Machine Learning for Data Science 1

(lecture notes, only for internal use)

Blaž Zupan, Erik Štrumbelj

May 17, 2021

Contents

Chapter 1

Introduction

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

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

1.1 Purpose of Machine Learning

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

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

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

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

1.2 Types of Machine Learning

Supervised Learning

Often, we are only interested in how a subset of variables is generated, while the remaining variables are used to explain the behavior of the variables of interest: $\{y_i, \boldsymbol{x}_i\}_{i=1}^n$. This is *supervised learning*, also known as *predictive learning* or predictions. Here, y is referred to as *response variable*, and $\boldsymbol{x}$ represents a vector of *features*. Depending on a branch of science that deals with machine learning, the dependent variable may also be referred to as a target variable, dependent variable (statistics), or label or class variable (machine learning). The independent variables are often referred to as covariates, independent variables, and predictors (statistics), or features and attributes (machine learning). Supervised learning starts with the training data, which includes n pairs of instantiations of independent and dependent variables. The goal is to learn about $p(y|\boldsymbol{x})$ so that we can make predictions for future or unobserved values of independent variable y for any combination of dependent variables $\boldsymbol{x}$ (see Table 1.1).

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

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

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

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

Unsupervised learning

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

A common approach to understanding the distribution $p(y)$ is to explain it with a smaller number of factors θ, that is, to learn $p(y|\theta)$, effectively projecting the data into a lower-dimensional space. We refer to such procedure as *dimensionality reduction*. An extreme example of dimensionality reduction is *clustering*, when we try to explain $p(y)$ with a single nominal factor (see Fig. 1.1). In essence, we are trying to group, or cluster, observations.

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

Reinforcement learning

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

1.3 Models and Learning

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

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

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

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

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

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

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

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

Parametric and Nonparametric Models

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

> **Example: 1-nearest neighbor model - a nonparametric model**
> $\mathcal{H}$ = { all functions f that can be expressed with a set of points (data instances) and the rule that $f(\boldsymbol{x}) = y_i$ of point x nearest to x_i, according to a chosen distance metric} (see Fig. 1.3)

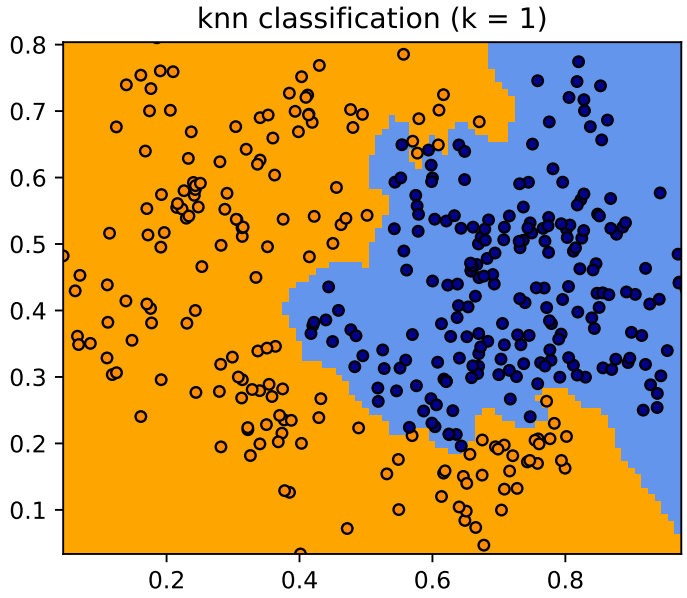


Figure 1.2: Decision boundary of a 1-nearest neighbor model trained on a two-featured binary classification data set with 380 data instances (170 in one class and 210 in the other) as shown on a figure. The decision boundary is complex and may substantially change with any addition on removal of the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

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

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

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

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

1.4 Challenges of Applied Machine Learning

Model Evaluation and Selection

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

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

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

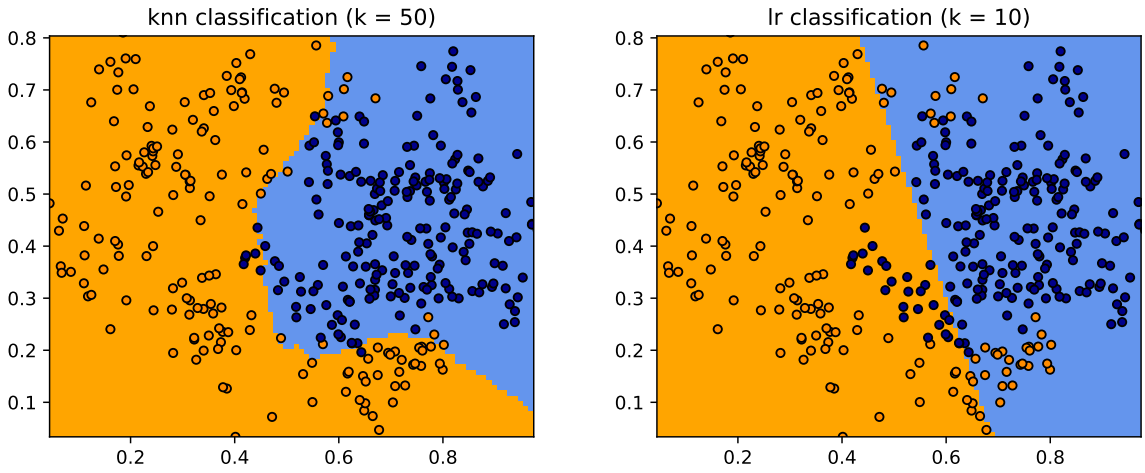


Figure 1.3: Decision boundary on a two-featured binary classification data set as inferred by the nearest neighbor algorithm with K=50 and by logistic regression. Which model would perform better on new data?

Overfitting

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

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

Model Complexity and Effective Number of Parameters

More complex models are more likely to overfit (see Figs. 1.6 and 1.7), but the "right" complexity of the model may depend on the amount of data we have (see Fig. 1.8). In parametric models, especially linear models, the model's complexity easily be measured in terms of

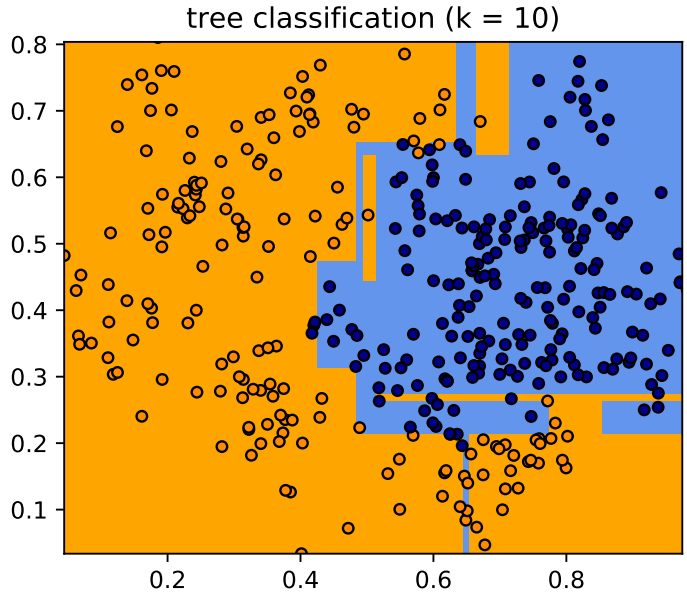


Figure 1.4: Classification trees would often overfit the training data. Figure shows decision boundary of a tree where the allowed maximum tree depth was 10. The decision boundary is complex and often covers single-case exceptions.

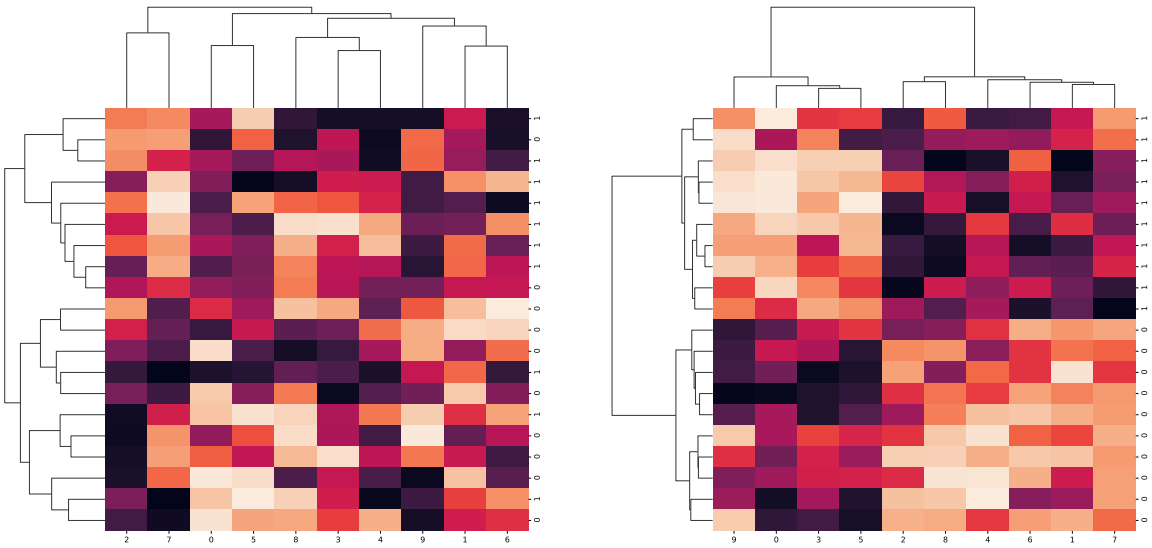


Figure 1.5: Co-clustering of a random data set with 20 instances and 10 features (left), and co-clustering of a similar data set with 10000 features, of which 10 features were selected that best correlate with a binary label (right). Notice a clear pattern of colors and shades in the right heatmap, which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

the number of parameters (or degrees of freedom). For non-parametric models, the theory is more complex (*e.g.*, Vapnik–Chervonenkis dimension) and introduces the concept of the *effective number of parameters*. Typically, non-parametric models have a higher number of effective parameters and are thus able to better fit the data but also more prone to overfitting. But they are more difficult to interpret.

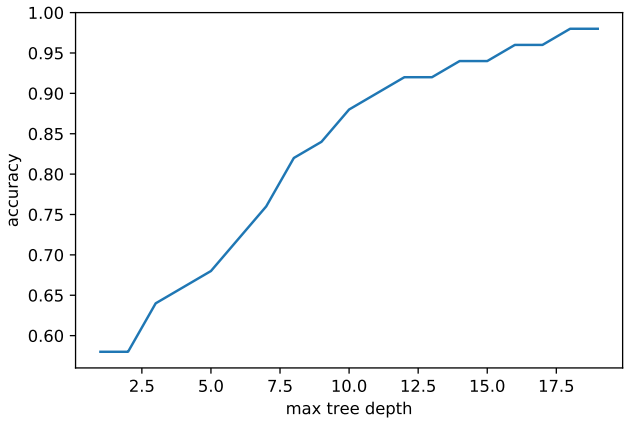


Figure 1.6: A classification tree accuracy on a random class-balanced binary classification data set with one feature and 50 data instances. Trees were grown to a specified maximal depth. More complex trees better fit the training data.

Practical Utility of Machine Learning Models

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

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

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

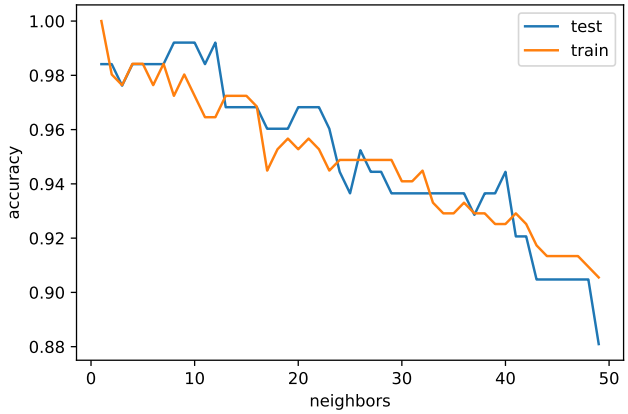


Figure 1.7: A training and the test-set tradeoff for k-nearest neighbor model. On the training data, the accuracy falls with raising k, while on the test data set, the accuracy peaks at around $k = 10$. Hyper-parameter estimation is one of the key issues when selecting the most appropriate model.

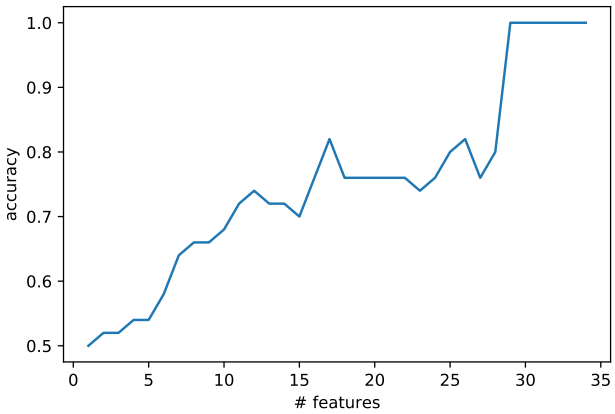


Figure 1.8: Logistic regression is more robust to overfitting than classification trees but succumbs as well when given a sufficient number of features. A graph shows a training error on a random 50-instance binary classification data set when adding up to 35 features.

- *Interpretability*, which often refers to the question if the model is readable or can be converted to a readable format. And if it is readable, is its interpretation easy (e.g., just a few if-then-rules) or impossible (e.g., a long list of rules or a large classification tree).
- *Explainability*, often confused with interpretability, places a model within a context of a problem domain and asks a question did we gain any new knowledge. To achieve explainability, one would often need to combine the interpretation of the model with extra formalized knowledge about the domain (*e.g.*, feature groups, ontologies, rules, and similar).

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

Curse of Dimensionality

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

The curse of dimensionality may also inhibit or even cripple some machine learning methods. For instance, k-nearest neighbors may work well on two-dimensional data, but as soon as the number of dimensions increases, to a few more dimensions, the algorithm fails. To illustrate this point, consider embedding a small d-dimensional cube of side s inside a larger unit cube (Fig. 1.9). Let the data be uniformly distributed within the unit cube. Suppose we estimate the density of a class labels around a test point $\boldsymbol{x}$ by growing a smaller hyper-cue until it contains a desired fraction f of the data points. The expected length of this cube will be $s(f,d) = f^{1/d}$. Say, with $d = 10$ and to base our estimate on 10% of the data, the length of the smaller cube would need to be $s = 0.8$. The approach, despite the name "nearest neighbor" is no longer very local, as even with the modest feature sizes, it relies on data points that are far away. Even with 1% coverage, the size of the small cube needs to be substantial, as $s(0.01, 10) = 0.63$. With a number of features growing, we quickly have to start taking into account points that are not close or risk increasing variance.

References

Freund, Y. and R.E. Shapire (1997). "A decision-theoretic generalization of on-line learning and an application to boosting". In: *Journal of Computer and System Sciences* 55, pp. 119–139.

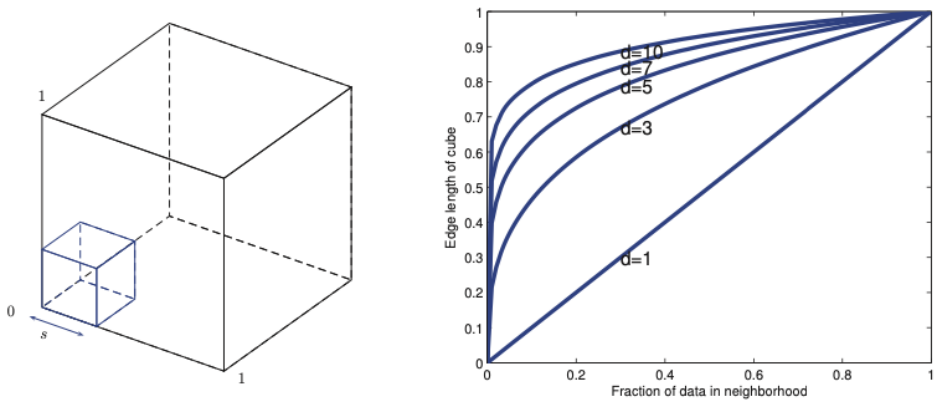


Figure 1.9: We embed a small cube within a unit cube (left) and assess the length of the edge of a small cube to cover a fraction of uniformly spread data. Graphs borrowed from **2012-Murphy**.

