Machine Learning for Data Science 1

(lecture notes, only for internal use)

Bla Zupan, Erik trumbelj

February 24, 2020

Contents

Chapter 1

Introduction

Machine learning is a set of approaches that can detect patterns in the data. Types of machine learning include predictive and descriptive reinforcement learning. Two major classes of predictive learning are classification and regression. Examples of unsupervised learning approaches include principal component analysis, clustering, and dimensionality reduction. We can formalize predictive and descriptive learning as density estimation, where we develop probabilistic formulations of the form $p(y|\boldsymbol{x}_i, \mathcal{D})$ for predictive, and formulations of the form $p(y|\mathcal{D})$ for unsupervised learning. Resulting probabilistic models $p(y|\theta)$ or $p(y|\boldsymbol{x}_i; \theta)$ may include fixed number of parameters, or their number may vary according to the size of the training data. Interesting concepts in machine learning include the curse of dimensionality, inductive bias, overfitting, model selection, and absence of a universally best model that would fit all kinds of problem domains. [a]

[a] These lecture notes follow Chapter 1 from Murphy [**2012-Murphy**]. Recommended additional reading is Chapter 2 from Hastie et al. [**2016-Hastie**].

1.1 The Purpose of Machine Learning

Machine learning is about learning models from data. More abstractly, given the training data $\mathcal{D}$, we would like to use the data to infer probability distributions. In other words, we would like to build models of the process $p(y)$ that generated the data.

The general task of learning $p(y|\mathcal{D})$, that is, inferring the conditional distribution of variables that define the processes given the data is very complex. In practice, we are, in most cases, not even interested in this general task. Instead, we are interested only in certain aspects of the distribution, and for these, apply specific types of machine learning, like classification, regression, or clustering.

In terms of applications, machine learning is a branch of artificial intelligence that pro-

vides algorithms that can automatically learn from experience without being explicitly programmed. While we will focus on theoretical aspects of machine learning, the reader of this text should place these in practical contexts and consider the tasks such as data acquistion, data cleaning, feature engineering, data cleaning and preprocessing, data visualisation, scoring and estimating the quality and utility of the developed models, and finally, their inclusion within working software and decision support systems. While practically of utmost importance, these engineering aspects will not be at the focus of this course.

1.2 Types of Machine Learning

Supervised learning

Often, we are only interested in how a subset of variables is generated, while the remaining variables are used to explain the behavior of the variables of interest: $\{y_i, \boldsymbol{x}_i\}_{i=1}^n$. This is *supervised learning*, also known as *predictive learning* or predictions. Here, y is referred to as *response variable*, and $\boldsymbol{x}$ represents a vector of *features*. Depending on a branch of science that deals with machine learning, the dependent variable may also be referred to as a target variable, dependent variable (statistics), or label or class variable (machine learning). The independent variables are often referred to as covariates, independent variables, and predictors (statistics), or features and attributes (machine learning). Supervised learning starts with the training data, which includes n pairs of instantiations of independent and dependent variables. The goal is to learn about $p(y|\boldsymbol{x})$ so that we can make predictions for future or unobserved values of independent variable y for any combination of dependent variables $\boldsymbol{x}$ (see Table 1.2).

When y is a nominal variable, we refer to this type of supervised learning as *classification*. When the nominal variable is two-valued, we deal with *binary classification*, and when the domain of the nominal variable includes three or more values, we refer to the problem as *multiclass classification*. When y is continuous, we refer to the problem as *regression*. Less common cases consider a count or ordinal dependent variable, where we refer to the suitable approaches as *count regression* and *ordinal regression*, respectively. In most cases, the dependent variable y will be a scalar, and we will refer to such cases as *univariate* classification or regression. When y is a vector, we will refer to the problem as *multivariate* classification or regression.

Note that while various machine learning approaches specialize in a particular case, it is often easy to generalize a specific approach to deal with other types of the dependent variable. For instance, it is not difficult to adapt classification trees to the regression problems, or even to extend this approach to address multivariate learning.

Table 1.1: A small sample from the famous Iris data set, where Iris flowers are described with four numerical features and are labeled with Iris species. A possible task for this data set is supervised learning, with aim to build a model that predicts species from leaf morphology.

sepal length	sepal width	petal length	petal width	iris
4.4	2.9	1.4	0.2	Iris-setosa
4.8	3.4	1.6	0.2	Iris-setosa
5.4	3.9	1.3	0.4	Iris-setosa
5.2	2.7	3.9	1.4	Iris-versicolor
6.0	2.2	4.0	1.0	Iris-versicolor

Unsupervised learning

We are using *unsupervised learning* when our problem does not include any response variable, that is, when the observations are not labelled. The goal of such learning is again to understand the data generative process $p(y)$, or at least to understand part of its structure.

A common approach to understanding the distribution $p(y)$ is to explain it with a smaller number of factors θ, that is, to learn $p(y|\theta)$, effectively projecting the data into a lower-dimensional space. We refer to such procedure as *dimensionality reduction*. An extreme example of dimensionality reduction is *clustering*, when we try to explain $p(y)$ with a single nominal factor (see Fig. 1.2). In essence, we are trying to group, or cluster, observations.

Figure 1.1: A two-dimensional visualisation of blood cells, originally described with expressions of thousands of features. The visualisation was constructed using t-SNE dimensionality reduction, and exposes potential clusters that need to be further analyzed.

Reinforcement learning

Reinforcement learning is about learning actions of software agents in an environment where the goal is to maximize reward. An example of reinforcement learning is to learn the actions of a robot that travels through the maze and receives sensor input. A reward, in this case, could be time spent in a maze. Reinforcement learning is different from supervised learning in not requiring labeled input. Instead, reinforcement learning aims to find the balance between the exploration of uncharted territory and the exploitation of current knowledge. While we will focus on unsupervised and supervised learning in this course, we will only dive into reinforcement learning in one of our final sessions.

1.3 Models and Learning

A *model* $\mathcal{H}$ is in the most abstract sense a collection of distributions (densities, functions, ...). The elements of a model depend on our task.

