Machine Learning for Data Science 1

(lecture notes, only for internal use)

Blaž Zupan, Erik Štrumbelj

March 12, 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 $p(y|x_i, \mathcal{D})$ for predictive, and formulations of the form $p(y|\mathcal{D})$ for unsupervised learning. Resulting probabilistic models $p(y|\theta)$ or $p(y|x_i;\theta)$ 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). Recommended additional reading is Chapter 2 from Hastie, Tibshirani, and Friedman (2016).

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 provides 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: $\{y_i, \mathbf{x}_i\}_{i=1}^n$. This is *supervised learning*, also known as *predictive learning* or predictions. Here, y is referred to as *response variable*, and $\mathbf{x}$ 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 $p(y|\mathbf{x})$ so that we can make predictions for future or unobserved values of independent variable y for any combination of dependent variables $\mathbf{x}$ (see Table 1.1).

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. 1.1). In essence, we are trying to group, or cluster, observations.

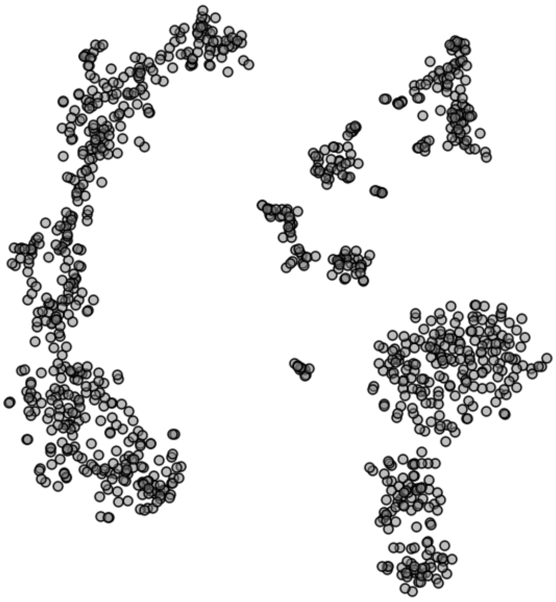


Figure 1.1: A two-dimensional visualisation of blood cells, originally described with expressions of thousands of features. The visualisation was constructed using t-SNE dimensionality reduction, and exposes potential clusters that need to be further analyzed.

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
> $\mathcal{H} = \left\{p(y|\boldsymbol{x}, \boldsymbol{\beta}, \alpha, \sigma) = dnorm(\boldsymbol{\beta x} + \alpha, \sigma^2), \boldsymbol{\beta}, \alpha \in \mathbb{R}, \sigma > 0\right\}$

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

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 $f(x) = y_i$ of point x nearest to x_i, according to a chosen distance metric$\}$ (see Fig. 1.3)

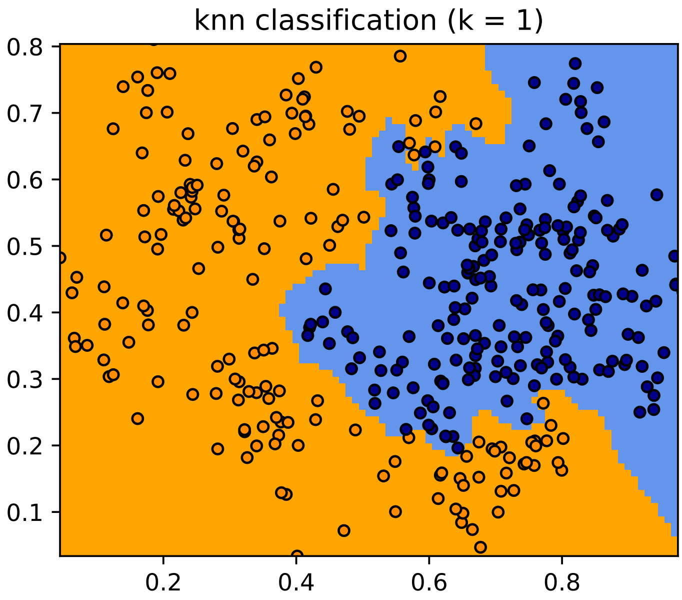


Figure 1.2: Decision boundary of a 1-nearest neighbor model trained on a two-featured binary classification data set with 380 data instances (170 in one class and 210 in the other) as shown on a figure. The decision boundary is complex and may substantially change with any addition on removal of the 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 (Wolpert and Macready, 2005).

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.

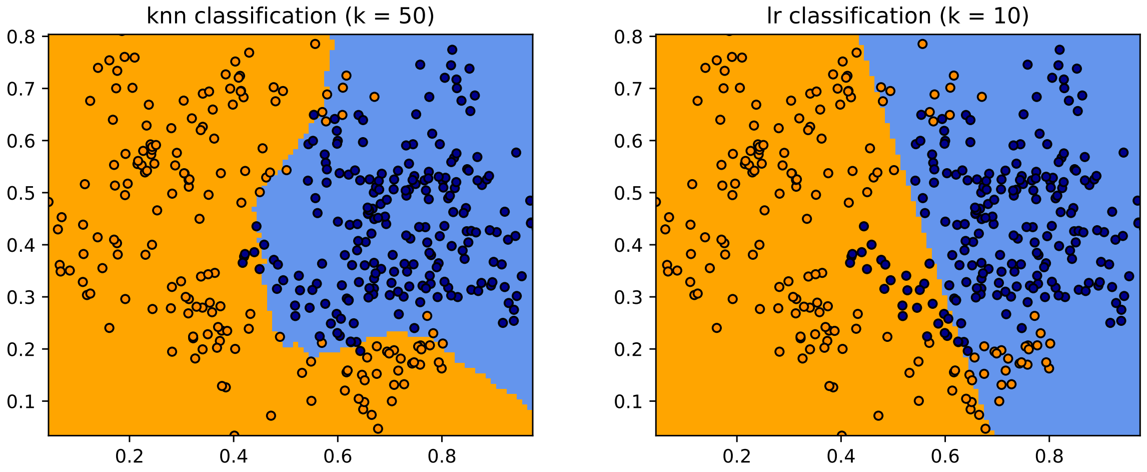


Figure 1.3: Decision boundary on a two-featured binary classification data set as inferred by nearest neighbor algorithm with K=50 and by logistic regression. Which model would perform better on new data?

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 (Simon et al., 2003). 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. (Vandewiele et al., 2019). 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 Figs. 1.6 and 1.7), but the "right" complexity of the model may depend on the amount of data we have (see Fig. 1.8). In parametric

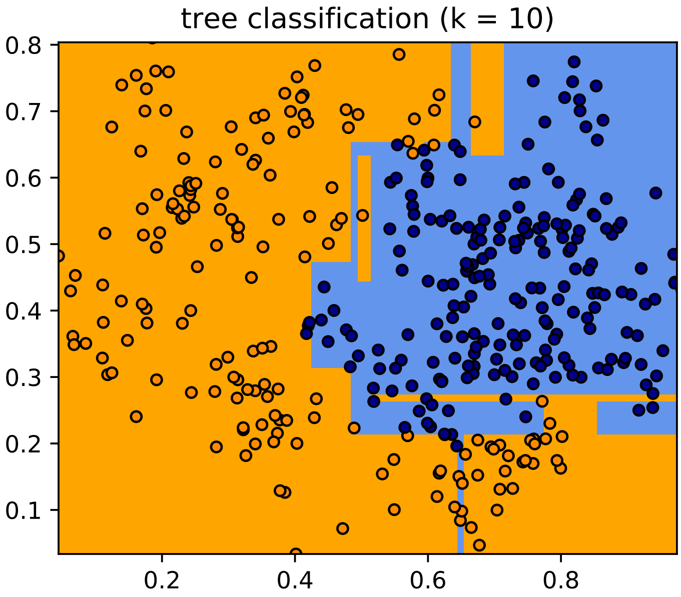


Figure 1.4: Classification trees would often overfit the training data. Figure shows decision boundary of a tree where the allowed maximum tree depth was 10. The decision boundary is complex and often covers single-case exceptions.