Chapter 2

Trees and Forests

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

2.1 Classification and Regression Trees (CART)

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

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

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

Learning kernel functions is on option, but is computationally expensive and requires a lot of data. An alternative approach is to forget about kernels, and instead infer useful features

$\boldsymbol{\phi}(\boldsymbol{x})$ directly from the training data. This is an approach used by adaptive basis function model, which takes the form

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

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

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

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

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

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

Basic Idea

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

The CART Algorithm

Finding the optimal partitioning of the input variable space is in general NP-complete, even if using axis-parallel splits only. That is, it is infeasible to check all possible partitions. In-

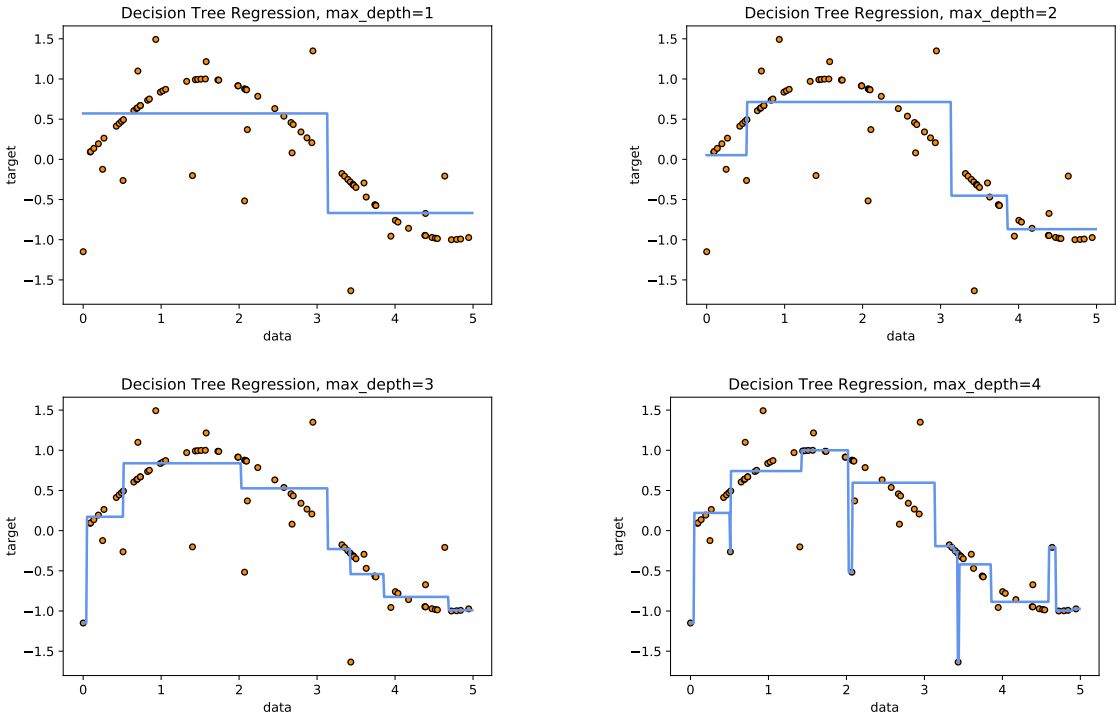


Figure 2.1: Regression trees fitted on data generated by a sine function with some noise. While the tree adapts well to the training data, its ability to overfit the training data is visible already with trees with of maximum depth of 4 (lower right).

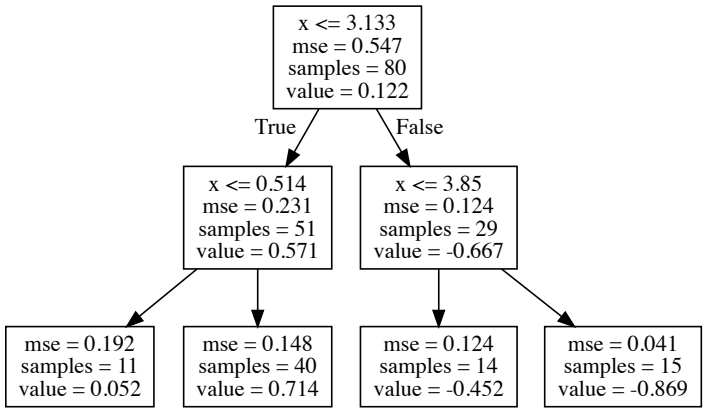


Figure 2.2: A regression tree with maximum depth of 2 from the data from Fig. 2.1.

stead, we will consider a greedy algorithm (CART) that uses binary recursive partitioning of the input space, at each step choosing the best possible split (according to some pre-selected criterion). Notice that this algorithm does not use any look-ahead, and while there are published studies of such algorithms, they have found no use in practice. Algorithm 1 shows the pseudocode of a CART algorithm.

Algorithm 1 CART

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

The CART algorithm uses several functions that require explanation:

- *createNode()*: This function creates an object that represents a tree node, which essentially stores the criterion that splits the data in a node and a possible reference to the data instances that are pertinent to the node. If the procedure finds a suitable node split, the node stores the information on its to siblings. Note that, as introduced above, the CART algorithm would construct binary trees.
- *split()*: The assumption here is that features are numerical or at least ordinal. We order every feature based on possible splits (based on unique values in the data, so we have a finite number of possible splits). And then, we go through all possible feature-split combinations to find the one that is optimal according to our splitting criterion – the one that minimizes the sum of the cost of the left and right subtrees. Below we discuss possible splitting criteria.
- *stoppingCriterionMet()*: The stopping condition, also referred to as *pre-prunning* of the trees, can be one or more of the following:
  - The partition is sufficiently homogeneous/pure. In particular, there is no point in splitting further if we have perfect homogeneity (all observations have the same value).
  - The gain Δ of splitting the data set in the current node (relative to stopping criterion) is below some pre-determined threshold, where

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

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

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

Choice of the Splitting Criterion

At each internal node, the inference method for the trees splits the training data set $\mathcal{D}$ pertinent to the node to maximize some splitting criterion. The split uses a single feature from the training data set and forms a condition on the value of this feature that evaluates to true or false. The condition splits the data $\mathcal{D}$ to two data sets, each pertinent to one of the two siblings of the node. The result is a binary tree. Notice that we could use other, non-binary, splitting mechanisms, but they would lead to over-fragmentation of the data, increase the variance, and lead to increased overfitting.

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

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

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

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

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

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

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

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

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

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

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

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

Discussion

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

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

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

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

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

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

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

treatment of missing values. Suppose our data has some missing predictor values in some or all of the variables. We might discard any observation with some missing values, but this could lead to severe depletion of the training set. Alternatively, we might try to fill in (impute) the missing values, with say the mean of that predictor over the non-missing observations. For tree-based models, there are two better approaches. The first

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

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

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

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

2.2 Bagging

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

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

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

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

Why Does Bagging Work?

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

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

One implication of the above is that models, where all points have the same or similar leverage, would not benefit from bagging. Similarly, models, where a single point has very little effect on the prediction, would also not benefit from bagging (robust models such as regularized regression or models that already contain some sort of bagging, such as random forests, which we discuss below). Therefore, high-variance models, such as non-pruned trees,

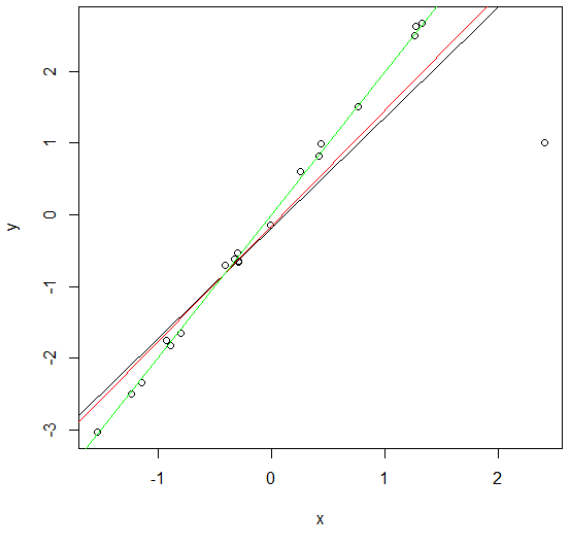
is where we would expect the most benefit.

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

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

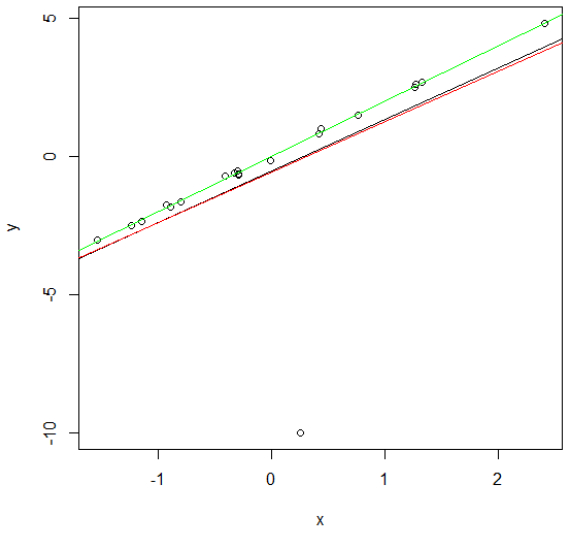
Example: Bagging on outliers, #1

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



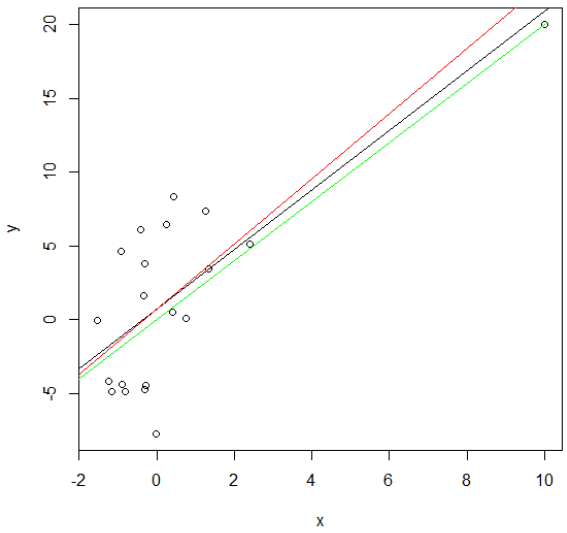
Example: Bagging on outliers, #2

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



Example: Bagging on outliers, #3

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



2.3 Random Forests

Random forests (**Brieman2001**) extend the idea of bagging but aim to develop even more decorrelated trees than those from bootstrap samples. The approach develops a possibly large

collection of trees $\{T_b\}_1^B$, where each tree is inferred from a bootstrap sample of the training data set. To additionally diversify the trees, the features on which to split each internal node of the tree are selected from a random sample of p variables. This is different from the normal growth of the trees which instead considers the entire set of predictors. Here, p is a user-specified parameter. To make a prediction of a new data point $\boldsymbol{x}$, we either average the predictions of individual trees in a case of regression,

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

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

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

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

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

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

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

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

Random forests do remarkably well in terms of accuracy, with very little or no tuning required (**Fernandez-Delgado2014**). Just like trees, they require almost no data preprocessing, can treat both continuous and discrete features, and can easily handle missing values. The inference of trees is fast and can be applied to any reasonably sized data set. With these

characteristics, random forests are a great baseline, that is, provide accuracies that need to be surpassed by more advanced approaches.

Out-of-Bag Estimates

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

Estimate of Feature Importance

One of the deficiencies of random forests is their overall complexity. If we agree that the trees are models that can be read and interpreted, we lose this ability with the forest simply because of the large collection of trees. With forests, interpretability is lost. To remedy the loss of interpretability, the author of the forests, **Brieman2001**, proposes to provide estimates of the importance of features in the forests using out-of-bag estimates. The procedure randomly permutes the value of a selected feature and estimates the out-of-bag error. The decrease of accuracy caused by random permutation now provides an estimate of the feature's importance. Notice that estimates obtained in this way can be substantially different from univariate estimates of the correlation between a feature and a class variable, taking into account possible feature interactions discovered by the trees in the forest.

Chapter 3

Linear and Logistic Regression

We start the chapter with a probabilistic view of linear regression. We first express the likelihood function for the regression and show that its maximization is equal to minimizing the residual sum squares, that is, to minimizing the square loss. We then compute the gradient of the square loss for linear regression, providing means to find the means of inferring parameters of the model from data through gradient descent. We also point to the alternative, closed-form solution. We use a similar approach for logistic regression, again starting with the likelihood function, and finding its gradient to be used in a gradient descent search for parameters that optimize the one-zero loss of the predictor on the training data.

3.1 Maximum Likelihood and Least Squares

Maximum likelihood estimation involves treating the problem as an optimization or search problem. We seek parameters that result in the best fit for the joint probability of the data sample. For a start, consider that we are given a regression (training) data set with pairs of values for dependent and vectors for independent variables, $\{(y_1, \boldsymbol{x}_1), \ldots, (y_N, \boldsymbol{x}_N)\}$. The target variable is continuous, and its estimate is given by

$$\hat{y}(\boldsymbol{x}) = f_{\boldsymbol{\beta}}(\boldsymbol{x}) = \beta_0 + \sum_{m=1}^{M} \beta_m \boldsymbol{x}_m. \tag{3.1}$$

We assume that the target variable y is given by a deterministic function $y(\boldsymbol{x}, \boldsymbol{\beta})$ with additive Gaussian noise, so that

$$y = y(\boldsymbol{x}, \boldsymbol{\beta}) + \epsilon. \tag{3.2}$$

We can then write

$$p(y|\boldsymbol{x}, \boldsymbol{\beta}, \sigma^2) = \mathcal{N}(y|y(\boldsymbol{x}, \boldsymbol{\beta}), \sigma^2), \tag{3.3}$$

which is to say that y is normally distributed around its (true) value provided by the deterministic function and with a variance of σ^2. That is, the distribution of noise in the data is $\epsilon = \mathcal{N}(0, \sigma^2)$. The probability of the approximation error, $e = y - \hat{y}$ is thus

$$p(e) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{e^2}{2\sigma^2}\right) \tag{3.4}$$

We are now ready to assign the probability for the estimated value of the target variable for the i-th data instances in the training data set:

$$p(y_i|\boldsymbol{x}_i, \boldsymbol{\beta}) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y_i - \hat{y}(\boldsymbol{x}))^2}{2\sigma^2}\right) \tag{3.5}$$

We can now compute the probability of observing the training data by our model, which we exclusively define through a set of parameters $\boldsymbol{\beta}$. We will assume that the data instances in the training data set are independent. We denote this probability with $L(\boldsymbol{beta})$, and write

$$\begin{aligned} L(\boldsymbol{\beta}) &= L(\boldsymbol{\beta}; \boldsymbol{X}, \boldsymbol{y}) && (3.6)\\ &= p(\boldsymbol{y}|\boldsymbol{X}; \boldsymbol{\beta}) && (3.7)\\ &= \prod_{i=1}^{N} p(y_i|\boldsymbol{x}_i; \boldsymbol{\beta}) && (3.8)\\ &= \prod_{i=1}^{N} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y_i - \hat{y}(\boldsymbol{x}_i))^2}{2\sigma^2}\right) && (3.9) \end{aligned}$$

We would like to maximize the probability with which we observe the target values y in the training data by our inferred model. In other words, we would like to find the parameters $\boldsymbol{\beta}$ to maximize the likelihood. For practical reasons, we compute the logarithm of the likelihood, and call it the log likelihood,

$$\begin{aligned} \ell(\boldsymbol{\beta}) &= \log L(\boldsymbol{\beta}) && (3.10)\\ &= \log \prod_{i=1}^{N} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y_i - \hat{y}(\boldsymbol{x}_i))^2}{2\sigma^2}\right) && (3.11)\\ &= \sum_{i=1}^{N} \log\left(\frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y_i - \hat{y}(\boldsymbol{x}_i))^2}{2\sigma^2}\right)\right) && (3.12)\\ &= N \log \frac{1}{\sigma\sqrt{2\pi}} - \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \hat{y}(\boldsymbol{x}_i))^2 . && (3.13) \end{aligned}$$

Considering the result, everything else is constant, and the only term that depends on $\boldsymbol{\beta}$ is the sum of squared approximation errors. To maximize the log-likelihood, we need to minimize

the sum of squared errors! That is, to train the model that maximizes the probability of observing the training data, we need to minimize the following criteria function:

$$J(\boldsymbol{\beta}) = \sum_{i=1}^{N} (y_i - \hat{y}(\boldsymbol{x}_i))^2 \tag{3.14}$$

$$= \sum_{i=1}^{N} (y_i - \beta^\intercal \boldsymbol{x}_i)^2 \tag{3.15}$$

3.2 Gradient Descent for Linear Regression

We are given a set of training data instances for which we would like to infer a linear regression model. We define the model with a set of parameters $\boldsymbol{\beta}$, so that $\hat{y}_i = \boldsymbol{\beta}_0 + \boldsymbol{\beta}_1 \boldsymbol{x}_{i1} + \ldots + \boldsymbol{\beta}_M \boldsymbol{x}_{iM}$, where $\boldsymbol{\beta}_j$ is the weight of the j-th independent variable and $\boldsymbol{\beta}_0$ is an intercept. Using gradient descent, we can start the search of the optimal set of parameters in some initial point, say, $\boldsymbol{\beta} = \vec{0}$. Our goal is to find $\boldsymbol{\beta}$ that minimize the criteria function $J(\boldsymbol{\beta})$. We do so by changing each $\boldsymbol{\beta}_j$ so that to make $J(\boldsymbol{\beta})$ smaller, that is, in the direction oposite to the partial derivative of the criteria function

$$\boldsymbol{\beta}_i \leftarrow \boldsymbol{\beta}_i - \alpha \frac{\partial}{\partial \boldsymbol{\beta}_i} J(\boldsymbol{\beta}), \tag{3.16}$$

where α is a learning rate whose value will determine the speed to which we proceed to the optimal value of the parameters. If α is too large, there is a chance the procedure will miss optimal point and get out of bounds. When α is too small, the convergence is slow.