> **Example: Simple linear regression - statistical model**
> The simple linear regression is a set of densities
> $\mathcal{H} = \left\{p(y|\boldsymbol{x}, \boldsymbol{\beta}, \alpha, \sigma) = dnorm(\boldsymbol{\beta}\boldsymbol{x} + \alpha, \sigma^2), \boldsymbol{\beta}, \alpha \in \mathbb{R}, \sigma > 0\right\}$

> **Example: Simple linear regression - function approximation**
> The simple linear regression is a set of functions $\mathcal{H} = \{f(\boldsymbol{x}, \boldsymbol{\beta}, \alpha) = \boldsymbol{\beta}\boldsymbol{x} + \alpha, \boldsymbol{\beta}, \alpha \in \mathbb{R}\}$

It is important to reinforce the view of a model as a set of hypotheses. Learning is the process of expressing a preference for certain hypotheses based on evidence (data). Choosing a particular machine learning algorithm, or in other words, choosing a particular model means expressing a preference for a certain type of hypothesis. The logistic regression model, in its basic form, will construct a model which will linearly separate the parameter space of the data instances to, prefarably, separate the data instances from either of the two classes. Separation plane constructed by classification trees may be much more complex and, implicitly, require many more parameters for its descriptions. The choice of the model also entails the choice of the complexity of the hypothesis, which in turn is related to the goodness of fit, overfitting, explainability, and other issues we expose in the text below.

A model is often also referred to as a hypothesis or set of hypotheses. Learning is often referred to as training the model, fitting the model/parameters, estimation.

Learning is the process of selecting elements of $\mathcal{H}$ based on some utility and using data. This is general. In practice, we can select a single element (a single density, function, distribution; as in the two function approximation examples above), a set of elements or even weight each element, for example, a distribution across all elements, as in Bayesian approaches.

Learning is in most cases just a problem in *computation* to be addressed through mathematical, numerical, algorithmic procedures. For parametric models, we typically do *least-*

squares or *maximum likelihood* estimations to obtain *point estimates* of the parameters of the model. That is, by learning, we select a single model. Learning thus becomes an *optimization* problem, or *Bayesian inference*, which is an *integration* or optimization problem, if we do some sort of *structural approximation* or *MAP*.

Parametric and Nonparametric Models

If the set $\mathcal{H}$ can be parametrized with a finite number of parameters, we call the model *parametric*. Otherwise, it is *nonparametric*. A parametric model captures all information about the properties of the data within its fixed number of parameters. Predictions using nonparametric models require knowledge about the current state of the system, that is, require access to the current data.

> **Example: 1-nearest neighbor model - a nonparametric model**
> $\mathcal{H}$ = { all functions f that can be expressed with a set of points (data instances) and the rule that $f(\boldsymbol{x}) = y_i$ of point x nearest to x_i, according to a chosen distance metric} (see Fig. 1.4)

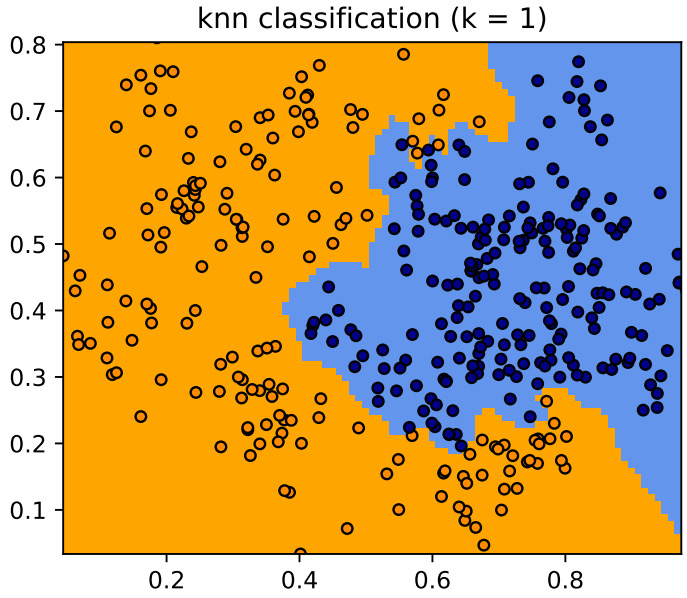


Figure 1.2: Decision boundary of a 1-nearest neighbor model trained on a two-featured binary classification data set with 380 data instances (170 in one class and 210 in the other) as shown on a figure. The decision boundary is complex and may substantially change with any addition on removal of the data.

Parametric models are easier to compute than non-parametric models. Non-parametric models are often more complex and grow with data. Parametric models depend on the data only through a finite number of parameters, while in non-parametric models, the complexity

of the model depends on the training set. Researchers mostly prefer parametric models because it may be easier to estimate its parameters, easier to perform predictions, and easier to tell a story about the data according to a parametric model (e.g., sensitivity analysis, effects of the changes in parameters, parameter interactions). In this sense, parametric models are more prone to interpretation by domain experts. In parametric models, the parameter estimates may have better statistical properties compared to those of non-parametric regression.

Parametric models make stronger assumptions about the data; the learning may be successfull if these assumptions are valid, but the inferred predictors may fail if these assumptions are violated. Think of modeling a sine curve with a linear regression model. A non-parametric algorithm is computationally slower but makes fewer assumptions about the data. In (overly) simplified view, the trade-offs between parametric and non-parametric algorithms are in computational cost and accuracy.

Notice that non-parametric models are related to *lazy learning*. Lazy learning methods generalize the training data at the time of prediction. This type of learning is an alternative to *eager learning*, where the system tries to generalize the training data before receiving queries. An example of the lazy learner is a K-nearest neighbor algorithm. Lazy learners may have an advantage in real-time environments, where the training data changes in time and models trained in the past become obsolete in a relatively short time due to emergent new data and changes of the distributions and underlying processes that generated the data.

1.4 Challenges of Applied Machine Learning

Model Evaluation and Selection

In theory, assuming uniform distribution over all possible datasets, there is no single best model. In fact, and again, in theory, no model is strictly better than any other model. This is in the literature referred to as *no free lunch theorem*, which states that any two optimization algorithms are equivalent when their performance is averaged across all possible problems [**2005-Wolpert-Macready**].

In practice, however, some characteristics are more common in datasets, so some models and algorithms perform better on average because their assumptions (*inductive bias*) better match the characteristics of the data generating process.

The above makes *model selection* a key part of applied machine learning. In order to train a model, we should define some measure of utility we would like to optimize. A trivial approach is then to select the model with the best utility on our available data. However, estimating the model's utility on the data it was trained on is biased and optimistic. In practice, the model's utility on the training data (so-called *in-sample* error) may be substantially better than on independent test data (*out-of-sample* error or *generalization error*). This effect is also known as *overfitting*, and it is something that we want to both detect and prevent.

