are relative to an orthonormal basis we have that

.. math::
   \|\v x\| = \sqrt{\langle \v x, \v x \rangle}

In case the angle between the two vectors is 90 degrees (or $\half\pi$ radians), i.e. the vectors are **orthogonal**, we have that $\v x\T\v y=0$.

An $n\times n$ matrix A whose columns form an *orthonormal* basis is called an **orthogonal matrix**. Interpreted as linear mappings an orthogonal matrix preserves the length (norm) of vectors: $\|A\v x\|=\|\v x\|$. The **inverse of an orthogonal matrix** equals the transpose: $A\inv = A\T$.

Eigenvectors and Eigenvalues

Consider a linear mapping from $\setR^n$ onto itself. Such a mapping can be represented with a square $n\times n$ matrix A. In case for a

specific vector $\v x$ we have that:

.. math::
   A\v x = \lambda\v x

we say that $\v x$ is an **eigenvector** of A with corresponding **eigenvalue** λ.

Let A be a symmetric $n\times n$ matrix then it can be shown that there are n eigenvectors that form an orthonormal basis of A. Let $\v u_i$ for $i=1,\ldots,n$ be the n eigenvectors with corresponding eigenvalues λ_i then we can write:

.. math::
   A = U D U\inv

where $U = \matvec{c}{\v u_1,\ldots,\v u_n}$ is the matrix whose columns are the n eigenvectors and D is a diagonal matrix whose entries are the corresponding eigenvalues. The above equation is called the **eigenvalue decomposition**.

Singular Value Decomposition

Given any matrix A, the **singular value decomposition (SVD)** writes the matrix as a product of a orthogonal matrix U, a 'diagonal' matrix D and a second orthogonal matrix V:

.. math::
   \underbrace{A}_{m\times n} =
   \underbrace{U}_{m\times m} \quad
   \underbrace{D}_{m\times n}\quad \underbrace{V\T}_{n\times n}

Evidently the matrix D cannot be truly diagonal in general as A is not necessarily a square matrix. It is nevertheless called diagonal because only the elements D_{ii} can be different from zero.

.. [#transpose] The $\v x\T$ denotes the *transpose* of a vector. For now you can read it as a way to make a column vector out of a row vector.

Multivariate Functions
==========

In programming we are accustomed to the fact that a function may take more then one argument to produce a result. For mathematical functions we also have functions with more then one argument: *multivariate

functions*. As a simple example consider

.. math::
   f(x,y) = x^2+y.

Plotting a multivariate function

For a univariate function (one argument) plotting the graph of the function is easy. These are the function plots that you are all familiar with.

.. math::
   f(x) = x^2

.. exec_python:: parabola session_multivariate
   :linenumbers:
   :code: show
   :code_label: Show code for figure
   :results: hide

   import numpy as np
   import matplotlib.pyplot as plt

   x = np.linspace(-3, 3, 1000)
   plt.clf()
   plt.plot(x, x**2)
   plt.savefig('source/images/parabola.png')

Multivariate functions in two arguments (we will call them 2D functions) are possible (given our capabilities of interpreting 2D drawings of 3D objects).

.. math::
   f(x,y) = x^2+y.

.. exec_python:: fxy session_multivariate
   :linenumbers:
   :code: show
   :code_label: Show code for figure
   :results: hide

   x,y = np.meshgrid(np.linspace(-3, 3, 50), np.linspace(-3, 3, 50));
   z = x**2 + y;
   ax = plt.figure().add_subplot(projection='3d')
   ax.plot_surface(x, y, z, linewidth=0,

```
                cmap=plt.cm.copper, rstride=1, cstride=1, shade=True);
plt.savefig('source/images/func2d.png')
```

.. sidebar:: **Parameterized functions**

   For a fixed value $y=a$ we may write $f_a(x)=f(x,a)=x^2+a$ which you might recognize as a *parameterized family of functions*. For every value of a we have a new function. A 2D function can be viewed in a lot of ways as a parameterized family. We could fix x instead of y. Doing this we see that for any a the function $f(a,y)$ is a linear function: $f(a,y) = a^2+y$.

The shape of the function surface is easy to understand from the function recipe. For a $y=0$ we have $f(x,0)=x^2$ and as a function of x that is the parabola. For any value $y=a$ we have a parabola in x: $f(x,a)=x^2+a$. For any value $x=a$ we have straight line in y: $f(a,y)=a^2+y$.

Math doesn't end with 2D functions. In fact in many branches of computer science (statistical learning techniques for example) functions with hundreds of arguments are quite common. But our methods of visualizing multivariate functions do end with 2D functions. Beyond that function (data) visualization involves some interpretation too, we have to select how to render the information in a 3D space that can be visualized. We refer to lectures on scientific visualization.

Differentiating a multivariate function

.. sidebar:: **Continuity and Differentiability**

   A continuous 1D function is a function such that a very small change in the x value (say $x+dx$) leads to a very small change in the function value ($f(x+dx)=f(x)+df$). The notion of ''very small'' can be given a rigerous definition in mathematics and we then refer to *infinitesimals*.

   Note that not all continuous functions are differentiable. For instance the function $|x|$ (the absolute value function) is not differentiable because for $x=0$ the derivative is not uniquely defined. 'Looking' to the left from $x=0$ the function has a slope -1 and looking to the right the slope is $+1$.

   For more details see any textbook on calculus or look at Wikipedia for 'continuity'_ and 'differentiability'_.

   .. _continuity: http://en.wikipedia.org/wiki/Continuous_function

   .. _differentiability: http://en.wikipedia.org/wiki/Derivative

Remember the derivative of a univariate function:

.. math::

   \frac{df}{dx}(x) = \lim_{h\rightarrow0} \frac{f(x+h)-f(x)}{h}

So the derivative measures something like the rate of change. It gives the change in value when we change the argument a little bit. Derivatives are therefore the mathematical way of describing change. For a univariate function the derivative at $x=a$ is the slope of the tangent line to the function in $(a,f(a))$.

The derivative of a function f is also a function. Taking the derivative thus transforms a function into a new function. The derivative function is often denoted as f'.

We can take the derivative of the derivative. Then we calculate the change in the slope as we move a little along the horizontal axis. The *second derivative* is denoted as $f''=\frac{d^2f}{dx^2}$. In the same way we can calculate the derivative up to any order (3th, 4th etc).

Differentiating a multivariate function is somewhat more complex. The idea is the same: what happens to the function value when i change the input just a little? But what do i mean now with changing the input? Should all the arguments be changed, or just one? Well actually it is your choice, there is a need for both these options in practice.

The simplest one is to change only one of the arguments and see what happens to the function value. Consider the function f in two arguments, say x and y. Let us change x to $x+h$, while keeping y fixed! We could then calculate what is known as the *partial derivative* of f with respect to its first argument which is called x.

.. math::

   \frac{\partial}{\partial x} f(x,y) = \lim_{h\rightarrow0}
   \frac{f(x+h,y)-f(x,y)}{h}

Observe that instead of df/dx we write $\partial f/\partial x$ to distinguish between the derivative of a univariate function and the derivative with respect to just one argument (the one that by convention is called x) of a multivariate function. Be sure to understand that the partial derivative of a multivariate function results in a multivariate function in the same number of arguments.

The partial derivative in the y argument is:

.. math::

   \frac{\partial}{\partial y} f(x,y) = \lim_{h\rightarrow0}
   \frac{f(x,y+h)-f(x,y)}{h}

At high school you have learned how to calculate the derivatives of functions. Better said you were given the derivatives of some basic functions (like x^n, $\cos(x)$, $\log(x)$ and others) and the rules to calculate the derivatives of compound functions (like the chain

rule and the product rule). Can we use this knowledge for partial differentiation as well? Yes we can. Remember when we are (partially) differentiating f with respect to x we keep y fixed and thus while differentiating anything in the formula with a y in it, it is treated as a constant.

For instance consider $f(x,y)=x^2+y$. Differentiating with respect to x leads to $2x$. The rules to follow here are:

* the derivative of a sum is the sum of the derivatives, so we may take the derivative of x^2 plus the derivative of y.

* the derivative of x^2 with respect to x is equal to $2x$

* the derivative of y *with respect to* x is 0.

The partial derivative with respect to y is a constant function equal to 1 everywhere (note that the term x^2 now is taken to be fixed, i.e. constant with zero derivative).

Also for partial derivatives we may repeat the differentiation. So the second order derivative in the x argument is denoted as $\partial^2 f / \partial x^2$. But now we can do something else as well: first take the derivative in x direction followed by taking the derivative in y direction. Or the other way around: first y then x. It can be shown that for most functions the order in which the derivatives are taken does not matter. We have:

.. math::

   \frac{\partial^2 f}{\partial x \partial y} = \frac{\partial^2 f}{\partial y \partial x}

The partial derivatives play an important role in the analysis of local structure in images. To make notation a bit simpler there we will use the subscript notation for partial derivatives. Let f be a 2D function with arguments we name x and y. The partial derivative with respect to x is denoted as f_x, and the second partial derivative both with respect to x as f_{xx}. The mixed second order derivative if f_{xy}.

We will also sometimes use the notation ∂_x to denote $\frac{\partial}{\partial x}$.

The order of differentiation of a multivariate function is the total number of times we do a differentiation not matter in which argument. So f_x and f_y are first order derivatives, whereas f_{xx}, f_{xy} and f_{yy} are second order derivatives. Note that f_{xxy} is a third order derivative.

The Chain Rule of Differentiation

==========

A chain of functions

.. sidebar:: Example

   .. math::
      f(x) &= x^2 \\
      g(x) &= \exp(-x) \\
      h(x) &= C x

   .. math::
      k(x) &= h(g(f(x))) \\
      &= C \exp(-x^2)

   .. math::
      k'(x) = -2 C x \exp(-x^2)

Consider the functions f, g and h. We consider the chain composition of these functions $h\after g\after f$ defined as:

.. math::
   y = k(x) = (h\after g\after f)(x) = h(g(f(x)))

In a flow graph the function composition is sketched in :numref:‘fig-hogof‘.

The Chain Rule

Let’s start with the composition of two functions:

.. math::
   h(x) = (g\after f)(x) = g(f(x))

When we differentiate a function we are looking at the change in the function value as a consequence of an infinitesimal change in the independent variable x. We have in first order:

.. math::
   h(x+dx) = h(x) + h'(x) dx

Now we do the same for $h(x)=g(f(x))$:

.. math::
   h(x+dx) &= g(f(x+dx))\\
   &= g(f(x)+f'(x)dx)\\
   &= g(f(x)) + g'(f(x))f'(x)dx\\
   &= h(x) + g'(f(x)) f'(x) dx

Comparing both expressions for $h(x+dx)$ we get:

.. math::
   h'(x) = g'(f(x)) f'(x)

the familiar chain rule that all of us learned at high school.

.. sidebar:: Example

   .. math::
      f(x) &= x^2 \\
      g(x) &= \exp(-x) \\
      h(x) &= C x

   .. math::
      k(x) &= h(g(f(x))) \\
      &= C \exp(-x^2)

   .. math::
      f'(x) &= 2x\\
      g'(x) &= -\exp(x)\\
      h'(x) &= C

   .. math::
      k'(x) &= h'(g(f(x))\;g'(f(x))\; f'(x)\\
      &= C \; (-\exp(x^2)) \; (2 x)\\
      &= -2Cx\exp(-x^2)

The derivative of the composed function $k = h\after g\after f$ is given by the chain rule

.. math::
   :label: eq-chainrule

   \frac{dk}{dx} = (h\after g\after f)'(x) = h'(g(f(x))\;g'(f(x))\; f'(x)

This is easily proved given the chain rule for the composition of two functions.

In functional terms we can write:

.. math::
   (h\after g\after f)' = (h'\after g \after f)\;(g'\after f)\;f'

where the multiplication of functions like $f g$ is defined as $(fg)(x)=f(x)g(x)$.

The Leibnitz Notation

Consider again the function composition

.. math::
   (h\after g\after f)(x) = h(g(f(x)))

If we define $u = f(x)$ and $v = (g\after f)(x)$ and indicate the values in the flow graph (see :numref:‘fig-hogof-uv‘)

The derivative of the composed function is:

.. math::
   (h\after g\after f)’(x) = h’(v) g’(u) f’(x)

or equivalently:

.. math::
   \frac{dy}{dx} = \frac{dy}{dv} \frac{dv}{du} \frac{du}{dx}

From here the traditional Leibnitz notation gets a bit sloppy, confusing at the start but convenient once you get used to it. Instead of using a new symbol for the result of a function in the chain (in our example $u=f(x)$ and $v=g(f(x))$) we will use the function names for its output value as well. The chain rule in Leibnitz form then becomes:

.. math::
   \frac{dh}{dx} = \frac{dh}{dg}\;\frac{dg}{df}\;\frac{df}{dx}

In summary: for every node (function) in the chain we diffentiate the output with respect to its input. And multiplying these ’one-function-derivatives’ we can calculate the derivative of (part of) the chain.

Multivariate Chain Rule

In the machine learning context we will be using the chain rule a lot for multivariate functions. Consider a simple one

.. math::
   f(u, v)

We will look at how the value of f will change if both u and v change a small bit, say from u to $u+du$ and from v to $v+dv$. The change df in f then is:

.. math::
   df = f(u+du, v+dv) - f(u, v)

Note that in the univariate case we have

.. math::
   f(x+dx) = f(x) + f_x(x) dx

if we now only consider the change in u and keep the second argument constant we get:

.. math::

df = f(u, v+dv) + f_u(u, v+dv) du - f(u,v)

Now for both terms we keep u constant and apply the same line of reasoning to the second argument of f we get:

.. math::
df &= f(u,v) + f_v(u,v)dv + (f_u(u,v) + f_{uv}(u,v)dv)du - f(u,v)\\
&= f_u(u,v)du + f_v(u,v)dv + f_{uv}(u,v) du dv

Note that $dudv$ is neglible small when compared with both du and dv and thus within first order:

.. math::
df = f_u(u,v) du + f_v(u,v) dv

This is called the **total derivative** of f. The above is an intuitive introduction of the concept but it can be given a solid mathematical foundation. **The total derivative states that the change of** f **is the sum of the contributions due to the changes in its arguments.**

Now we consider the situation where both u and v are functions of x. Then we have

.. math::
df &= f_u(u(x), v(x)) du(x) + f_v(u(x),v(x))dv(x)\\
&= f_u(u(x), v(x)) u_x(x) dx + f_v(u(x),v(x)) v_x(x) dx

and thus

.. math::
\frac{df}{dx} = f_u(u(x), v(x)) u_x(x) + f_v(u(x),v(x)) v_x(x)

or in Leibnits notation:

.. math::
\frac{df}{dx} = \pfrac{f}{u} \frac{du}{dx} + \pfrac{f}{v} \frac{dv}{dx}

In general for a function f with n arguments each depending on x:

.. math::
g(x) = f(u_1(x),\cdots,u_n(x))

we have:

.. math::
g'(x) = \sum_{i=1}^n f_{u_i}(u_1(x),\cdots,u_n(x)) u_i'(x)

Or in Leibnitz notation

.. math::
\frac{dg}{dx} = \sum_{i=1}^n \pfrac{f}{u_i} \frac{du_i}{dx}

And finally if we are given a multivariate function f whose n

arguments all depend on m arguments $x_1,\ldots,x_m$, i.e.

.. math::
  g(x_1,\ldots,x_m) = f(u_1(x_1,\ldots,x_m),\ldots,u_n(x_1,\ldots,x_m)),

we get

.. math::
  \pfrac{g}{x_j} = \sum_{i=1}^n \pfrac{f}{u_i} \pfrac{u_i}{x_j}

This last expression is of great importance in the machine learning context.

.. langere omslachtige manier
  In the machine learning context we will be using the chain rule a lot
  for multivariate functions. Consider a simple one

  .. math::
    g(x) = f(u(x), v(x))

  thus f is a multivariate function with two arguments. The first
  argument is a function u in x and the second argument is a
  function v in x.

  Differentiating g with respect to x we are looking in what way the
  value of g changes in case x changes a bit from x to
  $x+dx$.

  .. math::
    g(x+dx) = f(u(x+dx), v(x+dx))

  To simplify the notation a bit we write

  .. math::
    g(x+dx) = f(u+du, v+dv)

  where we omitted the argument x for the u and v function (but
  remember they are still functions of x). Look at $u+du$:

  .. math::
    u+du = u(x+dx) = u(x)+u'(x)dx

  from which we conclude that $du = u'dx$ (again omitting the argument
  x from u'). In the same way we see that $dv = v'dx$.

  We continue with the term $f(u+du, v+dv)$. Concentrating on the first
  argument $u+du$ we can write in first order

  .. math::
    f(u+du, v+dv) = f(u, v+dv) + f_u(u, v+dv) du,

  where we have used the (commonly used) notation

  .. math::
    f_u = \pfrac{f}{u}.

For the term $f(u,v+dv)$ we can do the same but then for the second argument of f:

.. math::
   f(u, v+dv) = f(u,v) + f_v(u,v)dv

For the term $f_u(u,v+dv)$ we have

.. math::
   f_u(u, v+dv) = f_u(u,v) + f_{uv}(u,v)dv

Combining all the terms we have

.. math::
   f(u+du, v+dv) = f(u,v) + f_u(u,v)du + f_v(u,v)dv + f_{uv}(u,v)dudv

but as we are still interested in the first order behavior we can omit the last term (if du and dv are infinitesimally small then $du dv$ is negligible in comparison).

.. math::
   f(u+du, v+dv) = f(u,v) + f_u(u,v)du + f_v(u,v)dv

Now we reintroduce the x arguments

.. math::
   :label: eq-1stf

   f(u(x+dx), v(x+dx)) &= f(u(x), v(x)) + \\
   &f_u(u(x),v(x))u'(x)dx + f_v(u(x),v(x))v'(x)dx

Note that $f(u(x+dx), v(x+dx))=g(x+dx)$ by definition and so

.. math::
   :label: eq-1stg

   g(x+dx) = g(x)+g'(x)dx

Comparing :eq:‘1stf‘ and :eq:‘1stg‘ we get

.. math::
   g(x)+g'(x)dx = f(u(x), v(x)) + f_u(u(x),v(x))u'(x)dx + f_v(u(x),v(x))v'(x)dx

Remember that $g(x)=f(u(x), v(x))$ so these values on the left and right hand side cancel and then also the dx factors cancel out and we are left with:

.. math::
   g'(x) = f_u(u(x),v(x))u'(x) + f_v(u(x),v(x))v'(x)

or in Leibnitz notation

.. math::
   \frac{dg}{dx} = \pfrac{f}{u}\frac{du}{dx} + \pfrac{f}{v}\frac{dv}{dx}

This long derivation was done just to show that when **using the chain rule on a multivariate function the contributions of all the arguments add up in the calculation of the derivative.** In general when

.. math::
   g(x) = f(u_1(x),\cdots,u_n(x))

then

.. math::
   g'(x) = \sum_{i=1}^n f_{u_i}(u_1(x),\cdots,u_n(x)) u_i'(x)

Or in Leibnitz notation

.. math::
   \frac{dg}{dx} = \sum_{i=1}^n \pfrac{f}{u_i} \frac{du_i}{dx}

And finally if we are given a multivariate function g to start with:

.. math::
   g(x_1,\ldots,x_m) = f(u_1(x_1,\ldots,x_m),\ldots,u_n(x_1,\ldots,x_m))

then

.. math::
   \pfrac{g}{x_j} = \sum_{i=1}^n \pfrac{f}{u_i} \pfrac{u_i}{x_j}

This last expression is of great importance in the machine learning context.

Matrix Calculus
==========

Essential in machine learning is optimization. Almost all machine learning algorithms start with optimization of a scalar loss (or cost) function with respect to an input vector $\v x$ or parameter vector $\v p$.

Things become more complicated when we differentiate all elements of a vector with respect to all elements of another vector. Or even when matrices are involved.

Matrix calculus provides the tools to elegantly deal with these derivatives.

This section is based on the Wikipedia article on matrix calculus. We make a choise for the so called denominator layout as explained in the Wikipedia article.

The derivative of a scalar function with respect to a vector

Let y be a scalar function of all elements x_i in vector $\v x$. By definition we state:

.. math::
   \pfrac{y}{\v x} = \begin{pmatrix}
   \pfrac{y}{x_1}\\ \vdots \\ \pfrac{y}{x_n}
   \end{pmatrix}

The derivative of such a scalar function is often called the **gradient** of the function.

The derivative of a vector with respect to a scalar

Let $\v y$ be a vector and let x be a scalar then:

.. math::
   \pfrac{\v y}{x} = \begin{pmatrix}
   \pfrac{y_1}{x} & \cdots & \pfrac{y_m}{x}
   \end{pmatrix}

The derivative of a vector with respect to a vector

.. math::
   \pfrac{\v y}{\v x} = \begin{pmatrix}
   \pfrac{y_1}{x_1} & \cdots & \pfrac{y_m}{x_1}\\
   \vdots & \ddots & \vdots\\
   \pfrac{y_1}{x_n} & \cdots & \pfrac{y_m}{x_n}
   \end{pmatrix}

Some important derivatives in the machine learning context

For A not a function of $\v x$:

.. math::
   :label: eq_pAxpx

   \pfrac{A\v x}{\v x} = A\T

.. math::
   :label: eq_pxTApx

   \pfrac{x\T A}{\v x} = A

.. math::
   :label: eq_pxTAxpx

   \pfrac{\v x\T A \v x}{\v x} = (A+A\T)\v x

Let $\v u$ be a vector function such that $\v x\in\setR^2\mapsto\v u(\v x)\in\setR^m$, then:

.. math::
   :label: eq_pAupx

   \pfrac{A\v u}{\v x} = \pfrac{\v u}{\v x} A\T

or more generally

.. math::
   \pfrac{\v f(\v u)}{\v x} = \pfrac{\v u}{\v x} \pfrac{\v f(\v u)}{\v u}

where $\v f$ is a vector valued function. Note that the choice $\v f(\v u)=A\v u$ and using :eq:‘pxTAxpx‘ leads to the result in :eq:‘eq_pAupx‘.

Let f be a scalar function then with $f\cdot(\v x)$ we denote the elementwise application of the function f to the vector $\v x$:

.. math::
   f\cdot(\v x) = \begin{pmatrix}
   f(x_1)\\ \vdots \\ f(x_n)\end{pmatrix}

We introduce this special notation to prevent confusion with a scalar function, say g, that has a vector as argument an produces a scalar: $g(\v x)$. Again let $\v y = \v y(\v x)$, then:

.. math::
   :label: eq_pfdypx

   \pfrac{f\cdot(\v y)}{\v x} &= \pfrac{}{\v x}\begin{pmatrix}
   f(y_1)\\ \vdots \\ f(y_m)\end{pmatrix}\\
   &= \begin{pmatrix}
   \pfrac{f(y_1)}{x_1} & \cdots & \pfrac{f(y_m)}{x_1}\\
   \vdots & \ddots & \vdots\\
   \pfrac{f(y_1)}{x_n} & \cdots & \pfrac{f(y_m)}{x_n}
   \end{pmatrix}\\
   &= \begin{pmatrix}
   f'(y_1)\pfrac{y_1}{x_1} & \cdots & f'(y_m)\pfrac{y_m}{x_1}\\
   \vdots & \ddots & \vdots\\
   f'(y_1)\pfrac{y_1}{x_n} & \cdots & f'(y_m)\pfrac{y_m}{x_n}
   \end{pmatrix}\\
   &= \pfrac{\v y}{\v x} \text{diag}(f'\cdot(\v y))

Examples from Machine Learning

Linear Regression
.....

