Regularization

Lecture series „Machine Learning“

Niels Landwehr

Research Group „Data Science“
Institute of Computer Science
University of Hildesheim

Agenda For Lecture

- Training and test error
- Model complexity and overfitting
- Model selection and regularization

Agenda For Lecture

- Training and test error
- Model complexity and overfitting
- Model selection and regularization

Review: Supervised Learning

- Review: in **supervised learning**, the goal is to make predictions about objects

Input: an **instance** (object) $\mathbf{x}$ that lives in an instance space $\mathcal{X}$

$$\mathbf{x} \mapsto y$$
$$\mathbf{x} \in \mathcal{X} \qquad y \in \mathcal{Y}$$

Output: a **label** or **target** y that lives in a target space $\mathcal{Y}$

- To obtain predictions, we are looking for a **model** f that produces a prediction $f(\mathbf{x}) \in \mathcal{Y}$ for an input instance $\mathbf{x}$

$$f : \mathcal{X} \to \mathcal{Y}$$

Input: instance $\mathbf{x}$

$$\mathbf{x} \mapsto f(\mathbf{x})$$

Output: prediction $f(\mathbf{x})$

- Model will be inferred from **training data**: a set of instances with observed targets

$$\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$$

Training instances $\mathbf{x}_n \in \mathcal{X}$: observed objects in training data, for example flowers, images of digits, or emails

Observed labels or targets $y \in \mathcal{Y}$ in training data, for example classes of flowers, digits 0...9, or spam/legitimate classifications

Review: Assumptions About Data

- For learning to work, we have to assume that there is some reasonably stable relationship between inputs and outputs that can be captured by a model

- Assumption: training example are independently drawn from (constant) joint distribution over inputs and outputs:

$$(\mathbf{x}_n, y_n) \sim p(\mathbf{x}, y)$$

- Because $p(\mathbf{x}, y) = p(\mathbf{x})p(y \mid \mathbf{x})$, the assumption can be reformulated as

$$\mathbf{x}_n \sim p(\mathbf{x})$$
$$y_n \sim p(y \mid \mathbf{x}_n)$$

- The instances $\mathbf{x}_n$ are sampled from a probability distribution over instances.
- $p(\mathbf{x})$ describes distribution over population of objects
- For example, certain flowers, digits, or email texts are encountered with a certain probability

- Given an instance $\mathbf{x}_n$, its label is drawn from a distribution $p(y \mid \mathbf{x}_n)$ that represents the relationship between input and output.
- The relationship could be deterministic (probabilities 0 or 1) but this formulation also allows for randomness or noise in data

Goal: Low Error at Application Time

- Model f is often a parameterized function $f_{\boldsymbol{\theta}}$, and its parameters are learned by minimizing a loss function on the training data:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$$

$$L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N} \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n)$$

- Alternatively, can maximize a „gain function“ such as likelihood on training data

- **However, the eventual goal of learning is not to perform well on training data**
  - After training, the model will be deployed in an application domain
  - After deployment, the model has to make predictions for novel instances which have not been part of the training set
  - These are assumed to be drawn from the same distribution as the training data
  - The goal of learning is to obtain a model that performs well on these novel instances, not the training instances

Test Error

- The predictive performance on novel instances at application time can be defined by the expected loss or error of the model on a new, randomly drawn instance
  - Assume we are drawing a novel instance (not part of training data) with corresponding label

$$\mathbf{x}_{new} \sim p(\mathbf{x})$$
$$y_{new} \sim p(y \mid \mathbf{x}_{new})$$

  - Given a function ℓ_{eval} that measures the loss or error between the true label y_{new} and prediction $f_{\boldsymbol{\theta}}(\mathbf{x}_{new})$, we can measure

$$\ell_{eval}(y_{new}, f_{\boldsymbol{\theta}}(\mathbf{x}_{new})) \in \mathbb{R}$$

  - The expectation of this quantity is obtained by integrating over $\mathbf{x}$ and y :

$$R(f_{\boldsymbol{\theta}}) = \mathbb{E}\left[\ell_{eval}(y, f_{\boldsymbol{\theta}}(\mathbf{x}))\right] = \iint \ell_{eval}(y, f_{\boldsymbol{\theta}}(\mathbf{x}))\, p(\mathbf{x}, y)\, d\mathbf{x}\, dy$$

- The quantity $R(f_{\boldsymbol{\theta}})$ is also called the expected risk or expected test error of the model
- Goal is to find a model with low $R(f_{\boldsymbol{\theta}})$

Loss or Error Function On Test Data

- How do we choose the evaluation loss ℓ_{eval}?
- For regression:
  - often squared loss $\ell_{eval}(y, f_{\boldsymbol{\theta}}(\mathbf{x})) = (y - f_{\boldsymbol{\theta}}(\mathbf{x}))^2$
  - or absolute loss $\ell_{eval}(y, f_{\boldsymbol{\theta}}(\mathbf{x})) = | y - f_{\boldsymbol{\theta}}(\mathbf{x}) |$
- For classification:
  - most widely used is classification error:

$$\ell_{eval}(y, f_{\boldsymbol{\theta}}(\mathbf{x})) = \begin{cases} 0 : y = f_{\boldsymbol{\theta}}(\mathbf{x}) \\ 1 : y \neq f_{\boldsymbol{\theta}}(\mathbf{x}) \end{cases}$$

Expectation over this function: fraction of cases in which the model makes an error

  - other options include likelihood as in training (higher is better) or application-specific losses

Training Versus Test Error

- To estimate the expected risk of a model on novel instances, we can use a **test set** of instances also drawn from the joint distribution $p(\mathbf{x}, y)$:

$$\mathcal{D}_{test} = \{(\overline{\mathbf{x}}_1, \overline{y}_1), \ldots, (\overline{\mathbf{x}}_{\overline{N}}, \overline{y}_{\overline{N}}) \qquad (\overline{\mathbf{x}}, \overline{y}) \sim p(\mathbf{x}, y)$$

- The expected risk can then be approximated by

$$\hat{R}_{test}(f_{\boldsymbol{\theta}}) = \frac{1}{\overline{N}} \sum_{n=1}^{\overline{N}} \ell_{eval}(\overline{y}_n, f_{\boldsymbol{\theta}}(\overline{\mathbf{x}}_n))$$

Unbiased estimator: will approximate $R(f_{\boldsymbol{\theta}})$ well given large enough $\overline{N}$ (more details in next lecture)

- We can also measure the average risk of the model on the training data:

$$\hat{R}_{train}(f_{\boldsymbol{\theta}}) = \frac{1}{N} \sum_{n=1}^{N} \ell_{eval}(y_n, f_{\boldsymbol{\theta}}(\mathbf{x}_n))$$

- Depending on the choice of ℓ_{eval} , $\hat{R}_{train}(f_{\boldsymbol{\theta}})$ is similar (or identical) to the loss function used for training,

$$L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n)$$

Agenda For Lecture

- Training and test error
- Model complexity and overfitting
- Model selection and regularization

Training Versus Test Error

- During learning, we choose the model only based on the training data
- The chosen model $f_{\boldsymbol{\theta}^*}$ will typically have a low $\hat{R}_{train}(f_{\boldsymbol{\theta}^*})$
- If everything goes well,
  - The trained model $f_{\boldsymbol{\theta}^*}$ has picked up the $(\mathbf{x}, y)$-relationship that stems from the joint distribution $p(\mathbf{x}, y)$
  - This $(\mathbf{x}, y)$-relationship then also holds on new data, therefore the model $f_{\boldsymbol{\theta}^*}$ will also perform well on new data ($R(f_{\boldsymbol{\theta}^*})$ and $\hat{R}_{test}(f_{\boldsymbol{\theta}^*})$ will also be low)