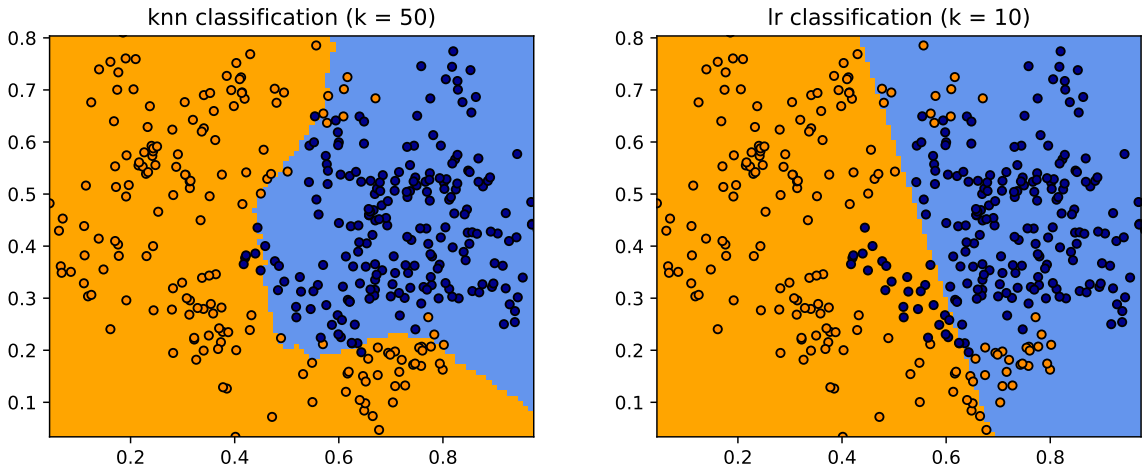


Figure 1.3: Decision boundary on a two-featured binary classification data set as inferred by nearest neighbor algorithm with K=50 and by logistic regression. Which model would perform better on new data?

Overfitting

Overfitting occurs when machine learning model trained on a (limited) data set captures noise of the data instead of the underlying data generation processes. This modelling error occurs when an inferred hypothesis is too closely fit to a limited set of data points, or when a model is too complex for a given data (e.g., Fig. 1.4).

The related practical challenges include knowing when overfitting occurs and finding the right remedy for overfitting. Another challenge is to avoid modeling procedures that led to overfitting. None of these challenges is trivial, and beginners or even quite experienced practitioners often make mistakes that lead to overfitting and consequentially report over-optimistic scores for their modeling procedures. For instance, it has been found that some (if not most) of significant reports on the analysis of microarray gene expression data sets at the break of the century included overfitting [**2003-Simon**]. Common reported mistakes included feature selection before cross-validation or class label-informed feature selection before data visualization. Reports on good accuracies are, despite teachings in data science, present also in recent literature, as reported by Vandewiele et al. [**2019-Vandewiele**]. The authors examined reports on the analysis of a collection of electrohysterogram signals. There, related reports oversampled the data prior to cross-validation, and hence falsely obtained almost perfect accuracies.

Model Complexity and Effective Number of Parameters

More complex models are more likely to overfit (see Fig. 1.4), but the “right” complexity of the model may depend on the amount of data we have (see Fig. **??**). In parametric models,

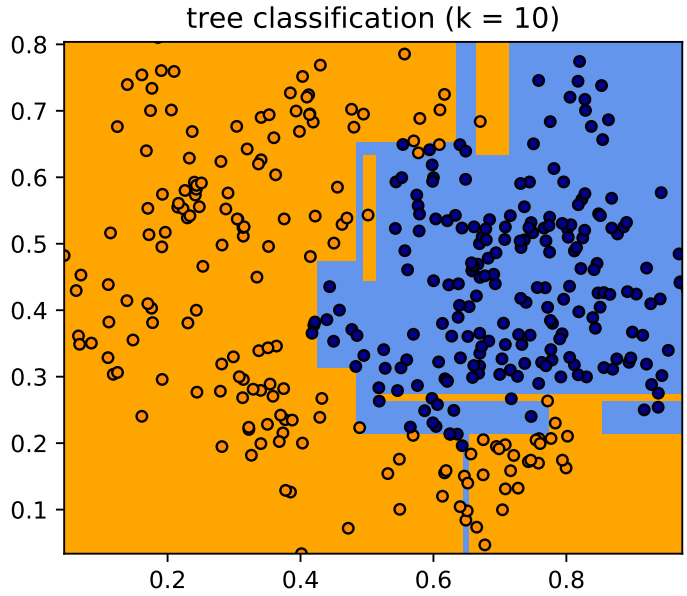


Figure 1.4: Classification trees would often overfit the training data. Figure shows decision boundary of a tree where the allowed maximum tree depth was 10. The decision boundary is complex and often covers single-case exceptions.

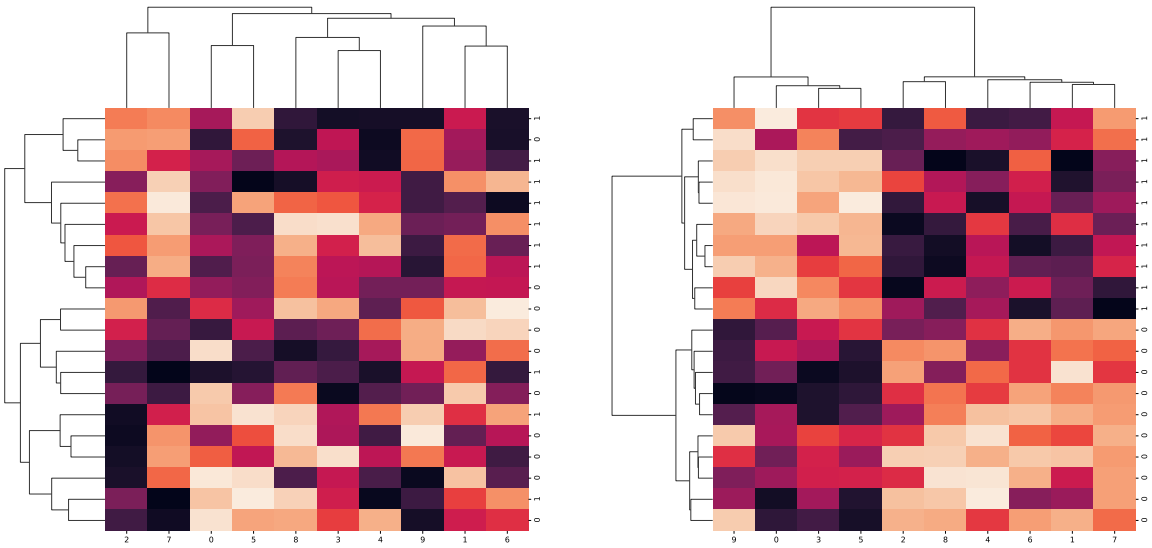


Figure 1.5: Co-clustering of a random data set with 20 instances and 10 features (left), and co-clustering of a similar data set with 10000 features, of which 10 features were selected that best correlate with a binary label (right). Notice a clear pattern of colors and shades in the right heatmap, which should not be there if correct data preprocessing procedures were applied.

especially linear models, the model's complexity easily be measured in terms of the number of parameters (or degrees of freedom). For nonparametric models the theory is more complex (*e.g.*, VapnikChervonenkis dimension) and introduces the concept of the *effective number of parameters*. Typically, nonparametric models have a higher number of effective parameters and are thus able to better fit the data but also more prone to overfitting. But they are more difficult to interpret.