The cost function in linear regression is

.. math::
   J(\v\theta) = \frac{1}{2m} \|\hv X \v\theta - \v y\|^2

The gradient function then is:

.. math::
   \pfrac{J(\v\theta)}{\v\theta} &= \frac{1}{2m}\pfrac{}{\v\theta}
   \left((\hv X \v\theta - \v y)\T(\hv X \v\theta - \v y)\right)\\
   &= \frac{1}{2m}\pfrac{}{\v\theta} \left(
   \v\theta\T\hv X\T\hv X \v\theta - 2 \theta\T\hv X\T\v y + \v y\T\v y \right)\\
   &= \frac{1}{2m}\left(2\hv X\T\hv X \v\theta - 2\hv X\T \v y\right) \\
   &= \frac{1}{m} \hv X\T (\hv X\v\theta - \v y)

Logistic Regression
.....

The cost function in logistic regression is:

.. math::
   J(\v\theta) = \frac{1}{m} \left(-\v y \T \log\cdot(g\cdot(\hv
   X\v\theta)) - (\v 1 - \v y)\T \log\cdot(\v 1 - g\cdot(\hv X \v\theta))
   \right)

In calculating the gradient we first consider the term:

.. math::
   \pfrac{}{\v\theta} \v y\T \log\cdot(g\cdot(\hv X\v\theta)) = \pfrac{}{\v\theta} \v y\T f\cdot(\hv X \v\theta)

where we introduced $f = \log \circ g$ (the composition of g after
$\log$. Using :eq:‘eq_pAupx‘ we get

.. math::
   \pfrac{}{\v\theta} \v y\T f\cdot(\hv X \v\theta) = \pfrac{f\cdot(\hv X \v\theta)}{\v\theta} \v y

and then using :eq:‘eq_pfdypx‘ we get

.. math::
   \pfrac{}{\v\theta} \v y\T f\cdot(\hv X \v\theta) &= \pfrac{\hv X\v\theta}{\v\theta} \text{diag}(f'\cdot(\hv X\v\th
eta)) \v y\\
   &= \hv X\T \text{diag}(f'\cdot(\hv X\v\theta)) \v y

Observe that because $g'(v) = g(v)(1-g(v))$ we have that $f'(v) = 1 - g(v)$ and thus

.. math::
   \pfrac{}{\v\theta} \v y\T f\cdot(\hv X \v\theta) =
   \hv X\T \text{diag}((1-g)\cdot(\hv X\v\theta)) \v y

For the second term in the gradient we get:

.. math::
   \pfrac{}{\v\theta}(\v 1- \v y)\T (\log\cdot(1-g)\cdot(\hv X \v\theta) =
   - \hv X\T \text{diag}(g\cdot(\hv X\v\theta))(\v 1-\v y)

For the sum of both terms:

.. math::
   \hv X\T\left(\text{diag}((1-g)\cdot(\hv X\v\theta))\v y
   - \text{diag}(g\cdot(\hv X\v\theta))(\v 1 - \v y) \right)

Computational Graphs and the Chain Rule of Differentiation
==========

We start by introducing computational graphs as a simple visualization of the flow of data within a typical machine learning system (neural networks as prime examples) by defining the sequence(s) of computations necessary to calculate the end result.

The final step in a computational graph (when learning) is the calculation of the loss that quantifies the error that the system makes when processing specific data.

These computational graphs provide not only describe the ’forward’ flow of the data but in case we are interested in the derivatives of

In order to understand automatic differentiation we first have to look at computational graphs. Then using these graphs we look at automatic differentiation.

Computational Graph

A computational graph describes the ’flow of data’ throughout a computation. Consider the expression $z=log(3x^2 + 5 xy)$ in a graph this can be depicted as:

Automatic Differentiation

We are going to use automatic differentiation to calculate $\prt z/\prt x$ and $\prt z/\prt y$. To start we look at one node representing the expression

.. math::
   z = z(x,y)

and imagine that this node is part of a larger graph ending up in the

final scalar value ℓ. Now assume that somehow we were capable of calculating $\prt \ell/\prt z$. What then are the derivatives $\prt\ell/\prt x$ and $\prt\ell/\prt y$?

For the derivative with respect to x we have:

.. math::
  \pfrac{}{x} \ell(z(x,y)) = \pfrac{\ell}{z} \pfrac{z}{x}

and for the derivative with respect to y:

.. math::
  \pfrac{}{y} \ell(z(x,y)) = \pfrac{\ell}{z} \pfrac{z}{y}

It should be noted that the notation $\ell(z(x,y))$ is a bit misleading. It is just to illustrate that ℓ depends on z, in a complex graph it will probably depend on many other values.

The important observation is that for any node in the graph, given the derivative of the output ℓ of the entire graph with respect to the output of this one node, we can calculate the derivative of ℓ with respect to the inputs of this node. If that is true, then we are able to 'propagate' the derivative calculations from the end of the graph 'backwards' to the beginning. This is essentially what backpropagation does in neural network learning.

Example

Let's do it for our first example:

.. math::
  z = \log(3 x^2 + 5 x y)

Using the chain rule and some basic derivative rules we have:

.. math::
  \pfrac{z}{x} &= \frac{6x+5y}{3x^2+5xy}\\
  \pfrac{z}{y} &= \frac{5x}{3x^2+5xy}\\

where we have assumed that $\log$ is the natural logarithm (as most programming languages do as well). Let's see whether we can reproduce these results using the computational graph. First we redraw the graph and name all arrows (values) in the graph for ease of reference.

So we have:

.. math::
  a &= x^2\\
  b &= 3 a\\
  c &= x y\\
  d &= 5 c\\
  e &= b +d\\
  z &= \log(e)

and no, e is *not* the
Working from the end to the start we get

.. math::
   \pfrac{z}{e} &= \frac{1}{e}\\
   \pfrac{z}{b} &= \pfrac{z}{e}\\
   \pfrac{z}{d} &= \pfrac{z}{e}\\
   \pfrac{z}{a} &= \pfrac{z}{b} 3\\
   \pfrac{z}{c} &= \pfrac{z}{d} 5\\
   \pfrac{z}{y} &= \pfrac{z}{c} x

The above steps are easy, but what about the derivative of z with respect to x. There we see that the value of x is fed into two nodes in the graph. So from the two 'streams' starting at x when considering the backward pass we see to derivatives

.. math::
   \pfrac{a}{x} \pfrac{z}{a}\quad \text{and}\quad \pfrac{c}{x} \pfrac{z}{c}

to calculate $\prt z/\prt z$ we have to add these contributions:

.. math::
   \frac{z}{x} &= \pfrac{a}{x} \pfrac{z}{a} + \pfrac{c}{x} \pfrac{z}{c}\\
               &= 2x \pfrac{z}{a} + y \pfrac{z}{c}

This leads to

.. math::
   \pfrac{z}{x} &= 2x \pfrac{z}{a} + y \pfrac{z}{c}\\
                &= \frac{6x}{3x^2+5 xy} + \frac{5y}{3x^2+5xy}
  &= \frac{6x+5y}{3x^2+5 xy}

The derivation of $\prt z/\prt y$ is left to the reader as an exercise.

Numerical Example

Computational graphs and automatic differentiation can be used to derive formula's but that is not the way they are used in practice. In practice they are used to calculate the derivatives numerically for a given set of inputs. Consider again our example graph but now with input $x=2$ and $y=3$.

Calculating the intermediate values then constitutes the *forward pass*:

.. math::
   a = 4\\
   b = 12\\
   c = 6\\

$$
d = 30\\
e = 42\\
z = \log(42) = 3.73
$$

The backward pass numerically is simple and we have indicated the values below the edges in the graph above. Be sure to understand how the purple values are calculated.

Vectorial Graphs

In neural networks the nodes in graphs take vectorial input and produce vectorial output. Consider the node that takes $\v x\in\setR^n$ as input and produces $\v y\in\setR^m$ as output. In formula:

$$
\v y = \v f(\v x)
$$

Note that $\v f$ is a function that takes a vector and produces a vector. In element notation we have:

$$
\begin{pmatrix}y_1\\\vdots\\y_m\end{pmatrix}
= \begin{pmatrix}
f_1(x_1,\ldots,x_n)\\
\vdots\\
f_m(x_1,\ldots,x_n)
\end{pmatrix}
$$

Note that we may write $\ell(y_1,\ldots,y_m)$ to indicate the ℓ is dependent on all components of $\v y$. Note that eacht y_j is dependent on all x_i, i.e. $y_j = f_j(x_1,\ldots,x_n)$. Constructing $\prt \ell/\prt \v x$ elementwise we have:

$$
\pfrac{\ell}{x_i} = \pfrac{\ell(y_1,\ldots,y_m)}{x_i}
= \sum_{j=1}^{m} \pfrac{\ell}{y_j} \pfrac{f_j}{x_i}
$$

We define the matrix J such that

$$
J_{ij} = \pfrac{y_j}{x_i}
$$

then:

$$
\pfrac{\ell}{x_i} = \sum_{j=1}^{m} J_{ij} \pfrac{\ell}{y_j}
$$

as vectors we can write:

$$
\pfrac{\ell}{\v x} = J \pfrac{\ell}{\v y}
$$

The matrix J is called the *Jacobian* of the function $\v f$ and is often denotes as

.. math::
   J = \pfrac{\v f}{\v x}

When using vectorial graphs in neural networks we will see that the Jacobian matrices are often of a special form due to the choice of the function $\v f$.

Some Bits of Information Theory
==========

Entropy

Information is linked to probability: an unlikely event is associated with a high information content. Claude Shannon was the first to give a solid mathematical model for this intuition. For an event A with probability $P(A)$ he defined **self information**

.. math::
   I(A) = -\log_b(P(A))

An event with probability one has zero self information. Nowing that it will rain every single day does not really provide any information. An event with very low probability is so rare that in case it does occur carries with it a lot of information. Note that an event with $P(A)=0$ is not really an event in the sense that it will never occur. In information theory we will only consider events for which $0<P(A)\leq 1$. For events that are very unlikely, i.e. $P(A)\ll 1$, the self information is very high. It is a very rare event and if such an event really occurs the information is high.

The unit of self information depends on the choice for the base b of the logarithm. For $b=2$ we express the self information in **bits**, for $b=e$ we use **nats** and for $b=10$ we use **decimals**. From now on we will use $b=2$. Note that bits in this definition are not bits as we know in computer science (but related).

Consider a discrete random variable X. The expected self information in observing one value is called the **entropy** of X and can be calculated with

.. math::
   H = E(I(X=x)) &= \sum_{x} I(X=x)\, P(X=x)\\
     &= - \sum_{x} P(X=x) \log_2 P(X=x)

Consider a uniformly distributed random variable with probability

function $p(x)=P(X=x)$:

.. math::
   p(x) = \begin{cases}
   \tfrac{1}{8} &: 1\leq x \leq 8\\
   0 &: \text{elsewhere}
   \end{cases}

The entropy for this distribution is $H=3\, \text{bits}$. Note that
this number is equal to the number of computer bits needed to encode a
number in the range from 0 to 7 (i.e. 8 different numbers).

Now consider a random variable with probability function

.. math::
   p(x) = \begin{cases}
   \tfrac{1}{2} &: 1\leq x \leq 2\\
   0 &: \text{elsewhere}
   \end{cases}

For this distribution the entropy is

.. math::
   H &= -\tfrac{1}{2} \log_2(\tfrac{1}{2}) - \tfrac{1}{2} \log_2(\tfrac{1}{2})\\
     &= \tfrac{1}{2} + \tfrac{1}{2}\\
     &= 1

And for the distribution with no uncertainty at all:

.. math::
   p(x) = \begin{cases}
   1 &: x=1\\
   0 &: \text{elsewhere}
   \end{cases}

the entropy is $H=0$.

These simple examples that the entropy H characterizes the disorder
in the system characterized with the given probability function. The
uniformly distributed system has the highest entropy, it is total
chaos: all outcomes are equally probable. The system with $H=0$ is
totally ordered, the outcome is always the same.

Bit Encoding

Consider the English language and assume that we write a text just
using the 26 capital letters from A to Z. We start with assuming that
each letter is equally probable. In that case the entropy is

.. math::
   H = - 26 \tfrac{1}{26} \log \tfrac{1}{26} = 4.70

But the letters are evidently not uniformly. The frequencies of

letters in the English languare are given in the table below.

Letter	Frequency	Letter	Frequency
E	12.02	M	2.61
T	9.10	F	2.30
A	8.12	Y	2.11
O	7.68	W	2.09
I	7.31	G	2.03
N	6.95	P	1.82
S	6.28	B	1.49
R	6.02	V	1.11
H	5.92	K	0.69
D	4.32	X	0.17
L	3.98	Q	0.11
U	2.88	J	0.10
C	2.71	Z	0.07

Note that the frequencies (when interpreted as probabilities) certainly do not come from a uniform distribution.

Let’s calculate the entropy in a small Python program.

.. ipython:: python

```
plt.clf()

letters = "ETAOINSRHDLUCMFYWGPBVKXQJZ"
freq = np.array([12.02, 9.10, 8.12, 7.68, 7.31,
                 6.95, 6.28, 6.02, 5.92, 4.32, 3.98, 2.88, 2.71,
                 2.61, 2.30, 2.11, 2.09, 2.03, 1.82, 1.49, 1.11,
   0.69, 0.17, 0.11, 0.10, 0.07])
print(len(letters), len(freq))
prob = freq/100
H = - np.sum(prob * np.log2(prob))
print(H)
plt.stem(prob, use_line_collection=True);
plt.xticks(np.arange(len(letters)), letters);
plt.savefig('source/images/englishletters.png');
```

We see that the entropy dropped to 4.18 indicating there is somewhat less disorder in a sequence of letters in an English text.

Shannon showed that the entropy of an ’alphabet’ (the possible outcomes of our random variable) is the lower bound on the number of computer bits (0 or 1) needed to encode a typical (long) message. Indeed for a distribution for the English language we can do better then the 5 bits we classically need to encode 26 different letters.

The crux then is to use fewer bits for often occuring letters and use more bits for letters that occur less frequent. A nice example of such a coding scheme is the Morse alphabet. There every letter (and digit) is encodes with a sequence of 'dots' and 'dashes' (short and long tones). The international Morse codes are given in the table below

Observe that indeed the number of dots and dashes are indeed roughly increasing with increasing frequency of the letter.

Unfortunately there is no constructive proof of Shannon's result. Therefore there is no optimal algorithm for bit encoding. Perhaps the most famous one is Huffman encoding.

Without proof we give the Huffman encoding of the English alphabet:

Letter	Code	#bits	Letter	Code	n = #bits
E	011	3	M	00111	5
T	000	3	U	01001	5
A	1110	4	Y	00100	5
H	0101	4	B	101100	6
I	1100	4	G	111100	6
N	1010	4	P	101101	6
O	1101	4	V	001010	6
R	1000	4	W	111101	6
S	1001	4	K	0010111	7
C	01000	5	J	001011001	9
D	11111	5	Q	001011010	9
F	00110	5	X	001011011	9
L	10111	5	Z	001011000	9

Note that the number of bits is roughly inversely proportional to the probability of the letters. For a typical English text you would need on average

.. math::
  \sum_{x\in{A,\ldots,Z}} p(x) n(x) = 4.21 \,\text{bits}

a number that is rather close to the theoretical minimum of 4.18.

A *bit allocation* algorithm based on the frequencies of the different symbols, like Huffman encoding, is used in many compression schemes (e.g. JPEG image encoding).

Cross Entropy

Now consider the situation where we have a coding scheme (bit allocation) that is optimized for distribution $q(x)$. However it turns out that in practice the true distribution is $p(x)$. The expected number of bits per symbol then is:

.. math::
   H(p,q) = - \sum_{x} p(x) \log_2(q(x))

This is called the **cross entropy** between distributions p and q.

The cross entropy is often used in machine learning where we have a true a posteriori distribution $p(x)$ over all class labels x and a distribution $q(x)$ as predicted by our model. For binary logistic classification were $p(x)\in\{y,1-y\}$ and $p(q)\in\{\hat y, 1-\hat y\}$ where $\hat y$ is the probability according to our model that $x=1$. The cross entropy in that case is

.. math::
   H(p,q) = - y\log\hat y - (1-y)\log(1-\hat y)

This is the loss function (for one feature vector) as defined in logistic regression.

==========
Probability and Statistics
==========

What is the probability of throwing a 6 with a fair dice? Almost everyone immediately will answer: $1/6$ -th. But what do we mean by that? Two obvious lines of reasoning to arrive at that answer are:

- If we throw a dice there are six possible outcomes and because it is a fair dice every outcome is equally probable, the probabilities of each possible outcome should add up to one, and so the probability of throwing a 6 is equal to $1/6$ -th.

- Let's repeat the experiment a large number of times, say N. After throwing the dice N times we estimate the probability as:

  .. math::
     \text{Probability(throwing 6)} = \frac{\text{#\{throwing 6\}}}{N}

  This estimate will approach the true probability if we let $N\rightarrow\infty$.

This line of reasoning is part of what is called the **frequentist** approach to probability. We will follow this route quite often in this course.

But now consider the question: what is the probability that it will rain tomorrow? Neither of the two lines of reasoning above can help us here? Tomorrow it will either rain or not but these possible outcomes are clearly not necessarily equally probable. And if we repeat the experiment N times (say we take $N=10\times365$, then we have data

over 10 years) and use the frequentist approach we undoubtly end up with the probability that it will rain on any randomly given day in the Netherlands. Certainly not the probability that it will rain *tomorrow*. Assigning a probability of 70% for rain tomorrow quantifies our **belief** that it will rain tomorrow. A belief that is based on data and models.

Fortunately the interpretation of probability (being a frequentist view or a belief based view) is of lesser importance. The classical mathematical rules for probability and statistics that we will look at in this course are largely independent of the interpretation. Loosely speaking we might say that **probability is the mathematical language of choice when dealing with uncertainty.**

Probabilty theory deals with random experiments and random processes. Experiments and processes that are not deterministic, it is simply not possible to know beforehand exactly what will be the result of such an experiment or process. In probability theory we assume that for each of the possible outcomes of a random experiment the corresponding probabilities are well defined.

With **statistics** we enter the realm of everyday life. We may assume that there exists a function that assigns a probability to each possible outcome of a random experiment, but alas all we have are observations (a sample or 'steekproef' in Dutch) of the random experiment. Statistics then deals (among others) with the question what can be known about the underlying random experiment using only the observations in the sample? Because the sample consists of random numbers the conclusions from statistics about the random experiment are in a sense random as well. We can never be completely sure about our (numerical) conclusions. A lot a statistics deals with this important question: can we quantize the probability that we arrive at the right (or wrong) conclusions?

Probability Space and Probability Axioms

Random Experiments and Probability Spaces

To describe a random experiment in mathematical terms we need what is called a probability space. Such a space consists of three things:

#. The set of all possible outcomes of the random experiment. We will denote this set as U for **universe**.

#. A collection of subsets of U. A subset A if U (i.e. $A\subset U$) corresponds with an **event**. For example when throwing a dice

a possible event is to throw a 6. Another possibiliy is the event corresponding with throwing an odd number of points.

#. A probability measure $\P$ that maps each subset $A\subset U$ (event) onto a positive scalar $\P(A)$

Probability Axioms

Let A and B both be subsets of the probability space (the universe) U, i.e. A and B are events. Surprisingly maybe, the whole theory of probability is based upon just three axioms:

.. proof:axiom:: Probability Axioms

   .. math::
      (A_1)&\quad \P(A)\geq 0\\
      (A_2)&\quad \P(U)=1\\
      (A_3)&\quad A\cap B=\emptyset \Rightarrow \P(A\cup B) = \P(A) + \P(B)

The first axiom simply states that probabilities are non negative (real) numbers.

The second axioma states that the probability for the universe U equals 1. This is intuitively clear as the universe contains everything that could ever happen.

The third axiom is the only axiom that relates probabilities of several events: if event A and event B are mutually exclusive (i.e. you can’t have A and B being the simulatanuous outcome of one random experiment, e.g. throwing with a dice and having events "6" and "odd") then the probability of either event A or event B occuring, i.e. $A\cup B$, equals the sum of $\P(A)$ and $\P(B)$.

Venn Diagrams

To gain insight and some intuition of probability space it is useful to depict that space as a two dimensional subset of the plane (the **Venn diagrams** of sets).

Consider the Venn diagram corresponding with the third axiom. We have the universe U and the two events A and B, both subset of U. The assumption that $A\cap B =\emptyset$ is visualised as the empty intersection of the two sets.

You may take the visual interpretation somewhat further and assume the relative area of set A with respect to the area of set U corresponds with the probability $\P(A)$:

.. math::
   \P(A) = \frac{\text{area}(A)}{\text{area}(U)}

Please note that this is NOT the definition of $\P(A)$! It is stating the rule to interpret the Venn diagrams.

Simple Probability Theorems

The definitions and axioms in a mathematical theory are the starting point, but the interesting parts are the theorems, the things that can be proven to be true given the definitions and the axioms. Probability theory is no exception.

Let us start with a simple definition: the complement. Let A be an event, i.e. subset of U. Then we define the **complement** $\neg A$ as:

.. math::
   \neg A = U \setminus A

where $\setminus$ denotes the 'set minus', i.e. everything in the universe that is not in A. This is just a definition. A theorem however is the statement:

.. proof:theorem::

   .. math::
      \P(\neg A) = 1 - \P(A)

.. proof:proof::

   The proof obviously needs the third axiom as we have to relate the probabilities of two events: A and $\neg A$. Observe that:

   .. math::
      U = A \cup \neg A

   By definition we have that A and $\neg A$ are **disjunct**, i.e. $A\cap \neg A=\emptyset$, and thus we can utilize the third axiom:

   .. math::
      \P(U) = \P(A) + \P(\neg A)

   Using the second axiom and rewriting the equation we get:

   .. math::
      \P(\neg A) = 1 - \P(A)

   and that completes the proof.

Now consider the situation where events A and B are not disjunct. Then we cannot apply the third axiom directly. Look at a Venn diagram of two subsets A and B or U with non empty

intersection. From such a figure it is intuively clear that

.. proof:theorem::

   .. math::
      \P(A\cup B) = \P(A) + \P(B) - \P(A\cap B).

.. proof:proof::

   The proof is simple as well. We leave that to the reader in an
   exercise.

Conditional Probabilities
=========================

The conditional probability $\P(A\given B)$ is the probability for
event A given that we know that event B has occured too. For
example we may ask for the probability of throwing a 6 with a fair die
given that we have thrown an even number of points.

.. proof:definition:: Conditional Probability

   The conditional probability of A given B is:

   .. math::
      \P(A\given B) = \frac{\P(A\cap B)}{\P(B)}

In practical applications in machine learning we find ourselves in the
situation were we would like to calculate the probability of some event,
say $\P(A)$, but only the conditional probabilities $\P(A\given B)$
and $\P(A\given \neg B)$ are known. Then the following theorem can be
used.