- However, a low training error $\hat{R}_{train}(f_{\boldsymbol{\theta}^*})$ does not guarantee a low test error $\hat{R}_{test}(f_{\boldsymbol{\theta}^*})$
- If things go not so well,
  - The trained model $f_{\boldsymbol{\theta}^*}$ will perform well on the training data not because it has picked up an $(\mathbf{x}, y)$-relationship that will transfer to new data, but because it has picked up spurious, random patterns that are specific to the particular training set
  - In this case, $\hat{R}_{train}(f_{\boldsymbol{\theta}^*})$ will be low, but $\hat{R}_{test}(f_{\boldsymbol{\theta}^*})$ will be high
  - Fundamental problem in machine learning known as **overfitting**

Example: Toy Sine Data Set

- As an example, let us look at a simple toy data set $\mathcal{D} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$
  - inputs $x_n \in \mathbb{R}$, randomly drawn from a uniform distribution over interval [0,1]:

$$x_n \sim p(x) = \mathcal{U}_{[0,1]}$$

  - (x, y) -relationship is given by a sine curve plus normally distributed noise:

$$y_n = \sin(2\pi x_n) + \epsilon_n \qquad \epsilon_n \sim \mathcal{N}(\epsilon \mid 0, \sigma^2)$$

  - That is, $p(y \mid x_n) = \mathcal{N}(y \mid \sin(2\pi x_n), \sigma^2)$

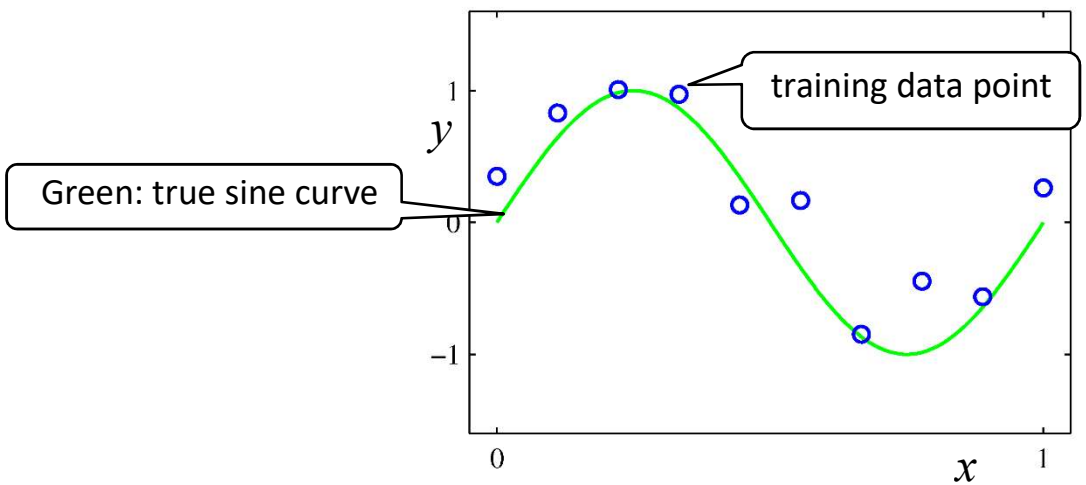
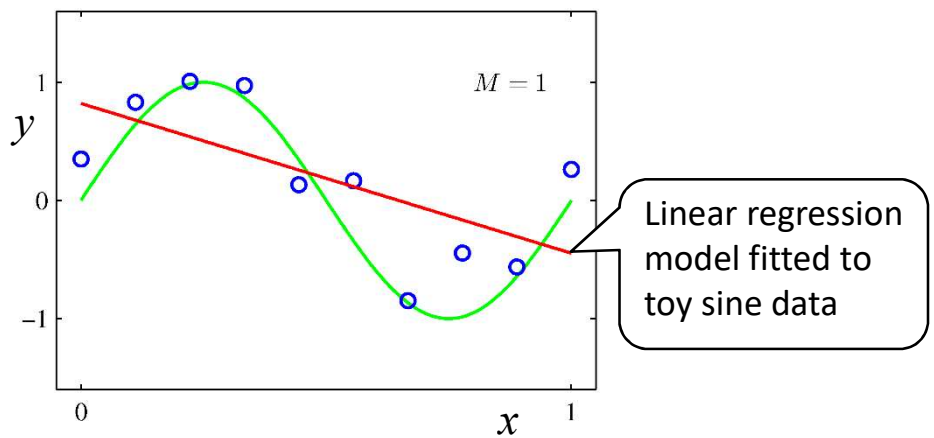


Figure: C. M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006

Example: Polynomial Regression

- What would be a good model for the toy sine data set?
- Clearly, the (*x,y*)-relationship in the data is not linear:



- Idea: we could estimate a model $f_{\boldsymbol{\theta}}$ for the toy data set using a polynomial regression of the form

$$f_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_M x^M$$

- Here, *M* is a parameter of the model class under consideration
  - We need to pick a value for the parameter *M* a priori, before learning
  - *M* can be called a **hyperparameter** to distinguish from parameters $\theta_0, \ldots, \theta_M$

Figure: C. M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006

Example: Polynomial Feature Map

- We can estimate a model $f_{\boldsymbol{\theta}}$ for the toy data set using a polynomial regression model:

$$f_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_M x^M$$

  where *M* determines the degree of the polynomial

- How do we implement such a non-linear polynomial model?
- The simplest way to implement a polynomial model is to transform the original data using a **polynomial feature map**

New, transformed input:

Old input:

$$x \in \mathbb{R} \quad \Rightarrow \quad \mathbf{x} = \begin{pmatrix} 1 \\ x \\ x^2 \\ \ldots \\ x^M \end{pmatrix} \in \mathbb{R}^{M+1}$$

Example: Polynomial Feature Map

- The original training data $\mathcal{D} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ is replaced with new training data $\mathcal{D}' = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, where $\mathbf{x}_n = (1, x_n, x_n^2, \ldots, x_n^M)^{\mathrm{T}}$
- A linear model $f_{\boldsymbol{\theta}} : \mathbb{R}^{M+1} \to \mathbb{R}$ on the transformed feature representation than has the form

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 \cdot 1 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_M x^M$$

- The linear model on the transformed feature representation is thus exactly identical to the polynomial model shown above
- Implementation is easy: Transform the old, one-dimensional input data $\mathcal{D}$ into new data $\mathcal{D}'$, then learn a linear regression models as discussed in earlier lecture
- For example, can fit a polynomial of degree *M=3* to toy sine data set:

Polynomial model captures non-linear (*x,y*)-relationship in data quite well

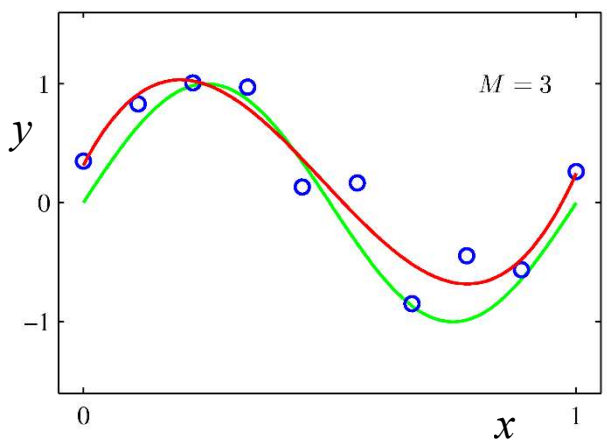


Figure: C. M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006

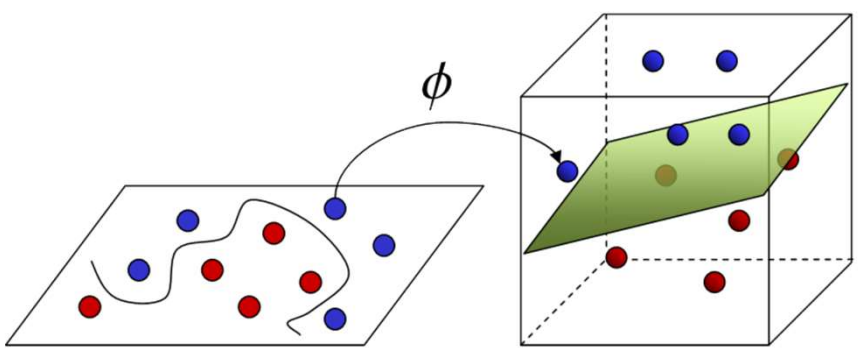
Nonlinear Feature Maps in General

- Nonlinear feature maps are a general and powerful tool in machine learning: Capture non-linear dependency in data without moving away from linear models

- Nonlinear feature maps are not only applicable to one-dimensional data:
  - If the original data is of the form $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$ with $\mathbf{x}_n \in \mathbb{R}^M$, can define a non-linear feature map

$$\Phi : \mathbb{R}^M \to \mathbb{R}^{M'}$$

  - With the nonlinear feature map, can transform the original data to $\mathcal{D}' = \{(\Phi(\mathbf{x}_1), y_1), \ldots, (\Phi(\mathbf{x}_N), y_N)\}$
  - Can then learn a linear model $f_{\boldsymbol{\theta}}(\Phi(\mathbf{x})) = \Phi(\mathbf{x})^{\mathrm{T}}\boldsymbol{\theta}$ on the transformed data



Example: General Polynomial Feature Map

- For example, as a feature map $\Phi : \mathbb{R}^M \to \mathbb{R}^{M'}$ can use a general polynomial feature map of degree *d,*

$$\Phi(\mathbf{x}) = \begin{pmatrix} \Phi_1(\mathbf{x}) \\ \dots \\ \Phi_{M'}(\mathbf{x}) \end{pmatrix}$$

  where the $\Phi_m(\mathbf{x})$ contain all multivariate polynomials of degree $\leq d$

- The linear model is then given by

$$f_{\boldsymbol{\theta}}(\Phi(\mathbf{x})) = \theta_0 + \sum_{m=1}^{M} \theta_m x_m \qquad (\text{degree } d = 1)$$

All polynomials of degree 1 (=original input features)

$$f_{\boldsymbol{\theta}}(\Phi(\mathbf{x})) = \theta_0 + \sum_{m=1}^{M} \theta_m x_m + \sum_{m=1}^{M} \sum_{l=1}^{M} \theta_{m,l} x_m x_l \qquad (\text{degree } d = 2)$$

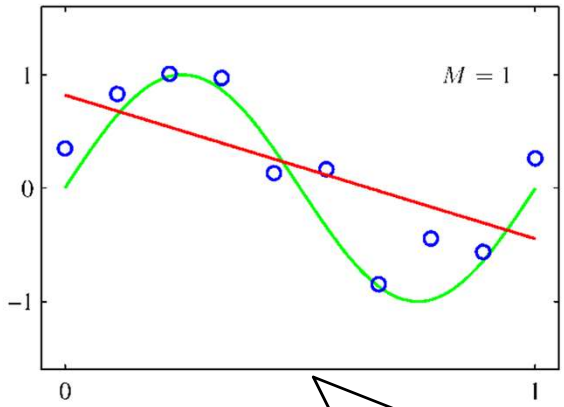
All polynomials of degree 2 in original input features

$$f_{\boldsymbol{\theta}}(\Phi(\mathbf{x})) = \theta_0 + \sum_{m=1}^{M} \theta_m x_m + \sum_{m=1}^{M} \sum_{l=1}^{M} \theta_{m,l} x_m x_l + \sum_{m=1}^{M} \sum_{l=1}^{M} \sum_{k=1}^{M} \theta_{m,l,k} x_m x_l x_k \qquad (\text{degree } d = 3)$$

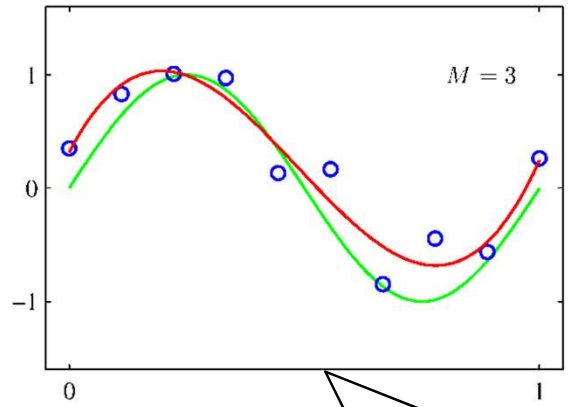
All polynomials of degree 3 in original input features

Example: Overfitting on Toy Data

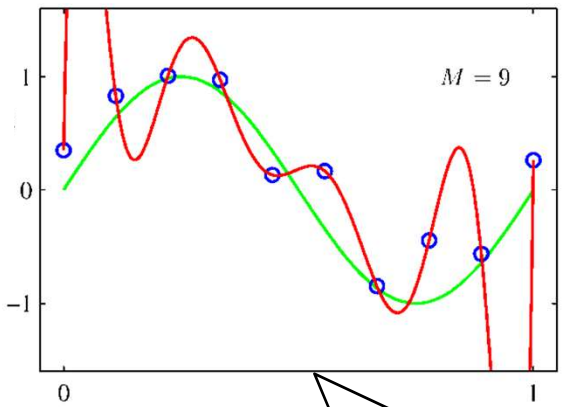
- **Back to the problem of overfitting**: learning spurious, random patterns that are specific to the particular training set rather than an (*x,y*)-relationship that transfers well to new data
- Learned models for different polynomial degrees on the toy sine data set (squared error as loss function):



Polynomial degree 1 (linear model): too simple (not flexible enough), cannot capture non-linear relationship in data



Polynomial degree 3: just right, captures non-linear relationship in data but does not capture much random patterns in data



Polynomial degree 9: too complex (too flexible), starts fitting random patterns that are specific to these data points. **Overfitting**, will not work well on test data

Figure: C. M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006

Example: Overfitting on Toy Data

- For the polynomial models on the toy sine data set, we can also plot training and test error as a function of the polynomial degree:

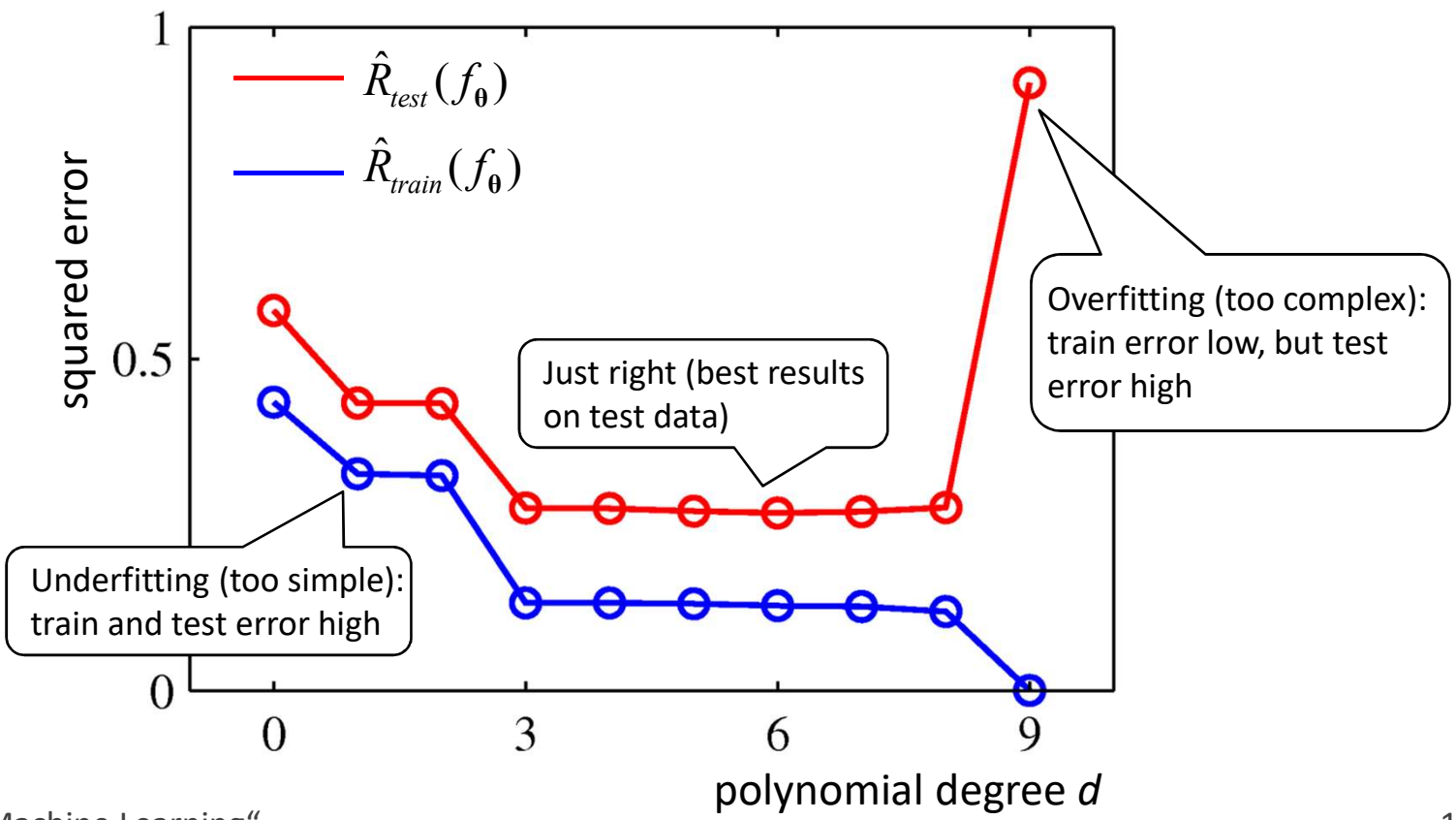
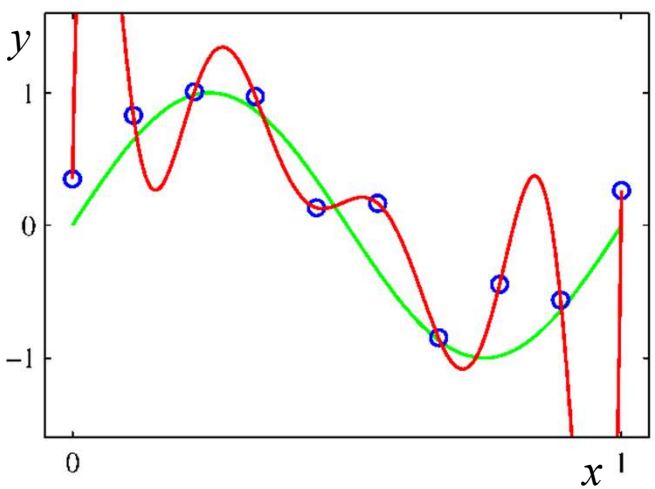


Figure: C. M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006

Overfitting Versus Number of Examples

- The problem of overfitting can decrease if more training examples are available:
  - A polynomial model of degree *d=9* overfits on the (small) toy data set with *N=10* training examples
  - If we create a larger toy data set with *N=100* training examples, a polynomial model of the same degree will not overfit

degree *D=9*, *N=10* examples



degree *D=9*, *N=100* examples

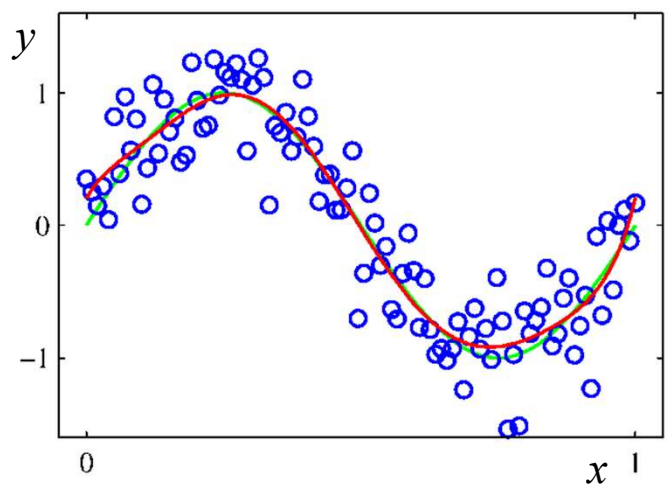
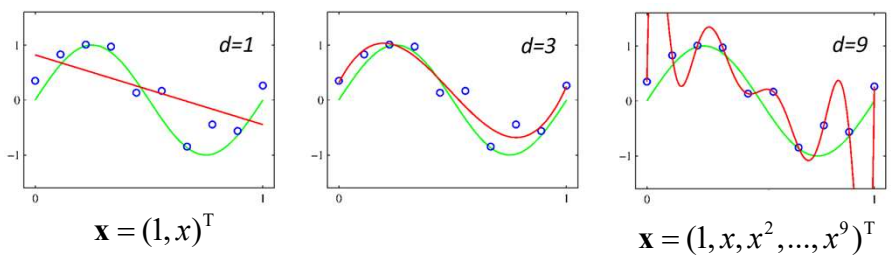


Figure: C. M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006

Summary: Overfitting on Toy Data

- Summary overfitting on the toy sine data set:
  - Polynomial models for the toy sine data set can be learned by transforming the original one-dimensional data using a polynomial feature map
  - The higher the degree of the polynomial, the more flexible/complex the resulting model
  - **Equivalently, we can say that model flexibility/complexity increases by adding more features to the data** (e.g. $\mathbf{x} = (1, x)^{\mathrm{T}}$ versus $\mathbf{x} = (1, x, x^2, \ldots, x^9)^{\mathrm{T}}$)
  - Training error decreases with model complexity, test error first decreases then increases
  - More training instances reduce overfitting



