Classic ML

Supervised Learning Scenario

map input to output: $y = f(\boldsymbol{x})$ (estimated: $\hat{f}(\boldsymbol{x})$)

random variables Y and $\boldsymbol{X} = (X_1, X_2, \cdots, X_p)$ ← usually high-dimensional

training (curve fitting / parameter estimation):

fit data set of i.i.d. $(y_i, \boldsymbol{x}_i)$ pairs → minimize deviations between y and $\hat{f}(\boldsymbol{x})$

consider discriminative models:

predict conditional density function $p(y|\boldsymbol{x}_i)$

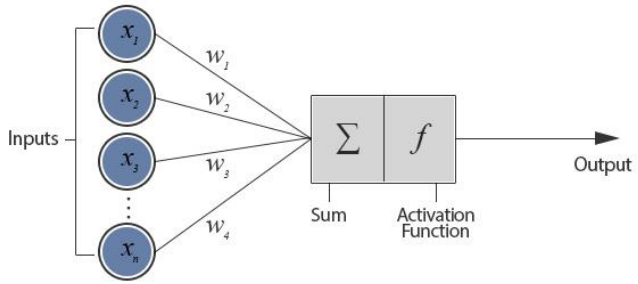
as opposed to generative models predicting $p(y, \boldsymbol{x})$ (or just $p(\boldsymbol{x})$) → $\boldsymbol{x}$ not given, more difficult

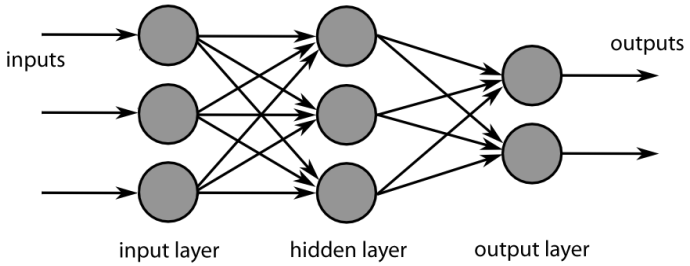
Algorithmic Families of Supervised Learning

parametric models

linear regression

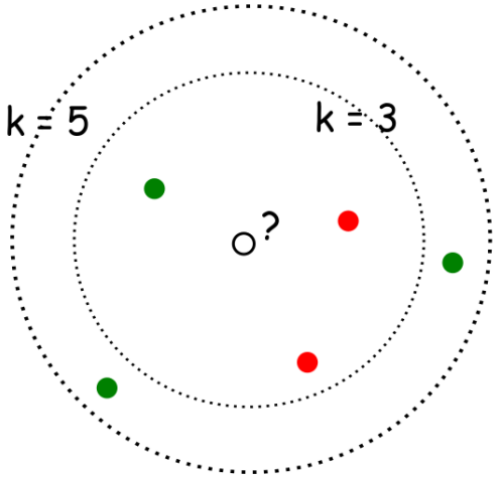
neural networks: many linear models, non-linear by means of activation functions





from wikipedia

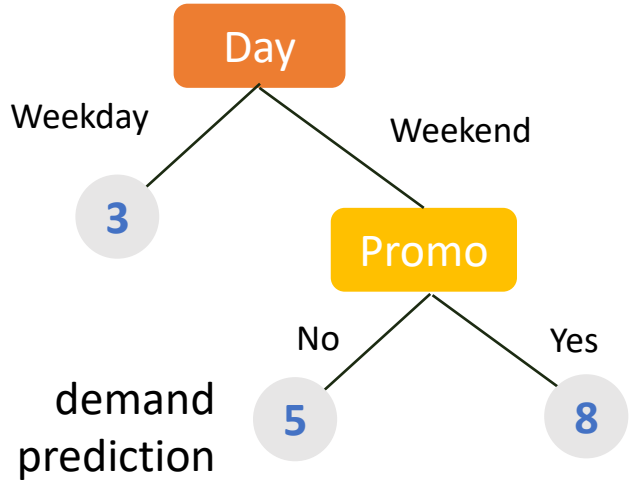
nearest neighbors (local methods, instance-based learning) – non-parametric models



with k = 3,
with k = 5,

kernel/support-vector machines: linear model (maximum-margin hyperplane) with kernel trick

