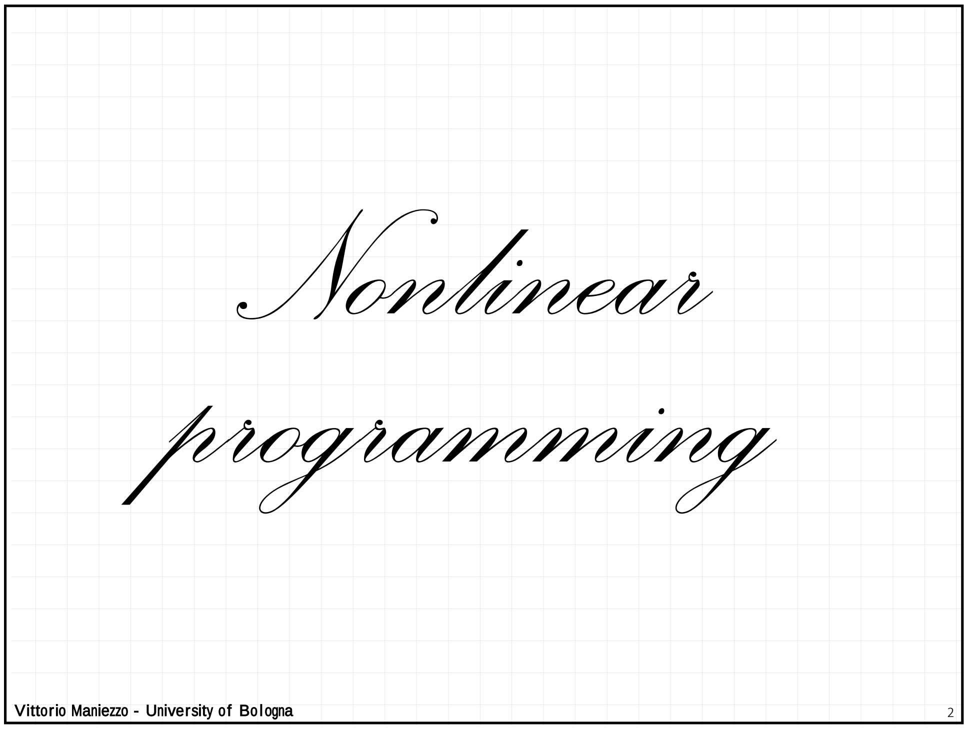
Optimization

A gentle introduction

Vittorio Maniezzo
Univ. Bologna

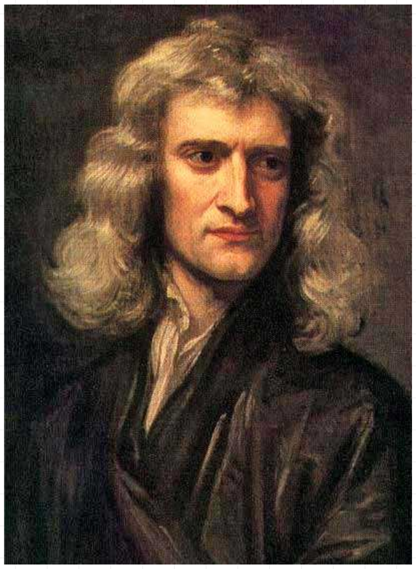
1



Nonlinear programming

2

The shoulders of giants

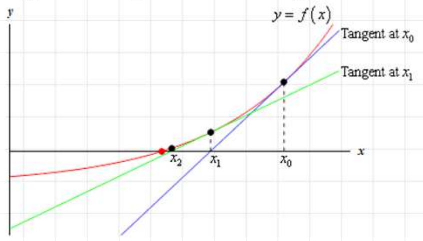


Newton's method

A root finding method that uses linear approximations of the function.

It tries to guess a solution x_0 of the equation $f(x)=0$, compute the linear approximation of $f(x)$ at x_0 and then finds the x-intercept ($y=0$) of the linear approximation.

Let $f(x)$ be a differentiable function.
If x_0 is near to a solution of $f(x)=0$, we can approximate $f(x)$ by the tangent line at x_0 and compute the x_0-intercept of the tangent line (should be near to the solution!).



Newton's method

The equation of the tangent line at x_0 is

$$y = f'(x_0)(x - x_0) + f(x_0)$$

The x-intercept is the solution x_1 of the equation

$$0 = f'(x_0)(x - x_0) + f(x_0)$$

solving for x_1

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Repeating this procedure, we obtain a sequence given by the recursive formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

which (potentially) converges to a solution of the equation $f(x)=0$

Newton's method

Advantages: when it converges, Newton's method usually converges very quickly.

Disadvantages:

- there's no guarantee that the method converges to a solution. We should set a maximum number of iterations so that our implementation ends if we don't find a solution.
- the stopping criteria for Newton's method doesn't know how close it is to a solution. All we can compute is the value $f(x)$ and so we implement a stopping criteria based on $f(x)$.
- the method requires computing values of the derivative of $f(x)$, a disadvantage if the derivative is difficult to compute.

Newton's method

Python implementation (1d). Specifications:
Input: f, f', x_0, ε and *max_iter* and
Output: an approximation of a solution of $f(x)=0$.

The function may terminate in 3 ways:

- If $\text{abs}(f(x_n)) < \varepsilon$, the algorithm found an approximate solution and returns x_n.
- If $f'(x_n) = 0$, the algorithm stops and returns None.
- If the number of iterations exceed *max_iter*, the algorithm stops and returns None.

```
def newton(f,Df,x0,epsilon,max_iter):
    xn = x0
    for n in range(0,max_iter):
        fxn = f(xn)
        if abs(fxn) < epsilon:
            print('Found solution after {0} iterations.'.format(n))
            return xn
        Dfxn = Df(xn)
        if Dfxn == 0:
            print('Zero derivative. No solution found.')
            return None
        xn = xn - fxn/Dfxn
    print('max_iter exceed. No solution found.')
    return None
```

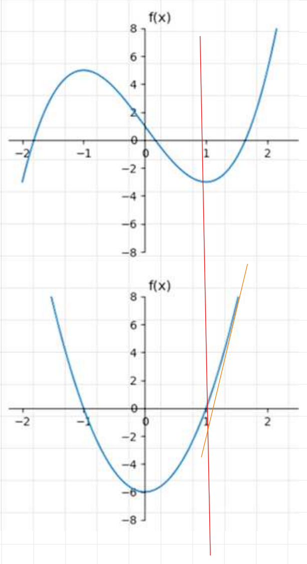
Newton's method for optimization

Optima (minima, maxima) are points where the derivative is null. We look for the roots of the derivative.

