Machine Learning for Molecules and Materials

Johannes C. B. Dietschreit

October 1, 2024

Contents

1 Introduction

In the natural sciences and engineering, a significant part of our work revolves around prediction, modeling, and control. We predict market trends, build models of complex systems or components, and automate tasks such as driving a vehicle. The methods for tackling these problems have evolved significantly over time. This course will provide an introduction to addressing these issues (with a clear focus on chemical problems) through the lens of machine learning. Initially, we will focus on prediction, as it is the most straightforward machine learning problem to grasp and effectively illustrates the core concepts of machine learning.

There are various types of predictions we can make. For example, we can predict future events like market movements or weather conditions, anticipate human actions such as purchasing behavior, or predict pedestrian behavior for autonomous vehicles. In addition, we can predict unknown properties, such as whether a chemical is water-soluble, identify objects in images, or translate sentences between languages. A common challenge in these prediction problems is that crafting a solution using explicit rules or code is extremely difficult. Instead, it is much more feasible to frame these issues as machine learning problems by providing examples of the desired outcomes. For example, using historical weather data, we can demonstrate a known condition (weather a week ago) and the subsequent outcome (e.g., sunny or cloudy the next day). This approach of learning solutions from examples is what makes machine learning so powerful and widespread.

In this course, we will explore how to mathematically express these "learning-from-examples" machine learning problems, determine which algorithms to use, and understand the necessary assumptions for effective performance. We will begin with the simplest prediction problem: binary classification. Here, the goal is to learn to classify examples, such as biological samples, images, or text, into two distinct categories.

1.1 Learning a Classifier

Today's data are often in unstructured textual forms like news articles, books, product reviews, emails, and tweets. Manually filtering this information is impractical, so we use automated tools to do it for us, similar to spam filters. Rather than creating cumbersome "if-then" rules, it is easier to label a small set of documents as positive (wanted) or negative (unwanted) and let a machine learning algorithm learn the rules from these examples. This labeled set is called the training set and our goal is to learn a classifier to apply to new documents.

However, computers do not understand "documents" directly. We need to represent each document as a feature vector, where each coordinate measures something useful about the document. A common method is the "bag-of-words" representation, where each coordinate corresponds to a word in the vocabulary, and its value is the word's frequency in the document. The same method of vector creation must be used for both the training set and new documents to ensure consistency.

This approach is not limited to text. For example, to classify tissue samples as containing tumor cells or not, we can use biological assays to create feature vectors based on gene expression levels. Similarly, for images, instead of using raw pixel values, we can use features such as edges and

textures to improve the classification accuracy.

In essence, any problem that can be mapped to labeled feature vectors can be tackled with machine learning algorithms. The flexibility of this method allows it to be applied across various domains, from documents to biological samples to images, enabling us to predict labels for new examples based on the training set.

1.2 Basic Components and Notation

1.2.1 Summary of Notation

Scalar values are denoted with lowercase letters (e.g., c), vectorial quantities are displayed in bold and lowercase (e.g., $\mathbf{x}$), and matrices as bold and uppercase letters (e.g., $\mathbf{A}$). In addition, special symbols are reserved for:

Features	Vector $\boldsymbol{\phi}(\mathbf{x}^{(i)})$ of dimension d. Can be reals or integers.
Label	Integer or real $y^{(i)}$, usually a scalar.
Labeled Data	Set of n tuples $(\mathbf{x}^{(i)}, y^{(i)})$.
Unlabeled Data	Set of n feature vectors $\boldsymbol{\phi}(\mathbf{x}^{(i)})$ that may have unknown labels y.
Data generation process	The unseen process that produces the label for a given state $\mathbf{x}$, this can be experimental measurements, other computer simulations, human annotation of pictures, etc.
Model	A function $\hat{h}(\mathbf{x})$ that returns the prediction $\hat{y}$ for a given sample $\mathbf{x}$.
Predictions	$\hat{y}$, the predicted output for a given input $\mathbf{x}$ (or more precisely $\boldsymbol{\phi}(\mathbf{x})$).
Parameters	The entirety of parameters of a model h are denoted with $\boldsymbol{\theta}$. If the model is linear, then we use $\mathbf{w}$ for the weights and b for the bias.

1.2.2 Feature vectors, training set

To look at these classification problems more formally, we will use $\boldsymbol{\phi}(\mathbf{x}) = [\phi(\mathbf{x})_1, \ldots, \phi(\mathbf{x})_d]^\top \in \mathbb{R}^d$ to denote each feature (column) vector of dimension d. We will also use $\mathbf{x}$ to denote the original object (e.g., sample, image, document). If we do not use any special technique, for example, when $\mathbf{x}$ is already a geometrical point, then simply $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$.

In classification, each training example $\mathbf{x}$ is associated with a binary label $y \in \{-1, 1\}$. For new examples, we will have to predict the label. Assuming that we have n training examples available to learn from, we will index the training examples with superscripts, $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n)}$ and similarly for the corresponding labels $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$. All that the classification method knows about the problem is the training data as n pairs $(\mathbf{x}^{(i)}, y^{(i)}), i = 1, \ldots, n$. Let us call this training set S_n where the subscript n highlights the number of labeled examples we have.

1.2.3 The Set of Classifiers

A classifier h is a mapping from feature vectors to labels: $h : \mathbb{R}^d \to \{-1, 1\}$. When applying the classifier to a particular example $\mathbf{x}$, we write $\hat{y} = h(\mathbf{x})$ as the predicted label. The notation $h(\mathbf{x})$ is a shorthand for $h(\boldsymbol{\phi}(\mathbf{x}))$ as we always apply the model to the machine-processable feature vector.

Any classifier divides the space $\mathbb{R}^d$ into positive and negative regions according to the predicted label. There could be a number of distinct regions that are marked as positive (that is, $\mathbf{x}$ such that $h(\mathbf{x}) = +1$). We are usually confronted with a set of classifiers $\mathscr{H}$ (a set of hypotheses about the rules that govern how labels are related to examples), and then select one $\hat{h} \in \mathscr{H}$ based on the training set S_n. The goal is to select $\hat{h} \in \mathscr{H}$ that would have the best chance of correctly classifying new examples that were not part of the training set. Note the key difficulty here. All the information we have on the classification problem is the training set S_n, but we are actually interested in doing well on examples that were not part of the training set. In other words, in the end we are interested in the prediction.

1.2.4 Training Error and Learning Criterion

Let us define how we evaluate whether a specific classifier is good in light of the training examples we have. For example, we can always calculate the training error of a classifier h. This is simply the fraction of errors that the classifier makes on the training examples. More formally,

$$\epsilon_n(h) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}[h(\mathbf{x}^{(i)}) \neq y^{(i)}] \,, \tag{1.1}$$

where $\mathbb{1}[\text{true}] = 1$ (when h makes an error) and $\mathbb{1}[\text{false}] = 0$ (when h is correct). We can evaluate the training error $\epsilon_n(h)$ for any classifier $h \in \mathscr{H}$. Of course, there may be many classifiers that achieve a low or even zero training error for any specific training set S_n. We typically have to incorporate additional guidance known as regularization to decide which classifier to prefer in light of the training set.

1.2.5 A Learning Algorithm

A learning algorithm is a mapping from the training set S_n and the set of classifiers $\mathscr{H}$ to a particular classifier $\hat{h} \in \mathscr{H}$. In other words, its job is to select the classifier that we want based on the training set. Mostly, we will use learning algorithms that are optimization algorithms whose goal is to minimize the learning criterion that we pose, such as the training error, minimized with respect to the choice of the classifier. However, since our goal is prediction rather than just performance on the training set, the best optimization algorithm is not necessarily the best learning algorithm. Indeed, a classifier that does not minimize the training error may perform better out of sample (outside of the training distribution), i.e., it may be more accurate on examples that were not part of the training set (which is what we want). The learning algorithm would ideally be tailored to find such classifiers.

1.2.6 Test Error and Generalization

The prediction goal is always to minimize the test error

$$\epsilon_{n'}(h) = \frac{1}{n'} \sum_{j=1}^{n'} \mathbb{1}[h(\mathbf{x}^{(j)}) \neq y^{(j)}] \,, \tag{1.2}$$

which is just like the training error but evaluated on the basis of a distinct set of n' completely new examples. The problem is that while we wish to minimize $\epsilon(h)$, we cannot evaluate it since the test examples (e.g., future weather patterns) are not available at the time that we have to commit to a classifier $\hat{h}$.

Although the situation is not entirely without hope, it is reasonable to assume that the test samples are largely similar to the training examples. Specifically, we can envision both training and test samples, along with their labels, as being randomly selected from a vast underlying pool of labeled data (formally, a joint distribution). Consequently, having more training examples tends to help because the training error (which we can measure) is more likely to reflect the test error (which we aim to minimize). We refer to the classifier's performance on future test samples as *generalization*. In other words, a classifier that generalizes well will perform on new examples in a manner similar to its performance on the training set, which can be measured. Creating learning criteria and algorithms that yield classifiers with good generalization involves multiple factors: the choice of feature vectors, the number of training examples (n), the selection of the set of classifiers $\mathscr{H}$, and how the learning algorithm selects $\hat{h} \in \mathscr{H}$.

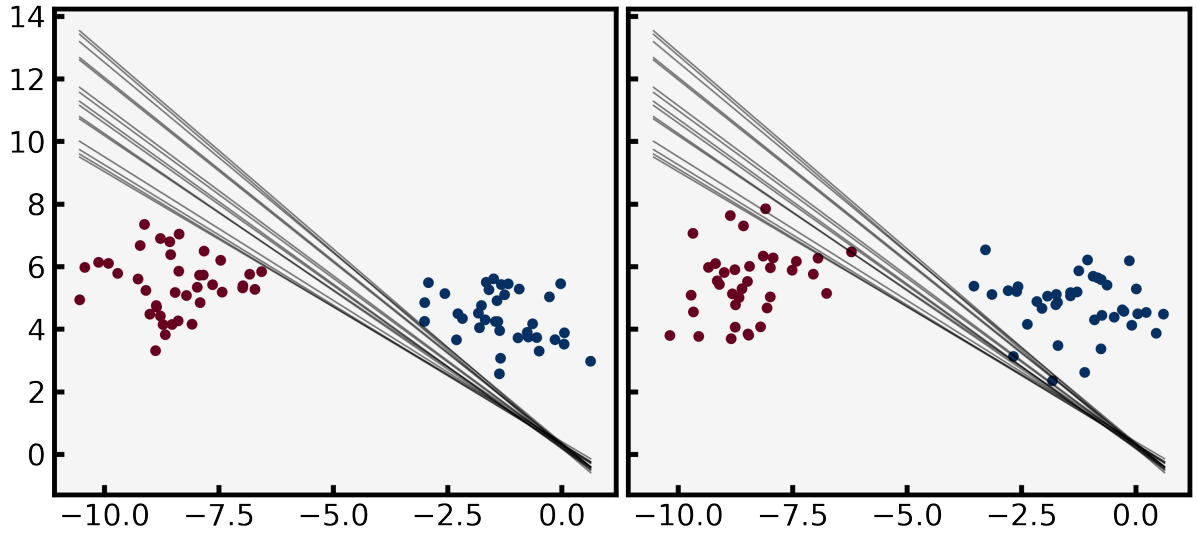

Figure 1.1: Left: All members of the set of trained classifiers that have a training error $\epsilon_n(h) = 0$. Right: Same set of classifiers that do not all have a test error $\epsilon_{n'}(h) = 0$. Both sets of points (train and test) are sampled from the same underlying distribution.

1.3 Understanding Generalization

There are two key parts to designing a machine learning model that will generalize well. The choice of the set of possible models $\mathscr{H}$ (a.k.a. *model selection*) and how $\hat{h} \in \mathscr{H}$ is selected (a.k.a. *parameter fitting*). It seems counterintuitive, but it is actually advantageous to keep the set of possible classifiers $\mathscr{H}$ small. In other words, it is helpful to *a priori* restrict the classifiers that we can choose from in response to the training set S_n, because our goal is prediction rather than simply minimizing the training error.

Suppose $\mathscr{H}$ contains only a single classifier. In other words, we are not training anything. The only thing we can do is to evaluate how well this specific (already chosen) classifier performs on

the training set. From this perspective, the training examples play the role of new test examples (which are not used to select h), so the classifier will generalize well in the sense that its training performance is very much like the test performance. However, it is probably not a good classifier as it was not tailored to the task at all.

In the other extreme where $\mathscr{H}$ contains all possible classifiers, we can easily find one or many functions h that have zero training error. However, these classifiers are likely to have large test errors, as they are specifically tailored to the learning set. For example, we could think of an absurd classifier that returns $+1$ exactly for those $\mathbf{x}$ that had a positive label in S_n and -1 for everything else. Since the test samples are distinct, this would result in a test performance of $1/2$, as this classifier effectively only returns -1 for any input.

Hence, the more heavily we rely on the training examples to select $\hat{h}$, the further apart the training and testing error may be. The goal is to explore sets of classifiers $\mathscr{H}$ that are small enough to ensure that we generalize well, yet large enough that some $\hat{h} \in \mathscr{H}$ has a low training error. We will start with a widely useful set of classifiers, the set of *linear classifiers*.

2 Linear Classification I

Any classifier h divides the input space into two halves based on the predicted label, i.e., cuts $\mathbb{R}^d$ into two sets $\{\mathbf{x} : h(\mathbf{x}) = -1\}$ and $\{\mathbf{x} : h(\mathbf{x}) = 1\}$ which, as sets, can be quite complicated. Linear classifiers make this division in a simple geometric way. In two dimensions, each linear classifier can be represented by a line that divides the space into two halves based on the predicted label. In three dimensions, the division is made by a plane and in higher dimensions by a hyperplane.

2.1 The Set of Linear Classifiers

A linear classifier is defined as

$$h(\mathbf{x}; \mathbf{w}, b) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) = \begin{cases} +1 & \mathbf{w} \cdot \mathbf{x} + b > 0 \\ -1 & \mathbf{w} \cdot \mathbf{x} + b \leq 0 \end{cases} \tag{2.1}$$

where $\mathbf{w} \cdot x = \mathbf{w}^\top x$ and $\mathbf{w} = (w_1, \ldots, w_d)^\top$ is a column vector of real-valued parameters and b is a scalar. If $b = 0$, then the plane of separation goes through the origin. Different settings of the parameters give rise to different classifiers in this set. Any two classifiers corresponding to different parameters would produce a different prediction for at least some input examples $\mathbf{x}$. We call the points $\mathbf{x}$ that fulfill $\mathbf{w} \cdot \mathbf{x} + b = 0$ the decision boundary. All parameters of a model, in the case of linear classifiers $\mathbf{w}$ and b, will be denoted with $\boldsymbol{\theta}$.

2.2 The Power of Linear Classifiers

Let us try to understand a little better what linear classifiers can do. We would like to understand whether we can find a linear classifier that perfectly separates the training examples into positive and negative. This is certainly not always the case, only when the examples are linearly separable.

More generally, when the set of classifiers we consider has at least one member (a classifier) that correctly classifies the training (and test) examples, the setting is known as the realizable case. Note that both the set of classifiers and the way the examples are chosen determine whether the problem is realizable.