decision trees: rule learning



often used in ensemble methods

- bagging: random forests
- boosting: gradient boosting

Linear Regression

fit: $\hat{f}(\boldsymbol{x}_i)$

$$y_i = \hat{\alpha} + \sum_{j=1}^{p} \hat{\beta}_j \, x_{ij} + \varepsilon_i$$

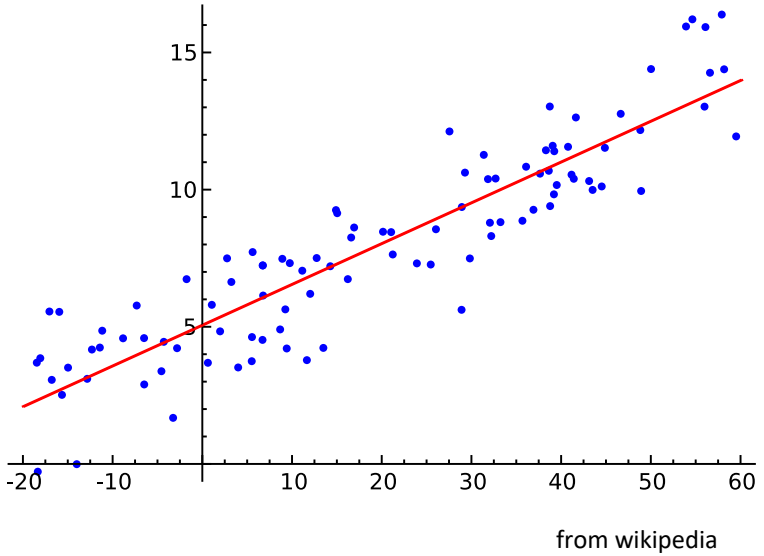
Gaussian

predict:

$$\hat{y}_i = E[Y|\boldsymbol{X} = \boldsymbol{x}_i] = \hat{f}(\boldsymbol{x}_i) = \hat{\alpha} + \sum_{j=1}^{p} \hat{\beta}_j \, x_{ij}$$

$$p(y|\boldsymbol{x}_i) = \mathcal{N}(y; \hat{y}_i, \hat{\sigma}^2)$$

Gaussian mean variance



from wikipedia

parameters to be estimated: $\hat{\alpha}, \hat{\boldsymbol{\beta}}$

$\rightarrow \hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{f}(\boldsymbol{x}_i)\right)^2$

(approximating assumed true $\alpha, \boldsymbol{\beta}, \sigma$)

Classification: Logistic Regression

predict probability p_i for $y = 1$ respectively $y = 0$ for each sample

- logit (log-odds) as link function
- Y following Bernoulli distribution

$$\mathrm{logit}(E[Y|\boldsymbol{X} = \boldsymbol{x}_i]) = \ln\left(\frac{p_i}{1-p_i}\right) = \hat{\alpha} + \sum_{j=1}^{p} \hat{\beta}_j \, x_{ij}$$

Generalization

core of ML:

empirical risk minimization (training error) as proxy for minimizing unknown population risk (test error, aka generalization error or out-of-sample error)

generalization gap: difference between test and training error

curse of dimensionality: many features (dimensions) → lots of data needed to densely sample volume

but reality is friendly: most high-dimensional data sets reside on lower-dimensional manifolds (manifold hypothesis) → enabling effectiveness of ML

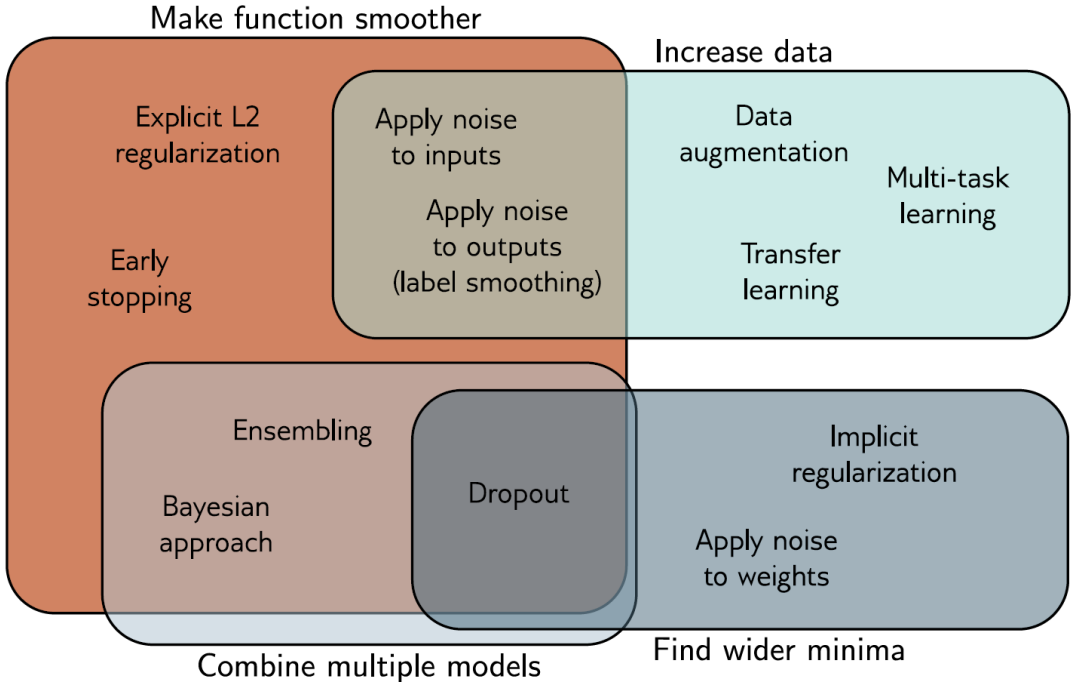
→ need for appropriate **inductive bias** (different forms: model design, regularization, …)