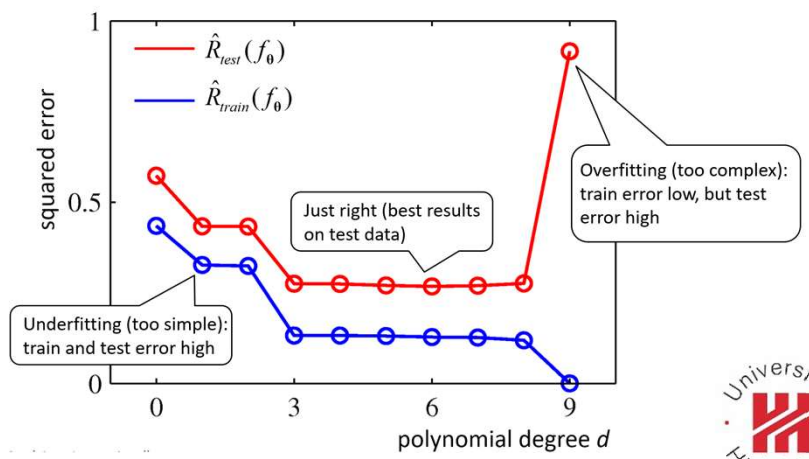


Figure: C. M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006

Agenda For Lecture

- Training and test error
- Model complexity and overfitting
- Model selection and regularization

Model Selection: Empirical

- Given a family of models whose complexity is controlled by a hyperparameter, how do we pick the best value?
- For example, polynomial degree *d*:

$$f_{d,\boldsymbol{\theta}}(x) = \sum_{m=1}^{d} \theta_m x^m$$

- One approach: choose the hyperparameter empirically, by comparing model performance on a separate test data set

Empirical approaches to model selection are widely used in practice

More details in evaluation lecture

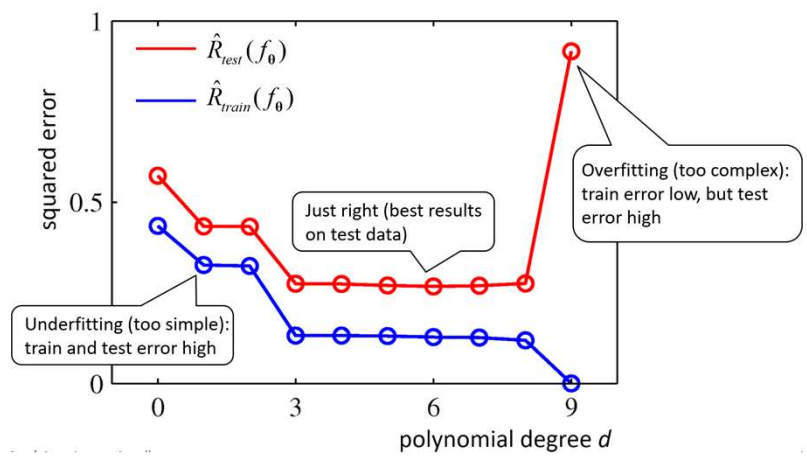


Figure: C. M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006

Model Selection Measures

- Can we measure which model fits the data best without looking at a separate test set?
  - If we just look at the error, loss or likelihood on the training set, a more complex model will always outperform a simpler model
  - For example, any low-degree polynomial can be represented by a high-degree polynomial by setting some coefficients to zero
  - Therefore minimizing the loss for the high-degree model will lead to a lower or equal loss compared to the low-degree model

- A **model selection measure** tries to trade off training loss or likelihood against model complexity:

Minimize: low loss is good,
but low complexity is also good

model selection measure = training loss + complexity

Maximize: high likelihood is good,
but low complexity is also good

model selection measure = training likelihood - complexity

- More complex models are penalized compared to simpler models: try to strike a balance between training data fit and complexity

Model Selection Measures: AIC and BIC

- **Akaike information criterion (AIC):**
  - Trade off model complexity, as measured by number of model parameters, against the likelihood obtained on the training data for the learned parameters

$$\text{Maximize} \quad AIC := \log L - p \qquad \text{(equivalent formulation: minimize } -2\log L + 2p\text{)}$$

L is the likelihood on training data of the learned model

p is the number of parameters in the model

- **Bayes information criterion (BIC):**
  - Trade off model complexity, as measured by number of model parameters, against the likelihood obtained on the training data for the learned parameters
  - Also takes into account number of training examples: to avoid that the log likelihood term dominates for large *N*, increase penalty with increasing *N*

$$\text{Maximize} \quad BIC := \log L - \frac{p}{2}\log N$$

L is the likelihood on training data of the learned model

p is the number of parameters in the model

Model Selection

- If we have a family of models whose complexity is controlled by a hyperparameter, we can learn one model for each hyperparameter and then select a model afterwards by AIC or BIC
- For example, learn polynomial models of different degrees *d:*

$$f_{d,\boldsymbol{\theta}}(x) = \sum_{m=1}^{d} \theta_m x^m$$

- Models with higher degree *d* will have a lower error on the training data, and (under a probabilistic model – see lecture on Bayesian learning) higher likelihood. However, because they have more parameters, their AIC or BIC is not necessarily better
- From all models, can pick the best one by maximizing AIC or BIC

Variable Selection: Projection

- The number of parameters a model has is one measure of its complexity
  - Models with more parameters tend to be more susceptible to overfitting
  - This idea is used in model selection measures such as AIC and BIC
- For linear models, the number of parameters is determined by the number of input variables
- **Idea of variable selection**: to prevent or reduce overfitting, try to select a subset of all available input variables to be included in the model
- We can project instances in a data set onto a subset of input variables V as follows:
  - Let $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$ denote a data set with $\mathbf{x}_n \in \mathbb{R}^M$
  - Let $\pi_V : \mathbb{R}^M \to \mathbb{R}^{\tilde{M}}$ denote a projection operator that projects an instance $\mathbf{x} = (x_1, \ldots, x_M)^{\mathrm{T}} \in \mathbb{R}^M$ onto a subset of its attributes,

    $$\pi_V((x_1, \ldots, x_M)) = (x_{m_1}, \ldots, x_{m_{\tilde{M}}})$$

    where $V = \{x_{m_1}, \ldots, x_{m_{\tilde{M}}}\} \subseteq \{x_1, \ldots, x_M\}$ is the subset of attributes
  - We can apply the projection operator to a complete data set by

    $$\pi_V(\mathcal{D}) = \{(\pi_V(\mathbf{x}_1), y_1), \ldots, (\pi_V(\mathbf{x}_N), y_N)\}$$

Variable Selection: Problem Setting

- We can formalize the problem of variable selection as follows:

- **Given**
  - A data set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$ with $\mathbf{x}_n \in \mathbb{R}^M$
  - A learning algorithm $\mathcal{A}$ that produces a model $f_{\boldsymbol{\theta}^*}$ from a data set $\mathcal{D}$, $f_{\boldsymbol{\theta}^*} = \mathcal{A}(\mathcal{D})$
  - A scoring function $\mathcal{S}(f_{\boldsymbol{\theta}^*})$ that returns a score for a model $f_{\boldsymbol{\theta}^*}$. The scoring function could be a model selection measure such as AIC or BIC, or the accuracy of the model on a separate test data set