.. proof:theorem:: Total Probability

   .. math::
      \P(A) = \P(A\given B)\,\P(B) + \P(A\given\neg B)\,\P(\neg B)

.. proof:proof::

   The proof starts with observing that:

   .. math::
      A = (A\cap B) \cup (A\cap \neg B)

   and because $A\cap B$ and $A\cap \neg B$ are disjunct we may apply
   the third axiom and obtain:

   .. math::

\P(A) &= \P(A\cap B) + \P(A\cap \neg B)\\
&= \frac{\P(A\cap B)}{\P(B)}\,\P(B) +
\frac{\P(A\cap \neg B)}{\P(\neg B)}\,\P(\neg B)\\
&= \P(A\given B)\,\P(B) + \P(A\given\neg B)\,\P(\neg B)

This theorem may be extended to **partitions** of the universe U. A partition of U is a collection of subsets B_i for $i=1,\ldots,n$ such that $B_i \cap B_j=\emptyset$ for any $i\not=j$ and $B_1\cup B_2\cup\cdots\cup B_n=U$.

.. proof:theorem:: **Total Probability**

For any partition $\{B_i\}$ of U we have:

.. math::
\P(A) = \sum_{i=1}^{n} \P(A \given B_i)\,\P(B_i)

The proof is a generalization of the proof for the partition $\{B, \neg B\}$.

.. proof:theorem:: Bayes Rule

Bayes rule allows us to write $\P(A\given B)$ in terms of $\P(B\given A)$:

.. math::
\P(A\given B) = \frac{\P(A)}{\P(B)}\,\P(B\given A)

The proof of Bayes rule simply follows from the definition of the conditional probability.

.. proof:definition:: Chain Rule

The definition of the conditional probability can be written in another form:

.. math::
\P(A\,B) = \P(A\given B)\,\P(B)

In this form it is known as the **chain rule** (or **product rule**). This rule can be generalised as:

.. math::
\P(A_1\,A_2\cdots A_n) = \P(A_1\given A_2\,\ldots,A_n)\,\P(A_2\given A_3\,\ldots,A_n) \cdots
\P(A_{n-1}\given A_n)\,\P(A_n)

==========
Probability Trees
==========

When you have learned to make probability trees in high school you have used conditional probabilities and the chain rule probably without knowing it. Consider the following random experiment. There are two vases labelled V_1 and V_2. In the vases are marbles of three colors. The following table shows how many of each color are in the vases.

Vase	#Red	#Green	#Blue
1	2	3	4
2	3	5	1

First we randomly select a vase (no preference for a vase) and then from the selected vase we pick a marble. The probability tree looks like:

Starting at the root node on the left we first select Vase 1 or Vase 2. From either the node 'Vase 1' or the node 'Vase 2' we then pick a marble from the vase. The marble is either 'Red', 'Green' or 'Blue'.

The probability of picking either vase is $1/2$. Picking a Red marble from Vase 1 has probability $\P(\text{Red}\given \text{Vase 1})=2/9$. So the probability of ending up with a Red marble from Vase 1 is given by:

.. math::
\P(\text{Vase 1}\cap \text{Red}) =
\P(\text{Vase 1}) \, \P(\text{Red}\given \text{Vase 1}) = \frac{1}{2}\,\frac{2}{9}

Thus the 'trick' that you have been using in highschool to multiply the probablities along the edges in the tree is nothing else then the application of the chain rule of conditional probabilities.

Independent Events

.. proof:definition:: Independent Events

We define two events A and B to be **independent** in case:

.. math::
\P(A\,B) = \P(A)\,\P(B)

and we often write $A\perp B$ to denote independent events.

When two events are independent it is simple to show that the conditional probability $\P(A\given B)$ equals $\P(A)$ i.e. knowledge

about B does not influence the probability of A. Evidently we also have $A\perp B\Longrightarrow\P(B\given A)=\P(B)$.

A well known example is throwing with two dices. The outcome of the first dice in no way influences the outcome of throwing with the second dice. Therefore throwing 2 times 6 in a row is $1/6\times 1/6 = 1/36$.

An example of dependent events can be found in the previous section about the marbles from the vases. The events ·Vase 1' and 'Red' are evidently not independent.

Now consider events A and B and a third event C.

.. proof:definition:: Conditional Independence

   Events A and B given C are conditional independent in case:

   .. math::
      \P(AB\given C) = \P(A\given C) \P(B\given C)

Independence of A and B does not imply conditional independence or vice versa.

==========
 Random Variables
==========

Remember the random experiment of throwing with a dice? We defined events then as "6" or "odd". The events "1",...,"6" of course are the elementary events in the universe U of all possible outcomes. Note that we write "1" here instead of 1 to stress that we are labelling events.

Let $u\in U$ be such an elementary outcome in the universe U. Then we can set up a mapping $X: u\in U\mapsto X(u)\in B$ where B is either $\setZ$ or $\setR$. Such a mapping is called a **random variable**. A RV is a numerical observation of the outcome of a random experiment.

Consider again the experiment of throwing with a fair dice. In this case $u\in\{\text{"1"}, \text{"2"}, \ldots,\text{"6"}\}$. An obvious mapping in this case is $X(\text{"1"}) = 1$, $X(\text{"2"}) = 2$ etc. This is an example of a **discrete random variable**, the range of the mapping X is the set of integers $\setZ$.

Now consider the experiment where we measure the length of a randomly selected person, so U is the set of all persons. The length can be any real number in the range of 0 to ∞ (of course not all lengths are equally probable). The random variable in this case is $X(u) = \text{length of person } u$, i.e. $X: u\in U \mapsto \setR$. Such a RV is called a **continuous random variable**.

Discrete Random Variables
==========

A discrete random variable X maps the elementary events $u\in U$ unto a value in $\setZ$. i.e. $X: u\in U \mapsto X(u)\in\setZ$. Note that the random variable X in iteself is not an event. We can specify an event A, associated with outcome $X=x$, by specifying

.. math::
   A = \{ u \bigm| X(u)=x \}

We will most often abbreviate this and write the event A as $X=x$. Carefully note the distinction between capital X and x. The capital X always refers to random variables and lower case letters refer to values from the range of the mapping X.

A discrete variable is completely characterized in case for all $x\in\setZ$ we specify $\P(X=x)$. We will do this with the **probability mass functions** p_X:

.. math::
   p_X(x) = \P(X=x)

As an example consider our fair dice again, there we have:

.. math::
   p_X(x) = \begin{cases}
   0 &: x < 1\\
   \frac{1}{6} &: 1\leq x \leq 6\\
   0 &: x > 6
   \end{cases}

Note that we assume that the events $X=x_1$ and $X=x_2$ are disjunct in case $x_1\not=x_2$ and futhermore that all $X=x$ for $x\in\setZ$ form a partition of U and therefore:

.. math::
   \sum_{x\in\setZ} p_X(x) = 1

As an example of a discrete random variable let us set a simple random experiment. We throw with two dices and count the number of both dice.

.. math::
   X=x\quad\text{x is the sum of both dices}

It is evident that $p_X(x)=0$ for $x<2$ and $x>12$. Later on in this chapter we will discuss how to calculate the probabilities for $2\le x\le 12$. Here we will assume that we are able to simulate this random

experiment by randomly generating two numbers in the range from 1 to 6 and adding these numbers. With this simulation we can estimate the probabilities based on our frequentist view on probability. The result is shown in the figure below.

Carefully note that this is only an estimate of the probability mass function. In this case it is easy to calculate the exact function (we will do that in a later section) in other situations observing a random experiment and estimating the probability mass function from the sample is needed.

Continuous Random Variables

If we measure the length of a random person then the length X is a **continous random variable**, in principle the length can be any value $x\in\setR$. The peculiarity of continuous random variables is that $\P(X=x)=0$ for *all* $x\in\setR$. What else could the probability of finding someone with length $180+\pi$ cm be? Therefore a probility mass function is non sensical for continuous random variables.

Instead we define the **probability density function** of a continuous RV as the function $f_X: x\in\setR \mapsto f_X(x)\in\setR$ such that:

.. math::
   \P(a\leq X \leq b) = \int_a^b f_X(x)\,dx

Note that:

#. The probability for $-\infty\leq x \leq\infty$ should be one (the entire real line of course is the universe for this RV) and therefore:

   .. math::
      \int_{-\infty}^{\infty} f_X(x)\,dx = 1

#. A probability *density* is not a probability, we have to integrate the density over a subset in $\setR$ to get a probability.

#. We have

   .. math::
      f_X(x)\geq 0

   but be aware that densities can be larger then 1.

Cumulative Distribution Function

==========

The cumulative distribution function brings the discrete and continuous RV's together. For a RV X the cumulative distribution function (often called the distribution function) is defined as:

.. math::
   F_X(x) = \P(X\leq x)

Note that $x\in\setR$ even in case X is a discrete RV. We have:

.. math::
   F_X(x) = \begin{cases}
   \sum_{k=-\infty}^{\lfloor x\rfloor} p_X(k) &\text{Discrete X}\\
   \int_{-\infty}^{x} f_X(y)\, dy &\text{Continuous X}
   \end{cases}

Below a probability mass function p_X is plotted and the corresponding F_X.

.. exec_python:: rvPMF_CDF rvCumulative
   :linenumbers:
   :code: shutter
   :Code_label: Show code for figure
   :results: hide

   import numpy as np
   import matplotlib.pyplot as plt

   plt.clf()

   pX = np.array([0,0,0,0,1,2,4,5,7,6,5,3,1,0,0,0])
   pX = pX / np.sum(pX)
   x = np.arange(len(pX))-7
   cpX = np.cumsum(pX)

   plt.subplot(211)
   plt.title(r"Probability Mass Function")
   plt.stem(x, pX, use_line_collection=True)
   plt.subplot(212)
   plt.title(r"Cumulative Distribution Function")
   plt.step(x, cpX, where='post')
   plt.savefig('source/figures/cumprobfunc.png')

And a plot of a probability density function and its corresponding cumulative distribution function.

The cumulative distribution function follows from the probability density function by integration. We can go the other way as well:

.. math::

$$f_X = \frac{d}{dx} F_X$$

With some mathematical leniency we can say that this also holds for a discrete random variable.

Expectation and Variance

The **expectation** $\E(X)$ of a random variable is the value that is to be expected 'on average'. That is if we repeat the random experiment a large number of times the average value of all obtained numbers is called the expectation.

Consider a simple die with 6 sides and assume it to be a fair die. Let X denote the number (from 1 to 8) thrown with the die. Then we have:

$$p_X(x) = \begin{cases}\tfrac{1}{6} &: 1\leq x \leq 6\\ 0 &: \text{elsewhere}\end{cases}$$

Consider the experiment we throw the die 6,000,000 times then we intuitively expect we throw any possible outcome 1,000,000 times. Leading to average:

$$\frac{1,000,000 \times 1 + \cdots 1,000,000 \times 6}{6,000,000}$$

which is equal to

$$\frac{1}{6}\times 1 + \cdots + \frac{1}{6}\times 6$$

So we take the sum of $x p_X(x)$ for each possible outcome x. That is exactly how the expectation is defined. Evidently the number of times we see one of the six possible values will not be exactly 1,000,000 but something close to it.

.. proof:definition:: Expectation

For a discrete RV we define the expectation as:

$$\E(X) = \sum_{x=-\infty}^{\infty} x\,p_X(x)$$

For a continuous RV the summation becomes an integral and thus the definition of the expectation becomes:

$$\E(X) = \int_{-\infty}^{\infty} x\,f_X(x)\,dx$$

Given a discrete random variable with probability mass function p_X a new random variable is constructed as $Y=g(X)$, i.e. if we get a value for X (say x) we just calculate $y=g(x)$ as a sample from the random variable $Y=g(X)$. A simple example would be to take X to be the random variable representing the outcome of throwing a die and set $Y=X^2$, so if we throw a 6 the value for Y would be 36.

For this new random variable Y it is simple to derive the probability mass function:

.. math::
   p_Y(y) = \begin{cases}
   \frac{1}{6} &: y\in\{1,4,9,16, 25, 36\} \\
   0 &: \text{otherwise}
   \end{cases}

And using this expression for the probability mass function for Y the expectation for Y can be calculated. In other cases it is not so simple to come up with an expression for p_Y and then the following result comes in handy.

.. proof:theorem:: $\E(g(X))$

   For a discrete random variable X and a function $g: \setR\rightarrow\setR$ we have:

   .. math::
      \E(g(X)) = \sum_{x=-\infty}^{\infty} g(x) p_X(x)

   For a continuous random variable with probability density function f_X we have

   .. math::
      \E(g(X)) = \int_{-\infty}^{\infty} g(x) f_X(x) dx

Using the above theorem we can prove an important property of the expectation: the **scaling property**. In case we make a new RV by multiplying the RV X with a constant value a and adding a constant b we have:

.. proof:theorem:: Scaling Random Variable

   .. math::
      \E(a X + b) = a\E(X) + b

.. proof:proof::

   We give the proof for a continuous RV.

   .. math::
      \E(aX+b) &= \int_{-\infty}^{\infty} (ax+b)\,f_X(x)\,dx\\

```
&= \int_{-\infty}^{\infty} a x\,f_X(x)\,dx +
\int_{-\infty}^{\infty} b\,f_X(x)\,dx\\
&= a\,\int_{-\infty}^{\infty} x\,f_X(x)\,dx +
   \,\int_{-\infty}^{\infty} f_X(x)\,dx\\
&= a \E(X) + b
```

The expectation of a random variable X tells us what value to expect on average. It does not say anything about the spread of individual values from the random variable.

The (population) variance is a way to quantify the spread around the mean and is defined as:

.. proof:definition:: Variance

   The (population) **variance** $\Var(X)$ is defined as:

   .. math::
      \Var(X) = \E((X-\E(X))^2)

   it equals the expected quadratic difference from the mean of X.

Note that the variance is expressed in the square of the unit in which the random variable is expressed (e.g. if X is expressed in meters (m) than $\Var(X)$ is expressed in meters squared (m^2). Taking the square of the variance leads to the **standard deviation** expressed in the same units as the random variable and will be denoted as $\std(X)$.

.. proof:definItion:: Standard Deviation

   The standard deviation is the square root of the variance:

   .. math::
      \std(X) = \sqrt{\Var(X)}

For the variance we have the following scaling property:

.. proof:theorem:: Scaling Random Variables

   .. math::
      \Var(aX+b) = a^2 \Var(X)

Note that adding a constant does not change the variance and that the scaling factor a becomes a quadratic factor for the variance. We leave the proof as an exercise.

==========

Conditional Random Variables

Consider two discrete random variables X and Y. Then we can define the conditional probability

.. math::
  \P(X=x \given Y=y)

Note that for any value y we have a random variable $X\given Y=y$ with probability mass function

.. math::
  X \given Y=y \sim p_{X\given Y=y}(x)

The notation for conditional random variables is not the same in al literature. You often find the notation

.. math::
  p_{X\given Y}(x\given y)

which I find a bit confusing as i like to reserve the $\given$ symbol to precede the conditioning event and not a mere value. But be warned that in case you advance to Bayesian inference statistics the $\given$ symbol will be used very often to denote the dependence on parameter values.

Now consider the situation where X is a continuous random variable whereas Y is a discrete random variable. In that case we have a probability density function for the random variable $X\given Y=y$:

.. math::
  X\given Y=y \sim f_{X\given Y=y}(x)

The conditional random variable $Y\given X=x$ is a discrete random variable.

.. math::
  Y\given X=x \sim p_{Y\given X=x}(y)

And yes although the probability $\P(X=x)=0$ evidently $X=x$ can be the outcome of the random experiment (there always is some outcome and that can be x of course).

To illustrate this situation consider apples (1) and pears (0) to be the possible outcome of random variable Y and let X denote the weight of a piece of fruit (either an apple of a pear). Then we may wonder what the probability is of a piece of fruit with weight x to be a pear or an apple i.e. $\P(Y=y\given X=x)$. It is tempting to use Bayes rule directly and write

.. math::
  \P(Y=y\given X=x) = \frac{\P(X=x\given Y=y)\P(Y=y)}{\P(X=x)}
  \quad\text{THIS IS WRONG}

It is obviously wrong as X and $X\given Y=y$ are both continuous random variables (and hence the above expression evaluates to $0/0$). This naive application of Bayes rule doesn't imply that we can't use it. To make the correct use of Bayes rule a bit more intuitive then just stating the result we introduce the (admittedly sloppy) notation:

.. math::
   X\approx x

for the event $x\leq X \leq x+dx$ with probability $f_X(x)dx$ for $dx\rightarrow 0$, i.e. for infinitessimally small dx:

.. math::
   \P(X\approx x) = \P(x\leq X \leq x+dx) = f_X(x)dx

(remember that probability density times interval length is a probability). With this definition we have:

.. math::
   \P(Y=y\given X=x) = \frac{\P(X\approx x\given Y=y)\P(Y=y)}{\P(X\approx x)} =
   \frac{f_{X\given Y=y}(x) dx \P(Y=y)}{f_X(x)dx}

the dx factors in nominator and denominator cancel out and so:

.. math::
   \P(Y=y\given X=x) = \frac{f_{X\given Y=y}(x) \P(Y=y)}{f_X(x)}

where:

- $\P(Y=y\given X=x)$: The **a posteriory probability** of class y given the value x

- $\P(Y=y)$: The **a priori** probability of class y

- $f_{X\given Y=y}$: The **class conditional probability density** function for $X\given Y=y$.

- f_X: The **evidence**, i.e. the probability density for X.

==========
Probability Distributions
==========

There are a lot of probability mass functions and probability density functions for which an analytical expression is known. These functions are parameterized in the sense that only a few parameters define the value for every value x.

In this section we discuss some of the most often used functions. In a later section we describe one method to estimate the parameters given a sample from a distribution.

Discrete Distributions

A discrete distribution is completely characterized with its probability mass function p_X. Remember that for a probability mass function we should have that the sum over all possible outcomes x equals 1 and that all probabilities are greater than or equal to zero. Given that function we will calculate (or only give the result) both the expectation and the variance.

Bernoulli Distribution
.....

The simplest of all probabilistic experiments perhaps is tossing a coin. The outcome is either heads or tails, or 0 or 1. Such an experiment is called a Bernoully experiment and the corresponding distribution is called the **Bernoulli distribution**.

The Bernoulli distribution has parameter p. In case the random variable X is Bernoulli distributed we write $X\sim\Bernoulli(p)$.

The probability mass function is:

.. math::
   p_X(x) = \begin{cases}
   1-p &: x=0\\
   p &: x=1\\
   0 &: \text{elsewhere}
   \end{cases}

The expectation equals:

.. math::
   \E(X) &= \sum_{x=-\infty}^{\infty} x\,p_X(x)\\
         &= 0\times (1-p) + 1\times p\\
         &= p

and its variance:

.. math::
   \Var(X) = p(1-p)

Binomial Distribution
.....

Consider a Bernoulli distributed random variable $Y\sim\Bernoulli(p)$, and let us repeat the experiment n

times. What then is the probabilty of k successes. A success is defined as the outcome $Y=1$ for the Bernoulli experiment.

So we define a new random variable X that is the sum of n outcomes of repeated independent and identicallu distributed (iid) Bernoulli experiments.

The outcomes of X run from 0 to n and the probability of finding k successes is given as:

.. math::
  p_X(k) = P(X=k) = {n \choose k}\,p^k\,(1-p)^{n-k}

This is called the **Binomial Distribution**. For a random variable X that has a binomial distribution we write $X\sim\Bin(n,p)$.

The expectation is:

.. math::
  \E(X) = n\,p

and the variance:

.. math::
  \Var(X) = n\,p\,(1-p)

.. exec_python:: rvPMF rvDiscrete
  :linenumbers:
  :code: shutter
  :Code_label: Show code for figure
  :results: hide

  import numpy as np
  import matplotlib.pyplot as plt

  from scipy.stats import binom

  n = 20
  for p, c in zip([0.05, 0.4, 0.8], ['r', 'g', 'b']):
      plt.stem(np.arange(0, n+1), binom.pmf(np.arange(0, n+1), n, p),
               linefmt=c, markerfmt=c+'o',
               label=f'p={p}')

      plt.legend()
  plt.xticks(np.arange(0, n+1))
  plt.savefig('source/figures/binomialdistribution.png')

Uniform Distribution
.....

The **discrete uniform distribution** is used in case all possibe outcomes of a random experiment are equally probable. Let $X\sim\Uniform(a,b)$, with $a<b$ and $a,b\in\setZ$, then

.. math::

$$
p_X(x) = \begin{cases}
\frac{1}{b-a+1} &: a\leq x \leq b\\
0 &: \text{elsewhere}
\end{cases}
$$

Its expectation is:

$$
\E(X) = \frac{a+b}{2}
$$

and its variance

$$
\Var(X) = \frac{(b-a+1)^2-1}{12}
$$

Continuous Distributions

Uniform Distribution
...

The **continuous uniform distribution** characterizes a random experiment in which each real valued outcome in the interval $[a,b]\subset\setR$ is equally probable. The probability density function of $X\sim\Uniform(a,b)$ is given by

$$
f_X(x) = \begin{cases}
\frac{1}{b-a} &: a \leq x \leq b\\
0 &: \text{elsewhere}
\end{cases}
$$

The expectation of the continuous uniform distribution is

$$
\E(X) = \frac{a+b}{2}
$$

and its variance

$$
\Var(X) = \frac{(b-a)^2}{12}
$$

Observe that:

- the normal distribution is symmetrical around its expectation,

- the standard deviation

Normal Distribution
...

The **normal distribution** is undoubtly the most often used distribution in probability and machine learning. For good reasons: a lot of natural phenomena of random character turn out to be normally distributed (at least in good approximation). Furthermore the normal distribution is a nice one to work with from a mathematical point of view.

Let $X\sim\Normal(\mu,\sigma^2)$ then the probability density function is:

.. math::
   f_X(x) = \frac{1}{\sigma\sqrt{2 \pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}

where the parameters μ and σ^2 are called the mean and variance for good reason:

.. math::
   &\E(X) = \mu\\
   &\Var(X) = \sigma^2

==========
Joint Distributions
==========

Two Discrete Distributions
==========

Consider a random experiment where we observe two random variables X and Y. Assume both RV's are discrete. The probability for outcomes $X=x$ and $Y=y$ is given by the **joint probability mass function p_{XY}

.. math::
   p_{XY}(x,y) = \P(X=x,Y=y)

Again the sum of all possible outcomes of the experiment should be 1:

.. math::
   \sum_{x=-\infty}^{\infty} \sum_{y=-\infty}^{\infty} p_{XY}(x,y) = 1

Note that we don't have to run the summation over all of $\setZ$ in case we know that $p_{XY}(x,y)=0$ outside a given interval for x and y.

Let's consider an example where X can have the values 1,2 and 3, and Y can take on the values 1 and 2. Assume we know all probabilities we can set up the **joint distribution table**:

.. list-table::

```
   :widths: 1, 1, 2
   :header-rows: 1

   * - $x$
     - $y$
     - $p_{XY}(x,y)$

   * - 1
     - 1
     - 0.10


   * - 1
     - 2
     - 0.20

   * - 2
     - 1
     - 0.20

   * - 2
     - 2
     - 0.25

   * - 3
     - 1
     - 0.15

   * - 3
     - 2
     - 0.10
```

The joint distribution probability functions (and hence the joint distribution table) is all there is to know about the random experiment. So we may also calculate $P(X=x)$ from it:

```
.. math::
   \P(X=x) = p_X(x) = \sum_{y} p_{XY}(x,y)
```

We can also calculate:

```
.. math::
   \P(X=1\given Y=1) &= \frac{\P(X=1,Y=1)}{\P(Y=1)}\\
                     &= \frac{p_{XY}(1,1)}{\sum_x p_{XY}(x,1)}\\
                     &= \frac{0.10}{0.10+0.20+0.15}\\
                     &= \frac{0.10}{0.45}
```

The indepence of two discrete random variables X and Y is defined as:

```
.. math::
   \forall x, \forall y : p_{XY}(x,y) = p_X(x)\, p_Y(y)
```

Two Continuous Distributions

In case X and Y are continuous random variables we have a joint probability density function $f_{XY}(x,y)$ let $A\subset\setR^2$ then

.. math::
   \P((X,Y)\in A) = \iint_A f_{XY}(x,y)\,dxdy

Note that f_{XY} is a density function in two variables. Therefore $f_{XY}(x,y)dxdy$ is a probability.