Regularization

reduce test error, possibly at expense of increased training error

many ways:

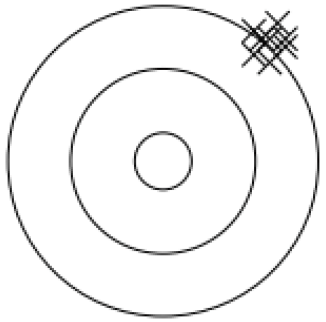
explicit constraints, adding penalties, priors, parameter sharing, data set augmentation, early stopping, dropout, bagging, …



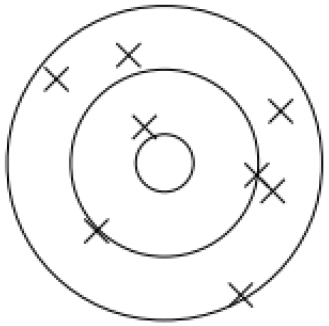
source

Bias, Variance, Irreducible Error

think of fitting ML algorithms as repeatable processes with different (i.i.d.) data sets



bias:
due to too simplistic model
(same for all training data sets)
“underfitting”



variance:
due to sensitivity to specifics (noise)
of different training data sets
“overfitting”

irreducible error (aka Bayes error):

inherent randomness (target generated from random variable following probability distribution)

→limiting accuracy of ideal model

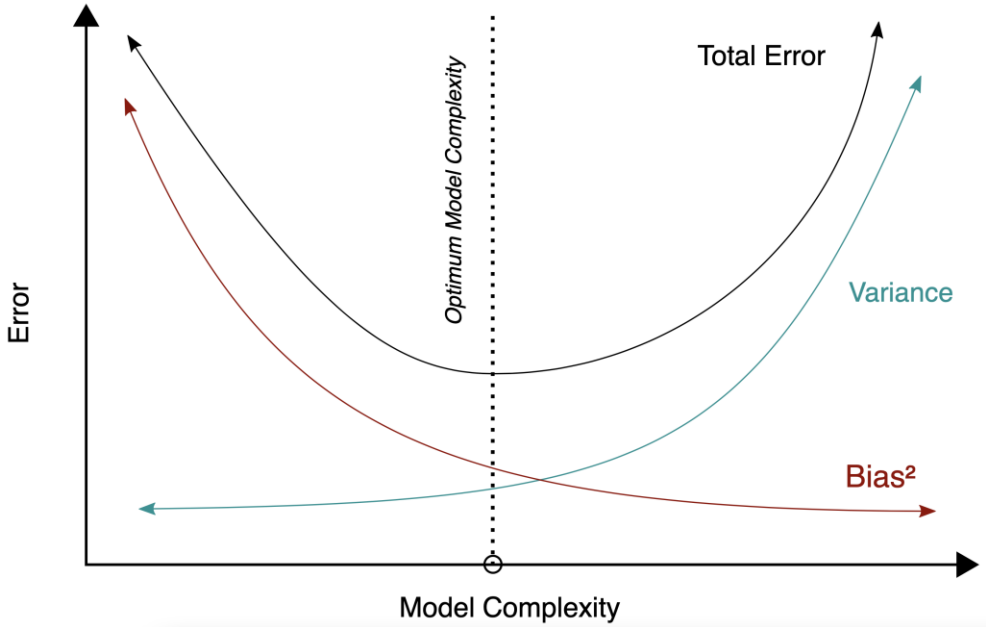
different potential reasons for inherent randomness (noise): complexity, missing information, …

