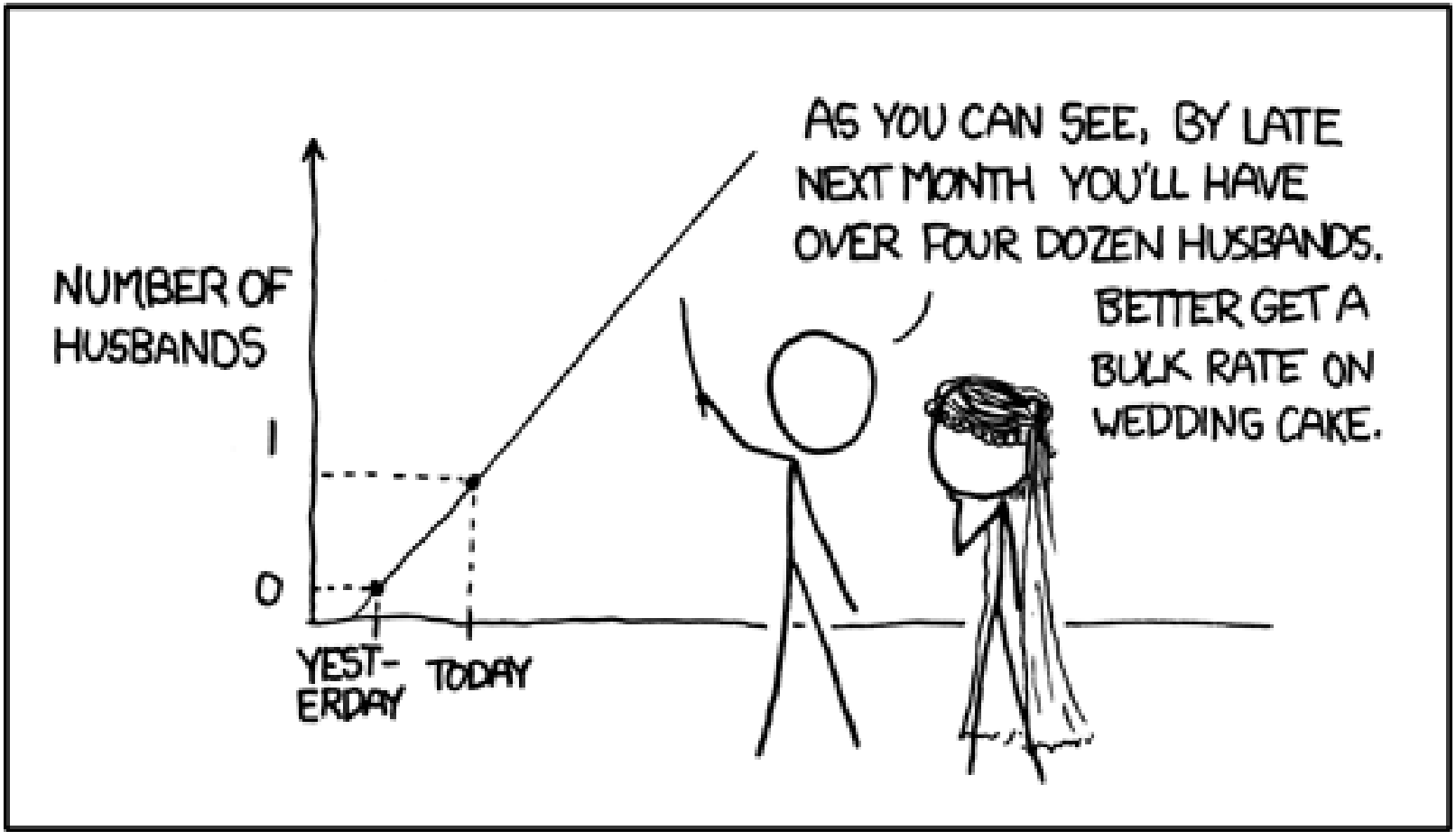
Advanced course in machine learning
582744
Lecture 2

Arto Klami

MY HOBBY: EXTRAPOLATING



Outline

Machine learning as optimization
- The learning problem
- Empirical risk minimization (Section 6)

Optimization (Sections 8.3, 8.5, 13.4)
- Convex functions
- Gradient descent
- Other descent methods

Machine learning process

- A *model* M describes data and typically has some unknown parameters $\boldsymbol{\theta}$
- A *data set* D is some collection of observations we want to model, often $D = \{\mathbf{x}_n, y_n\}_{n=1}^N$ or $D = \{\mathbf{x}_n\}_{n=1}^N$ ($\mathbf{x}$ is input, y is output)
- *Learning* or *model fitting* means choosing the parameters $\boldsymbol{\theta}$ based on the data D
- This is fundamentally an optimization problem: Some loss function $L(D, M(\boldsymbol{\theta}))$ needs to be minimized or maximized over the parameters $\boldsymbol{\theta}$
- Given the solution $\hat{\boldsymbol{\theta}}$ we can make predictions with the model: $p(\tilde{\mathbf{x}}|M, \hat{\boldsymbol{\theta}})$ or $p(\tilde{y}|\tilde{\mathbf{x}}, M, \hat{\boldsymbol{\theta}})$

Machine learning process: example

- Model: $p(y|\mathbf{x}, \boldsymbol{\theta}) = N(\boldsymbol{\theta}^T \mathbf{x}, 1)$
- Data set: $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ...\}$
- Loss function: $L(D, \boldsymbol{\theta}) = \frac{1}{N} \sum_n \|y_n - \boldsymbol{\theta}^T \mathbf{x}_i\|^2$
- Fit: $\hat{\boldsymbol{\theta}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}\mathbf{y}$
- Prediction: $p(y|\tilde{\mathbf{x}}, \hat{\boldsymbol{\theta}}) = N(\hat{\boldsymbol{\theta}}^T \tilde{\mathbf{x}}, 1)$

Loss functions

The loss function $L(D, M(\boldsymbol{\theta}))$

- Depends on the data D and the model/parameters $\boldsymbol{\theta}$
- We can always think of minimizing it; if the problem looks like maximization just put minus sign in front of it
- The loss function defines the goal of the learning task
- Examples:
  $L(y, f(\mathbf{x}, \boldsymbol{\theta})) = \|y - f(\mathbf{x}, \boldsymbol{\theta})\|^2$
  $L(y, f(\mathbf{x}, \boldsymbol{\theta})) = |y - f(\mathbf{x}, \boldsymbol{\theta})| \quad y \in [0, 1]$
  $L(y, f(\mathbf{x}, \boldsymbol{\theta})) = \log p(y|\mathbf{x}, \boldsymbol{\theta})$
  $L(\mathbf{x}, \boldsymbol{\theta}) = \log p(\mathbf{x}|\boldsymbol{\theta})$

Minimizing the loss is called *optimization*, and there is a whole field of science that studies that

The most important slide of the course

What are we really interested in?

- Training error: The loss for a given data set D
- Generalization error: The loss for a future data set $\tilde{D}$

The most important slide of the course

What are we really interested in?

- Training error: The loss for a given data set D
- Generalization error: The loss for a future data set $\tilde{D}$

Machine learning is about optimizing the generalization error!

The most important slide of the course

What are we really interested in?

- Training error: The loss for a given data set D
- Generalization error: The loss for a future data set $\tilde{D}$

Machine learning is about optimizing the generalization error!
...but while fitting the model we only have the given data

The most important slide of the course

What are we really interested in?

