HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Lecture Notes

5812
Machine Learning

Prof. Dr. Markus Schneider

Colophon

This document was typeset in LaTeX
using the *Lucida* typeface and math fonts

Lecture in *Machine Learning*
Prof. Dr. Markus Schneider
revision: d2249b8

Contents

Introduction 1
Linear Regression 9
Linear Classification 14
Nonlinear Modeling and Generalization 27
Kernel Functions 45
Perceptrons and Support Vector Machines 53
Nonparametric Algorithms 62
Decision Trees 68
Ensemble Methods 75
Neural Networks and Deep Learning 85
Supervised Cheat Sheet 108

Nomenclature

MachineLearning

σ	network activation function→ p. 85
$h \in \mathscr{H}$	hypothesis function (model)→ p. 7
H	ensemble of models→ p. 76
J	objective function→ p. 8
K	kernel matrix→ p. 47
ℓ	loss function→ p. 7
η	learning rate→ p. 4
m	data set size→ p. 7
n	input dimension→ p. 7
$\mathbf{P}_{XY}$	data distribution→ p. 7
λ	regularization parameter→ p. 38
$\Omega(\boldsymbol{w})$	regularizer→ p. 38
sign	sign function→ p. 13
$\boldsymbol{w} \in \mathscr{W}$	parameter→ p. 7
$\boldsymbol{x}, \boldsymbol{z} \in \mathscr{X}$	feature (input)→ p. 7
$\hat{\boldsymbol{x}} \in \mathscr{X}$	query input→ p. 6
$y \in \mathscr{Y}$	label (output)→ p. 7
$\hat{y} \in \mathscr{Q}$	prediction→ p. 7

Miscellaneous

$\mathbf{N}$,$\mathbf{Z}$,$\mathbf{Q}$,$\mathbf{R}$,$\mathbf{C}$	set of all natural, integer, rational, real and complex numbers; $0 \in \mathbf{N}$.
$[a,b]$,(a,b)	closed and open intervals.
$\|\cdot\|$,$\|\cdot\|_1$	Euclidean norm, ℓ_1 norm.

Introduction

There is a common, systematic approach to define machine learning algorithms. This chapter provides a first glance on how to train a machine learning model, make predictions and analyze the results. These ideas will lead to a more formal definition of machine learning, detailed in the following chapters.

Predicting Peak Electrical Power Consumption

We start with a simple example where our goal is to predict the peak electricity demand during the next day. In this case we will consider real data from San Diego, CA, served by the San Diego Gas & Electric Company.

Predicting electricity demand is actually a very important problem as the biggest concern in power grids is to balance between the generation of energy and load. This becomes more difficult as renewable generation is added to the grid. Intermittent renewable energy sources like wind and solar power generation cannot be scheduled and cannot be predicted accurately.

What makes the task difficult is, that there are various factors which influence the power consumption, such as temperature, day of week, seasons, special events — such as holidays — and many more. Some of these, like the temperature, are themselves difficult to predict. Hence, it would be very challenging to come up with a model for electrical demand based on expert knowledge.

What we can do, however, is to collect data about past energy consumption as well as information about the factors that affect consumption such as the weather during the relevant periods as, for example, shown in Table 1.1.

It seems that electricity consumption in San Diego in the summer heavily correlates with the outside temperature (Fig. 1.1). On days with high tem-

Date	Peak Demand [GW]	Temperature [°C]	Day of Week	Season
2014-01-01	2.77	19.4	Wednesday	Winter
2014-01-02	2.96	20.6	Thursday	Winter
2014-01-03	2.91	18.9	Friday	Winter
⋮	⋮	⋮	⋮	⋮
2017-07-17	3.49	24.4	Monday	Summer
2017-07-18	3.48	26.7	Tuesday	Summer
2017-07-19	3.56	27.2	Wednesday	Summer
2017-07-20	3.56	24.4	Thursday	Summer
⋮	⋮	⋮	⋮	⋮
2018-10-28	2.67	22.2	Sunday	Summer
2018-10-29	2.85	22.2	Monday	Summer
2018-10-30	2.81	21.7	Tuesday	Summer

Table 1.1 The peak demand data set merges the maximal energy consumption with the highest temperature on that day.

perature, the peak demand was also relatively high. This may be due to air conditioning. It is therefore not unreasonable to assume, that a *linear* model can make predictions for the peak demand, given the temperature on that day, *i.e.*,

$$\text{peak demand} \approx w_2 \cdot \text{temperature} + w_1,$$

where w_2 is the "slope" of the line and w_1 is the intercept. These are the *parameters* of our model, and we will collect them in a vector $\boldsymbol{w} = (w_1, w_2)$. We do not expect the prediction to be perfect, but with good choices for these parameters, and, given a forecast of tomorrow's weather, we can predict the likely peak demand on that day fairly well.

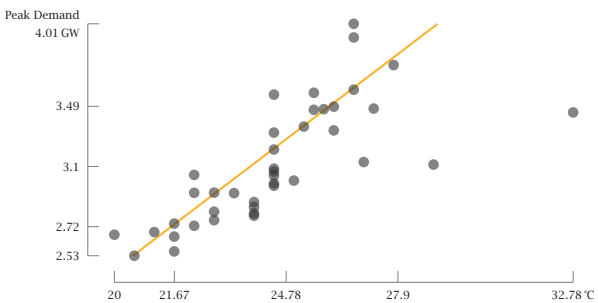


Figure 1.1 High temperature vs. peak demand for a sample of summer months (June – August). Tick marks are at min., max., mean ±std. The data exhibit a clear linear trend.

There are many reasonable models (lines) through the data, each corresponding to certain choices for $\boldsymbol{w}$. In such cases we call the model identifiable with the parameters $\boldsymbol{w}$. The question arises, which one of these lines fit the data "best" and then, how to find the corresponding parameters for this linear model. We have to define a metric, which tells us, how "good" each model performs on the given data. This is the *objective function*.

The Objective Function

The objective function is one of the key components of any machine learning problem. Different choices for the objective function will lead to different algorithms. There are some well-studied standard objective functions to choose from which will later be discussed in more detail.

For the application at hand, we will use the *squared error* between the predictions of the model and the actual value of *peak demand* given the *high temperature* value as

$$(\text{prediction} - \text{peak demand})^2 = (w_2 \cdot \text{temperature} + w_1 - \text{peak demand})^2$$

for all instances in the data set.

To ease notation, let $\boldsymbol{x}_i$ denote the input, *i.e.*, the temperature ith day. Likewise, the label y_i denotes the peak demand on day i and $\hat{y}_i$ its prediction from the model. The inputs and labels are collected in a data set $\{ (\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m) \}$ with recordings of m days.

The performance of a model (identified by the parameters $\boldsymbol{w}$) on the whole data set is then measured by the *objective function* $J(\boldsymbol{w})$ defined as

$$J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 = \frac{1}{m} \sum_{i=1}^{m} (w_2 \cdot \boldsymbol{x}_i + w_1 - y_i)^2$$

which is just the average squared error on the data set. The lower the value of the objective function is for a given model, the better it performs on the data in terms of the squared error. The goal is therefore to minimize $J(\boldsymbol{w})$ to find the best model in this context. Notice that the objective is explicitly a function of the model parameters $\boldsymbol{w}$, as these are the only variables / unknowns that can be tweaked in order to improve the model performance.

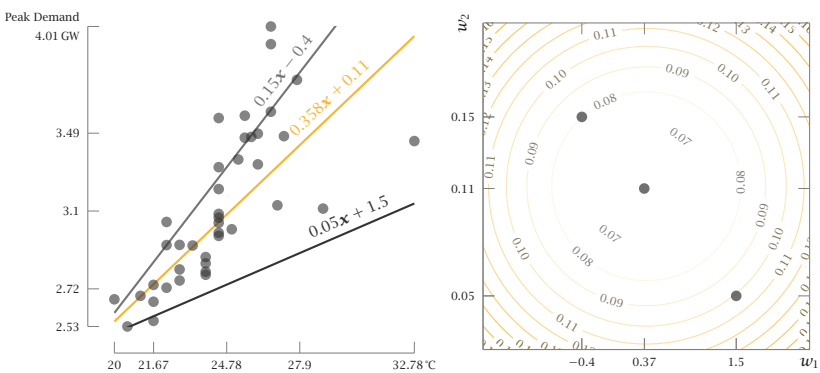


Figure 1.2 Every model (line) corresponds to a certain choice of $\boldsymbol{w}$. The performance of a model is then measured by the *objective function* $J(\boldsymbol{w})$ whose contours are overlaid in the parameter space on the right. The model identified with $\boldsymbol{w} = (0.36, 0.11)$ has the lowest object of the three and seems to have the best fit to the data.

Minimizing the Objective

The goal is now to find the best parameters $\boldsymbol{w}$, which are the ones that minimize the objective function $J(\boldsymbol{w})$. One of the simplest strategies to achieve this goal is based on the derivatives of the objective function and will ultimately lead to the *gradient decent* algorithm.

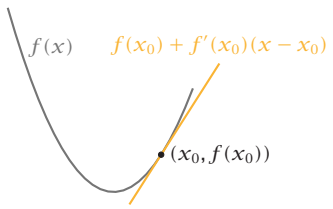


Figure 1.3 The derivative defines the first order Taylor approximation to the function f around a pointx_0 and is the slope of the function.

The idea is as follows: Recall that for a scalar function $f : \mathbf{R} \to \mathbf{R}$, its derivative is the slope of the function at any point (Fig. 1.3). If we want to minimize the function, starting at some initial value x_0, it makes thus sense to move in the direction of the *negative* derivative (called the *descent direction*) until we reach a minimum; or at least until we are very close to one. This simple intuition also holds for vector-valued functions $f : \mathbf{R}^n \to \mathbf{R}$ and forms the basis for the gradient descent algorithm.

The peak demand model has two parameters which needs to be optimized: w_1 and w_2. The descent direction of both is determined by their partial derivatives of the objective function.

$$\begin{aligned}
\frac{\partial J(\boldsymbol{w})}{\partial w_1} &= \frac{\partial}{\partial w_1} \frac{1}{m} \sum_{i=1}^{m} (w_2 \cdot \boldsymbol{x}_i + w_1 - y_i)^2 \\
&= \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial w_1} (w_2 \cdot \boldsymbol{x}_i + w_1 - y_i)^2 \quad \text{(by linearity of the derivative)} \\
&= \frac{1}{m} \sum_{i=1}^{m} 2(w_2 \cdot \boldsymbol{x}_i + w_1 - y_i) \qquad \text{(chain rule)}
\end{aligned}$$

Analogous, we can compute the partial derivatives of $J(\boldsymbol{w})$ with respect to the parameter w_2 as

$$\begin{aligned}
\frac{\partial J(\boldsymbol{w})}{\partial w_2} &= \frac{\partial}{\partial w_2} \frac{1}{m} \sum_{i=1}^{m} (w_2 \cdot \boldsymbol{x}_i + w_1 - y_i)^2 \\
&= \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial w_2} (w_2 \cdot \boldsymbol{x}_i + w_1 - y_i)^2 \quad \text{(by linearity of the derivative)} \\
&= \frac{1}{m} \sum_{i=1}^{m} 2(w_2 \cdot \boldsymbol{x}_i + w_1 - y_i) \cdot \boldsymbol{x}_i \quad \text{(chain rule)}
\end{aligned}$$

where all the steps are identically to the previous derivation, except for the fact that the coefficient term multiplying $\boldsymbol{w}_2$ in the last equation is always equal to $\boldsymbol{x}_i$.

Taking repeatedly steps in the direction of the negative derivatives leads parameters that have a low objective values and are therefore considered good. The steps are realized by

$$w_1 \leftarrow w_1 - \eta \frac{\partial J(\boldsymbol{w})}{\partial w_1} \quad \text{and} \quad w_2 \leftarrow w_2 - \eta \frac{\partial J(\boldsymbol{w})}{\partial w_2}$$

where $\eta > 0$ is called the *step size*. This is the gradient descent algorithm.

Feature Scaling

Most machine learning algorithms usually do not perform well when the numerical attributes (features) have very different scales. In the peak demand application, the values for temperature in degrees Celsius are relatively large compared to the bias term. Without normalization, it would be necessary to define different step sizes for the two parameters w_1 and w_2. While this is possible in principle, it is much easier to normalize the data and only have to deal with a single step size parameter. Especially in applications that involve many more parameters. Thus, we scale all data to be in roughly the same range before running the machine learning algorithm.

There are several strategies for feature normalization: Two common methods are *min-max scaling* and *standardization*. Min-max scaling is quite simple as the values are shifted and rescaled such that they end up in the range $[0, 1]$ or $[-1, 1]$. Standardization makes the values of each feature have zero-mean (first subtracting the mean from the data) and unit-standard deviation (by dividing through its standard deviation). For most algorithms, it does not matter which features scaling method is used.

Standardization was applied to the peak demand data in Fig. 1.4. As expected, the shape of the data did not change, just its scale. As a consequence, the units no longer correspond to physical quantities (like degrees Celsius here), but are some linear transformations of these.

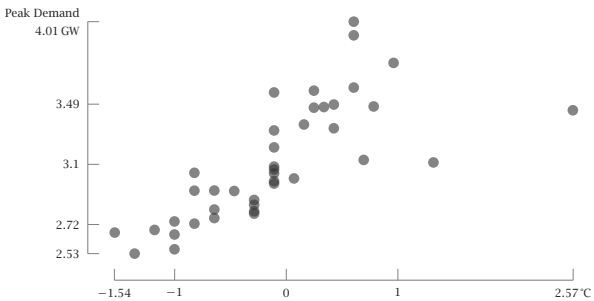


Figure 1.4 High temperature vs. peak demand for summer months, where the temperature variable was scaled such that has zero-mean and unit-standard deviation.

It is recommended to apply feature scaling beforehand with one exception: If the difference in scale is actually significant for the problem, then rescaling might throw away useful information and should not be used.

Visualizing Gradient Descent

Fig. 1.5 illustrates the converges of gradient descent with a step size of $\eta = 0.05$ on the peak demand data set. The parameters $\boldsymbol{w}$ are initialized to $(1, 1)$; however, we will show later that the initialization does not matter for this task. In each gradient descent iteration, the current parameter vector gets closer to the optimum of $\boldsymbol{w}^\star = (3.11, 0.27)$. The improvement is also reflected by the values of the objective function which becomes monotonically smaller. This is not surprising as the gradient descent algorithm is explicitly designed to descents on the objective function. The error decrease rapidly until the parameters are close to the optimum $\boldsymbol{w}^\star$ after about 30 iterations. The fast convergence rate is one of the desirable properties of the squared error function.

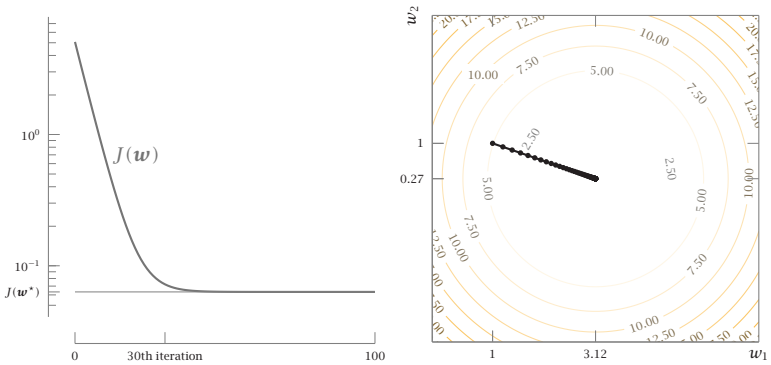


Figure 1.5 The gradient descent algorithm converges to the optimal parameters $\boldsymbol{w}^\star$ on the peak demand data sets. The plot on the left shows how the objective function value becomes monotonically smaller with each iteration of gradient descent. On the right are the individual parameters $\boldsymbol{w}$ used in each iteration overlaid with objective function contours. The parameters converge from their initial value of $(1, 1)$ to $\boldsymbol{w}^\star = (3.11, 0.27)$.

Once the optimal parameters are estimated on the normalized data, the solution can be converted back to the original scale in order to make predictions. Equivalently (and often preferable), new query points $\hat{\boldsymbol{x}}$ can also be normalized with the same procedure before plugged into the model for predictions.

Knowing that the high temperature tomorrow will be, for example, 28 °C we can forecast a likely energy peak demand of

$$\hat{y} = w_2 \cdot 28\,^\circ\mathrm{C} + w_1 \approx 3.51\,\mathrm{GW}$$

for that day. The energy production can then be adjusted accordingly.

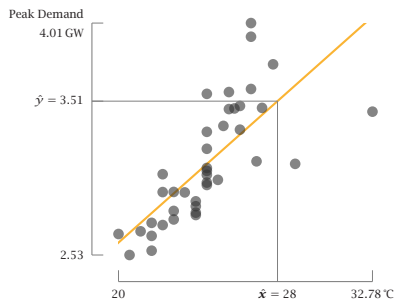


Figure 1.6 The prediction of peak electricity demand for a day with 28 °C temperature is 3.51 GW, according to model.

Machine Learning Terminology

The approach we took in the peak demand prediction was a typical example of a *machine learning* algorithm. The basic idea is, that we are confronted with a problem which is hard to model manually, but relatively easy to collect data for. Machine learning then provides a way to automatically infer the predictive model from data by optimizing an objective function.

To define machine learning in more general terms, it is necessary to introduce some terminology.

Input Features: $\boldsymbol{x} \in \mathscr{X}$. Inputs or features are the quantities from the input space $\mathscr{X}$ for which we would like to make predictions. Typically, the inputs are real values vectors with n components. In this case $\mathscr{X} = \mathbf{R}^n$.

Output Labels: $y \in \mathscr{Y}$. The *labels* are the targets from the label space $\mathscr{Y}$ we are interested in predicting. Common labels are real-valued scalars (for regression tasks), or binary and multivariate discrete quantities (for classification tasks).

Parameters: $\boldsymbol{w} \in \mathscr{W}$. These are the parameters of the model which are optimized during the learning phase. Often, $\mathscr{W} = \mathbf{R}^d$ but more extensive sets of parameters are possible.

Predictions: $\hat{y} \in \mathscr{Q}$. This is the *output* of the model (hypothesis function) for a given input. The prediction space is not necessarily the same as the label space $\mathscr{Y}$, but represents quantities that can easily be transformed to the label space. This is typical for classification tasks (*e.g.*, "Is this email spam or not?"), where it is often useful predict probabilities instead of class labels.

Hypothesis Function or *Model:* $h : \mathscr{X} \to \mathscr{Q}$. A model is a mapping from the input space $\mathscr{X}$ to the prediction space $\mathscr{Q}$. Most models depend on some parameters $\boldsymbol{w}$ which is sometimes emphasized by writing $h_{\boldsymbol{w}}$. The collection of all models is denoted as $\mathscr{H}$.

Loss Function: $\ell : \mathscr{Q} \times \mathscr{Y} \to \mathbf{R}_+$. The loss function measures how good or bad a prediction from the model was. The first argument is the prediction, the second argument the true label. The goal of every machine learning model is to make the loss as small as possible.

Data Distribution: $\mathbf{P}_{XY}$. This is the true, but unknown probability distribution the data comes from. If this distribution would be known, then perfect predictions become trivial. However, in general, only a finite set of samples $\{ (\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_m, y_m) \}$ from this distribution is available.

Empirical Risk Minimization

Given a collection of input features and labels $\{ (\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_m, y_m) \}$, and a hypothesis class $\mathscr{H}$ the goal of any machine learning algorithm is to find parameters $\boldsymbol{w}$ which minimize the empirical risk defined as

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} \ell(h(\boldsymbol{x}_i), y_i)$$

which is essentially the average of losses on the training data.

Virtually *all* machine learning algorithms have this form. They require

- The hypothesis class $\mathscr{H}$ (the set of functions h we can possibly learn).
- The loss function ℓ, which measures the performance of the model.
- A learning algorithm that minimizes the empirical risk $J(\boldsymbol{w})$.

This unifying view of machine learning algorithms as *optimization problems* is rather new and hence most techniques do not make these three elements explicit.

The name empirical risk minimization stems from the fact that, we would like to minimize the expected loss on the true data distribution $\mathbf{P}_{XY}$, *i.e.*

$$\mathbf{E}[\ell(h(X), Y)] = \int \ell(h(X), Y) \, \mathrm{d}\mathbf{P}_{XY} \tag{1.1}$$

called the risk. However, since $\mathbf{P}_{XY}$ is unknown, the empirical risk serves as a proxy for the above expectation as the strong law of large numbers guarantees that J converges to $\mathbf{E}[\ell(h(X), Y)]$ almost surely.

Linear Regression

This chapter reiterates over the peak power consumption regression problem, but now in the context of the new machine learning terminology and notation.

Problem Statement

For notational convenience, we introduce a new constant feature (sometimes called the bias term) to the input vector as

$$\boldsymbol{x} = \begin{bmatrix} 1 \\ \text{temperature} \end{bmatrix} \quad \text{or} \quad \boldsymbol{x} = \begin{bmatrix} 1 \\ \text{temperature} \\ \text{is weekday} \end{bmatrix}$$

which allows us to simply write

$$h(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle = \sum_{j=1}^{n} w_j x_j$$

for the linear hypothesis function with $\boldsymbol{w} \in \mathbf{R}^n$. Since the first feature is always 1, the first coefficient w_1 has the role of an offset or bias parameter.

Using the new terminology, the setup for the peak demand prediction task is as follows:

- vector valued inputs and parameters: $\mathscr{X} = \mathscr{W} = \mathbf{R}^n$
- real outputs and predictions: $\mathscr{Y} = \mathscr{Z} = \mathbf{R}$
- linear hypothesis class $\mathscr{H} = \{ \langle \boldsymbol{w}, \boldsymbol{x} \rangle : \boldsymbol{w} \in \mathscr{W} \}$
- squared error loss: $\ell_{\text{sq}}(h(\boldsymbol{x}), y) = (h(\boldsymbol{x}) - y)^2$
- overall optimization problem

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} (\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle - y_i)^2$$

This special setup — a linear model with a square loss function — is called least squares regression and has some desirable properties. One if them is, that there is an analytic solution to the optimization problem above.

Gradient Descent

Before we explore the exact analytic solution, we reformulate gradient descent now in a more general way using vector notation and gradients.

For a multivariate function $f\colon \mathbf{R}^n \to \mathbf{R}$, its gradient is an n-dimensional vector containing the partial derivatives with respect to each dimension:

$$\nabla_{\boldsymbol{w}} f(\boldsymbol{w}) = \Big(\frac{\partial f(\boldsymbol{w})}{\partial w_1}, \dots, \frac{\partial f(\boldsymbol{w})}{\partial w_n}\Big)$$

If there is no confusion which element we take the gradient with respect to, we simply write $\nabla f(\boldsymbol{w})$ instead of $\nabla_{\boldsymbol{w}} f(\boldsymbol{w})$.

The gradient of the least squares objective function $J(\boldsymbol{w})$ is given by

$$\nabla J(\boldsymbol{w}) = \nabla \frac{1}{m} \sum_{i=1}^{m} (\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle - y_i)^2 = \frac{2}{m} \sum_{i=1}^{m} \boldsymbol{x}_i (\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle - y_i).$$

This leads to the following *gradient descent update rule* when using a linear model and a square loss function:

$$\textit{repeat: } \boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \frac{2}{m} \sum_{i=1}^{m} \boldsymbol{x}_i (\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle - y_i)$$

Algorithm 2.1 The gradient descent update rule for least squares regression.

The advantage of the vector notation is, that it does not matter how many features input vector $\boldsymbol{x}$ has, the algorithm always stays the same. We can, for example, include an additional feature indicating whether the data corresponds to weekday or weekend as

$$\text{is weekday} = \begin{cases} 1 & \text{if weekday,} \\ 0 & \text{if weekend} \end{cases}$$

with the hope that this helps to reduce the variance within the data and hence improve the predictions.

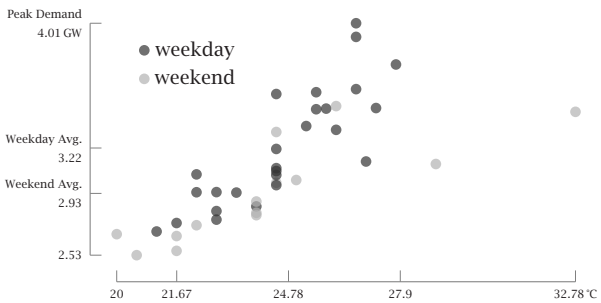


Figure 2.1 A partition of the data set into weekdays and weekends shows, that (on average) the peak electricity consumption is higher on the weekdays than the weekends.

The scatter plot in Fig. 2.1 shows, that — on average — the peak electricity consumption is higher on the weekdays than the weekends. This is a great finding as it explains some variability in the data. What previously just looked like random noise can now be explained. The variance in both

classes individually is much smaller than the variance within the entire data set. This is a great example of *feature engineering*.

Running gradient descent again, but this time on the data set with the new *is weekday* feature produces a different model than before, illustrated in Fig. 2.2. Although this looks like two individual hypothesis functions, it is just one. Depending on whether the feature *is weekday* is one or zero, the second element of $\boldsymbol{w}$ is added or vanishes:

$$\begin{aligned} h(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle &= w_3 \cdot \text{temperature} + w_2 \cdot \text{is weekday} + w_1 \\ &= \begin{cases} w_3 \cdot \text{temperature} + w_2 + w_1 & \text{if weekday,} \\ w_3 \cdot \text{temperature} + w_1 & \text{if weekend.} \end{cases} \end{aligned}$$

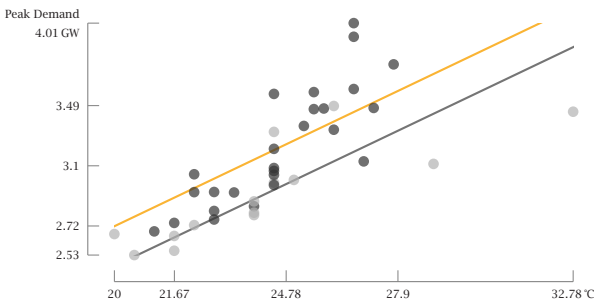


Figure 2.2 Using additional features, the model can offset its prediction, depending on if the day is a weekend or weekday. That way, the predictions become better.

The drawbacks of these features are evident: It is not possible to have two different slopes, which seems to be more appropriate for the data. Obviously, it would be possible to use two individual models, one for weekdays and one for weekends. The same can be achieved by a single model with an intelligent design of features:

$$\boldsymbol{x} = \begin{bmatrix} \text{(is weekday)} \\ \text{(1}-\text{is weekday)} \\ \text{temperature}\cdot\text{(is weekday)} \\ \text{temperature}\cdot\text{(1}-\text{is weekday)} \end{bmatrix}$$

which yields two linear models in one

$$h(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle = \begin{cases} w_3 \cdot \text{temperature} + w_1 & \text{if weekday,} \\ w_4 \cdot \text{temperature} + w_2 & \text{if weekend.} \end{cases}$$

This new model seems to fit the data much better than the previous one (Fig. 2.3) — all due to a different choice of features.

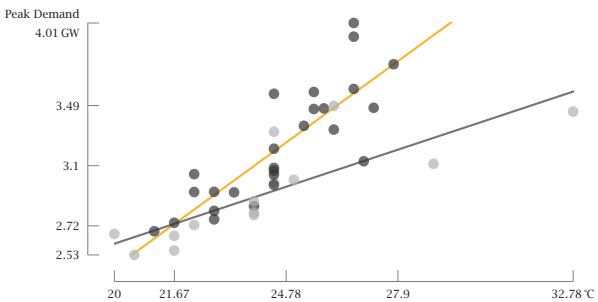


Figure 2.3 Use a different choice of features yields a model that is more flexible and fits the data better than the previous one.

An analytic Solution to Least Squares

For many optimization problems, there is no analytic solution and one has to resort to iterative methods like gradient descent. However, for least squares in particular, it is possible to analytically derive the optimum $\boldsymbol{w}^\star$ in closed form.

Notice, that in order for $\boldsymbol{w}^\star$ to be an optimum, it is necessary, that the gradient of the objective function is zero at this point. We already calculated the gradient of J with respect to $\boldsymbol{w}$, to be

$$\nabla J(\boldsymbol{w}) = \frac{2}{m}\sum_{i=1}^{m} \boldsymbol{x}_i(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle - y_i).$$

While it is possible to set this equation to zero and solve for $\boldsymbol{w}$, it turns out, that we can describe the sums also as matrices and vectors, making the analytic solution much simpler. For this purpose, define the so-called design matrix X as the $m \times n$ matrix that contains the inputs as rows and an m-dimensional vector $\boldsymbol{y}$ of all outputs.

$$X = \begin{bmatrix} - \boldsymbol{x}_1^\mathsf{T} - \\ \vdots \\ - \boldsymbol{x}_m^\mathsf{T} - \end{bmatrix} \in \mathbf{R}^{m\times n}, \quad \boldsymbol{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \in \mathbf{R}^m$$

notice that $X\boldsymbol{w} - \boldsymbol{y}$ is an m-dimensional vector with $\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle - y_i$ as its ith component. Hence, we can rewrite the gradient equation above as

$$\nabla J(\boldsymbol{w}) = \frac{2}{m} X^\mathsf{T}(X\boldsymbol{w} - \boldsymbol{y}). \tag{2.1}$$

Setting the gradient to zero yields

$$\begin{aligned} \nabla J(\boldsymbol{w}^\star) = 0 &\iff X^\mathsf{T}(X\boldsymbol{w}^\star - \boldsymbol{y}) = 0 \\ &\iff X^\mathsf{T}X\boldsymbol{w}^\star - X^\mathsf{T}\boldsymbol{y} = 0 \\ &\iff X^\mathsf{T}X\boldsymbol{w}^\star = X^\mathsf{T}\boldsymbol{y} \end{aligned}$$

where $X^\mathsf{T}X\boldsymbol{w}^\star = X^\mathsf{T}\boldsymbol{y}$ is also known as the normal equations. Thus, the value of $\boldsymbol{w}^\star$ that minimizes J is given in closed form by the equation

$$\boldsymbol{w}^\star = (X^\mathsf{T}X)^{-1}X^\mathsf{T}\boldsymbol{y}.$$

The design matrix also allows us to express the least squares objective function more compactly as

$$J(\boldsymbol{w}) = \frac{1}{m}\|X\boldsymbol{w} - \boldsymbol{y}\|^2. \tag{2.2}$$

Proof.

$$\begin{aligned} J(\boldsymbol{w}) = J(\boldsymbol{w}) &= \frac{1}{m}\sum_{i=1}^{m}(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle - y_i)^2 \\ &= \frac{1}{m}\langle X\boldsymbol{w} - \boldsymbol{y}, X\boldsymbol{w} - \boldsymbol{y}\rangle \\ &= \frac{1}{m}\|X\boldsymbol{w} - \boldsymbol{y}\|^2. \end{aligned}$$

□

Loss Functions for Regression

This section considers alternative loss functions for the regression. Often, the collected data is not ideal: It may contain measurement errors or wrongly labeled instances. Assume there is a single outlier of 2.7 GW at 35 °C in the peak demand dataset (see Fig. 2.4). The least square loss is very sensitive to such outliers in the data set since each error in prediction will lead to a penalty of the *squared* magnitude of that error. Therefore, the outlier causes a drastic shift of the fitted model.

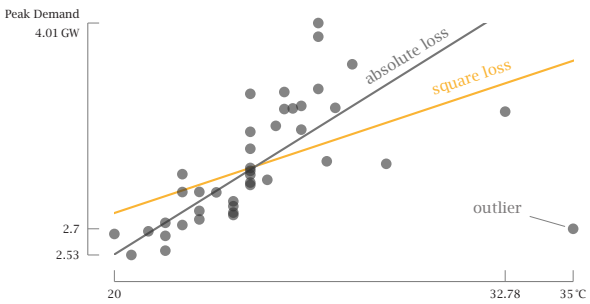


Figure 2.4 The least square loss is heavily influenced by the outlier, but not the absolute loss. The model obtained by the absolute loss is nearly identical to the one obtained on the outlier-free data set.

To make outliers less influential, one could use the absolute loss instead of a squared one:

$$\ell_{\text{abs}}(h(\boldsymbol{x}), y) = |h(\boldsymbol{x}) - y|. \tag{2.3}$$

Now each error in prediction will only lead to a penalty of the same magnitude. The only thing we have to do, is to swap the loss in the empirical risk minimization problem, which then becomes

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} |\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle - y_i|$$

which is sometimes referred to as the mean absolute error. The resulting algorithm is called robust regression hinting at its robustness against outliers.

Unfortunately, the absolute loss no longer has a closed form solution for the optimal $\boldsymbol{w}$ (the existence of a closed form solution is one of the reasons why the square loss is so popular, despite its disadvantages).

In addition, the objective function with absolute loss is not continuous differentiable anymore, and therefore we do not have gradients everywhere. A slight modification of gradient descent with the *subgradient*

$$\nabla J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{x}_i \operatorname{sign}(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle - y_i)$$

leads to *subgradient descent*.

The sign function of a real number x is defined as

$$\operatorname{sign}(x) = \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases}$$

If $\boldsymbol{x}$ is a vector, the function is applied component-wise.

Linear Classification

In the previous chapter we focused on regression problems, *i.e.*, predicting real-valued outputs. We now turn to classification, where the predictions are *discrete* labels. Each label represents a class correspondence. Our goal is still to solve the empirical risk minimization problem and, surprisingly, the only thing that changes is the loss function.

Classifying Rock Types

Our next real world example will be from geology.[1] More precisely from *rock physics*, which studies the relation between physical and elastic properties of rocks and is the basis of quantitative seismic interpretation.

In this scenario, we got the following data set with the task to distinguish between shale and sand rock types. As sample of the data is listed in the following table.

Depth [m]	Acoustic Impedance [$\mathrm{kg\,s^{-1}\,g\,cm^{-3}}$]	Shear/Pressure Velocity	Rock Type
62	5270.1	2.7	shale
68	5279.6	2.8	shale
603	6680.7	2.1	other
586	6526.8	2.0	sand
192	5373.6	2.8	shale
510	5481.2	2.0	sand
334	5278.0	3.2	sand
206	5068.4	2.8	shale
226	5485.1	2.7	shale
⋮	⋮	⋮	⋮
89	5430.8	2.6	other

[1] The data set is available from the quantitative seismic interpretation book website. Even though the data is real, the classification task is made artificially simple on purpose.

Figure 3.1 Mining site.

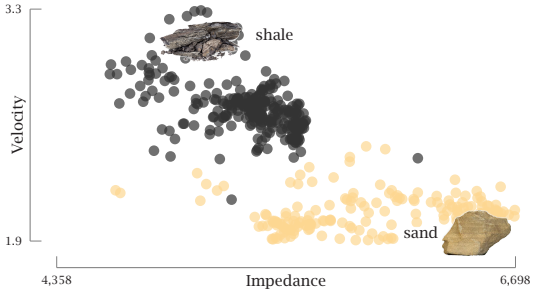


Figure 3.2 The scatter plot of the rock type data set reveals that the classes *sand* ● and *shale* ● are not linearly separable.

A scatter plot of the data (Fig. 3.2) reveals that the classes *sand* ● and *shale* ● form two main clusters with some outliers. A low acoustic impedance and high velocity ratio indicates a high probability for shale. The opposite is true for sand.

Whereas in linear regression our goal was to fit a line (or hyperplane) to the data, we now aim for a line which separates the two classes. The hypothesis class is therefore again $\mathscr{H} = \{ \langle \boldsymbol{w}, \boldsymbol{x} \rangle : \boldsymbol{w} \in \mathscr{W} \}$. This line is called decision boundary or separating hyperplane. Obviously, it is not possible to find a line which perfectly partitions all data points into the two classes. In such cases we say the data is *not* linearly separable. However, some separating lines are clearly better than others in that they make fewer classification errors (in instance is on the wrong side of the hyperplane).

In the context of classification, most elements of the empirical risk minimization setup stays the same. The one difference is that the output space, $\mathscr{Y}$, is now a *discrete* space instead of a continuous one. For the rock type data set with only two classes the label space $\mathscr{Y} = \{ -1, +1 \}$ is a reasonable choice.[1] In the following, let the label -1 corresponds to sand and $+1$ to shale.

[1] While it certainly would be possible to use $\mathscr{Y} = \{ 0, 1 \}$ as the label space, this would require a less convenient notation for the loss function.

Ideally, the predictions are not only *sand* or *shale*, but reflect the *confidence* in the prediction of the class correspondence. Some algorithms also allow for probabilistic predictions. We therefore choose $\mathscr{Z} = \mathbf{R}$ as the output/prediction space.

The magnitude of the prediction shall indicate how certain the model is in its class assignment. A large negative value of $h(\boldsymbol{x})$ indicates a high confidence for the class -1 (sand ●) for the input $\boldsymbol{x}$, whereas a large positive value indicates that the class is probably $+1$ (shale ●). Values of $h(\boldsymbol{x})$ around zero can be interpreted as a high uncertainty in the prediction. If a final class assignment is required, the model output can be thresholded at zero.

The input space will be $\mathscr{X} = \mathbf{R}^3$ with input vectors of *impedance*, *velocity*, and a constant bias term.

Loss Functions for Classification

We are now looking for a loss function $\ell : \mathcal{Z} \times \mathcal{Y} \to \mathbf{R}_+$ which is suitable for classification. Unlike in the regression case, $\mathcal{Z}$ and $\mathcal{Y}$ are different. $\mathcal{Z} = \mathbf{R}$ is continuous, indicating the confidence in prediction and $\mathcal{Y} = \{-1, +1\}$ is discrete and represents the class labels.

Why is a different loss needed in the first place? Why is it not possible to simply use the square loss (or some other regression loss) for the classification task and simply threat the discrete labels $\{-1, +1\}$ as continuous?

The problem becomes evident on the small toy data set in Fig. 3.3. The data is perfectly separable with a line (—●—). However, the least squares loss results in a predictor (—♦—) which misclassifies the left-most point. Remember, that the predicted label is $+1$, if the model output is non-negative and -1 otherwise. Hence, the decision boundary is at the point where the line crosses the x-axis.