The integral of the pdf over $\setR^2$ should equal 1 (because the universe in this case is $\setR^2$:

.. math::
   \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f_{XY}(x,y)dx dy = 1

Like for the discrete counterpart we can calculate f_X from f_{XY}:

.. math::
   f_X(x) = \int_{-\infty}^{\infty} f_{XY}(x,y)dy

Independent continuous random variables X and Y are defined with:

.. math::
   \forall x, \forall y : f_{XY}(x,y) = f_X(x)\,f_Y(y)

Multiple Random Variables

Now we consider the case that we are observing n random variables from one random experiment. In case that $X_1,X_2,\ldots,X_n$ are all discrete RV's we have:

.. math::
   p_{X_1 X_2\cdots X_n}(x_1,\ldots,x_n) = \P(X_1=x_1,\ldots,X_n=x_n)

For continuous RV's we have:

.. math::
   \P((X_1,\ldots,X_n)\in A) = \int\cdots\int_A f_{X_1 X_2\cdots X_n}(x_1,\ldots,x_n) dx_1 dx_2\cdots dx_n

Calculations with Random Variables

Scaled and Shifted Distributions

We have seen simple calculations with continuous random variables before where we defined $Y=aX+b$ where X is a random variable and a and b are constant values. When discussing the expectation and variance of random variables we have shown that

.. math::
  &\E(aX+b) = a\E(X)+b\\
  &\Var(aX+b) = a^2\Var(X)

Evidently Y is a random variable itself. In general when you make calculations involving random variables the result will be a random variable.

But what about the distribution of Y. For this operation on X the distribution of Y can be expressed as a scaled and shifted version of the distribution of X.

We start with $Y=aX+b$ where $a>0$. The cumulative distribution function F_Y then is:

.. math::
  \begin{align}
  F_Y(y) &= P(Y\leq y) \quad && \text{by definition}\\
      &= P(aX+b\leq y) && \text{substituting } Y=aX+b\\
  &= P(aX \leq y - b)&& \text{rearranging terms}\\
  &= P(X \leq \frac{y-b}{a}) && \text{because }a>0\\
  &= F_X(\frac{y-b}{a}) && \text{by definition}
  \end{align}

so F_Y is a translated and horizontally scaled version of the distribution F_X. Using the fact that f_X is the derivative of F_X we have:

.. math::
  p_Y(y) &= \frac{d}{dy}F_y(y)\\
      &= \frac{d}{dy} F_X(\frac{y-b}{a})\\
  &= F^\prime_X(\frac{y-b}{a}) \frac{1}{a}\\
  &= \frac{1}{a}p_X(\frac{y-b}{a})

Again we have the horizontal shift and horizontal scaling but now also a vertical scaling (that will keep the area under the curve equal to 1).

Such scaled and shifted versions of a pdf are very common. For instance in case you need a uniformly distributed number in the range from $-\pi/6$ to $\pi/6$ but you only have a random number generator $X\sim\Uniform(0,1)$ every programmer intuively will write code defining the random variable Y:

.. math::

$$Y = \frac{\pi}{6}(2 X - 1)$$

Scaling and shifting of the normal distribution is of great practical importance as well. Given a random variable $X\sim\Normal(\mu,\sigma^2)$ we have:

$$Z = \frac{X-\mu}{\sigma} \sim \Normal(0,1)$$

The random variable Z is often called the **Z-score**. It transforms any normally distributed value into a standard normal distributed value.

In the old pre computer days only the standard normal distribution was available in tables and scaling and shifting was needed to obtain the values for any normal distribution.

Whereas the Z-score takes a $\Normal(\mu, \sigma^2)$ distribution and transforms it into a standard normal distribution $\Normal(0,1)$ we can go the opposite way as well. Let $X\sim\Normal(0,1)$ then

$$Y = \sigma X + \mu \sim \Normal(\mu,\sigma^2)$$

The Sum of Random Variables

Consider the random variables X and Y and their sum $Z=X+Y$. First consider the expectation:

$$\E(Z) = \E(X+Y) = \E(X) + \E(Y)$$

To prove this we have to start with the joint probability mass or density function. Here we assume both X and Y are discrete RV's. Then:

$$\begin{aligned}
\E(Z) = \E(X+Y) &= \sum_x \sum_y (x+y)\, p_{XY}(x,y)\\
&= \sum_x \sum_y x\, p_{XY}(x,y) + \sum_x \sum_y y\, p_{XY}(x,y)\\
&= \sum_x x \sum_y p_{XY}(x,y) + \sum_y y \sum_x p_{XY}(x,y)\\
&= \sum_x x\, p_X(x) + \sum_y y\, p_Y(y)\\
&= \E(X) + \E(Y)
\end{aligned}$$

For the variance of Z things are a bit more complicated:

$$\Var(X+Y) = \Var(X) + \Var(Y) + 2\Cov(X,Y)$$

where

$$\Cov(X,Y) = \E\left((X-\E(X)) (Y-\E(Y) \right)$$

is called the **covariance** of X and Y. Covariance and independence are related but certainly not equal. We have that if X and Y are independent then $\Cov(X,Y)=0$. Please note that a zero covariance (we also say then that X and Y are **uncorrelated**) does not imply that X and Y are independent.

Descriptive Statistics

In practice the probability (density) functions are seldomly known. We just have a lot of data that we think come from one distribution. Almost always we assume the data values are independent realizations of the same random variable.

What we would like to do in many cases is to estimate the probability (density) function of the underlying random variable. Here we assume that the random variable is a continuous one. So the problem is: given the observations

$$x\ls 1,x\ls 2,...,x\ls m$$

The indexing with a superscript may strike you as a little odd but it is quite common in machine learning where sometimes a lot of indices are needed to identify an object. We use parentheses not to confuse x^2 with x^2 (sic!).

Histograms

A first step in any statistical or machine learning tasks is to get to know your data. The simplest way to get an overview of all possible values in the dataset $x\ls i$ is to make a histogram of the data.

A histogram is a visual representation of a frequency table. For continuous data first we divide the value range $\setR$ into bins (intervals) and then we count how many of the observation $x\ls i$ are within each bin. The bins are chosen such that they cover the entire range of values in the data set without overlap and that the bins are contiguous.

We then make a histogram of the bin values. Note that for a histogram of continuous data the bars in the plot should have no space inbetween them.

Note that the choice for the bins is somewhat arbitrary. In case you select to few bins you can not assess the distribution of values in your dataset. But in case of too many bins (i.e. a bin width that is

very small) you will probably end up with 0 or 1 data value in bins.

There are methods to select the bin width given the size of the data set and also methods that look at statistics of the data. We strongly suggest you experiment with the bin width.

.. ipython:: python

```
from matplotlib.pylab import *
from sklearn import datasets
bc = datasets.load_breast_cancer()
x = bc.data[:,0]
y = bc.target
x0 = x[y==0]
x1 = x[y==1]

be = linspace(x.min(), x.max(), 50)
h0, _ = histogram(x0, bins=be)
h1, _ = histogram(x1, bins=be)
bar(be[:-1],  h0, be[1]-be[0], color='r', alpha=0.5);
@savefig mean_radius_breast_cancer.png
bar(be[:-1],  h1, be[1]-be[0], color='g', alpha=0.5);
```

In the figure above the histograms of two datasets are drawn. The data is taken from a study into breast cancer. The feature value is the mean radius of a spot on the mammography. The red histogram is of malignant spots, the green histogram of benign spots.

This plot shows the central problem in machine learning: one feature of objects is most often not enough to classify the objects without error as can be seen by the fact that the two histograms overlap. More features are needed and even then there will probably be cases where classification will be wrong.

Statistics
==========

The most well known statistics of a set of numbers are the **mean** and the **variance**. The mean simply is:

.. math::
   m = \frac{1}{m} \sum_{i=1}^m x\ls i

and the **variance** is

.. math::
   s^2 = \frac{1}{m} \sum_{i=1}^m (x\ls i - m)^2

The square root of the variance s is called the **standard deviation**.

Carefully note the difference between the expectation and variance of

a distribution and the mean and variance of a data set. It is true that in the limit where $m\rightarrow\infty$ the two are equal. But in general m and s^2 are only estimates of $\E(X)$ and $\Var(X)$.

There are many more statistics that can be calculated given a data set of values. Alternatives for the mean are the **mode** or the **median**. Alternatives for standard deviation (measure of spread) are quartile distances among others.

Estimators for Distribution Parameters

A lot of statistical and machine learning methods are based on learning the probability distribution from data. Assume that random variable X is distributed according to the probability mass function $p_X(x;\theta)$ or probability density function $f_X(x;\theta)$ where θ are the parameters characterizing the distribution.

The question then is: can we give estimate of the parameters θ given a sample of m values of this distribution ? In machine learning it would be called the learning set.

A very common way to come up with an estimator is the maximum likelihood estimator.

Maximum Likelihood Estimator

Consider a discrete distribution with parameters θ. The pmf is then written as $\P(X=x) = p_X(x;\theta)$. We assume the values in the sample $x\ls 1,\ldots,x\ls m$ are realization of m independent and identically distributed random variables $X_1,\ldots,X_m$. Therefore:

.. math::
   \P(X_1=x\ls 1, X_2=x\ls 2, \ldots, X_m=x\ls m) =
   p_{X_1 X_2\cdots X_m}(x\ls1,\ldots,x\ls m \,;\, \theta)

where $p_{X_1 X_2\cdots X_m}$ is the joint probability mass function.

Assuming the random variables are iid we have:

.. math::
   p_{X_1 X_2\cdots X_m}(x\ls1,\ldots,x\ls m \,;\, \theta) =
   \prod_{i=1}^m p_X(x\ls i\, ; \,\theta)

This probability can also be interpreted as a function of θ, it then gives the likelihood for the parameter θ to explain the data:

.. math::
   \ell(\theta) = \prod_{i=1}^m p_X(x\ls i\, ; \,\theta)

The maximum likelihood estimator then is:

.. math::
  \hat \theta = \arg\max_{\theta} \ell(\theta)

In practice we often look at the log likelihood:

.. math::
  \hat\theta &= \arg\max_{\theta} \log\ell(\theta)\\
        &= \arg\max_{\theta} \sum_{i=1}^m \log\left(p_X(x\ls i\, ; \,\theta)\right)

In the following sections we will look at a few distributions and
their MLE's. Evidently the final expression for the estimator depends
on the probability mass function p_X and its dependence on θ.

In case X is a continuous random variable the probability density
function is used:

.. math::
  \hat\theta = \arg\max_{\theta} \sum_{i=1}^m \log\left(f_X(x\ls i\, ; \,\theta)\right)

Also note that the base of the logarithm is not important for our
purpose.

Bernoulli Distribution
=======================

The one parameter that needs to be estimated for the Bernoulli
distribution is the probability $p = \P(X=1)$. Using the sample
$x\ls 1,\ldots,x\ls m$ an obvious way to calculate this is:

.. math::
  \hat p = \frac{\sum_{i=1}^m x\ls i}{m}

i.e. we just calculate the relative frequency of the value $X=1$ in
our sample.

This estimator is the **maximum likelihood estimator** (MLE) for p as we
will show. The sample set of values are the outcomes of m iid random
variables $X_1,\ldots,X_m$. Assuming $X_i\sim\Bernoulli(p)$ we have:

.. math::
  \P(X_1=x\ls 1,\ldots,X_m=x\ls m\,;\,p) = \prod_{i=1}^m \P(X_i=x\ls i\,;\,p)
  = \prod_{i=1}^m p^{x\ls i} (1-p)^{1-x\ls i}

This probability when viewed as a function of p for fixed sample set
$x\ls 1,\ldots,x\ls m$ is called the likelihood function:

.. math::
  \ell(p) = \prod_{i=1}^m p^{x\ls i} (1-p)^{1-x\ls i}

The maximum likelihood estimator for p now finds the value for p
that maximizes $\ell(p)$. Instead of maximizing ℓ we maximize the

logarithm of ℓ leading to the maximum likelihood estimator:

.. math::
  \hat p = \arg\max_{p} \log(\ell(p))

where

.. math::
  \log(\ell(p))
  &= \log\left(\prod_{i=1}^m p^{x\ls i} (1-p)^{1-x\ls i}\right)\\
  &= \sum_{i=1}^m \left({x\ls i}\log(p) + (1-x\ls i)\log(1-p) \right)\\
  &= \left(\sum_{i=1}^m x\ls i\right) \log(p) + \left(\sum_{i=1}^m (1-x\ls i)\right) \log(1-p)\\
  &= m_1\log(p) + m_0\log(1-p)

where m_1 is the sum of the $x\ls i$, i.e. the number of values 1 in the sample and m_0 is the number of values 0 in the sample.

We can find the maximal value by calculating the derivative of ℓ with respect to p and solving for $\ell'(p)=0$. We have:

.. math::
  \frac{d}{dx}\ell(p) = \frac{m_1}{p} - \frac{m_0}{1-p}

setting this equal to 0 and solving for p leads to:

.. math::
  \hat p = \frac{m_1}{m_1+m_0} = \frac{m_1}{m}

With relatively small sample sizes you might run into the situation where $m_1=0$ or $m_0=0$. This will lead to either $p=0$ or $1-p=0$, a situation that in a lot of machine learning applications you would like to avoid. In such cases the **add 1 Laplace smoothing** is often used. Now we use

.. math::
  \hat p = \frac{m_1+1}{m+2}

Note that in case we have no data at all we just assume that $\hat p=\tfrac{1}{2}$, i.e. a uniform distribution.

Categorical Distribution
========================

For the categorical distribution there are K probabilities to be estimated from the sample $x\ls 1,\ldots,x\ls m$. The possible values range from 1 to K. Let's denote the number of outcomes equal to i with m_i, i.e.:

.. math::
  m_i = \sum_{j=1}^{m} \left[x\ls j = i\right]

here $\left[B \right]$ is the Iverson bracket:

.. math::

\left[B \right] = \begin{cases}
1 &: B \text{ is True}\\
0 &: \text{otherwise}
\end{cases}

The MLE for p_i is:

.. math::
\hat p_i = \frac{m_i}{m}

The estimator using add 1 Laplace smoothing is given by:

.. math::
\hat p_i = \frac{m_i+1}{m+K}

Normal Distribution
==========

Let $X\sim\Normal(\mu,\sigma^2)$ and let $x\ls 1,\ldots,x\ls m$ be a sample from X.

The MLE estimators for μ and σ^2 are:

.. math::
\hat \mu = \frac{1}{m}\sum_{i=1}^{m} x\ls i

and

.. math::
\hat \sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (x\ls i-\hat\mu)^2

Random Vectors
==========

We have looked at multiple random variables before. Let $X_1,X_2,\ldots, X_n$ be random variables then we defined the **joint distribution function** $F_{X_1,X_2,\ldots,X_n}$:

.. math::
F_{X_1,X_2,\ldots,X_n}(x_1,x_2,\ldots,x_n) = \P(X_1\leq x_1
\cap \cdots \cap X_n\leq x_n)

In case all random variables are continuous the **joint probability density function** is defined as

.. math::
f_{X_1,X_2,\ldots,X_n}(x_1,x_2,\ldots,x_n) =
\frac{\partial^n F_{X_1,X_2,\ldots,X_n}}{\partial x_1\cdots\partial x_n}
(x_1,x_2,\ldots,x_n)

To calculate the probability for an event such that the values of the

random variables are within a subset :math:‘A\subset\setR^n‘ we calculate the multivariate integral:

.. math::
   P((X_1,\ldots,X_n)\in A) =
   \int\cdots\int_{A} f_{X_1,X_2,\ldots,X_n}(x_1,x_2,\ldots,x_n)
   dx_1\cdots dx_n

Note that :math:‘dx_1dx_2\cdots dx_n‘ is an infinitesimal small hypervolume element. Multiplying the probability *density* with this volume element results in a probability, adding these probabilities for all volume elements in the set :math:‘A‘ results in the desired probability.

Very often there is the need to characterize all random variables with one entity. Think of situations where there are hundreds or even thousands of random variables involved, a situation that often occurs in practice. For this the **random vectors** are introduced:

.. math::
   \v X = \matvec{c}{X_1\\ X_2\\ \vdots\\ X_n}

A random vector is a vector whose elements are random variables. With this notation the distribution function can be abbreviated as :math:‘F_{\v X}‘ and the probability density function as :math:‘f_{\v X}‘. Using the vectorial notation the probability for the event :math:‘\v X\in A‘ equals:

.. math::
   P(\v X \in A) = \int_A f_{\v X}(\v x) d\v x

For discrete random variables the distribution function is defined equivalently but instead of defining a probability density function the **probability mass function** is defined:

.. math::
   p_{X_1,X_2,\ldots,X_n}(x_1,x_2,\ldots,x_n) = P(X_1=x_1\cap \cdots \cap X_n=x_n)

or equivalently in vectorial notation:

.. math::
   p_{\v X}(\v x) = P(\v X =\v x)

The probability :math:‘\v X\in A‘ for a discrete random variable is then a sum:

.. math::
   P(\v X\in A) = \sum_{\v x \in A} p_{\v X}(\v x)

and for a continuous random variable we get an integral:

.. math::
   P(\v X\in A) = \int_{\v x \in A} f_{\v X}(\v x) d\v x.

Note that :math:‘f_{\v X}(\v x)‘ is the probability density that multiplied with the infinitesimal volume :math:‘d\v x‘ is a probability. Thus we are summing probabilities and in the limit for inifinitesimal small intervals (:math:‘dx\rightarrow0‘) we are integrating a probability density.

It is possible to define random vectors with a mixture of discrete and continuous random variables. In this lecture series we stick with either pure discrete or pure continuous random vectors.

Consider two random vectors :math:‘\v X‘ and :math:‘\v Y‘. We define these two random vectors to be **independent** in case each element :math:‘X_i‘ is independent of each element :math:‘Y_j‘. Note that elements of :math:‘\v X‘ might be dependent! In a formula we can express this notion of independence with:

.. math::
   f_{X_1,X_2,\ldots,X_n,Y_1,Y_2,\ldots,Y_m}(x_1,x_2,\ldots,x_n,y_1,y_2,\ldots,y_m) =
   f_{X_1,X_2,\ldots,X_n}(x_1,x_2,\ldots,x_n)\,
   f_{Y_1,Y_2,\ldots,Y_m}(y_1,y_2,\ldots,y_m)

This formula is again a clear demonstration of the power of using vectors (linear algebra) to describe multivariate statistics where we can write:

.. math::
   f_{\v X, \v Y}(\v x, \v y) = f_{\v X}(\v x)\, f_{\v Y}(\v y)

==========
 Overview of Machine Learning
==========

Machine learning is all about interpreting data, lots of data. Observing and characterizing one object or situation in the real world through sensors or formal description results in what is called a **data (or feature) vector** $\v x\in\setR^n$. Such a vector can be low dimensional, say $n=3$ for an RGB vector describing the color of a pixel in the image, or very high dimensional like a video sequence of a few seconds (amounting to megabytes of elements in the data vector).

In our restricted view on machine learning we want to assign a result vector $\hat{\v y}$ to each possible observation $\v x$ where $\hat{\v y}$ is either a label characterizing the object or situation associated with that feature vector or it is a (set of) numerical values.

In supervised machine learning a set of examples $(\v x\ls i, \v y\ls i)$ is available for learning a function f that maps the feature vector $\v x$ onto the desired target value $\hat{\v y}$. We assume that the ’true’ target $\v y$ is given in the learning set (hence the term **supervised learning**). In practice the mapping f that is learned will not always produce the correct target value.

In these lecture notes we restrict ourselves to the principal categories of machine learning systems, namely:

Dimensionality Reduction.

The dimension of the feature vector representing one object or situation can be quite high. Consider an image of several mega pixels or worse, imagine a video sequence. Let's take an image of size 1000×1000. The feature vector $\v x$ then is a vector in $\setR^{3000000}$ assuming we have a red, green and blue value for each pixel. Now assume that an R,G and B value takes a byte to encode than there are $(256\times256\times256)^{10000000}$ possible images that are distinct from each other. But for most of these images it is true that they are semantically indistinguishable from other images (a horse is a horse of course of course) or depict totally nonsense to the human eye or are simply noise. The point is that the effective dimension (degrees of freedom) is much lower than the dimension of the raw data.

In machine learning several methods are known to reduce the dimensionality of the data without changing its interpretation. As a pre processing step it can help to reduce the time complexity of machine learning algorithms. Evidently dimensionality reduction is also central in the (lossy) compression of data.

In these notes we will look at **principal component analysis** as a way to reduce the dimensionality of our data.

Regression.

Now consider the case that our feature vector gives a lot of numerical data describing a house (area code, number of bedrooms, area, etc). Given that feature vector we would like to predict the house price. So $\hat y = f(\v x)$ is the predicted house price.

In general the regression task is to come up with a function f to be learned from the learning set that minimizes the error$\v y - \hat{\v y}$, i.e. the differences between the targets $\v y\ls i$ and the predicted value $f(\v x\ls i)$.

As with most supervised machine learning methods regression doesn't search for an arbitrary function to match the data. Instead we use a parameterized function for the prediction $\hat{\v y} = f(\v x, \v p)$ where the function f is fixed (to be chosen a priori) and all that is left to do is find the parameter vector $\v p$ that best fits the data.

In these lecture notes we look at linear regression methods. The 'hello world' example of a linear regression problem is the task to fit a straight line through a set of data points. It will become clear that linear regression does not owe it name to the straight line fitting, Much more complex functions can be fitted to data.

Regression is not restricted to mappings from $\setR^n$ to

$\setR$, i.e. resulting in scalar values. An example where the result $\v y$ is a vector as well is to predict *where* in am image an object is to be seen at what size. Then the input feature vector is a vector representation of the input image and the result is a vector $\v y$ representing the bounding box encompassing the object visible in the image (something like (y_1,y_2) for the position of the bounding box and (y_3,y_4) for the width and height of the boundingbox.

We also take a brief look at using a neural network for regression. But reading that section is best postponed to after learning about the use of neural networks for classification.

Classification

In classification the feature vector $\v x$ is mapped onto a label y (or set of probabilistic labels $\v y$) characterizing the object that is represented by the feature vector. An example is classifying pieces of fruit as either apple, pear or grapes etc given a picture of the piece of fruit.

Each point in the **feature space** $\setR^n$ is assigned to one of the possible labels. With this the feature space is divided into regions corresponding with one of the labels. The boundaries of these regions are called decision boundaries.

In these notes we will discuss several methods for classification: naive Bayes classification, logistic regression and neural networks.

Clustering

In clustering you are given a dataset with only feature vectors and no target value. The task then is to divide these feature vectors (i.e. points in feature space) into clusters such that points in one cluster share a common interpretation and points in different clusters are semantically different.