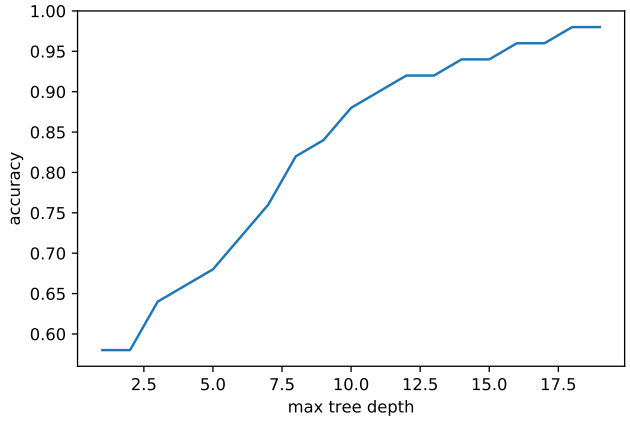


Figure 1.6: A classification tree accuracy on a random class-balanced binary classification data set with one feature and 50 data instances. Trees were grown to a specified maximal depth. More complex trees better fit the training data.

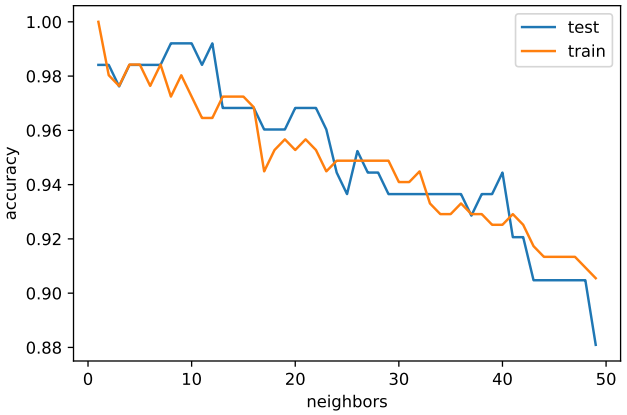


Figure 1.7: A training and test-set tradeoff for k-nearest neighbor model. On the training data, the accuracy falls with raising k, while on the test data set the accuracy peaks at around $k = 10$. Hyper-parameter estimation is one of the key issues when selecting the most appropriate model.

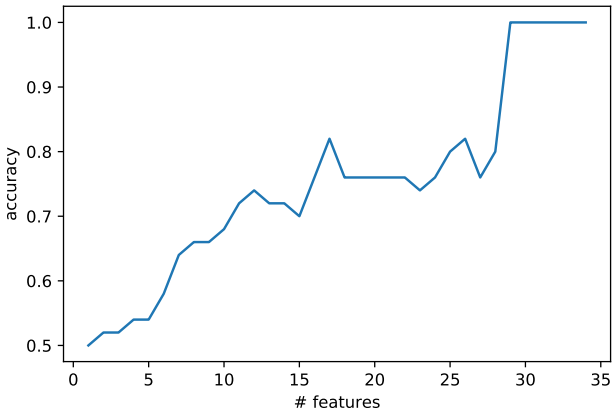


Figure 1.8: Logistic regression is more robust to overfitting than classification trees, but succombs as well when given sufficient number of features. A graph shows a training error on a random 50-instance binary classification data set when adding up to 35 features.

Practical Utility of Machine Learning Models

In practical applications, there are other dimensions (other than predictive accuracy, etc.) that we need to consider:

- *Computational aspects* include runtime complexity and resource consumption and are specifically relevant when modeling large data sets and streaming data, where models need to be adapted frequently and where there is inherent concept drift. Notice that while some computational can be mitigated with modern hardware, but we must understand that even taking into account pairwise feature interactions requires computation squared in the number of features, not even counting the number of data points. All alternatives are based on discarding some information: data subsampling (sublinear learning algorithms), feature selection, or discarding higher-order feature interactions.

- *Implementation aspects*, where data scientists need to decide which parts of the analysis procedures to implement on their own, gaining in flexibility, and for which to rely on already existing implementations. These later may also be limited in terms of data type (e.g., sparse or full), scalability (multi-core, multi-processor, or multi-GPU computing), and data access (e.g., Excel tables, SQL databases, or data in the cloud).

- *Interpretability*, which often refers to the question if the model is readable, or can it be converted to a readable format. And if it is readable, is its interpretation easy (e.g., just a few if-then-rules) or impossible (e.g., a long list of rules, or a large classification tree).

- *Explainability*, often confused with interpretability, places a model within a context of a problem domain and asks a question did we gain any new knowledge. To achieve

explainability, one would often need to combine the interpretation of the model with extra formalized knowledge about the domain (*e.g.*, feature groups, ontologies, rules, and similar).

Every modeling paradigm we introduce in this course should and will be discussed from these perspectives. Notice that most data science courses often focus on predictive accuracy alone; the intended audience may often forget that in practice, other issues are equally or even more important.

1.4.1 Curse of Dimensionality

In practice, the complexity of the models we want to fit is not bound only by computational resources but also the fact that a linear increase in the number of variables can result in exponential increases in the number of possible configurations. Therefore, the amount of data that would be required to distinguish between these configurations is impractical.

The curse of dimensionality may also inhibit, or even cripple some machine learning methods. For instance, k-nearest neighbors may work well on two-dimensional data, but as soon as the number of dimensions increases, to a few more dimensions, the algorithm fails. To illustrate this point, consider embedding a small d-dimensional cube of side s inside a larger unit cube. Let the data be uniformly distributed within the unit cube. Suppose we estimate the density of a class labels around a test point $\boldsymbol{x}$ by growing a smaller hyper-cue until it contains a desired fraction f of the data points. The expected length of this cube will be $s(f, d) = f^{1/d}$. Say, with $d = 10$ and to base our estimate on 10% of the data, the length of the smaller cube would need to be $s = 0.8$. The approach, despite the name "nearest neighbor" is no longer very local, as even with the modest feature sizes, it relies on data points that are far away. Even with 1% coverage, the size of the small cube needs to be substantial, as $s(0.01, 10) = 0.63$. With a number of features growing, we quickly have to start taking into account points that are not close or risk increasing variance.