- **Find**
  - A subset of variables $V = \{x_{m_1}, \ldots, x_{m_{\tilde{M}}}\} \subseteq \{x_1, \ldots, x_M\}$ that maximizes model score:

$$V^* = \arg \max_{V \subseteq \{x_1, \ldots, x_M\}} \underbrace{\mathcal{S}(\underbrace{\mathcal{A}(\underbrace{\pi_V(\mathcal{D})}_{\text{data set reduced to subset of variables}})}_{\text{model learned on reduced data set}})}_{\text{score of model learned on reduced data set}}$$

Variable Selection as Search

- How do we solve the optimization problem?

$$V^* = \arg \max_{V \subseteq \{x_1,\ldots,x_M\}} \mathcal{S}(\mathcal{A}(\pi_V(\mathcal{D})))$$

- In principle, can go through all 2^M subsets $V \subseteq \{x_1,\ldots,x_M\}$, build the reduced data set $\pi_V(\mathcal{D})$, learn a model $f_{\boldsymbol{\theta}^*} = \mathcal{A}(\pi_V(\mathcal{D}))$, compute the score $\mathcal{S}(\mathcal{A}(\pi_V(\mathcal{D})))$, and select the best subset V
- However, this will be computationally infeasible unless *M* is very small
- Instead, usually use greedy heuristic approaches
  - in **forward search**, start with $V = \emptyset$ and successively add variables to the set, choosing at each step the variable to add that improves the score $\mathcal{S}(\mathcal{A}(\pi_V(\mathcal{D})))$ the most
  - in **backward search**, start with $V = \{x_1,\ldots,x_M\}$, and successively delete variables from the set, choosing at each step the variable to remove that improves the score $\mathcal{S}(\mathcal{A}(\pi_V(\mathcal{D})))$ the most

- Both forward and backward search are not optimal, that is, they will generally not solve the optimization problem above

Variable Selection: Forward Search

- Greedy forward search in pseudocode:

Algorithm variable-selection-forward-search

Input : training data $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, learning algorithm $\mathcal{A}$, scoring function $\mathcal{S}$

Output : set of parameters V

1. $V_{used} := \emptyset$
2. $V_{left} := V$
3. $improvement = \text{True}$
4. while ($improvement$):
5. $\quad \text{gain}_{best} = 0$
6. $\quad$ for $v \in V_{left}$:
7. $\quad\quad gain := \mathcal{S}(\mathcal{A}(\pi_{V_{used} \cup \{v\}}(\mathcal{D}))) - \mathcal{S}(\mathcal{A}(\pi_{V_{used}}(\mathcal{D})))$
8. $\quad\quad$ if $gain > gain_{best}$:
9. $\quad\quad\quad \text{gain}_{best} := gain$
10. $\quad\quad\quad \text{v}_{best} := v$
11. $\quad improvement := (gain_{best} > 0)$
12. $\quad$ if $improvement$:
13. $\quad\quad \text{V}_{used} := \text{V}_{used} \cup \{v_{best}\}$
14. $\quad\quad \text{V}_{left} := \text{V}_{left} \setminus \{v_{best}\}$
15. return V_{used}

Initially, no input variables used in model

Try adding any of the input variables not currently used in model

Gain in score when adding that variable (positive or negative)

If a variable was found that improved score when added to model, add it to set of variables and remove it from candidate set

Variable Selection: Backward Search

- Greedy backward search in pseudocode:

Algorithm variable-selection-backward-search

Input : training data $\mathcal{D} = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_N, y_N)\}$, learning algorithm $\mathcal{A}$, scoring function $\mathcal{S}$

Output : set of parameters V

1. $V_{used} := V$
2. $improvement = \text{True}$
3. while ($improvement$):
4. $\quad \text{gain}_{best} = 0$
5. $\quad$ for $v \in V_{used}$:
6. $\quad\quad gain := \mathcal{S}(\mathcal{A}(\pi_{V_{used} \setminus \{v\}}(\mathcal{D}))) - \mathcal{S}(\mathcal{A}(\pi_{V_{used}}(\mathcal{D})))$
7. $\quad\quad$ if $gain > gain_{best}$:
8. $\quad\quad\quad \text{gain}_{best} := gain$
9. $\quad\quad\quad \text{v}_{best} := v$
10. $\quad improvement := (gain_{best} > 0)$
11. $\quad$ if $improvement$:
12. $\quad\quad \text{V}_{used} := \text{V}_{used} \setminus \{v_{best}\}$
13. return V_{used}

Initially, all input variables are used

Try removing any of the input variables currently in model

Gain in score when removing that variable (positive or negative)

If a variable was found that improved score when removed from model, remove it from set of variables

Motivation: Shrinkage

- For linear models such as linear regression or logistic regression, which have the form

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = g(\sum_{m=1}^{M} \theta_m x_m)$$

g is identity function (for linear regression) or sigmoid/softmax (for logistic regression)

removing a variable x_m from the model is equivalent to forcing its model parameter θ_m to zero

- **Idea:** rather than forcing a parameter to zero, can make it small
  - smaller parameters can be interpreted as a simpler/less complex model
  - limiting the parameters to small values can therefore reduce overfitting

- This idea can be implemented by adding a term to the objective function during learning (loss or likelihood) that penalizes large parameter values
  - During learning, we trade off training loss or likelihood versus small parameters
  - Called **shrinkage**, because the term „shrinks“ parameters towards zero

L2 and L1 Regularization

- There are various types of shrinkage techniques for different problem settings
- Techniques such as shrinkage that add a term to control model complexity to the optimization objective during learning are also called **regularization** techniques

- Most widely used are so-called **L2 regularization** and **L1 regularization**:

L2 („Tikhonov“) regularization: added term is $\lambda \sum_{m=1}^{M} \theta_m^2 = \lambda \|\boldsymbol{\theta}\|_2^2$

L1 („Lasso“) regularization: added term is $\lambda \sum_{m=1}^{M} |\theta_m| = \lambda \|\boldsymbol{\theta}\|_1$

- L1 and L2 regularization can also be combined:

„Elastic Net“ regularization: added term is $\lambda_1 \|\boldsymbol{\theta}\|_1 + \lambda_2 \|\boldsymbol{\theta}\|_2^2$

Ridge Regression

- Review: learning a linear regression model with quadratic loss function

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \qquad L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}(f_{\boldsymbol{\theta}}(\mathbf{x}_n) - y_n)^2$$

- Adding an L2 regularization term results in

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \qquad L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}(f_{\boldsymbol{\theta}}(\mathbf{x}_n) - y_n)^2 \boxed{+\lambda\|\boldsymbol{\theta}\|_2^2}$$

- Linear regression with L2 regularization is also called **ridge regression**
- The optimization problem of ridge regression has a closed-form solution:

$$\boldsymbol{\theta}^* = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + N\lambda\mathbf{I})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} \qquad \mathbf{I} = \begin{pmatrix} 1 & \dots & 0 \\ \dots & \ddots & \dots \\ 0 & \dots & 1 \end{pmatrix} \in \mathbb{R}^M, \text{ "identity matrix"}$$

- The value λ is is a hyperparameter that trades off model complexity and fit to data
  - low λ: do not control model complexity much ($\lambda = 0$: normal regression)
  - high λ: emphasize low model complexity, avoid overfitting