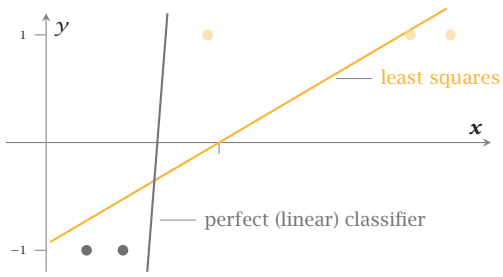


Figure 3.3 Using the square loss for classification is not always a good idea. Even though the data is perfectly separable by a line, the least squares loss yields a sub-optimal solution.

The issue is, that least squares aims to predict the ± 1-labels exactly, where, in reality, we only care about the correct sign of the prediction. In other words: Least square optimizes the *wrong* objective for the problem.

The perhaps the most obvious loss for classification is the one that counts the number of errors the classifier makes. An input is wrongly classified if the sign of $h(\boldsymbol{x}_i)$ is different from the sign of the corresponding label y_i. This is known as the zero-one loss $\ell_{0/1} : \mathbf{R} \times \{-1, +1\} \to \mathbf{R}_+$, defined as

$$\ell_{0/1}(h(\boldsymbol{x}), y) = \begin{cases} 0 & \text{if } \operatorname{sign}(h(\boldsymbol{x})) = y, \\ 1 & \text{otherwise.} \end{cases}$$

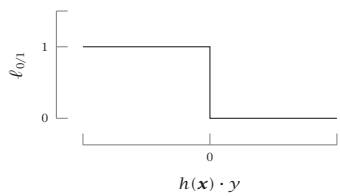


Figure 3.4 The zero-one loss. If $h(\boldsymbol{x}) \cdot y$ is positive (the prediction has the same sign as the label), the loss is zero and one otherwise.

The behavior of the zero-one loss is illustrated in Fig. 3.4: As long as $h(\boldsymbol{x})$ has the same sign as y, the loss is zero. Whenever the signs disagree, the loss becomes one — independent of the magnitude of the model prediction. While zero-one loss is very natural to consider for classification tasks, it has two major drawbacks.

1 The zero-one loss is nonconvex and its derivation is zero nearly everywhere, except for the origin. This renders optimization techniques, like gradient descent, virtually useless.
2 It can be shown, that for binary classification with a linear separator, there are 2^m possible dividing planes to check for an exact solution. As of today, there is no efficient algorithm known for this task other than exhaustive search with exponential runtime, making it unfeasible.

Thus, we should consider alternative loss functions for classification tasks that we can actually calculate (optimize) efficiently. Ideally, the surrogate loss bounds the zero-one loss from above such that, when minimizing the surrogate, the zero-one loss (and therefore the number of misclassifications) is also minimized.

There are several loss functions for classification which can be used alternatively to the zero-one loss, each one leading to a different classification algorithm.

Logistic Loss: $\ell_{\text{logistic}}(h(\boldsymbol{x}), y) = \log(1 + \exp(-y \cdot h(\boldsymbol{x})))$
The logistic loss grows linearly for negative values which makes it less sensitive to outliers. This is also one of the most popular loss functions for classification, since its outputs can be interpreted as probabilities.

Hinge Loss: $\ell_{\text{hinge}}(h(\boldsymbol{x}), y) = \max\{0, 1 - y \cdot h(\boldsymbol{x})\}$
The hinge loss provides a relatively tight, convex upper bound on the zero-one loss. This loss will lead to the famous support vector machine.

Exponential Loss: $\ell_{\text{exp}}(h(\boldsymbol{x}), y) = \exp(-y \cdot h(\boldsymbol{x}))$
The exponential Loss is another function that upper bounds on the zero-one loss, but is very aggressive. The penalty for a misclassification increases exponentially with the distance to the decision boundary. We will see this loss again, when deriving the AdaBoost algorithm.

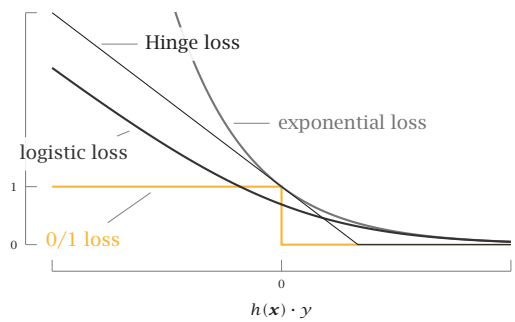


Figure 3.5 Loss functions for classification. The zero-one loss is the least sensitive to outliers, the exponential loss the most. Both, Hinge and exponential loss, upper bound on the zero-one loss.

Unlike regression, there is virtually no loss function for classification with an analytical solution to the empirical risk minimization problem. Therefore, iterative methods (like gradient descent) are the only way to learn the parameters of models for classification.

Logistic Regression

Logistic regression, despite its name, is a classification algorithm that uses a linear model in combination with the logistic loss function. This gets us

$$\begin{aligned}\ell_{\text{logistic}}(h(\boldsymbol{x}), y) &= \log(1 + \exp(-y h(\boldsymbol{x}))) \\ &= \log(1 + \exp(-y\langle \boldsymbol{w}, \boldsymbol{x}\rangle))\end{aligned}$$

The partial derivatives for gradient descent of the logistic loss is given by

$$\begin{aligned}\nabla J(\boldsymbol{w}) &= \frac{1}{m}\sum_{i=1}^{m} \nabla \log(1 + \exp(-y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle)) && (\nabla \log(x) = {}^{1}/_{x}) \\ &= \frac{1}{m}\sum_{i=1}^{m} \frac{-y_i \boldsymbol{x}_i \exp(-y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle)}{1 + \exp(-y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle)} && (\exp(-x) = {}^{1}/_{\exp(x)}) \\ &= \frac{1}{m}\sum_{i=1}^{m} \frac{-y_i \boldsymbol{x}_i}{\exp(y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle) + \exp(0)}\end{aligned}$$

The gradient descent update rule for logistic regression is therefore

$$\textit{repeat: } \boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \frac{1}{m}\sum_{i=1}^{m} \frac{-y_i \boldsymbol{x}_i}{1 + \exp(y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle)}$$

Algorithm 3.1 The gradient descent update rule for logistic regression.

and again, surprisingly simple.

For the rock type classification task, we normalize the data using *standardization* such that, the features *impedance* and *velocity* have zero mean and unit standard deviation. Subsequently, we run gradient descent with a learning rate of $\eta = 1$ for 2500 steps on the objective function.

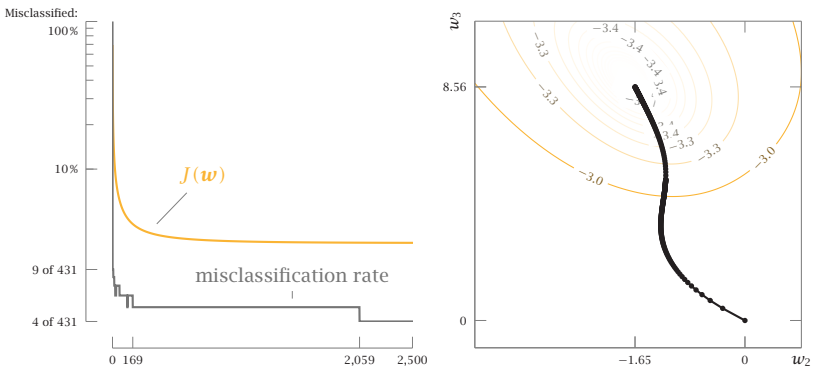


Figure 3.6 The gradient descent algorithm converges to the optimal parameters $\boldsymbol{w}^\star$ on the rock types data sets. The plot on the left shows that the misclassification rate decreases, although we are optimizing the logistic loss objective. On the right are the individual parameters $\boldsymbol{w}$ used in each iteration overlaid with objective function contours. The parameters converge from their initial value of $(0, 0)$ to $\boldsymbol{w}^\star = (-1.65, 8.56)$.

The left curve in Fig. 3.6 shows a rapid and smooth decrease of the logistic regression objective function as gradient descent progresses. The misclassification rate (the percentage of wrongly classified instances) also decreases overall, but is much less smooth and increases temporarily. This effect is not surprising, since minimizing with the logistic loss and not the zero-one loss directly. Hence, we do not expect the error to decrease monotonically throughout the training, nevertheless it should decrease overall.

The logistic regression decision boundary of the model after training, which consists of all points in the set $\{\langle \boldsymbol{w}, \boldsymbol{x}\rangle = 0 : \boldsymbol{x} \in \mathscr{X}\}$, is visualized in Fig. 3.7. All points above the line ($\langle \boldsymbol{w}, \boldsymbol{x}\rangle > 0$) will be classified as *shale* ●; all points below the line ($\langle \boldsymbol{w}, \boldsymbol{x}\rangle < 0$) as *sand* ●. A larger magnitude indicates more confidence in the prediction.

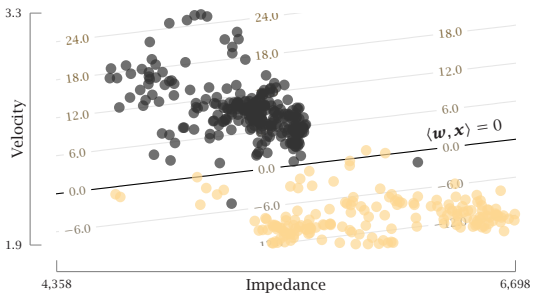


Figure 3.7 Contour plot of the logistic regression decision boundary after 2500 steps of gradient descent. The decision is made at $h(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x}\rangle = 0$. All points above the line ($\langle \boldsymbol{w}, \boldsymbol{x}\rangle > 0$) will be classified as *shale* ●, below ($\langle \boldsymbol{w}, \boldsymbol{x}\rangle < 0$) as *sand* ●. A larger magnitude indicates more confidence in the prediction.

Ideally, we would like to use a model to directly predict the probability of a class correspondence, since probabilities are easier to deal with than scores. At least for logistic regression, there is a statistical justification to transform the predictions through the logistic function $z \mapsto 1/(1+\exp(-z))$ to obtain probabilities. That means it is possible to model the probability distribution

$$\mathbf{P}(Y = 1 \mid X = \boldsymbol{x}) = \frac{1}{1 + \exp(-h(\boldsymbol{x}))}$$

which maps the predictions to the interval $(0, 1)$, now representing the probability of input $\boldsymbol{x}$ having label 1. The probability of the other class is then evidently $\mathbf{P}(Y = -1 \mid X = \boldsymbol{x}) = 1 - \mathbf{P}(Y = 1 \mid X = \boldsymbol{x})$.

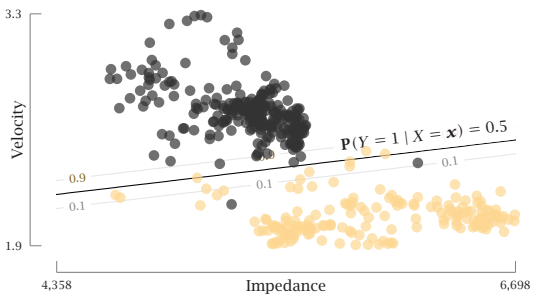


Figure 3.8 The predictions of the logistic model where mapped through the logistic function.

Linear Support Vector Machine

The (linear) support vector machine (SVM) is a liner model which optimizes the hinge loss:

$$\begin{aligned}\ell_{\text{hinge}}((h\boldsymbol{x}), y) &= \max\{\, 0, 1 - yh(\boldsymbol{x}) \,\} \\ &= \max\{\, 0, 1 - y\langle \boldsymbol{w}, \boldsymbol{x}\rangle \,\}\end{aligned}$$

There is also a nice geometric interpretation of the support vector machine as a classifier that maximize the "margin" of separation between examples of different classes which will be discussed later.

It has to be noted, that the support vector machine objective actually includes another term besides average loss on the data set: The *regularization* term $\sum_{j=1}^{n} w_j^2 = \|\boldsymbol{w}\|^2$ penalizes the (squared) magnitude of the sum of parameters, yielding

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} \max\{\, 0, 1 - y_i \cdot \langle \boldsymbol{w}, \boldsymbol{x}_i\rangle \,\} + \lambda\|\boldsymbol{w}\|^2$$

as the regularized objective. Since the topic of regularization will be covered shortly in great detail, we just mention here that this additional term has the effect of balancing out the "complexity" of the classifier (which increases with larger parameters) with the goal of minimizing the loss. The regularization parameter $\lambda \geq 0$ controls the trade-off between these complementary goals.

The gradient of the objective decomposes into two parts: The gradient of hinge loss and the gradient of the regularization term.

$$\begin{aligned}\nabla J(\boldsymbol{w}) &= \frac{1}{m}\sum_{i=1}^{m} \nabla \max\{\, 0, 1 - y_i \cdot \langle \boldsymbol{w}, \boldsymbol{x}_i\rangle \,\} + \nabla\lambda\|\boldsymbol{w}\|^2 \\ &= \frac{1}{m}\sum_{i=1}^{m} \begin{cases} -\boldsymbol{x}_i y_i & \text{if } y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle \leq 1, \\ 0 & \text{otherwise} \end{cases} + 2\lambda\boldsymbol{w}.\end{aligned}$$

In practice, the *linear* SVM and logistic regression are usually comparable. The decision boundary obtained by the SVM algorithm is very similar to the one obtained by logistic regression, although not identical.

The main difference between the hinge loss and the logistic loss is, that the latter always tries to make the confidence for the correct class higher, whereas the hinge loss does not care about the magnitude of the predictions, as long as it is greater than one.

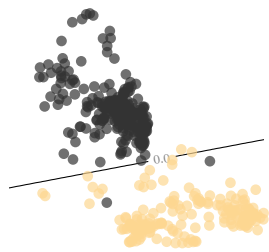


Figure 3.9 The linear support vector machine decision boundary after 2500 steps of gradient descent is nearly identical to the one obtained by logistic regression.

Multiclass Classification

So far, our task was to distinguish between two different classes often referred to as binary classification. However, there are many problems with more than two possible outputs. In this case, the label space is $\mathcal{Y} = \{1, \dots, k\}$, which is then known as multiclass classification.

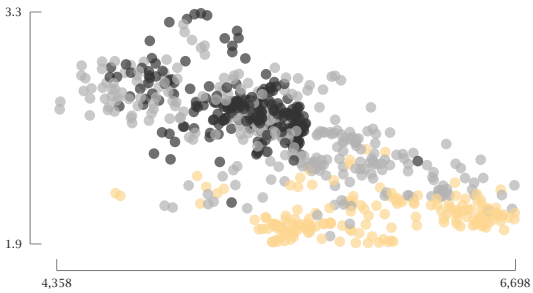


Figure 3.10 A classification problem with three classes.

There are basically two common approaches for this task. We can either reduce the task to simple binary classification, or use a loss function which is especially crafted to handle multiclass setups. Both approaches are discussed in turn.

Reduction to Binary Classification

The simplest reduction of a multiclass problem to binary classification is the one versus all method. Here, an ensemble H of k binary classifiers $h_1, \dots, h_k$ are trained, one for each of the k distinct classes. The task of the tth classifier is then to decide, if an input belongs to the class t or to *any* of the other classes. This also requires k data sets, where the tth class has label $+1$ and the instances of all other classes get labeled -1 as in Fig. 3.11.

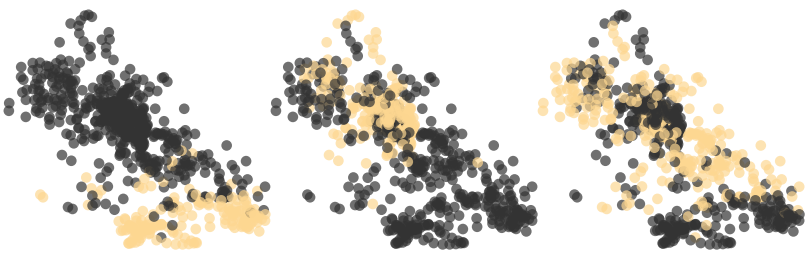


Figure 3.11 Relabeling of data for the one versus all method.

In order to make a prediction for a point $\boldsymbol{x}$, all k classifiers are evaluated and the prediction of the one with the highest confidence is chosen. Formally

$$H(\boldsymbol{x}) = \underset{t \in \{1,\dots,k\}}{\arg\max}\, h_t(\boldsymbol{x})$$

which explicitly allows negative confidence values (prediction).

Although this strategy is popular, it is a heuristic that suffers from several problems: Firstly, the scale of the confidence values may differ between the binary classifiers. Second, even if the class distribution is balanced in the original set, the binary classifiers see unbalanced data because the set of negatives they get is much larger than the set of positives.

The one versus one reduction is less common as it requires $k(k-1)/2$ binary classifiers, where each receives the instances of a pair of classes from the original training set, and must learn to distinguish these two classes. At prediction time, a voting scheme is applied, where all $k(k-1)/2$ classifiers make a prediction for a query point and the class that got the highest number of $+1$ predictions gets selected by the ensemble classifier. This approach does not suffer the same class imbalance problem as *one versus all*, however is much more computationally expensive.

Multiclass Loss

In the binary classification setting, the model $h\colon \mathscr{X} \to \mathbf{R}$ maps to a real value which indicates the confidence of predicting class $+1$ if positive and -1 if negative. Instead of a single number, we could modify the output of h to a k-dimensional prediction, one dimension for each class label. Then, the prediction space becomes $\mathscr{Q} = \mathbf{R}^k$ and the model is a map $h\colon \mathscr{X} \to \mathbf{R}^k$. As in the binary setting, these predictions are not discrete class labels, as the jth entry in the prediction vector $\hat{\boldsymbol{y}}$ is a real-valued indicator of the relative confidence that the input belongs to class j.

In the linear setting, the multiclass model is defined as

$$\begin{aligned} h\colon \mathscr{X} &\to \mathbf{R}^k \\ \boldsymbol{x} &\mapsto W\boldsymbol{x} = \hat{\boldsymbol{y}} \end{aligned} \tag{3.1}$$

for some parameter matrix $W \in \mathbf{R}^{k\times n}$ and vector-valued predictions $\hat{\boldsymbol{y}}$.

A change in the prediction space also requires a different loss function. Assume for a moment, that the multiclass classifier outputs a vector of *probabilities* $q_{\hat{Y}}$, where the jth entry corresponds to the probability that the input $\boldsymbol{x}$ belongs to class j. Then, this prediction forms a probability distribution over the classes. Similar, the correct label y can be interpreted as a discrete distribution p_Y where one entry (the correct class) has probability 1 and all other classes have probability 0 (see the illustration Fig. 3.12).

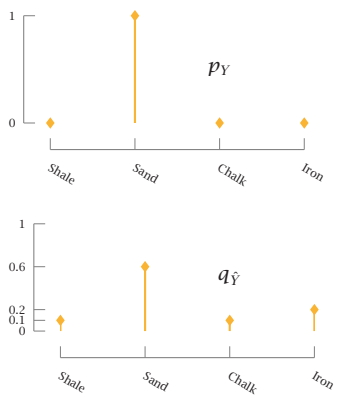


Figure 3.12 Top: The true label encoded as a probability mass function p_Y. All probabilities are zero expect the correct class *Sand*. Bottom: The model predictions $\hat{\boldsymbol{y}}$ normalized to a probability mass function $q_{\hat{Y}}$. Classes which are more likely under the model get assigned a higher probability. Ideally, these two distributions are identical.

Ideally, the predicted distribution of the model is as close as possible to the label distribution. The cross entropy can be used as a measure of divergence between the two. The cross entropy $H(\mathbf{P}_Y, \mathbf{Q}_{\hat{Y}})$ between two discrete probability distributions $\mathbf{P}_Y$ and $\mathbf{Q}_{\hat{Y}}$ with support $\mathscr{Y}$ is defined as

$$\ell(q_{\hat{Y}}, p_Y) = H(\mathbf{P}_Y, \mathbf{Q}_{\hat{Y}}) = -\sum_{y\in\mathscr{Y}} p_Y(y) \log q_{\hat{Y}}(y)$$

where p_Y and $q_{\hat{Y}}$ are the probability mass functions of $\mathbf{P}_Y$ and $\mathbf{Q}_{\hat{Y}}$, respectively. Notice, that the order of arguments in $\ell(q_{\hat{Y}}, p_Y)$ and $H(\mathbf{P}_Y, \mathbf{Q}_{\hat{Y}})$ are flipped since the cross entropy is not symmetric.

Before cross entropy can be used as loss function, two things need to happen:

1. The class labels $y \in \{1, \dots, k\}$ must be mapped to a vector p_Y of k components where jth entry must be set to 1 if $y = j$.
2. The model predictions $\hat{\boldsymbol{y}}$ must be normalized such that all entries are nonnegative and sum to one. Only then, they represent discrete probability distributions. Nonnegativity can be achieved by mapping each component of $\hat{\boldsymbol{y}}$ with $z \mapsto \exp(z)$. Afterwards, divide the resulting vector by its sum of components to get the distribution $q_{\hat{Y}}$.

The operations in the second step are sometimes called softmax operation.

It turns out, that there are natural analogues of most binary losses in the multiclass setting. The multiclass hinge loss, for example, encourages the correct class to a have higher confidence than the incorrect classes by a margin of at least one, and is defined as

$$\ell(\hat{\boldsymbol{y}}, y) = \sum_{j \neq y} \max\{0, 1 - \hat{\boldsymbol{y}}_y + \hat{\boldsymbol{y}}_j\}$$

If any other class has a score higher than $\hat{\boldsymbol{y}}_y - 1$ the loss will be positive and zero otherwise.

Remark 3.1

You may sometimes encounter the following expression for the cross entropy loss.

$$\begin{aligned} \ell &: \mathbf{R}^k \times \{1, \dots, k\} \to \mathbf{R}_+ \\ \ell(\hat{\boldsymbol{y}}, y) &= -\hat{\boldsymbol{y}}_y + \log \sum_{j=1}^{k} \exp(\hat{\boldsymbol{y}}_j) \end{aligned}$$

This is essentially equivalent to the two-step procedure above:

$$\begin{aligned} \ell(\hat{\boldsymbol{y}}, y) &= -\sum_{z \in \mathcal{Y}} p_Y(z) \log q_{\hat{Y}}(z) \\ &= -\log q_{\hat{Y}}(y) && \text{(since } p_Y(z) = 0 \text{ if } z \neq y\text{)} \\ &= -\log\left(\frac{\exp \hat{\boldsymbol{y}}_y}{\sum_{j=1}^{k} \exp \hat{\boldsymbol{y}}_j}\right) && \text{(normalization of } \hat{\boldsymbol{y}}\text{)} \\ &= -\hat{\boldsymbol{y}}_y + \log \sum_{j=1}^{k} \exp \hat{\boldsymbol{y}}_j && \text{(equivalent formulation)} \end{aligned}$$

Evaluation of Classifiers

How good is a classifier after training? Although, the algorithm optimizes a surrogate loss function (recall, that the zero-one loss is infeasible from a practical point of view) it might still be useful to report the number of classification mistakes — or, more common, its mean which is also known as the error rate. Another common metric is the accuracy, which measures the mean number of correctly classified instances.

However, there is a problem with reporting only the error rate or accuracy. If the data set is unbalanced, *e.g.*, in credit card fraud detection: In 2020 alone, around 390 billion card payments were made worldwide, from which 18.5 million transactions were fraudulent. A machine learning model which ignores the input and always predict *genuine* would have an accuracy of approximately 0.999952, but would still be useless.

The idea is to distinguish between misclassification of positive instances and negative instances in form of a confusion matrix. Take for example logistic regression from Fig. 3.7. It misclassifies two shale instances as sand and three sand instances as shale. The other 256 shale and 170 sand instances where correctly classified.

actual label \ predicted class	sand	shale
sand	170	3
shale	2	256

actual label \ predicted class	positive	negative
positive	TP	FN
negative	FP	TN

Good results correspond to large numbers down the main diagonal and small, ideally zero, off-diagonal elements in the confusion matrix.[1]

[1] How does the confusion matrix of a perfect classifier look like?

Typically, one class is named *positive*, the other *negative*. The correctly classified instances of each class are the true positives (TP) and true negatives (TN) respectively. Consequently, wrongly classified instances are called false positives (FP) if the correct class is negative, but positive was predicted and false negatives (FN) if the correct class is positive, but negative was predicted. Remember, the first term tells us if the prediction was right (true) or wrong (false) and the second term tells us what we predicted (positive or negative class).

actual label \ predicted class	cat	mouse	bird
cat	32	5	1
mouse	2	18	6
bird	8	1	25

Figure 3.13 The confusion matrix can be extended to multiclass classification.

If a classifier outputs a score or a probability, ROC curves are a useful visual tool for measuring how different thresholds on the output effect the performance of classifier. The name ROC stands for receiver operating

characteristic. An ROC curve shows the trade-off between the true positive rate (proportion of positive instances that are correctly identified) and the false-positive rate (proportion of negative instances that are incorrectly identified as positive) for a given model.

The for example the following table with predictions $\hat{y}$ of our model. Depending on the threshold τ (here the horizontal line marks $\tau = 0.5$) the model makes more false positive or false negative errors. A perfect classification is not possible in this example.

$\boldsymbol{x}$ (features)	$\hat{y}$ (predicted)	$\hat{y} \geq \tau$	y (actual)
...	0.95	+1 ●	+1 ●
...	0.80	+1 ●	+1 ●
...	0.60	+1 ●	−1 ●
...	0.45	−1 ●	+1 ●
...	0.40	−1 ●	−1 ●
...	0.35	−1 ●	−1 ●
...	0.25	−1 ●	−1 ●

For each choice of τ (moving the line up and down), the ROC curve plots true positive rate and false-positive rate resulting from τ (see Fig. 3.14). A graphical depiction of the various thresholds will often be far more revealing than presenting a single "optimal" decision.

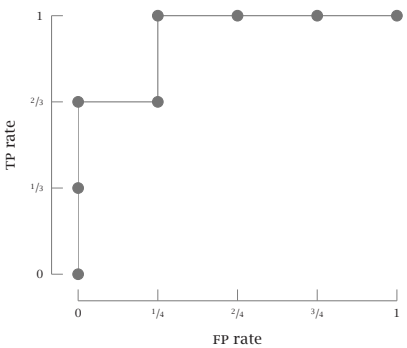


Figure 3.14 For different thresholds τ we get different points on the ROC curve. The area under the curve is $^8/_9$.

Ideally, we are in the upper left-hand corner which corresponds to zero FP and retrieving all TP. All models will be above the diagonal, or we would invert their prediction to obtain a better model. The plot also shows a diagonal line where for every true positive of such a model, we are just as likely to encounter a false positive. So the diagonal itself can be interpreted as a model that just randomly guesses the labels. So the operating part of the ROC diagram is the upper left triangle, and the farther to the northwest the better.

To assess the performance of a model in a single number, we can measure the area under the curve (AUC). The closer the area is to 0.5, the less accurate the corresponding model is. A model with perfect predictive performance will have an area of 1. This number is independent of the threshold τ and hence gives a more objective assessment of the model performance.

Cost-Sensitive Classification

Until now, we treated all errors as equal. However, there are cases where some errors are more serve than others, *e.g.* in medical diagnosis. Here, an actual positive case should not go undetected, but a false positive may only entail the costs of some additional tests. If the costs for errors are known, they can be incorporated into the confusion matrix, multiplying each loss with the associated costs.

A slight modification of the objective function allows the optimizer to respect the costs:

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} c(\boldsymbol{x}_i) \ell(h(\boldsymbol{x}_i), y_i)$$

where the function $c : \mathscr{X} \to \mathbf{R}$ assign a cost to each input. The higher the cost factor, the higher is the penalty induced by the loss.[1]

[1] The same method can also be applied to regression if some instances are more important to get right than others.

Nonlinear Modeling and Generalization

This chapter endows our linear models with the ability to fit nonlinear relationships between the inputs and outputs. However, this will lead to an undesired behavior: The model will have a very low loss on the data set we train it on, but will perform poorly on previously unseen data. The problem is, the model is *overfitting* the training data set and does not *generalize*.

Nonlinear Modeling

There was a good reason why we only considered the summer months in our task of predicting the peak electrical demand from high temperature in the beginning, as this relationship is well-modeled by a *linear* function.

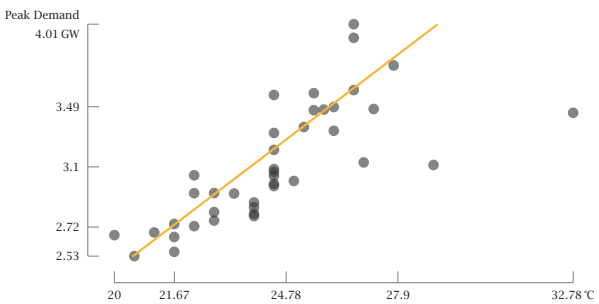


Figure 4.1 High temperature vs. peak demand for a sample of summer months (June – August). Tick marks are at min., max., mean ±std. The data exhibit a clear linear trend.

We reasoned that the electricity demand in the hotter summer months is dominated by air conditions. With higher temperature, more electricity for air conditioning is required. This assumption does certainly not hold if we consider the whole year. Air conditioning is not used in winter months. Still, our previously fitted linear model believes that with lower temperatures we would continue to have a steadily lower electricity consumption.

However, this is not the cases in reality as shown in Fig. 4.2, where the electricity demand rises again, as temperature declines. Since, at least in

San Diego, heating is mostly done with gas rather than with electricity, the demand in colder months does not increase as steeply as in the hotter seasons.

A linear model does a very poor job of fitting this data with a clear nonlinear relationship. Although the line in Fig. 4.2 is actually the best fit in the least squares sense, *i.e.*, the best we can hope to achieve with a linear model.

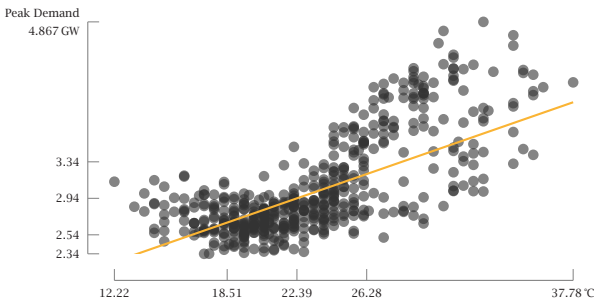


Figure 4.2 Best fit of a line to the full peak demand data set with least squares regression.

In order to do better, we need a way to express the nonlinear relationship between the inputs and outputs. Surprisingly, such a behavior can be realized while simultaneously keeping a linear model for which we already know how to optimize it and also have all kinds of theoretical guarantees.

Instead of making the model more complex, we simply augment the input space with new features. For example, replacing the original inputs such as *temperature* or *weekday*, by features that include nonlinear functions of these quantities. This is done by a so-called feature function ϕ. An example would be the inclusion of the squared *temperature* into the input vector:

$$\phi : \boldsymbol{x} \mapsto \begin{bmatrix} 1 \\ \text{temperature} \\ \text{temperature}^2 \end{bmatrix}$$

The linear hypothesis functions becomes essentially a nonlinear model as

$$h(\boldsymbol{x}) = \langle \boldsymbol{w}, \phi(\boldsymbol{x}) \rangle = w_3 \cdot \text{temperature}^2 + w_2 \cdot \text{temperature} + w_1$$

The model is now a *quadratic* function of the high temperature (see Fig. 4.3). Importantly, however, the hypothesis function is still a *linear* function of the parameters $\boldsymbol{w}$. Therefore, though they may seem limited, linear hypothesis classes are very powerful in combination with appropriate feature functions.

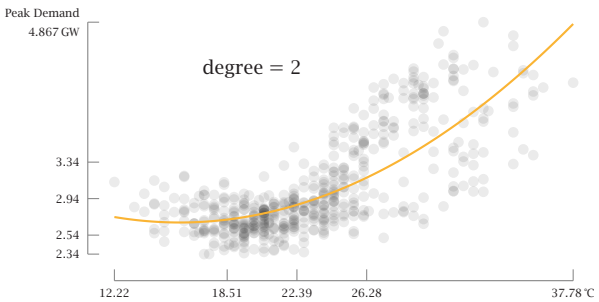


Figure 4.3 A quadratic function of the high temperature (polynomial of degree two) fitted to the data.

This already looks better than the line, but quadratic functions are symmetric around their minimum, and the data is definitely not. However,

the feature function allows us to effortless include additional higher order terms of the high temperature input, which yields higher order polynomials.

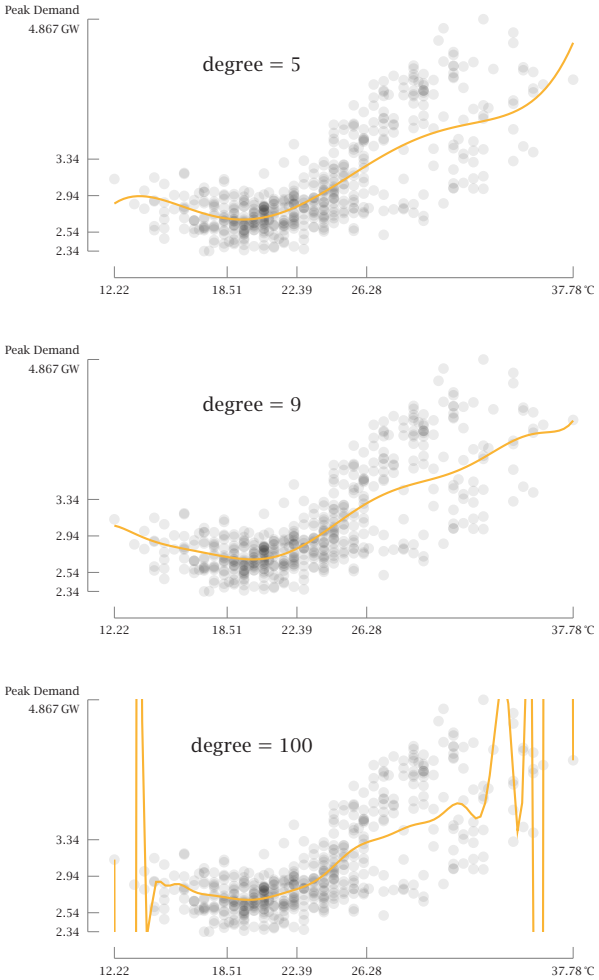


Figure 4.4 Linear regression with polynomial features. The polynomials of degree 5 and 9 seem to be a good model for the data. A polynomial of degree 2 is not sufficient as shown before. A degree 100 polynomial *overfits* the data.

The fit seems to become better with polynomials of higher order, however when choosing the degree too high a very odd pattern emerges: The degree 100 polynomial heavily oscillates with very high amplitude. Our intuition should be, that this is not a good approximation to the prediction task as nobody can possibly think a demand of 124 057 GW at 37.5 ℃ is reasonable.

Nonetheless, the important point here is, that the degree 100 polynomial is actually a very good fit to the data in a least squares sense. The high order polynomial passes through many of the data points *exactly*, yielding a zero loss for these points. This is exactly what we asked the optimizer (gradient descent) to do: minimize the objective.

Measuring Model Performance

The fundamental problem is, that we are optimizing something we actually do not care for. By minimizing the mean loss on the (training) data set, *i.e.*,

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} \ell(h(\boldsymbol{x}_i), y_i)$$

we are building a prediction model, which can accurately predict the peak demand on *past* data. However, what we really care about is to make predictions about the peak demand *tomorrow*, given the high temperature forecast. Thus, when building prediction models, the primary goal should be to make a model that most accurately predicts the desired target value for new, previously unseen data, *i.e.* we want the model to generalize well.

The higher degree polynomial in Fig. 4.4 shows signs of *overfitting*: It actually has a very low loss on the training data, called training error, but we expect the obtained model to have a high error on previously unseen data, called generalization error.

Mathematically, the generalization error is defined as the expectation

$$\mathbf{E}[\ell(h(X), Y)] = \int \ell(h(X), Y) \, \mathrm{d}\mathbf{P}_{XY}$$

In other words: The generalization error is the loss we expect to see when predictions are made for new instances which come from the *same* distribution as the training data.

Naturally, any model is highly optimized for the data it was trained on. The expected error the model has on new data will always be higher than the error it has on the training data (in expectation).

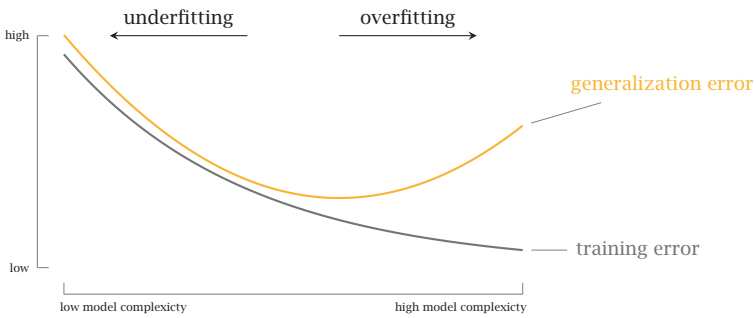


Figure 4.5 As model becomes more complex, training loss always decreases; generalization loss decreases to a point, then starts to increase.

When the model complexity is low, both training and generalization error are high, as a very simple model (as in Fig. 4.2) is not able to capture the relationship between input and output: the model is underfitting.

Increasing the model complexity will decrease the model training error. Very complex models with many parameters should be able to perfectly

predict every single point in the training data set and the training error should be near zero. The additional complexity allows the model to fit the points in the training set exactly, but leads to bad predictions on new data, thus increasing the generalization error. This is a case of overfitting and has to be prevented, in order to build robust models that generalize well.

The goal is therefore to find the model complexity with the lowest generalization error. The problem is, that we do not have access to the true generalization error, since we only have a data set of training points of past events. However, there are techniques to estimate the generalization error.

Cross Validation

One of the simplest techniques to estimate the generalization error is the holdout set method. It is a statistical model validation method for assessing how a model will generalize to an independent data set of unseen instances. The idea is to randomly split the available data into two groups, a new training set and a test set.

The idea is to ignore the test set during training and then measure the generalization error by pretending, that the test set is new data. This approach works because, unlike the training set, the model parameters are *not* optimized based upon the test set and therefore, it is not possible to overfit on this data. Thus, the test set provides a reasonable estimate for the generalization error.