Although linear classifiers may appear too simple, they are not. We will show later how it is easy to turn linear classifiers into very powerful classifiers by adding features to input examples, i.e., adding coordinates that aid in separating positive and negative examples. As a general rule, linear classifiers based on high-dimensional feature vectors are quite powerful. For now, however, we will continue to assume that feature vectors are simple enough, i.e., low-dimensional enough that linear separation is an effective constraint to guarantee good generalization.

2.3 Learning Linear Classifiers

Now that we have chosen our set of classifiers and have understood a bit what it can and cannot do, we are left with the problem of selecting a classifier from this set in response to the training

set $S_n = \{(\mathbf{x}^{(i)}, y^{(i)}), i = 1, \ldots, n\}$. There is a long history of designing algorithms and developing theory for learning linear classifiers from data. In general, algorithms are derived in one of two ways. First, we can design simple on-line algorithms (an algorithm that processes inputs piece by piece) that consider each training example in turn, updating parameters slightly in response to possible errors. Such algorithms are simple to understand (they respond by correcting mistakes) and efficient to run, since the amount of computation they require scales linearly with the size of the training set. However, they often do not have (strictly speaking) well-defined objective functions that the algorithm minimizes with respect to the parameters. They are motivated by goals such as minimizing the training error, but cannot be strictly seen to do so. They do, however, typically accompany theoretical guarantees of generalization, which is what we are after in the first place. We will discuss the perceptron algorithm in this category.

The second type of algorithm is a method that directly minimizes a clearly specified objective function with respect to the classifier parameters. The objective function may reflect the 'error', 'cost', 'loss', or 'risk' of a linear classifier. By putting a lot of effort into defining the objective, we can use an optimization algorithm whose goal is to directly minimize the objective as the learning algorithm. Methods in this category include, e.g., the Support Vector Machine (SVM).

2.3.1 Training Error as the Learning Objective

As a starting point, let us take the training error as our (approximate) learning criterion. Note that since our set of classifiers is quite constrained, finding a classifier that works well on the training set is likely to also work well on the test set, i.e., generalize well. For now, we try to find $\mathbf{w}$ and b that minimize

$$\epsilon_n(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}[y^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \leq 0] \,, \tag{2.2}$$

where $\mathbb{1}[\,]$ returns 1 if the logical expression in the argument is true, and zero otherwise. The training error here is the fraction of training examples for which the classifier with parameters $\mathbf{w}$ and b predicts the wrong label. Note that the last expression on the right calls all predictions for which $y(\mathbf{w}^\top \mathbf{x} + b) \leq 0$ errors. This may differ slightly from $\mathbb{1}[y^{(i)} \neq h(\mathbf{x}^{(i)}; \mathbf{w}, b)]$ when the points fall right on the boundary. We will use $y(\mathbf{w}^\top \mathbf{x} + b) \leq 0$ as the more accurate measure of error (if we don't know, it should be an error). Now, the training error $\epsilon_n(\mathbf{w}, b)$ can be evaluated for any choice of parameters $\mathbf{w}, b$. As a result, the error can also be minimized with respect to $\mathbf{w}, b$.

So, what would a reasonable algorithm be for finding $\hat{\mathbf{w}}, \hat{b}$ that minimizes $\epsilon_n(\mathbf{w}, b)$? Unfortunately, this is not an easy problem to solve in general, and we will have to settle for an algorithm that approximately minimizes the training error.

2.4 The Perceptron Algorithm

Some of the simplest and most useful algorithms for learning linear classifiers are mistake-driven online algorithms. They are guaranteed to find a solution with zero training error if such a solution exists (realizable case), though their behavior in the not-realizable case is less well defined. For simplicity, we will begin with the problem of learning linear classifiers through origin and bring back the offset parameter later. Now, the mistake-driven algorithms in this case start with a simple classifier, e.g., $\mathbf{w} = 0$ (zero vector), and successively try to adjust the parameters, based on each training example in turn, so as to correct any mistakes. Of course, since each step of the algorithm

does not necessarily decrease $\epsilon_n(\mathbf{w})$, we will have to separately prove that it will indeed find a solution with zero error.

The simplest (and oldest!) algorithm of this type is the perceptron algorithm. In this algorithm, we go through the training set T times, adjusting the parameters slightly in response to any mistake. No update is made if the example is classified correctly. Since the parameters are changed only when we encounter a mistake, we can also track their evolution as a function of the mistakes, i.e., denote $\mathbf{w}^{(k)}$ as the parameters obtained after exactly k mistakes on the training set. By definition $\mathbf{w}^{(0)} = 0$. Looking at the parameters in this way will be helpful in understanding what the algorithm does.

Algorithm 1 Perceptron Algorithm (no offset)

```
procedure PERCEPTRONSIMPLE({(x^(i), y^(i))}_{i=1,...,n}, T)
    w = 0 (vector)
    for t = 1, ..., T do
        for i = 1, ..., n do
            if y^(i)(w^⊤ x^(i)) ≤ 0 then
                w ← w + y^(i) x^(i)
            end if
        end for
    end for
    return w
end procedure
```

We should first establish that the perceptron updates tend to correct mistakes. Note that when we make a mistake, the product $y^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)})$ is negative or zero. Suppose that we make a mistake on $\mathbf{x}^{(i)}$. Then the updated parameters are given by $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + y^{(i)}\mathbf{x}^{(i)}$. If we consider classifying the same example $\mathbf{x}^{(i)}$ again right after the update, using the new parameters $\mathbf{w}^{(k+1)}$, then

$$\begin{aligned} y^{(i)}\left[\mathbf{w}^{(k+1)} \cdot \mathbf{x}^{(i)}\right] &= y^{(i)}\left[(\mathbf{w}^{(k)} + y^{(i)}\mathbf{x}^{(i)}) \cdot \mathbf{x}^{(i)}\right] \\ &= y^{(i)}(\mathbf{w}^{(k)} \cdot \mathbf{x}^{(i)}) + (y^{(i)})^2(\mathbf{x}^{(i)} \cdot \mathbf{x}^{(i)}) \\ &= y^{(i)}(\mathbf{w}^{(k)} \cdot \mathbf{x}^{(i)}) + ||\mathbf{x}^{(i)}||^2 \end{aligned}$$

In other words, the value of $y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)})$ increases as a result of the update (becomes more positive). If we consider the same example repeatedly, then we will necessarily change the parameters such that the example will be classified correctly, i.e., the value of $y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)})$ becomes strictly positive. Of course, mistakes on other examples may steer the parameters in different directions, so it may not be clear that the algorithm converges to something useful if we repeatedly cycle through the training examples. Luckily, we can show that the algorithm converges in the realizable case (we assume that T, the number of passes through the training set, is large enough to allow this).

The number of mistakes that the algorithm makes as it passes through the training examples depends on how easy or difficult the classification task is. If the training examples are well separated by a linear classifier, the perceptron algorithm converges quickly, i.e., it makes only a few mistakes in total until all the training examples are correctly classified. The convergence guarantee holds independently of the order in which the points are traversed. Although the order does impact the number of mistakes that the algorithm makes, it does not change the fact that it converges after a finite number of mistakes. Similarly, we could use any (finite) initial setting of $\mathbf{w}$ without losing

the convergence guarantee, but again affecting the number of errors accumulated. The perceptron algorithm and the above statements about convergence naturally extend to the case with the offset parameter.

Algorithm 2 Perceptron Algorithm

procedure PERCEPTRON($\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1,\dots,n}, T$)
  $\mathbf{w} = 0$ (vector), $b = 0$ (scalar)
  for $t = 1, \dots, T$ **do**
    for $i = 1, \dots, n$ **do**
      if $y^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \leq 0$ **then**
        $\mathbf{w} \leftarrow \mathbf{w} + y^{(i)}\mathbf{x}^{(i)}$
        $b \leftarrow b + y^{(i)}$
      end if
    end for
  end for
  return $\mathbf{w}, b$
end procedure

To understand why the offset is updated in this way, think of it as a parameter associated with an additional coordinate that is set to 1 for all examples. In other words, we map our examples $\mathbf{x} \in \mathbb{R}^d$ to $\mathbf{x}' \in \mathbb{R}^{d+1}$ such that $\mathbf{x}' = (x_1, \dots, x_n, 1)^\top$, and our parameters $\mathbf{w} \in \mathbb{R}^d$ to $\mathbf{w}' \in \mathbb{R}^{d+1}$ such that $\mathbf{w}' = (w_1, \dots, w_n, b)^\top$. In this setup, the perceptron algorithm through origin will reduce exactly to the algorithm shown above. In terms of convergence, if training examples are linearly separable (not necessarily through the origin), then the above perceptron algorithm with the offset parameter converges after a finite number of mistakes. If the training examples are not linearly separable, then the algorithm will continue to make updates during each of the T passes. It cannot converge.

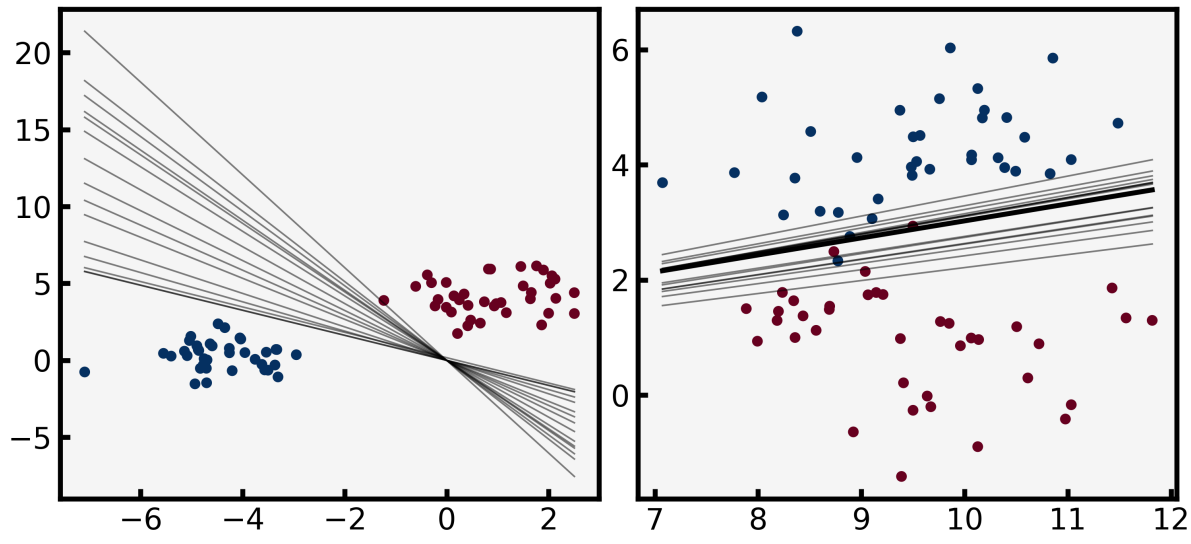

Figure 2.1: Left: Realizable case that can be separated linearly with the perceptron algorithm (Alg. 2). Right: This training set is not separable. The thin lines indicate solutions of the perceptron, whereas the thick black line is the averaged perceptron (Alg. 3) that yields more sensible solution.

2.5 Beyond the Realizable Case

When the training data are not linearly separable, the perceptron algorithm will not converge and continues to update until stopped (after T passes). We can adjust the algorithm a little to extract a reasonable classifier even in this case. In the course of the algorithm, the parameters that remain intact for a large number of examples are better than others. It seems therefore reasonable that we should track the parameters as the algorithm runs and take their average as the final answer. In other words, we average the parameters that are present at each step, not just the parameters after each update, to emphasize the parameters that seem to work longer than others. This is called the *averaged perceptron* and works quite well in practice.

Algorithm 3 Averaged Perceptron Algorithm

procedure AVERAGEDPERCEPTRON($\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1,\dots,n}, T$)
  $\mathbf{w} = 0$ (vector), $b = 0$ (scalar)
  $\bar{\mathbf{w}} = 0$ (vector), $\bar{b} = 0$ (scalar)
  $c = 1$
  for $t = 1, \dots, T$ **do**
    for $i = 1, \dots, n$ **do**
      if $y^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \leq 0$ **then**
        $\mathbf{w} \leftarrow \mathbf{w} + y^{(i)}\mathbf{x}^{(i)}$
        $b \leftarrow b + y^{(i)}$
        $\bar{\mathbf{w}} \leftarrow \bar{\mathbf{w}} + c\,y^{(i)}\mathbf{x}^{(i)}$
        $\bar{b} \leftarrow \bar{b} + c\,y^{(i)}$
      end if
      $c \leftarrow c - \frac{1}{nT}$
    end for
  end for
  return $\bar{\mathbf{w}}, \bar{b}$
end procedure

3 Understanding and Creating Features

A linear classifier is not indifferent to the way objects are represented as vectors. In fact, the "feature encoding" step in which we concatenate the properties of the object into a vector can easily make or break the classifier in terms of performance. This is true for both training performance, i.e., whether we are able to accurately classify the training examples, and test performance, i.e., whether the resulting classifier generalizes well. We will begin by trying to understand how to make linear classifiers more powerful (better able to fit training examples) by adding additional features and thinking about practically useful ways of encoding information into feature vectors in a real context.

Consider a training set $S_n = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1,\dots,n}$. We will use $\mathbf{x}$ to denote the object we wish to classify. It may already be represented as a feature vector, such as exam scores for a student, or it may need to be transformed into one, for example, if $\mathbf{x}$ is a chemical or written sentence. We now return to using $\boldsymbol{\phi}(\mathbf{x})$ to denote the resulting feature vector associated with object $\mathbf{x}$. So, while x may or may not already be a vector, $\boldsymbol{\phi}(\mathbf{x})$ always is.

The second half of this chapter will present a related but different topic. How do we make chemicals machine-readable? In the realm of computational chemistry and machine learning, transforming molecular structures into a format that algorithms can process is crucial. This transformation is (also) called featurization and involves converting the geometrical and compositional details of molecules into machine-readable formats. In other words, we transform the chemical composition into a vector or a set of vectors that a computer can process.

3.1 Polynomial Features

Let us start by looking at what makes a linear classifier powerful in terms of its ability to predict training labels. Suppose that the input x is just a scalar. A linear classifier based solely on this characteristic is not very flexible, since classification decisions are made by thresholding a linear function $wx + b$. Thus, linear classifiers in this case can select at most one point on the line of real numbers and decide which half is positive. Therefore, none of these classifiers could correctly fit the training examples $(x^{(1)} = -1; y^{(1)} = +1), (x^{(2)} = 0; y^{(2)} = -1), (x^{(3)} = 1; y^{(3)} = +1)$. In order to be able to classify these correctly and still claim to do linear classification, we must first map x to a feature vector $\boldsymbol{\phi}(x)$ and learn a linear classifier based on $\{(\boldsymbol{\phi}(x^{(i)}), y^{(i)})\}_{i=1,\dots,n}$ rather than the original training set. The resulting classifier is no longer linear in x but linear in $\boldsymbol{\phi}(x)$.