Let us now compute the partial derivate of our criteria function for $\boldsymbol{\beta}_i$. Notice that these partial derivatives form a vector, or the gradient of the criteria function. For now, we will compute the gradient taking into account only one data instance from the training set, $(y, \boldsymbol{x})$.

$$\frac{\partial}{\partial \boldsymbol{\beta}_i} J(\boldsymbol{\beta}) = \frac{\partial}{\partial \boldsymbol{\beta}_i} \left(f_\beta(x) - y \right)^2 \tag{3.17}$$

$$= 2(f_\beta(x) - y) \frac{\partial}{\partial \boldsymbol{\beta}_i} \left(f_\beta(\boldsymbol{x}) - y \right) \tag{3.18}$$

$$= 2(f_\beta(x) - y) \frac{\partial}{\partial \boldsymbol{\beta}_i} (\boldsymbol{\beta}_0 x_0 + \ldots + \boldsymbol{\beta}_i \boldsymbol{x}_i + \ldots \boldsymbol{\beta}_n x_n - y) \tag{3.19}$$

$$= 2(f_\beta(x) - y)\boldsymbol{x}_i \tag{3.20}$$

Iterative correction of $\boldsymbol{\beta}_i$ will be therefore

$$\boldsymbol{\beta}_i \leftarrow \boldsymbol{\beta}_i - \frac{\alpha}{N} (f_\beta(\boldsymbol{x}) - y)\boldsymbol{x}_i \tag{3.21}$$

and considering all the data instances in the training set

$$\boldsymbol{\beta}_i \leftarrow \boldsymbol{\beta}_i - \frac{\alpha}{N}\sum_{j=1}^{N}(f_{\boldsymbol{\beta}}(\boldsymbol{x}_j) - y_j)x_{ji} \tag{3.22}$$

In the linear regression or least squares approach described above, the criteria function $J(\boldsymbol{\beta})$ is a quadratic function with a single minimum, so we do not need to fear that the optimization would stop at some local minimum. However, as we have already written above, at large values of α the gradient descent may overshoot and miss the minimum and start moving further and further away from it. It helps, of course, to reduce α to a value at which the optimization is stable and converges to optimal value of parameters $\boldsymbol{\beta}$.

3.3 Closed-Form Solution for Linear Regression

Let us rewrite the criteria function $J(\boldsymbol{\beta})$ for linear regression in a matrix-vector form,

$$J(\boldsymbol{\beta}) = \sum_{i=1}^{N}(y_i - \hat{y}(\boldsymbol{x}_i))^2 \tag{3.23}$$

$$= (\boldsymbol{y} - \boldsymbol{X\beta})^\intercal(\boldsymbol{y} - X\boldsymbol{\beta}) \tag{3.24}$$

We are looking for the parameters $\boldsymbol{\beta}$ which minimize the value of the criteria function, that is, the value of the parameters where the gradient is $\mathbf{0}$. Let us first compute the gradient:

$$\frac{\partial J(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -2\boldsymbol{X}^\intercal(\boldsymbol{y} - \boldsymbol{X\beta}). \tag{3.25}$$

Equating the gradient with $\mathbf{0}$, we obtain

$$\boldsymbol{X}^\intercal(\boldsymbol{y} - \boldsymbol{X\beta}) = \mathbf{0} \tag{3.26}$$

$$\boldsymbol{X}^\intercal - \boldsymbol{X}^\intercal\boldsymbol{X\beta} = \mathbf{0} \tag{3.27}$$

$$\boldsymbol{X}^\intercal X\boldsymbol{\beta} = \boldsymbol{X}^\intercal\boldsymbol{y} \tag{3.28}$$

$$\boldsymbol{\beta} = (\boldsymbol{X}^\intercal\boldsymbol{X})^{-1}\boldsymbol{X}^\intercal\boldsymbol{y} \tag{3.29}$$

Above is a closed form solution to the computation of parameters $\boldsymbol{\beta}$ of a linear regression model. Note that from here we can also compute the approximations of the values for indepent variable:

$$\hat{\boldsymbol{y}} = X\boldsymbol{\beta} \tag{3.30}$$

$$= \boldsymbol{X}(\boldsymbol{X}^\intercal\boldsymbol{X})^{-1}\boldsymbol{X}^\intercal\boldsymbol{y} \tag{3.31}$$

3.4 Linear Regression as a Classifier

In theory, linear regression could also be used for classification. Consider the following example and the data from Table 3.1. The data includes the mesaurements of body temperature and the record of the state of the visitor of doctor's office. We would like to infer the model that predicts the state (sick or healthy) from the body temperature. We encode the class variable with a number, 0 for healthy and 1 for sick, and present the data in a graph (Fig. 3.1).

Table 3.1: Body temperature and state of the visitor at doctor's office, where we state the class with a categorical variable and encode it with a number, a class variable y.

body temperature	state	y
36,5	healthy	0
36,6	healthy	0
36,8	healthy	0
36,9	sick	1
37,0	healthy	0
37,2	sick	1
37,5	sick	1
37,6	sick	1
39,5	sick	1

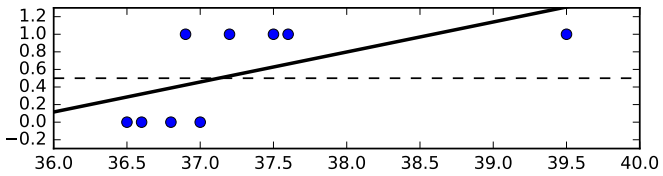


Figure 3.1: An attempt to develop a classifier with a linear regression.

In the graphically presented data (Fig. 3.1) we dare to fit the linear function $\hat{y} = f(x)$. We first notice that the range of the function $f(x)$ is inappropriate, since it ranges from $-\infty$ to ∞. In the interval of body temperatures around the point 37°, the function has a value between

0 and 1. The question arises how to interpret the estimated value, or how to convert it into probability. Recall that each probability function returns values between 0 and 1, and our linear function is limited to this interval only in a certain range of values of the input attribute. Additional problem is with outliers, or visitors with extreme value of the temperature. If we were to add another case of a patient with a very high temperature to the data, our function would change considerably and shift to the right. We need to instead develop a function that would output the probabilities, soften the output at left and right edges of the data, and then pose the problem in probabilistic and formal terms.

3.5 Logistic Function

Before we continue with a formal introduction of logistic regression, let us introduce a logistic function, a function that converts a real value into an interval from 0 to 1:

$$g(z) = \frac{1}{1 + e^{-z}} \tag{3.32}$$

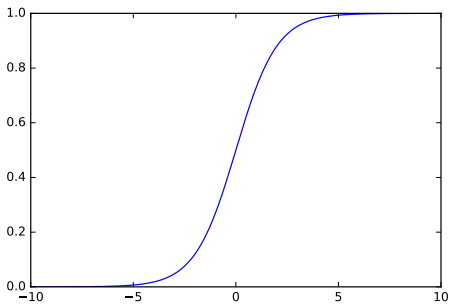


Figure 3.2: Logistic function.

The logistic function (Fig. 3.2) is continuous, monotonic; with $z \to -\infty$ it converges to 0

and with $z \to \infty$ it converges to 1. It derivative exists for all values of its parameter:

$$\begin{aligned}
\frac{dg(z)}{dz} &= \frac{d}{dz}\frac{1}{1+e^{-z}} \\
&= \frac{1}{(1+e^{-z})^2}e^{-z} \\
&= \frac{1}{1+e^{-z}}\frac{e^{-z}}{1+e^{-z}} \\
&= \frac{1}{1+e^{-z}}\frac{1+e^{-z}-1}{1+e^{-z}} \\
&= \frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right) \\
&= g(z)[1-g(z)]
\end{aligned} \tag{3.33}$$

3.6 Likelihood for Logistic Regression

Following is an equation for a logistic regression model. In the core, this is a linear model, that is, a weighted sum of the values of the input variables.

$$\begin{aligned}
f_\beta(x) &= g(\boldsymbol{\beta}_0 + \boldsymbol{\beta}_1 x_1 + \boldsymbol{\beta}_2 x_2 + \ldots \boldsymbol{\beta}_M x_M) \\
&= g(\boldsymbol{\beta}^\intercal \boldsymbol{x}) \\
&= \frac{1}{1+e^{-\beta^\intercal x}}
\end{aligned} \tag{3.34}$$

Notice that the linear combination of $\boldsymbol{\beta}$ and values of input variables, when equated to 0, defines the hyperplane in the feature space. Notice also the all the points that lie on the plane have a value of this function of 0. The value 0 is transformed by logistic function to 0.5. For any other data point that does not lie on the hyperpane the logistic function transforms this value to the value in the interval of $(0.5, 1.0]$ for the points on one side of the hyperplane, or the the value between $(0, 0.5]$ for the points on the other, negative side of the hyperplane. Conveniently, the farther away the point is from the hyperplane, the more to the extremes (0 or 1) will be its value of logistic regression. The hyperplane defined within the logistic regression model can thus serve as a decision boundary between two classes.

In what follows, we will use the logistic function so that our model $f_\beta(\boldsymbol{x})$ returns values between 0 and 1. We will continue to assume that our class variable is two-valued, but assume that its target value is a class marked with $y = 1$ and that for it, the model $f_\beta(\boldsymbol{x})$ returns the probability of this class. Notice that the parameters $\boldsymbol{\beta}$ fully define our logistic

regression model. Thus, we can write:

$$p(y = 1|\boldsymbol{x};\boldsymbol{\beta}) = f_{\boldsymbol{\beta}}(x) \tag{3.35}$$

$$p(y = 0|\boldsymbol{x};\boldsymbol{\beta}) = 1 - f_{\boldsymbol{\beta}}(x) \tag{3.36}$$

The expression for $p(y = 1|\boldsymbol{x};\boldsymbol{\beta})$ therefore gives us the probability that the target class of the data instance described by the vector of attribute values $\boldsymbol{x}$ is equal to 1. That is, it gives the probability that the independent variable takes the value of 1 for the data instance described with $\boldsymbol{x}$, where the model is parametrized with parameters $\boldsymbol{\beta}$. We can combine the two equations into a single one as

$$p(y|x;\boldsymbol{\beta}) = (f_{\boldsymbol{\beta}}(x))^y(1 - f_{\boldsymbol{\beta}}(x))^{1-y} \tag{3.37}$$

The expression for $p(y|\boldsymbol{x};\boldsymbol{\beta})$ in the above equation therefore gives the probability for a certain value of the independent variable and a certain vector of attribute values in the model given by $\boldsymbol{\beta}$. Now imagine that the values of the elements of the vector $\boldsymbol{\beta}$ change. Certainly, in this way, the probability for a given class in a selected case will also change; once this probability will be higher, another time lower. Change in the parameters of the model changes the probabilities with which we observe the data from the training set.

Let us now freeze the parameters of the model and compute the joint probability $L(\boldsymbol{\beta})$, the likelihood, for all the instances in the training set. We assume that the examples from the training set are independent and therefore the probability, with which we observe the values of the classes of particular data instanes described with the vectors $\boldsymbol{x}$ can be written as the product of the probabilities of individual data instances:

$$\begin{aligned} L(\boldsymbol{\beta}) &= p(\boldsymbol{y}|\boldsymbol{X};\boldsymbol{\beta}) \\ &= \prod_{i=1}^{N} p(y_i|x_i;\boldsymbol{\beta}) \\ &= \prod_{i=1}^{N} f_{\boldsymbol{\beta}}(x_i)^{y_i}(1 - f_{\boldsymbol{\beta}}(x_i))^{1-y_i} \end{aligned} \tag{3.38}$$

Again, just like with the linear regression, we would like to infer the model that maximizes the likelihood. That is, the parameters where we maximize the probability with which the model observes the training set. For convenience of deriving the $\boldsymbol{\beta}$ which maximize the likelihood, we compute its logarithm, and then maximize the log-likelihood:

$$\begin{aligned} \ell(\boldsymbol{\beta}) &= \log L(\boldsymbol{\beta}) \\ &= \sum_{i=1}^{N} \Big[y_i \log f_{\boldsymbol{\beta}}(x_i) + (1 - y_i) \log(1 - f_{\boldsymbol{\beta}}(x_i)) \Big] \end{aligned} \tag{3.39}$$

3.7 Gradient Descent for Logistic Regression

We are therefore looking for such $\boldsymbol{\beta}$ that maximizes the log-likelihood $\ell(\boldsymbol{\beta})$. Since we will use the gradient method, and since $\boldsymbol{\beta}$ is a vector $[\beta_0\ \beta_1\ \ldots\ \beta_M]$, we need to calculate the partial derivatives of our criterion function:

$$\begin{aligned}
\frac{\partial}{\partial \beta_j}\ell(\boldsymbol{\beta}) &= \sum_{i=1}^{M}\frac{\partial}{\partial \beta_j}\Big[y_i \log f_{\boldsymbol{\beta}}(x_i) + (1-y_i)\log(1-f_{\boldsymbol{\beta}}(x_i))\Big] \\
&= \sum_{i=1}^{M}\Big[y_i\frac{1}{g(\boldsymbol{\beta}^\intercal x_i)} - (1-y_i)\frac{1}{1-g(\boldsymbol{\beta}^\intercal x_i)}\Big]\frac{\partial}{\partial \beta_j}g(\boldsymbol{\beta}^\intercal x_i) \\
&= \sum_{i=1}^{M}\Big[\frac{y_i}{g(\boldsymbol{\beta}^\intercal x_i)} - \frac{(1-y_i)}{1-g(\boldsymbol{\beta}^\intercal x_i)}\Big]g(\boldsymbol{\beta}^\intercal x_i)(1-g(\boldsymbol{\beta}^\intercal x_i))\frac{\partial}{\partial \beta_j}\boldsymbol{\beta}^\intercal x_i \\
&= \sum_{i=1}^{M}\Big[\frac{y_i - g(\boldsymbol{\beta}^\intercal x_i)}{g(\boldsymbol{\beta}^\intercal x_i)(1-g(\boldsymbol{\beta}^\intercal x_i))}\Big]g(\boldsymbol{\beta}^\intercal x_i)(1-g(\boldsymbol{\beta}^\intercal x_i))x_{ji} \\
&= \sum_{i=1}^{M}(y_i - g(\boldsymbol{\beta}^\intercal x_i))x_{ji} \\
&= \sum_{i=1}^{M}(y_i - f_{\boldsymbol{\beta}}(x_i))x_{ji} \qquad (3.40)
\end{aligned}$$

Very easy! Have we seen such a result or such an equation before? Of course! In linear regression. Partial derivatives are identical to those for linear regression. Of course with a small difference. This time our function $f_{\boldsymbol{\beta}}$ uses the logistic function, while in linear regression $f_{\boldsymbol{\beta}}$ was only the weighted sum of attribute values.

The iterative process using gradient descent to find the parameters of the logistic regression model is identical to that of linear regression. Of course, with the small difference that this time the $f_{\boldsymbol{\beta}}(x)$ is a logistic regression model. However, we write down the step for refreshing the value of the parameter $theta_j$:

$$\beta_j \leftarrow \beta_j + \alpha\sum_{i=1}^{N}\Big(y_i - f_{\boldsymbol{\beta}}(x_i)\Big)x_{ji} \qquad (3.41)$$

Again, the α learning rate is typically small for normalized data (e.g., 0.001).

A warning applies here. The gradient descent approach is slow, and even for medium-sized data, many iterations of correcting the values of the $\boldsymbol{\beta}$ parameters are required. Instead, we typically use optimization techniques that have faster convergence. One of these is the L-BFGS, which is typically used from an accessible Python library.

Chapter 4

Generalized Linear Models

This chapter is in writing, and we should complete it soon. Meanwhile, please check the literature cited on the course's web site—sincere apologies for any inconvenience.

Chapter 5

Feature Selection and Model Regularization

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

5.1 Relation to Dimensionality Reduction

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

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

mensionality reduction are:

- *Feature transformation* that embeds the data into a lower-dimensional space, replacing original features with a new set that retains as much of information as possible. Approaches of this kind include principal component analysis and deep autoencoders, some of which we will cover in later chapters.

- *Feature selection*, also known as feature subset selection, variable selection, or attribute selection. This approach removes the dimensions (*e.g.* columns) from the input data and results in a reduced data set for model inference.

- *Regularization*, where we are constraining the solution space while doing optimization. Here, we add adding the regularisation terms to which an optimization algorithm must adhere to when minimizing the loss function, apart from having to minimize the error between the true y and the predicted $\hat{y}$. In lasso regularization, for instance, optimization is instructed to find model parameters so that to minimize their absolute sum. This type of regularization may lead to some of the parameters be equal to zero, effectively imposing zero weight to corresponding features, essentially canceling them out from the model. Hence, we can also regard regularization as a feature selection, where the inference method per se includes the feature selection procedure.

5.2 Feature Selection

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

- Because we have a discrete search space, we can, in general, not find the optimal solution, unless we perform a global search and evaluate all 2^n solutions.

- Unless the number of original features n is very small, examining all 2^n solutions is infeasible.

- In practice, the best we can do is to use heuristic search methods with a good inductive bias. This approach tends to work well because we can impose assumptions that are more likely to hold on the data we encounter.

- Any search method that operates on discrete search spaces can also be applied to feature selection.

Filter Methods

Filter methods (**Guyon2003**) perform feature selection before the inference of the model. They rely on a feature scoring function that assigns the score to a feature according to how useful the feature could be in the model. Notice that the scoring is performed before and independently of the model. Features are scored and then ranked, and usually, a top k features are selected, where k is a user-defined parameter of the procedure. Alternatively, feature scores could be compared to their null-distribution that, in practice, could be obtained through feature scoring on a randomly permuted data set. In such cases, k is replaced with user-defined probability p that a particular (or higher) feature score could be obtained and randomly-permuted data.