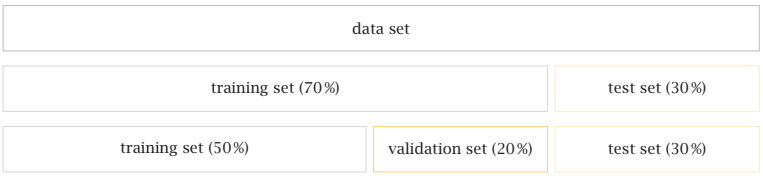


Figure 4.6 Good: A split of the data set into train and test set. Better: A split of the data set into train, validation, and test.

However, in practice, we need an additional split: the validation set. The reason is, that we are only allowed to use the test set *exactly once*. If we want to make a decision based on the generalization error (like the selection of model complexity), we have no more unseen data available to estimate how good this decision was. The idea is to use the validation set to select the model complexity based on a first estimate of the generalization error.

After a satisfying model complexity was found, the training and validation set can be combined to train a final model whose final generalization error is then measured on the test set. After this, a new test set has to be collected.

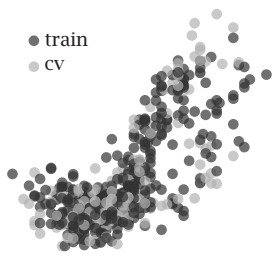


Figure 4.7 The peak demand data split into training and validation set to be used for cross validation.

The effect of the polynomial degree on the model performance can be estimated with holdout set method as follows. We split the peak demand data set into 70% training (1279 instances) and 30% validation (547 instances) as illustrated in Fig. 4.7. Then train a linear model with features of various polynomial degrees and estimate the generalization error. Careful: Not

only the parameter estimation, but also the calculation of the values for feature scaling (and all other preprocessing steps) have to be performed on the training set alone and must never include data from the validation or test set as these are there to simulate new data.

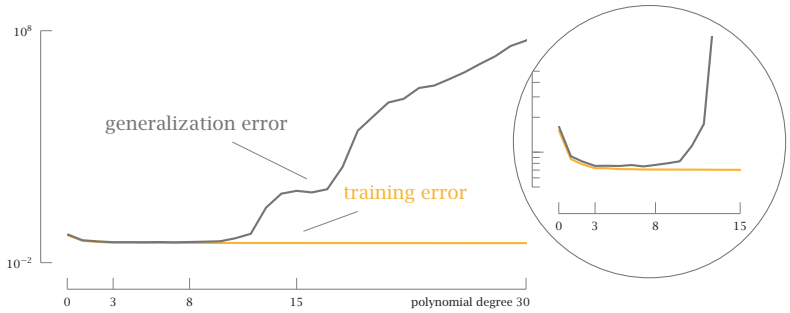


Figure 4.8 Mean squared error on the peak demand data set averaged over 10 runs with random splits of the data set into training and validation (70% and 30% respectively). Polynomials between degree three to eight seem to have the lowest cross validation error.

As expected, the error on the training and validation set is high in the beginning and both start to decline as we increase the degree of the polynomial (the model complexity) as shown in Fig. 4.8 However, as we continue, there is a drastic increase in the generalization error, caused by the high curvature of the polynomial which tries to fit the training points exactly. For this particular problem, we would choose a polynomial degree somewhere between three and eight, but with a tendency to the lower complexity models.

k-fold cross validation

The disadvantage of the holdout set method is, that there is a significant portion of the data, that is never seen during training and another one never used to estimate the generalization error.

A solution to this problem is called *k*-fold cross validation, an approach that holds out less data. It partitions the data set into k folds, train on $k-1$ folds and use the remaining one for validation. For example, using 5-fold cross validation on the peak demand data set with 1826 instances would create 5 folds of approximately 365 data points each. The estimation of the generalization error is repeated 5 times. Each time four of the folds are combined (resulting in 1460 data points) and used to train the model. Then the 5th fold of 365 points, which was not used to fit the model, is used to estimate the generalization error.

The validation errors of all five runs are finally *averaged* and called cross validation error. As an additional advantage, this methodology does not only get a statistically more accurate estimation of the generalization error by averaging, but also provides *standard deviations*.

Fold	data set					Validation Error
1	validation	training	training	training	training	ϵ_1
2	training	validation	training	training	training	ϵ_2
3	training	training	validation	training	training	ϵ_3
4	training	training	training	validation	training	ϵ_4
5	training	training	training	training	validation	ϵ_5

Figure 4.9 Illustration of dataset division for k-fold cross validation. In this example five-fold.

The choice on the number of folds is always a compromise: If the folds are smaller, the estimates of the generalization error are better, but the procedure becomes overall computationally more expensive. On the extreme, we can have one fold for each data point ($k = m$), which is then known as leave-one-out cross validation. This approach trains on all but one of the instances, tests on that one instance, and repeats the whole process m times. In practice however, a 5-fold or 10-fold cross-validation appears to be often sufficient for and estimation of the generalization error.

Cross validation is probably the most widely-used method to estimate how a model will perform in the future. However, this approach only works, the test data really "looks" like the training data: both must come from the *same* distribution $\mathbf{P}_{XY}$. There are situations where this assumption does not hold. For example, we trained a model to predict the electricity demand of San Diego based on temperature. We can expect the model to perform well for cities with similar development and geographical location. It might, however, fail to make predictions for countries where air condition is not commonly installed in homes or the temperatures are totally different.

It is also not always to split the data set randomly. If, for example the data is of *temporal* nature, we must always use past data to predict the future as it would be the case after deployment of the model.

In summary, the best approach is to evaluate the model "in the wild" where it will see truly novel instances. If this is not possible, we can use parts of our data set (which are not used for training) to "simulate" novel examples:

1. Split all data into training/testing sets (*e.g.*, 70% / 30%).
2. Split the training set into training/validation set (*e.g.*, 70% / 30% using the holdout set method or split into k folds).
3. Choose the hypothesis class with the help of cross validation.
4. (Optional) once a good hypothesis class is found, retrain the model on the combiend training and validation set.
5. Finally, evaluate the performance on the test set (which is then never allowed to use a second time).

Bias-Variance Tradeoff

Now, that we know how to measure the performance, we need to understand the different sources of errors in prediction, which then guides us in various decisions we have to make during the modeling and fitting process.

Figure 4.10 Graphical illustration of bias and variance after Scott Fortmann-Roe. Therein, $h^\star(X)$ is the center of the target. Each dot represents a model $h(X)$ trained on a particular data set. Hence, $h(X) - \mathbf{E}[h(X)]$ measures the deviation of each training from the average model and $\mathbf{E}[h(X)] - h^\star(X)$ measures how much the average model deviates from the optimal one.

Consider again the data distribution $\mathbf{P}_{XY}$. We imagine that there is an unknown function $h^\star$ which relates the input and output (random) variables in the form of

$$Y = h^\star(X) + \epsilon \tag{4.1}$$

where ϵ is a zero mean random variable reflecting the ambiguity, noise or other unknown effects in the data set. So, what we observe are noisy versions of $h^\star(X)$.

The goal is to decompose the generalization error $\mathbf{E}[\ell(h(X), Y)]$ into *bias* (how much does the mean of the model differ from the optimal model) and *variance* (how much does the model vary about its mean for different training datasets) terms. Consider, *e.g.*, the square loss. Then

$$\begin{aligned} \mathbf{E}[(Y - h(X))^2] &= \mathbf{E}[\mathbf{E}[h(X)] - h^\star(X)]^2 && \text{(bias}^2\text{)} \\ &+ \mathbf{E}[(h(X) - \mathbf{E}[h(X)])^2] && \text{(variance)} \\ &+ \mathbf{V}[\epsilon] && \text{(noise).} \end{aligned} \tag{4.2}$$

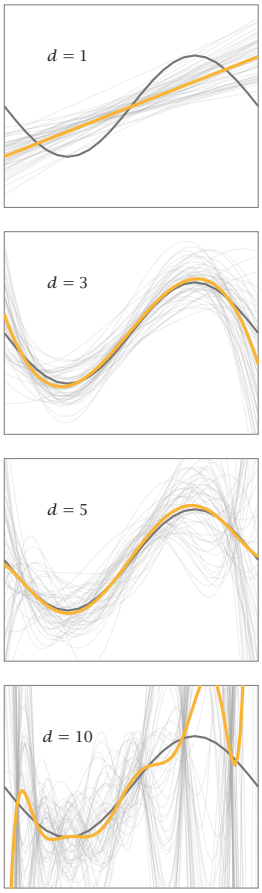


Figure 4.11 Samples of $h(X)$ (——) and their expectation $\mathbf{E}[h(X)]$ (——) using polynomial regression of various degrees d. The true function $h^\star(X)$ is plotted as ——. High bias arises from a low model complexity. In contrast, a high model complexity results in a high variance.

Error due to bias: This is the inherent error made by the model, even if we would have access to infinite training data. We had this problem as we tried to fit a line to the peak demand data set with nonlinear relationships between inputs and outputs (Fig. 4.2). No amount of additional training data would improve this underfitting model.

Error due to variance: This is the variability of a model prediction for a given data point, *i.e.*, how much the fitted model depends on the particular samples from the data distribution or (informally) a training set. An overfitting model which passes through each training point exactly would look very different on varied data sets.

Error due to noise: This error measures ambiguity due to your data distribution and feature representation. This is an aspect of the data itself and cannot be reduced without additional features such as, for example, the weekday feature of the peak demand data set (Fig. 2.3).

An *underfitting model* is characterized by high bias, low variance. Both, training and generalization error are high. In general, such a model is too simple, and we need to increase its complexity. An *overfitting model* is characterized by low bias, high variance. Only the generalization error is high. In general, such a model is too complex. In a *good model*, bias and variance are balanced and the generalization error is low.

Detect and Correct high Variance and Bias

Learning curves show the training error and validation error in dependence on the sample size of the training data. The complexity of the model stays constant during the analysis. We can use learning curves to answer the question of how much training data is needed and how much bias or variance is present in the model. The former is quite simple to answer: As long as both, the training error increases and the generalization error decreases, add more training data until eventually a high bias situation emerges.

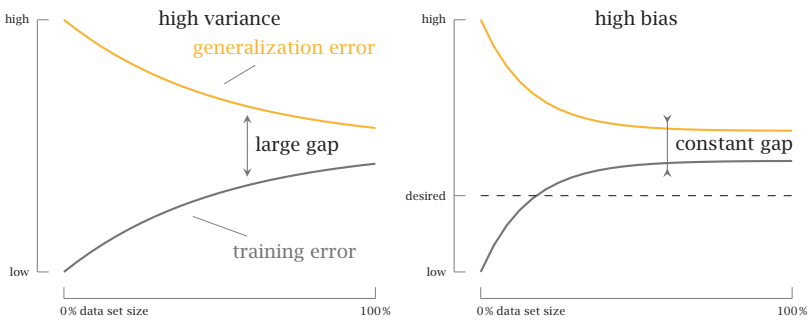


Figure 4.12 Idealized learning curves detecting high variance (left) and high bias (right).

High Variance

symptoms: training error is much lower than generalization error

remedies: add more data, reduce model complexity, bagging

High Bias

symptoms: training error is higher than desired

remedies: increase model complexity, add features, kernels, boosting

A model has *high variance*, if the training and generalization error are quite different from each other, with a large gap between both. In this case, the training data set is fitted much better than the validation data, which indicates problems with the generalization capabilities of the model. In this case, we should add more training data or reduce the model complexity.

A model has *high bias*, if the training and generalization error converge to a value which is much larger than the desired generalization error. This

indicates an underfitting of the model and increasing the sample size will not improve the results. We should use a model with higher complexity or add more features.

Bias and variance should always be balanced. Since bias is reduced and variance is increased in relation to model complexity, it is really about dealing with over- and underfitting. As we add more parameters to the model, its complexity increases and variance becomes our primary concern while bias steadily declines.

Figure 4.13 Bias and variance contributing to total error. As the model complexity is increased, variance becomes our primary concern while bias steadily declines.

Understanding bias and variance is critical for understanding the behavior of prediction models, but ultimately, we want to minimize the overall error. Unfortunately, there is no analytical way to achieve this. Instead, we have to accurately measure the generalization error for different levels of model complexity and then choose the one that minimizes the overall error.

Proof 4.1 (Bias Variance Decomposition). First notice, that the trained models depend on the random samples which form the training data set and are therefore random variables (random functions) themselves with distribution denoted by $\mathbf{P}_H$. Hence, the notation in Section 9.1 is a little imprecise: we are interested in $\mathbf{E}_{XY}\,\mathbf{E}_H[(Y - h(X))]$. This is one of the main confusion about the bias variance decomposition in literature. To ease notation, let $\overline{h} = \mathbf{E}_H[h(X)]$ in the following.

$$
\begin{aligned}
&\mathbf{E}_{XY}\,\mathbf{E}_H[(Y - h(X))^2] = \iint (Y - h(X))^2 \,\mathrm{d}\mathbf{P}_H \,\mathrm{d}\mathbf{P}_{XY} \\
&= \mathbf{E}_{XY}\,\mathbf{E}_H[(Y - \overline{h} + \overline{h} - h(X))^2] \\
&= \mathbf{E}_{XY}\,\mathbf{E}_H[(Y - \overline{h})^2 + (\overline{h} - h(X))^2 - 2(Y - \overline{h})(\overline{h} - h(X))] \\
&= \mathbf{E}_{XY}\,\mathbf{E}_H[(Y - \overline{h})^2] + \mathbf{E}_{XY}\,\mathbf{E}_H[(\overline{h} - h(X))^2] - 2\,\mathbf{E}_{XY}[(\mathbf{E}_H[Y] - \overline{h})(\underbrace{\overline{h} - \overline{h}}_{=0})] \\
&= \mathbf{E}_{XY}\,\mathbf{E}_H[(Y - \overline{h})^2] + \mathbf{E}_{XY}\,\underbrace{\mathbf{E}_H[(\overline{h} - h(X))^2]}_{\text{bias}^2}
\end{aligned}
$$

Notice, that in the first term, neither Y nor $\overline{h}$ depends on H, and we can therefore omit the expectation $\mathbf{E}_H$. This gets us the variance and noise terms as

$$
\begin{aligned}
\mathbf{E}_{XY}\,\mathbf{E}_H[(Y - \overline{h})^2] &= \mathbf{E}_{XY}[(Y - \overline{h})^2] \\
&= \mathbf{E}_{XY}[(h^\star(X) + \epsilon - \overline{h})^2] \\
&= \mathbf{E}_{XY}[(h^\star(X) - \overline{h})^2] + \mathbf{E}_{XY}[\epsilon^2] - 2\,\mathbf{E}_{XY}[\underbrace{\epsilon}_{\mathbf{E}_{XY}[\epsilon]=0}(h^\star(X) - \overline{h})] \\
&= \mathbf{E}_{XY}[(h^\star(X) - \overline{h})^2] + \mathbf{V}_{XY}[\epsilon]
\end{aligned}
$$

which leaves us with the bias variance decomposition

$$\begin{aligned}
\mathbf{E}_{XY}\,\mathbf{E}_H\big[(Y - h(X))^2\big] &= \mathbf{E}_{XY}\,\mathbf{E}_H\big[\mathbf{E}[h(X)] - h^\star(X)\big]^2 && (\text{bias}^2) \\
&+ \mathbf{E}_{XY}\big[(h(X) - \mathbf{E}[h(X)])^2\big] && (\text{variance}) \\
&+ \mathbf{V}_{XY}[\epsilon] && (\text{noise}).
\end{aligned}$$

□

Regularization

For far, we used the degree of the polynomial feature function as a means to control model complexity. Recall, that the degree 100 polynomial overfits the peak demand data since it tries to pass through many of the data points with the aim to minimizing the objective function (these points contribute a loss of zero). Now, for the polynomial to pass exactly through the points requires extremely high values for the coefficients. Remember from Fig. 4.8, that we only need a degree three to nine polynomial to build a model that generalizes well on this particular data set. If we include additional polynomial terms, least squares exploits these to pass through some points exactly, minimizing the training error further. This is accomplished by very high weights on the high-order coefficients.

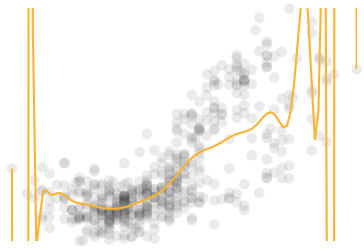


Figure 4.14 A degree 100 polynomial overfitting on the peak demand data set.

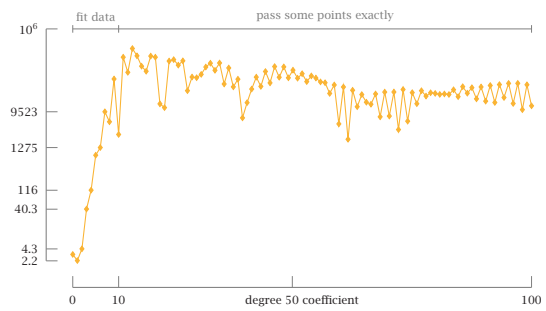


Figure 4.15 Absolute value of polynomial coefficients. Extremely high values for the coefficients are necessary, in order to pass through some data points exactly.

The idea of regularization is to control the complexity of the model class by restricting the parameters in one way or another (in our case reducing their magnitude). This leads to the regularized loss minimization problem, where, instead of simply minimizing the average loss, we minimize the average loss plus a term $\Omega(\boldsymbol{w})$ that penalizes the model complexity:

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} \ell(h(\boldsymbol{x}_i), y_i) + \lambda \Omega(\boldsymbol{w}) \tag{4.3}$$

where the regularization parameter $\lambda \geq 0$ balances the penalty for a high average loss and the penalty for complex models. By varying λ from zero (no regularization) to infinity (infinite regularization, meaning parameters will all be zero), we can adjust for different sets of model complexity. This is the type of regularization we mentioned in our discussion about the linear support vector machine.

As with different loss functions, there are several regularizers Ω available, each then again leading to a different machine learning algorithm with its own advantages and disadvantages. Two common regularized regression techniques are discussed in the following: *Ridge regression* and *lasso*.

One of the simplest regularization functions is $\Omega(\boldsymbol{w}) = \sum_{i=1}^{n} w_i^2 = \|\boldsymbol{w}\|^2$, often called *Tikhonov* or ℓ_2 regularization. Applying the Tikhonov regularization to linear regression with the square loss yields the objective

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle - y_i)^2 + \lambda\|\boldsymbol{w}\|^2 \tag{4.4}$$

which is then known as the ridge regression machine learning algorithm.

The gradient of the objective function is simply the gradient of the average loss plus the gradient of the regularizer.

$$\begin{aligned}\nabla J(\boldsymbol{w}) &= \nabla\frac{1}{m}\sum_{i=1}^{m}(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle - y_i)^2 + \nabla\lambda\langle \boldsymbol{w}, \boldsymbol{w}\rangle \\ &= \frac{2}{m}\sum_{i=1}^{m}\boldsymbol{x}_i(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle - y_i) + 2\lambda\boldsymbol{w} \\ &= \frac{2}{m}X^\mathsf{T}(X\boldsymbol{w} - \boldsymbol{y}) + 2\lambda\boldsymbol{w}\end{aligned}$$

The gradient descent rule for ridge regression is thus as follows.

repeat: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta\frac{2}{m}X^\mathsf{T}(X\boldsymbol{w} - \boldsymbol{y}) + 2\lambda\boldsymbol{w}$

Algorithm 4.1 The gradient descent update rule for ridge regression.

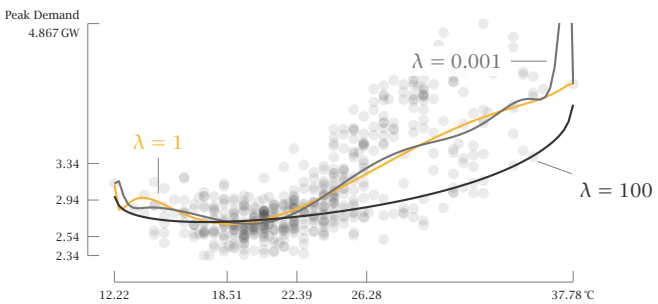


Figure 4.16 The effect of regularization on the fitted linear regression model with polynomial degree 100. Regularization with $\lambda = 1$ (—♦—) seem to have a positive effect. A lower regularization (—●—) *overfits* the data, a higher regularization (—■—) *underfits*.

Ridge regression, like least squares, has a closed from solution $\boldsymbol{w}^\star$ which can be obtained by setting the gradient equal to zero, as

$$\begin{aligned}\nabla J(\boldsymbol{w}^\star) = 0 &\iff \frac{2}{m}X^\mathsf{T}(X\boldsymbol{w}^\star - \boldsymbol{y}) + 2\lambda\boldsymbol{w}^\star = 0 \\ &\iff X^\mathsf{T}X\boldsymbol{w}^\star + m\lambda\boldsymbol{w}^\star = X^\mathsf{T}\boldsymbol{y} \\ &\iff (X^\mathsf{T}X + m\lambda I)\boldsymbol{w}^\star = X^\mathsf{T}\boldsymbol{y}\end{aligned}$$

and thus the value of $\boldsymbol{w}^\star$ that minimizes J can be obtained by the equation

$$\boldsymbol{w}^\star = (X^\mathsf{T}X + m\lambda I)^{-1}X^\mathsf{T}\boldsymbol{y}, \tag{4.5}$$

in closed form.[1]

[1] Some authors merge the constant m into the regularization parameter λ and hence omit it in the equation.

In Fig. 4.16, we apply ridge regression with different regularization strengths to the peak demand data set. The resulting model with a regularization parameter of $\lambda = 1$ looks substantially better than the one obtained without regularization. If more regularization is added ($\lambda = 100$), the model essentially underfits since nearly all weights are forced to be close to zero.

As we know from the previous discussion, a lower-degree polynomial is sufficient to fit the data reasonable well. Adding enough regularization to the optimization objective forces the algorithm to concentrate on these low-order terms for fitting and avoids putting much weight on the high-order terms of the polynomial. This is also reflected in Fig. 4.17. The model puts very little weight on the coefficients after degree nine or ten. Therefore, adding regularization to a complex model has a similar effect as using a simple one model.

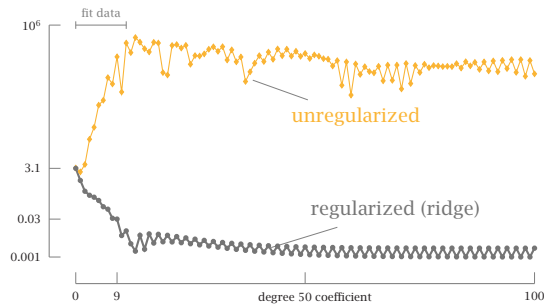


Figure 4.17 Sufficient regularization has the effect of less weight on the high-order terms since a lower-degree polynomial can already achieve a low error on the data set.

We can now apply a similar analysis as in Fig. 4.8 and relate the amount of regularization to the performance of the model on the training and validation sets.

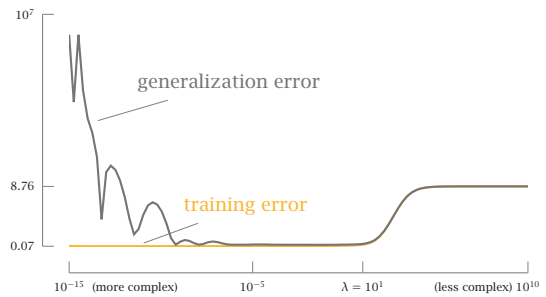


Figure 4.18 Mean squared error on the peak demand data set averaged over 100 runs with random splits of the data set into training and validation (70 % and 30 % respectively). A regularization with λ between 10^{-5} and 10^1 appears to perform best on this data set.

This actually looks quite similar to Fig. 4.8, just in reverse order as higher regularization leads to simpler models. In the beginning, our model overfits the training set, leading to a high generalization error. As more and more regularization is added, the model continues to generalize better until, eventually the model is too restricted (essentially corresponding to all zero coefficients) to fit the training set and underfits. As we see from the logarithmic scale on the axis, regularization is typically applied on a scale of orders of magnitude. This should be considered in a search for a good regularization parameter.

Lasso Regression

Similar to ridge regression, there is another regularized regression technique called lasso regression which uses ℓ_1 regularization, defined as $\Omega(\boldsymbol{w}) = \sum_{i=1}^{n} |w_i| = \|w\|_1$. The only difference between lasso regression and ridge regression is the type of regularization.

If the regularization parameter λ is big enough, some weights are forced to be exactly zero, which makes it easier to understand the data and yields a model that is faster to evaluate. A solution $\boldsymbol{w}$ with many zeros is said to be *sparse* and can be observed in Fig. 4.19, were the weights of all coefficients above degree 30 are (within numerical accuracy) essentially zero.

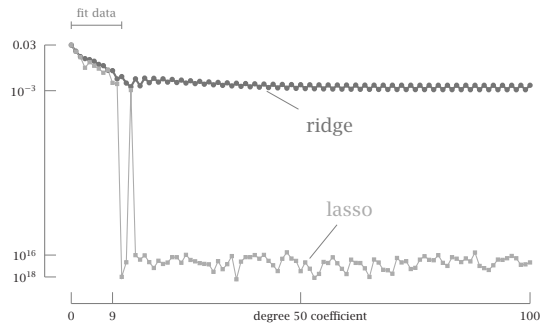


Figure 4.19 With ℓ_1 regularization, most of the weights are forced to be essentially zero. ℓ_2 regularization only drives the parameters small.

Sparsity comes at the cost of having no closed form solution anymore. Since the absolute value function is non-differentiable at zero, the objective function does not have a gradient, but only a *sub*gradient at this point, *i.e.*

$$\begin{aligned}\nabla J(\boldsymbol{w}) &= \nabla \frac{1}{m} \sum_{i=1}^{m} (\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle - y_i)^2 + \nabla \lambda \sum_{i=1}^{n} |w_i| \\ &= \frac{2}{m} \sum_{i=1}^{m} \boldsymbol{x}_i (\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle - y_i) + \lambda \sum_{i=1}^{n} \operatorname{sign}(w_i)\end{aligned}$$

Even though, no analytical solution is available for lasso regression, it is still possible to apply (sub-) gradient descent to minimize this objective.

The advantage of a sparse solution is, that all features with zero weight can be discarded, as these will not contribute anything to the prediction of the model. In addition, these zero weight features can be seen as irrelevant to the problemm which makes it easier to understand the models decisions.

Regularization as Constraint Optimization

Why does lasso regression yield sparse solutions? To answer this question, it helps to change the look at regularizers from a penalty to a constraint: It can be shown, that for each $\lambda \geq 0$, there exists a $B \geq 0$ such that

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} \ell(h(\boldsymbol{x}_i), y_i) + \lambda \Omega(\boldsymbol{w}) \qquad \text{and}$$

$$\begin{aligned} \underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad & J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} \ell(h(\boldsymbol{x}_i), y_i) \\ \text{subject to} \quad & \Omega(\boldsymbol{w}) \leq B \end{aligned}$$

are equivalent.[1] This means we are looking for parameters $\boldsymbol{w}$ that minimize the (unregularized) objective function — these are the ones that are closest to $\boldsymbol{w}^\star$ — but do not escape the "ball" of radius B.

[1] The KKT conditions explain the relationship between constrained minimization and regularization. The upper formulation is the generalized Lagrangian of the constrained optimization problem below.

This concept is illustrated in Fig. 4.20 on the initial linear regression problem on the peak demand data set. The best feasible points will be on the border of the constraints. In the case of ℓ_1 regularization, w_1 is exactly zero since, if not, we can further decrease the objective function by walking along the edge of the ℓ_1-norm ball. In contrast, when using ℓ_2 regularization, w_1 becomes only very small, but not zero since the optimal solution is exactly at the intersection of the line from the origin to $\boldsymbol{w}^\star$ on the ℓ_2-norm ball.

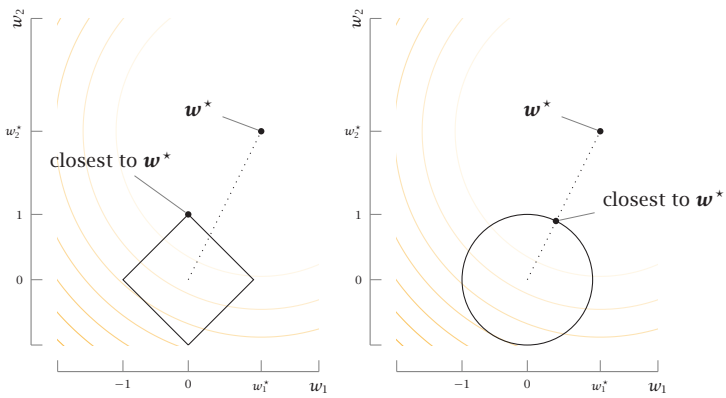


Figure 4.20 The objective function contours overlayed with the ℓ_1-norm unit ball (left) and ℓ_2-norm unit ball (right).

In Fig. 4.20, the connection between regularization and early stopping becomes obvious. The latter stops optimization after $t \geq 0$ gradient steps, even if optimization has not yet converged. It is computationally cheaper, but without explicit constraints, the points within the parameter space, that can be reached with early stopping, heavily depend on the initialization.

Nonlinear Classification

Nonlinear classification in the empirical risk minimization framework is not different from nonlinear regression; it just requires a different choice of loss and an appropriate regularizer.

Consider again the rock type data set, which was not linearly separable. This time we use polynomial features to obtain a nonlinear decision boundary. In contrast to the peak demand data set we do not have only one input variable, but two ($\boldsymbol{x} \in \mathbf{R}^2$). If we consider all polynomial terms of degree up to degree $d = 3$, we get

$$\phi(\boldsymbol{x}) = (1,\ x_1,\ x_1^2,\ x_1^3,\ x_2,\ x_2^2,\ x_2^3,\ x_1x_2,\ x_1x_2^2,\ x_1^2x_2)$$

as a feature function. For general input spaces of n dimensions, we express this in set builder notation as

$$\phi(\boldsymbol{x}) = \left\{ \prod_{i=1}^{n} x_i^{d_i} : d_i \in \mathbf{N} \text{ and } \sum_{i=1}^{n} d_i \leq d \right\}$$

which generates $\binom{n+d}{d}$ terms.

The objective obtained by polynomial features $\phi(\boldsymbol{x})$ together with the logistic loss function and ℓ_2 regularization is

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} \log(1 + \exp(-y_i \langle \boldsymbol{w}, \phi(\boldsymbol{x}_i) \rangle)) + \lambda \|\boldsymbol{w}\|^2$$

for which we already know how to derive the gradient.

A comparison of various amounts of regularization on a linear regression model with a degree 10 polynomial is illustrated in Fig. 4.21.

The same techniques we used to choose the model complexity in regression can also be applied for classification (and any other machine learning task).

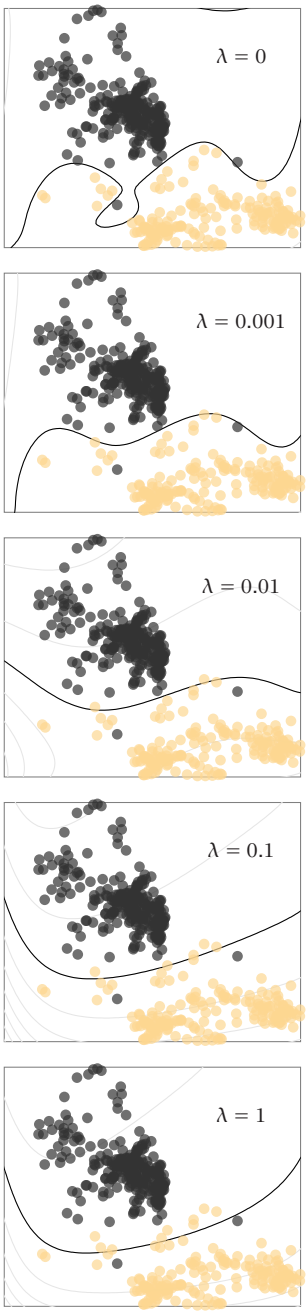
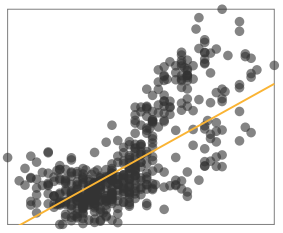
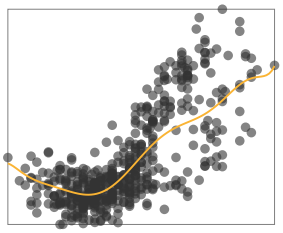
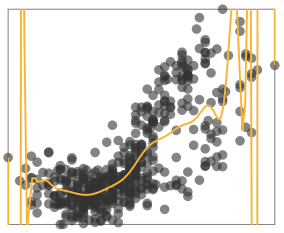
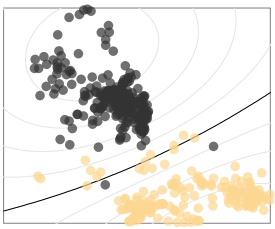
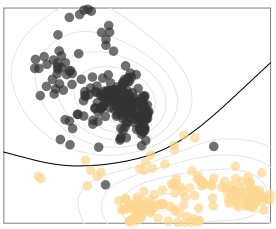
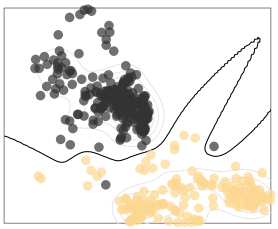
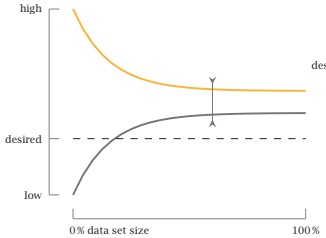
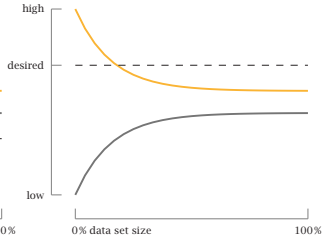
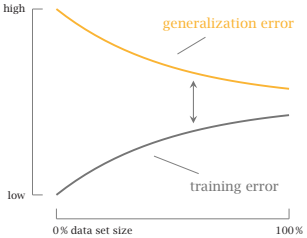


Figure 4.21 The effect of regularization on a logistic regression model with polynomial features of degree 10. Regularization with $\lambda = 0.01$ seems to work best for this setup.

	Underfitting	Good	Overfitting
Symptoms	• high training error • training error close to test error • high bias	• generalization error below desired value	• low training error • training error much lower than test error • high variance
Regression			
Classfication			
Performace Curve			
Remedies	• increase modelcomplexity • add features • kernelize • boosting		• add more data • reduce model complexity • bagging

Kernel Functions

The previous chapter showed that linear methods can fit nonlinear data if an appropriate feature transformations ϕ is applied to the inputs. We effectively gain nonlinear models with the same theoretical guarantees and optimization techniques of linear models without their disadvantages.

However, approach has some other drawbacks: It is, for example, not obvious which feature functions to use, since good features are highly domain dependent. Also, the new feature vector $\phi(\boldsymbol{x})$ might be very high dimensional which can make both, learning and prediction slow.

At their core, kernels are a technique for working with high dimensional feature vectors without explicitly forming the feature vectors themselves. Assume that an algorithm can entirely be expressed in terms of inner products $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z}) \rangle$ between any two inputs $\boldsymbol{x}$ and $\boldsymbol{z}$. We will see, that instead of choosing a transformation ϕ it is possible to choose a kernel k and replace the inner product by the kernel value, as $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z}) \rangle = k(\boldsymbol{x}, \boldsymbol{z})$. The evaluation of the kernel function will be much faster and memory efficient than the original inner product.

The idea behind kernels is based on the representer theorem, which tells us, that for a $\boldsymbol{w}$ minimizing the objective function for a *linear model*, *i.e.*,

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} \ell(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i) \rangle, y_i) + \Omega(\boldsymbol{w})$$

there exists an $\boldsymbol{\alpha} \in \mathbf{R}^m$, such that the optimal solution $\boldsymbol{w}$ will always be a *linear combination* of the features themselves, namely

$$\boldsymbol{w} = \sum_{i=1}^{m} \alpha_i \phi(\boldsymbol{x}_i). \tag{5.1}$$

Proof (Representer Theorem). This constructive proof uses the descent update rule for linear models $h(\boldsymbol{x}) = \langle \boldsymbol{w}, \phi(\boldsymbol{x}) \rangle$. Although, the proof can be conducted with *any* loss function, we adopt in the square loss for clarity.

In the first step, we show that the gradient of the objective function is a linear combination of $\boldsymbol{\alpha}$.

$$\begin{aligned}
\nabla J(\boldsymbol{w}) &= \frac{1}{m}\sum_{i=1}^{m} \nabla \ell(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle, y_i) \\
&= \frac{1}{m}\sum_{i=1}^{m} \nabla(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle - y_i)^2 \\
&= \frac{1}{m}\sum_{i=1}^{m} 2(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle - y_i)\cdot \phi(\boldsymbol{x}_i) \\
&= \sum_{i=1}^{m} \gamma_i \phi(\boldsymbol{x}_i)
\end{aligned}$$

with $\gamma_i = \frac{1}{m}2(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle - y_i)$. The gradient descent update rule is then

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \sum_{i=1}^{m} \gamma_i^{(t)} \phi(\boldsymbol{x}_i).$$

The claim that all parameters can be written as $\boldsymbol{w} = \sum_{i=1}^{m} \alpha_i \phi(\boldsymbol{x}_i)$ can now be proved with mathematical induction.

Base Case: If $\boldsymbol{w}^{(0)} = 0$, then it has the form $\boldsymbol{w}^{(0)} = \sum_{i=1}^{m} \alpha_i \phi(\boldsymbol{x}_i)$ by setting all α_i equal to zero.

Induction Step(s): For all $t \geq 0$, applying the gradient descent update rule

$$\begin{aligned}
\boldsymbol{w}^{(t+1)} &= \boldsymbol{w}^{(t)} - \eta \sum_{i=1}^{m} \gamma_i^{(t)} \phi(\boldsymbol{x}_i) \\
&= \sum_{i=1}^{m} \alpha_i^{(t)} \phi(\boldsymbol{x}_i) - \eta \sum_{i=1}^{m} \gamma_i^{(t)} \phi(\boldsymbol{x}_i) \\
&= \sum_{i=1}^{m} (\alpha_i^{(t)} - \eta \gamma_i^{(t)}) \phi(\boldsymbol{x}_i) \\
&= \sum_{i=1}^{m} \alpha_i^{(t+1)} \phi(\boldsymbol{x}_i).
\end{aligned}$$

which is again a linear combination of the features $\phi(\boldsymbol{x}_i)$.