At any point, for a 1D function, the derivative points to a direction that increases it (nearby): given a function $f(x)$, if its derivative $f'(x)$ is positive, then increasing x will increase $f(x)$ and decreasing it will decrease $f(x)$.

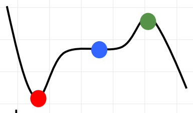
To minimize $f(x)$, we should take small steps opposite to the sign of $f'(x)$.

(ex. $f=2x^3 - 6x + 1$ $f'(x) = 6x^2 - 6$)



Newton's method for optimization

In optimization, Newton's method is thus applied to the derivative f' of a twice-differentiable function f to find the roots of the derivative (solutions to $f'(x) = 0$).



The roots are known as stationary points of f.

These solutions may be minima, maxima, or saddle (aka inflection) points.

If there are multiple variables (say x and y), we can have a derivative with each of them: we get a 2D vector, the *gradient* of the function.

Moving along the direction of the gradient will increase the function while moving opposite to it will decrease it (for small steps).

Newton's method, example

Function $f(x,y) = x^2+y^2$. This is a paraboloid, minimum $x=0$ and $y=0$.

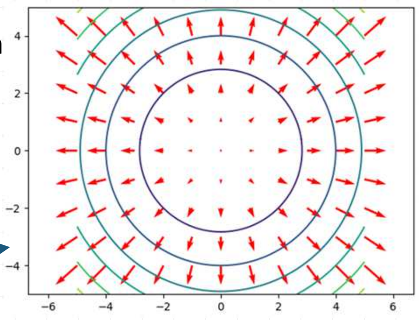
The derivatives with respect to x and y are $2x$ and $2y$.

The gradient (Jacobian of a scalar function) is the vector $\nabla f = [2x, 2y]$.

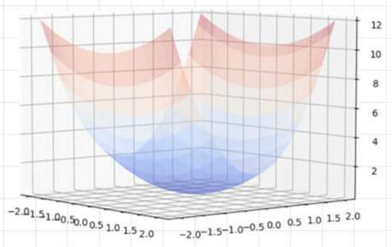
∇f is zero only when $x=0$ and $y=0$. Otherwise, the gradient points in a direction where $f(x,y)$ will increase and the direction opposite the gradient will decrease $f(x,y)$.

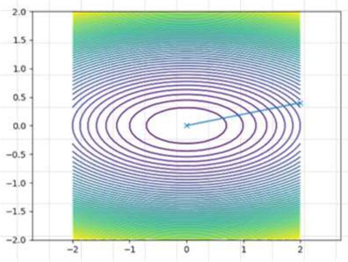
In an unconstrained optimization problem, a necessary (not sufficient) condition to be at a local optima is $\nabla f = 0$

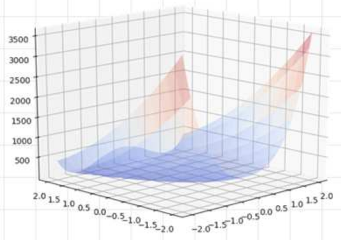
Gradient field

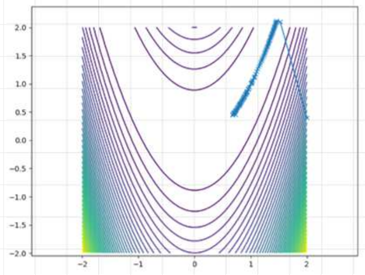


Newton's method, example









Exercises

Derivative-free methods:

- Simple line search ($x_{t+1} = x_t + \lambda_t d_t$), with d heuristic direction.
- Dichotomous search.
- Golden section method (α=0.618)

Gradient descent

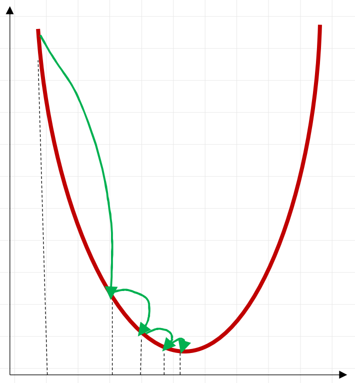
Gradient (or steepest) descent (Cauchy,1847) is the core of many learning algorithms (machine learning, deep learning, …).

GD is a first-order iterative algorithm for finding the local minimum of a differentiable function $f(x)$ (it's *gradient ascent* for the maximum).

To find a local minimum it starts from a point x_0 and takes steps proportional to the negative of the gradient (or of an approximate gradient) of the function at the current point.

The length of the steps is dictated by a parameter, the learning rate (λ).

$$x_{t+1} = x_t - \lambda \nabla f(x_t)$$



Vittorio Maniezzo - University of Bologna 13

13

Gradient descent, example

Cost function $f(x) = x^3 - 4x^2 + 6$

Derivative $f'(x) = 3x^2 - 8x$ (the slope of $f(x)$ at any given x)

Start with any value of x, say 0.5 and $\lambda = 0.05$

Iterate a number of times, always re-calculating the value of x.

$x = x - \lambda * f'(x)$

$x = 0.5 - (-3.25*0.05) = 0.6625$

Use $x = 0.6625$ in second iteration

$x = 0.6625+(3.983*0.05) = 0.86165$

and so on for a number of iterations, or until a predefined value of precision is reached.