How does one select $\boldsymbol{\phi}(x)$? Its coordinates should somehow yield more power than the scalar x alone. The key notion here is linear independence. Monomials such as $1, x, x^2, x^3, \dots$ are, as functions, linearly independent. What this means is that, in general, $(x^{(i)})^k$ cannot be obtained by a linear transformation of all $x^{(i)}$. This is true as long as the number of samples n is greater than the rank of the polynomial k. Therefore, the $3 \times n$ matrix

$$\begin{bmatrix} 1 & \dots & 1 \\ x^{(1)} & \dots & x^{(n)} \\ (x^{(1)})^2 & \dots & (x^{(n)})^2 \end{bmatrix} \tag{3.1}$$

has linearly independent rows if $n \geq 3$ and the points are distinct. When $n = 3$, the columns must also be linearly independent, as the matrix is square. The linear independence of the columns then ensures that we can find b, w_1, w_2 such that

$$[b, w_1, w_2] \begin{bmatrix} 1 \\ x^{(i)} \\ (x^{(i)})^2 \end{bmatrix} = b + \mathbf{w} \cdot \boldsymbol{\phi}(x^{(i)}) \tag{3.2}$$

takes any value we want for $i = 1, 2, 3$. In other words, we can find parameters of a linear classifier $\mathbf{w} = [w_1, w_2]^\top$ and b operating on feature vectors $\boldsymbol{\phi}(x) = [x, x^2]^\top$ such that $b + \mathbf{w} \cdot \boldsymbol{\phi}(x)$ is positive or negative as desired for any three distinct points. Furthermore, if $\boldsymbol{\phi}(x)$ is expanded up to degree p rather than just 2, then the same holds for any $p + 1$ distinct points. The conclusion is that by adding linearly independent functions of x as coordinates in the feature vectors, we clearly increase the power of the linear classifiers that operate on the resulting feature vectors.

Although the focus of this discussion has been on creating more powerful classifiers, it is good to keep in mind that our goal is generalization, not fitting to the training data. More powerful features are not always useful, even if they make training examples linearly separable. Indeed, intuitively speaking, for a classifier to generalize well, the choice of this classifier must become more constrained as we get more training data. In other words, training examples must increasingly narrow down what the classifier should be to the extent that additional examples barely change the classifier. This is formally known as *stability*. Clearly, stability runs counter to making the classifier all powerful, able to fit any data.

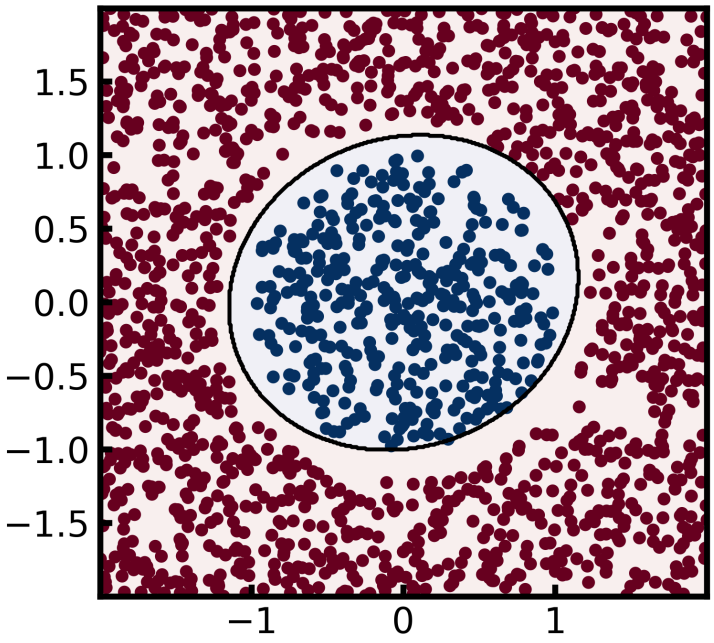

Figure 3.1: Here we have a set of points with a circular decision boundary. The points can clearly not be separated linearly with the feature vectors $\boldsymbol{\phi}(\mathbf{x}) = [x_1, x_2]$. However, the perceptron algorithm can perfectly classify the points when using $\boldsymbol{\phi}(\mathbf{x}) = [x_1, x_2, x_1^2, x_2^2, x_1 x_2]$ instead.

3.2 String Representations

In the fields of computational chemistry and cheminformatics, compounds are typically represented as strings. The most prominent examples from machine learning include SMILES, SELFIES, and SMARTS, which are popular for their simplicity and adaptability. This section offers an overview of these string-based representations, along with examples to demonstrate their usage. It is important to note that these string representations must undergo further processing to be converted into floating-point number vectors.

3.2.1 Sequence Based Representations

Sequence-based representations capture the linear order and relationships between elements in a sequence, which is used to understand the structure and behavior of molecules, especially polymers. In the context of machine learning, a sequence is an ordered list of elements that can represent various scientific entities. For chemists and materials scientists, sequences often take the form of molecular structures, crystal lattices, or polymer chains. Each element in the sequence can represent atoms, monomers, or other fundamental units. The most well-known sequences from biochemistry are the one-letter codes used for amino acids in proteins or bases in DNA and RNA. Those bio molecules are, after all, polymers.

- **Molecular Sequences:** Molecules can be represented as sequences of atoms connected by chemical bonds. A common representation is the SMILES (Simplified Molecular Input Line Entry System) notation, which encodes the structure of a molecule in a linear string of characters.
- **Polymer Sequences:** Polymers are long chains of repeating units (monomers). The sequence of monomers in a polymer chain can significantly influence its properties. Representing polymers as sequences allows for the prediction of properties and behaviors using machine learning models.
- **Crystal Structures:** Crystals can be described by sequences of unit cells arranged in a periodic lattice. The arrangement and type of atoms within each unit cell and their repetition pattern can be represented as a sequence.

Techniques for Sequence-Based Representations

Several machine learning techniques are employed to handle sequence data effectively:

- **Recurrent Neural Networks (RNNs):** RNNs are designed to handle sequential data by maintaining a hidden state that captures information from previous elements in the sequence. Variants such as long-short-term memory (LSTM) and gated recurrent unit (GRU) are particularly effective in capturing long-range dependencies in sequences.
- **Convolutional Neural Networks (CNNs):** While CNNs are traditionally used for image data, they can be adapted for sequences by using one-dimensional convolutions, which scan through the sequence to capture local patterns.
- **Transformers:** Transformer models, such as BERT and GPT, have revolutionized sequence-based learning by using self-attention mechanisms to capture global dependencies within sequences. These models have shown great promise in natural language processing and are increasingly being applied to chemical and material sequences.

3.2.2 SMILES (Simplified Molecular Input Line Entry System)

SMILES is a notation that allows a user to represent a chemical structure in a way that can be used by computer programs. It encodes molecular structures using short ASCII strings. The problem with SMILES is that they are not always valid. Another problem is that they are not unique, different software will generate SMILES for the same molecule, but the strings are not identical, even though the molecule is.

SMILES Syntax

- **Atoms** are represented by their atomic symbols (e.g., C for carbon, O for oxygen).
- **Bonds** are represented by symbols such as '-', '=', and '#' for single, double, and triple bonds, respectively. More often than not, bonds are not explicitly written down.
- **Branches** are denoted by parentheses.
- **Rings** are indicated by numbers after the atomic symbol, each number appearing twice to indicate where a ring opens and where it closes.

Examples

- Ethanol: `CCO`
- Benzene: `c1ccccc1`
- Cyclohexane: `C1CCCCC1`

3.2.3 SELFIES (Self-Referencing Embedded Strings)

SELFIES is a more robust alternative to SMILES, designed to be a 100%-robust molecular string representation. It guarantees that any string generated under its grammar is valid, which makes it particularly useful for generative models in machine learning. However, it turns out that empirically people seem to prefer SMILES over SELFIES as the nonvalidity of SMILES seems to "teach" the model something.

SELFIES Syntax

SELFIES uses a set of predefined tokens to represent atoms and bonds. The tokens ensure that any sequence generated will correspond to a valid molecule.

Examples

- Ethanol: `[C][C][O][H][H][H][H][H]`
- Benzene: `[C][=C][C][=C][C][=C]`
- Cyclohexane: `[C][C][C][C][C][C]`

3.2.4 SMARTS (SMILES Arbitrary Target Specification)

SMARTS is an extension of the SMILES notation, allowing for the specification of substructural patterns in molecules. It is useful for querying databases for molecules containing specific substructures, and it can be used to express reactions/substitutions.

SMARTS Syntax

SMARTS builds on the SMILES notation with additional symbols and operators to denote wildcards, recursive patterns, and logical operators.

Examples

- Any aliphatic carbon: `[C]`
- Any aromatic carbon: `[c]`
- Ketone functional group: `C(=O)C`
- Hydroxyl group: `[OH]`

3.2.5 InChI Keys

InChI (International Chemical Identifier) Keys are a standardized textual representation for chemical substances, developed by the International Union of Pure and Applied Chemistry (IUPAC). InChI keys provide a way to encode chemical information into string format.

Overview of InChI Keys

An InChI key is a fixed length, 27-character alphanumeric string derived from a more detailed InChI string that describes the molecular structure in a hierarchical manner. The InChI Key simplifies the sharing and searching of chemical information by providing a condensed and standardized form of the full, more complex InChI string. For example, the InChI of ethanol is `'InChI=1S/C2H6O/c1-2-3/h3H,2H2,1H3'`, whereas `LFQSCWFLJHTTHZ-UHFFFAOYSA-N` is the InChI key.

The InChI Key consists of three parts: i) the first block, which consists of 14 characters that encode the molecular skeleton, including connectivity and hydrogen atoms; ii) the second block, which is 8 characters long and encodes the remaining structural information, such as stereochemistry and isotopic variations; and iii) the final check character, which is a single letter that acts as a checksum to verify the integrity of the InChI Key.

InChI keys are generated using the IUPAC InChI algorithm. This ensures that each unique molecular structure corresponds to a unique InChI key, enabling consistent identification and comparison of chemical substances.

Advantages and Limitations of InChI Keys

Advantages:

- **Standardization:** InChI Keys provide a standardized format for representing chemical structures, ensuring consistency and interoperability across different systems and databases.

- **Compactness:** The fixed-length, 27-character InChI Key is much shorter and easier to handle than the full InChI string, making it suitable for database indexing and web searching.
- **Efficiency:** InChI Keys enable fast and efficient searching and comparison of chemical compounds, crucial for high-throughput screening and large-scale data analysis.

Limitations:

- **Loss of Detail:** The InChI Key is a condensed version of the full InChI string and does not capture all the detailed structural information. For complete molecular characterization, the full InChI string is required.
- **Non-Human-Readable:** While InChI Keys are more compact than full InChI strings, they are not easily interpretable by humans. One needs software tools to decode and interpret InChI Keys.
- **Case Sensitivity:** InChI Keys are case-sensitive, which can lead to issues with consistency and data entry errors.

3.3 Molecular Fingerprints

Molecular fingerprints are a type of molecular descriptor used extensively in cheminformatics for representing molecules in a way that facilitates comparison, similarity search, and machine learning applications. They are a compact, binary representation of molecular structures, capturing essential features that can be used to differentiate between molecules.

3.3.1 Overview

A molecular fingerprint is a binary vector in which each bit represents the presence or absence of a particular substructure or chemical feature within the molecule. This is also called one-hot encoding, meaning that the vector values are either 1 or 0, where $x_i = 1$ means that feature i is present. The length of the fingerprint varies depending on the type and design, but typically ranges from a few hundred to several thousand bits. These fingerprints are generated using various algorithms that scan the molecular structure and encode specific information into the bit string.

3.3.2 Types of Molecular Fingerprints

There are several types of molecular fingerprints, each with its own method of encoding information about the molecule.

Structural Fingerprints

Structural fingerprints, also known as substructure fingerprints, are generated by identifying predefined substructures or fragments within a molecule. Each bit in the fingerprint corresponds to a particular substructure, and the bit is set to 1 if the substructure is present in the molecule or 0 if it is absent.
Examples:

- **Daylight Fingerprints:** One of the earliest and most widely used types of fingerprints, based on the presence/absence predefined chemical substructures. They were developed by the Daylight Chemical Information System.
- **MACCS Keys:** A set of 166 predefined substructures commonly used for substructure search and similarity analysis.

Circular Fingerprints

Circular fingerprints, such as the **Extended-Connectivity Fingerprints (ECFP)**, encode information about circular neighborhoods around each atom in the molecule. They consider atom environments up to a specified radius, capturing local structural information. They are useful in similarity searches and machine learning. They are also known as **Morgan fingerprints**.

Topological Fingerprints

Topological fingerprints represent the molecular graph by encoding paths, rings, and other topological features of the molecule. They encode the connectivity of atoms and the overall molecular shape. One such example would be **Path-Based Fingerprints**, which encode information about linear paths within the molecule, up to a certain length.

Pharmacophore Fingerprints

Pharmacophore fingerprints capture the spatial arrangement of pharmacophoric features, such as hydrogen bond donors, acceptors, aromatic rings, and hydrophobic regions. These fingerprints are particularly useful in drug discovery and virtual screening. An example would be **PharmacoPFP**, which encodes pharmacophoric features and their spatial relationships within the molecule.

3.3.3 Generation of Molecular Fingerprints

The process of generating molecular fingerprints involves several steps:

1. **Molecular Structure Parsing:** The molecular structure is parsed to identify atoms, bonds, and their connectivity.
2. **Feature Extraction:** Relevant features or substructures are extracted from the molecule based on the type of fingerprint generated.
3. **Encoding:** The extracted features are encoded into a binary vector, where each bit represents the presence or absence of a specific feature.

3.3.4 Advantages and Limitations

Advantages:

- **Compact Representation:** Fingerprints provide a compact and efficient representation of molecular structures.
- **Scalability:** Suitable for handling large datasets due to their binary nature and fixed length.

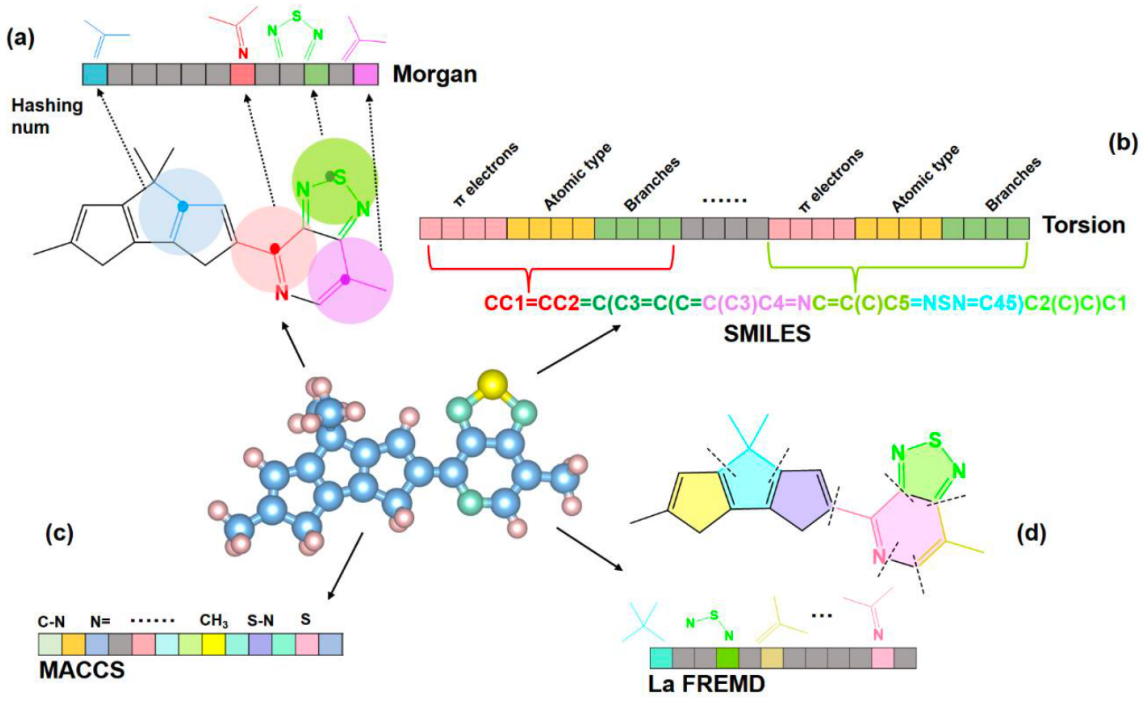

Figure 3.2: Diagrammatic representations of several molecular fingerprints. Morgan fingerprints (a) are made up of hash indexes of substructures that are located within a specific distance of every atom in the structure. (b) Torsion: Linear paths of 4 atoms at a time to form torsion fingerprints, which record the number of π electrons, atomic species, and nonhydrogen branch chains of the four atoms. (c) MACCS: One-hot encoding is used, and each bit in the MACCS code indicates a small substructure. (d) La FREMD: A molecular fragment created by clipping the input in accordance with predetermined guidelines makes up the La FREMD fingerprint produced by RDKIT. Every molecular fragment was represented using one-hot coding. Taken from [1]

- **Versatility:** Applicable to various cheminformatics tasks, including similarity searching, clustering, and machine learning.