- Training error: The loss for a given data set D
- Generalization error: The loss for a future data set $\tilde{D}$

Machine learning is about optimizing the generalization error!
...but while fitting the model we only have the given data

If not for this discrepancy, the field of optimization would have already solved most of the problems

The most important slide of the course

What are we really interested in?

- Training error: The loss for a given data set D
- Generalization error: The loss for a future data set $\tilde{D}$

Machine learning is about optimizing the generalization error!
...but while fitting the model we only have the given data

If not for this discrepancy, the field of optimization would have already solved most of the problems

Dreaming of becoming a manager? Always ask "...but how well does it generalize?"

Why are they different?

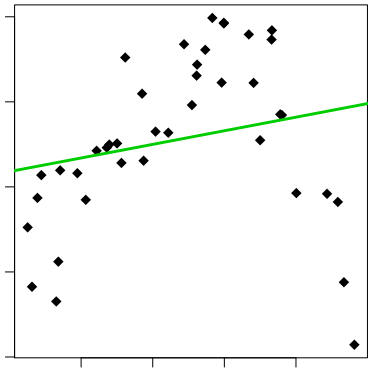
- The data D is a sample from an uncertain process (that is, it is a random variable)
- Another sample $\tilde{D}$ from the same process would require different parameters $\tilde{\boldsymbol{\theta}}$ to minimize the loss
- Hence: A finite data sample D does not uniquely determine the optimal solution
- Instead, we need to find parameters that are good for the underlying distribution $p(D)$ that generated the data

Why are they different?

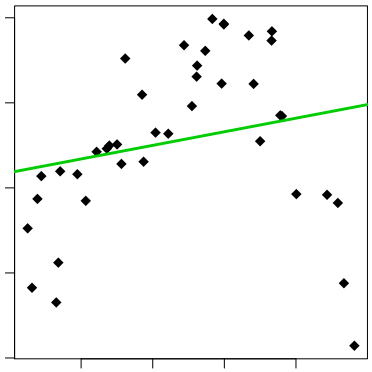
- The data D is a sample from an uncertain process (that is, it is a random variable)
- Another sample $\tilde{D}$ from the same process would require different parameters $\tilde{\boldsymbol{\theta}}$ to minimize the loss
- Hence: A finite data sample D does not uniquely determine the optimal solution
- Instead, we need to find parameters that are good for the underlying distribution $p(D)$ that generated the data

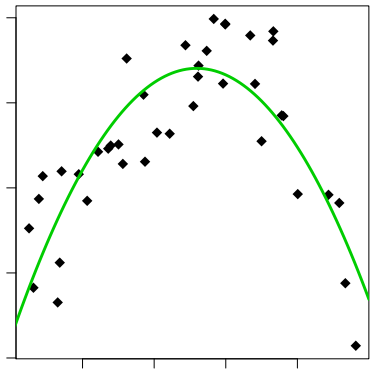
If D is not uncertain then you probably do not need ML

Illustration

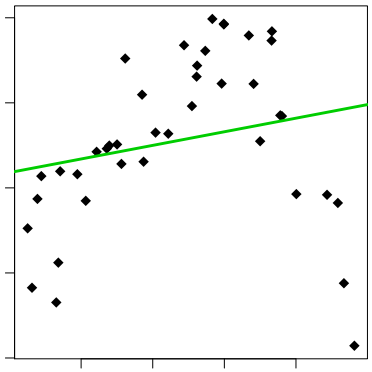


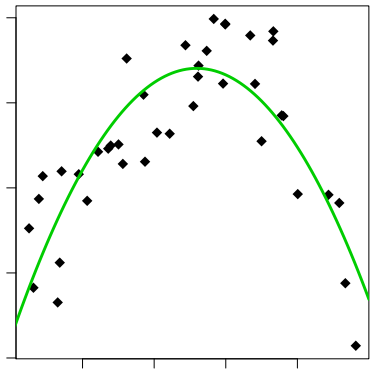
Illustration

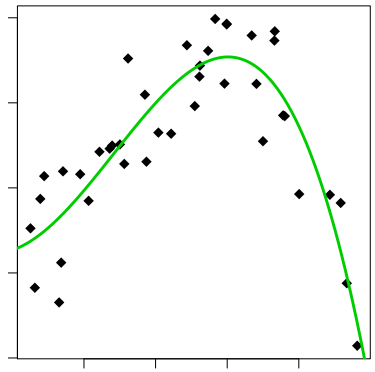




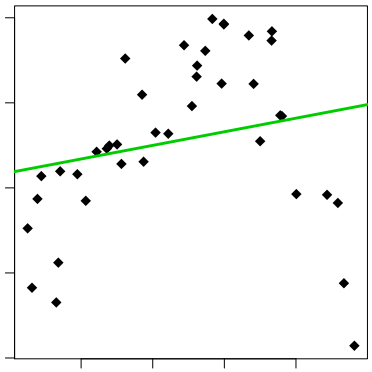
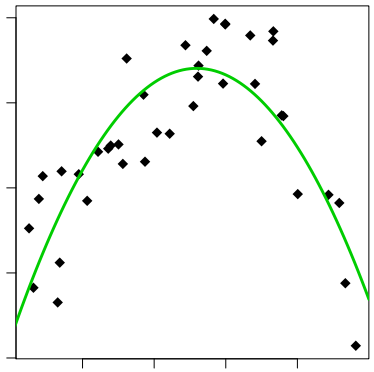
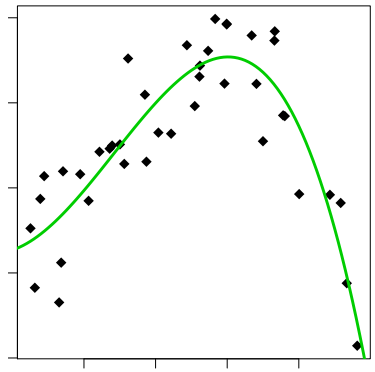
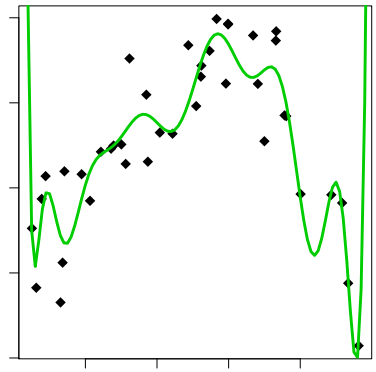
Illustration