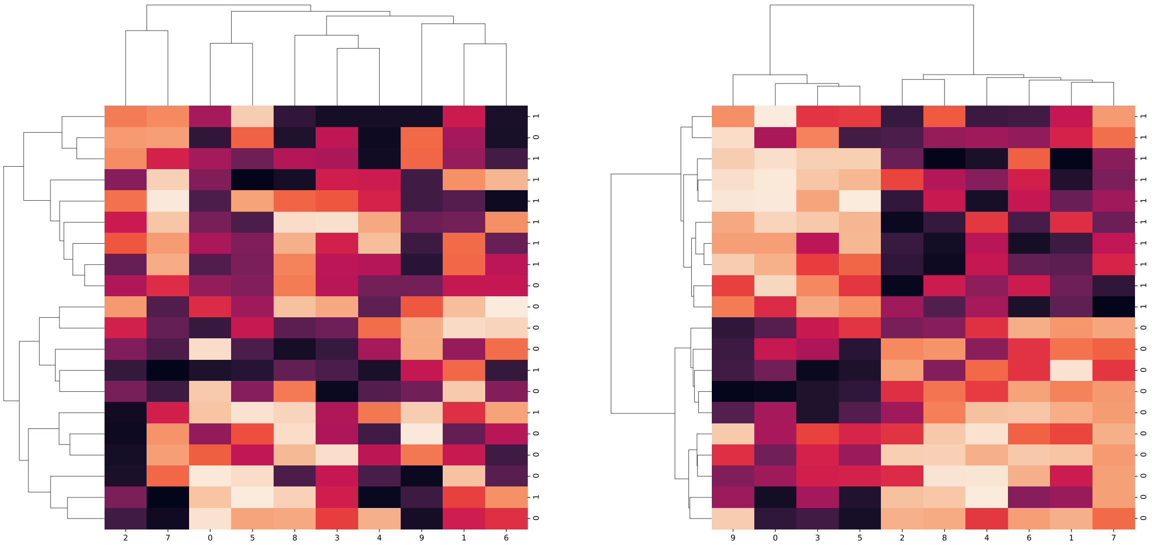


Figure 1.5: Co-clustering of a random data set with 20 instances and 10 features (left), and co-clustering of a similar data set with 10000 features, of which 10 features were selected that best correlate with a binary label (right). Notice a clear pattern of colors and shades in the right heatmap, which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

models, 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.*, Vapnik–Chervonenkis 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.

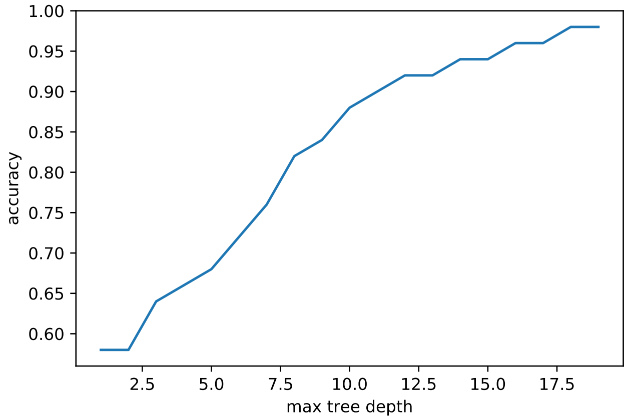


Figure 1.6: A classification tree accuracy on a random class-balanced binary classification data set with one feature and 50 data instances. Trees were grown to a specified maximal depth. More complex trees better fit the training data.

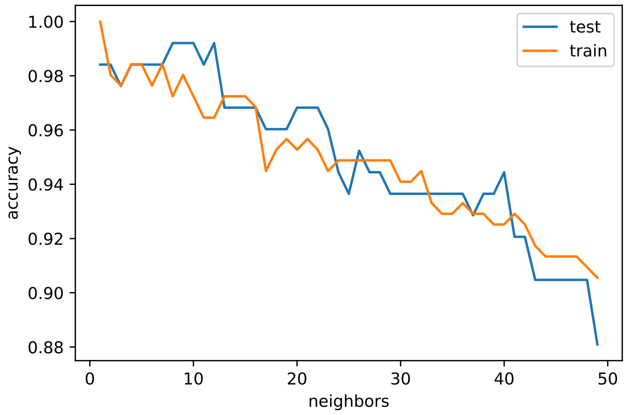


Figure 1.7: A training and test-set tradeoff for k-nearest neighbor model. On the training data, the accuracy falls with raising k, while on the test data set the accuracy peaks at around $k = 10$. Hyper-parameter estimation is one of the key issues when selecting the most appropriate model.

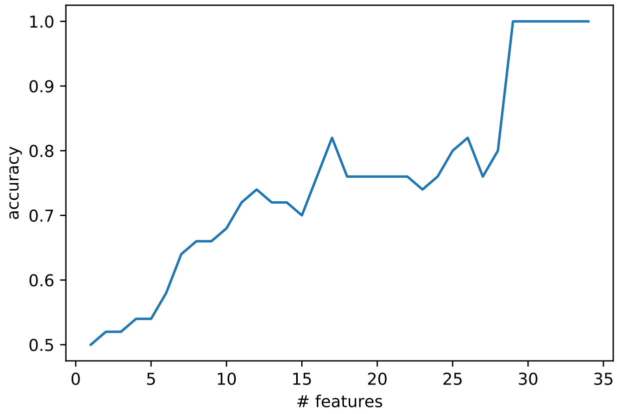


Figure 1.8: Logistic regression is more robust to overfitting than classification trees, but succombs as well when given sufficient number of features. A graph shows a training error on a random 50-instance binary classification data set when adding up to 35 features.

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.

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 (Fig. 1.9). Let the data be uniformly distributed within the unit cube. Suppose we estimate the density of a class labels around a test point 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.

References

Hastie, T., R. Tibshirani, and J. Friedman (2016). *The Elements of Statistical Learning*. 2nd ed. Springer.

Murphy, K. P. (2012). *Machine Learning: a Probabilistic Perspective*. 1st ed. The MIT Press.

Simon, R. et al. (2003). "Pitfalls in the use of DNA microarray data for diagnostic and prognostic classification". In: *J. Natl. Cancer Inst.* 95, pp. 14–18.

Vandewiele, G. et al. (2019). "A Critical Look at Studies Applying Over-Sampling on the TPEHGDB Dataset". In: *Proc. Artificial Intelligence in Medicine*. Springer, pp. 355–364.

Wolpert, D.H. and W.G. Macready (2005). "Coevolutionary free lunches". In: *IEEE Transactions on Evolutionary Computation* 9.6, pp. 721–735.