Limitations:

- **Information Loss:** Simplification of molecular structures into binary vectors may lead to the loss of detailed chemical information.
- **Sensitivity to Algorithm Choice:** The performance and accuracy of fingerprints can vary depending on the chosen algorithm and parameters.

3.4 Molecular Descriptors

Molecular descriptors are quantitative representations of molecular structures and properties used in cheminformatics, drug discovery, and materials science. They provide another way to encode

chemical information into numerical values. Unlike the molecular fingerprints, they do not encode the presence of a certain pattern but rather the value of a certain physicochemical quantity. Molecular descriptors can be classified into several categories, including topological, geometric, electronic, and thermodynamic descriptors. Each category captures different aspects of the molecular structure and properties.

Molecular descriptors are often used to predict the quantitative structure-activity relationship (QSAR) and the quantitative structure-property relationship (QSPR).

3.4.1 Topological Descriptors

Topological descriptors are derived from the molecular graph, representing the connectivity and arrangement of atoms within a molecule without considering the three-dimensional geometry. Examples include:

- **Wiener Index:** The sum of the shortest path distances between all pairs of vertices in a molecular graph.
$$W = \sum_{i<j} d(v_i, v_j)$$
where $d(v_i, v_j)$ is the shortest path distance between vertices v_i and v_j.
- **Zagreb Indices:** The first and second Zagreb indices are defined as:
$$M_1 = \sum_i d_i^2, \quad M_2 = \sum_{(i,j)\in E} d_i \cdot d_j$$
where d_i is the degree of vertex v_i (number of all edges connected to the node, in a directed graph one distinguishes between the in degree and out degree) and E is the set of edges in the molecular graph.
- **Randic Index:** A measure of molecular branching, defined as:
$$R = \sum_{(i,j)\in E} (d_i \cdot d_j)^{-1/2}$$

3.4.2 Geometrical Descriptors

Geometrical descriptors take into account the three-dimensional arrangement of atoms in a molecule. These descriptors are derived from the spatial coordinates of the atoms. Examples include:

- **Molecular Volume:** The volume occupied by a molecule, often calculated using atomic (van der Waals) radii and the spatial coordinates of the atoms.
- **Surface Area:** The total surface area of a molecule, which can be important for understanding intermolecular interactions. One can distinguish between the normal surface and the solvent accessible surface area, where a sphere with the diameter of the solvent is rolled over the superposition of van der Waals spheres.
- **Radius of Gyration:** A measure of the compactness of a molecule, defined as:
$$R_g = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\mathbf{r}_i - \mathbf{r}_{\text{cm}})^2}$$

where $\mathbf{r}_i$ are the atomic coordinates and $\mathbf{r}_{\text{cm}}$ is the center of mass of the molecule.

- **Shape Indexes:** These descriptors, such as the Asphericity and the Oblateness, quantify the shape of a molecule, providing insight into its three-dimensional geometry.

$$\text{Asphericity} = \frac{(\lambda_1 - \lambda_2)^2 + (\lambda_2 - \lambda_3)^2 + (\lambda_3 - \lambda_1)^2}{2(\lambda_1 + \lambda_2 + \lambda_3)^2} \tag{3.3}$$

where $\lambda_1, \lambda_2, \lambda_3$ are the eigenvalues of the molecule's inertia tensor.

3.4.3 Electronic Descriptors

Electronic descriptors capture information about the electronic structure of a molecule, which is crucial for understanding reactivity and interactions. Examples are:

- **HOMO and LUMO Energies:** The energies of the highest occupied molecular orbital (HOMO) and the lowest unoccupied molecular orbital (LUMO), which are important for predicting chemical reactivity.
- **Dipole Moment:** A measure of the separation of positive and negative charges in a molecule, which affects interactions with external electric fields.
- **Electronegativity:** An estimate of the molecule's ability to attract electrons, often calculated as the weighted average of the atomic electronegativities.

3.4.4 Thermodynamic Descriptors

Thermodynamic descriptors are related to the energetic properties of molecules and can provide insights into stability and reactivity. Examples are:

- **Heat of Formation:** The change in enthalpy when a molecule is formed from its constituent atoms in their standard states.
- **Entropy:** A measure of the disorder or randomness in a system, which can be related to the number of accessible microstates.
- **Gibbs Free Energy:** A thermodynamic potential that combines enthalpy and entropy, used to predict the spontaneity of a reaction.

$$\Delta G = \Delta H - T\Delta S$$

3.5 Graph Representations

Graph representations are essential for capturing the complex relationships and interactions within structured data. In cheminformatics, bioinformatics, and social network analysis, graph representations provide a powerful and flexible way to model data as sets of nodes (vertices) connected by edges (links). Each node typically represents an entity, and each edge represents a relationship or interaction between entities.

A Lewis structure of a molecule is nothing but a graph, where each vertex is an atom, and each bond is an edge. We will talk more about graphs at the end of this course.

3.5.1 Basic Concepts

A graph G is defined as an ordered pair $G = (V, E)$, where:

- V is a set of vertices (nodes), $V = \{v_1, v_2, \ldots, v_n\}$.
- E is a set of edges (links), $E \subseteq \{e_{ij} = (v_i, v_j) \mid v_i, v_j \in V\}$.

Graphs can be undirected or directed:

- **Undirected Graph:** Edges have no direction, i.e., the edge e_{ij} is identical to e_{ji}.
- **Directed Graph (Digraph):** Edges have a direction, i.e., $e_{ij} \neq e_{ji}$.

There are neural force fields with directed and undirected graphs.

3.5.2 Types of Graph Representations

Graphs can be represented in various forms to facilitate their processing and analysis.

Adjacency Matrix

An adjacency matrix A is a square matrix used to represent a graph, where A_{ij} indicates the presence or absence of an edge between vertices v_i and v_j. For an undirected graph:

$$A_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

For a directed graph, the matrix is not necessarily symmetric.

Adjacency List

An adjacency list is an array of lists. The list at index i contains all vertices j such that there is an edge (v_i, v_j) in the graph. This representation is more space-efficient for sparse graphs.

They are often used for nearest-neighbor lists, as each atom has its own environment, the amount of neighbors will therefore greatly vary, e.g., a bound atom has several close neighbors due to the chemical bonds surrounding it, but an ion in solution has fewer neighbors as it does have a single covalent bond.

Edge List

An edge list is a collection of all the edges in the graph. Each edge is represented as a pair (v_i, v_j), indicating a connection between vertices v_i and v_j. This representation is compact, but less efficient for querying connectivity.

3.6 3D Geometrical Representations of Molecules

Three-dimensional (3D) geometrical representations of molecules are essential for understanding and predicting molecular properties, behaviors, and interactions. These representations capture the spatial arrangement of atoms in a molecule, which is crucial for studying stereochemistry, reactivity, and molecular dynamics. Several methods and descriptors are used to encode the 3D geometry of molecules, each providing different levels of detail and utility for various applications.

3D geometrical representations are often linked to graph representations, since the graph encodes the chemical information of formed bonds.

3.6.1 Cartesian Coordinates

The most straightforward 3D representation of a molecule is through Cartesian coordinates, which specify the positions of atoms in a 3D space using (x, y, z) coordinates. Each atom in the molecule is assigned a coordinate triplet, which provides a complete picture of the molecular structure.

$$\mathbf{r}_i = (x_i, y_i, z_i) \tag{3.4}$$

where $\mathbf{r}_i$ represents the coordinates of the i-th atom.

3.6.2 Bond Lengths, Angles, and Dihedrals

Another common representation involves describing the molecule in terms of internal coordinates: bond lengths, bond angles, and dihedral angles. This approach is particularly useful for understanding the conformational flexibility of molecules. Internal coordinates are especially common in geometry optimization and have been used as a Z-matrix in quantum chemistry packages.

- **Bond Length:** The distance between two bonded atoms, denoted as r_{ij}.

$$r_{ij} = \|\mathbf{r}_i - \mathbf{r}_j\|_2 \tag{3.5}$$

- **Bond Angle:** The angle between three consecutive atoms, denoted as θ_{ijk}.

$$\theta_{ijk} = \cos^{-1}\left(\frac{(\mathbf{r}_i - \mathbf{r}_j)\cdot(\mathbf{r}_k - \mathbf{r}_j)}{\|\mathbf{r}_i - \mathbf{r}_j\|_2 \|\mathbf{r}_k - \mathbf{r}_j\|_2}\right) \tag{3.6}$$

- **Dihedral Angle:** The angle between two planes formed by four consecutive atoms, denoted as ϕ_{ijkl}.

$$\phi_{ijkl} = \cos^{-1}\left(\frac{(\mathbf{r}_j - \mathbf{r}_i)\times(\mathbf{r}_k - \mathbf{r}_j)\cdot(\mathbf{r}_l - \mathbf{r}_k)}{\|(\mathbf{r}_j - \mathbf{r}_i)\times(\mathbf{r}_k - \mathbf{r}_j)\|_2 \|(\mathbf{r}_k - \mathbf{r}_j)\times(\mathbf{r}_l - \mathbf{r}_k)\|_2}\right) \tag{3.7}$$

4 Linear Classification II: Objective Functions

One way to frame machine learning problems is to write them down as optimization problems. The solution to the optimization problem is then the model we seek. Thinking about problems in this way is often beneficial since the optimization problem expresses the key trade-offs we want to balance. For example, we might wish to find a classifier that separates most if not all of the training points by a large margin. The key trade-off in this case is between how well we can classify each point (expressed through a loss function) and the value of the margin we attain (what we call a regularizer). The larger the margin that we require the classifier to attain, the fewer points will actually satisfy it. The trade-off between the loss measured on training points and the margin we can achieve is therefore real and needs to be balanced.

In general, if we consider classifiers specified by parameters $\mathbf{w}, b$, e.g., the set of linear classifiers, we first try to turn the learning problem into an optimization problem over these parameters. We want a classifier that minimizes some objective function that may be composed of two or more parts that characterize the key trade-offs. We typically try to minimize

$$J(\mathbf{w}, b) = \underbrace{\frac{1}{n}\sum_{i=1}^{n} \text{Loss}(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)}_{\text{average loss on training set}} + \underbrace{\lambda R(\mathbf{w}, b)}_{\text{regularization}} \tag{4.1}$$

with respect to $\mathbf{w}, b$ with the idea that it is the minimizing parameter values we are really after rather than the value of the objective. The objective function here has two key parts: 1) how the classifier performs on the training examples or average loss, and 2) the default answer that we wish to bias the classifier towards in the absence of any evidence from the training examples, expressed via the regularizer. $\lambda > 0$ here is known as the regularization parameter (or regularization strength), and it is used to quantify how we wish to balance these two competing goals. A large value of λ will emphasize the regularizer and thus steer the classifier towards the default answer. A small value of λ, on the other hand, discounts the regularizer in favor of the mean losses, resulting in $\mathbf{w}, b$ that attain small training losses. The key motivation for introducing the concept of a loss function is that not all errors should be treated the same. Points near the decision boundary are inherently uncertain and should be penalized less than those that are placed far on the wrong side of the boundary. Formally, we need a loss function that simply specifies numerically how badly we classify each example and balance such loss values against the regularizer.

4.1 Perceptron and Optimization

The perceptron algorithm does not fit well within the optimization framework and therefore does not quite have a loss function. However, we can think of the perceptron algorithm minimizing so-called zero-one loss (just the error), i.e.,

$$\text{Loss}_{0,1}(y(\mathbf{w}\cdot\mathbf{x} + b)) = \mathbb{1}[y(\mathbf{w}\cdot\mathbf{x} + b) \leq 0] \tag{4.2}$$

Note that while this loss function is a function of the "agreement" (whether prediction and label have the same sign) it only cares about whether the agreement is positive (correct classification) or not (mistake). It does not depend on the magnitude of the agreement (or lack thereof). The Perceptron algorithm also does not have any regularizer or default answer. It starts with all parameters set to zero, but the perceptron algorithm can get very far from this initial setting as it continues to iterate through the training examples.

4.2 Large Margin Classifier

Our goal is to derive an objective function for learning linear classifiers. Furthermore, we want the classifier to separate the points with a large margin. We can evaluate the margin (the signed distance from the decision boundary) for each training example relative to a linear classifier by

$$\gamma_i(\mathbf{w}, b) = \frac{y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)}{||\mathbf{w}||} \tag{4.3}$$

The margin is positive when the training example lies on the correct side of the decision boundary, and negative otherwise. The absolute value of the margin is always the distance of the training point from the decision boundary. We want a classifier that achieves a large margin for most or all training examples.

To translate the large-margin goal into an objective function, we will introduce an additional parameter $\gamma_{\text{ref}} > 0$ into the optimization problem. γ_{ref} specifies the margin that we would like to achieve for all examples. We may not know *a priori* what the reasonable value of γ_{ref} would be, but we want it to be large. The reason for introducing this reference margin is that we have a good idea of how it should be regularized. We wish to maximize γ_{ref} or, equivalently, minimize $1/\gamma_{\text{ref}}$, or $1/\gamma_{\text{ref}}^2$. We also need to specify the loss function, i.e., measure how well the reference margin γ_{ref} agrees with the actual margin attained, that is, $\gamma_i(\mathbf{w}, b)$. For example, we can define this loss in terms of the ratio $\gamma/\gamma_{\text{ref}}$

$$\text{Loss}_h(\gamma/\gamma_{\text{ref}}) = \begin{cases} 1-\gamma/\gamma_{\text{ref}} & \text{if} \gamma < \gamma_{\text{ref}} \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

In other words, if the ratio is higher than one, the margin attained is larger than the reference, and we incur no loss at all. If the margin for a particular example drops below the reference, we incur a loss of $1-\gamma/\gamma_{\text{ref}}$. This loss function is called the Hinge loss.