Bias-Variance Tradeoff

- fundamental concept in classical statistical learning theory
- models of higher complexity have lower bias but higher variance (given the same number of training examples)

- generalization error follows U-shaped curve: overfitting once model complexity (number of parameters) passes certain threshold
- overfitting: variance term dominating test error

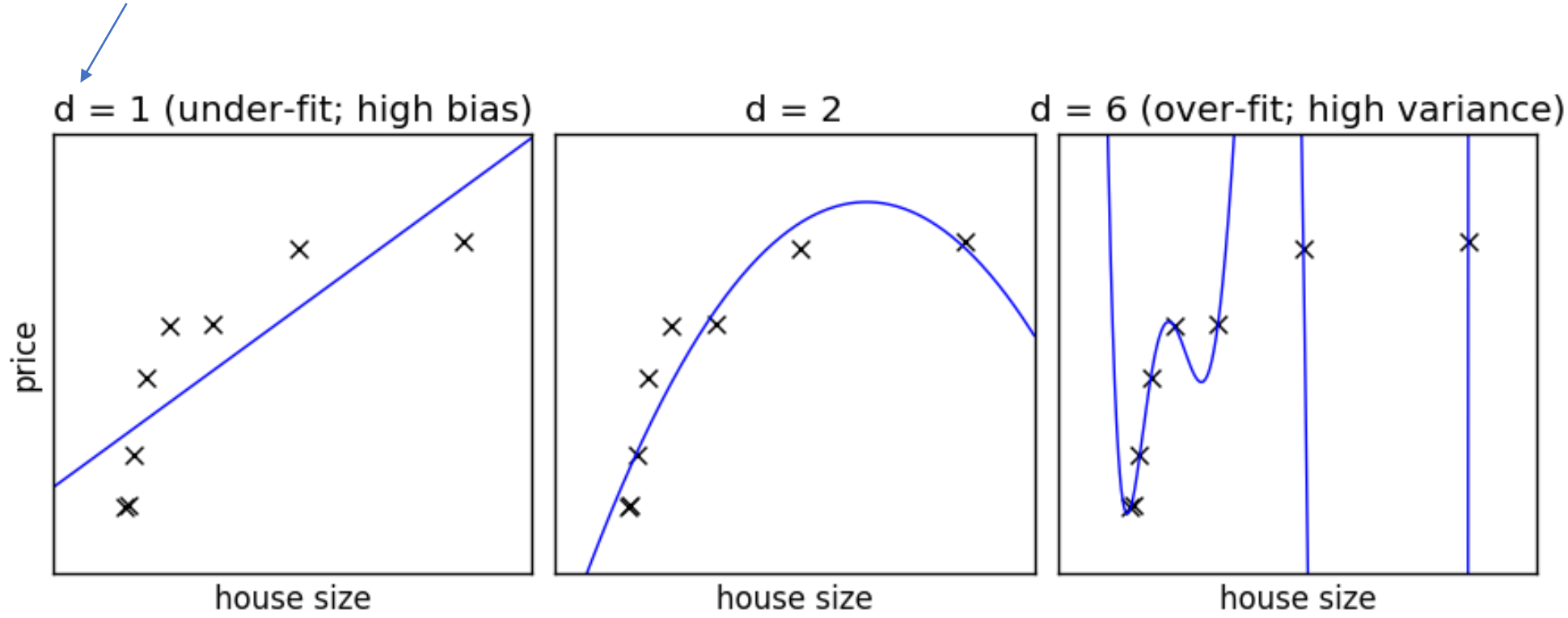
→ increasing model complexity increases test error



from wikipedia

Example: Non-Linear Function Approximation

degree of fitted polynomial



from scikit-learn documentation

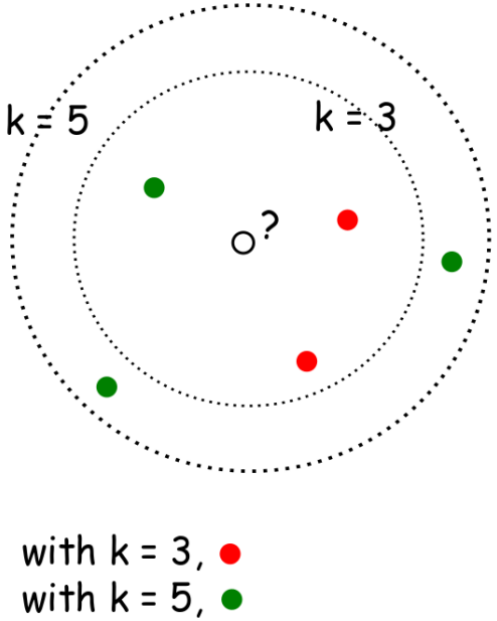
Example: k-Nearest Neighbors

- local method, instance-based learning
- non-parametric
- distance defined by metric on $\boldsymbol{x}$ (e.g., Euclidean)