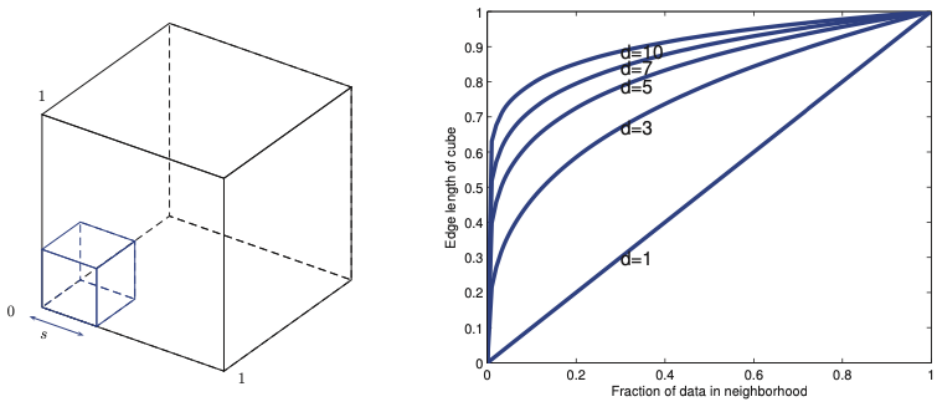


Figure 1.9: We embed a small cube within a unit cube (left) and assess a lenght of the edge of a small cube to cover a fraction of uniformly spread data. Graphs borrowed from Murphy [**2012-Murphy**].

Chapter 2

Trees and Forests

We here introduce learning via recursive partitioning of the input variable space. Depending on the learning taks, the algorithm used is classification trees or regression trees, respectively. We then describe the upgrade of the tree-learning approach with construction of a set of trees, random forests, and, as part of the latter, we introduce a more general approach of bootstrap aggregation (bagging).

2.1 Classification and Regression Trees (CART)

Classification and regression trees, somehow surprisingly, conceptually relate to other advanced machine learning approaches, such as kernel methods, generalized linear models, and adaptive basis function models. While we have yet to discuss them, let us visit them briefly for some motivation. The (generalized) linear modelling paradigm, as introduced in the next lecture, assumes that the data generating process can be interpreted with a family of distributions whose parameters are in a (transformed) linear relationship with the input variables. These are parametric models. For kernel methods, the prediction takes the form of a weighted sum $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x})$, where $\boldsymbol{w}$ is a weight vector and ϕ is a vector of similarities with an input example $\boldsymbol{x}$, such that

$$\boldsymbol{\phi}(\boldsymbol{x}) = [\kappa(\boldsymbol{x}, \boldsymbol{\mu}_1), \ldots, \kappa(\boldsymbol{x}, \boldsymbol{\mu}_n)]$$

where $\boldsymbol{\mu}_k$ are either all the training data or some sample, and κ is a kernel function. Kernel function are, in general, defined in advance, and coming up with a good kernel is hard and may depend on the problem domain.

Learning kernel functions is on option, but is computationally expensive and requires a lot of data. An alternative approach is to forget about kernels, and instead infer useful features $\boldsymbol{\phi}(\boldsymbol{x})$ directly from the training data. This is an approach used by adaptive basis function

model, which takes the form

$$f(\boldsymbol{x}) = w_0 + \sum_{m=1}^{M} w_m \phi_m(\boldsymbol{x})$$

where $\phi_m(\boldsymbol{x})$ is the m-th basis function inferred from the training data. The basis functions are parametric, so that we can write $\phi_m(\boldsymbol{x}) = \phi_m(\boldsymbol{x}; \boldsymbol{v}_m)$, where $\boldsymbol{v}_m$ are the parameters of the basis function itself. The CART approach can be viewed as a special case of adaptive basis function model. CART recursively partitions the input space and defines a simplified local model in each resulting region. Recursive partitioning can be represented as a tree, where partitioning conditions are stored in internal nodes and region models in the leaves. The model takes the following form

$$f(\boldsymbol{x}) = \mathbb{E}[y|\boldsymbol{x}] \tag{2.1}$$

$$= \sum_{m=1}^{M} w_m \mathbb{I}(\boldsymbol{x} \in R_m) \tag{2.2}$$

$$= \sum_{m=1}^{M} w_m \phi(\boldsymbol{x}; \boldsymbol{v}_m) \tag{2.3}$$

where R_m denotes the m'th regio and w_m is, simplified, the man response in the region. The set $\boldsymbol{v}_m$ encodes the choice of the variable to split on and the related threshold value in the path from the root of the tree to specific leaf. Notice that in CART the regions do not overlap, and that the training example falls in only and exactly one of the constructed regions. The region splits are defined on exactly one of the variables and are thus axis parallel.

Basic Idea

From the viewpoint of model construction and compared to generalized linear models, kernel methods, and inference of adaptive basis function models, CART introduces a fundamentally different modelling paradigm. One that assumes that the data generating process can be interpreted as a partition of the input variable space into homogeneous (pure) regions – regions where there is little or no uncertainty left about the target variable. For regression, the target variable for the data instances within this region is almost constant (see Fig. 2.1). For classification, a majority of data instanes in the region have the same value of the target variable.

The CART Algorithm

Finding the optimal partitioning of the input variable space is in general NP-complete, even if using axis-parallel splits only. That is, it is infeasible to check all possible partitions. Instead, we will consider a greedy algorithm (CART) that is based on binary recursive partitioning of