□

The last two steps of the proof also show how gradient descent updates for *linear* models can be performed entirely in terms of α without ever expressing $\boldsymbol{w}$ explicitly namely

repeat: $\alpha_i \leftarrow \alpha_i - \eta \gamma_i$

Algorithm 5.1 The gradient descent update rule for linear models.

Given $\boldsymbol{w}$ in the form of $\boldsymbol{w} = \sum_{i=1}^{m} \alpha_i \phi(\boldsymbol{x}_i)$, the original hypothesis function of a linear model $h(\boldsymbol{x}) = \langle \boldsymbol{w}, \phi(\boldsymbol{x})\rangle$ becomes

$$h_{\boldsymbol{\alpha}}(\boldsymbol{x}) = \sum_{i=1}^{m} \alpha_i \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x})\rangle \tag{5.2}$$

where we use the notation $h_{\boldsymbol{\alpha}}$ to emphasize that the parameters of the model are now $\boldsymbol{\alpha} \in \mathbf{R}^m$ and not $\boldsymbol{w}$ anymore. Within this equation, the feature maps ϕ only occur in the form of inner products and never alone.

This leads us to the key idea of kernel functions, *i.e.*, that we can replace $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z})\rangle$ by a kernel k such that we can compute the inner product between these feature vectors without ever explicitly computing the vectors.

Why are feature functions alone problematic, but not their inner products? Consider, for example, the polynomial feature function of degree $d = 2$ for a two-dimensional input $\boldsymbol{x} \in \mathbf{R}^2$:

The additional constants $\sqrt{2}$ are just there to ease the notation in the following.

$$\phi \colon \mathbf{R}^2 \to \mathbf{R}^6, \qquad \phi(\boldsymbol{x}) = (1,\ \sqrt{2}x_1,\ x_1^2,\ \sqrt{2}x_2,\ x_2^2,\ \sqrt{2}x_1x_2)$$

The inner product between two of these feature vectors is then

$$\begin{aligned}\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z})\rangle &= 1 + 2x_1z_1 + x_1^2z_1^2 + 2x_2z_2 + x_2^2z_2^2 + 2x_1x_2z_1z_2 \\ &= 1 + 2(x_1z_1 + x_2z_2) + (x_1z_1 + x_2z_2)^2 \\ &= 1 + 2\langle \boldsymbol{x}, \boldsymbol{z}\rangle + \langle \boldsymbol{x}, \boldsymbol{z}\rangle^2 \\ &= (\langle \boldsymbol{x}, \boldsymbol{z}\rangle + 1)^2.\end{aligned}$$

This demonstrates that, in order to evaluate the inner product of two six-dimensional vectors, we actually only have to calculate the inner product of two two-dimensional ones, add one, and then square it. We get the *exact* same result as with the actual inner product, but without ever forming the feature vectors $\phi(\boldsymbol{x})$ and $\phi(\boldsymbol{z})$ explicitly.

This does not seem to accomplish much in the simple example above. But consider the general case of polynomial features with degree d for inputs of dimension n. The feature vector then has $\binom{n+d}{d}$ terms, which is in general way too large to even store explicitly, let alone to evaluate. But, the inner product between any two such features is simply

$$\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z})\rangle = (\langle \boldsymbol{x}, \boldsymbol{z}\rangle + 1)^d$$

which requires only $\mathbf{O}(n)$ (linear) time and $\mathbf{O}(1)$ (constant) space.

If an algorithm can now entirely be expressed in terms of inner products $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z})\rangle$, we can substitute them with $k(\boldsymbol{x}, \boldsymbol{z}) = (\langle \boldsymbol{x}, \boldsymbol{z}\rangle + 1)^d$ and effectively have polynomial features of dimension d, but without their computational and storage costs.

What if we do not want *polynomial* features but something else which might be better to model the data at hand? It turns out, that $k(\boldsymbol{x}, \boldsymbol{z}) = (\langle \boldsymbol{x}, \boldsymbol{z}\rangle + 1)^d$ is not the only possible substitution. There are many maps $k \colon \mathscr{X} \times \mathscr{X} \to \mathbf{R}$, called kernel functions, which can be used; each one leading to different (implicit) feature maps ϕ. The different variants will be discussed shortly.

However, it is *not* possible to use an arbitrary function as a kernel. The map $k \colon \mathscr{X} \times \mathscr{X} \to \mathbf{R}$ must be *symmetric* and *positive semi-definite*. Only then, it is (theoretically) possible to find a corresponding feature map ϕ such that $k(\boldsymbol{x}, \boldsymbol{z}) = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z})\rangle$. This statement is also known as Mercer's theorem.

Proof. Recall, that a function is positive semi-definite iff for any finite set $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_m\}$, the corresponding kernel matrix $K \in \mathbf{R}^{m \times m}$, defined as

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) \tag{5.3}$$

is positive semi-definite.

Symmetry: If there exists feature map ϕ such that $k(\boldsymbol{x}, \boldsymbol{z}) = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z}) \rangle$, then k must be symmetric, since

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle = \langle \phi(\boldsymbol{x}_j), \phi(\boldsymbol{x}_i) \rangle = k(\boldsymbol{x}_j, \boldsymbol{x}_i)$$

Positive Semi-Definiteness: For any vector $\boldsymbol{v} \in \mathbf{R}^m$ and kernel matrix K

$$\begin{aligned}
\boldsymbol{v}^\top K \boldsymbol{v} &= \sum_{i,j} v_i K_{ij} v_j \\
&= \sum_{i,j} v_i \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle v_j \\
&= \langle \sum_i v_i \phi(\boldsymbol{x}_i), \sum_j v_j \phi(\boldsymbol{x}_j) \rangle \\
&= \| \sum_i v_i \phi(\boldsymbol{x}_i) \|^2 \geq 0,
\end{aligned}$$

which tells us that k must be positive semi-definite.

□

Instead of proving Mercer's theorem for a potential kernel function we would like to use, it is also possible to combine existing kernel functions into new ones. In the following we review some approaches, which are known to yield valid kernel functions.

Base Kernels: $k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^\top \boldsymbol{z}$ and $k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^\top S \boldsymbol{z}$ are valid kernel functions for any symmetric, positive semi-definite matrix S.

Sum and Scaling of Kernels: Let k_1 and k_2 be kernel functions and $\alpha, \beta \geq 0$, then the conical sum $\alpha k_1 + \beta k_2$ is also a kernel function.

Product of Kernels: Let k_1 and k_2 be kernel functions, then $k_1(\boldsymbol{x}, \boldsymbol{z}) \cdot k_2(\boldsymbol{x}, \boldsymbol{z})$ is a kernel function.

Exponentiate If k is a kernel function, so is $\exp(k(\boldsymbol{x}, \boldsymbol{z}))$.

Polynomial: If k is a kernel function, so is $g(k(\boldsymbol{x}, \boldsymbol{z}))$, if $g : \mathbf{R} \to \mathbf{R}$ is a polynomial with positive coefficients.

Product with Function: If k is a kernel function, so is $f(\boldsymbol{x}) k(\boldsymbol{x}, \boldsymbol{z}) f(\boldsymbol{z})$ for any function $f : \mathscr{X} \to \mathbf{R}$.

The squared exponential kernel — one of the most popular kernel functions — can be derived with these rules.

With this kernel, all algorithms discussed so far become *universal approximators.* That means, they can approximate any function arbitrary well. The squared exponential itself is defined as

$$k(\boldsymbol{x}, \boldsymbol{z}) = \exp\Big(-\frac{\|\boldsymbol{x} - \boldsymbol{z}\|^2}{2\sigma^2}\Big)$$

for some $\sigma > 0$. The value $k(\boldsymbol{x}, \boldsymbol{z})$ is large if $\boldsymbol{x}$ and $\boldsymbol{z}$ are close together and small if $\boldsymbol{x}$ and $\boldsymbol{z}$ are far apart. It can thus be interpreted as a measure of *similarity* between inputs, a property shared by most kernel functions. This has the advantage, that it is often easier to come up with a similarity (or distance) measure than a good feature function.[1] The length-scale σ

[1] In fact, you can replace the norm in the squared exponential by any valid metric. A popular choice is the Mahalanobis distance, which makes this kernel extremely flexible.

controls what distances are considered "close" and which are "far".

Interestingly, if we would like to decompose the squared exponential kernel into feature maps, we would need polynomial kernels with degrees $d \to \infty$. To see this, we use the Taylor series expansion of the exponential, to get

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{z}) &= \exp\Big(-\frac{\|\boldsymbol{x} - \boldsymbol{z}\|^2}{2\sigma^2}\Big) \\ &= \exp\Big(-\frac{\|\boldsymbol{x}\|^2}{2\sigma^2}\Big) \exp\Big(-\frac{\|\boldsymbol{z}\|^2}{2\sigma^2}\Big) \sum_{d=0}^{\infty} \frac{\langle \boldsymbol{x}, \boldsymbol{z}\rangle^d}{\sigma^{2d} d!}, \end{aligned}$$

where the sum goes over all polynomial kernels of degree $d = 0, \dots, \infty$. Informally, that means, using the squared exponential kernel is equivalent to mapping the inputs into an infinite-dimensional feature space. This is sometimes advantageous since, if the dimension of the feature space is high enough, the data always becomes linearly separable; a property which is, assumed, *e.g.*, by logistic regression and support vector machines.

It is not only possible to define kernel function over vectors $\boldsymbol{x}, \boldsymbol{z} \in \mathbf{R}^n$ but arbitrary structures. If, for example, P and Q are two *sets* over the same finite domain, then

$$k(P, Q) = \exp(|P \cap Q|)$$

is a valid kernel over sets, sometimes called the *bag-of-words* kernel.

Kernelization

By now, the advantages of kernel functions over feature maps should be evident. In order to use them in an algorithm, the following steps have to be performed, which are sometimes referred to as kernelization.

1. Rewrite the model h and the optimization algorithm such that *all* inputs $\boldsymbol{x} \in \mathscr{X}$ only appear in inner products with other inputs $\boldsymbol{z} \in \mathscr{X}$. That is either $\langle \boldsymbol{x}, \boldsymbol{z}\rangle$ or $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z})\rangle$.
2. Choose or define a kernel function k and replace the inner products by the kernel map, *i.e.* substitute $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z})\rangle$ with $k(\boldsymbol{x}, \boldsymbol{z})$.

The kernelization of *linear* hypothesis functions only requires us to replace

$$h_{\boldsymbol{w}}(\boldsymbol{x}) = \langle \boldsymbol{w}, \phi(\boldsymbol{x})\rangle \quad \text{with} \quad h_{\boldsymbol{\alpha}}(\boldsymbol{x}) = \sum_{i=1}^{m} \alpha_i \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x})\rangle,$$

which is again a *linear* model, but now with respect to the parameters $\boldsymbol{\alpha}$. As a result, we can apply all the same techniques we developed previously to find parameters $\boldsymbol{\alpha}$ that minimize the objective, such as gradient descent.

In the following two kernelized linear models are derived: Kernel ridge regression and kernel logistic regression.

Kernel Ridge Regression

To derive kernel ridge regression we follow the two-step procedure before. Observe, that the ℓ_2 regularizer can also be reformulated in terms of $\boldsymbol{\alpha}$ as

$$\|\boldsymbol{w}\|^2 = \langle \boldsymbol{w}, \boldsymbol{w} \rangle = \boldsymbol{\alpha}^\mathsf{T} K \boldsymbol{\alpha}, \tag{5.4}$$

using the opposite direction as in the proof of Mercer's theorem.[1] Thus, the ridge regression objective can therefore be written as

$$\underset{\boldsymbol{\alpha} \in \mathbf{R}^m}{\text{minimize}} \quad J(\boldsymbol{\alpha}) = \frac{1}{m} \sum_{i=1}^{m} \Big(\sum_{j=1}^{m} \alpha_j k(\boldsymbol{x}_j, \boldsymbol{x}_i) - y_i \Big)^2 + \lambda \boldsymbol{\alpha}^\mathsf{T} K \boldsymbol{\alpha}$$

which is equivalent to

$$\underset{\boldsymbol{\alpha} \in \mathbf{R}^m}{\text{minimize}} \quad J(\boldsymbol{\alpha}) = \frac{1}{m} \|K\boldsymbol{\alpha} - \boldsymbol{y}\|^2 + \lambda \boldsymbol{\alpha}^\mathsf{T} K \boldsymbol{\alpha} \tag{5.5}$$

in matrix notation. The gradient of the objective, now with respect to $\boldsymbol{\alpha}$, is

$$\begin{aligned} \nabla_{\boldsymbol{\alpha}} J(\boldsymbol{\alpha}) &= \nabla_{\boldsymbol{\alpha}} \frac{1}{m} \|K\boldsymbol{\alpha} - \boldsymbol{y}\|^2 + \nabla_{\boldsymbol{\alpha}} \lambda \boldsymbol{\alpha}^\mathsf{T} K \boldsymbol{\alpha} \\ &= \frac{2}{m} K(K\boldsymbol{\alpha} - \boldsymbol{y}) + 2\lambda K \boldsymbol{\alpha}. \end{aligned}$$

Setting this quantity equal to zero leads to the analytic solution. It is

$$\begin{aligned} \nabla_{\boldsymbol{\alpha}} J(\boldsymbol{\alpha}^\star) = 0 &\iff \frac{2}{m} K(K\boldsymbol{\alpha}^\star - \boldsymbol{y}) + 2\lambda K \boldsymbol{\alpha}^\star = 0 \\ &\iff KK\boldsymbol{\alpha}^\star + m\lambda K \boldsymbol{\alpha}^\star = K\boldsymbol{y} \\ &\iff K(K + m\lambda I)\boldsymbol{\alpha}^\star = K\boldsymbol{y} \\ &\iff (K + m\lambda I)\boldsymbol{\alpha}^\star = \boldsymbol{y} \end{aligned}$$

and thus the optimal $\boldsymbol{\alpha}^\star$ that minimizes $J(\boldsymbol{\alpha})$ has closed form

$$\boldsymbol{\alpha}^\star = (K + m\lambda I)^{-1} \boldsymbol{y}. \tag{5.6}$$

[1] $$\begin{aligned} \|\boldsymbol{w}\|^2 &= \langle \boldsymbol{w}, \boldsymbol{w} \rangle \\ &= \langle \sum_{i=1}^{m} \alpha_i \phi(\boldsymbol{x}_i), \sum_{j=1}^{m} \alpha_j \phi(\boldsymbol{x}_j) \rangle \\ &= \sum_{i,j}^{m} \alpha_i \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle \alpha_j \\ &= \boldsymbol{\alpha}^\mathsf{T} K \boldsymbol{\alpha}, \end{aligned}$$

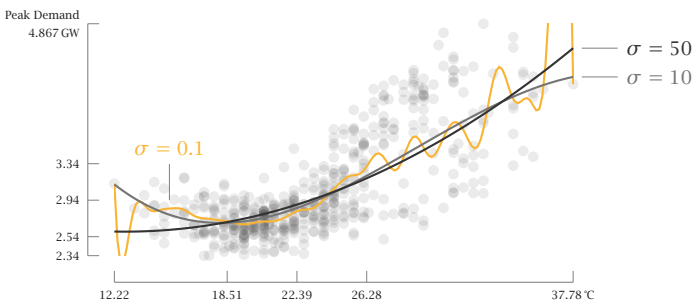


Figure 5.1 The effect of the length-scale kernel parameter σ on the fitted ridge regression model with squared exponential kernel. A length-scale of $\sigma = 0.1$ (—◆—) seem to *overfit* the data; with a length-scale of $\sigma = 50$ (—■—), the model *underfits*. Regularization was set to a low value of $\lambda = 1 \times 10^{-6}$.

Figure 5.1 shows kernel ridge regression with the squared exponential kernel on the peak demand data set for different characteristic length-scales σ. At a small length-scale, local effects and noise have more influence on the regression line. A large length-scale tends to underfitting the data.

Kernel Logistic Regression

Kernelized logistic regression has no analytic solution and iterative methods have to be used. The gradient of the kernelized logistic regression objective

$$J(\boldsymbol{\alpha}) = \frac{1}{m}\sum_{i=1}^{m}\log(1+\exp(-y\cdot\sum_{j=1}^{m}\alpha_j k(\boldsymbol{x}_j,\boldsymbol{x}_i))) \tag{5.7}$$

with respect to $\boldsymbol{\alpha}$ is defined by the partial derivatives

$$\frac{\partial}{\partial\alpha_j}J(\boldsymbol{\alpha}) = \frac{1}{m}\sum_{i=1}^{m}\frac{-y_i\cdot k(\boldsymbol{x}_i,\boldsymbol{x}_j)}{1+\exp(y_i\cdot h_{\boldsymbol{\alpha}}(\boldsymbol{x}_i))}. \tag{5.8}$$

If needed, regularization can be added to the objective and gradient.

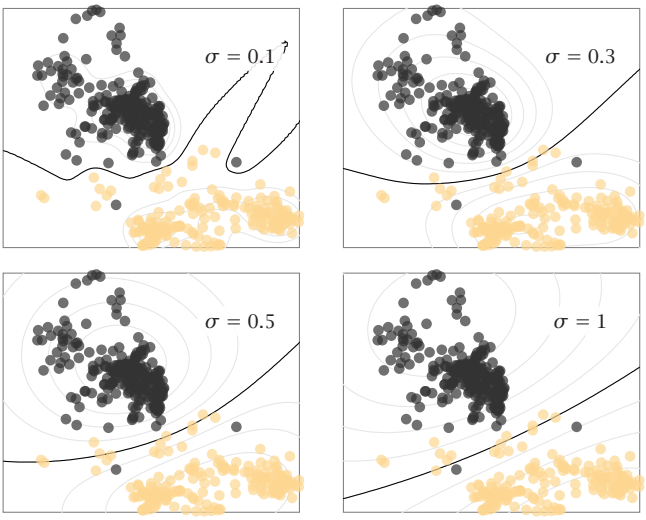


Figure 5.2 Kernelized logistic regression with different length-scales for the squared exponential kernel.

Computational Considerations

It should be clear by now, that kernel functions have two advantages: First, it is possible to use input feature transformations which cannot be expressed explicitly. This happens if the kernel feature map has no analytic form or cannot be represented by today's computer architectures. Second, a kernel $k(\boldsymbol{x},\boldsymbol{z})$ can (intuitively) be seen as a similarity measure between its arguments $\boldsymbol{x}$ and $\boldsymbol{z}$. This can be useful as it is sometimes easier to define how similar two objects are than to design a suitable feature transformation.

One problem arises with the use of kernel functions: There is one α_i for each of the m instances in the training set. Hence, the number of parameters grows with the training set size. As a consequence, we have to keep the

full training set in order to make predictions since the model[1] needs them. This is a disadvantage over the algorithms discussed so far, where the data set can be discarded once $\boldsymbol{w}$ is determined.

[1] $h_{\boldsymbol{\alpha}}(\boldsymbol{x}) = \sum_{i=1}^{m} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x})$

Surprisingly, it can be shown, that for kernelized support vector machines, most of the α_i becomes exactly zero, which allows us to abandon the corresponding training points after the model was fitted.

Perceptrons and Support Vector Machines

We already discussed the (linear) support vector machine in the context of loss functions and regularization. In this chapter, we will have a closer look at its origins as a maximum margin classifier and subsequently derive its kernelized variant.

Most of the linear classifiers we discussed so far, estimated a linear model, that is $h(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$, and then predicted the positive class, if $h(\boldsymbol{x}) \geq 0$. Hence, the hyperplane $\{ \langle \boldsymbol{w}, \boldsymbol{x} \rangle = 0 \}$ divides the input space into two partitions; one for each class. Both, the perceptron and the support vector machine optimization problems are formulated in a way to directly find the plane $\{ \langle \boldsymbol{w}, \boldsymbol{x} \rangle = 0 \}$. However, this is mostly a notational difference.

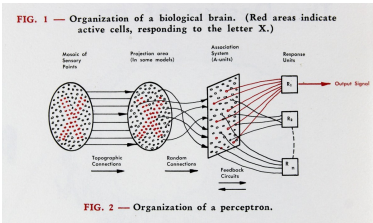


Figure 6.1 An image of the perceptron from Rosenblatt's "The Design of an Intelligent Automaton", Summer 1958. Image: Cornell University Library, Division of Rare and Manuscript Collections.

The Perceptron

The perceptron, one of the first machine learning algorithms, was invented in 1958 at the Cornell Aeronautical Laboratory by Frank Rosenblatt. It assumes that the data is linearly separable, *i.e.*, it is possible to find a hyperplane $\{ \langle \boldsymbol{w}, \boldsymbol{x} \rangle = 0 \}$, which divides the space into two partitions, where one half is "positive" and one half is "negative". The perceptron can therefore be seen as a model that explicitly finds a linear decision boundary.

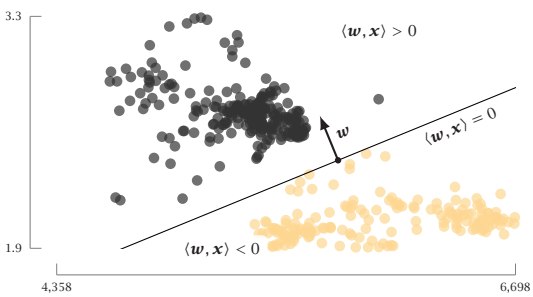


Figure 6.2 The perceptron algorithm tries to find a decision boundary such that $\langle \boldsymbol{w}, \boldsymbol{x} \rangle > 0$ for shale (●) instances and $\langle \boldsymbol{w}, \boldsymbol{x} \rangle < 0$ for sand (●) instances.

The perceptron is equivalent to the linear model we used for linear classification. Geometrically, the set $\{ \langle \boldsymbol{w}, \boldsymbol{x} \rangle = 0 \}$ comprises all points which are perpendicular to the vector $\boldsymbol{w}$, as their inner product with $\boldsymbol{w}$ has to be zero.

The perceptron learning algorithm tries to find a separating hyperplane by minimizing the distance of *misclassified* points to the decision boundary. Using the label space $\mathscr{Y} = \{ -1, +1 \}$ and the fact that $y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \leq 0$ for misclassified instances, the perceptron object can be compactly written as

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} \max\{ 0, -y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \}$$

which is very similar to the hinge loss objective. Notice that only misclassified instances contribute to the sum; the perceptron does not care about the correctly classified data. The gradient of the objective is then given by

$$\nabla J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} \begin{cases} -y_i \boldsymbol{x}_i & \text{if } y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \leq 0, \\ 0 & \text{otherwise.} \end{cases}$$

However, the perceptron learning algorithm does not use full gradient updates, but *stochastic* gradient descent: It chooses an instance $(\boldsymbol{x}_i, y_i)$ at random and calculates the gradient with respect to this single instance as

$$\tilde{\nabla} J(\boldsymbol{w}) = \begin{cases} -y_i \boldsymbol{x}_i & \text{if } y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

making it effectively a stochastic gradient. A stochastic gradient is not an actual gradient, only in expectation. That requires $\mathbf{E}[\tilde{\nabla} J(\boldsymbol{w})] = \nabla J(\boldsymbol{w})$.

The perceptron update rule with stochastic gradient descent is as follows

repeat: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \tilde{\nabla} J(\boldsymbol{w})$

Algorithm 6.1 The gradient descent update rule for the perceptron.

which is repeated until all points are correctly classified, or it loops forever if no separating plane exists. If the classes are linearly separable, it can be shown that the perceptron converges to a separating hyperplane in a finite number of steps.

Proof. If the data set is linearly separable, there must be a unit vector $\boldsymbol{w}^\star$ such that $y_i \langle \boldsymbol{w}^\star, \boldsymbol{x}_i \rangle > 0$ for all points $\{ (\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m) \}$. We define the margin γ as the smallest distance to the plane, that is

$$\gamma = \min_{(\boldsymbol{x}_i, y_i)} y_i \langle \boldsymbol{w}^\star, \boldsymbol{x}_i \rangle.$$

Evidently, $y_i \langle \boldsymbol{w}^\star, \boldsymbol{x}_i \rangle \geq \gamma$ for all instances. Furthermore, let $r = \max_i \|\boldsymbol{x}_i\|$ be the radius of the ball enclosing all data points in the data set.

Starting with some initial parameter vector $\boldsymbol{w}^{(0)}$, we will show, that in every optimization step, the angle between $\boldsymbol{w}^{(t)}$ and $\boldsymbol{w}^\star$ decreases.

$$\begin{aligned} \langle \boldsymbol{w}^\star, \boldsymbol{w}^{(t)} \rangle &= \langle \boldsymbol{w}^\star, \boldsymbol{w}^{(t-1)} + y_i \boldsymbol{x}_i \rangle && \text{(update rule)} \\ &= \langle \boldsymbol{w}^\star, \boldsymbol{w}^{(t-1)} \rangle + y_i \langle \boldsymbol{w}^\star, \boldsymbol{x}_i \rangle && \text{(inner product identities)} \\ &\geq \langle \boldsymbol{w}^\star, \boldsymbol{w}^{(t-1)} \rangle + \gamma && \text{(by definition of } \gamma) \end{aligned}$$

Thus, at every update step, the projection of $\boldsymbol{w}^{(t)}$ onto $\boldsymbol{w}^\star$ increases by at least γ. It follows by induction, that $\langle \boldsymbol{w}^\star, \boldsymbol{w}^{(t)} \rangle \geq t\gamma$.

Next, we need to show that this increase is due to $\boldsymbol{w}^{(t)}$ getting closer to $\boldsymbol{w}^\star$ and not just $\boldsymbol{w}^{(t)}$ getting longer:

$$\begin{aligned} \|\boldsymbol{w}^{(t)}\|^2 &= \|\boldsymbol{w}^{(t-1)} + y_i \boldsymbol{x}_i\|^2 \\ &= \|\boldsymbol{w}^{(t-1)}\|^2 + y_i^2 \|\boldsymbol{x}_i\|^2 + 2 y_i \langle \boldsymbol{w}^\star, \boldsymbol{x}_i \rangle \end{aligned}$$

Since $y_i \in \{-1, +1\}$ and $\|\boldsymbol{x}_i\|^2 \leq r$ we have $y_i^2 \|\boldsymbol{x}_i\|^2 \leq r^2$. As the update is only performed for misclassified points, it holds that $y_i \langle \boldsymbol{w}^{(t-1)}, \boldsymbol{x}_i \rangle \leq 0$. With these two facts we can conclude $\|\boldsymbol{w}^{(t)}\|^2 \leq \|\boldsymbol{w}^{(t-1)}\|^2 + r^2 + 0$, which implies $\|\boldsymbol{w}^{(t)}\|^2 \leq t r^2$ by induction.

The first part of the proof gave us $t\gamma \leq \langle \boldsymbol{w}^\star, \boldsymbol{w}^{(t)} \rangle \leq \|\boldsymbol{w}^\star\| \|\boldsymbol{w}^{(t)}\|$, which implies $t\gamma \leq \|\boldsymbol{w}^{(t)}\|$ since $\|\boldsymbol{w}^\star\|$ is a unit vector. The second part gave us $\|\boldsymbol{w}^{(t)}\| \leq \sqrt{t} r$. Combining both results yields

$$t\gamma \leq \sqrt{t} r \iff t^2 \gamma^2 \leq t r^2 \iff t \leq \frac{r^2}{\gamma^2}$$

This means after r^2/γ^2 steps we cannot make any more progress. □

Although we have a proven convergence guarantee, there are several problems with the perceptron algorithm.

- If the data is not linearly separable, the perceptron will not converge and cycle. These cycles may be hard to detect.
- If the data is linearly separable, there are many solutions[1] which depend on the initialization $\boldsymbol{w}^{(0)}$.
- The convergence is slow when the margin γ is small.

[1] The one found, for example, in Fig. 6.2 is not particularly good, as it is too close to the data.

Although the perceptron initially seemed promising, it was quickly proven that it could *not* be trained to recognize many simple patterns. This caused the field of neural network research to stagnate for many years (known as the "AI winter"), before it was recognized that neural networks with more layers had greater processing power than the perceptron with one layer.

The Support Vector Machine

In contrast to the perceptron, the support vector machine (SVM) does not try to find any separating hyperplane, but the one which separates the data with *large margin*: all instances are not only on the correct side of the separating hyperplane but also far away from it.

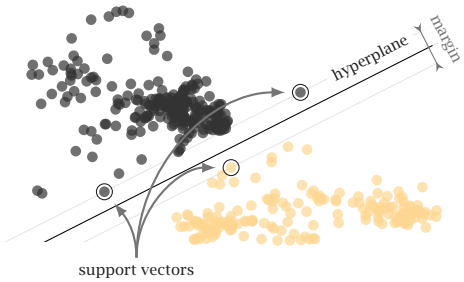


Figure 6.3 The support vector machine solves a *constrained optimization* problem to find this *optimal hyperplane*.

The support vector machine does not include the bias term into the feature vector $\boldsymbol{x}$, but explicitly carry along an offset b in the hyperplane equation as $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b = 0$, which will make some of the following notations easier.

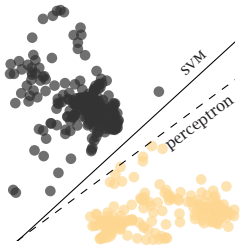


Figure 6.4 Comparison support vector machine and perceptron.

As before, the hyperplane defines the decision boundary: Everything on the side $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b > 0$ belongs class $+1$, and everything on the side $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b < 0$ belongs class -1 with label space $\mathscr{Y} = \{-1, +1\}$. The points closest to the separating hyperplane, *i.e.*, the instances directly on the margin, are known as support vectors.

The derivation of the support vector machine require to prove two things:

1 Maximizing the margin means minimizing $\frac{1}{2}\|\boldsymbol{w}\|^2$.
2 An instance $(\boldsymbol{x}_i, y_i)$ is only correctly classified if $y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b) \geq 1$.

Starting with the latter, notice that the distance of an instance $(\boldsymbol{x}_i, y_i)$ to the separating hyperplane is given by $y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b)$. We define the γ to be the smallest distance to the plane, formally $\gamma = \min_{(\boldsymbol{x}_i, y_i)} y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle$.

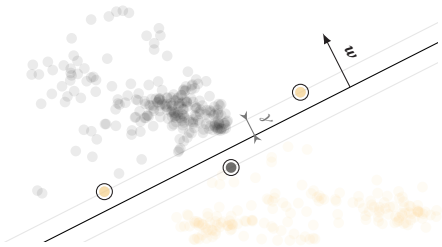


Figure 6.5 The smallest distance to the hyperplane is denoted by γ. If $\boldsymbol{w}$ and b are divided by γ, this distance becomes exactly 1.

The first thing to realize is, that we can scale the parameters $\boldsymbol{w}$ and b arbitrarily, without changing the hyperplane location, as evident from

$$\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b = 0 \iff c\langle \boldsymbol{w}, \boldsymbol{x} \rangle + cb = 0$$

for any $c > 0$. Dividing $\boldsymbol{w}$ and b by γ, the smallest distance (and hence the distance of any support vector) to the plane becomes exactly 1. This then implicitly requires $y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b) \geq 1$ for a correctly classified instance.

The one thing that is left to show is, that maximizing the margin means minimizing $\frac{1}{2}\|\boldsymbol{w}\|^2$. To see why, let u denote the width of the margin. Pick any two support vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ with labels -1 and $+1$, respectively (see Fig. 6.6). Then, the margin u is the vector $(\boldsymbol{x}_j - \boldsymbol{x}_i)$ projected onto a unit vector in the direction of $\boldsymbol{w}$. Such a unit vector can be obtained from $\boldsymbol{w}$ by dividing $\boldsymbol{w}$ though its length $\|\boldsymbol{w}\|$ which yields

$$u = \langle \boldsymbol{x}_j - \boldsymbol{x}_i, \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|} \rangle = \frac{1}{\|\boldsymbol{w}\|}(\langle \boldsymbol{w}, \boldsymbol{x}_j \rangle - \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle) = \frac{2}{\|\boldsymbol{w}\|},$$

where the last equality holds, since $\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b = -1$ and $\langle \boldsymbol{w}, \boldsymbol{x}_j \rangle + b = 1$ by definition as support vectors are exactly on the margin.

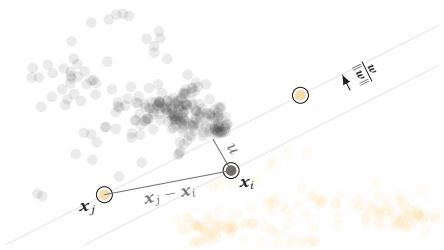


Figure 6.6 Derivation of the margin u with the help of two support vectors.

Instead of maximizing $\frac{2}{\|\boldsymbol{w}\|}$, which is undefined whenever $\|\boldsymbol{w}\|$ is zero, we can equivalently minimize $\frac{1}{2}\|\boldsymbol{w}\|^2$, which behaves nicer. This works, since $\|\boldsymbol{w}\|^2$ is a strictly decreasing[1] function of $\frac{1}{\|\boldsymbol{w}\|}$.

[1] As $\frac{1}{\|\boldsymbol{w}\|}$ gets bigger, $\|\boldsymbol{w}\|^2$ gets smaller.

Putting things together, the support vector machine solves

$$\begin{aligned} \underset{\boldsymbol{w}, b}{\text{minimize}} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 \\ \text{subject to} \quad & y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b) \geq 1 \quad i = 1, \dots, m \end{aligned}$$

which we derived as the constraint optimization problem that yields a separating hyperplane with a largest margin.

Soft Constraints

As before, we can transform this linear SVM classifier into a nonlinear one, by replacing the inputs $\boldsymbol{x}$ thought feature functions $\phi(\boldsymbol{x})$. If we project the inputs into a sufficiently high feature space, it will always be possible for the SVM to find a separating hyperplane for the data. However, this is likely to cause the support vector machine to overfit as, *e.g.* as shown in Fig. 6.7. The overfitting is caused by the constraints

$$y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i) \rangle + b) \geq 1 \qquad i = 1, \dots, m$$

which force the optimizer to find a hyperplane where all training points lie outside the corresponding margin. The optimization problem with these constraints is therefore called the hard margin support vector machine.

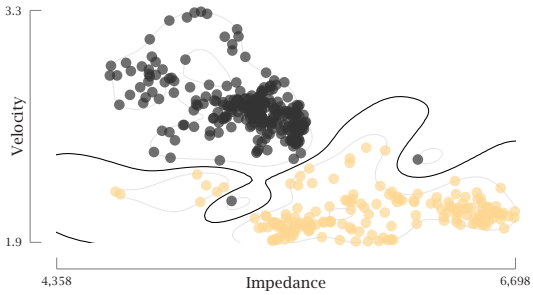


Figure 6.7 The hard margin support vector machine overfitting on the rock type data set since it forces all instances to be on the correct side of the decision boundary.

These hard constraints can be "softened" by introducing so-called slack variables ξ_i which permits individual training instances to be inside (and even on the wrong side) of the margin. This leads to the common soft margin support vector machine optimization problem which is defined as

$$\begin{aligned} \underset{\boldsymbol{w}, b, \boldsymbol{\xi}}{\text{minimize}} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{m}\xi_i \\ \text{subject to} \quad & y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle + b) \geq 1 - \xi_i \quad i = 1, \dots, m, \\ & \xi_i \geq 0 \qquad\qquad i = 1, \dots, m \end{aligned}$$

Whenever a slack variable is "active", *i.e.*, $\xi_i > 0$, the objective is increased by $C\xi_i$, where the constant $C \geq 0$ controls how bad it is to violate a constraint, acting as a regularization parameter. The effects are shown in Fig. 6.8.

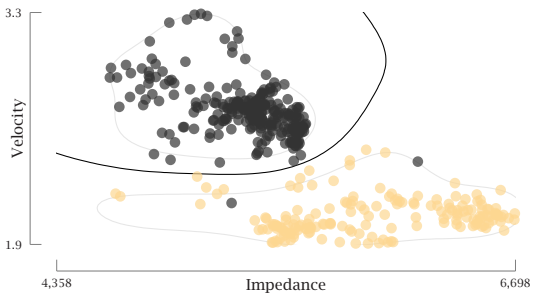


Figure 6.8 The soft margin support vector machine permits training instances to be inside (and even on the wrong side) the margin. This decision boundary looks much better than the one of the hard margin SVM.

An Unconstrained Formulation using Hinge Loss

In the discussion about loss functions, we claimed that the support vector machine is nothing else than a linear classifier which optimizes the hinge loss, namely

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}\max\{\,0, 1 - y_i \cdot \langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle\,\} + \lambda\|\boldsymbol{w}\|^2$$

We now make the connection between this formulation and the soft margin support vector machine. First, we determine the optimal values of the slack

variables ξ_i if $C \neq 0$. The constraints can equivalently be written as

$$\begin{aligned} y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle + b) &\geq 1 - \xi_i \\ \iff \xi_i &\geq 1 - y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle + b). \end{aligned}$$

Since the optimizer will try to minimize ξ_i as much as possible it sets

$$\xi_i = \begin{cases} 0 & \text{if } y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle + b) \geq 1 \\ 1 - y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle + b) & \text{if } y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle + b) < 1 \end{cases}$$

which can be compactly written as $\xi_i = \max\{\, 0, 1 - y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle + b)\,\}$.

Plug these ξ_i back into the objective and scale everything it by $2\lambda/Cm$ to get the hinge loss version of the soft margin SVM.

However, there is a subtle difference: The original support vector machine does not incorporate the bias term b into the parameter vector $\boldsymbol{w}$ and hence, b will not be regularized. This is often a desirable property, but can also achieve by using a constant feature much larger than the usual 1, which then minimizes the side effects of regularizing the bias.

Kernelized Support Vector Machines

From the chapter about kernel functions we know, that in order to kernelize an algorithm, we have to rewrite it in such a way, that it can entirely be expressed in terms of inner products $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z})\rangle$ between the inputs $\boldsymbol{x}$ and $\boldsymbol{z}$. This can be achieved for support vector machines by forming the corresponding *dual* optimization problem; an approach which has some desirable properties.

