Annotated Version

Machine Learning Course - CS-433

Regression

Sept 15, 2020

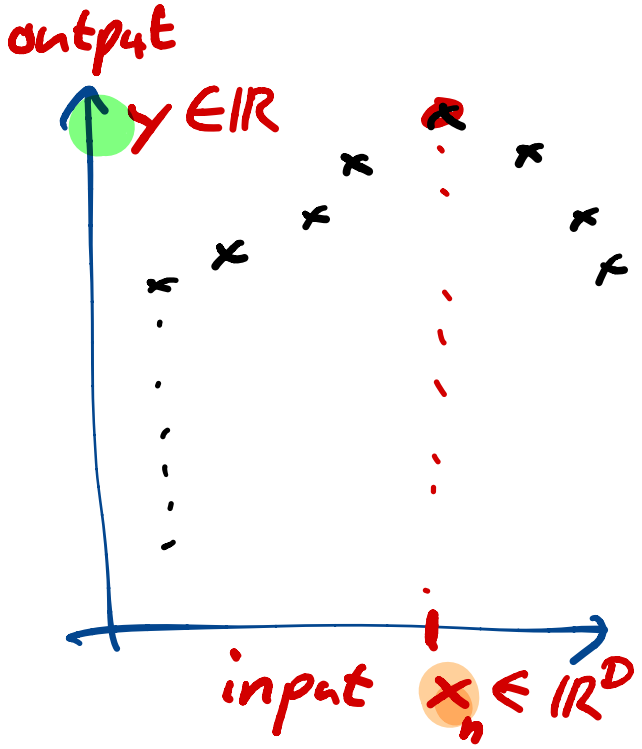
minor changes by Martin Jaggi 2020,2019,2018,2017,2016;

Last updated on: September 14, 2020

EPFL

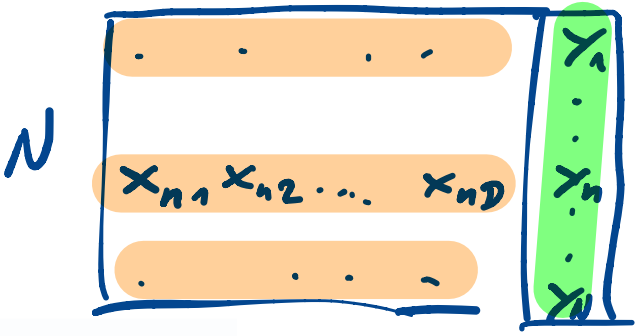
What is regression?

Regression is to relate input variables to the output variable, to either predict outputs for new inputs and/or to understand the effect of the input on the output.



$$\text{Data} = \{(x_n, y_n)\}_{n=1}^{N}$$

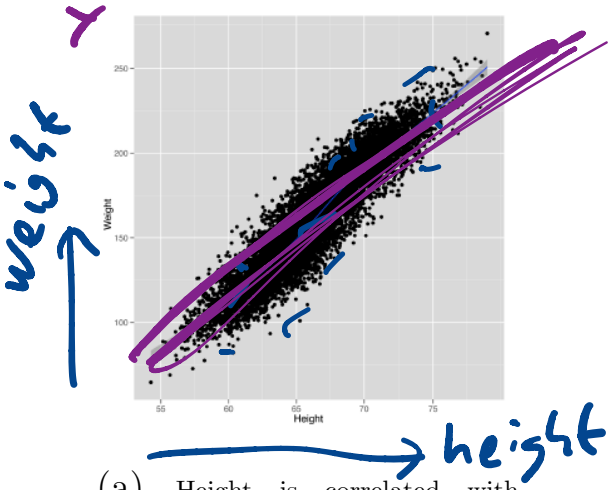
$x_n \in \mathbb{R}^D$, $y_n \in \mathbb{R}$



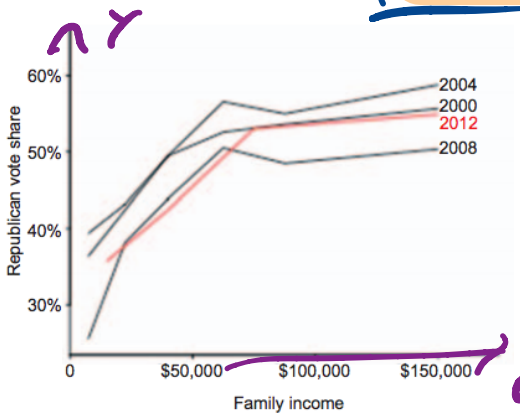
Dataset for regression

In regression, data consists of pairs $(\mathbf{x}_n, y_n)$, where y_n is the n'th output and $\mathbf{x}_n$ is a vector of D inputs. The number of pairs N is the data-size and D is the dimensionality.

Examples of regression

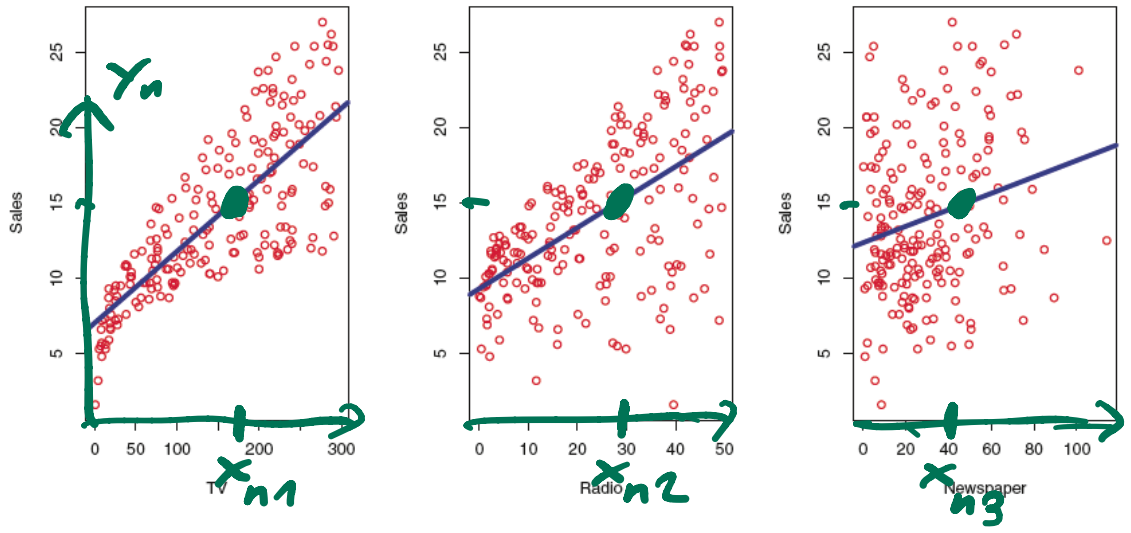


(a) Height is correlated with weight. Taken from "Machine Learning for Hackers"



(b) Do rich people vote for republicans? Taken from Avi Feller et. al. 2013, Red state/blue state in 2012 elections.

one-dimensional
D = 1



(c) How does advertisement in TV, radio, and newspaper affect sales? Taken from the book "An Introduction to statistical learning"

Two goals of regression

In prediction, we wish to predict the output for a new input vector, e.g. what is the weight of a person who is 170 cm tall?

In interpretation, we wish to understand the effect of inputs on output, e.g. are taller people heavier too?

The regression function

For both the goals, we need to find a function that approximates the output "well enough" given inputs.

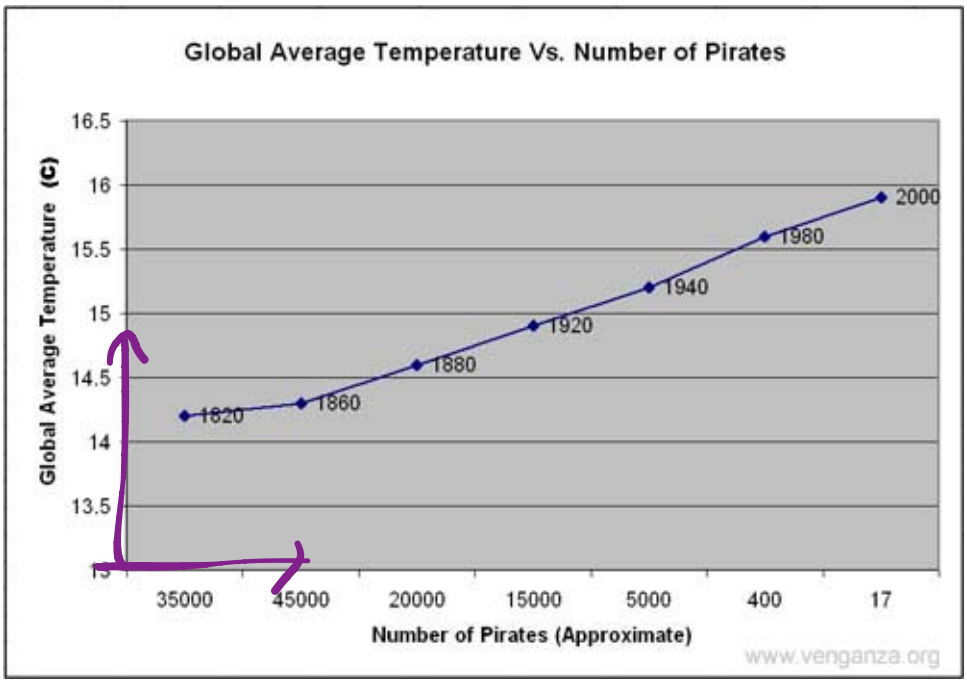
$$y_n \approx f(\mathbf{x}_n), \text{ for all } n$$

model, parameterized by $\mathbf{w}$

Additional Notes

Correlation $\neq$ Causation

Regression finds correlation not a causal relationship, so interpret your results with caution.



This image is taken from www.venganza.org. You can see many more examples at this page: Spurious correlations page.

Machine Learning Jargon for Regression

X_n *Input variables* are also known as features, covariates, independent variables, explanatory variables, exogenous variables, predictors, regressors.

Y_n *Output variables* are also known as target, label, response, outcome, dependent variable, endogenous variables, measured variable, regressands.

Prediction vs Interpretation

Some questions to think about: are these prediction tasks or interpretation task?

1. What is the life-expectancy of a person who has been smoking for 10 years?
2. Does smoking cause cancer?
3. When the number of packs a smoker smokes per day doubles, their life span gets cut in half?
4. A massive scale earthquake will occur in California within next 30 years.
5. More than 300 bird species in north America could reduce their habitat by half or more by 2080.

annotated version

Machine Learning Course - CS-433

Linear Regression

Sept 15, 2020

minor changes by Martin Jaggi 2020,2019,2018,2017,2016;

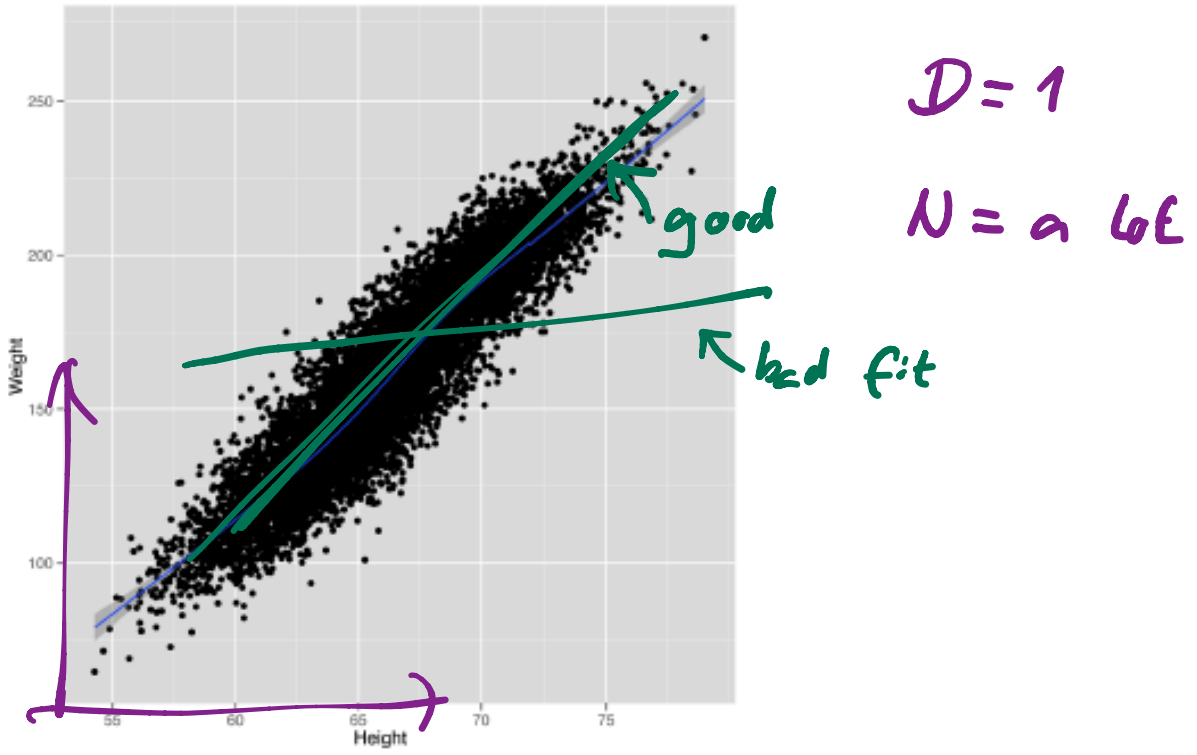
Last updated on: September 14, 2020

EPFL

1 Model: Linear Regression

What is it?

Linear regression is a model that assumes a linear relationship between inputs and the output.



Why learn about linear regression?

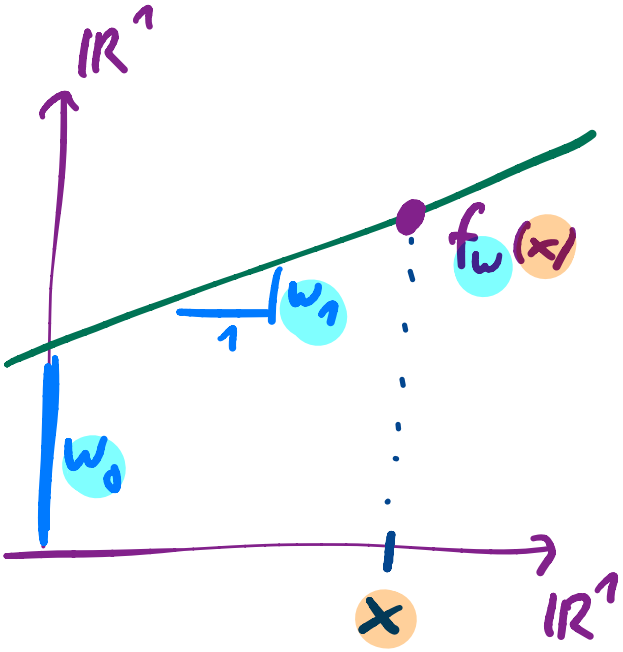
Plenty of reasons: simple, easy to understand, most widely used, easily generalized to non-linear models. Most importantly, you can learn almost all fundamental concepts of ML with regression alone.

Simple linear regression

With only one input dimension, we get simple linear regression.

$$y_n \approx f(\mathbf{x}_n) := w_0 + w_1 x_{n1}$$

Here, $\mathbf{w} = (w_0, w_1)$ are the two parameters of the model. They describe f.



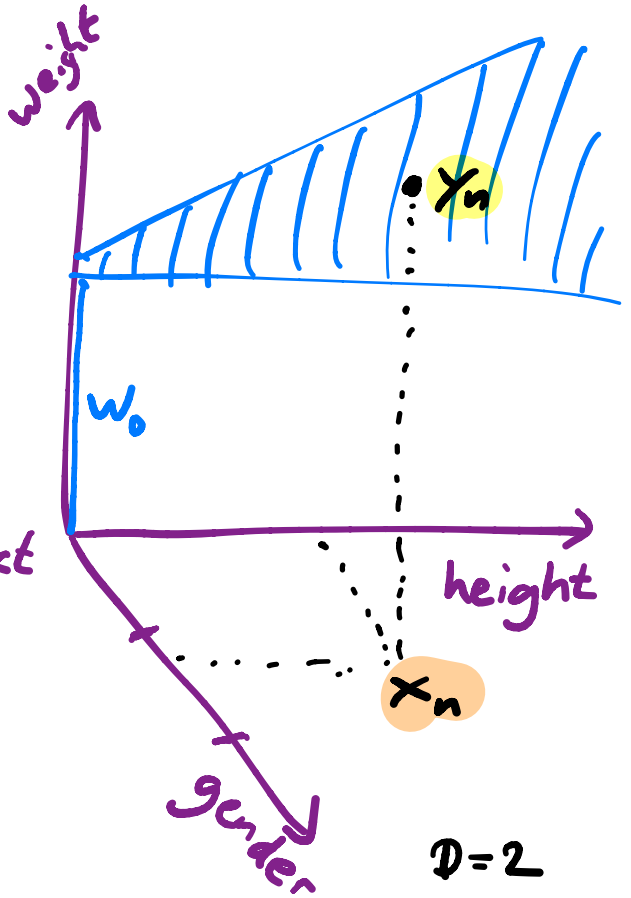
Multiple linear regression

If our data has multiple input dimensions, we obtain multivariate linear regression.

$$\begin{aligned} y_n \approx f(\mathbf{x}_n) &:= w_0 + w_1 x_{n1} + \ldots + w_D x_{nD} \\ &= w_0 + \mathbf{x}_n^\top \begin{pmatrix} w_1 \\ \vdots \\ w_D \end{pmatrix} \\ &=: \tilde{\mathbf{x}}_n^\top \tilde{\mathbf{w}} \end{aligned}$$

inner product

Note that we add a tilde over the input vector, and also the weights, to indicate they now contain the additional offset term (a.k.a. bias term).



$$\tilde{\mathbf{x}}_n := \begin{pmatrix} 1 \\ \mathbf{x}_n \end{pmatrix} = \begin{pmatrix} 1 \\ x_{n1} \\ \vdots \\ x_{nD} \end{pmatrix} \in \mathbb{R}^{D+1}$$

$\tilde{\mathbf{w}}$

Learning / Estimation / Fitting

Given data, we would like to find $\tilde{\mathbf{w}} = [w_0, w_1, \ldots, w_D]$. This is called learning or estimating the parameters or fitting the model.
To do so, we need an optimization algorithm, which we will discuss in the chapter after the next.

Additional Notes

Alternative when not using an 'offset' term

Above we have used $D + 1$ model parameters, to fit data of dimension D. An alternative also often used in practice, in particular for high-dimensional data, is to ignore the offset term w_0.

$$\begin{aligned} y_n \approx f(\mathbf{x}_n) &:= w_1 x_{n1} + \ldots + w_D x_{nD} \\ &= \mathbf{x}_n^\top \mathbf{w} \end{aligned}$$

in this case, we have just D parameters to learn, instead of $D + 1$.

As a warning, you should be aware that for some machine learning models, the number of weight parameters (elements of $\mathbf{w}$) can in general be very different from D (being the dimension of the input data). For an ML model where they do not coincide, think for example of a neural network (more details later).

Matrix multiplication

To go any further, one must revise matrix multiplication. Remember that multiplication of $M \times N$ matrix with a $N \times D$ matrix results in a $M \times D$ matrix. Also, two matrices of size $M \times N_1$ and $N_2 \times M$ can only be multiplied when $N_1 = N_2$.

classic: $N > D$
modern DL: $N < D$

The $D > N$ Problem

datapoints
features

Consider the following simple situation: You have $N = 1$ and you want to fit $y_1 \approx w_0 + w_1 x_{11}$, i.e. you want to find $\mathbf{w} = (w_0, w_1)$ given one pair (y_1, x_{11}). Is it possible to find such a line?

This problem is related to something called $D > N$ problem (in statistics typically named $p > n$). It means that the number of parameters exceeds number of data examples. In other words, we have more variables than we have data information. For many models, such as linear regression, this makes the task *under-determined*. We say that the model is *over-parameterized* for the task.

deep learning

Using regularization is a way to avoid the issue described, which we will learn later.

Annotated version

Machine Learning Course - CS-433

Cost Functions

a.k.a. Loss- ..

Sept 17, 2020

minor changes by Martin Jaggi 2020,2019,2018,2017,2016; ©Mohammad Emtiyaz Khan 2015

Last updated on: September 15, 2020

EPFL

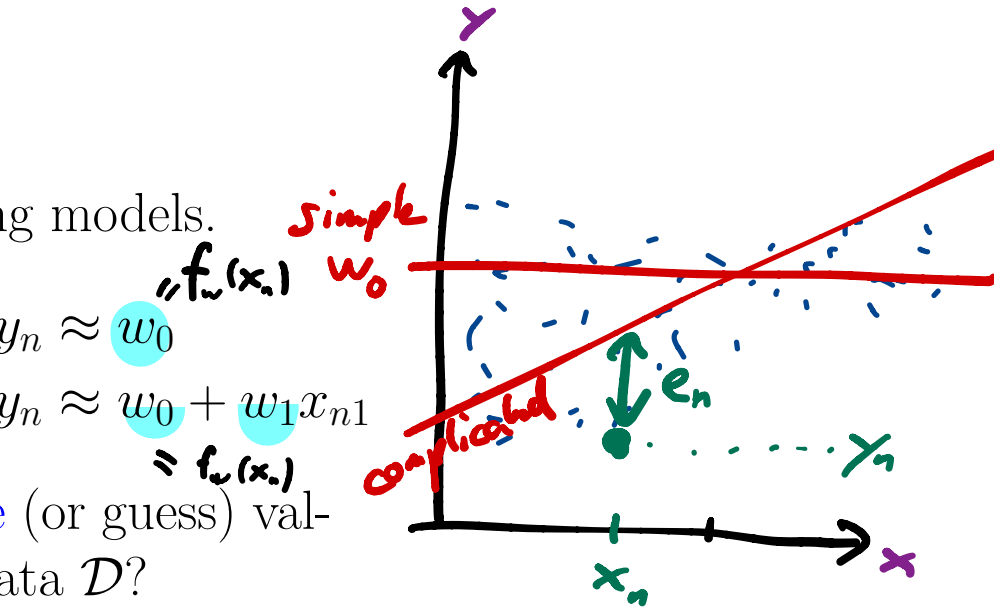
Motivation

Consider the following models.

1-parameter model: $y_n \approx w_0$

2-parameter model: $y_n \approx w_0 + w_1 x_{n1}$

How can we estimate (or guess) values of $\mathbf{w}$ given the data $\mathcal{D}$?



What is a cost function?

$e_n := y_n - f_w(x_n)$

A cost function (or energy, loss, training objective) is used to learn parameters that explain the data well. The cost function quantifies how well our model does - or in other words how costly our mistakes are.

$$\mathcal{L}(w) = \frac{1}{N} \sum_{n=1}^{N} \text{cost}(e_n)$$

Two desirable properties of cost functions

When the target y is real-valued, it is often desirable that the cost is symmetric around 0, since both positive and negative errors should be penalized equally.

Also, our cost function should penalize "large" mistakes and "very-large" mistakes similarly.

Statistical vs computational trade-off

If we want better statistical properties, then we have to give-up good computational properties.

Mean Square Error (MSE)

MSE is one of the most popular cost functions.

$$\text{MSE}(\mathbf{w}) := \frac{1}{N}\sum_{n=1}^{N}\underbrace{\left[y_n - f_{\mathbf{w}}(\mathbf{x}_n)\right]^2}_{\text{cost of point } n}$$

e_n

Does this cost function have both mentioned properties?

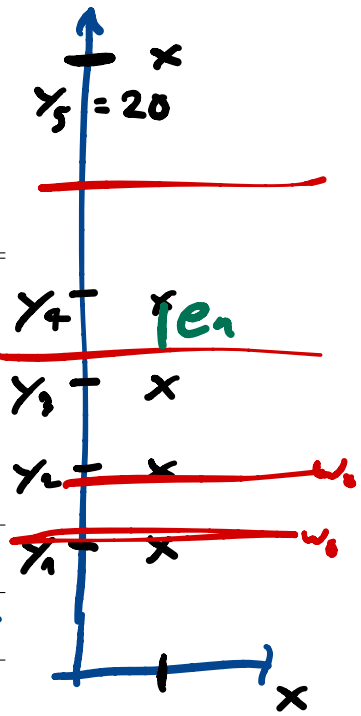
An exercise for MSE

Compute MSE for 1-param model: $f_w(x) = w_0$

$$\mathcal{L}(w_0) := \frac{1}{N}\sum_{n=1}^{N}\left[y_n - w_0\right]^2$$

"$f_w(x)$"

$w_0 =$	1	2	3	4	5	6	7
$y_1 = 1$	0^2	1^2	2^2	3^2	.	.	
$y_2 = 2$	1^2	0^2	1^2	2^2	.	.	
$y_3 = 3$	2^2	1^2	0^2	1^2	.	.	
$y_4 = 4$	3^2	2^2	1^2	0^2	.	.	
$\text{MSE}(\mathbf{w}) \cdot N$	14	6	6	14	30	54	
$y_5 = 20$	19^2	18^2	17^2	16^2	15^2	14^2	13^2
$\text{MSE}(\mathbf{w}) \cdot N$	.	.	.	.	.	250	



Some help: $19^2 = 361, 18^2 = 324, 17^2 = 289, 16^2 = 256, 15^2 = 225, 14^2 = 196, 13^2 = 169$.

best model w 4

best model with 5

Outliers

Outliers are data examples that are far away from most of the other examples. Unfortunately, they occur more often in reality than you would want them to!

MSE is not a good cost function when outliers are present.

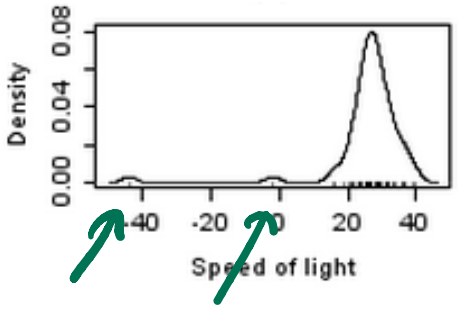
not very robust

Here is a real example on speed of light measurements

(Gelman's book on Bayesian data analysis)

28	26	33	24	34	-44	27	16	40	-2
29	22	24	21	25	30	23	29	31	19
24	20	36	32	36	28	25	21	28	29
37	25	28	26	30	32	36	26	30	22
36	23	27	27	28	27	31	27	26	33
26	32	32	24	39	28	24	25	32	25
29	27	28	29	16	23				

(a) Original speed of light data done by Simon Newcomb.



(b) Histogram showing outliers.

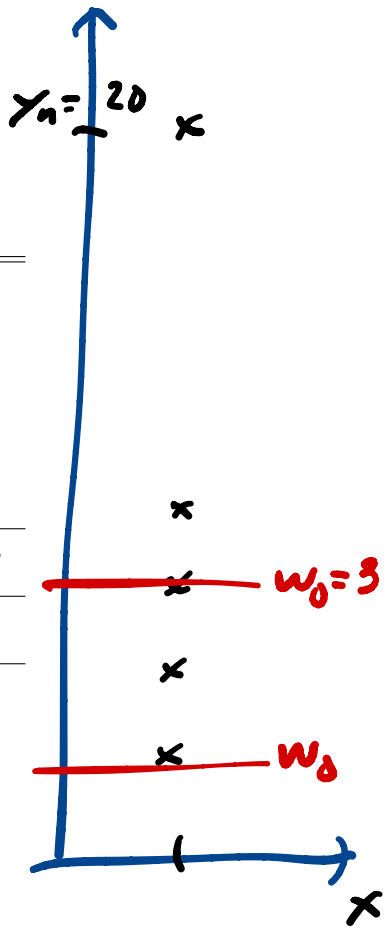
Handling outliers well is a desired *statistical* property.

Mean Absolute Error (MAE)

$$\mathrm{MAE}(\mathbf{w}) := \frac{1}{N}\sum_{n=1}^{N} |y_n - f_{\mathbf{w}}(\mathbf{x}_n)|$$

Repeat the exercise with MAE.

	$w_0 =$ 1	2	3	4	5	6	7
$y_1 = 1$	0	1	2	3			
$y_2 = 2$	1	0	1	2			
$y_3 = 3$	2	1	0	1	.		
$y_4 = 4$	3	2	1	0	.	.	.
$\mathrm{MAE}(\mathbf{w}) \cdot N$	6	4	4	6	10	14	18
$y_5 = 20$	19	18	17	16	15	14	13
$\mathrm{MAE}(\mathbf{w}) \cdot N$	25	22	21	22	25	.	.



Can you draw MSE and MAE for the above example?

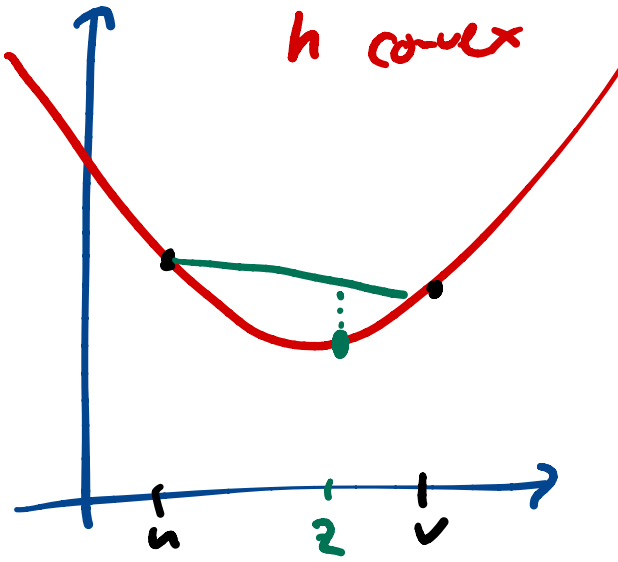
best model ~ 4 pt
$w_0^* = 2,3$

best model ~ 5 pt
$w_0^* = 3$

later:
⇒ computational advantage

Convexity

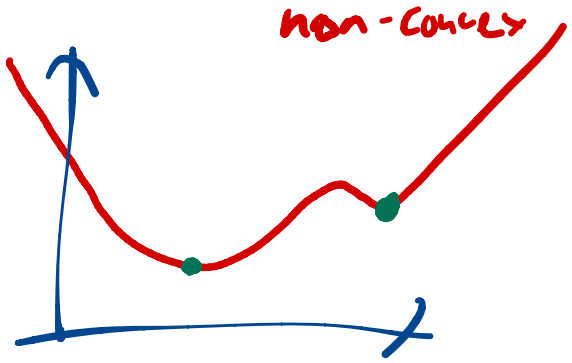
Roughly, a function is convex iff a line joining two points never intersects with the function anywhere else.



A function $h(\mathbf{u})$ with $\mathbf{u} \in \mathcal{X}$ is convex, if for any $\mathbf{u}, \mathbf{v} \in \mathcal{X}$ and for any $0 \leq \lambda \leq 1$, we have:

$$h(\lambda\mathbf{u} + (1-\lambda)\mathbf{v}) \leq \lambda h(\mathbf{u}) + (1-\lambda)h(\mathbf{v})$$

A function is strictly convex if the inequality is strict.



Importance of convexity

A strictly convex function has a unique global minimum $\mathbf{w}^\star$. For convex functions, every local minimum is a global minimum.

as a function of weights w_0

Sums of convex functions are also convex. Therefore, MSE is convex.

Convexity is a desired *computational* property.

Can you prove that the MAE is convex? (as a function of the parameters $\mathbf{w} \in \mathbb{R}^D$, for linear regression $f_{\mathbf{w}}(\mathbf{x}) := f(\mathbf{x}, \mathbf{w}) := \mathbf{x}^\top \mathbf{w}$)

Homework

$\text{cost}(y_n - f_w(x_n))$

e_n

Computational VS statistical trade-off

So which loss function is the best?

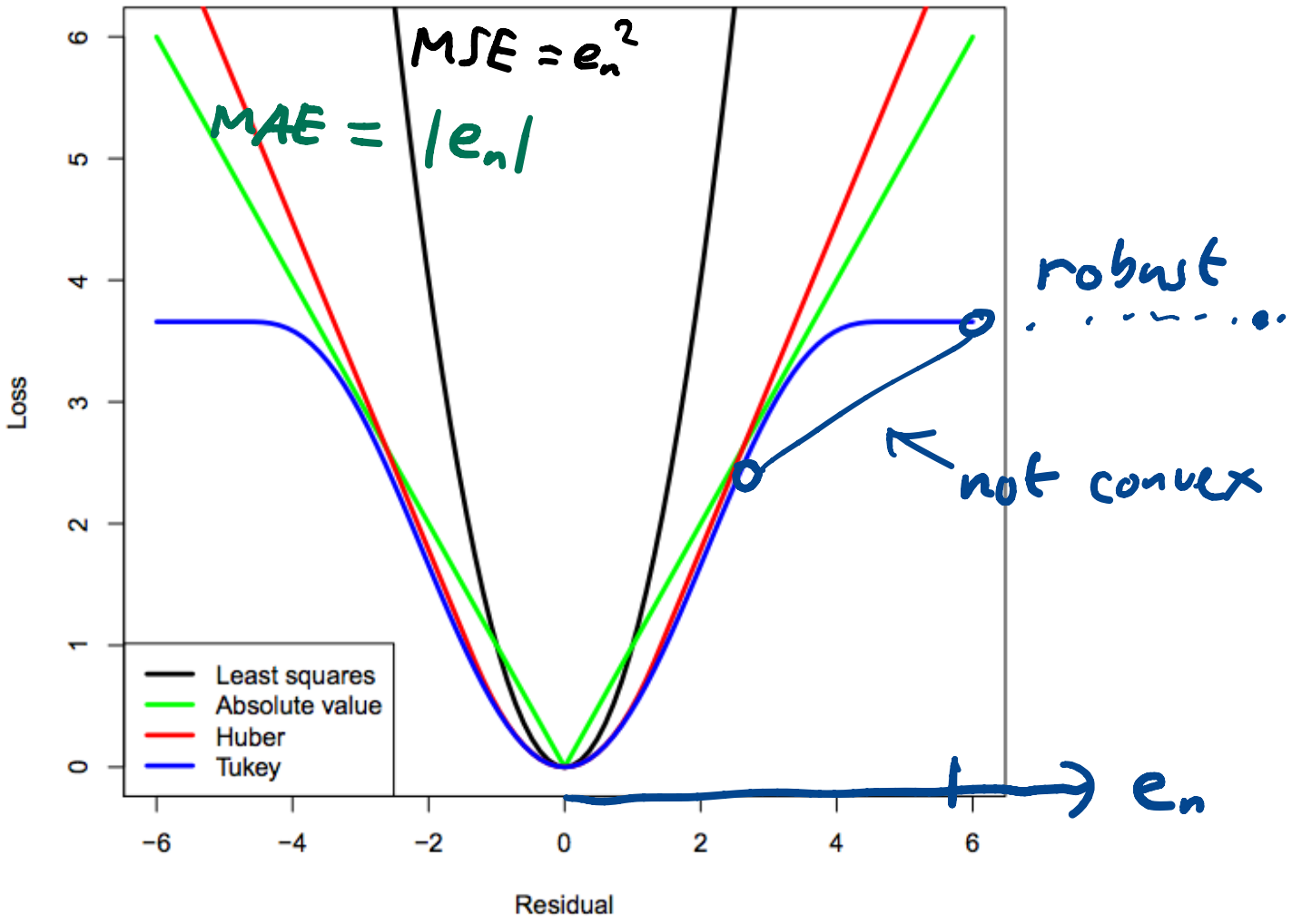


Figure taken from Patrick Breheny's slides.

If we want better statistical properties, then we have to give-up good computational properties.

Additional Reading

Other cost functions

Huber loss

$$Huber(e) := \begin{cases} \frac{1}{2}e^2 & , \text{if } |e| \leq \delta \\ \delta|e| - \frac{1}{2}\delta^2 & , \text{if } |e| > \delta \end{cases} \tag{1}$$

Huber loss is convex, differentiable, and also robust to outliers. However, setting δ is not an easy task.

Tukey's bisquare loss (defined in terms of the gradient)

$$\frac{\partial \mathcal{L}}{\partial e} := \begin{cases} e\{1 - e^2/\delta^2\}^2 & , \text{if } |e| \leq \delta \\ 0 & , \text{if } |e| > \delta \end{cases} \tag{2}$$

Tukey's loss is non-convex, but robust to outliers.

Additional reading on outliers

- Wikipedia page on "Robust statistics".
- Repeat the exercise with MAE.
- Sec 2.4 of Kevin Murphy's book for an example of robust modeling

Nasty cost functions: Visualization

See Andrej Karpathy's Tumblr post for many cost functions gone "wrong" for neural networks. http://lossfunctions.tumblr.com/.

Annotated version

Machine Learning Course - CS-433

Optimization

Sep 22+24, 2020

minor changes by Martin Jaggi 2020,2019,2018,2017;

Last updated on: September 22, 2020

EPFL

Learning / Estimation / Fitting

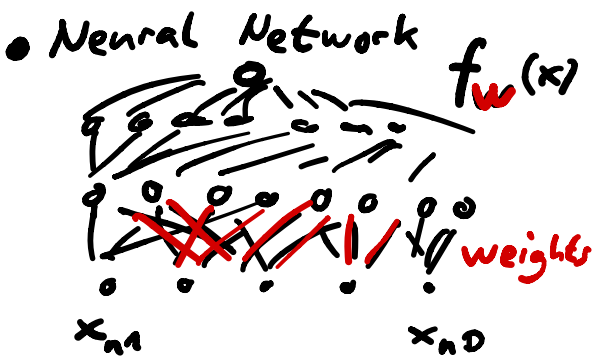
Given a cost function $\mathcal{L}(\mathbf{w})$, we wish to find $\mathbf{w}^\star$ which minimizes the cost:

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) \quad \text{subject to } \mathbf{w} \in \mathbb{R}^D$$

This means the *learning* problem is formulated as an optimization problem.
We will use an optimization algorithm to solve the problem (to find a good $\mathbf{w}$).

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(y_n - f_{\mathbf{w}}(x_n))^2$$

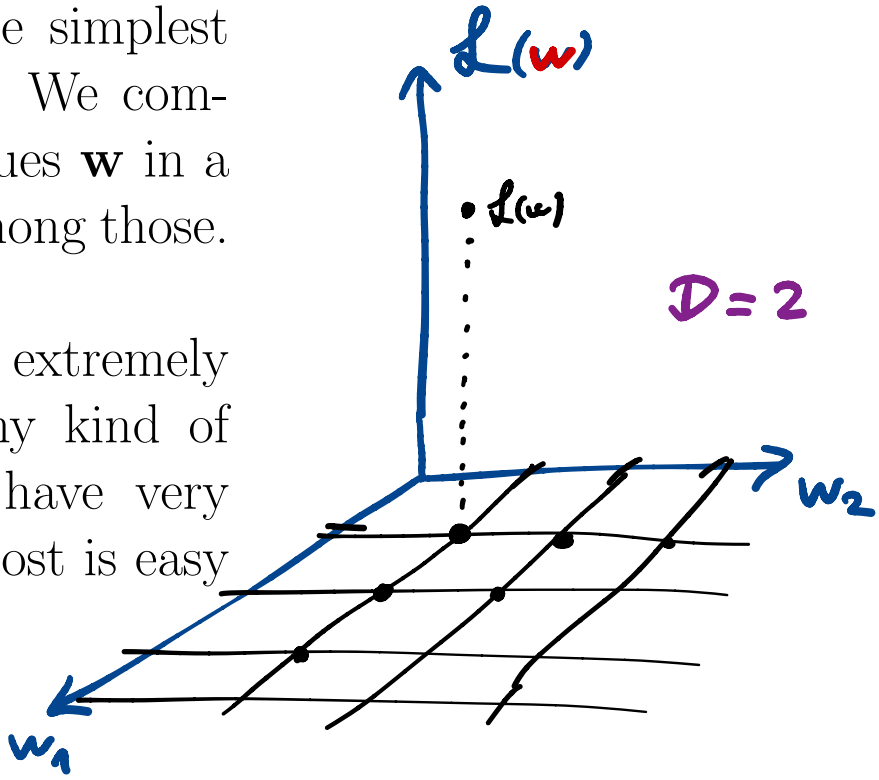
Examples
- Linear Model
$$f_{\mathbf{w}}(x) = \mathbf{w}^T x$$
- Neural Network



Grid Search

Grid search is one of the simplest optimization algorithms. We compute the cost over all values $\mathbf{w}$ in a grid, and pick the best among those.

This is brute-force, but extremely simple and works for any kind of cost function when we have very few parameters and the cost is easy to compute.



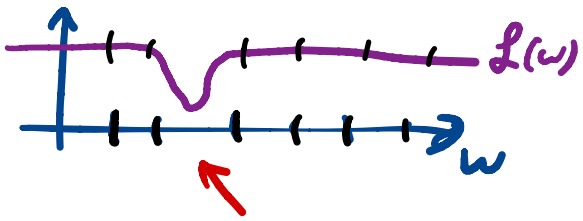
1. For a large number of parameters D, however, grid search has too many "for-loops", resulting in an exponential computational complexity:

$w \in \mathbb{R}^D$

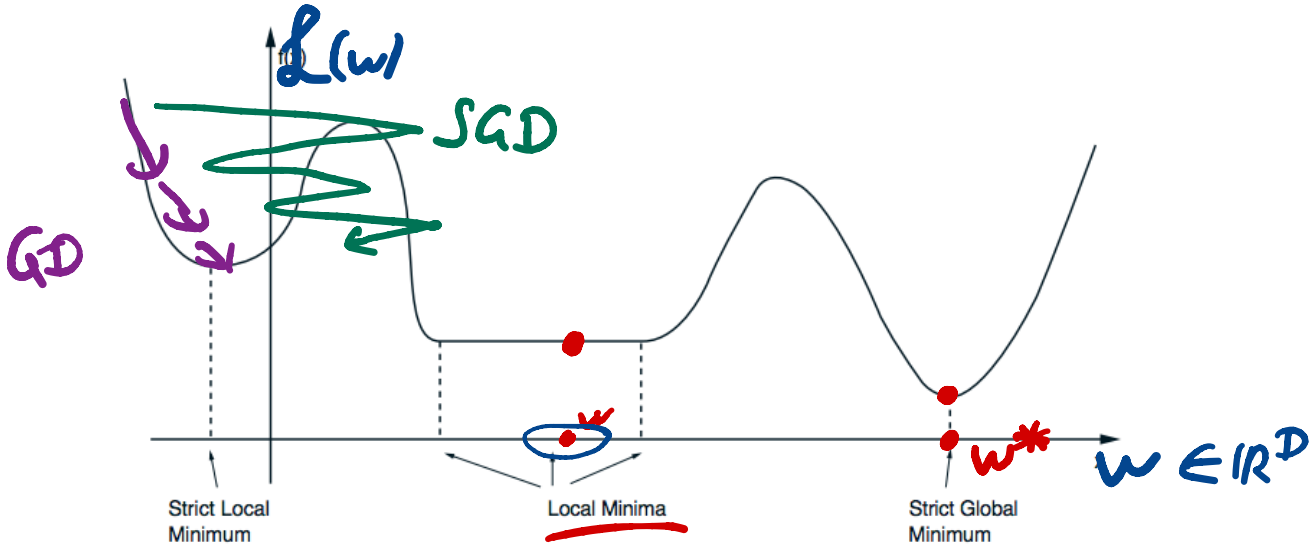
If we decide to use 10 possible values for each dimension of $\mathbf{w}$, then we have to check 10^D points. This is clearly impossible for most practical machine learning models, which can often have $D \approx$ millions of parameters. Choosing a good range of values for each dimension is another problem.

tries ($\mathcal{L}(w)$ eval) $= 10^D$

2. *Other issues:* No guarantee can be given that we end up close to an optimum.



Optimization Landscapes



The above figure is taken from Bertsekas, Nonlinear programming.

A vector $\mathbf{w}^\star$ is a local minimum of $\mathcal{L}$ if it is no worse than its neighbors; i.e. there exists an $\epsilon > 0$ such that,

$$\mathcal{L}(\mathbf{w}^\star) \leq \mathcal{L}(\mathbf{w}), \quad \forall \mathbf{w} \text{ with } \|\mathbf{w} - \mathbf{w}^\star\| < \epsilon$$

A vector $\mathbf{w}^\star$ is a global minimum of $\mathcal{L}$ if it is no worse than all others,

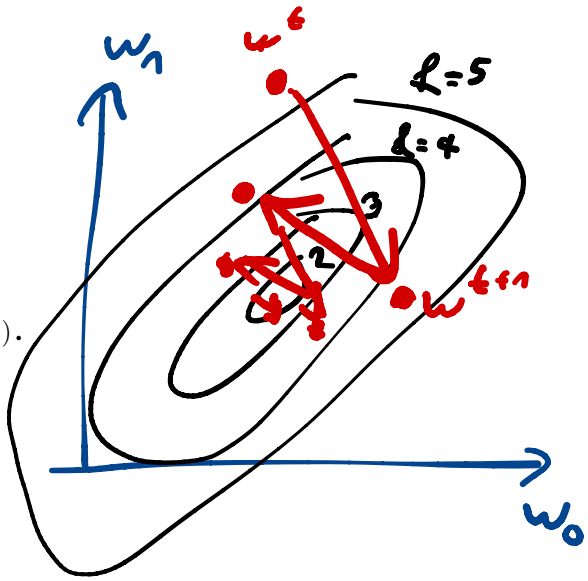
$$\mathcal{L}(\mathbf{w}^\star) \leq \mathcal{L}(\mathbf{w}), \quad \forall \mathbf{w} \in \mathbb{R}^D$$

A local or global minimum is said to be strict if the corresponding inequality is strict for $\mathbf{w} \neq \mathbf{w}^\star$.

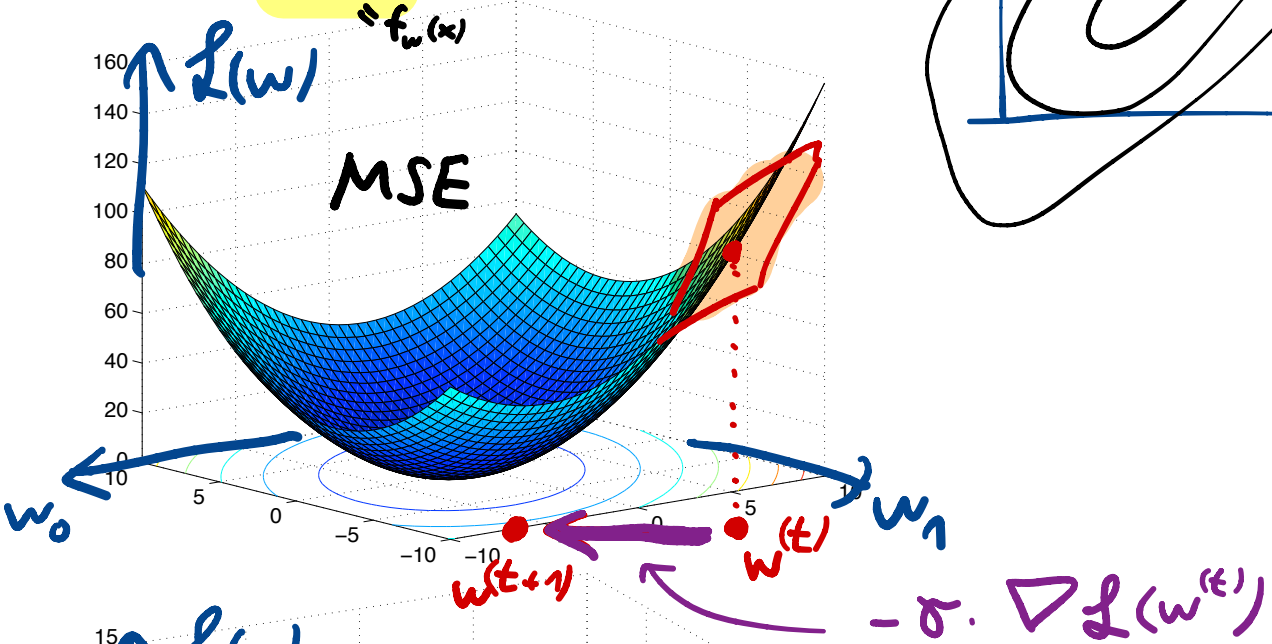
Smooth Optimization

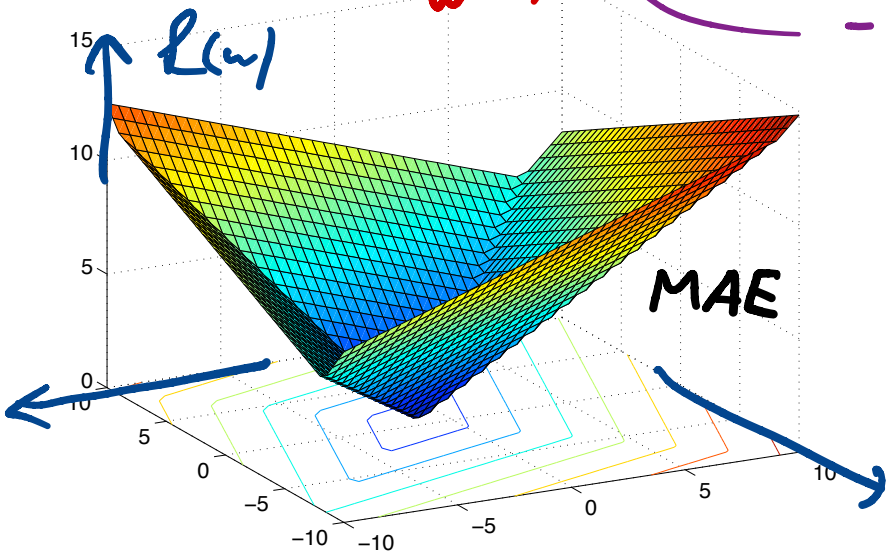
Follow the Gradient

A gradient (at a point) is the slope of the tangent to the function (at that point). It points to the direction of largest increase of the function.



For a 2-parameter model, MSE($\mathbf{w}$) and MAE($\mathbf{w}$) are shown below.
(We used $\mathbf{y}_n \approx w_0 + w_1 x_{n1}$ with $\mathbf{y}^\top = [2, -1, 1.5]$ and $\mathbf{x}^\top = [-1, 1, -1]$).





Definition of the gradient:

$$\nabla \mathcal{L}(\mathbf{w}) := \left[\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_D}\right]^\top$$

This is a vector, $\nabla \mathcal{L}(\mathbf{w}) \in \mathbb{R}^D$.

Gradient Descent

To minimize the function, we iteratively take a step in the (opposite) direction of the gradient

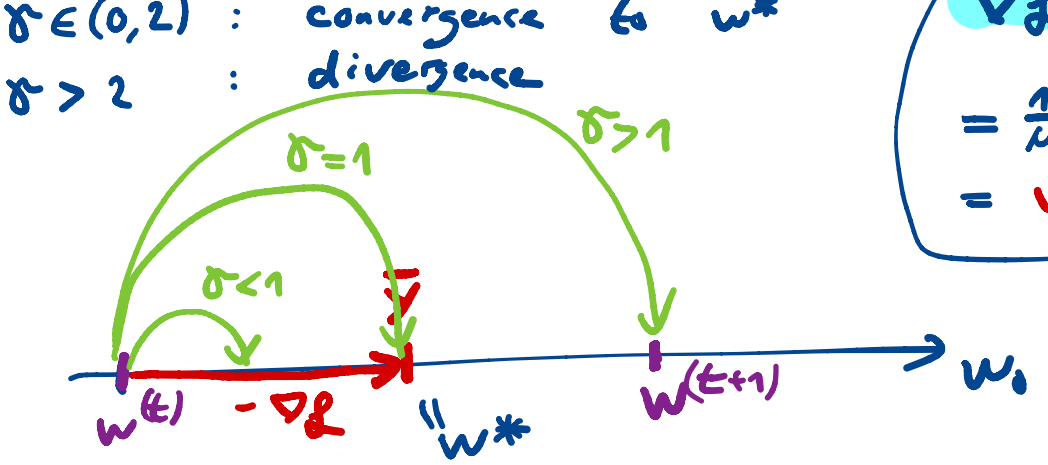
$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \gamma \nabla \mathcal{L}(\mathbf{w}^{(t)})$$

where $\gamma > 0$ is the step-size (or learning rate). Then repeat with the next t.

Example: Gradient descent for 1-parameter model to minimize MSE:

$$w_0^{(t+1)} := (1-\gamma) w_0^{(t)} + \gamma \bar{y}$$

where $\bar{y} := \sum_n y_n / N$. When is this sequence guaranteed to converge?



Gradient Descent for Linear MSE

For linear regression

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}, \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1D} \\ x_{21} & x_{22} & \dots & x_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \dots & x_{ND} \end{bmatrix}$$

← datapoint 1

We define the error vector $\mathbf{e}$:

$$\mathbf{e} = \mathbf{y} - \mathbf{X}\mathbf{w}$$

matrixcalculus.org

and MSE as follows:

$$\mathcal{L}(\mathbf{w}) := \frac{1}{2N} \sum_{n=1}^{N} (y_n - \mathbf{x}_n^\top \mathbf{w})^2$$

$f_{\mathbf{w}}(x)$

$$= \frac{1}{2N} \mathbf{e}^\top \mathbf{e}$$

$$= \frac{1}{2N} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$$

then the gradient is given by

$$\nabla \mathcal{L}(\mathbf{w}) = -\frac{1}{N} \mathbf{X}^\top \mathbf{e}$$

e_n

$$\frac{\partial}{\partial w_1} \mathcal{L} = \frac{1}{2N} \sum_{n=1}^{N} -2(y_n - \mathbf{x}_n^\top \mathbf{w}) x_{n1}$$

$$= -\frac{1}{N} (\mathbf{X}_{:1})^\top \mathbf{e}$$

$$\vdots$$

$$\frac{\partial}{\partial w_D} \mathcal{L} = -\frac{1}{N} (\mathbf{X}_{:D})^\top \mathbf{e}$$

Computational cost. What is the complexity (# operations) of computing the gradient?

a) starting from $\mathbf{w}$ and

b) given $\mathbf{e}$ and $\mathbf{w}$?

① compute $\mathbf{e}$

cost: $\mathcal{O}(N \cdot D) + \mathcal{O}(N)$

for $\mathbf{X}\mathbf{w}$ for $\mathbf{y} - \dots$

② compute $\nabla \mathcal{L}$ given $\mathbf{e}$

cost: $\mathcal{O}(N \cdot D) + \mathcal{O}(D)$

for $\mathbf{X}^\top \mathbf{e}$ for $\cdot \frac{1}{N}$

total: $\mathcal{O}(N \cdot D)$

Variant with offset. Recall: Alternative trick when also incorporating an offset term for the regression:

artificial feature per datapoint

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \qquad \widetilde{\mathbf{X}} = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1D} \\ 1 & x_{21} & x_{22} & \dots & x_{2D} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & x_{N2} & \dots & x_{ND} \end{bmatrix}$$

Exercise: compute $\nabla \mathcal{L}(w)$

$w = (w_0, w_1, \dots, w_D)$

Stochastic Gradient Descent

Sum Objectives. In machine learning, most cost functions are formulated as a sum over the training examples, that is

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}_n(\mathbf{w}) \,,$$

$\mathcal{L}_n$: cost of one datapoint n

where $\mathcal{L}_n$ is the cost contributed by the n-th training example.

Q: What are the $\mathcal{L}_n$ for linear MSE?

The SGD Algorithm. The stochastic gradient descent (SGD) algorithm is given by the following update rule, at step t:

1. Sample one datapoint $n \in \{1, \dots, N\}$ uniformly at random
2. $\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \gamma \nabla \mathcal{L}_n(\mathbf{w}^{(t)}) \,.$

g „stochastic gradient"

Theoretical Motivation. *Idea:* Cheap but unbiased estimate of the gradient!
In expectation over the random choice of n, we have

$$\mathbb{E}\left[\nabla \mathcal{L}_n(\mathbf{w})\right] = \nabla \mathcal{L}(\mathbf{w})$$

which is the true gradient direction. (check!)

$$\mathbb{E}\left[\nabla \mathcal{L}_n(\mathbf{w})\right] = \frac{1}{N}\sum_{n=1}^{N} \nabla \mathcal{L}_n(\mathbf{w}) = \nabla \frac{1}{N}\sum_n \mathcal{L}_n = \nabla \mathcal{L}(\mathbf{w}) = \mathbf{g}$$

$\Rightarrow$ unbiased

Mini-batch SGD. There is an intermediate version, using the update direction being

$$\mathbf{g} := \frac{1}{|B|}\sum_{n\in B} \nabla \mathcal{L}_n(\mathbf{w}^{(t)})$$

again with

$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \gamma \mathbf{g}.$$

In the above gradient computation, we have randomly chosen a subset $B \subseteq [N]$ of the training examples. For each of these selected examples n, we compute the respective gradient $\nabla \mathcal{L}_n$, at the same current point $\mathbf{w}^{(t)}$.

Randomly select $|B|$ datapoints $B \subseteq \{1, \dots N\}$

Examples:

- $B = \{n\}$ $\quad |B| = 1$
  $\Rightarrow$ SGD
- $B = \{1, 2, \dots, N\}$
  $\Rightarrow$ (full) gradient descent
- $|B| = 5$
  $\Rightarrow$ mini-batch SGD

The computation of $\mathbf{g}$ can be parallelized easily. This is how current deep-learning applications utilize GPUs (by running over $|B|$ threads in parallel).
Note that in the extreme case $B := [N]$, we obtain (batch) gradient descent, i.e. $\mathbf{g} = \nabla \mathcal{L}$.

SGD for Linear MSE

See Exercise Sheet 2.

Computational cost. For linear MSE, what is the complexity (# operations) of computing the stochastic gradient?
(using only $|B| = 1$ data examples)

a) starting from $\mathbf{w}$?

GD: $\mathcal{L}(w) = \frac{1}{N} \sum_{n=1}^{N} \frac{1}{2}(y_n - x_n^T w)^2$

$D\mathcal{L}(w) = -\frac{1}{N} X^T (y - Xw)$

$e \in \mathbb{R}^N$

cost: $\mathcal{O}(N \cdot D)$

SGD: $\mathcal{L}_n(w) = \frac{1}{2}(y_n - x_n^T w)^2$

$D\mathcal{L}_n(w) = -x_n^T (y_n - x_n^T w)$

$e_n \in \mathbb{R}$

cost: $\mathcal{O}(D)$

per iteration

N times cheaper

Non-differentiable Optimization

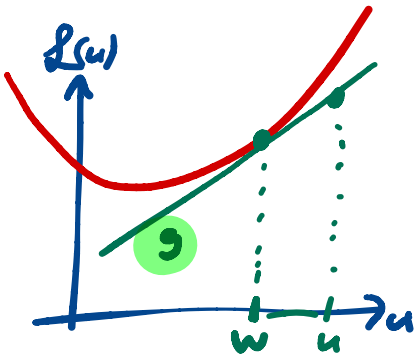
Non-Smooth Optimization

An alternative characterization of *convexity*, for differentiable functions is given by

$$\mathcal{L}(\mathbf{u}) \geq \mathcal{L}(\mathbf{w}) + \nabla\mathcal{L}(\mathbf{w})^\top(\mathbf{u}-\mathbf{w}) \quad \forall \mathbf{u}, \mathbf{w}$$

1st order Taylor

meaning that the function must always lie above its linearization.

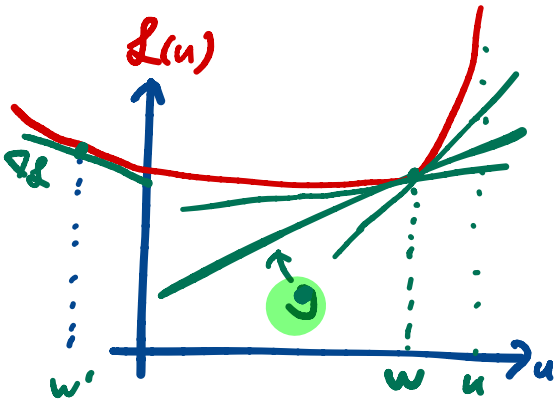


Subgradients

A vector $\mathbf{g} \in \mathbb{R}^D$ such that

$$\mathcal{L}(\mathbf{u}) \geq \mathcal{L}(\mathbf{w}) + \mathbf{g}^\top(\mathbf{u}-\mathbf{w}) \quad \forall \mathbf{u}$$

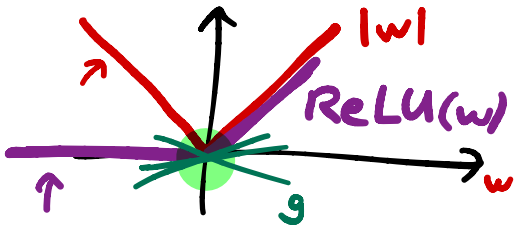
is called a subgradient to the function $\mathcal{L}$ at $\mathbf{w}$.