Clustering is a useful technique for several reasons. It is used in exploratory data analysis to look for patterns in the data (which articles in a supermarked tend to end up in the same cart and what does that tell us about these customers). It is also of more technical interest. For instance when considering all the colors in an image (a lot of points in 3D space) we can often find a limited number of clusters in color space. The cluster centers than can serve as a limited set of colors to paint the image without too much visual degradation of the image appearance.

In these lecture notes we consider a simple clustering algorithm: k-means clustering.

Besides the four basic categories of machine learning mwthods we will look at the methodology for applying machine learning in practice. How to utilize the data for learning and testing. How to quantify the

performance of your machine learning system.

It is important to note that this introductory course on machine learning barely scratches the surface of the field of machine learning. Not all classical methods are discussed, the support vector machine and all other kernel methods in learning, the state of the art in machine learning only two decades ago are skipped. Also methods like decision trees are not discussed in these notes. Still these methods are not without their special merits and use even today. For instance decision trees are useful when dealing with categorical data without the need to transform categorical data into numerical data as is needed for most other methods.

And besides not mentioning methods from the past, we also don't look at state of the art modern methods. Auto encoders, transformer networks, LSTM networks, generative adversary networks and diffusion models are only a few of the modern developments that we did not discuss. These will looked at in master level courses on machine and deep learning.

Principal Component Analysis

The underlying assumption in principal component analysis (PCA) is that the data is 'concentrated' in an affine subspace of dimension d of feature space (of dimension n). We model our data with the random vector X with

.. math::
  \v\mu = E(\v X), \quad \Sigma = \Cov(\v X)

To make it into a real subspace we subtract the mean $\v \mu$ from $\v X$:

.. math::
  \v X_z = \v X - \v \mu

Maximizing Directional Variance

We have allready seen that the directional variance in direction $\v r$ is given by

.. math::
  \Var(\v r\T \v X_z) = \v r\T \Cov(\v X_z) \v r = \v r\T \Sigma \v r

The direction of maximal variance (let's call it direction $\v u_1$ then is

.. math::
  \v u_1 = \arg\max_{\v r: \|\v r\|=1} \v r\T \Sigma \v r

Conditional optimization can be done with the Lagrange multipliers method
to incorporate the constraint in the function to be optimized:

.. math::
   \v u_1, \lambda_1 = \arg\max_{\v r, \lambda_1} \v r\T \Sigma \v r - \lambda_1 (\|\v r\|^2-1)

The conditions for a maximum are then given by

.. math::
   \Sigma \v u_1 - \lambda_1 \v u_1 &= 0\\
   \v u_1\T\v u_1 - 1 &= 0

The second equation is of course the condition we started with. The
first equation is the important one:

.. math::
   \Sigma \v u_1 = \lambda \v u_1

showing that $\v u_1$ is an eigenvector of Σ. The directional
variance then is $\v u_1\T \Sigma \v u_1 = \lambda$ (remember
eigenvectors of a real symmetric matrix have unit length). As we are
looking for the maximal value $\v u_1$ should be the eigenvector with
maximal eigenvalue λ_1.

Next we could look for another direction, say $\v u_2$ that is
perpendicular to $\v u_1$ and. under that extra constraint, maximizes
the variance. You correctly guessed, $\v u_2$ is the eigenvector with
the second largest eigenvalue. The same reasoning applies for all the
eigenvectors.

.. Next we need to find the direction, *perpendicular to $\v u_1$*, that
   has maximum variance:

   .. math::
      \v u_2, \lambda_2, \mu_2 = \arg\max_{r,\lambda,\mu}

A Linear Algebra View

Principal component analysis transforms the zero mean data in such a
way that $\v Y = A \v X_z$ has a diagonal covariance matrix. In that
case the elements of $\v Y$ are uncorrelated. It is not to difficult
to prove that $A=U\T$ where U is the matrix of eigenvectors of
Σ such that $U\T\v X_z$ has a covariance matrix D.

Proof: The eigen decomposition of Σ is

.. math::
   \Sigma = U D U\T

where the columns of U are the eigenvectors and the diagonal
elements of the diagonal matrix D are the eigenvectors of

Σ. Because for a real symmetrix matrix the eigenvectors form an orthogonal basis we can write:

.. math::
   D = U\T \Sigma U

Remember that

.. math::
   \Cov(A\v X) = A\Cov(\v X)A\T = A\Sigma A\T

Thus we see that $\v Y=U\T \v X_z)$ leads to $\Cov(\v Y) = D$. $\blacksquare$

Thusfar we have done any dimensionality reduction whatsoever. The random vector $\v Y$ is of the same dimension as the original vector $\v X_z$. But in case we assume that the eigen values in D are in decreasing order, the last elements of $\v Y$ are in the directions of lowest variance. It is in the eigen vector basis that we may decide to leave out the basis vectors (eigen vectors) with low variance. We use only the first k eigenvectors:

.. math::
   \v Y_k = \begin{pmatrix}Y_1\\Y_2\\\vdots\\Y_k\end{pmatrix}

We can calculate $\v Y_k$ as

.. math::
   \v Y_k = U_k\T \v X_z

where U_k is the matrix with only the first k columns (eigenvectors) of U.

The elements of the vector $\v Y_k$ are the coordinates of the projected vector X_zm with respect to the (reduced) eigenvector basis. Note that the number of elements in $\v Y_k$ is less than the number of elements in $\v X$ (i.e. $k<n$) and we thus have reduced the number of real values needed to represent the vector $\v X$. That representation is only approximatively though.

With respect to the original basis we have

.. math::
   \v X_\text{zm} & \approx U_k \v Y_k\\
   & \approx U_k U_k\T \v X_\text{zm}

Note that in case $k=n$, i.e. $U_k = U$ we have an equality in the above expression.

Summarizing:

.. math::
   \v Y_k = U_k\T (\v X - E(\v X))

is the coordinate vector of $\v X - E(\v X)$ with respect to the
eigenvector basis and

.. math::
   \v X & \approx U_k \v Y_k + E(\v X)\\
        & \approx U_k U_k\T (\v X - E(\v X)) + E(\v X)

PCA in Practice

In practice PCA is used very often for dimensionality reduction. In
practice however neither the expectation nor the covariance matrix are
known. All we have is a sample of $\v X$: $\v x\ls 1,\ldots,\v x\ls m$.

Given this sample we may estimate the mean and covariance matrix:

.. math::
   \widehat{E(\v X)} = \bar{\v x} = \frac{1}{m} \sum_{i=1}^{m} \v x\ls i

and:

.. math::
   \widehat{\Cov(\v X)} = S = \frac{1}{m} \sum_{i=1}^{m} (\v x\ls i - \bar{\v x})(\v x\ls i - \bar{\v x})\T

With these two the basic formulas for PCA are:

.. math::
   S &= U D U\T\\
   \v y_k &= U_k\T (\v x - \bar{\v x})\\
   \v x &\approx U_k \v y_k + \bar{\v x}

Please note that in case PCA is used for dimensionality reduction as a
means of compressing the data, you must take into accound that U_k
and $\bar{\v x}$ are needed as well to approximately reconstrict $\v
x$ from its reduced representation $\v y_k$.

Linear Regression
=================

The canonical example of linear regression, fitting a straight line
through data points, is a misleading one. It suggests that the name
linear regression stems from the straight line that is fitted. This
is not true. Linear regression is about fitting a parameterized
function, the **hypothesis** $h_{\v\theta}(\v x)$, **that is linear in
its parameters** $\v\theta$, to the data points $(\v x\ls i, y\ls i)$.

Nevertheless we also start with the canonical example: the straight
line fit in the first section. Because of its simplicity we can easily
visualize the workings of **gradient descent optimization** that is
central in most machine learning algorithms.

Univariate Linear Regression

==========

One Learning Examples at a Time

Let $(x\ls i, y\ls i)$ for $i=1,\ldots,m$ be the set of
points through which we want to fit a straight line. The hypothesis or
model function in this case is:

.. math::
   h_{\v\theta}(x) = \theta_0 + \theta_1 x

Here the parameter vector $\v\theta$ is given by:

.. math::
   \v\theta = \matvec{c}{\theta_0 \\ \theta_1}

Fitting a straight line then amounts to finding the "optimal"
parameters θ_0 and θ_1 such that
$y\ls i \approx h_{\v\theta}(x\ls i)$
for all i. Optimal parameters are found by
minimizing a **cost function**:

.. math::
   J(\v\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_{\v\theta}(x\ls i) - y\ls i \right)^2

Such that the optimal parameter vector is found with:

.. math::
   \v\theta^\star = \arg\min_{\v\theta} J(\v\theta)

All we have to do from here is to find the $\v \theta$ vector that
minimizes the cost function. For this particular cost function it is
doable to analytically calculate the unique optimal parameter vector
(we will do so in a later section). Here we will use a numerical
iterative algorithm to 'search' for the optimal parameter
vector. There are very many numerical optimization algorithms. We only
discuss one: the **Gradient Descent Algorithm** which in its (over)
simplified form can be written as:

.. math::
   \text{iterate until done:}\qquad \v\theta := \v\theta -
   \alpha \frac{\partial J(\v\theta)}{\partial \v\theta}

Evidently the stopping criterion is ill defined here. It it hard to
define such a criterion. Often the best way to define such a criterion
on is to observe the value of J while iterating and set a maximum on
the number of iterations.

The gradient vector can be easily calculated to be:

.. math::
   \frac{\partial J(\v\theta)}{\partial \v\theta} = \matvec{c}{

$$
\frac{J(\v\theta)}{\theta_0} \\
\frac{J(\v\theta)}{\theta_1}} =
\frac{1}{m}\sum_{i=1}^{m} \left(h_{\v\theta}(x\ls i) - y\ls i \right)
\matvec{c}{1\\ x\ls i}
$$

We can tidy up this equation a bit using:

$$
\hv x\ls i = \matvec{c}{1\\x\ls i}
$$

resulting in:

$$
\frac{\partial J(\v\theta)}{\partial \v\theta} =
\frac{1}{m}\sum_{i=1}^{m} \left(\v\theta\T \hv x\ls i - y\ls i \right)\hv x\ls i
$$

The algorithm then is:

$$
\text{iterate until done:}\qquad \v\theta := \v\theta -
\alpha \frac{1}{m}\sum_{i=1}^{m} \left(\v\theta\T \hv x\ls i - y\ls i \right)\hv x\ls i
$$

Note that in the formulation given above the entire learning set (i.e. all m examples) are used to calculate every step in the gradient descent procedure. Later on in these notes when discussing neural networks we will restrict the calculation of the gradient to a subset of all examples (and in the extreme case we use only one example to calculate the step to be made in the gradient descent procedure). This type of learning is called **stochastic gradient descent**.

Vectorization of the Learning Set

We consider the same problem as in the previous section, i.e. minimization of the cost function:

$$
\begin{aligned}
J(\v\theta) &= \frac{1}{2m} \sum_{i=1}^{m} \left(h_{\v\theta}(x\ls i) - y\ls i \right)^2\\
&= \frac{1}{2m} \sum_{i=1}^{m} \left(\v \theta\T \hv x\ls i) - y\ls i \right)^2
\end{aligned}
$$

Let the vector $\v y$ contain all target values from the learning set:

$$
\v y = \matvec{c}{y \ls 1\\ \vdots \\ y \ls m}
$$

and let $\v X$ be the matrix

$$
\hv X = \matvec{c}{\hv x \ls 1 \T \\ \vdots \\
\hv x \ls m \T }
$$

Note that the example vectors are now the *rows* in the matrix. Now the expression for the cost function can be written as:

.. math::
   J(\v \theta) = \frac{1}{2m} \| \hv X \v \theta - \v y \|^2

The gradient of the cost function using an explicit summation over the
learning examples was derived to be:

.. math::
   \frac{\partial J(\v\theta)}{\partial \v\theta} =
   \frac{1}{m}\sum_{i=1}^{m} \left(\v\theta\T \hv x\ls i - y\ls i \right)\hv x\ls i

In vectorized form we have:

.. math::
   \frac{\partial J(\v\theta)}{\partial \v\theta} = \frac{1}{m} \hv X\T (\hv X \v\theta - \v y)

Leading to the gradient descent algorithm:

.. math::
   \text{iterate until done:}\qquad \v\theta := \v\theta -
   \alpha \frac{1}{m} \hv X\T (\hv X \v\theta - \v y)

Then we extend linear regression to deal with more then one feature
value, this leads to multivariate linear regression.

Multivariate Linear Regression
==============================

From 1 to n Features

Now suppose we have not one feature value x but n feature values
$x_1,\ldots,x_n$. We collect these values in a **feature vector** $\v
x = (x_1\ldots x_n)\T$.

Now the target value y depends on all these n values and a simple
linear model is:

.. math::
   h_{\v\theta}(x_1, x_2, \ldots, x_n) = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n

Again we augment our feature vector to deal with the bias term θ_0:

.. math::
   \hv x = \matvec{c}{1\\ \v x} = \matvec{c}{1\\ x_1 \\ \vdots\\ x_n}

Convince yourself that from here it is downhill and completely as we
have seen in the univariate case. The cost function is:

.. math::
   J(\v \theta) = \frac{1}{2m} \| \hv X \v \theta - \v y \|^2

and the gradient vector is:

.. math::
   \frac{\partial J(\v\theta)}{\partial \v\theta} = \frac{1}{m} \hv X\T (\hv X \v\theta - \v y)

Note that the increased complexity with respect to the univariate case is completely 'hidden' in the dimensions of the vector $\v\theta$ and the matrix $\hv X$.

Extended Features

Evidently not all functions that we would like to learn using a regression technique are linear in the data values. A standard way is to use an extended feature set. Let's consider the univariate case again. Instead of only considering the value of x in a linear model we now add 'new' feature values. For instance we generate the augmented feature vector $\hv x$:

.. math::
   \hv x = \matvec{c}{1 \\ x \\ x^2 \\ x^3}

And again *computationally* not much changes: the cost function is the same and the gradient vector is the same (albeit with other dimensions of the data matrix $\hv X$ and parameter vector $\v\theta$.

Note that we are not restricted to the monomials as feature values. Consider a sinussoidal function $a + b\sin(2x + c)$ where a, b and c are parameters of our model (we assume that the frequency is known). The function as defined in this way is *not* suitable for linear regression (why not?). We therefore write:

.. math::
   h_{\v\theta}(x) = A + B \sin(2 x) + C \cos(2 x)

as an equivalent expression. In the following figures we show a data set that van be modelled with the above hypothesis but is corrupted by noise. Using linear reqression (we use the analytical solution in the code but could have equally well use the gradient descent procedure) we estimate the parameters A, B and C and plot the learned model and compare with it with the 'ground truth' (possible because we have generated the noisy data ourselves).

.. ipython:: python

   from matplotlib.pylab import *
   x = 10 * random_sample(50)
   xx = linspace(0,10,1000)
   x = sort(x)
   def generatey(x):
       return 5 + 3 * sin(2 * x + pi/3)

   yt = generatey(x)

```
y = yt + 0.9 * randn(*x.shape)
subplot(2,1,1);
plot(x, y, '.');
plot(xx, generatey(xx));

tmp = (ones_like(x), sin(2*x), cos(2*x))
X = vstack(tmp).T
p = inv(X.T @ X) @ X.T @ y
subplot(2,1,2);
plot(xx, generatey(xx), 'g');
def model(x, p):
    return p[0] + p[1] * sin(2*x) + p[2] * cos(2*x)

@savefig sinefit.png
plot(xx, model(xx,p) , 'b');
```

Observe that we only need 50 learning examples to get a fairly accurate estimation of the model parameters. This is only true in general in case the data is modelled accurately by the selected hypothesis function *and* the noise (deviations from the model) can be accurately described as additive normally distributed. In the last section of the Regression chapter we will derive the univariate regression algorithm from basic principles.

In case we have a lot of features the hypothesis is getting more complex and we run into the risk of **overfitting**. The the model learns to adopt to the noise and natural variation in the data leading to a small error on the learning examples but a large(r) error on examples not seen in the learning phase. To tackle this problem we will look at **regularization**.

Dealing with Overfitting using Regularization
==========

Consider the hypothesis function:

.. math::
   h_{\v\theta} = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots \theta_n x^n

We like to find the parameter vector $\v\theta = (\theta_0\;\theta_1\;\cdots\;\theta_n)\T$ such that $h_{\v\theta}$ fits the data points given below in the figure. Note that the 'true' function is depicted with a gray line.

.. ipython:: python
   :suppress:

```
np.random.seed(87234334)

def f(x):
    return 1/2 * (x - 5)**3 + 3*(x - 5) + 4
```

```
xx = np.linspace(0, 10, 1000)
yy = f(xx)
plt.plot(xx, yy, color='lightgray');

N = 12
x = np.random.uniform(low=0, high=10, size=N)
y = f(x) + 10*np.random.randn(N)
plt.plot(x, y, 'x');

plt.plot()

def augment_extended(x, n):
    X = np.empty((0,len(x)))
    for i in range(n):
        X = np.vstack((X,x**i))
    return X.T

order = 8
tX = augment_extended(x, order)
print(tX.shape)
theta = np.linalg.inv(tX.T @ tX) @ tX.T @ y
print(theta)
tXX = augment_extended(xx,order)
yy_pred = tXX @ theta

plt.plot(xx, yy_pred, label='standard fit')
plt.ylim(-80,80)

lmb = 100000
nu = np.eye(order); nu[0,0] = 0
theta_reg = np.linalg.inv(tX.T @ tX + lmb*nu) @ tX.T @ y
yy_pred_reg = tXX @ theta_reg

plt.plot(xx, yy_pred_reg, label='regularized fit')
plt.legend()
plt.title('Linear Regression with(out) Regularization\n' \
        r'$h_\theta(x)=\theta_0+\theta_1 x + \theta_2 x^2 + \cdots +\theta_7 x^7$')

@savefig linregr_regularization.png
plt.plot()
```

The standard linear regression function is plotted in orange. Because the model is too complicated for the data it will **overfit**. The model will do well on the learning set but will perform poorly on the test set.

Instead of selecting a less complex model (e.g. a lower order polynomial) we can use **regularization**. The regularization approach is based on the observation that the parameters of an overfitted model tend to get large. We then try to keep the parameters small by penalizing large parameters. This is done by adding a term to the cost function:

.. math::
   J(\v\theta) = \frac{1}{2m}\|\hv X\v\theta - \
   \v y\|^2 + \frac{\lambda}{2m} \sum_{i=1}^n \theta_i^2

where λ is the **regularization parameter** that determines the relative importance of the regularization cost with respect to the fitting cost. Note that the θ_0, the bias parameter, is not taken into account in calculating the regularization cost.

For $\lambda=0$ there is no regularization and we end up with the classic linear regression solution. For $\lambda\rightarrow\infty$ regularization dominates completely and all parameters, except θ_0 will be zero.

Introducing the matrix:

.. math::
   D_0 = \begin{pmatrix}
   0 & 0 & 0 & \cdots & 0\\
   0 & 1 & 0 & \cdots & 0\\
   0 & 0 & 1 & \cdots & 0\\
   \vdots & \vdots & \vdots & \ddots & \vdots\\
   0 & 0 & 0 & \cdots & 1
   \end{pmatrix}

we can write:

.. math::
   J(\v\theta) = \frac{1}{2m}\|\hv X\v\theta - \v y\|^2 + \
   \frac{\lambda}{2m} \| D_0 \v\theta \|^2

The gradient then becomes:

.. math::
   \pfrac{J(\v\theta)}{\v\theta} = \frac{1}{m}\hv X\T(\hv X\v\theta-\v y) + \
   \frac{\lambda}{m}D_0 \v\theta

With this new gradient vector finding the optimal parameter vector is done in the same way as before. Start at a random initial vector and do the gradient descent iteration.

In the figure above besides the standard (unregularized) fit (in orange) also the regularized fit (in green) is shown.

The regularization parameter λ is a so called **meta parameter**. It is not a real parameter of the model itself, but it is a parameter of the cost function. Selecting the best regularization parameter requires 'learning' too. The learning procedure then is:

#. Divide your set of examples $\{\v x\ls i, y\ls i\}$ into three
   parts: the learning set, the test set *and* the **cross validation
   set**.

#. For several values, say N, of the meta parameter λ learn
   the model on the learning set. This results in N possibly
   different models (parameter vectors $\v\theta$).

#. Select the best out of the N models using the results based on
   the cross validation set.

#. Test this model on the test set to find the accuracy of your model.

Solving a linear regression problem with a gradient descent procedure
might seem like too much work as an analytical solution is known.

Analytical Solution for Linear Regression
==========

For the general case of linear regression we have the cost function:

.. math::
   J(\v \theta) = \frac{1}{2m} \| \hv X \v \theta - \v y \|^2

and the gradient of the cost function:

.. math::
   \frac{\partial J(\v\theta)}{\partial \v\theta} = \frac{1}{m} \hv X\T (\hv X \v\theta - \v y)

We are looking for a minimum of the cost function. Remember from high
school that a nescessary condition for a minimum of a univariate
function is that the derivative is equal to zero. For multivariate
function we must have that all partical derivatives are equal to
zero. I.e. we have to set the gradient to zero and solve for
$\v\theta$.

.. math::
   \hv X\T (\hv X \v\theta - \v y) = 0

or equivalently:

.. math::
   \v\theta = (\hv X\T \hv X)\inv \hv X\T \v y

It can be shown that this parameter vector is indeed the point where
the cost function attains its global minimum. The cost function is a
convex function with a unique minimum. We will not give a formal
proof here. Note that a convex function does not imply that the
minimum can be calculated analytically. In another chapter we will
discuss logistic regression as an example of an optimization problem
that has a unique solution but still needs a numerical technique to
find that minimum.

For examples of linear regression in these notes we will use the
analytical solution. In practice with real life examples where the
number of features tend to be large the analytical solution is not
always the best choice. See the discussion in the Coursera Machine

Learning Course of Andrew Ng.

Throughout our analysis of linear regression we have simply stated that a quadratic cost function had to be used. But why is that? Is there a reason for it? In the last section of this chapter we will derive our cost function from basic principles.

Linear Regression from Basic Principles

A Statistical View on Regression

We consider a system with scalar input x and scalar output y. We hypothesize that in the absence of 'noise' we have $y = m(x)$. The exact analytical form of the function m is not known, all we have are examples $(x\ls i,y\ls i)$. Unfortunately we know that $y\ls i \not= m(x\ls i)$ due to noise. We assume that the noise is iid for all x and normal distributed. This makes our observation a stochastic variable:

.. math::
   Y = m(x) + R

where R is the noise random variable. We assume $R\sim\Normal(\mu,\sigma^2)$. Thus

.. math::
   Y \sim \Normal(m(x), \sigma^2)

The goal of linear regression is to come up with an expression for the function m. To make this mathematically tractable we approximate m with a parameterized hypothesis function $h_{\v\theta}$ arriving at:

.. math::
   Y \sim \Normal(h_{\v\theta}(x), \sigma^2)

With our learning set $(x\ls i,y\ls i)$ for $i=1,\ldots,m$ the regression task is to find the parameter vector $\v\theta$ that best 'fits' the data.