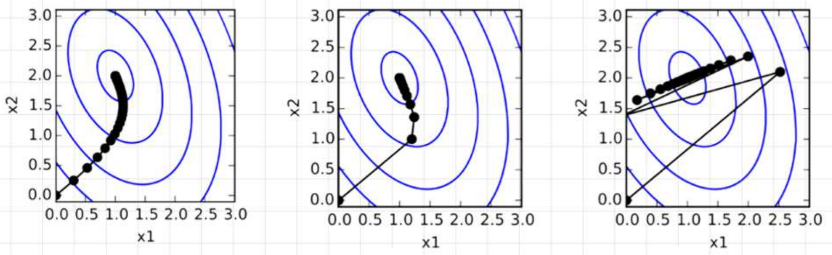
Vittorio Maniezzo - University of Bologna 14

14

Step size

Choice of λ matters a lot in practice:

$$\underset{x}{\text{minimize}}\ 2x_1^2 + x_2^2 + x_1x_2 - 6x_1 - 5x_2$$



$\lambda = 0.05$ $\lambda = 0.2$ $\lambda = 0.42$

Gradient descent, least squares

Example: linear regression where we estimate the coefficients of the equation $y = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$

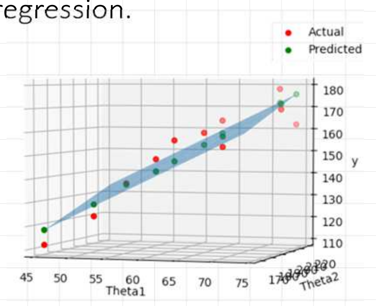
The optimal set of coefficients is determined by minimizing loss (cost, objective) function $\|y - \Theta' X\|^2$, which is the maximum likelihood estimator for the task of linear regression.

Simple example: 1D array of n points.

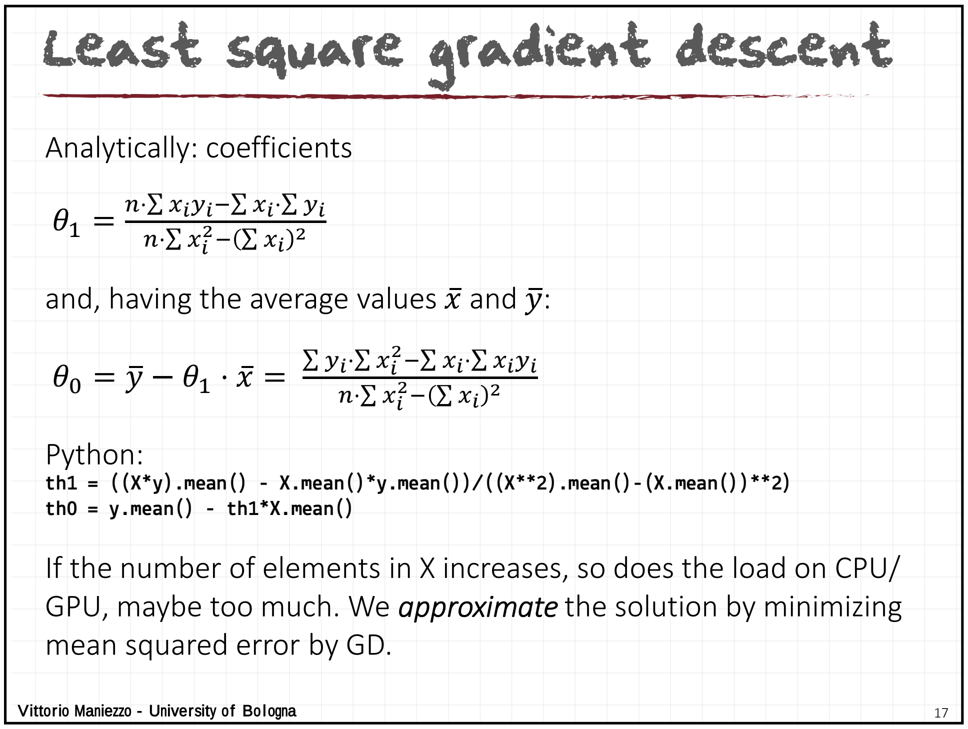
We look for the interpolation line

$y = \theta_0 + \theta_1 x$ that minimizes the

mean square error on all data pairs (x_i, y_i):



$$e(\boldsymbol{\Theta}) = \sum \big((x_i, y_i) - (x_i, \theta_0 + \theta_1 x_i)\big)^2$$



Least square gradient descent

Analytically: coefficients

$$\theta_1 = \frac{n \cdot \sum x_i y_i - \sum x_i \cdot \sum y_i}{n \cdot \sum x_i^2 - (\sum x_i)^2}$$

and, having the average values $\bar{x}$ and $\bar{y}$:

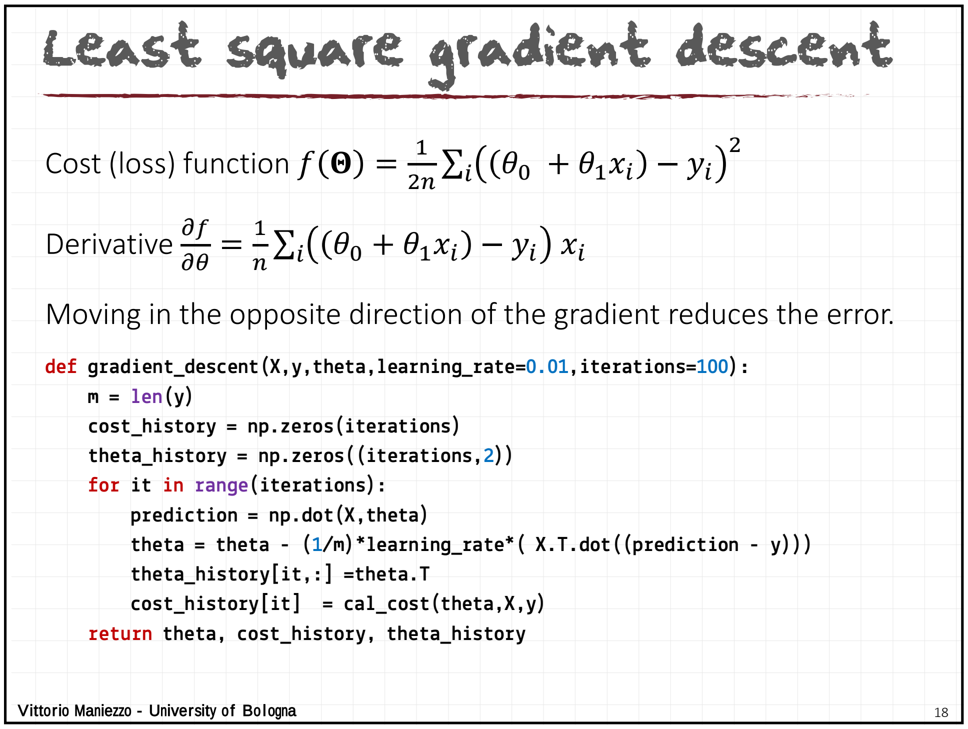
$$\theta_0 = \bar{y} - \theta_1 \cdot \bar{x} = \frac{\sum y_i \cdot \sum x_i^2 - \sum x_i \cdot \sum x_i y_i}{n \cdot \sum x_i^2 - (\sum x_i)^2}$$