This definition makes sense for objectives $\mathcal{L}$ which are not necessarily differentiable (and not even necessarily convex).

If $\mathcal{L}$ is convex and differentiable at $\mathbf{w}$, then the only subgradient at $\mathbf{w}$ is $\mathbf{g} = \nabla\mathcal{L}(\mathbf{w})$.

Subgradient Descent

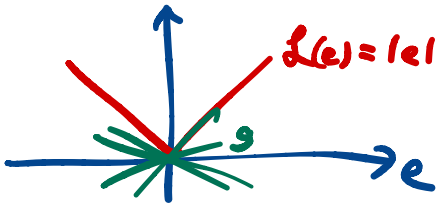


Identical to the gradient descent algorithm, but using a subgradient instead of gradient. Update rule

$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \gamma\, \mathbf{g}$$

for $\mathbf{g}$ being a subgradient to $\mathcal{L}$ at the current iterate $\mathbf{w}^{(t)}$.

Example: Optimizing Linear MAE



1. Compute a subgradient of the absolute value function

   $h : \mathbb{R} \to \mathbb{R}\,,\ h(e) := |e|.$

   toy example

   $\mathbb{R}$ ∋ w ↦ $g \in \begin{cases} -1 & \text{if } e<0 \\ [-1,1] & \text{if } e=0 \\ 1 & \text{if } e>0 \end{cases}$ subgradients at e

2. Recall the definition of the mean absolute error:

   $$\mathcal{L}(\mathbf{w}) = \mathrm{MAE}(\mathbf{w}) := \frac{1}{N}\sum_{n=1}^{N} |y_n - f_{\mathbf{w}}(\mathbf{x}_n)|$$

   For linear regression, its (sub)gradient is easy to compute using the chain rule. Compute it!

   for subgradients

$\mathcal{L}(w) = h(q(w))$

h: non-differentiable

q: differentiable

⇒ subgradient of $\mathcal{L}(w)$

$g \in \partial h(q(w)) \cdot \nabla q(w)$

set of subgradients of h at $q(\cdot)$

See Exercise Sheet 2.

Stochastic Subgradient Descent

Stochastic SubGradient Descent (still abbreviated SGD commonly).

$\mathcal{L}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}_n(\mathbf{w})$

possibly non-differentiable

Same, $\mathbf{g}$ being a subgradient to the randomly selected $\mathcal{L}_n$ at the current iterate $\mathbf{w}^{(t)}$.

Exercise: Compute the SGD update for linear MAE.

for linear models

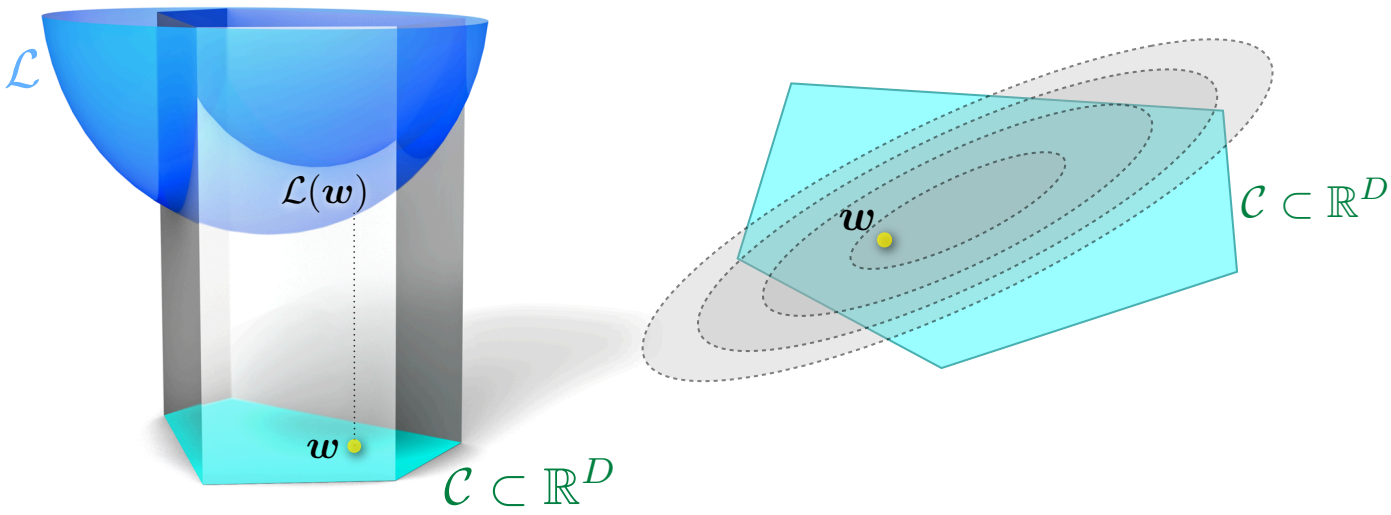
computational cost	smooth MSE	non-smooth MAE
GD $\mathcal{L}$	gradient $\nabla\mathcal{L}$ $\mathcal{O}(N \cdot D)$	subgradient of $\mathcal{L}$ $\mathcal{O}(N \cdot D)$
SGD $\mathcal{L}_n$	stoch. grad. $\nabla\mathcal{L}_n$ $\mathcal{O}(D)$	stoch. subgradient of $\mathcal{L}_n$ $\mathcal{O}(D)$

Constrained Optimization

Sometimes, optimization problems come posed with additional constraints:

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}), \quad \text{subject to } \mathbf{w} \in \mathcal{C}.$$

The set $\mathcal{C} \subset \mathbb{R}^D$ is called the constraint set.



Solving Constrained Optimization Problems

A) Projected Gradient Descent

B) Transform it into an *unconstrained* problem

Convex Sets

A set $\mathcal{C}$ is convex *iff* the line segment between any two points of $\mathcal{C}$ lies in $\mathcal{C}$, i.e., if for any $\mathbf{u}, \mathbf{v} \in \mathcal{C}$ and any θ with $0 \leq \theta \leq 1$, we have

$$\theta\mathbf{u} + (1-\theta)\mathbf{v} \in \mathcal{C}.$$

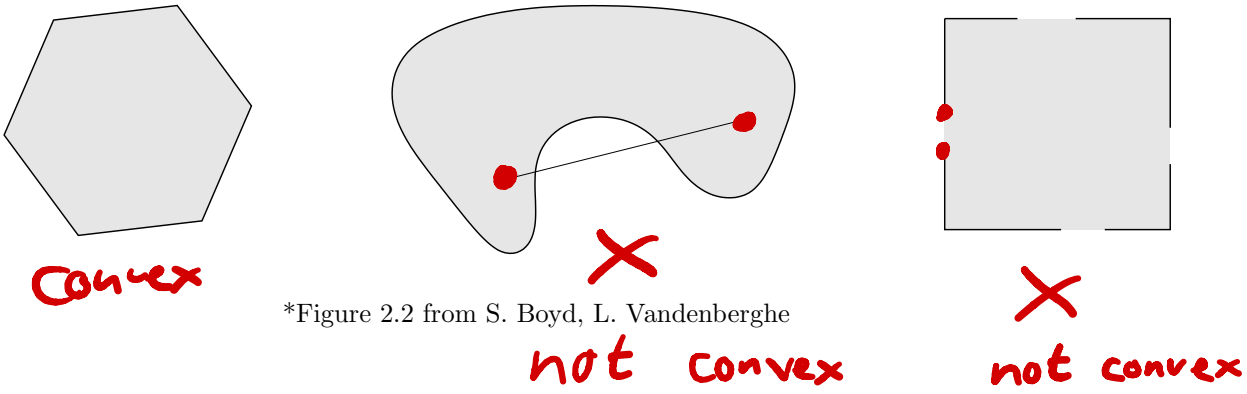


*Figure 2.2 from S. Boyd, L. Vandenberghe

Properties of Convex Sets

- Intersections of convex sets are convex
- Projections onto convex sets are *unique*. (and often efficient to compute)
  Formal definition:
  $P_{\mathcal{C}}(\mathbf{w}') := \arg\min_{\mathbf{v}\in\mathcal{C}} \|\mathbf{v} - \mathbf{w}'\|$.

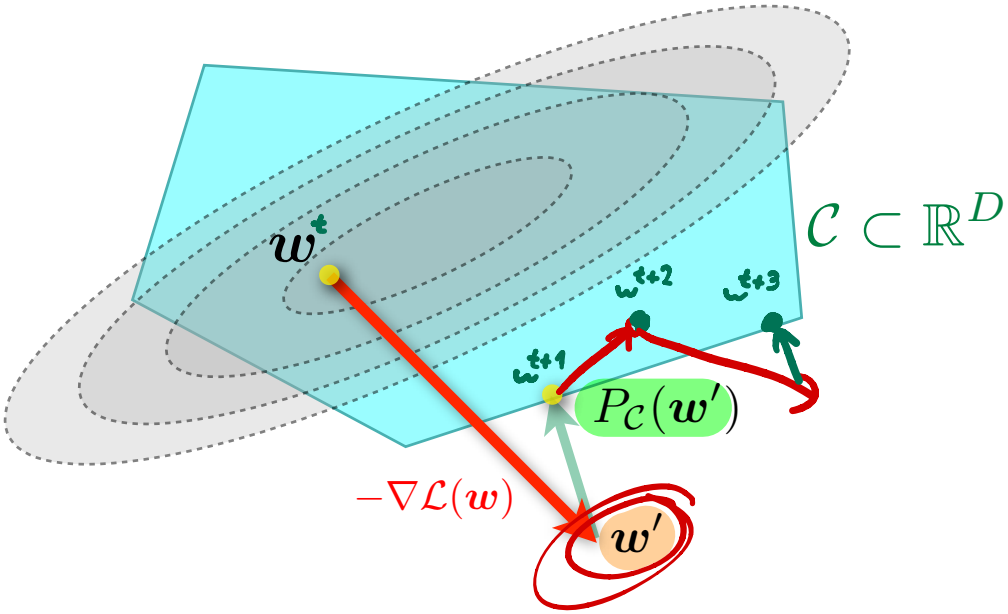
Projected Gradient Descent

Idea: add a projection onto $\mathcal{C}$ after every step:

$$P_{\mathcal{C}}(\mathbf{w}') := \arg\min_{\mathbf{v}\in\mathcal{C}} \|\mathbf{v} - \mathbf{w}'\| \ .$$

Update rule:

$$\mathbf{w}^{(t+1)} := P_{\mathcal{C}}\left[\mathbf{w}^{(t)} - \gamma\, \nabla\mathcal{L}(\mathbf{w}^{(t)})\right] \ .$$



Projected SGD. Same SGD step, followed by the projection step, as above. Same convergence properties.

Computational cost of projection? Crucial!

depends on geometry of set $\mathcal{C}$

2

Turning Constrained into Unconstrained Problems

(Alternatives to projected gradient methods)

Use penalty functions instead of directly solving $\min_{\mathbf{w}\in\mathcal{C}} \mathcal{L}(\mathbf{w})$.

- "*brick wall*" (indicator function)

$$I_{\mathcal{C}}(\mathbf{w}) := \begin{cases} 0 & \mathbf{w} \in \mathcal{C} \\ \infty & \mathbf{w} \notin \mathcal{C} \end{cases}$$

$$\Rightarrow \min_{\mathbf{w}\in\mathbb{R}^D} \mathcal{L}(\mathbf{w}) + I_{\mathcal{C}}(\mathbf{w})$$

(disadvantage: non-continuous objective)

penalty function

original cost

- Penalize error. *Example:*

$\mathcal{C} = \{\mathbf{w} \in \mathbb{R}^D \mid A\mathbf{w} = \mathbf{b}\}$

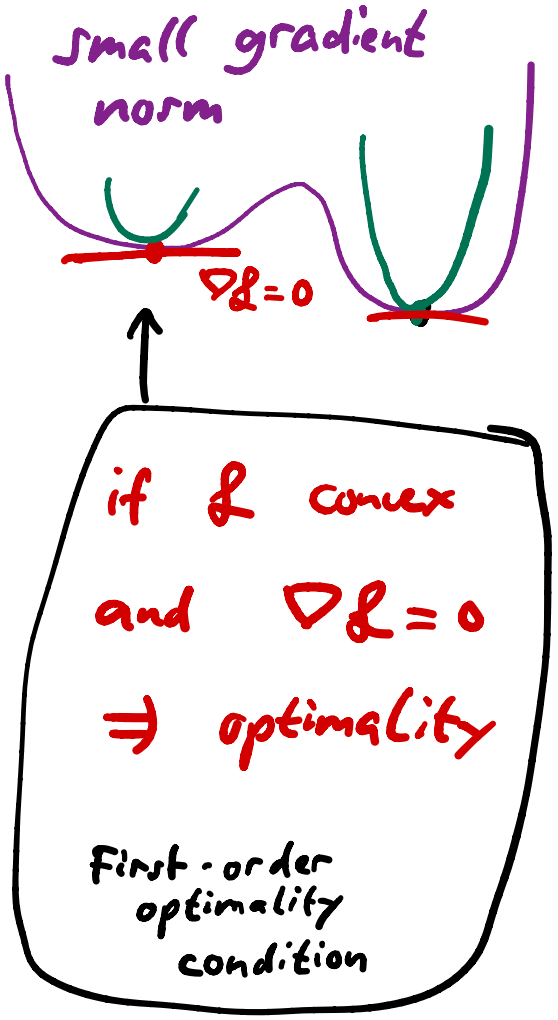
$$\Rightarrow \min_{\mathbf{w}\in\mathbb{R}^D} \mathcal{L}(\mathbf{w}) + \lambda \|A\mathbf{w} - \mathbf{b}\|^2$$

trade-off parameter λ

- Linearized Penalty Functions (see Lagrange Multipliers)

Implementation Issues

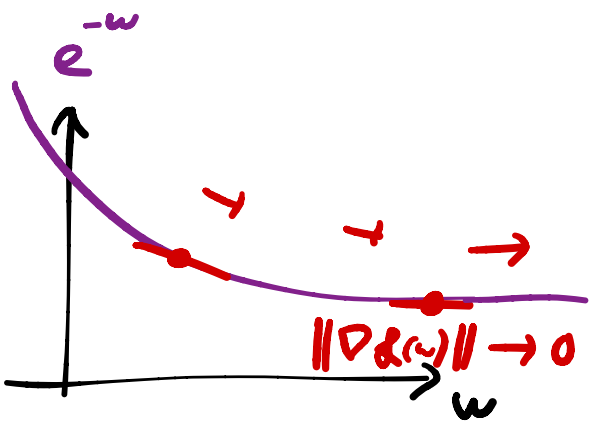
For gradient methods:



Stopping criteria: When $\|\nabla \mathcal{L}(\mathbf{w})\|$ is (close to) zero, we are (often) close to the optimum value.

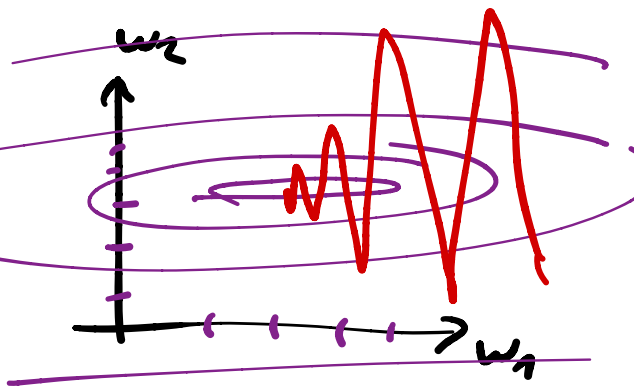
Optimality: If the second-order derivative is positive (positive semi-definite to be precise), then it is a (possibly local) minimum. If the function is also convex, then this condition implies that we are at a global optimum. See the supplementary section on Optimality Conditions.

Step-size selection: If γ is too big, the method might diverge. If it is too small, convergence is slow. Convergence to a local minimum is guaranteed only when $\gamma < \gamma_{min}$ where γ_{min} is a fixed constant that depends on the problem.



Line-search methods: For some objectives $\mathcal{L}$, we can set step-size automatically using a line-search method. More details on "back-tracking" methods can be found in Chapter 1 of Bertsekas' book on "nonlinear programming".

Feature normalization and pre-conditioning: Gradient descent is very sensitive to ill-conditioning. Therefore, it is typically advised to normalize your input features. In other words, we pre-condition the optimization problem. Without this, step-size selection is more difficult since different "directions" might converge at different speed.



Non-Convex Optimization

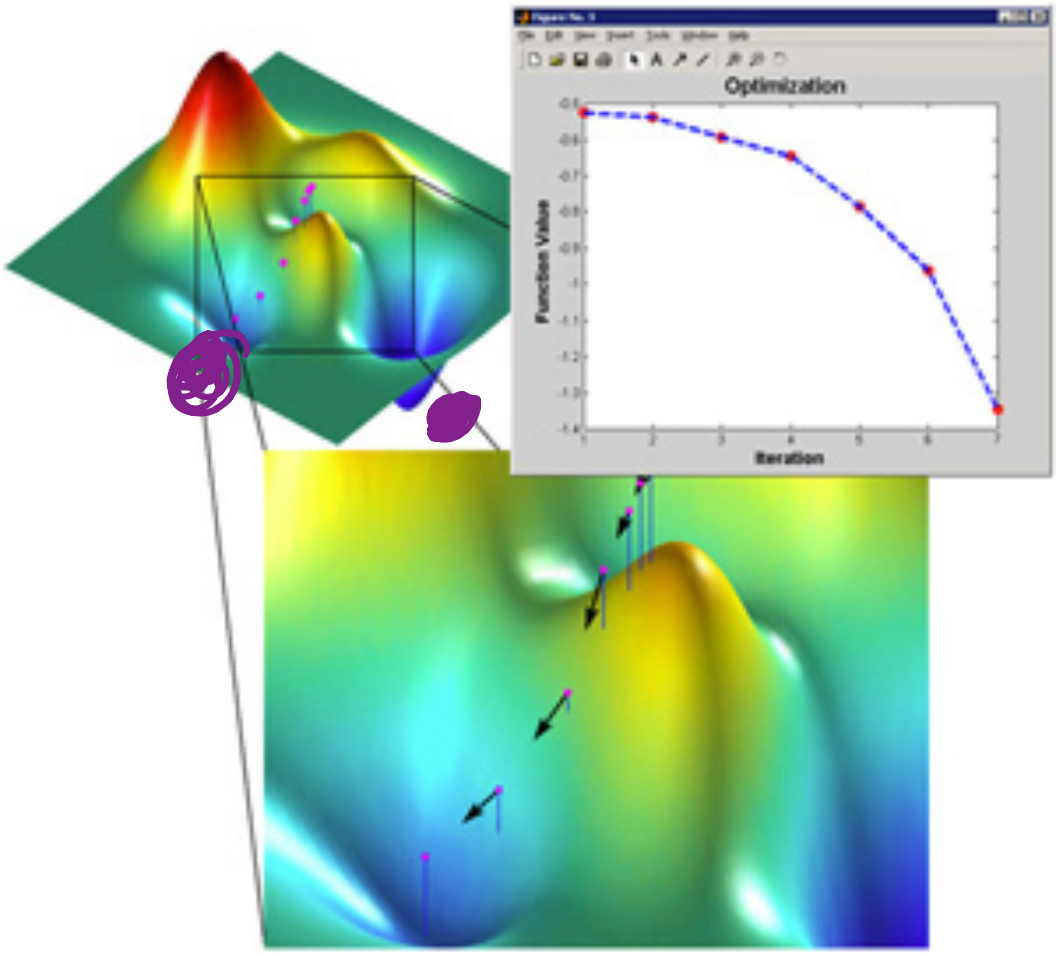


*image from mathworks.com

Real-world problems are not convex!

All we have learnt on algorithm design and performance of convex algorithms still helps us in the non-convex world.

Additional Notes

Grid Search and Hyper-Parameter Optimization

Read more about grid search and other methods for "hyperparameter" setting:
en.wikipedia.org/wiki/Hyperparameter_optimization#Grid_search.

Computational Complexity

The computation cost is expressed using the big-$\mathcal{O}$ notation. Here is a definition taken from Wikipedia. Let f and g be two functions defined on some subset of the real numbers. We write $f(x) = \mathcal{O}(g(x))$ as $x \to \infty$, if and only if there exists a positive real number c and a real number x_0 such that $|f(x)| \leq c|g(x)|, \quad \forall x > x_0$.

Please read and learn more from this page in Wikipedia:
en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations#Matrix_algebra .

- What is the computational complexity of matrix multiplication?
- What is the computational complexity of matrix-vector multiplication?

Optimality Conditions

For a *convex* optimization problem, the first-order *necessary* condition says that at *an* optimum the gradient is equal to zero.

$$\nabla \mathcal{L}(\mathbf{w}^\star) = \mathbf{0} \tag{1}$$

The second-order *sufficient* condition ensures that the optimum is a minimum (not a maximum or saddle-point) using the Hessian matrix,

which is the matrix of second derivatives:

$$\mathbf{H}(\mathbf{w}^\star) := \frac{\partial^2 \mathcal{L}(\mathbf{w}^\star)}{\partial \mathbf{w} \partial \mathbf{w}^\top} \quad \text{is positive semi-definite.} \tag{2}$$

The Hessian is also related to the convexity of a function: a twice-differentiable function is convex if and only if the Hessian is positive semi-definite at all points.

SGD Theory

As we have seen above, when N is large, choosing a random training example $(\mathbf{x}_n, y_n)$ and taking an SGD step is advantageous:

$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \gamma^{(t)} \nabla \mathcal{L}_n(\mathbf{w}^{(t)})$$

For convergence, $\gamma^{(t)} \to 0$ "appropriately". One such condition called the Robbins-Monroe condition suggests to take $\gamma^{(t)}$ such that:

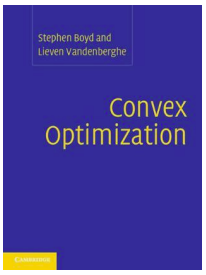
$$\sum_{t=1}^{\infty} \gamma^{(t)} = \infty, \qquad \sum_{t=1}^{\infty} (\gamma^{(t)})^2 < \infty \tag{3}$$

One way to obtain such sequences is $\gamma^{(t)} := 1/(t+1)^r$ where $r \in (0.5, 1)$.

More Optimization Theory

If you want, you can gain a deeper understanding of several optimization methods relevant for machine learning from this survey:

Convex Optimization: Algorithms and Complexity - by Sébastien Bubeck



(> 35 000 citations)

And also from the book of Boyd & Vandenberghe (both are free online PDFs)

Exercises

1. Chain-rule

   If it has been a while, familiarize yourself with it again.

2. Revise computational complexity (also see the Wikipedia link in Page 6 of lecture notes).

3. Derive the computational complexity of grid-search, gradient descent and stochastic gradient descent for linear MSE (# steps and cost per step).

4. Derive the gradients for the linear MSE and MAE cost functions.

5. Implement gradient descent and gain experience in setting the step-size.

6. Implement SGD and gain experience in setting the step-size.

annotated version

Machine Learning Course - CS-433

Least Squares

Sept 29, 2020

changes by Martin Jaggi 2020,2019,2018, changes by Rüdiger Urbanke 2017, minor changes by Martin Jaggi 2016

Last updated on: September 29, 2020

EPFL

Motivation

In rare cases, one can compute the optimum of the cost function analytically. Linear regression using a mean-squared error cost function is one such case. Here the solution can be obtained explicitly, by solving a linear system of equations. These equations are sometimes called the normal equations. This method is one of the most popular methods for data fitting. It is called least squares.

To derive the normal equations, we first show that the problem is convex. We then use the optimality conditions for convex functions (see the previous lecture notes on optimization). I.e., at the optimum parameter, call it $\mathbf{w}^\star$, it must be true that the gradient of the cost function is $\mathbf{0}$. I.e.,

$$\nabla \mathcal{L}(\mathbf{w}^\star) = \mathbf{0}.$$

This is a system of D equations.

Find w

$\min_w \mathcal{L}(w) = MSE(w)$

$= \frac{1}{N}\sum_{n=1}^{N} \frac{1}{2}(y_n - x_n^T w)^2$

Exercise: 1-param. model

$\frac{1}{N}\sum_{n=1}^{N} \frac{1}{2}(y_n - w_0)^2$

① convex in w? ✓

② $\frac{\partial \mathcal{L}}{\partial w_0} = \frac{1}{N}\sum_{n} (y_n - w_0)(-1)$

$= w_0 - \frac{1}{N}\sum_{n} y_n$

$\bar{y}$

$\stackrel{!}{=} 0$

$\Leftrightarrow w_0 = \bar{y}$

global optimum

Normal Equations

Recall that the cost function for linear regression with mean-squared error is given by

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}(y_n - \mathbf{x}_n^\top \mathbf{w})^2 = \frac{1}{2N}(\mathbf{y}-\mathbf{X}\mathbf{w})^\top(\mathbf{y}-\mathbf{X}\mathbf{w}),$$

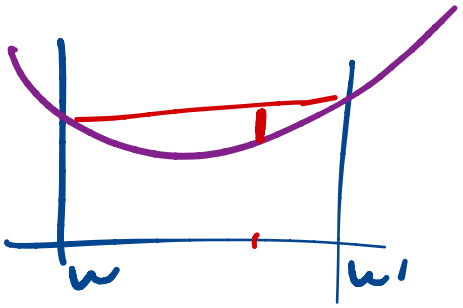
where

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}, \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1D} \\ x_{21} & x_{22} & \dots & x_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \dots & x_{ND} \end{bmatrix}$$

We claim that this cost function is *convex* in the $\mathbf{w}$. There are several ways of proving this:

1. Simplest way: observe that $\mathcal{L}$ is naturally represented as the sum (with positive coefficients) of the simple terms $(y_n - \mathbf{x}_n^\top \mathbf{w})^2$. Further, each of these simple terms is the composition of a linear function with a convex function (the square function). Therefore, each of these simple terms is convex and hence the sum is convex.

2. Directly verify the definition, that for any $\lambda \in [0,1]$ and $\mathbf{w}, \mathbf{w}'$,



$$\mathcal{L}(\lambda \mathbf{w} + (1-\lambda)\mathbf{w}') - (\lambda \mathcal{L}(\mathbf{w}) + (1-\lambda)\mathcal{L}(\mathbf{w}')) \leq 0.$$

Computation: LHS =

$$-\frac{1}{2N}\lambda(1-\lambda)\|\mathbf{X}(\mathbf{w}-\mathbf{w}')\|_2^2,$$

which indeed is non-positive.

3. We can compute the second derivative (the Hessian) and show that it is positive semidefinite (all its eigenvalues are non-negative). For the present case a computation shows that the Hessian has the form

$$\left(\frac{\partial^2 \mathcal{L}(\mathbf{w})}{\partial w_i \partial w_j}\right)_{ij}$$

$$\nabla^2 \mathcal{L}(\mathbf{w}) = \frac{1}{N}\mathbf{X}^\top \mathbf{X}.$$

This matrix is indeed positive semidefinite since its non-zero eigenvalues are the squares of the non-zero singular values of the matrix $\mathbf{X}$.

(2)

Now where we know that the function is convex, let us find its minimum. If we take the gradient of this expression with respect to the weight vector $\mathbf{w}$ we get

$$\nabla \mathcal{L}(\mathbf{w}) = -\frac{1}{N}\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\mathbf{w}).$$

If we set this expression to $\mathbf{0}$ we get the normal equations for linear regression,

$$\mathbf{X}^\top \underbrace{(\mathbf{y} - \mathbf{X}\mathbf{w})}_{\text{error}} = \mathbf{0}.$$

$\nabla \mathcal{L}(w) \overset{!}{=} 0$

$= 0$

① convex + ② $\nabla = 0$ $\Rightarrow$ ok!

$= e$

↑ rows of X^T = cols of X = feature vectors $\in \mathbb{R}^N$

Geometric Interpretation

The error is orthogonal to all columns of $\mathbf{X}$.

The span of $\mathbf{X}$ is the space spanned by the columns of $\mathbf{X}$. Every element of the span can be written as $\mathbf{u} = \mathbf{X}\mathbf{w}$ for some choice of $\mathbf{w}$. Which element of $span(\mathbf{X})$ shall we take? The normal equations tell us that the optimum choice for $\mathbf{u}$, call it $\mathbf{u}^\star$, is that element so that $\mathbf{y} - \mathbf{u}^\star$ is orthogonal to $span(\mathbf{X})$. In other words, we should pick $\mathbf{u}^\star$ to be equal to the projection of $\mathbf{y}$ onto $span(\mathbf{X})$.

$e := y - Xw \quad \in \mathbb{R}^N$

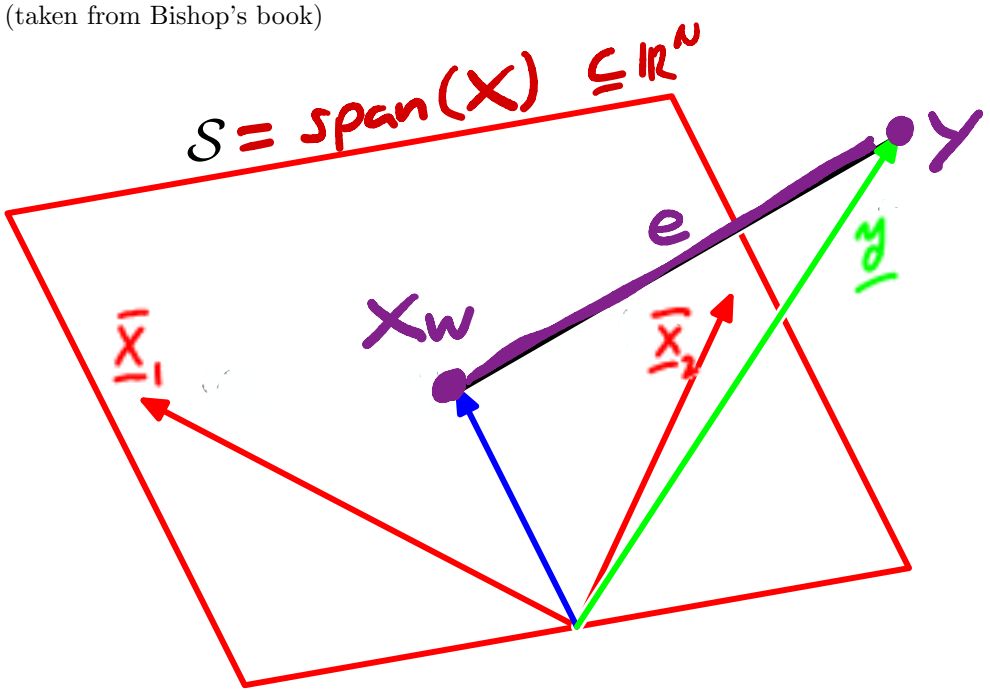
Normal equation

$X^T e \doteq 0$

$\min_w \underbrace{L(w)}_{\doteq \|e\|^2}$

The following figure illustrates this:

(taken from Bishop's book)



$\in \mathbb{R}^N$

Least Squares

The matrix $\mathbf{X}^\top\mathbf{X} \in \mathbb{R}^{D\times D}$ is called the Gram matrix. If it is invertible, we can multiply the normal equation by the inverse of the Gram matrix from the left to get a closed-form expression for the minimum:

$$\mathbf{w}^\star = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}.$$

We can use this model to predict a new value for an unseen datapoint (test point) $\mathbf{x}_m$:

$$\hat{y}_m := \mathbf{x}_m^\top\mathbf{w}^\star = \mathbf{x}_m^\top(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}.$$

Invertibility and Uniqueness

Note that the Gram matrix $\mathbf{X}^\top\mathbf{X} \in \mathbb{R}^{D\times D}$ is invertible if and only if $\mathbf{X}$ has full column rank, or in other words $rank(\mathbf{X}) = D$.

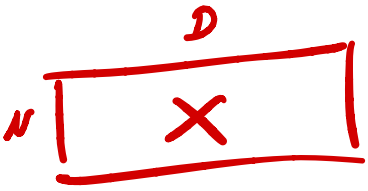
Proof: To see this assume first that $rank(\mathbf{X}) < D$. Then there exists a non-zero vector $\mathbf{u}$ so that $\mathbf{X}\mathbf{u} = \mathbf{0}$. It follows that $\mathbf{X}^\top\mathbf{X}\mathbf{u} = \mathbf{0}$, and so $rank(\mathbf{X}^\top\mathbf{X}) < D$. Therefore, $\mathbf{X}^\top\mathbf{X}$ is not invertible.
Conversely, assume that $\mathbf{X}^\top\mathbf{X}$ is not invertible. Hence, there exists a non-zero vector $\mathbf{v}$ so that $\mathbf{X}^\top\mathbf{X}\mathbf{v} = \mathbf{0}$. It follows that

$$\mathbf{0} = \mathbf{v}^\top\mathbf{X}^\top\mathbf{X}\mathbf{v} = (\mathbf{X}\mathbf{v})^\top(\mathbf{X}\mathbf{v}) = \|\mathbf{X}\mathbf{v}\|^2.$$

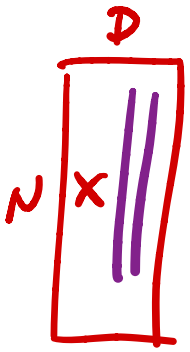
This implies that $\mathbf{X}\mathbf{v} = \mathbf{0}$, i.e., $rank(\mathbf{X}) < D$.

Rank Deficiency and Ill-Conditioning

Unfortunately, in practice, $\mathbf{X}$ is often rank deficient.



- If $D > N$, we always have $rank(\mathbf{X}) < D$
  (since row rank = col. rank)
- If $D \leq N$, but some of the columns $\mathbf{x}_{:d}$ are (nearly) collinear, then the matrix is ill-conditioned, leading to numerical issues when solving the linear system.



Can we solve least squares if $\mathbf{X}$ is rank deficient? Yes, using a linear system solver.

$$\frac{\lambda_{max}(X^T X)}{\lambda_{min}(X^T X)}$$

↑ zero if rank-deficient

Summary of Linear Regression

We have studied three types of methods:

1. Grid Search
2. Iterative Optimization Algorithms
   (Stochastic) Gradient Descent

SGD

3. Least squares
   closed-form solution, for linear MSE

compute inverse

comp. cost $O(D^2N + D^3)$

Additional Notes

Closed-form solution for MAE

Can you derive closed-form solution for 1-parameter model when using MAE cost function?
See this short article: http://www.johnmyleswhite.com/notebook/2013/03/22/modes-medians-and-means-an-unifying-perspective/.

Implementation

There are many ways to solve a linear system, but using the QR decomposition is one of the most robust ways. Matlab's backslash operator and also NumPy's linalg package implement this in just one line:

```
w = np.linalg.solve(X, y)
```

For a robust implementation, see Sec. 7.5.2 of Kevin Murphy's book.

Annotated Version

Machine Learning Course - CS-433

Maximum Likelihood

Sept 29, 2020

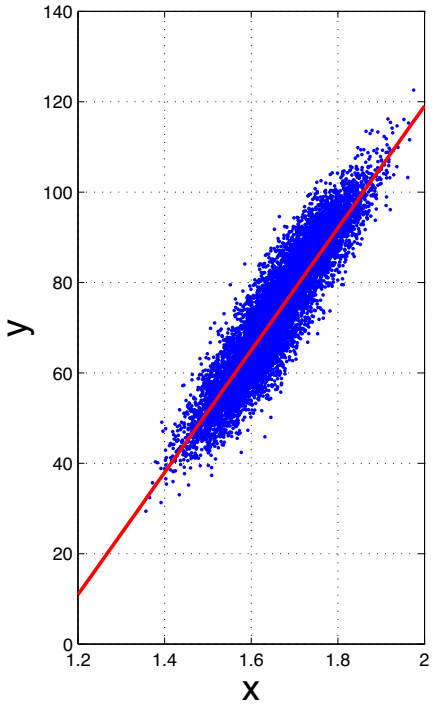
minor changes by Martin Jaggi 2020,2019,2018, minor changes by Rüdiger Urbanke 2017,
minor changes by Martin Jaggi 2016 ©Mohammad Emtiyaz Khan 2015

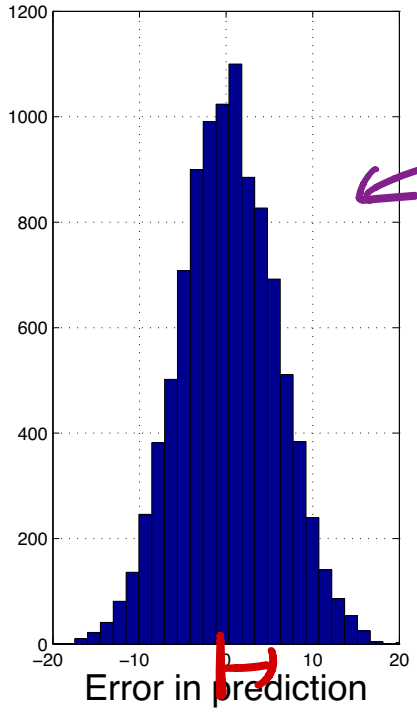
Last updated on: September 29, 2020

EPFL

Motivation

In the previous lecture we arrived at the least-squares problem in the following way: we postulated a particular cost function (square loss) and then, given data, found that model that minimizes this cost function. In the current lecture we will take an alternative route. The final answer will be the same, but our starting point will be probabilistic. In this way we find a second interpretation of the least-squares problem.





Histogram of errors

$e_n = y_n - x_n^T w$

Gaussian distribution and independence

Recall the definition of a Gaussian random variable in $\mathbb{R}$ with mean μ and variance σ^2. It has a density of

$y \in \mathbb{R}$

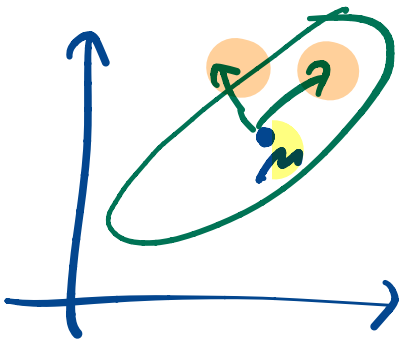
$$p(y \,|\, \mu, \sigma^2) = \mathcal{N}(y \,|\, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y-\mu)^2}{2\sigma^2}\right].$$

In a similar manner, the density of a Gaussian random vector with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$ (which must be a positive semi-definite matrix) is

$\mathbf{y} \in \mathbb{R}^N$

$$\mathcal{N}(\mathbf{y} \,|\, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^D \, det(\boldsymbol{\Sigma})}} \exp\left[-\frac{1}{2}(\mathbf{y}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{y}-\boldsymbol{\mu})\right].$$

Also recall that two random variables X and Y are called *independent* when $p(x, y) = p(x)p(y)$.



A probabilistic model for least-squares

We assume that our data is generated by the model,

$$y_n = \mathbf{x}_n^\top \mathbf{w} + \epsilon_n,$$

where the ϵ_n (the noise) is a zero-mean Gaussian random variable with variance σ^2 and the noise that is added to the various samples is independent of each other, and independent of the input. Note that the model $\mathbf{w}$ is unknown.

Therefore, given N samples, the likelihood of the data vector $\mathbf{y} = (y_1, \cdots, y_N)$ given the input $\mathbf{X}$ (each row is one input) and the model $\mathbf{w}$ is equal to

$$p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) = \prod_{n=1}^{N} p(y_n \mid \mathbf{x}_n, \mathbf{w}) = \prod_{n=1}^{N} \mathcal{N}(y_n \mid \mathbf{x}_n^\top \mathbf{w}, \sigma^2).$$

The probabilistic view point is that we should maximize this likelihood over the choice of model $\mathbf{w}$. I.e., the "best" model is the one that maximizes this likelihood.

Defining cost with log-likelihood

Instead of maximizing the likelihood, we can take the logarithm of the likelihood and maximize it instead. Expression is called the log-likelihood (LL).

$$\mathcal{L}_{\text{LL}}(\mathbf{w}) := \log p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) = -\frac{1}{2\sigma^2} \sum_{n=1}^{N} (y_n - \mathbf{x}_n^\top \mathbf{w})^2 + \text{const.}$$

Compare the LL to the MSE (mean squared error)

$$\mathcal{L}_{\text{LL}}(\mathbf{w}) = -\frac{1}{2\sigma^2} \sum_{n=1}^{N} (y_n - \mathbf{x}_n^\top \mathbf{w})^2 + \text{const}$$

$$\mathcal{L}_{\text{MSE}}(\mathbf{w}) = \frac{1}{2N} \sum_{n=1}^{N} (y_n - \mathbf{x}_n^\top \mathbf{w})^2$$

Maximum-likelihood estimator (MLE)

It is clear that maximizing the LL is equivalent to minimizing the MSE:

$$\arg\min_{\mathbf{w}} \mathcal{L}_{\text{MSE}}(\mathbf{w}) = \arg\max_{\mathbf{w}} \mathcal{L}_{\text{LL}}(\mathbf{w}).$$

This gives us another way to design cost functions.

MLE can also be interpreted as finding the model under which the observed data is most likely to have been generated from (probabilistically). This interpretation has some advantages that we discuss now.

Properties of MLE

MLE is a *sample* approximation to the *expected log-likelihood*:

$$\mathcal{L}_{\text{LL}}(\mathbf{w}) \approx \mathbb{E}_{p(y,\mathbf{x})}\big[\log p(y \mid \mathbf{x}, \mathbf{w})\big]$$

MLE is consistent, i.e., it will give us the correct model assuming that we have a sufficient amount of data. (can be proven under some weak conditions)

$$\mathbf{w}_{\text{MLE}} \longrightarrow^{p} \mathbf{w}_{\text{true}} \quad \text{in probability}$$

The MLE is asymptotically normal, i.e.,

$$(\mathbf{w}_{\text{MLE}} - \mathbf{w}_{\text{true}}) \longrightarrow^{d} \frac{1}{\sqrt{N}} \mathcal{N}(\mathbf{w}_{\text{MLE}} \mid \mathbf{0}, \mathbf{F}^{-1}(\mathbf{w}_{\text{true}}))$$

where $\mathbf{F}(\mathbf{w}) = -\mathbb{E}_{p(\mathbf{y})}\left[\frac{\partial^2 \mathcal{L}}{\partial \mathbf{w} \partial \mathbf{w}^\top}\right]$ is the Fisher information.

MLE is efficient, i.e. it achieves the Cramer-Rao lower bound.

$$\text{Covariance}(\mathbf{w}_{\text{MLE}}) = \mathbf{F}^{-1}(\mathbf{w}_{\text{true}})$$

Another example

We can replace Gaussian distribution by a Laplace distribution.

$$p(y_n \mid \mathbf{x}_n, \mathbf{w}) = \frac{1}{2b} e^{-\frac{1}{b}|y_n - \mathbf{x}_n^\top \mathbf{w}|}$$

Annotated version

Machine Learning Course - CS-433

Underfitting and Overfitting

Oct 1, 2020

minor changes by Martin Jaggi 2020,2019,2018, minor changes by Rüdiger Urbanke 2017,
minor changes by Martin Jaggi 2016 ©Mohammad Emtiyaz Khan 2015

Last updated on: September 29, 2020

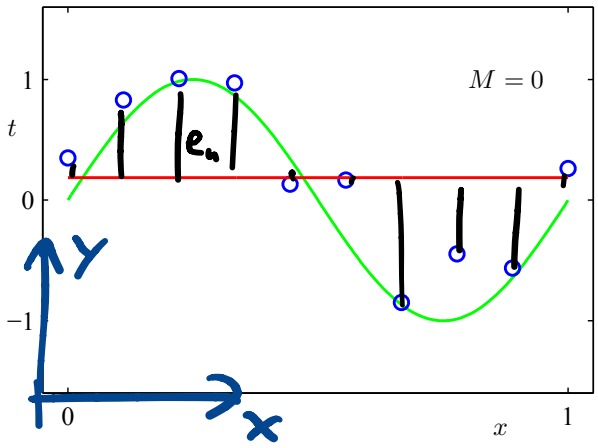
EPFL

Motivation

Models can be *too limited* or they can be *too rich*. In the first case we cannot find a function that is a good fit for the date in our model. We then say that we underfit. In the second case we have such a rich model family that we do not just fit the underlying function but we in fact fit the noise in the date as well. We then talk about an overfit. Both of these phenomena are undesirable. This discussion is made more difficult since all we have is data and so we do not know a priori what part is the underlying signal and what part is noise.

Underfitting with Linear Models

It is easy to see that linear models might underfit. Consider a scalar case as shown in the figure below.



$f_w(x) := w_0$

The solid curve is the underlying function and the circles are the actual data. E.g., we assume that there is a scalar function $g(x)$ but that we do not observe $g(x_n)$ directly but only a noisy version of it, $y_n = g(x_n) + Z_n$, where Z_n is

the noise. The noise might be due for example to some measurement inaccuracies. The y_n are shown as blue cirles. If our model family consists of only linear functions of the scalar input x, i.e., $\mathcal{F} = \{f_w(x) = wx\}$, where w is a scalar constant (the slope of the function), then it is clear that we cannot match the given function accurately, regardless how many samples we get and how small the noise is. We therefore will underfit.

Extended/Augmented Feature Vectors From the above example it might seem that linear models are too simple to ever overfit. But in fact, linear models are highly prone to overfitting, much more so than complicated models like neural nets.

Since linear models are inherently not very rich the following is a standard "trick" to make them more powerful.

In order to increase the representational power of linear models we typically "augment" the input. E.g., if the input (feature) is one-dimensional we might add a polynomial basis (of arbitrary degree M),

before: $x_n \in \mathbb{R}$ $\quad$ $\mathbb{R}$ $\quad$ alternative $\sin(x_n)$, $\log(x_n)$

after: $\phi(x_n) \in \mathbb{R}^{M+1}$ $\quad$ ψ

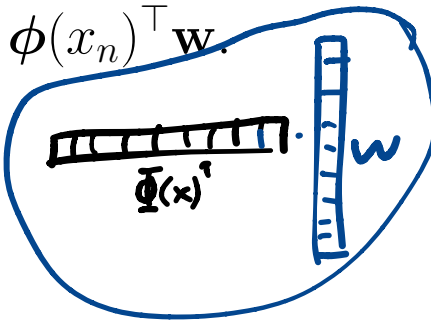
$$\phi(x_n) := [1, x_n, x_n^2, x_n^3, \ldots, x_n^M]$$

so that we end up with an extended feature vector.

We then fit a linear model to this extended feature vector $\phi(x_n)$:

$$y_n \approx w_0 + w_1 x_n + w_2 x_n^2 + \ldots + w_M x_n^M =: \phi(x_n)^\top \mathbf{w}$$

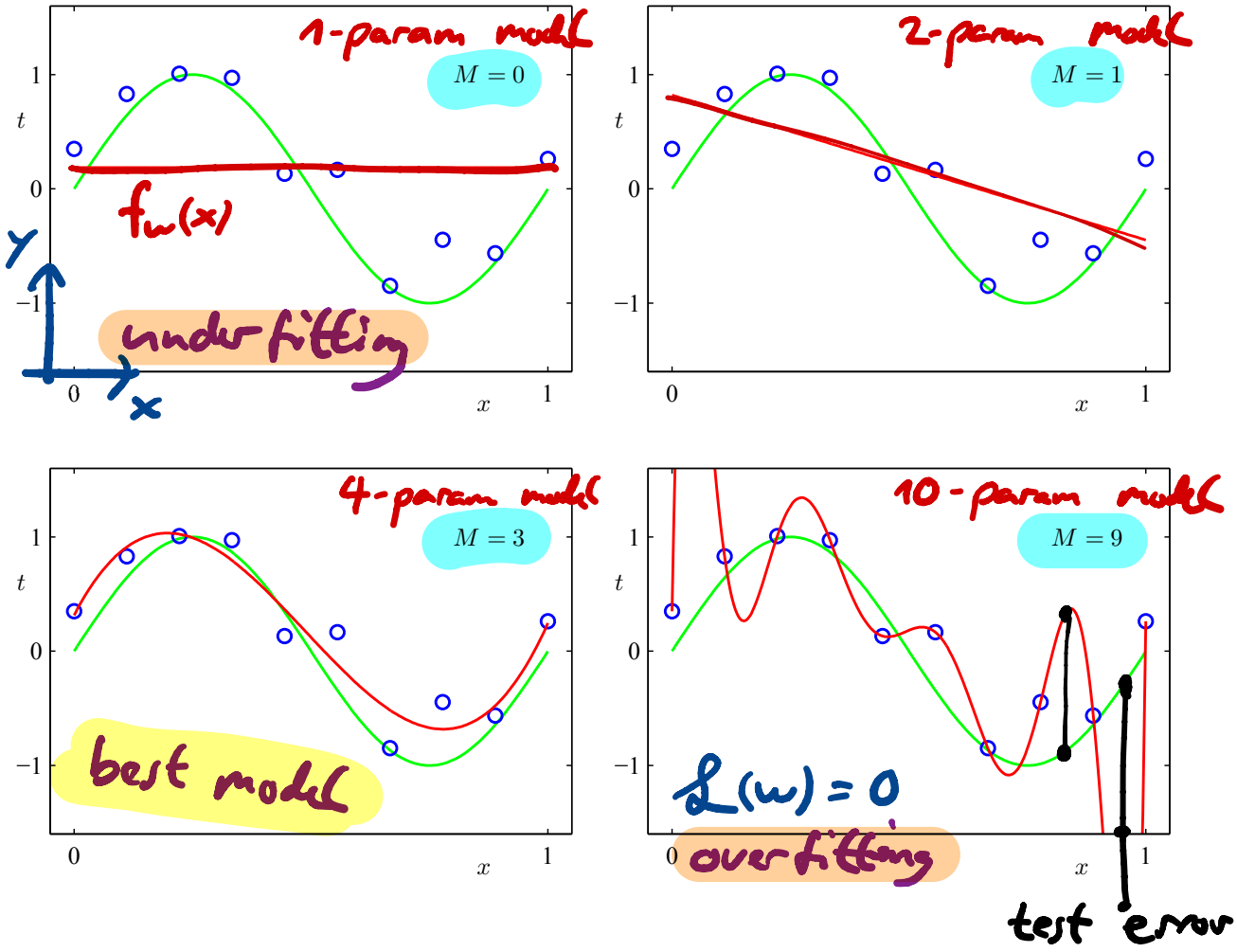
$f_w(\phi(x_n)) = f_w(x_n)$



M: model complexity

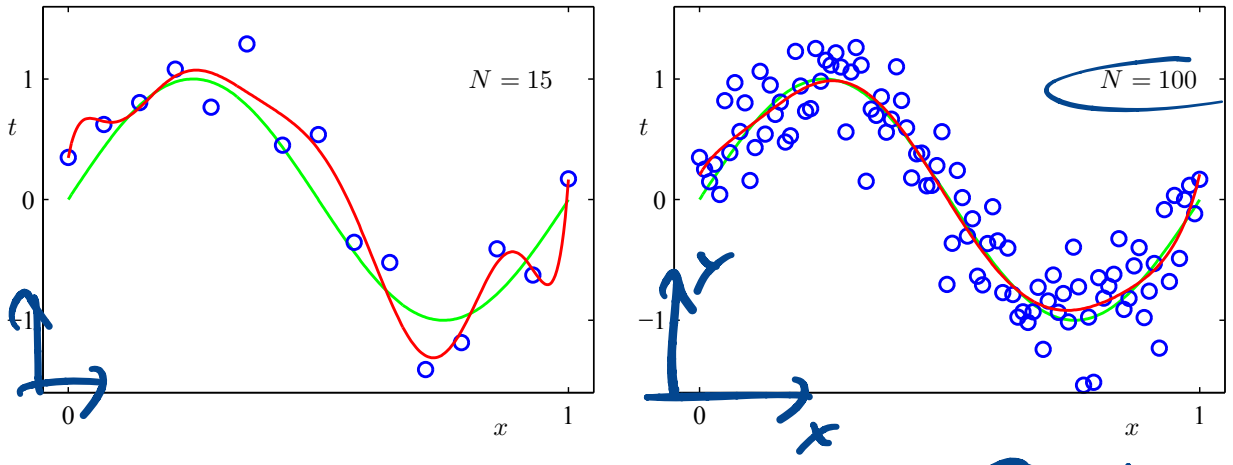
Overfitting with Linear Models

In the following four figures, circles are data points, the green line represents the "true function", and the red line is the model. The parameter M is the maximum degree in the polynomial basis.



For $M = 0$ (the model is a constant) the model is underfitting and the same is true for $M = 1$. For $M = 3$ the model fits the data fairly well and is not yet so rich as to fit in addition the small "wiggles" caused by the noise. But for $M = 9$ we now have such a rich model that it can fit every single data point and we see severe overfitting taking place. What can we do to avoid overfitting? If you increase the amount of data (increase N, but keep M fixed), overfitting

might reduce. This is shown in the following two figures where we again consider the same model complexity $M = 9$ but we have extra data ($N = 15$ or even $N = 100$).



Fight overfitting ① simpler model ② more data

A Word About Notation

If it is important to distinguish the original input x from the augmented input then we will use $\phi(x)$ to denote this augmented input vector. But we can consider this augmentation as part of the pre-processing, and then we might simply write $\mathbf{x}$ to denote the input. This will save us a lot of notation.

Additional Materials

Read about overfitting in the paper by Pedro Domingos (Sections 3 and 5 of "A few useful things to know about machine learning").

Annotated Version

Machine Learning Course - CS-433

Regularization: Ridge Regression and Lasso

Oct 1, 2020

minor changes by Martin Jaggi 2020,2019,2018, minor changes by Rüdiger Urbanke 2017,
changes by Martin Jaggi 2016 ©Mohammad Emtiyaz Khan 2015

Last updated on: September 30, 2020

EPFL

Motivation

We have seen that by augmenting the feature vector we can make linear models as powerful as we want. Unfortunately this leads to the problem of overfitting. *Regularization* is a way to mitigate this undesirable behavior.

We will discuss regularization in the context of linear models, but the same principle applies also to more complex models such as neural nets.

Occam's razor

„simple models are prefered"

Regularization

Through regularization, we can penalize complex models and favor simpler ones:

fit

complexity of model w

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) + \Omega(\mathbf{w})$$

The second term Ω is a regularizer, measuring the complexity of the model given by $\mathbf{w}$.

L_2-Regularization: Ridge Regression

$x_n, w \in \mathbb{R}^D$

The most frequently used regularizer is the standard Euclidean norm (L_2-norm), that is

tradeoff-parameter $\lambda > 0$

$$\Omega(\mathbf{w}) = \lambda \|\mathbf{w}\|_2^2$$

where $\|\mathbf{w}\|_2^2 = \sum_i w_i^2$. Here the main effect is that large model weights w_i will be penalized (avoided), since we consider them "unlikely", while small ones are ok. When $\mathcal{L}$ is MSE, this is called ridge regression:

$\lambda \to 0$ no regularization — potential overfitting

$\lambda \to \infty$ underfitting — $\|w\| \to 0$

$\mathcal{L}(w)$ $\Omega(w)$

$$\min_{\mathbf{w}} \frac{1}{2N} \sum_{n=1}^{N} \left[y_n - \mathbf{x}_n^\top \mathbf{w} \right]^2 + \lambda \|\mathbf{w}\|_2^2$$

$P = \mathcal{L} + \Omega$

Least squares is a special case of this: set $\lambda := 0$.

Explicit solution for $\mathbf{w}$: Differentiating and setting to zero:

$$\mathbf{w}_{\text{ridge}}^\star = (\mathbf{X}^\top \mathbf{X} + \lambda' \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}$$

(here for simpler notation $\frac{\lambda'}{2N} = \lambda$)

First-order optimality

- P convex
- $\nabla P(w) \stackrel{!}{=} 0$

$\Rightarrow$ w optimal

$$\nabla \mathcal{L}(w) = -\frac{1}{N} X^T (y - Xw)$$

$$+ \quad \nabla \Omega(w) = 2\lambda \cdot w$$

$$\stackrel{!}{=} 0$$

$$\Leftrightarrow \quad (X^T X + \lambda' I) w = X^T y$$

Ridge Regression to Fight Ill-Conditioning

The eigenvalues of $(\mathbf{X}^\top\mathbf{X} + \lambda'\mathbf{I})$ are all at least λ' and so the inverse always exists. This is also referred to as *lifting the eigenvalues*.

Proof: Write the ~~singular~~-value decomposition of $\mathbf{X}^\top\mathbf{X}$ as $\mathbf{USU}^\top$. We then have

$$\begin{aligned}\mathbf{X}^\top\mathbf{X} + \lambda'\mathbf{I} &= \mathbf{USU}^\top + \lambda'\mathbf{UIU}^\top \\ &= \mathbf{U}[\mathbf{S} + \lambda'\mathbf{I}]\mathbf{U}^\top.\end{aligned}$$

We see now that every ~~singular~~ value is "lifted" by an amount λ'.