Now, our optimization problem is over $\mathbf{w}, b$ as well as $\gamma_{\text{ref}} > 0$. Specifically, we wish to find values for these parameters so as to minimize

$$J(\mathbf{w}, b, \gamma_{\text{ref}}) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}_h(\gamma_i/\gamma_{\text{ref}}) + \lambda(1/\gamma_{\text{ref}}^2) \tag{4.5}$$

Note that $\mathbf{w}, b$ only appear in the optimization problem through the margins γ_i. If we fix $\mathbf{w}, b$, there will be an optimal value for γ_{ref}. If we fix γ_{ref}, we can find the optimal $\mathbf{w}, b$ for this particular reference margin. In other words, γ_{ref} specifies how much space we want to have separating the training examples and $\mathbf{w}, b$ are optimized to best accommodate this wish. For large values of γ_{ref}, not all examples can be separated with that large margin and we start incurring some losses. The solution is then obtained by finding a balance between the desire for a large margin (the regularizer) and the losses incurred (average Hinge loss).

It turns out that we can simplify this optimization problem further by harnessing an unused degree of freedom in the linear classifier. Specifically, we can fix $||\mathbf{w}||$ to any positive value and still be able to find any linear decision boundary. For example, let us say we want $||\mathbf{w}|| = 1$. Then, the decision boundary of any linear classifier $\mathbf{w}, b$ coincides with that of $\mathbf{w}' = \mathbf{w}/||\mathbf{w}||, b' = b/||\mathbf{w}||$. Our definition of the margin $\gamma_i(\mathbf{w}, b)$ as the distance from the boundary is similarly unconstrained by the choice of $||\mathbf{w}||$. So, we can express γ_{ref} as a function of $||\mathbf{w}||$ so that the optimization problem will only be over $\mathbf{w}, b$. Specifically, we set $\gamma_{\text{ref}} = 1/||\mathbf{w}||$, so that

$$\gamma_i(\mathbf{w}, b)/\gamma_{\text{ref}} = \gamma_i(\mathbf{w}, b)\ ||\mathbf{w}|| = y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \tag{4.6}$$

Our resulting objective function for large margin classification is then given by

$$J(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}_h(y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)) + \frac{\lambda}{2}||\mathbf{w}||^2 \tag{4.7}$$

The goal is to find $\mathbf{w}, b$ that minimize $J(\mathbf{w}, b)$. Now, the choice of $||\mathbf{w}||$ matters a lot, as it is tied to the margin we are after, i.e., the preferred distance to the boundary. For later beautification of derivatives, we will use $\lambda/2$ as the regularization parameter instead of just λ.

We have introduced a more sensitive loss function for linear classification than mere error by penalizing any prediction for which the agreement drops below one (as opposed to zero) and increasing the penalty for greater violations. Specifically, we use the Hinge loss, now taking the agreement as its argument

$$\text{Loss}_h(y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)) = \max\{0,\ 1 - y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)\}$$

By introducing this loss, we do not exclude any linear classifier from being selected but will strongly bias the selection towards particular kinds of classifiers, thereby effectively restricting our choices (in a good way) and improving generalization.

4.3 Gradient Descent

Gradient descent is an iterative optimization algorithm that is used to minimize a function by adjusting its parameters. It is widely used in machine learning for training models. The main idea behind gradient descent is to move towards the minimum of a function by taking steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point.

4.3.1 Mathematical Formulation

Given a function $f(\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is the vector of all parameters, the goal of gradient descent is to find the parameters that minimize f. The gradient of f at $\boldsymbol{\theta}$, denoted as $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$, points in the direction of the steepest increase of the function. Therefore, the negative gradient points in the direction of the steepest decrease.

The gradient descent update rule is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_\theta f(\boldsymbol{\theta}_t) \tag{4.8}$$

where:

- $\boldsymbol{\theta}_t$ is the parameter vector at iteration t,
- η is the learning rate, a positive scalar determining the step size,
- $\nabla_\theta f(\boldsymbol{\theta}_t)$ is the gradient of the function at $\boldsymbol{\theta}_t$.

4.3.2 Learning Rate

The learning rate η is a crucial hyperparameter in gradient descent, as it scales the size of the parameter update. If η is too small, the algorithm will converge slowly. If η is too large, the algorithm might overshoot the minimum and fail to converge or even diverge. Selecting an appropriate learning rate is often done through experimentation or adaptive methods like learning rate schedules.

4.3.3 Types of Gradient Descent

There are several variants of gradient descent, each suited to different scenarios:

- **Batch Gradient Descent**: Uses the entire data set to compute the gradient. Although it can converge to the global minimum for convex functions, it is computationally expensive for large datasets.
- **Stochastic Gradient Descent (SGD)**: Uses one training example to compute the gradient for each optimization step. It is much faster, but introduces more noise into the optimization process, which can help escape local minima, but may also result in more fluctuations in the convergence path.
- **Mini-Batch Gradient Descent**: Uses a subset of the training data (mini-batch) to compute the gradient. Thus, it balances computational efficiency and stability of the convergence process. One usually refers to Mini-Batch GD when using the term SGD, as the training set is shuffled anew after every epoch (full iteration over the entire training set) before renewed batch selection.

4.3.4 Convergence

Gradient descent converges when the changes in the parameter vector $\boldsymbol{\theta}$ become sufficiently small or when the gradient $\nabla_\theta f(\boldsymbol{\theta})$ is close to zero. Monitoring these criteria helps determine when to stop the iterations.

Gradient descent is a foundational technique in machine learning, enabling the efficient training of a wide variety of models by iteratively refining their parameters to minimize a given objective function.

4.4 Support Vector Machine

We are ready to use the optimization problem to find the maximum-margin classifier. The resulting method is called the Support Vector Machine (SVM). The objective function of the support vector machine tries to 1) minimize the average Hinge loss on the training examples while 2) pushing the margin boundaries apart by reducing $||\mathbf{w}||$. These two goals are in opposition to each other, as discussed above. The more we emphasize the Hinge losses, the more the resulting classifier is determined by the few examples near the decision boundary. In contrast, if we extend the margin boundaries, we incur losses on examples that are further away, and the solution will be defined by where the bulk of the positive and negative examples are. By including the regularization term we have made the problem well posed even when data are conflicting or absent.

4.4.1 Gradient Descent

The parameter update occurs according to

$$\begin{aligned}
\mathbf{w} &\leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} J(\mathbf{w}, b) \\
&= \mathbf{w} - \eta \nabla_w \left[\frac{\lambda}{2} ||\mathbf{w}||^2 + \frac{1}{n} \sum_{i=1}^{n} \text{Loss}_h(y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)) \right] \\
&= \mathbf{w} - \eta \left[\nabla_w \frac{\lambda}{2} ||\mathbf{w}||^2 + \frac{1}{n} \sum_{i=1}^{n} \nabla_w \text{Loss}_h(y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)) \right] \\
&= \mathbf{w} - \eta\lambda\mathbf{w} + \eta \frac{1}{n} \sum_{i=1}^{n} \begin{cases} y^{(i)}\mathbf{x}^{(i)} & \text{if } y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \leq 1 \\ 0 & \text{o.w.} \end{cases}
\end{aligned} \tag{4.9}$$

$$\begin{aligned}
b &\leftarrow b - \eta \nabla_b J(\mathbf{w}, b) \\
&= b + \eta \frac{1}{n} \sum_{i=1}^{n} \begin{cases} y^{(i)} & \text{if } y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \leq 1 \\ 0 & \text{o.w.} \end{cases}
\end{aligned} \tag{4.10}$$

While it would be good to find such a solution, this update does not scale well with the number of training examples. In fact, it takes about $O(dn^3)$ computation to solve it reasonably well in practice, limiting it to smaller problems (on the order of 10,000 training examples without additional tricks).

4.4.2 Stochastic Gradient Descent (Pegasos algorithm)

Algorithm 4 Pegasos algorithm (online SVM)

procedure PEGASOS($\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1,\ldots,n}, \lambda, T$)
  $\mathbf{w} = 0$ (vector), $b = 0$ (scalar)
  for $t = 1, \ldots, T$ **do**
    Select $i \in \{1, \ldots, n\}$ at random
    $\eta = 1/t$
    if $y^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \leq 1$ **then**
      $\mathbf{w} \leftarrow (1 - \eta\lambda)\mathbf{w} + \eta y^{(i)} \mathbf{x}^{(i)}$
      $b \leftarrow b + \eta y^{(i)}$
    else
      $\mathbf{w} \leftarrow (1 - \eta\lambda)\mathbf{w}$
    end if
  end for
  return $\mathbf{w}, b$
end procedure

We can always try to solve the same SVM minimization problem approximately using much simpler (and scalable) stochastic gradient methods. These methods, as online algorithms, consider each training example in turn, move the parameters slightly in the negative gradient direction, and move on. This is similar but not exactly the same as the passive-aggressive algorithm (perceptron). The difference is that the regularization term now explicitly controls the norm $||\mathbf{w}||$ (i.e., 1/margin) rather than indirectly by keeping the sequence of updates small.

We can reformulate the loss and then, Instead of using all data at once, use a single training example to update the parameters

$$J(\mathbf{w}, b) = \frac{1}{n}\left[\sum_{i=1}^{n} \text{Loss}_h(y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)) + \frac{\lambda}{2}||\mathbf{w}||^2\right]$$

where η is the learning rate. If we decrease the learning rate appropriately, these gradient descent updates are guaranteed to converge to the same SVM solution. This is necessary, so that we can move to the optimum, and that the "noise" inherent in the gradient updates (stochastic choice of terms, finite step size) vanishes in the end. For example, $\eta_k = 1/(k+1)$ is a valid (albeit somewhat slow) choice, where k is the epoch or step number.

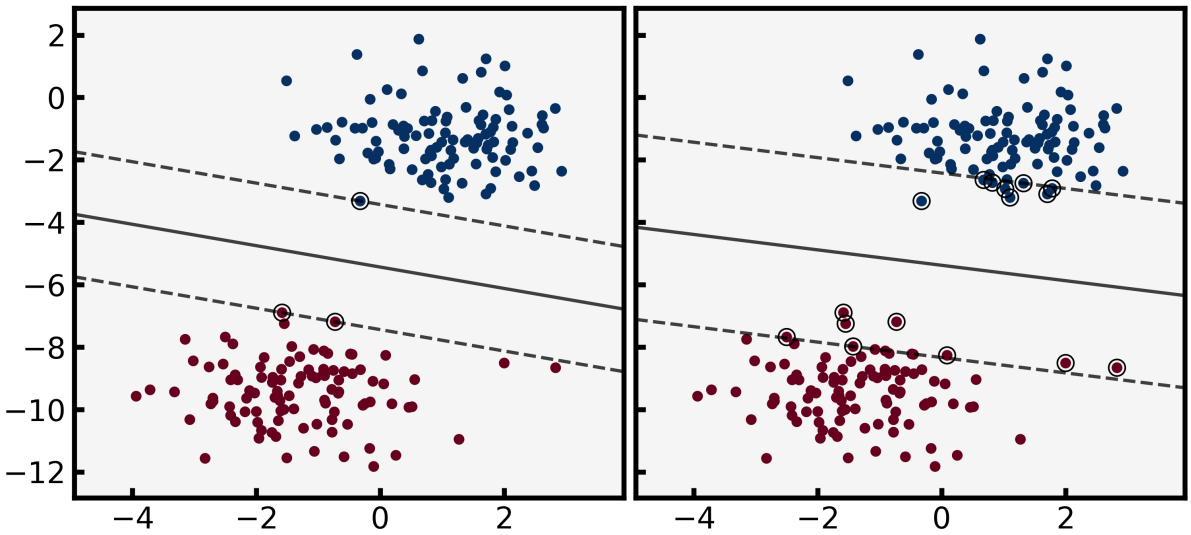

Figure 4.1: Support Vector Machine classification, with the decision boundary as a slid line and the margin dashed. The circled points are the support vectors. Left: Very weak regularization, very few points within the margin. Right: Strong regularization, more points within the margin, and flatter slope.

5 Linear Regression

Until now, we have attempted to forecast a binary label $y \in \{-1, +1\}$ for each given input $\mathbf{x}$. However, often the target we aim to predict is a real-valued number. For instance, instead of merely determining if a chemical's solubility is above or below a certain limit, we might want to predict the precise value of its solubility product.

Supervised learning tasks where the output is a continuous value are known as regression problems. Formally, our objective is to discover a function $f : \mathbb{R}^d \to \mathbb{R}$ such that $f(\mathbf{x}) \approx y$ for all test examples. As before, we select f based on a finite set of training data before observing any test cases. To generalize effectively to unseen test examples and avoid overfitting, we must constrain the family of potential functions, similar to classification. We will start with simple linear functions, similar to how we approach linear classification.

A linear regression function is simply a linear function of the feature vectors, i.e.,

$$f(\mathbf{x}; \mathbf{w}, b) = \mathbf{w} \cdot \mathbf{x} + b = \sum_{i=1}^{d} w_i x_i + b \tag{5.1}$$

Each setting of the parameters $\mathbf{w}, b$ gives rise to a slightly different regression function. Together, they constitute the set of functions $\mathscr{F}$ that we consider. Although this set of functions seems quite simple (just linear), the power of $\mathscr{F}$ is again hidden in the feature vectors. In fact, we can often construct different feature representations for objects. For example, there are many different ways in which we can map molecular descriptors into a feature vector.

To start with, we assume that a suitable representation has been found, denoting feature vectors simply as $\mathbf{x}$ for convenience. Our learning task is to choose one $f \in \mathscr{F}$, i.e., choose parameters $\hat{\mathbf{w}}$ and $\hat{b}$, based on the training set $S_n = \{\mathbf{x}^{(i)}, y^{(i)}), i = 1, \ldots, n\}$, where $y^{(i)} \in \mathbb{R}$. As before, our goal is to find $f(\mathbf{x}; \hat{\mathbf{w}}, \hat{b})$ that would yield accurate predictions on yet unseen examples. There are several problems to address:

1. How do we measure error? What is the criterion by which we choose $\hat{\mathbf{w}}$ and $\hat{b}$ based on the training set?

2. What algorithm can we use to optimize the training criterion? How does the algorithm scale with the dimension (feature vectors may be high dimensional) or the size of the training set (the dataset may be large)?

3. When the size of the training set is not large enough in relation to the number of parameters (dimension), there may be degrees of freedom, i.e., directions in the parameter space, that remain unconstrained by the data. How do we set those degrees of freedom? This is part of a broader problem known as regularization. The question is how to softly constrain the set of functions $\mathscr{F}$ to achieve better generalization.