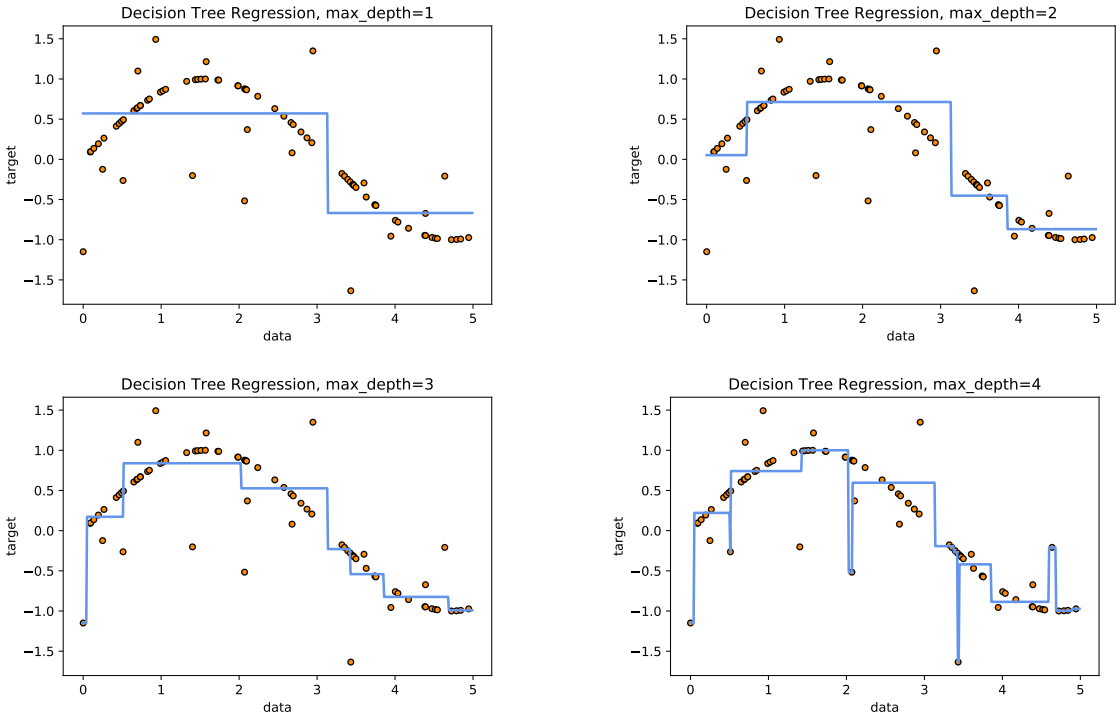


Figure 2.1: Regression trees fitted on data generated by a sine function with some noise. While the tree adapts well to the training data, its ability to overfit the training data is visible already with trees with of maximum depth of 4 (lower right).

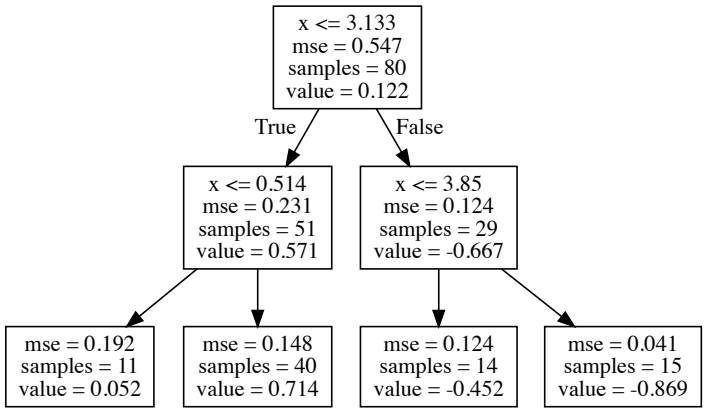


Figure 2.2: A regression tree with maximum depth of 2 from the data from Fig. 2.1.

the input space, at each step choosing the best possible split (according to some pre-selected criterion). Notice that this algorithm does not use any look-ahead, and while such algorithms were studied in the literature, they are not used in practice. A simplified and abstracted CART algorithm is encoded as Algorithm 1.

Algorithm 1 CART

```
1: procedure FITTREE(D)
2:     (D_L, D_R, criterion) ← split(data)
3:     node ← createNode(criterion, data)
4:     if (stoppingCriterionMet(...)) then return node
5:     node.L ← fitTree(D_L)
6:     node.R ← fitTree(D_R)
7:     return node
```

The CART algorithm uses several functions that require explanation:

- *createNode()*: This function creates an object that represents a tree node, which essentially stores the criterion on which the data in the node is split, and a possible reference to the data instances that are pertinent to the node. If a suitable node split is found, the node stores the information on its to siblings. Note that, as introduced above, the CART algorithm would construct binary trees.

- *split()*: The assumption here is that features are numerical or at least ordinal. We order every feature based on possible splits (based on unique values in the data, so we have a finite number of possible splits). And then we basically go through all possible feature-split combinations to find the one that is optimal according to our splitting criterion – the one that minimizes the sum of the cost of the left and right subtrees. Possible splitting criterions are discussed below.

- *stoppingCriterionMet()*: The stopping condition, also referred as *pre-prunning* of the trees, can be one or more of the following:

  - The partition is sufficiently homogeneous/pure. In particular, there is no point in splitting further if we have perfect homogeneity (all observations have the same value).

  - The gain Δ of splitting the data set in the current node (relative to stopping criterion) is bellow some pre-determined threshold, where

$$\Delta = \text{cost}(\mathcal{D}) = \text{cost}(\mathcal{D}) - (\frac{|\mathcal{D}_L|}{|\mathcal{D}|}\text{cost}(\mathcal{D}_L) + \frac{|\mathcal{D}_R|}{|\mathcal{D}|}\text{cost}(\mathcal{D}_R))$$

  - The algorithm has reached pre-determined maximum tree depth.

– Splitting the data set in the node would yield a leaf with number of observations below some pre-determined minimum.

Choice of the Splitting Criterion

At each internal node, the inference methods for the trees splits the training dataset $\mathcal{D}$ pertinent to the node to maximize some splitting criterion. The split is performed using a single feature from the training data set and forming a condition on the value of this feature that evaluates to true or false. According to this condition, the data $\mathcal{D}$ is then split to two datasets, each pertinent to one of the two siblings of the node. This type of splitting results in a binary tree. Notice that other, non-binary, splitting mechanisms could be used, but they would lead to over-fragmentation of the data, increase the variance, and lead to increased overfitting.

Splitting criteria are related to dataset purity, costs, loss, or estimated errors, and have to specifically address the type of the target feature, this being either numerical or discrete. Note that since the introduction of classification and regression trees, many different criteria were propose and while, at least on the surface, these take different forms, the practical differences regarding overall accuracies and ordering of the features are often neglectable. The costs of the splitting is most often estimated for each of the resulting siblings (leaves), and then weigthed according to the estimated probability that the data instance will fall in one of the two constructed regions

$$\text{cost}(\textit{node}, \textit{criterion}) = \frac{|\mathcal{D}_L|}{|\mathcal{D}|}\text{cost}(\mathcal{D}_L) + \frac{|\mathcal{D}_R|}{|\mathcal{D}|}\text{cost}(\mathcal{D}_R))$$

For regression trees, the most often used splitting criterion is the mean squared error of predicting with the subtree mean

$$\textit{cost}(\mathcal{D}) = \sum_{i \in \mathcal{D}} (y_i - \overline{y})^2$$

where $\overline{y} = \frac{1}{|\mathcal{D}| \sum_{i \in \mathcal{D}} y_i}$ is the mean of the target variable in the resulting dataset.