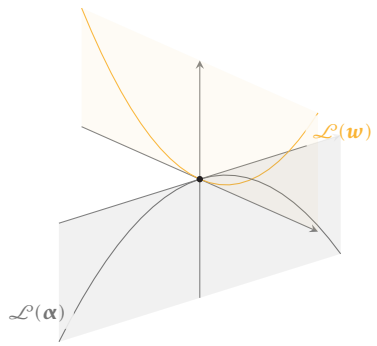


Figure 6.9 In optimization theory, duality is the principle that optimization problems may be viewed from either of two perspectives, the *primal* problem or the *dual* problem. For convex problems, their solutions are equal.

We first have a look at the *primal* constrained optimization problem of the soft margin support vector machine. In order to form the generalized Lagrangian, we have to rewrite all inequality constraints equivalently as

$$\begin{aligned} 1 - y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle + b) - \xi_i \leq 0 \qquad &\text{and} \\ -\xi_i \leq 0. \end{aligned}$$

The Lagrangian $\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\gamma})$ is then equal to

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^m \xi_i + \sum_{i=1}^m \alpha_i[1 - y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle + b) - \xi_i] + \sum_{i=1}^m \gamma_i[-\xi_i]$$

from which we can find the dual by minimizing $\mathcal{L}$ with respect to $\boldsymbol{w}$, b and $\boldsymbol{\xi}$. We will do so, by calculating the corresponding derivatives as

$$\begin{aligned} \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\gamma}) &= \boldsymbol{w} - \sum_{i=1}^m \alpha_i y_i \phi(\boldsymbol{x}_i) \\ \nabla_b \mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\gamma}) &= \sum_{i=1}^m \alpha_i y_i \\ \nabla_{\xi_i} \mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\gamma}) &= C - \alpha_i - \gamma_i. \end{aligned}$$

The optimal values $\boldsymbol{w}^\star$, $b^\star$ and $\xi_i^\star$ can be obtained by setting the respective gradients to zero, which consequently implies

$$\boldsymbol{w}^\star = \sum_{i=1}^{m} \alpha_i y_i \phi(\boldsymbol{x}_i), \quad \sum_{i=1}^{m} \alpha_i y_i = 0 \quad \text{and} \quad \alpha_i \leq C,$$

where the latter inequality holds, since $C = \alpha_i + \gamma_i$ and the fact that the γ_is are Lagrange multipliers which are always nonnegative.

If we take $\boldsymbol{w}^\star$ and plug that back into the Lagrangian, this becomes

$$\mathcal{L}(\boldsymbol{\alpha}) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle$$

after simplification; an expression that now only depends on $\boldsymbol{\alpha}$.

Recall that the dual function always has the constraints that $\alpha_i \geq 0$. Putting this together with the above formulation of the Lagrangian, as well as the constraints we obtained during the derivation, we get

$$\begin{aligned} \underset{\boldsymbol{\alpha} \in \mathbf{R}^m}{\text{maximize}} \quad & \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C \quad i = 1, \dots, m, \\ & \sum_{i=1}^{m} \alpha_i y_i = 0 \end{aligned}$$

as the dual support vector machine optimization problem. Notice, that we reached our initial goal of expressing the SVM entirely in terms of inner products between the inputs, which allows us to utilize any kernel function.

Making Predictions

From the derivation above, we know that the optimal parameter $\boldsymbol{w}^\star$ is

$$\boldsymbol{w}^\star = \sum_{i=1}^{m} \alpha_i y_i \phi(\boldsymbol{x}_i).$$

Interestingly, optimization theory tells us something about $\boldsymbol{\alpha}$, depending on where the training points land with respect to the margin. It holds, that

$$\begin{aligned} & y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i) \rangle + b) \geq 1 \text{ if } \alpha_i = 0 \\ & y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i) \rangle + b) \leq 1 \text{ if } \alpha_i = C \\ & y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i) \rangle + b) = 1 \text{ if } 0 < \alpha_i < C \end{aligned}$$

by the KKT complementary slackness conditions.

Once we solved the dual, we know that all $\boldsymbol{x}_i$ for which $0 < \alpha_i < C$ are support vector. Furthermore, we do not have to keep any point which is correctly classified and outside the margin; these are the ones with $\alpha_i = 0$.

The one thing missing is the value of $b^\star$. For any support vector we have

$$y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i) \rangle + b) = 1 \iff \langle \boldsymbol{w}, \phi(\boldsymbol{x}_i) \rangle + b = y_i,$$

since $y_i \in \{-1, +1\}$. Hence, using a support vector[1] $(\boldsymbol{x}_i, y_i)$ we can derive

$$\begin{aligned} b &= y_i - \langle \boldsymbol{w}, \phi(\boldsymbol{x}_i) \rangle \\ &= y_i - \sum_{j=1}^{m} \alpha_j y_i \langle \phi(\boldsymbol{x}_j), \phi(\boldsymbol{x}_i) \rangle \end{aligned}$$

which we then can use in the support vector machine model

$$\begin{aligned} h(\boldsymbol{x}) &= \langle \boldsymbol{w}, \phi(\boldsymbol{x}) \rangle + b \\ &= \sum_{i=1}^{m} \alpha_i y_i \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}) \rangle + b \end{aligned}$$

to make predictions. In conclusion, both training and prediction can be done using just the results of inner products, and can therefore be kernelized.

[1] For numerical stability, it is best practice to not use a single support vector, but to average over all support vectors.

Nonparametric Algorithms

An algorithm is called nonparameteric if the number of parameters grows with the number of points in the training set m — an admittedly poorly chosen name, as this is the opposite of having no parameters. Often, such algorithms are "lazy" in the sense that they just store the training set in the learning/optimization step and do all the work at prediction time. A typical example are nearest neighbor algorithms which just look up the training instances which are closest to the query point. This chapter reviews some of these nonparameteric algorithms and finally anticipates decision trees.

Locally Weighted Linear Regression

The idea of locally weighted (linear) regression (LWR) is, that linear models might be not very good at modeling nonlinear behavior on a *global* scale, but *locally*. That means, if we want to make a prediction for an input $\hat{\boldsymbol{x}}$, we use only the data in the neighborhood of $\hat{\boldsymbol{x}}$ to fit a linear model (Fig. 7.1).

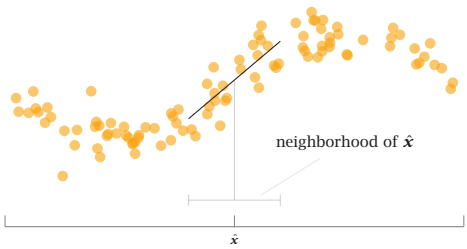


Figure 7.1 Only the points in the neighborhood of $\hat{\boldsymbol{x}}$ are used to fit a linear model.

Locally weighted regression is slightly more sophisticated: it does not use a fixed neighborhood, but weights each training instance such that points close to $\hat{\boldsymbol{x}}$ get a higher weight than points far away (indicated by the size of the points in Fig. 7.2). Intuitively, if the weight for a particular point $\boldsymbol{x}_i$ is large, the optimizer tries hard to make $\ell(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle, y_i)$ small. On the other hand, if its weight is small, the induced loss can be pretty much ignored.

The squared exponential kernel is a popular choice for a weighting function:

$$k(\boldsymbol{x}, \hat{\boldsymbol{x}}) = \exp\Big(-\frac{\|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2}{2\sigma^2}\Big)$$

The length-scale parameter σ allows an easy control on how quickly the weight of a training set instance falls off with its distance to the query $\hat{\boldsymbol{x}}$.

Overall, locally weighted regression solves the following optimization problem

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} k(\boldsymbol{x}_i, \hat{\boldsymbol{x}}) \cdot \ell(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle, y_i) \tag{7.1}$$

for *each* query input $\hat{\boldsymbol{x}}$. Note that each predicted point is associated to a single regression curve. So the overall model is obtained by combining the locally weighted linear regression models at each point as shown in Fig. 7.2.

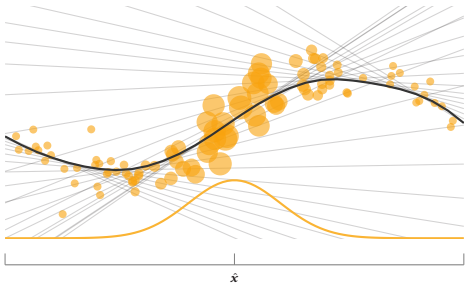


Figure 7.2 Locally weighted regression fit to the data. The weighting function for a single query $\hat{\boldsymbol{x}}$ is shown at the bottom. The size of each training point indicates its weight according to the weighting function.

Nearest Neighbors

The k-nearest neighbors (k-NN) algorithm can be used for classification and regression. It assumes that similar inputs have similar labels and therefore retrieves the k points closest to $\hat{\boldsymbol{x}}$ from the data to predict the label for $\hat{\boldsymbol{x}}$.

Nearest Neighbors Regression

In the regression task, the prediction for a query point $\hat{\boldsymbol{x}}$ is the average label of its k nearest neighbors in the data set, which is formally defined as

$$h(\hat{\boldsymbol{x}}) = \frac{1}{k} \sum_{i=1}^{m} y_i \delta_{\text{NN}}(\boldsymbol{x}_i, \hat{\boldsymbol{x}}), \tag{7.2}$$

where $\delta_{\text{NN}}(\boldsymbol{x}_i, \hat{\boldsymbol{x}})$ is a function which is one, if $\hat{\boldsymbol{x}}$ is a k-NN of $\boldsymbol{x}_i$ and zero otherwise. We see, there is no learning or optimization step involved. k-NN simply stores the training set and retrieves the nearest neighbors of a query.

The parameter k controls the model complexity: Low values lead to overfitting; high values to underfitting as illustrated in Fig. 7.3. Use cross validation to find a satisfactory value for k.

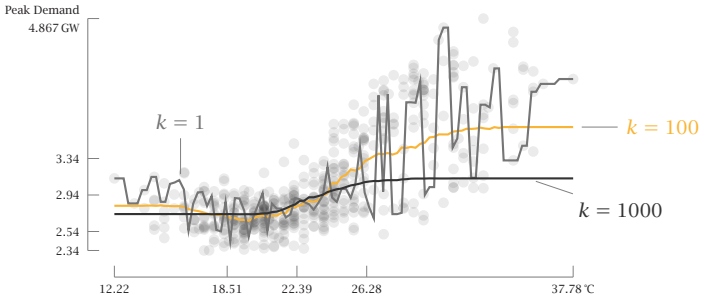


Figure 7.3 The k-nearest neighbors regression algorithm for different values of k. A low value of $k = 1$ (—●—) *overfits* the data; a high value of $k = 1000$ (—■—) *underfits*. A middle ground like $k = 100$ (—◆—) seems to perform well.

Nearest Neighbors Classification

In k-nearest neighbor classification, an instance is classified by a plurality vote of its neighbors, with the query point being assigned to the class which is most common among its k nearest neighbors. For $k = 1$, the query point is simply assigned to the class of that single nearest neighbor.

For the label space $\mathcal{Y} = \{-1, +1\}$, the k-NN classification algorithm can be expressed in the same way as in the regression setting, namely

$$h(\boldsymbol{x}) = \frac{1}{k} \sum_{i=1}^{m} y_i \delta_{\text{NN}}(\boldsymbol{x}_i, \boldsymbol{x}).$$

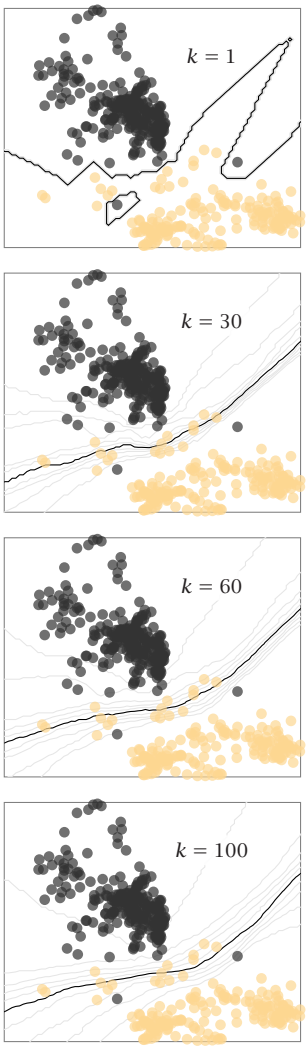


Figure 7.4 The k-nearest neighbors classification algorithm for different values of k.

The Distance Metric

The k-nearest neighbor algorithm fundamentally relies on a good distance metric. The better the metric reflects label similarity, the better the predictions. A popular metric choice is the Minkowski distance defined as

$$d(\boldsymbol{x}, \boldsymbol{z}) = \Big(\sum_{i=1}^{n} |x_i - z_i|^p \Big)^{\frac{1}{p}}.$$

The Minkowski distance can be considered as a generalization of the Manhattan distance ($p = 1$), the Euclidean distance ($p = 2$) and the Chebyshev distance ($p = \infty$).

Connection to Support Vector Machines

There is an interesting connection between kernelized support vector machines and the k-nearest neighbor algorithm. Compare the k-NN model with the one of the kernelized SVM which had been defined as

$$h(\boldsymbol{x}) = \sum_{i=1}^{m} \frac{1}{k} y_i \delta_{\text{NN}}(\boldsymbol{x}_i, \boldsymbol{x}) \qquad \text{and}$$
$$h(\boldsymbol{x}) = \sum_{i=1}^{m} \alpha_i y_i k(\boldsymbol{x}_i, \boldsymbol{x}) + b,$$

respectively. Both are very similar, but in contrast to k-NN, the SVM does not limit its decision to the k-nearest neighbors of $\boldsymbol{x}$. The SVM considers

all training points, but puts more weight to those that are closer to $\boldsymbol{x}$, *i.e.*, those which have larger kernel values $k(\boldsymbol{x}_i, \boldsymbol{x})$ — a similar idea as in LWR.

Using a kernel that decays with distance, such as the squared exponential, the SVM can be seen as a soft nearest neighbor algorithm as the exponential decay will assign almost no weight to all but the neighboring points of $\boldsymbol{x}$. In some surprising sense, linear classification in infinite dimensional spaces can be seen as nearest neighbor classification in the original space.

Moreover, the SVM is smart enough to discard useless training points by setting many α_i to zero and delegating their "voting weight" to other training points. This makes the SVM faster and more memory efficient than k-NN.

The Curse of Dimensionality

The curse of dimensionality — a term due to Richard Bellman — refers to the fact that problems can get a lot harder to solve in high-dimensional spaces. For example, the k-NN classifier makes the assumption that close points share similar labels. However, it can happen that in very high dimensional spaces, all points are equally far way, *i.e.*, there are no *nearest* neighbors.

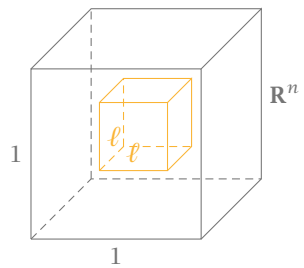


Figure 7.5 An n-dimensional hypercube with edge length of 1. We want to choose the inner box such that 1% of the observations are enclosed by it.

To get an intuition for the curse of dimensionality, suppose the large unit cube in Fig. 7.5 is the data universe with a uniform distribution of data points across the whole volume. The task is to find the smallest inner cube with edge length ℓ, such that 1% of the observations are enclosed inside it.

Let n be the dimension of the data. Then the observations lie in the $[0, 1]^n$ hypercube. We assume $m = 1000$ points are sampled *uniformly* at random within this cube. Then, we would like to capture 1%, ($k = 10$ points), that is

$$\ell^n \approx \frac{k}{m} \iff \ell \approx \left(\frac{k}{m}\right)^{\frac{1}{n}} \to 1 \text{ for } n \to \infty$$

which tells us that, the inner cube needs to be the same size as the outer box if the data dimension n gets large. The interior of the cube is basically empty and virtually all points (at least 99%) are located at the cube's edge.

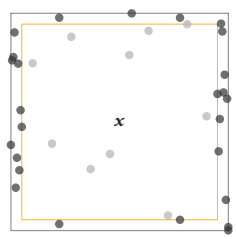


Figure 7.6 The k-NN of $\boldsymbol{x}$ must be at the edge of the inner box, or else we could make the box smaller.

The k-NN of $\boldsymbol{x}$ must be at the edge of the inner box, or else we could make the box smaller. The points which are *not* k-NN of $\boldsymbol{x}$ are between the two boxes, right next to the nearest neighbor points. It follows, that all points must have roughly the same distance. This violates the assumption of k-NN.

Intrinsic Dimensionality

When the curse of dimensionality affects the k-NN algorithm so much, why does it work on natural images with thousands of dimensions? Because natural images are *not* uniformly distributed. The true dimensionality of the data is much lower than its ambient space: It requires much fewer features to describe a face (*e.g.* sex, eyes, hair, ...) than number of pixels used for its picture. k-NN can be applied if the data has low intrinsic dimensionality.

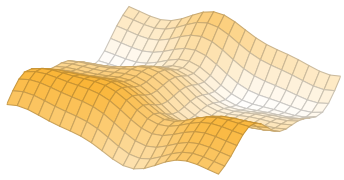


Figure 7.7 Sketch of a low dimensional manifold in $\mathbf{R}^n$.

k-dimensional Trees

Obviously, we would like to have a large data set to achieve a good accuracy when using k-nearest neighbor techniques. However, k-NN is a very slow algorithm as it has to find the nearest neighbors of a query point, which takes $\mathbf{O}(nm)$ time, when considering a training set of m points in n dimensions.

The general idea of k-d trees (invented in the 1970s by Jon Bentley) is to partition the data set into regions, such that we only have to search for the nearest neighbor of a point in the same region and not the whole data set.

A k-d tree is build by iteratively splitting the data in half along one input dimension (feature) using its median (Fig. 7.8). The input dimension can be rotated, but in practice, using the one with highest variance works the best.

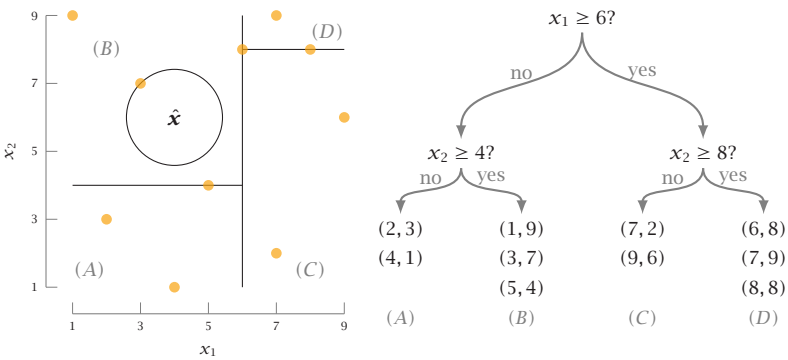


Figure 7.8 An example of a tree dividing the input space at feature x_1 and then in each of the resulting partitions at x_2.

To find the nearest neighbor of a query point, for example, $\hat{\boldsymbol{x}} = (4, 6)$ we just walk down the tree to find its partition; then search for its nearest neighbor within this partition. In Fig. 7.8, the nearest neighbor of $\hat{\boldsymbol{x}} = (4, 6)$ is $(3, 7)$ with a Euclidean distance of $\sqrt{2}$. If no separating plane is within this distance,[1] we are done, as none of the data points inside the other partitions can be closer to $\hat{\boldsymbol{x}}$. Otherwise, we have to review the partitions which are too close using *depth-first* tree search as, *e.g.*, shown in Fig. 7.9.

[1] We have to check if any separating plane intersects the circle with radius $\sqrt{2}$ and center $(3, 7)$.

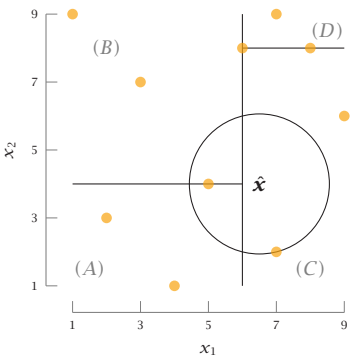


Figure 7.9 The query point $\hat{\boldsymbol{x}} = (6.5, 4)$ is in partition (C) with its nearest neighbor at a Euclidean distance of approximately 2.06, intersecting two planes. We therefore have to go up one node in the k-d tree and check the condition "$x_2 \geq 8$?". No intersection with this plane as $4 + 2.06 < 8$. We go again one node up and check the condition "$x_1 \geq 6$?". Since $6.5 - 2.06 < 6$, we are intersecting the plane and have to go down the "no"-branch. Next, check the node with "$x_2 \geq 4$?". Since $4 + 2.06 \geq 4$ and $4 - 2.06 < 4$, we have to audit both partitions (A) and (B) to find the true nearest neighbor $(5, 4)$ in (B).

When searching for not only one, but k-nearest neighbors, we have to check if there is any separating plane within the distance of the kth closest point.

Notice, that a k-d tree is a binary tree which is always balanced since we split each time at the median. Finding the correct leaf node of the tree therefore takes $\mathbf{O}(\log m)$ time. However, in the worst case the circle is intersection every plane, and we gain nothing as we have to search each partition of the tree, eventually comparing the whole data set to the query point. This happens more often, if the input dimension is large as it is then more likely, that a point is at one or more boundaries of its hypercube. Hence, the curse of dimensionality makes k-d trees ineffective in high dimensional spaces. Even worse: In contrast to vanilla nearest neighbors, it is not advantageous, if the data has a low intrinsic dimensionality as the splits are all axis aligned.

If it is not necessary to get the exact nearest neighbors of a query point, we can derive an approximate nearest neighbor algorithm by simply never checking other partitions, even if they potentially contain a closer neighbor.

Decision Trees

Tree-based methods come in flavors of regression and classification algorithms. They partition the input space into distinct sets and fit a simple model, typically a *piecewise constant*, to each partition. This chapter explores how trees fit into our optimization based view of machine learning.

Introduction

Before formalizing decision trees, we introduce the ideas on a simple toy example: The people of medieval Romania had several vampire accidents. They therefore decided to create a simple guide in form of a decision tree to allow everybody to identify vampires. They recorded the following data.[1]

[1] The data set is inspired by MIT 6.034 lectured by Patrick H. Winston.

Has Shadow	Likes Garlic	Accent	Age	Vampire
yes	no	heavy	17	no
yes	yes	falsene	23	no
?	yes	odd	45	no
yes	no	heavy	55	no
?	no	falsene	58	yes
?	yes	falsene	68	no
?	no	odd	201	yes
no	no	heavy	423	yes

Before developing a guide in form of a decision tree, we establish some notation. Without the guide, the a priori probability of a stranger being a human or a vampire is $\mathbf{P}(\text{human}) = 5/8$ and $\mathbf{P}(\text{vampire}) = 3/8$ respectively.

We can evaluate how bad this a priori prediction is, using a classification loss, *e.g.*, the cross entropy loss which is defined as

$$\ell(q_{\hat{Y}}, p_Y) = H(\mathbf{P}_Y, \mathbf{Q}_{\hat{Y}}) = -\sum_{y \in \mathscr{Y}} p_Y(y) \log q_{\hat{Y}}(y)$$

where $q_{\hat{Y}}$ is the predicted and p_Y the true discrete probability distribution.

In this example, let the first entry in $\hat{\boldsymbol{y}}$ and $\boldsymbol{y}$ correspond to $\mathbf{P}(\text{human})$ and the second entry to $\mathbf{P}(\text{vampire})$. With this convention, $\hat{\boldsymbol{y}}$ and $\boldsymbol{y}$ correspond to the discrete probability distributions $q_{\hat{Y}}$ and p_Y, respectively.

How good is the a priori prediction of $\hat{\boldsymbol{y}} = (5/8, 3/8)$ for all entries in the table with respect to the loss? The true class probabilities are $\boldsymbol{y} = (1, 0)$ for human entries and $\boldsymbol{y} = (0, 1)$ for vampire entries which yield losses of

$$\ell\big((5/8, 3/8), (1, 0)\big) = -1 \cdot \log 5/8 - 0 \cdot \log 3/8 \approx 0.47 \qquad \text{and}$$
$$\ell\big((5/8, 3/8), (0, 1)\big) = -0 \cdot \log 5/8 - 1 \cdot \log 3/8 \approx 0.98.$$

The objective value for this constant a priori prediction is therefore

$$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} \ell(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i) = \frac{1}{m}\sum_{i=1}^{m} \ell\big((5/8, 3/8), \boldsymbol{y}_i\big) \approx 0.66$$

which will serve as a baseline (the best we can do without questing or testing the stranger). The goal now is to improve this baseline with a decision tree.

What properties about the stranger should we query in order to improve our predictions? Of course, we would like to ask as few questions as possible, as not to rise suspicion. If we could only ask one question, what question would that be? We want to find a question that is *most useful* for vampire detection, where usefulness is measured in terms of overall loss reduction.

We can, for example, ask if the stranger *likes garlic*. Based on the answer, the data set is split into two partitions, which we name S_1 and S_2. S_1 comprises all past recordings of people that like garlic and S_2 of those who do not.

Has Shadow	Likes Garlic	Accent	Age	Vampire	Partition
?	yes	falsene	68	no	
yes	yes	falsene	23	no	S_1
?	yes	odd	45	no	
?	no	falsene	58	yes	
no	no	heavy	423	yes	
?	no	odd	201	yes	S_2
yes	no	heavy	17	no	
yes	no	heavy	55	no	

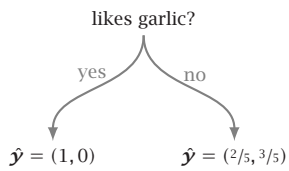


Figure 8.1 The garlic test.

If the stranger likes garlic, we look at S_1. According to the records, no vampire likes garlic. The prediction is therefore $\hat{\boldsymbol{y}} = (1, 0)$. If the answers is "no", we look at partition S_2. The records show that $2/5$ of the humans in this part of the country do not like garlic either. The best prediction, with respect to the cross entropy loss, is therefore $\hat{\boldsymbol{y}} = (2/5, 3/5)$ (see Fig. 8.1).

Although asking only one question, the prediction based on the *garlic* test is doing much better than the previous a priori prediction in terms of $J(\boldsymbol{w})$:

$$J(\boldsymbol{w}) = \frac{1}{m}\sum_{\boldsymbol{x}_i \in S_1} \ell\big((1, 0), \boldsymbol{y}_i\big) + \frac{1}{m}\sum_{\boldsymbol{x}_i \in S_2} \ell\big((2/5, 3/5), \boldsymbol{y}_i\big) \approx 0.42$$

The first term sums up all losses in the first partition (likes garlic), and the second term sums up all losses of the second partition (doesn't like garlic).

Alternatively to the garlic test, we could also see if the stranger casts a *shadow*. However, the records are missing some values and there are therefore three categories/partitions S_1, S_2 and S_3 for "yes", "no" and "?", respectively. This test yields an even lower objective of approximately 0.35.

Has Shadow	Likes Garlic	Accent	Age	Vampire	Partition
yes	yes	falsene	23	no	
yes	no	heavy	17	no	S_1
yes	no	heavy	55	no	
no	no	heavy	423	yes	S_2
?	yes	falsene	68	no	
?	no	falsene	58	yes	S_3
?	no	odd	201	yes	
?	yes	odd	45	no	

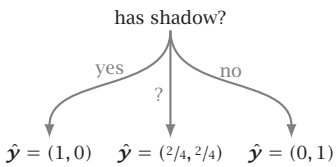


Figure 8.2 The shadow test.

It is also possible to split on numeric properties like *age*. For this, we evaluate all possible thresholds τ for *age* $(17, 23, \ldots, 423)$ and choose the one which yields the lowest objective. Here, a test on "age $\geq$ 58" yields an objective of 0.28 and the following partitions.

Has Shadow	Likes Garlic	Accent	Age	Vampire	Partition
?	yes	falsene	68	no	
?	no	falsene	58	yes	S_1
no	no	heavy	423	yes	
?	no	odd	201	yes	
yes	yes	falsene	23	no	
yes	no	heavy	17	no	S_2
yes	no	heavy	55	no	
?	yes	odd	45	no	

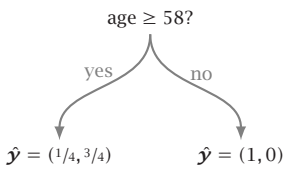


Figure 8.3 The age test.

It turns out, that the *accent* does not help much in identifying vampires, and we therefore discard this question entirely.

None of our "guides" so far is perfect, but we can improve them with further tests. Unfortunately, tracking all possible combinations of tests is infeasible, as there exponentially many. Hence, we apply a greedy approach which continuous with the best partitions so far and divides these further.

The best partitions so far came from the last decision tree using the *age* test. We need to check both partitions and see if an additional test brings an improvement: Partition S_2 already has a zero loss and there is nothing

to do. However, partition S_1 may be improved with a further test. Probably we should ask if the stranger *likes garlic* to create an additional partition.

Has Shadow	Likes Garlic	Accent	Age	Vampire	Partition
?	yes	falsene	68	no	S_1
?	no	falsene	58	yes	
no	no	heavy	423	yes	S_3
?	no	odd	201	yes	
yes	yes	falsene	23	no	
yes	no	heavy	17	no	S_2
yes	no	heavy	55	no	
?	yes	odd	45	no	

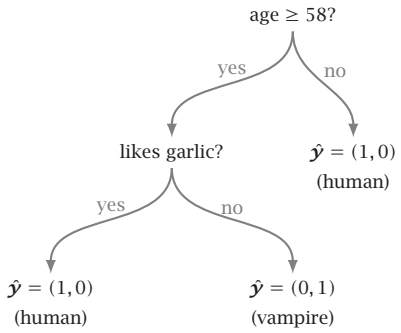


Figure 8.4 The age and garlic test.

We remark, that the same feature can be used multiple times. An alternative to the *garlic* test would thus be first "age ≥ 68" and then "age < 201".

Formalizing Decisions Trees

The decision tree algorithm is no different from other machine learning techniques. It requires a loss function (regression or classification) and an approach to minimize the objective with some regularization.

To simplify matters, we restrict our attention to binary trees. That means each node of the tree splits the data set on some feature $\boldsymbol{x}_j$ with a certain threshold τ into exactly *two* partitions. Then, one or both of these splits are sliced again into two more partitions. This process is continued, until some stopping criteria is met, and we end up with P partitions $\{ S_1, \dots, S_P \}$.

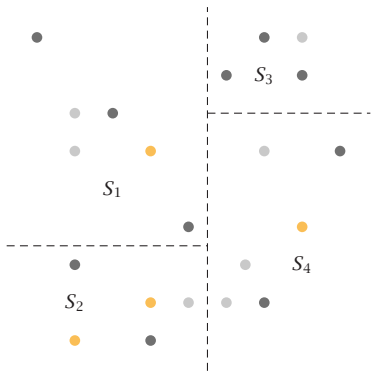
A decision tree can only make a constant prediction in each leave. We denote these predictions with $\{ \hat{y}_1, \dots, \hat{y}_P \}$. The decision tree model is then

$$h(\boldsymbol{x}) = \sum_{p=1}^{P} \begin{cases} \hat{y}_p & \text{if } \boldsymbol{x} \in S_p, \\ 0 & \text{otherwise} \end{cases}$$

essentially predicting $\hat{y}_p$ if $\boldsymbol{x}$ falls into partition S_p.

What the best prediction $\hat{y}_p$ in a partition S_p is, depends on the selected loss function. If we choose the cross entropy loss, as in the vampire detection task before, the best possible prediction is the proportion of each class label in that partition (as we did). On the other hand, if we would decide on the square loss, then the optimal prediction in partition S_p would be

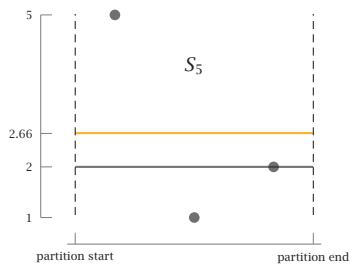
$$\hat{y}_p = \frac{1}{|S_p|} \sum_{\boldsymbol{x}_i \in S_p} y_i$$



Classification Trees: The best prediction with respect to the *cross entropy loss* the proportion of each class in that partition. For example $\hat{y}_1 = (3/6\,\bullet, 1/6\,\bullet, 2/6\,\bullet)$ for partition S_1.

namely, the (empirical) *mean* of all labels in that partition. In general, the optimal prediction $\hat{y}_p$ for any given loss ℓ can be obtained by solving

$$\hat{y}_p = \underset{\hat{y}_p \in \mathcal{Y}}{\arg\min} \sum_{\boldsymbol{x}_i \in S_p} \ell(\hat{y}_p, y_i). \tag{8.1}$$



Regression Trees: The best prediction for this partition (S_5) with three training instances is: the median ($\hat{y}_5 = 2$) if the *absolute loss* is used; the mean ($\hat{y}_5 = 2.66$) if the *squared loss* was chosen.

Since we now know how to derive the optimal predictions $\hat{y}_p$ in each split, we only have to find an optimal partition $\{ S_1, \dots, S_P \}$ of the input space. As said, finding the best binary partition is generally computationally infeasible. Instead, we have to resort to greedy methods which incrementally build the tree, minimizing the objective function one decision tree node at a time.

Starting without a split on the full data set S, the decision tree corresponds to a constant function. Then, we systematically search through all feature dimensions j and all thresholds τ, splitting S into

$$S^+ = \{ \boldsymbol{x} \in S : x_j \geq \tau \} \quad \text{and} \quad S^- = \{ \boldsymbol{x} \in S : x_j < \tau \}$$

and evaluate the objective on a tree with this new additional partition. Finally, we pick the best pair (j, τ) for which the cost was the lowest. If no split produced a sufficiently lower objective, the algorithm terminates. Otherwise, the sets S^+ and S^- are scanned for new splits in the same way.

Notice, that by splitting a set into two new partitions, the objective *never* increases. Hence, (assuming no training instances are contradictory) decision trees can always obtain zero training error if they are just deep enough, that is, with a sufficiently high number of partitions. In the extreme case, every training point ends up in its own partition. Therefore, decision trees are also *universal function approximators*, and hence prone to overfitting.

Cross validation can be used to determine the right amount of regularization, in terms of the tree depth or preferably the number of tree nodes.

Notice, that $\boldsymbol{w}$ is not a vector anymore, but the set of all the parameters that define a tree, such as the splits (j, τ) and the partition predictions $\hat{y}_p$.

Case Studies

The following section implements regression and classification trees for the peak demand and rock type data sets, respectively.

Regression Trees

Regression trees are no different from classification trees. We split on some features and, depending on in which partition the input lands, we make a prediction. This prediction will be constant within the partition.

If the square loss is used, the best prediction $\hat{y}_p$ within partition S_p is the *mean* of all labels of points in that partition. If the absolute loss is used instead, *e.g.*, to make the model more robust against outliers, the best prediction would be the *median* of all labels.

The tree depth or the total number of nodes (splits) can be used to regulate the model complexity. Figure 8.5 shows regression trees with a different number of nodes on the peak demand dataset.

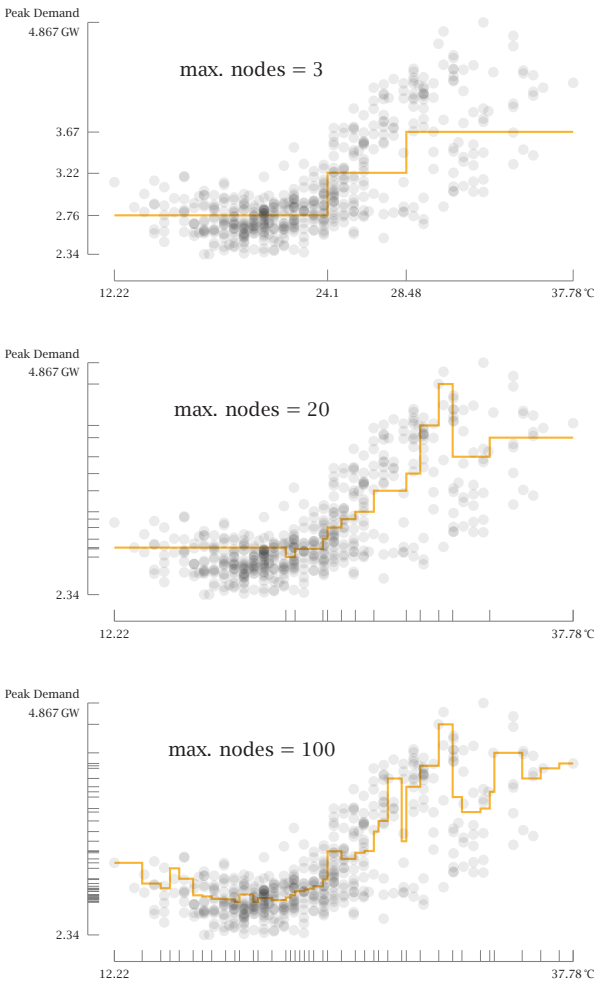


Figure 8.5 Regression trees with square loss on the peak demand dataset. Increasing the total number of nodes (splits) of the tree increases the model complexity. In the plot, the axis ticks mark the split points.

Since decision trees are universal function approximators, they overfit if grown too deep (see Fig. 8.6), that is, we have too many partitions. Cross validation helps to choose a sufficient depth (or sufficient number of splits) on a validation set. However, this number is now a discrete integer and not a continuous regularization parameter like λ as before and does therefore

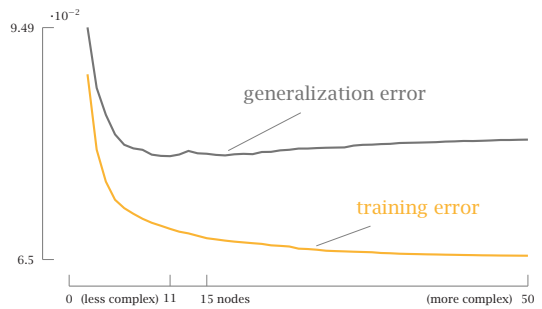


Figure 8.6 Regression tree mean squared error on the peak demand data. Since there are no extreme values in the predictions of the regression trees, the generalization error rises moderately if the number of nodes (splits) is increased.

not allow for a fine grain tuning of the model complexity. In this example, we would select the number of splits to be somewhere between 10 and 15.

Classification Trees

For classification, we only have to decide on a different loss function. The greedy algorithm for partitioning (splitting the nodes in the tree) stays identical to regression. Typical choices are the zero-one or logistic loss (binary classification) or the cross entropy loss for multiclass classification.