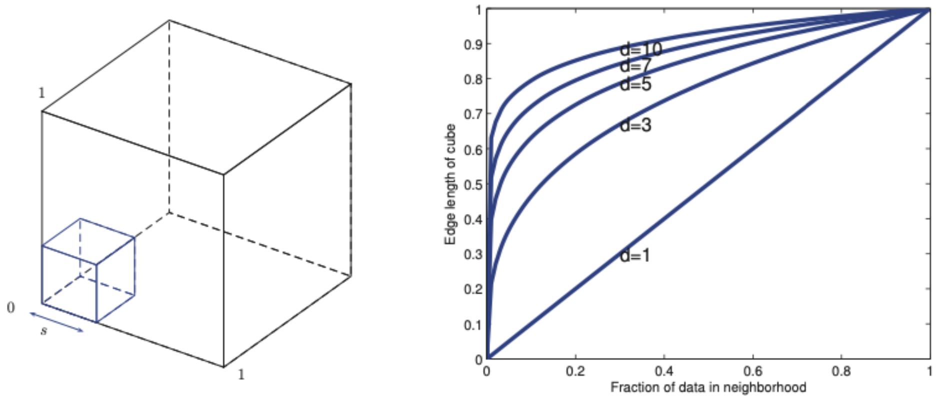


Figure 1.9: We embed a small cube within a unit cube (left) and assess a lenght of the edge of a small cube to cover a fraction of uniformly spread data. Graphs borrowed from Murphy (2012).

Chapter 2

Trees and Forests

Trees introduce learning via recursive partitioning of the input variable space. Depending on the learning task, the algorithm used is either a classification tree or a regression tree, respectively. The inference of trees is fast but leads to models that are not stable and have high variance. To reduce the variance and increase stability, the upgrade of the tree-learning approach may construct a set of trees. We define two such procedures; one called bootstrap aggregation (bagging) and the other random forest.

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 $f(x) = \boldsymbol{w}^\intercal \boldsymbol{\phi}(x)$, where $\boldsymbol{w}$ is a weight vector and ϕ is a vector of similarities with an input example $\boldsymbol{x}$, such that

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

where $\boldsymbol{\mu}_k$ are either all the training data or some sample, and κ is a kernel function. Kernel functions 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

$$\begin{aligned} f(\boldsymbol{x}) &= \mathbb{E}[y|\boldsymbol{x}] && (2.1)\\ &= \sum_{m=1}^{M} w_m \mathbb{I}(\boldsymbol{x} \in R_m) && (2.2)\\ &= \sum_{m=1}^{M} w_m \phi(\boldsymbol{x}; \boldsymbol{v}_m) && (2.3) \end{aligned}$$

where R_m denotes the m'th region and w_m is, simplified, the main 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 the 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 instances 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,

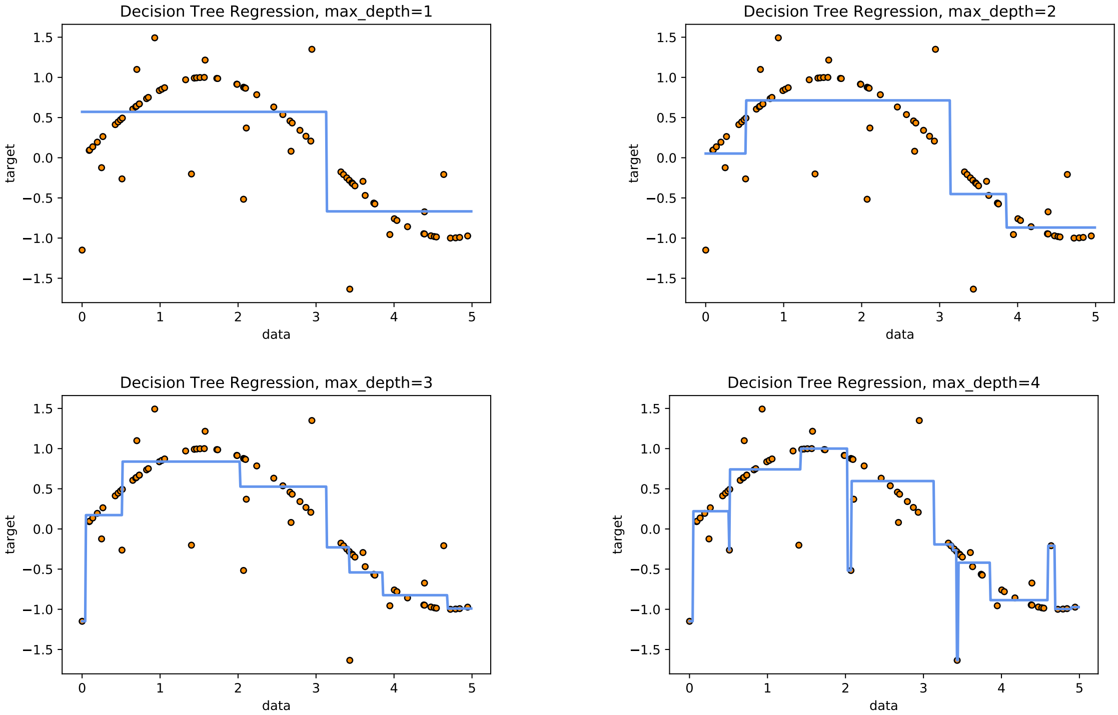


Figure 2.1: Regression trees fitted on data generated by a sine function with some noise. While the tree adapts well to the training data, its ability to overfit the training data is visible already with trees with of maximum depth of 4 (lower right).

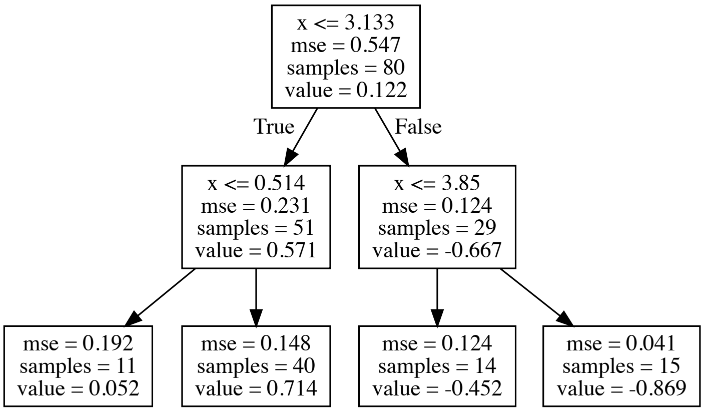


Figure 2.2: A regression tree with maximum depth of 2 from the data from Fig. 2.1.