Here is an alternative proof. Recall that for a symmetric matrix $\mathbf{A}$ we can also compute eigenvalues by looking at the so-called Rayleigh ratio,

$$R(\mathbf{A}, \mathbf{v}) = \frac{\mathbf{v}^\top\mathbf{Av}}{\mathbf{v}^\top\mathbf{v}}.$$

Note that if $\mathbf{v}$ is an eigenvector with eigenvalue λ then the Rayleigh coefficient indeed gives us λ. We can find the smallest and largest eigenvalue by minimizing and maximizing this coefficient. But note that if we apply this to the symmetric matrix $\mathbf{X}^\top\mathbf{X} + \lambda'\mathbf{I}$ then for any vector $\mathbf{v}$ we have

$$\frac{\mathbf{v}^\top(\mathbf{X}^\top\mathbf{X} + \lambda'\mathbf{I})\mathbf{v}}{\mathbf{v}^\top\mathbf{v}} \geq \frac{\lambda'\mathbf{v}^\top\mathbf{v}}{\mathbf{v}^\top\mathbf{v}} = \lambda'.$$

L_1-Regularization: The Lasso

$\|w\|_1 := \sum_{j=1}^{D} |w_j|$

As an alternative measure of the complexity of the model, we can use a different norm. A very important case is the L_1-norm, leading to L_1-regularization. In combination with the MSE cost function, this is known as the Lasso:

MSE = $\mathcal{L}$ $\quad$ $\Omega(w) = \|w\|_1$

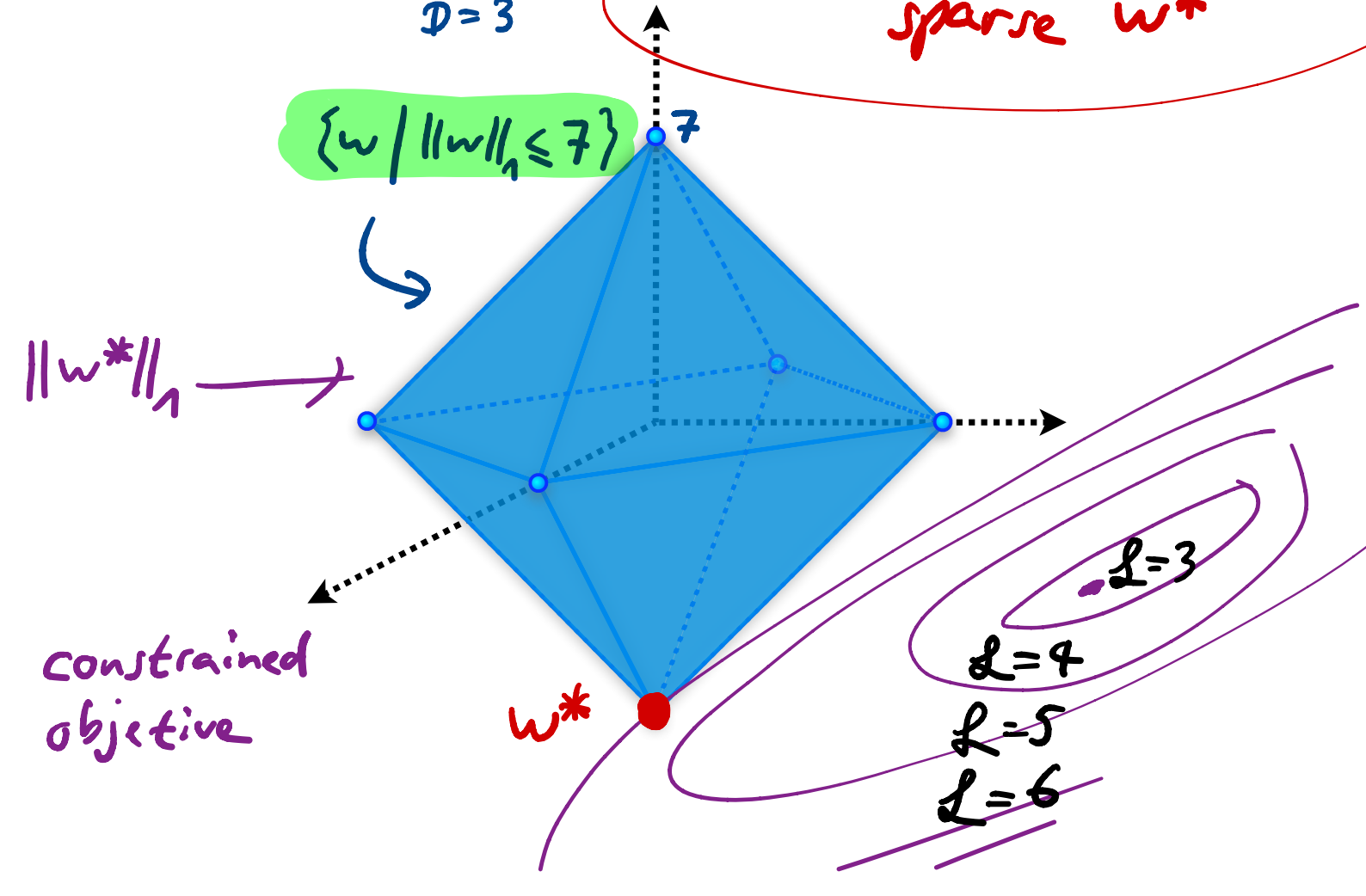
$$\min_{\mathbf{w}} \frac{1}{2N} \sum_{n=1}^{N} [y_n - \mathbf{x}_n^\top \mathbf{w}]^2 + \lambda \|\mathbf{w}\|_1$$

penalized objective

where

$$\|\mathbf{w}\|_1 := \sum_i |w_i|.$$

→ encourages sparse w^*



Explanation:

The figure above shows a "ball" of constant L_1 norm. To keep things simple assume that $\mathbf{X}^\top\mathbf{X}$ is invertible. We claim that in this case the set

$$\{\mathbf{w} : \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 = \alpha\} \tag{1}$$

is an ellipsoid and this ellipsoid simply scales around its origin as we change α. We claim that for the L_1-regularization the optimum solution is likely going to be sparse (only has few non-zero components) compared to the case where we use L_2-regularization.

Why is this the case? Assume that a genie tells you the L_1-norm of the optimum solution. Draw the L_1-ball with that norm value (think of 2D to visualize it). So now you know that the optimal point is somewhere on the surface of this "ball". Further you know that there are ellipsoids, all with the same mean and rotation that describes the equal error surfaces incurred by the first term. The optimum solution is where the "smallest" of these ellipsoids just touches the L_1-ball. Due to the geometry of this ball this point is more likely to be on one of the "corner" points. In turn, sparsity is desirable, since it leads to a "simple" model.

How do we see the claim that (1) describes and ellipsoid? First look at $\alpha = \|\mathbf{X}\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w}$. This is a quadratic form. Let $\mathbf{A} = \mathbf{X}^\top \mathbf{X}$. Note that $\mathbf{A}$ is a symmetric matrix and by assumption it has full rank. If $\mathbf{A}$ is a diagonal matrix with strictly positive elements a_i along the diagonal then this describes the equation

$$\sum_i a_i \mathbf{w}_i^2 = \alpha,$$

which is indeed the equation for an ellipsoid. In the general case, $\mathbf{A}$ can be written as (using the SVD) $\mathbf{A} = \mathbf{U}\mathbf{B}\mathbf{U}^T$, where $\mathbf{B}$ is a diagonal matrix with strictly positive entries. This then corresponds to an ellipsoid with rotated axes. If we now look at $\alpha = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$, where $\mathbf{y}$ is in the column space of $\mathbf{X}$ then we can write it as $\alpha = \|\mathbf{X}(\mathbf{w}_0 - \mathbf{w})\|^2$ for a suitable chosen $\mathbf{w}_0$ and so this corresponds to a shifted ellipsoid. Finally, for the general case, write $\mathbf{y}$ as $\mathbf{y} = \mathbf{y}_\parallel + \mathbf{y}_\perp$, where $\mathbf{y}_\parallel$ is the component of $\mathbf{y}$ that lies in the subspace spanned by the columns of $\mathbf{X}$ and $\mathbf{y}_\perp$ is the component that

is orthogonal. In this case

$$\begin{aligned}\alpha &= \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 \\ &= \|\mathbf{y}_{\|} + \mathbf{y}_{\perp} - \mathbf{X}\mathbf{w}\|^2 \\ &= \|\mathbf{y}_{\perp}\|^2 + \|\mathbf{y}_{\|} - \mathbf{X}\mathbf{w}\|^2 \\ &= \|\mathbf{y}_{\perp}\|^2 + \|\mathbf{X}(\mathbf{w}_0 - \mathbf{w})\|^2.\end{aligned}$$

Hence this is then equivalent to the equation $\|\mathbf{X}(\mathbf{w}_0 - \mathbf{w})\|^2 = \alpha - \|\mathbf{y}_{\perp}\|^2$, proving the claim. From this we also see that if $\mathbf{X}^{\top}\mathbf{X}$ is not full rank then what we get is not an ellipsoid but a cylinder with an ellipsoidal cross-section.

Additional Notes

Other Types of Regularization

Popular methods such as shrinkage, dropout and weight decay (in the context of neural networks), early stopping of the optimization are all different forms of regularization.

Another view of regularization: The ridge regression formulation we have seen above is similar to the following constrained problem (for some $\tau > 0$).

$$\min_{\mathbf{w}} \quad \frac{1}{2N} \sum_{n=1}^{N} (y_n - \mathbf{x}_n^\top \mathbf{w})^2, \qquad \text{such that } \|\mathbf{w}\|_2^2 \leq \tau$$

The following picture illustrates this.

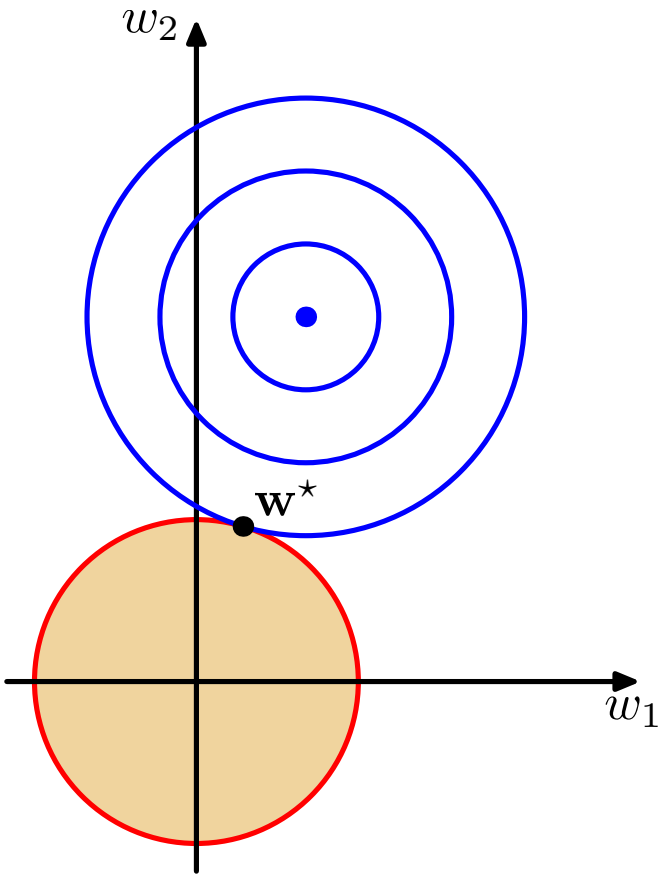


Figure 1: Geometric interpretation of Ridge Regression. Blue lines indicating the level sets of the MSE cost function.

For the case of using L_1 regularization (known as the Lasso, when used with MSE) we analogously consider

$$\min_{\mathbf{w}} \quad \frac{1}{2N}\sum_{n=1}^{N}(y_n - \mathbf{x}_n^\top \mathbf{w})^2, \qquad \text{such that } \|\mathbf{w}\|_1 \leq \tau$$

This forces some of the elements of $\mathbf{w}$ to be strictly 0 and therefore enforces sparsity in the model (some features will not be used since their coefficients are zero).

- Why does L_1 regularizer enforce sparsity? *Hint:* Draw the picture similar to above for Lasso, and locate the optimal solution.
- Why is it good to have sparsity in the model? Is it going to be better than least-squares? When and why?

Ridge Regression as MAP estimator

Recall that classic *least-squares linear regression* can be interpreted as the maximum likelihood estimator:

$$
\begin{aligned}
\mathbf{w}_{\text{lse}} &\overset{(a)}{=} \arg\min_{\mathbf{w}} && -\log p(\mathbf{y}, \mathbf{X}|\mathbf{w}) \\
&\overset{(b)}{=} \arg\min_{\mathbf{w}} && -\log p(\mathbf{X}|\mathbf{w}) p(\mathbf{y}|\mathbf{X}, \mathbf{w}) \\
&\overset{(c)}{=} \arg\min_{\mathbf{w}} && -\log p(\mathbf{X}) p(\mathbf{y}|\mathbf{X}, \mathbf{w}) \\
&\overset{(d)}{=} \arg\min_{\mathbf{w}} && -\log p(\mathbf{y}|\mathbf{X}, \mathbf{w}) \\
&\overset{(e)}{=} \arg\min_{\mathbf{w}} && -\log \left[\prod_{n=1}^{N} p(y_n|\mathbf{w}_n, \mathbf{w}) \right] \\
&\overset{(f)}{=} \arg\min_{\mathbf{w}} && -\log \left[\prod_{n=1}^{N} \mathcal{N}(y_n \,|\, \mathbf{x}_n^\top \mathbf{w}, \sigma^2) \right] \\
&= \arg\min_{\mathbf{w}} && -\log \left[\prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(y_n - \mathbf{x}_n^\top \mathbf{w})^2} \right] \\
&= \arg\min_{\mathbf{w}} && -N \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \sum_{n=1}^{N} \frac{1}{2\sigma^2}(y_n - \mathbf{x}_n^\top \mathbf{w})^2 \\
&= \arg\min_{\mathbf{w}} && \frac{1}{2\sigma^2} \sum_{n=1}^{N} (y_n - \mathbf{x}_n^\top \mathbf{w})^2
\end{aligned}
$$

In step (a) on the right we wrote down the negative of the log of the likelihood. The maximum likelihood criterion choses that parameter $\mathbf{w}$ that minimizes this quantity (i.e., maximizes the likelihood). In step (b) we factored the likelihood. The usual assumption is that the choice of the input samples $\mathbf{x}_n$ does not depend on the model parameter (which only influces the output given the input. Hence, in step (c) we removed the conditioning. Since the factor $p(\mathbf{X})$ does not depend on $\mathbf{w}$, i.e., is a constant wrt to $\mathbf{w}$) we can remove it. This is done in step (d). In step (e) we used the assumption that the samples are iid. In step (f) we then used our assumption that the samples have the form $y_n = \mathbf{w}_n^\top \mathbf{w} + Z_n$,

where Z_n is a Gaussian noise with mean zero and variance σ_2. The rest is calculus.

Ridge regression has a very similar interpretation. Now we start with the posterior $p(\mathbf{w}|\mathbf{X}, \mathbf{y})$ and chose that parameter $\mathbf{w}$ that maximizes this posterior. Hence this is called the maximum-a-posteriori (MAP) estimate. As before, we take the log and add a minus sign and minimize instead. In order to compute the posterior we use Bayes law and we assume that the components of the weight vector are iid Gaussians with mean zero and variance $\frac{1}{\lambda}$.

$$
\begin{aligned}
\mathbf{w}_{\text{ridge}} &= \arg\min_{\mathbf{w}} && -\log p(\mathbf{w}|\mathbf{X}, \mathbf{y}) \\
&\overset{(a)}{=} \arg\min_{\mathbf{w}} && -\log \frac{p(\mathbf{y}, \mathbf{X}|\mathbf{w})p(\mathbf{w})}{p(\mathbf{y}, \mathbf{X})} \\
&\overset{(b)}{=} \arg\min_{\mathbf{w}} && -\log p(\mathbf{y}, \mathbf{X}|\mathbf{w})p(\mathbf{w}) \\
&\overset{(c)}{=} \arg\min_{\mathbf{w}} && -\log p(\mathbf{y}|\mathbf{X}, \mathbf{w})p(\mathbf{w}) \\
&= \arg\min_{\mathbf{w}} && -\log \left[p(\mathbf{w}) \prod_{n=1}^{N} p(y_n|\mathbf{w}_n, \mathbf{w}) \right] \\
&= \arg\min_{\mathbf{w}} && -\log \left[\mathcal{N}(\mathbf{w} \,|\, 0, \tfrac{1}{\lambda}\mathbf{I}) \prod_{n=1}^{N} \mathcal{N}(y_n \,|\, \mathbf{x}_n^\top \mathbf{w}, \sigma^2) \right] \\
&= \arg\min_{\mathbf{w}} && -\log \left[\frac{1}{(2\pi\frac{1}{\lambda})^{D/2}} e^{-\frac{\lambda}{2}\|\mathbf{w}\|^2} \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(y_n - \mathbf{x}_n^\top \mathbf{w})^2} \right] \\
&= \arg\min_{\mathbf{w}} && \sum_{n=1}^{N} \frac{1}{2\sigma^2}(y_n - \mathbf{x}_n^\top \mathbf{w})^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2.
\end{aligned}
$$

In step (a) we used Bayes' law. In step (b) and (c) we eliminated quantities that do not depend on $\mathbf{w}$.

Machine Learning Course - CS-433

Generalization, Model Selection, and Validation

Oct 6, 2020

minor changes by Nicolas Flammarion 2020, minor changes by Martin Jaggi 2019, changes by Martin Jaggi 2018, 2016 changes by Rüdiger Urbanke 2019, 2017 ©Mohammad Emtiyaz Khan 2015

Last updated on: October 17, 2020

EPFL

Motivation

Assume that your friend has trained a model on some data and now claims to have found the "perfect" regression function f. How can you verify this claim and have confidence that f will have good performance? This leads us to the question of *generalization*.

As a second motivation consider the following problem. We have seen in ridge regression that the regularization parameter $\lambda > 0$ can be tuned to reduce overfitting by reducing model complexity,

$$\min_{\mathbf{w}} \quad \frac{1}{2N} \sum_{n=1}^{N} (y_n - \mathbf{x}_n^\top \mathbf{w})^2 \quad + \quad \lambda \|\mathbf{w}\|^2$$

The parameter λ is a *hyper*-parameter.

In a similar manner, we can enrich the model complexity, by augmenting the feature vector $\mathbf{x}$. E.g., consider a polynomial feature expansion. Here the degree d is a hyperparameter.

To see a final example consider neural nets. Here we have tens or hundreds of hyperparameters: architechture, width, depth, type of the network etc.

In all these cases we are faced with the same problem: how do we choose these hyperparameters? This is the *model selection* problem.

Data Model and Learning Algorithm

In order to give a meaningful answer to the above questions we first need to specify our data model.
We assume that there is an (unknown) underlying distribution $\mathcal{D}$, with range $\mathcal{X} \times \mathcal{Y}$. The data set we see, call it S, consists of independent samples from $\mathcal{D}$:

$$S = \{(\mathbf{x}_n, y_n) \text{ i.i.d.} \sim \mathcal{D}\}_{n=1}^{N}.$$

The *learning algorithm* takes the data and outputs a model within the class of models that it is given. Often this is done by optimising a cost function. We have seen that we can use e.g. (stochastic) gradient descent or least-squares for the ridge-regression model as an efficient way of implementing this learning. Write $f_S = \mathcal{A}(S)$, where $\mathcal{A}$ denotes the learning algorithm.
If we want to indicate that f_S also depends on parameters of the model, e.g., the λ in the ridge regression model, we can add a subscript to write $f_{S,\lambda}$.

True Error, Emperical Error, and Training Error

Given a model f, how can we assess if f is any good? We should compute the *expected* error over all samples chosen according to $\mathcal{D}$, i.e., we should compute

$$L_{\mathcal{D}}(f) = \mathbb{E}_{\mathcal{D}}[\ell(y, f(\mathbf{x}))],$$

where $\ell(\cdot, \cdot)$ is our loss function. E.g., for ridge regression

$$\ell(y, f(\mathbf{x})) = \tfrac{1}{2}(y - f(\mathbf{x}))^2,$$

The quantity $L_{\mathcal{D}}(f)$ has many names: *(true/expected) (risk/loss/*
This is the quantity we are fundamentally interested in, but we cannot compute it since $\mathcal{D}$ is not known.
But we are given some data S. It is therefore natural to compute the equivalent *empirical* quantity

$$L_S(f) = \frac{1}{|S|} \sum_{(\mathbf{x}_n, y_n) \in S} \ell(y_n, f(\mathbf{x}_n)). \tag{1}$$

This is called the *(empirical) (risk/loss/error).*
There is one added complication. Assume that we are given the data S. If we first learn the model from S, i.e., we compute $f_S = \mathcal{A}(S)$ and then we compute the empirical risk of f_S using the *same data* S then in fact we are computing

$$L_S(f_S) = \frac{1}{|S|} \sum_{(\mathbf{x}_n, y_n) \in S} \ell(y_n, f_S(\mathbf{x}_n)).$$

This is called the *training (risk/loss/error)* and we have already discussed in previous lectures that this training error might not be representative of the error we see on "fresh" samples.
The reason that $L_S(f_S)$ might not be close to $L_{\mathcal{D}}(f_S)$ is of course overfitting.
So let us summarize. If somebody hands us a function f then there is the true risk $L_{\mathcal{D}}(f)$ associated to f, but we cannot in general compute it. If we have some data S that was not used to determine f then we can compute the empirical risk of f with respect to S. This is denoted by $L_S(f)$. We will shortly discuss the relationship between $L_{\mathcal{D}}(f)$ and $L_S(f)$. And if S was used to learn f, then we denote it by f_S and call the resulting empirical error the training error, $L_S(f_S)$.

Splitting the Data and Test Error

Before we go on and explore the relationship between the true risk and the empirical risk, let us first see how we can adress the potential overfitting problem that ocurs if we train and test the model on the same data.
Problem: Validating model on the same data subset we trained it on!
Fix: Split the data into a *training* and a *test* set (a.k.a. *validation* set), call them S_{train} and S_{test}, respectively.
We apply the learning algorithm $\mathcal{A}$ to the training set S_{train} and compute the function $f_{S_{\text{train}}}$. We then compute the error on the test set, i.e.,

$$L_{S_{\text{test}}}(f_{S_{\text{train}}}) = \frac{1}{|S_{\text{test}}|} \sum_{(y_n, \mathbf{x}_n) \in S_{\text{test}}} \ell(y_n, f_{S_{\text{train}}}(\mathbf{x}_n)).$$

This is called the *(test/validation) (risk/loss/error)*. Since S_{test} is a "fresh" sample we can hope that $L_{S_{\text{test}}}(f_{S_{\text{train}}})$ is close to the quantity $L_{\mathcal{D}}(f_{S_{\text{train}}})$.
But we payed a price. We had to split the data and now have less data both for the learning as well as the validation (test) task.
"Cross validation" as described below is more efficient in using data, but hard to analyze.

True Error, Test Error, and Generalization Error

We now get back to our original question. Assume that we have a model f, and that our loss function $\ell(\cdot, \cdot)$ is bounded,

lets say in $[a, b]$. We are given a test set S_{test} chosen i.i.d. from the underlying distribution $\mathcal{D}$ (and this test set was *not* used to train the model). The test/empnrial error is

$$L_{S_{\text{test}}}(f) = \frac{1}{|S_{\text{test}}|} \sum_{(\mathbf{x}_n, y_n) \in S_{\text{test}}} \ell(y_n, f(\mathbf{x}_n)).$$

The true error is

$$L_{\mathcal{D}}(f) = \mathbb{E}_{(y,\mathbf{x})\sim\mathcal{D}}[\ell(y, f(\mathbf{x}))].$$

How far are these apart? This is called the *generalization error* and it is given by

$$|L_{\mathcal{D}}(f) - L_{S_{\text{test}}}(f)|.$$

First note that in expectation they are the same, i.e.,

$$L_{\mathcal{D}}(f) = \mathbb{E}_{S_{\text{test}}\sim\mathcal{D}}[L_{S_{\text{test}}}(f)], \tag{2}$$

where the expectation is over the samples of the test set. But we need to worry about the variation. We claim that

$$\mathbb{P}\Big[\ \big|L_{\mathcal{D}}(f) - L_{S_{\text{test}}}(f)\big| \geq \sqrt{\frac{(b-a)^2 \ln(2/\delta)}{2|S_{\text{test}}|}}\ \Big] \leq \delta. \tag{3}$$

Insights: The error decreases as $\mathcal{O}(1/\sqrt{|S_{\text{test}}|})$ with the number test points. The more data points we have therefore, the more confident we can be that the empirical loss we measure is close to the true loss. If we want δ to be smaller we only need to increase the size of the test set slightly.

Proof of (3):

Since we assumed that each data sample $(\mathbf{x}_n, y_n)$ in the test set S_{test} is chosen independently, the associated losses $\ell(y_n, f(\mathbf{x}_n))$, given a fixed model f, are also i.i.d. random variables, taking values in $[a, b]$ by assumption. Call each such loss Θ_n. The expected value of $\Theta_n = \ell(y_n, f(\mathbf{x}_n))$ is equal to the true loss

$$L_{\mathcal{D}}(f) = \mathbb{E}[\ell(y_n, f(\mathbf{x}_n))].$$

The empirical loss on the other hand is equal to the average of $|S_{\text{test}}|$ such i.i.d. values.
We want to know the chance that the empirical loss $L_{S_{\text{test}}}(f)$ deviates from its true value by more than a given constant. This is a classical problem addressed in the following lemma.

Lemma 0.1 (Chernoff Bound)**.** *Let $\Theta_1, \ldots, \Theta_N$ be a sequence of i.i.d. random variables with mean $\mathbb{E}[\Theta]$ and range $[a, b]$. Then, for any $\varepsilon > 0$,*

$$\mathbb{P}\left[\left|\frac{1}{N}\sum_{n=1}^{N}\Theta_n - \mathbb{E}[\Theta]\right| \geq \varepsilon\right] \leq 2e^{-2N\varepsilon^2/(b-a)^2}.$$

Using Lemma 0.1 let us show (3). Equating $2e^{-2|S_{\text{test}}|\varepsilon^2/(b-a)^2}$ with δ we get that $\varepsilon = \sqrt{\frac{(b-a)^2 \ln(2/\delta)}{2|S_{\text{test}}|}}$ as claimed.

How to Use This Bound

Let us summarize. Assume at first that somebody hands you a function f and you have a data set S. Then you can assert

that

$$\mathbb{P}\Big[\ L_{\mathcal{D}}(f) > L_S(f) + \sqrt{\frac{(b-a)^2 \ln(2/\delta)}{2|S|}}\ \Big] \leq \delta.$$

In words, we can compute a probabilistic upper bound on the true risk since both $L_S(S)$ as well as the error term can be computed with the data at hand.
In the more general case, we are given a data set S and we want to first learn a function and then compute an upper bound on the true risk of the learned function. In this case we should split S into $S = S_{\text{train}} \cup S_{\text{test}}$. We then let $f_{S_{\text{train}}} = \mathcal{A}(S_{\text{train}})$. Then we can assert that

$$\mathbb{P}\Big[L_{\mathcal{D}}(f_{S_{\text{train}}}) > L_{S_{\text{test}}}(f_{S_{\text{train}}}) + \sqrt{\frac{(b-a)^2 \ln(2/\delta)}{2|S_{\text{test}}|}}\ \Big] \leq \delta.$$

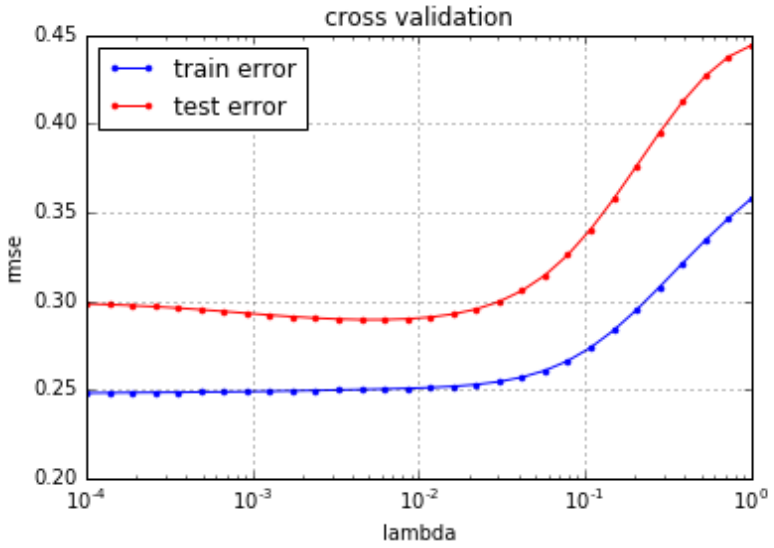
So also in this case do we get a a computable probabilistic upper bound. In the second case we pay a price for splitting the data and the bound will in general be less tight.

Model selection

Let us now get to our second, related, problem. We are looking for a way to select the hyperparameters of our model, like the parameter λ for the ridge regression problem.
We split our data into a training set S_{train} and a test set S_{test} and we think of them as having been generated independently and sampled according to the underlying but unknown distribution $\mathcal{D}$. We have in addition a set of values for a parameter of the model, e.g., the parameter λ in the ridge

regression problem. Let these values be λ_k, $k = 1, \ldots, K$. To keep things simple we assume that K is some finite value. We run the learning algorithm K times on the same training set S_{train} to compute the K prediction functions $f_{S_{\text{train}},\lambda_k}$. For each such prediction function we compute the test error $L_{S_{\text{test}}}(f_{S_{\text{train}},\lambda_k})$. We then choose that value of the parameter λ which gives us the smallest such test error.

In the figure below, we plot the test error (red) as well as the training error (blue) for many values of λ (grid search).



The same procedure can is applied to select other model hyperparameters, such as the degree in case of a polynomial feature expansion.

Model Selection Based on Test Error

If for all models $f_{S_{\text{train}},\lambda_k}$ the test error was exactly equal to the true error then it makes sense that we pick the best of these models. But we have seen that even if we compare the

test to the true error of a single function in general there is a difference (the generalization error). And in the model selection problem we use the same test set K times to compute the test error for each of the K models. How confident can we be that the suggested procedure gives us meaningful results? This is a slight extension of our previous analysis.
Assume hence that we have K models f_k given as candidates, $k = 1, \ldots, K$. Again assume that our loss function $\ell(\cdot, \cdot)$ is bounded in $[a, b]$. Given a test set S_{test} chosen i.i.d. from $\mathcal{D}$. what is the maximum generalization error, i.e., what is the maximum difference between the true error and the test error?
Similarly as in the case of a single model (3), we claim that we can now bound the maximum deviation for all K candidates, by

$$\mathbb{P}\Big[\ \max_k \big|L_{\mathcal{D}}(f_k) - L_{S_{\text{test}}}(f_k)\big| \geq \sqrt{\frac{(b-a)^2 \ln(2K/\delta)}{2|S_{\text{test}}|}}\ \Big] \leq \delta. \tag{4}$$

Insights: The error decreases as $\mathcal{O}(1/\sqrt{|S_{\text{test}}|})$ with the number test points. Now that we test K hyper-parameters, our error only goes up by a very small factor which is proportional to $\sqrt{\ln(K)}$. So we can test many different models without incurring a large penalty.
The proof of this statement follows (3), which has answered the special case of $K = 1$.

For a general K, if we check the deviations for K models and ask for the probability that for at least one such model

we get a deviation of at least ε then by the union bound this probability is at most K times as large as in the case where we are only concerned with a single instance. I.e., the upper bound becomes $2Ke^{-2|S_{\text{test}}|\varepsilon^2/(b-a)^2}$.
Hence, equating now $2Ke^{-2|S_{\text{test}}|\varepsilon^2/(b-a)^2}$ with δ we get that $\varepsilon = \sqrt{\frac{(b-a)^2 \ln(2K/\delta)}{2|S_{\text{test}}|}}$ as stated.
Let

$$k^* = \text{argmin}_k L_{\mathcal{D}}(f_k).$$

In words, f_{k^*} is that function that has the smallest *true* risk. Further, let

$$\hat{k} = \text{argmin}_k L_{S_{\text{test}}}(f_k).$$

In words, $\hat{k}$ is that function that has the smallest *empirical* risk. Then a little thought shows that

$$\mathbb{P}\Big[\ L_{\mathcal{D}}(f_{\hat{k}}) > L_{\mathcal{D}}(f_{k^*}) + 2\sqrt{\frac{(b-a)^2 \ln(2K/\delta)}{2|S_{\text{test}}|}}\ \Big] \leq \delta. \quad (5)$$

In words, if we choose the "best" function according to the empirial risk then its true risk is not too far away from the true risk of the optimal choice.

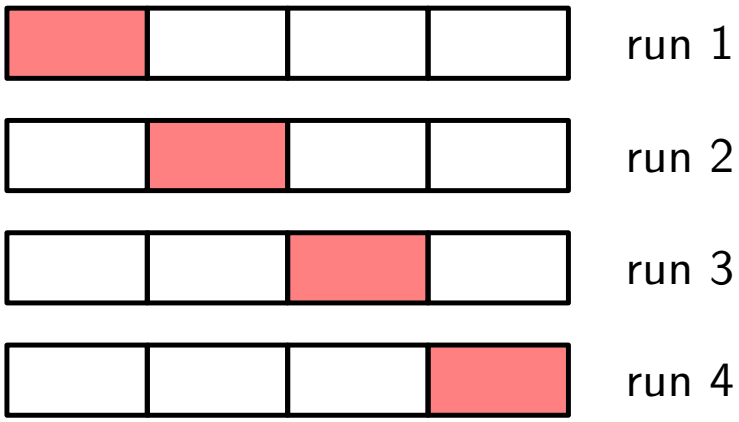
Extension to Infinitely Many Model Choices The basic idea of this bound can be carried over to the case where we have infinitely many models. In this case a more sophisticated concept, called the VC-dimension, is used: as long as we have models with a *finite* VC-dimension then the bound one can show has the same form, with K replaced by the VC dimension.

Cross-validation

Splitting the data once into two parts (one for training and one for testing) is not the most efficient way to use the data. Cross-validation is a better way:

K-fold cross-validation is a popular variant. Randomly partition the data into K groups. Now train K times. Each time leave out exactly one of the K groups for testing and use the remaining $K-1$ groups for training. Average the K results.

Note: Have used all data for training, and all data for testing, and used each data point the same number of times.



Cross-validation returns an unbiased estimate of the *generalization error* and its variance.

Additional Notes

Proof of Lemma 0.1

Instead of considering the setup in the lemma we can equivalently assume that $\mathbb{E}[\Theta] = 0$ and that the Θ_n take values in $[a, b]$, where $a \leq 0 \leq b$. We will show that

$$\mathbb{P}\{\frac{1}{N}\sum_{n=1}^{N}\Theta_n \geq \varepsilon\} \leq e^{-2N\varepsilon^2/(b-a)^2}.$$

This, together with the equivalent bound

$$\mathbb{P}\{\frac{1}{N}\sum_{n=1}^{N}\Theta_n \leq -\varepsilon\} \leq e^{-2N\varepsilon^2/(b-a)^2}$$

will prove the claim. We have

$$\begin{aligned}
\mathbb{P}\{\frac{1}{N}\sum_{n=1}^{N}\Theta_n \geq \varepsilon\} &\overset{s>0}{=} \mathbb{P}\{e^{s\frac{1}{N}\sum_{n=1}^{N}\Theta_n} \geq e^{s\varepsilon}\} \\
&\overset{(a)}{\leq} \min_{s>0}\mathbb{E}[e^{s\frac{1}{N}\sum_{n=1}^{N}\Theta_n}]e^{-s\varepsilon} \\
&\overset{(b)}{=} \min_{s>0}\prod_{n=1}^{N}\mathbb{E}[e^{\frac{s\Theta_n}{N}}]e^{-s\varepsilon} \\
&= \min_{s>0}\mathbb{E}[e^{\frac{s\Theta}{N}}]^N e^{-s\varepsilon} \\
&\overset{(c)}{\leq} \min_{s>0} e^{s^2(b-a)^2/(8N)} e^{-s\varepsilon} \\
&\overset{s=4N\varepsilon/(b-a)^2}{=} e^{-2N\varepsilon^2/(b-a)^2}.
\end{aligned}$$

Here, step (a) follows from the Markov inequality. In step (b) we have used the fact that the random variables Θ_n, and

hence the random variables $e^{\frac{s\Theta_n}{N}}$, are independent so that the expectation of the product is equal to the product of the expectations. Finally, in step (c) we have used the so-called Hoeffding lemma. It states that for any random variable X, with $\mathbb{E}[X] = 0$ and $X \in [a, b]$ we have

$$\mathbb{E}[e^{sX}] \leq e^{\frac{1}{8}s^2(b-a)^2}.$$

To give a rough outline, consider the convex function e^{sx}, $s \geq 0$. In the range $[a, b]$ it is upper bounded by the chord (the line that is equal to the function at the two boundaries)

$$e^{sx} \leq \frac{x-a}{b-a}e^{sb} + \frac{b-x}{b-a}e^{sa}.$$

If we now take the expectation with respect to X and recall that $\mathbb{E}[X] = 0$ by assumption then we get

$$\mathbb{E}[e^{sX}] \leq \frac{b}{b-a}e^{sa} - \frac{a}{b-a}e^{sb} \leq e^{s^2(b-a)^2/8}.$$

The last step on the right requires several steps but this is now a pure calculus problem and we skip the details.

Machine Learning Course - CS-433

Bias-Variance Decomposition

Oct 6, 2020

minor changes by Nicolas Flammarion 2020, minor changes by Rüdiger Urbanke 2019, changes by Martin Jaggi 2018, changes by Rüdiger Urbanke 2017

Last updated on: October 6, 2020

EPFL

Motivation

Last time we saw how to assess if a given function was good. In particular, we discussed how we can bound the difference between the true risk of the function and the emperical risk. We then used the same ideas and discussed how to choose the "best" out of a finite number of models. This led us to the idea of splitting the data into a *train* set and a *test* set. Our motivation for the model selection problem was that typically we need to optimize hyper-parameters. E.g., in the ridge regression problem the hyper-parameter was λ. These hyper-parameters often control the "complexity" of the class of models that we allow.

Today we will focus on how the risk (true or emperical) behaves as a function of the complexity of the model class. This will lead to the important concept of the bias - variance trade-off when we perform the model selection. It will help us to decide how "complex" or "rich" we should make our model.

Let us discuss a very simple example. Consider linear regression with a one-dimensional input and using polynomial feature expansion. The maximum degree d regulates the complexity of the class. We will see that the following is typically true.

Assume that we only allow simple models, i.e., we restrain the degree to be small:

- We then typically will get a large bias, i.e., a bad fit.
- On the other hand the variance of $L_{\mathcal{D}}(f_S)$ as a function of the random sample S is typically small.

We say that we have high bias but low variance.
Assume that we allow complex models, i.e., we allow large degrees:

- We then typically will find a model that fits the data very well. We will say that we have small bias.
- But we likely observe that the variance of $L_{\mathcal{D}}(f_S)$ as a function of the random sample S is large.

We say that we have low bias but high variance.

Data Generation Model

Assume that the data is generated as

$$y = f(\mathbf{x}) + \varepsilon,$$

where f is some (arbitrary and unknown) function and ε is additive *noise* with distribution $\mathcal{D}_\varepsilon$ that is independent from sample to sample and independent from the data. Assume the noise has zero mean (otherwise this constant can be absorbed into f). Note that f is in general not *realizable*, i.e., it is in general not in our model class.
We further assume that $\mathbf{x}$ is generated according to some fixed but unknown distribution $\mathcal{D}_{\mathbf{x}}$. Finally, we assume that the loss function $\ell(\cdot, \cdot)$ is the square loss. Let $\mathcal{D}$ denote the joint distribution on pairs $(\mathbf{x}, y)$.

Error Decomposition

As always, we have given some training data S_{train}, consisting of iid samples according to $\mathcal{D}$. Given our learning

algorithm $\mathcal{A}$, we compute the prediction function $f_{S_{\text{train}}} = \mathcal{A}(S_{\text{train}})$. We are ultimately interested in how the true error

$$\mathbb{E}_{\mathbf{x}\sim\mathcal{D}}[(f(\mathbf{x}) + \varepsilon - f_{S_{\text{train}}}(\mathbf{x}))^2]$$

behaves as a function of the training set S_{train} and the complexity of the model class.
But the decomposition we will discuss already applies "pointwise", i.e., for a single sample $\mathbf{x}$. It is therefore simpler if we fix $\mathbf{x}_0$, and only consider

$$(f(\mathbf{x}_0) + \varepsilon - f_{S_{\text{train}}}(\mathbf{x}_0))^2.$$

We imagine that we are running the experiment many times: we create S_{train}, we learn the model $f_{S_{\text{train}}}$, and then we evaluate the performance by computing the square loss for this fixed element $\mathbf{x}_0$.
So let us look at the expected value of this quantity:

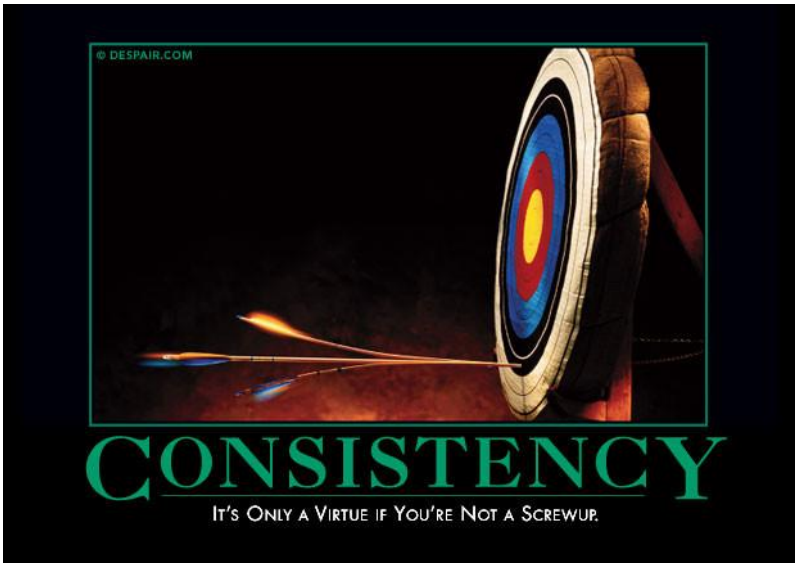
$$\mathbb{E}_{S_{\text{train}}\sim\mathcal{D},\, \varepsilon\sim\mathcal{D}_\varepsilon}[(f(\mathbf{x}_0) + \varepsilon - f_{S_{\text{train}}}(\mathbf{x}_0))^2].$$

We will now show that we can rewrite the above quantity as a sum of *three non-negative terms* and this decomposition

has a natural interpretation. We write

$$
\begin{aligned}
&\mathbb{E}_{S_{\text{train}} \sim \mathcal{D},\, \varepsilon \sim \mathcal{D}_\varepsilon}\big[(f(\mathbf{x}_0) + \varepsilon - f_{S_{\text{train}}}(\mathbf{x}_0))^2\big] \\
&\overset{(a)}{=} \mathbb{E}_{\varepsilon \sim \mathcal{D}_\varepsilon}[\varepsilon^2] + \mathbb{E}_{S_{\text{train}} \sim \mathcal{D}}\big[(f(\mathbf{x}_0) - f_{S_{\text{train}}}(\mathbf{x}_0))^2\big] \\
&\overset{(b)}{=} \mathrm{Var}_{\varepsilon \sim \mathcal{D}_\varepsilon}[\varepsilon] + \mathbb{E}_{S_{\text{train}} \sim \mathcal{D}}\big[(f(\mathbf{x}_0) - f_{S_{\text{train}}}(\mathbf{x}_0))^2\big] \\
&\overset{(c)}{=} \underbrace{\mathrm{Var}_{\varepsilon \sim \mathcal{D}_\varepsilon}[\varepsilon]}_{\text{noise variance}} \\
&\qquad + \underbrace{\big(f(\mathbf{x}_0) - \mathbb{E}_{S'_{\text{train}} \sim \mathcal{D}}[f_{S'_{\text{train}}}(\mathbf{x}_0)]\big)^2}_{\text{bias}} \\
&\qquad + \mathbb{E}_{S_{\text{train}} \sim \mathcal{D}}\Big[\underbrace{\big(\mathbb{E}_{S'_{\text{train}} \sim \mathcal{D}}[f_{S'_{\text{train}}}(\mathbf{x}_0)] - f_{S_{\text{train}}}(\mathbf{x}_0)\big)^2}_{\text{variance}}\Big].
\end{aligned}
$$

Note that here S'_{train} is a second training set, also sampled from $\mathcal{D}$ that is independent of the training set S_{train}.



Details:
In step (a), we omitted the third term

$$\mathbb{E}_{S_{\text{train}}\sim\mathcal{D},\,\varepsilon\sim\mathcal{D}_\varepsilon}\big[2\varepsilon(f(\mathbf{x}_0)-f_{S_{\text{train}}}(\mathbf{x}_0))\big].$$

But since the noise ε is independent from S_{train} we can first average over the noise, and by observing that the noise has mean zero, we see that this term is in fact zero.
Further, since the noise has zero mean, the second moment is equal to the variance. This explains step (b).
In step (c) we have added and substracted the constant term $\mathbb{E}_{S'_{\text{train}}\sim\mathcal{D}}[f_{S'_{\text{train}}}(\mathbf{x}_0)]$ to the expression and then expanded the square.
The expansion yields the two expressions which are stated (termed "bias" and "variance"). In addition it yields the cross term (to save space we omit the factor 2 and the $_{\text{'train'}}$-subscript)

$$\begin{aligned}
&\mathbb{E}_{S\sim\mathcal{D}}\Big[\big(f(\mathbf{x}_0)-\mathbb{E}_{S'\sim\mathcal{D}}[f_{S'}(\mathbf{x}_0)]\big)\cdot\big(\mathbb{E}_{S'\sim\mathcal{D}}[f_{S'}(\mathbf{x}_0)]-f_S(\mathbf{x}_0)\big)\Big]\\
&=\big(f(\mathbf{x}_0)-\mathbb{E}_{S'\sim\mathcal{D}}[f_{S'}(\mathbf{x}_0)]\big)\cdot\mathbb{E}_{S\sim\mathcal{D}}\big[(\mathbb{E}_{S'\sim\mathcal{D}}[f_{S'}(\mathbf{x}_0)]-f_S(\mathbf{x}_0))\big]\\
&=\big(f(\mathbf{x}_0)-\mathbb{E}_{S'\sim\mathcal{D}}[f_{S'}(\mathbf{x}_0)]\big)\cdot\big(\mathbb{E}_{S'\sim\mathcal{D}}[f_{S'}(\mathbf{x}_0)]-\mathbb{E}_{S\sim\mathcal{D}}[f_S(\mathbf{x}_0)]\big)\\
&=0.
\end{aligned}$$

Interpretation of Decomposition

Each of the three terms is non-negative. Hence each of them is a lower bound on the true error for the input $\mathbf{x}_0$.
The noise imposes a strict lower bound on what error we can achieve. This contribution is given by the term $\mathrm{Var}_{\varepsilon\sim\mathcal{D}_\varepsilon}[\varepsilon]$.
The bias term is the square of the difference between the actual value $f(\mathbf{x}_0)$ and the expected prediction $\mathbb{E}_{S'\sim\mathcal{D}}[f_{S'}(\mathbf{x}_0)]$, where the expectation is over the training sets. (E.g. simple models can not fit well, so have a large bias)
The variance term is the variance of the prediction function. If we consider very complicated models then small variations in the data set can produce vastly different models and our prediction for an input $\mathbf{x}_0$ will vary widely.

Examples

The following four figures are take from the book by James, Witten, Hastie, and Tibshirani (Introduction to Statistical Learning).
The first three pictures show three different functions each (the true function is the black curve). The first function has medium "complexity", the second is very simple, and the third is the most complicated. In each case, three different predictions are done based on models of increasing complexity.

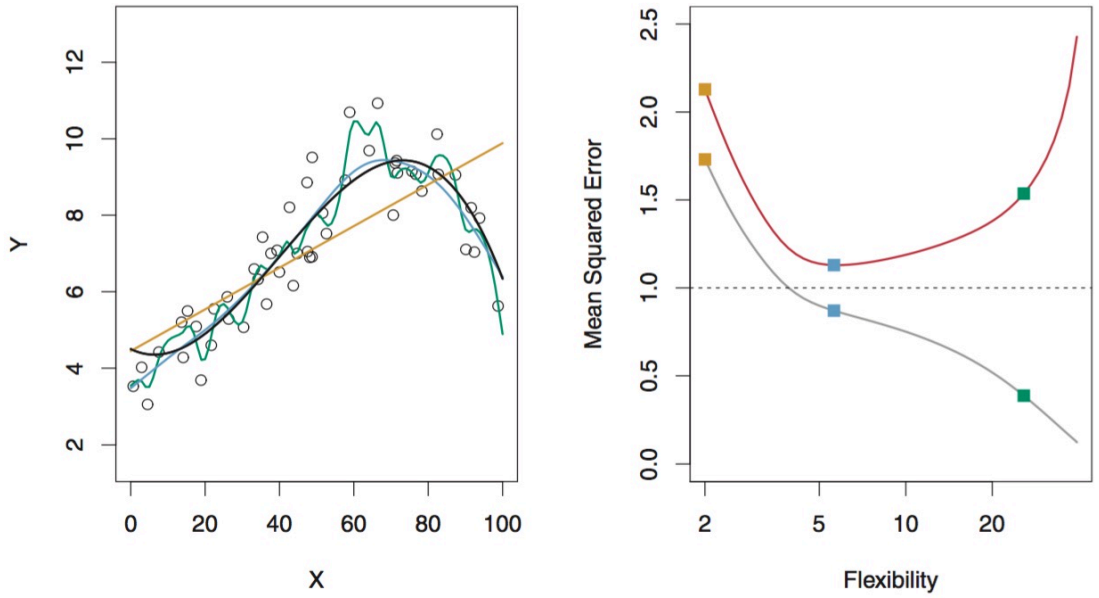


FIGURE 2.9. Left: *Data simulated from f, shown in black. Three estimates of f are shown: the linear regression line (orange curve), and two smoothing spline fits (blue and green curves).* Right: *Training MSE (grey curve), test MSE (red curve), and minimum possible test MSE over all methods (dashed line). Squares represent the training and test MSEs for the three fits shown in the left-hand panel.*

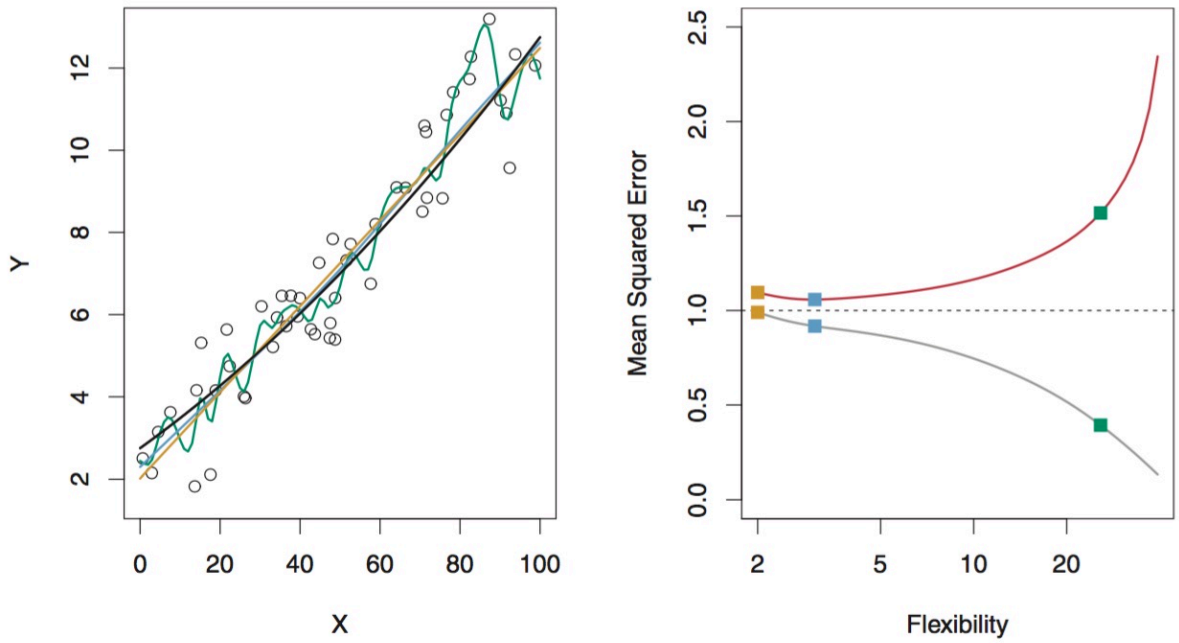


FIGURE 2.10. *Details are as in Figure 2.9, using a different true f that is much closer to linear. In this setting, linear regression provides a very good fit to the data.*

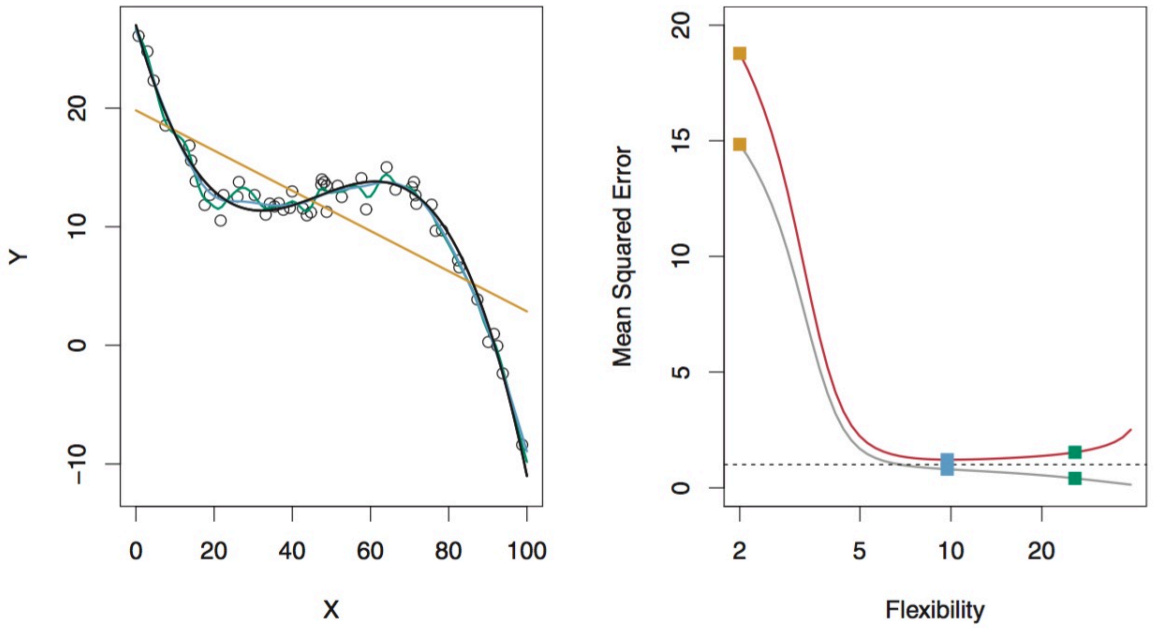


FIGURE 2.11. *Details are as in Figure 2.9, using a different f that is far from linear. In this setting, linear regression provides a very poor fit to the data.*

The final figure shows the bias-variance decomposition for each of these three models as a function of increasing complexity.

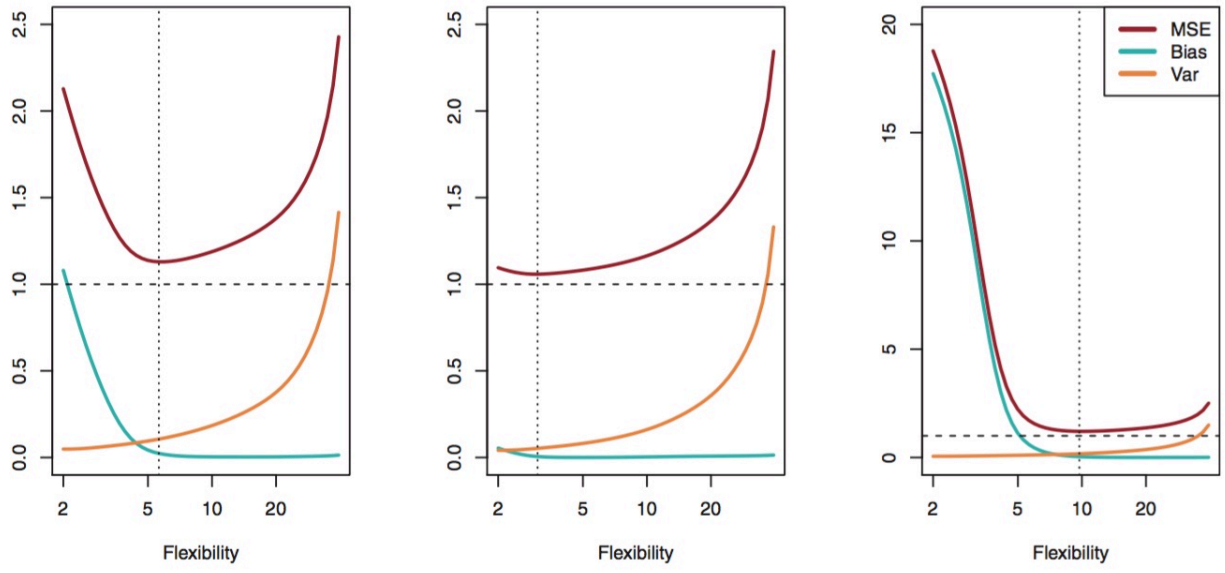


FIGURE 2.12. *Squared bias (blue curve), variance (orange curve), $Var(\epsilon)$ (dashed line), and test MSE (red curve) for the three data sets in Figures 2.9–2.11. The vertical dotted line indicates the flexibility level corresponding to the smallest test MSE.*

Additional Notes

You can find a very readable article about this topic by Scott Fortmann-Roe, here
scott.fortmann-roe.com/docs/BiasVariance.html
You can also find a nice article about the double descent phenomenon by Mikhail Belkin et al, here
https://www.pnas.org/content/116/32/15849.short

Machine Learning Course - CS-433

Classification

Oct 13, 2020

Minor changes by Nicolas Flammarion 2020; changes by Rüdiger Urbanke 2019,2018,2017,2016;

Last updated on: October 12, 2020

Classification

Similar to regression, classification relates the input variable $\mathbf{x}$ to the output variable y, but now y can only take on discrete values. We say that y is a *categorical* variable.

Binary classification

When y can only take on two values, it is called binary classification. Sometimes we refer to the two discrete values abstractly as $y \in \{\mathcal{C}_1, \mathcal{C}_2\}$. The $\mathcal{C}_i$ are called class labels or simply classes. Other times it is more conveniently to assume that $y \in \{-1, +1\}$ or $y \in \{0, 1\}$. Note that even if the class labels are real values, there is typically no ordering implied between the two classes.

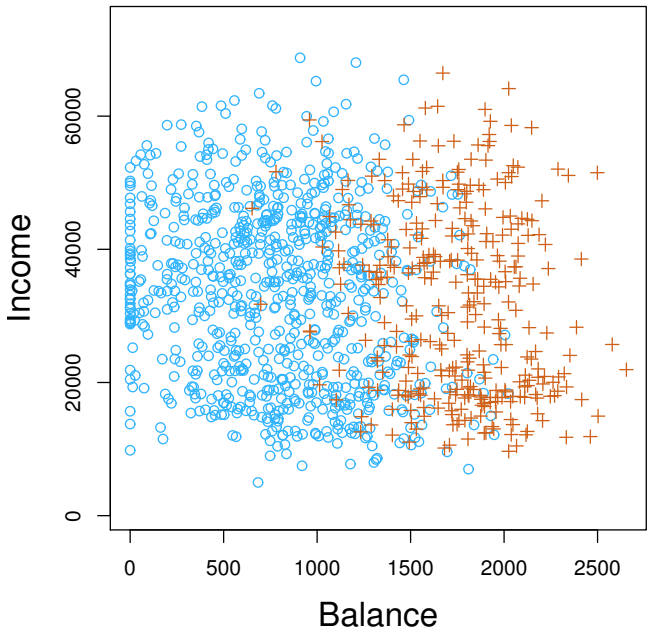
Multi-class classification

In a multi-class classification, y can take on more than two values, i.e., $y \in \{\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_{K-1}\}$ for a K-class problem. Again, even though there is in general no ordering among these classes, we sometimes use the labels $y \in \{0, 1, 2, \ldots, K-1\}$.

Examples of classification problems

A credit card service must be able to determine whether or not a requested transaction is fraudulent. They might have at their disposal the users IP address (if the transaction

happens on the web), past transaction history, and perhaps some other features.
An example from the book *Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman is shown below. We have at our disposal the annual incomes and monthly credit card balances of a number of individuals. The individuals who defaulted on their credit card payments are shown in orange, and those who did not default are shown in blue.



To consider another example, a person arrives at the emergency room with a set of symptoms that could possibly be attributed to one of three medical conditions. Which of the three conditions does the individual have?

Classifier

A classifier will divide the input space into a collection of regions belonging to each class. The boundaries of these re-

gions are called decision boundaries. A classifier can be linear or nonlinear. This distinction is less strict than it might seem at first. E.g., if you look at the classifier in the right-hand side of "Figure 4.1" you will see that the decision regions are non-linear (not straight lines). But in fact the classifier that led to these region is linear – we added some non-linear features to the original feature vector (think polynomial basis) before performing the linear classification. The decision boundaries appear non-linear since the plot is made in the original feature space and not the extended feature space.

What is the aim of classification?

In some situations we are interested in classification in itself. This means, we are constructing a predictor based on a training set and are interested in applying this predictor to "new" data. Consider e.g. the example of the credit card company that wants to predict if a customer will default or not.

But in some instances we are interested in more. We would like to "understand" the cause. Think e.g. of a disease prediction. Not only are we interested to know who is at risk but we would like to understand *why* somebody is at risk. So we are interested in the "interpretation" of the prediction. In particular for this second task it is important to have *simple* models and to have means of eliminating features that do not contribute significantly to the prediction.