Review: Gradient Descent for Linear Regression

- Can also learn ridge regression via gradient descent
- Review: gradient descent for standard (non-regularized) linear regression
  - gradient of loss function, as derived in lecture on linear regression:

$$L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}(f_{\boldsymbol{\theta}}(\mathbf{x}_n) - y_n)^2$$

$$\nabla L(\boldsymbol{\theta}) = \frac{1}{N}(-\mathbf{X})^{\mathrm{T}} 2(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})$$

  - Optimize model by gradient descent:

1. $\boldsymbol{\theta}_0 = \text{randomInitialization}()$
2. for $i = 0,\ldots,i_{max}$:
3. $\quad\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta\nabla L(\boldsymbol{\theta}_i)$
4. $\quad$ if $L(\boldsymbol{\theta}_i) - L(\boldsymbol{\theta}_{i+1}) < \epsilon$:
5. $\quad\quad$ return $\boldsymbol{\theta}_{i+1}$
6. raise Exception("Not converged in i_{max} iterations")

Gradient Descent for Ridge Regression

- Gradient for the objective function of ridge regression can be derived as

$$L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}(f_{\boldsymbol{\theta}}(\mathbf{x}_n) - y_n)^2 \boxed{+\lambda\|\boldsymbol{\theta}\|_2^2}$$

$$\nabla L(\boldsymbol{\theta}) = \frac{\partial}{\partial\boldsymbol{\theta}}\frac{1}{N}\sum_{n=1}^{N}(f_{\boldsymbol{\theta}}(\mathbf{x}_n) - y_n)^2 \boxed{+\frac{\partial}{\partial\boldsymbol{\theta}}\lambda\|\boldsymbol{\theta}\|_2^2} = \frac{1}{N}(-\mathbf{X})^{\mathrm{T}}2(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) \boxed{+2\lambda\boldsymbol{\theta}}$$

Derivative of regularization term:

$$\frac{\partial}{\partial\boldsymbol{\theta}}\|\boldsymbol{\theta}\|_2^2 = \frac{\partial}{\partial\boldsymbol{\theta}}\langle\boldsymbol{\theta},\boldsymbol{\theta}\rangle = 2\boldsymbol{\theta}$$

- Optimize model by gradient descent:

Gradient descent algorithm

1. $\boldsymbol{\theta}_0 = \text{randomInitialization}()$
2. for $i = 0,\ldots,i_{max}$:
3. $\quad\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta\nabla L(\boldsymbol{\theta}_i)$
4. $\quad$ if $L(\boldsymbol{\theta}_i) - L(\boldsymbol{\theta}_{i+1}) < \epsilon$:
5. $\quad\quad$ return $\boldsymbol{\theta}_{i+1}$
6. raise Exception("Not converged in i_{max} iterations")

Review: Logistic Regression

- Similarly as for linear regression, can also add a regularization term to logistic regression
- Review (see lecture „Linear Classification“): learning logistic regression
  - Objective function is log-likelihood of training data (maximize):

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} L_{cll}(\boldsymbol{\theta}) \qquad L_{cll}(\boldsymbol{\theta}) = \log p(y_1,\ldots,y_N \mid \mathbf{x}_1,\ldots,\mathbf{x}_N,\boldsymbol{\theta})$$

  - We have derived the gradient $\nabla L_{cll}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \mathbf{x}_n (y_n - f(\mathbf{x}_n))$
  - Learn model using gradient ascent in likelihood

Gradient ascent algorithm

1. $\boldsymbol{\theta}_0 = \text{randomInitialization()}$
2. for $i = 0,\ldots,i_{max}$:
3. $\quad\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \eta \nabla L_{cll}(\boldsymbol{\theta}_i)$
4. $\quad$ if $L_{cll}(\boldsymbol{\theta}_{i+1}) - L_{cll}(\boldsymbol{\theta}_i) < \epsilon$:
5. $\quad\quad$ return $\boldsymbol{\theta}_{i+1}$
6. raise Exception("Not converged in i_{max} iterations")

L2-Regularized Logistic Regression

- Add L2 regularization to logistic regression: because we are maximizing objective function, need to substract the penalty term

$$L_{cll}(\boldsymbol{\theta}) = \log p(y_1,\ldots,y_N \mid \mathbf{x}_1,\ldots,\mathbf{x}_N,\boldsymbol{\theta}) \boxed{-\lambda \|\boldsymbol{\theta}\|_2^2}$$

- The gradient then becomes

$$\nabla L_{cll}(\boldsymbol{\theta}) \;=\; \sum_{n=1}^{N} \mathbf{x}_n (y_n - f(\mathbf{x}_n)) \boxed{-2\lambda\boldsymbol{\theta}}$$

- Learn parameters by gradient ascent as before

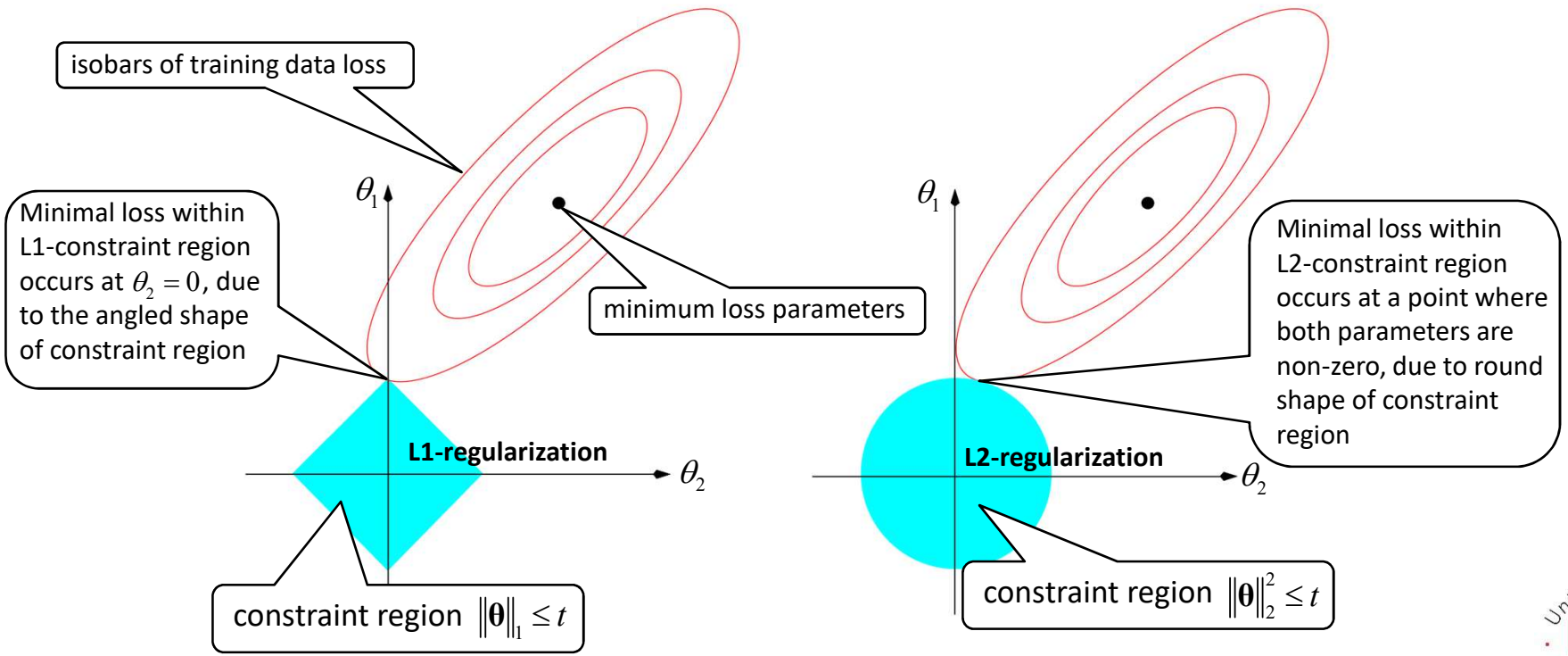
Gradient ascent algorithm

1. $\boldsymbol{\theta}_0 \;= \text{randomInitialization}()$
2. for $i = 0,\ldots,i_{max}$:
3. $\quad\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \eta \nabla L_{cll}(\boldsymbol{\theta}_i)$
4. $\quad$ if $L_{cll}(\boldsymbol{\theta}_{i+1}) - L_{cll}(\boldsymbol{\theta}_i) < \epsilon$:
5. $\quad\quad$ return $\boldsymbol{\theta}_{i+1}$
6. raise Exception("Not converged in i_{max} iterations")