Python:
```
th1 = ((X*y).mean() - X.mean()*y.mean())/((X**2).mean()-(X.mean())**2)
th0 = y.mean() - th1*X.mean()
```

If the number of elements in X increases, so does the load on CPU/ GPU, maybe too much. We *approximate* the solution by minimizing mean squared error by GD.

Vittorio Maniezzo - University of Bologna 17

17



Least square gradient descent

Cost (loss) function $f(\mathbf{\Theta}) = \frac{1}{2n} \sum_i \left((\theta_0 \ + \theta_1 x_i) - y_i \right)^2$

Derivative $\frac{\partial f}{\partial \theta} = \frac{1}{n} \sum_i \left((\theta_0 + \theta_1 x_i) - y_i \right) x_i$

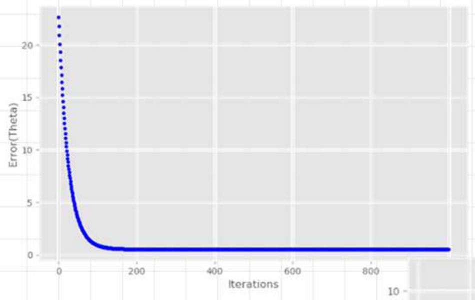
Moving in the opposite direction of the gradient reduces the error.

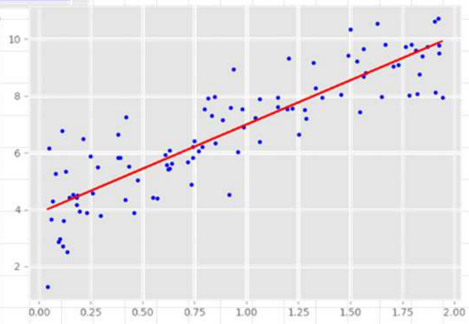
```
def gradient_descent(X,y,theta,learning_rate=0.01,iterations=100):
    m = len(y)
    cost_history = np.zeros(iterations)
    theta_history = np.zeros((iterations,2))
    for it in range(iterations):
        prediction = np.dot(X,theta)
        theta = theta - (1/m)*learning_rate*( X.T.dot((prediction - y)))
        theta_history[it,:] =theta.T
        cost_history[it]  = cal_cost(theta,X,y)
    return theta, cost_history, theta_history
```

Vittorio Maniezzo - University of Bologna 18

18

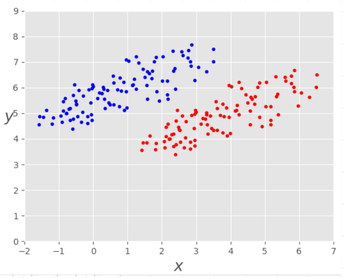
Least square gradient descent





GD linear classifier

Two clusters of points, find a line that separates (if possible) the two clusters.

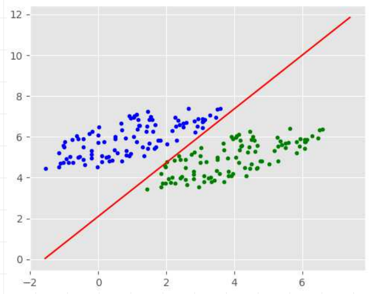


```
# generate points inside ellipse of center x0 y0
def get_random_ellipse(n, x0, y0):
    xout = np.zeros(n)
    yout = np.zeros(n)
    nkeep=0
    while nkeep < n:
        # equation of ellipse (x/x0) 2 + (y/y0) 2 = 1
        x=2*x0*(np.random.random(n-nkeep) - 0.5)
        y=2*y0*(np.random.random(n-nkeep) - 0.5)
        w,=np.where( ( (x/x0)**2 + (y/y0)**2 ) < 1 )
        if w.size > 0:
            xout[nkeep:nkeep+w.size] = x[w]
            yout[nkeep:nkeep+w.size] = y[w]
            nkeep += w.size
    return xout,yout
```

GD linear classifier

As with least square, but errors computed only on the misclassified points.

Line will be close (on) some points.
Not robust, see support vector machines (SVM).



The line acts as a linear classifier, separating all class A points on one side from all class B points on the other side.

Project idea: Logistic classifier (based on something like $z = \sigma(\theta x + b)$ with $\sigma = \frac{1}{1+e^{-z}}$) should be the go-to method for binary classification problems. More complex math.

Scipy

Newton (method='newton')

Newton's method minimizes f(x) as of a second order approximation of the function f(x) = f(x_0) + ∇f(x_0)'Δx + Δx'H(x_0)Δx, where H(x_0) is the Hessian matrix. The descent direction is given by:

d = -H(x_0)$^{-1}$*∇f(x_0).

```
from scipy.optimize import newton
def f(x):
    return (1.0/4.0)*x**3+(3.0/4.0)*x**2-(3.0/2.0)*x-2
x = 4
x0 = newton(f, x, fprime=None, args=(), tol=1.48e-08, maxiter=50, fprime2=None)
```

Gradient (method='gradient')

The gradient algorithm minimizes f(x) by first-order approximation: f(x) = f(x_0) + ∇f(x_0)'Δx, with descent direction, *d*, given by:

d = -∇f(x_0).