Consider "Figure 4.12". In this data set various risk factors are given for a particular heart disease. These risk factors are blood pressure (sbp), tobacco usage, family history, obesity,

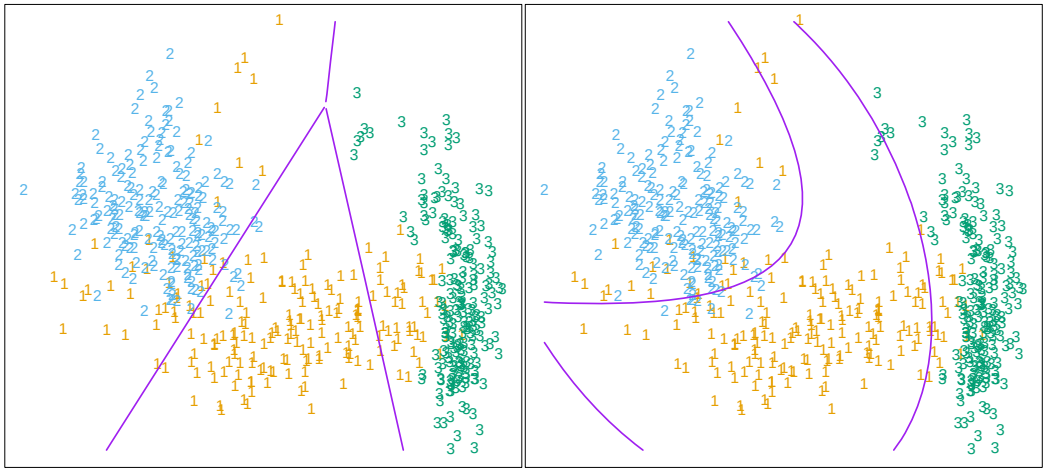


FIGURE 4.1. *The left plot shows some data from three classes, with linear decision boundaries found by linear discriminant analysis. The right plot shows quadratic decision boundaries. These were obtained by finding linear boundaries in the five-dimensional space* $X_1, X_2, X_1X_2, X_1^2, X_2^2$. *Linear inequalities in this space are quadratic inequalities in the original space.*

alcohol consumption, and age. For each pair of these risk factors the figure shows how the cases (people who have this disease) separate from the controls (people who do not have the desease). Such plots can help to decide which risk factors should be included in a model and which might have little predictive power.

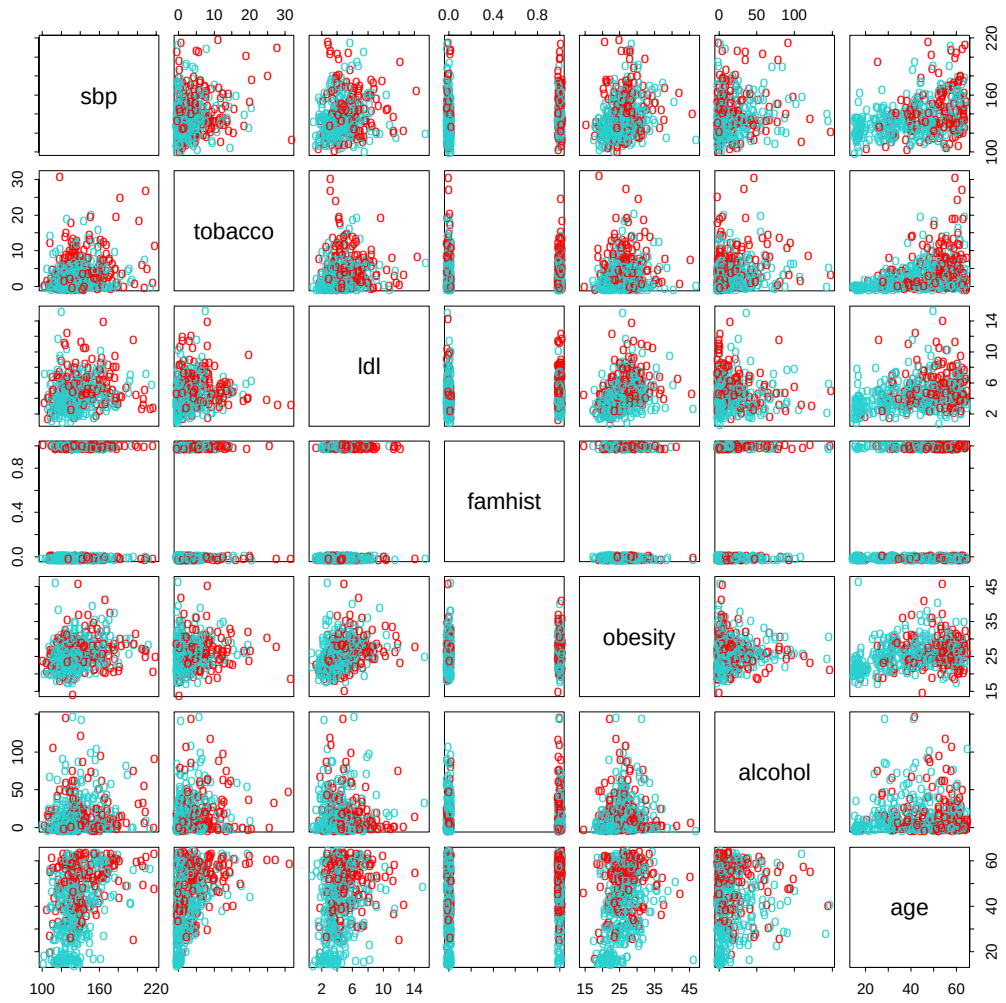
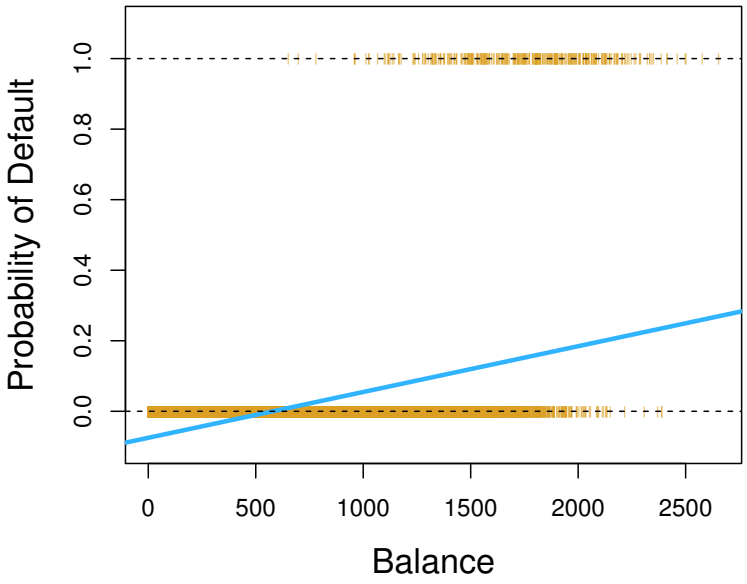


FIGURE 4.12. *A scatterplot matrix of the South African heart disease data. Each plot shows a pair of risk factors, and the cases and controls are color coded (red is a case). The variable* family history of heart disease (`famhist`) *is binary (yes or no).*

Classification as a special case of regression

From the very definition we see that classification is a *special case* of regression. It is a special case since the output is restricted to a small discrete set. So it might appear that there is not much new and that we should simply apply our

standard regression techniques to this special case.
E.g., we can assign $y = 0$ for $\mathcal{C}_1$ and $y = 1$ for $\mathcal{C}_2$ and, given a training set S_{train}, we can use (regularized) least-squares to learn a prediction function $f_{S_{\text{train}}}$ for this regression problem. To convert the regression into a classification it is then natural to decide on class $\mathcal{C}_1$ if $f_{S_{\text{train}}}(\mathbf{x}) < 0.5$ and $\mathcal{C}_2$ if $f_{S_{\text{train}}}(\mathbf{x}) > 0.5$.
In the figure below this approach is applied to the credit-card default problem. To keep things simple we use as feature only the *balance* (as we have seen in a previous plot the second feature (the *income*) does not contain much information).
We think of $y = 0$ as "no default" and $y = 1$ as "default". The dots we see corresponds to the various data points and all dots are either on the line $y = 0$ or $y = 1$. The horizontal axis corresponds to the input $\mathbf{x}$, the *balance*.
In the figure the output y is labeled as *probability*. This is just a convenient way of interpretation. Since the desired label y is either 0 or 1 we can think of y as the probability of a default.
So let us now run a regression on the training data S_{train}. To keep things simple we run a linear regression and learn the linear function $f_{S_{\text{train}}}(\mathbf{x})$. The result is the blue curve that is indicated.
We might want to interpret the value $f_{S_{\text{train}}}(\mathbf{x})$ as the probability of a default and then assign a label depending on whether this "probability" is smaller or larger than 0.5. Of course, this "probability" can be negative or be larger than 1 so such an interpretation has to be taken with a grain of salt.



Why classification is not just a special case of regression

It is not hard to see that this method can lead to questionable results. Even if the cases and controls are well separated, the general "position" of the line will depend crucially on how many points are in each class and where these points lie. E.g., if we add a few points with $y = 1$ and a very large balance this will shift the curve significantly even though only a few points changed. This is clearly not a desirable property.
Why does this happen? The squared loss function that we used for regression is not a good match to our objective. We would like that the *fraction of misclassified cases* is small. But the mean-squared error is only very loosely related to this objective. In particular, the mean-squared error counts positive and negative deviations from the class label equally bad, although only one of them can potentially lead to a misclassification. If we do have a very small mean-squared error then indeed we can guarantee a small classification error

but the opposite is not true – a regression function can have arbitrarily large mean-squared error even though the fraction of misclassified cases is arbitrarily small. We therefore might have to work "much harder" than we should in bringing down the mean-squared error and so to have a guarantee on the misclassification error.
Based on the above observation, we see that classification is not just a special form of regression with a simple loss function like the mean-squared error.

Some basic ideas of how to perform classification

Many different approaches have been developed over the years of how to efficiently perform classification. Our aim right now is not to give an exhaustive account of all possible techniques. Rather, let us quickly discuss some basic ideas. We will come back and discuss those in more detail in later lectures.

Nearest Neighbor

In some cases it is reasonable to postulate that inputs that are "close" are also likely to have the same label attached. Here "close" might e.g. be measured by the Euclidean distance. If we believe that this assumption is good, then, given an input $\mathbf{x}$ and a training set S_{train}, we can look for that point $\mathbf{x}^*$ which is closest to $\mathbf{x}$ and an element of S_{train} and then output y^*, the label attached to $\mathbf{x}^*$.

The good point about such a classifier is that it might work well even in cases where the decision boundaries are very irregular (see the Figure 2.3 below). But, as we will discuss in a later lecture, such a scheme fails miserably in high dimensions since in this case the geometry renders the notion of "close by" meaningless.

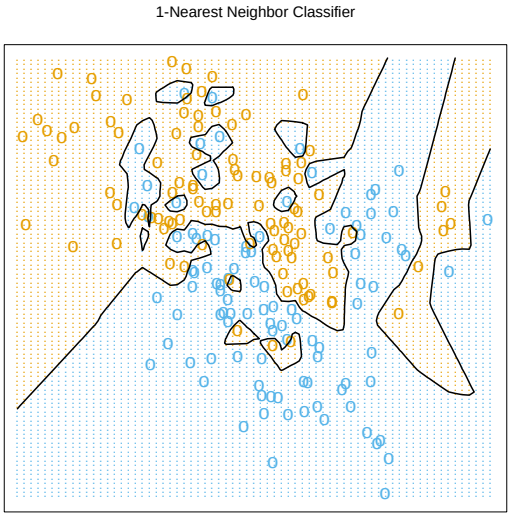


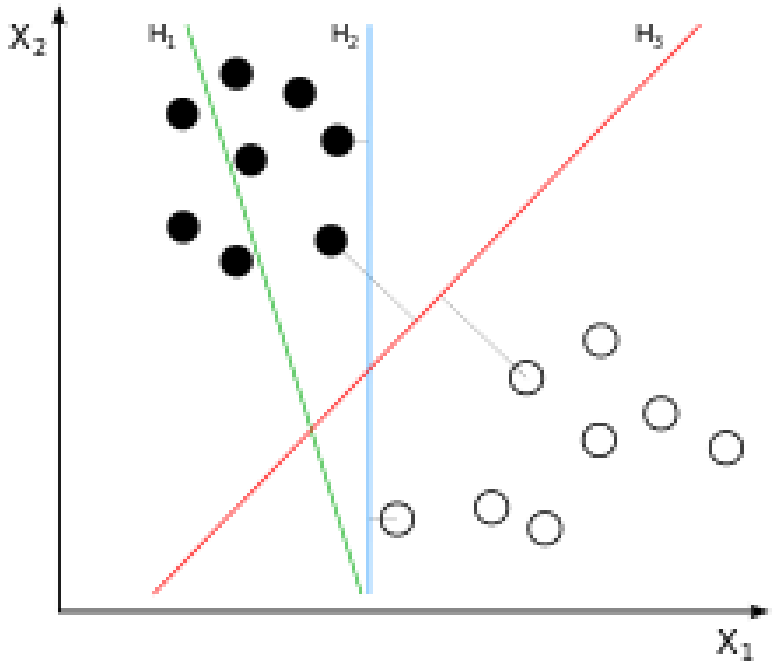
FIGURE 2.3. *The same classification example in two dimensions as in Figure 2.1. The classes are coded as a binary variable* (BLUE $= 0,$ ORANGE $= 1$)*, and then predicted by* 1*-nearest-neighbor classification.*

There are many natural generalizations of this concept. Instead of using a single neighbor we can use lets say the k nearest neighbors or we can take a weighted linear combination of elements in our neighborhood. The latter idea leads to to *smoothing kernels.*

Linear decision boundaries

One starting point is to assume that decision boundaries are linear (hyperplanes). Consider e.g. a binary classification problem and look at schemes where the boundary between

the two classes is a hyperplane. We can ask how to pick this boundary. To keep things simple, assume that there exists a "separating hyperplane" i.e., a hyperplane so that no point in the training set is misclassified.
In general, there might be many hyperplanes that do the trick (assuming there is at least one). So which one should we pick?
One idea is to pick a hyperplane so that the decision has as much "robustness/margin" with respect to the training set as is possible. I.e., if we slightly change the training set by "wiggling" the inputs we would like that the number of misclassifications stays low.
In the figure below (taken from Wikipedia) we see that H_1 does not separate the data, but H_2 and H_3 do. Between H_2 and H_3, H_3 is preferable, since it has a larger "margin." This idea will lead us to support vector machines (SVM) as well as logistic regression.



Non-linear decision boundaries

In many cases linear decision boundaries will not allow us to separate the data and non-linearities are needed. One option is to augment the feature vector with some non-linear functions. (The *kernel trick* is a method of doing this in an efficient way. We will learn about this at a later stage.)
Another way is to find an appropriate non-linear transform of the input so that the transformed input is then linearly separable. This is what is done when we are using *neural networks.*

Optimal classification for known generating model

It is instructive to think about how one could classify in an *optimal* fashion, i.e., how one could minimize the probability of misclassification, if the distribution of the generating model was known. To be concrete, assume that we know the joint distribution

$$p(\mathbf{x}, y)$$

and that y takes on elements in a discret set $\mathcal{Y}$.
Given the "observation" (input $\mathbf{x}$), let $\hat{y}(\mathbf{x})$ be our estimate of the class label. What is the optimum choice for this function? Note that our estimate is only a function of the input $\mathbf{x}$. Further, for a given input $\mathbf{x}$, the probability that the "correct" label is y is $p(y \mid \mathbf{x})$, according to our model. So if our estimate is $\hat{y}(\mathbf{x})$ then we will be correct a fraction $p(\hat{y}(\mathbf{x}) \mid \mathbf{x})$ of the time. We conclude that if we want to maximize the probability of guessing the correct label then we should chose

the decision rule

$$\hat{y}(\mathbf{x}) = \text{argmax}_{y \in \mathcal{Y}} p(y \mid \mathbf{x}).$$

This is called the maximum a-posteriori (MAP) criterion since we maximize the posterior probability (it is called posterior probability, since it is the probability of a class label *after* we have observed the input $\mathbf{x}$). This classifier is also called the Bayes classifier.
The probability of a correct guess is then the average (over all inputs $\mathbf{x}$) of this probability, i.e.,

$$\mathbb{P}\{\hat{y}(\mathbf{x}) = y\} = \int p(\mathbf{x}) p(\hat{y}(\mathbf{x}) \mid \mathbf{x}) dx.$$

In practice we do not know the joint distribution $p(\mathbf{x}, y)$. But we could use such an approach by using the data itself to learn the distribution (perhaps by assuming that the distribution is Gaussian and then just fitting the parameters from the data).

Machine Learning Course - CS-433

Logistic Regression

Oct 13, 2020

minor changes by Nicolas Flammarion 2020; changes by Rüdiger Urbanke 2019,2018,2017,2016;

Last updated on: October 12, 2020

EPFL

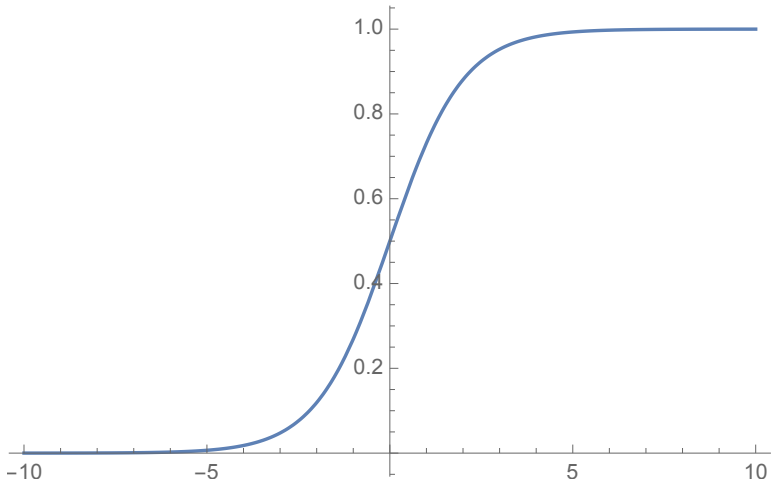
Logistic regression

Recall that in the previous lecture we discussed what happens if we treat binary classification as regression with lets say $y = 0$ and $y = 1$ as the two possible (target) values and then decide on the label by looking if the predicted value is smaller or larger than 0.5.
We have also discussed that it is tempting to interpret the predicted value as probability.
But there are problems: (i) the predicted values are in general not in $[0, 1]$; further, (ii) very large ($y \gg 1$) or very small ($y \ll 0$) values of the prediction will contribute to the error if we use the squared loss, even though they indicate that we are very confident in the resulting classification.
It is therefore natural that we *transform* the predictions that take values in $(-\infty, \infty)$ into a true probability by applying an appropriate function. There are several possible such functions. The *logistic function*

$$\sigma(z) := \frac{e^z}{1 + e^z}$$

is a natural and popular choice, see the next figure.[1]

[1] If you implement this function note that you are applying the exponential function to potentially large (in magnitude) values.



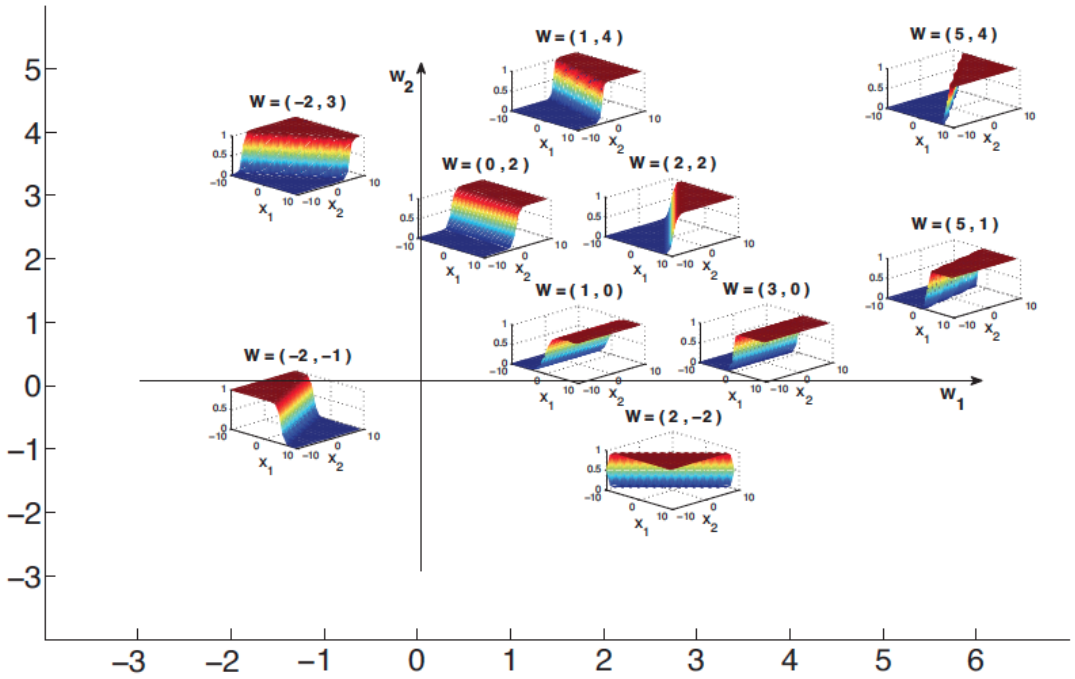
Consider the binary classification case and assume that our two class labels are $\{0, 1\}$. We proceed as follows. Given a training set S_{train} we learn a weight vector $\mathbf{w}$ (we will discuss how to do this shortly) and a "shift" (scalar) w_0. Given a "new" feature vector $\mathbf{x}$, we predict the (posterior) *probability* of the two class labels given $\mathbf{x}$ by means of

$$p(1 \mid \mathbf{x}, \mathbf{w}) = \sigma(\mathbf{x}^\top \mathbf{w} + w_0),$$
$$p(0 \mid \mathbf{x}, \mathbf{w}) = 1 - \sigma(\mathbf{x}^\top \mathbf{w} + w_0).$$

Note that we predict a real value (a probability) and not a label. This is the reason it is called logistic *regression*. But typically we use logistic regression as the first step of a classifier. In the second step we quantize the value to a binary value, typically according to whether the predicted prob-

This can lead to overflows. One work around is to implement this function by first checking the value of x and by treating large (in magnitude) values separately.

ability is smaller or larger than 0.5.
So very large and very small (large negative) values of $\mathbf{x}^\top \mathbf{w} + w_0$ correspond to probabilities $p(1 \mid \mathbf{x}, \mathbf{w})$ very close to 1 and 0, respectively.
The following figure visualizes the probabilities obtained for a 2-D problem (taken from KPM Chapter 7). More precisely, this is a case with two features and hence two weights that we learn. We see the effect of changing the weight vector on the resulting probability function.
It is easy to see what the roles of $\mathbf{w}$ and w_0 are. The vector $\mathbf{w}$ is orthogonal to the "surface of transition" and the w_0 allows us to shift the transition point along the vector $\mathbf{w}$. E.g., if $\mathbf{w} = (1, 0)$ and $w_0 = 0$ then the transition between the two levels happens at the $x_1 = 0$ plane. By scaling $\mathbf{w}$ we can make the transition faster or slower and by changing w_0 we can shift the decision region along the $\mathbf{w}$ vector.



At this point it is hopefully clear how we use logistic regression to do classification. To repeat, given the weight vector $\mathbf{w}$ we predict the probability of the class label 1 to be $p(1 \mid \mathbf{x}, \mathbf{w}) = \sigma(\mathbf{x}^\top \mathbf{w} + w_0)$ and then quantize. What we need to discuss next is how we learn the model, i.e., how we find a good weight vector $\mathbf{w}$ given some training set S_{train}.

A word about notation

In the beginning of this course we started with an arbitrary feature vector $\mathbf{x}$. We then discussed that often it is useful to add the constant 1 to this feature vector and we called the resulting vector $\widetilde{\mathbf{x}}$. We also discussed that often it is useful to add further features and we called then the resulting vec-

tor $\phi(\mathbf{x})$. Note that in particular for the logistic regression it is crucial that we have the constant term contained in $\mathbf{x}$ since this allows us to "shift" the decision region.
We will assume from now on that the vector $\mathbf{x}$ *always contains the constant term as well as any further features we care to add.* This will save us from a flood of notation.
Hence, from now on we no longer need the extra term w_0 but the term $\mathbf{x}^\top \mathbf{w}$ suffices since it contains already the constant.

Training

As always we assume that we have our training set S_{train}, consisting of iid samples $\{(\mathbf{x}_n, y_n)\}_{n=1}^N$, sampled according to a fixed but unknown distribution $\mathcal{D}$.
Exploiting that the samples $(\mathbf{x}_n, y_n)$ are independent, the probability of $\mathbf{y}$ (vector of all labels) given $\mathbf{X}$ (matrix of all inputs) and $\mathbf{w}$ (weight vector) has a

simple product form:

$$
\begin{aligned}
p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) &= \prod_{n=1}^{N} p(y_n|\mathbf{x}_n) \\
&= \prod_{n:y_n=1} p(y_n = 1|\mathbf{x}_n) \prod_{n:y_n=0} p(y_n = 0|\mathbf{x}_n) \\
&= \prod_{n=1}^{N} \sigma(\mathbf{x}_n^\top \mathbf{w})^{y_n} [1 - \sigma(\mathbf{x}_n^\top \mathbf{w})]^{1-y_n}.
\end{aligned}
$$

It is convenient to take the logarithm of this probability to bring it into an even simpler form. In addition we add a minus sign to the expression. In this way our objective will be to minimize the resulting cost function (rather than maximizing it). This is consistent with our previous examples, where we always minimized the cost function. We call the resulting cost function $\mathcal{L}(\mathbf{w})$,

$$
\begin{aligned}
\mathcal{L}(\mathbf{w}) &= -\sum_{n=1}^{N} y_n \ln \sigma(\mathbf{x}_n^\top \mathbf{w}) + (1 - y_n) \ln[1 - \sigma(\mathbf{x}_n^\top \mathbf{w})] \\
&= \sum_{n=1}^{N} \ln[1 + \exp(\mathbf{x}_n^\top \mathbf{w})] - y_n \mathbf{x}_n^\top \mathbf{w}.
\end{aligned}
$$

In the last step we have used the specific form of the logistic function $\sigma(x)$ to bring the cost function into a nice form.

Before we continue note the following. In principle we should have written down the likelihood of the data $(\mathbf{y}, \mathbf{X})$ given the parameter $\mathbf{w}$, i.e., $p(\mathbf{y}, \mathbf{X} \mid \mathbf{w})$. But

$$\begin{aligned} p(\mathbf{y}, \mathbf{X} \mid \mathbf{w}) &= p(\mathbf{X} \mid \mathbf{w}) p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) \\ &= p(\mathbf{X}) p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}), \end{aligned}$$

where in the second step we have made the natural assumption that the $\mathbf{X}$ data does not depend on the parameter we choose in our model. Note that this is an assumption and part of our model. But now note that the factor $p(\mathbf{X})$ is a constant wrt to the choice of $\mathbf{w}$, and hence plays no role when we apply the maximum likelihood criterion.

Maximum likelihood criterion

Recall what we did so far. Under the assumption that the samples are independent we have written down the likelihood of the data given a particular choice of weights $\mathbf{w}$. We then choose the weights $\mathbf{w}$ that maximize this likelihood.
Equivalently, we choose the weights that maximize the *log*-likelihood. This is called the *maximum-likelihood criterion*. In a final reformulation, we added a negative sign to bring the cost function to our standard form and called it $\mathcal{L}(\mathbf{w})$. In this form,

we are looking for the weights $\mathbf{w}$ that minimize $\mathcal{L}(\mathbf{w})$. In formulae, we choose the weight $\mathbf{w}^\star$, so that

$$\mathbf{w}^\star = \text{argmin}_{\mathbf{w}}\ \mathcal{L}(\mathbf{w}).$$

As we discussed in that context of the probabilistic interpretation of the least squares problem, one justification of the maximum-likelihood criterion is that, under some mild technical conditions, it is *consistent*. I.e., if we assume that the data was *generated* according a model in this class and we have iid samples and we use this procedure to estimate the underlying parameter, then our estimate will converge to the true parameter if we get more and more data. Of course, in practice the data is unlikely being generated in this way and there might not be *any* probabilistic model underlying it. But nevertheless, this gives our method a theoretical justification.

Conditions of optimality

As we want to minimize $\mathcal{L}(\mathbf{w})$, let us look at the stationary points of this function by computing the gradient, setting it to zero, and solving for $\mathbf{w}$. Note

that

$$\frac{\partial \ln[1+\exp(x)]}{\partial x} = \sigma(x).$$

Therefore

$$\begin{aligned}\nabla \mathcal{L}(\mathbf{w}) &= \sum_{n=1}^{N} \mathbf{x}_n(\sigma(\mathbf{x}_n^\top \mathbf{w}) - y_n) \\ &= \mathbf{X}^\top [\sigma(\mathbf{X}\mathbf{w}) - \mathbf{y}].\end{aligned}$$

Recall that by our convention the matrix $\mathbf{X}$ has N rows, one per input sample. Further, $\mathbf{y}$ is the column vector of length N which represents the N labels corresponding to each sample.
Therefore, $\mathbf{X}\mathbf{w}$ is a column vector of length N. The expression $\sigma(\mathbf{X}\mathbf{w})$ means that we apply the function σ to each of the N components of $\mathbf{X}\mathbf{w}$. In this manner we can express the gradient in a compact manner.
There is no closed-form solution for this equation. Let us therefore discuss how to solve this equation in an iterative fashion by using gradient descent or the Newton method.

Convexity

Since we are planning to iteratively minimize our cost function, it is good to know that this cost function is convex.

Lemma. *The cost function*

$$\mathcal{L}(\mathbf{w}) = \sum_{n=1}^{N} \ln[1 + \exp(\mathbf{x}_n^\top \mathbf{w})] - y_n \mathbf{x}_n^\top \mathbf{w}$$

is convex in the weight vector $\mathbf{w}$*.*

Proof. Recall that the sum (with non-negative weights) of any number of (strictly) convex functions is (strictly) convex. Note that $\mathcal{L}(\mathbf{w})$ is the sum of $2N$ functions. N of them have the form $-y_n \mathbf{x}_n^\top \mathbf{w}$, i.e., they are linear in $\mathbf{w}$ and a linear function is convex. Therefore it suffices to show that the other N functions are convex as well. Let us consider one of those. It has the form $\log[1 + \exp(\mathbf{x}_n^\top \mathbf{w})]$. Note that $\ln(1 + \exp(x))$ is convex – it has first derivative $\sigma(x)$ and second derivative

$$\frac{\partial^2 \ln(1+\exp(x))}{\partial x^2} = \frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x)), \quad (1)$$

which is non-negative.
The proof is complete by noting that $\ln[1+\exp(\mathbf{x}_n^\top \mathbf{w})]$ is the composition of a linear function with a convex function, and is therefore convex. □

Note: Alternatively, to prove that a function is convex (strictly convex) we can check that the Hessian (matrix consisting of second derivatives) is positive semi-definite (positive definite). We will do this shortly.

Gradient descent

As we have done for other cost functions, we can apply a (stochastic) gradient descent algorithm to minimize our cost function. E.g. for the batch version we can implement the update equation

$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \gamma^{(t)} \nabla \mathcal{L}(\mathbf{w}^{(t)}),$$

where $\gamma^{(t)} > 0$ is the step size and $\mathbf{w}^{(t)}$ is the sequence of weight vectors.

Newton's method

The gradient method is a *first-order* method, i.e., it only uses the gradient (the first derivative). We get a more powerful optimization algorithm if we use also the second order terms. Of course there is a trade-off. On the one hand we need fewer steps to converge if we use second order terms, on the other hand every iteration is more costly. Let us describe now a scheme that also makes use of second order terms. It is called *Newton's method*.

Hessian of the Log-Likelihood

Let us compute the *Hessian* of the cost function $\mathcal{L}(\mathbf{w})$, call it $\mathbf{H}(\mathbf{w})$. What is the Hessian? If $\mathbf{w}$

has D components then this is the $D \times D$ symmetric matrix with entries

$$\mathbf{H}_{i,j} = \frac{\partial^2 \mathcal{L}(\mathbf{w})}{\partial w_i \partial w_j}.$$

Recall that the cost function $\mathcal{L}(\mathbf{w})$ is a sum of N terms, all of the same form. So let us first compute the Hessian corresponding to one such term. We already computed the *gradient* of one such term and got

$$\mathbf{x}_n(\sigma(\mathbf{x}_n^\top \mathbf{w}) - y_n).$$

Recall, that this gradient is a vector of length D (the dimension of the feature vector $\mathbf{x}$ and hence also the dimension of the weight vector) where the i-th component is the derivative of $\mathcal{L}(\mathbf{w})$ with respect to $\mathbf{w}_i$. If you look at the above expression you see that this gradient is equal to $\mathbf{x}$ (a vector) times the scalar $(\sigma(\mathbf{x}_n^\top \mathbf{w}) - y_n)$. Note that $\mathbf{x}$ does not depend on $\mathbf{w}$ and neither does y_n. The only dependence on $\mathbf{w}$ is in the term $\sigma(\mathbf{x}_n^\top \mathbf{w})$. Therefore, the Hessian associated to one term will be

$$\mathbf{x}_n(\nabla \sigma(\mathbf{x}_n^\top \mathbf{w}))^\top.$$

We have already seen that $\sigma'(x) = \sigma(x)(1-\sigma(x))$. Therefore, by the chain rule one such term gives

rise to the Hessian

$$\mathbf{x}_n \mathbf{x}_n^\top \sigma(\mathbf{x}_n^\top \mathbf{w})(1 - \sigma(\mathbf{x}_n^\top \mathbf{w})).$$

It remains to do the sum over all N samples. Rather than just summing, let us put this again in a compact form by using the data matrix $\mathbf{X}$. We get

$$\mathbf{H}(\mathbf{w}) = \mathbf{X}^\top \mathbf{S} \mathbf{X},$$

where $\mathbf{S}$ is a $N \times N$ diagonal matrix with diagonal entries

$$S_{nn} := \sigma(\mathbf{x}_n^\top \mathbf{w})[1 - \sigma(\mathbf{x}_n^\top \mathbf{w})].$$

Note that the diagonal entries of $\mathbf{S}$ are non-negative. Hence $\mathbf{H}(\mathbf{w})$ is non-negative definite. This gives us an alternative proof that our original cost function is convex.

Newton's Method

Gradient descent uses only first-order information and takes steps in the direction opposite to the gradient. This makes sense since the gradient points in the direction of increasing function values and we want to *minimize* the function.
Newton's method uses second-order information and takes steps in the direction that minimizes a

quadratic approximation. More precisely, it approximates the function locally by a quadratic form and then moves in the direction where this quadratic form has its minimum. The update equation is of the form

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \gamma^{(t)}(\mathbf{H}^{(t)})^{-1}\nabla\mathcal{L}(\mathbf{w}^{(t)}).$$

Where does this update equation come from?
Recall that the Taylor series approximation of a function (up to second order terms) around a point $\mathbf{w}^\star$ has the form

$$\begin{aligned}\mathcal{L}(\mathbf{w}) \approx \mathcal{L}(\mathbf{w}^\star) + \nabla\mathcal{L}(\mathbf{w}^\star)^\top(\mathbf{w} - \mathbf{w}^\star) \\ + \frac{1}{2}(\mathbf{w} - \mathbf{w}^\star)^\top\mathbf{H}(\mathbf{w}^\star)(\mathbf{w} - \mathbf{w}^\star).\end{aligned}$$

The right-hand side is a *local approximation* of $\mathcal{L}(\mathbf{w})$. Assume that we take the right-hand side to be an exact representation of our cost function. We want to minimize this function. So let us look where the right-hand side takes its minimum value. If we think that this approximation is reasonably good, then it makes sense to move the new weight vector to the position of this minimum.
Let us take the gradient of the right hand side and set it to zero. We get

$$\nabla\mathcal{L}(\mathbf{w}^\star) + \mathbf{H}(\mathbf{w}^\star)(\mathbf{w} - \mathbf{w}^\star) = \mathbf{0}.$$

Solving for $\mathbf{w}$ gives us $\mathbf{w} = \mathbf{w}^\star - \mathbf{H}(\mathbf{w}^\star)^{-1}\nabla\mathcal{L}(\mathbf{w}^\star)$. This corresponds exactly to the stated update equation, except that in this update we have an extra step size γ. Why do we need this factor?
Recall that the right-hand side is only an approximation. Caution therefore dictates that we only move part of the way to the indicated minimum.

Regularized Logistic Regression

Although the cost-function for logistic regression is lower bounded by 0 we get issues if the data is linearly separable. In this case there is no finite-weight vector $\mathbf{w}$ which gives us this minimum cost function and if we continue to run the optimization the weights will tend to infinity.
To avoid this problem, as for standard regression problems, we can add a penalty term. E.g., we consider the cost function

$$\operatorname{argmin}_{\mathbf{w}} - \sum_{n=1}^{N} \ln p(y_n \,|\, \mathbf{x}_n^\top \mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2.$$

When the data is linearly separable, even if the gradient descent algorithm is not converging to a finite-weight vector, the direction in which the iterates are diverging to infinity is still very informative: the predictor converges to the direction of the

max-margin solution. You can find a very interesting and related article by Nati Srebro here https://www.jmlr.org/papers/volume19/18-188/18-188.pdf

Machine Learning Course - CS-433

Exponential Families and Generalized Linear Models

Oct 20nd, 2020

minor changes by Nicolas Flammarion 2020; changes by Rüdiger Urbanke 2019,2018,2017,2016;

Last updated on: October 18, 2020

EPFL

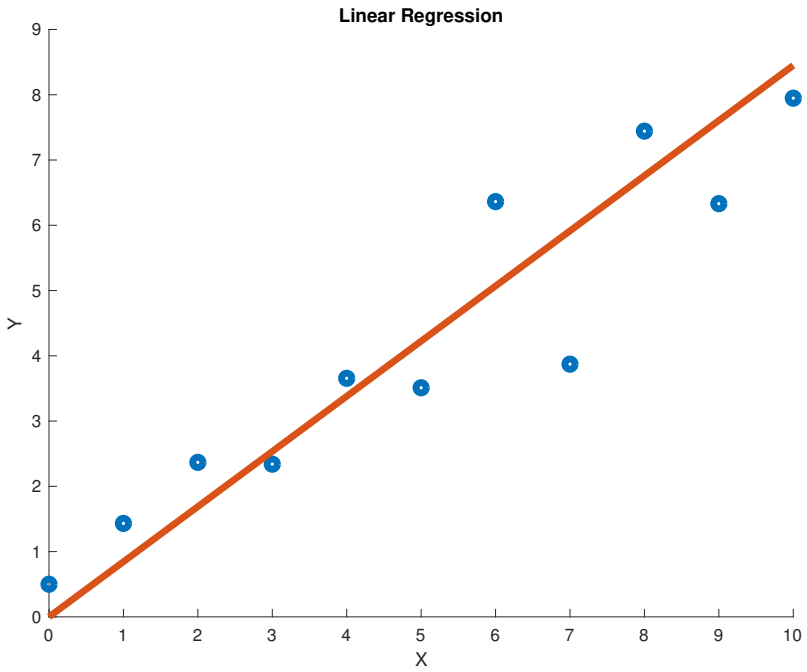


Figure 1:

Motivation

Let us go back to regression. Consider the very simple one-dimensional example in Fig. 1. The horizontal axis represents the input x and the vertical axis the output y. Our aim is to find a model for this data. It is very natural in this case that we try a linear model: $y = xw_1 + w_0 + Z$. I.e., we model the data as a line plus noise. Perhaps the most natural choice for the noise is a zero-mean Gaussian with some variance σ^2. As we discussed, this leads to least squares, assuming that we think of the data samples as independent and that we maximize the likelihood. This is what is typically meant when people talk about linear models (of course the data could be higher dimensional).

Now consider the data given in Fig. 2. In this case a linear

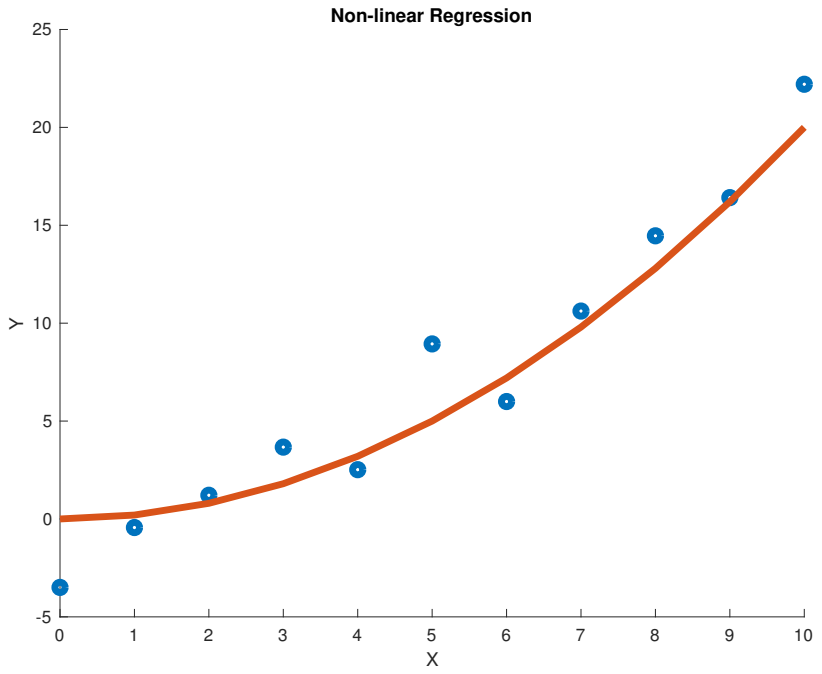


Figure 2:

model would not be a good fit. We have seen how we can get around this problem. Just add some additional features, e.g., x^2 and x^3. If we now use again a linear model, but in the extended feature space then we should be able to model the data well. So the idea was to augment or transform the feature space.

But this is not the only option we have. Note that in the example above the linear model predicts the *mean* of a distribution from which we then assume the data was sampled. Explicitly, we had $y = xw_1 + w_0 + Z$, where $xw_1 + w_0$ is the prediction of the linear model and represents the mean (i.e., the putatively "true" value for this data point) and then we get a noisy version as a sample. Here is now the extra degree of freedom we have: Instead of using the linear model to predict the mean of the distribution we can use it to predict

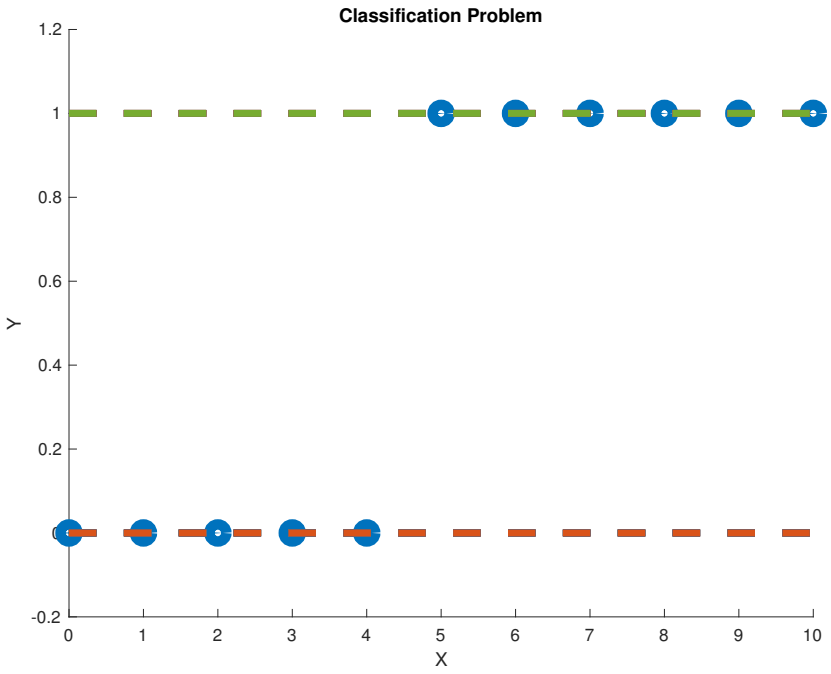


Figure 3:

a different quantity.

We have already seen an example when we talked about logistic regression. Consider the data given in Fig 3, where all the y values are in $\{0, 1\}$. This might correspond to a binary classification problem. Recall that in logistic regression we model the probability of the two classes $\{0, 1\}$ given the data $\mathbf{x}$ by

$$p(y = 1|\eta) = \sigma(\eta),$$
$$p(y = 0|\eta) = 1 - \sigma(\eta),$$

where η as a shorthand for $\mathbf{x}^\top \mathbf{w}$. This can be written compactly as

$$p(y|\eta) = \frac{e^{\eta y}}{1 + e^{\eta}} = \exp\left[\eta y - \log(1 + e^{\eta})\right],$$

where $y \in \{0, 1\}$. Note that linear model predicts η, $\eta = \mathbf{x}^\top \mathbf{w}$, and that η is *not* the mean of the distribution. Rather, η is related to the mean μ by the non-linear relation $\eta = \ln \frac{\mu}{1-\mu}$ or $\mu = \sigma(\eta)$. This relation between the parameter we predict by the linear model and the mean is called the *link function*. It is exactly this nonlinear link function that makes it possible to use a linear model in this context.

Outline

As you can see, we rewrote this distribution used in logistic regression in a very specific form. Our aim for today will be to generalize this form. We will see that there are many other distributions that can be written in this form. This will lead us to the class of distributions known as exponential families. We will first spend some time to talk about this family. We will see that many distributions (but not all) fit into this framework and that distributions in this family have many nice properties. We will only discuss some of these properties. Exponential families are also those distributions that have maximum entropy given some moment constraints and they are extremal also in other contexts. You are likely going to come across this family in other courses, and you will definitely see them if later on you will work in this area. As a second step we then discuss how exponential families can be used in the context of ML. In essence, by using different families we use different link functions, i.e., different relationships between the paramter η that the linear model predicts and the mean of the distribution. And this degree of freedom can be useful when we are trying to find a good

model for a given set of data.
In the subsequent discussion we consider various exponential families and then compute the corresponding link functions. But conceptually it can also be fruitful to think in the reverse way. What should the relationship be between the parameter that the linear model predicts and the mean of the distribution in order to fit the data well. I.e., perhaps we start with a desired link function and then find the exponential family that gives us this relationship.

Exponential family – Definition

Let y be a scalar and $\boldsymbol{\eta}$ be a vector. We will say that a distribution belongs to the *exponential family* if it can be written in the form

$$p(y|\boldsymbol{\eta}) = h(y) \exp\left[\boldsymbol{\eta}^\top \boldsymbol{\phi}(y) - A(\boldsymbol{\eta})\right]. \tag{1}$$

Let us look at the various components of this distribution. The parameter vector, η is often referred to as the natural or canonical parameter. The quantity $\boldsymbol{\phi}(y)$ is in general a vector and it is called a *sufficient statistics*. Why is $\boldsymbol{\phi}(y)$ called a sufficient statistics? Assume that we are given independent samples from this distribution. We do know $\boldsymbol{\phi}(y)$ and $h(y)$ but we do not know the parameter η. It turns out that in order to optimally estimate $\boldsymbol{\eta}$ given these samples all we need is the empirical average of the $\boldsymbol{\phi}(\mathbf{y})$. In other words, $\boldsymbol{\phi}(\mathbf{y})$ contains all the relevant information.
Note that the expression in (1) is non-negative if $h(y) \geq 0$. So we only need to ensure that it is properly normalized, i.e.,

we require that

$$\int_y h(y) \exp\left[\boldsymbol{\eta}^\top \boldsymbol{\phi}(y) - A(\boldsymbol{\eta})\right] dy] = 1.$$

Rewriting this we see that

$$\int_y h(y) \exp\left[\boldsymbol{\eta}^\top \boldsymbol{\phi}(y)\right] dy] = e^{A(\boldsymbol{\eta})}. \tag{2}$$

We see from the last expression that the only role of $A(\boldsymbol{\eta})$ is to ensure a proper normalization. $A(\boldsymbol{\eta})$ is sometimes called the *cumulant* and some times it is called the *log partition* function. We will see shortly that despite the fact that $A(\boldsymbol{\eta})$ is *only* there for normalization purposes it plays a crucial role and contains valuable information.

If you look at the definition of the exponential family, you will see that we have several "degrees of freedom" to define an element of the family. We can choose the factor $h(y)$, we can choose the vector $\boldsymbol{\phi}(y)$, and we can choose the parameter $\boldsymbol{\eta}$. For every choice we will get an element of the exponential family. The term $A(\boldsymbol{\eta})$ is then determined for each such choice and ensures that the expression is properly normalized as dicussed. Of course it can happen that for some parameters $\boldsymbol{\eta}$, $h(y) \exp\left[\boldsymbol{\eta}^\top \boldsymbol{\phi}(y)\right]$ is such that we cannot normalize the expression because the integral is infinity. E.g., set $h(y) = 1$, $\boldsymbol{\phi}(y) = y^2$ and $\boldsymbol{\eta} = 1$. We will exclude such parameters by only looking at the set of parameters

$$M := \{\boldsymbol{\eta} : \int_y h(y) \exp\left[\boldsymbol{\eta}^\top \boldsymbol{\phi}(y)\right] dy] < \infty\}.$$

As a final remark concerning $A(\boldsymbol{\eta})$ note that from (2) we have

$$A(\boldsymbol{\eta}) = \ln\Big[\int_y h(y)\exp\left[\boldsymbol{\eta}^\top \boldsymbol{\phi}(y)\right] dy\Big]. \tag{3}$$

Exponential family – Examples

Let us look at a few examples which are probably familiar to you but you might not have seen them written in this form. *Example:* We claim that the Bernoulli distribution is a member of the exponential family. We write

$$\begin{aligned} p(y|\mu) &= \mu^y(1-\mu)^{1-y}, \text{ where } \mu \in (0,1) \\ &= \exp\left[(\ln\frac{\mu}{1-\mu})y + \ln(1-\mu)\right] \\ &= \exp\left[\eta\phi(y) - A(\eta)\right]. \end{aligned}$$

Mapping this to (1) we see that

$$\begin{aligned} \phi(y) &= y, \\ \eta &= \ln\frac{\mu}{1-\mu}, \\ A(\eta) &= -\ln(1-\mu) = \ln(1+e^\eta), \\ h(y) &= 1. \end{aligned}$$

In this case $\phi(y)$ is a scalar, reflecting the fact that this family only depends on a single parameter. In fact, we have a 1-1 relationship between η and μ,

$$\eta = g(\mu) = \ln\frac{\mu}{1-\mu} \iff \mu = g^{-1}(\eta) = \frac{e^\eta}{1+e^\eta}.$$

As we mentioned in the very beginning, this function g is known as the *link* function (it links the mean of $\boldsymbol{\phi}(y)$ to the parameter η.)
Note that this is *exactly* the same distribution that we encountered when we discussed *logistic regression*.
Example: Consider the Poisson distribution with mean μ. We have, for $y \in \mathbb{N}$,

$$\begin{aligned} p(y|\mu) &= \frac{\mu^y e^{-\mu}}{y!} \\ &= \frac{1}{y!} e^{y\ln(\mu)-\mu} \\ &= h(y)e^{\eta\phi(y)-A(\eta)}, \end{aligned}$$

where $h(y) = 1/y!$, $\phi(y) = y$, $\eta = g(\mu) = \ln(\mu)$, and $\mu = g^{-1}(\eta) = e^{\eta}$. Here again, $g(\mu)$ *links* the mean to the parameter η.
Example: The Gaussian distribution with mean μ and variance σ^2 as parameters is also a member of the exponential family. We write

$$\begin{aligned} p(y|\mu) &= \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}, \mu \in \mathbb{R}, \sigma^2 \in \mathbb{R}^+ \\ &= \exp\left[(\mu/\sigma^2, -1/(2\sigma^2))(y, y^2)^\top - \frac{\mu^2}{2\sigma^2} - \frac{1}{2}\ln(2\pi\sigma^2)\right]. \end{aligned}$$

Mapping this again to (1) we see that

$$\begin{aligned}
\boldsymbol{\phi}(y) &= (y, y^2)^\top \\
\boldsymbol{\eta} &= (\eta_1 = \mu/\sigma^2, \eta_2 = -1/(2\sigma^2))^\top, \\
A(\boldsymbol{\eta}) &= \frac{\mu^2}{2\sigma^2} + \frac{1}{2}\ln(2\pi\sigma^2), \\
&= -\frac{\eta_1^2}{4\eta_2} - \frac{1}{2}\ln(-\eta_2/\pi), \\
h(y) &= 1.
\end{aligned}$$

Note that this time $\boldsymbol{\phi}(y)$ is a vector of length two, reflecting the fact that the distribution depends on two parameters. In fact, we have the 1-1 relationship between $\boldsymbol{\eta} = (\eta_1, \eta_2)$ and (μ, σ^2).

$$\eta_1 = \frac{\mu}{\sigma^2}; \eta_2 = -\frac{1}{2\sigma^2} \iff \mu = -\frac{\eta_1}{2\eta_2}; \sigma^2 = -\frac{1}{2\eta_2}.$$

Basic Properties

Convexity of $A(\boldsymbol{\eta})$

Lemma. *The cumulant $A(\boldsymbol{\eta})$ is convex as a function of $\boldsymbol{\eta}$ on M (the set of parameters $\boldsymbol{\eta}$ where the cumulant is finite).*

Proof. Let $\boldsymbol{\eta}_1$ and $\boldsymbol{\eta}_2$ be two parameters in M. Define $\boldsymbol{\eta} = \lambda\boldsymbol{\eta}_1 + (1-\lambda)\boldsymbol{\eta}_2$. We start with (2) and apply Hoelder's inequality. Recall that Hoelder's inequality reads $\|fg\|_1 \leq \|f\|_p\|g\|_q$, where $p, q \in [1, \infty]$ and $1/p + 1/q = 1$. Here,

$$\|f\|_p = \Big(\int |f(y)|^p dy\Big)^{\frac{1}{p}}.$$

You might not have seen Hoelder's inequality before, but you surely have seen the special case when $p = q = 2$. In this case you get the Cauchy-Schwarz inequality.
Let us go back to the proof. Pick $p = 1/\lambda$ and $q = 1/(1-\lambda)$. Then $p, q \in [1, \infty]$ and $1/p + 1/q = \lambda + (1 - \lambda) = 1$. We have

$$
\begin{aligned}
& e^{A(\boldsymbol{\eta})} \\
=& \int_y h(y) \exp\left[\boldsymbol{\eta}^\top \boldsymbol{\phi}(y)\right] dy \\
=& \int_y \underbrace{[h(y)^\lambda \exp\left[\lambda \boldsymbol{\eta}_1^\top \boldsymbol{\phi}(y)\right]]}_{f(y)} \underbrace{[h(y)^{1-\lambda} \exp\left[(1-\lambda)\boldsymbol{\eta}_2^\top \boldsymbol{\phi}(y)\right]]}_{g(y)} dy \\
\leq& (\int_y h(y) \exp\left[\boldsymbol{\eta}_1^\top \boldsymbol{\phi}(y)\right] dy)^\lambda (\int_y h(y) \exp\left[\boldsymbol{\eta}_2^\top \boldsymbol{\phi}(y)\right] dy)^{1-\lambda} \\
=& e^{\lambda A(\boldsymbol{\eta}_1)} e^{(1-\lambda) A(\boldsymbol{\eta}_2)}.
\end{aligned}
$$

Taking the log of this chain proves the claim,

$$A(\boldsymbol{\eta}) \leq \lambda A(\boldsymbol{\eta}_1) + (1 - \lambda) A(\boldsymbol{\eta}_2).$$

□

Derivatives of $A(\boldsymbol{\eta})$ and moments

Another useful property is that the gradient and Hessian (first and second derivatives) of $A(\boldsymbol{\eta})$ are related to the mean and the variance of $\boldsymbol{\phi}(y)$.

Lemma.

$$
\begin{aligned}
\nabla A(\boldsymbol{\eta}) &= \mathbb{E}[\boldsymbol{\phi}(y)], \\
\nabla^2 A(\boldsymbol{\eta}) &= \mathbb{E}[\boldsymbol{\phi}(y)\boldsymbol{\phi}(y)^\top] - \mathbb{E}[\boldsymbol{\phi}(y)]\mathbb{E}[\boldsymbol{\phi}(y)]^\top.
\end{aligned}
$$

Note that this in particular shows that the Hessian of $A(\boldsymbol{\eta})$ is a covariance matrix and hence is positive semi-definite. This gives us a second proof that $A(\boldsymbol{\eta})$ is convex.
Before we prove this, let us check this for our two running examples. Recall that for the Bernoulli distribution $\boldsymbol{\phi}(y)$ is a scalar, namely y. So in this case the first derivative should be the mean of the Bernoulli distribution and the second derivative the variance. Let us verify this. We get

$$\frac{dA(\eta)}{d\eta} = \frac{d\ln(1+e^{\eta})}{d\eta} = \frac{e^{\eta}}{1+e^{\eta}} = \sigma(\eta) = \mu,$$
$$\frac{d^2A(\eta)}{d\eta^2} = \frac{d\sigma(\eta)}{d\eta} = \sigma(\eta)(1-\sigma(\eta)) = \mu(1-\mu),$$

which confirms the claim.
For the Gaussian distribution our vector $\boldsymbol{\phi}(y)$ is of the form $(y, y^2)^{\top}$. So the first derivative (gradient) should give us the mean and the second moment of the Gaussian. The second derivative should give us the variance of various moments of y. We get

$$\frac{\partial A(\boldsymbol{\eta})}{d\eta_1} = \frac{\partial(-\frac{\eta_1^2}{4\eta_2} - \frac{1}{2}\ln(-\eta_2/\pi))}{\partial\eta_1} = -\frac{\eta_1}{2\eta_2} = \mu,$$
$$\frac{\partial A(\boldsymbol{\eta})}{d\eta_2} = \frac{\partial(-\frac{\eta_1^2}{4\eta_2} - \frac{1}{2}\ln(-\eta_2/\pi))}{\partial\eta_2} = (\frac{\eta_1^2 - 2\eta_2}{4\eta_2^2}) = \mu^2 + \sigma^2,$$

which are exactly the expected value and the second moment of y, as claimed. To do one more computation, let us

compute

$$\frac{\partial^2 A(\boldsymbol{\eta})}{d\eta_1^2} = \frac{\partial(-\frac{\eta_1}{2\eta_2})}{\partial \eta_1} = -\frac{1}{2\eta_2} = \sigma^2,$$

which is the variance of y, again as expected.

Proof. Let us just write down the proof regarding the first derivative. The proof for the second derivative proceeds in a similar fashion. We have

$$\begin{aligned}
\nabla A(\boldsymbol{\eta}) &= \nabla \ln[\int_y h(y) \exp\left[\boldsymbol{\eta}^\top \boldsymbol{\phi}(y)\right] dy] \\
&= \frac{\int_y \nabla h(y) \exp\left[\boldsymbol{\eta}^\top \boldsymbol{\phi}(y)\right] dy}{\int_y h(y) \exp\left[\boldsymbol{\eta}^\top \boldsymbol{\phi}(y)\right] dy} \\
&= \frac{\int_y h(y) \exp\left[\boldsymbol{\eta}^\top \boldsymbol{\phi}(y)\right] \boldsymbol{\phi}(y) dy}{\exp(A(\boldsymbol{\eta}))} \\
&= \int_y h(y) \exp\left[\boldsymbol{\eta}^\top \boldsymbol{\phi}(y) - A(\boldsymbol{\eta})\right] \boldsymbol{\phi}(y) dy \\
&= \mathbb{E}[\boldsymbol{\phi}(y)].
\end{aligned}$$

In the second step we have exchange the derivative with the integral. Note that the exchange of differentiation and integration is permitted if the resulting integral is finite (which it is in our case).

□

Link function

As we have seen already in two specific cases (Bernoulli and Poisson), there is a relationship between the "mean" $\boldsymbol{\mu} :=$

$\mathbb{E}[\boldsymbol{\phi}(y)]$ and $\boldsymbol{\eta}$ defined using a so-called *link function* $\mathbf{g}$.

$$\boldsymbol{\eta} = \mathbf{g}(\boldsymbol{\mu}) \iff \boldsymbol{\mu} = \mathbf{g}^{-1}(\boldsymbol{\eta}).$$

For the Gaussian, we started with the parameters (μ, σ^2) and we have seen that there is a 1-1 relationship to the vector $(\boldsymbol{\eta}_1, \boldsymbol{\eta}_2)$. But we could have started with the parameters $(\mu, \mu^2+\sigma^2)$ (which now corresponds to $\mathbb{E}[\boldsymbol{\phi}(y)] = \mathbb{E}[(y, y^2)^\top]$ instead). And again we would have found that there is a 1-1 relationship between $\mathbb{E}[\boldsymbol{\phi}(y)]$ and the vector $\boldsymbol{\eta}$.
For a list of such link functions for various distributions see the chapter on "Generalized Linear Model" in the KPM book.

Applications in ML

Let us now look at two applications of exponential families in ML.