Scoring functions depend on the type of machine learning problem and type of the scored feature. For instance, for unsupervised learning, we may disregard features with near-constant values by selecting features with the highest deviance, that is, with the highest ratio between variance and the mean. Scoring functions for classification or regression most often consider the correlation between a predictor and the dependent variable. One of the most popular empirical estimates is the mutual information between the i-th predictor and the target y (**Guyon2003**) :

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

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

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

Notice that mutual information and all similar feature scoring techniques are univariate and assess the information held by the feature in the absence of the context of other features. The scoring function of this type would undervalue features that are in some interactions with other features and only combined with these provide information about the class. A typical example of such a combination is an exclusive disjunction, where participating features may provide no information about the class on their own, yet are information-rich when they are considered together with complementing argument. A field that studies the discovery and ranking of such features is called feature interaction analysis (**Jakulin2005**; **Anastassiou2007**). The approaches cited here rely on an exhaustive search for feature interactions, which are prohibitive in complexity even for reasonably-sized data sets. A bigger problem, though, is that estimates of feature interactions may report about highly interactive features simply by chance and due to an extremely high number of feature combinations

explored.

Interestingly, however, note that there are feature score estimators that take into consideration the contexts and are sensitive to feature interactions. The most prominent of these is Relief, an algorithm originally developed by **Kira1992**. The algorithm assumes that each feature in the data set has been scaled to the interval $[0, 1]$. Let $\boldsymbol{w}$ be a feature weight vector initialized to $\mathbf{0}$. The algorithm randomly draws a data instance $\boldsymbol{x}_i$ and updates the weight vector, such that:

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

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

Kononenko1997 proposed several extensions and improvements of Relief. On the surface, Relief looks like a perfect feature scoring algorithm that can indeed cope with any hidden feature interactions. However, it relies on finding similar data instances and thus suffers from the same problem as any nearest neighbors approach. In other words, Relief would start failing in data sets with a higher number of features, which is precisely where feature interaction discovery would be of the highest value.

Wrapper Methods

Wrapper methods score feature sets according to the estimated accuracy or utility of the algorithm used for learning. The most common search approach is forward/backward stepwise selection (**Guyon2003**). Forward selection is an iterative procedure that starts with an empty set of features and adds one feature at a time, where the benefit of adding a feature is observed in raised accuracy of the inferred model when that feature is added to the feature set. Backward selection starts with a full set of features, and then eliminates one feature at the time, each time selecting feature with the smallest impact on the accuracy of the model. Notice, of course, that backward selection can actually increase the accuracy of the model, as we expect that there is an optimal feature subset with corresponding highest accuracy.

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

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

Embedded Methods

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

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

A Rough Summary on Feature Selection Techniques

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

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

- feature selection has an advantage over feature transformation as it keeps the original features, which helps with interpretability,
- feature selection is related to explanation/interpretability also through methods that assess variable importance, that is, provides a ranked set of features which can be scrutinized by the domain experts,
- as computational power grows and data sets get larger, filter methods are used less and less and give way to models whose inference relies on optimization and gradient-based search of parameter space.

5.3 Regularization

Let us start with considering linear regression, where $y_i = \beta^\intercal x_i + \epsilon_i$ and where we assume that the error term is distributed normally, so that $\epsilon_i \sim_{\text{iid}} N(0, \sigma^2)$. The data, conveniently, includes a constant column, such that $x_0 = 1$, consequently using β_0 as an intercept, a constant term in linear combination. Linear regression aims to find β^* that minimizes residual sum of squares,

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

so that

$$\beta_{\text{OLS}} = \arg\min_{\beta} \sum_{i=1}^{n}(y_i - \beta^T x_i)^2.$$

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

Linear regression also has a closed form solution (**ESL**). Let X denote $n \times (1 + p)$ matrix with n training data instances described with p features. The first column of the matrix is a unit vector. Residual sum of squares can then be expressed as:

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

Differentiating with respect to β we obtain

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

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

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

The main problem that we will address with regularization are cases with few observations in the data set that are described with comparably many features. In other words, cases where the training data matrix X has relatively few rows compared with columns. Here, we are likely to overfit the data, that is, develop a complex model that includes many features

and fits training data well but performs poorly in prediction on new data. If we have more columns than rows, we will have colinearity, so $\boldsymbol{X}^\intercal\boldsymbol{X}$ will not be invertible, and if we want to solve it by minimizing the sum of squares, we will have an infinite number of equivalent solutions.

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

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

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

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

Closed-Form Solution for L2 Regularization

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

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

Note that if we differentiate again, we get $2\boldsymbol{X}^\intercal\boldsymbol{X} + 2\lambda\boldsymbol{I}$. This is always positive definite

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

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

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

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

Equivalence of Penalized and Constrained Forms

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

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

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

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

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

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

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

Now we write the Lagrangian of the constrained formulation:

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

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

continuously differentiable constraints:

$$\frac{d}{d\boldsymbol{\beta}}L(\boldsymbol{\beta},\mu) = 0 \tag{5.1}$$

$$\mu \geq 0 \tag{5.2}$$

$$\mu(\|\boldsymbol{\beta}\|_2^2 - c) = 0 \tag{5.3}$$

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

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

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

L1 regularization

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

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

or, in vector notation:

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

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

Bayesian Interpretation of Regularization

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

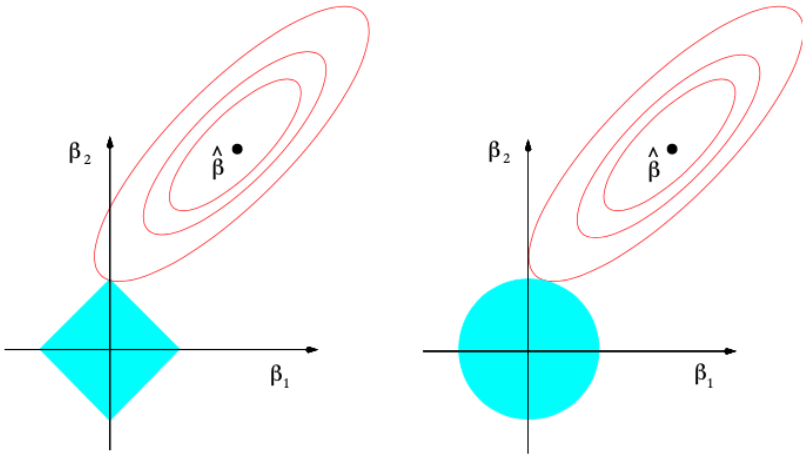


Figure 5.1: Geometric presentation of the optimization problem for the lasso (L1, left) and ridge (L2, right) regression. Shown are conturs of the penalty (least squares error), and the constrain regions $|\beta_1| + |\beta_2| \leq c$ and $\beta_1^2 + \beta_2^2 \leq t^2$. The sharp corners of the constraint region of the lasso yield sparse solutions. In high dimensions, sparsity arises from corners and edges of the lasso's constraint region (from **Tibshirani2014**).

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

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

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

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

Indeed, maximizing the log-likelihood, we obtain

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

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

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

The answer should be obvious: $\prod_{i=1}^{k} \exp(-\frac{\boldsymbol{\beta}_i^2}{1/\lambda})$. In terms of $\boldsymbol{\beta}_i$ this is proportional to the product of normal likelihoods $\boldsymbol{\beta}_i \sim_{iid} N(0, \frac{1}{\sqrt{\lambda}})$. So, if we look at this in terms of the Bayesian

(log)posterior $\log p(\boldsymbol{\beta}|\boldsymbol{y},\boldsymbol{x}) \propto \log p(\boldsymbol{y}|\boldsymbol{\beta},\boldsymbol{x}) + \log p(\boldsymbol{\beta})$, we see that if we place a normal prior on the coefficients, we get a posterior maximum that corresponds to the L2 regularized MLE solution. In other words, the Bayesian interpretation of L2 regression is that we express a prior opinion that coefficients are normally distributed around 0 with some variance that is a function of $\sigma^2 = \frac{1}{\sqrt{\lambda}}$. Higher variance implies lower λ (less regularization), while lower variance implies higher λ (more regularization).

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

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

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

Final Remarks

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

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

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

Chapter 6

Kernels

Machine learning often considers problems where we profile objects with attribute-value vectors. This representation has, in principle, several limitations. Objects may be complex, and their vector-based representation is not trivial. Consider text documents, molecular structures, trees, graphs, and networks. For these, an alternative to feature-based representation is the utility of a function that can measure object-to-object similarity. Moreover, even if feature-based representation is available, it may be too weak to allow simpler models, like linear and logistic regression, to model more complex relations, like feature interactions. One way to surpass such limitations is kernels. In general, kernels are functions that map a pair of input objects to a number. One use of kernels is to consider a prototype object and then map input space into a latent representation, where selected modeling techniques may be more successful. When applied to a pair of data instances, we can regard kernels as functions that measure similarity. Smoothing kernels as used in kernel density estimation, which has a substantially different meaning. In this chapter, we look at a range of typical kernels and approaches that can use kernels in model inference.

In the previous chapters, we have introduced machine learning models that consider a set of training data to infer a predictive model. The training data is then discarded, and predictions for new inputs are formed entirely based on the model and its inferred parameters.

In this chapter, we introduce a different class of machine learning techniques that keeps the training data and uses it within the prediction phase. We have already exposed one such algorithm, namely k-nearest neighbors. We refer to algorithms of this kind as *lazy*, or *memory-based*. They typically require a metric of similarity of any two data points from the input space. We can recast many linear parametric models into an equivalent dual representation in which the predictions are based on a linear combination of a *kernel function* evaluated in the original, input space. For models, which are based on a fixed non-linear *feature space*

mapping ϕ, the kernel function is given by the relation

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \phi(\boldsymbol{x})^\intercal \phi(\boldsymbol{x}').$$

The kernel function is, from the definition, symmetric. Again, and importantly, the definition above says that instead of computing the dot product between the vectors in the latent, mapped space, we can compute the kernel function in the space of original features. This concept, thus formulated as an inner product in a feature space, allows us to develop extensions of many well-known regression and classification methods. All we need to do is to reformulate the methods to operate with dot products of input vectors. When introducing transformation to latent space, this product is then replaced with the kernel. We show the utility of the kernel trick in detail for one regression and for one classification approach: linear regression with ridge regularization and support vector machines.

In particular, support vector machines have received much attention in the past, but their importance has been in decay with the introduction of recent approaches, including deep networks. A particularly important driver of support vector machine's success was the utility of kernels on structured objects, like text, voice, images, and graphs.

6.1 Examples of Kernel Functions

A *kernel function*, or just a *kernel* is defined as:

$$\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R},$$

where $\mathcal{X}$ is our variable space or typically, an input space. A kernel is thus a function $\kappa(x, x')$ that takes a pair of elements from the input space $x, x' \in \mathcal{X}$ and maps them to a real number. In practice we typically deal with kernel functions where $\kappa(x, x') \geq 0$ and $\kappa(x, x') = \kappa(x', x)$. That is, non-negative and symmetric kernel functions, which allows us to interpret them as *similarity measures*.

Notice that *kernel* has different meanings in different contexts. We will be here covering three:

- First, we will look at kernels in a very general sense - as functions that map a pair of elements from our (input, feature) space to a number.
- Then we will move on to positive-definite (or Mercer) kernels, which are a special case of the former (that is, with additional requirements) and allow for more efficient computation that is the basis for models such as SVM and kernel ridge regression.
- And third, we will introduce smoothing kernels that are used in kernel density estimation and has a substantially different meaning.

Polynomial Kernel

A standard kernel that is related to a transformation to a latent space that can, for instance, yield linearly-inseparable data instances manageable under linear models (e.g., Fig. 6.4) is a polynomial kernel:

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\intercal \boldsymbol{x}' + 1)^n.$$

For $n = 2$, and assuming that $x = [u_1 u_2]^\intercal$ and $x' = [v_1 v_2]^\intercal$ we get

$$\begin{aligned}(\boldsymbol{x}^\intercal \boldsymbol{x}' + 1)^2 &= (u_1 v_1 + u_2 v_2 + 1)^2 \\ &= u_1^2 v_1^2 + u_2^2 v_2^2 + 1 + 2u_1 v_1 + 2u_2 v_2 + 2u_1 v_1 u_2 v_2 \\ &= \langle 1,\ \sqrt{2}\ u_1,\ \sqrt{2}\ u_2, u_1^2,\ \sqrt{2}\ u_1 u_2, u_2^2\rangle^\intercal \langle 1,\ \sqrt{2}\ v_1,\ \sqrt{2}\ v_2, v_1^2,\ \sqrt{2}\ v_1 v_2, v_2^2\rangle\end{aligned}$$

Polynomial kernel of second degree returns a dot product of vectors in six-dimensional space.

Radial Basis Function Kernels

The squared exponential kernel, or *Gaussian* kernel is defined by:

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}')^\intercal \Sigma^{-1} (\boldsymbol{x} - \boldsymbol{x}')\right)$$

When Σ is diagonal, this kernel can be expressed as:

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{1}{2}\sum_{j=1}^{D} \frac{1}{\sigma_j^2}(x_j - x_j')^2\right)$$

We can interpret σ_j as a characteristic length scale of the dimension. If we assume that all characteristic length scales are equal, then we can write this kernel as:

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2\sigma^2}\right)$$

where σ^2 is known as a bandwidth. Since this kernel depends only on a function $\|\boldsymbol{x} - \boldsymbol{x}'\|$, that is, only on a distance between a point $\boldsymbol{x}$ and, say, a reference $\boldsymbol{x}'$, this kernel is a radial basis function and is often referred to as an *RBF kernel*.

Notice that an RBF kernel has a parameter, σ, that needs to be either set by the user given some domain knowledge or inferred from the data through, for example, internal cross-validation.

Linear Kernel

When $\phi(\boldsymbol{x}) = \boldsymbol{x}$, we get a linear kernel defined as:

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^\top x'$$

This kernel is useful if the original data is already high dimensional, and if the original set of features is informative. Examples of such data sets are frequent in text mining and related with a bag of words representation of text documents, or data sets from molecular biology that involve thousands of genes or millions of single-nucleotide polymorphisms. In these cases, a linear combination of features may represent a sufficiently accurate decision boundary, and it may not be required to use some other latent representation.

Kernels for Comparing Text Documents

Notice that a kernel provides a proxy for the similarity of data instances. Given two objects, we will be able to construct regressors or classifiers by only computing the kernels, that is, estimating the similarity between two objects. If the objects are text documents, we can represent the document with vectors that contain word frequencies. We often refer to this presentation as *bag of words*. Because we can consider documents of different lengths, the Euclidean distance would fail (why?). We can instead normalize the bag-of-words representation according to the document length or use the *cosine similarity* [1]:

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \frac{\boldsymbol{x}^\top \boldsymbol{x}'}{||\boldsymbol{x}||\,||\boldsymbol{x}'||}$$

Cosine similarity measures the cosine of the angle between the two vectors $\boldsymbol{x}$ and $\boldsymbol{x}'$ that represent the corresponding documents. Since both vectors are count vectors, the cosine similarity will be between 0 and 1.

Bag of words representation may include punctuations and frequently occurring words, so-called stop words, that may obscure the differences between documents and yield document representations too similar to each other. Various techniques for text pre-processing to avoid this effect were proposed in the literature. Among the most frequently used are stop-words removal and transform called *term frequency-inverse document frequency*, which replaces word counts with weights to expose less frequent words.

String Kernels

Kernels that operate on strings report on string sequence similarity. In these times, it may not be hard to consider RNA sequences of viruses that have infected people at different continents.

[1] Euclidean distance between normalized vectors and cosine similarity are in practice almost identical. Find what is their relation mathematically!

Due to mutations, their sequence may be different, and so is the effect on a phenotype of a patient. We may predict these phenotypes using kernels that measure sequence similarity. Consider the following three sequences:

```
TCGGTTTAACGGATTATGGTAC
TCGGTCCAACGGATAATGGAAC
TCGGCGATTTAACGGATCGATTTATGGTAC
```

To compare them, we may, for instance, use edit distance, that is, the measure that reports how many atomic changes like deletions, insertions, or single nucleotide mutations we have to introduce in one sequence to derive another one. Sequence similarity may also be computed through the count of the substrings the two strings have in common, or through the length of the longest common substring, or similar. There is vast research on string similarity measures and means to compute them through sequence alignment in the literature of molecular biology, and interested readers should consult algorithms such as BLAST or CLUSTAL.

6.2 Kernelized Linear Models

One simple way to use kernels for classification or regression is to use *kernel machines*. A kernel machine is a generalized linear model where the input vector has the form

$$\phi(\boldsymbol{x}) = [\kappa(\boldsymbol{x}, \boldsymbol{\mu_1}), \ldots, \kappa(\boldsymbol{x}, \boldsymbol{\mu_K})]$$

where $\boldsymbol{\mu}_k$ is a set of *centroids* or *prototypes*, that is, a subset of examples from the training set. Considering kernels as proximity functions, the above-defined *kernelized feature vector* can be, for a given data instance, regarded as a vector of similarity to the prototype data instances. The general idea is that wherever we use a linear term $\boldsymbol{\beta}^T\boldsymbol{x}$, that is, in linear regression, generalized linear models, ordinal regression or similar, we could instead transform the input vector x via a kernel function with respect to a set of *prototype* observations $\boldsymbol{\mu}_1$, ..., $\boldsymbol{\mu}_k$. After constructing these new features, we proceed with the inference as we would with the original modeling method, we have a different, transformed input space (the term used now is $\beta^T\phi(x)$.

Prototypes, if appropriately selected, may help linear models to model feature spaces with feature interactions. Consider an XOR problem and logistic regression. Using RBF kernel and four different prototypes, logistic regression would be able to infer a perfect model for this otherwise hard classification case. Notice, though, that choice of the prototypes here is essential. In general, though:

- The number of prototypes K can be less, above, or equal to the dimension of the original training set.
- We can choose the prototypes using some systematic approach, like clustering. Alternatively, we could use every training example, x, for a prototype.