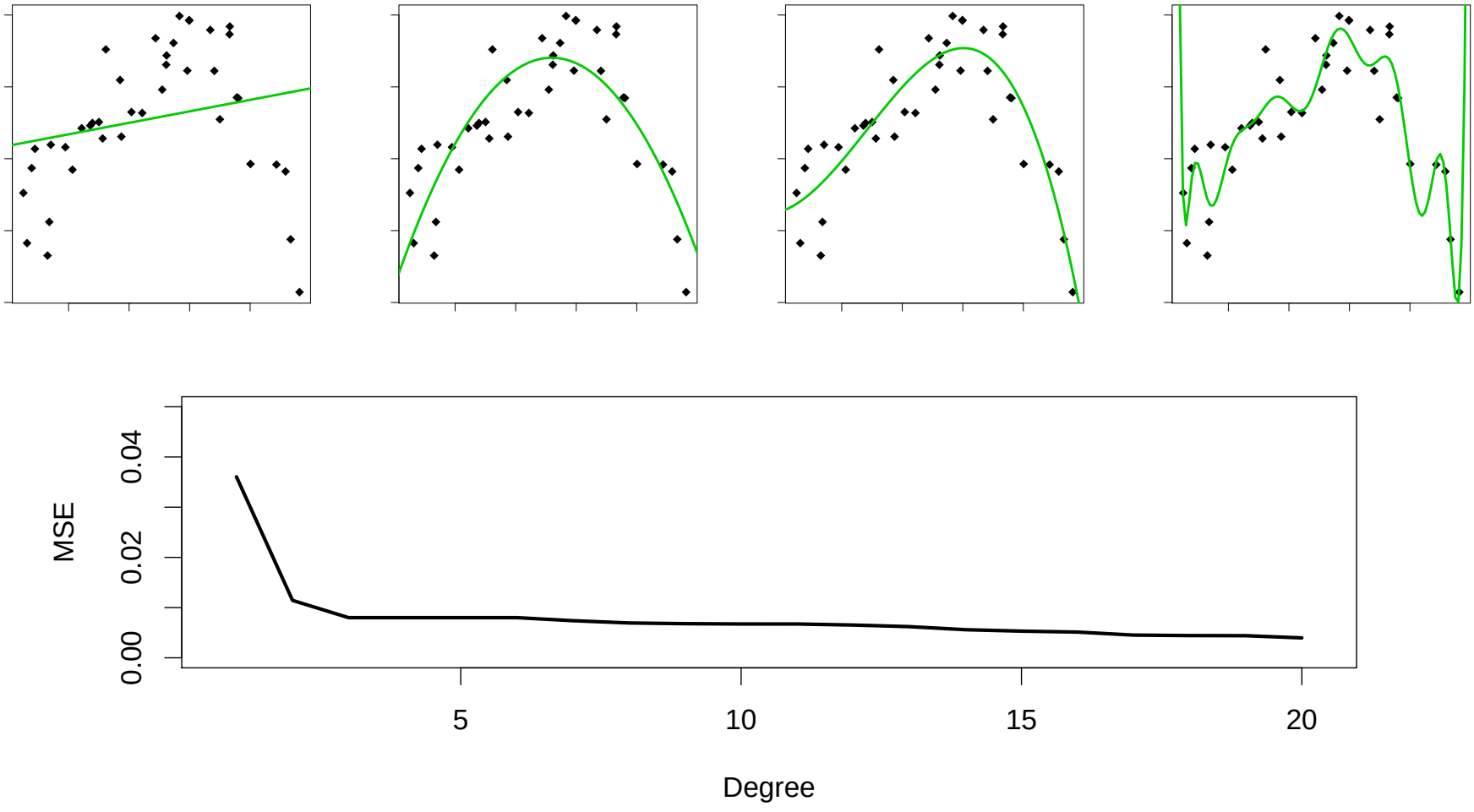




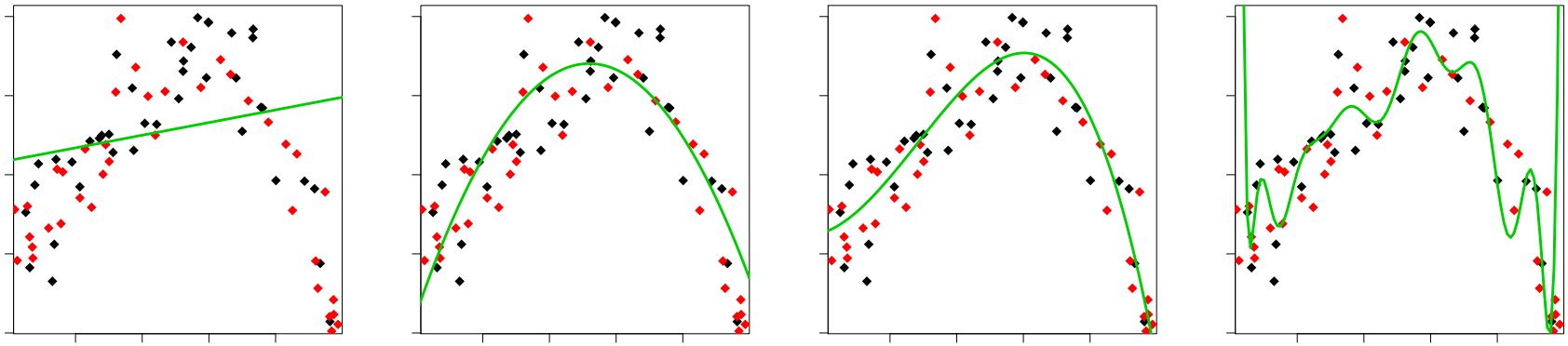
Illustration

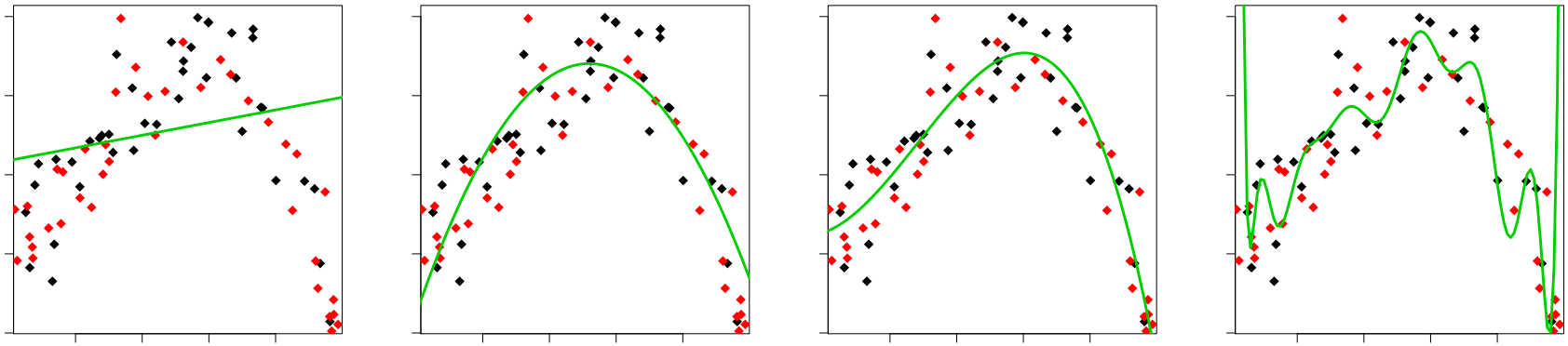
Illustration

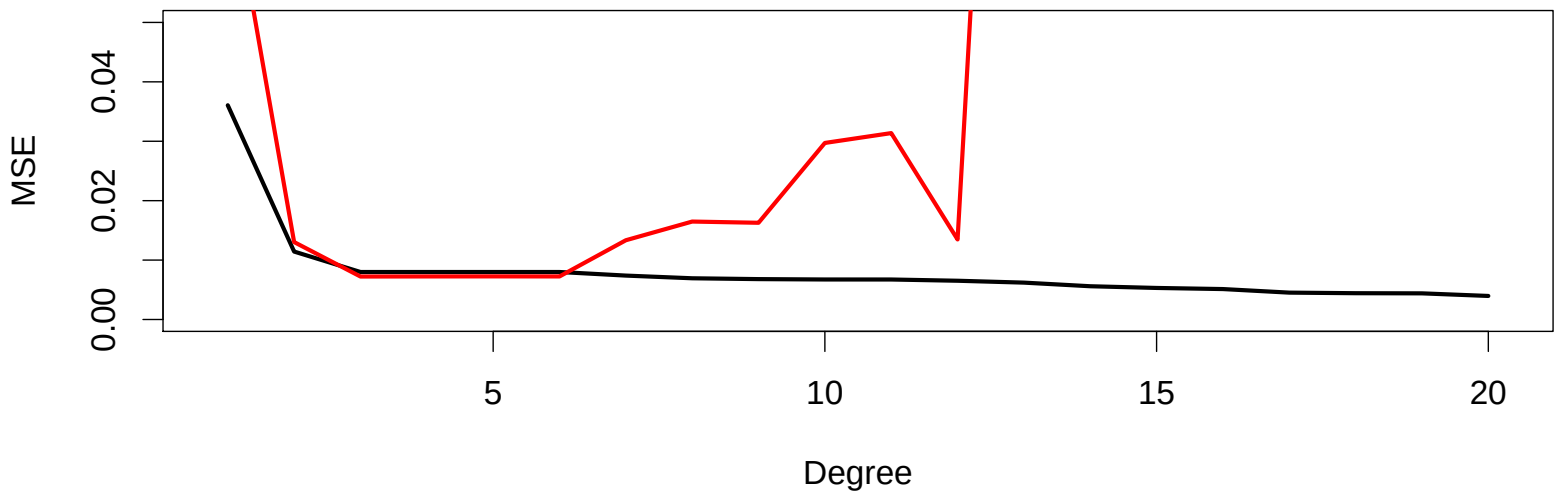


Illustration

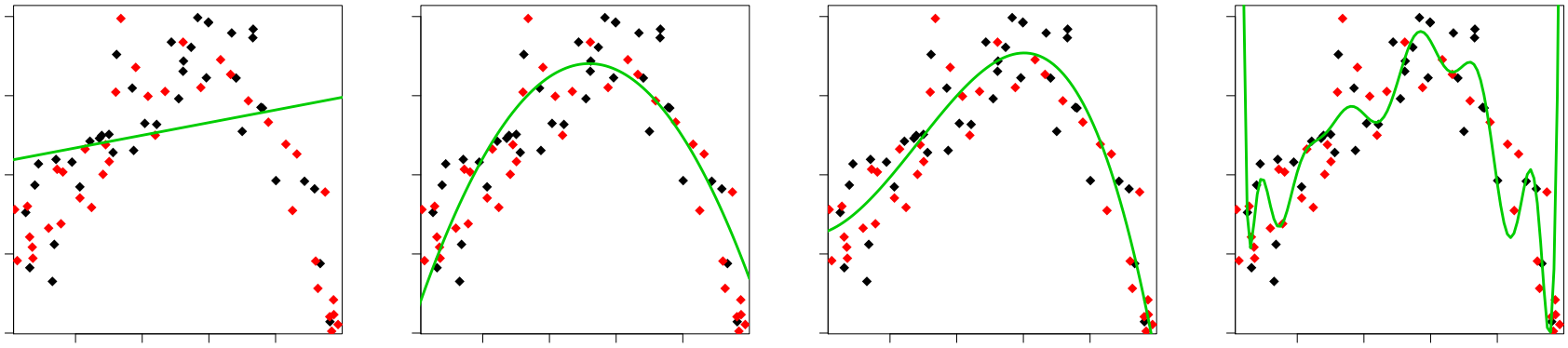


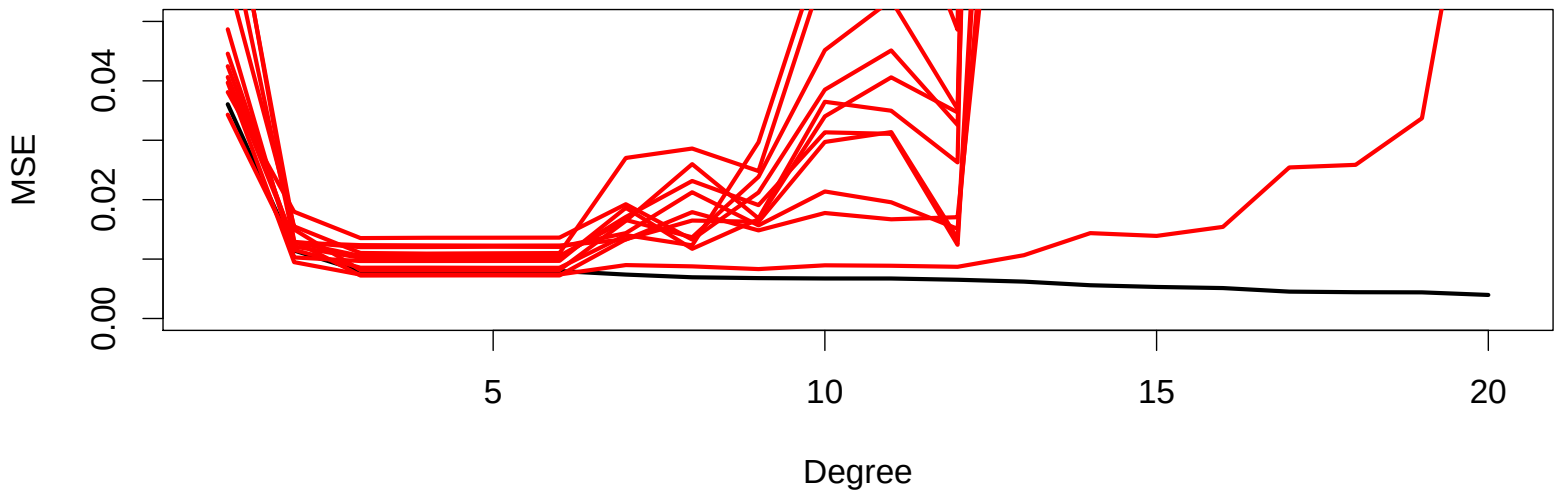
Illustration





Illustration





Approaching the generalization error

- Formal definitions
- How to estimate it?
- How to prevent it?

Risk: the expected loss

For simplicity of notation, let us consider classification problems where we output some y for each $\mathbf{x}$, and denote by $\delta(D)$ the classifier that makes the decisions

The *risk* is defined as the expected loss over the distribution generating the data points:

$$R(\delta) = \mathbb{E}_{p(y,\mathbf{x})}[L(y, \delta(\mathbf{x}))] = \int_{\mathbf{x},y} L(y, \delta(\mathbf{x}))p(y, \mathbf{x})d\mathbf{x}dy$$

Risk: the expected loss

For simplicity of notation, let us consider classification problems where we output some y for each $\mathbf{x}$, and denote by $\delta(D)$ the classifier that makes the decisions

The *risk* is defined as the expected loss over the distribution generating the data points:

$$R(\delta) = \mathbb{E}_{p(y,\mathbf{x})}[L(y, \delta(\mathbf{x}))] = \int_{\mathbf{x},y} L(y, \delta(\mathbf{x}))p(y, \mathbf{x})d\mathbf{x}dy$$

Simple, right? But we do not know $p(y, \mathbf{x})$...