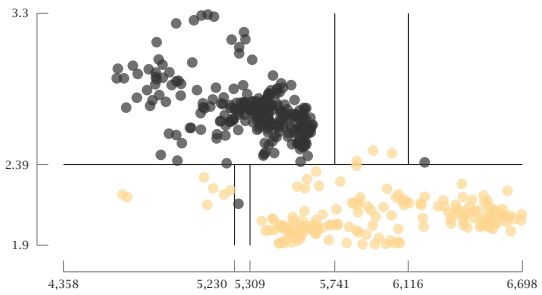


Figure 8.7 A binary classification tree with 100% accuracy on the rock type data set.

Ensemble Methods

In this chapter we will combine models to an ensemble such that the ensemble has (on average) a better performance than the individual models. An *ensemble* is created by *averaging* (in case of regression) or *voting* (in case of classification) of base learners. We will study two popular ensemble methods: *bagging* which reduces the variance of complex models, and *boosting* which reduces bias to promote weak learners to strong learners.

Bagging

Bagging, short for bootstrap aggregating, was introduced by Breiman (1996) to reduce the prediction error of learning algorithms with high variance. Recall that the bias-variance decomposition comprises the following terms:

$$\begin{aligned}\mathbf{E}[(Y - h(X))^2] &= \mathbf{E}[\mathbf{E}[h(X)] - h^\star(X)]^2 && (\text{bias}^2)\\ &+ \mathbf{E}[(h(X) - \mathbf{E}[h(X)])^2] && (\text{variance})\\ &+ \mathbf{V}[\epsilon] && (\text{noise}).\end{aligned}$$

The goal is now to reduce the contribution by variance. High variance algorithms are those, whose predicted output undergoes major changes in response to small changes in the training data. This is typical for models that overfit, as these try to hit some training instances exactly.

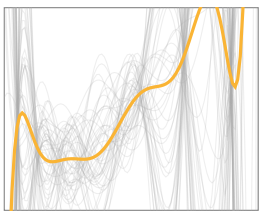


Figure 9.1 Samples of $h(X)$ ( ) and their expectation $\mathbf{E}[h(X)]$ ( ).

To reduce variance, it is necessary that the individual (training data set dependent) models $h(X)$ become closer to the "average" model $\mathbf{E}[h(X)]$ we would get after many fits to different training sets from $\mathbf{P}_{XY}$ (see Fig. 9.1).

This endeavor would be trivial if we would have access to T different data sets $D_1, \ldots, D_T$ sampled from $\mathbf{P}_{XY}$. We could then train one model $h_1, \ldots, h_T$ (referred to as base learner) on each of these data sets and create an ensemble by averaging their predictions, formally

$$H(\boldsymbol{x}) = \frac{1}{T}\sum_{t=1}^{T} h_t(\boldsymbol{x}). \tag{9.1}$$

The law of large numbers guarantees that the ensemble gets closer to the expectation with high probability as more samples being used, that is

$$H(X) = \frac{1}{T}\sum_{t=1}^{T} h_t(X) \to \mathbf{E}[h(X)] \quad \text{as} \quad T \to \infty$$

and, we would have reached the initial goal of reducing the model variance.

The only problem is, that we do not have access to T different data sets sampled from $\mathbf{P}_{XY}$, but only one. A solution is to draw bootstrap copies from the original data set and train a model on each copy. Each bootstrap copy is a set of the same size m as the original training set, where the instances are samples with replacement. This is bagging.

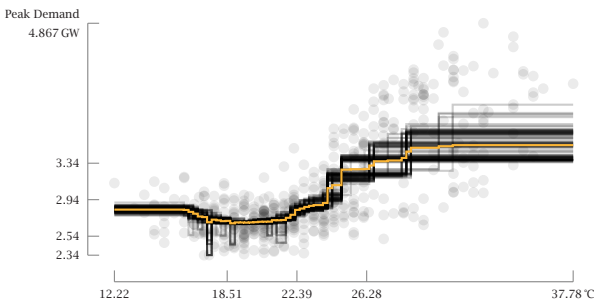


Figure 9.2 The effect of bagging 100 regression tree base learner (——) to an ensemble (——).

Why do we have to use sampling *with replacement*? Intuitively, we need to draw samples from the unknown distribution $\mathbf{P}_{XY}$, but have only the set $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)\}$ available. However, since this is a sample representation of $\mathbf{P}_{XY}$, we hope it is representative for random draws from $\mathbf{P}_{XY}$. If the drawn samples would be removed from the set, it would change the distribution. It is therefore important to sample *with* replacement.

If m samples are drawn with replacement from a data set of size m, then what is the probability for any particular data point ending up in the bootstrap copy? It is easier to turn this question around and determine the probability that an instance is *not* selected as

$$(1 - \tfrac{1}{m})^m \quad \text{with} \quad \lim_{m\to\infty}(1 - \tfrac{1}{m})^m = \tfrac{1}{\mathrm{e}} \approx 0.368$$

and therefore, the probability that a sample is chosen from a large data set, is very roughly $2/3$. This is another way to explain the variance reduction: Outliers and noise are likely to occur just in a few bootstrap copies and therefore have only a very limited influence on the overall ensemble.

In summary, bagging reduces variance by model averaging and thus prevents overfitting. Although bagging is usually applied to decision trees, it can *and should* be used with any high variance machine learning algorithm.

Moreover, since the ensemble is an average of models, it can provide besides the prediction *mean* also a *variance*. This variance which can be interpreted as the uncertainty in prediction. Especially in regression tasks, such uncertainties are otherwise hard to obtain.

Random Forest

The random forest algorithm is essentially bagged decision trees, with a slightly modified splitting criteria: Draw a bootstrap copy and split until no more improvement can be made (we overfit). However, do not search through all feature dimensions for a split. Instead, randomly subsample $k \leq n$ features (without replacement) and only consider these for splitting.

Random forests are very popular and work extremely well out of the box: The algorithm has only two parameters, T and k and is extremely *insensitive* to both. Common choices are $k = \sqrt{n}$ and T as large as we can afford. Feature scaling can also be omitted since decision trees do not care about numerical ranges. There is basically nothing to do but throwing data at it.

Out of Bag Error

For bagged models, there is no need for cross validation or a separate validation set to get an unbiased estimate of the generalization error. Since each model is constructed using a different bootstrap sample from the original data, about one-third of the cases are left out of the bootstrap copy and are not used in the construction of a particular model. Each data point, that was not involved in the training of a model, can now be used as an estimate for the generalization error: This way, a prediction on a "validation set" is obtained in about one-third of the models.

Out of Bag Error Algorithm:

1. For each instance $\boldsymbol{x}_i$ in the training set, find all models that are *not* trained on this particular instance.
2. Make a prediction for $\boldsymbol{x}_i$ with only these models (for example by averaging or majority vote) and compare it to the actual label y_i.
3. Compile a metric over all instances.

Bias-Variance Tradeoff for Bagging

Bagging uses a model average $H(\boldsymbol{x}) = \frac{1}{T}\sum_{t=1}^{T} h_t(\boldsymbol{x})$ to drive the ensemble closer to the model expectation $\mathbf{E}[h(X)]$ in order to reduce the variance. Remarkably, this has no effect on the bias term (which normal increases if the variance is decreased). As a result, the generalization error steadily decreases with the ensemble size as shown in Fig. 9.3 and does not rise again at a certain point.

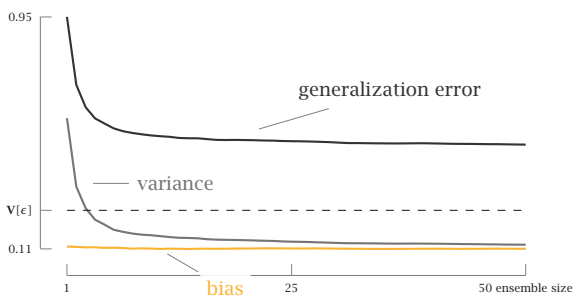


Figure 9.3 Bias-variance decomposition of bagged decision trees on a synthetic data set. Even though the variance decreases with a larger ensemble, the bias does not increase.

Boosting

Boosting is one of the most powerful ideas in machine learning. A boosting ensemble drastically improves the accuracy of weak learners, turning them into strong learning algorithms. Intuitively, one can think of a weak learner as an algorithm that performs just slightly better than a random guess. A typical example of a weak learner is a decision tree stump, *i.e.*, a decision tree with exactly one node that creates two partitions. However, any classification or regression algorithm can be used as a weak learner.

The Idea of Boosting

The idea of boosting is easiest to explain in the least squares regression setting. Suppose the hypothesis class of weak learners $\mathscr{H}$ consists of the aforementioned decision tree stumps. The weak learner $h_1 \in \mathscr{H}$ fitted to the peak demand data set looks as follows.

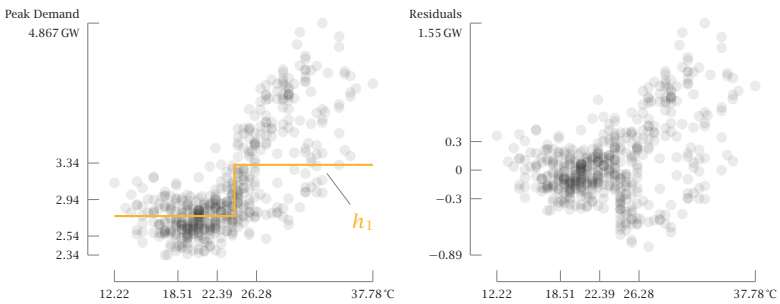


Figure 9.4 A decision tree stump h_1 fitted to the peak demand data set (left) and the residuals $y - h_1(\boldsymbol{x})$ (right).

There is not only noise, but a lot of structure in the residuals which might be captured with an additional model. We fit a second model h_2 to the residuals, trying to correct the errors the first model made by forming an ensemble $H(\boldsymbol{x}) = h_1(\boldsymbol{x}) + h_2(\boldsymbol{x})$.

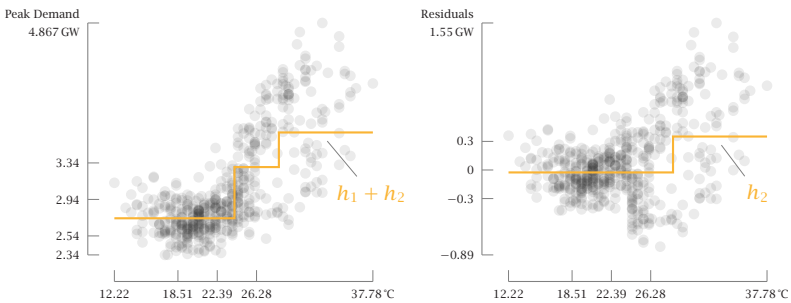


Figure 9.5 A second model h_2 fitted to the residuals (right) and their ensemble $H = h_1 + h_2$ (left).

This procedure can be continued for T iterations until the residuals look like standard normal distributed noise as illustrated in Fig. 9.6.

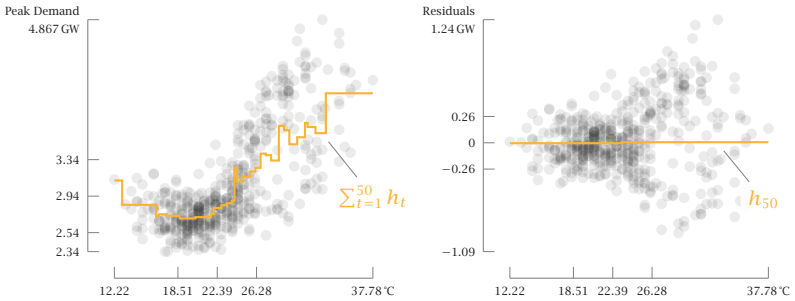


Figure 9.6 The ensemble of T weak learners (left) and the remaining residuals fitted model h_t (right).

That is the big picture of boosting: Combine weak learners $h_1, \dots, h_T$ to a strong learner $H(\boldsymbol{x}) = \sum_{t=1}^{T} h_t(\boldsymbol{x})$. A close look at the update rule for the ensemble, reveals that a surprising similarity to gradient descent with learning rate $\eta = 1$ and gradients $-h$:

repeat: $H \leftarrow H + \eta h$

The difference is, that we are not updating parameters $\boldsymbol{w}$, but *functions*. What we derived in this example, was only a special case of a general family of boosting algorithms, that all perform gradient descent in *functional space*.

Gradient Descent in Functional Space

A boosting ensemble H will combine T base learners $h_1, \dots, h_T$ to

$$H(\boldsymbol{x}) = \sum_{t=1}^{T} \eta_t h_t(\boldsymbol{x}),$$

where the $\eta_t \in \mathbf{R}$ can be interpreted as *confidence weights*, which determines how much we "trust" the prediction of the individual base learners. Boosting builds this ensemble in an iterative fashion, where, in iteration t, the model h_t is trained and added to the ensemble.

The process of constructing such an ensemble in a stage-wise fashion is very similar to gradient descent. However, instead of updating model parameters, functions are added to the ensemble. Thus, boosting performs *gradient descent in functional space*.

The goal is, as before, to minimize an objective J, but this time we try to directly find a model (ensemble) H that minimizes J for some loss function:

$$\underset{H}{\text{minimize}} \quad J(H) = \frac{1}{m} \sum_{i=1}^{m} \ell(H(\boldsymbol{x}_i), y_i),$$

Suppose we already have some ensemble H and wish to find the next model $h \in \mathscr{H}$, to add to H, such that the objective $J(H + \eta h)$ decreases. If we look at this from an optimization point of view, then h should be in the direction

of the negative gradient of the objective with respect to H, namely we should choose $h = -\nabla_H J$. This insight follows from the first order Taylor expansion which permits the approximation $J(H + \eta h) \approx J(H) + \langle \nabla J_H(H), h \rangle$.

There is only one problem: The new model h must be chosen from from $\mathscr{H}$ and it very unlikely that the wanted function $-\nabla_H J$ is in this set. Instead, we can try to find some $h \in \mathscr{H}$ which is as close as possible to $-\nabla_H J$:

$$\underset{h \in \mathscr{H}}{\text{minimize}} \quad \|h - (-\nabla_H J)\|^2$$

which is sometimes regerred to as the *gradient projection* step since we are projecting $-\nabla_H J$ onto $\mathscr{H}$. The algorithm will make progress as long as h and $-\nabla_H J$ do not point in different directions (or are orthogonal), that is we always want $\langle h, \nabla_H J \rangle < 0$.

After a satisfactory base learner h was found, the gradient descent step $H \leftarrow H + \eta h$ can be applied to yield a new ensemble for the next iteration.

The steps described so far build the generic boosting framework called AnyBoost. To popular manifestations of boosting are *least squares boosting* for regression and *AdaBoost* for classification; both are discussed in the following.

```
H ← 0
for t ← 1, ..., T do
    h ← minimize_{h∈ℋ}  ‖h + ∇_H J‖²
    if ⟨h, ∇_H J⟩ ≥ 0 then
        break
    H ← H + ηh
return ensemble H
```

Algorithm 9.1 Generic Boosting (AnyBoost).

Remark 9.1
Finding the next base learner h requires the inner products[1] between functions. The inner product of two Lebesgue integrable functions f and g is defined as

$$\langle f, g \rangle = \int f(\boldsymbol{x}) g(\boldsymbol{x}) \, \mathrm{d}\mathbf{P}_X(\boldsymbol{x}) = \mathbf{E}[fg].$$

If one does not have access to the distribution $\mathbf{P}_X$ (as in our scenario), the intergral can be estimates as

$$\langle f, g \rangle \approx \frac{1}{m} \sum_{i=1}^{m} f(\boldsymbol{x}_i) g(\boldsymbol{x}_i)$$

from samples $\boldsymbol{x}_1, \dots, \boldsymbol{x}_m$, again justified by the law of large numbers.

Whenever we need to calculate $\langle h, \nabla_H J \rangle$, this sample estimate tells us, that we only need to evaluate h and $\nabla_H J$ at $\boldsymbol{x}_1, \dots, \boldsymbol{x}_m$ to get an approximation.

[1] Recall that $\|\boldsymbol{v}\|^2 = \langle \boldsymbol{v}, \boldsymbol{v} \rangle$.

Least Squares Boosting

The introductory example on the peak demand data set was a special case of the AnyBoost algorithm with square loss[2] $\ell(H(\boldsymbol{x}), y) = {}^1\!/_2 (H(\boldsymbol{x}) - y)^2$ known as least squares boosting. The partial derivatives of J with respect to H evaluated at $\{ \boldsymbol{x}_1, \dots, \boldsymbol{x}_m \}$ can be derived as

$$\nabla_H J = (H(\boldsymbol{x}_1) - y_1, \dots, H(\boldsymbol{x}_m) - y_m)$$

[2] The loss is scaled by the constant factor ½ to make the result more readable.

The projection of the negative gradient onto $\mathscr{H}$ then yields

$$\begin{aligned}\underset{h\in\mathscr{H}}{\text{minimize}} \quad \|h-(-\nabla_H J)\|^2 &= \tfrac{1}{m}\sum_{i=1}^{m}(h(\boldsymbol{x}_i)-(y_i-H(\boldsymbol{x}_i)))^2\\ &= \tfrac{1}{m}\sum_{i=1}^{m}(h(\boldsymbol{x}_i)-r_i)^2\end{aligned}$$

which implies that the next base learner h should be the one that minimizes the square loss with respect to the residuals $r_i = y_i - H(\boldsymbol{x}_i)$ of the previous ensemble. This is exactly what we did in the introductory example.

AdaBoost

There are two reasons why AdaBoost, is one of the most popular algorithms for binary classification: it analytically finds the optimal step size η and is surprisingly robust against overfitting. AdaBoost adopts the *exponential loss* $\ell(H(\boldsymbol{x}), y) = \exp(-yH(\boldsymbol{x}))$ and thus minimizes the objective

$$\underset{H}{\text{minimize}} \quad J(H) = \tfrac{1}{m}\sum_{i=1}^{m}\exp(-y_i H(\boldsymbol{x}_i))$$

Although, the individual base learners for AdaBoost are restricted to have a binary output, namely $h(\boldsymbol{x}) \in \{-1, +1\}$, the ensemble predicts a signed confidence score for each class in $\mathscr{Y} = \{-1, +1\}$.

The first thing to show is that the gradient projection step implies that AdaBoost fits in each iteration a base learner that minimizes an objective with *weighted* zero-one loss, namely

$$J(h) = \sum_{i=1}^{m} w_i \cdot \ell_{0/1}(h(\boldsymbol{x}_i), y_i)$$

with weights $w_i = \frac{1}{Z}\exp(-y_i H(\boldsymbol{x}_i))$ and $Z = \sum_{j=1}^{m} w_j$. If we define F to be the set of all wrongly classified instances, *i.e.*, $F = \{(\boldsymbol{x}_i, y_i) : h(\boldsymbol{x}_i) \neq y_i\}$ and $\epsilon = \sum_{(\boldsymbol{x}_i, y_i)\notin F} w_i$, this objective can be written compactly as $J(h) = \epsilon$.

We will see, that this objective encourages the next model h to get instances eventually correctly classified that where wrongly discriminated by the previous ensemble H.

```
H ← 0 (the constant 0 function)
w_i ← 1/m
J(h) = Σ_{i=1}^m w_i ℓ_{0/1}(h(x_i), y_i)
for t ← 1, ..., T do
    h ← minimize_{h∈H} J(h)
    ε ← J(h)
    if ε ≥ 1/2 then
        break
    η ← 1/2 ln (1−ε)/ε
    w_i ← w_i exp(−y_i η h(x_i))
    w_i ← w_i / Σ_{i=1}^m w_i
    H ← H + ηh
return ensemble H
```

Algorithm 9.2 AdaBoost

Proof. We show that gradient projection implies that the next base learner h minimizes $J(h) = \sum_{(\boldsymbol{x}_i, y_i)\in F} w_i = \epsilon$. Notice that, since the weights sum up to one, it holds that $\sum_{(\boldsymbol{x}_i, y_i)\notin F} w_i = 1 - \sum_{(\boldsymbol{x}_i, y_i)\in F} w_i$.

Since the gradient of J with respect to H evaluated at $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_m\}$ is

$$\nabla_H J = (-y_i \exp(-y_1 H(\boldsymbol{x}_1)), \dots, -y_i \exp(-y_m H(\boldsymbol{x}_m))),$$

the projection of the negative gradient onto $\mathscr{H}$ yields

$$\begin{aligned}
\underset{h\in\mathscr{H}}{\text{minimize}} \quad & \|h - (-\nabla_H J)\|^2 \\
&\equiv \underset{h\in\mathscr{H}}{\text{minimize}} \quad \sum_{i=1}^{m} (h(\boldsymbol{x}_i) - y_i w_i)^2 \\
&= \underset{h\in\mathscr{H}}{\text{minimize}} \quad \sum_{i=1}^{m} (h(\boldsymbol{x}_i)^2 - y_i w_i h(\boldsymbol{x}_i) + y_i^2 w_i^2) \\
&\equiv \underset{h\in\mathscr{H}}{\text{minimize}} \quad \sum_{i=1}^{m} -y_i w_i h(\boldsymbol{x}_i) \\
&= \underset{h\in\mathscr{H}}{\text{minimize}} \quad \sum_{(\boldsymbol{x}_i,y_i)\in F} w_i - \sum_{(\boldsymbol{x}_i,y_i)\notin F} w_i = \epsilon - (1-\epsilon) \\
&\equiv \underset{h\in\mathscr{H}}{\text{minimize}} \quad \sum_{(\boldsymbol{x}_i,y_i)\in F} w_i = \epsilon
\end{aligned}$$

where we used the fact, that $h(\boldsymbol{x}) \in \{-1, +1\}$ several times. □

Now, one great property of AdaBoost is, that we can find the optimal learning rate η for each gradient descent step in *closed form* to be

$$\eta = \frac{1}{2} \ln \frac{1-\epsilon}{\epsilon}. \tag{9.2}$$

Proof. A necessary condition for $\eta^\star$ to be optimal is that the gradient of $J(H + \eta h)$ with respect to η is zero at $\eta^\star$:

$$\begin{aligned}
\frac{\partial}{\partial\eta} J(H + \eta^\star h) &= \sum_{i=1}^{m} y_i h(\boldsymbol{x}_i) \exp(-y_i H(\boldsymbol{x}_i) - \eta^\star y_i h(\boldsymbol{x}_i)) = 0 \\
&\iff \sum_{i=1}^{m} w_i y_i h(\boldsymbol{x}_i) \exp(-\eta^\star y_i h(\boldsymbol{x}_i)) = 0 \\
&\iff \mathrm{e}^{\eta^\star} \sum_{(\boldsymbol{x}_i,y_i)\in F} w_i = \mathrm{e}^{-\eta^\star} \sum_{(\boldsymbol{x}_i,y_i)\notin F} w_i \\
&\iff \mathrm{e}^{\eta^\star} \epsilon = \mathrm{e}^{-\eta^\star} (1-\epsilon) \\
&\iff \mathrm{e}^{2\eta^\star} = \frac{1-\epsilon}{\epsilon} \\
&\iff \eta^\star = \frac{1}{2} \ln \frac{1-\epsilon}{\epsilon}
\end{aligned}$$

□

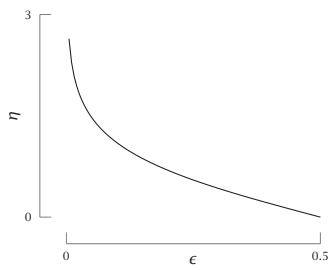


Figure 9.7 The learning rate η depends on the error ϵ of the base learner. Models with a lower training error will have more weight in the ensemble.

Notice, that the optimal step size η depends on the performance of the base learner h, which is hidden in the definition of $\epsilon = \sum_{(\boldsymbol{x}_i,y_i)\notin F} w_i$ with F beeing the set of all instances that are wrongly classified by h. As a result, models with a lower training error — and therefore their predictions — will have more weight in the ensemble.

Also notice, that the update step $H \leftarrow H + \eta h$ causes the weights for the next iteration to be

$$\begin{aligned}
w_i^{(t+1)} &= \exp(-y_i(H(\boldsymbol{x}_i) + \eta h(\boldsymbol{x}_i))) \\
&= w_i^{(t)} \exp(-y_i \eta h(\boldsymbol{x}_i)) \\
&= \begin{cases} w_i^{(t)} \mathrm{e}^{\eta} & \text{if } h(\boldsymbol{x}_i) \neq y_i \\ w_i^{(t)} \mathrm{e}^{-\eta} & \text{if } h(\boldsymbol{x}_i) = y_i \end{cases} \\
&= \begin{cases} w_i^{(t)} \sqrt{\frac{1-\epsilon}{\epsilon}} & \text{if } h(\boldsymbol{x}_i) \neq y_i \\ w_i^{(t)} \sqrt{\frac{\epsilon}{1-\epsilon}} & \text{if } h(\boldsymbol{x}_i) = y_i \end{cases}
\end{aligned}$$

Since h is a weak learner, its accuracy is by definition larger than $1/2$ which implies that ϵ is always less than $1/2$. Therefore, in each iteration, the weights of correctly classified instances are *decreased* and the weights of misclassified examples are *increased*. If a training instance is consecutively misclassified, its weight becomes so large that it will eventually be classified correctly.

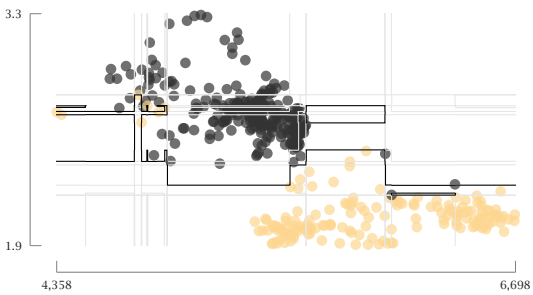


Figure 9.8 AdaBoost with zero training error on an extended version of the rock types data set. This might not be a good decision boundary in terms of generalization, but illustrates that AdaBoost can fit very complex data.

Another consequence of the closed form solution of the optimal step size is, that AdaBoost converges extremely quick. If the weak learner returns a model with error $\epsilon \leq 1/2 - \gamma$, then, after T iterations

$$J(H) \leq (1 - 4\gamma^2)^{\frac{T}{2}}$$

which means that the training error is decreasing exponentially fast.

Proof. We need to bound

$$J(H) = \frac{1}{m}\sum_{i=1}^{m} \exp(-y_i H(\boldsymbol{x}_i)) = \frac{1}{m}\sum_{i=1}^{m} w_i.$$

In each iteration, the weights get updates by

$$\begin{aligned} w_i^{(t+1)} &= w_i^{(t)} \exp(-y_i \eta h(\boldsymbol{x}_i)) \\ \sum_{i=1}^{m} w_i^{(t+1)} &= \sum_{i=1}^{m} w_i^{(t)} \exp(-y_i \eta h(\boldsymbol{x}_i)) \\ &= \mathrm{e}^{\eta} \sum_{(\boldsymbol{x}_i, y_i) \in F} w_i + \mathrm{e}^{-\eta} \sum_{(\boldsymbol{x}_i, y_i) \notin F} w_i \\ &= \mathrm{e}^{\eta}\epsilon + \mathrm{e}^{-\eta}(1-\epsilon) \\ &= 2\sqrt{\epsilon(1-\epsilon)} \end{aligned}$$

substituting the optimal step size $\eta = \frac{1}{2}\ln\frac{1-\epsilon}{\epsilon}$ in the last step.

Starting with $\sum_{i=1}^{m} w_i^{(0)} = m$, it follows that, after T iterations

$$\begin{aligned} J(H) &= \frac{1}{m}\sum_{i=1}^{m} \exp(-y_i H(\boldsymbol{x}_i)) = \frac{1}{m}\sum_{i=1}^{m} w_i^{(T)} \\ &= \frac{1}{m}\sum_{i=1}^{m} w_i^{(0)} \prod_{t=1}^{T} 2\sqrt{\epsilon(1-\epsilon)} \\ &\leq (2\sqrt{1/4 - \gamma^2})^T \\ &= (1 - 4\gamma^2)^{\frac{T}{2}} \end{aligned}$$

with proves the claim. □

Since the exponential loss upper bounds the zero-one loss, the misclassification rate decreases with at least the same speed (Fig. 9.10). Consequently,

if $J(H) < 1/m$ there must be no training errors (if the data set does not contain any contradictions). This happens after $\mathbf{O}(\ln m)$ iterations:

$$
\begin{aligned}
& J(H) \leq (1-4\gamma^2)^{\frac{T}{2}} < \tfrac{1}{m} \\
& \iff (1-4\gamma^2)^{-\frac{T}{2}} > m \\
& \iff \tfrac{T}{2} \ln\left(\frac{1}{1-4\gamma^2}\right) > \ln m \\
& \iff T > \frac{2 \ln m}{\ln\left(\frac{1}{1-4\gamma^2}\right)}
\end{aligned}
$$

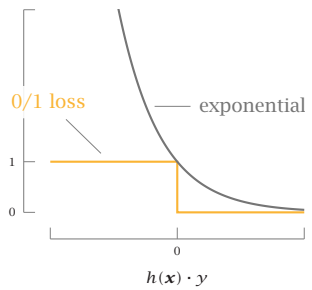


Figure 9.9 The exponential loss upper bounds the zero-one loss. Consequently, if the latter decreases with the former.

Notice, that the objective never reaches zero and always declines, pushing the training instances further from the decision boundary. This makes AdaBoost, just like the SVM, a large margin classifier. So, contrary to our intuition, there is no danger to overfit if we keep training after reaching zero training error. It is quite the opposite. By pushing points further from the decision boundary, the generalization error gets improved.

In practice AdaBoost is quite robust to overfitting. As the number of weak learners increases, the bias decreases exponentially fast, but the variance increases by geometrically diminishing magnitudes. However, AdaBoost is sensitive to the labeling noise as can be seen from Fig. 9.8.

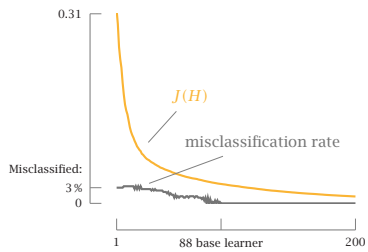


Figure 9.10 Since the exponential loss upper bounds the zero-one loss, the misclassification rate decreases with at least the same speed.

Bias-Variance Tradeoff for Boosting

Boosting directly combats high bias, leaving the error due to variance essentially unchanged. Hence, as we increase the model complexity (the number of weak learners T) the generalization error does not go up, but steadily decreases as shown in Fig. 9.11.

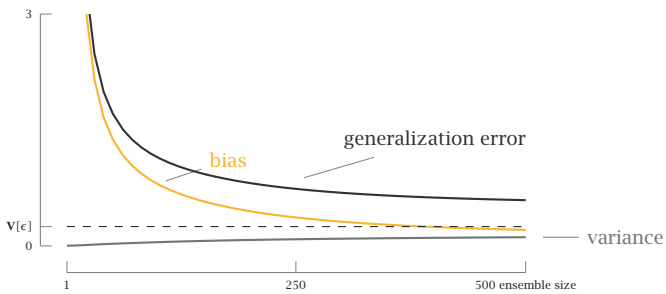


Figure 9.11 Bias-variance decomposition of boosted decision trees on a synthetic data set. Even though the bias decreases with a larger ensemble, the variance does not increase significantly.

Neural Networks and Deep Learning

We have seen, that simple linear models become very powerful, when the inputs are projected into a well-chosen feature space using

$$h(\boldsymbol{x}) = \langle \boldsymbol{w}, \phi(\boldsymbol{x}) \rangle + b,$$

where ϕ is the feature function, performing the mapping that creates the (typically nonlinear) features. Common examples for ϕ are polynomial features, bag-of-words, HOC, SIFT; to name a few. However, the performance of these algorithms crucially depends on the choice of *good* features, which is often the most difficult task when modeling a machine learning problem.

Can we *automatically* learn a feature function ϕ for a given task/domain?

Feedforward Neural Networks

The neural network was originally known under the name multi-layer perceptron (MLP), and is essentially a feature function of the form

$$\phi(\boldsymbol{x}) = \sigma(U\boldsymbol{x} + \boldsymbol{c}) \tag{10.1}$$

with a matrix U and vector $\boldsymbol{c}$ as additional parameters that can be learned / optimized. Here, $\sigma(\boldsymbol{z})$ is a *nonlinear* activation function, which operates element-wise on its input vector $\boldsymbol{z}$. Without the activation function, any linear model that uses such a feature map would stay linear as evident from

$$h(\boldsymbol{x}) = \boldsymbol{w}^\top \phi(\boldsymbol{x}) = \boldsymbol{w}^\top (U\boldsymbol{x} + \boldsymbol{c}) = (\boldsymbol{w}^\top U)\boldsymbol{x} + \boldsymbol{w}^\top c = W\boldsymbol{x} + b$$

setting $W = \boldsymbol{w}^\top U$ and $b = \boldsymbol{w}^\top \boldsymbol{c}$. Hence, the apparent introduced complexity is not providing any additional representational power if σ is omitted.

Some common choices for activation functions are listed the following.

sigmoid activation: $\boldsymbol{z} \mapsto 1/(1 + e^{-\boldsymbol{z}})$
The sigmoid or logistic activation function squashes its input between zero

and one. It has a continuous differentiable gradient. However, for very large or very small inputs, the gradient essentially vanishes.

tanh activation: $\boldsymbol{z} \mapsto (e^{2\boldsymbol{z}} - 1)/(e^{2\boldsymbol{z}} + 1)$
The hyperbolic tangent activation function is essentially like the sigmoid, sharing its advantages and disadvantages, however it is symmetric around zero and hence keeps negative inputs negative.

ReLU activation: $\boldsymbol{z} \mapsto \max\{0, \boldsymbol{z}\}$
The rectified linear unit (ReLU) is one of the most widely used activation functions for neural networks. It performs a threshold operation to each input element where negative values are set to zero, eliminating the vanishing gradient problem observed by sigmoid or tanh activation functions. In addition, it is extremely fast to compute. Due to its popularity, there are various extensions and modifications of the ReLU.

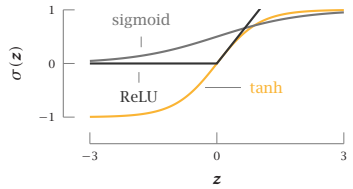


Figure 10.1 Common activation functions for neural networks.

To analyze how neural network function, consider again the peak demand task using a linear model with rectified linear unit activation function, *i.e.*,

$$\begin{aligned} h(\boldsymbol{x}) &= \boldsymbol{w}^\top \phi(\boldsymbol{x}) + b = \boldsymbol{w}^\top \max\{0, U\boldsymbol{x} + \boldsymbol{c}\} + b \\ &= \sum_j w_j \max\{0, \langle \boldsymbol{u}_j, \boldsymbol{x}\rangle + c_j\} + b \end{aligned}$$

where $\boldsymbol{u}_j^\top$ is the jth row of the matrix U.[1] From the last equation, it becomes evident that the neural network model h is a piecewise linear approximation to the data: Each $\langle \boldsymbol{u}_j, \boldsymbol{x}\rangle + c_j$ is a line (or hyperplane in high dimensions), which becomes "active", whenever its positive (Fig. 10.2). Notice that, downward slopes (below zero) are possible by using a negative value for w_j.

[1] The matrix U has the following form.

$$U = \begin{bmatrix} - \boldsymbol{u}_1^\top - \\ - \boldsymbol{u}_2^\top - \\ \vdots \end{bmatrix}$$

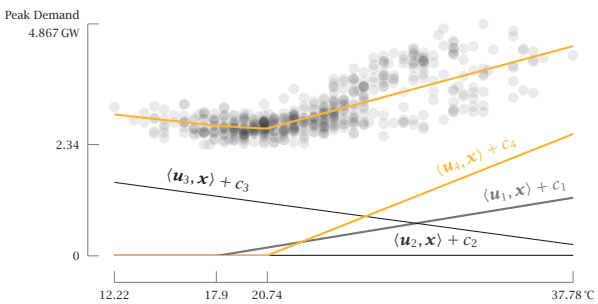


Figure 10.2 The learned model consists of the piecewise linear components $\langle \boldsymbol{u}_j, \boldsymbol{x}\rangle + c_j$ (bottom). In this example, $U = (0.78, 0.2, -0.65, 1.9)$ and $\boldsymbol{c} = (0.43, -0.49, 0.89, 0.64)$. The piecewise linear approximation to the data (top) is obtained by combining them with the parameters $\boldsymbol{w} = (0.29, 0.2, 0.8, 0.83)$ and offset $b = 1.72$.

It is not surprising, that with sufficiently many piecewise linear components, we are able to approximate *any* continuous function arbitrary well. A neural network is therefore also a *universal function approximator*. However, in practice, this property is not that relevant (similar to how polynomials can fit any function), and the more important aspect is that neural networks appear to work very well for many different domains.

A deep neural network is a simple extension of this idea, where the learnable feature functions are stacked or layered. This looks as follows.

$$\begin{aligned} h(\boldsymbol{x}) &= \boldsymbol{w}^\top \phi(\phi'(\phi''(\boldsymbol{x}))) + b \\ &= \boldsymbol{w}^\top \sigma(U_3 \sigma(U_2 \sigma(U_1 \boldsymbol{x} + \boldsymbol{c}_1) + \boldsymbol{c}_2) + \boldsymbol{c}_3) + b \\ &= U_4 \sigma(U_3 \sigma(U_2 \sigma(U_1 \boldsymbol{x} + \boldsymbol{c}_1) + \boldsymbol{c}_2) + \boldsymbol{c}_3) + c_4 \end{aligned}$$

setting $U_4 = \boldsymbol{w}^\mathsf{T}$ and $c_4 = b$ in the last equation for a consistent notation.

Algorithmically, the forward pass of a k-layer network can be expressed as

$$\begin{aligned} \boldsymbol{z}_1 &= \boldsymbol{x} \\ \boldsymbol{z}_{i+1} &= \sigma_i(U_i \boldsymbol{z}_i + \boldsymbol{c}_i), \quad i = 1, \dots, k-1 \\ h(\boldsymbol{x}) &= U_k \boldsymbol{z}_k + \boldsymbol{c}_k \end{aligned}$$

where we, from now on, set $U_k = \boldsymbol{w}^\mathsf{T}$ and $c_k = b$, such that we can reuse $\boldsymbol{w}$ to denote the set of *all* parameters which have to be optimized. These are

$$\boldsymbol{w} = \{ U_1, \dots, U_k, \boldsymbol{c}_1, \dots, \boldsymbol{c}_k \}$$

in this concrete example. Note, there might be more parameters, depending on the network architecture and the activation functions. Architectural choices (how many layers, type of activation functions, *e. t. c.*), define the hypothesis class $\mathscr{H}$ and are not part of the learnable parameter set $\boldsymbol{w}$.