Many more splitting criterion were proposed for the classification setting, and most of them rely on estimating class-conditional probabilities

$$\hat{\pi}_c = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \mathbb{I}(y_i \equiv c)$$

For instance, we can measure the *entropy* (or *deviance*) of the resulting dataset

$$\mathbb{H}(\hat{\boldsymbol{\pi}}) = -\sum_{c=1}^{C} \hat{\pi}_c \log \hat{\pi}_c$$

or can measure the expected error rate in the form of a *gini index*

$$\sum_{c=1}^{C} \hat{\pi}_c(1 - \hat{\pi}_c) = \sum_c \hat{\pi}_c - \sum_c \hat{\pi}_c^2 = 1 - \sum_c \hat{\pi}_c^2$$

where $\hat{\pi}_c$ is the probabilty a random entry in the leaf belongs to class c, and $1 - \hat{\pi}_c$ is the probability for this entry to be misclassified. Other criterion may include information gain, information gain ratio, chi-squared test, and similar. Note that with all the above criteria, splitting the training data set will never decrease the quality (and increase the cost) and in the worst case the quality will remain the same if the node's data set is already homogeneous. Notice that we are estimating all the costs on the training set and thus potentially overfitting the data.

Discussion

There are seevral issues with growing and using the classification and regression trees. The trees have some advantages many disadvantages. While, on their own, the trees are rather mediocre predictors, their enhancements in terms of ensambling discussed in the following sections of this chapter ellevate them to at least a formidable baseline, if not state-of-the-art approch. Therefore, let us first review some of the issues that are pertinent to development and utility of CART, that is, induction of single trees.

Interpretation. Decision trees are easy to interpret. In fact, according to the current research, the interpretability of the trees is only behind decision tables and individual rules when it comes to non-expert users. This is somewhat marred by the fact that the standard decision tree algorithms are susceptible to changes in the inputs. A small change in the training data set can result in a substantially different tree. What good is an in-depth interpretation of the model if this is inherently unstable? We can mitigate instability by using bootstrapping to check if the algorithm produces stable trees before proceeding with the analysis. Also, learning a (stable) tree that mimics a more complex model such as a tree ensemble and neural networks is one of the most common approaches to explaining how the complex model works. This approach, though, has gained recent criticism that if one is after the explanation, it should primarily build interpretable models in the first place, and not represent complex models with simple ones [**Rudin2019**].

Low computational complexity. Trees are fast to training and very fast in prediction. They scale well to large data sets. The only exception to this observation is in treatment of the sparse data, that is, data with many unknown values. A thorough treatment of unknown values may invalidate the divide-and-conquer approach with the passing of full data sets to leaves and potentially visiting the entire tree when predicting. A

potential remedy of this side effect is to impute the missing values before training or prediction.

Poor inductive bias. Compared to more sophisticated methods, including ensembles of trees and neural networks, classification and regression trees have a relatively weak inductive bias. That is, they will not perform the best (or close to) in terms of predictive quality on most practical problems. The two main issues are a *lack of smoothness* and *difficulty of capturing additive relationships*. See Hastie *et al.* for further details.

Possible complex treatment of categorical input variables. When splitting a predictor having q possible unordered values, there are $2^{q-1}-1$ possible partitions of the q values into two groups and the computations become prohibitive for large q. For example, consider the treatment of postal codes in the data sets. There are possible heuristic approaches to cope with such cases, though. For binary target variables, we can order the predictor classes according to the proportion falling in outcome class 1. Then we split this predictor as if it were an ordered predictor. One can show this gives the optimal split, in terms of cross-entropy or Gini index, among all possible splits. This result also holds for a quantitative outcome and squared error lossthe categories are ordered by increasing the mean of the outcome. The proof for binary outcomes is given in Breiman et al. [**Brieman1984**] and Ripley [**Ripley1996**]; the proof for quantitative outcomes can be found in Fisher [**Fisher1958**]. For multicategory outcomes, no such simplifications are possible, although various approximations have been proposed [**Loh1988**].

The partitioning algorithm tends to favor categorical features with many values; the number of partitions grows exponentially in q, and the more choices we have, the more likely we can find an (arbitrarily) good one for the data at hand. This can lead to severe overfitting if q is significant, and such variables should either be avoided or some preprocessing by means of a grouping of similar feature values, such as clustering, should be used. Also, note that dummy (one-hot) encoding of categorical variables can lead to the opposite problem of individual binary variables not being selected over many features represented encoded variables.

The benefits of binary splits. Rather than splitting each node into just two groups at each stage, we might consider multiway splits into more than two groups. While this can sometimes be useful, it is not a good general strategy. The problem is that multiway splits fragment the data too quickly, leaving insufficient data at the next level down. Hence we would want to use such splits only when needed. Since multiway splits can be achieved by a series of binary splits, the latter is preferred.

treatment of missing values. Suppose our data has some missing predictor values in some or all of the variables. We might discard any observation with some missing values, but this could lead to severe depletion of the training set. Alternatively, we might try to

fill in (impute) the missing values, with say the mean of that predictor over the non-missing observations. For tree-based models, there are two better approaches. The first is applicable to categorical predictors: we make a new category for missing. From this, we might discover that observations with missing values for some measurement behave differently than those with non-missing values. The second more general approach is the construction of surrogate variables. When considering a predictor for a split, we use only the observations for which that predictor is not missing. Having chosen the best (primary) predictor and split point, we formed a list of surrogate predictors and split points. The first surrogate is the predictor and corresponding split point that best mimics the split of the training data achieved by the primary split. The second surrogate is the predictor and relevant split point that does second best, and so on. When sending observations down the tree either in the training phase or during prediction, we use the surrogate splits in order, if the primary splitting predictor is missing. Surrogate splits exploit correlations between predictors to try and alleviate the effect of missing data. The higher the correlation between the missing predictor and the other predictors, the smaller the loss of information due to the missing value.

Tree pruning. If the tree is allowed to grow until the leaves are entirely (or nearly) homogenous, we are likely to be overfitting. In some cases that is desirable - we will see such an example later with random forests, where we want an individual tree in the ensemble to be very biased. However, in most cases, it is not. To prevent overfitting, we can carefully tune the stopping criteria. However, growing the entire tree and then post-processing it by *pruning* individual branches can sometimes lead to better results. The basic idea is to check each split if not making that split would not result in a significant increase in error. Additionally, we can use cross-validation to prune based on an estimate of the generalization error, making the process more robust. Note that cross-validation could (should), in theory, also be used when growing the tree. The reason why we make splits based on what is essentially training set error is that cross-validation would be computationally infeasible in most practical scenarios.

Model trees. An alternative to report on average values in the tree leaves, we can use non-trivial models. Many approaches combine trees with generalized linear (additive) models in the leaves. This can lead to improved results in problems that are a combination of crisp rules and (local) linear behavior while retaining most of the interpretability. However, it comes at the cost of computational complexity because it requires a more complex model evaluation when splitting the tree.