Maximum Likelihood Estimator

We are looking for the parameter vector $\v\theta$ that makes the observed data most plausible. This can be casted in a form that is called a **maximum likelihood estimator**. Let f be the joint probability density function for all observations $y\ls i$ given the $x\ls i$ values and parameter vector $\v\theta$. The MLE then can be written as:

.. math::
   \hat{\v\theta} = \arg\max_{\v\theta} f(y\ls 1, \ldots,y\ls m \bigm|

x\ls 1, \ldots, x\ls m; \v\theta)

As we have assumed that the noise in observations are iid and normally distributed we can write:

.. math::
f(y\ls 1, \ldots,y\ls m \bigm|
x\ls 1, \ldots, x\ls m; \v\theta) &=
\prod_{i=1}^{m} f(y\ls i\bigm| x\ls i; \v\theta)\\
&= \prod_{i=1}^{m} \frac{1}{\sigma\sqrt{2\pi}}
\exp\left(- \frac{(y\ls i - h_{\v\theta}(x\ls i))^2}{2\sigma^2}\right)

Our goal is to maximize the probability density and thus we may maximize the logarithm of it. That function of $\v\theta$ is called the **log likelihood**:

.. math::
\ell(\v\theta)
&= \log\left(\prod_{i=1}^{m} \frac{1}{\sigma\sqrt{2\pi}}
\exp\left(- \frac{(y\ls i - h_{\v\theta}(x\ls
i))^2}{2\sigma^2}\right) \right)\\
&= \sum_{i=1}^{m} \left(
-\log(\sigma\sqrt{2\pi}) - \frac{(y\ls i - h_{\v\theta}(x\ls
i))^2}{2\sigma^2}
\right)

Our goal it to maximize ℓ and that is equivalent with *minimizing* the sum of squared errors:

.. math::
\hat{\v\theta} = \arg\min_{\v\theta} \sum_{i=1}^{m} \left(
y\ls i - h_{\v\theta}(x\ls i)
\right)^2

This is exactly the same expression we started with when introducing linear regression. In this section we see that linear regression is optimal in case we have normally distributed noise. In practice that is not very often the case (especially in the machine learning context) but even then linear regression is often used (mostly with the addition of a regularization term).

Bayesian Classification

Let $\v X$ be the random (feature) vector $\v X=(X_1\cdots X_n)\T$ characterizing an object. The class of this object is characterized with the discrete random variable Y. The goal in classification is to come up with a classification function $\hat y = \classify(\v x)$ that assigns a class to a feature vector $\v x$.

To quantify the succes of the classifier we introduce the **loss function** L such that $L(\hat y, y)$ indicates the loss of an incorrect classification. The loss function is a mechanism to make a distinction in the errors a classifier can make. For a medical test it

is often considered less of a problem if a patient is incorrectly diagnosed with a disease than the opposite case where the patient is declared healthy where in reality she is not.

In many classification problems the **zero-one loss** is used, then we take $L(\hat y, y)=[\hat y\not=y]$. I.e. the loss function is equal to 1 in case the classifier is wrong and zero if it is right.

The squared error $L(\hat y,y) = (\hat y-y)^2$ is also used as a loss function (for instance in neural nets) but is more often associated with *regression*.

The expected loss given feature vector $\v X = \v x$ for a classifier $\hat y = \classify(\v x)$ equals:

.. math::
  \mathcal L(\hat y;\v x) = \E(L(\hat y,Y)\given \v X=\v x)
  = \sum_y L(\hat y, y)\, \P(Y=y\given \v X = \v x)

The **Bayesian classifier** then finds the class $\hat y$ with minimal expected loss:

.. math::
  \classify(\v x) = \arg\min_{\hat y} \mathcal L(\hat y; \v x)
  = \arg\min_{\hat y} \sum_y L(\hat y, y)\, \P(Y=y\given \v X = \v x)

We will look in somewhat more detail at the zero-one loss Bayesian classifier. First we show that the zero-one loss function leads to the **Maximum a Posteriori** classifier. Secondly we consider the **Naive Bayes Classifier**.

Maximum a Posteriori Classifier

The Bayes classifier was defined as:

.. math::
  \classify(\v x) = \arg\min_{\hat y} \sum_y L(\hat y, y)\, \P(Y=y\given \v X = \v x)

In case we set $L(\hat y, y)=[\hat y\not= y]$ we get:

.. math::
  \classify(\v x) &= \arg\min_{\hat y} \sum_y [\hat y\not= y] \, \P(Y=y\given \v X = \v x)\\
  &= \arg\min_{\hat y} \sum_{y\not= \hat y} \P(Y=y\given \v X = \v x)\\
  &= \arg\min_{\hat y} \left(1 - \P(Y=\hat y\given \v X = \v x) \right)\\
  &= \arg\max_{\hat y} \P(Y=\hat y\given \v X = \v x)

I.e. the Bayes classifier for zero-one loss is equal to the maximum a posteriori classsifier.

Using Bayes rule this can be rewritten for a discrete random vector
$\v X$ as:

.. math::
   \classify(\v x) &= \arg\max_{\hat y} \P(Y=\hat y\given \v X = \v x)\\
                   &= \arg\max_{\hat y} \frac{\P(\v X=\v x\given Y=\hat y)\,\P(Y=\hat y)}{\P(\v X=\v x)}

Because the evidence $\P(\v X=\v x)$ is not dependent on $\hat y$ and
is positive we have:

.. math::
   \classify(\v x) = \arg\max_{\hat y} \P(\v X=\v x\given Y=\hat y)\,\P(Y=\hat y)

For a continuous random vector $\v X$ we get:

.. math::
   \classify(\v x) &= \arg\max_{\hat y} \P(Y=\hat y\given \v X = \v x)\\
                   &= \arg\max_{\hat y} \frac{f_{\v X\given Y=\hat y}(\v x)\,\P(Y=\hat y)}{f_{\v X}(\v x)}

Again the evidence $f_{\v X}(\v x)$ is positive and independent of
$\hat y$ and thus:

.. math::
   \classify(\v x) = \arg\max_{\hat y} f_{\v X\given Y=\hat y}(\v x)\,\P(Y=\hat y)

==========
Naive Bayes Classifier
==========

Discrete Naive Bayes Classifier
==========

We start with the simple example of a Naive Bayesian spam
classifier. An (email) text message is characterized with a n
boolean random variables. Each variable W_i has value 1 in case the
i th word from a dictionary is present in the email. Here we take
the dictionary for granted. In a real spam filtering system setting up
the dictionary and mapping the text onto the words in the dictionary
if of course a very important aspect of such a system.

Let S be the random variable that characterizes a spam email: $S=1$
is spam and $S=0$ is a non spam email.

A Bayesian classifier classifies an email characterized with
$W_1=w_1,\dots,W_n=w_n$ as spam in case:

.. math::
   \P(S=1\given W_1=w_1,\dots,W_n=w_n) \geq \P(S=0\given W_1=w_1,\dots,W_n=w_n)

The joint probability function for this problem is

.. math::
   p_{W_1\cdots W_n\,S}(w_1,\ldots,w_n,s) = \P(W_1=w_1,\dots,W_n=w_n, S=s)

This joint probability function encoded as a table giving the value
for each possible combination of the variables is quite big. The table
will have 2^{1001} entries. Way too big to learn.

Using Bayes rule we first rewrite the classicification rule:

.. math::
   \frac{\P(W_1=w_1,\dots,W_n=w_n \given S=1) \P(S=1)}{\P(W_1=w_1,\dots,W_n=w_n)} \geq
   \frac{\P(W_1=w_1,\dots,W_n=w_n \given S=0) \P(S=0)}{\P(W_1=w_1,\dots,W_n=w_n)}

The a priori probability for the data $\P(W_1=w_1,\dots,W_n=w_n)$ is
called the **evidence** and can be left out on both sides of the
equation.

.. math::
   \P(W_1=w_1,\dots,W_n=w_n \given S=1) \P(S=1) \geq
   \P(W_1=w_1,\dots,W_n=w_n \given S=0) \P(S=0)

But still this does not lead to an improvement. We now have two tables
each of 2^{1000} entries.

The **Naive Bayesian Classifier** assumes that the W_i ’s are
(conditionally) independent, i.e.

.. math::
   \P(W_1=w_1,\dots,W_n=w_n \given S=s) = \prod_{i=1}^{n} \P(W_i=w_i \given S=s)

Using this naive assumption leads to the classifier:

.. math::
   \P(S=1)\,\prod_{i=1}^{n} \P(W_i=w_i \given S=1) \geq
   \P(S=0)\,\prod_{i=1}^{n} \P(W_i=w_i \given S=0)

Now we have $2n+2$ probabilities that need to be known to use such a
classifier. These probabilities can easily be learned from data.

Often the classifier is written as:

.. math::
   \frac{\prod_{i=1}^{n} \P(W_i=w_i \given S=1)}{
   \prod_{i=1}^{n} \P(W_i=w_i \given S=0)} \geq
   \frac{\P(S=0)}{\P(S=1)}

In the case of a spam filter it turns out that in practice
$\P(S=1)\gg\P(S=0)$. Instead of using the a priori probabilities for
spam and non-spam a threshold T is chosen.

.. math::
   \frac{\prod_{i=1}^{n} \P(W_i=w_i \given S=1)}{
   \prod_{i=1}^{n} \P(W_i=w_i \given S=0)} \geq T

In practice the value of T is learned from the data. In that

situation a value of T can be selected that not only considers the number of classification errors but that weigh the two types of error differently. In the case of spam filtering it is better to misclassify some span emails as non-spam then to classify non-spam emails as spam (and never read them...).

Continuous Naive Bayes Classifier

We now consider the case where we have n continuous random variables $X_1,\ldots,X_n$ and based upon their values have to classify the object into one of k classes $C=1$ to $C=k$.

The MAP classifier then is:

.. math::
   c^\star = \arg \max_c \P(C=c\given X_1=x_1, \ldots, X_n=x_n)

Using Bayes rule we can write:

.. math::
   \P(C=c\given X_1=x_1, \ldots, X_n=x_n) =
   \frac{f_{X_1\cdots X_n\given C=c}(x_1,\ldots,x_n) \, \P(C=c)}{f_{X_1\cdots X_n}(x_1,\ldots,x_n)}

Leading to:

.. math::
   c^\star &= \arg \max_c
   \frac{f_{X_1\cdots X_n\given C=c}(x_1,\ldots,x_n) \, \P(C=c)}{f_{X_1\cdots X_n}(x_1,\ldots,x_n)}\\
   &= \arg \max_c f_{X_1\cdots X_n\given C=c}(x_1,\ldots,x_n) \, \P(C=c)

The last equation is true because the evidence for the data is not dependent of the class. Note carefully that the value that is being maximized in this last expression is *not* a probability anymore.

The naive assumption again is that the feature values $X_1,\ldots,X_n$ are *class conditionally independent*, i.e.

.. math::
   f_{X_1\cdots X_n\given C=c}(x_1,\ldots,x_n) = \prod_{i=1}^{n} f_{X_i\given C=c}(x_i)

Using the Naive Bayes classifier for continuous random variables in practice is most often done in case we may assume that the X_i are distributed according to some parameterized pdf. Then we need to estimate the parameters for the distribution of each of the classes.

Nearest Neighbor Classification

In nearest neighbor classification we start with a learning set of feature vectors $\v x\ls i$ and associated class labels $y\ls i$. We

assume that each example in the learning set $(\v x\ls i,y\ls i)$ is the simultanuous realization of i.i.d. contiuous random variables $\v X_i$ and discrete random variables Y_i for $i=1,\ldots,m$.

Bayesian classification assigns a feature vector $\v x$ to class y such that

.. math::
   y = \arg\max_{y'} \P(Y=y'\given \v X=x)

We have already shown that this can be rewritten as:

.. math::
   y = \arg\max_{y'} f_{\v X\given Y=y'}(\v x) \P(Y=y')

Generative classifiers try to estimate the class conditional data distributions functions from the learning set.

A nearest neighbor classifier is based on the same principle but *without* explicit learning of probability distributions. Instead the entire learning set is memorized and to classify an unknown $\v x$ we simply assign it to the class of the closest example in the learning set. Evidently in case the class conditional probability is high we are bound to find an example in the learning set close to the unknown $\v x$ that is close to one of the learning examples.

We will discuss not only the simple nearest neighbor classifier but also the k-nearest neighbor classifier where the resulting class is the taken to be the most often occuring class in the k nearest neighbors. Then we generalize the classifier to a weighted classifier using in principle all examples in the learning set.

Simple Nearest Neighbor Classifier

The classifier is defined as:

.. math::
   \text{classify}(\v x) = y\ls i,\quad\text{where}\quad i = \arg\min_j \|\v x - \v x\ls j\|

i.e. we first look for the example $\v x\ls i$ in the learning set that is closest to the given $\v x$ and then assign its label $y\ls i$ to the unknown feature vector $\v x$.

This very simple classifier is:

* only **accurate** in case a lot of learning data is available, essentially we need enough data to be able to estimate the class conditional probability density functions.

* only **efficient** in case the learning set is not too large and the dimensionality of the feature space is not too large either. The first demand is of course contradictory to the requirement for an accurate classifier.

* very prone to **overfitting** in the case of 'outliers' of a class
  in the feature space region that is more probable for another class.

In the next section we define the k-nearest neighbor classifier as a way to deal with the overfitting problem.

k-Nearest Neighbor Classifier

For the k-nearest neighbor classifier we don't only consider the closest element in the learning set. Instead we consider the k closest elements in the learning set. For a two class problem we then classify an unknown feature vector for the class that has the majority in the k closest elements from the learning set.

The algorithm is simple:

1. First calculate the distance from the feature vector $\v x$ to all vectors in the learning set.

   .. math::
      d_i = \|\v x - \v x\ls i\|

2. Sort all distances and set the *sorted index map* I_s such that

   .. math::
      \forall i\leq j: d_{I_s(i)} \leq d_{I_s(j)}

   Note: I_s is what the 'argsort' function in Numpy returns.

3. For $i=1,\ldots,k$ count which class $y\ls{I_s(i)}$ is in the majority. That class will be the result of the classifier.

The k-NNb classifier can be interpreted as a voting algorithm: the closest k vectors $\v x\ls i$ in the learning set vote (with equal voting strength) for their class $y\ls i$. In the next section we consider a generic voting scheme where the voting strength is a function of the distance of the vector $\v x$ to each of the vectors in the learning set.

Weighted Nearest Neighbor Classification

In the k nearest neighbor classifier each of the k nearest feature vectors in the learning set cast a vote for their class. And all votes are of the same strength.

We may soften (regularize) the hard voting by assigning soft votes. Most often the vote strength is taken to be dependent on the distance d_i of feature vector $\v x$ to the i-th learning example $\v x\ls i$. Let the function w be the voting strenght function then

weighted NNb classification works as follows.

1. First calculate the distance from the feature vector $\v x$ to all
   vectors in the learning set.

   .. math::
      d_i = \|\v x - \v x\ls i\|

2. Then perform the voting procedure resulting in a function V that
   maps a class label to the total voting strength for that class.

   .. math::
      V(y) = \sum_{i=1}^{m} w(d_i) \left[y\ls i = y \right]

   where $\left[\text{cond} \right]$ is the Iverson bracket that
   equals one in case cond is true and zero otherwise.

   Note that the expression for V is not the right way to implement
   the voting procedure. Be sure to understand the algorithm that is
   of order m and not dependent on the number of classes.

3. The unknown $\v x$ is then classified as:

   .. math::
      \text{classify}(\v x) = \arg\max_y V(y)

Logical choices for the kernel function w are such that:

- $w(d)>0$ for all d,
- $w(d)$ attains its maximum for $d=0$, and
- $w(d)$ is not increasing for increasing d.

==========
 Logistic Regression
==========

Logistic regression, despite its name, it a classification technique. We have seen that the Bayes classifier assigns the class $\hat y = \classify(\v c)$ to an object characterized with feature vector $\v x$ based on:

.. math::
   \classify(\v x) = \arg\max_y \P(Y=y\given \v X=\v x)

For the Bayesian classifier the a posteriori probability :math:‘\P(Y=y\given \v X = \v x)‘ is then expressed in the class conditional probabilities of the data and the a priori probabilities of the classes. Effectively the joint distribution of the feature vector and class is estimated. A Bayesian classifier is therefore a **generative classifier** (given the joint distribution you could generate examples).

The logistic regression classifier is an example of a **discriminative

classifier** and it starts with the same a posteriori probability as given above. But unlike the Bayes classifier it does not calculate this a posteriori probability from (an estimate of) the joint distribution but it estimates the a posteriori probability directly from the training set using a 'simple' parameterized model. As such the logistic regression classifier concentrates more on the **decision boundaries** of the different classes and not on an estimator for the probability.

Logistic Regression Model
=========================

We start with a simple two class problem $Y=0$ or $Y=1$. The logistic regression classifier assumes that a hyperplane separates the two classes in feature space. In two dimensional feature space the hyperplane is a straight line, feature vectors on one side of the plane belong to the $Y=0$ class, vectors on the other side belong to the $Y=1$ class.

The a posteriori probability $\P(Y=1\given \v X=\v x)$ is modelled with the hypothesis function $h_{\v\theta}(\v x)$

.. math::
   h_{\v\theta}(\v x) = \P(Y=1\given \v X=\v x) = g(\theta\T \tilde{\v x})

where $\tilde{\v x} = (1\; \v x)\T$ is the feature vector $\v x$ is augmented with a bias term. Note that the parameter vector $\v\theta$ thus is a $n+1$ dimensional vector.

The function g is the **sigmoid** or **logistic** function and is defined as:

.. math::
   g(v) = \frac{1}{1+e^{-v}}

Plotting this function show that $g(v)$ is in the range from 0 to 1 (what is to be expected for a probability of course).

.. ipython:: python
   :suppress:
   :okwarning:

   def g(v):
       return 1/(1+np.exp(-v))

   v = np.linspace(-10,10)
   plt.plot(v, g(v))
   @savefig sigmoidfunction.png width=10cm align=center
   plt.show()

Note that for a 2D (nD) feature space:

* $\v\theta\T\tilde{\v x} = 0$ defines a line (hyperplane) in feature space.

* This line (hyperplane) is called the **decision boundary**, points $\v x$ where $h_{\v\theta}(\v x)>0.5$ are classified as $y=1$, whereas points where $h_{\v\theta}(\v x)<0.5$ are classified as $y=0$.

* On this line (hyperplane) we have $h_{\v\theta}(\v x)=0.5$.

* On lines (hyperplanes) parallel to the above line (hyperplane) the hypothesis value is constant.

Training a logistic regression classifier thus amounts to learning the optimal parameter vector $\v\theta$ given a set of training examples $(\v x\ls i, y\ls i)$ for $i=1,\ldots,m$. In the next section we will describe this learning process as a maximum likelihood estimator for $\v\theta$.

==========
Maximum Likelihood Estimator
==========

We have defined in a previous section that:

.. math::
   P(Y=1\given \v X=\v x) = h_{\v\theta}(\v x)

then of course for a 2 class problem:

.. math::
   P(Y=0\given \v X=\v x) = 1 - h_{\v\theta}(\v x)

We thus have that :math:‘Y\given\v X=\v x \sim \Bernoulli(h_{\v\theta}(\v x))‘. The probability function then is:

.. math::
   \P(Y=y\given\v X=\v x) = \begin{cases}
                              h_{\v\theta}(\v x) &: y=1\\
    1 - h_{\v\theta}(\v x) &: y=0
      \end{cases}

or equivalently:

.. math::
   \P(Y=y\given\v X=\v x) = \left(h_{\v\theta}(\v x)\right)^y \,
                            \left(1-h_{\v\theta}(\v x)\right)^{1-y}

Note that in the above expression we have used the fact that y is either zero or one.

The training set $(\v x\ls i, y\ls i)$ for $i=1,\ldots,m$ can be considered to be the realization of m i.i.d. random vectors and variables $(\v X\ls i, Y\ls i)$. Probability for the entire training set then is

.. math::
  \P(Y\ls 1=y\ls 1,\ldots,Y\ls m=y\ls m \given \v X\ls 1 = \v x\ls 1,\ldots, \v X\ls m = \v x\ls m) = \\
   \prod_{i=1}^{m} \P(Y\ls 1=y\ls i\given \v X\ls i = \v x\ls i)\\
   = \prod_{i=1}^{m} \left(h_{\v\theta}(\v x\ls i)\right)^{y\ls i} \, \left(1-h_{\v\theta}(\v x\ls i)\right)^{1-y\ls i}

The above expression can also be interpreted as the **likelihood** of the
data (the training set) given the parameter vector $\v\theta$:

.. math::
  \ell(\v\theta) = \prod_{i=1}^{m} \left(h_{\v\theta}(\v x\ls i)\right)^{y\ls i}
                \, \left(1-h_{\v\theta}(\v x\ls i)\right)^{1-y\ls i}

The maximum likelihood estimator for theta is then given by:

.. math::
  \hat{\v\theta} = \arg\max_{\v\theta} \log(\ell(\v\theta))

Finding the optimal $\v\theta$ has to be done using a numerical
technique, unlike the case for linear regression there is no
analytical solution for logistic regression. We thus have to calculate
the gradient of $-\log\ell(\v\theta)$ and then iterate the gradient
descent steps as we have done for the linear regression.

You may skip the next subsection and take the gradient derivation for
granted.

Gradient Descent
================

Logistic regression finds the optimal $\v\theta$ given by:

.. math::
  \hat{\v\theta} = \arg\max_{\v\theta} \log(\ell(\v\theta))

where

.. math::
  \ell(\v\theta) = \prod_{i=1}^{m} \left(h_{\v\theta}(\v x\ls i)\right)^{y\ls i}
                \, \left(1-h_{\v\theta}(\v x\ls i)\right)^{1-y\ls i}

The maximum likelihood problem can be casted into the minimization of
a cost function by taking the negative likelihood and taking the
average over m examples:

.. math::
  J(\v\theta) &= - \frac{1}{m} \log(\ell(\v\theta))\\
           &= \frac{1}{m}\sum_{i=1}^{m} \left(- y\ls i\,\log h_{\v\theta}(\v x\ls i) -
        (1 - y\ls i)\log(1-h_{\v\theta}(\v x\ls i)) \right)

The gradient is:

.. math::
  \frac{\partial J(\v\theta)}{\partial \v\theta} =
  \frac{1}{m} \sum_{i=1}^{m} \left(- y\ls i\,\frac{\partial}{\partial \v\theta}\log h_{\v\theta}(\v x\ls i) -
        (1 - y\ls i)\frac{\partial}{\partial \v\theta}\log(1-h_{\v\theta}(\v x\ls i)) \right)

For the first partial derivative in the above expression we have:

.. math::
  \frac{\partial}{\partial \v\theta}\log h_{\v\theta}(\v x\ls i)
  &= \frac{\partial}{\partial \v\theta} \log g(\v\theta\T \tilde{\v x}\ls i)\\
  &= \frac{1}{g(\v\theta\T \tilde{\v x}\ls i)}\,g'(\v\theta\T \tilde{\v x}\ls i)\, \tilde{\v x}\ls i

For the logistic function we have $g'(v) = g(v)\,(1-g(v))$ and thus:

.. math::
  \frac{\partial}{\partial \v\theta}\log h_{\v\theta}(\v x\ls i)
  &= (1 - g(\v\theta\T \tilde{\v x}\ls i))\,\tilde{\v x}\ls i\\

For the second partial derivative we get:

.. math::
  \frac{\partial}{\partial \v\theta}\log(1-h_{\v\theta}(\v x\ls i))
  &= \frac{\partial}{\partial \v\theta}\log(1-g(\v\theta\T \tilde{\v x}\ls i))\\
  &= \frac{-1}{1-g(\v\theta\T \tilde{\v x}\ls i)}\, g'(\v\theta\T \tilde{\v x}\ls i)\, \tilde{\v x}\ls i \\
  &= - g(\v\theta\T \tilde{\v x}\ls i)\, \tilde{\v x}\ls i

This leads to the following expression for the gradient:

.. math::
  \frac{\partial J(\v\theta)}{\partial \v\theta}
  &= \frac{1}{m} \sum_{i=1}^{m} -y\ls i (1 - g(\v\theta\T \tilde{\v x}\ls i))\,\tilde{\v x}\ls i +
    (1-y\ls i)g(\v\theta\T \tilde{\v x}\ls i)\, \tilde{\v x}\ls i\\
  &= \frac{1}{m} \sum_{i=1}^{m} \left(g(\v\theta\T \tilde{\v x}\ls i) - y\ls i \right) \tilde{\v x}\ls i\\
  &= \frac{1}{m} \sum_{i=1}^{m} \left(h_{\v\theta}(\v x\ls i) - y\ls i \right) \tilde{\v x}\ls i

This last expression is equal to the expression for the gradient for linear regression. Carefully note that the hypothesis function is different in these cases.