we will consider a greedy algorithm (CART) that is based on binary recursive partitioning of 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

```
procedure FITTREE($\mathcal{D}$)
    $(\mathcal{D}_L, \mathcal{D}_R, criterion) \leftarrow split(\mathcal{D})$
    $node \leftarrow createNode(criterion, \mathcal{D})$
    if stoppingCriterionMet(criterion, $\mathcal{D}$) then return node
    $node.L \leftarrow fitTree(\mathcal{D}_L)$
    $node.R \leftarrow fitTree(\mathcal{D}_R)$
    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 criteria are discussed below.

- *stoppingCriterionMet()*: The stopping condition, also referred to 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 below some pre-determined threshold, where

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

    - 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 data set $\mathcal{D}$ 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 data sets, 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 data set 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 proposed 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 weighted according to the estimated probability that the data instance will fall in one of the two constructed regions

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

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

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

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

Many more splitting criteria 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 data set

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

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 several issues with growing and using the classification and regression trees. The trees have some advantages and many disadvantages. While, on their own, the trees are rather mediocre predictors, their enhancements in terms of ensembling discussed in the following sections of this chapter elevate them to at least a formidable baseline, if not state-of-the-art approach. 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 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, one should primarily build interpretable models in the first place, and not represent complex models with simple ones (Cynthia, 2019).

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 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.

Weak 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, Tibshirani, and Friedman, 2016) 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 loss—the categories are ordered by increasing the mean of the outcome. The proof for binary outcomes is given in Breiman et al. (Breiman et al., 1984) and Ripley (Ripley, 1996); the proof for quantitative outcomes can be found in Fisher (Fisher, 1958). For multicategory outcomes, no such simplifications are possible, although various approximations have been proposed (Loh and Vanichsetakul, 1988).

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 include little modelling bias. 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 go over each split and check 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. As an alternative to reporting on average values in 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 to 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 Does Bagging Work?

Note that most of the arguments we state here are from (Grandvalet, 2004). Some authors, including (Hastie, Tibshirani, and Friedman, 2016), 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.

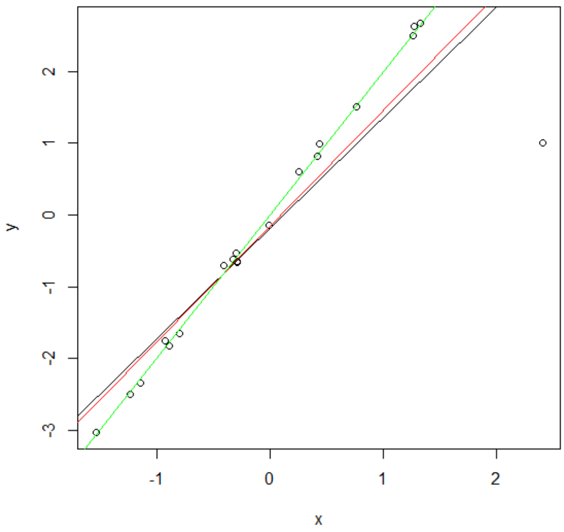
One implication of the above is that models, where all points have the same or similar leverage, would not benefit from bagging. Similarly, models, where a single point has very little effect on the prediction, would also not benefit from bagging (robust models such as regularized regression or models 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 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 center of mass (x-axis only) have more leverage.

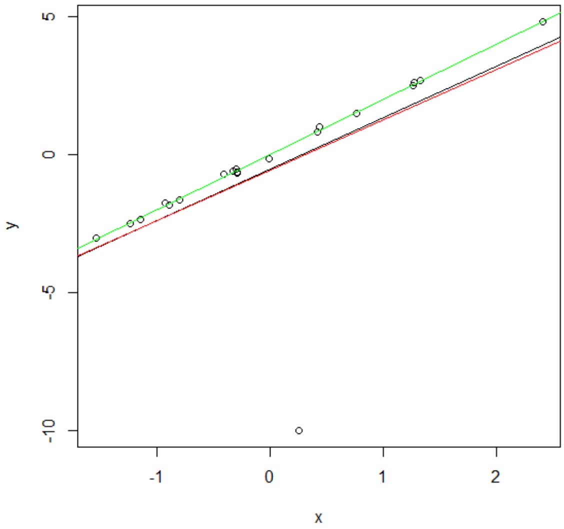
Example: Bagging on outliers, #1

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.



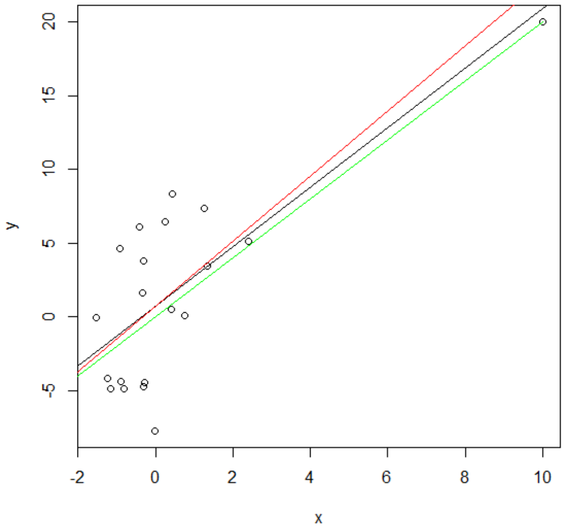
Example: Bagging on outliers, #2

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

Random forests (Brieman, 2001) extend the idea of bagging but aim to develop even more decorrelated trees than those from bootstrap samples. The approach develops a possibly large collection of trees $\{T_b\}_1^B$, where each tree is inferred from a bootstrap sample of the training data set. To additionally diversify the trees, the features on which to split each internal node of the tree are selected from a random sample of p variables. This is different from the normal growth of the trees which instead considers the entire set of predictors. Here, p is a user-specified parameter. To make a prediction of a new data point $\boldsymbol{x}$, we either average the predictions of individual trees in a case of regression,

$$\hat{f}_{\mathrm{rf}}^{B}(\boldsymbol{x}) = \frac{1}{B}\sum_{b=1}^{B} T_b(\boldsymbol{x})$$

or choose a class using a majority vote in the case of classification.

Trees are ideal for the described averaging procedure. If they are grown sufficiently deep, they have a relatively low bias. As they are notoriously noisy, they can benefit from averaging. Since each tree in bagging is identically distributed, the expectation of an average of B such trees is the same as the expectation of any of them. The bias of the bagged trees is the same as that of the individual trees. Hence, in random forests, it is recommended that the trees are not pruned but instead developed to the depth.

The benefits of trees can also be examined from the viewpoint of variance. Notice that the average of B i.i.d. random variables, each with a variance of σ^2, has a variance

$$\frac{1}{B}\sigma^2.$$

If the variables are only identically distributed but not necessarily independent with a positive pairwise correlation of ρ, the variance of the average is

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2.$$

The aim of the random forest is to reduce the variance. We see that with a large number of the trees and hence large values of B we decrease the value of the second term in the variance as expressed above. The first term, $\rho\sigma^2$, can then only be minimized by minimizing ρ. Hence, we prefer the trees that are different, and whose correlations in predictions is minimized. We, of course, prefer accurate trees, but those whose precision is focused on different parts of the parameter space. For this reason, besides bootstrap sampling, random forests engage extra randomization procedures, like arbitrarily choosing p features when examining which feature to use at each split. In practice, p can be relatively small and equal to $p = \sqrt{D}$ for classification and $p = D/3$ for regression.

Random forests do remarkably well in terms of accuracy, with very little or no tuning required (Fernández-Delgado, Carnadas, and Barro, 2014). Just like trees, they require almost no data preprocessing, can treat both continuous and discrete features, and can easily handle missing values. The inference of trees is fast and can be applied to any reasonably sized data set. With these characteristics, random forests are a great baseline, that is, provide accuracies that need to be surpassed by more advanced approaches.

Out-of-Bag Estimates

Bootstrap sampling, on the average, leaves $e^{-1} = 0.368$ of data instances out of sample. An out-of-bag estimate is the mean prediction error on each training sample x_i, using only the trees that did not have x_i in their bootstrap sample. With a fixed number of trees B in the forest, this estimate would converge to the estimate we would obtain through, say, cross-validation. Alternatively, we can use the out-of-bag estimate to observe the convergence of estimated error and stop the growth of the trees when the error stabilizes. In practice, forests are usually grown to include up to a few hundreds of trees.