- If the number of prototypes is large, we could use any of the sparsity-promoting priors on $\boldsymbol{\beta}$, as discussed in the chapter on regularization. We refer to such an approach is called *sparse vector machine*. The most natural choice is ℓ_1 regularization, an approach we refer to as ℓ_1*-regularized vector machine*, or L1VM. Another popular approach of creating a sparse kernel machine is a support vector machine, discussed in detail below.
- And, most importantly, worth emphasizing again – we can use this approach within any linear model we have learned so far. With this approach, we can produce non-parametric versions of those parametric approaches, where their expressiveness can grow with the number of data points.

6.3 Mercer Kernels

Mercer kernels are related to approaches that use kernels through the so-called kernel trick and where we define models where input data appears within the inner products of the input data instances. We here define the Mercer kernels, establish the equivalence between Mercer kernels and inner products in transformed space, and outline some of the rules to follow when constructing a new kernel.

Definition Mercer's Theorem

A kernel function κ of the form $\mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is said to be *symmetric positive semidefinite* if it is (a) symmetric: $\kappa(x, x') = \kappa(x', x)$ and (b) for any integer $m > 0$ and any set of m vectors $x_i \in \mathbb{R}^d$ the matrix

$$K = \begin{bmatrix} \kappa(x_1, x_1) & \dots & \kappa(x_1, x_m) \\ \vdots & \ddots & \vdots \\ \kappa(x_m, x_1) & \dots & \kappa(x_m, x_m) \end{bmatrix}$$

is positive semidefinite. This matrix is also called the *Gram* matrix.

Definition 1 *Mercer Kernel. A symmetric positive semidefinite kernel κ is also called a* Mercer kernel.

Definition 2 *Mercer's Theorem. If κ is a Mercer kernel then it is an inner product (dot product) $\kappa(x, x') = \langle \phi(x), \phi(x') \rangle$ for some (possibly infinite dimensional) mapping $\phi(x) : \mathbb{R}^m \to \mathcal{D}$.*

Note that the mapping ϕ is often called the *basis function* and the space $\mathcal{D}$ is called the *feature space*.

This theorem is fundamental. It justifies the *kernel trick* that we will see a couple of times later. The kernel trick is just a direct application of Mercer's theorem - if we have a method that depends only on the inner products, we can replace those inner products with any Mercer

kernel to obtain the application of that method on the feature space $\mathcal{D}$ determined by that kernel. Of course, computing the kernel is, in most cases, much easier.

Polynomial Kernel Revisited

We can start by considering one-dimensional polynomial regression, which is equivalent to using the basis function $\phi : x \to (1, x, x^2, ..., x^r)$. For example, for quadratic polynomial regression ($r = 2$), $\phi : x \to (1, x, x^2)$ and for cubic polynomial regression ($r = 3$), $\phi : x \to (1, x, x^2, x^3)$. Similarly, for higher dimensional x, we have $\phi : (x_1, x_2) \to (1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$.

In practice, it turns out that the basis functions for polynomial kernels use slightly different weights. For the cubic example, instead of $\phi : x \to (1, x, x^2, x^3)$, we use $\phi : x \to (1, \sqrt{3}x, \sqrt{3}x^2, x^3)$. The squared-roots do not make any difference when considering linear combinations, as the inner two coefficients will just be scaled by $\sqrt{3}$. However, it does make the computation much more convenient:

$$\langle \phi(x), \phi(x') \rangle = 1 + 3xx' + 3x^2(x')^2 + x^3(x')^3 = (1 + xx')^3.$$

This can be generalized to arbitrary power r, obtaining the *polynomial kernel* $\kappa(x, x') = (1 + xx')^r$. The beauty of this kernel is that the computation of the inner product in feature space, which has $\binom{r+m}{m}$ dimensions, actually requires just one inner product in the original space (dimension m), adding one and taking the power.

RBF Kernel Revisited

With no proof, we should state here that:

1. Radial basis function (RBF) kernel is a Mercer kernel.
2. RBF kernel corresponds to a basis function that transforms the data instance to an infinite-dimensional feature space, that is, to a space of an infinite sum of polynomial kernels. So not only is the feature space inner product more convenient to compute in original space, it would be impossible to compute in feature space.

Other Mercer Kernels

In general, it is difficult to verify if a kernel is a Mercer kernel. There are, however, operations that preserve the property. If κ_1 and κ_2 are Mercer kernels defined on the same feature space, then the following are also Mercer kernels (**ShaweTaylorCristianini2004**):

- $\kappa(x, x') = c\kappa_1(\boldsymbol{x}, \boldsymbol{x}')$, where $c > 0$ is a constant
- $\kappa(x, x') = f(\boldsymbol{x})\kappa_1(\boldsymbol{x}, \boldsymbol{x}')f(\boldsymbol{x}')$, where $f(x)$ is a real function with nonnegative coefficients

- $\kappa(x, x') = f(\kappa_1(\boldsymbol{x}, \boldsymbol{x}'))$
- $\kappa(x, x') = \kappa_1(\boldsymbol{x}, \boldsymbol{x}') + \kappa_2(\boldsymbol{x}, \boldsymbol{x}')$
- $\kappa(x, x') = \kappa_1(\boldsymbol{x}, \boldsymbol{x}')\kappa_2(\boldsymbol{x}, \boldsymbol{x}')$
- $\kappa(x, x') = \kappa_1(\phi(\boldsymbol{x}), \phi(\boldsymbol{x}'))$
- $\kappa(x, x') = \boldsymbol{x}^\intercal \boldsymbol{A} \boldsymbol{x}'$, where $\boldsymbol{A}$ is symmetric positive semidefinite matrix

Let us start with a linear function $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\intercal \boldsymbol{x}')$, which is a kernel since $\boldsymbol{A} = \boldsymbol{I}$ is a positive definite matrix, (and as such also positive semi-definite). Then a simple polynomial kernel $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\intercal \boldsymbol{x}')^2$ = contains only terms of degree two, and is equal to the linear kernel squared, and is also a kernel. A slightly generalized function $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\intercal \boldsymbol{x}' + c)^2$ with $c > 0$ is also a kernel since its expansion contains linear combinations of linear or polynomial kernels of degree two. Through similar reasoning, we can find that $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\intercal \boldsymbol{x}' + c)^M$ is a kernel for any $c \geq 0$.

Another commonly used kernel, a Gaussian kernel, takes the form

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\|\boldsymbol{x} - \boldsymbol{x}'\|^2 / 2\sigma^2\right)$$

This is a valid kernel, since by expanding the square

$$\|\boldsymbol{x} - \boldsymbol{x}'\|^2 = \boldsymbol{x}^\intercal \boldsymbol{x} + (\boldsymbol{x}')^\intercal \boldsymbol{x}' - 2\boldsymbol{x}^\intercal \boldsymbol{x}'$$

we can write

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \exp(-\boldsymbol{x}^\intercal \boldsymbol{x}/2\sigma^2) \exp(\boldsymbol{x}^\intercal \boldsymbol{x}'/\sigma^2) \exp(-(\boldsymbol{x}')^\intercal \boldsymbol{x}'/x\sigma^2)$$

and see that this is a valid kernel.

6.4 Application of the Theory of Mercer Kernels to Modelling

We want to express an existing linear method in terms of inner products and then replace them with a kernel to obtain the linear method in the transformed feature space, consequently constructing a non-linear model. That is, we apply the *kernel trick*. Of course, for this to work, the kernel needs to be a Mercer kernel, not just any kernel function. Below, we show that regularized linear regression has a dual form that uses dot products of input vectors, and hence it can be kernelized. We also derive support vector machines, a linear classification algorithm that uses a weighted sum of the dot product of pairs of data instances from the training data set. Furthermore, we briefly discuss a kernelized version of the k-nearest neighbors. For other methods, like support vector machine regression, kernelized principal component analysis, or kernelized k-means clustering, see (**2012-Murphy**).

Kernelized Ridge Regression and the Kernel Trick

First, recall what we've learned about ridge regression, that is, L_2-regularized linear regression. Again, let $X \in \mathbb{R}^{N \times D}$ be our independent variables and $y \in \mathbb{R}^N$ our dependent variable. Given a regularization parameter λ the objective is to find coefficients β that minimize the squared error and the squared sum of coefficients β,

$$\hat{\beta} = \arg\min_{\beta} \left(\|X\beta - y\|_2^2 + \lambda \|\beta\|_2^2 \right).$$

We know that this has a closed-form solution $\hat{\beta} = (X^T X + \lambda I_d)^{-1} X^T y$.

Now we will rewrite this solution in an alternative (dual) form, which will facilitate the use of a kernel. Observe that

$$(X^T X + \lambda I_d) X^T = X^T X X^T + \lambda X^T = X^T (X X^T + \lambda I_n).$$

Multiplying the leftmost and rightmost terms by $(X^T X + \lambda I_d)^{-1}$ on the left and $(X X^T + \lambda I_d)^{-1}$ on the right-hand side, we get

$$X^T (X X^T + \lambda I_n)^{-1} = (X^T X + \lambda I_d)^{-1} X^T.$$

So, we have found an alternative formulation of the closed-form solution:

$$\hat{\beta} = X^T (X X^T + \lambda I_n)^{-1} y.$$

Using this new formulation as depicted above, we can define the prediction for a new observation as

$$\hat{y}(x') = \hat{\beta}^T x' = (x')^T \hat{\beta} = (x')^T X^T (X X^T + \lambda I_n)^{-1} y.$$

The critical observation here is that *the prediction depends on X and x' only through standard inner products*.

More precisely, $(x')^T X^T = \begin{bmatrix} \langle x', x_1 \rangle \\ \vdots \\ \langle x', x_n \rangle \end{bmatrix}^T$ and $X X^T = \begin{bmatrix} \langle x_1, x_1 \rangle & \dots & \langle x_1, x_n \rangle \\ \vdots & \ddots & \vdots \\ \langle x_n, x_1 \rangle & \dots & \langle x_n, x_n \rangle \end{bmatrix}$ is the Gram matrix.

So, we can apply the *kernel* trick and replace these inner products with the more general kernelized formulation.

$$\hat{y}(x') = k(x')(K + \lambda I_n)^{-1} y,$$

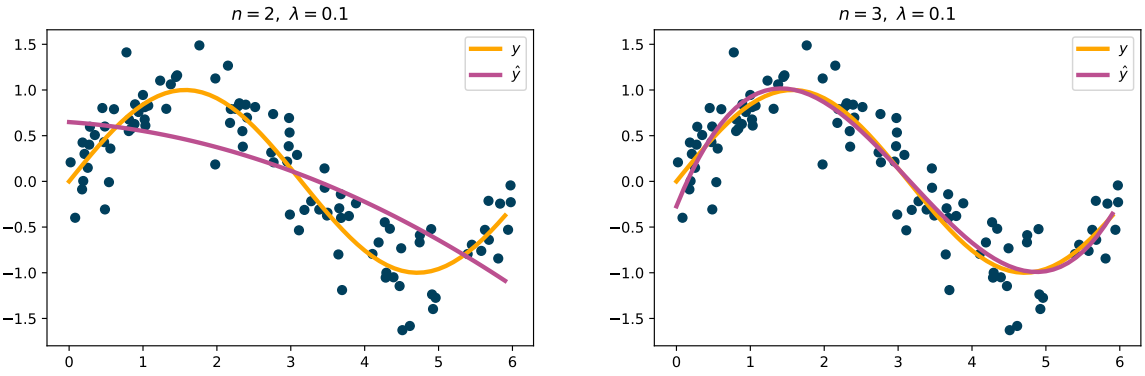


Figure 6.1: Kernelized ridge regression with polynomial kernel on one-dimensional data. Choice of hyper-parameter of the kernel can greatly influence the degree of fit.

where $k(x') = \begin{bmatrix} \kappa(x', x_1) \\ \vdots \\ \kappa(x', x_n) \end{bmatrix}^T$ and $K = \begin{bmatrix} \kappa(x_1, x_1) & \dots & \kappa(x_1, x_n) \\ \vdots & \ddots & \vdots \\ \kappa(x_n, x_1) & \dots & \kappa(x_n, x_n) \end{bmatrix}$.

That is, using a kernel, we can perform ridge regression in the space whose inner product is represented by the kernel.

An alternative view is to explicitly introduce the (dual) variable $\alpha = (K + \lambda I_n)^{-1} y$ to allow us to express the closed-form solution as

$$\hat{\beta} = X^T \alpha = \sum \alpha_i x_i.$$

The solution to the problem is just a linear combination of the observations!

Plugging this into the prediction for the new observation, we get

$$\hat{y}(x') = (x')^T \sum \alpha_i x_i = \sum \alpha_i (x')^T x_i = \sum \alpha_i \kappa(x', x_i),$$

which illustrates that the prediction for a new observation is just a weighted sum of training observations' values of y (weighted by the similarity of those observations with the new observation x')! For example, for linear regression, where we just have the standard inner product, observations that have a smaller angle (closer to 0 or 180) have a higher weight.

The degree of fit of kernelized ridge regression depends on the choice of kernel and its parameters, and on the choice of degree of regularization (see Figs. 6.5 and **??**).

Support Vector Machines

Support vector machines are, in essence, linear classifiers that infer separating hyper-planes with maximal margins to the neighboring data points. We here derive the formulation of this classifier, show how to use its power with a kernel trick, and describe extension for non-linearly separable cases. We also derive the formulation for support vector machines using

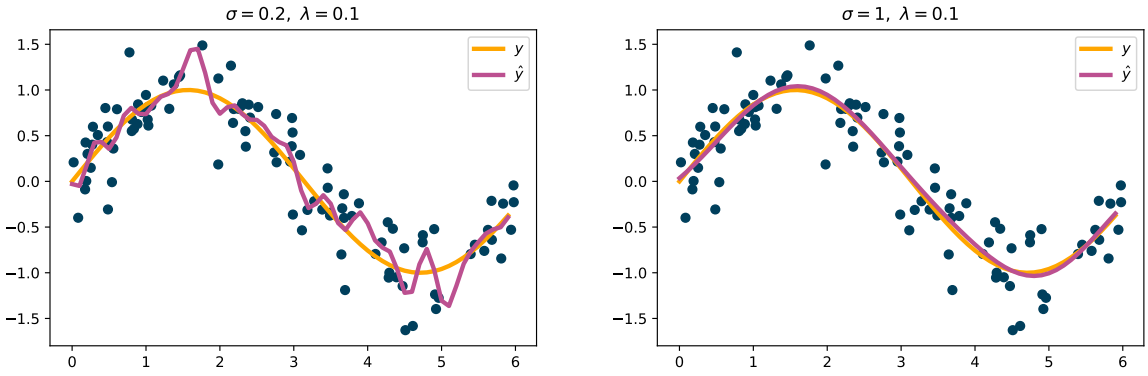


Figure 6.2: Wrong choice of hyperparameters can lead to overfitting, as shown in the plots of regression on one-dimensional data set with Gaussian kernel.

hinge loss.

The Large Margin Principle

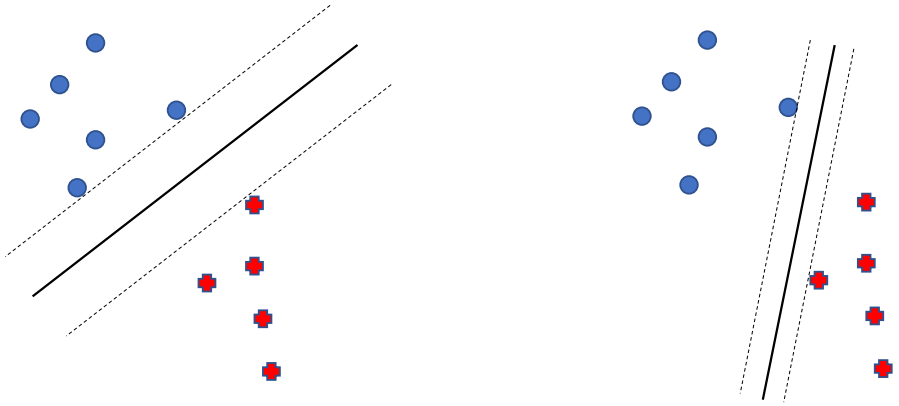


Figure 6.3: A separating hyper-plane with large (left) and small margin (right).