Vectorized Logistic Regression Algorithm

Like we did for linear regression we can vectorize the expressions for logistic regression and 'hide' the summation over the examples in the learning set in matrix vector multiplications. Without proof we give the vectorized cost function:

.. math::
  J(\v\theta) = \frac{1}{m} \left(-\v y \T \log(g(\hv
  X\v\theta)) - (\v 1 - \v y)\T \log(\v 1 - g(\hv X \v\theta))
  \right)

where g is the logistic function and $\v 1$ is an $m\times1$ vector of ones. The gradient vector is:

.. math::
   \frac{\partial J(\v\theta)}{\partial\v\theta} =
   \frac{1}{m} \hv X\T (g(\hv X\v\theta) - \v y)

This leads to the following algorithm:

.. math::
   \text{iterate until done:}\qquad \v\theta := \v\theta -
   \alpha \frac{1}{m} \hv X\T (g(\hv X \v\theta) - \v y)

Multi Class Logistic Regression

Thusfar we have used logistic regression only for two class problems. Then an hypothesis that returns just one value modelling the a posteriori probability of assigning class $y=1$ to a feature vector $\v x$ suffices (simply because the probability for class $y=0$ is the complementary probability).

For a multi class problem things are different. Let the target value y be a value from 1 to C (for a C-class problem). Then we could think of the multi class classification problem as a regression problem. This is most often *not* a good idea.

Most regression algorithms take the output $\hat y = h_{\v\theta}(\v x)$ to be an orderable value and most often a value defined on an interval scale. This implies that for the loss function the error between $y=1$ and $\hat y=2$ will be less than the error between $y=1$ and $\hat y=4$. The scale of y is definitely not orderable in most situations let alone that differences are usefull entities.

We will distinguish three ways to ’generalize’ the two class approach to C classes. The first two ways do circumvent the problem described above, they both implement multi class classification by using several two class classifications. The third way is a real generalization as it learns a vectorial hypothesis function (with C elements) such that these values represent a probability distribution over the classes.

One vs All Multi Class

For a C class problem we train C two class classifiers: For each class c we train a classifier to distinguish class c from all other classes. To make a prediction we calculate all C hypotheses and make the choice for the class with the largest value.

One vs One Multi Class

Now we train a lot more classifiers. One for every pair of classes c and c' (with $c\not=c'$). For each classifier we then predict the class and for of these predictions we select the class with most 'votes'.

Softmax Multi Class

Softmax is nowadays the standard way to implement a multiclass classification system. Also for multi class logistic regression. Consider C linear units, each with input $\v x$, i.e.

.. math::
   z_i = \v\theta_i\T \v x

The individual sigmoid units for each of the linear combinations z_i are replaced with a **softmax layer**:

.. math::
   \hat y_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}

Note that by definition the $\hat y_i$ add up to one, so it can be interpreted as the a posteriory probability function over the class labels.

Also observe that for this scheme we need one-hot encoding of the target value. For instance if $y=1$ the target vector should be $(1\,0\cdots\,0)\T$. For $y=3$ the target vector should be $(0\,0\,1\,0\cdots\,0)\T$.

The loss function ℓ for one training sample $\v x$ to be used in these cases is the :doc:'cross entropy </LectureNotes/Math/InformationTheory>':

.. math::
   \ell(\v x) =
   - \sum_{i=1}^{C} \left(y_i \log \hat y_i + (1-y_i)\log(1-\hat y_i)\right)

where y_i are the elements of the one-hot encoding of the target class and $\hat y_i$ is the i-th element of the C linear combinations $\v\theta_i\T \v x$.

Neural Networks
==========

From a technical point of view a neural network is a collection of interconnected logistic regression units. One logistic regression unit takes in n input values, calculates a linear combination of those values using the 'weights' (parameters), adds a bias value and passes the resulting sum through a non-linear activation function. The

resulting value (in case classification is the goal) approximates the a posteriori probability of the class $y=1$ given the n feature values. Symbolically this is depicted in :numref:‘fig-nn_logistic‘.

The input values $x_1, \ldots,x_4$ are each multiplied with associated weight (parameter) θ_i, summed (and a bias weight θ_0 is added) and then fed through the activation function. The result is the value $\hat y$ associated with that one output node in the graph.

To build a neural network we start by extendig this was logistic regression unit to m units where each of those units is fed with the same n input values. If we have m of those logistic regression units working in parallel on the same input values we end up with something like is depicted in :numref:‘fig-nn_m_logistic‘.

From here on we can go on adding layers of nodes going from left to right. Adding two more layers to our example network ending in just one output node we end up with a network as shown in :numref:‘fig-nn_network‘.

In the context of neural networks one logistic regression unit is called a **neuron**. Read the next section to learn that neural networks were indeed inspired by comparison with the brain where neurons are the basic information processing building blocks.

In a neural network we always have an input layer and an output layes. The number of nodes (values) in the input layer equals the number of feature values. The number of nodes in the output layer depends on the use of the neural network. For boolean classification we most often will have one node, whereas for a k classes classification task we often have k output nodes, one for each class.

The layers inbetween the input and output layers are called the **hidden layers**. Again note that the bias values can be used for all (logistic regression) nodes but are not shown in the graph.

As may be clear from this last network graph, the complexity becomes large and in a detailed description we need to keep track of all the nodes in all layers and all the weights involved. We will do in both the more classical way that closely resembles the graph in :numref:‘fig-nn_network‘ and in a more modern formalism.

Learning the weights in a network with several layers needs a technique called back propagation. Remember that learning the parameters of the simple logistic regression (in :numref:‘fig-nn_logistic‘) requires comparing the error (loss) between the desired output (the target value) and the value that is calculated by the hypothesis function. For the hidden layers in a neural network there is no target value available of course. Back propagation learning is really nothing else than using the chain rule of differentiation in a systematic way.

Important questions to be answered when using a neural network in practice are:

- How many layers do i need for optimal performace and

- how many nodes in each layer?

The first question has a surprising answer: only three layers are enough (i.e. only one hidden layer). Only one hidden layer can be proven to result in an hypothesis function that can approximate any functional relation between input values and output value.

There is a big catch there. The number of nodes needed in the hidden layer might become unpractically high. And furthermore such an ’optimal’ network proves to be hard to learn using back propagation, the risk of overfitting is high.

Modern deep learning research has shown that very deep networkds (i.e. with a lot of layers in the order of hundreds) are indeed slow learners but with the best possible results.

To this day deciding on the number of layers and the number of nodes in a layer is more of an art than a science. Experimentation is key in this area.

To get some intuitive understanding of what a neural network is doing i urge you to play with a neural network at the ‘TensorFlow playground <https://playground.tensorflow.org/>‘_ (see :numref:‘fig-nnplayground‘ for a screenshot), an online demonstration that works on a two dimensional input vector $(x_1\;x_2)\T$ (and possibly some augmented features) with variable number of hidden layers and number of nodes in the hidden layers.

==========
Neural Networks: the Third Revolution
==========
Yes indeed we are in the middle of the third neural network revolution. The first one was in the forties when McCullogh and Pitts came with a simplified model of a human neuron. Their analysis was based on classical logic but unfortunately there was no learning algorithm known for the model.

In the mid fifties Rosenblatt proposed the perceptron. Each neuron was essentially a logistic regression classifier albeit it with a non differentiable activation function.

Rosenblatt came up with a learning rule that is close to the learning rule of the logistic classifier although their rule was only guaranteed to converge to a solution in case the learning set was indeed linear separable. Rosenblatt even made a hardware implementation of the perceptron that was able to learn to recognize visual patterns (of size 20×20).

In 1960 Widrow and Hoff proposed the ADALINE (adaptive linear

neuron). In the running phase it performed just like the perceptron. But in the learning phase the non linear activation function was not used, only the linear summation part. This lead to linear regression model and learning rule. The ADALINE learning rule is known as LMS (least mean squares) rule, or as the Widrow-Hoff rule or simple as the delta rule. A three layer network based on ADALINE was called MADALINE and there were in fact learning rules for these networks (including one that was proven later to be the backpropagation rule, for that network the activation function was indeed the sigmoid function).

Then at the end of the sixties, Minsky and Papert wrote a famous book: "Perceptrons". They criticized perceptrons by investigating the limitations of networks. They showed that a one neuron perceptron could not solve the XOR problem. They knew it could be done with an extra layer but at the time there was no learning algorithm to derive the parameters by learning. This book started the "winter of neural networks", very few scientists dare upon touching the subject as these two famous scientists decided that perceptrons were fundamentally flawed.

Minsky's and Papert's research went much further that just the XOR problem. They showed that not even a (finite) multi layer network of perceptrons can decide upon the connectivity of sets (as inputs they considered binary images just like the classical perceptron). This is what is illustrated on the cover of their book. You might think that both red drawings are one line. But in fact one drawing shows one red line and the other shows two red lines. The human visual brain is topoagnostic as well (at least i don't know anyone that is capable of seeing the difference between the two drawings at once).

Then in the eighties things got better for the neural network afficionado. The backpropagation algorithm was discovered. And as with many important ideas this one has many fathers as well. Dreyfuss, Werbos and Rumelhart are often cited to be one of the fathers of backpropagation.

Backpropagation essentially is the way to learn all the parameters in a feed forward network of simple perceptron neurons. With the important change that the activation function was chosed to be the sigmoid function: a differentiable function. Backpropagation through a modern looking glass is nothing else then a clever way of using the chain rule of differentation.

Backpropagation marked the start of the second revolution of neural networks. But interest decreased over time. Neural networks did not significantly outperform methods that we more firmly grounded on statistical theory. It was (and is) hard to really explain what a neural network is doing; in what way it comes to its decisions. Support vector machines became the machine learning tool of choice in the ninetees and beginning of the 21st century.

Neural networks were still being used though and most notably (especcially in hindsight) Le Cun et al basically reinvented the neocognitron of Fukushima (1980) but added the backpropagation

learning in LeNet (1989) the first deep learning network based on convolutions for letter and digit recognition. This architecture still is much like modern days deep learning network architectures.

With the increase of computer power, availabilty of advanced tools (like automatic differentiation) and development of powerful extentions of the classical gradient descent optimization, deep learning architectures with tens and even hundreds of (convolutional) layers became popular and the results are spectacular.

Yes we are indeed witnessing the third revolution of neural networks.

The Classical View on Neural Networks

In the classical view on neural networks graphs are used as well as in the modern view. In the classical graphs each node represents both a function as well as the value that it the outcome of that function., the edges in the graph indicate both the flow of values as well as the 'connection strengths' (the weight values).

A two layer network with one output node is the same as a logistic regression unit. The input layer contains n values (in the input nodes the values are not processed). In the node in the second layer we first calculate a weighted sum of the input values a_i and add a bias term to it

.. math::
   z = \theta_0 + \theta_1 a_1 + \cdots + \theta_n a_n = \v\theta\T\hv a

where $\v a$ is the vector $(a_1\cdots a_n)\T$ and $\hv a$ is the augmented vector $(1 \; a_1\cdots a_n)\T$. The extra element equal to 1 is added to 'pick up' the bias term θ_0. The final value is

.. math::
   a_{\text{out}} = g(z)

where g is the **activation function.**. For now think of the logistic function as it is also used in logistic regression.

.. sidebar:: **Elementwise function application**

   The notation $f(\v x)$ is read as applying a function that maps a vector $\v x$ and returns a scalar. Essentially $f(\v x)$ is short hand notation for

   .. math::
      f(\v x) = f(x_1,\ldots,x_n)

   In machine learning we often
   find the need to apply a scalar function to each of the elements in a vector (just like Numpy applies a function to all elements in an array). For this special case we will write $f\aew(\v x)$:

.. math::
f\aew(\v x) = \begin{pmatrix}
f(x_1)\\ \vdots\\ f(x_n)\end{pmatrix}

To get an entire layer in a nearal network we feed our input vector $\hv a$ not in one logistic regression node but in m nodes. Each of those nodes is characterized with its own parameter vector $\v\theta_j$ for $j=1,\ldots,m$. Combining these into a weight matrix Θ

.. math::
\Theta = \begin{pmatrix}
\v\theta_1\T \\ \v\theta_2\T \\ \vdots\\ \v \theta_m\T
\end{pmatrix}

The m outputs then are combined into one vector ${\v a}_\text{out}$. Given an input ${\hv a}_\text{in}$ we have:

.. math::
\v z_\text{out} &= \Theta \hv a_\text{in}\\
\v a_\text{out} &= g\aew(\v z_\text{out})

Carefully note that whereas the input is augmented (with the 1 element) the output is *not* augmented.

Now we can build a complete network with these layers of 'logistic regression units'.

Note that in the classical view this is called a 4 layer network as the input nodes are called a layer as well (although nothing is calculated in that 'layer'). A L-layer network is thus characterized with $L-1$ weight matrices.

The values in the nodes in one layer are called the *activation* of that layer and we will write the activation in layer l as $\v a\ls l$. We start numbering at 1 so $\v a\ls 1$ is the input $\v x$ to our network. Going from layer 1 to layer 2 we have:

.. math::
\v z\ls 2 &= \Theta\ls 1 \hv a\ls 1 \\
\v a\ls 2 &= g\aew(\v z\ls 2)

In general going from layer l to layer $l+1$ we have:

.. math::
\v z\ls {l+1} &= \Theta\ls l \hv a\ls l \\
\v a\ls {l+1} &= g\aew(\v z\ls {l+1})

The final output of the network is $\v a\ls L$. Formally we write

.. math::
   h_\v\Theta(\v x) = \v a\ls L

where the $\v\Theta$ is used to indicate all weight matrices in the network, $\v\Theta =\{\Theta\ls 1,\cdots\Theta\ls{L-1}\}$. Note that the above expression is sloppy math, the $\v x$ in the left hand side is nowhere to be found on the right hand side. But evidently $\v a\ls L$ depends ultimately on $\v x = \v a\ls 1$.

Finding all the parameters in a neural network boils down to finding all weight matrices in the network. Assuming a mean squared error loss over a learning set of tuples $(\v x\ls i, \v y\ls i)$ for $i=1,\ldots,m$, we have the following loss (cost) to be minimized:

.. math::
   J(\v\Theta) = \frac{1}{m}
   \sum_{i=1}^{m} \| h_\v\Theta(\v x\ls i) - \v y\ls i \|^2

We have left out a possible regularization term. Also note that the cross entropy error measure as used in logistic regression can be used here as well,

The minimization of the cost (or loss) function is the same as for all other minimization procedures that we have encountered before: gradient descent:

.. math::
   \forall l: \quad \Theta\ls l := \Theta\ls l -
   \alpha \frac{\partial J(\v\Theta)}{\partial\Theta\ls l}

In these lecture notes you will not find the expressions for the cost gradients (in this form). Instead we will first rewrite the neural network into the computational graph formalism and do the backpropagation in that formalism. The reason that we go that route is twofold. First the derivations are easier (IMHO). Secondly and more importantly the ’language’ used is the one from the modern view on neural networks paving the way for a somewhat easier understanding of deep learning methodologies.

==========
 A Modern View On Neural Networks
==========

In the more modern view on neural networks we will use graphs where the data is represented as the arrows in the graph and the nodes indicate the processing blocks.

The flow from input vector $\v x$ to output vector y is represented in :numref:‘fig-nnblock‘. The parameters that determine the processing are indicated with a vector $\v p$ (later on we will use parameter matrices as well). Symbolically we can write:

.. math::

\v y = f(\v x; \v p)

When we cascade these blocks to form a network of fully connected processing nodes we get something like (assuming 3 processing nodes) depicted in :numref:‘fig-nncascade‘.

The input $\v x$ - output $\hat{\v y}$ relation for this network is:

.. math::
:label: eq-cascade3

\hat{\v y} = f\ls 3(f\ls 2(f\ls 1(\v x; \v p\ls 1); \v p\ls 2); \v p\ls 3)

where we use $\hat{\v y}$ as the output of the network to indicate that it is an estimation of some desired target value $\v y$.

When learning all the parameters in the network (in our case these are $\v p\ls1$, $\v p\ls2$ and $\v p\ls3$) in a supervised learning setting we calculate the loss ℓ (cost) in comparing the output $\hat{\v y}$ given input $\v x$ with the known target value $\v y$.

Extending the cascade (graph) with the loss calculation we have:

We sketch the graph for the loss calculation for one learning example $(\v x, \v y)$. For all values in the learning set (of for a subset of the learing set: a **batch**) the total loss is simply the sum of the individual losses.

Given the loss function, for instance the quadratic loss function:

.. math::
:label: eq-loss

\ell(\hat{\v y}, \v y) = \tfrac{1}{2}(\hat{\v y} - \v y)^2

The learning procedure tries to minimize the loss function using a gradient descent procedure. I.e. for all $i=1,2,3$ it iterates steps:

.. math::
\v p\ls i := \v p\ls i - \alpha \pfrac{\ell}{\v p\ls i}

Note that ℓ is dependent on $\hat{\v y}$ and thus dependent on all $\v p\ls i$. In case we calculate the loss for the entire learning set or for a batch of examples the gradient is simply the sum of the gradients for the individual examples. Therefore we look at one example only in this introduction. In subsequent sections we will use larger batches of examples.

Also note that for differentiable functions $f\ls i$ equations :eq:‘eq-cascade3‘ and :eq:‘eq-loss‘ show that the gradient can be calculated analytically and will require as many uses of the chain rule of differentiations as there are blocks in the cascade.

To guide us in our calculation of the gradients of the loss function with respect to the parameters in our network we will do it block by

block starting at the output and working our way towards the input.

In :numref:‘fig-nnforwardbackward‘ the neural network with 4 layers (i.e. 3 processing blocks) and the loss block is sketched again but this time with backward arrows we also indicate the flow to calculate the derivatives of the loss function with respect to intermediate values in the network (including the parameters). For each block given the derivative of the loss with respect to its output we can calculate the derivative of the loss with respect to its inputs (including the parameters). That derivative requires only knowledge of that particular block. It is the chain rule that will tie all these derivatives together.

In a fully connected neural network each of the processing blocks $f\ls i$ in the graphs above can be decomposed in steps as depicted in :numref:‘fig-nnoneblock‘.

Note that the parameter vector $\v p\ls i$ is now split into the parameter matrix $W\ls i$ and bias vector $\v b\ls i$. In subsequent sections we will look at these blocks individually and then finally do it for an entire network (i.e. chain of processing blocks).

In order to appreciate the rest of this section (and chapter) you better refresh your understanding of the chain rule reading the math chapter on the :doc:‘chain rule in differentiation</LectureNotes/Math/chain_rule>‘.

Linear Block

The computational graph of a fully connected linear module is depicted in the figure below.

The input is an $s_\in$-dimensional vector $\v x$ and the output is a vector $\v u$ that is $s_\out$ dimensional. We have:

.. math::
  \underbrace{\v u}_{(s_\out\times 1)}
  = \underbrace{W}_{(s_\out\times s_\in)} \underbrace{\v x}_{(s_\in\times1)}

where W is an $s_\out\times s_\in$ matrix. Assuming $\pfrac{\ell}{\v y}$ is known we can calculate $\pfrac{\ell}{\v x}$:

.. math::
  \frac{\prt\ell}{\prt\v x} &= \frac{\prt \v u}{\prt\v
  x}\frac{\prt \ell}{\prt\v u}\\
  &= W\T \frac{\prt\ell}{\prt\v u}

The proof of this result is rather simple. We could either dive into **matrix calculus** (see :doc:‘/LectureNotes/Math/vectorderivatives‘) or we can give a straightforward proof by looking at the components of the vectors keeping in mind the chain rule for multivariate functions

(see :doc:‘/LectureNotes/Math/multivariate‘). We will follow the
second route.

.. math::
   \pfrac{\ell}{x_i} &= \sum_{j=1}^{s_\out} \pfrac{u_j}{x_i}\pfrac{\ell}{u_j}\\
   &= \sum_{j=1}^{s_\out} W_{ji}\pfrac{\ell}{u_j}\\
   &= \sum_{j=1}^{s_\out} (W\T)_{ij} \pfrac{\ell}{u_j}

and thus

.. math::
   \pfrac{\ell}{\v x} = W\T \pfrac{\ell}{\v u}

Next we need to know the derivative $\prt\ell/\prt W$ in order to
update the weights in a gradient descent procedure. Again we start
with an elementwise analysis:

.. math::
   \pfrac{\ell}{ W_{ij}}
   = \sum_{k=1}^{s_\out} \pfrac{u_k}{W_{ij}} \pfrac{\ell}{u_k}

where

.. math::
   \pfrac{u_k}{W_{ij}} = \pfrac{}{W_{ij}}
   \sum_{l=1}^{s_in} W_{kl} x_l = \begin{cases}x_j &: i=k\\0 &: i\not=k\end{cases} = x_j \delta_{ik}

substituting this into the expression for $\prt \ell/\prt W_{ij}$ we get:

.. math::
   \pfrac{\ell}{W_{ij}}
   = \sum_{k=1}^{s_\out} x_j \delta_{ik} \pfrac{\ell}{u_k}
   = x_j \pfrac{\ell}{u_i}

or equivalently:

.. math::
   \pfrac{\ell}{W} = \pfrac{\ell}{\v u} \v x\T

Let X be the data matrix in which each *row* is an input vector and
U is the matrix in which each row it the corresponding output vector
then

.. math::
   U\T = W X\T

or

.. math::
   U = X W\T

where each *row* in U is the linear response to the corresponding
row in X. In this case:

.. math::
   \pfrac{\ell}{X\T} = W\T \pfrac{\ell}{U\T}

or

.. math::
   \pfrac{\ell}{X} = \pfrac{\ell}{U} W

For the derivative with respect to the weight matrix we have:

.. math::
   \pfrac{\ell}{W} = \left(\pfrac{\ell}{U}\right)^\top X

Bias Block
==========

The forward pass of the bias block is simple:

.. math::
   \v v = \v u + \v b

The graph for this block is given in :numref:‘fig-nn_bias‘.

it simply adds a vector $\v b$ to the input. Evidently this block
passes the the derivative of ℓ with respect to its output
directly to its input:

.. math::
   \pfrac{\ell}{\v u} = \pfrac{\ell}{\v v}

and also to the parameter vector:

.. math::
   \pfrac{\ell}{\v b} = \pfrac{\ell}{\v v}

we leave the proofs as exercises for the reader (again the easiest way
to do so is by considering all elements of the vectors).