Estimate of Feature Importance

One of the deficiencies of random forests is their overall complexity. If we agree that the trees are models that can be read and interpreted, we lose this ability with the forest simply because of the large collection of trees. With forests, interpretability is lost. To remedy the loss of interpretability, the author of the forests, Brieman (2001), proposes to provide estimates of the importance of features in the forests using out-of-bag estimates. The procedure randomly permutes the value of a selected feature and estimates the out-of-bag error. The decrease of accuracy caused by random permutation now provides an estimate of the feature's importance. Notice that estimates obtained in this way can be substantially different from univariate estimates of the correlation between a feature and a class variable, taking into account possible feature interactions discovered by the trees in the forest.

References

Breiman, L. et al. (1984). *Classification and Regression Trees*. Chapman and Hall/CRC.

Brieman, L. (2001). "Random Forests". In: *Machine Learning* 45, pp. 5–32.

Cynthia, R. (2019). "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead". In: *Nature Machine Intelligence* 1.5, pp. 206–215.

Fernández-Delgado, Manuel, Eva Carnadas, and Sanén Barro (2014). "Do we need hundreds of classifiers to solve real world classification problems?" In: *Journal of Machine Learning Research* 15, pp. 3133–3181.

Fisher, W.D. (1958). “On Grouping for Maximum Homogeneity”. In: *Journal of the American Statistical Association* 53.284, pp. 789–798.

Grandvalet, Y. (2004). “Bagging equalizes influence”. In: *Machine Learning* 55.3, pp. 251–270.

Hastie, T., R. Tibshirani, and J. Friedman (2016). *The Elements of Statistical Learning*. 2nd ed. Springer.

Loh, W.Y. and N. Vanichsetakul (1988). “Tree-structured classification via generalized discriminant analysis”. In: *Journal of the American Statistical Association* 83.493, pp. 715–725.

Ripley, B. D. (1996). *Pattern Recognition and Neural Networks*.

Chapter 3

Feature Selection and Model Regularization

> **The data may contain features that are either redundant or irrelevant, and their removal may have no or only small effect on model accuracy. The reduction of feature space may also help us avoid overfitting. By selecting the most informative features, we may reduce running times and computational complexity and increase the interpretability of results due to the inference of simpler models. Three main approaches to feature selection use filter, wrapper, and embedded methods. In the filter approach, we select the most informative features before modeling. Wrapper methods select features according to the observed performance of inferred models and treat the modeling technique as a black box. With embedded methods, we refer to modeling techniques, which include feature selection within a model inference procedure. In this chapter, we will dive into filter and wrapper approaches, and for embedded methods focus on model regularization.**

3.1 Relation to Dimensionality Reduction

Dimensionality reduction is an essential part of quantitative data analysis whose aim is to reduce the dimensions of the data considered in the inference of the model. A positive effect of dimensionality reduction is a decreased model complexity and shortened inference time. Another potential benefit is increased interpretability due to the inference of simpler models. The central premise of dimensionality reduction is that this procedure will have little or no effect on the accuracy of the model.

Dimensionality reduction is effective if the input data includes redundant or irrelevant features, or if we can use new features to replace a subset of original features so that to encapsulates all their information. The three most common families of approaches for dimensionality reduction are:

- *Feature transformation* that embeds the data into a lower-dimensional space, replacing original features with a new set that retains as much of information as possible. Approaches of this kind include principal component analysis and deep autoencoders, some of which we will cover in later chapters.
- *Feature selection*, also known as feature subset selection, variable selection, or attribute selection. This approach removes the dimensions (*e.g.* columns) from the input data and results in a reduced data set for model inference.
- *Regularization*, where we are constraining the solution space while doing optimization. Here, we add adding the regularisation terms to which an optimization algorithm must adhere to when minimizing the loss function, apart from having to minimize the error between the true y and the predicted $\hat{y}$. In lasso regularization, for instance, optimization is instructed to find model parameters so that their absulute sum is minimized. This type of regularization may lead to some of the parameters be equal to zero, effectively imposing zero weight to corresponding features, essentially canceling them out from the model. Hence, regularization can be also regarded as a feature selection, where the selection procedure is embedded within an inference method.

3.2 Feature Selection

Feature selection is an optimization problem. The search space is the set of all possible subsets of features, that is, the power set, with 2^n possible solutions. We are trying to find the best solution under some utility and constraints. An example of utility could be the accuracy of the model when inferred from the reduced feature set, and a constraint may be expressed through a maximal number of features. Viewing feature selection as an optimization problem leads to the following properties of the procedure:

- Because we have a discrete search space, we can, in general, not find the optimal solution, unless we perform a global search and evaluate all 2^n solutions.
- Unless the number of original features n is very small, examining all 2^n solutions is infeasible.
- In practice, the best we can do is to use heuristic search methods with a good inductive bias. This approach tends to work well because we can impose assumptions that are more likely to hold on the data we encounter.
- Any search method that operates on discrete search spaces can also be applied to feature selection.

Filter Methods

Filter methods (Guyon and Elisseeff, 2003) perform feature selection before the inference of the model. They rely on a feature scoring function that assigns the score to a feature according to how useful the feature could be in the model. Notice that the scoring is performed before and independently of the model. Features are scored and then ranked, and usually, a top k features are selected, where k is a user-defined parameter of the procedure. Alternatively, feature scores could be compared to their null-distribution that, in practice, could be obtained through feature scoring on a randomly permuted data set. In such cases, k is replaced with user-defined probability p that a particular (or higher) feature score could be obtained and randomly-permuted data.

Scoring functions depend on the type of machine learning problem and type of the scored feature. For instance, for unsupervised learning, we may disregard features with near-constant values by selecting features with the highest deviance, that is, with the highest ratio between variance and the mean. Scoring functions for classification or regression most often consider the correlation between a predictor and the dependent variable. One of the most popular empirical estimates is the mutual information between the i-th predictor and the target y (Guyon and Elisseeff, 2003) :