5.1 Empirical risk and the least squares criterion

As in the classification setting, we will measure training error in terms of the average loss or *empirical risk*

$$R_n(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}(y^{(i)} - f(\mathbf{x}^{(i)}; \mathbf{w}, b)) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}(y^{(i)} - (\mathbf{w}\cdot\mathbf{x}^{(i)} + b)) \tag{5.2}$$

Note that, unlike in classification, this loss function now depends on the difference between the real valued target $y^{(i)}$ and the corresponding linear prediction $f(\mathbf{x}^{(i)}; \mathbf{w}, b))$. There are many possible ways to define the loss function. We will use here a simple squared error:

$$\text{Loss}(z) = z^2/2\ , \tag{5.3}$$

where the division by 2 is only for convenience. The idea is to permit small discrepancies (we expect the responses to include noise), but heavily penalize large deviations that typically indicate poor parameter choices. As a result, we have

$$R_n(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(y^{(i)} - (\mathbf{w}\cdot\mathbf{x}^{(i)} + b)\right)^2 \tag{5.4}$$

As discussed in Sec. 7.4.1, this expression is equal to half the mean squared error. Recall that our learning goal in supervised learning is not to minimize $R_n(\boldsymbol{\theta})$; it is just the best we can do (for now). Minimizing $R_n(\boldsymbol{\theta})$ is a surrogate or proxi criterion since we don't have a direct access to the test or generalization error

$$R^{\text{test}}_{n'}(\boldsymbol{\theta}) = \frac{1}{n'}\sum_{j=1}^{n'} \frac{1}{2}\left(y^{(j)} - (\mathbf{w}\cdot\mathbf{x}^{(j)} + b)\right)^2 \tag{5.5}$$

Let's briefly consider how $R_n(\boldsymbol{\theta})$ and $R^{\text{test}}_{n'}(\boldsymbol{\theta})$ are related. In the simplest setting, we select all parameters $(\boldsymbol{\theta})$ minimizing $R_n(\boldsymbol{\theta})$, but our performance will eventually be measured according to the test error $R^{\text{test}}_{n'}(\boldsymbol{\theta})$. This test error can be large for two different reasons. (i) First, we may have a large estimation error. This means that even if the true relationship between $\mathbf{x}$ and y was linear, it would be difficult for us to estimate or recover it based on a small (and potentially noisy) training set S_n. Our estimated parameters $\hat{\boldsymbol{\theta}}$ will not be entirely correct. The larger the training set, the smaller the estimation error. (ii) The second type of error that also affects our test error is structural error. This means that we may estimate a linear mapping from x to y when the true underlying relationship is highly non-linear. Clearly, we could not do very well in this case, regardless of how large the training set was. In order to reduce structural error, we would have to use a larger set of functions $\mathscr{F}$, e.g., by including polynomial features as additional coordinates. A larger set of functions ensures that there would at least be one $f \in \mathscr{F}$ that approximates the true underlying mapping well. But the problem is that, given only a noisy training set S_n, it will be harder to select this correct function from a large set of candidates. Our estimation error would necessarily increase. Finding the balance between estimation and structural errors is at the heart of effective learning.

When we formulate a linear regression problem as a statistical problem, we imagine that the targets have been generated by some underlying function with noise added. In this case, we can talk about the structural error as *bias*, i.e., how far off from the true function we are in expectation over different choices of training sets. The estimation error, on the other hand, corresponds to the *variance* of the function or the parameter estimator $\hat{\boldsymbol{\theta}}(S_n)$. The parameters $\hat{\boldsymbol{\theta}}$ are obtained with the help of training data and can thus be viewed as functions of S_n. An estimator is a mapping from the

data to the parameters. If we consider a large set of functions, our bias is low, but the estimated parameters will vary quite a bit from one training set to another (high variance), since we can easily find a function that fits the noise as well.

5.2 Optimizing the least squares criterion

Perhaps the simplest way to optimize the least squares objective (the sum of squared errors or residuals)

$$J_n(\boldsymbol{\theta}) = R_n(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}\left(y^{(i)} - (\mathbf{w}\cdot\mathbf{x}^{(i)} + b)\right)^2 \tag{5.6}$$

is to use the stochastic gradient descent method discussed earlier in the classification context. Our case here is easier, since $J_n(\boldsymbol{\theta})$ is everywhere differentiable. At each step of the algorithm, we select one training example at random, and nudge parameters in the opposite direction of the gradient

$$\begin{aligned}\nabla_{\mathbf{w}}\frac{(y^{(t)} - (\mathbf{w}\cdot\mathbf{x}^{(t)} + b))^2}{2} &= (y^{(t)} - (\mathbf{w}\cdot\mathbf{x}^{(t)} + b))\nabla_{\mathbf{w}}(y^{(t)} - (\mathbf{w}\cdot\mathbf{x}^{(t)} + b)) \\ &= -(y^{(t)} - (\mathbf{w}\cdot\mathbf{x}^{(t)} + b))\mathbf{x}^{(t)} \qquad (5.7)\\ \nabla_{b}\frac{(y^{(t)} - (\mathbf{w}\cdot\mathbf{x}^{(t)} + b))^2}{2} &= (y^{(t)} - (\mathbf{w}\cdot\mathbf{x}^{(t)} + b))\nabla_{b}(y^{(t)} - (\mathbf{w}\cdot\mathbf{x}^{(t)} + b)) \\ &= -(y^{(t)} - (\mathbf{w}\cdot\mathbf{x}^{(t)} + b)) \qquad (5.8)\end{aligned}$$

As a result, the algorithm can be written as where η_t is the learning rate (e.g., $\eta_t = 1/(t+1)$).

Algorithm 5 Stochastic Gradient Descent

procedure LINEARREGSGD($\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1,\ldots,n}, T$)
  $\mathbf{w} = 0$ (vector), $b = 0$ (scalar)
  for $t = 1, \ldots, T$ **do**
    Select $i \in \{1, \ldots, n\}$ at random
    $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} + \eta_t(y^{(i)} - (\mathbf{w}^{(t)}\cdot\mathbf{x}^{(i)} + b^{(t)}))\mathbf{x}^{(i)}$
    $b^{(t+1)} \leftarrow b^{(t)} + \eta_t(y^{(i)} - (\mathbf{w}^{(t)}\cdot\mathbf{x}^{(i)} + b^{(t)}))$
  end for
  return $\mathbf{w}, b$
end procedure

Recall that, in the perceptron algorithm, an update was performed only if a mistake was made. Now the update is proportional to the discrepancy $(y^{(i)} - (\mathbf{w}^{(t)}\cdot\mathbf{x}^{(i)} + b^{(t)}))$, so any error, however small, counts. As in the classification context, the update is "self-correcting", i.e., if we were to consider the same sample immediately again, the deviation between the label and the prediction would be smaller.

5.2.1 Closed Form Solution

We can also try to minimize $J_n(\boldsymbol{\theta})$ directly by setting the gradient to zero. Indeed, since $J_n(\boldsymbol{\theta})$ is a convex function of the parameters, the minimum value is obtained at a point (or a set of points)

where the gradient is zero. So, formally, we find $\hat{\boldsymbol{\theta}}$ for which $\nabla_\theta J_n(\boldsymbol{\theta}) = 0$. More specifically

$$\begin{aligned}
\nabla_\theta J_n(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}} &= \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta \left.\frac{\left(y^{(i)} - (\boldsymbol{\theta}\cdot\boldsymbol{\phi}(\mathbf{x}^{(i)}))\right)^2}{2}\right|_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}} \\
&= \frac{1}{n}\sum_{i=1}^{n} -\left(y^{(i)} - (\hat{\boldsymbol{\theta}}\cdot\boldsymbol{\phi}(\mathbf{x}^{(i)}))\right)\boldsymbol{\phi}(\mathbf{x}^{(i)}) \\
&= -\frac{1}{n}\underbrace{\sum_{i=1}^{n} y^{(i)}\boldsymbol{\phi}(\mathbf{x}^{(i)})}_{\mathbf{c}} + \frac{1}{n}\underbrace{\sum_{i=1}^{n}(\boldsymbol{\phi}(\mathbf{x}^{(i)})\boldsymbol{\phi}(\mathbf{x}^{(i)})^\top)}_{\mathbf{A}}\hat{\boldsymbol{\theta}} \\
\nabla_\theta J_n(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}} &= \mathbf{A}\hat{\boldsymbol{\theta}} - \mathbf{c} \stackrel{!}{=} 0
\end{aligned} \tag{5.9}$$

where we have used the the trick to choose a additional featurization $\boldsymbol{\phi}(\mathbf{x}^{(i)}) = [x_1, \ldots, x_d, 1]^\top$ that has dimension $d+1$ where the last dimension is simply a 1, so that addition of the bias b can be written as a scalar product, $\boldsymbol{\theta} = [w_1, \ldots, w_d, b]^\top$. Furthermore, we used the fact that $\hat{\boldsymbol{\theta}}\cdot\boldsymbol{\phi}(\mathbf{x}^{(i)})$ is a scalar and subsequently rewritten the inner product as $\hat{\boldsymbol{\theta}}\cdot\boldsymbol{\phi}(\mathbf{x}^{(i)}) = \boldsymbol{\phi}(\mathbf{x}^{(i)})^\top\hat{\boldsymbol{\theta}}$. As a result, the equation for the parameters can be expressed in terms of a $(d+1)\times 1$ column vector $\mathbf{c}$ and a $(d+1)\times(d+1)$ matrix $\mathbf{A}$ as $\mathbf{A}\hat{\boldsymbol{\theta}} = \mathbf{c}$. When the matrix $\mathbf{A}$ is invertible, we can solve for the parameters directly:

$$\hat{\boldsymbol{\theta}} = \mathbf{A}^{-1}\mathbf{c} \tag{5.10}$$

In order for $\mathbf{A}$ to be invertible, the training points must fully span $\mathbb{R}^{d+1}$. Naturally, this can happen only if $n \geq d+1$ and is therefore more likely to be the case when the dimension d is small in relation to the size of the training set n. Another consideration is the cost of actually inverting $\mathbf{A}$. Roughly speaking, you will need $O(d^3)$ operations for this. If $d = 10\,000$, this can be computationally expensive, making stochastic gradient updates more attractive.

In solving linear regression problems, the matrix $\mathbf{A}$ and the vector $\mathbf{c}$ are often written in a slightly different way. Specifically, $\mathbf{X} = [\boldsymbol{\phi}(x^{(1)}), \ldots, \boldsymbol{\phi}(x^{(n)})]^\top$. In other words, $\mathbf{X}^\top$ has each training feature vector as a column; $\mathbf{X}$ has them stacked as rows. If we also define $\mathbf{y} = [y^{(1)}, \ldots, y^{(n)}]^\top$ (column vector), then you can easily verify that

$$\mathbf{c} = \mathbf{X}^\top\mathbf{y} \quad \mathbf{A} = \mathbf{X}^\top\mathbf{X} \tag{5.11}$$

5.3 L_2 Regularization

However, what if a closed-form solution cannot be obtained? In this case, the training data provide no guidance on how to set some of the parameter directions. For example, if one of the coordinates $x_k^{(i)} = 0$ for all $i = 1, d \ldots n$, we would not know how to set the corresponding parameter θ_k. In this case, we say that the learning problem is ill-posed. The same issue also affects the stochastic gradient method, although initialization $\boldsymbol{\theta}(0) = 0$ helps set the parameters to zero for directions outside the span of the training examples, thus correcting the simple problematic case described above. The simple fix does not solve the broader problem, however, including when the coordinates of feature vectors are linearly dependent. How should we set the parameters when we have insufficient training data?

We will add a regularization term to the estimation criterion, as we did for classification. The purpose of this term is to bias the parameters towards a default answer such as zero. The regularization term will "resist" chainging parameters away from zero, even when the training data may

weakly tell us otherwise. This resistance is helps to ensure that our predictions generalize well. The intuition is that we opt for the "simplest answer" when the evidence is absent or weak.

There are many possible regularization terms that fit the above description. In order to keep the resulting optimization problem easily solvable, we will use $||\mathbf{w}||_2^2/2$ as the penalty. Specifically, we will minimize

$$J_{n,\lambda}(\boldsymbol{\theta}) = \frac{\lambda}{2}||\mathbf{w}||_2^2 + R_n(\boldsymbol{\theta}) = \frac{\lambda}{2}||\mathbf{w}||_2^2 + \frac{1}{2n}\sum_{i=1}^{n}\left(y^{(i)} - (\mathbf{w}\cdot\mathbf{x}^{(i)} + b)\right)^2 \tag{5.12}$$

where the *regularization parameter* $\lambda > 0$ quantifies the trade-off between keeping the parameters small (minimizing the squared norm $||\mathbf{w}||^2$) and fitting to the training data (minimizing the empirical risk $R_n(\boldsymbol{\theta})$). The offset b is usually not affected by the regularization. The use of this modified objective is known as **Ridge regression**.

While important, the regularization term introduces only small changes to the two estimation algorithms. For example, in the stochastic gradient descent algorithm, in each step, we will now move in the reverse direction of the gradient

$$\nabla_{\mathbf{w}}\left[\frac{\lambda}{2}||\mathbf{w}||_2^2 + \frac{1}{2}\left(y^{(i)} - (\mathbf{w}\cdot\boldsymbol{\phi}(\mathbf{x}^{(i)}) + b)\right)^2\right]\Big|_{\theta=\theta^{(k)}} = \lambda\mathbf{w}^{(k)} - \left(y^{(i)} - (\mathbf{w}^{(k)}\cdot\boldsymbol{\phi}(\mathbf{x}^{(i)}) + b)\right)\boldsymbol{\phi}(\mathbf{x}^{(i)})$$

As a result, the algorithm can be rewritten as:

Algorithm 6 Ridge Regression

procedure RIDGEREG($\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1,\ldots,n}, T$)
  $\mathbf{w}, b = \mathbf{0}, 0$
  for $t = 1, \ldots, T$ **do**
    Select $i \in \{1, \ldots, n\}$ at random
    $\mathbf{w}^{(t+1)} \leftarrow (1 - \lambda\eta_t)\mathbf{w}^{(t)} + \eta_t(y^{(i)} - (\mathbf{w}^{(t)}\cdot\boldsymbol{\phi}(\mathbf{x}^{(i)}) + b^{(t)}))\boldsymbol{\phi}(\mathbf{x}^{(i)})$
    $b^{(t+1)} \leftarrow b^{(t)} + \eta_t(y^{(i)} - (\mathbf{w}^{(t)}\cdot\boldsymbol{\phi}(\mathbf{x}^{(i)}) + b^{(t)}))$
  end for
  return $\mathbf{w}, b$
end procedure

As one might expect, compared to non-regularized regression, there is now a new factor $1 - \eta_t\lambda$ multiplying the current parameters $\boldsymbol{\theta}^{(t)}$, shrinking them towards zero with each update. When solving for the parameters directly, the regularization term only modifies the matrix $\mathbf{A} = \lambda\mathbf{I} + 1/n\,\mathbf{X}^\top\mathbf{X}$, where $\mathbf{I}$ is the identity matrix. The resulting matrix is always invertible as long as $\lambda > 0$. However, the cost of inverting it remains the same.

5.4 L_1 Regularization

L_1 regularization changes our objective function to be:

$$J_{n,\lambda}(\boldsymbol{\theta}) = \frac{\lambda}{2}||\boldsymbol{\theta}||_1 + R_n(\boldsymbol{\theta}) = \frac{\lambda}{2}||\boldsymbol{\theta}||_1 + \frac{1}{2n}\sum_{i=1}^{n}\left(y^{(i)} - (\mathbf{w}\cdot\mathbf{x}^{(i)} + b)\right)^2 \tag{5.13}$$

It almost seems that this is identical to L_2, but in fact, the L_1 regularization has a powerful benefit: it induces sparsity. L_2 causes the regression coefficients to be lower on average, but L_1 forces

some coefficients to be 0. This yields a kind of "automatic" feature selection. This is called **Lasso Regression** when you combine L_1 regularization with linear regression. A linear regression problem that includes both, L_1 and L_2, is referred to as **Elastic Net**.

The terms L_1 and L_2 come from the definition of norms. They indicate the coefficient used in the norm:

$$L_p(\mathbf{w}) = ||\mathbf{w}||_p = \left(\sum_{i=1}^{d} |w_i|^p \right)^{1/p}, \tag{5.14}$$

where $p = 1$ for L_1 and $p = 2$ for L_2. Others exist, such as $p = 0$ which simply counts dimensions and $p = \infty$ which selects the maximum element. The "L" comes from the word Lebesgue integral. If used as a distance and not as a norm, it is called the Minkowski distance after a Polish mathematician.