Oblique feature space splitting. Axis-parallel partitioning**: Most tree-based algorithms (including the one described above) limit themselves axis-parallel splits. This can lead to very complicated trees if the boundaries between homogeneous regions do not follow this assumption. As an alternative, non-axis-parallel (oblique) algorithms have

been developed. However, this comes at the cost of interpretability and computational complexity.

2.2 Bagging

Before we proceed with random forests, we will first introduce a component of random forests that has more general applicability. *Bagging* (Bootstrap Aggregation) is a technique that can improve the predictive quality of any models, in particular when the data set is small and/or we are dealing with a high-variance model that can easily overfit the training data. A prime example of such a model is a non-pruned tree.

The basic idea is straightforward: instead of using our model $\hat{f}$ that was trained on all the training data, we take B bootstrap samples of the training data and re-train the model on each sample, resulting in B models $\hat{f}_b$. The bootstrapped prediction is the aggregate (average) of the individual bootstrap models:

$$\hat{f}_{\text{boot}}(\boldsymbol{x}) = \sum_{b=1}^{B} \frac{1}{B} \hat{f}_b(\boldsymbol{x}).$$

In essence, we are using the bootstrap, where the functional of the data is the model's prediction for $\boldsymbol{x}$. And, as we already know, the sampling error can be made arbitrarily small by increasing B.

Why dDoes Bagging Work?

Note that most of the arguments we state here are from Grandvalet [**Grandvalet2004**]. Some authors, including Hastie *et al.* [**2016-Hastie**], claim that the bagging estimate will be the same as the original model if the model is linear. That does not imply that bagging will produce the same estimate if used on linear regression. Overall, there is little rigorous theoretical justification of why bagging should work, but there is ample empirical evidence that it often does work. Here we will offer some empirical justification for the underlying mechanisms that make bagging work (and sometimes fail).

Grandvalet (2004) argues that bagging equalizes the influence of individual points on the prediction. As the most influential points (points with high leverage) are typically outliers and have a bad influence on predictive quality, reducing their influence will improve performance by reducing the variance. This is a more general explanation to the more common explanation that bagging improves predictions because it reduces variance, in particular, because bagging can also increase variance. That is, if points of high leverage have a positive influence, bagging will decrease predictive quality.

One implication of the above is that models where all points have the same or similar leverage would not benefit from bagging. Similarly, models where a single point has very

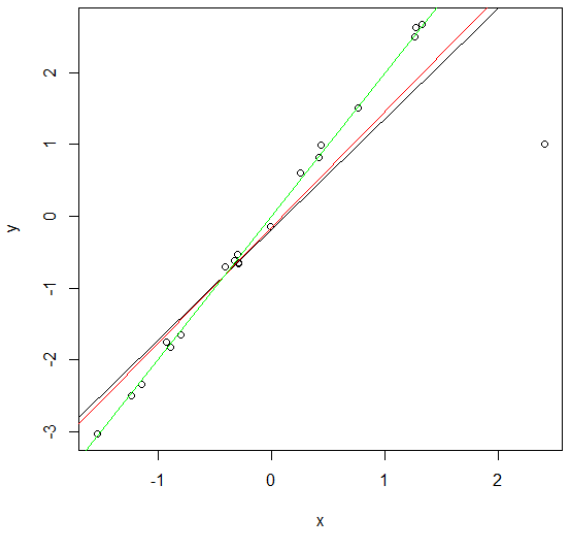
little effect on the prediction would also not benefit from bagging (robust models such as regularized regression or models that already contain some sort of bagging, such as random forests, which we discuss below). Therefore, high-variance models, such as non-pruned trees, is where we would expect the most benefit.

A prototypical example of where all points have the same leverage (and bagging does nothing) is predicting with the training set average. With enough bootstrap samples, every point will be included in the bootstrap sample approximately the same number of times and every point has the same influence. Indeed, the bootstrap prediction will be approximately the same as the prediction of the model that uses the entire training set.

In general, every point will be included in the bootstrap sample approximately the same number of times, but what is at first maybe even somewhat surprising, not every point has the same influence on the prediction. The fact that some points have more *leverage* on a prediction can be illustrated with simple linear regression, where points further away from the centre of mass (x-axis only) have more leverage.

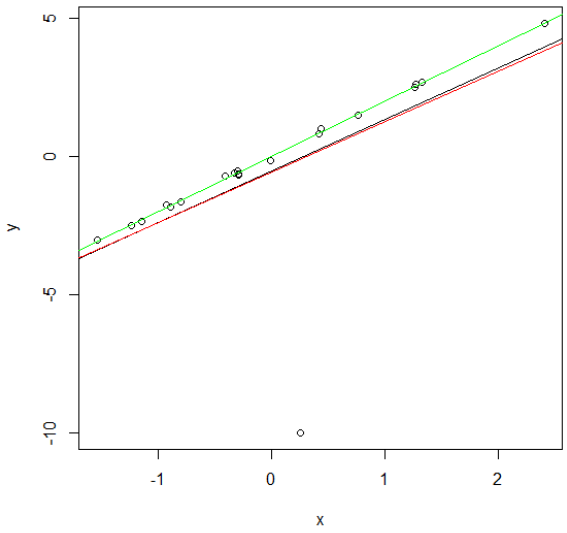
Example: Bagging on outliers, #1

The outlier (bad point) is a high-leverage point, hence bootstrapping improves performance. Points in green denote true data generating process mean, points in black denote predictions by linear regression, and points in red predictions by bootstrapped linear regression.



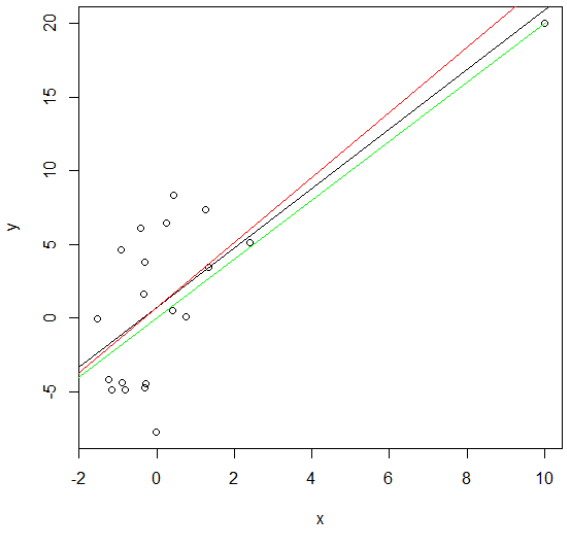
Example: Bagging on outliers, #2

The outlier (bad point) is a low-leverage point. Bootstrapping gives it more influence, slightly decreasing performance.



Example: Bagging on outliers, #3

The outlier (this time it's a good point) is a high-leverage point. Bootstrapping gives it less influence, slightly decreasing performance.



2.3 Random Forests