regression:

$$\hat{f}(\boldsymbol{x}_0) = \frac{1}{k}\sum_{j=1}^{k} y_j$$

with j running over k nearest neighbors of $\boldsymbol{x}_0$



- low k: low bias but high variance
- high k: low variance but high bias

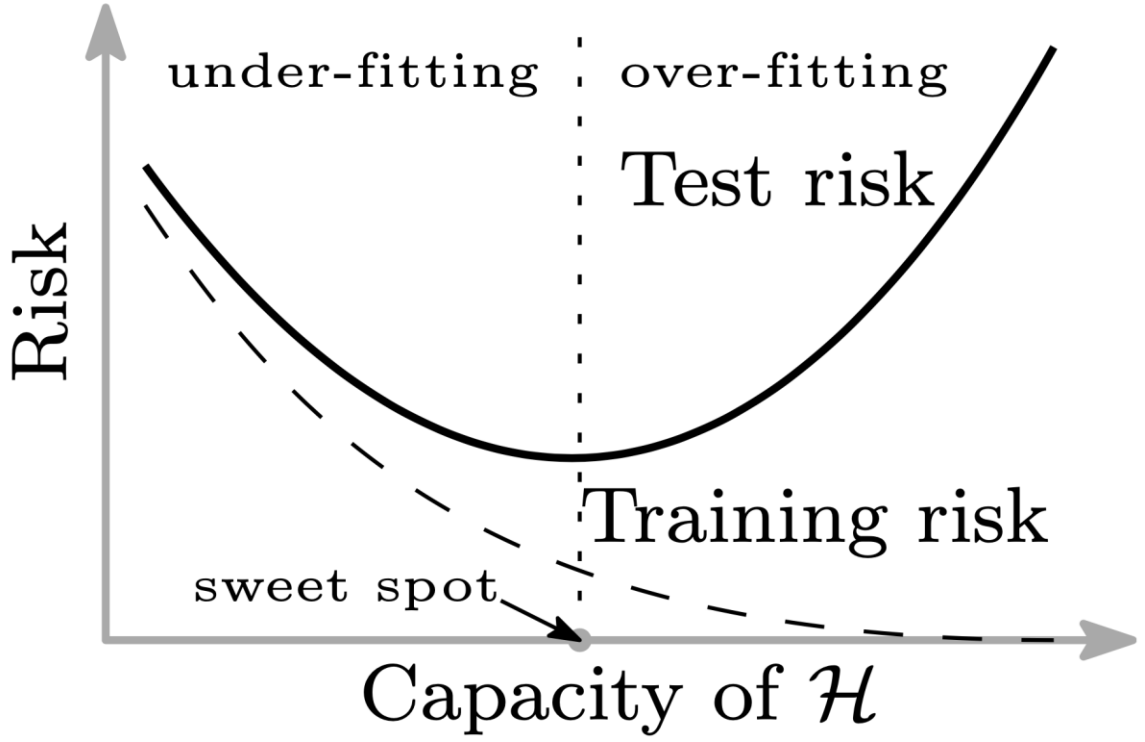
$$bias = f(\boldsymbol{x}) - \frac{1}{k}\sum_{j=1}^{k} y_j$$

$$var = \frac{\sigma^2}{k}$$

Problem of Finding Complexity Sweet Spot

- training/in-sample error keeps decreasing with more complex model (less bias)
- test/out-of-sample error not easily accessible during training