It is common to regularize the parameters $\boldsymbol{w}$. The term weight decay (in the neural network context) referrers thereby to standard ℓ_2 regularization.

Neural Network Optimization

The nonlinear activation functions make both, the neural network and its objective nonconvex functions of the parameters. Usually, the objective has numerous local minima (depicted in Fig. 10.3) and we must resort to *nonconvex optimization* methods.

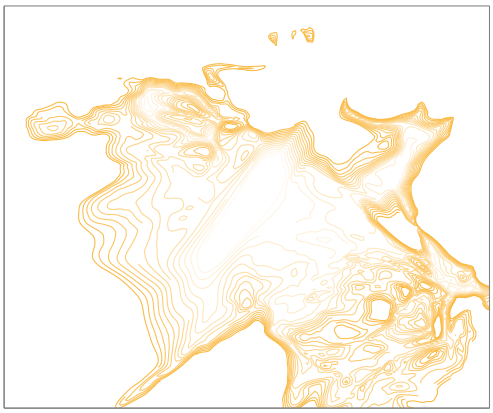


Figure 10.3 Loss landscape of a deep neural network (resnet56) with many narrow minima and one wide minimum in the middle.

One possibility is to keep using gradient descent. However, we will then converge to a *local minimum* which is almost guaranteed not to be a global one. Many of these local minima might be equally good in terms of training error, but have widely different generalization performances. It is assumed,

that *sharp* narrow local minima are problematic and *flat* wide local minima generalize much better, and we should try to converge to the latter.

Sophisticated optimization techniques, like Newton's method, are very good at finding narrow local minima, and should therefore be avoided. *Stochastic gradient descent*, on the other hand, seems to miss the small minima and appears to wide ones with better generalization properties.

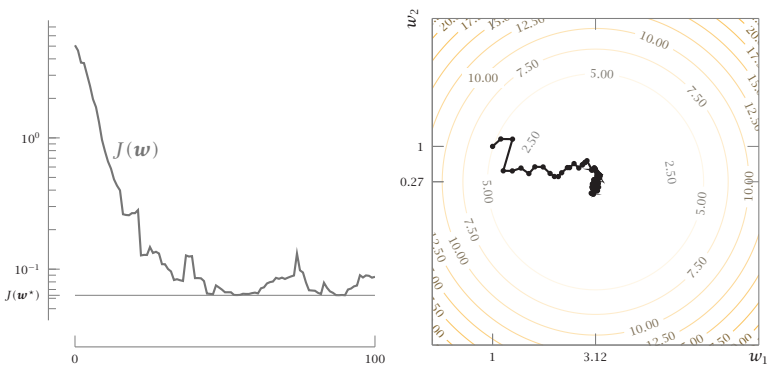
Instead of the true gradient $\nabla J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} \nabla \ell(h(\boldsymbol{x}_i), y_i)$, stochastic gradient descent uses a single random sample[1] $(\boldsymbol{x}_i, y_i)$ from the data set to estimate a stochastic gradient as

$$\tilde{\nabla} J(\boldsymbol{w}) = \nabla \ell(h(\boldsymbol{x}_i), y_i)$$

and then adjusts the parameters in this direction, namely

repeat: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \tilde{\nabla} J(\boldsymbol{w})$

As long as the *expectation* of the stochastic gradient is equal to the true gradient, *i.e.*, $\mathbf{E}[\tilde{\nabla} J(\boldsymbol{w})] = \nabla J(\boldsymbol{w})$, we are guaranteed to converge to a local minimum if a sufficiently small step size η is used.



Another advantage of stochastic gradient descent is, that it is *fast*. To calculate the true gradient, we have to iterate over the whole training data set. This is affordable if its size m is moderate, but not for really large data sets which are typical for deep neural network applications. Stochastic gradient descent, in contrast, needs only a single instance to perform an update step and thereby making progress on minimizing the objective.[2]

It is important to note, that, since the objective is nonconvex, a good initialization of the parameters is crucial. Unlike in the convex setting, we cannot simply set all of them to zero and are guaranteed to converge to a global optimum. It is common to initialize the weights of a neural network with small, zero mean normally distributed values. This approach has an advantageous side effect: Since each initialization will converge to a different local minimum (and hence yields a different model), there is no need for sampling when neural networks are *bagged*. We can simply train T networks and average their output to obtain a bagged neural network.

[1] It is important to not select the samples sequentially from the data set as this would violate the assumption that $\mathbf{E}[\nabla \ell(h(\boldsymbol{x}_i), y_i)] = \nabla J(\boldsymbol{w})$ of a stochastic gradient.

Algorithm 10.1 The stochastic gradient descent update rule.

Figure 10.4 The *stochastic* gradient descent algorithm converges to the optimal parameters $\boldsymbol{w}^\star$ on the peak demand data sets, but much slower when compared to vanilla gradient descent (Fig. 1.5). It also oscillates around the minimum and will only converge if the learning rate is decreased in each iteration.

[2] To make the gradient estimates a little less noisy, one can use a minibatch of, *e.g.*, 64 or 128 samples for the estimate instead of just a single instance. However, if too many samples are used, we again may converge to a bad narrow local minima.

How do we compute the (stochastic) gradients $\nabla \ell(h(\boldsymbol{x}_i), y_i)$? Remember $\boldsymbol{w}$ denotes an entire *set* of parameters, *e.g.*, $\boldsymbol{w} = \{U_1, \dots, U_k, \boldsymbol{c}_1, \dots, \boldsymbol{c}_k\}$, and we therefore require the gradients of the loss ℓ with respect to all of them. It is unrealistic, that the derivation of these gradients can be done analytically, even for a moderate number of layers. A way forward is to use *automatic differentiation*, discussed in the following, as an efficient way to compute gradients using the chain rule and "caching" intermediate calculations.

Acceleration and Momentum

Gradient descent is a first order method, which means it only utilizes the first-derivative (gradient). First order methods are especially suited for machine learning, since they are *fast* to evaluate, but their rate of convergence is not the best. The question is, can we improve the convergence rate without moving to higher order methods which are much more costly at each iteration?

In order to stay with first order methods, we need to exploit past gradients to adapt the next step size. The idea is, that if, the sign of the gradient is different from the previous one, then the step size is too large, because we are overshooting. Conversely, if the gradients are staying in the same direction, then the step size is too small, and larger steps are possible. Maybe we can use this observation to make steps smaller when gradients reverse sign and larger when gradients are consistently in the same direction.

Momentum gives gradient descent a short-term memory. It initializes the vector $\boldsymbol{z}$ with zero and then applies the gradient descent update rule

$$\begin{aligned} \boldsymbol{z} &\leftarrow \beta \boldsymbol{z} - \eta \nabla J(\boldsymbol{w}) \\ \boldsymbol{w} &\leftarrow \boldsymbol{w} + \boldsymbol{z} \end{aligned} \tag{10.2}$$

This modification is computational negligible, but works very well in practice. The parameter β controls the influence of the past gradients (*e.g.*, $\beta \approx 0.9$). When $\beta = 0$, it is vanilla gradient descent. The momentum update consists of the sum of $\beta \boldsymbol{z}$ and the vanilla gradient step $-\eta \nabla J(\boldsymbol{w})$ (Fig. 10.6).

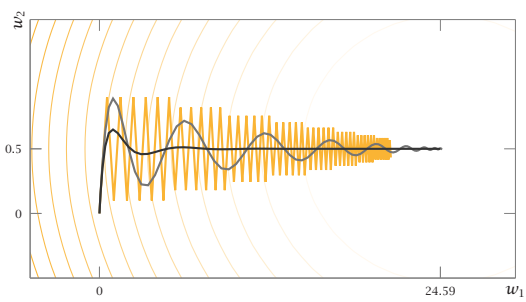


Figure 10.5 The gradients (——) are oscillating, which leads to a slow convergence. Momentum (——) can counter this problem. Even better is to use Nesterov's accelerated gradient descent (——).

The momentum acts like a smoothing on the gradient updates. We move a little further in the direction where the current gradient is the same direc-

tion as the previous one, and less far when it's in the opposite direction. In Fig. 10.5, the vanilla gradients change sign in the w_2 component (vertical axis) very rapidly. Momentum will average these oscillations out and increase the steps in the w_1 direction where the sign does not change.

Nesterov's algorithm is lightly different, but with very strong theoretical guarantees. Instead of evaluating the gradient at $\boldsymbol{w}$ and then adding the momentum to it, Nesterov's update looks ahead where momentum would take it, *i.e.*, at the position $\boldsymbol{w} + \beta z$, and then evaluates the gradient there yielding the following update rule.

$$\begin{aligned} \boldsymbol{z} &\leftarrow \beta \boldsymbol{z} - \eta \nabla J(\boldsymbol{w} + \beta \boldsymbol{z}) \\ \boldsymbol{w} &\leftarrow \boldsymbol{w} + \boldsymbol{z} \end{aligned} \tag{10.3}$$

Unfortunately, the intuition why the algorithm works so well is quite difficult to grasp. However, in practice, it works consistently slightly better than the vanilla momentum method.

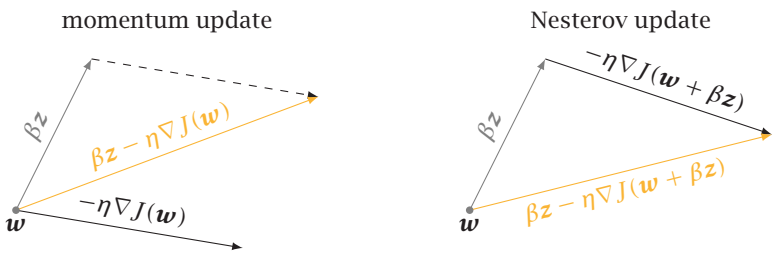


Figure 10.6 The momentum update (left) in comparison with Nesterov's accelerated gradient descent update (right). The latter does not evaluate the gradient at the current position $\boldsymbol{w}$, but looks ahead where momentum would take it.

Adaptive Learning Rates

Features normalization works well for linear models, but not for layered functions as present in deep neural networks, because it is not guaranteed, that the feature dimensions stay scaled after the first layer. Ideally, we have an individual learning rate for each parameter instead of a single global one. Of course, these learning rate then need to be set automatically.

The idea of Adagrad is simply to divide each component of the gradient by the mean squared root of its past gradient components, that is

$$\begin{aligned} \boldsymbol{g} &\leftarrow \boldsymbol{g} + \nabla J(\boldsymbol{w}) \odot \nabla J(\boldsymbol{w}) \\ \boldsymbol{w} &\leftarrow \boldsymbol{w} - \eta \frac{\nabla J(\boldsymbol{w})}{\sqrt{\boldsymbol{g}} + \epsilon} \end{aligned}$$

where $\odot$ denotes the component-wise multiplication of two vectors. Here, the role of $\epsilon \approx 1 \times 10^{-8}$ is to avoid a division by zero.

The newer Adam update rule can be seen as Adagrad with momentum.

$$\begin{aligned} \boldsymbol{z} &\leftarrow \beta_1 \boldsymbol{z} + (1 - \beta_1) \nabla J(\boldsymbol{w}) \\ \boldsymbol{g} &\leftarrow \beta_2 \boldsymbol{g} + (1 - \beta_2) \nabla J(\boldsymbol{w}) \odot \nabla J(\boldsymbol{w}) \\ \boldsymbol{w} &\leftarrow \boldsymbol{w} - \eta \frac{\boldsymbol{z}}{\sqrt{\boldsymbol{g}} + \epsilon} \end{aligned}$$

Automatic Differentiation and Backpropagation

In this section we develop an intuitive understanding of automatic differentiation, a family of techniques similar to, but more general than backpropagation for efficiently and accurately evaluating derivatives of numeric functions expressed as *computational graphs*. This is done by a recursive application of the chain rule.

Automatic differentiation can be thought of as *augmenting* a computation with the calculation of various derivatives. Since all numerical computations are ultimately compositions of a finite set of elementary operations for which derivatives are known (*e.g.*, $+$, $\cdot$, ln, sin, ...), we can apply the chain rule to these operations to get the derivative of the overall composition. Using automatic differentiation allows us therefore to calculate derivatives of complicated functions *automatically* and *exact* (up to machine precision).

Consider for example the ReLU activation function applied to the output of a linear model, namely $g(\boldsymbol{x}) = \max\{0, \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b\}$. The function g can be expressed as the following computation graph (shown in Fig. 10.7) which utilizes intermediate variables to cache computations.

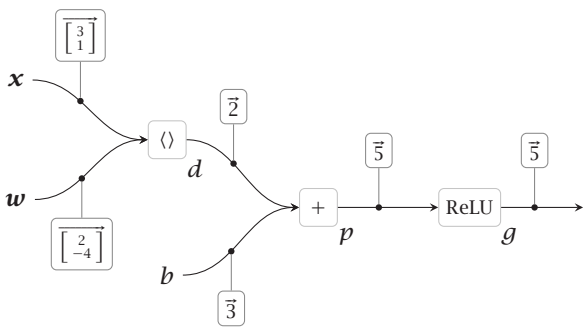


Figure 10.7 The computational graph for $g(\boldsymbol{x}) = \max\{0, \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b\}$. d and p are intermediate variables to cache computations. The pinned boxes indicate concrete values of the variables "flowing" through the computational graph.

The pinned boxes indicate concrete values of the variables "flowing" through the computational graph. Here, $\boldsymbol{x} = (3, 1)$ and $\boldsymbol{w} = (2, -4)$. Their inner product is 2 and stored in the variable d. The function output is $g(\boldsymbol{x}) = 5$.

We are now interested in the derivative g with respect to the parameters, *i.e.*, $\partial g / \partial \boldsymbol{w}$ (and also $\partial g / \partial b$), which can be obtained via chain rule, that is

$$\frac{\partial g}{\partial \boldsymbol{w}} = \frac{\partial g}{\partial d} \frac{\partial d}{\partial \boldsymbol{w}} = \frac{\partial g}{\partial d} \frac{\partial \langle \boldsymbol{w}, \boldsymbol{x} \rangle}{\partial \boldsymbol{w}} = \frac{\partial g}{\partial d} \boldsymbol{x}.$$

The derivation is not complete, since we do not know $\partial g / \partial d$ (yet). However, we can again apply the chain rule to obtain

$$\frac{\partial g}{\partial d} = \frac{\partial g}{\partial p} \frac{\partial p}{\partial d} = \frac{\partial g}{\partial p} \frac{\partial (d + b)}{\partial d} = \frac{\partial g}{\partial p} 1$$

now missing the derivative $\partial g / \partial p$. This will continue until we reach the end of the computation graph with $\partial g / \partial g = 1$. It might therefore be a good idea

to directly start at the end and work backwards from there. This approach is known as the reverse accumulation mode of automatic differentiation.

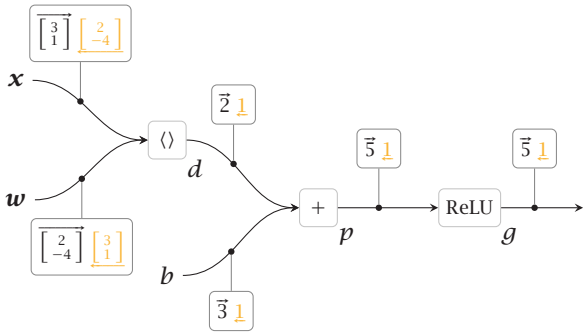


Figure 10.8 The computational graph for $g(\boldsymbol{x}) = \max\{0, \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b\}$ in reverse accumulation mode.

For the given forward pass with $\boldsymbol{x} = (3, 1)$, $b = 3$ and $\boldsymbol{w} = (2, -4)$, the derivatives of g are $\partial g / \partial \boldsymbol{w} = (3, 1)$ and $\partial \sigma / \partial b = 1$, obtained from the computational graph in Fig. 10.8; starting at the end of the graph.

Importantly, automatic differentiation can not only be applied to closed form expressions, but also to algorithms making use of control flow such as branching, loops, recursion, and function calls, giving it an important advantage over symbolic differentiation.

Notice, that in the backwards pass, each of the computational nodes $\langle\rangle$, $+$ and ReLU receive incoming gradients from successor nodes and propagate new gradients back to all predecessors after the chain rule was applied. The application of the chain rule may require the node to store some information during the forward pass. For example, the derivative of the ReLU unit is 1 if it received a value greater than zero during the forward pass, and 0 otherwise — its gradient depends on the values during the forward pass.

The nodes of the computational graph can store the required information completely independent, without knowing the entire computational graph.

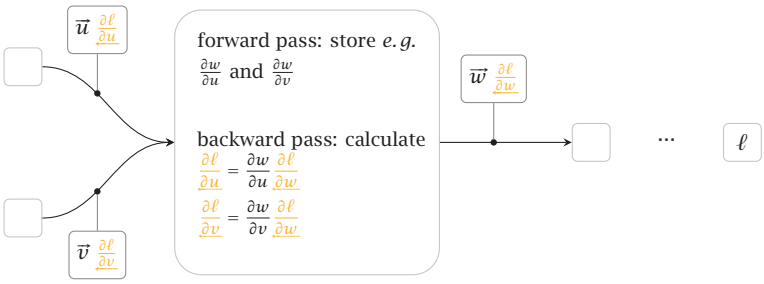


Figure 10.9 Anatomy of a computational node. In the forward pass, it stores all necessary information to apply the chain rule when receiving a gradient in the backward pass.

Before we can calculate derivatives of deep neural networks, we have to generalize the notion of gradients to vector valued functions, *e.g.*, $\boldsymbol{x} \mapsto U\boldsymbol{x}$, and we need a way to *efficiently* apply the chain rule on such functions.

Thus far, we discussed partial derivatives and gradients of real valued functions of the form $f \colon \mathbf{R}^n \to \mathbf{R}$. However, many operations in a neu-

ral network map vector to vectors, such as $U\boldsymbol{x}$. We therefore require a generalized notion of gradients to functions with signature $f\colon \mathbf{R}^n \to \mathbf{R}^m$.

For input vectors $\boldsymbol{x} \in \mathbf{R}^n$ and functions $f\colon \mathbf{R}^n \to \mathbf{R}^m$ the Jacobian is defined as the $m \times n$ matrix

$$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1(\boldsymbol{x})}{\partial x_1} & \frac{\partial f_1(\boldsymbol{x})}{\partial x_2} & \cdots & \frac{\partial f_1(\boldsymbol{x})}{\partial x_n} \\ \frac{\partial f_2(\boldsymbol{x})}{\partial x_1} & \frac{\partial f_2(\boldsymbol{x})}{\partial x_2} & \cdots & \frac{\partial f_2(\boldsymbol{x})}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m(\boldsymbol{x})}{\partial x_1} & \frac{\partial f_m(\boldsymbol{x})}{\partial x_2} & \cdots & \frac{\partial f_m(\boldsymbol{x})}{\partial x_n} \end{bmatrix}.$$

Using this definition, the Jacobian of a *linear transformation* $U\boldsymbol{x} \in \mathbf{R}^n \to \mathbf{R}^m$ is the $\partial U\boldsymbol{x}/\partial \boldsymbol{x} = U$.
The Jacobian for an element-wise applied activation function $\boldsymbol{z} = \sigma(\boldsymbol{x})$ is the diagonal matrix

$$\frac{\partial \sigma(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial \sigma(x_1)}{\partial x_1} & 0 & \cdots & 0 \\ 0 & \frac{\partial \sigma(x_2)}{\partial x_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\partial \sigma(x_n)}{\partial x_n} \end{bmatrix}$$

Note that, the Jacobian matrix of scalar-valued functions $f\colon \mathbf{R}^n \to \mathbf{R}$ is the *transpose* of the gradient which is a common source for errors.

With Jacobian matrices, the chain rule generalizes to the (transposed) Jacobian-vector product defined as

$$\frac{\partial \ell}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}^{\mathsf{T}} \frac{\partial \ell}{\partial \boldsymbol{z}}, \tag{10.4}$$

where $\partial \ell / \partial \boldsymbol{z}$ is the incoming gradient and $\partial \boldsymbol{z}/\partial \boldsymbol{x}$ is the Jacobian of the computational node. Recall, that all we ever need for the backward pass is the *result* of the Jacobian-vector product, which gives an opportunity for optimization: Consider an activation function σ. Its Jacobian will be a diagonal matrix with $\partial \sigma(\boldsymbol{x}_j)/\partial \boldsymbol{x}_j$ entries. Any incoming gradient vector which is multiplied by a diagonal Jacobian will have its entries scaled by the factors in the diagonal matrix. So, instead calculating the potentially huge Jacobian matrix of an element-wise applied function, we can simply scale each element of the incoming gradient by the corresponding factors without ever explicitly forming the Jacobian.

In fact, the only thing required for automatic differentiation are Jacobian-vector products. One of the most complicated operations is the matrix-matrix multiplication $Z = UX$ (which generalizes all matrix-vector and vector-vector operations). The Jacobian-vector products for $Z = UX$ are

$$\begin{aligned} \frac{\partial \ell}{\partial X} &= \frac{\partial Z}{\partial X}^{\mathsf{T}} \frac{\partial \ell}{\partial Z} = U^{\mathsf{T}} \frac{\partial \ell}{\partial Z} \qquad \text{and} \\ \frac{\partial \ell}{\partial U} &= \frac{\partial Z}{\partial U}^{\mathsf{T}} \frac{\partial \ell}{\partial Z} = \frac{\partial \ell}{\partial Z} X^{\mathsf{T}} \end{aligned} \tag{10.5}$$

where $\partial \ell / \partial Z$ is the incoming gradient/Jacobian matrix.

With these considerations, a stochastic gradient descent step for an entire neural network with a squared loss function, namely

$$\ell(\hat{y}, y) = \tfrac{1}{2}(\langle \boldsymbol{w}, \phi(\boldsymbol{x}) \rangle + b - y)^2 = \tfrac{1}{2}(\boldsymbol{w}^{\mathsf{T}} \max\{0, U\boldsymbol{x} + c\} + b - y)^2$$

can be derived with the following computational graph.

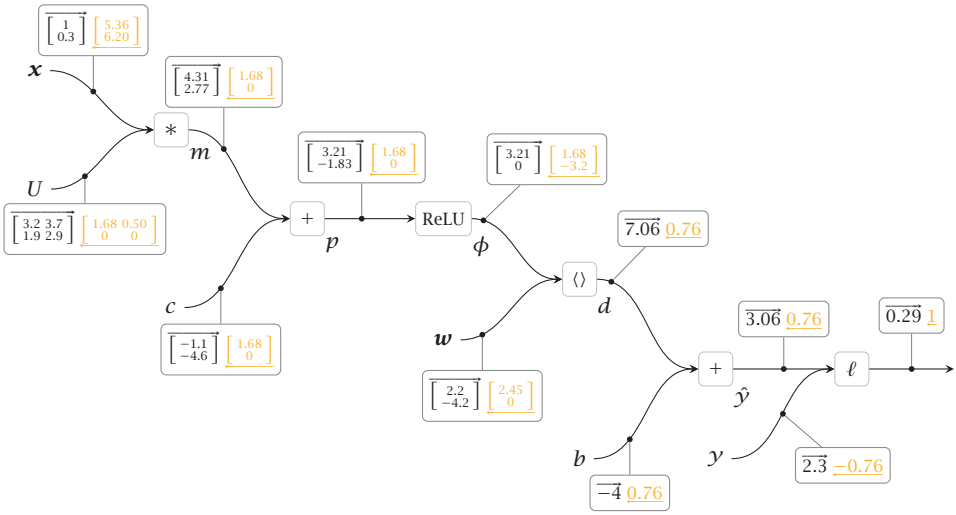


Figure 10.10 The computational graph for the objective $\frac{1}{2}(\langle \boldsymbol{w}, \phi(\boldsymbol{x})\rangle + b - y)^2$ of a full neural network.

Neural Network Regularization

As already mentioned, any common regularization can be applied to penalize the magnitude of the parameters, as for example ℓ_2 regularization. Another form of regularization is *dropout* which randomly "drops" neurons in the forward pass by setting their output to zero or *early stopping* which essentially interrupts the optimization algorithm after several iterations.

Dropout

During training, dropout randomly selects a subset of neurons and sets their output to zero. Then, during backpropagation, only the gradient of neurons that were kept "alive" during the forward pass are propagated. This is done in each forward/backward pass. Dropout is a kind of bagging since we are essentially training on much smaller networks in each pass and then average their predictions, essentially reducing the model variance.

After training, the entire network is used to make predictions. However, in the prediction step, the output of each neuron must then be scaled such that its magnitude is approximately the same as during training.

Early Stopping

Another form of regularization is early stopping, where the optimizer is interrupted before the parameters are close to a local optimum (Fig. 10.11). Typically, the number of gradient descent steps is not fixed in advanced, but determined using cross validation. Every time the error on the validation

set improves, we store a copy of the model parameters. After training, the parameters with the lowest *generalization error* is taken, and not the ones with the lowest objective. Early stopping thus has the effect of restricting the feasible parameters to a relatively small, but not fixed, ball in the neighborhood of the initialization.

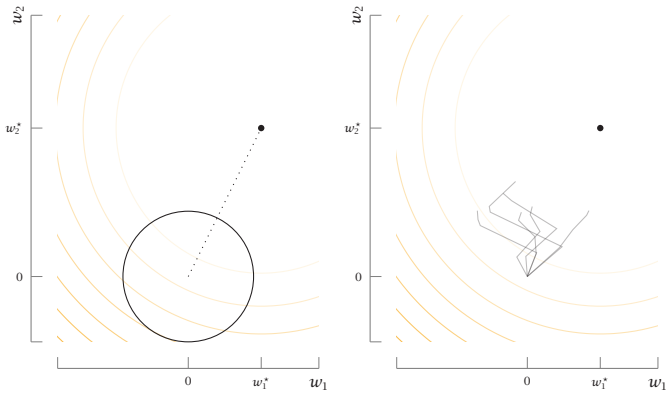


Figure 10.11 Early stopping (right) compared to ℓ_2 regularization (left). Each run of stochastic gradient descent was stopped after five steps, leading to a different parameter estimate each time, even though the initial position was identical.

Recall, that digital images are just matrices with entries between 0 and 255, where each value corresponds to the brightness of one pixel in the image as, *e.g.*, the 256×256 matrix in Fig. 10.12. Images can have multiple channels. If Fig. 10.12 was RGB image, we would have three 256×256 matrices, one for the red, green and blue channel. Instead of three individual matrices, we then would talk about a $256 \times 256 \times 3$ *tensor*.

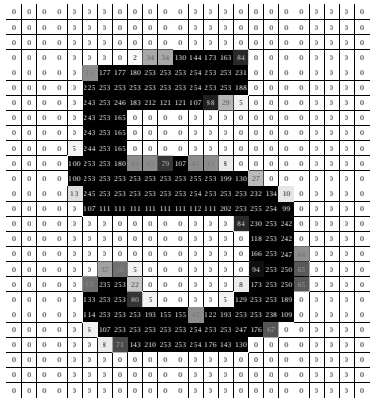


Figure 10.12 Each pixel of an image is just a number between 0 and 255.

Most of the machine learning algorithm considered so far, expected vectors of numbers as inputs and not images. Although, it is possible to convert images to vectors, by simply stacking the columns of the image matrix, we should avoid this, as we would throw away valuable information.

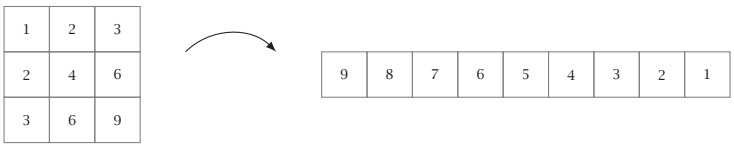


Figure 10.13 Vectorial representation of an image by stacking its columns.

It turns out, that natural images have very specific properties we would like to exploit in order to make better predictions. For example, if we would move the digit 5 in Fig. 10.12 one pixel to the right, it would still be the digit 5. However, the vectorial representation we would get by stacking the columns would be drastically different from the one we had before the shift.

This concept is called (local) translation invariance: If we move any image by a small amount, the semantic interpretation of the image does not change. This is a very crucial property for image classification and the reason convolutional neural networks (CNN) use a different feature function.

Recall, that feedforward neural networks get an input vector $\boldsymbol{x}$ transform it through a series of feature functions ϕ_i, each one producing an intermediate vector $\boldsymbol{z}_{i+1}$, and finally make a prediction based on the last transformation:

$$\begin{aligned} \boldsymbol{z}_1 &= \boldsymbol{x} \\ \boldsymbol{z}_{i+1} &= \phi(\boldsymbol{z}_i), \quad i = 1, \dots, k-1 \\ h(\boldsymbol{x}) &= U_k \boldsymbol{z}_k + \boldsymbol{c}_k \end{aligned}$$

A convolutional neural network takes advantage of the fact that the input is an image, and designs the intermediate results in a more sensible way, keeping the "image layout" as long as possible: each $\boldsymbol{z}_i$ is again an image or, more precise, a $w \times h \times d$ tensor with $w \times h$ "pixels" and d channels.

There are three main types of layers to build CNN architectures: Convolutional layers, pooling layers, and fully connected layers, illustrated in Fig. 10.14. The latter is the same as in regular feedforward neural networks.

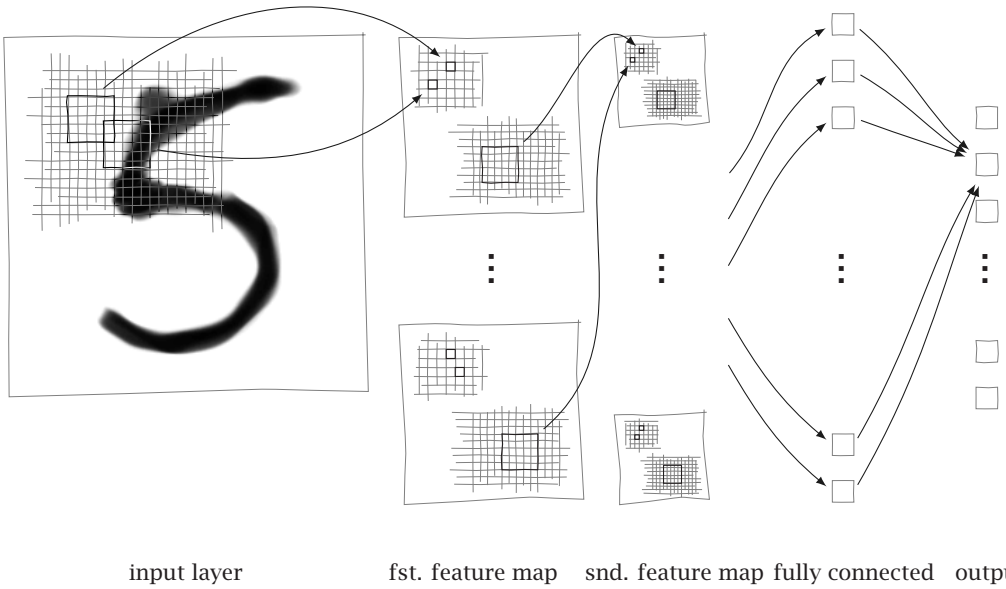


Figure 10.14 Graphical illustration of a convolutional neural network with two convolutional and two fully connected layers (after Mike O'Neill).

Convolutional Layer

The convolutional layer is the core building block of a CNN. Its parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input tensor. For example, the first layer in Fig. 10.14 uses k 5×5 filters which are combined to a single $5 \times 5 \times 1 \times k$ tensor: 5 pixels width, 5 pixels height, 1 depth (because the input image is gray scale with only one "color" channel) and k output images. If the input was a RGB image, we would have used a $5 \times 5 \times 3 \times k$ tensor. Each of these k filters produce an output "image" as we slide (more precisely, convolve) each one across the width and height of the input tensor and compute dot products between the entries of the filter and the input at any position of the image.

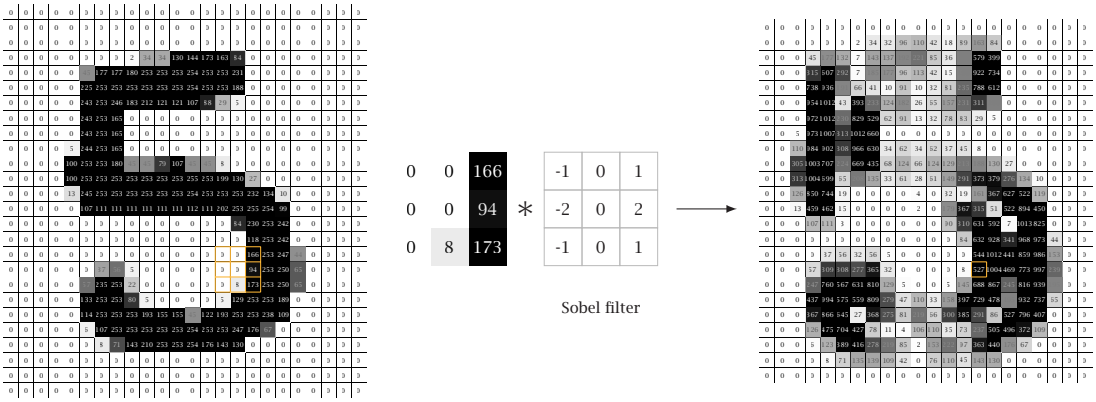


Figure 10.15 Vertical edge detection with a 3×3-Sobel filter. The image on the right show the absolute values after the filter was applied.

In Fig. 10.15, we manually specified the nine entries of the filter. However, in a convolutional neural network, these values are learned by defining an objective function and then find its minimum via optimization.

Notice, that the resulting image after convolution in Fig. 10.15 is two pixels smaller in width and height. Since the input image is 24×24 pixels and the filter is 3×3 spatially, we can only move it 22 times in each direction. To combat the shrinking effect, one can add a frame of zeros around the input image; a technique called zero padding.

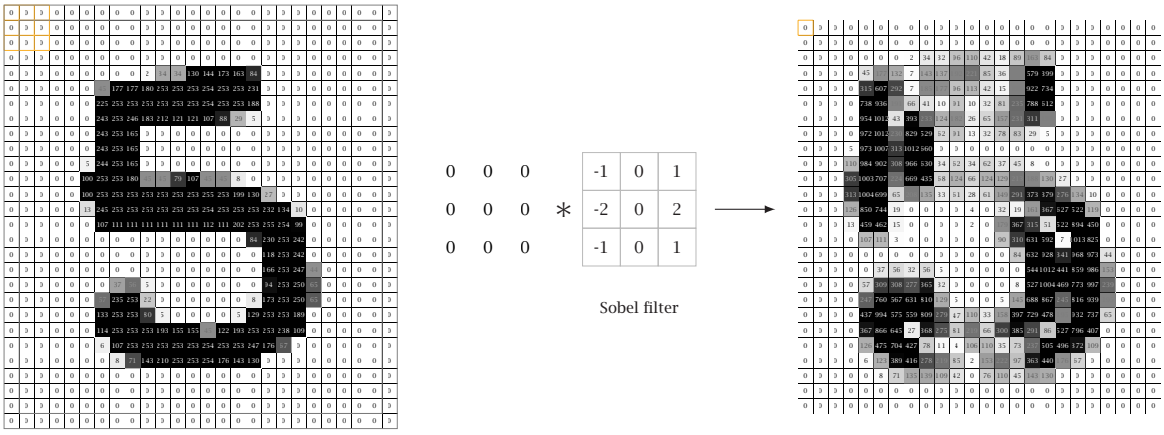


Figure 10.16 The size of the output image does not change, if zero padding is applied to the input image.

Another parameter of a convolution layer is the stride. If the stride is 1, the filters are moved one pixel at a time. With a stride of 2 (more is rare in practice) the filters jump two pixels at a time as we slide them around.

Notice, that the convolution is a *linear* operator and therefore requires a nonlinear activation function afterwards. This can be any of the activation functions discussed so far, but most common is the ReLU or its variants.

Pooling Layer

It is common to periodically insert a pooling layer in-between successive convolutional layers. Its function is to progressively reduce the size of the intermediate representations and hence shrink the amount of parameters of the network. The pooling layer operates independently on every channel and resizes it spatially, using the max operation. The most common form is a pooling layer with filters of size 2×2 applied with a stride of 2. This shifts the filter not one but two pixels every time.

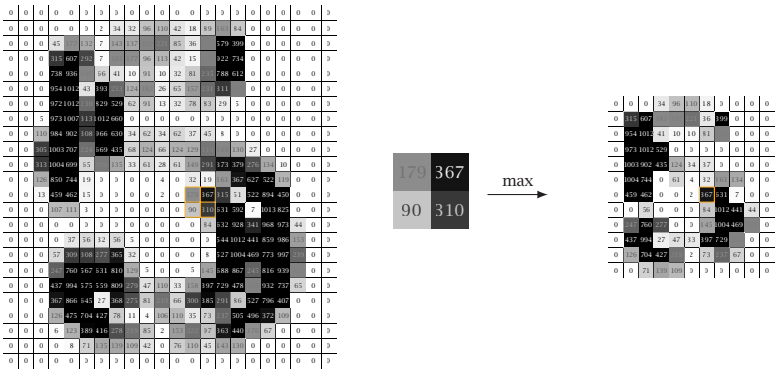


Figure 10.17 Max pooling with a filter of size 2×2 and stride 2.

Every max operation would, in this example, be taking a max over 4 numbers (little 2×2 region in some channel), reducing the image size by 75 %.

Fully Connected Layer

Although fully connected layers can be realized as in regular neural networks, *i.e.*, $\boldsymbol{z} \mapsto \sigma(U\boldsymbol{z} + c)$, it is much more common to use again convolutions. If, for example, the last convolution yields a feature map of $7 \times 7 \times 512$, and we need scores for ten different classes, we can use a convolution with filter size $7 \times 7 \times 512 \times 10$ which outputs a tensor of shape $1 \times 1 \times 10$ — each channel corresponding to a particular class score.

Feature Map Visualization

Since neural networks are not only differentiable with respect to the parameters $\boldsymbol{w}$, but also to their inputs $\boldsymbol{x}$, we can use gradient descent to iteratively tweak the input until it produces a high value in the feature map. This allows us to visualize what individual filters detect in the image (Fig. 10.18).