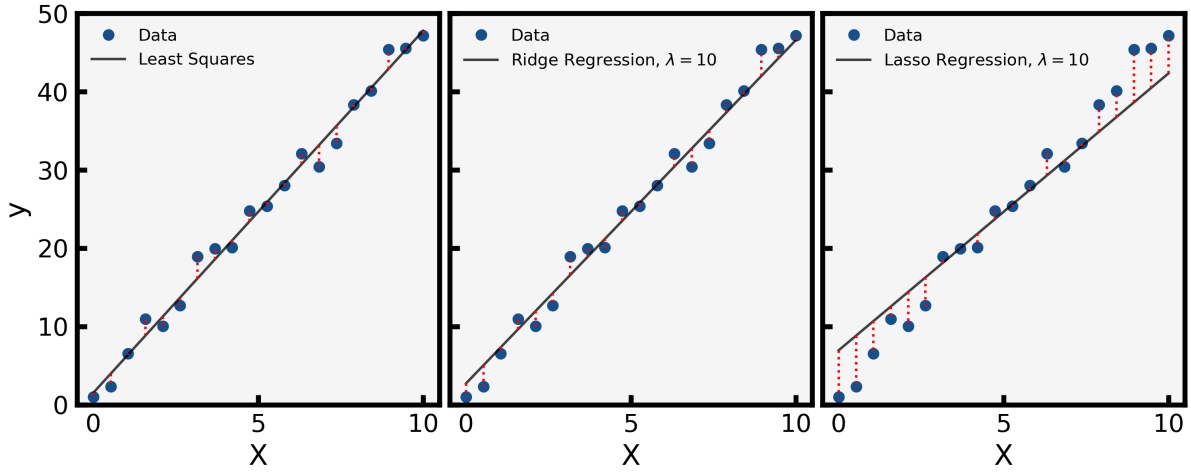

Figure 5.1: Comparison of (optimal) least squares regression with its regularized versions. Data as blue circles, linear fit in black, and residuals dashed in red. Left: Unregularized least squares fit. Center: Ridge Regression (L_2) with a regularization strength of $\lambda = 10$. Right: Lasso Regression (L_1) with a regularization strength of $\lambda = 10$, usually creates much stronger regularization.

5.5 The effect of regularization

The regularization term shifts the emphasis away from the training data. As a result, we expect that higher values of λ will have a negative impact on the training error. Specifically, let $\hat{\boldsymbol{\theta}} = \hat{\boldsymbol{\theta}}(\lambda)$ denote the parameters that we would find by minimizing the regularized objective $J_{n,\lambda}(\boldsymbol{\theta})$. We view $\hat{\boldsymbol{\theta}}(\lambda)$ here as a function of λ. We claim then that $R_n(\hat{\boldsymbol{\theta}}(\lambda))$, i.e.,mean squared training error, increases as λ increases. If the training error increases, what is the benefit? Larger values of λ often lead to a lower generalization error, as the noise in the data has a smaller impact. Put another way, it becomes harder to overfit the training data. This benefit accrues for a while as λ increases, then turns hindering learning. Biasing the parameters towards zero too strongly, even when the data tell us otherwise, will eventually hurt generalization performance. As a result, you will see a typical U-shaped curve in terms of how the generalization error depends on the regularization parameter.

6 Soft Classification

We saw that for many hard classifiers (only returning labels that are class labels), we need special training algorithms. The perceptron and the averaged perceptron had their own way of training. Even the support vector machine could be trained via gradient descent, but always with the caveat whether a training point was outside the margin or not.

However, for the remainder of this course, we will use SGD-like algorithms to train our models. Therefore, we need classification algorithms that can be trained without an if-else logic.

6.1 Softmax Classifier or Logistic Regression

The key idea is to transform the hard class label into a soft, differentiable one. Here, 'soft' implies that it can take any value between 0 and 1, indicating the probability with which the model predicts the positive class. This can be accomplished with a minor modification to the original function:

$$h(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + b) \tag{6.1}$$

where σ is the **sigmoid** function

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{6.2}$$

The sigmoid function accepts inputs within the range $[-\infty, \infty]$ and produces outputs that are probabilities within $[0, 1]$. The values fed into the sigmoid are known as log-odds, or logits. Odds represent the ratio of probabilities; for instance, odds of 1 indicate equal probabilities of 0.5 for both class 1 and class 0, while odds of 2 mean a probability of 0.67 for class 1 and 0.33 for class 0. Log-odds are the natural logarithms of these ratios, implying that a log-odds of 0 corresponds to odds of 1 and an output probability of 0.5. This type of binary classifier is often referred to as **logistic regression** due to its reliance on performing regression on logits.

In essence, we have substituted the perceptron's inequality with a smooth and differentiable alternative. This approach gives us a soft classification, as we provide probabilities of class membership rather than definitive assignments. Consequently, we must also adjust our loss function.

6.2 Cross-Entropy

An alternative loss function that is more effective for classification tasks is cross-entropy. Although you can try using mean absolute error or mean squared error (both are discussed in the next Section) for classification, cross-entropy typically outperforms them.

Cross-entropy is a loss function that measures the difference between two probability distributions. When it is minimized, the two distributions become more similar. Cross-entropy is a simpler form of the Kullback–Leibler divergence, which is a directional measure of the difference between two probability distributions. Therefore, it is technically not a distance because it is not symmetric with respect to its inputs, but in practical terms, it is sufficiently similar to a distance, so we treat it as such.

For multi-class classifier, cross-entropy is defined as

$$L = -\sum_{k}^{K} y_k \ln \hat{y}_k \tag{6.3}$$

where K is the number of classes and we assume that $\sum_k^K y_k = 1$ as well as $\sum_k^K \hat{y}_k = 1$. Since, for a binary classifier, we only have two classes whose probabilities have to sum up to one, we can simplify the above expression to

$$L(\mathbf{x}^{(i)}, \boldsymbol{\theta}; y^{(i)}) = -[y \ln \hat{y} + (1-y)\ln(1-\hat{y})] \tag{6.4}$$

Hence, the non-regularized objective function would be

$$J_n(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y^{(i)} \ln \sigma(\mathbf{w}\cdot\mathbf{x}^{(i)} + b) + (1-y^{(i)})\ln(1-\sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b))\right] \tag{6.5}$$

However, to avoid numerical instability in practice when using cross-entropy sigmoid classification you should use a model that outputs the logits and you use a loss function that works on logits instead of probability, e.g.,

$$L(\mathbf{x}^{(i)}, \boldsymbol{\theta}; y^{(i)}) = \max(\mathbf{w}\cdot\mathbf{x}^{(i)} + b, 0) - \left(\mathbf{w}\cdot\mathbf{x}^{(i)} + b\right) y^{(i)} + \ln(1 + e^{-|\mathbf{w}\cdot\mathbf{x}^{(i)}+b|})$$

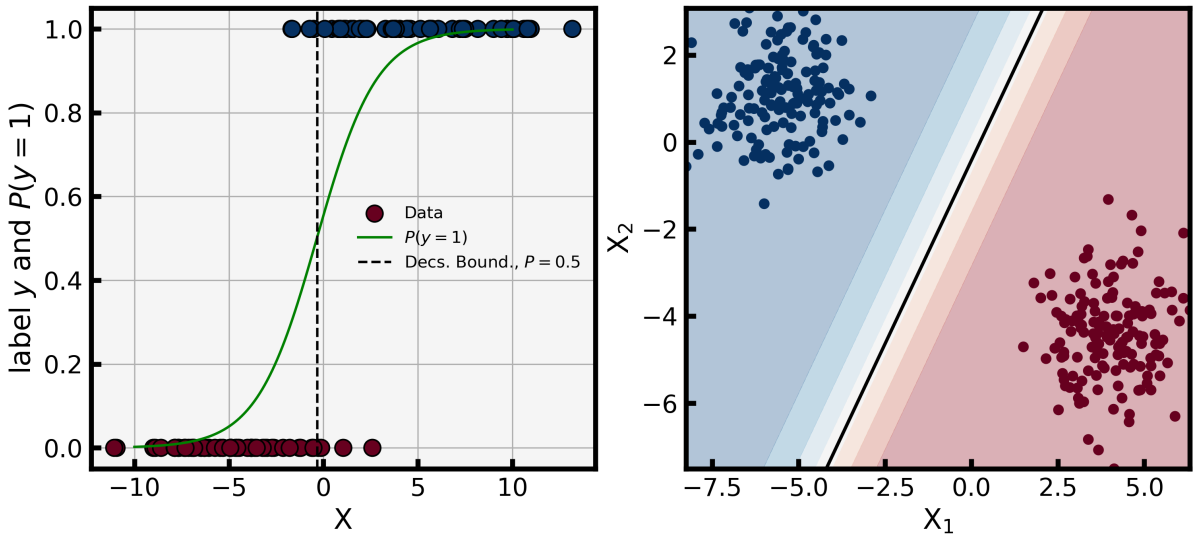

Figure 6.1: Depictions of Logistic Regression. Left: One dimensional features and class labels 0 and 1. The green curve shows the learned, sigmoidal probability of $y = 1$. The decision boudary dashed line is considered to be at $P = 0.5$. Right: Two dimensional example, the class probability is visualized as background color. Decision boundary as solid balck line.

7 Model Evaluation and more Details on Training

In this section, we shall take a closer look at how to train a model, to observe the training progress, and finally how to judge the trained model. We will also discuss different strategies for choosing hyperparameters, of which we have encountered so far one, the regularization strength λ.

7.1 Learning Curve and Batched Learning

The curve shown in figure 7.1 is called a *training curve* for parameter updates based on all available data at once. These curves show whether the loss is decreasing during training, indicating that the model is learning. Training curves are also called *learning curves*. The x-axis may be the sample number, total iterations through the data set (called epochs), or some other measure of the amount of data used for training the model.

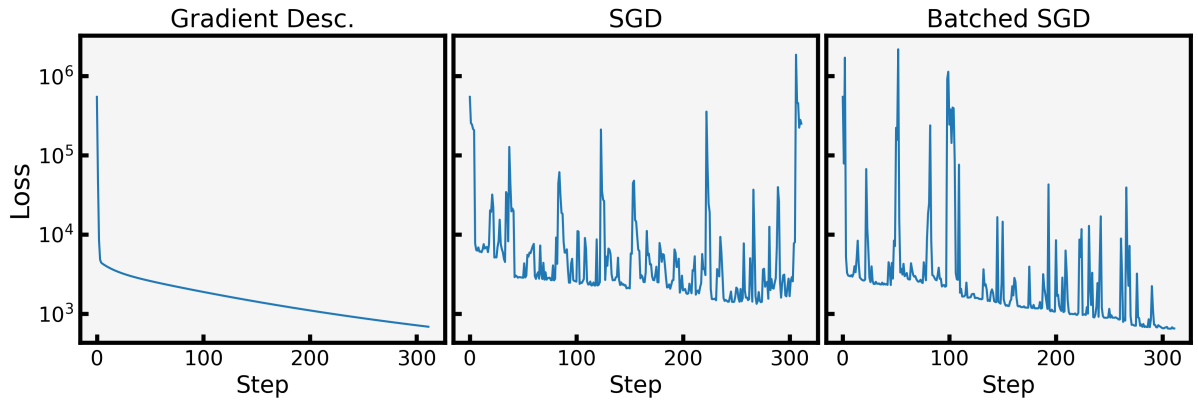

Figure 7.1: Learning Curves, loss (mean squared error) computed for current parameter values for the entire data set. Left: Gradient descent, each step includes computing the objective function and its gradient for the entire data set. The curve is extremely smooth. Center: Stochastic gradient descent that uses only a single sample per update step. Every step creates very different parameter updates, hence the loss function changes rapidly. Here only a fraction of the data set is used. Right: Batched stochastic gradient descent, with a batch size of 32. At every iteration a random 32 samples are used to compute the gradients of the objective function and update the parameters. At the end of the curve each data point has been used once.

As discussed previously, using all data for a single parameter update is costly and can actually lead to be mislead by the noise in the data set. However, using only a single sample as shown so far in stochastic gradient descent might be too noisy to reach convergence. Therefore, one usually settles for a balanced approach, called batching, which is how training is mostly done in machine

learning. The small change is that, rather than using all data at once, we only take a small batch of data. Batching provides two benefits: It reduces the amount of time it takes to compute an update to our parameters and makes the training process somewhat random. The randomness makes it possible to escape local minima that might stop training progress. Hence, the algorithm is also stochastic and is therefore also referred to as stochastic gradient descent (SGD). SGD and variations of it are the most common methods of training in deep learning.

When comparing the panels in Fig. 7.1, there are three changes to note:

1. The loss is lower at the end than without batching.
2. There are more steps, even though we iterated over our dataset once instead of 10 times.
3. The loss does not always go down.

The reason the loss is lower is, because we are able to take more steps even though we only see each data point once. That's because we update the parameters at each batch, giving more updates per iteration over the data set. Specifically, if B is batch size, there are n/B updates for every 1 update in the original gradient descent, which used all n examples. The reason why the loss does not always decrease is that each time we evaluate it, it is on a different subset of S_n. Some examples are harder to predict than others. Furthermore, each update step we take to minimize the objective function may not be correct because we only update the parameters based on one batch. Assuming that the batches are mixed though, we will always improve on average.

7.2 Feature Normalization

When featurizing real-life objects, such as molecules, it is often the case that the features have drastically different sizes and units, e.g., boiling points, solubilities, and molar weights. This can cause learning to stall, as some gradients are very large and some are very small, due to the different magnitudes of the features. Each of these features must use the same learning rate, η, which has the right size for some and is too large or small for others. A standard trick is to make all the features have the same magnitude, using standardization:

$$\phi_j(\mathbf{x}^{(i)}) \leftarrow \frac{\phi_j(\mathbf{x}^{(i)}) - \bar{\phi}_j}{\sigma(\phi_j)} \tag{7.1}$$

where $\bar{\phi}_j$ is mean of the j^{th} feature and $\sigma(\phi_j)$ its standard deviation. To avoid contaminating training data with test data (i.e., leaking information between train and test data), training data should only be used to compute the mean and standard deviation. We want our test data to approximate how the model would perform on unseen data, therefore, we cannot know what these unseen features means and standard deviations might be, and thus cannot use them at training time for standardization.

7.3 Analyzing Classifier Performance

In evaluating the performance of a classifier, several metrics and visual tools are used to understand how well the model performs on a given data set. These metrics help to assess the strengths and weaknesses of the classifier and compare different models.

Basically, all the metrics presented below are based on the confusion matrix, a tabular representation of the correct and incorrect prediction of the labels -1 and $+1$.

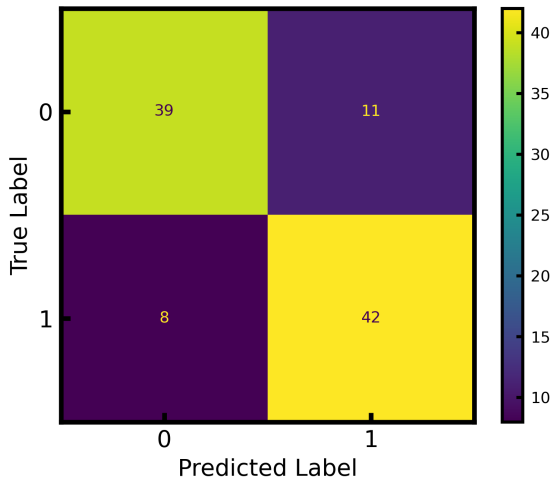

Figure 7.2: Confusion matrix that counts and divides up the samples by comparing ground truth and predicted label. The entries of the confusion matrix are used to determine the quality of the classifier.