Maximum Likelihood Parameter Estimation

Assume that we have a set of samples $\{y_n\}_{n=1}^N$. We assume that these are independent samples from some distribution. Further, we assume that they come from some exponential family with a given $h(y)$ and sufficient statistics $\boldsymbol{\phi}(y)$ but unknown parameter $\boldsymbol{\eta}$ (or we simply want to find that element of this family of distributions that is closest). Our aim is to estimate the parameter $\boldsymbol{\eta}$. We use our maximum likelihood

principle to find this parameter. Hence we minimize

$$\begin{aligned}\mathcal{L}(\boldsymbol{\eta}) &= -\ln(p(\mathbf{y}|\boldsymbol{\eta})) \\ &= \sum_{n=1}^{N}[-\ln(h(y_n)) - \boldsymbol{\eta}^\top \boldsymbol{\phi}(y_n) + A(\boldsymbol{\eta})].\end{aligned}$$

We see that this is a convex function in $\boldsymbol{\eta}$ since $A(\boldsymbol{\eta})$ is a convex function. Further, if we assume that we can determine the link function we can derive the solution in an explicit form by taking the gradient and setting it to zero:

$$\frac{1}{N}\nabla\mathcal{L}(\boldsymbol{\eta}) = -(\frac{1}{N}\sum_{n=1}^{N}\boldsymbol{\phi}(y_n)) + \mathbb{E}[\boldsymbol{\phi}(y)] = 0.$$

This equation makes sense intuitively. It says that we should pick $\boldsymbol{\eta}$ in such a way that the expected value of the sufficient statistics is equal to its empirical value! In formulae,

$$\boldsymbol{\eta} = \mathbf{g}(\frac{1}{N}\sum_{n=1}^{N}\boldsymbol{\phi}(y_n)).$$

We now see the justification for why we called $\boldsymbol{\phi}(y)$ a sufficient statistics.

Generalized Linear Models

Given an element from the exponential family with a scalar $\phi(y)$, we can construct from this a data model by assuming that a sample $(\mathbf{x}, y)$ follows the distribution

$$p(y \mid \mathbf{x}, \mathbf{w}) = h(y)e^{\mathbf{x}^\top \mathbf{w}\phi(y) - A(\mathbf{x}^\top \mathbf{w})}.$$

We call such a model a *generalized linear model*. Just note: If we had chosen $\mathbf{w}$ to be a matrix instead of a vector then we could build generalized linear models using exponential families with non-scalar parameters. Not much changes. To keep things simple we stick to the scalar case.
It is a generalization of the data model we used for logistic regression. As we will now discuss, for such a model the maximum likelihood problem is particularly easy to solve. Assume that we have given a training set S_{train} consisting of N independent samples $(\mathbf{x}_n, y_n)$. Assume further that we fit a generalized linear model to this data. This means that we assume that samples obey a distribution of the form

$$p(y_n \mid \mathbf{x}_n, \mathbf{w}) = h(y_n)e^{\eta_n \phi(y_n) - A(\eta_n)}$$

with $\eta_n = \mathbf{x}_n^\top \mathbf{w}$. Given S_{train}, we then write down the likelihood and look for that weight vector $\mathbf{w}$ that maximizes this likelihood.
In more detail, we consider the cost function

$$\begin{aligned}\mathcal{L}(\mathbf{w}) &= -\sum_{n=1}^{N} \ln p(y_n | \mathbf{x}_n^\top \mathbf{w}) \\ &= -\sum_{n=1}^{N} \ln(h(y_n)) + \mathbf{x}_n^\top \mathbf{w}\phi(y_n) - A(\mathbf{x}_n^\top \mathbf{w}).\end{aligned}$$

We want to minimize this cost function (we added a minus sign). First, note that this cost function is convex, hence a greedy algorithm should work well.

Let us take the gradient of this expression. We get

$$\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) = -\sum_{n=1}^{N} \mathbf{x}_n \phi(y_n) - \mathbf{x}_n g^{-1}(\mathbf{x}_n^\top \mathbf{w}).$$

where we used the fact that

$$\nabla A(\eta) = g^{-1}(\eta).$$

If we set this equation to zero we get the condition of optimality. In particular, if we rewrite this sum by using our matrix notation we get

$$\nabla\mathcal{L}(\mathbf{w}) = \mathbf{X}^\top \left[g^{-1}(\mathbf{X}\mathbf{w}) - \phi(\mathbf{y})\right] = 0,$$

where, as before, the scalar functions (g^{-1} and ϕ) are applied to each vector component-wise.
To compare, for the case of the logistic regression we got the equation

$$\nabla\mathcal{L}(\mathbf{w}) = \mathbf{X}^\top \left[\sigma(\mathbf{X}\mathbf{w}) - \mathbf{y}\right] = 0.$$

As we have discussed, for the logistic case (Bernoulli distribution) we have the relationship $g^{-1} = \sigma$, which confirms that our previous derivation was just a special case.
Note also that we have already shown that $A(\mathbf{x}^\top\mathbf{w})$ is a convex function (A is convex and $A(\mathbf{x}^\top\mathbf{w})$ is the composition of a linear function with a convex function). Therefore $\mathcal{L}(\mathbf{w})$ is convex (the other terms are constant or linear), just as we have seen this for the logistic regression. As a consequence, greedy iterative algorithms (like gradient descent) to find the optimum weight vector $\mathbf{w}$ are expected to work well in this context.

Machine Learning Course - CS-433

Nearest Neighbor Classifiers and the Curse of Dimensionality

Oct 20th, 2020

minor changes by Nicolas Flammarion 2020; changes by Rüdiger Urbanke 2019,2018,2017,2016;

Last updated on: November 1, 2020

EPFL

Classification example

In many cases a linear model is not optimal.

FIGURE 2.1. *A classification example in two dimensions. The classes are coded as a binary variable (*BLUE $= 0$, ORANGE $= 1$*), and then fit by linear regression. The line is the decision boundary defined by* $x^T\hat{\beta} = 0.5$*. The orange shaded region denotes that part of input space classified as* ORANGE*, while the blue region is classified as* BLUE*.*

*All figures taken from Chapter 2 of "The Elements of Statistical Learning" by Hastie, Friedman, and Tibshirani.

The k-nearest-neighbor classifier/regressor is an entirely different way of performing classification/regression. It performs best if we are working in low dimensions and if we have reason to believe that points $(\mathbf{x}, y)$ that are "close" in input have similar labels/values.

k-Nearest Neighbor (k-NN)

Assume that S_{train} is our training data. We are given a "fresh" input $\mathbf{x}$ and want to make a prediction (regression). A natural k-NN prediction for $\mathbf{x}$ is,

$$f_{S_{\text{train}},k}(\mathbf{x}) = \frac{1}{k} \sum_{n:\mathbf{x}_n \in \text{nbh}_{S_{\text{train}},k}(\mathbf{x})} y_n \ ,$$

where $\text{nbh}_{S_{\text{train}},k}(\mathbf{x})$ is the set of k input points in S_{train} that are closest to $\mathbf{x}$. There are many possible variants. E.g., instead of taking the empirical average we could weigh individual samples the heavier the closer they are to $\mathbf{x}$.
For the classification problem it is natural to output that label that appears the most frequently,

$$f_{S_{\text{train}},k}(\mathbf{x}) = \text{majority element}\{y_n : \mathbf{x}_n \in \text{nbh}_{S_{\text{train}},k}(\mathbf{x})\}.$$

For the binary case it is good to pick k to be odd so that there is a clear winner. For the simplest case of $k = 1$ we return the label of the closest neighbor.
Figures "2.3" and "2.2" show this rule applied to a binary classification problem with $k = 1$ an $k = 15$.

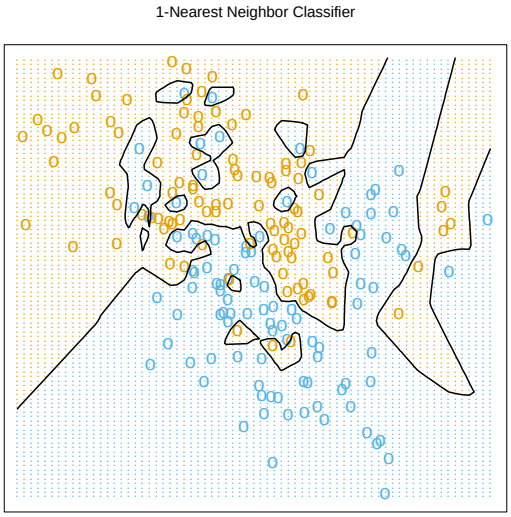


FIGURE 2.3. *The same classification example in two dimensions as in Figure 2.1. The classes are coded as a binary variable* (BLUE $= 0$, ORANGE $= 1$)*, and then predicted by* 1*-nearest-neighbor classification.*

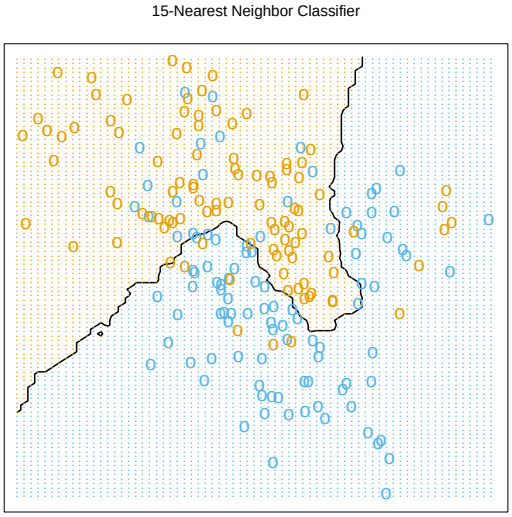


FIGURE 2.2. *The same classification example in two dimensions as in Figure 2.1. The classes are coded as a binary variable* (BLUE $= 0$, ORANGE $= 1$) *and then fit by* 15*-nearest-neighbor averaging as in (2.8). The predicted class is hence chosen by majority vote amongst the* 15*-nearest neighbors.*

Bias-variance revisited

Note that if we pick a large value of k then we are smoothing/averaging over a large area. In the extreme case where

k is equal to the size of the training data our prediction will become a constant. Therefore – large k equals simple model, small k equals complex model.
Hence, if we pick k small we expect a small bias but large variance and if we pick k large we expect a large bias but small variance. Figure "2.4" shows the test error as a function of the complexity of the model. It shows the usual "U" shaped curve. The left part of the figure corresponds to large k (small complexity) and the right part corresponds to small k (large complexity. On the left the test error is large due to a model that is too simple (too much averaging), whereas on the right it is large due to a large variance.

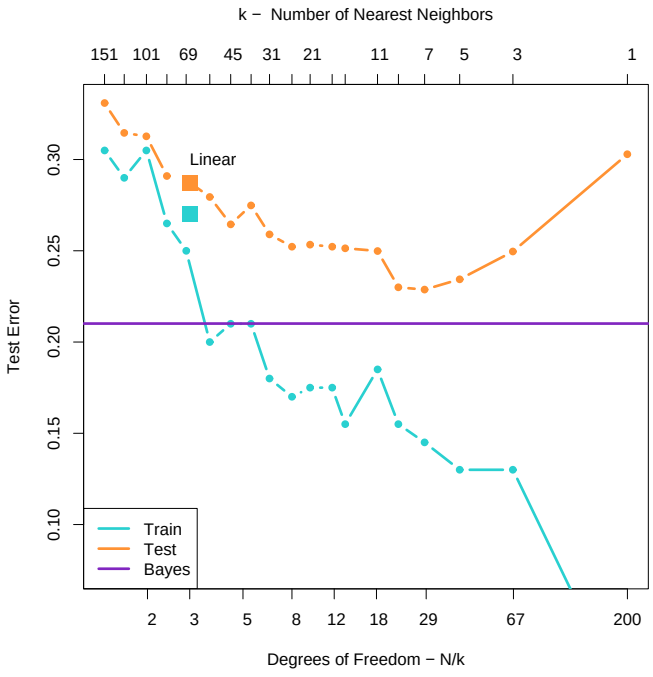


FIGURE 2.4. *Misclassification curves for the simulation example used in Figures 2.1, 2.2 and 2.3. A single training sample of size* 200 *was used, and a test sample of size* $10,000$. *The orange curves are test and the blue are training error for k-nearest-neighbor classification. The results for linear regression are the bigger orange and blue squares at three degrees of freedom. The purple line is the optimal Bayes error rate.*

Curse of dimensionality

According to Pedro Domingos: "Intuition fails in high dimensions". This is also known as the curse of dimensionality (Bellman, 1961).

Claim 1: "Generalizing correctly becomes exponentially harder as the dimensionality grows because fixed-size training sets cover a dwindling fraction of the input space."

To see this, assume that our data is uniformly distributed in the box $[0, 1]^d$. Take the point at the center of this box, i.e., the point $(\frac{1}{2}, \cdots, \frac{1}{2})$, and draw a (small) box around it of side length r. What fraction of the total volume does this smaller box cover? It is r^d. Therefore, in expectation a fraction r^d of the data lies in this small box. Let the desired fraction be α. Then for $\alpha = 0.01$ in 10 dimensions we need r to be 0.63 and if we want to capture a fraction 0.1 of the data then we need $r = 0.8$ (recall the large box has a side length of only 1!). So we need to explore almost the whole range in each dimension!

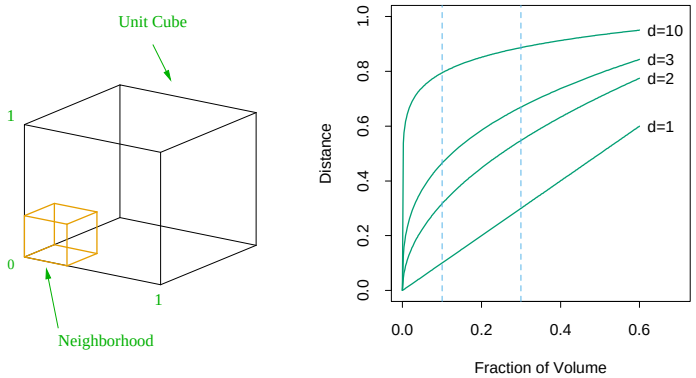


FIGURE 2.6. *The curse of dimensionality is well illustrated by a subcubical neighborhood for uniform data in a unit cube. The figure on the right shows the side-length of the subcube needed to capture a fraction r of the volume of the data, for different dimensions p. In ten dimensions we need to cover* 80% *of the range of each coordinate to capture* 10% *of the data.*

We will see shortly that as a result we have to scale the number of samples exponentially in the dimension if we want our risk to stay constant.

Claim 2: In high-dimension, data-points are far from each other. Consequently, "as the dimensionality increases, the choice of nearest neighbor becomes effectively random."

Consider again N data points uniformly distributed in the box $[0,1]^d$. We consider a nearest-neighbor estimate at the point $(\frac{1}{2}, \cdots, \frac{1}{2})$. Assume that we center around this point a box of side-length r. How large do we have to pick r so that the chance that at least one point falls into this smaller box is $\frac{1}{2}$?

We claim that

$$r = \left(1 - \frac{1}{2}^{1/N}\right)^{1/d}$$

For $N = 500, d = 10$, this number is 0.52. So the box has to have a side length of half the one of the large box before we can hope to find a neighbor inside it.

To see this claim, what is the chance that a random sample inside the unit box is not inside this small box? This probability is $1 - r^d$, since the small box covers a fraction r^d of the volume of the big box. Therefore, the chance that none of N iid samples fall inside the small box is equal to $(1 - r^d)^N$. We want this probability to be $\frac{1}{2}$ (we are looking for the median). Solving for r confirms the above claim.

Analysis of nearest neighbor rule

Our aim is to analyze the simplest setting. We consider the nearest neighbor classifier and we will compare its performance to the optimal (Bayes) classifier. This will confirm our intuition about the curse of dimensionality and tell us how good this classifier performs compared to the best we can hope for. In particular, if we have plenty of data we will see a simple and pleasing connection.

Here is our set-up. We assume that $\mathbf{x}$ takes values in $\mathcal{X} = [0,1]^d$ and that $\mathcal{Y} = \{0,1\}$ (binary classification). We assume further that there is some unknown distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$ from which samples are drawn. Our training set S_{train} will consist of N iid samples. Associated to the distribution $\mathcal{D}$ there is the conditional probability

$$\mathbb{P}\{y = 1 \mid \mathbf{x}\}.$$

To ease our notation, let us introduce

$$\eta(\mathbf{x}) = \mathbb{P}\{y = 1 \mid \mathbf{x}\}.$$

In words, $\eta(\mathbf{x})$ is the conditional probability that the label is 1 given that the input is $\mathbf{x}$.

If we knew the distribution $\mathcal{D}$ (and hence $\eta(\mathbf{x})$) we would employ the classifier $f_\star$ which outputs

$$f_\star(\mathbf{x}) = \mathbf{1}_{\{\eta(\mathbf{x}\} > \frac{1}{2}\}}.$$

This is the *Bayes* classifier (also called maximum a posteriori or MAP classifier). It has the smallest probability of misclassification of any classifier, namely

$$\mathcal{L}(f_\star) = \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}[\min\{\eta(\mathbf{x}), 1 - \eta(\mathbf{x})\}].$$

It is instructive to compare the classification error of our nearest neighbor classifier to this optimal classifier.
It is clear that we cannot hope to perform well if there is no correlation between the "position" $\mathbf{x}$ and the associated label. E.g., consider the case where the label y is chosen randomly and independently from $\mathbf{x}$. It is then of no help to know the labels of points close by. So we need to make a suitable assumption regarding $\mathcal{D}$. Here is one way of doing this. We demand that the distribution $\mathcal{D}$ is such that for some constant c we have

$$|\eta(\mathbf{x}) - \eta(\mathbf{x}')| \leq c||\mathbf{x} - \mathbf{x}'||,$$

where on the right we have the Euclidean distance. In words, we ask that the conditional probability $\mathbb{P}\{y = 1 \mid \mathbf{x}\}$ (seen as a function of $\mathbf{x}$) is Lipschitz continuous with Lipschitz constant c. This seems a reasonable assumption: if we move a point only slightly, we want that the probability of this point to have an associated label $y = 1$ only varies slightly as well.
Let $\mathcal{L}(f_\star)$ denote the risk (classification error) for this setting assuming that we use the Bayes classifier and let $\mathcal{L}(f_{S_{\text{train}}})$ denote the risk (classification error) for this setting if we use the nearest neighbor classifier.

Lemma.

$$\begin{aligned}\mathbb{E}_{S_{train}}[\mathcal{L}(f_{S_{train}})] &\leq 2\mathcal{L}(f_\star) + c\,\mathbb{E}_{S_{train};\mathbf{x}\sim\mathcal{D}}[\|\mathbf{x} - nbh_{S_{train},1}(\mathbf{x})\|] \\ &\leq 2\mathcal{L}(f_\star)] + 4c\sqrt{d}N^{-\frac{1}{d+1}}.\end{aligned}$$

Before we see where this bound comes from, let us interpret this result. We see that the risk of the nearest neighbor

classifier is upper bounded by twice the risk of the optimal classifier plus a "geometric" term. This term is the average distance of a randomly chosen point to the nearest point in the given training set times the Lipschitz constant c.

It is very natural to have this second term present. Our assumption was that nearby points are likely to have the same label. So it is reasonable to expect in the bound a term that depends on the average distance.

So let us discuss what happens in certain limiting cases. Assume at first that we have a fixed dimension and that the size of the training data tends to infinity. Then the second term on the right will converge to zero and we see that in this case we have a risk which is at most twice the Bayes risk. So if we have plenty of data then we are doing well!

On the other hand, if we fix the size of the data, namely N, but increase the dimension we see from the bound that very quickly we should expect a large error. The second term on the right is proportional to $1/N$ to the power $1/(d+1)$ (there is an extra term $4c\sqrt{d}$ in the bound but this is relatively minor and hence let us ignore it). More precisley, in order to keep the second term on the right constant, lets say α, $\alpha << 1$, we need to let N grow like $(1/\alpha)^{d+1}$, i.e., exponential in the dimension. This is another way of seeing the curse of dimensionality.

Note that the nearest neighbor rule is an interpolating classification method (since the training labels are predicted at the training points, it obtains zero classification error on the training data). Thus the previous result is going against the common belief that interpolating classifiers are not gen-

eralizing and will alway lead to overfitting. However this method is still not consistent (when the number of training data is going to infinity, the difference between its classification error and the Bayes risk, i.e., the minimal risk possible, is not going to zero) and saturates at twice the Bayes risk. Interested readers can read the following paper of Balkin (https://arxiv.org/pdf/1806.05161.pdf) where related classification rules are shown to be nearly consistent.

Proof. Let us now explain how to derive such a bound. We follow the proof in Chapter 19 of the "Understanding Machine Learning" book. You can find a more detailed explanation there.

We start with the second inequality. How do we bound $\mathbb{E}_{S_{\text{train}};\mathbf{x}\sim\mathcal{D}}[\|\mathbf{x}-\text{nbh}_{S_{\text{train}},1}(\mathbf{x})\|]$ by $4\sqrt{d}N^{-\frac{1}{d+1}}$? Take the unit cube $[0,1]^d$. Recall that this is the space of inputs. Cut this "large" cube into small little cubes of side length ϵ. Consider now what happens when we want to predict a label for a "fresh" input.

Consider the cube which contains $\mathbf{x}$. If we are lucky, the same cube also contains elements from the training set S_{train}. In this case $\mathbf{x}$ has a neighbor in S_{train} at distance at most $\sqrt{d}\epsilon$. This is good since by our assumption a short distance leads to good predictions. But if $\mathbf{x}$ lies in a cube which contains no such element from the training data, then its nearest neighbor might be much further, at worst at a distance $\sqrt{d}$. In this case the prediction is probably not very reliable.

So what is the chance that a randomly chosen point $\mathbf{x}$ ends up in a specific box and this box does not have a training point in there?

If the probability of $\mathbf{x}$ landing in a particular box i is lets say $\mathbb{P}_i$, then the chance that none of the N training symbols are in box i is $(1-\mathbb{P}_i)^N$. This is the main step in this proof. We do not know the distribution $\mathcal{D}$ and hence cannot determine the probabilities $\mathbb{P}_i$. But it turns out that this is not important. No matter how the probability is distributed over the boxes we are fine. The reason is simple. If $\mathbb{P}_i$ is large (i.e., this case happens often) we are fine since then it is very likely that we also have at least one training point in this box. But if $\mathbb{P}_i$ is small (and hence also the probability that we have a training point in there is small) we are fine since by definition this does not happen very often.
The rest is calculus, carefully choosing the right scaling for ϵ in order to get a good bound.
It still remains to explain the term $2\mathcal{L}(f_\star)$. Consider the following experiment. We are given two points $\mathbf{x}$ and $\mathbf{x}'$, both elements of $[0,1]^d$. Assign to these two points labels y and y' according to the distribution $\eta(\cdot)$. What is the chance that these two labels are not the same? We claim that

$$\mathbb{P}\{y \neq y'\} \leq 2\eta(\mathbf{x})(1-\eta(\mathbf{x})) + c\|\mathbf{x}-\mathbf{x}'\|. \qquad (1)$$

This is easily explained. Assume at first that we draw the two labels y and y' independently but according to the *same* conditional probability $\eta(\mathbf{x})$. In this case the probability that the two labels are different is exactly equal to $2\eta(\mathbf{x})(1-\eta(\mathbf{x}))$. But according to our model we draw the label for $\mathbf{x}$ according to the conditional probability $\eta(\mathbf{x})$ and the label for $\mathbf{x}'$ independently but according to the conditional probability $\eta(\mathbf{x}')$. These two quantities are in general not the same. But

we have

$$\begin{aligned}
&\eta(\mathbf{x})(1-\eta(\mathbf{x}'))+\eta(\mathbf{x}')(1-\eta(\mathbf{x})) \\
=&2\eta(\mathbf{x})(1-\eta(\mathbf{x}))+(2\eta(\mathbf{x})-1)(\eta(\mathbf{x})-\eta(\mathbf{x}')) \\
\leq&2\eta(\mathbf{x})(1-\eta(\mathbf{x}))+|\eta(\mathbf{x})-\eta(\mathbf{x}')| \\
\leq&2\eta(\mathbf{x})(1-\eta(\mathbf{x}))+c\|\mathbf{x}-\mathbf{x}'\|.
\end{aligned}$$

Consider now the following experiment. Draw a set S_{train} according to the distribution $\mathcal{D}$ but hide the labels y_n and only reveal the inputs $\mathbf{x}_n$. Then draw one more "fresh" sample $(\mathbf{x}, y)$ but hide also in this case the label y and only reveal $\mathbf{x}$. Find the point in S_{train} that is closest to $\mathbf{x}$, call it $\text{nbh}_{S_{\text{train}},k}(\mathbf{x})$. Now reveal the label y associated to $\mathbf{x}$ as well as the label $y_{i\text{nbh}_{S_{\text{train}},k}(\mathbf{x})}$ associated to this closest point $\text{nbh}_{S_{\text{train}},k}(\mathbf{x})$. What is the probability that these two labels do not agree?

This is exactly the risk $\mathbb{E}_{S_{\text{train}}}[\mathcal{L}(f_{S_{\text{train}}})]$. And according to (1) we can bound this probability by averaging the right hand side over all choices of $\mathbf{x}$ and $\text{nbh}_{S_{\text{train}},k}(\mathbf{x})$. The average of the term $2\eta(\mathbf{x})(1-\eta(\mathbf{x}))$ is upper bounded by $2\mathcal{L}(f_\star)$ since $2\eta(\mathbf{x})(1-\eta(\mathbf{x})) \leq 2\min\{\eta(\mathbf{x}), 1-\eta(\mathbf{x})\}$. And the average of the term $c\|\mathbf{x}-\mathbf{x}'\|$ is $c\,\mathbb{E}_{S_{\text{train}};\mathbf{x}\sim\mathcal{D}}[\|\mathbf{x}-\text{nbh}_{S_{\text{train}},1}(\mathbf{x})\|]$.

□

Machine Learning Course - CS-433

Support Vector Machines

Oct 27, 2020

changes by Nicolas Flammarion, changes by Martin Jaggi 2019, changes by Rüdiger Urbanke 2018, changes by Martin Jaggi 2016, 2017

Last updated on: November 2, 2020

EPFL

Motivation

Let us re-consider binary classification with data pairs $(\mathbf{x}_n, \tilde{y}_n)$, $\tilde{y}_n \in \{0, 1\}$. We use here the notation $\tilde{y}_n$ to denote variables that take values in $\{0, 1\}$ since soon we will transform our labels to take values in $\{\pm 1\}$ and we want to avoid confusion. As we had discussed, if we use least squares (not recommended!) and ignore right now any potential regularization term this lead us to the minimization

$$\min_{\mathbf{w}} \sum_{n=1}^{N} (\tilde{y}_n - \mathbf{x}_n^\top \mathbf{w})^2.$$

If instead we use logistic regression and optimize the log-likelihood, we solve

$$\min_{\mathbf{w}} \sum_{n=1}^{N} \log(1 + e^{\mathbf{x}_n^\top \mathbf{w}}) - \tilde{y}_n \mathbf{x}_n^\top \mathbf{w}.$$

In the following it will be slightly more convenient to assume that $y_n \in \{\pm 1\}$, where we have the mappings $0 \leftrightarrow -1$ and $1 \leftrightarrow 1$. We can write this compactly as $y_n = 2\tilde{y}_n - 1$, or equivalently, $\tilde{y}_n = \frac{1}{2}(y_n + 1)$.

Consider first the least squares problem. Assume, as always, that the feature vector contains the constant feature 1 and that this is component 0. Define $\tilde{\mathbf{w}} = \frac{1}{2}(\mathbf{w} + e_0)$, where e_0 is the vector of length D (the dimension of the feature vector) that has a 1 at component 0 and 0 at all other components. Note that this defines a one-to-one mapping between $\mathbf{w}$ and

$\tilde{\mathbf{w}}$. We then have

$$\begin{aligned}
4\sum_{n=1}^{N}(\tilde{y}_n - \mathbf{x}_n^\top \tilde{\mathbf{w}})^2 &= 4\sum_{n=1}^{N}(\frac{1}{2}(y_n+1) - \frac{1}{2}\mathbf{x}_n^\top(\mathbf{w}+e_0))^2 \\
&= \sum_{n=1}^{N}((y_n+1) - \mathbf{x}_n^\top(\mathbf{w}+e_0))^2 \\
&= \sum_{n=1}^{N}(y_n - \mathbf{x}_n^\top \mathbf{w})^2 \\
&\overset{(a)}{=} \sum_{n=1}^{N}(1 - y_n\mathbf{x}_n^\top \mathbf{w})^2 \\
&= \sum_{n=1}^{N}\mathrm{MSE}(\mathbf{x}_n^\top \mathbf{w}, y_n),
\end{aligned}$$

where

$$\mathrm{MSE}(z,y) = (1-yz)^2.$$

In step (a) we have used the fact that $y_n\{\pm 1\}$ so that $y_n^2 = 1$. In a similar manner, for logistic regression we have

$$\begin{aligned}
\sum_{n=1}^{N}\log(1+e^{\mathbf{x}_n^\top \mathbf{w}}) - \tilde{y}_n\mathbf{x}_n^\top \mathbf{w} &= \sum_{n=1}^{N}\log(1+e^{-y_n\mathbf{x}_n^\top \mathbf{w}}) \\
&= \sum_{n=1}^{N}\mathrm{LogisticLoss}(\mathbf{x}_n^\top \mathbf{w}, y_n),
\end{aligned}$$

where

$$\mathrm{LogisticLoss}(z,y) = \log(1+\exp(-yz)).$$

This is most easily seen by checking the two cases $\tilde{y}_n = 0 \leftrightarrow y_n = -1$ and $\tilde{y}_n = 1 \leftrightarrow y_n = 1$ separately.
Note that above, z is the prediction based on the given data point and y is the label associated to the data point. We get support vector machines (SVMs), if instead we use the loss

$$\text{Hinge}(z, y) = [1 - yz]_+ = \max\{0, 1 - yz\},$$

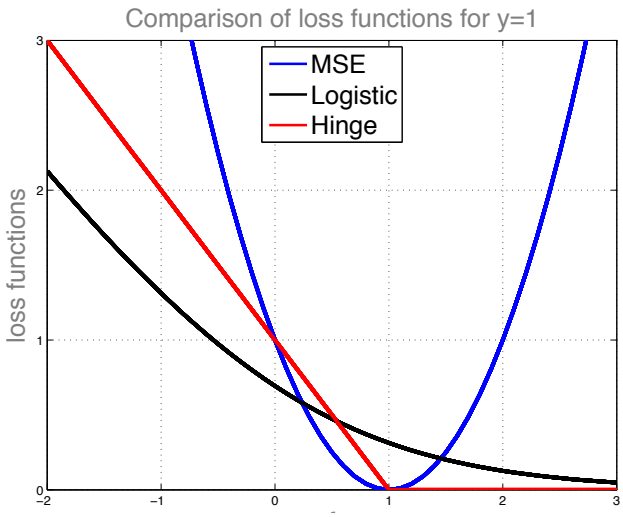
and add a regularization term.

Support Vector Machine

As just mentioned, SVMs correspond to the following optimization problem:

$$\min_{\mathbf{w}} \sum_{n=1}^{N} \left[1 - y_n \mathbf{x}_n^\top \mathbf{w}\right]_+ + \tfrac{\lambda}{2}\|\mathbf{w}\|^2.$$

The next figure compares the cost functions of the mean-squared loss, the logistic loss, and the hinge loss. Note that

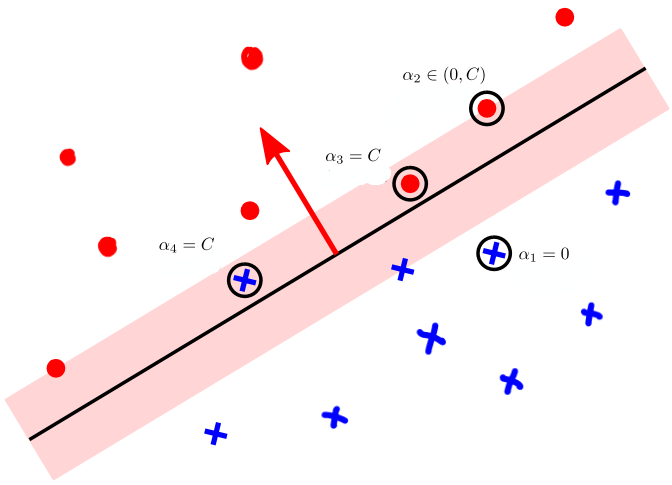


for least squares we incur a cost whenever we fail to represent the desired value exactly and the cost is symmetric around

the target value. For logistic regression we always incur a cost but the cost is asymmetric – it becomes smaller the further we are "on the right side" and it becomes larger the further we are "on the wrong side." The hinge loss acts differently. Once we are "sufficiently far" on the right side we no longer pay a cost. But if we are not yet far enough on the correct side or if we are on the wrong side we do pay a cost and the cost increases linearly the further we are away.

In the figure below you see a region in pink. This region is called the "margin." It is defined as follows: Take the normal vector $\mathbf{w}$ that defines the hyperplane. Look at all feature vectors $\mathbf{x}$ so that $|\mathbf{x}^\top \mathbf{w}| \leq 1$. This is the margin. Note that the margin does not only depend on the direction of the vector $\mathbf{w}$ but also its norm. In fact the total width of the margin is equal to $2/\|\mathbf{w}\|$.

If you are looking for an interpretation: It is the region where our prediction is not yet sufficiently "confident" (assuming that we get the sign right). The intuition is strongest if we

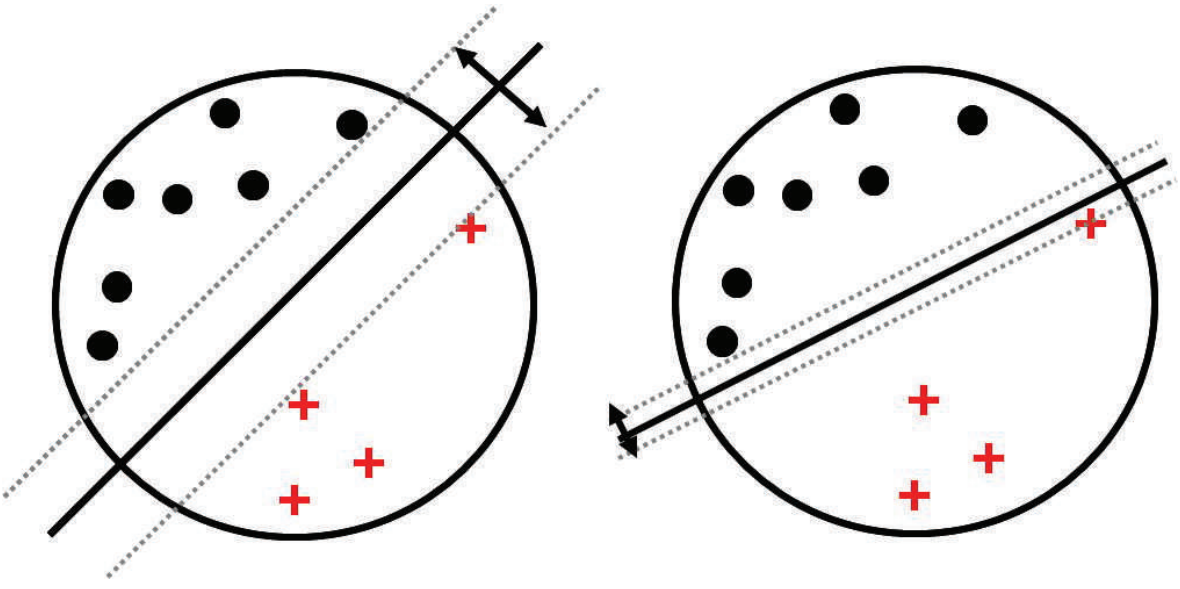


look at the case of separable data. This is shown in the following two figures. Assume that λ is small so that the cost function is dominated by the sum over the hinge losses

on the left. What will the optimal $\mathbf{w}$ look like in this case? It is clear that we want the following:

1. A separating hyperplane.
2. A scaling of $\mathbf{w}$ so that no point of the data is in the margin.
3. That separating hyperplane and scaling for which the margin is the largest.

Conditions (1) and (2) ensure that there is no cost incurred in the first expression (the sum over the terms $\left[1 - y_n \mathbf{x}_n^\top \mathbf{w}\right]_+$). Since by assumption that λ is small this is the dominant term, we cannot hope to do better than having this term to be 0. The condition (3) ensures that we have the minimum possible cost associated for the regularization term, i.e. the minimal possible normed squared $\|\mathbf{w}\|^2$. Geometrically this corresponds to a hyperplane with maximal "spacing" to the left and right, i.e., a hyperplane with maximal margin.



NOTE: We have introduced our formulation of SVMs for the general case where the data is not necessarily linearly separable. This is sometimes called the *soft-margin* formulation. The *hard-margin* formulation concerns the case where the data is linearly separable and where we insist that the decision region is given in terms of a separating hyperplane. In this case the formulation would require us to find a separating hyperplane with minimal $\|\mathbf{w}\|^2$. In other words, the optimal solution is a separating hyperplane with maximal margins. And in this case there must be some feature vectors which lie exactly on the boundaries (otherwise we could enlarge the margin, contradicting optimality). These feature vectors "support" the boundaries and hence the name "support vector." For the soft-margin case this interpretation is only approximately true as we have explained in the previous paragraphs.

Optimization

Now where we have established *what function* we are optimizing, let us look at the question *how* we can optimize it efficiently.
Note that the function is convex and and has a subgradient (in $\mathbf{w}$):

$$\min_{\mathbf{w}} \sum_{n=1}^{N} \left[1 - y_n \mathbf{x}_n^\top \mathbf{w}\right]_+ + \tfrac{\lambda}{2}\|\mathbf{w}\|^2$$

We can therefore use SGD with subgradients. This is good news!

Duality: The big picture

We have just seen that we can use SGD in order to find the optimal parameters for the SVM. We will now discuss an alternative but equivalent formulation via the concept of *duality*. In some cases this leads to a more efficient implementation. But perhaps more importantly, once we have derived this alternative representation it will point us naturally to a more general formulation. This is called the *kernel trick*. We will explicitly discuss this technique in a separate lecture.

Let us say that we are interested in minimizing a function $\mathcal{L}(\mathbf{w})$. Assume that we can define an auxiliary function $G(\mathbf{w}, \boldsymbol{\alpha})$ so that

$$\mathcal{L}(\mathbf{w}) = \max_{\boldsymbol{\alpha}} \; G(\mathbf{w}, \boldsymbol{\alpha}).$$

We can therefore solve our original problem by solving

$$\min_{\boldsymbol{w}} \max_{\boldsymbol{\alpha}} G(\mathbf{w}, \boldsymbol{\alpha}).$$

We call this the *primal* problem. In some cases it might be much easier to find

$$\max_{\boldsymbol{\alpha}} \min_{\boldsymbol{w}} G(\mathbf{w}, \boldsymbol{\alpha}).$$

We call this the *dual* problem. This leads us naturally to the following questions:

1. How do we find a suitable $G(\mathbf{w}, \boldsymbol{\alpha})$?
2. When is it OK to switch $\min_{\boldsymbol{w}}$ and $\max_{\boldsymbol{\alpha}}$?

3. When is the dual easier to optimize than the primal?

Q1: How do we find a suitable $G(\mathbf{w}, \boldsymbol{\alpha})$? There is a general theory on this topic (see e.g., Bertsekas' "Nonlinear Programming" for more formal details). But rather than talk about this in the abstract, let us look at our specific problem. We have

$$[z]_+ = \max\{0, z\} = \max_{\alpha \in [0,1]} \alpha z.$$

Therefore,

$$[1 - y_n \mathbf{x}_n^\top \mathbf{w}]_+ = \max_{\alpha_n \in [0,1]} \alpha_n (1 - y_n \mathbf{x}_n^\top \mathbf{w}).$$

So we can rewrite the SVM problem as:

$$\min_{\boldsymbol{w}} \max_{\boldsymbol{\alpha} \in [0,1]^N} \underbrace{\sum_{n=1}^{N} \alpha_n (1 - y_n \mathbf{x}_n^\top \mathbf{w}) + \tfrac{\lambda}{2} \|\mathbf{w}\|^2}_{G(\boldsymbol{w}, \boldsymbol{\alpha})}$$

Note that $G(\boldsymbol{w}, \boldsymbol{\alpha})$ is convex in $\mathbf{w}$ and linear, hence concave, in $\boldsymbol{\alpha}$.

Q2: When is it OK to switch max and min?

Note that it is always true that

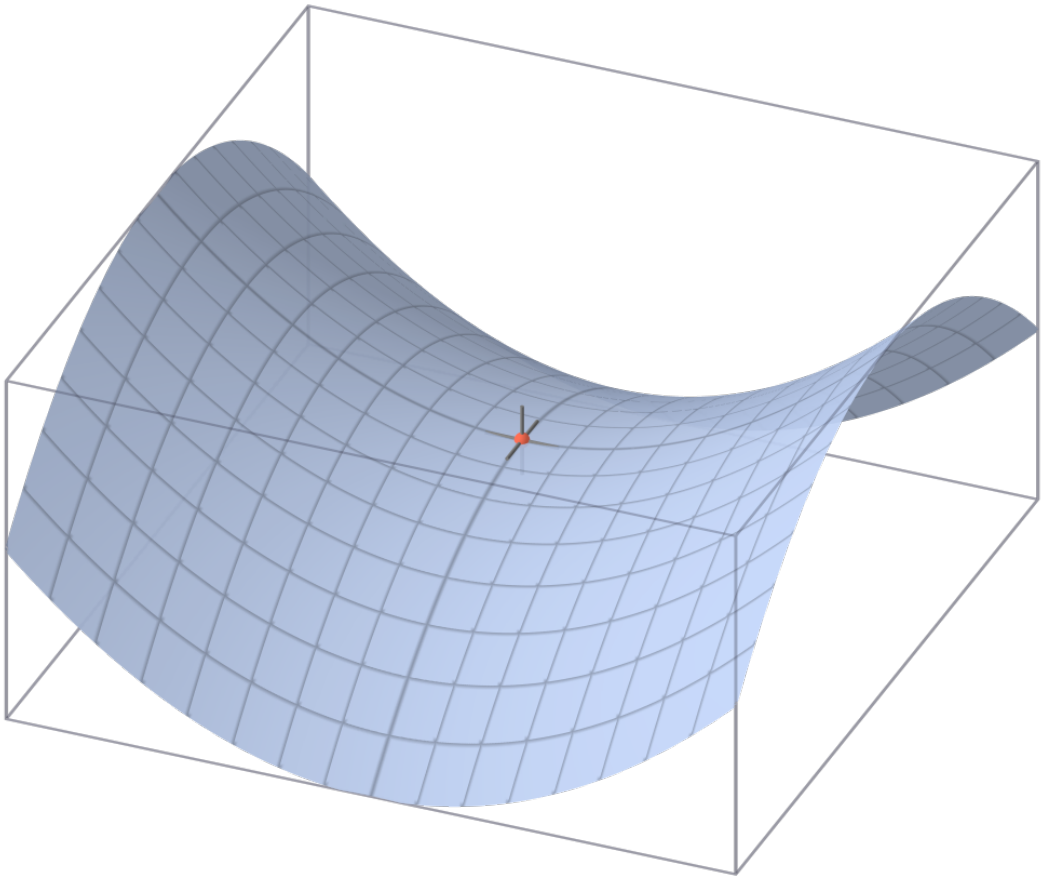
$$\max_{\boldsymbol{\alpha}} \min_{\boldsymbol{w}} G(\mathbf{w}, \boldsymbol{\alpha}) \leq \min_{\boldsymbol{w}} \max_{\boldsymbol{\alpha}} G(\mathbf{w}, \boldsymbol{\alpha}).$$

This is easy to see:

$$\begin{aligned}
\min_{\boldsymbol{w}'} G(\mathbf{w}', \boldsymbol{\alpha}) &\leq G(\mathbf{w}, \boldsymbol{\alpha}) \;\; \forall \mathbf{w}, \boldsymbol{\alpha}, \Leftrightarrow \\
\max_{\boldsymbol{\alpha}} \min_{\boldsymbol{w}'} G(\mathbf{w}', \boldsymbol{\alpha}) &\leq \max_{\boldsymbol{\alpha}} G(\mathbf{w}, \boldsymbol{\alpha}) \;\; \forall \mathbf{w} \Leftrightarrow \\
\max_{\boldsymbol{\alpha}} \min_{\boldsymbol{w}'} G(\mathbf{w}', \boldsymbol{\alpha}) &\leq \min_{\boldsymbol{w}} \max_{\boldsymbol{\alpha}} G(\mathbf{w}, \boldsymbol{\alpha}).
\end{aligned}$$

We get equality if $G(\mathbf{w}, \boldsymbol{\alpha})$ is a continuous function that is convex in $\mathbf{w}$, concave in $\boldsymbol{\alpha}$, and the domain of $\mathbf{w}$ and $\boldsymbol{\alpha}$ are both compact and convex. I.e., in this case we have

$$\min_{\boldsymbol{w}} \max_{\boldsymbol{\alpha}} G(\mathbf{w}, \boldsymbol{\alpha}) = \max_{\boldsymbol{\alpha}} \min_{\boldsymbol{w}} G(\mathbf{w}, \boldsymbol{\alpha}).$$



In other words, we get equality if we have functions that look like saddles as in the previous figure.
For SVMs the condition is fulfilled and we can switch the min and max. This leads to the following formulation

$$\max_{\boldsymbol{\alpha} \in [0,1]^N} \min_{\boldsymbol{w}} \sum_{n=1}^{N} \alpha_n (1 - y_n \mathbf{x}_n^\top \mathbf{w}) + \tfrac{\lambda}{2} \|\mathbf{w}\|^2. \tag{1}$$

Taking the derivative w.r.t. $\mathbf{w}$ we get

$$\nabla_{\boldsymbol{w}} G(\mathbf{w}, \boldsymbol{\alpha}) = -\sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n + \lambda \mathbf{w}.$$

Equating this to $\mathbf{0}$, we can explicitly solve for $\mathbf{w}$ for any given $\boldsymbol{\alpha}$. We get

$$\mathbf{w}(\boldsymbol{\alpha}) = \tfrac{1}{\lambda}\sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n = \tfrac{1}{\lambda}\mathbf{X}^\top \mathbf{Y}\boldsymbol{\alpha},$$

where $\mathbf{Y} := \mathrm{diag}(\mathbf{y})$.
Plugging this $\mathbf{w} = \mathbf{w}(\boldsymbol{\alpha})$ back into the saddle-point formulation (1), gives rise to the following dual optimization problem:

$$\begin{aligned} &\max_{\boldsymbol{\alpha}\in[0,1]^N} \sum_{n=1}^{N} \alpha_n(1 - \tfrac{1}{\lambda} y_n \mathbf{x}_n^\top \mathbf{X}^\top \mathbf{Y}\boldsymbol{\alpha}) + \tfrac{\lambda}{2}\|\tfrac{1}{\lambda}\mathbf{X}^\top \mathbf{Y}\boldsymbol{\alpha}\|^2 \\ = &\max_{\boldsymbol{\alpha}\in[0,1]^N} \boldsymbol{\alpha}^\top \mathbf{1} - \tfrac{1}{2\lambda}\boldsymbol{\alpha}^\top \mathbf{Y}\mathbf{X}\mathbf{X}^\top \mathbf{Y}\boldsymbol{\alpha}. \end{aligned}$$

Q3: When is the dual easier to optimize than the primal, and why?

(1) The dual is a differentiable (but constrained) quadratic problem.

$$\max_{\boldsymbol{\alpha}\in[0,1]^N} \boldsymbol{\alpha}^\top \mathbf{1} - \tfrac{1}{2\lambda}\boldsymbol{\alpha}^\top \mathbf{Q}\boldsymbol{\alpha},$$

where $\mathbf{Q} := \mathrm{diag}(\mathbf{y})\mathbf{X}\mathbf{X}^\top \mathrm{diag}(\mathbf{y})$. Optimization is easy by using coordinate descent, or more precisely coordinate ascent since this is a maximization problem. Crucially, this method will be changing only one α_n variable a time.

(2) Note that in the dual formulation the data only enters in the form $\mathbf{K} := \mathbf{X}\mathbf{X}^\top$. This product $\mathbf{X}\mathbf{X}^\top$ is called

the "kernel." We hence often say that this formulation is *kernelized*. As we will discuss in the next lecture, this has a very pleasing consequence.

(3) The solution $\boldsymbol{\alpha}$ is typically sparse, and is non-zero only for the training examples that are instrumental in determining the decision boundary.

Recall the how the parameters α_n were introduced:

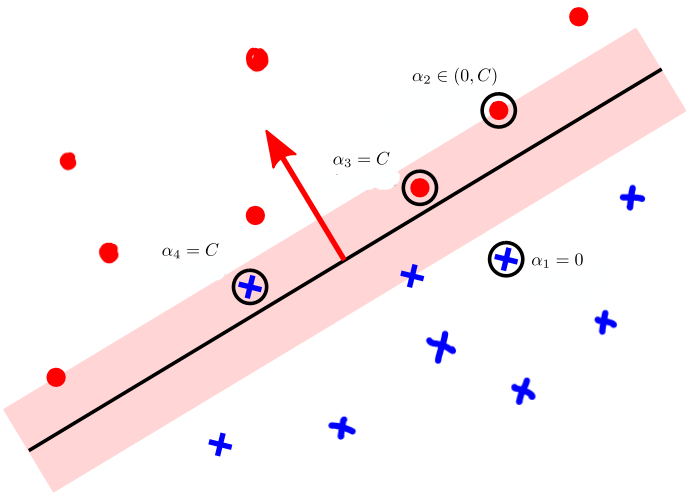
$$[1 - y_n \mathbf{x}_n^\top \mathbf{w}]_+ = \max_{\alpha_n \in [0,1]} \alpha_n (1 - y_n \mathbf{x}_n^\top \mathbf{w})$$

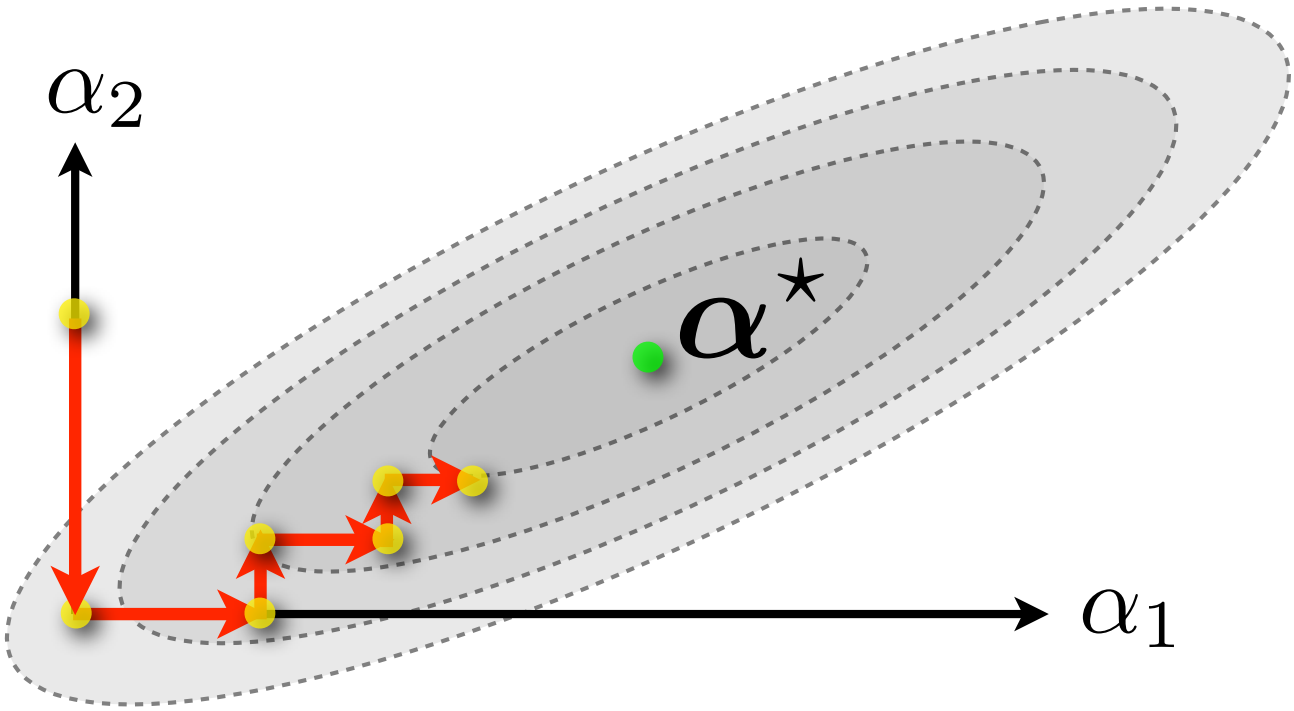
From this formulation we can see that there are three distinct cases we should consider:

a) Examples that lie on the correct side and outside the margin. For those $1 - y_n \mathbf{x}_n^\top \mathbf{w} < 0$. Hence, $\alpha_n = 0$.

b) Examples that lie on the correct side and just "on the margin". I.e., for those we have $1 - y_n \mathbf{x}_n^\top \mathbf{w} = 0$. Therefore $\alpha_n \in [0, 1]$.

c) Examples that lie strictly inside the margin, or on the wrong side. I.e., for those we have $1 - y_n \mathbf{x}_n^\top \mathbf{w} > 0$. Therefore $\alpha_n = 1$.





We call the $\mathbf{x}_n$ for which $\alpha_n = 0$ *non-support* vectors, the $\mathbf{x}_n$ for which $\alpha_n \in (0, 1)$ *essential support vectors* and *bound support vectors* when $\alpha_n = 1$. The above consideration explains why we expect most α_n to be zero.

Coordinate Descent

Goal: Find $\boldsymbol{\alpha}^\star \in \mathbb{R}^N$ maximizing or minimizing $g(\boldsymbol{\alpha})$.
Yet another optimization algorithm?

Idea: Update one coordinate at a time, while keeping others fixed.

initialize $\boldsymbol{\alpha}^{(0)} \in \mathbb{R}^N$
for t = 0:maxIter **do**

sample a coordinate n randomly from $1 \dots N$.
optimize g w.r.t. that coordinate:

$$u^\star \leftarrow \arg\min{}^{1}{}_{u \in \mathbb{R}} \, g\big(\alpha_1^{(t)}, \dots, \alpha_{n-1}^{(t)}, u, \alpha_{n+1}^{(t)}, \dots, \alpha_N^{(t)}\big)$$

update $\alpha_n^{(t+1)} \leftarrow u^\star$
$\alpha_{n'}^{(t+1)} \leftarrow \alpha_{n'}^{(t)}$ for $n' \neq n$ *(unchanged)*

end for

Issues with SVM

- There is no obvious probabilistic interpretation of SVM.
- Extension to multi-class is non-trivial (see Section 14.5.2.4 of KPM book).

[1] The pseudocode here is for coordinate **de**scent, that is to minimize a function. For the equivalent problem of maximizing (coordinate **a**scent), either change this line to arg max, or use the arg min of minus the objective function.

Machine Learning Course - CS-433

Kernel Ridge Regression and the Kernel Trick

Oct 27, 2020

changes by Nicolas Flammarion 2020, changes by Martin Jaggi 2019, changes by Rüdiger Urbanke 2018, changes by Martin Jaggi 2016, 2017 ©Mohammad Emtiyaz Khan 2015

Last updated on: October 25, 2020

EPFL

Motivation

In our last lecture we formulated the optimization problem corresponding to SVMs. We then derived an alternative formulation using duality. We saw that in this alternative formulation the data only enters in the form of a "kernel" $\mathbf{K} = \mathbf{X}\mathbf{X}^\top$.

The aim of today is the following. First, we will discuss a second problem, namely ridge regression, that admits an alternative formulation that is "kernelized," i.e., the alternative formulation depends on the data only via the kernel $\mathbf{K} = \mathbf{X}\mathbf{X}^\top$. Second, we will see that for any kernelized problem we can apply the *kernel trick*. This trick will allow us to use a significantly augmented feature vector without incurring extra costs. We will discuss what kernel functions are *admissible* for this trick and how to construct new kernel functions from old ones.

Even though we present the kernel trick in the context of ridge regression it will hopefully be clear by the end that the same trick can be applied to any problem that can be brought into kernelized form.

Alternative formulation of ridge regression

Recall that ridge regression corresponds to the following optimization problem:

$$\min_{\mathbf{w}} \quad \tfrac{1}{2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \tfrac{\lambda}{2}\|\mathbf{w}\|^2.$$

It has the solution

$$\mathbf{w}^\star = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_D)^{-1}\mathbf{X}^\top\mathbf{y}.$$

We claim that this solution can be written alternatively as

$$\mathbf{w}^\star = \mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1}\mathbf{y}. \tag{1}$$

This second formulation can be proved using the following identity: let $\mathbf{P}$ be an $N \times M$ matrix and $\mathbf{Q}$ be an $M \times N$ matrix. Then, trivially,

$$\mathbf{P}(\mathbf{Q}\mathbf{P} + \mathbf{I}_M) = \mathbf{P}\mathbf{Q}\mathbf{P} + \mathbf{P} = (\mathbf{P}\mathbf{Q} + \mathbf{I}_N)\mathbf{P}.$$

If we now assume that $(\mathbf{Q}\mathbf{P} + \mathbf{I}_M)$ and $(\mathbf{P}\mathbf{Q} + \mathbf{I}_N)$ are invertible we have the identity

$$(\mathbf{P}\mathbf{Q} + \mathbf{I}_N)^{-1}\mathbf{P} = \mathbf{P}(\mathbf{Q}\mathbf{P} + \mathbf{I}_M)^{-1}.$$

To derive from this general statement our alternative representation, let $\mathbf{P} = \mathbf{X}^\top$ and $\mathbf{Q} = \frac{1}{\lambda}\mathbf{X}$.
Why is this alternative representation useful?

1. Define $\boldsymbol{\alpha}^\star = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1}\mathbf{y}$. Then we can write

$$\mathbf{w}^\star = \mathbf{X}^\top \boldsymbol{\alpha}^\star. \tag{2}$$

   From this representation we see that $\mathbf{w}^\star$ lies in the column space of $\mathbf{X}^\top$, i.e., the space spanned by the feature vectors.

2. The original formulation involves computation of order $O(D^3 + ND^2)$, while the second can be computed in time $O(N^3 + DN^2)$. Hence it depends on the size of D and N which of the two is more efficient.

3. We will see that (1) and (2) are the crucial ingredients for the kernel trick to work.

The representer theorem

The representer theorem generalizes this result: for a $\mathbf{w}^\star$ minimizing the following function for any $\mathcal{L}_n$,

$$\min_{\mathbf{w}} \sum_{n=1}^{N} \mathcal{L}_n(\mathbf{x}_n^\top \mathbf{w}, y_n) + \tfrac{\lambda}{2}\|\mathbf{w}\|^2$$

there exists $\boldsymbol{\alpha}^\star$ such that $\mathbf{w}^\star = \mathbf{X}^\top \boldsymbol{\alpha}^\star$.

Such a general statement was originally proved by *Schölkopf, Herbrich and Smola (2001)*.

Kernelized ridge regression

Let us go back to ridge regression. The representer theorem allows us to write an equivalent optimization problem in terms of $\boldsymbol{\alpha}$:

$$\mathbf{w}^\star = \arg\min_{\mathbf{w}} \quad \tfrac{1}{2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \tfrac{\lambda}{2}\|\mathbf{w}\|^2$$

$$\boldsymbol{\alpha}^\star = \arg\min_{\boldsymbol{\alpha}} \; \frac{1}{2}\boldsymbol{\alpha}^\top(\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)\boldsymbol{\alpha} - \boldsymbol{\alpha}^\top \mathbf{y}.$$

To see that these two problems have equivalent solutions, note that if we take the gradient of the second expression we get $(\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)\boldsymbol{\alpha} - \mathbf{y}$. Setting this to 0 and solving for $\boldsymbol{\alpha}$ results in

$$\boldsymbol{\alpha}^\star = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1}\mathbf{y}.$$

If we combine this with the representer theorem $\mathbf{w}^\star = \mathbf{X}^\top \boldsymbol{\alpha}^\star$ we find back the solution (1).