$$I(i) = \int_{x_i}\int_{y} p(x_i, y) \log \frac{p(x_i, y)}{p(x_i)p(y)} dxdy,$$

where $p(x_i)$ and $p(y)$ are the probability densities of predictor x_i and dependent variable y, and $p(x_i, y)$ is their joint density. These densities are all unknown and are hard to estimate from the data. The easiest of all is the case of nominal variables, where integral becomes a sum and where probabilities are then estimated from frequency counts:

$$I(i) = \sum_{x_i}\sum_{y} P(X = x_i, Y = y) \log \frac{P(X = x_i, Y = y)}{P(X = x_i)P(Y = y)}.$$

Notice that mutual information and all similar feature scoring techniques are univariate and assess the information held by the feature in the absence of the context of other features. The scoring function of this type would undervalue features that are in some interactions with other features and only combined with these provide information about the class. A typical example of such a combination is an exclusive disjunction, where participating features may provide no information about the class on their own, yet are information-rich when they are considered together with complementing argument. A field that studies the discovery and ranking of such features is called feature interaction analysis (Jakulin, 2005; Anastassiou, 2007). The approaches cited here rely on an exhaustive search for feature interactions, which are prohibitive in complexity even for reasonably-sized data sets. A bigger problem, though, is that estimates of feature interactions may report about highly interactive features simply by chance and due to extremely high number of feature combinations explored.

Interestingly, however, note that there are feature score estimators that take into consideration the contexts and are sensitive to feature interactions. The most prominent of these is Relief, an algorithm originally developed by Kira and Rendell (1992). The algorithm assumes that each feature in the data set has been scaled to the interval $[0, 1]$. Let $\boldsymbol{w}$ be a feature weight vector initialized to $\mathbf{0}$. The algorithm randomly draws a data instance $\boldsymbol{x}_i$ and updates the weight vector, such that:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - (x_i - \text{nearHit}(x_i))^2 + (x_i - \text{nearMiss}(x_i))^2$$

where $\text{nearHit}(x_i)$ is the closest same-class data instance to x_i, and $\text{nearMiss}(x_i)$ is the closest different-class data instance to x_i. Notice that the weight of any given feature decreases if it differs from that feature in nearby instances of the same class more than nearby instances of the other class, and increases in the reverse case. In a local neighborhood, the best features should distinguish between instances of the different class and should be similar across instances of the same class. The score $\boldsymbol{w}$ is updated for m random draws, and the features with the highest scores are those that should be selected.

Several extensions and improvements of Relief were proposed by (Kononenko, Šimec, and Robnik-Šikonja, 1997). On the surface, Relief looks like a perfect feature scoring algorithm that can indeed cope with any hidden feature interactions. However, it relies on finding similar data instances and thus suffers from the same problem as any nearest neighbors approach. In other words, Relief would start failing in data sets with a higher number of features, which is precisely where feature interaction discovery would be of the highest value.

Wrapper Methods

Wrapper methods score feature sets according to estimated accuracy or utility of the algorithm used for learning. The most common search approach are forward/backward stepwise selection (Guyon and Elisseeff, 2003). Forward selection is an iterative procedure that starts with an empty set of features and adds one feature at a time, where the benefit of adding a feature is observed in raised accuracy of the inferred model when that feature is added to the feature set. Backward selection starts with a full set of features, and then eliminates one feature at the time, each time selecting feature with the smallest impact on the accuracy of the model. Notice, of course, that backward selection can actually increase the accuracy of the model, as we expect that there is an optimal feature subset with corresponding highest accuracy.

Forward and backward stepwise selection do not necessary yield the same feature sets. Notice that in the presence of strong feature interactions (*e.g.* consider exclusive disjunction) forward selection would miss including features that interact, while backward selection may leave interactive features in the selected set, provided that the underlying machine learning algorithm can detect and use the interactions.

There are other, more elaborate discrete space search procedures that could be used in combination of the wrapper approaches. Consider, for instance, local search algorithms, simulated annealing, or genetic algorithms.

Embedded Methods

Embedded methods refer to feature selection techniques that are part of the learning algorithm itself. A typical example of such a method are classification trees, where the inferred model often includes only a subset of most informative features. Perhaps more elaborate technique in this respect are random forests, where the set of used features may be larger than those from a single tree and where out-of-bag examples can be used for feature scoring and hence ranking.

Below, we will consider a special approach to embedded feature selection that is based on regularization, a constrained optimization that jointly considers accuracy of the inferred model and the magnitude of model parameters and with it the coplexity of the model.

A Rough Summary on Feature Selection Techniques

Of the three approaches to feature selection stated above, wrapper methods are the most general and may yield the best results, but are computationally intensive and often infeasible even with simple brute force forward or backward search. This is especially the case with data domains which include tens or houndreeds of thousands of features that are common in areas like genomics, text, sound and image mining. Filter methods typically work one-feature-at-a-time and are faster, but they might provide suboptimal results, because the selection is decoupled from actual learning. Embedded methods are kind of the best of both worlds, but they require adaptation of the algorithm that implements them.

We already mentioned other approaches to dimensionality reduction that, instead of feature selection, rely on inference of new set of (latent) features. Examples of such feature transformation techniques include matrix factorization, principal component analysis, and deep autoencoders. In comparison with these techniques, please note that:

- feature selection has an advantage over feature transformation as it keeps the original features, which helps with interpretability,

- feature selection is related to explanation/interpretability also through methods that assess variable importance, that is, provides a ranked set of features which can be scrutinized by the domain experts,

- as computational power grows and data sets get larger, filter methods are used less and less and give way to models whose inference relies on optimization and gradient-based search of parameter space.

3.3 Regularization

Let us start with considering linear regression, where $y_i = \boldsymbol{\beta}^\intercal \boldsymbol{x}_i + \epsilon_i$ and where we assume that the error term is distributed normally, so that $\epsilon_i \sim_{\text{iid}} N(0, \sigma^2)$. The data, conveniently, includes a constant column, such that $x_0 = 1$, consequently using $\boldsymbol{\beta}_0$ as an intercept, a constant term in linear combination. Linear regression aims to find $\boldsymbol{\beta}^*$ that minimizes residual sum of squares,