Consider a binary classification problem, with $y = \pm 1$, where data instances of the two classes are linearly separable. Our aim is to define a separating hyper-plane that splits the feature space so that the margin between the closest positive and the closest negative data instance is the largest, as seen on Fig. 6.3. Intuitively, a large margin would steer us from overfitting and may yield best accuracy on yet unseen data. Let us denote the margin with γ. We would therefore like to find $\boldsymbol{w}$ that defines the direction of a separating hyper-plane $\boldsymbol{w}^\intercal \boldsymbol{x} + w_0 = 0$, where w_0 is the intercept, with largest margin γ. With no loss of generality, let us choose w so that the data points on the margin are one unit away from separating

hyper-plane. The equations for the margins are thus:

$$\boldsymbol{w}^\intercal \boldsymbol{x} + w_0 = 1$$
$$\boldsymbol{w}^\intercal \boldsymbol{x}_\perp + w_0 = -1$$

Consider now a point $\boldsymbol{x}$ on one margin and its projection $\boldsymbol{x}_\perp$ across the separating hyperplane to the other margin:

$$\boldsymbol{x} = \boldsymbol{x}_\perp + 2\gamma \frac{\boldsymbol{w}}{||\boldsymbol{w}||}$$

Using the equations of the margin, so that $\boldsymbol{w}^\intercal \boldsymbol{x} + w_0 = 1$ and $\boldsymbol{w}^\intercal \boldsymbol{x}_\perp + w_0 = -1$, we obtain:

$$\boldsymbol{w}^\intercal \left(\boldsymbol{x}_\perp + 2\gamma \frac{\boldsymbol{w}}{||\boldsymbol{w}||} \right) + w_0 = 1,$$

$$\boldsymbol{w}^\intercal \boldsymbol{x} + w_0 + 2\gamma \frac{\boldsymbol{w}^\intercal \boldsymbol{w}}{||\boldsymbol{w}||} = 1,$$

$$-1 + 2\gamma \frac{\boldsymbol{w}^\intercal \boldsymbol{w}}{||\boldsymbol{w}||} = 1,$$

and finally, considering $\boldsymbol{w}^\intercal \boldsymbol{w} = ||\boldsymbol{w}||^2$,

$$\gamma = \frac{1}{||\boldsymbol{w}||}$$

To maximize the margin, we need to minimize the length of $\boldsymbol{w}$. Formally, and considering that all data points in our linearly separable classification problem have to lie on or outside the margin, we therefore want to opimize

$$\min_{\boldsymbol{w}, w_0} \frac{1}{2} ||w||^2 \;\; s.t. \; y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i + w_0) \geq 1, \; i = 1 \ldots N$$

For convenience, we have added $\frac{1}{2}$ and are optimizing the squared norm instead of the norm, which, of course, should be mathematically more convenient and leads to the same solution. We can now use the method of Lagrange multipliers to find the minima of our criteria function that fulfills the set of constrains. The primal Lagrangian is:

$$L(\boldsymbol{w}, w_0, \boldsymbol{\alpha}) = \frac{1}{2} ||w||^2 - \sum_{i=1}^{N} \alpha_i [y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i + w_0) - 1]$$

where $\alpha_i \geq 0$ are Lagrange multipliers. Optimal value of parameters that define the separating

hyper-plane is where the gradient of the Lagrangian is zero:

$$\begin{aligned}\nabla L &= \frac{\partial L}{\partial \boldsymbol{w}} \\ &= \boldsymbol{w} - \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i = 0\end{aligned}$$

Therefore,

$$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i \tag{6.1}$$

Separating hyperlane is defined through a normal vector that is a weighted sum of vectors that define our training data instances!

Similar holds for w_0,

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{N} \alpha_i y_i = 0$$

and hence

$$\sum_{i=1}^{N} \alpha_i y_i = 0$$

We now insert the values of our optimal parameters back to the Lagrangian:

$$\begin{aligned}L &= \frac{1}{2}\left(\sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i\right)\left(\sum_{j=1}^{N} \alpha_j y_j \boldsymbol{x}_j\right) - \left(\sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i\right)\left(\sum_{j=1}^{N} \alpha_j y_j \boldsymbol{x}_j\right) - \sum_{i=1}^{N} \alpha_i y_i + \sum_{i=1}^{N} \alpha_i \\ &= \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i \boldsymbol{x}_j\end{aligned}$$

Support vector classifier is then defined through the following optimization problem:

$$\begin{aligned}&\text{maximize} && L(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i \boldsymbol{x}_j \\ &\text{subject to} && \sum_{i=1}^{N} y_i \alpha_i = 0, \\ & && \alpha_i \geq 0,\ i = 1, \ldots, N\end{aligned}$$

The problem of finding the separating hyper-plane with maximal margin thus translates to quadratic programming, for which a standard solver can be used.

At this stage, notice that finding $\boldsymbol{\alpha}$ through quadratic programming provides us the weights $\boldsymbol{w}$ (see Eq. 6.1) and consequently provides for b. Notice that the solution of a quadratic programming problem, that is, the vector $\boldsymbol{\alpha}$ is typically sparse. That is, most α_i will be zero. That

is, the weights $\boldsymbol{w}$ will be defined by only a small number of input data instances. These data instances are those that define the margin and are called *support vectors*. The support vectors are, therefore, training data instances for which $\alpha_i > 0$.

The decision rule for classification of data instance $\boldsymbol{x}$ to the positive class ($y = 1$) is

$$\boldsymbol{w}^\top \boldsymbol{x} + b \geq 0$$

and if we substitute for $\boldsymbol{w}$ we obtain:

$$\sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x_i}^\top \boldsymbol{x} + b \geq 0$$

Notice that in essence we do not need to compute the weights $\boldsymbol{w}$ for classification. All we need is to remember the instances for the training data set and compute the scalar product between training data instances and the vector the represents data instance to be classified. We can notice that in the quadratic programming formulation, where we solve for $\boldsymbol{\alpha}$, its dependency on the training data is again expressed through the scalar product $\boldsymbol{x}_i^\top \boldsymbol{x}_j$.

Support Vector Machines and Kernel Trick

Consider a data set with linearly non-separable classes (Fig. 6.4, left panel), and a transformation $\Phi(\boldsymbol{x}) = \langle x_1^2, x_2^2, \sqrt{2}\, x_1 x_2 \rangle$ that takes each two-dimensional data instance and transforms it into a latent space with three dimension. The transformed data set becomes linearly separable (Fig. 6.4, right panel). If $\boldsymbol{x}$ and $\boldsymbol{x}'$ are two vectors in original space, what is their dot product $\Phi(\boldsymbol{x})\Phi(\boldsymbol{x}')$ in transformed space?

$$\begin{aligned}
\Phi(\boldsymbol{x})\Phi(\boldsymbol{x}') &= \langle x_1^2, x_2^2, \sqrt{2}\, x_1 x_2 \rangle^\top \langle x_1'^2, x_2'^2, \sqrt{2}\, x_1' x_2' \rangle \\
&= x_1^2 x_1'^2 + 2 x_1 x_2 x_1' x_2' + x_2^2 x_2'^2 \\
&= (x_1 x_1' + x_2 x_2') \\
&= (\boldsymbol{x}^\top \boldsymbol{x}')^2
\end{aligned}$$

So, if, instead of using a dot product between two vectors $\boldsymbol{x}^\top \boldsymbol{x}'$ in formulation of SVM, we can replace it with a $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\top \boldsymbol{x}')^2$. This substitution is called a kernel trick. The kernel trick avoids the explicit mapping that is needed to, say, get a linear model to learn a nonlinear function or decision boundary. For all $\boldsymbol{x}$ and $\boldsymbol{x}'$ in the input space, certain functions $\kappa(\boldsymbol{x}, \boldsymbol{x}')$ can be expressed as an inner product in another space. The function κ is referred to as a kernel or a kernel function. Notice that the kernel trick allows us to never transform the data to the latent space, that is, never to use the transformation function $\Phi(\boldsymbol{x})$. In this way, the latent space could, in principle, have infinite dimensions, yet all we need is to compute the scalar product in this space. For any kernel, there is a corresponding function that transforms

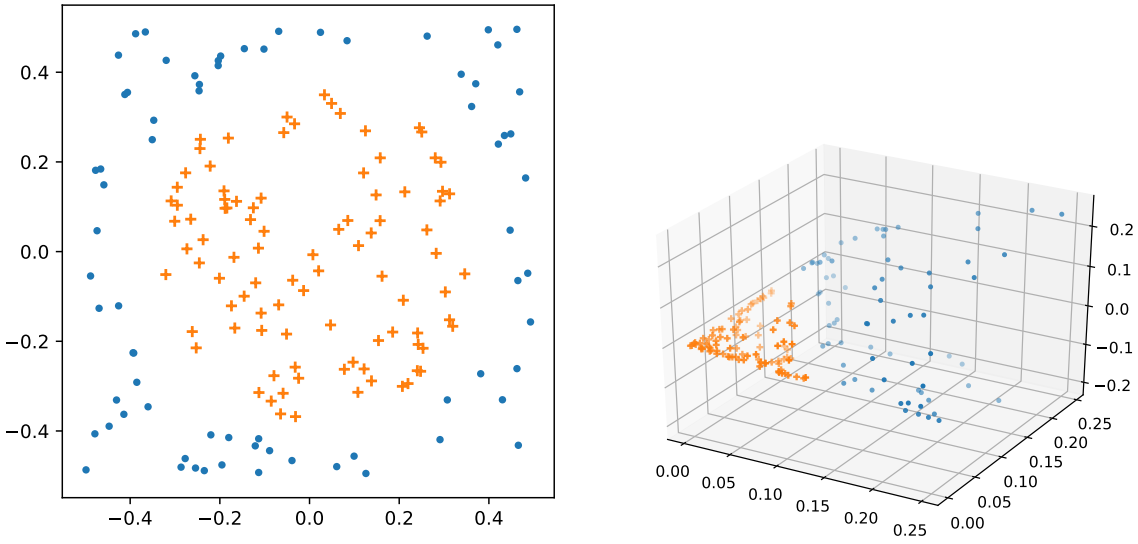


Figure 6.4: A two-feature binary classification data set (left), where the data points of different classes are not linearly separable. The same data set, where each point $\boldsymbol{x}$ was transformed through $\Phi(\boldsymbol{x})$, so that $\Phi(\boldsymbol{x}) = \langle x_1^2, x_2^2, \sqrt{2}\, x_1 x_2 \rangle$ (right).

the data from original space to latent space, and there are some kernels where the required dimensionality of latent space is infinite. We will discuss various kernel functions and their properties a bit later in this chapter.

Non-Linearly Separable Data

If the data is not linearly separable, which is expected from any real data that contains some noise, the support vector machine that we have defined so far does not have any solutions. For such cases, we need to introduce slack variables $\xi_i \geq 0$ such that slack is zero if the data instance is on or inside the correct margin boundary, or the slack is positive and equal to

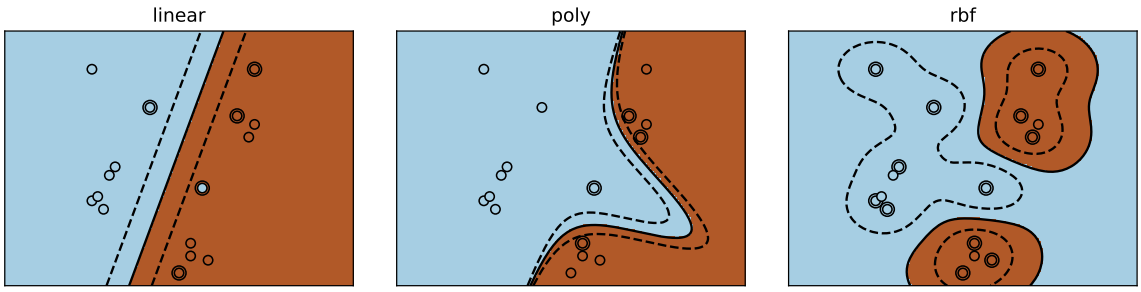


Figure 6.5: Classification in two-dimensional feature space shows decision boundaries of support vector machines with linear, polynomial ($n = 3$), and Gaussian kernel. Double-circled data points are support vectors. Code for figures by Gaël Varoquaux.

the distance to the corresponding margin. Notice that if $0 < \xi_i \geq 1$ the point lies inside the margin, but on the correct side of decision boundary. To solve for this problem, we replace the hard constrains with *soft margin constraints* with the new objective, where, as before, we would like to maximize the margin with minimal use of slack:

$$\min_{\boldsymbol{w},w_0,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\xi_i \;\; s.t.\; \xi_i \geq 0,\; y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i + w_0) \geq 1 - \xi_i,\; i = 1 \ldots N$$

The corresponding Lagrangian is

$$L(\boldsymbol{w}, w_0, \boldsymbol{\alpha}) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\alpha_i[y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i + w_0) - 1 + \xi_i] - \sum_{i=1}^{N}\mu_i\xi_i$$

After computing the corresponding gradients and replacing the results in the primal Lagrangian, the dual form of Lagrangian is the same as before,

$$\sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j \boldsymbol{x}_i\boldsymbol{x}_j$$

with the only exception of a box constraint $0 \leq \alpha_i \leq C$. Notice that the parameter C is a regularization parameter that controls the number of errors we are willing to tolerate on the training set. This parameter is commonly defined as $C = 1/(\nu N)$, where $0 < \nu \leq 1$ is the fraction of misclassified points we allow during the training phase. This version of the algorithm is called ν-SVM classifier.

Multi-Class Classification

We have defined support vectors classifiers for supervised learning on binary classification data. Unlike some other approaches, like softmax for logistic regression, SVMs do not have a natural extension that would be appropriate to treat multi-class data. Standard approaches that solve multi-class problems through inference of a set of binary classifiers include a one-versus-all approach and one-versus-one approach. However, the problem to use them is that SVM, in its original formulation, is not a probabilistic classifier.

Choosing the Value of Hyper-Parameters

Regularization parameters like C and ν are parameters of the algorithm, that is, hyper-parameters that need to be set before the inference of the model. To search for the most appropriate value, we can use approaches such as the search over a fixed set of parameters and estimating their appropriateness through cross-validated accuracy on the training data. We then use the parameter value with the highest estimated accuracy to infer the model from

the entire training set.

Hinge Loss

An alternative derivation of support vector machine comes from the utility of *hinge loss*, a variant of a loss function for binary classification, defined as:

$$L_{\text{hinge}}(y, \eta) = \max(0, 1 - y\eta) = (1 - y\eta)_+$$

Here, $\eta = f(\boldsymbol{x}) = \boldsymbol{w}^\intercal x + w_0$ can be regarded as "confidence" in choosing label $y = 1$. The overall objective is again to maximize the margin while minimizing the loss, therefore:

$$\min_{\boldsymbol{w}, w_0} \frac{1}{2} \|\boldsymbol{w}\|^2 + C \sum_{i=1}^{N} (-y_i f(\boldsymbol{x}_i))_+$$

This objective function is non-differentiable because of the max term. We can replace this term with slack variables, and request in the revised objective that the slack is minimized. The new objective function

$$\min_{\boldsymbol{w}, w_0, \xi} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{N} \xi_i \;\; s.t.\; \xi_i \geq 0,\; y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i + w_0) \geq 1 - \xi_i,\; i = 1 \ldots N$$

is exactly the same as the one derived above, in the section on the treatment of non-linearably separable data.

Kernelized k-Nearest Neighbors

k-nearest neighbor algorithms are based on distance, that is, they find the observations closest to the one we are predicting for. If using an Euclidean distance, this can be expressed in terms of inner products

$$\|x, x'\|^2 = \|x\|^2 + \|x'\|^2 - 2\langle x, x' \rangle = \langle x, x \rangle^2 + \langle x', x' \rangle^2 - 2\langle x, x' \rangle.$$

Thus, we can replace these dot products with a kernel, and in this way perform k-nearest neighbors in the feature space of a chosen kernel.

Support Vector Machines Regression

Here, we motivate SVM with the problem of kernelized ridge regression, and actually with other approaches discussed so far that use the kernel trick, where the solution is not sparse. More precisely, the solution of kernelized ridge regression $\hat{\beta} = \sum \alpha_i \boldsymbol{x}_i$ and subsequently the predictions $\hat{y}(x') = \sum \alpha_i \kappa(\boldsymbol{x}', \boldsymbol{x}_i)$ depend on all training observations.

A key idea of SVMs is to introduce sparsity through the loss function. Vapnik proposed the *epsilon insensitive loss function*:

$$L_\epsilon(y, \hat{y}) = |y - \hat{y}| - \epsilon \text{ if } |y - \hat{y}| \geq \epsilon \text{ and } 0 \text{ otherwise.}$$

Basically, anything that has an absolute error less than ϵ is not penalized. The objective function we want to minimize is then

$$J = C \sum L_\epsilon(y_i, \hat{y}_i) + \frac{1}{2}\|\beta\|^2,$$

where $\hat{y}_i = f(\boldsymbol{x}_i) = \beta^T \boldsymbol{x}_i + \beta_0$ and $C = \frac{1}{\lambda}$ is a regularization constant.

This optimization problem is non-differentiable because of the absolute value in the loss function. Typically, it is reformulated as a constrained optimization problem by introducing slack variables ξ:

$$y_i - f(\boldsymbol{x}_i) \leq +\epsilon + \xi_i^+,$$
$$y_i - f(\boldsymbol{x}_i) \geq -\epsilon - \xi_i^-.$$

Now we can reformulate the objective function.

$$J = C \sum (\xi_i^+ + \xi_i^-) + \frac{1}{2}\|\boldsymbol{\beta}\|^2.$$

This is a quadratic function of $\boldsymbol{\beta}$ and we need to minimize it, subject to the above constraints and $\xi_i^+ \geq 0$, $\xi_i^- \geq 0$. Standard quadratic programming black-box solvers can be used.

If it is not immediately clear what we did with the slack variables: First, we deliberately put the $f(\boldsymbol{x}_i)$ on the left-hand side of the inequality constraints above. This is to illustrate the fact that for observations that we can fit within ϵ, the constraint will immediately be true, and both ξ_i can be 0. For all other observations, one of the inequality constraints will be violated, but not both. So, one of the slack variables will be 0, and the other will be exactly the amount the fit exceeds ϵ. This is what forces the $f(\boldsymbol{x}_i)$ as close to y_i as possible, in the absolute sense; the only way to make ξ_i as close to 0 as possible, again, in the absolute sense, is to make the $f(x_i)$ as close to y_i as possible. That is, in the optimal solution, the sum of ξ will be exactly the total sum of all exceedances of ϵ), so by minimizing the one, we minimize the other. But we got rid of the absolute term.