Risk formulations

$$R(\delta) = \mathbb{E}_{p(y,\mathbf{x})}[L(y, \delta(\mathbf{x}))] = \int_{\mathbf{x},y} L(y, \delta(\mathbf{x}))p(y, \mathbf{x})d\mathbf{x}dy$$

Statistical learning theory:
Treat the data generating process as truly unknown, and try to somehow bound the risk, for example by considering the worst-case scenario

Risk formulations

$$R(\delta) = \mathbb{E}_{p(y,\mathbf{x})}[L(y, \delta(\mathbf{x}))] = \int_{\mathbf{x},y} L(y, \delta(\mathbf{x}))p(y, \mathbf{x})d\mathbf{x}dy$$

Statistical learning theory:
Treat the data generating process as truly unknown, and try to somehow bound the risk, for example by considering the worst-case scenario

Bayesian approach:
Assume we know the correct model but are just unsure of the parameters. Then we can average over the parameters conditional on the data:
$R(\delta) = \mathbb{E}_{p(\boldsymbol{\theta}|D)}[L(y, \delta(\mathbf{x}))] = \int_{\boldsymbol{\theta}} L(y, \delta(\mathbf{x}))p(y, \mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta}|D)d\boldsymbol{\theta}$

Risk formulations

$$R(\delta) = \mathbb{E}_{p(y,\mathbf{x})}[L(y, \delta(\mathbf{x}))] = \int_{\mathbf{x},y} L(y, \delta(\mathbf{x}))p(y, \mathbf{x})d\mathbf{x}dy$$

Statistical learning theory:
Treat the data generating process as truly unknown, and try to somehow bound the risk, for example by considering the worst-case scenario

Bayesian approach:
Assume we know the correct model but are just unsure of the parameters. Then we can average over the parameters conditional on the data:
$$R(\delta) = \mathbb{E}_{p(\boldsymbol{\theta}|D)}[L(y, \delta(\mathbf{x}))] = \int_{\boldsymbol{\theta}} L(y, \delta(\mathbf{x}))p(y, \mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta}|D)d\boldsymbol{\theta}$$

The book is largely based on the latter, but does not very clearly state its limitation: The expectation is wrong if the model is wrong

Empirical risk minimization

$$R(\delta) = \mathbb{E}_{p(y,\mathbf{x})}[L(y, \delta(\mathbf{x}))] = \int_{\mathbf{x},y} L(y, \delta(\mathbf{x}))p(y, \mathbf{x})d\mathbf{x}dy$$

The training error is called "empirical risk" and simply plugs in the observations; $1/N$ weight for each of the N training samples: $R_e \approx \frac{1}{N}\sum_n L(y_n, \delta(\mathbf{x}_n))$

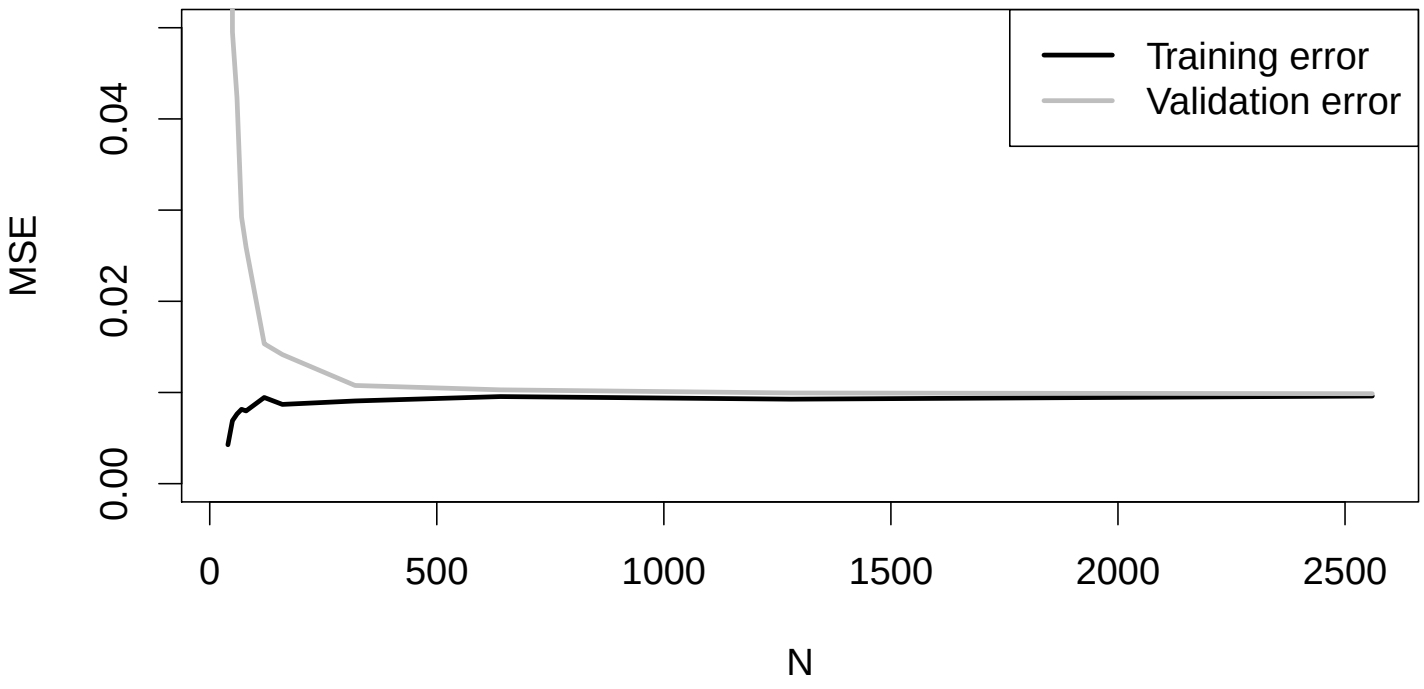
A *consistent estimator* gives the right solution by minimizing the empirical risk when $N \to \infty$

Estimating the risk

Instead of using the empirical error on the training data, we can get an unbiased (but typically high-variance) estimate on the expected risk by looking at the error on data samples not used for training

$R_v \approx \frac{1}{M} \sum_m L(y_m, \delta(\mathbf{x}_m))$ for $\{(x_m, y_m)\}$ not used in training

Consistency



Cross-validation error

A practical way of getting a lower variance estimate

- Split the data randomly into K folds
- Estimate θ based on samples in $K-1$ folds
- Evaluate the error on the remaining one
- Estimate the risk as $\frac{1}{K} R_{v(k)}$, where $R_{v(k)}$ is the empirical risk for the kth fold

Leave-one-out validation: $K = N$, always leaving out one sample at a time

Risk minimization

Now we can recognize models with high risk, but how do we avoid it?

Three standard ways:

- Early-stopping: Keep on optimizing only as long as the estimated risk goes down
- Regularize the loss function: Modified optimization problem that is less likely to overfit
- Average predictions over multiple models

Regularized risk minimization

Idea: Limiting the complexity should help avoiding overfitting

Optimize $R(\delta) + \lambda C(\delta)$ instead, where $C(\delta)$ somehow measures the complexity

We also need to set λ to control the amount of regularization. This is typically done by cross-validation; we estimate the risk for each possible choice of λ and pick the best:

$$\hat{\lambda} = \text{argmin}\, R_v(\delta_\lambda)$$

Statistical learning theory

(One) formal justification for regularization comes from *statistical learning theory*: Prove an upper bound for the risk over all possible data generating distributions $p(D)$

For binary classification, Hoeffding's inequality states that
$p(\max_\delta |R_e(\delta) - R(\delta)| > \epsilon) \leq 2|\mathcal{H}|e^{-2N\epsilon^2}$
where $|\mathcal{H}|$ is the size of the hypothesis space

Makes sense: Larger N reduces the bound, and larger hypothesis space means the classifier is more flexible – suggests that restricting the classifier with regularization is a good idea

Statistical learning theory

What is $|\mathcal{H}|$?

- The number of possible models in a finite set
- ...but usually we have infinitely many
- Vapnik-Chevronenkis (VC) dimension: The maximum number of points that can be arranged so that we make no mistakes in binary classification for any set of labels given for the points
- *Probably appxorimately correct* (PAC) if we can find a function that has low empirical risk and the hypothesis space is small
- In practice: Computing VC dimension is hard and the bound can still be loose

Statistical learning theory

What is $|\mathcal{H}|$?