The vectorized expressions are:

.. math::
   V = U + \v 1 \v b\T

note that $\v 1\v b\T$ constructs a matrix in which all the rows are
$\v b\T$. And also note that Numpy is smart enough to broadcast this
correctly in case we write :code:‘V = U + b‘ where :code:‘V‘ is an
array of shape :code:‘(m,n)‘ and :code:‘b‘ is an array of shape
:code:‘(n,)‘.

For the backpropagation pass we have:

.. math::
   \pfrac{\ell}{U} = \pfrac{\ell}{V}.

For the derivative of ℓ with respect to parameter vector $\v b$
we have to sum all $\prt\ell/\prt \v b = \prt\ell / \prt \v v\ls i$ for all examples in the
batch:

.. math::
   \pfrac{\ell}{\v b} = \sum_{i=1}^{m} \pfrac{\ell}{\v v\ls i}
   = \pfrac{\ell}{V}\T \v 1

where $\v 1$ is a vector with all elements equal to 1. Note that in
Numpy we can simply sum over all columns in array V.

Activation Block
==========

Consider a network of two linear layers. Let $\v x$ be the input and
$\v y$ be the output of the first layer and input to the second
layer. The total output $\v z$ of the two layers is given by:

.. math::
   \v z = W_2 (W_1 \v x + \v b_1) + \v b_2

It is because of the linearity that

.. math::
   \v z = W \v x + \v b

where $W = W_2 W_1$ and $\v b = W_2 \v b_1 + \v b_2$. So a neural
network with only linear units doesn't need hidden layers as it always
will act like one layer.

Therefore a non linear activation function is inserted. One layer then
functions as

.. math::
   \v y = g\aew(W\v x + \v b)

Note the use of $g\aew$ to indicate the elementwise application of the
function g to all elements in a vector (or matrix). It is this
strange operator in the linear algebra context that will be the cause
for some strange looking formula's in the back propagation procedure.

But first we concentrate on the activation function itself:

.. math::
   \v y = g\aew(\v v)

where we write $g\aew(\v v)$ (note the dot) to indicate the element
wise application of the function g, i.e. $y_i = g(v_i)$ for all i.

For the derivative of the loss with respect to the input vector we
have to remember that

.. math::
   y_i = g(v_i)

and thus

.. math::
   \pfrac{\ell}{v_i} = \pfrac{\ell}{y_i}\pfrac{y_i}{v_i} = g'(v_i)\pfrac{\ell}{y_i}

The vector $\prt\ell/\prt v$ then is the element wise multiplication
of the vector $g'\aew(\v v)$ and $\prt\ell/\prt\v y$. Standard linear
algebra has no concise notation for that. In the ML literature the
operator $\odot$ is used for that purpose:

.. math::
   \pfrac{\ell}{\v v} = g'\aew(\v v) \odot \pfrac{\ell}{\v y}

Element wise multiplication is of course easy to do in Numpy, but very
special in linear algebra (it is not linear of course).

In modern days use of neural networks it is especcially the ReLu
activation function that is used quite often. We refer to the
'Wikipedia page <https://en.wikipedia.org/wiki/Activation_function>'_
on activation functions for a nice overview of a lot of different
activation functions. That page tabulates both the expressions for the
activation function g but also the expression for the
derivative. Implementation in Python/Numpy should be easy from there.

The vectorized expression for batches V and Y is:

.. math::
   \pfrac{\ell}{V} = g'\aew(V) \odot \pfrac{\ell}{Y}

where the activation function is applied elementwise to all elements
in the matrix V and is element wise multiplied with the matrix
$\pfrac{\ell}{Y}$.

Loss Function
=============

In this section we only consider the derivations for the L_2 loss,
i.e. the quadratic loss function:

.. math::
   \ell = \ell(\hat{\v y}, \v y) = \half \|\hat{\v y} - \v y\|^2

where $\v y$ is the target value. Note that $\hat{\v y}$ is
the output of the entire neural network. In our loss function
we have not incorporated any regularization term.

Here we have:

.. math::
   \pfrac{\ell}{\hat{\v y}} = \hat{\v y} - \v y

We see that is the error, i.e. the difference between what the network calculates for output ($\hat{\v y}$) and what it should be according to our learning set $\v y$, that starts of the chain of backpropagation steps through all the layers from the front all the way backwards to the input.

Vectoring the loss function and derivate we get the cost for the entire batch as the sum of the individual losses.

.. math::
   J = \frac{1}{m} \sum_{i=1}^m \ell(\hat{\v y}\ls i, \v y\ls i) = \frac{1}{2m} \|\hat{Y} - Y\|_F^2

where $\|\cdot\|_F$ is the Frobenius norm of a matrix (sum of all squared elements), $\hat{Y}$ is the matrix of all outputs of the neural net (as rows in the matrix), and Y is the matrix of all target vectors as rows. Now the derivative becomes:

.. math::
   \pfrac{\ell}{\hat{Y}} = \frac{1}{m}(\hat{Y} - Y)

One Layer in a Neural Network
==========

In the figure below one layer in a fully connected neural network is sketched. Not only the arrows and formula's for the forward pass are given but also the arrows and formula's for the backward pass.

The first sketch is for one training example, input $\v x$ and output $\hat{\v y}$.

The formulas for the forward pass by concatenating the actions of the three blocks in one layer (see previous sections). In the formula's below we write W instead of $W\ls i$ and $\v b$ instead of $\v b\ls i$.

.. math::
   \v u &= W \v x\\
   \v v &= \v u + \v b &= W\v x + \v b\\
   \hat{\v y} &= g\aew(\v v) &= g\aew(W\v x + \v b)

For the backpropagation of the 'error' we have:

.. math::
   \frac{\prt\ell}{\prt \v v} &= g'\aew(\v v) \odot \frac{\prt\ell}{\prt \hat{\v y}} &\\
   \frac{\prt\ell}{\prt \v u} &= \frac{\prt\ell}{\prt \v v} &= g'\aew(\v v) \odot \frac{\prt\ell}{\prt \hat{\v y}}\\

\frac{\prt\ell}{\prt \v x} &= W\T \frac{\prt\ell}{\prt \v u} &=
W\T \left(g'\aew(\v v) \odot \frac{\prt\ell}{\prt \hat{\v y}}\right)

and the gradients of the error with respect to the parameters W and
$\v b$ are given by:

.. math::
\frac{\prt\ell}{\prt \v b} &= \frac{\prt\ell}{\prt \v v} = g'(\v v) \odot \frac{\prt\ell}{\prt \hat{\v y}} \\
\frac{\prt\ell}{\prt W} &= \frac{\prt\ell}{\prt \v u} \v x\T =
\left(g'(\v v) \odot \frac{\prt\ell}{\prt \hat{\v y}}\right) \v x\T\\

Now let us consider an input batch X and corresponding output batch
Y. In this case the graph looks like:

The forward pass is described with:

.. math::
U &= X W\T \\
V &= U + \v 1 \v b\T &= X W\T + \v 1 \v b\T\\
\hat{Y} &= g\aew(V) &= g\aew(X W\T + \v 1 \v b\T)

For the backpropagation of the 'error' we have:

.. math::
\frac{\prt\ell}{\prt V} &= g'(V) \odot \frac{\prt\ell}{\prt \hat{Y}} &\\
\frac{\prt\ell}{\prt U} &= \frac{\prt\ell}{\prt V} &= g'(V) \odot \frac{\prt\ell}{\prt \hat{Y}}\\
\frac{\prt\ell}{\prt X} &= \frac{\prt\ell}{\prt U} W
&= \left(g'(V) \odot \frac{\prt\ell}{\prt \hat{Y}}\right) W

and the gradients of the error with respect to the parameters W and
$\v b$ are given by:

.. math::
\frac{\prt\ell}{\prt \v b} &= \frac{\prt\ell}{\prt V}\T \v 1 =
\left(g'(V) \odot \frac{\prt\ell}{\prt \hat{Y}}\right)\T \v 1 \\
\frac{\prt\ell}{\prt W} &= \frac{\prt\ell}{\prt U}\T X =
\left(g'(V) \odot \frac{\prt\ell}{\prt \hat{Y}}\right)\T X\\

Fully Connected Neural Network
==========

Below you find the graph for one layer but this time we use indexed
symbols for the variables to make it easier to come up with an
algorithm for the entire network.

The first layer has index $i=0$ (to make Python coding adhere to the
formules) and thus

.. math::
A_0 = X

where X is the input (batch) for the network. The shape of X is (m,n) where m is the number of samples in the batch and n is the number of features. The forward pass going from A_i to A_{i+1} is given by:

.. math::
   Z_i &= A_i W\T + \v 1 \v b_i\T\\
   A_{i+1} &= g\aew(Z_i)

Python code for the forward pass in the entire network is easy (assuming all arrays are properly initialized).

.. code:: python

   A[0] = X
   for i in range(L):
      Z[i] = A[i] @ W[i].T + b[i]
      A[i+1] = g(Z[i])
   return(A[i+1])

The backward pass is not too difficult either. First in formula's:

.. math::
   \Delta_{L-1} &= (A_{L-1} - Y)/m\\
   \Delta_{i} &= (g'\aew(Z_i) \odot \Delta_{i+1}) W

where Y are the target vectors collected in a matrix. In Python this is something like:

.. code:: python

   D[L-1] = (A[L-1] - Y)/m
   for i in range(L-2,0,-1):
      D[i] = (g_prime(Z[i]) * D[i+1]) @ W

Note that we could have calculated D[0] too but we don't need it in calculating the derivatives with respect to all parameters.

Calculating the derivatives with respect to the parameters is needed to make one step in a gradient descent procedure. In formula:

.. math::
   b_i := b_i - \alpha (g'\aew(Z_i) \odot \Delta_{i+1})\T \v 1\\
   W_i := W_i - \alpha (g'\aew(Z_i) \odot \Delta_{i+1})\T A_i

and in Python:

.. code:: python

   for i in range(L):
      b[i] -= alpha * sum(g_prime(Z[i]) * D[i+1], axis=0)
      W[i] -= alpha * (g_prime(Z[i]) * D[i+1]).T @ A[i]

We leave it as an exercise to the reader to combine these code snippets into working code to train and use a fully connected neural network.

Just like in the gradient descent procedure for linear or logistic regression the parameters need to be initialized before the first step is made. Whereas for linear and logistic regression taking initial values to be the same (and zero) is. This is not true for a neural network. We refer to the literature (Andrew Ng's coursera course has a nice explanation for this).

Clustering
==========

Clustering is the process to group closely related feature vectors. We will consider clustering of real valued feature vectors $\v x\ls i\in\setR^n$. Clustering is an **unsupervised learning algorithm**: only the set of feature vectors is known, there are no target values.

.. ipython:: python
   :suppress:

   plt.clf()
   from clustering import make_datasets, plot_datasets
   dsets = make_datasets(500)
   plot_datasets(dsets, pltlabels=True)
   plt.savefig('source/figures/clustering_examples.png', bbox_inches='tight')

   Clusters in 2D data sets. In every column the data set is shown in the top row, in the second row the same data set but then the 'true' cluster label is used to distinguish the clusters. Note a 'true' cluster label in practice is not known. (the clusters are generated with standard sklearn code).

In :numref:'fig-clusters' some 2D datasets are shown. In all cases it is quite obvious what the clusters should approximately be (with the possible exception of the clusters in column (b)).

The clusters in the first three columns are relatively simple. A cluster is well represented with its center and membership to a cluster can be expressed in terms of distance to the cluster center. Note that in columns (b) en (c) the distances should take the shape/size of the clusters into account.

The clusters in the first three columns are the types of clusters that can be found using (variants of) the k-Means algorithm. In this chapter we only discuss these algoritms in depth.

The clusters in column (d) en (e) are more difficult. The exists a variety of algorithms that can deal with these types of clusters. We refer to the manual of sklearn for this.

k-Means Clustering
==================

k-Means clustering is a **partional clustering** approach in which each cluster is associated with a centroid $\v\mu_j$ (center or prototype point). The number of clusters k is considered fixed and has to be chosen a priori. Each point $\v x\ls i$ in the data set is then assigned to the cluster with the closest centroid.

Let c_i be the cluster assigned to point $\v x\ls i$. Then the goal of k-means clustering is to minimize the following cost function:

.. math::
   J(c_1,\ldots,c_m, \v\mu_1,\ldots,\v\mu_k) =
   \sum_{j = 1}^k \sum_{i: c_i = j} \| \v x\ls i - \v\mu_j \|^2

Observe that the cluster labels c_i are discrete values. Gradient descent is therefore not a logical choice. The cluster centers $\v\mu_j$ are continuous (n dimensional) variables. It can be shown that finding the optimal parameters c_i and $\v\mu_j$ is an NP hard problem.

The k-means optimization problem is therefore done in a two step process. First we fix the cluster centers and find the c_i that minimize the cost. Evidently if we set:

.. math::
   :label: eq-labels

   c_i = \arg\min_{j} \|\v x\ls i - \v\mu_j\|

the distance from $\v x\ls i$ to the selected cluster center is minimal and doing that for all points $\v x\ls i$ minimizes the cost.

Then given the classification we fix the c_i and calculate the optimal cluster centers $\v\mu_j$. To do so we calculate the gradient:

.. math::
   \pfrac{ J(c_1,\ldots,c_m, \v\mu_1,\ldots,\v\mu_k)}{\v\mu_j} &=
   \sum_{i:c_i=j} \pfrac{\|\v x\ls i - \v\mu_j\|^2}{\v\mu_j}\\
   &= \sum_{i:c_i=j}\pfrac{(\v x\ls i - \v\mu_j)\T(\v x\ls i - \v\mu_j)}{\v\mu_j}\\
   &= \sum_{i:c_i=j}\pfrac{\left((\v x\ls i)\T\v x\ls i -
   2\v\mu_j\T\v x\ls i + \v\mu_j\T\v\mu_j\right)}{\v\mu_j}\\
   &= \sum_{i:c_i=j} \left(-2\v x\ls i + 2\v\mu_j \right)

Observe that this minimization can be solved analytically. We set the gradient (with respect to $\v\mu_j$ equal to zero and solve for $\v\mu_j$:

.. math::
   \sum_{i:c_i=j}\v x\ls i &= \sum_{i:c_i=j}\v\mu_j\\
   &= \v\mu_j \sum_{i:c_i=j} 1

Note that the summation in the right hand side just calculates the

number of points in cluster j. Let’s call that number m_j then we arrive at:

.. math::
   :label: eq-clusters

   \v\mu_j = \frac{1}{m_j} \sum_{i:c_i=j} \v x\ls i

So given a fixed set of clusters the optimal cluster center $\v\mu_j$ is the mean of all points assigned to that cluster.

The k-means algorithm consists of selecting k cluster centers at random and then alternating these two optimization steps:

.. literalinclude:: /python/clustering.py
   :pyobject: kmeans_clustering

In an exercise you have to implement four functions:

.. code-block:: python

   select_initial_clusters(data, n_clusters)

this function should return a ‘‘(n_clusters, n)‘‘ shaped array of cluster centers, each row a cluster center. A simple way to do that is to randomly select ‘‘n_clusters‘‘ points from the data set.

.. code-block:: python

   assign_labels(data, clusters, n_clusters)

this function should return a ‘‘(m,)‘‘ shaped array of label values (from 0 to n_clusters) indicating which cluster center is closest. This function implements :eq:‘eq-labels‘.

.. code-block:: python

   calculate_clusters(data, labels, n_clusters)

this function should return a ‘‘(n_clusters, n)‘‘ shaped array of cluster centers. This function implements :eq:‘eq-clusters‘.

Finally you have to implement the function

.. code-block:: python

   test_if_done(data, labels, clusters, new_clusters, n_clusters)

As a default implementation you could just check whether the new clusters are close to the previous clusters.

.. ipython:: python

```
   :suppress:

   plt.clf()
   from clustering import make_datasets, kmeans_clustering, plot_datasets_labels
   dsets = make_datasets(500)
   plot_datasets_labels(dsets, [kmeans_clustering])
   plt.savefig('source/figures/kmeansclustering.png', bbox_inches='tight')
```

In :numref:'fig-kmeans' the 5 data sets from the introduction are shown as clustered by the k-means algorithm.

From the figure it is clear that k-means clustering only really works for the first data set. In (a) clusters of spherical shape of about the same size are shown. In (b) we do have spherical clusters but of varying sizes, k-means tend to assign points from the larger clusters to smaller clusters. In (c) where we see ellipsoid clusters, a situation that k-means cannot deal with. For the non ellipsoid clusters in (d) and (e) k-means clustering is totally insufficient.

k-Means, towards Gaussian Mixture Models
==

We start with a simple extension of the k-means algorithm in order to deal with spherical blobs of varying size. This is done by keeping track of the 'variance'

```
.. math::
   \sigma_j^2 = \frac{1}{m_j} \sum_{i:c_i=j} \|\v x\ls i - \v\mu_j\|^2
```

and measuring the distance of a point $\v x\ls i$ to a cluster $\v\mu_j$ relative to the variance:

```
.. math::
   c_i = \arg\min_{j} \frac{\|\v x\ls i - \v\mu_j\|}{\sigma_j}
```

With this variance weighted distance measure the distance to a cluster center is effictively measured with a ruler for which the unit length is equal to the standard deviation of that cluster of points. The clustering of the 5 datasets is depicted below in :numref:'fig-kmeansvariance'.

```
.. ipython:: python
   :suppress:
   :okwarning:

   from clustering import make_datasets, kmeans_clustering, \
                          kmeans_variance_clustering, kmeans_covariance_clustering, \
     plot_datasets_labels
   dsets = make_datasets(500)
```

```
  np.random.seed(170)
  plot_datasets_labels(dsets,
                       [kmeans_clustering, kmeans_variance_clustering])
  plt.savefig('source/figures/kmeansvarianceclustering.png', bbox_inches='tight')
```

From the figure it is clear that variance weighted k-means is better suited to deal with spherical clusters of different sizes. But still it can't make sense of the ellipsoidal clusters in the 3rd column.

To deal with these ellipsoidal shaped clusters we will calculate the covariance matrix for each of the classes:

```
.. math::
  \Sigma_j = \frac{1}{m_j} \sum_{i:c_i=j} (\v x\ls i - \v\mu_j)(\v x\ls i - \v\mu_j)\T
```

and use the (squared) Mahalanobis distance for the label assignment:

```
.. math::
  c_i = \arg\min_j (\v x\ls i - \v\mu_j)\T \Sigma_j\inv (\v x\ls i - \v\mu_j)
```

where

```
.. math::
  d_M(\v x) = \sqrt{(\v x-\v\mu)\T \Sigma\inv (\v x-\v\mu)}
```

is the **Mahalanobis distance** from a point $\v x$ to the distribution with mean $\v\mu$ and covariance matrix Σ. Note that for a scalar x and distribution with mean μ and variance σ^2 the Mahalanobis distance is equal to the distance measure where the length of the difference is divided by the standard deviation (as used at the start of this section).

The Mahalanobis distance generalises this idea of using the variances (in each direction) as yardstick. That is we measure distance in terms of how many standard deviations point $\v x$ is away from the center of the distribution. Thus we take the 'shape' of the distribution into account.

Using the Mahalanobis distance will correctly cluster the data set in column (c) in :numref:'fig-kmeansvariance'. But will not deal with the data sets in the last two columns. Other ways of clustering are needed in these cases.

Clustering can be taken one (large) step further by using a Gaussian mixture model (GMM). In a GMM not only the Mahalanobis distance is estimated and used as a distance measure but the assignment to a cluster is not 'hard' (in the sense that the closest cluster wins) but 'soft' in the sense that each point is assigned to each cluster with a 'strength' that is inversely proportional to its distance to that cluster center.

A detailed description of the GMM is outside the scope of these lecture notes.

Evaluating Classifiers
======================

We will be looking at a binary (two class) classifier for which Bayes classification rule tells us:

.. math::
   \hat y = \classify(\v x) = \begin{cases}1 &: P(Y=1|\v X=\v x)\geq P(Y=0|\v X=\v x)\\
   0 &: P(Y=1|\v X=\v x)< P(Y=0|\v X=\v x)
   \end{cases}

or equivalently:

.. math::
   \hat y = \classify(\v x) = \begin{cases}1 &: P(Y=1|\v X=\v x)\geq \tfrac{1}{2}\\
   0 &: P(Y=1|\v X=\v x)< \tfrac{1}{2}
   \end{cases}

In our evaluation of the classifier it is not important whether the a posteriori probability is calculated by estimating the class conditional probability (density) of the data or is estimated directly with an hypothesis function $h_{\v\theta}$.

The important thing is that we can make this into a parameterized classifier by selecting a **classification threshold** (or **decision threshold**) different than $1/2$:

.. math::
   \hat y = \classify(\v x; t) = \begin{cases}1 &: P(Y=1|\v X=\v x)\geq t\\
   0 &: P(Y=1|\v X=\v x)< t
   \end{cases}

The results of the classifier on a dataset can be summarized in a **confusion matrix** where we call $y=1$ POSITIVE and $y=0$ NEGATIVE (often in classification the y value indicates POSITIVE or NEGATIVE (TRUE or FALSE, PASS or FAIL, etc).

$\quad$	$y=0$	$y=1$
$\hat y = 0$	TN	FN
$\hat y = 1$	FP	TP

where

:TP:
  stands for True Positives indicating the number of elements $\v x$ in the test set that are correctly (TRUE) classified as 1 (POSITIVE), i.e. $\hat y=1$ and $y=1$.

:TN:
  stands for True Negatives, the number of elements that are correctly
  classified as NEGATIVE, i.e. $\hat y=0$ and $y=0$.

:FP:
  stands for False Positive indicating the number of elements that are
  incorrectly classified as positive, i.e. $\hat y=1$ and $y=0$.

:FN:
  stands for False Negative, the number of elements that are
  incorrectly classified as negaive, i.e. $\hat y=0$ and $y=1$.

From the confusion matrix we can easily calculate the total
accuracy as the fraction of correctly classified examples, i.e.

.. math::
   \text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}

Achieving the maximum accuracy of a classifier is not always the
goal. Consider a decision where the false positive error has more
severe negative consequences than a false positive error. For instance
when testing a patient on the corona virus it is better to tolerate
some more false positives (thinking a patient might have covid-19) and
have less false negative errors. Note that a false negative means that
a patient who *has* covid-19 is not treated as such. By selecting a
threshold t we can balance FP and FN errors (possibly at the expense
of some of the accuracy).

In situations where the True Positives are much more likely than the
True Negatives the accuracy also isn't very indicative. Consider for
instance spam classification where around 99% of the emails are
spam. Setting a classifier in that case to always return a positive
result (spam in this case) leads to a 99% accuracy.

In these situations the **precision** and **recall** measures might be
more interesting. Precision is defined as:

.. math::
   \text{Precision} = \frac{TP}{TP + FP}

i.e. the fraction of positive classifications that were actual
positive as well.

The recall is defined as:

.. math::
   \text{Recall} = \frac{TP}{TP + FN}

i.e. what fraction of actual positives were classified as such (again
note that FN indicate the actual positive cases that are classified as
negative).

..

In order to get a feeling for the influence of the decision threshold on the confusion matrix (and hence on the accuracy, precision and recall) let’s consider a simple classification problem on a 2D dataset as sketched in :numref:‘fig-decisionthresholdclassifier‘.