L2 Versus L1 Regularization

- L2 regularization is most widely used, but L1 regularization has the advantage that it can lead to sparse solutions, where some parameters are zero

  Example: two-dimensional parameter space, $\boldsymbol{\theta} = (\theta_1, \theta_2)$.
  If we minimze the loss under a constraint for the parameter norm, L1 can lead to sparse solutions



Hastie, Tibshirani, and Friedman. *Elements of Statistical Learning,* 2001.

Toy Example: Regularization and Overfitting

- Regularization can control model complexity and thereby prevent overfitting
- For example, in the toy sine data set, training a ploynomial model of degree *d=9,*

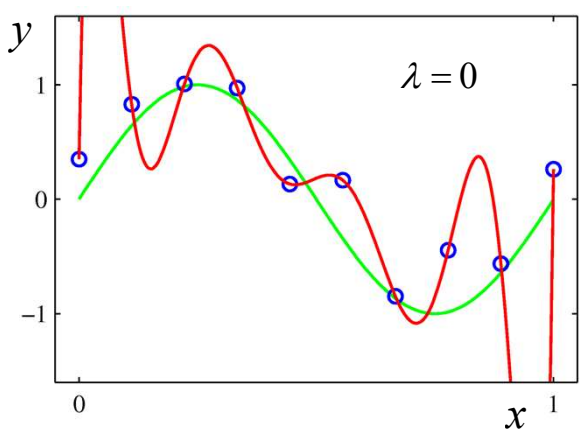
$$f_{\boldsymbol{\theta}}(x) = \sum_{m=1}^{9} \theta_m x^m$$

but with a L2-regularization term in the objective,

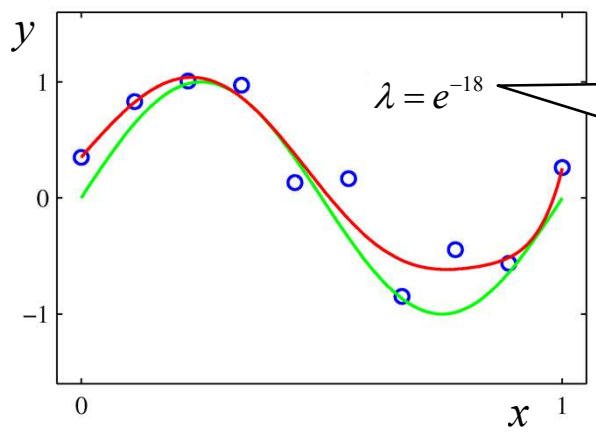
$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \qquad L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}(f_{\boldsymbol{\theta}}(\mathbf{x}_n) - y_n)^2 + \lambda \|\boldsymbol{\theta}\|_2^2$$

does not result in overfitting for *N=10* training data points:

without regularization, model overfits



with regularization, no overfitting



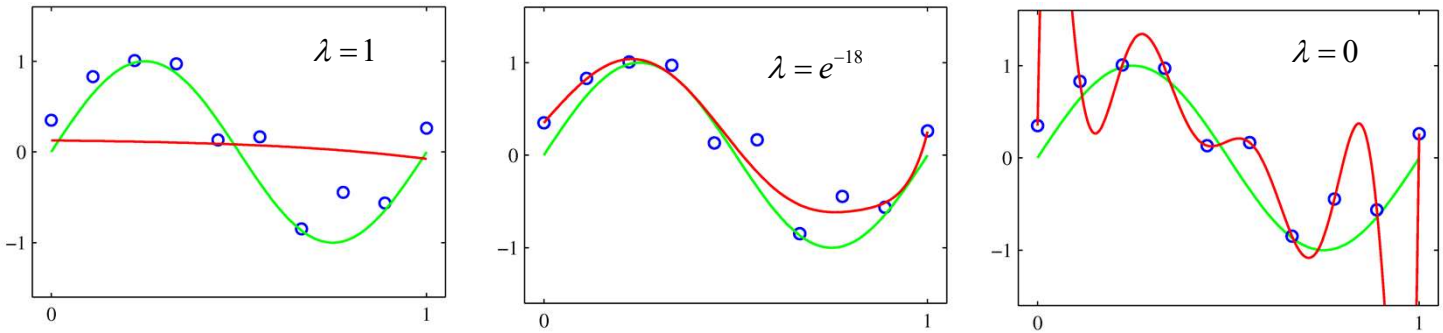
Regularization weight is set to $\log \lambda = -18$. The hyperparameter λ typically moves on a logarithmic scale.

Figure: C. M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006

Regularization and Overfitting

- The regularization weight λ now controls model complexity, in a similar way as the polynomial degree *d* before
- Learning polynomial models of degree *d=9,* with different regularization weights:

overregularized model, underfits | **good regularization, good fit** | **no regularization, overfits**



- The regularization weight also has a corresponding effect on training and test error:

Regularization too low: low training error, but high test error

Regularization correct: ok training error, lowest test error

Regularization too high: high training error, high test error

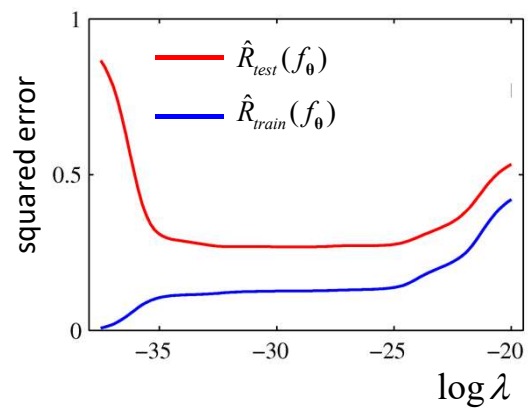


Figure: C. M. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006

Summary

- When learning a model from data, the complexity/flexibility of the model needs to be matched to the data set
  - If the model is too complex, it might pick up spurious, random patterns that are specific to the particular training set, called **overfitting** the training data
  - If the model is not complex enough, it might not be able to fully pick up the true (x,y)-relationship represented in the data, called **underfitting** the training data

- **Regularization** controls model complexity and thereby reduces overfitting
  - An additional term is added to the objective function during learning that penalizes complex models
  - During learning, there is thus a trade-off between fitting the training data well while not making the model too complex

- The most widely used regularization technique is parameter shrinkage, where the regularization term penalizes large values of model parameters