$$\begin{aligned}\text{RSS}(\boldsymbol{\beta}) &= \sum_{i=1}^{n}(y_i - f(\boldsymbol{x}_i))^2 \\ &= \sum_{i=1}^{n}(y_i - \boldsymbol{\beta}^T \boldsymbol{x}_i)^2\end{aligned}$$

so that

$$\boldsymbol{\beta}_{\text{OLS}} = \arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{n}(y_i - \boldsymbol{\beta}^T \boldsymbol{x}_i)^2.$$

Notice that this criteria function actually stems from the maximum likelihood estimation, where parameters $\boldsymbol{\beta}$ are estimated by maximizing a likelihood function, so that under the assumed statistical model the observed data is most probable.

Linear regression also has a closed form solution Hastie, R. Tibshirani, and Friedman, 2016. Let $\boldsymbol{X}$ denote $n \times (1+p)$ matrix with n training data instances described with p features. The first column of the matrix is a unit vector. Residual sum of squares can then be expressed as:

$$\text{RSS}(\boldsymbol{\beta}) = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^\intercal (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})$$

Differentiating with respect to $\boldsymbol{\beta}$ we obtain

$$\begin{aligned}\frac{\partial \text{RSS}}{\partial \boldsymbol{\beta}} &= -2\boldsymbol{X}^\intercal(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) \\ \frac{\partial^2 \text{RSS}}{\partial^2 \boldsymbol{\beta}} &= -2\boldsymbol{X}^\intercal \boldsymbol{X}\end{aligned}$$

Setting the first derivative to zero and assuming that $\boldsymbol{X}$ is nonsingular and hence $\boldsymbol{X}^\intercal \boldsymbol{X}$ is positive definite, we obtain

$$\begin{aligned}\boldsymbol{X}^\intercal(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}_{\text{OLS}}) &= 0 \\ \boldsymbol{\beta}_{\text{OLS}} &= (\boldsymbol{X}^\intercal \boldsymbol{X})^{-1}\boldsymbol{X}^\intercal \boldsymbol{y}\end{aligned}$$

The main problem that we will address with regularization are cases with few observations in the data set that are described with comparably many features. In other words, cases where the training data matrix $\boldsymbol{X}$ has relatively few rows compared with columns. Here, we are likely to overfit the data, that is, develop a complex model that includes many features and fits training data well but misperforms in prediction on new data. In fact, if we have more columns than rows, we will have colinearity, so $\boldsymbol{X}^\intercal \boldsymbol{X}$ will not be invertible and if we want to solve it by minimizing the sum of squares, we will have an infinite number of equivalent solutions.

We will look at regularization from a few different perspectives. The first and the most commonly used one is addressing the overfit by penalizing deviations of model coefficients from zero. For linear regression, if the value of a coefficient is 0, we regard that the corresponding feature is not used in the model. A common approach to supress the magnitude of model coefficients is to use a quadratic penalty term, leading to the modified optimization problem:

$$\boldsymbol{\beta}_{L2} = \arg\min_{\boldsymbol{\beta}} \left(\sum_{i=1}^{n} (y_i - \boldsymbol{\beta}^\intercal \boldsymbol{x}_i)^2 + \lambda \sum_{i=1}^{k} \boldsymbol{\beta}_i^2 \right),$$

where $\lambda \geq 0$ is the regularization parameter or regularization weight. Notice that λ is an additional parameter of the optimizatio problem whose value must either be set manually or determined through some optimization procedure that can involve estimation of accuracy of resulting model by cross-validation or using a validation data set. There are two extreme regularization cases: if $\lambda = 0$, we get non-regularized regression; if $\lambda = \infty$, the regularization penalty is so high that the optimal solution is to select $\boldsymbol{\beta}_i = 0$ for all $i \geq 1$. Note that the intercept, $\boldsymbol{\beta}_0$, is not regularized and in this case becomes equal to the mean value of the outcome variable. The optimal value of λ lies somewhere between these two extremes, and penalizes the coefficients just enough to prevent overfitting, but not too much to interfere with the learning, that is, not obfuscating the likelihood term.

The quadratic (L2 norm) penalty is not the only one we can use. We will later discuss the other commonly used penalty, the absolute or L1 norm penalty. And we can here note that another, potentially useful penalty uses the L0 norm (counting), which penalizes for the number of features selected, that is, the number of nonzero $\boldsymbol{\beta}_i$. But first we will explore how L2 penalty term transforms the initial optimization problem of finding the maximum of the likelihood.

Closed-Form Solution for L2 Regularization

Similar to how we derived the above closed-form solution to the least squares problem we can also derive a closed-form solution to the penalized regression. We want to minimize $\left(\sum_{i=1}^{n} (\boldsymbol{\beta}^T \boldsymbol{x}_i - y_i)^2 + \lambda \sum_{i=1}^{k} \boldsymbol{\beta}_i^2\right)$ or, in matrix shorthand $||\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}||_2^2 + \lambda ||\boldsymbol{\beta}||_2^2$. Again, we find the extreme the usual way by differentiating and checking where the gradient is 0:

$$\frac{d}{d\boldsymbol{\beta}} \left(||\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}||_2^2 + \lambda ||\boldsymbol{\beta}||_2^2 \right) = 2(\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y})^\intercal \boldsymbol{X} + 2\lambda\boldsymbol{\beta} = 2\boldsymbol{\beta}^\intercal \boldsymbol{X}^\intercal \boldsymbol{X} - 2\boldsymbol{y}^\intercal \boldsymbol{X} + 2\lambda\boldsymbol{\beta}^\intercal$$

Note that if we differentiate again, we get $2\boldsymbol{X}^\intercal \boldsymbol{X} + 2\lambda \boldsymbol{I}$. This is always positive definite for $\lambda > 0$, so we have a minimum. This is in contrast with non-penalized regression, where we rely on the additional assumption that $\boldsymbol{X}$ has full rank, making $2\boldsymbol{X}^\intercal\boldsymbol{X}$ positive definite by itself.

Finally, the extreme is where the gradient is zero, so that $2\boldsymbol{X}^\intercal\boldsymbol{X}\boldsymbol{\beta}_{L2} - 2\boldsymbol{X}^\intercal\boldsymbol{y} + 2\lambda\boldsymbol{\beta}_{L2} = (2\boldsymbol{X}^\intercal\boldsymbol{X} + 2\lambda\boldsymbol{I})\boldsymbol{\beta}_{L2} - 2\boldsymbol{X}^\intercal\boldsymbol{y} = 0$ or $(\boldsymbol{X}^\intercal\boldsymbol{X} + \lambda\boldsymbol{I})\boldsymbol{\beta}_{L2} = \boldsymbol{X}^\intercal\boldsymbol{y}$, which leads to

$$\boldsymbol{\beta}_{L2} = (\boldsymbol{X}^\intercal\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^\intercal\boldsymbol{y}$$

. Note that the term is invertible for the reasons discussed above. In essence, we make $\boldsymbol{X}^\intercal\boldsymbol{X}$ invertible by adding at least a tiny number to its diagonal elements, making the resulting matrix invertible even if it was not invertible by itself. Besides constraining the solution space, L2 regularization solves the problem of non-invertibility that we can encounter when using plain linear regression.

Equivalence of Penalized and Constrained Forms

L2 regularization, as stated above, can be viewed as penalized optimization, where we deal with the objective and a penalty:

$$\text{minimize}_{\beta}\left\{||\boldsymbol{\beta}^\intercal\boldsymbol{x}_i - \boldsymbol{y}_i||_2^2 + \lambda||\boldsymbol{\beta}||_2^2\right\}.$$

This optimization problem can be formulated as an alternative way that provides additional insight into what regularization does geometrically:

$$\text{minimize}_{\beta}\left\{||\boldsymbol{\beta}^\intercal\boldsymbol{x}_i - \boldsymbol{y}_i||_2^2\right\}, \text{subject to } ||\boldsymbol{\beta}||_2^2 \leq c,$$