The ultimate iteration step is given by the minimization of f(α) = f(x_0) - α∇f(x_0) with a lineserach substep.

```
import numpy as np
from scipy.optimize import minimize
def f_and_grad(x):
   return x**2, 2*x
minimize(f_and_grad, [1], jac=True)
```

Stochastic gradient descent

If the dataset is big (say, a million samples) Gradient Descent has to use all the samples for completing each iteration, until the minima is reached. It can be computationally very expensive.

In Stochastic Gradient Descent (SGD), only a few samples (a “batch”) are randomly selected at each iteration.

In basic GD, the batch is taken to be the whole dataset: few iterations but very expensive.

Pure Stochastic Gradient Descent uses only a single sample, i.e., a batch size of one, to perform each iteration: many iterations very cheap.

Mini-batch GD trades-off the effectiveness with the complexity of each iteration.

SGD is usually at the heart of the backpropagation learning algorithm.

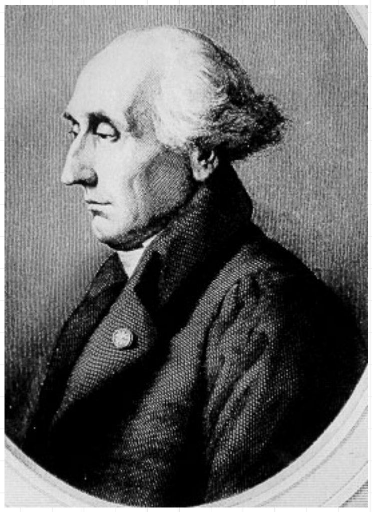
Project ideas

- Logistic regression (using only numpy and pandas)
- Logistic classifier (using only numpy and pandas)
- SGD for linear regression
- (Backpropagation, using only numpy and pandas. More complex.)
- Subgradient optimization (for people mathematically bent).
- iterated Newton-Raphson
- L-BFGS optimization.

The shoulders of giants

Giuseppe Luigi Lagrangia
(or Joseph-Louis Lagrange or Giuseppe Ludovico De la Grange Tournier)
25 January 1736 – 10 April 1813.

Moves into *constrained optimization*.

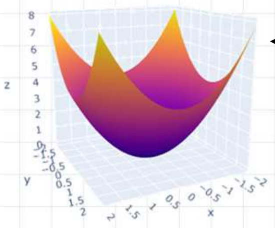


Lagrangian optimization

Appealing approach: you have constraints you do not like to have?

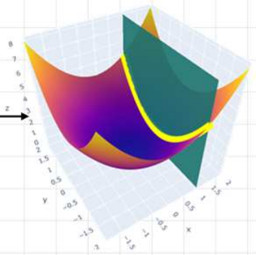
- Pretend they are not there!
- Solve as if they weren't there, but
- declare yourself the less happy the more a solution does not meet the constraints.

Ex. $\min f(x) = x^2 + y^2$ but subject to the constraint $x = 1$



unconstrained $\longleftarrow$

constrained $\longrightarrow$

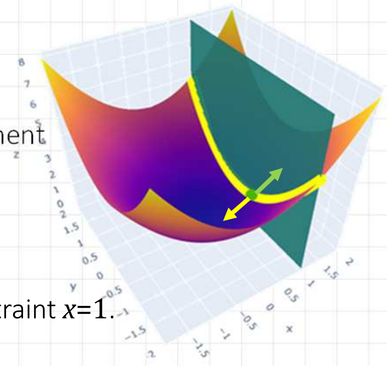


Equality constraints

Equality constraint may forbid to get zero gradient.

Gradient can be non-zero if moving in any direction that has a positive gradient component causes the violation of the constraint.

This can only happen when the plane of the constraint is perpendicular to the gradient



Example *min* $f(x,y)=x^2+y^2$ with equality constraint $x=1$.

Constraint: stay on the green plane.
If the gradient has a projection along the plane, we should move in the direction opposite to that projection. This will keep us on the plane, while reducing the objective function.

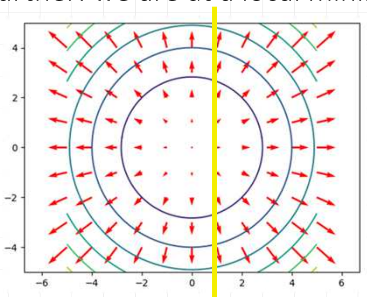
At the green point the yellow arrow (opposite to the gradient of objective function) has no projection along the plane: the yellow arrow is perpendicular to the plane (parallel to the gradient of the constraint).

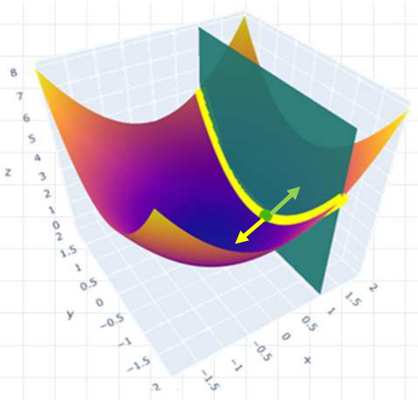
Equality constraints, example

To decrease the objective function, we need to move in a direction that has a component along the negative gradient.

But at the green point this forces us leaving the plane of the constraint.

The constraint makes it impossible to decrease the objective function further: we are at a local minimum.





Equality constraints

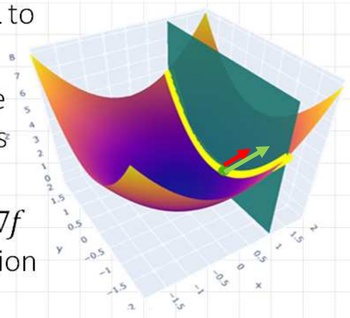
An easy way to check local optimality is to require that the red gradient of the objective function is parallel to the green gradient of the constraint plane (gradient of the plane is ⊥ to the plane, magnitude 1 along x).