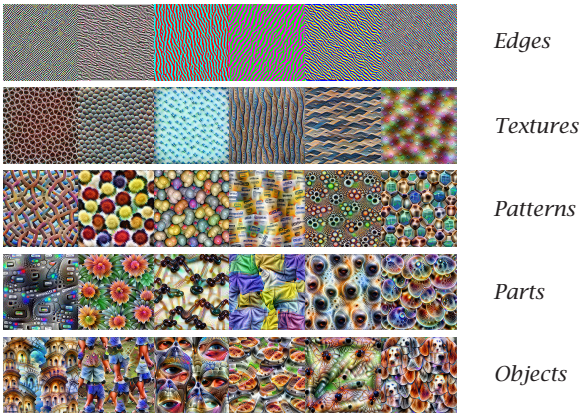


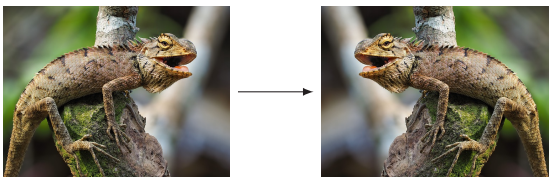
Figure 10.18 Feature visualization allows us to see how GoogLeNet, trained on the ImageNet dataset, builds up its understanding of images over many layers.

Olah, C., Mordvintsev, A., & Schubert, L. (2017). Feature Visualization. Distill. https://doi.org/10.23915/distill.00007

Data Augmentation

Deep neural networks perform remarkably well on many computer vision tasks. The more data these networks have access to, the more effective it can be, even when the data is of low quality. Data augmentation encompasses a suite of techniques that artificially supplement the size of the training set.

Flipping The image is flipped on one axis. Flipping on the vertical axis is much more common than flipping the horizontal axis. If the flipping operation preserves whatever we are trying to recognize in the image, then this would be a good data augmentation technique to use; otherwise not.



Cropping Another commonly used technique is random cropping which takes a slightly shifted central patch of each image.

Rotation Rotation augmentations are done by rotating the image clockwise or counterclockwise. Often, we can only apply a slight rotation or the semantic of the image changes, as for example, with digit recognition tasks.

Translation Shifting images left, right, up, or down can be a very useful transformation to avoid positional bias in the data. If, for example, a face data set has only centered faces, it is very likely that the trained model can only detect faces if they are perfectly centered in an image.

Color Space Transformations Given an image, we can add different distortions to the color channels. The images can also be brightened or darkened. There is a lot of freedom for creativity with color space augmentations.

Noise Injection To make the model more resilient against noise, we can artificially tamper the input image with small values, usually drawn from a normal distribution.

Typically, data augmentation techniques can be combined, *e.g.*, before an image is fed into the network, we randomly rotate it, change the color space and add some noise. By chance, we flip it on the vertical axis.

Transfer Learning

In practice, very few people train an entire convolutional neural network *from scratch*, since it is relatively rare to have a data set of sufficient size. Instead, deep neural networks are pretrained on very large data sets. The parameters obtained that way are then either used as an *initialization* for the actual network, or taken as a fixed *feature extractor* for other algorithms.

Sequence Models

A deep neural network (or any model discussed so far) takes a single input $\boldsymbol{x}$ and makes a prediction $\hat{y} = h(\boldsymbol{x})$. For example $\boldsymbol{x}_1$ could be an image, and the task is to classify its contents. After that, the next image $\boldsymbol{x}_2$ can be classified. It is assumed, that the inputs are *independent*, *i.e.*, the label of input $\boldsymbol{x}_i$ does not tell us anything about the label of $\boldsymbol{x}_j$. In the following, such networks are called one-to-one represented by the graph in Fig. 10.19.

Figure 10.19 A feedforward neural network maps one input to one output and is therefore called *one-to-one* network. Each rectangle is a vector and arrows represent computations.

For some tasks, the independence assumption is correct, but not for all. In practice, we often want to predict a sequence of outputs, given a sequence of inputs. Just predicting each output independently would miss crucial information from the context. A typical example is the classification of video or movie data. Certainly, video frames are not independent, and many-to-many networks are better suited for such tasks than one-to-one.

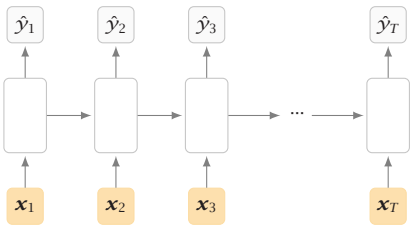


Figure 10.20 A *many-to-many* network predicts a sequence of outputs, given a sequence of inputs.

If a single prediction at the end of the sequence is required, we call this a many-to-one network. A typical application would be *sentiment analysis*, where the goal is to classify if a given sentence expresses a positive or a negative sentiment after its entire content was processed by the network.

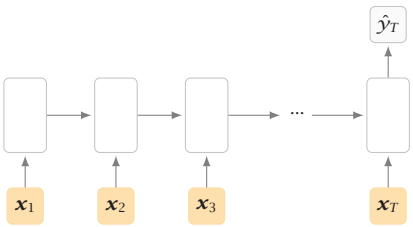


Figure 10.21 A *many-to-one* network that predicts a single label at the end of the input sequence.

The opposite of a many-to-one architecture is a one-to-many network, which can be used, for example, to generate a caption (a sequence of words) for an image (single input).

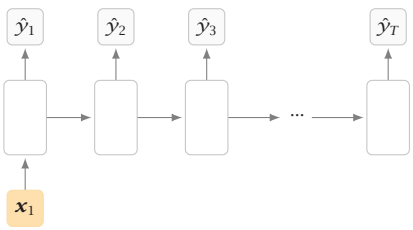


Figure 10.22 A *one-to-many* network predicts a sequence of outputs, given a single input.

Recurrent Neural Networks

A recurrent neural network (RNN) belongs to a family of neural networks, that are specialized for processing sequential data. The idea is to maintain a *hidden state* $\boldsymbol{z}_t$ over time that encodes or summaries all past information.

In the original recurrent neural network,[1] the hidden state is function of the *current input* and the *previous hidden state*, formally

$$\boldsymbol{z}_t = \sigma(U_{\boldsymbol{x}}\boldsymbol{x}_t + U_{\boldsymbol{z}}\boldsymbol{z}_{t-1} + \boldsymbol{c}_{\boldsymbol{x}})$$
$$\hat{y}_t = U_y\boldsymbol{z}_t + \boldsymbol{c}_y$$

were the first hidden state $\boldsymbol{z}_0$ is typically initialized with all zeros.

[1] This is sometimes called a vanilla RNN or an Elman RNN after Prof. Jeffrey Elman.

The computational graph or a many-to-many recurrent neural network is depicted in Fig. 10.23 which can easily be adapted to other sequence problems like one-to-many or many-to-one. Notice, that the *same* set of parameters $\boldsymbol{w} = \{U_{\boldsymbol{x}}, \boldsymbol{c}_{\boldsymbol{x}}, U_y, \boldsymbol{c}_y, U_{\boldsymbol{z}}\}$ is used at every time step.

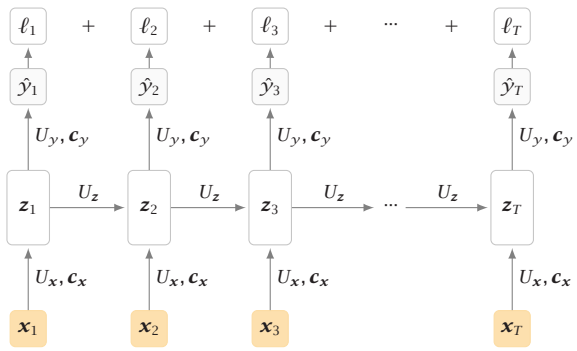


Figure 10.23 The computational graph or a many-to-many recurrent neural network.

During training, a loss function is evaluated at each time step of the sequence. After reaching the end of the input sequence, all individual losses are summed up to a final loss, which can be used to adjust all parameters in $\boldsymbol{w}$. Automatic differentiation will keep track of the whole sequence.

Example 10.1

In this example, we are given a letter and would like to know the most probable character following it. More precisely, want a likelihood score for each possible next character. This will then allow us to generate new text one character at a time.

Consider a dictionary of four letters $\{\mathrm{h}, \mathrm{o}, \mathrm{n}, \mathrm{r}\}$, which are encoded into a vector using the 1-of-k encoding. In the following, the word "honor" if fed into the network, one character at a time. The network outputs a vector of scores. Each score indicates how likely the corresponding encoded character is as the next one.

At each time step, we calculate the *cross entropy loss* between the label y and the prediction $\hat{y}$. After the end of the sequence is reached, all individual losses are summed up, and backpropagation is used to increase the score of the correct predictions (underlined) and decrease the score of

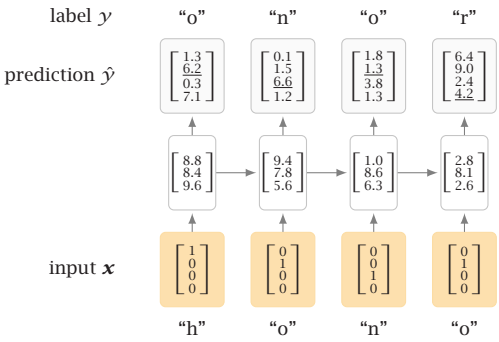


Figure 10.24 A recurrent neural network to predict the next character in a sequence. The score of the correct next character is underlined. During training, we optimize the parameters of the RNN, such that these scores become higher and all others lower.

all others. In practice, backpropagation can be run on smaller chunks of the sequence. This is especially useful, if the sequences are very long.

Notice, that the first time we want to map "o" to "n", and the second time "o" to "r". This shows that the input alone is not sufficient to predict the next character, and that it is necessary to keep a track of past inputs.

Encode-Decoder Architectures

There is also a variant of the *many-to-many* network, where the sequences are not synchronized and potentially of different length. A typical example is the automatic translation from one language into another. It turns out, this is best modeled with two recurrent neural networks: A many-to-one network, called the encoder, which summaries all information about the inputs sequence in its hidden vector. This hidden vector can then be provided as a single input to a one-to-many network, the decoder, which then produces the output sequence (illustrated in Fig. 10.25).

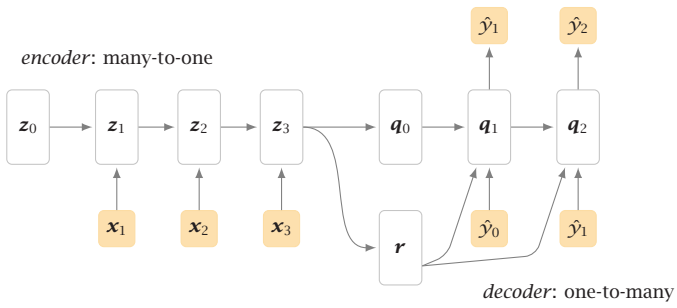


Figure 10.25 Predicting unsynchronized sequences of possible different length. The first many-to-one network encoders the input sequence into a single context vector $\boldsymbol{r}$, which is then provided as an input to the one-to-many network which produces an output sequence. The hidden state of the encoder is denoted by $\boldsymbol{z}$, the hidden state of the decoder by $\boldsymbol{q}$. The first input of the decoder $\hat{y}_0$ is typically a start token which signalizes the beginning of the sequence.

The encoder and decoder networks can be very different. In image captioning, for example, we want to output sequences of text in the context of an image. This can be achieved by a convolution neural network[1] which encodes the content of an image into a feature vector (the context). This feature vector is then passed to a recurrent neural network which is trained to output words that describe the content of that image (see Fig. 10.26).

To signal the start and end of the caption, we introduce two special tokens <S> and <E> respectively. At prediction time, the decoder gets the encoded

[1] We can, for example, take an already trained neural network for image classification and use the output of one of the last fully connected layers as a feature vector. The intuition is, that since the CNN is able to classify the content of the image based on these features, all necessary information about the image content must be encoded in these values.

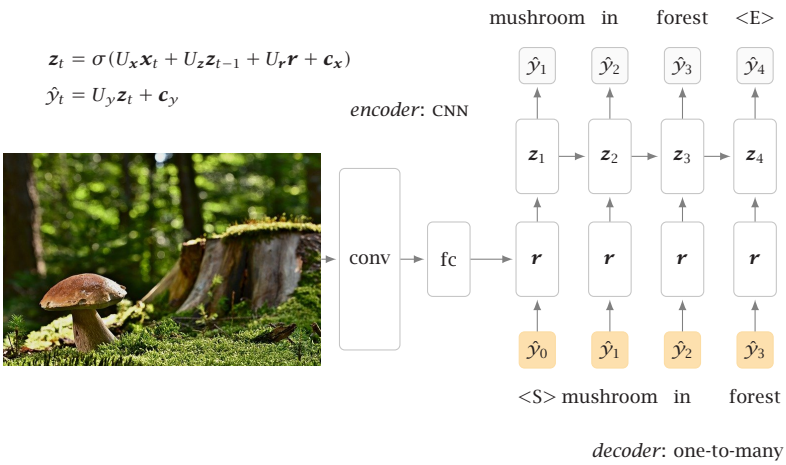


Figure 10.26 Image captioning with recurrent neural networks. In this example we use a CNN to encode the image into a feature vector which is used as context for the caption.

image and the start token as input and predicts a distribution (or scores) over possible words. One of these words is sampled and passed (again together with the encoded image features) back to decoder for the next iteration until the end token is sampled.

Vanishing and Exploding Gradients

The training of recurrent neural networks becomes problematic for longer sequences, which is best seen from the vanilla RNN forward pass:

$$\boldsymbol{z}_t = \sigma(U_{\boldsymbol{x}}\boldsymbol{x}_t + U_{\boldsymbol{z}}\boldsymbol{z}_{t-1} + \boldsymbol{c}_{\boldsymbol{x}})$$
$$\hat{y}_t = U_y\boldsymbol{z}_t + \boldsymbol{c}_y$$

By chain rule, the derivative $\partial\ell/\partial U$ involves many powers of U. As a consequence, if the largest eigenvalue of U is less than one, the gradients *vanish* and if it is greater than one, the gradients *explode*. More sophisticated architectures such as *attention networks* can avoid these problems.

Attention

The *sequence-to-sequence* model in Fig. 10.25 used a single context vector $\boldsymbol{r}$ (in this example it was the final hidden state of the encoder, *i.e.*, $\boldsymbol{r} = \boldsymbol{z}_T$) that encoded *all* necessary information about the input sequence, such that the decoder can produce an output sequence. For long input sequences however, the context vector $\boldsymbol{r}$ does not have the capacity to store such large amount of information. In order to overcome this shortcoming, we provide an individual context vector at each time step of the decoder network. These new context vectors encode only relevant parts of the input sequence, and allow the decoder to focus on these; a mechanism which is called attention.

The encoder part stays identical to the one in Fig. 10.25. New is an alignment (or similarity) function $f(\boldsymbol{q}, \boldsymbol{z})$, which calculates a score that tells the

decoder how much attention $\boldsymbol{a}$ it should put on the encoder hidden state $\boldsymbol{z}$ when the decoder is in hidden state $\boldsymbol{q}$. The new context vector $\boldsymbol{r}$ is then a weighted sum of the encoder hidden states and the attention scores. This way, the network learns by itself how much weight it wants to put on each of the hidden states of the input sequence.

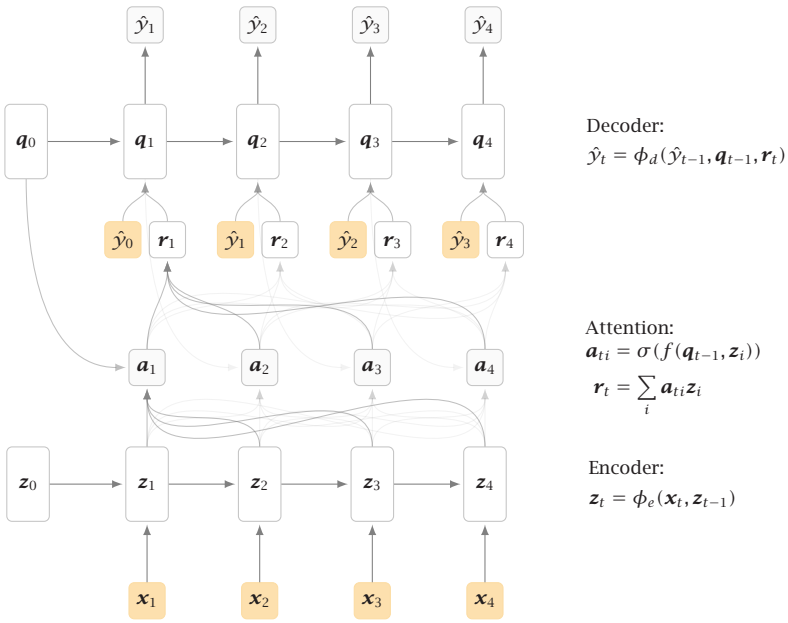


Figure 10.27 A sequence-to-sequence network with *attention* mechanism.

It turns out, that the attention mechanism is much more general and can be abstracted into a new neural network layer, the attention layer. This new layer is a mapping from a set of queries $\{\boldsymbol{q}_1, \boldsymbol{q}_2, \dots\}$ to a set of context vectors $\{\boldsymbol{r}_1, \boldsymbol{r}_2, \dots\}$. We collect both in design matrices Q and R respectively. Similar, the hidden encoder states $\{\boldsymbol{z}_1, \boldsymbol{z}_2, \dots\}$ are collected in the matrix Z, but these are called input vectors in the context of an attention layer.

$$Q = \begin{bmatrix} - \boldsymbol{q}_1^\mathsf{T} - \\ - \boldsymbol{q}_2^\mathsf{T} - \\ \vdots \end{bmatrix} \quad R = \begin{bmatrix} - \boldsymbol{r}_1^\mathsf{T} - \\ - \boldsymbol{r}_2^\mathsf{T} - \\ \vdots \end{bmatrix}$$

$$Z = \begin{bmatrix} - \boldsymbol{z}_1^\mathsf{T} - \\ - \boldsymbol{z}_2^\mathsf{T} - \\ \vdots \end{bmatrix}$$

Notice, that the vectors $\boldsymbol{z}_i$ are used two times: once in the calculation of $\boldsymbol{a}_{ti}$ and a second time in the weighted sum $\boldsymbol{r}_t$. To gain more flexibly, we can map $\boldsymbol{z}_i$ by some linear transformations U_k and U_v before being used. That way, we obtain design matrices of so-called key vectors $K = ZU_k$ and value vectors $V = ZU_v$.

query vectors: Q
input vectors: Z

key vectors: $K = ZU_k$
value vectors: $V = ZU_v$
attention: $A = \sigma(QK^\mathsf{T})$
result: $R = AV$

For efficiency, an attention layer does not allow a generic alignment function f, but defines the similarity between key K and query Q as an inner product QK^T. The overall mapping of the attention layer is then $R = \sigma(QK^\mathsf{T})V$. This is, in contrast to the vanilla RNN, highly parallelizeable.

One special case is the self-attention layer, where the query vectors are derived from the inputs as $Q = ZU_q$. This allows us to compare each input vector $\boldsymbol{z}_i$ to all other input vectors $\{\boldsymbol{z}_1, \boldsymbol{z}_2, \dots\}$. Current state of the art sequence-to-sequence models are solely build on self-attention layers and called transformer networks.

Supervised Cheat Sheet

General

Design matrices

$$X = \begin{bmatrix} - \boldsymbol{x}_1^\mathsf{T} - \\ \vdots \\ - \boldsymbol{x}_m^\mathsf{T} - \end{bmatrix} \in \mathbf{R}^{m \times n}, \quad \boldsymbol{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \in \mathbf{R}^m$$

A *gradient* is a vector containing the partial derivatives with respect to each dimension.

$$\nabla J(\boldsymbol{w}) = \Big(\frac{\partial J(\boldsymbol{w})}{\partial w_1}, \dots, \frac{\partial J(\boldsymbol{w})}{\partial w_n} \Big)$$

A *stochastic gradient* requires $\mathbf{E}[\tilde{\nabla} J(\boldsymbol{w})] = \nabla J(\boldsymbol{w})$.

Loss Functions

Regression:

Square Loss: $\ell_{\text{sq}}(h(\boldsymbol{x}), y) = (h(\boldsymbol{x}) - y)^2$.
Absolute Loss: $\ell_{\text{abs}}(h(\boldsymbol{x}), y) = |h(\boldsymbol{x}) - y|$.

Classification:

Zero-one Loss: $\ell_{0/1}(h(\boldsymbol{x}), y) = 0$ if $\text{sign}(h(\boldsymbol{x})) = y$, and 1 otherwise.
Logistic Loss: $\ell_{\text{logistic}}(h(\boldsymbol{x}), y) = \log(1 + \exp(-y \cdot h(\boldsymbol{x})))$.
Hinge Loss: $\ell_{\text{hinge}}(h(\boldsymbol{x}), y) = \max\{0, 1 - y \cdot h(\boldsymbol{x})\}$.
Exponential Loss: $\ell_{\text{exp}}(h(\boldsymbol{x}), y) = \exp(-y \cdot h(\boldsymbol{x}))$.

Regularization

ℓ_2 regularization: $\Omega(\boldsymbol{w}) = \sum_{i=1}^n w_i^2 = \|\boldsymbol{w}\|^2$.
ℓ_1 regularization: $\Omega(\boldsymbol{w}) = \sum_{i=1}^n |w_i| = \|\boldsymbol{w}\|_1$.

Gradient Descent

repeat: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla J(\boldsymbol{w})$ (ordinary)

repeat: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \tilde{\nabla} J(\boldsymbol{w})$ (stochastic)

Least Squares Regression

- Vector valued inputs and parameters: $\mathscr{X} = \mathscr{W} = \mathbf{R}^n$.
- Real labels and predictions: $\mathscr{Y} = \mathscr{Q} = \mathbf{R}$.
- Linear hypothesis class: $\mathscr{H} = \{ \langle \boldsymbol{w}, \boldsymbol{x} \rangle : \boldsymbol{w} \in \mathscr{W} \}$.
- Square loss: $\ell_{\text{sq}}(h(\boldsymbol{x}), y) = (h(\boldsymbol{x}) - y)^2$.
- Matrix notation: $J(\boldsymbol{w}) = \frac{1}{m} \|X\boldsymbol{w} - \boldsymbol{y}\|^2$.
- Gradient: $\nabla J(\boldsymbol{w}) = \frac{2}{m} X^\mathsf{T}(X\boldsymbol{w} - \boldsymbol{y})$.
- Closed form solution: $\boldsymbol{w}^\star = (X^\mathsf{T}X)^{-1}X^\mathsf{T}\boldsymbol{y}$.
- ℓ_2 regularization leads to *ridge regression* with $\boldsymbol{w}^\star = (X^\mathsf{T}X + m\lambda I)^{-1}X^\mathsf{T}\boldsymbol{y}$.
- ℓ_1 regularization leads to *lasso regression*.

Robust Regression

- Vector valued inputs and parameters: $\mathscr{X} = \mathscr{W} = \mathbf{R}^n$.
- Real labels and predictions: $\mathscr{Y} = \mathscr{Q} = \mathbf{R}$.
- Linear hypothesis class: $\mathscr{H} = \{ \langle \boldsymbol{w}, \boldsymbol{x} \rangle : \boldsymbol{w} \in \mathscr{W} \}$.
- Absolute loss: $\ell_{\text{abs}}(h(\boldsymbol{x}), y) = |h(\boldsymbol{x}) - y|$.
- Gradient: $\nabla J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^m \boldsymbol{x}_i \,\text{sign}(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle - y_i)$.

Logistic Regression

- Vector valued inputs and parameters: $\mathscr{X} = \mathscr{W} = \mathbf{R}^n$.
- Discrete labels: $\mathscr{Y} = \{-1, +1\}$.
- Real predictions: $\mathscr{Q} = \mathbf{R}$.
- Linear hypothesis class: $\mathscr{H} = \{ \langle \boldsymbol{w}, \boldsymbol{x} \rangle : \boldsymbol{w} \in \mathscr{W} \}$.
- Logistic loss: $\ell_{\text{logistic}}(h(\boldsymbol{x}), y) = \log(1 + \exp(-y \cdot h(\boldsymbol{x})))$.
- Gradient: $\nabla J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^m \frac{-y_i \boldsymbol{x}_i}{1 + \exp(y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle)}$.
- Logistic function: $z \mapsto 1/(1 + \exp(-z))$.

Empirical Risk Minimization

Given a collection of input features and labels $\{ (\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_m, y_m) \}$, and a hypothesis class $\mathscr{H}$ the goal of any machine learning algorithm is to find parameters $\boldsymbol{w}$ which minimize the *empirical risk* defined as

$$\underset{\boldsymbol{w} \in \mathscr{W}}{\text{minimize}} \quad J(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^m \ell(h(\boldsymbol{x}_i), y_i) + \lambda \Omega(\boldsymbol{w})$$

Bias-Variance Decomposition

$$\begin{aligned} \mathbf{E}\big[(Y - h(X))^2\big] &= \mathbf{E}\big[\mathbf{E}[h(X)] - h^\star(X)\big]^2 && (\text{bias}^2) \\ &+ \mathbf{E}\big[(h(X) - \mathbf{E}[h(X)])^2\big] && (\text{variance}) \\ &+ \mathbf{V}[\epsilon] && (\text{noise}). \end{aligned}$$

High Variance

symptoms: training error is much lower than generalization error

remedies: add more data, reduce model complexity, bagging

High Bias

symptoms: training error is higher than desired

remedies: increase model complexity, add features, kernels, boosting

Kernelization

1. Rewrite the model h and the optimization algorithm such that *all* inputs $\boldsymbol{x} \in \mathscr{X}$ are only appear in inner products with other inputs $\boldsymbol{z} \in \mathscr{X}$. That is $\langle \boldsymbol{x}, \boldsymbol{z} \rangle$ or $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z}) \rangle$.
2. Choose or define a kernel function k and replace the inner products by the kernel map, *i.e.* substitute $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z}) \rangle$ with $k(\boldsymbol{x}, \boldsymbol{z})$.

Kernel Ridge Regression: $J(\boldsymbol{\alpha}) = \frac{1}{m}\|K\boldsymbol{\alpha} - \boldsymbol{y}\|^2 + \lambda \boldsymbol{\alpha}^\mathsf{T} K \boldsymbol{\alpha}$ with analytical solution $\boldsymbol{\alpha}^\star = (K + m\lambda I)^{-1}\boldsymbol{y}$.

Kernel Logistic Regression:

$$J(\boldsymbol{\alpha}) = \frac{1}{m}\sum_{i=1}^{m} \log(1 + \exp(-y \cdot \sum_{j=1}^{m} \alpha_j k(\boldsymbol{x}_j, \boldsymbol{x}_i)))$$

has no analytical solution and requires partial derivatives

$$\frac{\partial}{\partial \alpha_j} J(\boldsymbol{\alpha}) = \frac{1}{m}\sum_{i=1}^{m} \frac{-y_i \cdot k(\boldsymbol{x}_i, \boldsymbol{x}_j)}{1 + \exp(y_i \cdot h_{\boldsymbol{\alpha}}(\boldsymbol{x}_i))}.$$

Perceptron

The perceptron learns the hyperplane $\{ \langle \boldsymbol{w}, \boldsymbol{x} \rangle = 0 \}$ by minimizing $J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} \max\{0, -y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle\}$ with stochastic gradient descent:

$$\tilde{\nabla} J(\boldsymbol{w}) = \begin{cases} -y_i \boldsymbol{x}_i & \text{if } y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \le 0 \\ 0 & \text{otherwise} \end{cases}$$

Support Vector Machine

The SVM solves a constraint optimization problem:

$$\begin{aligned} \underset{\boldsymbol{w}, b, \boldsymbol{\xi}}{\text{minimize}} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{m} \xi_i \\ \text{subject to} \quad & y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i) \rangle + b) \ge 1 - \xi_i \quad i = 1, \dots, m, \\ & \xi_i \ge 0 \qquad i = 1, \dots, m \end{aligned}$$

and has a kernelized variant, where most α_i are zero:

$$\begin{aligned} \underset{\boldsymbol{\alpha} \in \mathbf{R}^m}{\text{maximize}} \quad & \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j) \rangle \\ \text{subject to} \quad & 0 \le \alpha_i \le C \quad i = 1, \dots, m, \\ & \sum_{i=1}^{m} \alpha_i y_i = 0 \end{aligned}$$

Locally Weighted Regression

Locally weighted regression weighs each training instance $(\boldsymbol{x}_i, y_i)$ with $k(\boldsymbol{x}_i, \hat{\boldsymbol{x}})$:
$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} k(\boldsymbol{x}_i, \hat{\boldsymbol{x}}) \cdot \ell(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle, y_i)$

k-Nearest Neighbors

The k-nearest neighbors algorithm can be used for regression and classification.

$$h(\boldsymbol{x}) = \frac{1}{k}\sum_{i=1}^{m} y_i \delta_{\text{NN}}(\boldsymbol{x}_i, \boldsymbol{x}),$$

$\delta_{\text{NN}}(\boldsymbol{x}_i, \hat{\boldsymbol{x}})$ is a function which is one, if $\hat{\boldsymbol{x}}$ is a k-NN of $\boldsymbol{x}_i$ and zero otherwise.

Decision Trees

The decision tree model is

$$h(\boldsymbol{x}) = \sum_{p=1}^{P} \begin{cases} \hat{y}_p & \text{if } \boldsymbol{x} \in S_p, \\ 0 & \text{otherwise} \end{cases}$$

essentially predicting $\hat{y}_p$ if $\boldsymbol{x}$ falls into partition S_p. The optimal prediction $\hat{y}_p$ for any given loss ℓ can be obtained by solving

$$\hat{y}_p = \underset{\hat{y}_p \in \mathscr{Y}}{\arg\min} \sum_{\boldsymbol{x}_i \in S_p} \ell(\hat{y}_p, y_i).$$

The partitions are found by greedy methods which incrementally build the tree, minimizing the objective function one decision tree node at a time.

Bagging

Bagging uses a model average $H(\boldsymbol{x}) = \frac{1}{T}\sum_{t=1}^{T} h_t(\boldsymbol{x})$ to drive the ensemble closer to the model expectation $\mathbf{E}[h(X)]$ in order to reduce the variance.

Boosting

A boosting ensemble drastically improves the accuracy of weak learners, turning them into strong learning algorithms. Boosting perform gradient descent in *functional space.*

AnyBoost

```
H ← 0
for t ← 1, ..., T do
    h ← minimize_{h∈ℋ}  ‖h + ∇_H J‖²
    if ⟨h, ∇_H J⟩ ≥ 0 then
        break
    H ← H + ηh
return ensemble H
```

Least Squares Boosting

Least squares boosting is a regression algorithm that uses the square loss. The projection of the negative gradient onto $\mathscr{H}$ is.

$$\underset{h \in \mathscr{H}}{\text{minimize}} \quad \|h - (-\nabla_H J)\|^2 = \frac{1}{m}\sum_{i=1}^{m} (h(\boldsymbol{x}_i) - r_i)^2.$$

AdaBoost

AdaBoost is a classification algorithm that uses the exponential loss. The projection of the negative gradient onto $\mathscr{H}$ is.

$$\underset{h \in \mathscr{H}}{\text{minimize}} \quad \|h - (-\nabla_H J)\|^2 = \sum_{(\boldsymbol{x}_i, y_i) \in F} w_i = \epsilon$$

AdaBoost ican find the optimal learning rate η for each gradient descent step in *closed form* to be $\eta = \frac{1}{2} \ln \frac{1-\epsilon}{\epsilon}$.

$H \leftarrow 0$ (the constant 0 function)
$w_i \leftarrow 1/m$
$J(h) = \sum_{i=1}^{m} w_i \ell_{0/1}(h(\boldsymbol{x}_i), y_i)$
for $t \leftarrow 1, \dots, T$ *do*
    $h \leftarrow \text{minimize}_{h \in \mathscr{H}} J(h)$
    $\epsilon \leftarrow J(h)$
    if $\epsilon \geq 1/2$ *then*
        break
    $\eta \leftarrow \frac{1}{2} \ln \frac{1-\epsilon}{\epsilon}$
    $w_i \leftarrow w_i \exp(-y_i \eta h(\boldsymbol{x}_i))$
    $w_i \leftarrow w_i / \sum_{i=1}^{m} w_i$
    $H \leftarrow H + \eta h$
return ensemble H

If the weak learner returns a model with error $\epsilon \leq 1/2 - \gamma$, then, after T iterations $J(H) \leq (1 - 4\gamma^2)^{\frac{T}{2}}$.

Neural Networks

A neural network is a feature function $\phi(\boldsymbol{x}) = \sigma(U\boldsymbol{x} + \boldsymbol{c})$ which is *automatically* learned for a given task or domain. The feature functions can be layered

$$\begin{aligned} \boldsymbol{z}_1 &= \boldsymbol{x} \\ \boldsymbol{z}_{i+1} &= \sigma_i(U_i \boldsymbol{z}_i + \boldsymbol{c}_i), \quad i = 1, \dots, k-1 \\ h(\boldsymbol{x}) &= U_k \boldsymbol{z}_k + \boldsymbol{c}_k \end{aligned}$$

to obtain a k-layer deep neural network.

Neural networks are trained with *automatic differentiation*, a family of techniques for an efficient and accurate evaluation derivatives of numeric functions expressed as *computational graphs*. It requires Jacobian-vector products:

$$\frac{\partial \ell}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{z}^\top}{\partial \boldsymbol{x}} \frac{\partial \ell}{\partial \boldsymbol{z}},$$

The Jacobian-vector products for $Z = UX$ are

$$\begin{aligned} \frac{\partial \ell}{\partial X} &= \frac{\partial Z^\top}{\partial X} \frac{\partial \ell}{\partial Z} = U^\top \frac{\partial \ell}{\partial Z} \qquad \text{and} \\ \frac{\partial \ell}{\partial U} &= \frac{\partial Z^\top}{\partial U} \frac{\partial \ell}{\partial Z} = \frac{\partial \ell}{\partial Z} X^\top \end{aligned}$$

where $\partial \ell / \partial Z$ is the incoming gradient/Jacobian matrix.

Index

ℓ_1 regularization, 41
ℓ_2 regularization, 39

accuracy, 24
activation
    ReLU, 86
    sigmoid, 85
    tanh, 86
activation function, 85
AdaBoost, 81
Adagrad, 90
Adam, 90
AnyBoost, 80
approximate nearest neighbor, 67
area under the curve, 25
attention, 105
attention layer, 106
AUC, 25
automatic differentiation, 91

backpropagation, 91
bagging, 75
    neural networks, 88
base learner, 75
bias, 9
binary classification, 21
boosting
    AdaBoost, 81
    least squares, 80
bootstrap, 76
bootstrap aggregating, 75

characteristic length-scales, 50
classification
    AdaBoost, 81
    AnyBoost, 80
    binary, 21
    convolutional neural network, 96
    decision boundary, 19
    decision tree, 71
    kernel logistic regression, 51
    logistic regression, 18
    multiclass, 21
    nearest neighbors, 64
    softmax, 23
    support vector machine, 20
CNN, 96
computation graph, 91
confusion matrix, 24
constant feature, 9
convolutional layer, 97
convolutional neural networks, 96
cross entropy, 22
cross validation error, 32

data augmentation, 99
decision boundary, 19
decision tree, 71
decision tree stump, 78
deep neural network, 86
design matrix, 12
dropout, 94

early stopping, 42, 94
empirical risk, 8
empirical risk minimization, 8
ensemble, 75
    bagging, 75
    random forest, 77
error rate, 24

false negatives, 24
false positives, 24
feature function, 28
forward pass, 87

generalization error, 30
generalize, 30
gradient, 10

hard margin, 58
holdout set, 31

identifiable, 2
intrinsic dimensionality, 66

Jacobian, 93
Jacobian-vector product, 93

k-fold cross validation, 32
k-nearest neighbors, 63
kernel
    length-scale, 48
    logistic regression, 51
    ridge regression, 50
    squared exponential, 48
kernel functions, 47
kernel matrix, 47
kernel ridge regression, 50
kernelization, 49
kernelized logistic regression, 51
k-NN, 63

lasso regression, 41
learning curve, 35
least squares, 9
least squares boosting, 80
leave-one-out cross validation, 33
length-scale, 48
linearly separable, 15
locally weighted regression, 62
logistic regression, 18
loss
    absolute, 13
    exponential, 17
    hinge, 17
    logistic, 17
    square, 9
    zero-one, 16
LWR, 62

machine learning, 8
maximum margin classifier, 53
mean absolute error, 13
minibatch, 88
Minkowski distance, 64
momentum update, 89
multi-layer perceptron, 85
multiclass classification, 21

Nesterov's algorithm, 90
neural network, 85
    convolutional layer, 97
    filter, 97
    pooling layer, 98
    pretrain, 101
nonparameteric, 62
normal equations, 12

one versus all, 21
one versus one, 22
overfitting, 31

padding
    zero padding, 98
partial derivatives, 10
pooling layer, 98
pretrain, 101

random forest, 77
recurrent neural network, 103
regression
    AnyBoost, 80
    decision tree, 71
    kernel ridge, 50
    lasso, 41
    least squares, 9
    least squares boosting, 80
    locally weighted, 62
    nearest neighbors, 63
    ridge, 39
    robust, 13
regularization, 38
    dropout, 94
    early stopping, 94
    weight decay, 87
regularization parameter, 38
regularized loss minimization, 38
ReLU activation, 86
representer theorem, 45
reverse accumulation mode, 92
ridge regression, 39
risk, 8
RNN, 103
robust regression, 13

self-attention, 106
separating hyperplane, 15
sigmoid activation, 85
slack variables, 58
soft margin, 58
softmax, 23
squared exponential kernel, 48
stochastic gradient descent, 88
stride, 98
support vector machine, 20

support vectors, 56
SVM, 20

tanh activation, 86
test set, 31
Tikhonov regularization, 39
training data, 8
training error, 30
training set, 31
transformer networks, 106
translation invariance, 96
true negatives, 24
true positives, 24

underfitting, 30

validation set, 31

weak learner, 78
weight decay, 87

zero padding, 98
zero-one loss, 16