- The number of possible models in a finite set
- ...but usually we have infinitely many
- Vapnik-Chevronenkis (VC) dimension: The maximum number of points that can be arranged so that we make no mistakes in binary classification for any set of labels given for the points
- *Probably appxorimately correct* (PAC) if we can find a function that has low empirical risk and the hypothesis space is small
- In practice: Computing VC dimension is hard and the bound can still be loose

Just remember the intuition: If the hypothesis space is small and we have large N, the empirical risk is probably quite close to the true risk. If not, we probably underestimated the risk.

Typical regularizers

Most regularizers control the norm of the parameters:

- ℓ_2: The squared norm
- ℓ_1: The sum of the absolute elements
- ℓ_∞: The largest element

...while some attempt to directly measure the complexity:

- Bayesian/Akaike information criterion: degrees of freedom
- Minimum description length (MDL): See *Information theoretic modeling*

Typical regularizers

Most regularizers control the norm of the parameters:

- ℓ_2: The squared norm
- ℓ_1: The sum of the absolute elements
- ℓ_∞: The largest element

...while some attempt to directly measure the complexity:

- Bayesian/Akaike information criterion: degrees of freedom
- Minimum description length (MDL): See *Information theoretic modeling*

We will look at practical regularizers when we start talking about actual models

Bootstrap

Instead of searching for a single optimal $\boldsymbol{\theta}$, we can accept the fact that $\boldsymbol{\theta}$ depends on the data. For any given data set D_b we learn $\boldsymbol{\theta}_b$ and when making predictions we use averages $\frac{1}{B} f(\mathbf{x}, \boldsymbol{\theta}_b)$

Ideally we would draw the data sets D_b from the true unknown process, but in practice we create fake sets by re-sampling from the current set: Pick N samples with replacement

Optimization

ML is about solving the optimization problem $L(D, \boldsymbol{\theta}) + R(\boldsymbol{\theta})$
...so we need some optimization tools

Two kinds of problems: Convex and non-convex
...or "easy" and hard

Constrained or un-constrained; we mostly consider the former on this course

Convexity

- A convex function is a happy function
- Multiple definitions: secants are above the curve, the second derivatives are non-negative
- Strongly convex has positive second derivatives (above some ϵ)
- If $f(\cdot)$ and $g(\cdot)$ are convex then $af(\cdot) + bg(\cdot)$ is convex if $a, b \geq 0$
- ...and $\max f(\cdot) + g(\cdot)$ is convex
- ...and $g(f(\cdot))$ is convex if $g(\cdot)$ is also non-decreasing (think of $\exp(\cdot)$)
- Smooth (differentiable) vs non-smooth: Things are always easier for the former

Convex functions?

Which of these are convex?

- $f(x) = a * x + b$
- $f(x) = \exp(x)$
- $f(x) = \log(x)$
- $f(x) = \sin(x) \quad x \in [\pi, 2\pi]$

Convex losses in ML

- Least squares, many regularizers
- Convexified losses: Hinge loss instead of binary classification error
- Smooth functions in local neighborhoods of positive curvature; think of Taylor expansion
- Parts of more complex cost functions: often the cost is convex with respect to some parameters if the others are kept constant
- Very often in the form $\sum_n f(\mathbf{x}_n|\boldsymbol{\theta}) + g(\boldsymbol{\theta})$, where $f(\cdot)$ and $g(\cdot)$ are convex

Convex optimization

Convex optimization problems are kind of easy

- The gradient points towards better solutions
- Convex optimization studies the convergence rates (and other theoretical properties) of different kinds of algorithms
- The ML community can cherry-pick the most robust techniques

The gradient

$$\nabla L(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial L(\boldsymbol{\theta})}{\partial \theta_1} \\ \vdots \\ \frac{\partial L(\boldsymbol{\theta})}{\partial \theta_k} \end{bmatrix}$$

- High-school math: The gradient is zero at the optimum
- Already solves for example least-squares regression, but for most problems we cannot find the optimum analytically
- Instead, we need iterative algorithms

Gradient descent

We can minimize any convex function $g(\boldsymbol{\theta})$ with *gradient descent*:

- Start with some initial guess $\boldsymbol{\theta}_0$
- Repeat until convergence:
  1. Evaluate the gradient $\nabla g(\boldsymbol{\theta})$ at the current $\boldsymbol{\theta}_t$
  2. Update the parameters by moving a bit against the gradient direction: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla g(\boldsymbol{\theta})$

Gradient descent

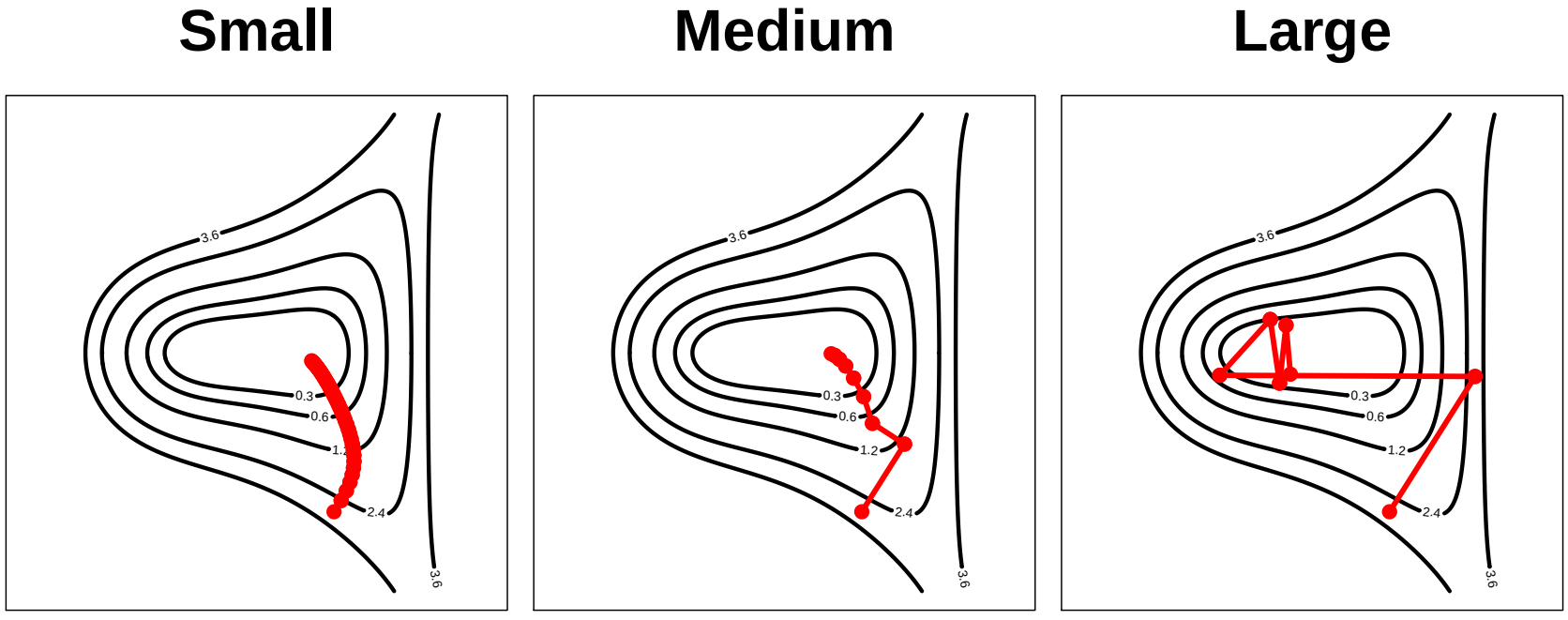
We can minimize any convex function $g(\boldsymbol{\theta})$ with *gradient descent*:

- Start with some initial guess $\boldsymbol{\theta}_0$
- Repeat until convergence:
  1. Evaluate the gradient $\nabla g(\boldsymbol{\theta})$ at the current $\boldsymbol{\theta}_t$
  2. Update the parameters by moving a bit against the gradient direction: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla g(\boldsymbol{\theta})$

The step-size α controls the magnitude of the updates

- Small α always works, but can be slow
- Large α can lead to oscillation

Gradient descent



Gradient descent: Step size

- Fixed value: Small is slow, large overshoots
- Line search: Accurate but slow, often not worth it
- Backtracking: Try with large value, divide by two if the cost did not decrease (enough) $f(\boldsymbol{\theta} - \alpha \nabla f) \leq f(\boldsymbol{\theta}) - \beta \alpha \nabla f$
- Adaptive: Modify the previous length by some rules, typically increasing it if the cost dropped and reverting back to some small value if we overshot

Gradient descent for risk minimization

The risk is defined as expectation of the loss over a data generating distribution: $R(\boldsymbol{\theta}) = \mathbb{E}_{p(y,\mathbf{x})}[L(y, f(\mathbf{x}, \boldsymbol{\theta}))]$

To apply gradient descent we need to take gradients of that expectation

Gradient descent for risk minimization

The risk is defined as expectation of the loss over a data generating distribution: $R(\boldsymbol{\theta}) = \mathbb{E}_{p(y,\mathbf{x})}[L(y, f(\mathbf{x}, \boldsymbol{\theta}))]$

To apply gradient descent we need to take gradients of that expectation

Not as daunting as it sounds; expectation is a linear operator and we can exchange the order of integration and differentiation: $\nabla \mathbb{E}[L(y, f(\mathbf{x}, \boldsymbol{\theta}))] = \mathbb{E}[\nabla L(y, f(\mathbf{x}, \boldsymbol{\theta}))]$

Gradient descent for risk minimization

The risk is defined as expectation of the loss over a data generating distribution: $R(\boldsymbol{\theta}) = \mathbb{E}_{p(y,\mathbf{x})}[L(y, f(\mathbf{x}, \boldsymbol{\theta}))]$

To apply gradient descent we need to take gradients of that expectation

Not as daunting as it sounds; expectation is a linear operator and we can exchange the order of integration and differentiation: $\nabla\mathbb{E}[L(y, f(\mathbf{x}, \boldsymbol{\theta}))] = \mathbb{E}[\nabla L(y, f(\mathbf{x}, \boldsymbol{\theta}))]$

...but we still need to compute the expectation of the gradient itself

Monte Carlo approximation

Luckily expectations can be estimated with Monte Carlo approximation: Draw samples $y, \mathbf{x} \sim p(y, \mathbf{x})$ and replace the expectation with sum

$$\mathbb{E}_{p(y,\mathbf{x})}[\nabla L(y, f(\mathbf{x}, \boldsymbol{\theta}))] \approx \frac{1}{M} \sum_{m=1}^{M} \nabla L(y_m, f(\mathbf{x}_m, \boldsymbol{\theta}))$$

Provides noisy but unbiased estimates of the actual gradient

- Already $M = 1$ samples are enough
- In practice we use the empirical estimate $p(y, \mathbf{x}) \approx \frac{1}{N} \delta(y_n, \mathbf{x}_n)$, picking individual training examples

Using such stochastic estimates in gradient descent is called *stochastic gradient descent*

Alternative derivation

Start with the empirical risk $R_e(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{n=1}^{N} L(y_n, f(\mathbf{x}_n, \boldsymbol{\theta}))$

Compute the gradient as $\nabla R_e(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{n=1}^{N} \nabla L(y_n, f(\mathbf{x}_n, \boldsymbol{\theta}))$

This is called the *batch gradient*; it is exact gradient for the empirical risk

Computing the exact gradient has $O(N)$ complexity, so we might want to approximate it with using only a subset of the data points

...which gives us SGD

Stochastic gradient descent

The real gradient is zero at the minimum, so we know when to stop

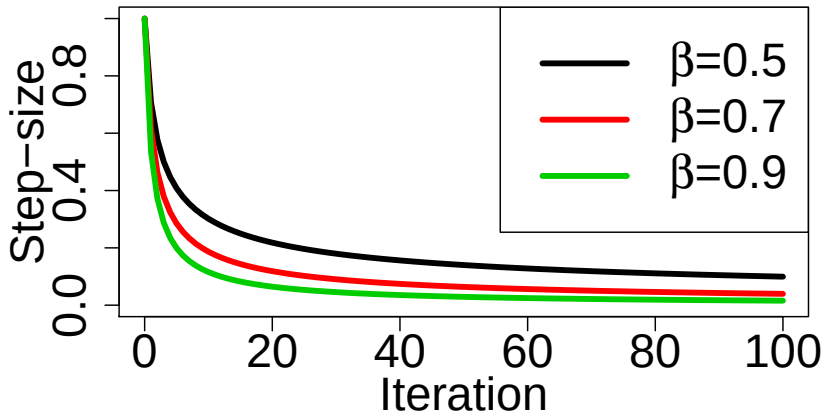
However, its stochastic estimate will typically not be zero

To make the algorithm converge, we need to make the step-size smaller during optimization

Not too fast and not too slow: $\sum_{t=1}^{\infty} \alpha_t = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2 \leq \infty$ (Robbins-Monro conditions)

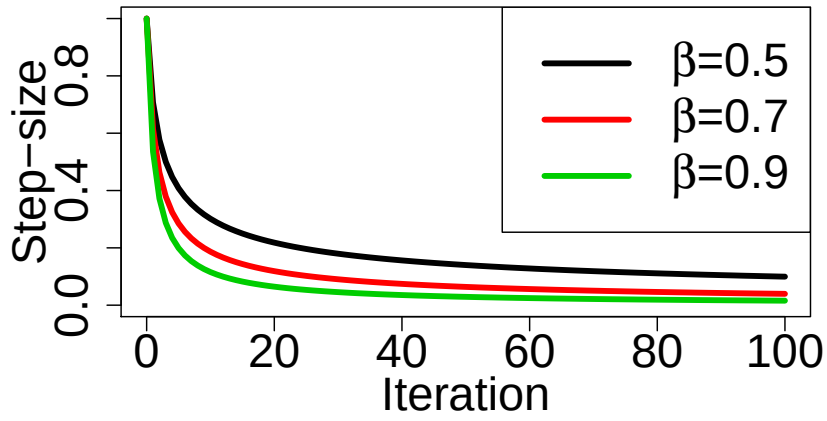
Stochastic gradient descent in practice

For example $\alpha_t = (\alpha_0 + t)^{-\beta}$ for for $\beta \in (0.5, 1]$ satisfies the R-M conditions



Stochastic gradient descent in practice

For example $\alpha_t = (\alpha_0 + t)^{-\beta}$ for for $\beta \in (0.5, 1]$ satisfies the R-M conditions



In practice you should use some adaptive rate instead

- Adagrad: Per-parameter step-size scaled by $1/\sqrt{\sum_t g_t^2}$
- Adam, Adadelta and RMSProp are other similar techniques; it is enough for you to know one of these