If the two vectors are parallel, we can write one as a multiple of the other. Let's call this multiple μ.

If the gradient of the objective function is ∇f and that of the constraint is ∇h, the condition above is:

$$\nabla f = \mu \nabla h$$

The μ is called a *Lagrangian multiplier*.



Inequality constraints

Inequality constraints can be of two types: geq or leq.

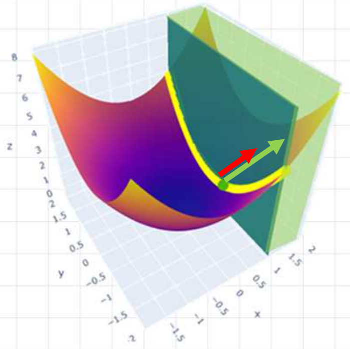
1) $g(x,y) = x\text{-}1 \geq 0$

The green point continues to be a local optimum, allowed space is beyond the constraint plane.

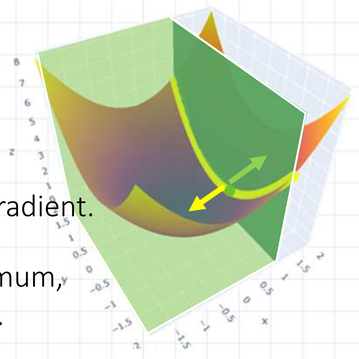
Since constraint's gradient (green) and the objective function's gradient (red) continue to point in the same direction, we have:

$$\nabla f = \lambda \nabla g$$

with $\lambda > 0$.



Inequality constraints



2) $g(x,y) = x - 1 \leq 0$

The feasible region is *before* the plane.
Movement is allowed along the negative gradient.

The green point is no longer the local optimum, we can move to (0,0), the global minimum.

At the green point, not a local optimum, we still have $\nabla f = \lambda \nabla g = 0$, but the yellow vector points opposite to the green vector, so we have $\lambda < 0$ and we can move feasibly toward decreasing o.f. values.

Inequality constraints

Putting it all together, for inequality constraints and minimization problems:

- For geq constraints, o.f. and constraint gradients are parallel and have the same direction, in P_0 we have $\nabla f = \lambda \nabla g$ with $\lambda > 0$: P_0 is a local minimum.
- For leq constraints, o.f. and constraint gradients are parallel but have the opposite directions, in P_0 we have $\nabla f = \lambda \nabla g$ with $\lambda < 0$: P_0 is not a local minimum.

For an inequality constraint, the condition $\nabla f = \lambda \nabla g$ indicates that we are at a local minimum only when $\lambda > 0$.

Complementary slackness

Recap, again. For the two inequality constraint:

- with x-1≥0, the green point was the solution. We were on the constraint plane $g(x) = x\text{-}1 = 0$. So, we had $g(x)=0$ and $\lambda>0$.
- with x-1≤0, another point $\mathbf{x'} = (0,0)$ was the local minimum, which was also the solution to the unconstrained problem. In x' we had $\nabla f=0$.
  The Lagrange condition requires $\nabla f = \lambda\nabla g$. In x' we get $\lambda\nabla g = 0$. But in x' $\nabla g\neq 0$ (not on the plane), therefore $\lambda=0$.

If the constraint is active ($g(\mathrm{x})=0$), we have $\lambda\geq 0$ while if it is not ($g(\mathrm{x})\neq 0$) we must have $\lambda=0$: one of the two must be zero in all cases.

This is called the complementarity slackness condition:

$\lambda_j\, g_j(\mathrm{x}) = 0$ for all $j \in \mathcal{I}$; ($\mathcal{I}$ inequality constraint set)

KKT (Karush-Kuhn-Tucker)

All in all, for a general optimization problem:

min	$f(x)$		
s.t.	$h_i(x) = 0$	$i \in \mathcal{E}$	(Equality constraint set)
	$g_j(x) \geq 0$	$j \in \mathcal{I}$	(Inequality constraint set)

The listed conditions can be summed up with reference to the so-called Lagrangian function of the problem

$$\mathcal{L}(x) = f(x) - \sum_{i\in E} \mu_i h_i(x) - \sum_{j\in I} \lambda_j g_j(x)$$

Upon differentiating and equating to 0 it becomes

$$\nabla\mathcal{L} = \nabla f - \sum_{i\in E} \mu_i \nabla h_i(x) - \sum_{j\in I} \lambda_j \nabla g_j(x) = 0$$

KKT (Karush-Kuhn-Tucker)

The full suite of KKT conditions required to be at a local optimum is:

1. $\nabla f = \sum_{i \in E} \mu_i \nabla h_i(x) + \sum_{j \in I} \lambda_j \nabla g_j(x)$ (o.f. gradient $\perp$ to all active constraints, optimality)
2. $h_i(x) = 0 \quad i \in \mathcal{E}$; (all equality constraints satisfied)
3. $g_j(x) \geq 0 \quad j \in \mathcal{I}$; (all inequality constraints satisfied)
4. $\lambda_j \geq 0 \qquad j \in \mathcal{I}$; (cannot decrease o.f. meeting inequalities)
5. $\lambda_j\, g_j(x) = 0$ for all $j \in \mathcal{I}$ (complementarity slackness)

These 5 eqs are the KKT necessary condition for local optimality.

KKT conditions

The KKT conditions are necessary conditions, not always sufficient.

To be sufficient the problem should have the functions $f(x)$ and $g_j(x)$ continuously differentiable and convex, and the functions $h_i(x)$ linear.

KKT conditions can be presented in different ways.

KKT for linear problems

For a standard form linear program, the KKT conditions become:

1. $c - \mu A - \lambda = 0$
2. $Ax = b$
3. $x \geq 0$
4. $\lambda \geq 0$.
5. $\lambda \cdot x = 0$ (dot product, complementary slackness)

The equations are non-linear due to the last, complementary slackness, condition

Lagrangian function

The above a condition to be at the optimum.