→ e.g., cross-validation, early stopping



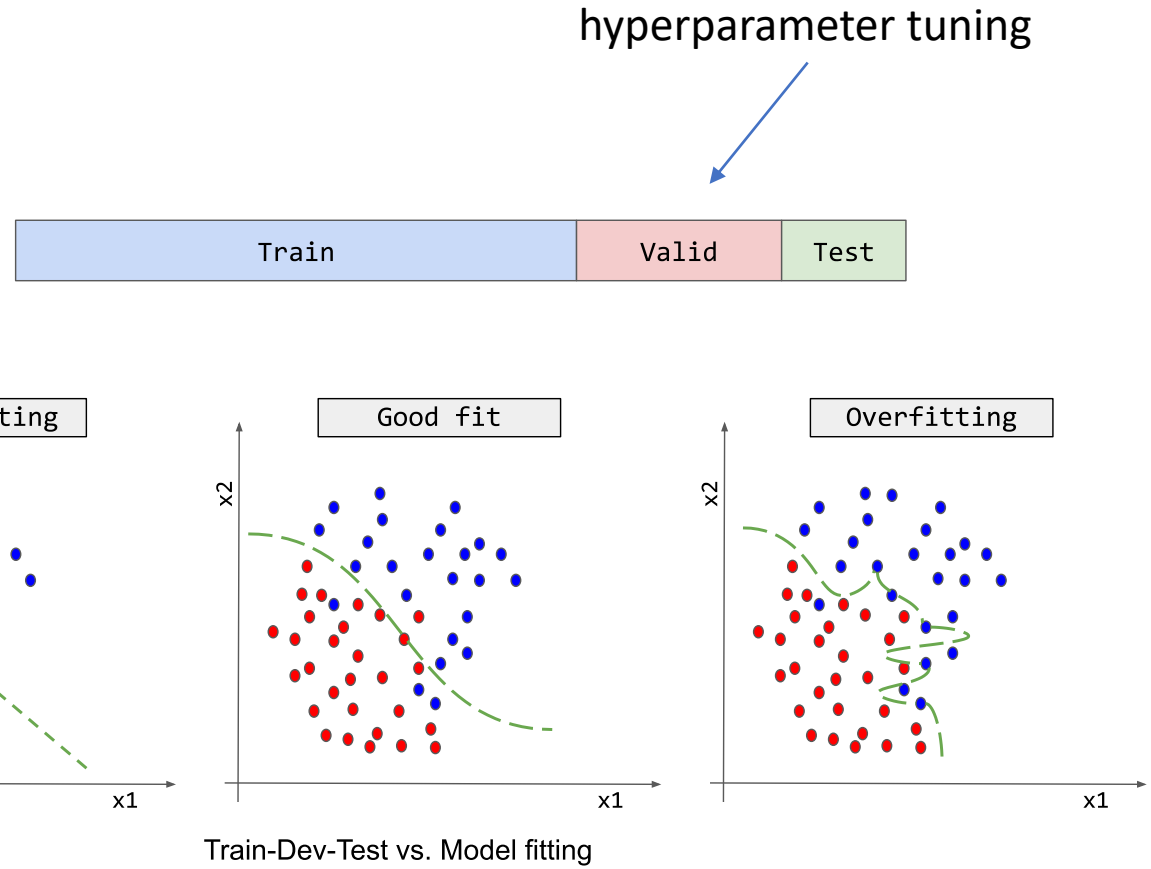
source

Hyperparameters

model complexity often controlled via hyperparameters (parameters not fitted but set in advance)

examples:

- degree of fitted polynomial d
- number of considered nearest neighbors k
- maximum depth of decision tree
- number of trees in random forest
- …



Train-Dev-Test vs. Model fitting

Methods for …

regularization

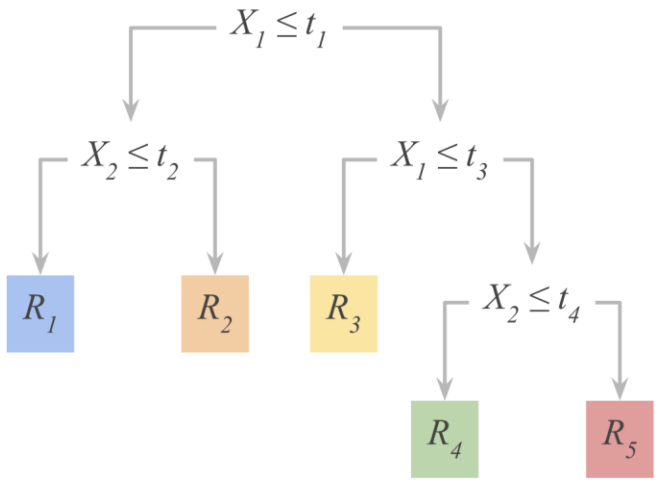
reducing bias:

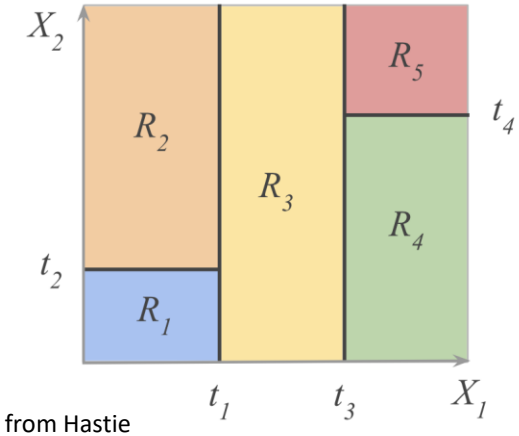
- kNN: consider less neighbors
- decision tree: greater depth
- ensemble: boosting
- neural network: more hidden nodes
- more model features
- larger training data set

reducing variance:

- kNN: consider more neighbors
- decision tree: shallower depth
- ensemble: bagging (kind of regularization)
- neural network: less hidden nodes
- dimensionality reduction, feature selection
- larger training data set

Decision Trees

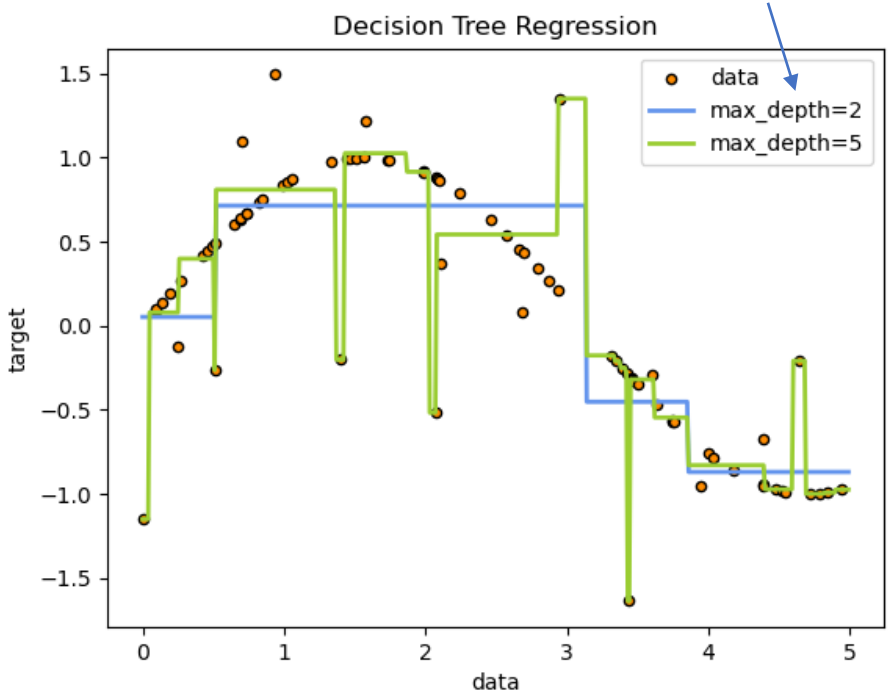




from Hastie

- non-parametric learning of simple decision rules
- usually, binary trees
- axis-parallel decision boundaries (box-shaped regions in feature space)
- fit constant $\hat{c}$ in each box
  - classification: majority class of target
  - regression: average of target
- fully explainable models

hyperparameter



from scikit-learn

Ensemble Learning

idea:

combine several individual models (often of the same type) to form an ensemble model with better predictive performance than each of its constituents

learning in several different ways from the training data

- the trainings of the individual constituent models are (kind of) independent
- outputs of individual models are combined (e.g., averaged)