where $c \geq 0$ is some constant. Now we show that these two are indeed equivalent. We will use $\boldsymbol{\beta}_P$ to denote the solution to the penalized form and $\boldsymbol{\beta}_C$ the solution of the constrained form. First, we show that for any $\boldsymbol{X}$, $\boldsymbol{y}$, and every c there exists a constant λ that does not depend on $\boldsymbol{X}$ and $\boldsymbol{y}$ such that the solutions of the two problems are the same, that is, $\boldsymbol{\beta}_C = \boldsymbol{\beta}_P$. In other words, we will show that every constraint formulation of the problem has an equivalent penalized formulation.

We already know the solution to the penalized formulation (we derived it above):

$$\boldsymbol{\beta}_P = (\boldsymbol{X}^\intercal\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^\intercal\boldsymbol{y}$$

Now we write the Lagrangian of the constrained formulation:

$$L(\boldsymbol{\beta}, \mu) = ||\boldsymbol{\beta}^\intercal\boldsymbol{x}_i - \boldsymbol{y}_i||_2^2 + \mu(||\boldsymbol{\beta}||_2^2 - c)$$

According to Karush–Kuhn–Tucker (KKT), we have the following conditions to guarantee an optimal solution, which are in this case sufficient, because we have a convex problem and continuously differentiable constraints:

$$
\begin{aligned}
\frac{d}{d\boldsymbol{\beta}} L(\boldsymbol{\beta}, \mu) &= 0 \qquad (3.1) \\
\mu &\geq 0 \qquad (3.2) \\
\mu(||\boldsymbol{\beta}||_2^2 - c) &= 0 \qquad (3.3)
\end{aligned}
$$

Observe that, for the first of the above conditions, the left hand side is the same as the gradient of the penalized form, just using μ instead of λ.

Now assume that $\boldsymbol{\beta}_P$ solves the penalized formulation for a given λ. Setting $\mu = \lambda$, $\boldsymbol{\beta} = \boldsymbol{\beta}_P$, and $c = ||\boldsymbol{\beta}_P||_2^2$ satisfies all three KKT conditions, so there exists for every λ a c such that the solutions to the two problems are the same. Conversely, if $\boldsymbol{\beta}_C, \mu$ solves the constrained formulation for a given c, then $\boldsymbol{\beta}_C$ solves the penalized formulation at $\lambda = \mu$. So, there exists for every c a λ such that the solutions to the two problems are the same. Thus the formulations are equivalent.

In essence, we have shown that penalizing the solution with the quadratic norm is equivalent to putting a hypersphere constraint on the solution. This equivalence of constrained and penalized forms applies in general to p-norms (Hastie, R. Tibshirani, and Friedman, 2016).

L1 regularization

The optimization problem of L1 regularization, also known as Lasso regression, is:

$$\boldsymbol{\beta}_{L1} = \arg\min_{\boldsymbol{\beta}} \left(\sum_{i=1}^{n} (\boldsymbol{\beta}^\intercal \boldsymbol{x}_i - \boldsymbol{y}_i)^2 + \lambda \sum_{i=1}^{k} |\boldsymbol{\beta}_i| \right)$$

or, in vector notation:

$$\boldsymbol{\beta}_{L1} = \arg\min_{\boldsymbol{\beta}} \left(||\boldsymbol{\beta}^\intercal \boldsymbol{x}_i - \boldsymbol{y}_i||_2^2 + \lambda ||\boldsymbol{\beta}||_1 \right).$$

With the result from the previous section on comparison of constrained and penalized formulation it is not too big a cheat if we immediately say that L1 is equivalent to a ’diamond’ constraint, although the proof is not so obvious. This also allows for the geometric discussion of why Lasso regression tends to set coefficients to zero, while L2 regularization usually infers small but non-zero values of coefficients.

Bayesian Interpretation of Regularization

Regularization is sometimes referred to as *a bet on sparsity*. That is, we are making an apriori assumption that not all (or even not most) of the input variables are relevant predictors. As soon as we introduce prior information it should not come as a great surprise that regularization has a very elegant Bayesian interpretation.

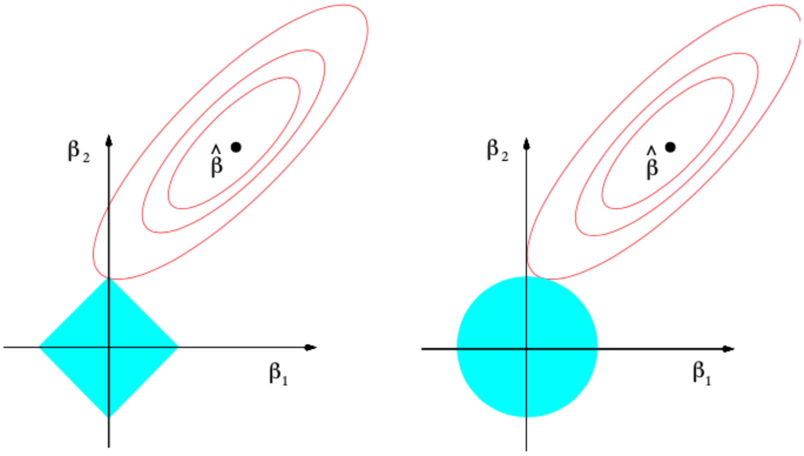


Figure 3.1: Geometric presentation of the optimization problem for the lasso (L1, left) and ridge (L2, right) regression. Shown are conturs of the penalty (least squares error), and the constrain regions $|\beta_1| + |\beta_2| \leq c$ and $\beta_1^2 + \beta_2^2 \leq t^2$. The sharp corners of the constraint region of the lasso yield sparse solutions. In high dimensions, sparsity arises from corners and edges of the lasso's constraint region (from R. J. Tibshirani, 2014).

To see this, we go back to the optimization goal of ordinary least squares regression from the beginning of the chapter:

$$\boldsymbol{\beta}_{OLS} = \arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} (\boldsymbol{\beta}^\intercal \boldsymbol{x}_i - y_i)^2,$$

and recalling that this minimization is equivalent to maximizing the normal (Gaussian) likelihood assumed by the linear regression model,

$$L(\boldsymbol{\beta}; ...) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(\boldsymbol{\beta}^\intercal \boldsymbol{x}_i - y_i)^2}{2\sigma^2})$$

Indeed, maximizing the log-likelihood, we obtain

$$\ell(\boldsymbol{\beta}; ...) = \sum_{i=1}^{n} \log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (\boldsymbol{\beta}^\intercal \boldsymbol{x}_i - y_i)^2$$

Note that the maximum w.r.t. the $\boldsymbol{\beta}$ does not depend on σ^2, only on mimimizing the sum of squares.

With what do we have to multiply the likelihood to produce the extra term $-\lambda \sum_{i=1}^{k} \boldsymbol{\beta}_i^2$ in the log-likelihood and therefore get the negative of this term which appears in the minimization problem of L2 regression?

The answer should be obvious: $\prod_{i=1}^{k} \exp(-\frac{\beta_i^2}{1/\lambda})$. In terms of $\boldsymbol{\beta}_i$ this is proportional to the product of normal likelihoods $\boldsymbol{\beta}_i \sim_{iid} N(0, \frac{1}{\sqrt{\lambda}})$. So, if we look at this in terms of the Bayesian (log)posterior $\log p(\boldsymbol{\beta}|\boldsymbol{y},\boldsymbol{x}) \propto \log p(\boldsymbol{y}|\boldsymbol{\beta},\boldsymbol{x}) + \log p(\boldsymbol{\beta})$, we see that if we place a normal prior on the coefficients, we get a posterior maximum that corresponds to the L2 regularized MLE solution. In other words, the Bayesian interpretation of L2 regression is that we express a prior opinion that coefficients are normally distributed around 0 with some variance that is a function of $\sigma^2 = \frac{1}{\sqrt{\lambda}}$. Higher variance implies lower λ (less regularization), while lower variance implies higher λ (more regularization).

Similarly we can find the analogue to L1 regularization - the absolute penalty in log-space corresponds to the Laplace distribution (pdf of Laplace is $p(\boldsymbol{x}) = \frac{1}{2b}\exp(-\frac{|x-\mu|}{b})$).

Note that the Bayesian approach to regularization lends itself to an alternative way of inferring λ. Instead of using a fixed λ or choosing the best λ via CV or similar procedure, we can instead treat λ as a parameter, put a hyper-prior on it, and infer it simultaneously with the coefficients.

Also note that regularization (adding a penalty term to the likelihood) is in statistical circles more often referred to as **penalized likelihood**.

Final Remarks

Regularization is not limited to linear regression. Although it might lead to more complex optimization/sampling problems, the basic principles apply to all models that have coefficients that we can penalize (all linear models, SVM and other kernel methods).

We typically do not regularize the constant (intercept) coefficient. Either that, or we demean the data and not use an intercept at all when regularizing. Regularizing it does not make sense, because it should fit the mean of the data and that is typically not 0 and we have no reason to have a prior opinion that it is related to the other coefficients.

A particular form of regularization, called elastic net regularization, combines lasso and ridge penalties with an additional weight parameter. This regularization, however, introduces another meta-parameter (this mixing between L1 and L2 penalties) that needs to be inferred from the data.