As we discussed previously, depending on the D, the dimension of the feature space, and N, the number of samples, one or the other of the two formulations might be more efficient. But there is an arguably even more important reason why the second expression is of interest. In this second expression the data only enters in terms of the kernel matrix $\mathbf{K} = \mathbf{X}\mathbf{X}^\top$.

Kernel functions

Recall that the kernel is defined as

$$\mathbf{K} = \mathbf{X}\mathbf{X}^\top = \begin{bmatrix} \mathbf{x}_1^\top\mathbf{x}_1 & \mathbf{x}_1^\top\mathbf{x}_2 & \dots & \mathbf{x}_1^\top\mathbf{x}_N \\ \mathbf{x}_2^\top\mathbf{x}_1 & \mathbf{x}_2^\top\mathbf{x}_2 & \dots & \mathbf{x}_2^\top\mathbf{x}_N \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}_N^\top\mathbf{x}_1 & \mathbf{x}_N^\top\mathbf{x}_2 & \dots & \mathbf{x}_N^\top\mathbf{x}_N \end{bmatrix}.$$

For reasons that will become clear shortly, we call this the *linear* kernel.

Assume that we had first augmented the feature space to $\boldsymbol{\phi}(\mathbf{x})$. The associated kernel with basis functions $\boldsymbol{\phi}(\mathbf{x})$ would then be $\mathbf{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^\top$. Explicitly,

$$\begin{bmatrix} \boldsymbol{\phi}(\mathbf{x}_1)^\top\boldsymbol{\phi}(\mathbf{x}_1) & \boldsymbol{\phi}(\mathbf{x}_1)^\top\boldsymbol{\phi}(\mathbf{x}_2) & \dots & \boldsymbol{\phi}(\mathbf{x}_1)^\top\boldsymbol{\phi}(\mathbf{x}_N) \\ \boldsymbol{\phi}(\mathbf{x}_2)^\top\boldsymbol{\phi}(\mathbf{x}_1) & \boldsymbol{\phi}(\mathbf{x}_2)^\top\boldsymbol{\phi}(\mathbf{x}_2) & \dots & \boldsymbol{\phi}(\mathbf{x}_2)^\top\boldsymbol{\phi}(\mathbf{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{\phi}(\mathbf{x}_N)^\top\boldsymbol{\phi}(\mathbf{x}_1) & \boldsymbol{\phi}(\mathbf{x}_N)^\top\boldsymbol{\phi}(\mathbf{x}_2) & \dots & \boldsymbol{\phi}(\mathbf{x}_N)^\top\boldsymbol{\phi}(\mathbf{x}_N) \end{bmatrix}.$$

We have already discussed that sometimes it is useful to augment the feature space. This will lead to a more powerful model. Here is a link to a video explaining this point in more detail: https://www.youtube.com/watch?v=3liCbRZPrZA

The kernel trick

The big advantage of using kernels is that rather than first augmenting the feature space and then computing the kernel, we can do both steps together, and we can do it more efficiently. Let us discuss how this works.
Let us define a "kernel function" $\kappa(\mathbf{x}, \mathbf{x}')$ and let us compute the (i, j)-th entry of $\mathbf{K}$ as $K_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$. For the right choice of kernel κ it turns out to be equivalent to first augmenting the features to some suitable $\boldsymbol{\phi}(\mathbf{x})$ and then computing the inner product

$$\boldsymbol{\phi}(\mathbf{x})^\top \boldsymbol{\phi}(\mathbf{x}')$$

in the augmented space. In other words, for the right choices we have

$$\kappa(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^\top \boldsymbol{\phi}(\mathbf{x}') \ .$$

This is probably best seen by looking at examples:

1. To start trivially, if we pick the linear kernel $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \mathbf{x}'$, then the corresponding feature map is of course $\boldsymbol{\phi}(\mathbf{x}') = \mathbf{x}'$.

2. Assume that $\mathbf{x} \in \mathbb{R}$, i.e., $\mathbf{x}$ is a scalar. The kernel $\kappa(x, x') = (xx')^2$ corresponds to $\phi(x) = x^2$.

3. Assume that $\mathbf{x} \in \mathbb{R}^3$, i.e., $\mathbf{x}$ is a vector of dimension 3. The kernel $\kappa(\mathbf{x}, \mathbf{x}') = (x_1 x_1' + x_2 x_2' + x_3 x_3')^2$ corresponds to

   $$\boldsymbol{\phi}(\mathbf{x})^\top = \left[\ x_1^2, \ x_2^2, \ x_3^2, \ \sqrt{2} x_1 x_2, \ \sqrt{2} x_1 x_3, \ \sqrt{2} x_2 x_3 \ \right] .$$

   This is an example of what is called a *polynomial kernel*.

4. The kernel

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left[-(\mathbf{x} - \mathbf{x}')^\top(\mathbf{x} - \mathbf{x}')\right]$$

corresponds to an infinite feature map! It is called the the *radial basis function* (RBF) kernel. In order to look at this more in detail, consider the simple case where the $\mathbf{x}$ and $\mathbf{x}'$ are scalars. In this case we have the expansion

$$\mathbf{K}(x, x') = e^{-(x)^2} e^{-(x')^2} \sum_{k=0}^{\infty} \frac{2^k (x)^k (x')^k}{k!}.$$

We see that we can think of this as the inner product of infinite-dimensional vectors whose k-th component, $k = 0, 1, \cdots$ is equal to

$$e^{-(x)^2} \sqrt{\frac{2^k}{k!}} (x)^k \text{ and } e^{-(x')^2} \sqrt{\frac{2^k}{k!}} (x')^k,$$

respectively. And although this is not obvious, let us state that this kernel cannot be represented as an inner product in a finite-dimensional space.

5. You can find many further examples in Section 14.2 of Murphy's book.

6. Building new kernels from old kernels.

   a) $\kappa(\mathbf{x}, \mathbf{x}') = a\kappa_1(\mathbf{x}, \mathbf{x}') + b\kappa_2(\mathbf{x}, \mathbf{x}')$ for all $a, b \geq 0$.

   Proof. By assumption κ_1 and κ_2 are valid kernels. Hence there exist feature maps ϕ_1 and ϕ_2 so that

$$\kappa_1(\mathbf{x}, \mathbf{x}') = \phi_1(\mathbf{x})^\top \phi_1(\mathbf{x}'),$$
$$\kappa_2(\mathbf{x}, \mathbf{x}') = \phi_2(\mathbf{x})^\top \phi_2(\mathbf{x}').$$

Hence,

$$\kappa(\mathbf{x}, \mathbf{x}') = a\phi_1(\mathbf{x})^\top \phi_1(\mathbf{x}') + b\phi_2(\mathbf{x})^\top \phi_2(\mathbf{x}').$$

This can be represented as an inner product via the feature map

$$(\sqrt{a}\phi_1(\cdot), \sqrt{b}\phi_2(\cdot)).$$

□

b) $\kappa(\mathbf{x}, \mathbf{x}') = \kappa_1(\mathbf{x}, \mathbf{x}')\kappa_2(\mathbf{x}, \mathbf{x}')$.

Proof. Let the two feature maps be ϕ_1 and ϕ_2. Assume that they are of dimensions d_1 and d_2. Then ϕ is a feature map of dimension $d_1 d_2$ of the form

$$\begin{aligned}\phi(\mathbf{x})^\top =&((\phi_1(\mathbf{x}))_1(\phi_2(\mathbf{x}))_1, \cdots, (\phi_1(\mathbf{x}))_1(\phi_2(\mathbf{x}))_{d_2}, \cdots, \\ &(\phi_1(\mathbf{x}))_{d_1}(\phi_2(\mathbf{x}))_1, \cdots, (\phi_1(\mathbf{x}))_{d_1}(\phi_2(\mathbf{x}))_{d_2}).\end{aligned}$$

□

c) $\kappa(\mathbf{x}, \mathbf{x}') = \kappa_1(f(\mathbf{x}), f(\mathbf{x}'))$ for any f from the domain to itself.

Proof. Let $\phi_1(\cdot)$ be the feature map corresponding to $\kappa_1(\cdot, \cdot)$. Then by direct inspection we see that $\phi(\cdot) = \phi_1(f(\cdot))$ is the feature map corresponding to $\kappa(f(\cdot), f(\cdot))$. Indeed,

$$\phi_1(f(\mathbf{x}))^\top \phi_1(f(\mathbf{x}')) = \kappa_1(f(\mathbf{x}), f(\mathbf{x}') = \kappa(\mathbf{x}, \mathbf{x}').$$

□

d) $\kappa(\mathbf{x}, \mathbf{x}') = f(\mathbf{x})f(\mathbf{x}')$ for any real-valued f. Clearly, $\phi(\mathbf{x}) = f(\mathbf{x})$ will be the corresponding feature map.

Classifying with the kernel K

We have seen so far how we can compute the optimal parameter vector $\boldsymbol{\alpha}$ using only the kernel (and not having to go to the extended feature space). We have also discussed that in some cases the feature space is in fact infinite. All this would not be useful if there was not also a way to do the prediction using only the kernel. Indeed this is possible.
Recall that the classifier predicts $y = \phi(\mathbf{x})^\top \mathbf{w}^*$ which, using (2), can be expressed as

$$y = \phi(\mathbf{x})^\top \phi(\mathbf{X})^\top \boldsymbol{\alpha} = \sum_{n=1}^{N} \kappa(\mathbf{x}, \mathbf{x}_n)\boldsymbol{\alpha}_n.$$

I.e., we can express the prediction in terms of the kernel function applied to the new feature vector and the data vector in the original space and do not need to go into the augmented space.
From this expression we can also clearly see that although the classifier in the extended space $\phi(\mathbf{x})$ is linear, if we look at the decision regions in the original space $\mathbf{x}$ it is non-linear.

Properties of kernels: Mercer's Condition

A natural question is the following: how can we ensure that there exists a $\boldsymbol{\phi}$ corresponding to a given kernel function κ?

I.e., how do we ensure that the kernel function is an inner-product in some feature space?
Mercer's condition statest that this is true if and only if the following two conditions are fulfilled. In the following, given the kernel function κ and some arbitrary input set $\{\mathbf{x}_n\}_{n=1}^N$, let K be the associated kernel matrix, $\mathbf{K}_{i,j} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$.

1. The kernel function κ must be symmetric, i.e. $\kappa(\mathbf{x}, \mathbf{x}') = \kappa(\mathbf{x}', \mathbf{x})$; equivalently, the kernel matrix $\mathbf{K}$ must be symmetric for all possible input sets.

2. The kernel matrix $\mathbf{K}$ must be positive semi-definite for all possible input sets.

Machine Learning Course - CS-433

Neural Nets – Basic Structure

November 3, 2020

changes by Nicolas Flammarion 2020, changes by Rüdiger Urbanke 2019,2018,2017;

Last updated on: November 3, 2020

EPFL

Outline

We started this course with a basic setup. We are given a training set $S_t = \{(y_n, \mathbf{x}_n)\}$ and our aim is classification. We have seen that simple linear classification schemes like logistic regression

$$p(y \mid \mathbf{x}^\top \mathbf{w}) = \frac{e^{\mathbf{x}^\top \mathbf{w} y}}{1 + e^{\mathbf{x}^\top \mathbf{w} y}}$$

can some times work very well but they have their limits.
The key to improving such schemes is to add well chosen features to the original data vector. E.g., assume that our data is two-dimensional, where all data with label $y = 0$ lies inside the unit circle and all data with label $y = 1$ lies outside the unit circle. A linear scheme, limited to the original input, cannot classify this data well. But if we add the features $\mathbf{x}_1^2 + \mathbf{x}_2^2$ and the constant to the input then the linear classification becomes trivial.
In "real" applications we are faced with the problem that we do not know a priori what features are useful. One option is to add as many features as possible. E.g., we could add all polynomial terms up to some order to our feature vector. But this quickly becomes computationally infeasible and can also lead to overfitting.
One way to address the computational issue is to use the "kernel trick." Alternatively, we can only add very few new features but have them designed by a domain experts.
But would it not be nice if we could *learn* the features from the data in the same way as we learn the weights of the linear classifier? This is what neural networks allow us to do.

There is currently a lot of excitement about neural networks and its many applications. At the end of this short tutorial you will unlikely be able to program a neural net to play Go like a grandmaster. Many small tricks and lots of patience and computing power are needed to train neural nets for complicated tasks. But you will be able to write small scripts to solve standard handwriting recognition challenges. We will focus on basic questions.
We highly recommend the web tutorial by Michael Nielsen, neuralnetworksanddeeplearning.com and we will follow it in many aspects.

The Basic Structure

Let us look at the structure of a neural network. It is shown in Figure 1. This is a neural net with one *input* layer of size D, L *hidden* layers of size K, and one *output* layer. It is a *feedfoward* network: the computation performed by the network starts with the input from the left and flows to the right. There is no feedback loop.
As always, we assume that our input is a D-dimensional vector. We see that there is a node drawn in Figure 1 for each of the D components of $\mathbf{x}$. We denote these nodes by $x_i^{(0)}$, where the superscript (0) specifies that this is the input layer.
The same network can be used for regression as well as classification. The only difference will be in the output layer.
Let us discuss the exact computation that is performed by this network. We already described the input layer. Let us now look at the hidden layers. Let us assume that there are

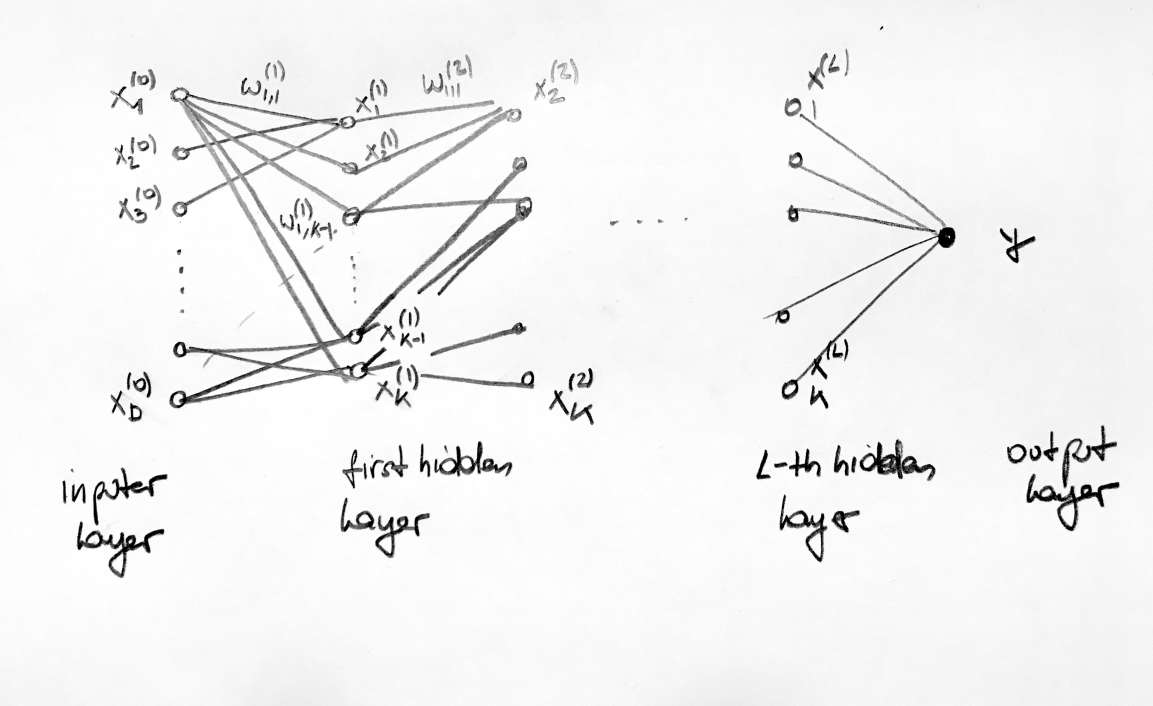


Figure 1: A neural network with one input layer, L hidden layers, and one output layer.

K nodes in each hidden layer, where K is a hyper-parameter that has to be chosen by the user and can/should be optimized via validation. There is no reason that all hidden layers should have the same size but we will stick to this simple model. How many layers are there typically? Not long ago, typical networks might have just had one or a few hidden layers. Modern applications have "deep" nets with sometimes hundreds of layers. Training such deep nets poses new and challenging problems and we will have more to say about this later.

Each node in the hidden layer l, $l = 1, \cdots, L$, is connected to all the nodes in the previous layer via a weighted edge. We denote the edge from node i in layer $l-1$ to the node j in layer l by $w_{i,j}^{(l)}$. The super-script (l) indicates that these are the weights of edges that lead to layer l.

The output at the node j in layer l is denoted by $x_j^{(l)}$ and it is given by

$$x_j^{(l)} = \phi\Big(\sum_i w_{i,j}^{(l)} x_i^{(l-1)} + b_j^{(l)}\Big).$$

In words, in order to compute the output we first compute the weighted sum of the inputs and then apply a function ϕ to this sum.

A few remarks are in order. The constant term $b_j^{(l)}$ is called the *bias term* and is a parameter like any of the weights $w_{i,j}^{(l)}$. The *learning part* will consist of choosing all these parameters appropriately for the task. The function $\phi(\cdot)$ is called the *activation* function. Many possibilities exist for choosing this function and we will explore some choices later one. For now, let us just mention one of the most popular one, namely the *sigmoid function* $\phi(x) = \frac{1}{1+e^{-x}}$. We have encountered this function already. It is the same as the logistic function. For future reference, Figure 2 shows a plot of this function. Note that the function is increasing from 0 to 1 and that for very small (negative) and very large (positive) values of the argument the function is very flat. As we will discuss, this will cause troubles when training the net since the derivative will vanish there.

It is crucial that this function is *non-linear*. Why is this? Assume not, then the whole neural-net would just be a highly factorized linear function of the input data and there would be no gain compared to standard linear regression/classification. Although it is possible to choose different activation functions for different nodes, it is common to choose the same function within the network.

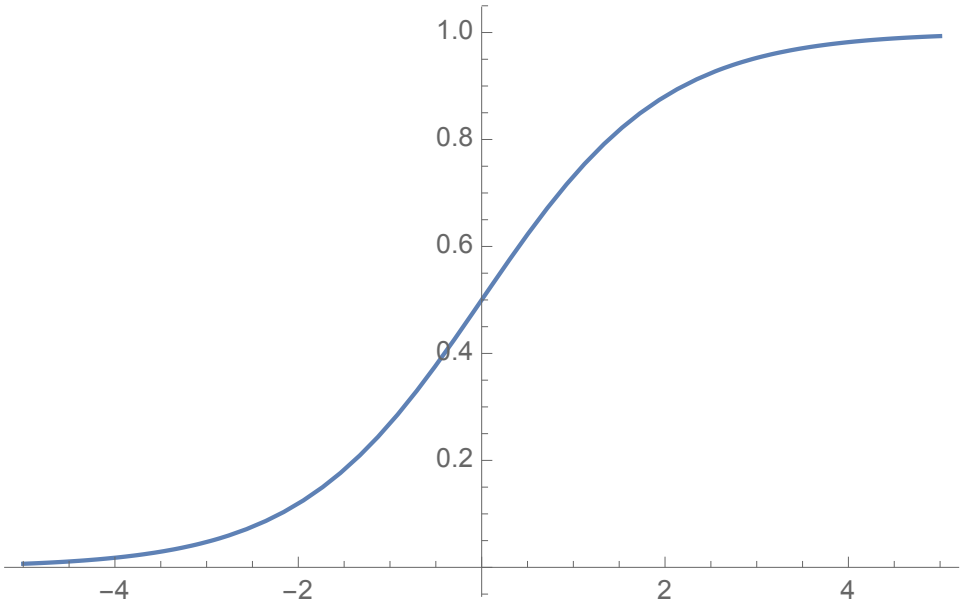


Figure 2: The sigmoid function $f(x) = \frac{1}{1+e^{-x}}$.

To connect back to our previous discussion we can decompose the neural network into two parts. The first part comprises the input layer and all the L hidden layers. The task of this part of the net is to transform the original input into a more suitable representation. In other words, this part of the net represents a *function* from $\mathbb{R}^D$ to $\mathbb{R}^K$. It performs the task that typically was done by domain experts, namely finding suitable features of the input. We will soon discuss what functions such a network can implement. We will see that this simple structure can approximate any continuous function arbitrarily closely provided only that we allow K to be sufficiently large.

Now that we have (hopefully) a suitable representation of the data, the final layer performs the desired ML task. This means that it is either our trusted linear regressor or perhaps a linear classifier. Presumably at this point the regression/classification task is easy. So a simple linear regres-

sor/classifier suffices.

Machine Learning Course - CS-433

Neural Nets – Representation Power

November 3, 2020

changes by Nicolas Flammarion 2020, changes by Rüdiger Urbanke 2019,2018,2017;

Last updated on: November 3, 2020

EPFL

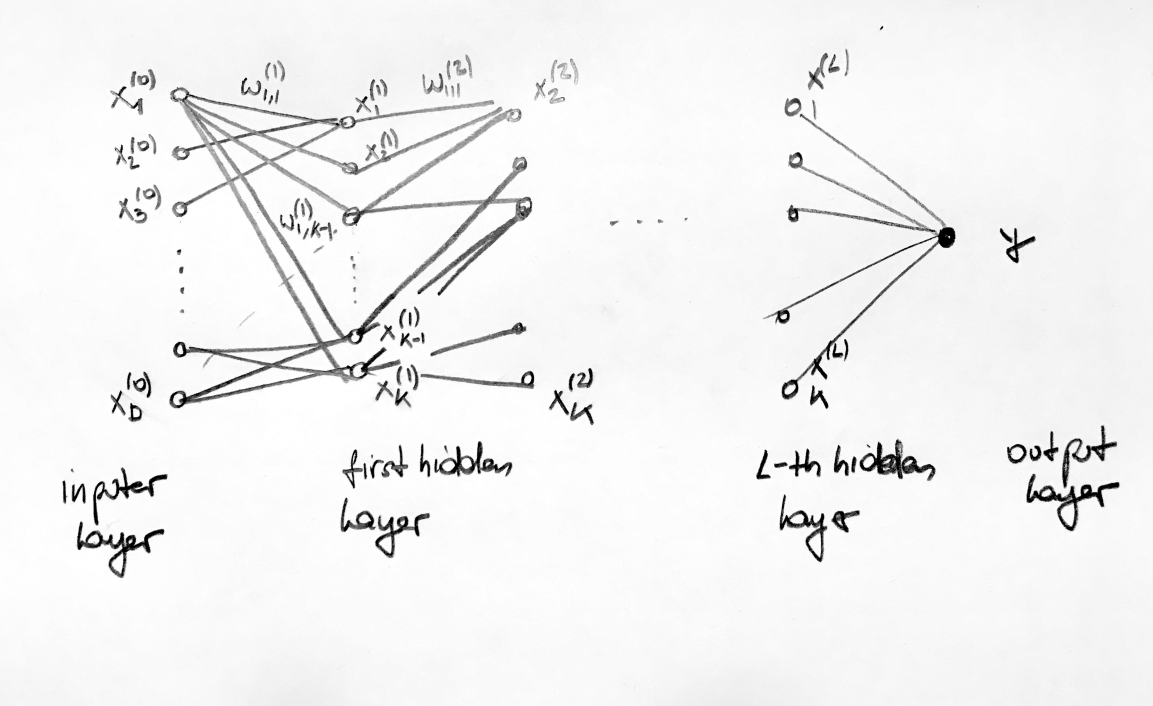


Figure 1: A neural network with one input layer, L hidden layers, and one output layer.

Motivation

In the last lecture we introduced the basic structure of a neural net. It is shown in Figure 1. Recall that we can split this network into two parts. The first part comprises the input layer and all the L hidden layers. The task of this part of the net is to transform the original input into a more suitable representation. In other words, this part of the net represents a *function* from $\mathbb{R}^D$ to $\mathbb{R}^K$. The second part, represented by the final layer performs the actual ML task (regression or classification).

We will now focus on the first part. We will ask: How "powerful" are neural nets? More precisely, what functions $f(\mathbf{x})$ can they represent, or better, what functions can they approximate? We will see that even relatively simple nets (with at most two hidden layers) are capable of approximating any

continuous function arbitrarily closely on a bounded domain, assuming only that we allow a large number of nodes and arbitrary weights and biases.

We will not take a rigorous approach. Rather, we will follow the lead of Michael Nielsen and give a heuristic but rather convincing argument which shows why neural nets are so expressive.

If you are interested in a rigorous approach we recommend that you read either "Approximation by superposition of a sigmoidal function" by Cybenko (1989), or "Universal approximation bounds for superpositions of a sigmoidal function" by Barron (1993). Both of these papers deal with networks with a single layer and sigmoids as approximation functions. Since then, many further approximation results for various network structures, activation functions, and approximation measures have been derived.

Before we get into our heuristic argument let us state the main theorem of the paper by Barron. This gives you a flavor of what kind of results can be proved.

Lemma. *Let $f : \mathbb{R}^D \to \mathbb{R}$ be a function such that*

$$\int_{\mathbb{R}^D} |\omega||\tilde{f}(\omega)|d\omega \leq C,$$

where

$$\tilde{f}(\omega) = \int_{\mathbb{R}^D} f(\mathbf{x})e^{-j\omega^\top \mathbf{x}}d\mathbf{x}$$

is the Fourier transform of $f(\mathbf{x})$.
Then for all $n \geq 1$, there exists a function f_n of the form

$$f_n(\mathbf{x}) = \sum_{j=1}^{n} c_j \phi(\mathbf{x}^\top \mathbf{w}_j + b_j) + c_0,$$

i.e., a function that is representable by a NN with one hidden layer with n nodes and "sigmoid-like" activation functions so that

$$\int_{|\mathbf{x}|\leq r} (f(\mathbf{x}) - f_n(\mathbf{x}))^2 d\mathbf{x} \leq \frac{(2Cr)^2}{n}.$$

Discussion: First note that the condition on the Fourier transform is a "smoothness condition." E.g., functions so that $\int_{\mathbb{R}^D} |\omega||\tilde{f}(\omega)|d\omega < \infty$ can be shown to be continuously differentiable.
Second note that the lemma only guarantees a good approximation in a bounded domain. The larger the domain, the more nodes we need in order to approximate a function to the same level (see the term r^2, where r is the radius of the ball where we want the approximation to be good, in the upper bound).

Third, this is an approximation "in average", more precisely in L_2-norm. We will mostly discuss approximations in this sense but come back to this point at the end.
Fourth, the approximation f_n with n terms corresponds exactly to our model of a neural net with one hidden layer containing n nodes and sigmoids as activation functions.
Fifth, the theorem applies to all activation functions that are "sigmoid-like," i.e., all activation functions whose left limit is 0, whose right limit is 1, and that are sufficiently smooth.
In words the lemma says that a sufficiently "smooth" function can be approximated by a neural net with one hidden layer and the approximation error goes down like one over the number of nodes in the hidden layer. Note that this is a very fast convergence.

Approximation in Average

We will now give a very simple and intuitive explanation why neural nets with a sigmoid as activation function and at most two hidden layers have already a large expressive power. This explanation will not be rigorous and it will fall far short of the stated lemma by Barron which proves that in fact we only need one hidden layer and not too many hidden nodes. We will search for an approximation "in average", i.e., an approximation so that the integral over the absolute value of the difference is arbitrarily small.

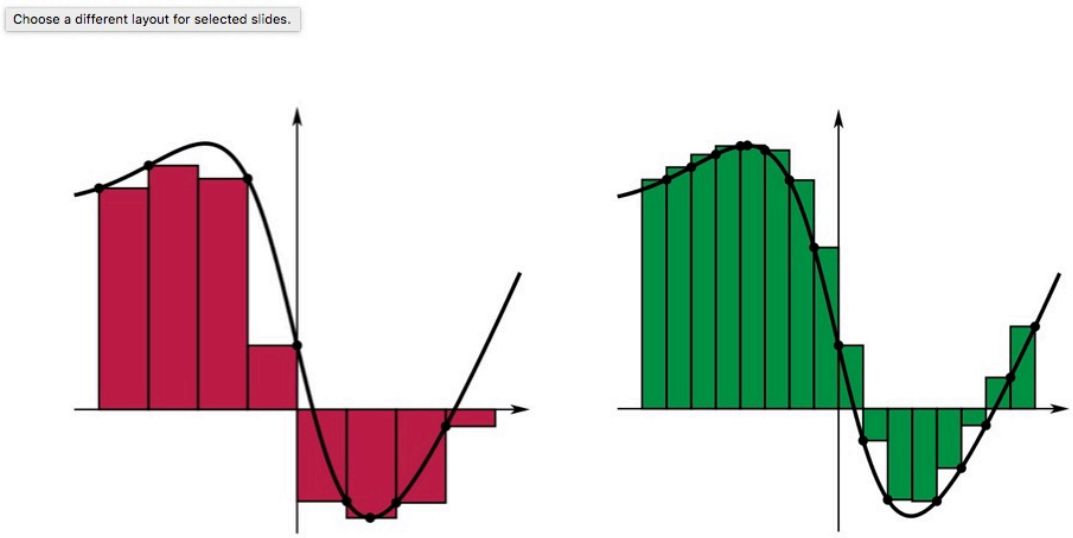


Figure 2: A lower and an upper Riemann sum.

Functions $\mathbb{R} \to \mathbb{R}$

We start with a scalar function $f(x)$ on a bounded domain. Recall that if this function is continuous then it is Riemann integrable, i.e., it can be approximated arbitrarily closely by "upper" and "lower" sums of rectangles, see Figure 2. Of course, we might need a lot of such rectangles to approximate the area with an error of at most ϵ, but for every $\epsilon > 0$ we can find such an approximation.

We will now show that if we do not limit the weights, then with two hidden nodes (of a neural network with one hidden layer) we can construct a function which is arbitrarily close to a given rectangle. But since, as we have just seen, a finite number of rectangles suffices to approximate a bounded continuous function arbitrarily closely, it follows that with a finite number of hidden nodes of a neural network with one hidden layer we can approximate any such function arbitrarily closely (in the sense that the integral of the absolute value

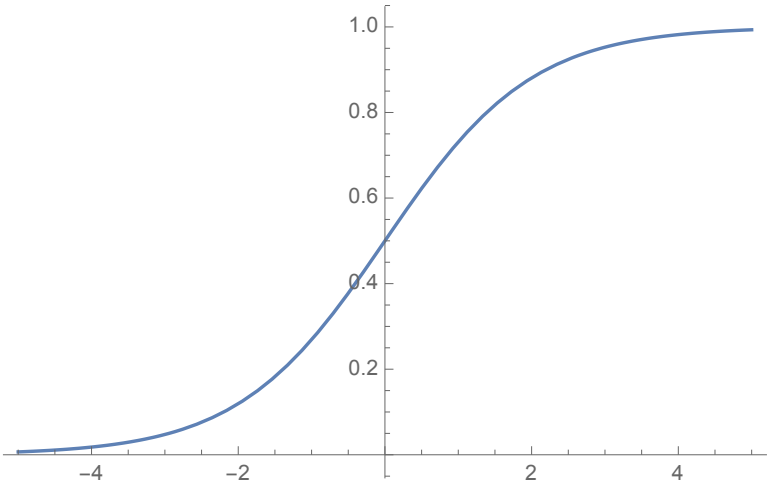


Figure 3: The sigmoid function.

of their difference is arbitrarily small).

The following idea is taken from the tutorial by Michael Nielsen, http://neuralnetworksanddeeplearning.com.

Let $\phi(x) = \frac{1}{1+e^{-x}}$ be the sigmoid function. We have encountered it already in the last lecture. But here it is again in Figure 3.

Consider the function $f(x) = \phi(w(x-b))$, where w is the *weight* of a particular edge and $-wb$ is the *bias* term.

Note that if $w \geq 0$ then $f(x)$ is an increasing function that increases smoothly from 0 to 1. Further, the "transition" happens at the spot $x = b$, i.e., at this value of x the function has the value $\frac{1}{2}$. The transition is the faster the larger we choose the weight w. In fact, if we set $b = 0$ so that the transition from 0 to 1 happens at $x = 0$, then the derivative of the function at 0 is $w/4$. In other words, the *width of the transition* is of order $4/w$.

Therefore, if we want to create a rectangle that jumps from 0 to 1 at $x = a$ and jumps back to 0 at $x = b$, $a < b$, then

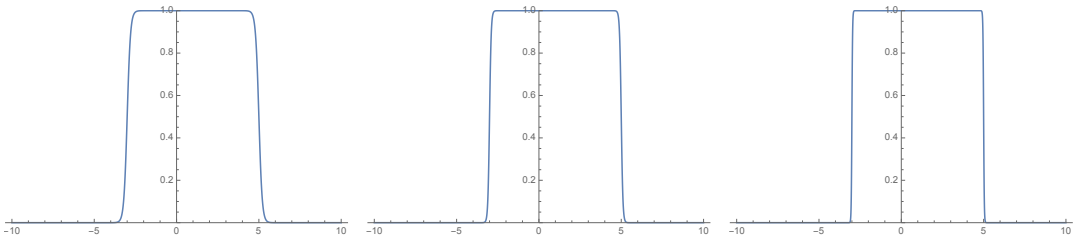


Figure 4: An approximate rectangle of the form $\phi(w(x-a))-\phi(w(x-b))$ with $w = 10, 20$, and 50, respectively.

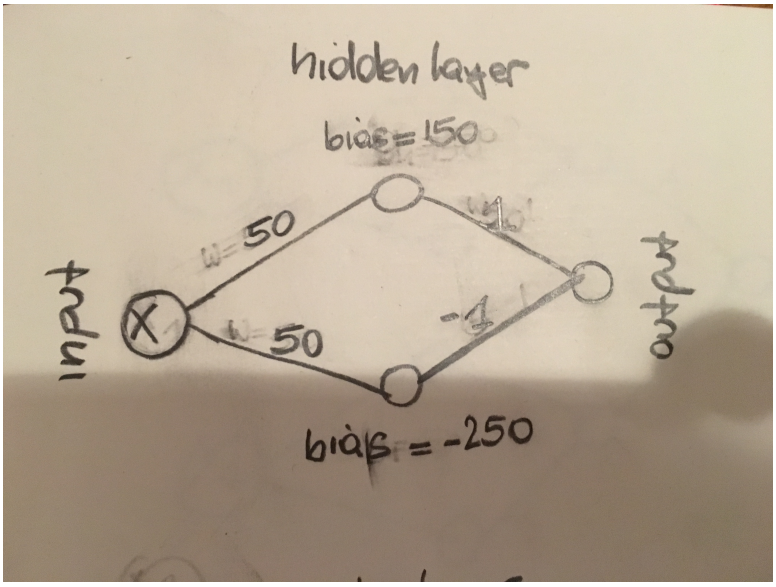


Figure 5: A simple NN implementation of equation (1).

we can accomplish this by taking

$$\phi(w(x-a))-\phi(w(x-b)), \tag{1}$$

and taking w to be very large. This is shown in Figure 4 where the three figures correspond to $w = 10, 20$ and 50, respectively and $a = -3$ and $b = 5$. We see that for large values of w the result is barely distinguishable from a true rectangle.

Note that equation (1) has a very simple representation in form of a neural net. This is shown in Figure 5. There is one input node which contains the value x. This value is

multiplied by some large weight (in the figure it is 50) and it is then forwarded to the two hidden nodes. One of these hidden nodes has a bias of 150 the other one has a bias of -250, so that the sums at these two hidden nodes are $50(x+3)$ and $50(x-5)$, respectively. Each node applies the sigmoid function and forwards the result to the output layer. The edge from the top hidden node to the output has weight 1 and the one from the bottom hidden node to the output has weight -1. The output node adds the two inputs. The result is $\phi(50(x+3)) - \phi(50(x-5))$, which is approximately a unit-height rectangle from -3 to 5. If we want a rectangle of height h, use the weights h and $-h$ in the second layer instead of the weights 1 and -1.

It is hopefully clear at this point why any continuous function on a bounded domain can be approximated via a neural network with one hidden layer. Let us summarize in telegram style: Take the function. Approximate it in the Riemann sense. Approximate each of the rectangles in the Riemann sum by means of two nodes in the hidden layer of a neural net. Compute the sum (with appropriate sign) of all the hidden layers at the output node. If we are using a Riemann sum with K rectangles we get therefore a neural network approximation with one hidden layer containing $2K$ nodes

A few remarks are in order:

1. The same intuition applies to many activation functions. All we have used is that the activation function has left limit 0 and right limit 1, just like for Barron's result.

2. Whereas Barron's result gave us a strong bound on

the number of required nodes, our intuitive explanation gave us no bound.

3. The above approximation only works if we allow the weights to become arbitrarily large. Of course, very large weights would likely cause problems in practice. It is also not clear why we should need them. There is no fundamental reason why we should first approximate rectangles very precisely which then in turn approximate a continous function.

Functions $\mathbb{R}^D \to \mathbb{R}$

So far we have talked about a real function of a single variable. What about functions over $\mathbb{R}^D$? We will see that the same principle applies but we will need one more layer. The extra idea that is needed to extend the above scheme from one to several dimensions is already present in $\mathbb{R}^2$. We will therefore stick to this case. You should have no trouble figuring out how to extend it to $\mathbb{R}^D$.

As before, it will suffice if we can show how to approximate any two-dimensional rectangle. In fact, all we need are two-dimensional rectangles that are parallel to the two axes.

Let the two inputs/axes be x_1 and x_2, represented by two nodes in the input layer. Assume that we want to represent a rectangle of height 1 that extends from a_1 to b_1, $a_1 < b_1$, on the x_1 axis and from a_2 to b_2, $a_2 < b_2$, on the x_2 axis.

Consider the neural net in Figure 6. It represents a rectangle that goes from a_1 to b_1 in the direction of the x_1 axis but is unbounded in the x_2 direction. The function it represents is

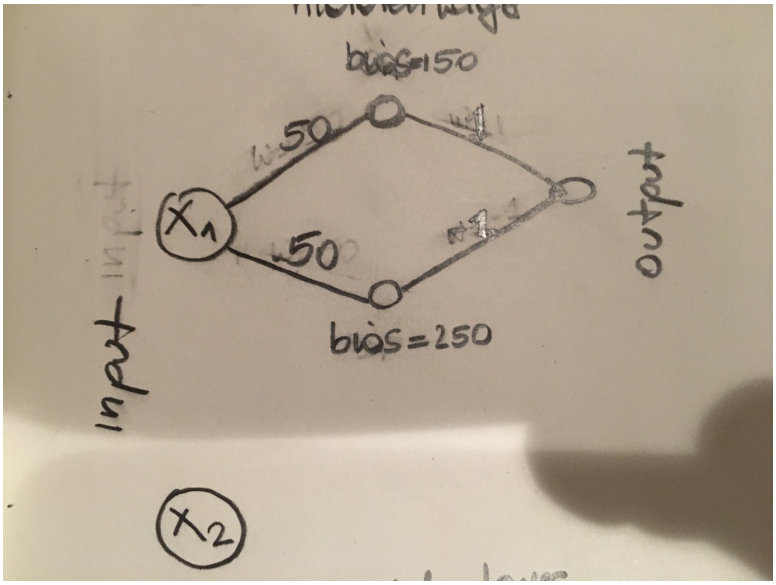


Figure 6: A 2-D rectangle bounded in one dimension and unbounded in the other.

shown Figure 7.

Let us add to this rectangle another one that is unbounded in the x_1 direction but extends from a_2 to b_2 in the direction of the x_2 axis. The corresponding neural net is shown in Figure 8 and the plot of the corresponding function is shown in Figure 9. This is close to what we want. In the region where we want the function to be 1 it is in fact close to 2 since we have there the sum of two functions, each of height 1. A trivial scaling by a factor 1/2 would bring it to the desired value in this region. But unfortunately we still have in addition the two "arms" that extend to infinity along each axis. Those have height 1.

But those additional unwanted "arms" are easily suppressed. Just apply the sigmoid function with some large weight and bias, lets say 3/2, to the node that sums up the two rectangles. A plot of the resulting function is shown in Figure 10. So instead of being the output node, as it was before, this

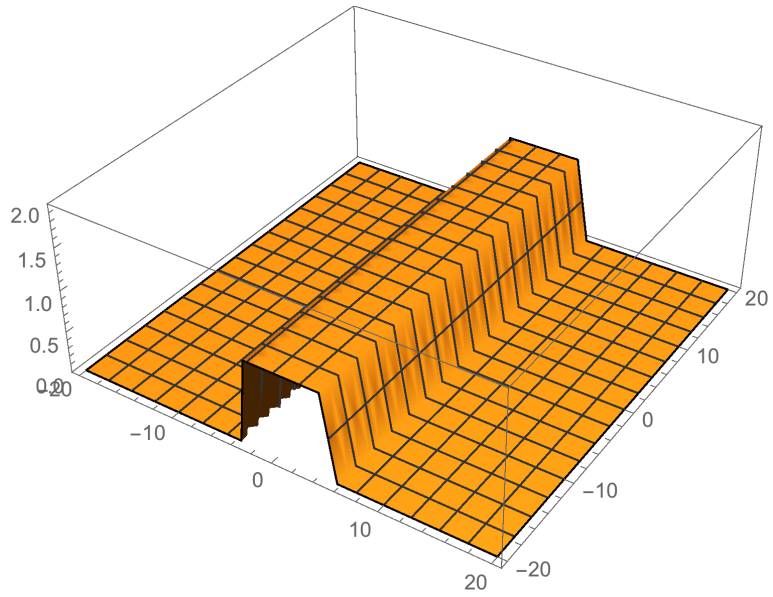


Figure 7: The function $\phi(w(x_1 + a_1)) - \phi(w(x_1 - b_1))$ corresponding to the NN shown in Figure 6. Here, $w = 50$, $a_1 = -3$, and $b_1 = 5$.

node now becomes a hidden node, forming one extra layer of hidden nodes. With this we have created the desired 2-dimensional rectangle. We can approximate any sufficiently smooth 2-dimensional function on a bounded domain by using a suitable number of those and adding them at the output node.

Point-wise Approximations and other Activation Functions

So far we have considered approximations "in average". I.e., we have seen for Barron's result that the L_2 norm can be made arbitrarily small. And for our intuitive derivation we took the Riemann integral as a starting point, i.e., we considered the L_1 norm.

We can also ask if a point-wise approximation with vanishing

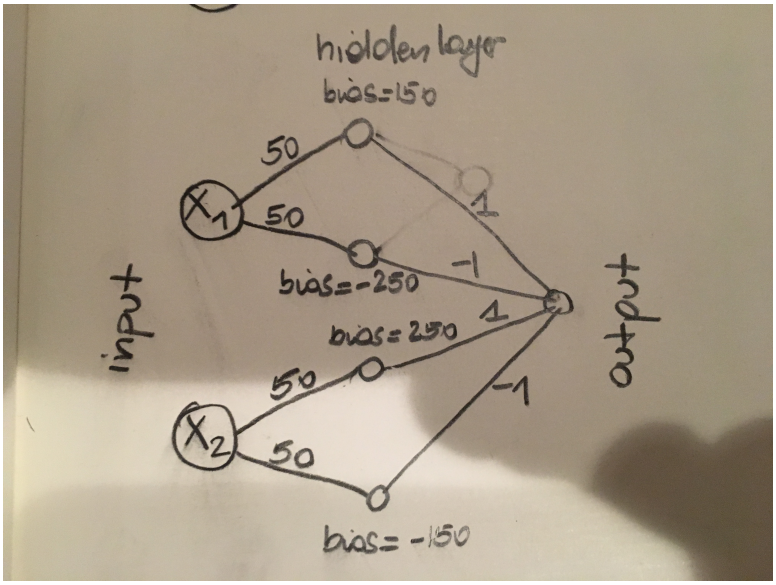


Figure 8: Sum of two rectangles, each bounded in one dimension and unbounded in the other.

error is possible, i.e., we can consider the L_∞ norm.
Further, we have limited our discussion so far to "sigmoid-type" activation functions, i.e., activation functions whose left limit is 0 and whose right limit is 1.
So let us now look at point-wise approximations with an activation function that is the *rectified linear function*,

$$(x)_+ = \max\{0, x\}.$$

Many other combinations (approximation criterion and activation function) are of course possible and have been considered.
Let $f(x)$ be a continuous function on a bounded domain. Without loss of generality we can assume that this domain is $[0, 1]$, rescaling and shifting the x-axis if necessary. The classical *Stone-Weierstrass* theorem says that for every $\epsilon > 0$, there exist a polynomial $p(x)$ so that for all $x \in [0, 1]$,

$$|f(x) - p(x)| < \epsilon.$$

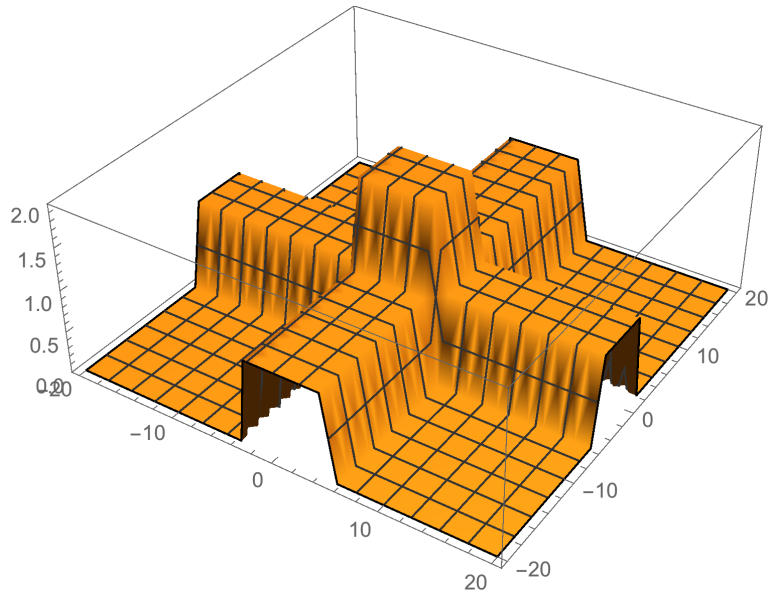


Figure 9: The function $g(x_1, x_2) = \phi(w(x_1+a_1)) - \phi(w(x_1-b_1)) + \phi(w(x_2+a_2)) - \phi(w(x_2-b_2))$ corresponding to the NN shown in Figure 8. Here, $w = 50$, $a_1 = -3$, $b_1 = 5$, $a_2 = -5$, $b_2 = 3$.

But such a function $f(x)$ can also be approximated in L_∞ norm by even simpler functions, namely continuous piecewise-linear functions, see Shektman, 1982. Let $q(x)$ be a continuous piecewise-linear function. Then it has the form

$$q(x) = \sum_{i=1}^{m} (a_i x + b_i) \mathbb{1}_{\{r_{i-1} \le x < r_i\}},$$

where $0 = r_0 < r_1 < \cdots < r_m = 1$ is a suitable partition of $[0, 1]$. Note that continuity imposes the constraints

$$a_i r_i + b_i = a_{i+1} r_i + b_{i+1}, i = 1, \cdots, m-1.$$

For our purpose it is more convenient to write it in the al-

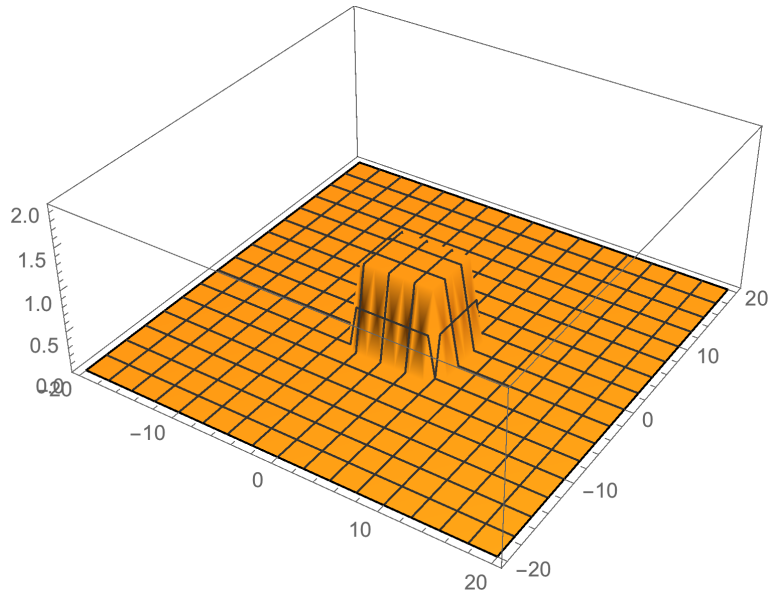


Figure 10: The function $\phi(w(g(x_1, x_2) - 3/2))$ where $g(x_1, x_2)$ is the function shown in Figure 9. This is now a good approximation of the desired rectangle.

ternative form

$$q(x) = \tilde{a}_1 x + \tilde{b}_1 + \sum_{i=2}^{m} \tilde{a}_i (x - \tilde{b}_i)_+.$$

Here, $\tilde{a}_1 = a_1$ and $\tilde{b}_1 = b_1$ and for $i = 2, \cdots, m$, the remaining parameters can be computed via the relations

$$a_i = \sum_{j=1}^{i} \tilde{a}_i,$$
$$\tilde{b}_i = r_{i-1}.$$

Each term in the sum on the right corresponds to one node in a hidden layer with input x, bias $-\tilde{b}_i$, and activation function $(x)_+$. The bias term $\tilde{b}_1$ can be absorbed into the bias term of the output node. This leaves the term $\tilde{a}_1 x$. This

term can also be represented by a node in the hidden layer with activation function $(x)_+$ by choosing $x_0 = 0$, since we only need the this representation to be correct in the range $[0, 1]$. So this shows that we can approximate any continuous function on a bounded domain by a neural net with one hidden layer in the L_∞ norm to arbitrary precision.
We have only considered the one-dimensional case. But it turns out that a similar scheme works also for higher dimensions. We skip the details.

Machine Learning Course - CS-433

Neural Nets – Training: SGD and Backpropagation

November 3, 2020

changes by Nicolas Flammarion 2020, changes by Rüdiger Urbanke 2019,2018,2017;

Last updated on: November 3, 2020

EPFL

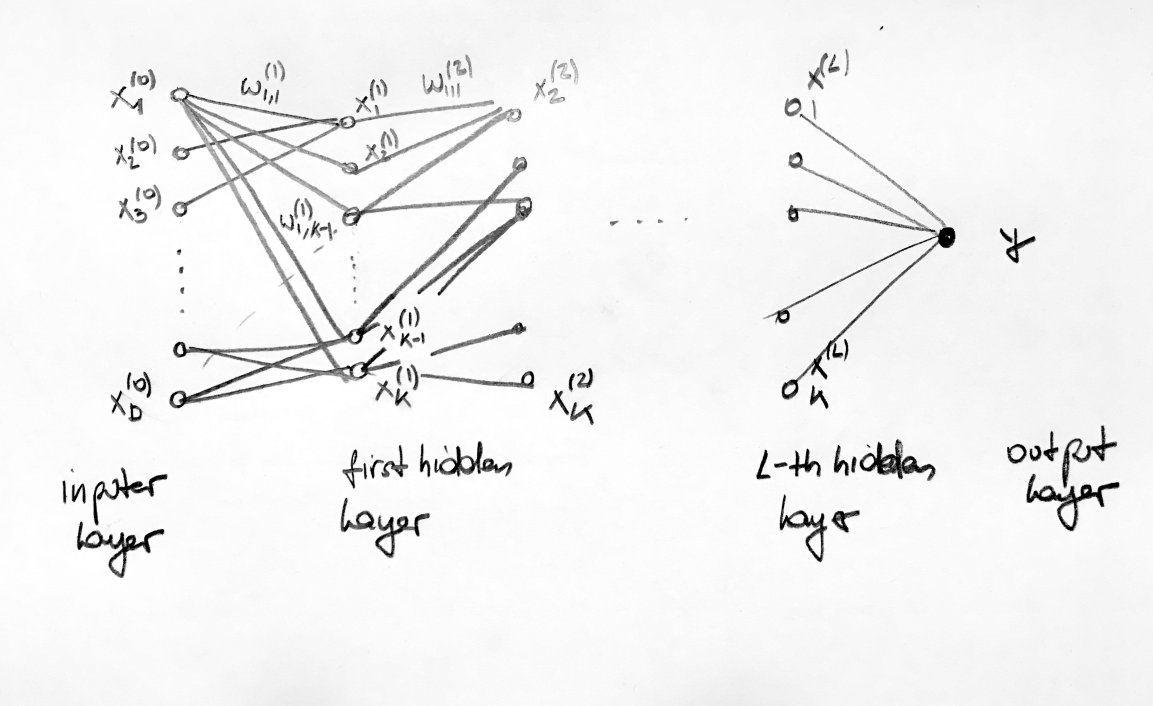


Figure 1: A neural network with one input layer, L hidden layers, and one output layer.

Motivation

Recall the structure of a neural network. For your convenience it is shown again in Figure 1. Assume that our task is regression. I.e., we have a training set $S_t = \{(\mathbf{x}_n, y_n)\}$. Let $f(\mathbf{x})$ be the function that is represented by the nn (including the last layer). I.e., $f(\mathbf{x})$ is the output of the nn.
Let us assume that we use our standard cost function

$$\mathcal{L} = \frac{1}{N} \sum_{n=1}^{N} (y_n - f(\mathbf{x}_n))^2.$$

We might want to add a regularization term to avoid overfitting, but this term is trivial to compute and to take into account. Therefore, we omit such a regularization term from our discussion in this lecture.

Given our training set S_t, our task is to train the network to minimize the cost function. *Training* here means choosing the parameters of the net, namely the *weights* of the edges and the *bias* terms in order to minimize the cost function. Our go-to technique for training models is stochastic gradient descent. We have seen how it works for simple linear regression models but also for the matrix factorization problem. It is also a natural candidate for training neural nets and the current state-of-the-art. Contrary to some other optimization problems we encountered this problem is not convex. In fact, we expect it to have many local minima. We therefore have no guarantee that gradient descent will get close to an optimal solution for the training set.

In addition, we have to worry about overfitting. Here the news is better: SGD is known to be *stable* when applied to a NN. Informally this means that the outcome of running SGD will not differ dramatically if we replace a single sample from our training set. More precisely, assume that we run two versions of SGD on the same net and with the same training set, except a single sample that differs in the two training sets.

Assume now that the net is initialized in the same state and that we make the same random choices in the two parallel versions. Then, as long as we do not run for too many rounds, (running a through the whole date a constant number over times is OK) the outcome of the two runs will differ only slightly. And, as a consequence, such a stable training algorithm is guaranteed not to overfit, i.e., the training error will be close to the true error.

As always when dealing with gradient descent we compute the gradient of the cost function for a particular input sample (with respect to all weights of the net and all bias terms) and then we take a small step in the direction opposite to this gradient.
As we will see, computing the derivative with respect a particular parameter amounts to applying the *chain rule* of calculus and is therefore familiar to all of us. So why discuss this matter?
Since in general there are many parameters it would not be efficient to do this computation for each parameter individually. We therefore will discuss how to compute all the derivatives *jointly* in an efficient manner. The algorithm for doing so is very natural and it is called *back propagation.*

Compact Description of Output

Let us start by writing down the output as a function of the input explicitly in compact form. It is natural and convenient to describe the function that is implemented by each layer of the network separately at first. The overall function is then the composition of these functions.
Let $\mathbf{W}^{(l)}$ denote the *weight* matrix that connects layer $l-1$ to layer l. The matrix $\mathbf{W}^{(1)}$ is of dimension $D \times K$, the matrices $\mathbf{W}^{(l)}$, $2 \leq l \leq L$, are of dimension $K \times K$, and the matrix $\mathbf{W}^{(L+1)}$ is of dimension $K \times 1$. The entries of each matrix are given by

$$\mathbf{W}^{(l)}_{i,j} = w^{(l)}_{i,j},$$

where we recall that $w_{i,j}^{(l)}$ is the weight on the edge that connects node i on layer $l-1$ to node j on layer l.
Further, let us introduce the *bias* vectors $\mathbf{b}^{(l)}$, $1 \leq l \leq L+1$, that collect all the bias terms. All these vectors are of length K, except the term $\mathbf{b}^{(L+1)}$, that is a scalar.
With this notation we can describe the function that is implemented by each layer in the form

$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}) = \phi((\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}), \qquad (1)$$

where the (generic) activation function is applied pointwise to the vector.
The overall function $y = f(\mathbf{x}^{(0)})$ can then be written in terms of these functions as the composition

$$f(\mathbf{x}^{(0)}) = f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}^{(0)}).$$

Cost Function

The cost function can be written as

$$\mathcal{L} = \frac{1}{N} \sum_{n=1}^{n} (y_n - f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n))^2.$$

Note that this cost function is a function of all weight matrices and bias vectors and that it is a composition of all the functions describing the transformation at each layer.
Note also that the specific form of the loss (squared loss, hinge loss, ...) does not really matter for the workings of the back propagation algorithm that we now discuss. Just to be specific we stick to the square loss. Only the initialization of the back recursion changes if we pick a different loss function.

The Backpropagation Algorithm

In SGD we compute the gradient of this function with respect to one single sample. Therefore, we start with the function

$$\mathcal{L}_n = (y_n - f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n))^2.$$

Recall that our aim is to compute

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}}, l = 1, \cdots, L+1,$$
$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}}, l = 1, \cdots, L+1.$$

It will be convenient to first compute two preliminary quantities. The desired derivatives are then easily expressed in terms of those quantities.
Let

$$\mathbf{z}^{(l)} = (\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}, \tag{2}$$

where $\mathbf{x}^{(0)} = \mathbf{x}_n$ and $\mathbf{x}^{(l)} = \phi(\mathbf{z}^{(l)})$, see (1). In words, $\mathbf{z}^{(l)}$ is the input at the l-th layer before applying the activation function. These quantities are easy to compute by a *forward* pass in the network. More precisely, start with $\mathbf{x}^{(0)} = \mathbf{x}_n$ and then apply this recursion for $l = 1, \cdots, L+1$, first always computing $\mathbf{z}^{(l)}$ via (2) and then computing $\mathbf{x}^{(l)} = \phi(\mathbf{z}^{(l)})$.
Further, let

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}.$$

Let $\delta^{(l)}$ be the corresponding vector at level l. Whereas the quantities $\mathbf{z}^{(l)}$ were easily computed by a forward pass, the quantities $\delta^{(l)}$ are easily computed by a *backwards* pass:

$$\begin{aligned}\delta_j^{(l)} &= \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} = \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \\ &= \sum_k \delta_k^{(l+1)} \mathbf{W}_{j,k}^{(l+1)} \phi'(z_j^{(l)}).\end{aligned}$$

In vector form we can write this as

$$\delta^{(l)} = (\mathbf{W}^{(l+1)} \delta^{(l+1)}) \odot \phi'(\mathbf{z}^{(l)}),$$

where $\odot$ denotes the Hadamard product (the pointwise multiplication of vectors).
Now that we have both $\mathbf{z}^{(l)}$ and $\delta^{(l)}$ let us get back to our initial goal. Note that

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} = \underbrace{\frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}}_{\delta_j^{(l)}} \underbrace{\frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}}}_{\mathbf{x}_i^{(l-1)}} = \delta_j^{(l)} \mathbf{x}_i^{(l-1)}.$$

Why could we drop the sum in the above expression? When we change the weight $w_{i,j}^{(l)}$ then it *only* changes the sum $z_j^{(l)}$. All other sums at level l stay unchanged.
In a similar manner,

$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_j^{(l)}} = \underbrace{\frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}}_{\delta_j^{(l)}} \underbrace{\frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}}_{1} = \delta_j^{(l)} \cdot 1 = \delta_j^{(l)}.$$

Summary of Backpropagation Algorithm for Computing the Derivatives

We are given a nn with L hidden layers. All weight matrices $\mathbf{W}^{(l)}$ and bias vectors $\mathbf{b}^{(l)}$, $l = 1, \cdots, L+1$, are fixed. We are given in addition a sample $(\mathbf{x}_n, y_n)$. We want to compute the derivatives

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}}, \quad \frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}}, l = 1, \cdots, L+1,$$

where

$$\mathcal{L}_n = (y_n - f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n))^2.$$

Forward Pass: Set $\mathbf{x}^{(0)} = \mathbf{x}_n$. Compute for $l = 1, \cdots, L+1$,

$$\mathbf{z}^{(l)} = (\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}, \quad \mathbf{x}^{(l)} = \phi(\mathbf{z}^{(l)}).$$

Backward Pass: Set $\delta^{(L+1)} = -2(y_n - \mathbf{x}^{(L+1)})\phi'(z^{(L+1)})$. If we are using a loss other than the squared loss, this initialization changes and this is the *only* change. Also note that the expression $\phi'(\cdot)$ refers to the derivative of the activation function used in the output layer. So if we are using a different activation function in the last layer (as we often do) use the appropriate derivative at this point. Compute for $l = L, \cdots 1$,

$$\delta^{(l)} = (\mathbf{W}^{(l+1)}\delta^{(l+1)}) \odot \phi'(\mathbf{z}^{(l)}).$$

Final Computation: For all parameters compute

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \delta_j^{(l)} \mathbf{x}_i^{(l-1)}, \frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \delta_j^{(l)}.$$

Now that we have the gradient with respect to all parameters, the SGD algorithm makes a small step in the direction opposite to the gradient, then picks a new sample $(\mathbf{x}_n, y_n)$, and repeats.