To find the optimum, we use condition 1, $\nabla f - \mu \nabla h - \lambda \nabla g = \mathbf{0}$ as a root finding equation of the derivative of the Lagrangian function

$$\mathcal{L}(x, \mu, \lambda) = f(x) - \mu\, h(x) - \lambda\, g(x)$$

Example:

$min \quad f(x, y) = 2x_1^2 + x_2^2 \Rightarrow min\ \mathcal{L}(\mathbf{x}, \mu) = 2x_1^2 + x_2^2 - \mu(x_2 + x_2 - 1)$

$s.t. \quad x_2 + x_2 = 1$

The constrained problem turned into unconstrained!

<u>*Often*</u> written as $min\ \mathcal{L}(\mathbf{x}, \mu) = 2x_1^2 + x_2^2 + \mu(1 - x_2 - x_2)$

Lagrangian problem

Lagrangian problem: $min\ \mathcal{L}(\mathbf{x},\mu)=2x_1^2+x_2^2+\mu(1-x_1-x_2)$

For each value of μ we have a different solution.

μ	x_1	x_2	$f(x_1,x_2)$	$h(x_1,x_2)$	$L(\mathbf{x},\mu)$
0	0	0	0	1	0
1	0.25	0.5	0.625	0.25	0.875
4/3	1/3	2/3	0.89	0	0.89
2	0.5	1	1.5	-0.5	0.5
3	0.75	1.5	2.625	-1.25	-1.125

For μ = 4/3 the constraint h is satisfied and f has a minimum at 0.89.

The Lagrangian $\mathcal{L}$ has a maximum where f has a feasible minimum.

True iff o.f. and constraints are all convex

Vittorio Maniezzo - University of Bologna 39

39

Lagrangian problem

How to optimize the Lagrangian function?

From KKT 1: $\nabla f=\mu\nabla h+\lambda\nabla g\Rightarrow\nabla f-\mu\nabla h-\lambda\nabla g=0$, we look for $\nabla\mathcal{L}$=0

$$\frac{\partial L}{\partial x_1}=4\mathrm{x}_1-\mu=0$$

$$\frac{\partial L}{\partial x_2}=2\mathrm{x}_2-\mu=0$$

$$\frac{\partial L}{\partial \mu}=1-\mathrm{x}_1-x_2=0$$

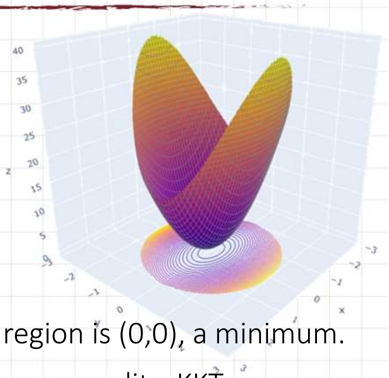
Solving this gives the solution x*=[1/3, 2/3], μ*= 4/3

Note that $\nabla f*$=[1/3x$_1^*$, 2/3x$_2^*$]=[4/3,4/3] is parallel to ∇h=[1,1]

Vittorio Maniezzo - University of Bologna 40

40

Lagrangian with inequalities



Find the maximum and minimum values of $f(x,y)=4x^2+10y^2$ on the disk $x^2+y^2\leq 4$

First order partial derivatives.

$$\frac{\partial f}{\partial x} = 8x = 0 \quad \Rightarrow x = 0$$
$$\frac{\partial f}{\partial y} = 20y = 0 \quad \Rightarrow y = 0$$

The only critical point inside the feasible region is (0,0), a minimum.

On the boundary, the constraint becomes an equality. KKT conditions are:

$$8x - 2\lambda x = 0$$
$$20y - 2\lambda y = 0$$
$$x^2+y^2=4$$

Lagrangian with inequalities

From the first equation (second would give the same) we get:

$2x(4-\lambda) = 0 \Rightarrow x = 0$ or $\lambda = 4$

- for $x=0$ the constraint gives $y=\pm 2$.
- for $\lambda=4$ the second equation gives $20y = 8y \Rightarrow y = 0$, thus the constraint $x=\pm 2$.

Lagrange Multipliers give four points to check: $(0,2)$, $(0,-2)$, $(2,0)$, $(-2,0)$.

To find the maxima and minima we plug these four points along with the critical point in the function.

$f(0,0)=0$	Minimum
$f(2,0)=f(-2,0)=16$	Saddle
$f(0,2)=f(0,-2)=40$	Maxima

In this case, the minimum was interior to the disk and the maxima were on the boundary of the feasibility region.

Optimizing multipliers

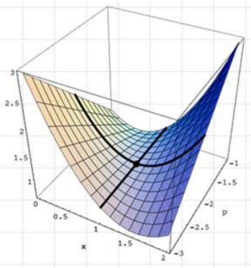
Need to find the stationary points of the Lagrangian function (those yielding $\nabla\mathcal{L} = 0$)

Unfortunately these points occur at saddle points, rather than at local maxima (or minima): changes in the original variables can decrease the Lagrangian, while changes in the multipliers can increase it. Gradient descent and the like only find max or min.

For example, minimize x^2 subject to $x=1$. The Lagrangian is $\mathcal{L}(x, \mu)=x^2+\mu(x-1)$, which has a saddle point at $x=1, \mu=-2$.

- For fixed μ, $\mathcal{L}$ is a parabola with minimum at $-\mu/2$ (1 when $\mu=-2$).
- For fixed x, $\mathcal{L}$ is a line with slope $x-1$ (0 when $x=1$).

Need of other algorithms for optimization.



Optimizing multipliers

Need to find the stationary points of the Lagrangian function (those yielding $\nabla\mathcal{L} = 0$)

The described method goes through computing $\nabla\mathcal{L}$=0.

What if the Lagrangian is non differentiable?

Need of other algorithms for optimization.

Best known is named subgradient optimization (out of scope for this course, *nice project*).