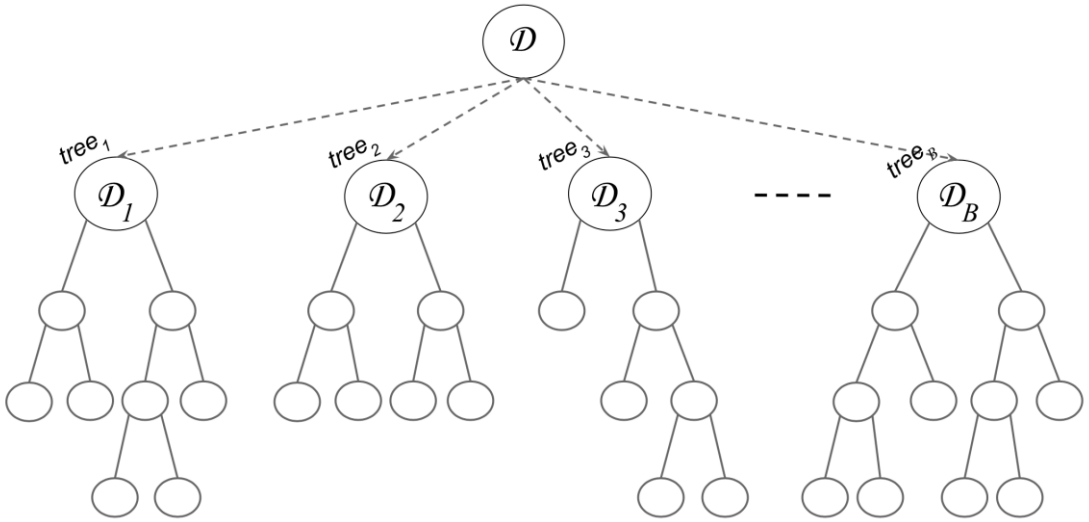
Bagging (Bootstrap Aggregating)

idea: train individual models of ensemble method on random (sub)sets of the training data (random sample with replacement)

- reduces variance of ensemble model compared to individual models
- but not bias → use low-bias base models

individual models combined by averaging (regression) or majority voting (classification) → committee of models

example with decision trees:



source

Random Subspace Method

aka feature bagging

idea: train different members (models) of an ensemble method on random subsets of all features (random sample with replacement)

→ reduce correlation between ensemble members

individual models combined by averaging (regression) or majority voting (classification) → committee of models

Random Forest

ensemble method using

- decision trees as individual models (usually, rather large, low-bias decision trees)
- combination of bagging and random subspace method (features sampled at each node in the trees)

compared to individual decision trees, random forests

+ reduce variance → improve accuracy

- lose explainability

one of the most popular off-the-shelf ML algorithms (often, good performance with little hyperparameter tuning and data preparation)

Boosting

idea: sequentially learn and combine several “weak” learners (such as small decision trees, but in principle any ML algorithm) to construct a “strong” one

→ gradually (in a greedy fashion) reducing bias of ensemble model

in simple terms:

building a model from the training data, then creating a second model that attempts to correct the errors from the first model, …

→ each subsequent weak learner is forced to concentrate on the examples that are missed by the previous ones in the sequence

not a committee of models

typically, use simple, high-bias methods as individual models

Forward Stagewise Additive Modeling

boosting can be understood as fitting an additive expansion in a set of basis functions $b(\boldsymbol{x}; \hat{\gamma}_m)$ (e.g., decision trees, with $\hat{\gamma}_m$ parametrizing splits): $\hat{f}(\boldsymbol{x}) = \sum_{m=1}^{M} \hat{\beta}_m \, b(\boldsymbol{x}; \hat{\gamma}_m)$

loss minimization of $\hat{f}(\boldsymbol{x})$ computationally expensive → boosting: sequentially add and fit individual basis functions without changing already fitted ones

boosting algorithm for sequential optimization using basis functions:

1. initialize $\hat{f}_0(\boldsymbol{x}) = 0$
2. for $m = 1$ to M
   a) compute $\underset{\hat{\beta}_m, \hat{\gamma}_m}{\operatorname{argmin}} \sum_{i=1}^{N} L\left(y_i, \hat{f}_{m-1}(\boldsymbol{x}_i) + \hat{\beta}_m \, b(\boldsymbol{x}_i; \hat{\gamma}_m)\right)$
   b) set $\hat{f}_m(\boldsymbol{x}) = \hat{f}_{m-1}(\boldsymbol{x}) + \hat{\beta}_m \, b(\boldsymbol{x}; \hat{\gamma}_m)$

Decision Tree Sizes for Boosting

consider degree to which features interact with each other in given data set (see ANOVA expansion)

- decision trees with single split (aka decision stumps): covering no interaction effects, just main effects of individual features
- decision trees with two splits: covering second-order interactions
- decision trees with three splits: covering third-order interactions

(usually, interaction order not much higher than ~ 5 in real-world data sets)

why boosting often works better than single large, low-bias model: uncorrelated learners, each focusing on a specific aspect of the data

So, let’s do some modeling …

scikit-learn

coding example: Kaggle House Prices

Assignments

- classification: Kaggle Spaceship Titanic

- regression: Kaggle Store Sales