One final note. In our next lecture we will encounter networks (convolutional neural nets) where several edges *share* the same weight or bias. The advantage of doing so is that the net has fewer parameters and so might be easier to train with a given amount of data. How do we proceed then. We can still compute the gradient in such scenarios using the backpropagation algorithm: Write down the network pretending that all the parameters are independent. Run the backpropagation algorithm. The gradient for a particular parameter for the model where some weights are equal is now just the sum of the gradients (of the model where weights are independent) of all the edges that share the same weight.

Machine Learning Course - CS-433

Neural Nets – Some Popular Activation Functions

November 3, 2020

changes by Nicolas Flammarion 2020, changes by Rüdiger Urbanke 2019,2018,2017;

Last updated on: November 2, 2020

EPFL

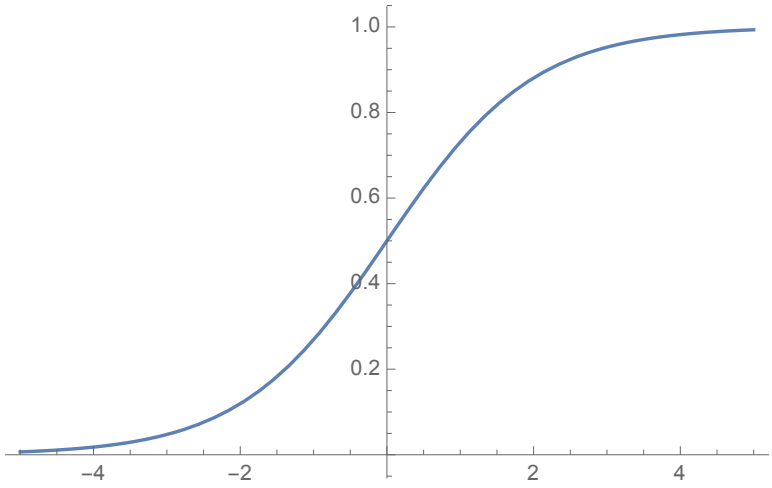


Figure 1: The sigmoid function $\phi(x)$.

Motivation

There are many activation functions that are being used in practice. Let us list here some of them and briefly discuss their merits.

Sigmoid

We start with the sigmoid $\phi(x)$, which we have encountered already several times. Just to summarize, it is defined by

$$\phi(x) = \frac{1}{1 + e^{-x}},$$

and a plot is shown in Figure 1. Note that the sigmoid is always positive (not really an issue) and that it is bounded. Further, for $|x|$ large, $\phi'(x) \sim 0$. This can cause the gradient to become very small (which is known as the "vanishing gradient problem"), sometimes making learning slow.

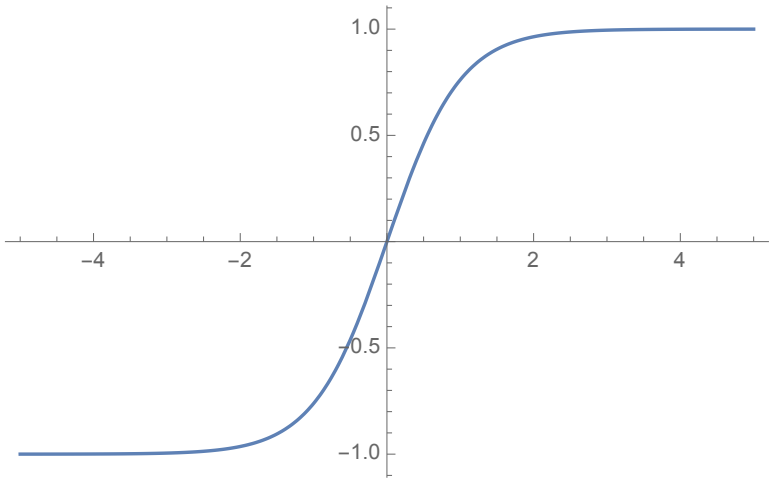


Figure 2: $\tanh(x)$.

Tanh

Very much related to the sigmoid is $\tanh(x)$. It is defined by

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\phi(2x) - 1,$$

and a plot is shown in Figure 2. Note that $\tanh(x)$ is "balanced" (positive and negative) and that it is bounded. But it has the same problem as the sigmoid function, namely for $|x|$ large, $\tanh'(x) \sim 0$. As mentioned, this can cause the gradient to become very small, sometimes making learning slow.

Rectified linear Unit – ReLU

Very popular is the rectified linear unit (ReLU) $(x)_+$ that we have also seen already. To recall, it is defined by

$$(x)_+ = \max\{0, x\},$$

and a plot is shown in Figure 3. Note that the ReLU is

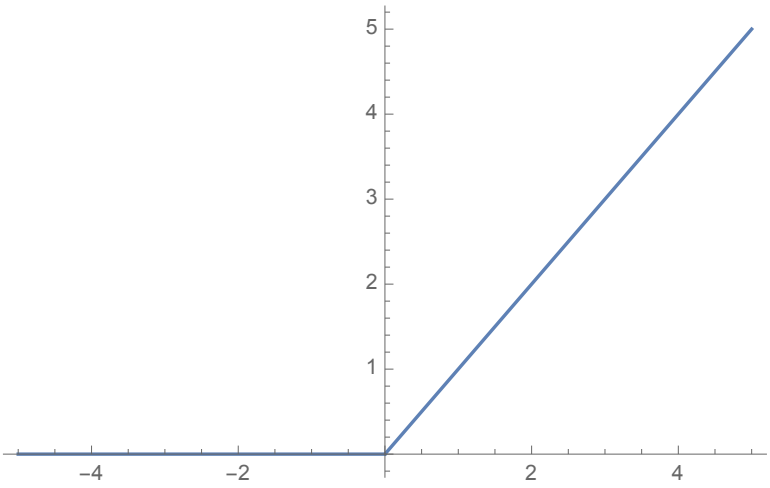


Figure 3: The ReLU $(x)_+$.

always positive and that it is unbounded. One nice property of the ReLU is that its derivative is 1 (and does not vanish) for positive values of x (it has 0 derivative for negative values of x though).

Leaky ReLU

In order to solve the 0-derivative problem of the ReLU (for negative values of x) one can add a very small slope α in the negative part. This gives rise to the leaky rectified linear unit (LReLU). It is defined by

$$f(x) = \max\{\alpha x, x\}$$

and a plot is shown in Figure 4. The constant α is of course a hyper-parameter that can be optimized.

Maxout

The maxout generalizes ReLU and LReLU. It is defined by

$$f(x) = \max\{\mathbf{x}^\top \mathbf{w}_1 + b_1, \cdots, \mathbf{x}^\top \mathbf{w}_k + b_k\}$$

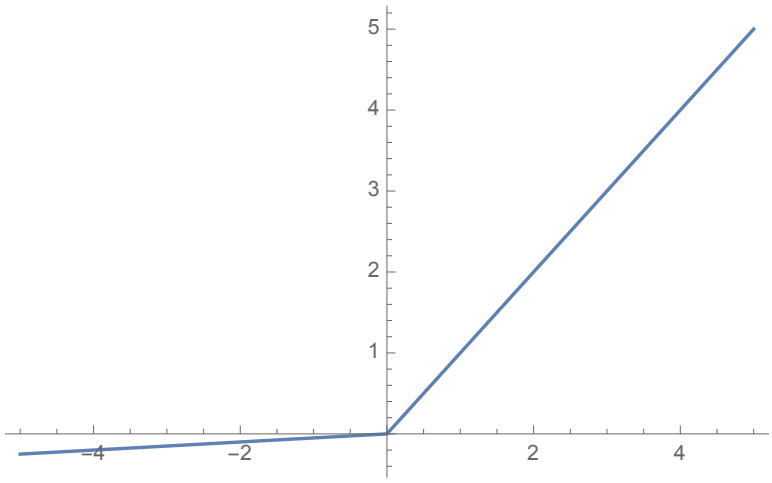


Figure 4: LReLU with $\alpha = 0.05$

and a plot is shown in Figure 5. The constants in this function are of course parameters that can be chosen for the particular application. Note that this activation function is quite different from the previous cases. In the previous cases we computed a weighted sum and then applied the activation function to it, whereas here we compute two or more different weighted sums and then choose the maximum.

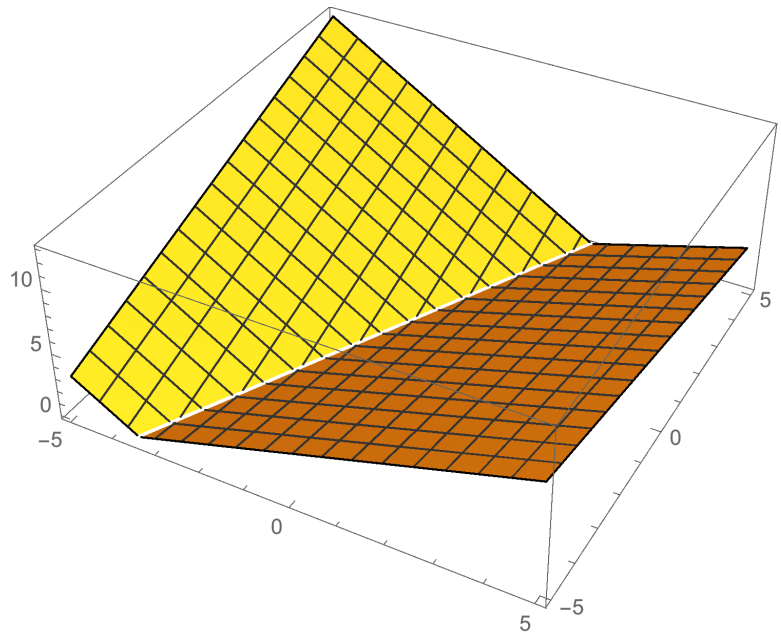


Figure 5: Maxout function with two terms, $\max\{x_1 - 0.5x_2 + 1, -2x_1 + x_2 - 2\}$.

Machine Learning Course - CS-433

Neural Nets – Convolutional Nets

Nov 10, 2020

changes by Nicolas Flammarion 2020, changes by Rüdiger Urbanke 2019,2018,2017;

Last updated on: November 9, 2020

EPFL

Basic Structure of Convolutional Nets

Recall that for standard neural network every node at level l is connected to every node at level $l-1$. The advantage of this structure is that it is very general and powerful. The disadvantage is that such a network has a large number of parameters and so generically lots of data is needed to train it.

In some scenarios it is intuitive that "local" processing of data should suffice. E.g., consider an audio stream given by samples $x^{(0)}[n]$. It is natural to process such a stream by running it through a linear time-invariant *filter*, whose output, call it $x^{(1)}[n]$, is given by the *convolution* of the input $x^{(0)}[n]$ and the filter $f[n]$,

$$x^{(1)}[n] = \sum_k f[k]\, x^{(0)}[n-k].$$

The filter $f[n]$ is often "local", i.e., $f[k] = 0$ for $|k| \geq K$. Much of signal processing is based on this type of operation. By choosing an appropriate type of filter we can bring out various aspects of the underlying signal. E.g., we can *smooth features* by averaging or we can *enhance differences* between neighboring elements by taking a so-called "high-pass" filter.

We have essentially the same scenario if we think of a picture. Now the signal $x^{(0)}[n,m]$ is two-dimensional and the convolution takes the form

$$x^{(1)}[n,m] = \sum_{k,l} f[k,l]\, x^{(0)}[n-k, m-l].$$

As before, we can take filters $f[n, m]$ that are "local" i.e., only have non-zero coefficients for small values of $|n|$ and $|m|$.

There are two important aspects about this structure. First, the output $x^{(1)}$ at position $[n, m]$ only depends on the value of the input $x^{(0)}$ at positions close to $[n, m]$. If we use this in a NN then we no longer get a fully connected network but the structure is much more sparse and local. This implies that we have significantly fewer parameters to deal with.

Second, this structure implies that we should use the *same* filter (e.g., not only the same connection-pattern but also the same weights) at every position! This is called *weight sharing*. Weight sharing drastically reduces the number of parameters further.

Figure 1 shows two layers of a very small NN where we see the difference between a fully connected network and one where connections are sparse and local.

Why is it meaningful to use the same filter at every position in the network? Consider e.g. a photo. Photos are inherently "shift invariant." We can imagine that there is an essentially infinite-sized 2D photo in reality and we are seeing a small portion of it. The portion we are seeing is more or less random and so the exact position of any object in this photo is more or less random as well. It therefore makes sense that we treat each position essentially equally.

Layout

It is common to lay out the data in a convolutional network according to its "natural" layout. E.g., if the input is a photo

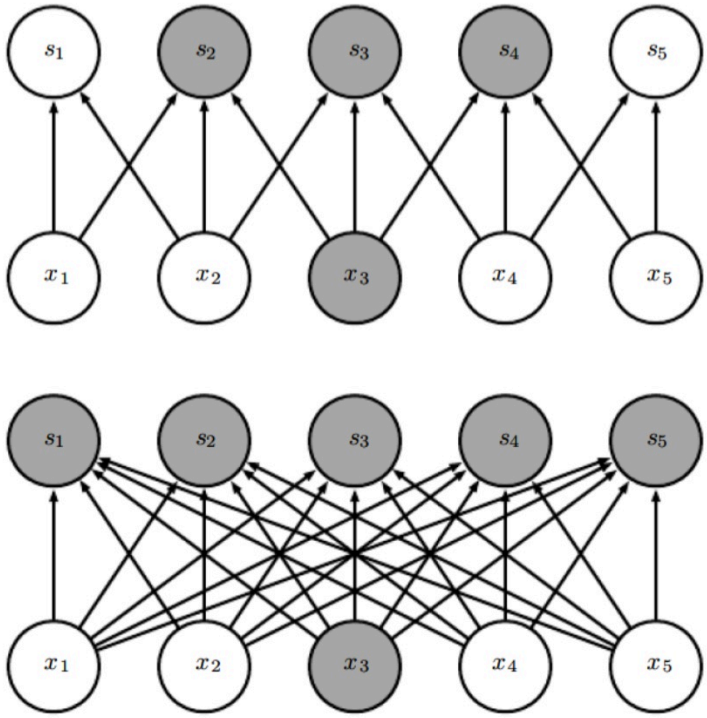


Figure 1: Sparse connections (upper illustration) versus fully connected (lower illustration). In the sparse case a node on the bottom layer only influences a limited number of nodes on the top layer. (Figure 9.2 from http://www.deeplearningbook.org/)

then it is natural to use a 2D layout, whereas if the data is naturally one-dimensional then it makes sense to use a 1D layout.

Handling of Borders

Consider a 2D case. Assume that we have an input of dimension $N_1 \times N_2$ and a kernel $f[l, m]$ so that $|f[m, l]| = 0$ if $|l| \geq K_1$ or $|m| \geq K_2$. There are several natural ways of dealing with the boundary.

The first is to *pad* the original input data with zeros at the boundary. More precisely, instead of considering the input

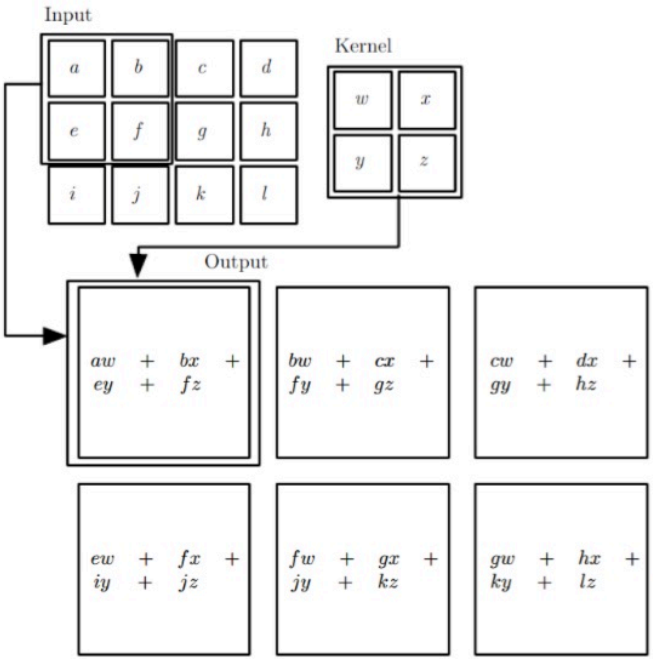


Figure 2: Handling the boundary using the *valid packing* method. (Figure 9.1 from http://www.deeplearningbook.org/)

of size $N_1 \times N_2$, consider the input to be of size $(N_1+2(K_1-1)) \times (N_2+2(K_2-1))$ where the "center piece" is the original data and there is an additional boundary of width $K_1 - 1$ and $K_2 - 1$ respectively which is set to 0. If we now perform the convolution then the output will again be of size $N_1 \times N_2$. For obvious reasons this is called *zero padding*.

The second option is to compute an output which is only of size $(N_1 - 2(K_1 - 1)) \times (N_2 - 2(K_2 - 1))$, i.e., to perform the convolution only for positions so that the whole filter lies inside the original data. This is called *valid padding*. Figure 2 shows this second method.

Multiple Channels

Assume we start with picture, i.e., a 2D structure. It is common to not only compute the output of a single filter but to use multiple filters. The various outputs are called *channels*. This introduces some additional parameters into the model.
If we add several channels we do not end up with a 2D output in the next level but in fact with a 3D output. We proceed in the same fashion in each further stage, computing several channels per input layer in the previous stage. In this manner, we will get more and more channels as we get deeper and deeper into the network. On the other hand the "size" of each picture typically gets smaller and smaller as we proceed through the layers, either due to the handling of the boundary or because we might perform subsampling. The whole structure then looks a little bit like a pyramid. It gets thinner towards the top but the sections become longer and longer (more and more channels). This is shown in Figure 3.

Training

As we discussed, there are two aspects that make CNN special. First, only some of the edges are present. This means that the weight matrices are sparse and this does not require any change in the learning steps when we use SGD and backpropagation.
Second, weight sharing is used, i.e., many edges use weights that are the same. As we already mentioned in the previous

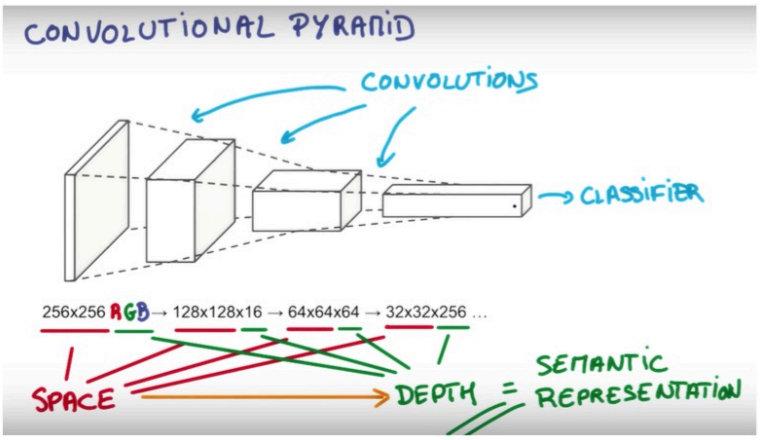


Figure 3: Structure of a typical CNN. Per layer we have increasingly more channels but a smaller "footprint." (from www.udacity.com/course/deep-learning–ud730)

lecture, with a small modification the back propagation algorithm can still be used to train a CNN with weight sharing: run backpropagation ignoring that some weights are shared, considering each weight on each edge to be an independent variable. Once the gradient has been computed for this network with independent weights, sum up the gradients of all edges that share the same weight. This gives us the gradient for the network with weight sharing.

To see that this is the correct thing to do, consider a simple example. Let $f(x, y, z)$ be a function from $\mathbb{R}^3 \to \mathbb{R}$. Let $g(x, y) = f(x, y, x)$. In words, z is no longer an independent variable but we set $z = x$. If we now want to compute the gradient

$$\left(\frac{\partial g(x,y)}{\partial x}, \frac{\partial g(x,y)}{\partial y}\right)$$

then we can compute this by first computing

$$\Big(\frac{\partial f(x,y,z)}{\partial x}, \frac{\partial f(x,y,z)}{\partial y}, \frac{\partial f(x,y,z)}{\partial z}\Big)$$

and then realizing that

$$\Big(\frac{\partial g(x,y)}{\partial x}, \frac{\partial g(x,y)}{\partial y}\Big) = \Big(\frac{\partial f(x,y,z)}{\partial x} + \frac{\partial f(x,y,z)}{\partial z}, \frac{\partial f(x,y,z)}{\partial y}\Big).$$

FFT

The convolutional structure can be used in order to compute the output of a CNN very efficiently. Whether this is more efficient than computing the output by directly implementing the sum depends on the size of the filter/kernel.
To keep things simple, assume that our data layout is one-dimensional and that each layer is connected via a convolution where we use zero-padding at the boundaries. Let us consider the first layer to be specific. We then have

$$x^{(1)}[n] = \sum_k f[k]\, x^{(0)}[n-k].$$

Assume that $|f[k]| = 0$ for $k \notin \{0, \cdots K-1\}$. I.e., the filter has at most K non-zero coefficients. And assume that the signal $x^{(0)}[n]$ has length N. Zero-pad both the signal and the filter to make them of length L, where $L \geq N+K-1$. More precisely, for the filter the first K positions are the non-zero positions of the original filter and the remaining $L-K$ are zeros. For the signal the first N positions are the non-zero coefficients of the signal and the remaining $L-N$ are zeros.

Call the resulting quantities $\tilde{f}$ and $\tilde{x}^{(0)}$ respectively. Compute the *cyclic* convolution of these two signals, i.e., compute

$$\tilde{x}^{(1)}[n] = \sum_{k=0}^{L-1} f[k]\, x^{(0)}[n - k \bmod L].$$

This convolution is just like the regular one except that we compute indices modulo L. A quick check shows that $x^{(1)}[n]$ (where we used zero-padding to deal with the boundary) and $\tilde{x}^{(1)}[n]$ are identical for the first $N - K + 1$ positions.
But this cyclic convolution can be computed by transforming both the signal as well as the filter into the Fourier domain, multiplying the two, and then converting the result back. This in turn can be accomplished efficiently by means of the Fast Fourier Transform (FFT) in $cL \log_2(L)$ operations, where c is a small constant.
More precisely, for a signal $x[n]$ of length L its discrete Fourier Transform (DFT) (also of length L) and its inverse are given by

$$\hat{x}[k] = \sum_{n=0}^{L-1} x[n] e^{-\frac{2\pi j}{L} kn},$$

$$x[n] = \frac{1}{L} \sum_{k=0}^{L-1} \hat{x}[k] e^{\frac{2\pi j}{L} kn}.$$

Machine Learning Course - CS-433

Neural Nets – Regularization, Data Augmentation, and Dropout

Nov 10, 2020

changes by Nicolas Flammarion 2020, changes by Rüdiger Urbanke 2019,2018,2017;

Last updated on: November 9, 2020

EPFL

Regularization

We have seen that for standard regression/classification tasks it is common to add a regularization term when learning. The same is true for neural networks. E.g., we might want to add a term of the form

$$\frac{1}{2}\sum_{l=1}^{L+1}\mu^{(l)}\|\mathbf{W}^{(l)}\|_F^2,$$

where $\mu^{(l)}$ is a non-negative constant that can depend on the layer. Note that it is common *not to penalize the bias terms* but only the weights.

Such a regularization term favors small weights and, combined with the right constants $\mu^{(l)}$, can avoid overfitting.

How does the gradient descent algorithm change if we use this form of regularization?[1] Assume that we use the same constant in all layers, i.e., $\mu^{(l)} = \mu$. Let $\Theta = w_{i,j}^{(l)}$ be the weight of the edge going from node i at layer $l-1$ to node j at layer l and let t be discrete time, increasing by one in each update step. We then get the update rule

$$\underbrace{\Theta[t+1]}_{\text{new value}} = \underbrace{\Theta[t]}_{\text{old value}} - \underbrace{\eta}_{\text{step size}} \Big(\underbrace{\nabla_\Theta \mathcal{L} + \mu\Theta[t]}_{\text{grad. data/regularization}} \Big)$$
$$= \underbrace{(1-\eta\mu)\Theta[t]}_{\text{weight decay}} - \eta\nabla_\Theta\mathcal{L}.$$

We see that in one update step the weight is decreased by a factor $1-\eta\mu$ and in addition we add a small step in the

[1] This discussion is not restricted to neural nets but applies generally. We just happen to discuss this topic in the context of neural nets.

negative direction of the gradient. We say that regularization leads to *weight decay*.

Another popular method (Hinton et al, 2013) is not to put a penalty on the square of the L_2 norm of the weights, but rather ask that the weight vector must have an L_2 norm no more than a constant, call it r. This can be easily incorporated into the gradient descent algorithm as follows. After each gradient step check whether the L_2 norm of the weight vector is above r or not. If it is, rescale the whole weight vector so that it has L_2 norm equal to r. This rescaling operation is trivial for the L_2 norm. The resulting algorithm is called the *projected* gradient descent algorithm (since we project after the gradient step the weight vector back to the sphere of radius r if necessary).

As a side remark – if we had added a regularization term which is the L_1 norm of the weight vector then this projection is not nearly as easy. But fortunately for neural nets the L_2 norm is the most useful regularization.

Dataset Augmentation

Data is scarce and valuable and the more data we have the better we can train. In some instances we can generate new data from the data we are given.

Consider a classification task with training set $S_t = \{(\mathbf{x}_n, y_n)\}$. Assume that there exists a transformation $\tau : \mathbb{R}^D \to \mathbb{R}^D$ that keeps the labels unchanged.

E.g., consider a hand-writing recognition task. We are given small squares (see Figure 1) each containing a digit from 0 to 9. The absolute position of the digit inside the square and

the exact orientation of the digit do not matter and can be changed without changing the label.
We can therefore take the data we are given, create variants of it, and add it to the date. Figure 1 shows some characters from the MNIST data set. Figure 2 shows some rotated variants and Figure 3 shows some shifted variants. This way

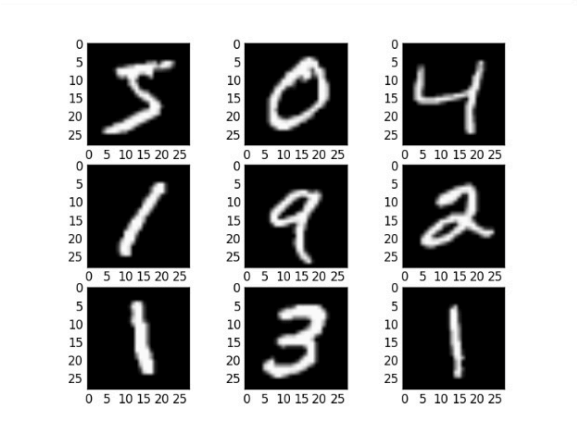


Figure 1: Some characters from the MNIST data set.

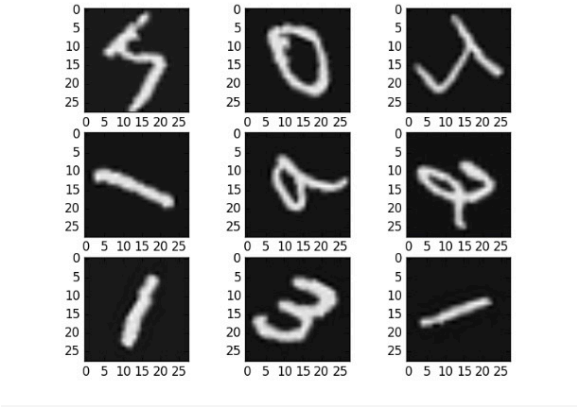


Figure 2: Rotated characters of the MNIST data set.

we can significantly increase our data which helps in training. In addition, if we train our network on this augmented data set then the network will automatically learn to become invariant to these transformations.

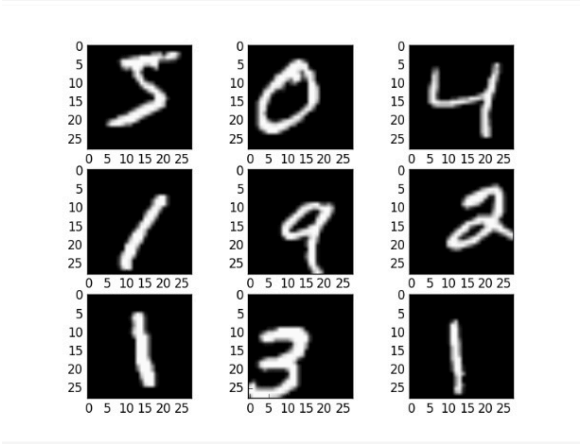


Figure 3: Shifted characters from the MNIST data set.

These transformations, if they exist, are very task specific. If we consider image recognition task some other possible transformations are *cropping* or *resizing*. More subtle, we can use the PCA and "compress" the image by only keeping the components corresponding to the largest singular values. This changes the photo globally but introduces only a minimal distortion (in the L_2 sense). In a similar sense, we can add some small amount of noise to our data.
There is another way in which we can "augment" our data. Assume that we have several distinct but related tasks. In this case we can train a network jointly whose "core" is used jointly for all tasks and where only the last layer is task specific. This is shown in Figure 4. The idea here is that for related tasks the same features are useful for the task.

Dropout

Dropout is a method both to avoid overfitting as well as to do model averaging (Hinton et al, 2012). By now, many variants have been proposed. Here is the original version.

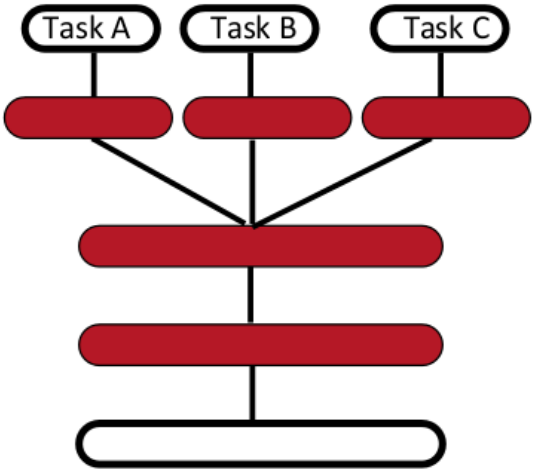


Figure 4: Neural network structure for multi-task learning.

Define the probability $p_i^{(l)}$. It is the probability of "keeping" the node i in layer l. Typical values are $p_i^{(0)} = 0.8$ (i.e., we keep in expectation 80 percent of all input nodes) and $p_i^{(l)} = 0.5$ for $l \geq 1$, (i.e., we keep in expectation 50 precent of all hidden nodes).
At every training step decide for each node i at level l according to the probability $p_i^{(l)}$ whether to keep this node or not. This defines a "subnetwork." Run one step of SGD (or perhaps a minibatch) and update the weights. Iterate until training is done.
For the prediction phase several variants are possible. Either generate K subnets in the same manner as before, predict for each and average the prediction. Alternatively use the whole network for the prediction. But in this case scale the output of node i at level l by the factor $p_i^{(l)}$. This guarantees that the expected input at each node stays the same as the expected input during training.
There are two benefits to this "dropout" procedure. First,

it has been observed that this procedure limits overfitting. Intuitively, nodes cannot "rely" on other nodes being present. Second, note that there is an exponential number of "sub-networks." This is shown in Figure 5 for a very small toy network. The effect of dropout is that we are performing an average over several (sub)networks. We either do this explicitly by running over several of them, computing the output, and then average. Or we do this implicitly by using the whole network but with reduced weights. We therefore get the advantage that comes with model averaging.

Averaging over many models is a standard ML trick and it is called *bagging*. It typically leads to improved performance. But dropout is quite different from standard bagging since we do not train K networks and then average. Rather, all these networks share the same weights. In fact, this characteristic seems to be an important component to explain their good performance.

In dropout we remove whole nodes. It is of course also possible to remove individual edges independently from each other.

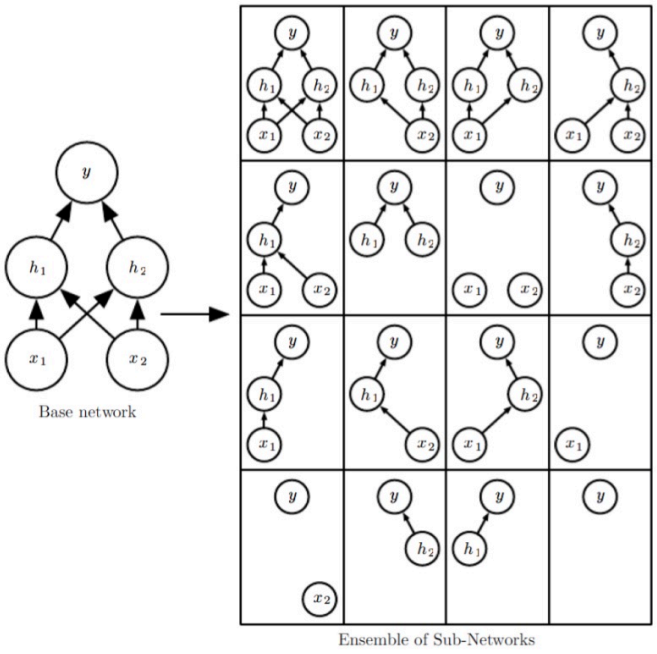


Figure 5: All the subnetworks of a given network.

Machine Learning Course - CS-433

Adversarial ML

Nov 10, 2020

Small changes by Nicolas Flammarion, small changes by Martin Jaggi 2019; ©Rüdiger Urbanke 2019

Last updated on: November 9, 2020

Introduction

Some ML tasks are inherently difficult. E.g., consider the examples in Figure 1. Even humans might not be able to classify all examples correctly. So we would not be surprised to see NNs struggle in such cases. But typically this is where they shine, having a performance on par or perhaps even surpassing humans. But to date, NNs (or other classifiers for that matter) are not as *robust* in their decisions as humans and can be easily tricked by adversarially chosen perturbations of the input, even if the perturbations are small. This can lead to problems.

Figure 1: Dog or mop?

In the sequel it might be good to have a concrete example in mind. E.g., think of self-driving cars. And we will assume that we are using NNs. But the basic principle applies to a much wider setting.

So consider a self-driving car. Such a device hopefully will be able to recognize and correctly interpret street signs with very high probability. If the ML algorithm that performs

this task can be easily tricked by slightly manipulating the input then insurance companies might not be happy![1]

The Basic Attack using Back Propagation

So let us look at the simplest instance. Assume that we have trained a binary classifier $f : \mathcal{X} \to \{\pm 1\}$ given some training set $\mathcal{S} = \{(\mathbf{x}_n, y_n)\}_{n=1}^N$. There is an underlying data distribution $\mathcal{D}$ that is unknown to us. Assume that we have trained the network well and that it has a small true risk, i.e.,

$$\mathcal{L}(f) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\mathbb{1}_{\{f(\mathbf{x})\neq y\}}] \leq \delta,$$

where $\delta > 0$ is a small number. We are happy.
But assume now that an adversary were allowed to manipulate, i.e., change, the input $\mathbf{x}$ slightly. How much worse can the risk be made? If we put no restriction on the "power" of the adversary then clearly she can increase the risk at will. So let us say that the adversary can change the given input $\mathbf{x}$ to $\tilde{\mathbf{x}}$, but that we require that $\|\mathbf{x} - \tilde{\mathbf{x}}\| \leq \varepsilon$ for some small $\varepsilon > 0$. What norm shall we pick? This depends on the

[1] Just to clear up any confusion. There are also GANs – which confusingly stands for generative adversarial networks. Even though there is also the name "adversarial" in this topic, this has nothing to do with the current set-up. In GANs the adversarial view-point helps to train a network. E.g., assume that the task of the network is to to create realistically looking faces given "random-like" input. Here the network is used in a "generative" way. It has at it's disposal a set of human faces. In order to train this network to perform its task better we use a second network that tries to distinguish between faces generated by the network and real faces. This is the adversary. Training now proceeds in two phases that are interlaced. Given a generative network we train a powerful adversary to perform the distinction and given an adversary we train a hopeful better generative network.

application and it is part of what is called the *threat model*. We will get back to this point later. It is customary in the literature to pick either ℓ_1, ℓ_2, or ℓ_∞. We can now define the adversarial risk, call it $\mathcal{R}(f, \varepsilon)$, as

$$\mathcal{R}(f, \varepsilon) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}\Big[\max_{\tilde{\mathbf{x}}:\|\mathbf{x}-\tilde{\mathbf{x}}\|\leq\varepsilon} \mathbb{1}_{\{f(\tilde{\mathbf{x}})\neq y\}}\Big].$$

In words, for every input $\mathbf{x}$ the adversary can find the worst perturbation allowed by the norm constraint.
Depending on whether you are interested in "breaking" the classifier or try to make it robust we are faced with numerous questions. Here are some in no particular order.

1. How do we find adversarial perturbations efficiently?
2. In order to find those perturbations, what access to the classifier do we need?
3. By how much worse can we make the risk?
4. Are there measures we can take to make a given classifier more robust?
5. Are there particular ways to train the classifier to make it robust?

We will explore some of these questions in the following sections.

White Box Attacks

So assume that we are given a binary classifier f. To be concrete let us assume that it is implemented by a NN. We

have complete access to the NN and are given an input $\mathbf{x}$. How do we find an adversarial perturbation $\tilde{\mathbf{x}}$ so that $\|\mathbf{x} - \tilde{\mathbf{x}}\| \leq \varepsilon$? You will explore this in the exercise session. Here is a the basic idea.

If f does not classify $\mathbf{x}$ correctly then we are done – we can just set $\tilde{\mathbf{x}} = \mathbf{x}$. So we might as well assume that it does indeed classify $\mathbf{x}$ correctly. To be specific, assume that the data is separable in principle, i.e., that there exists a "ground truth" encoded by the function $h : \mathcal{X} \to \{\pm 1\}$. In other words, the correct label is non-probabilistic and is given by $h(\mathbf{x})$. Further assume that $f(\mathbf{x})$ has the form

$$f(\mathbf{x}) = \begin{cases} 1, & 0 \leq \frac{1}{2} \leq g(\mathbf{x}) \leq 1, \\ -1, & 0 \leq g(\mathbf{x}) < \frac{1}{2}, \end{cases} \tag{1}$$

where $g : \mathcal{X} \to [0, 1]$ represents the probability that $y = 1$ given the input. I.e., we assume that, like in logistic regression, we first compute a probability and then we quantize.

Of course $g(\mathbf{x})$ depends on all the weights and bias terms within the NN but we omit this dependence in our notation since in the present context we are interested in variations due to changes in the input rather than changes in the parameters.

Why do we assume that f has this form? We will use gradient descent in order to find adversarial perturbations. For this to work we want the objective function to be smooth. We will therefore use g rather than the original f.

Using the backpropagation algorithm, we can efficiently compute $\nabla_{\mathbf{x}} g(\mathbf{x})$. Note that this is the gradient with respect to changes in the input rather than changes in the parameters.

Hence

$$h(\mathbf{x})\nabla_{\mathbf{x}} g(\mathbf{x})$$

is a vector of length D (the dimension of the input space) which is positive in positions where an increase in that input makes the prediction more correct (increases the probability of the correct label) and negative where an increase in that dimension makes the prediction less correct.

Assume that we are allowed to move the point $\mathbf{x}$ to a new point $\tilde{\mathbf{x}}$, with the restriction that $\|\tilde{\mathbf{x}} - \mathbf{x}\|_2 \leq \varepsilon$. Note that we have assumed an ℓ_2 constraint. Our aim is to make the prediction as bad as possible. This means that we would like to decrease the probability of the correct label as much as possible.

If ε is small then it makes sense to assume that the change in the probability is given by the first order change, i.e., that it is well predicted by the gradient. Given our "budget" in terms of ℓ_2 the optimum move is then to move in the opposite direction of the gradient, i.e., to define[2]

$$\tilde{\mathbf{x}} = \mathbf{x} - \varepsilon h(\mathbf{x}) \frac{\nabla_{\mathbf{x}} g(\mathbf{x})}{\|\nabla_{\mathbf{x}} g(\mathbf{x})\|_2}.$$

If, for a particular $\mathbf{x}$, we manage to "flip" the probability using a move that is bounded by ε then we have found an adversarial example. In general, we might not want to take a single step but rather only move partially in this direction and then iterate this process.

[2]Mathematically speaking this corresponds to the following. We are given a fixed vector $\mathbf{w}$ of unit norm (in the ℓ_2 sense). Then the inner product $\mathbf{w}^\top \mathbf{v}$ conditioned on $\|\mathbf{w}\|_2 \leq \varepsilon$ is maximized by chosing $\mathbf{v} = \varepsilon \mathbf{w}$. This can be seen e.g. by the Cauchy Schwartz inequality $\mathbf{w}^\top \mathbf{v} \leq \|\mathbf{w}\|_2 \|\mathbf{v}\|_2$.

The above attack is known as a "white box" attack – we have access to the details of the algorithm.
Here is a good problem to think about. Assume that we are given as above the gradient $\nabla_{\mathbf{x}} g(\mathbf{x})$. What is the locally optimal move if our constraint is $\|\tilde{\mathbf{x}}-\mathbf{x}\|_1 \leq \varepsilon$ or $\|\tilde{\mathbf{x}}-\mathbf{x}\|_\infty \leq \varepsilon$ instead of the ℓ_2 constraint we used above?

Black Box Attacks

In the above attack we needed access to the NN that implemented the prediction in order to compute the gradients. In general it is a good idea in anything involving security to assume that the adversary has this level of access. Assume e.g., autonomous cars. Typically those are sold in large quantities and it will not be difficult for an adversary to get a hold of one of those boxes.
But even if we limit the adversary it turns out that similar attacks can still be carried out. Those are typically known as "black box" attacks. In such attacks we assume that we can observe the input out relationship but we cannot look inside the box.
Early on in the development of adversarial machine learning it was assumed that adversarial attacks could be prevented by preventing access to the algorithm or by obfuscating or masking the gradients (e.g., by adding some small amount of noise to the input before passing it through the network or by adding noise to the output). But it has been shown that all such approaches can be easily broken and that black box access is enough.
In the simplest case assume the previous setting and assume

further that we have access to the "unquantized" output $g(\mathbf{x})$. Assume that we want to compute the derivative with respect to the i-th input component. We can do this numerically by asking for the input output pairs for both $\mathbf{x}$ and $\tilde{\mathbf{x}}$ which is equal to $\mathbf{x}$, except in the i-th component, where it is slightly perturbed.
But what can we do if we only see the quantized output, i.e., the decision. In this case we cannot compute the derivative. An ingenious approach for this case is the following. Given black box access to f create a new set of samples $\mathcal{S} = \{(\mathbf{x}, f(\mathbf{x}))\}$ by computing many input-output pairs. Given $\mathcal{S}$ train a new NN. This new NN does not have to have the same structure (depth, width, activation functions etc) as the original one. Now run a white box attack on the newly-trained NN. Those adversarial direction that are found in this way often also cause trouble for the original network. If we assume that we have a sufficiently large number of examples so that we can learn f perfectly then this is perhaps not so surprising. But this approach seems to work also in cases where the number of samples we have at our disposal is quite reasonable.

Adversarial Attacks on Physical Objects

So far in our discussion we have assumed that we are given an input $\mathbf{x}$ and that we are finding an adversarial perturbation. But there exist much more interesting and potentially more dangerous variants. Let us go back to the example of a self-driving car. The car presumably has a camera (or multiple cameras). Assume that the car approaches an in-

tersection and sees a traffic sign. Perhaps this traffic sign is a stop sign. How can the adversary perturb the input. Perhaps the adversary can "hack" the car itself. But a much simpler attack is to perturb the actual physical stop sign. Is it possible to change the stop sign slightly so that a human will still recognize it as a stop sign but that the car will likely mistake it for a different sign? Note that this is quite more complicated than our original set-up. We are not given a particular input $\mathbf{x}$ but a whole family of such inputs – this family comes about since the care might approach the same stop sign from various slightly different angles, distances, and under slightly different ambient lighting conditions. And we would like the same attack (the same physical manipulation of this stop sign) to work under a broad set of those conditions. Even in this cases it has been shown that adversarial changes can be found! This is quite non-trivial! Figure 2 shows what a perturbed stop sign might look like. Note that for a human the change is visible but it is unlikely that it will cause confusion.

Why Some ML Algorithms Might Not Be Robust – Non-Robust Features

The following example illustrates one reason why a ML algorithm might learn a classification rule that has low standard risk but high adversarial risk.
This is a toy example, but the basic idea is sound the ML algorithm might rely on a large set of "non-robust" features that can be easily tricked by perturbations.

Figure 2: To stop or not to stop.

Consider a binary classification task. We have $y \in \{\pm 1\}$. Let $\hat{\mathbf{x}}$ be the input vector. Assume that after applying a suitable transform we get the new input vector $\mathbf{x}$ that has the following simple structure.

We have $\mathbf{x} = (x_1, \cdots, x_D)$, where $x_i = a_i y + Z_i$, $i = 1, \cdots, D$, where Z_i is Gaussian zero-mean and unit-variance noise which is independent for each component. Further, $a_1 = 1$ and for $i = 2, \cdots, D$, we have $a_i = \sqrt{\frac{\log(D)}{D-1}}$. The exact values for a_i are not so important and are chosen simply for convenience. What is important is that the first component contains a strong signal component, whereas the other features have a very weak such component. Strong versus weak is with respect to the strength of the noise that is added (zero-mean Gaussian of unit variance). We will say that the first feature is *robust* whereas the other features are not robust.

To summarize, each of the D components is a scaled and

noisy version of the label and all components represent conditionally independent observations. Further, the first component contains a strong signal component. The remaining $D-1$ features contain extremely weak signal components but there are many of them (we assume that D is large). Finally, assume that we are given the prior on the label $p(y)$ and that it is uniform.

Assume at first that we are interested in the best classifier without adversarial perturbations, i.e., the classifier with the smallest possible risk (error probability). This is the Bayes classifier, i.e., we should compute the posterior probability and then choose that label that maximizes the posterior:

$$\begin{aligned}\text{argmax}_{\hat{y}\in\{\pm 1\}} p(y \mid \mathbf{x}) &= \text{argmax}_{\hat{y}\in\{\pm 1\}} \frac{p(\mathbf{x} \mid y)p(y)}{p(\mathbf{x})} \\ &= \text{argmax}_{\hat{y}\in\{\pm 1\}} \prod_{i=1}^{d} p(\mathbf{x}_i \mid y).\end{aligned}$$

In the last step we have used the fact that under our model the observations are conditionally independent so that we get a product of the probabilities and that we have a uniform

prior. This can further be simplified to

$$\begin{aligned}
\text{argmax}_{y\in\{\pm 1\}} p(y \mid \mathbf{x}) &= \text{argmax}_{\hat{y}\in\{\pm 1\}} \prod_{i=1}^{d} p(\mathbf{x}_i \mid \hat{y}) \\
&= \text{argmax}_{\hat{y}\in\{\pm 1\}} \log \prod_{i=1}^{d} p(\mathbf{x}_i \mid \hat{y}) \\
&= \text{argmax}_{\hat{y}\in\{\pm 1\}} \sum_{i=1}^{d} \log p(\mathbf{x}_i \mid \hat{y}) \\
&= \text{argmax}_{\hat{y}\in\{\pm 1\}} \sum_{i=1}^{d} \log \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(\mathbf{x}_i - \hat{y}a_i)^2} \\
&= \text{argmin}_{\hat{y}\in\{\pm 1\}} \sum_{i=1}^{d} (\mathbf{x}_i - \hat{y}a_i)^2 \\
&= \text{argmin}_{\hat{y}\in\{\pm 1\}} \sum_{i=1}^{d} (\mathbf{x}_i^2 - 2\mathbf{x}_i\hat{y}a_i + \hat{y}^2 a_i^2) \\
&= \text{argmax}_{\hat{y}\in\{\pm 1\}} \hat{y} \sum_{i=1}^{d} \mathbf{x}_i a_i.
\end{aligned}$$

Recall that we observe $\mathbf{x}$ which is the vector $y\big(a_1 = 1, a_2 = \sqrt{\frac{\log(D)}{D-1}}, \cdots, a_D = \sqrt{\frac{\log(D)}{D-1}}\big)$ under Gaussian noise with iid zero-mean components and unit variance in each dimension. Therefore the expression that we maximize over $\hat{y}$ above is equal to

$$\hat{y}y(\sum_{i=1}^{D} a_i^2) + \hat{y}\sum_{i=1}^{D} a_i Z_i = \hat{y}y(1 + \log(D)) + \hat{y}Z,$$

where Z is a Gaussian noise with variance $(\sum_{i=1}^{D} a_i^2) = 1 + \log(D)$. Scaling everything by $1/(1 + \log(D))$, we see that this is equivalent to observing the signal $y = \pm 1$ under a zero-mean Gaussian noise with variance $1/(1 + \log(D))$. Hence as D grows the variance tends to zero and our error probability will go to zero as well. In other words, if we train our ML algorithm well then we can hope to get close to zero standard risk when the dimension D grows.

But assume now that we allow the adversary to move the point $\mathbf{x}$ into $\tilde{\mathbf{x}}$ and that our norm is ℓ_∞ with $\varepsilon = 2\sqrt{\frac{\log D}{D-1}}$. In this case the adversary can do the following. She can first make an optimal decision based on the observation. As we have seen, with high probability she will know the correct label. She can then can move each of the non-robust features $i = 2, \cdots, D$ into the wrong direction by an amount $2\sqrt{\frac{\log D}{D-1}}$. This means that she can in effect flip the non-robust features. She can also move the first component somewhat but this has almost no effect. If we ignore the effect on the first component we see that with this change the classifier will misclassify every single sample with high probability!

In summary, we have seen an example where the optimal standard risk is essentially zero but the adversarial risk of the same classifier is almost 1. This is as bad as it gets. And we have seen that this is due to the fact that the classifier relied heavily on many very weak features that were easy to perturb.

Could we have constructed a more robust classifier? Yes, certainly, but at a price. Assume that we build a classifier based on the first feature only. The best classifier in this case

is again a Bayes classifier. A little bit of thought shows that it is given by simply taking the sign of $\mathbf{x}_1$. What is the risk of this classifier? We have a label that is either $+1$ or -1. We add a zero-mean unit-variance Gaussian random variable to it and ask what is the probability that this noise changes the sign. A little bit of thought shows that the incurred error probability in this case is equal to

$$\frac{1}{\sqrt{2\pi}} \int_1^\infty e^{-\frac{1}{2}x^2} dx \sim 0.16.$$

In words, this classifier incurs a standard risk of about sixteen percent. This is much higher than the standard risk of essentially zero we saw above. But in this case the risk almost does not change if we consider the adversarial setting since in our threat model we allow the adversary to move each component by only a very small amount. So we see in this model a trade-off between a small standard risk and a small adversarial risk. We cannot get both.

The above example might look a little construed and also it might not be clear how much of this is due to the fact that we allow the adversary to change the components in an ℓ_∞ sense. Indeed, it is more challenging to find simple examples if the threat model consist of changes in ℓ_2. But similar ideas still apply.

The idea for the above model and further details come from the paper "Robustness May Be at Odds with Accuracy," by Tsipras, Dimitris, Santurkar, Shibani, Engstrom, Logan, Turner, Alexander, and Madry, Aleksander.

Why Some ML Algorithms Might Not Be Robust – The Curse of Dimensionality – Again!

Let us talk about a simple setting to see why in high dimensions adversarial examples cannot be completely avoided.
We consider a binary classification example. Let us assume that $\mathcal{X} = \mathbb{R}^{500}$, i.e., $D = 500$. Our data is perfectly separable. For $y = -1$ the data is distributed uniformly on the surface of a sphere of radius 1 and for $y = 1$ the data is uniformly distributed on the surface of a sphere of radius 1.3. Both classes have equal size.
In the paper "The Relationship Between High-Dimensional Geometry and Adversarial Examples," by Justin Gilmer, Luke Metz, Fartash Faghri, Samuel S. Schoenholz, Maithra Raghu, Martin Wattenberg, and Ian Goodfellow the authors performed the following experiment.
They took a 2-hidden layer NN with ReLU activation functions and 500 nodes per hidden layer. They then trained with 50 million samples using SGD. The network trained very well, giving no error on 20 million test samples. Despite this they were easily able to find adversarial examples by moving test samples by a small amount (on the order of $1/\sqrt{D}$)!
Let us now discuss how this is an essentially unavoidable consequence of working in very high dimension.
Consider a classifier f that has a small but non-zero error probability. Consider lets say only those errors where $(\mathbf{x}, y)$ is such that $\|\mathbf{x}\|_2 = 1$ and $y = -1$ but $f(\mathbf{x}) = 1$. I.e., these are points lying on the inner sphere but that are classified

as belonging to the outer sphere. Let E be the set of such points that are misclassified.

For a reason that will become clear hopefully soon, assume that this set forms a spherical cap in the e_1 direction, i.e., these are the points $\mathbf{x}$ of the form $\|\mathbf{x}\|_2 = 1$ and $\mathbf{x}_1 \geq \alpha$ for some appropriate constant α. Without loss of generality we can assume that the error probability, call it p, fulfills $0 \leq p \leq \frac{1}{2}$ so that $\alpha \geq 0$.

Given p what is the value of α that gives us p (recall that we assume a uniform distribution of the data on the sphere)? This amounts to computing the surface area of the spherical cap, dividing it by the surface area of the whole sphere and equating this ratio to p. Recall that the sphere has radius 1. If we are using our low-D intuition we might think that as p varies from 0 to $\frac{1}{2}$, α will vary from 1 to 0. But in fact, we will see now that α is of order $1/\sqrt{D}$ and that p only modulates the constant in front of this expression! This means that as D becomes large, the spherical cap extends almost completely down to the equator. In other words, almost all the mass sits at the equator. As a consequence, adversarial examples are essentially unavoidable – if we randomly pick a point it is likely very close to the equator. And such a point is hence very likely $1/\sqrt{D}$-close to a point that f misclassifies. In other words, it is easy to find small adversarial moves that will cause the point to be misclassified!

It remains to clarify two things. First, let us check for the example above that indeed the spherical cap almost extends all the way down to the equator when D is large and the error probability is non-zero. Second, why did we assume

that the error set forms a spherical cap?
To answer the first question we will take a convenient short-cut. In principle the area of a spherical cap is explicitly known. But the expression is somewhat unwieldy. But note the following. Picking a point uniformly at random on a sphere of radius 1 in D-dimensional real space is almost the same as picking a Gaussian vector with iid zero-mean components of variance $\sigma^2 = 1/D$. Such a vector will be spherically symmetric and it will have a norm very close to 1.
But for the latter model it is very easy to assess the probability that this vector extends beyond a positive number β in the first component. This probability is given by

$$\int_\beta^\infty \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x_1^2}{2\sigma^2}} dx_1 = p.$$

One way to express this is to define

$$Q(x) = \int_x^\infty \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx.$$

E.g., $Q(1) = 0.16$, i.e., the probability that a zero-mean unit-variance Gaussian has a value larger than 1 is about 16 percent. With this function we get the expression

$$Q(\beta/\sigma) = p.$$

Since $\sigma = \frac{1}{\sqrt{n}}$, we see from this expression that β must be of the form $\alpha/\sqrt{n}$, as claimed. More precisely,

$$\beta = Q^{-1}(p)/\sqrt{n}.$$

To answer the second question consider the following. You likely know that, in any dimension and for a fixed volume,

the body that has the smallest surface area is the sphere. This is a famous so-called isoperimetric inequalities. Many such isoperimetric inequalities exist. The version we need is the following. Consider a sphere in D dimension and and a subset E on the surface of this sphere of a fixed size. Now add to this subset all points on the sphere that have distance no more than ϵ away from this set E. What is the smallest this set can be? It turns out that the answer is that the resulting set is the smallest if we start with a spherical cap! Applied to our problem this means that by assuming that the misclassified set was a spherical cap we in fact computed a lower bound on the adversarial misclassification rate!

Constructing a Robust Classifier from Scratch – Adversarial Training

So far we have assumed that we are given a classifier and we discussed what might happen if we allow adversarial changes to the input. But we can take a more pro-active approach. First, we can include the aim for adversarial robustness in the training phase. This is what we discuss now. Second, given a classifier we can ask if we can modify it to make it more robust. We will briefly discuss this second approach in the next section. In practice we might want to apply both techniques. We will be *very* brief.

The approach taken in the paper "Towards Deep Learning Models Resistant to Adversarial Attacks" by Madry, Aleksander, Makelov, Aleksandar, Schmidt, Ludwig, Tsipras, Dimitris, and Vladu is the following perhaps most natural idea.

Rather than minimizing the usual cost function

$$\min_{\Theta} \ \big[\mathcal{L}(f) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\mathbb{1}_{\{f_\Theta(\mathbf{x})\neq y\}}]\big]$$

over all parameters Θ of the model, why not directly minimize what matters, namely

$$\min_{\Theta} \ \big[\mathcal{R}(f,\varepsilon) = \min_{\Theta} \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\max_{\tilde{\mathbf{x}}:\|\mathbf{x}-\tilde{\mathbf{x}}\|\leq\varepsilon} \mathbb{1}_{\{f(\tilde{\mathbf{x}})\neq y\}}]\big].$$

This leads to a min-max formulation. As written, this function is not easy to optimize for several reasons. The first is that it is not smooth (due to the indicator function). Therefore let us instead look at the problem

$$\min_{\Theta} \ \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}\big[\max_{\tilde{\mathbf{x}}:\|\mathbf{x}-\tilde{\mathbf{x}}\|\leq\varepsilon} (\tfrac{1}{2} - y(g(\tilde{\mathbf{x}}) - \tfrac{1}{2}))\big].$$

Here we assumed that we deal with a binary classification problem, i.e., $y \in \{\pm 1\}$ and that the classifier $f(\mathbf{x})$ has the form (1) so that $(\frac{1}{2} - y(g(\tilde{\mathbf{x}}) - \frac{1}{2}))$ is the predicted probability of the incorrect label. Now we are dealing with a smooth function.
Next, we do not have access to the distribution but instead we have a sample $\mathcal{S}$. Therefore the equivalent problem is

$$\min_{\Theta} \frac{1}{N}\sum_{n=1}^{N} \max_{\tilde{\mathbf{x}_n}:\|\mathbf{x}_n-\tilde{\mathbf{x}}_n\|\leq\epsilon} (\frac{1}{2} - y_n(g(\tilde{\mathbf{x}}_n) - \tfrac{1}{2})).$$

It is still not completely obvious how to minimize this. It turns out that the following is correct. Compute for each $\mathbf{x}_n$ the worst perturbation $\tilde{\mathbf{x}}_n$. Then take the gradient of this

expression and move towards the negative gradient direction. Of course, this is computationally intensive since for each sample and each iteration we need to find the worst-case perturbation.

Let us apply this algorithm to our example with one robust and many non-robust features. To keep things simple let us assume that we even know that the optimum classifier is of the form as discussed, i.e., we first form the sum $\sum_{i=1}^{D} \mathbf{x}_i a_i$ and then take the sign, but we do not know what the best constants a_i are that we should use. Hence, we are led to the optimization task

$$\max_a \frac{1}{N} \sum_{n=1}^{N} \min_{\tilde{\mathbf{x}_n}: \|\mathbf{x}_n - \tilde{\mathbf{x}}_n\|_\infty \leq \epsilon} y_n \sum_{i=1}^{D} (\tilde{\mathbf{x}}_n)_i a_i + \lambda(\sum_{i=1}^{D} a_i^2 - 1 - \log(D)),$$

where $\epsilon = 2\sqrt{\frac{\log(D)}{D-1}}$ and where we added the term $\lambda(\sum_{i=1}^{D} a_i^2 - 1 - \log(D))$ in order to limit the scale of the coefficients.