Application: SVM

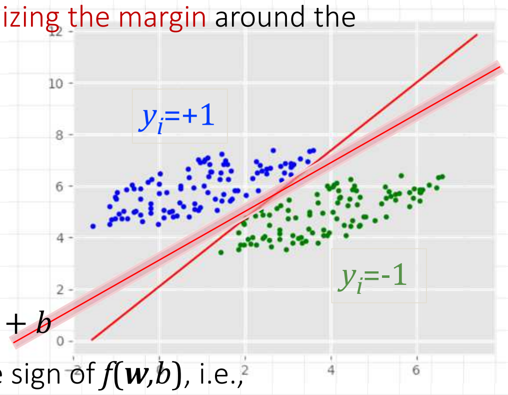
Support Vector Machines are machine learning classifiers which, given labeled training data (supervised learning), compute an optimal hyperplane which categorize new examples.

Optimality comes from maximizing the margin around the separating hyperplane.

This increases robustness in classification.

The separating hyperplane is determined by coefficients ($\boldsymbol{w}$,b):

$f(\boldsymbol{x_i}) = w_1 x_{i1} + \cdots + w_n x_{in} + b$



The classes are labelled by the sign of $f(\boldsymbol{w},b)$, i.e.,

$y_i = sign(f(x_i)) \in \{-1,1\}$

||w||

The distance between a point $\boldsymbol{x}$ and a hyperplane identified by ($\boldsymbol{w}$,b) is:

$$d(\boldsymbol{x}; \boldsymbol{w}, b) = \frac{|\boldsymbol{x} \cdot \boldsymbol{w} + b|}{||\boldsymbol{w}||}$$

with $||\boldsymbol{w}||$ *euclidean norm of vector* $\boldsymbol{w}$: $\sqrt{w_1^2 + w_2^2 + \ \ldots}$

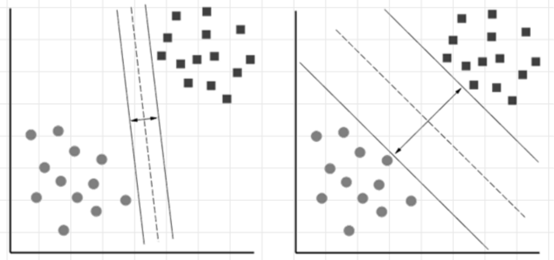
Thus, the distance between the hyperplane ($\boldsymbol{w}$,b) and the nearest point x of the dataset to separate is a function of $1/\|w\|$.

Minimizing the norm $||\boldsymbol{w}||$ corresponds to maximize the distance between the closest point and the hyperplane (its *margin*)

Example

Squares and circles are separated by an (hyper)plane with small margin (a), or bigger margin (b).

Case (b) promises a better robustness when classifying out-of-sample points.



Maximize margin

The optimal hyperplane is the one that maximizes its margin, i.e., the distance between itself and the nearest point to classify.

The optimal hyperplane permits to classify correctly all points into the two classes $y_i \in \{-1,1\}$ using $\boldsymbol{w}$ of smallest norm of.

The problem can be formulated as follows:

Minimize

$$\Phi(\boldsymbol{w}) = ½ \|\boldsymbol{w}\|^2 \qquad (1)$$

with $\boldsymbol{w},b$ subject to

$$y_i\,(\boldsymbol{w} \cdot x_i + b) \geq 1, \quad i = 1, \ldots, l \qquad (2)$$

Lagrangian reformulation

This problem can be solved by means of Lagrangian multipliers, introducing one multiplier for each constraint (2)

Tfe Lagrangian function is:

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\Lambda}) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{l} \lambda_i [y_i(\mathbf{w}\cdot\mathbf{x} + b) - 1] \qquad (3)$$

where $\Lambda=(\lambda_1\ldots\lambda_l)$ is the lagrangian multiplier vector, they must be non negative since deriving from inequelity constraints (2).

This Lagrangian function must be minimized w.r.t. $\boldsymbol{w}$ and b and maximized w.r.t. $\Lambda \geq \mathbf{0}$ (saddle point).

Lagrangian reformulation

Upon differencing (3) and applying KKT 1 we get:

$$\frac{\partial \mathcal{L}(\mathbf{w}, b, \boldsymbol{\Lambda})}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{l} \lambda_i y_i \mathbf{x}_i = 0 \qquad (4)$$

$$\frac{\partial \mathcal{L}(\mathbf{w}, b, \boldsymbol{\Lambda})}{\partial b} = \sum_{i=1}^{l} \lambda_i y_i = 0 \qquad (5)$$

From (4) we get:

$$\mathbf{w}^* = \sum_{i=1}^{l} \lambda_i^* y_i \mathbf{x}_i \qquad (6)$$

Quadratic objective

The optimal hyperplane can be obtained by a linear combination of the points to separate:

$$w^* = \sum_i \lambda_i^* y_i x_i = \Lambda^* yx$$

$$f(x) = w^* x + b = \Lambda^* yxx + b$$

The function to optimize is a *quadratic function*.

Lagrangian problem

The Lagrangian problem (3) can now be reformulated in this simpler form:

max

$$F(\mathbf{\Lambda}) = \sum_{i=1}^{l} \lambda_i - \frac{1}{2}\left\|\mathbf{w}^*\right\|^2 = \sum_{i=1}^{l} \lambda_i - \frac{1}{2}\sum_{i,j=1}^{l} \lambda_i \lambda_j y_i y_j \mathbf{x}_i \mathbf{x}_j \qquad (7)$$

Subject to

$$\lambda_i \geq 0 \qquad i = 1 \ldots l \qquad (8)$$

$$\sum_{i=1}^{l} \lambda_i y_i = 0 \qquad (9 = 5)$$

Support vectors

The classifier is finally given by:

$$f(\mathbf{x}) = sign\left(\sum_{i=1}^{l} y_i \lambda_i^* (\mathbf{x} \cdot \mathbf{x}_i) + b^*\right)$$

In the solution, all points x_i having the corresponding multiplier λ_i strictly positive are named support vectors.

All other points x_i have the corresponding $\lambda_i = 0$ and have no effect on the classifier.

The support vectors are the points closest to the separation hyperplane; if all other points were removed and learning re-run the result would be exactly the same.

Projects

From not-so-easy to complex.

- Numpy implementation of the described theory.
- Use of described linear SVM for regression.
- Introduction of a kernel function (only for those already knowledgeable of the topic).