Stochastic gradient descent in practice

The basic theory of SGD works already for $M = 1$, but in practice one should use bigger subset (*mini-batch*) of the data

SGD solves many of the 'big data' challenges:

- The complexity of the algorithm does not depend on the total amount of data, but only on M
- Most deep learning success stories use SGD with adaptive step-sizes
- SGD often works well even when the loss is not convex

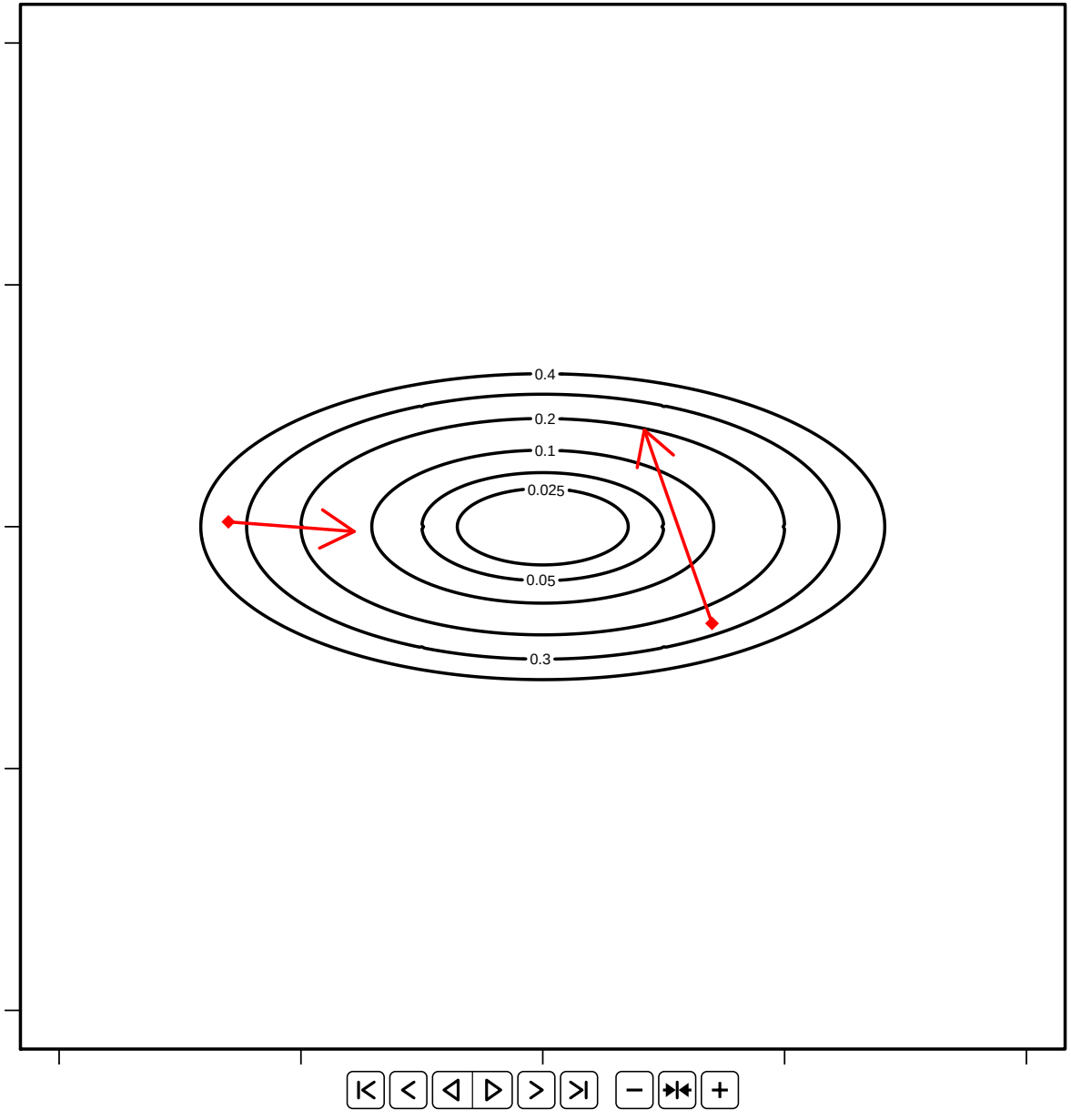
If you only learn one optimization algorithm, it should be SGD

Can we do better?

Gradient is an intuitive direction because it points towards the steepest descent. However, it need not be optimal

Instead of gradients, we can go towards any direction $\mathbf{d}$ that decreases the cost ($\mathbf{d}^T \nabla L < 0$). Often we can find better directions than the gradient

Geometry example



Gradient descent: second-order

Newton's method: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{H}(\boldsymbol{\theta})^{-1}\nabla f(\boldsymbol{\theta})$, where $H_{i,j} = \frac{\partial^2 H}{\partial \theta_i \partial \theta_j}$

- Fast but requires quite a bit of computation; we need the second derivatives and we have to invert the Hessian
- Can also be fraqile
- Does the geometric transformation locally
- Quasi-Newton methods: BFGS etc approximate the inverse of the Hessian based on the gradients, requiring less computation and memory – standard optimization libraries usually use these by default

Conjugate gradients

- Conjugate gradient algorithms modify the gradient direction based on the previous gradients
- Optimal for quadratic functions $\boldsymbol{\theta}^T \mathbf{A} \boldsymbol{\theta}$, converges in D steps
- Requires (sufficiently) exact line-search

Coordinate descent

Pick one dimension at a time and perform one-dimensional optimization in that direction
$\theta_i = \arg\min f(\boldsymbol{\theta} + \alpha \mathbf{e}_i)$

Can be useful with non-smooth functions, and is in general surprisingly efficient

Non-convex optimization

- The problem: We have multiple local optima and might miss the global optimal solution
- The cheap way: Just use convex optimization techniques and hope for the best
- ...and perhaps buy a few more lottery tickets by trying again with random initializations
- Momentum in gradient descent, simulated annealing, genetic algorithms, convexified loss functions, ...

On this course, we are happy with convex optimization techniques, but will frequently use them only for subproblems

Constrained optimization

- A big field in itself
- We only need two things: (1) Projecting back to the constrained set and (2) Lagrange multipliers
- Projecting back: Perform some gradient-based update, then find the closest solution that satisfies the constraints

Lagrange multipliers

- Given a loss $L(\boldsymbol{\theta})$ and a constraint $f(\boldsymbol{\theta}) = C$, we can use *Lagrange functions* to recognize optima
- Intuition: Along the curve of $f(\boldsymbol{\theta}) = C$ we still need to find the optimum of the loss, which is recognized as the gradient being perpendicular to the constraint: $\nabla L(\boldsymbol{\theta}) = \lambda \nabla f(\boldsymbol{\theta})$
- Augment the loss as $L' = L(\boldsymbol{\theta}) - \lambda(f(\boldsymbol{\theta}) - C)$ and find $L' = 0$

Non-smooth functions

What if we have no derivative at some point?

- Instead, we have a set of *subgradients*; the set of all functions $g(x)$ that touch the function $f(x)$ at the point of differentiation and are otherwise touching or below it
- If the zero vector belongs to that set then we are at a local optimum
- Proximal methods: Regularize the problem around the point where we have no gradient

ML definitions

- "Field of study that gives computers the ability to learn without being explicitly programmed" (A. Samuel, 1959)
- "...a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty" (K. Murphy)
- ML is optimization for loss functions that are expectations over unknown data generating processes (A. Klami)