Assume at first that we started with the optimum choice for the non-adversarial setting, i.e., $a_1 = 1$ and $a_i = \alpha = \sqrt{\frac{\log(D)}{D-1}}$. And now let us do one gradient step. The adversary will move all features in the "incorrect direction" by an amount 2α. This will leave the robust feature essentially unchanged but will "flip" all non-robust features. The gradient will have the form

$$\begin{aligned} (1 - 2\alpha + 2\lambda) + \mathcal{N}(0, \sigma^2 = \frac{1}{N}), \quad i = 1, \\ -\alpha(1 - 2\lambda) + \mathcal{N}(0, \sigma^2 = \frac{1}{N}), \quad i = 2, \cdots, D, \end{aligned}$$

and we will walk a little bit into the direction of this gradient (in the current setting we want to maximize and not

to minimize the given expression). If N is large then the additional noise is negligible and the direction will become deterministic. Further, think of λ as small. We see that the non-robust features will move towards zero.
Indeed, if we consider as a second example the case where we set $a_i = 0$, $i = 2, \cdots, D$, and pick a_1 to be positive, we see that this is a fixed point of the adversarial training algorithm. This training algorithm is called *adversarial training*.

Making an Existing Classifier Robust – Randomized Smoothing

In some instances we might be handed a classifier that has small standard risk but might be prone to adversarial errors and are asked to make it more robust rather than learning a new classifier from scratch. To date the perhaps most promising idea of accomplishing this is *randomized smoothing*.
Let f be the given classifier. Assume that it maps $\mathbb{R}^D$ to $\mathcal{Y}$, the set of labels. From this we derive the *smoothed* classifier, call it g. This new smoothed classifier g maps an input $\mathbf{x}$ to that class c that is most likely returned by f if presented the input $\mathbf{x} + \mathbf{z}$, where $\mathbf{z}$ is a vector of iid zero-mean random Gaussian variables with a variance of σ^2 per component. This idea was introduced by Lecuyer, M., Atlidakis, V., Geambasu, R., Hsu, D.,and Jana, S. in the paper entitled "Certified robustness to adversarial examples with differential privacy". It was shown to work well for various test data sets like ImageNet. There are several variants of

this. E.g., rather than using this probabilistic definition using Gaussians we could average over a ball of radius $\sqrt{D}\sigma$ (with a uniform distribution). The effect would be more or less the same. This resulting training algorithm variants are called *robust training*.

The basic idea why this adds robustness is the following. Consider e.g. the binary case. Take a point $\mathbf{x}$ and consider the original classifier f. Lets say that the label $y = 1$ is much more likely to be returned by f if we give it the point $\mathbf{x} + \mathbf{z}$ than the label $y = -1$. Now consider what happens if we give f the label $\tilde{\mathbf{x}} + \mathbf{z}$ instead, where $\|\tilde{\mathbf{x}} - \mathbf{x}\|_2 \leq \epsilon$ for some not too large ϵ. Since the resulting points in the two cases are picked with a probability that is not too different it is intuitive that in average we are still more likely to return $y = 1$ rather than $y = -1$. I.e., we will return the same label for points not too far away. Let us make this precise.

Consider first the case of $D = 1$, i.e., we are operating on a line. On the line each point x has a label $f(x) \in \{\pm 1\}$ attached to it. For a point x on this line define $p = \mathbb{E}[\mathbb{1}_{\{f(x+z)=1\}}]$. Let us assume that $\frac{1}{2} < p \leq 1$. This means that "more" of the points in the neighborhood of x are given the label $y = 1$ than the label $y = -1$. Consider now $\tilde{p} = \mathbb{E}[\mathbb{1}_{\{f(\tilde{x}+z)=1\}}]$ where $\|\tilde{\mathbf{x}} - \mathbf{x}\|_2 \leq \epsilon$. Assume e.g. that the new point $\tilde{x}$ is moved ϵ to the right from x. How small can $\tilde{p}$ be. A little bit of thought shows that the worst case is if all the points that were originally labeled $y = -1$ were on the right of the point x in the tail of the Gaussian. More precisely, the worst case happens if all the point to the left of $x + Q^{-1}((1-p)/\sigma)$ are labeled $y = 1$ and all points

to the right of this point are labeled $y = -1$. How far can we then move $\tilde{x}$ to the right away from x so that still the majority of the points is labeled $y = 1$. We see that we can move it at most by $Q^{-1}((1-p)/\sigma)$. This is pleasing. The larger p, i.e., the more biased the original average was, the more we are adversarially robust.

This was in 1-D. But exactly the same happens in any dimension since the noise is Gaussian with iid components. Hence, no matter what direction we are considering, in this direction we are dealing with a Gaussian with zero mean and variance σ^2.

So why not choose a very large σ so that we get a very large robustness radius? The averaging (smoothing) will in general increase the standard risk and it can do so considerably. I.e., there might be a considerable price to pay for the added robustness.

Machine Learning Course - CS-433

Unsupervised Learning

Nov 19, 2020

changes by Martin Jaggi 2020, 2019, changes by Rüdiger Urbanke 2018, changes by Martin Jaggi 2016, 2017 ©Mohammad Emtiyaz Khan 2015

Last updated on: November 17, 2020

EPFL

Unsupervised learning

How can systems learn when there are no labels available? How to learn a meaningful internal representation for data examples? I.e., to represent them in a way that reflects the semantic structure of the overall collection of input patterns? This question is the central focus of unsupervised learning.

In unsupervised learning, our data consists only of features (or inputs) $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$, vectors in $\mathbb{R}^D$, and there are **no outputs** y_n available.

Unsupervised learning seems to play an important role in how living beings learn. Variants of it seem to be more common in the brain than supervised learning.

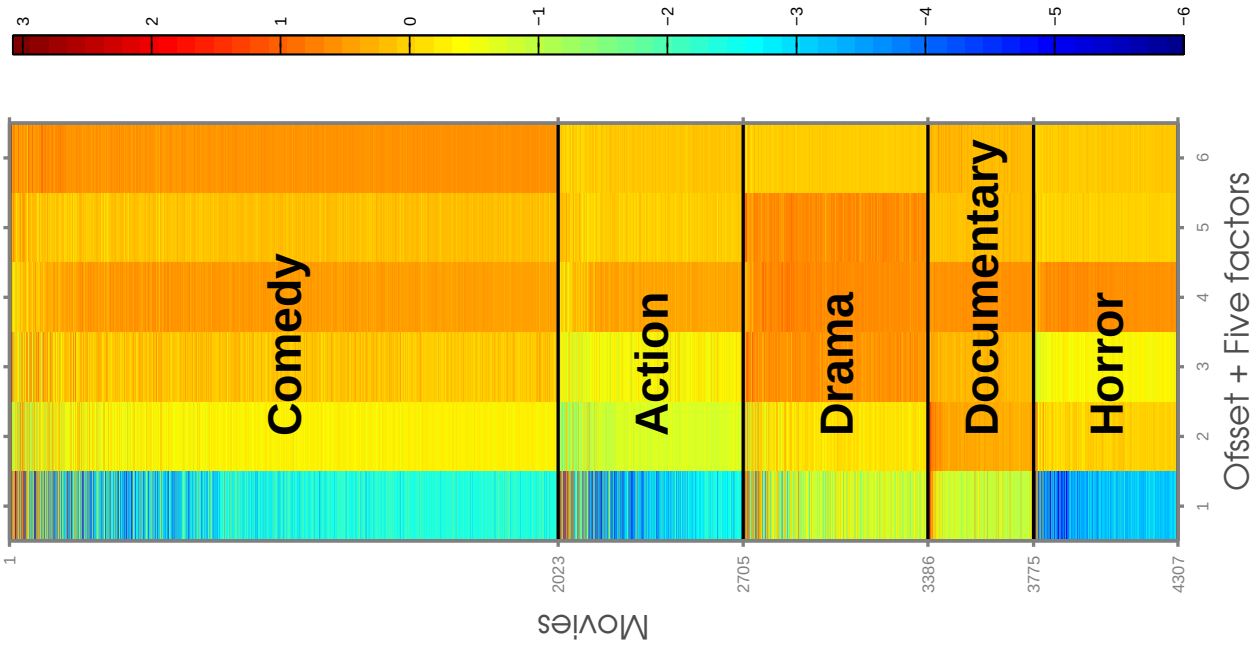
Two main directions in unsupervised learning are

- representation learning and
- density estimation & generative models

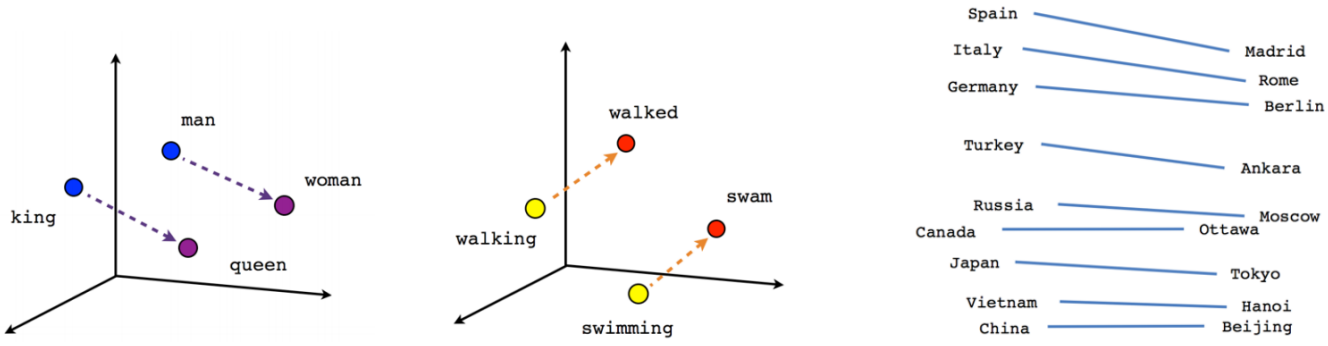
Examples

Examples for Representation Learning

Given ratings of movies and viewers, we use matrix factorization to extract useful features (see e.g. Netflix Prize).



Learning word-representations using matrix-factorizations, `word2vec` (Mikolov et al. 2013).



Male-Female | Verb tense | Country-Capital

Given voting patterns of regions across Switzerland, we use PCA to extract useful features (Etter et al. 2014).

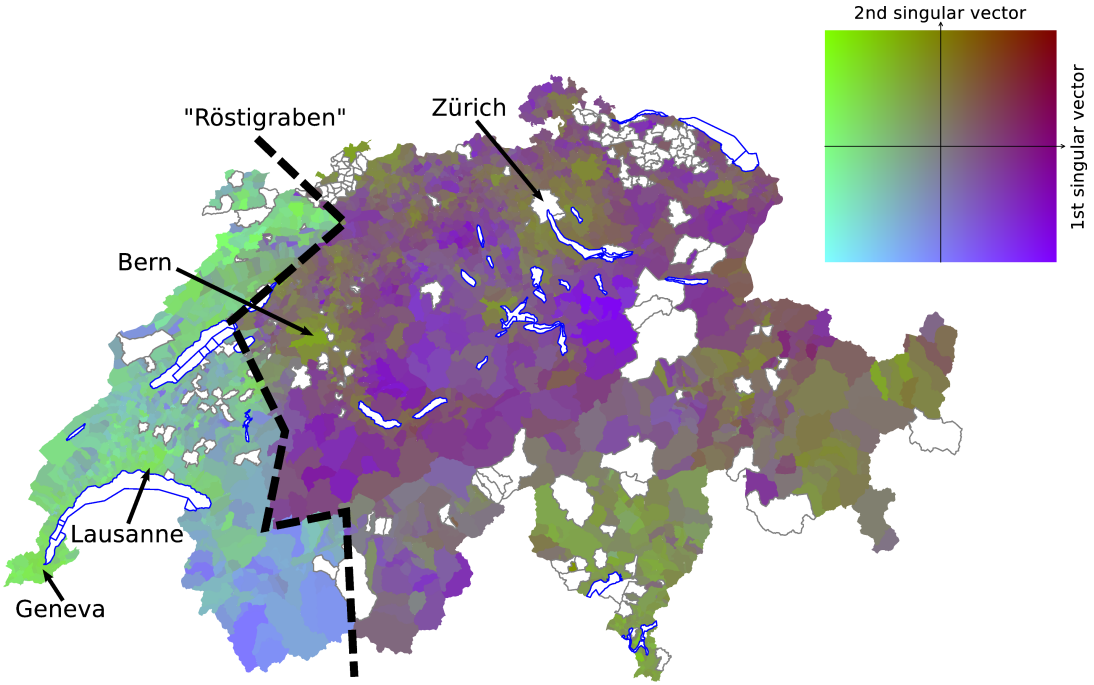
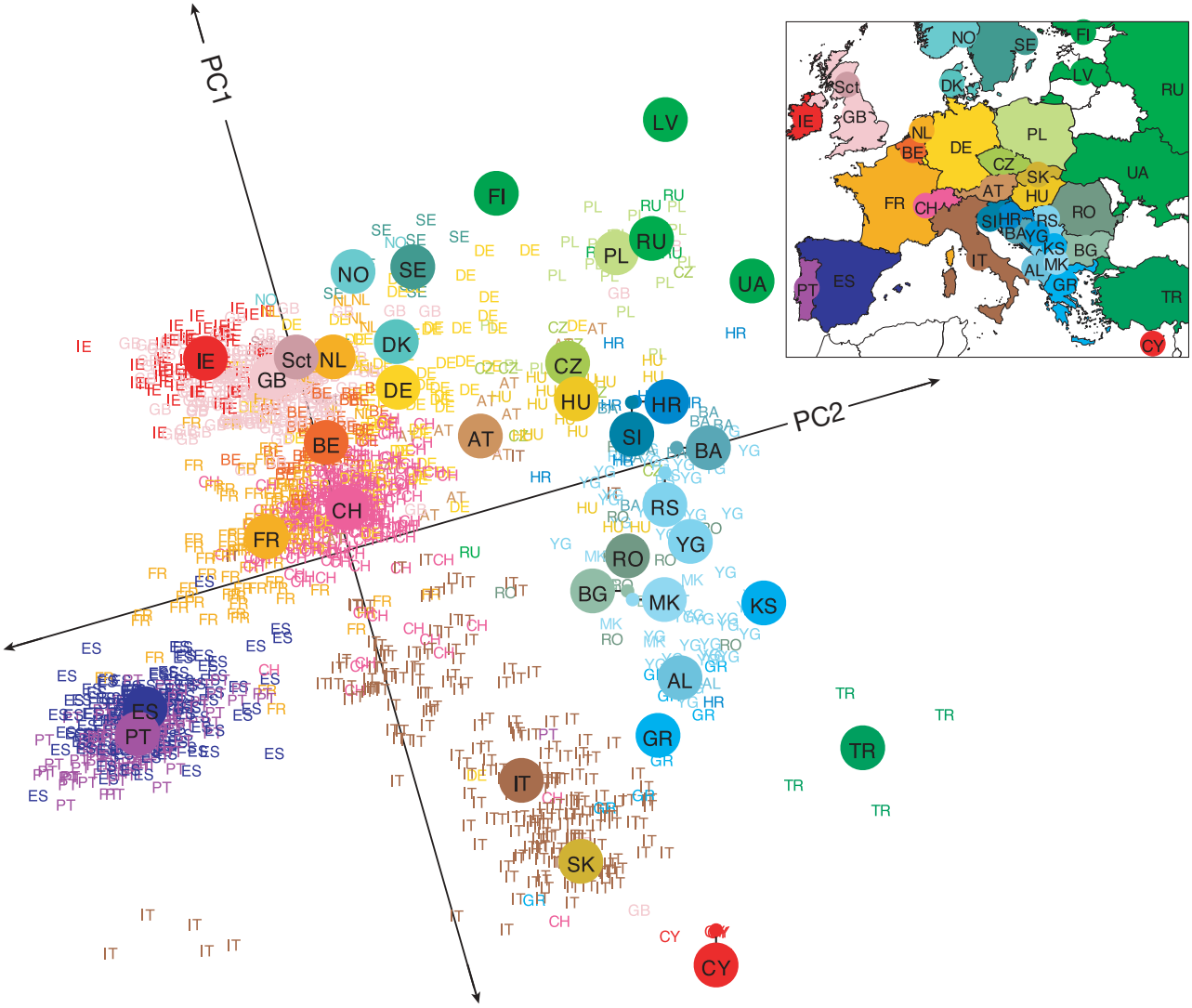


Figure 9: Voting patterns of Swiss municipalities. The color of a municipality is assigned using its location in Figure 8 and the color gradient shown in the upper right corner. Two municipalities with similar colors have similar voting patterns. The *Röstigraben*, corresponding to the cultural difference between French-speaking municipalities and German-speaking ones, is clearly visible from the difference in voting patterns. Regions shown in white are lakes or municipalities for which some vote results are missing (due to a merging of municipalities, for example). A more detailed map can be found online [2].

PCA Example 2: Genes mirror geography



Nature 2008, http://dx.doi.org/10.1038/nature07331

Examples for Clustering

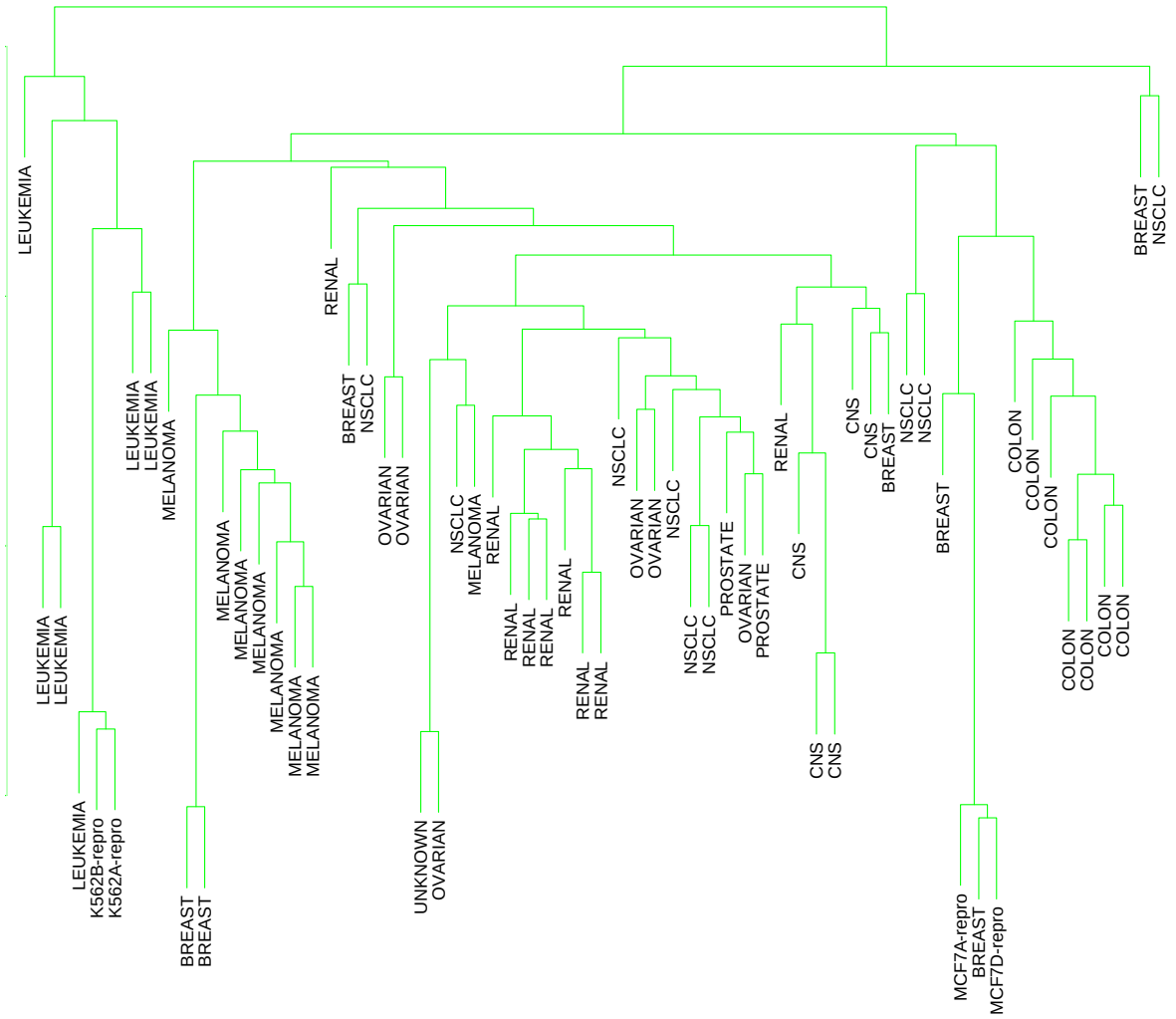
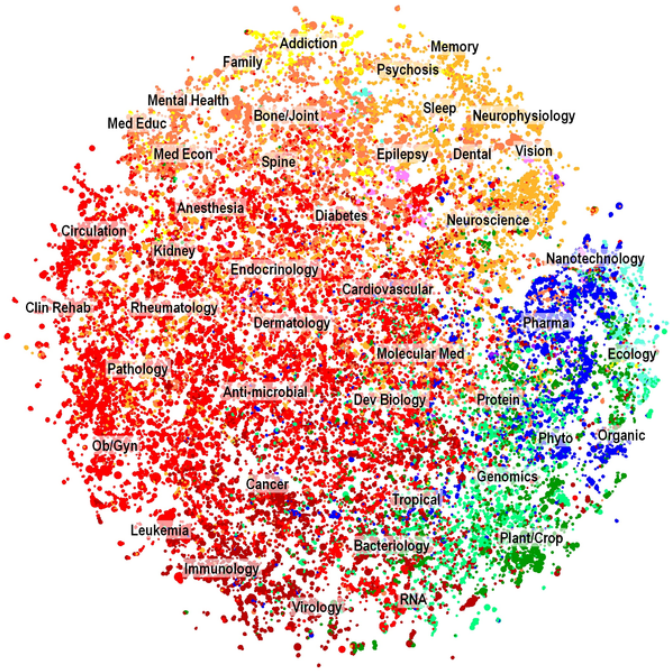


FIGURE 14.12. *Dendrogram from agglomerative hierarchical clustering with average linkage to the human tumor microarray data.*

Clustering more than two million biomedical publications (Kevin Boyack et.al. 2011)



Clustering articles published in Science (Blei & Lafferty 2007)

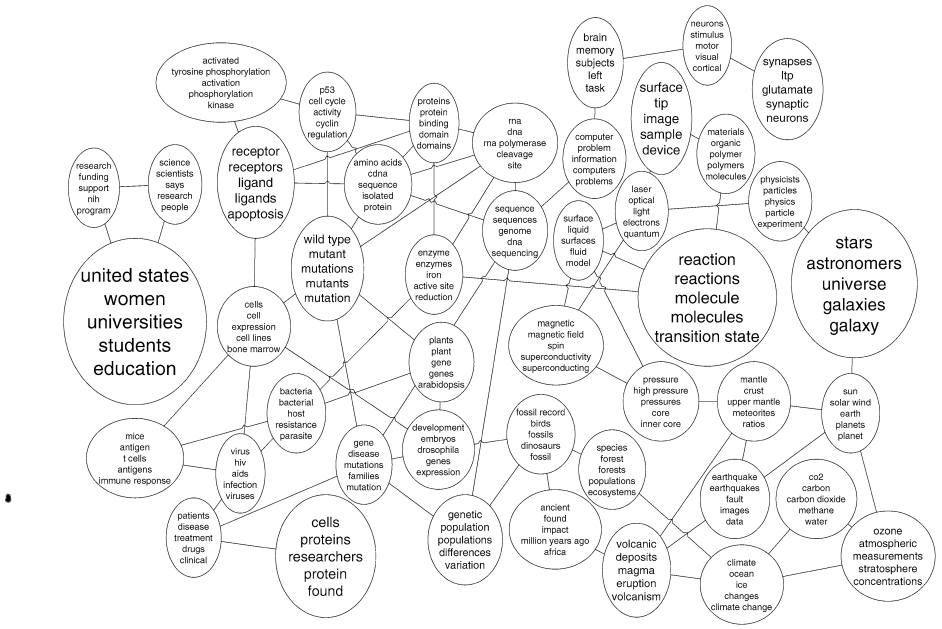


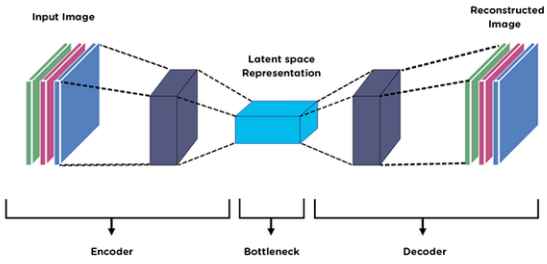
FIG. 2. *A portion of the topic graph learned from* 16,351 *OCR articles from Science* (1990–1999). *Each topic node is labeled with its five most probable phrases and has font proportional to its popularity in the corpus.* (*Phrases are found by permutation test.*) *The full model can be found in http://www.cs.cmu.edu/~lemur/science/ and on STATLIB.*

Unsupervised Representation Learning & Generation

How does it work?

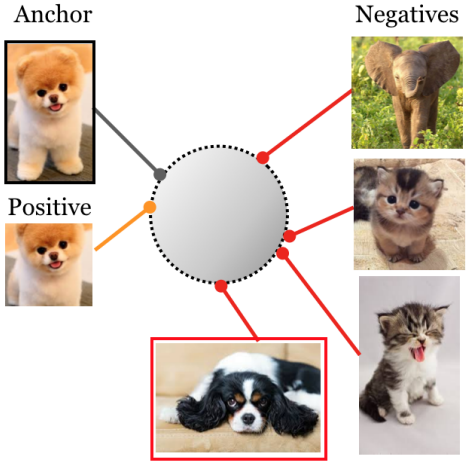
Define a unsupervised or self-supervised loss function, for

- Compression & Reconstruction
  (e.g. Auto-Encoder)



. image: Deepak Birla

- Consistency & Contrastive Learning
  (e.g. Noise-contrastive estimation)



. image: arxiv.org/pdf/2004.11362

- Generation
  (e.g. Auto-Encoder, Gaussian Mixture Model)

Examples:

(G = can be used as a generative model)

- Auto-Encoders (G)
  Invertible networks, learned compression, normalizing flows

- Image Representation Learning
  combining unsupervised representation learning (pre-training) with supervised learning

- Language models & sequential data
  text generation (G), or sequence continuation (G), BERT (G)
  video, audio & timeseries (auto-regressive, contrastive or G)

- Generative Adversarial Networks (GAN) (G)
  see also *predictability minimization*

"Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Network", ICLR 2016, https://arxiv.org/abs/1511.06434

Machine Learning Course - CS-433

K-Means Clustering

Nov 19, 2020

changes by Martin Jaggi 2020, 2019, changes by Rüdiger Urbanke 2018, changes by Martin Jaggi 2016, 2017 ©Mohammad Emtiyaz Khan 2015

Last updated on: November 17, 2020

EPFL

Clustering

Clusters are groups of points whose inter-point distances are small compared to the distances outside the cluster.

The goal is to find "prototype" points $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2,, \ldots, \boldsymbol{\mu}_K$ and cluster assignments $z_n \in \{1, 2, \ldots, K\}$ for all $n = 1, 2, \ldots, N$ data vectors $\mathbf{x}_n \in \mathbb{R}^D$.

K-means clustering

Assume K is known.

$$\min_{\mathbf{z}, \boldsymbol{\mu}} \mathcal{L}(\mathbf{z}, \boldsymbol{\mu}) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|_2^2$$

$$\text{s.t. } \boldsymbol{\mu}_k \in \mathbb{R}^D, z_{nk} \in \{0, 1\}, \sum_{k=1}^{K} z_{nk} = 1,$$

$$\text{where } \mathbf{z}_n = [z_{n1}, z_{n2}, \ldots, z_{nK}]^\top$$

$$\mathbf{z} = [\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_N]^\top$$

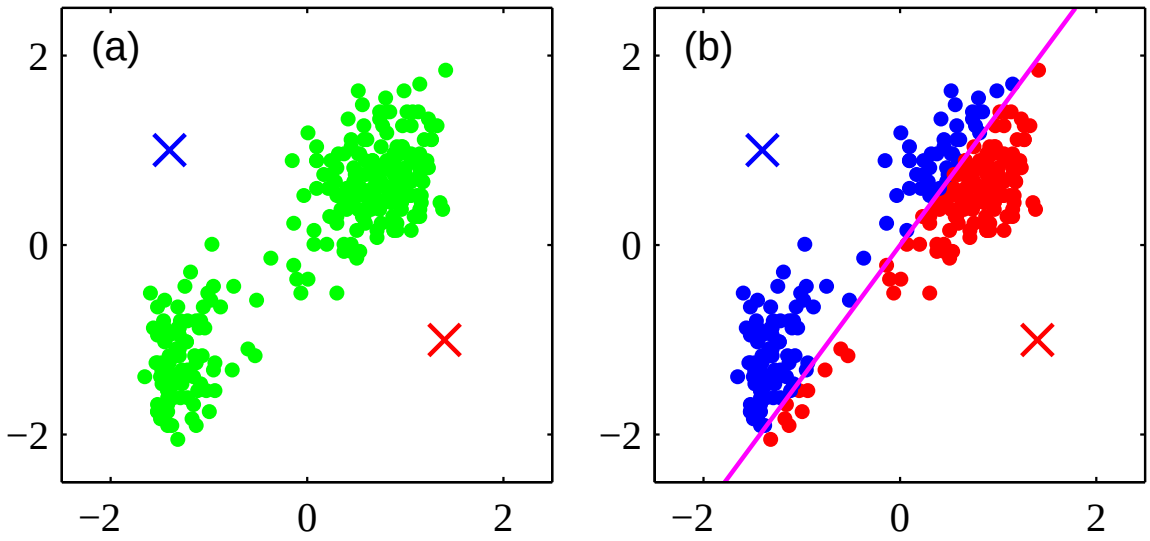
$$\boldsymbol{\mu} = [\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_K]^\top$$

Is this optimization problem easy?

Algorithm: Initialize $\boldsymbol{\mu}_k \, \forall k$,
then iterate:

1. For all n, compute $\mathbf{z}_n$ given $\boldsymbol{\mu}$.
2. For all k, compute $\boldsymbol{\mu}_k$ given $\mathbf{z}$.

Step 1: For all n, compute $\mathbf{z}_n$ given $\boldsymbol{\mu}$.

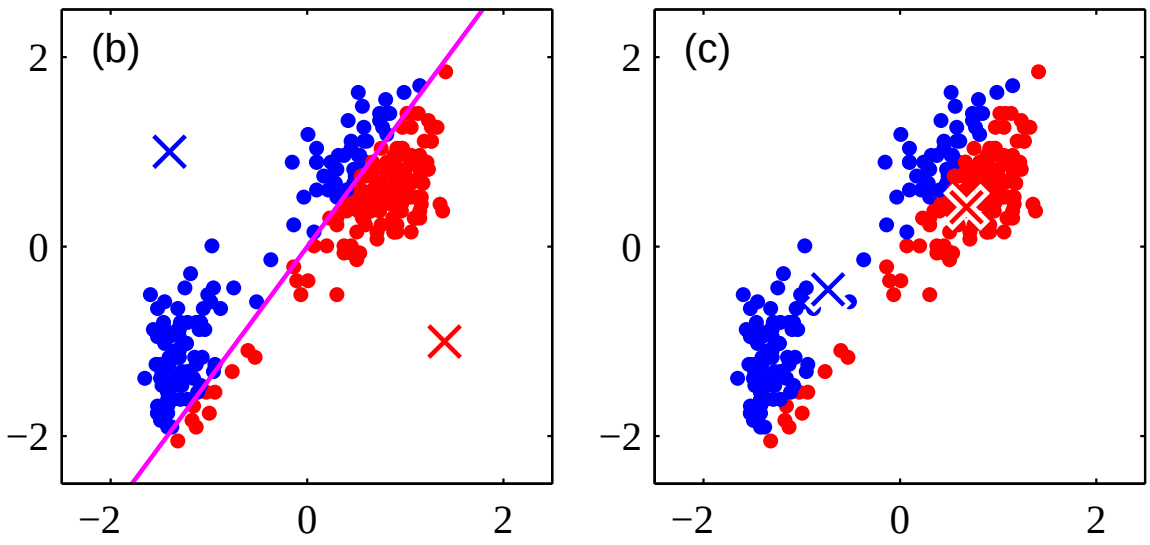


$$z_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_{j=1,2,\ldots K} \|\mathbf{x}_n - \boldsymbol{\mu}_j\|_2^2 \\ 0 & \text{otherwise} \end{cases}$$

Step 2: For all k, compute $\boldsymbol{\mu}_k$ given $\mathbf{z}$.
Take derivative w.r.t. $\boldsymbol{\mu}_k$ to get:

$$\boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} z_{nk} \mathbf{x}_n}{\sum_{n=1}^{N} z_{nk}}$$

Hence, the name 'K-means'.



Summary of K-means

Initialize $\boldsymbol{\mu}_k \, \forall k$, then iterate:

1. For all n, compute $\mathbf{z}_n$ given $\boldsymbol{\mu}$.

$$z_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_j \|\mathbf{x}_n - \boldsymbol{\mu}_j\|_2^2 \\ 0 & \text{otherwise} \end{cases}$$

2. For all k, compute $\boldsymbol{\mu}_k$ given $\mathbf{z}$.

$$\boldsymbol{\mu}_k = \frac{\sum_{n=1}^N z_{nk} \mathbf{x}_n}{\sum_{n=1}^N z_{nk}}$$

Convergence to a local optimum is assured since each step decreases the cost (see Bishop, Exercise 9.1).

Coordinate descent

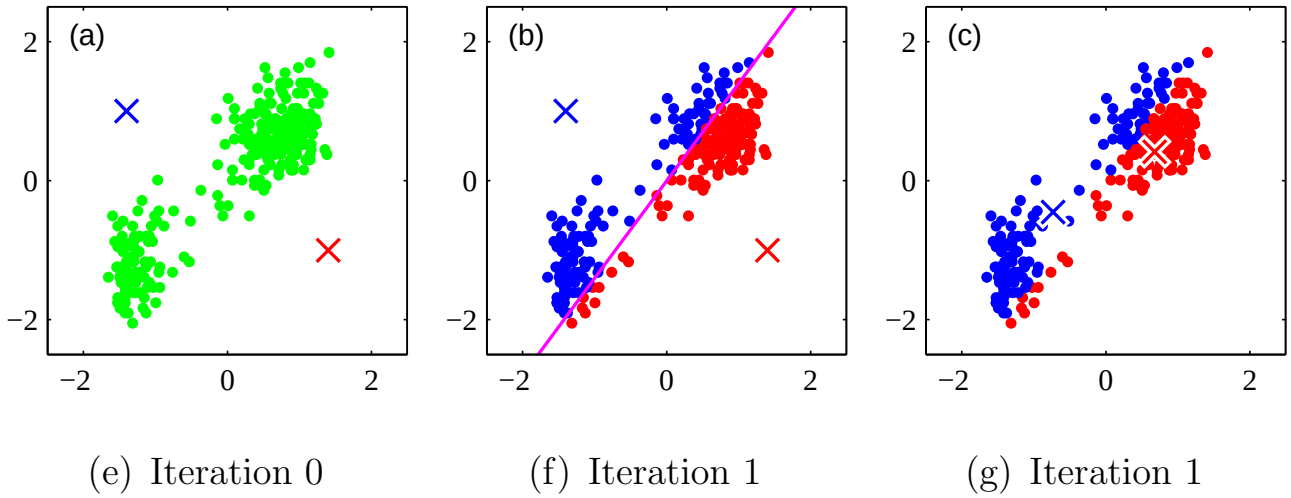
K-means is a coordinate descent algorithm, where, to find $\min_{\boldsymbol{z},\boldsymbol{\mu}} \mathcal{L}(\mathbf{z}, \boldsymbol{\mu})$, we start with some $\boldsymbol{\mu}^{(0)}$ and repeat the following:

$$\mathbf{z}^{(t+1)} := \arg\min_{\boldsymbol{z}} \; \mathcal{L}(\mathbf{z}, \boldsymbol{\mu}^{(t)})$$

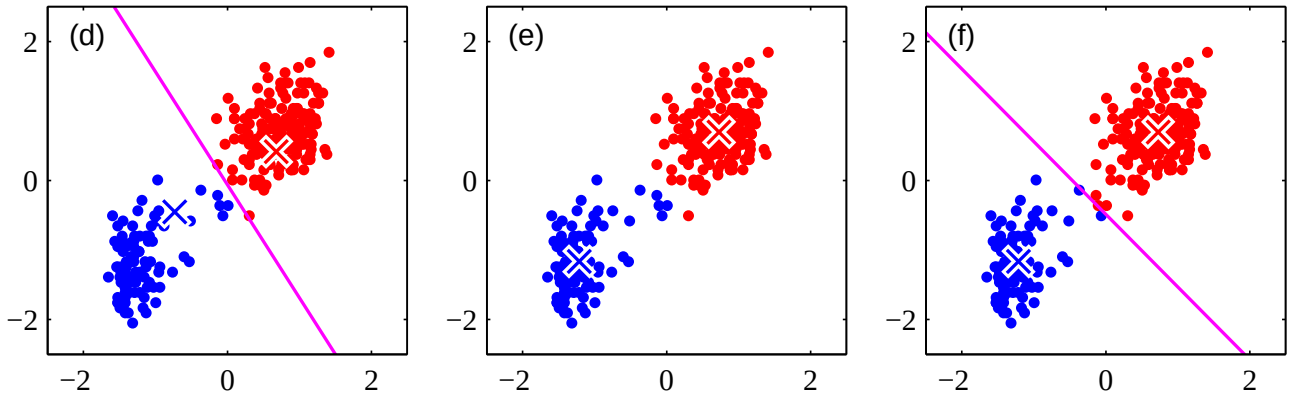
$$\boldsymbol{\mu}^{(t+1)} := \arg\min_{\boldsymbol{\mu}} \; \mathcal{L}(\mathbf{z}^{(t+1)}, \boldsymbol{\mu})$$

Examples

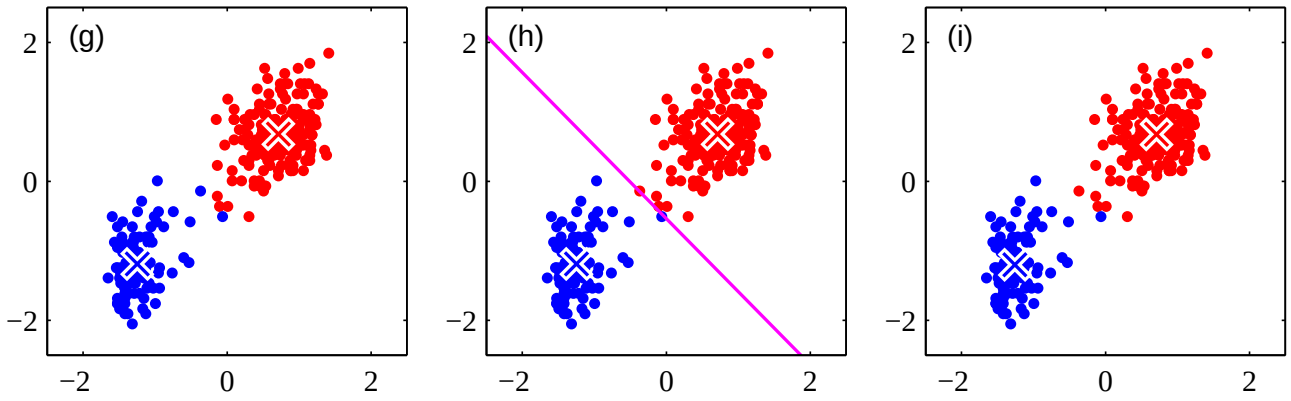
K-means for the "old-faithful" dataset (Bishop's Figure 9.1)



(e) Iteration 0 (f) Iteration 1 (g) Iteration 1

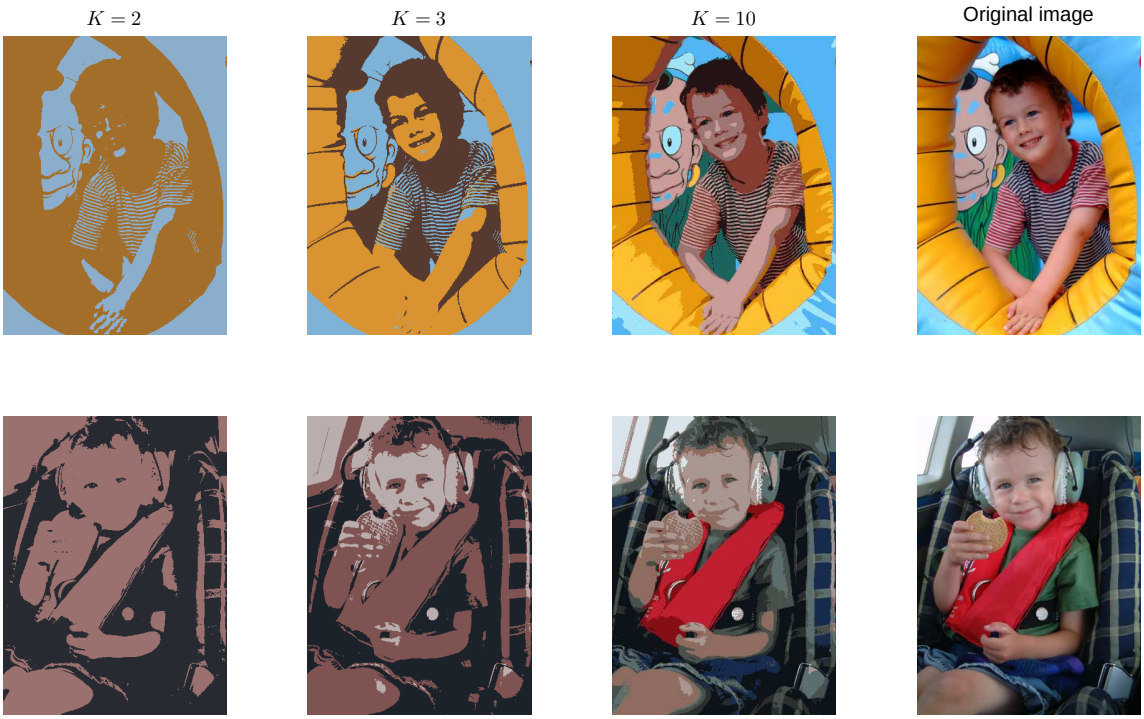


(h) Iteration 2 (i) Iteration 2 (j) Iteration 3



(k) Iteration 3 (l) Iteration 4 (m) Iteration 4

Data compression for images (this is also known as vector quantization).



Probabilistic model for K-means

K-means as a Matrix Factorization

Recall the objective

$$\min_{\mathbf{z},\boldsymbol{\mu}} \mathcal{L}(\mathbf{z},\boldsymbol{\mu}) = \sum_{n=1}^{N}\sum_{k=1}^{K} z_{nk}\|\mathbf{x}_n - \boldsymbol{\mu}_k\|_2^2$$
$$= \|\mathbf{X}^\top - \mathbf{M}\mathbf{Z}^\top\|_{\mathsf{Frob}}^2$$

$$\text{s.t. } \boldsymbol{\mu}_k \in \mathbb{R}^D,$$
$$z_{nk} \in \{0,1\}, \sum_{k=1}^{K} z_{nk} = 1.$$

Issues with K-means

1. Computation can be heavy for large N, D and K.
2. Clusters are forced to be spherical (e.g. cannot be elliptical).
3. Each example can belong to only one cluster ("hard" cluster assignments).

Machine Learning Course - CS-433

Gaussian Mixture Models

Nov 24, 2020

changes by Martin Jaggi 2020, 2019, changes by Rüdiger Urbanke 2018, changes by Martin Jaggi 2016, 2017

Last updated on: November 24, 2020

EPFL

Motivation

K-means forces the clusters to be *spherical*, but sometimes it is desirable to have *elliptical* clusters. Another issue is that, in K-means, each example can only belong to one cluster, but this may not always be a good choice, e.g. for data points that are near the "border". Both of these problems are solved by using Gaussian Mixture Models.

Clustering with Gaussians

The first issue is resolved by using full covariance matrices $\mathbf{\Sigma}_k$ instead of *isotropic* covariances.

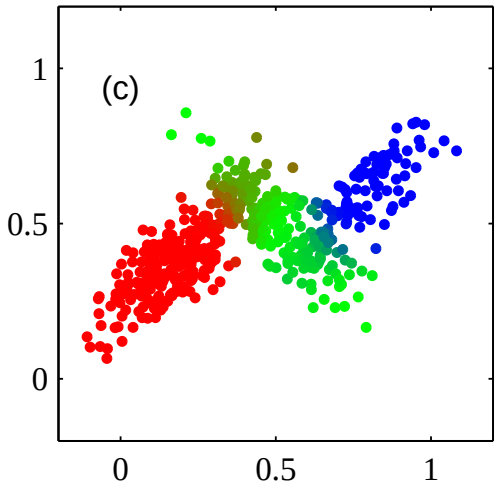
$$p(\mathbf{X}|\boldsymbol{\mu}, \mathbf{\Sigma}, \mathbf{z}) = \prod_{n=1}^{N} \prod_{k=1}^{K} [\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \mathbf{\Sigma}_k)]^{z_{nk}}$$

Soft-clustering

The second issue is resolved by defining z_n to be a random variable. Specifically, define $z_n \in \{1, 2, \ldots, K\}$ that follows a multinomial distribution.

$$p(z_n = k) = \pi_k \text{ where } \pi_k > 0, \forall k \text{ and } \sum_{k=1}^{K} \pi_k = 1$$

This leads to soft-clustering as opposed to having "hard" assignments.



Gaussian mixture model

Together, the likelihood and the prior define the joint distribution of Gaussian mixture model (GMM):

$$
\begin{aligned}
& p(\mathbf{X}, \mathbf{z} \,|\, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) \\
&= \prod_{n=1}^{N} p(\mathbf{x}_n | z_n, \boldsymbol{\mu}, \boldsymbol{\Sigma}) p(z_n | \boldsymbol{\pi}) \\
&= \prod_{n=1}^{N} \prod_{k=1}^{K} [\mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)]^{z_{nk}} \prod_{k=1}^{K} [\pi_k]^{z_{nk}}
\end{aligned}
$$

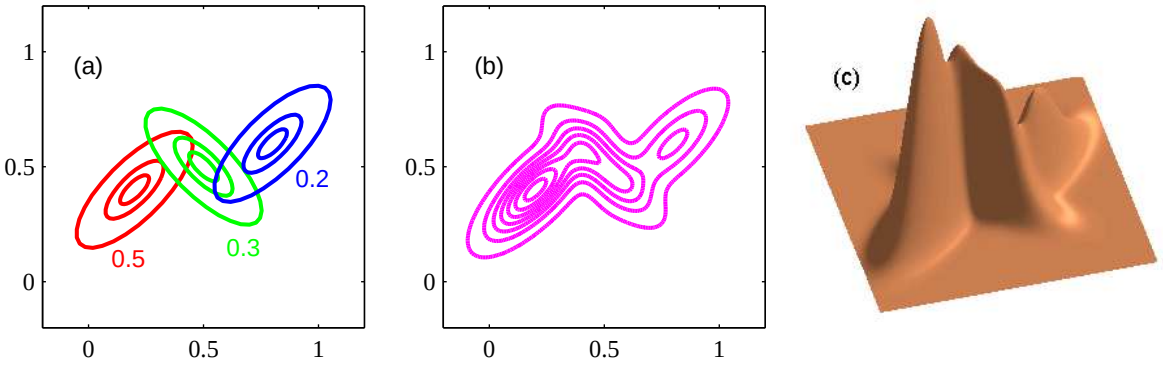
Here, $\mathbf{x}_n$ are observed data vectors, z_n are *latent* unobserved variables, and the unknown *parameters* are given by $\boldsymbol{\theta} := \{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K, \boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\Sigma}_K, \boldsymbol{\pi}\}$.

Marginal likelihood

GMM is a latent variable model with z_n being the unobserved (latent) variables. An advantage of treating z_n as latent variables instead of *parameters* is that we can *marginalize* them out to get a cost function that does not depend on z_n, i.e. as if z_n never existed.

Specifically, we get the following marginal likelihood by marginalizing z_n out from the likelihood:

$$p(\mathbf{x}_n|\boldsymbol{\theta}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$



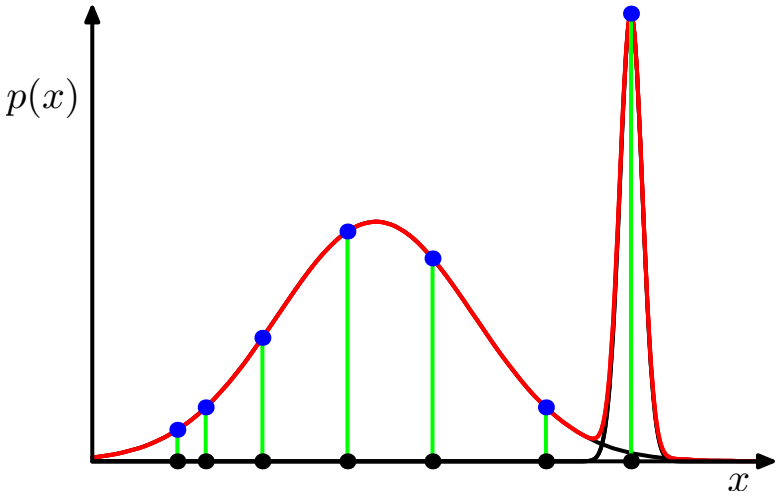
Deriving cost functions this way, is good for *statistical efficiency*. Without a latent variable model, the number of parameters grow at rate $O(N)$. After marginalization, the growth is reduced to $O(D^2K)$ (assuming $D, K \ll N$).

Maximum likelihood

To get a maximum (marginal) likelihood estimate of $\boldsymbol{\theta}$, we maximize the following:

$$\max_{\boldsymbol{\theta}} \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

Is this cost convex? Identifiable? Bounded?



Machine Learning Course - CS-433

Expectation-Maximization Algorithm

Nov 26, 2020

changes by Martin Jaggi 2020, 2019, changes by Rüdiger Urbanke 2018, changes by Martin Jaggi 2016, 2017 ©Mohammad Emtiyaz Khan 2015

Last updated on: November 24, 2020

Motivation

Computing maximum likelihood for Gaussian mixture model is difficult due to the log outside the sum.

$$\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) := \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

Expectation-Maximization (EM) algorithm provides an elegant and general method to optimize such optimization problems. It uses an iterative two-step procedure where individual steps usually involve problems that are easy to optimize.

EM algorithm: Summary

Start with $\boldsymbol{\theta}^{(1)}$ and iterate:

1. Expectation step: Compute a lower bound to the cost such that it is tight at the previous $\boldsymbol{\theta}^{(t)}$:

$\mathcal{L}(\boldsymbol{\theta}) \geq \underline{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t)})$ and
$\mathcal{L}(\boldsymbol{\theta}^{(t)}) = \underline{\mathcal{L}}(\boldsymbol{\theta}^{(t)}, \boldsymbol{\theta}^{(t)})$.

2. Maximization step: Update $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}^{(t+1)} = \arg\max_{\boldsymbol{\theta}} \underline{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t)}).$$

Concavity of log

Given non-negative weights q s.t. $\sum_k q_k = 1$, the following holds for any $r_k > 0$:

$$\log\left(\sum_{k=1}^{K} q_k r_k\right) \geq \sum_{k=1}^{K} q_k \log r_k$$

The expectation step

$$\log \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \geq \sum_{k=1}^{K} q_{kn} \log \frac{\pi_k \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{q_{kn}}$$

with equality when,

$$q_{kn} = \frac{\pi_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}$$

This is not a coincidence.

The maximization step

Maximize the lower bound w.r.t. $\boldsymbol{\theta}$.

$$\max_{\boldsymbol{\theta}} \sum_{n=1}^{N}\sum_{k=1}^{K} q_{kn}^{(t)} \left[\log \pi_k + \log \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right]$$

Differentiating w.r.t. $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k^{-1}$, we can get the updates for $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$.

$$\boldsymbol{\mu}_k^{(t+1)} := \frac{\sum_n q_{kn}^{(t)} \mathbf{x}_n}{\sum_n q_{kn}^{(t)}}$$

$$\boldsymbol{\Sigma}_k^{(t+1)} := \frac{\sum_n q_{kn}^{(t)} (\mathbf{x}_n - \boldsymbol{\mu}_k^{(t+1)})(\mathbf{x}_n - \boldsymbol{\mu}_k^{(t+1)})^\top}{\sum_n q_{kn}^{(t)}}$$

For π_k, we use the fact that they sum to 1. Therefore, we add a Lagrangian term, differentiate w.r.t. π_k and set to 0, to get the following update:

$$\pi_k^{(t+1)} := \frac{1}{N}\sum_{n=1}^{N} q_{kn}^{(t)}$$

Summary of EM for GMM

Initialize $\boldsymbol{\mu}^{(1)}, \boldsymbol{\Sigma}^{(1)}, \boldsymbol{\pi}^{(1)}$ and iterate between the E and M step, until $\mathcal{L}(\boldsymbol{\theta})$ stabilizes.

1. E-step: Compute assignments $q_{kn}^{(t)}$:

$$q_{kn}^{(t)} := \frac{\pi_k^{(t)} \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k^{(t)}, \boldsymbol{\Sigma}_k^{(t)})}{\sum_{k=1}^{K} \pi_k^{(t)} \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k^{(t)}, \boldsymbol{\Sigma}_k^{(t)})}$$

2. Compute the marginal likelihood (cost).

$$\mathcal{L}(\boldsymbol{\theta}^{(t)}) = \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k^{(t)} \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k^{(t)}, \boldsymbol{\Sigma}_k^{(t)})$$

3. M-step: Update $\boldsymbol{\mu}_k^{(t+1)}, \boldsymbol{\Sigma}_k^{(t+1)}, \pi_k^{(t+1)}$.

$$\begin{aligned}
\boldsymbol{\mu}_k^{(t+1)} &:= \frac{\sum_n q_{kn}^{(t)} \mathbf{x}_n}{\sum_n q_{kn}^{(t)}} \\
\boldsymbol{\Sigma}_k^{(t+1)} &:= \frac{\sum_n q_{kn}^{(t)} (\mathbf{x}_n - \boldsymbol{\mu}_k^{(t+1)})(\mathbf{x}_n - \boldsymbol{\mu}_k^{(t+1)})^\top}{\sum_n q_{kn}^{(t)}} \\
\pi_k^{(t+1)} &:= \frac{1}{N} \sum_n q_{kn}^{(t)}
\end{aligned}$$

If we let the covariance be diagonal i.e. $\boldsymbol{\Sigma}_k := \sigma^2 \mathbf{I}$, then EM algorithm is same as K-means as $\sigma^2 \to 0$.

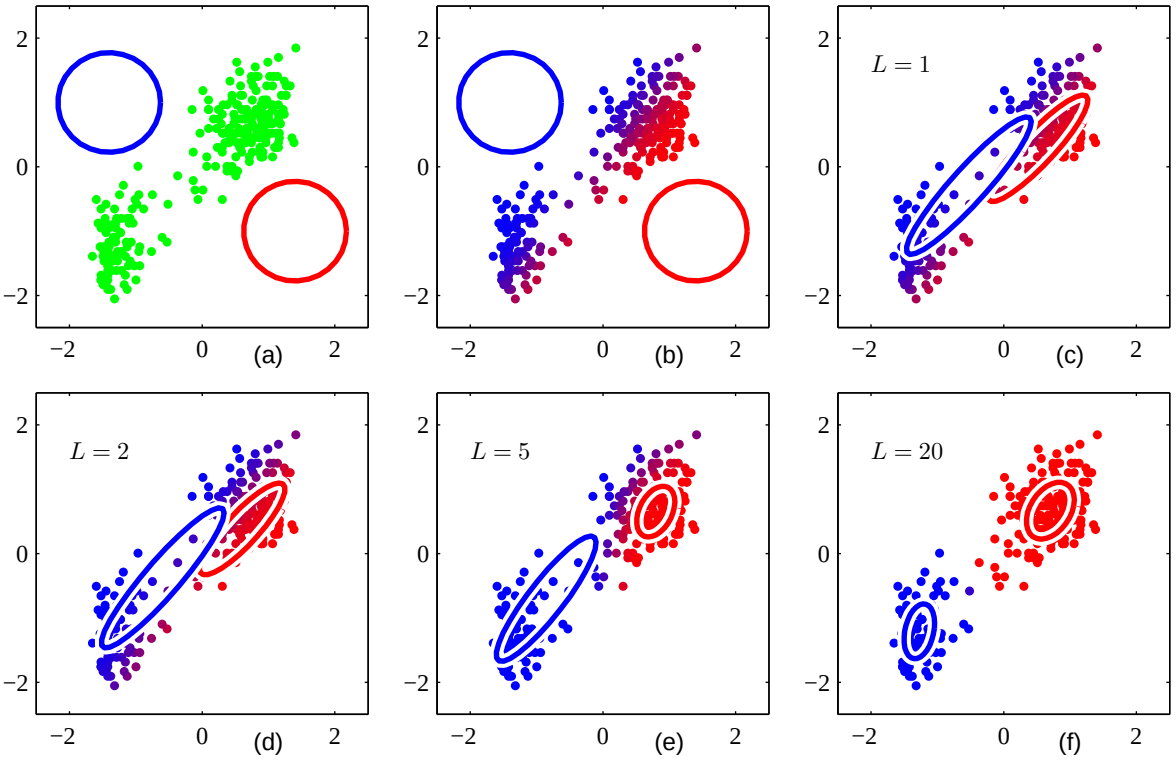
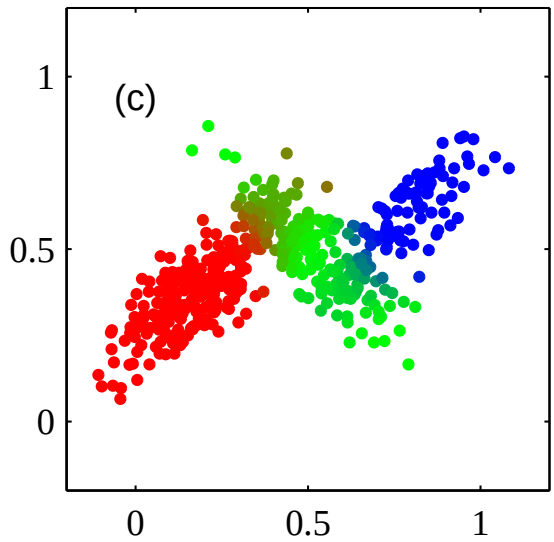


Figure 1: EM algorithm for GMM

Posterior distribution

We now show that $q_{kn}^{(t)}$ is the posterior distribution of the latent variable, i.e. $q_{kn}^{(t)} = p(z_n = k \mid \mathbf{x}_n, \boldsymbol{\theta}^{(t)})$

$$p(\mathbf{x}_n, z_n|\boldsymbol{\theta}) = p(\mathbf{x}_n|z_n, \boldsymbol{\theta})p(z_n|\boldsymbol{\theta}) = p(z_n|\mathbf{x}_n, \boldsymbol{\theta})p(\mathbf{x}_n|\boldsymbol{\theta})$$



EM in general

Given a general joint distribution $p(\mathbf{x}_n, z_n|\boldsymbol{\theta})$, the marginal likelihood can be lower bounded similarly:

The EM algorithm can be compactly written as follows:

$$\boldsymbol{\theta}^{(t+1)} := \arg\max_{\boldsymbol{\theta}} \sum_{n=1}^{N} \mathbb{E}_{p(z_n|\mathbf{x}_n,\boldsymbol{\theta}^{(t)})}\big[\log p(\mathbf{x}_n, z_n|\boldsymbol{\theta})\big]$$

Another interpretation is that part of the data is missing, i.e. $(\mathbf{x}_n, z_n)$ is the "complete" data and z_n is missing. The EM algorithm averages over the "unobserved" part of the data.