The solution of the above optimization problem (**2012-Murphy**) has, not surprisingly, the form

$$\hat{\boldsymbol{\beta}} = \sum_i \alpha_i \boldsymbol{x}_i,$$

where $\alpha_i \geq 0$. Notice that this solution is typically sparse, as most $\alpha_i = 0$. The $\boldsymbol{x}_i$ where $\alpha_i > 0$ are again called *support vectors*. We should emphasize that the sparseness comes from

the objective function (ϵ). In practice, and depending on training data, we sometimes do get unlucky as it turns out that most x_i from the training data are support vectors.

SVM regression is typically solved in its dual form. The form above is the primal form. The two forms are the same as the optimization problem is convex. The dual formulation is also where the result comes that the solution is a linear combination of observations, very similar to kernelized ridge regression.

As with kernelized ridge regression, the prediction for a new observation is

$$\begin{aligned}\hat{y}(x') &= \hat{\beta}_0 + x'^T\hat{\beta} \\ &= \hat{\beta}_0 + x'^T \sum \alpha_i x_i \\ &= \hat{\beta}_0 + \sum \alpha_i x'^T x_i \\ &= \hat{\beta}_0 + \sum \alpha_i \kappa(x', x_i)\end{aligned}$$

So, we can also use the kernel trick here and do SVM regression in a feature space determined by the choice of kernel.

Kernel Density Estimation

Here, we will discuss on a substantially different kind of kernels, the so-called *smoothing kernels*. We will use the smoothing kernels to create non-parametric density estimates $p(\boldsymbol{x})$, as well as for creating generative models for classification and regression of the form $p(y, \boldsymbol{x})$.

A smoothing kernel is a one-argument function that satisfies the following properties:

$$\int \kappa(x)dx = 1,$$

$$\int x\kappa(x)dx = 0,$$

$$\int x^2\kappa(x)dx > 0.$$

A simple example of a smoothing kernel is a *Gaussian kernel*,

$$\kappa(x) = \frac{1}{(2\pi)^{\frac{1}{2}}} e^{-x^2/2}$$

and we can control the width of the kernel by introducing a *bandwidth* parameter h:

$$\kappa_h(x) = \frac{1}{h}\kappa\frac{x}{h}$$

We can generalize this kernel to vector-valued inputs by defining an RBF kernel:

$$\kappa_h(\boldsymbol{x}) = \kappa_h(\|\boldsymbol{x}\|).$$

Kernel density estimators are different from, say, Gaussian mixture models, a parametric density estimator that requires speficying K prototypes. An alternative is to allocate one cluster center per data point, so that the kernel density estimator, also called the *Parzen window density estimator* becomes

$$\hat{p}(\boldsymbol{x}) = \frac{1}{N}\sum_{i=1}^{N} \kappa_h(\boldsymbol{x} - \boldsymbol{x}_i).$$

The advantage of kernel density estimator over the parametric models is no need for model fitting and no need to pick the prototypes. However, we need to tune for the bandwidth, which can be performed by internal cross-validation. Notice also that the choice of h influences the bias-variance tradeoff. More smoothing decreases bias but increases variance, less smoothing decreases variance but increases bias. The optimal point is somewhere in between.

Chapter 7

Boosting in Machine Learning

Boosting is a type of ensemble learning that combines weak learners to produce a powerful committee of prediction models. From this view, it resembles bagging, which averages the output of many, hopefully, uncorrelated models to reduce the variance. Bagging, however, is fundamentally different and instead employs forward staging additive modeling, where the data feed into a modeling procedure at the selected stage depends on the output of a growing ensemble developed in preceding stages. We start with examining AdaBoost.M1, show that it minimizes exponential loss, and extend the concept to other loss functions, both for regression and classification.

Somehow surprisingly, boosting, an approach to construct and use an ensemble of predictive models, is easy to implement, is very powerful, and should be in anybody's toolbox when predictive accuracy is at stake. Similar, to bagging and random forests, boosting also relies on combination of weak learners and implements ensemble learning. But while in bagging we construct learners in parallel from sampled data, we construct boosting ensemble from data that is re-weighted according to the performance of the previously inferred learners in the sequence (Fig. 7.1). In this chapter, we start with intuitive building blocks of boosted prediction models, and then dive into some of the related formalism.

7.1 AdaBoost In a Nutshell

Let us quickly skim through some of the concepts that are important in boosting machine learning. We use boosting both in regression and classification, and in fact, we can adapt it for any kind of supervised machine learning tasks. In this section, though, we assume we will use it for classification, and will further – to simplify the introduction – constrain the task to binary classification with equal class distribution.

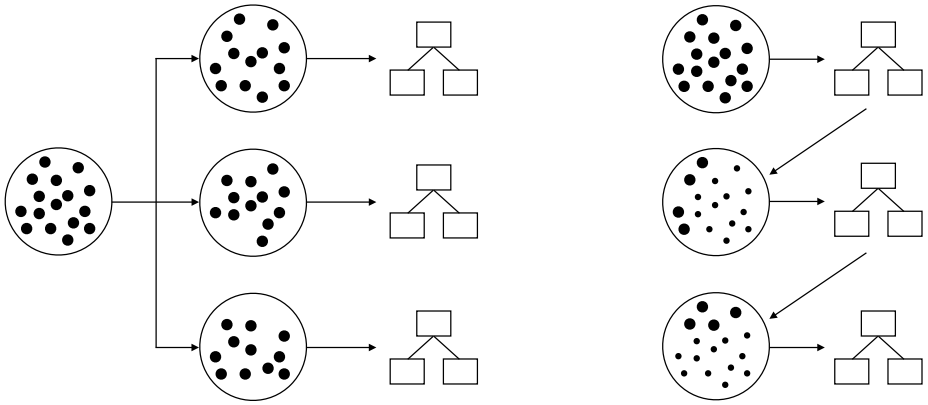


Figure 7.1: Bagging improves accuracy through "wisdom of the crowd", while boosting develops a sequence of models where each model specializes in prediction in the area where previous models in the sequence have failed.

Weak vs. Strong Classifiers

Consider an error rate, $\epsilon \in [0.0, 1.0]$ of a binary classifier with $y \in \{-1, 1\}$ for a problem with an equal class distribution. A weak learner would produce a weak classifier that would perform only slightly better than a random classifier. That is, for an assumed problem, its error on the training set would be just slightly below 0.5. On the other hand, in practice, we would like to develop strong classifiers with a very low error rate. Interestingly, and as a fundation for boosting, we can form a strong classifier from a set of weak classifier.

Consider a simple case of three classifiers, and let us denote them with $h^1(x)$, $h^2(x)$, and $h^3(x)$, where x is a vector that describes a data instance we would like to classify. If the three classifiers are substantially different, and they make wrong predictions in disjunct areas of the data space (Fig. 7.2), we can construct a simple, strong classifier as an ensemble that would join the output of the three classifiers in the following way:

$$H(x) = \text{sign}(h^1(x) + h^2(x) + h^3(x)) \tag{7.1}$$

The prediction of the classifier $H(x)$, under the assumption that the areas of erroneous prediction of each of the three classifiers do not overlap. It would be great if we could construct such classifiers, but in reality, of course, the areas where classifiers get it wrong would overlap, and hence a simple procedure for their ensembling would not necessary be beneficial.

We may try to develop classifiers that are of the type from Fig. 7.2, that is, where a classifier attempts to be correct in the area where a previous classifier, or a set of previous classifiers were wrong. We will do so by changing the data. We will build the first classifier, h^1 on the entire data set, but then try to distort the data to exaggerate on the data space where h^1 makes mistakes. A simple procedure we can use is through training data instance

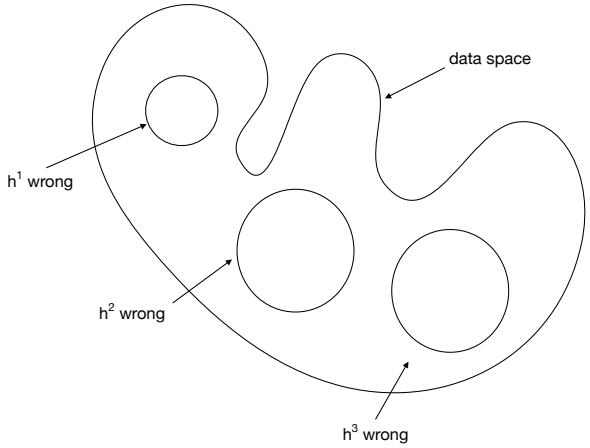


Figure 7.2: A hypothetical data space and three classifiers with disjunctive areas where their prediction is incorrect.

weighting: we will increase the weight of the data instances that were misclassified by h^1, thus instructing a new classifier h^2, developed on such distorted data set, to correctly classify the data instances which were misclassified by h^1. If we follow the idea from the previous paragraph and try to develop three different classifiers, we now can train the third one on a data set where the weights of the data instances would emphasize those x where $h^1(x) \neq h^2(x)$.

Just a quick note: thus far, our classification inference algorithm never considered data instance weights. It is not difficult to change the inference algorithm to do so. For instance, in inference of trees, instead of counts of the data instances we would sum up the weights. Where changing the training algorithms to handle weights is not possible, we could address the exaggeration with oversampling of the target data instances.

Hierarchy of predictors

If constructing a classifier by ensembling the predictors works well, we could improve those predictors as well through ensembling them from another, nested set of classifiers. By doing this we would shrink the area where the predictors make mistakes, making it easier for a base set of predictors not to overlap in their misclassification zones. Note that in theory we may continue building such hierarchy to arbitrary level, whereas practically and in most cases development of one serious of predictors suffices.

A Weak Classifier and Initial Data Instance Weights

We hinted about the use of weak classifiers, but have not yet introduced an algorithm to construct them. Here it is: a *decision tree stump*. This learning algorithm uses a decision tree inference algorithm, but limits the depth of the tree to one, that is, besides the root node there is only a next level with a set of leaves. Stumps use only one feature to decide which leave

node to use for prediction. In practice, we can employ shallow trees that are deeper than one level, but for an example of a weak classifier a decision tree stumps would do as well. Also notice that while our introduction talks about classifiers, we can use trees for regression as well and hence we will also be able to develop bootstrap predictors for regression problems.

Note that we can use boosting with any kind of prediction models. But due to speed and simplicity, and since they are weak classifiers, we use shallow trees in practical implementations.

The error rate of a classifier can be expressed as:

$$\epsilon = \sum_{\text{wrong}} \frac{1}{N}$$

where N is the number of data instances in the training set. Initially, all instances will have equal weight and since we would like the weights to form a distribution and hence sum to 1, the weights for each data instance i for the first classifier, that is h^1, are:

$$w_i^1 - \frac{1}{N}$$

Using the formulation above, we can express the error rate as a function of weights:

$$\epsilon = \sum_{\text{wrong}} w_i$$

Ensembling Classifiers

A more general way to combine classifiers, rather than summing their outputs, would be to construct their weighted combination:

$$H(x) = \text{sign}(\alpha^1 h^1(x) + \alpha^2 h^2(x) + \ldots + \alpha^T h^T(x))$$

where T is a number of weak classifiers which we would like to ensemble. This is again different from bagging. Bagging counts on wisdom of the crowds, while here we are summing up on series of classifiers which are different in the areas of misclassification and where we will weight them according to their error. In boosting, the wisdom of the crowds becomes the wisdom of the experts that specialize in different area of the data space.

The overall procedure to construct our ensemble is then as shown in Table 7.1.

Suppose that we define the weights as:

$$w_i^{t+1} = \frac{w_i^t}{z} \exp\left(-\alpha^t h^t(x_i) y_i\right)$$

where z is a normalizing factor so that the weights form a distribution and they sum to 1. The

Table 7.1: An overall structure of a bootstrap learner

$t \leftarrow 0$
$w_i^t \leftarrow \frac{1}{N}$
while $t \leq T$
    $t \leftarrow t + 1$
    pick h^t that minimizes ϵ^t
    pick α^t
    calculate w^{t+1}

equation above comes from mathematical convenience, but we will show that while initially proposed for boosting, they have a more intuitive and simpler interpretation. Notice that where the prediction $h^t(x_i)$ and the true class y_i agree, and assuming that the factors α are positive, the future weight w_i^{t+1} of the instance i is lowered. When prediction and the true class do not agree, the value of the future weight w_i^{t+1} is raised.

We would like to minimize the error for the ensemble. It turns out that to do this (**FreundSchapire1997**) we need to set the weight of the classifier according to the following:

$$\alpha^t = \frac{1}{2} \ln \frac{1 - \epsilon^t}{\epsilon^t} \tag{7.2}$$

$$= \ln \sqrt{\frac{1 - \epsilon^t}{\epsilon^t}} \tag{7.3}$$

If we combine this expression with the update for the data instance weights, and note that the product $h^t(x_i)y_i$ equals 1 for correct classification and -1 for misclassification, than we obtain

$$w_i^{t+1} = \frac{w_i^t}{z} \times \begin{cases} \sqrt{\frac{\epsilon^t}{1-\epsilon^t}}, & \text{if correct} \\ \sqrt{\frac{1-\epsilon^t}{\epsilon^t}}, & \text{if wrong.} \end{cases}$$

The weights have to add up to 1, thus we need to set the normalization factor to

$$z = \frac{\epsilon^t}{1 - \epsilon^t} \sum_{\text{correct}} w_i^t + \frac{1 - \epsilon^t}{\epsilon^t} \sum_{\text{wrong}} w_i^t$$

Notice, however, that the sum of the weights of the missclassified items is the error rate, that is, $\sum_{\text{wrong}} w_i^t = \epsilon^t$. Similarly, the sum of the weight over correct classifications is one minus the error rate, that is, $\sum_{\text{correct}} w_i^t = 1 - \epsilon^t$. Hence,

$$z = 2\sqrt{\epsilon^t(1 - \epsilon^t)}$$

Combining the expression for weight update and normalization, we obtain:

$$w_i^{t+1} = \frac{w_i^t}{2} \times \begin{cases} \sqrt{\frac{1}{1-\epsilon^t}}, & \text{if correct} \\ \sqrt{\frac{1}{\epsilon^t}}, & \text{if wrong.} \end{cases}$$

Now, if we add the weights for the correct classifications, we get

$$\begin{aligned} \frac{1}{2}\frac{1}{1-\epsilon^t}\sum_{\text{correct}} w^t &= \frac{1}{2}\frac{1}{1-\epsilon^t}(1-\epsilon^t) \\ &= \frac{1}{2} \end{aligned}$$

Similar is true for missclassifications,

$$\begin{aligned} \frac{1}{2}\frac{1}{\epsilon^t}\sum_{\text{wrong}} w^t &= \frac{1}{2}\frac{1}{\epsilon^t}\epsilon^t \\ &= \frac{1}{2} \end{aligned}$$

In other words, the boosting algorithm that we have described distrubutes the same amount of weights to correct and incorrect classification. If incorrect classifications will be in minority, which we hope they will, the misclassified examples will carry higher weights for the next instance of the training algorithm in the sequence. The procedure described here is called AdaBoost for adaptive boosting and was introduced in (**FreundSchapire1997**). Interestingly, while the math for Eq.7.2 was worked out in the original publication, the interpretation with notion that the weights for both misclassification and classification sum to one half, which does in a way simplify the update as well, was not noticed at the time.

Chapter 8

Artificial Neural Networks

Neural networks combine incredibly simple computational units to solve possibly the hardest problem in machine learning: discovery of feature interactions. Initially inspired by architectures of neurons and brains, they model these very loosely but equally build their power on the number of connections and parallel processing. In this lecture, we provide an elementary introduction to artificial neural networks. We focus on motivation, describe inspirations from biology, and delve into perceptron and its failures. Next, we introduce the artificial neuron and the combinations of neurons within the standard feed-forward neural network. We show how to compute the gradient of the cost function with respect to the parameters of the model. Computation of the gradients uses chain rules and led to the algorithm for weight updates called backpropagation. We finish with some ideas on optimization and avoidance of overfitting, and mention, but do not delve into, other types of neural networks.

The computational motivation for introduction of neural networks are to learn *hard* concepts. For instance, consider a concept depicted in Fig. 8.1: the classification rule that separates the classes needs to model interaction between the two features. Assuming the other features or combinations are not as informative as a combination from Fig. 8.1, and supposing that the training data set that contains 1.000 features, one would need to search among 999.000 feature pairs to find the informative pair. If a concept involves a feature triplet, the search size is larger and contains 332.334.000 triplets. Addressing such problem directly, through exhaustive search, is computationally not feasible.[1]

A possible alternative to exhaustive search of feature interactions are models that incorporate feature interaction, and that can possibly model any kind of interaction between any of the features. The problem we are facing is of course the data. Such models need substantial, if not huge amount of data for training to avoid overfitting. But if data is available – and

[1] Actually, and depending on a data set, it is also statistically not feasible, but we will leave this problem aside.

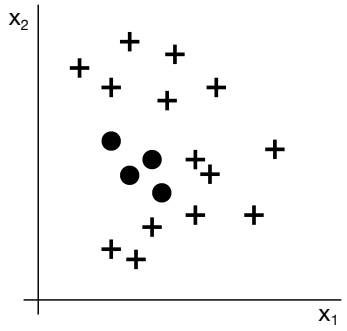


Figure 8.1: An example of a hard classification concept, where the classifier would need to recognize the interaction between two features, x_1 and x_2. Concepts like these are especially hard to model in the presence of many other features, which can be to a degree related to the class.

sometimes it is – then we better define the model that we can use in such cases. Notice that we are moving into direction where such models may be hard to explain, but this also is an issue we will deal with later, in our next chapter.

8.1 Motivation from biology

We start with disclaimer: artificial neural networks are very simplistic model of a brain, or any biological neural network. Biology is by orders of magnitude more complex: an axon, that is considered in artificial networks as a wire, has been studies in numerous projects and its structure and physiology has been reported in books of thousands of pages. With this warning, though, consider a realistic model of a neural cell in Fig. 8.3. Neural cell emits electric signals through the axon, but only when the potential in the cell body reaches a certain level, called *action potential*. The electrical potential of the body is a sum of potentials in the dendrites, and and this in turn depend on potential evoked from connected cells. Connections are established through synapses, which chemically transmit the electrical signal from the axon tips (inputs) to the dendrites. Neural cells thus, in a very simplified way, sum up the input signals and fire when the sum reaches specific threshold, emitting the signal through the axon and establishing a network with connected cells.

Human brain contains 8×10^{10} neurons, where, on average, each neuron is connected to 10.000 other neurons. The resulting network is huge and contains 10^{15}, that is, 1.000 trillion connections.

Synapses adapt, adjusting the quantity of required transmitters and available receptors, thus implementing one of the mechanisms for plasticity of the brain and learning. Synapses, on the other hand, implement chemical transmission of the signals and are thus slow, but they are many and function in parallel.

The brain is modular. Different regions perform different functions. Experimentally this

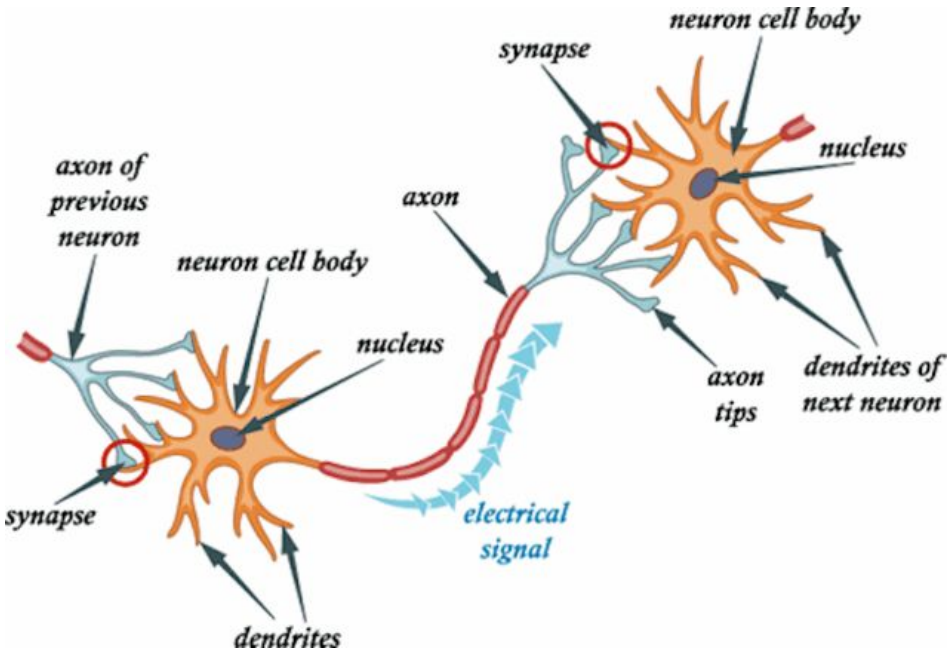


Figure 8.2: The structure of a neural cell, showing means of communication between two connected cells.

was observed in patients where local damages had specific effects. But the plasticity was observed as well: regions of brains can take over a specific function after the brain region originally carrying out this function was damaged. Damage in one region can therefore be alleviated through specialization of another region.

The idea of the network of neurons, neurons summing up the input signals, adaptivity of synapses which can weight the input, and a activation function implemented by a body of a neural cell are all concepts that are modeled by artificial neural networks. Brain plasticity and redundancy are modeled as well, and specifically addressed in larger, deeper neural networks.

8.2 Idealized neuron

Idealized neuron is a model of a neuronal cell with complicated details removed (Fig. **??**). It performs simple mathematics, resorts to basic principles, and is wrong since the communication is not binary. The simplest model sums-up the inputs through a weighted sum, where w_i is a weight for i-th input:

$$z = b + \sum_i x_i w_i, \tag{8.1}$$

and the output of the *linear neuron* is

$$\hat{y} = z \tag{8.2}$$

Other types of neurons incorporate other *activation functions*, that is, functions that take

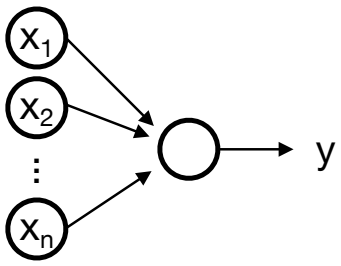


Figure 8.3: An idealized neuron with x_i representing its inputs and y an output variable.

a weighted sum of the inputs to compute the output of a neuron. Popular examples include *binary threshold neuron*,

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{else} \end{cases} \tag{8.3}$$

rectified linear neuron, or RELU,

$$\hat{y} = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{else} \end{cases} \tag{8.4}$$

and *sigmoid neurons*,

$$\hat{y} = \frac{1}{1 + e^{-z}} \tag{8.5}$$

Notice that a sigmoid neuron looks very much like one of the classification models we have already studied. Which one, and what are the differences, if any?

8.3 Perceptrons

Training with a single linear neuron, that is, a neuron implementing $z = w^\intercal x$ and a related classifier,

$$\hat{y} = h(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{else} \end{cases} \tag{8.6}$$

was popular in 1960s under the name perceptron. The perceptrons (Fig. 8.4, algorithm in Table 8.1) were proposed by Frank Rosenblatt, one of the pioneers of artificial intelligence, and were wrongly presented as a very powerful tool. In really, learning with perceptrons was very weak, could not handle noise, but is still historically interesting.

Notice that the perceptron training (Table 8.1) actually implements a stochastic gradient descent with a batch size of one and a learning rate of one. The training would succeed in cases where the classes are linearly separable, but fail otherwise. In linearly separable cases there would be infinitely many solutions where perceptron training would converge to a particular one. The process would fail under any interaction between input variables,

Table 8.1: Perceptron's learning procedure

initialize w
repeat
    choose (x, y) from the training set
    if $h(z) \neq y$
        if $h(z) = -1$
            $w \leftarrow w + x$
        else
            $w \leftarrow w - x$

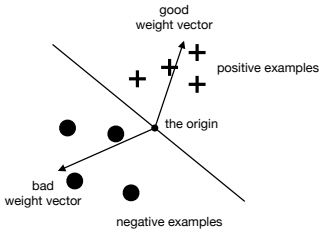


Figure 8.4: Several concepts in perceptron learning.

where a typical example would be that of XOR. Obviously, there, we would need hierarchy of concepts and a nested perceptrons to model interactions.

8.4 Artificial neural networks

Artificial neural network is a network of artificial neurons. Output of one neuron is fed into inputs of a set of neurons. While there is no limitation on the structure of the network, the typical network starts with a layer of input features, continues with a layer of neurons, and then with the next layers, where each layer is fully connected. That is, a neuron at layer L receives inputs from all neuron at previous level, level $L - 1$. The last layer is special, and set according to the problem at hand. For instance, for regression, the last layer may include only one neuron, whose activation models variable y. For classification, the last layer may have as many neurons as there are class values, where each activation reports on a class probability. We refer to all layers between an input layer and an output layer as *hidden* layers.

Just like with other machine learning techniques, we have to set a cost function, and define a procedure to optimize the weights for each of the neuron accordingly. This procedure is known as *back-propagation*, and actually implements a gradient descent. We develop the mathematics for it in the next section.

8.5 Back-propagation algorithm

We start with some conventions. We assume that all units of the neural network can take value between 0 and 1. We refer to this value as *activation* and will denote it with a. In the previous text, when introducing a single neuron, we have denoted it with $\hat{y}$, which we will now reserve for the output of the entire network. We will also assume that the output of the neural network corresponds either to the value of regression problem, or to class probabilities, where of form of softmax regression is used to guarantee that the class probabilities sum to one.

To introduce the notation, consider a simple neural network with one input feature x and one output $\hat{y}$, and one neuron in each of the two hidden layers (Fig. 8.5).

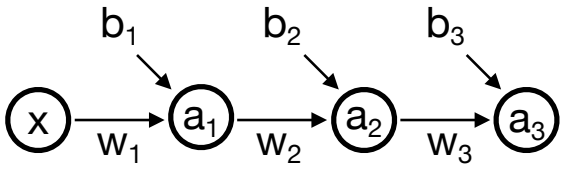


Figure 8.5: An example of a network with a single input and output and one neuron per layer.

Let us, for simplicity, assume we are dealing with only one example in the training set, and define a cost function as a squared error:

$$J(w_1, b_1, \ldots, w_3, b_3) = (a_3 - y)^2 \tag{8.7}$$

Until now, we have used the indices to denote the weights w, the offsets b and activation at each layer. Later, when dealing with more than one neuron at each layer, a notation which denotes the layer number will come handy. Apart from the layer with the input value, our simple network from Fig. **??** has three layers, $L = 3$. The activation a_3 belongs to the third layer and we will alternatively denote it with $a^{(L)}$. Similarly, $a^{(L-1)}$ will denote a_2. Same goes with other parameters and activation values. We can thus write that the weighted sum of inputs for neuron at layer L is equal to

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}, \tag{8.8}$$

the activation of that neuron is

$$a^{(L)} = \sigma(z^{(L)}), \tag{8.9}$$

and the cost function

$$J(w_1, b_1, \ldots, w_3, b_3) = (a^{(L)} - y)^2 \tag{8.10}$$

While we can use any activation function here, we will sigmoid activation function for conve-

nience.

To implement gradient descent, we need to find how does a cost function J depend on the values of the parameters of the neural network. For instance, how does J depend on the weight w_3, that is, the weight $w^{(L)}$? We can use a chain rule to compute the partial derivate, and while doing so, it helps us to examine the dependencies as depicted in Fig. 8.6:

$$\frac{\partial J}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \times \frac{\partial a^{(L)}}{\partial z^{(L)}} \times \frac{\partial J}{\partial a^{(L)}} \tag{8.11}$$

$$= a^{(L-1)} \times \sigma(z)(1 - sigma(z)) \times 2(a^{(L)} - y). \tag{8.12}$$

We can interpret the terms in this equation as $a^{(L-1)}$ denoting the power (or the weight) of the precious layer, $\sigma(z)(1 - sigma(z))$ denoting a derivative of an activation function, and term $2(a^{(L)}) - y)$ as the error of the prediction.

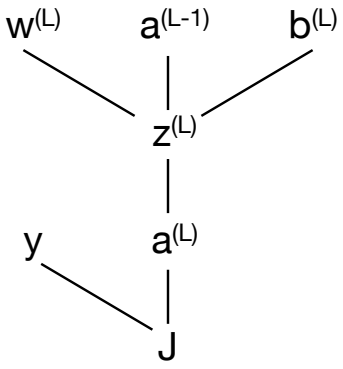


Figure 8.6: Dependency tree of the cost function J on some of the parameters from the neural network from Fig. 8.5.

Above we have assumed we are dealing with only one training example. To generalize the above assertions for a set of training instances, we first need to modify the definition of the cost function, which now becomes:

$$J = \sum_{j=0}^{N} \left(a_j^{(L)} - y_j \right)^2 \tag{8.13}$$

Notice that the only change when computing partial derivative of J according to $w^{(L)}$ is in computation of third term, $\frac{\partial J}{\partial a^{(L)}}$, which now becomes a sum of partial derivates.

Let us consider now a more general type of network, with a number of neurons at each layer, and the number of neurons at the output layer. We again restrict the training set to only one data instance. Consider a fragment of a network from Fig. 8.7, which depicts the relation

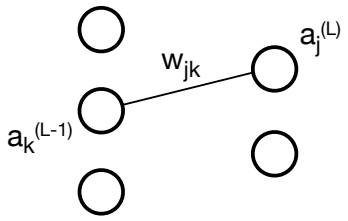


Figure 8.7: A fragment of a neural network exposing the relation between activation of the k-th neuron in layer $(L-1)$ and activation of a j-th neuron at leyer L.

between activation of the k-th neuron in layer $(L-1)$ and j-th neuron at leyer L. Notice that

$$z_j^{(L)} = \sum_i w_{ji}^{(L)} a_i^{(L-1)} + b_j^{(L)}, \tag{8.14}$$

$$a_j^{(L)} = \sigma(z_j^{(L)}), \tag{8.15}$$

$$J = \sum_j^{(} n_{L-1}(a_j^{(L)} - y_j)^2. \tag{8.16}$$

We assume the indices run from 0, replace the intercepts b for each k-th neuron with w_{0k}, and denote the number of neurons at layer L with n_L. Notice also that the weights have now two indices. The weight w_{jk} is a weight for a j-the neuron for the output of the k-th neuron from the previous layer. For a gradient descent, we again need to compute the change this weight invokes to the cost function,

$$\frac{\partial J}{\partial w_{jk}} = \sum_j \frac{\partial z_j^{(L)}}{\partial w_{jk}} \times \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \times \frac{\partial J}{\partial a_j^{(L)}} \tag{8.17}$$

The partial derivatives of the first two terms in the product are straightforward, and stem directly from the expression for $z_j^{(L)}$ and $a_j^{(L)}$. But the partial derivative in the last term is new, and we can break it down to

$$\frac{\partial J}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L} \frac{\partial z^{(L)}}{\partial a_k^{(L-1)}} \times \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \times \frac{\partial J}{\partial a_j^{(L)}} \tag{8.18}$$

With these two expressions, we can now chain back through the network and compute the influence of every of the network's parameters, thus providing means for the gradient descent. The procedure is known under the name *back propagation*, which, intuitively:

- converts discrepancy between output and and target into error derivative,
- computes the error derivatives in each hidden layer from error derivatives of the next layer,

- uses error derivative with respect to activations to get error derivative with respect to weights.

Let us express our derivations and back-propagation procedure in a matrix form. We will assume that the input data includes m data instances and n features, and will for convenience add a first column of 1's to the input data matrix, thus increasing its size to $m \times (n+1)$ and denoting this matrix with $\boldsymbol{X}'$. Let this matrix represent the activations of the neurons in the first, input layer:

$$\boldsymbol{A}^{(1)} = \boldsymbol{X}' \tag{8.19}$$

Then we can write the equations for the second layer:

$$\underset{m\times n_2}{\boldsymbol{Z}^{(2)}} = \underset{m\times n_1}{\boldsymbol{A}^{(1)}}\ \underset{n_1\times n_2}{\boldsymbol{W}^{(2)}} \tag{8.20}$$

$$\underset{m\times n_2}{\boldsymbol{A}^{(2)}} = \sigma\left(\underset{m\times n_2}{\boldsymbol{Z}^{(2)}}\right) \tag{8.21}$$

For the general l-th layer we can write:

$$A^{(l)} = \sigma\left(A^{(l-1)}W^{(l)}\right) \tag{8.22}$$

We start with the last layer,

$$\frac{\partial J}{\partial \boldsymbol{W}^{(L)}} = \frac{\partial \boldsymbol{Z}^{(L)}}{\partial \boldsymbol{W}^{(l)}} \times \frac{\partial \boldsymbol{A}^{(L)}}{\partial \boldsymbol{Z}^{(L)}} \times \frac{\partial J}{\partial \boldsymbol{A}^{(L)}} \tag{8.23}$$

where computing the first partial derivative is straightforward. Let us represent the product of the last two terms with d:

$$\underset{m\times n_L}{\mathrm{d}^{(L)}} = \left(\boldsymbol{A}^{(L)} - \boldsymbol{Y}\right) \odot \boldsymbol{A}^{(L)}\left(1 - \boldsymbol{A}^{(L)}\right) \tag{8.24}$$

$$\frac{\partial J}{\partial \boldsymbol{W}^{(L)}} = \frac{1}{m}\left(\boldsymbol{A}^{(L-1)}\right)^{\intercal} \times \mathrm{d}^{(L)} \tag{8.25}$$

where with $\odot$ we denote element-wise product. In a similar way we compute the partial derivative $\frac{\partial J}{\partial \boldsymbol{A}^{(L-1)}}$ as a function of partial derivate of $\frac{\partial J}{\partial \boldsymbol{A}^{(L)}}$, and repeat the computation of the above two equations for lower levels of the network.

8.6 Bag of tricks

As with training of other classifiers and regression models, like linear and logistic regression, there are technique which can speed up and improve convergence of the training of neural networks. These, in brief, include:

- To aim to prevent overfitting, we can use regularization, and include the sum of all the

weights of the network in a cost function. This procedure was also known as a *weight decay*, where after the computation of each weight these well further scaled down by some factor, say 0.99. Notice that if using sigmoid activation function, the small weights meant that we operate at the linear part of sigmoid at thus with regularization aim at derive close-to-linear model, thus simplifying it.

- Neural networks include many parameters, the problem we can alleviate through *weight sharing*. That is, neurons at some level would share some of the weights.
- Training of neural networks assumes large training data set. We can use all of the data instances from the training data, but for large data sets use *mini-batch* gradient descent with adaptive learning rate and use of momentum in the optimization.
- To further avoid overfitting, we can use *dropout*, where we randomly drop neurons along with their connections from the neural network during the training. Dropout prevents neurons from co-adapting. During training, dropout samples form an exponential number of different "thinned" networks, thus also, in some way, introduce ensembling.

These and other tricks and their details go beyond our discussion in this chapter. They are all implemented in today standard packages for neural network training. While one of the homeworks of this course will be on neural network implementation, this, and the derivation of related equations and their implementation would almost certainly be a once-in-a-lifetime attempt useful only for educational reasons.