7.3.1 Types of Error

When analyzing classifier performance, it is essential to consider the types of errors that the model can make. These errors are typically summarized in a confusion matrix (Fig. 7.2), which provides a comprehensive view of the classifier's performance.

- **True Positives (TP):** The number of correct positive predictions.
- **True Negatives (TN):** The number of correct negative predictions.
- **False Positives (FP):** The number of incorrect positive predictions (Type I error).
- **False Negatives (FN):** The number of incorrect negative predictions (Type II error).

From these values, several performance metrics can be derived, such as precision, recall, accuracy, and F1 score.

7.3.2 Accuracy vs. Precision

Accuracy and precision are commonly used metrics, but they measure different aspects of classifier performance.

Accuracy

Accuracy is the proportion of true results (both true positives and true negatives) among the total number of cases examined. It is defined as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{7.2}$$

While accuracy is a useful metric, it can be misleading, especially in cases of imbalanced datasets where one class significantly outnumbers the other.

Precision

Precision, also known as *positive predictive value* (PPV), measures the proportion of true positives among all positive predictions. It is defined as:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{7.3}$$

Precision is particularly useful when the cost of false positives is high. It provides insight into the exactness of the classifier.

7.3.3 Recall and F1 Score

Recall

Recall, also known as sensitivity or true positive rate, measures the proportion of true positives among all actual positives. It is defined as:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{7.4}$$

Recall is important when the cost of false negatives is high, as it provides insight into the classifier's ability to identify all positive instances.

F1 score

The F1 Score is the harmonic mean of precision and recall, providing a single metric that balances both concerns. It is defined as:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{7.5}$$

The F1 score is useful for comparing classifiers when there is an uneven class distribution.

7.3.4 Receiver Operating Characteristic (ROC) Curve

The **ROC curve** is a graphical representation of a soft classifier's performance across different threshold settings (it determines how a probability is converted to a positive label). It plots the true positive rate (recall) against the false positive rate (FPR), which is defined as:

$$\text{FPR} = \frac{FP}{FP + TN} \tag{7.6}$$

The area under the ROC curve (AUC) provides a single scalar value to summarize the overall performance of the classifier. An AUC of 1 represents a perfect classifier, while an AUC of 0.5 indicates a classifier with no discriminative power.

7.3.5 Other Metrics

Specificity

Specificity, also known as the true negative rate, measures the proportion of true negatives among all actual negatives. It is defined as:

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{7.7}$$

Specificity is crucial when the cost of false positives is high.

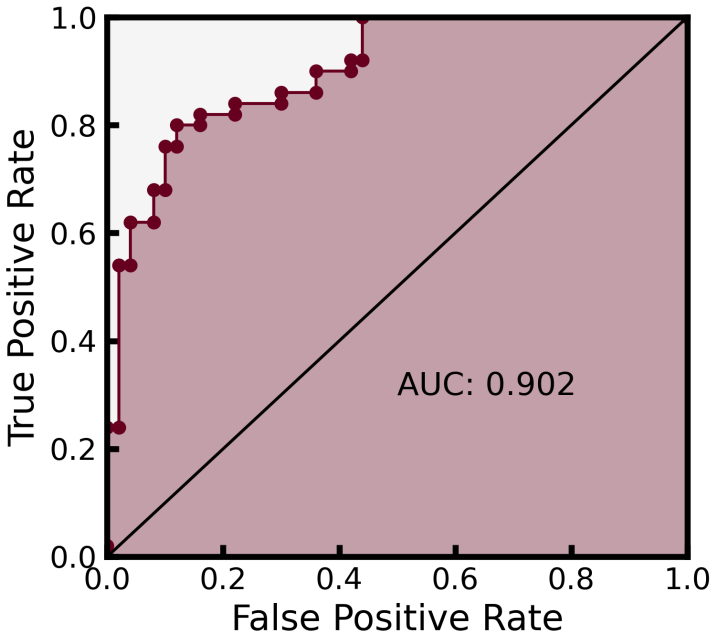

Figure 7.3: Receiver operating characteristic (ROC) curve for the same Logistic regression model that produced the confusion matrix in Fig. 7.2. The area under the curve (AUC) is significantly larger than 0.5, indicating a well performing model.

Matthews Correlation Coefficient

The Matthews Correlation Coefficient (MCC) is a balanced measure that accounts for all four quadrants of the confusion matrix and is especially useful for imbalanced datasets. It is defined as:

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{7.8}$$

The MCC ranges from -1 (total disagreement) to +1 (perfect prediction), with 0 indicating a random guess.

Balanced accuracy

Balanced accuracy is the average of sensitivity (recall) and specificity, providing a more balanced view of performance on imbalanced datasets. It is defined as:

$$\text{Balanced Accuracy} = \frac{\text{Sensitivity} + \text{Specificity}}{2} \tag{7.9}$$

This metric helps in understanding the classifier's performance by considering both the ability to identify positives and avoid false positives.

Cohen's Kappa

Cohen's Kappa measures the agreement between the predicted and actual classifications, adjusted for the possibility of random agreement. It is defined as:

$$\kappa = \frac{p_o - p_e}{1 - p_e} \tag{7.10}$$

where p_o is the observed agreement,

$$p_o = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$$

and p_e is the expected agreement by chance.

$$p_e = \frac{(\mathrm{TP} + \mathrm{FP})(\mathrm{TP} + \mathrm{FN}) + (\mathrm{TN} + \mathrm{FP})(\mathrm{TN} + \mathrm{FN})}{(\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN})^2}$$

Kappa values range from -1 (complete disagreement) to 1 (perfect agreement), with 0 indicating random chance.

7.4 Analyzing Regression Model Performance

This is a large topic, but the first thing one typically examines in supervised learning is a **parity plot**, which shows our predictions ($\hat{y}$) vs. the true label y. This plot is independent of the dimension of the features, as it just examines the label. A perfect fit would fall on the line at $y = \hat{y}$.

The final assessment of the model can be done with the loss function, but other metrics are typically used.

7.4.1 Mean squared error

The mean squared error (MSE) is often used for loss functions due to its smooth derivative and penalization of outliers. However, unlike all other error metrics described in the following, it does not have the unit of the label, but its square.

$$\mathrm{MSE} = \frac{1}{n'} \sum_{j}^{n'} \left(y^{(j)} - \hat{f}(\mathbf{x}^{(j)}) \right)^2 \tag{7.11}$$

7.4.2 Mean absolute error

The mean absolute error (MAE) measures the mean deviation of a prediction from its target, disregarding the sign.

$$\mathrm{MAE} = \frac{1}{n'} \sum_{j}^{n'} \left| y^{(j)} - \hat{f}(\mathbf{x}^{(j)}) \right| \tag{7.12}$$

Simply put, it is the average absolute vertical or horizontal distance between each point in a scatter plot and the $x = y$ diagonal line.

7.4.3 Mean signed deviation

The mean signed deviation (MSD) or sometimes also mean signed error is simply the mean of all deviations.

$$\mathrm{MSD} = \frac{1}{n'} \sum_{j}^{n'} y^{(j)} - \hat{f}(\mathbf{x}^{(j)}) \tag{7.13}$$

The MSD is thus sensitive to whether a model overestimates or underestimates the label. However, since models frequently do both, this can result in coincidental error cancelation, meaning that the

positive and negative ($y^{(j)} - \hat{f}(\mathbf{x}^{(j)})$) values offset each other, leading to low absolute MSD values. Due to this phenomenon, the MSD value is often much lower than the MAE, which can falsely suggest that the model is more accurate than it actually is. Therefore, relying solely on the MSD is not a reliable measure of a model's performance. Note that the abbreviation MSE typically refers to the mean squared error.

7.4.4 Root mean squared deviation

The root mean squared deviation (RMSD) or error (RMSE), defined as

$$\text{RMSD} = \sqrt{\frac{1}{n'}\sum_{j}^{n'}(y^{(j)} - \hat{f}(\mathbf{x}^{(j)}))^2} = \sqrt{\text{MSE}} \tag{7.14}$$

is similar to the MAE, but gives due to the square a larger weight to outliers. Therefore, RMSD≥MAE. MAE possesses advantages in interpretability over RMSD, as it is the average of the absolute values of the errors. Furthermore, each error influences MAE in proportion to the absolute value of the error, which is not the case for RMSD, as larger errors influence it more strongly.

7.4.5 Mean Absolute Percentage Error and Mean absolute scaled error

The mean absolute percentage error (MAPE) is defined as

$$\text{MAPE} = 100\frac{1}{n'}\sum_{j}^{n'}\left|\frac{y^{(j)} - \hat{f}(\mathbf{x}^{(j)})}{y^{(j)}}\right| \tag{7.15}$$

The absolute error between the prediction $\hat{f}(\mathbf{x}^{(j)})$ and the actual value $y^{(j)}$ is normalized by the absolute value of the target $y^{(j)}$. MAPE is sometimes utilized as a loss function in regression tasks and for model assessment, owing to its intuitive interpretation in terms of relative error. Although MAPE seems straightforward and persuasive, it presents significant drawbacks in practical use. It becomes unusable when y values are zero or near-zero, as this causes the division to approach infinity. Predictions that are too small in magnitude cannot exceed a 100% error, while there is no upper bound in the opposite direction, which poses an issue when dealing with variables that lack a meaningful zero or negative values, such as temperatures in Kelvin. This can introduce bias when using MAPE as a loss function. An alternative could be the logarithmic ratio $\log \frac{\hat{f}(\mathbf{x}^{(j)})}{y^{(j)}}$.

Another measure that offers a solution to many of these problems is the mean absolute scaled error (MASE), which for non-time series data is defined as

$$\text{MASE} = \frac{1}{n'}\sum_{j}^{n'}\frac{\left|y^{(j)} - \hat{f}(\mathbf{x}^{(j)})\right|}{\frac{1}{n'}\sum_{k}^{n'}\left|y^{(k)} - \bar{y}\right|} = \frac{\sum_{j}^{n'}\left|y^{(j)} - \hat{f}(\mathbf{x}^{(j)})\right|}{\sum_{k}^{n'}\left|y^{(k)} - \bar{y}\right|} \tag{7.16}$$

where $\bar{y}$ is the mean of the labels in the set. Hence, MASE is the ratio of Mean absolute error (MAE) and the Mean Absolute Deviation (MAD) within the data set from the mean.

7.4.6 Pearson correlation coefficient

In a regression setting, one often computes the Pearson correlation coefficient ρ, which measures the *linear* correlation between two sets of data. It is a normalized measurement of the covariance

such that the result always falls in the range $[-1, 1]$, where -1 is a perfect anticorrelation (as x increases y decreases linearly) and 1 is a perfect positive correlation (as x increases y also increases linearly). The Pearson correlation coefficient is mathematically defined as

$$\rho_{X,Y} = \frac{\text{cov(X,Y)}}{\sigma_X \sigma_Y} = \frac{\mathbb{E}[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y} = \frac{\mathbb{E}[XY]-\mathbb{E}[X]\mathbb{E}[Y]}{\sqrt{(\mathbb{E}[X^2]-\mathbb{E}[X]^2)(\mathbb{E}[Y^2]-\mathbb{E}[Y]^2)}}\,, \tag{7.17}$$

where $\mathbb{E}[X] = \mu_X$ of the random variable X is its expectation or mean.

7.4.7 Coefficient of determination

Another useful measure is the coefficient of determination, usually denoted by R^2 and actually pronounced "R squared". R^2 measures how well observed outcomes are predicted by the model, based on the proportion of total variation of outcomes explained by the model. Normally, R^2 ranges from 0 to 1, but can assume values in $[-\infty, 1]$. In cases where R^2 is negative, just taking the mean of the data would provide a better fit to the results than the model. R-squared is generally defined as

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} = 1 - \frac{\text{MSE}}{\text{var}(y^{(i)})}\,, \tag{7.18}$$

where SS_{res} is the sum of the squares of all residuals

$$SS_{\text{res}} = \sum_{j}^{n'} (y^{(j)} - \hat{f}(\mathbf{x}^{(j)}))^2$$

and SS_{tot} is the total sum of squares, measured from the mean of the data ($\bar{y}$)

$$SS_{\text{tot}} = \sum_{j} (y^{(j)} - \bar{y})^2$$

$R^2 = 1$ means that the modeled values perfectly match the observed values, hence $SS_{\text{res}} = 0$. A baseline would be a model that always predicts $\bar{y}$. This would yield $R^2 = 0$.

7.4.8 Spearman's rank correlation coefficient

Spearman's rank correlation coefficient denoted by either ρ or r_s is a measure of rank correlation. It assesses how well the relationship between two variables describes a monotonic function. The Spearman correlation between two variables is equal to the Pearson correlation between the rank values of those two variables, where the ranking of a series of numbers is their index if the numbers were sorted from small to large. Where Pearson's correlation measures how linear the relationship between variables is, Spearman's correlation assesses only if the relationship is monotonic (linear or not). A perfect Spearman correlation coefficient of $+1$ occurs when one variable increases when the other increases, -1 occurs when one decreases whenever the other increases.

As mentioned above, Spearman's rank correlation coefficient is equal to the Pearson's correlation coefficient between the rank variables:

$$r_s = \rho_{R(X),R(Y)} = \frac{\text{cov}(R(X), R(Y))}{\sigma(R(X))\sigma(R(Y))} \tag{7.19}$$

where $R(X)$ denotes the rank coefficient of the variable X.

The rank correlation is important, as sometimes we may not be able to learn the generating function, but getting the ranking right is enough. That means that when we compare two samples $\mathbf{x}$ and $\mathbf{x}'$ we can confidently say which will have the larger signal, even if it is not the ground truth value.

7.5 Overfitting and the Effect of Number of Features

7.5.1 Overfitting

Overfitting often occurs when training a complex model with many parameters on a small data set. This means that the model fits the training data perfectly but performs poorly on unseen data. As illustrated in the left side of Fig. 7.4, both training and test loss initially decrease during training. However, beyond a certain point, the training loss continues to decrease while the test loss increases, indicating overfitting. Overfitting results from training for too many steps or using too many parameters, often caused by having unnecessary features, leading the model to learn noise in the training data rather than the underlying process.

Experimental data invariably contains noise, meaning that even if our model $\hat{f}(\mathbf{x})$ perfectly replicates the generating process $f(\mathbf{x})$, it won't fit the data perfectly due to measurement noise:

$$y = f(\mathbf{x}) + \epsilon \tag{7.20}$$

where ϵ is (ideally) Gaussian noise with zero mean and unknown variance. When fitting our model, the noise is fixed because the labels $y^{(i)}$ are fixed for each datum. Consequently, we may inadvertently learn the sum $f(\mathbf{x}) + \epsilon$ instead of just $f(\mathbf{x})$. This learned noise degrades the performance on new data, which has different noise.

Overfitting is almost inevitable with real data due to noise and imperfect features. It can be assessed by splitting the data into training and test sets, and further mitigated using a train-test-validation split, which will be discussed in the context of deep models. Overfitting occurs when the model has too many parameters or uses features that do not correlate with the labels.

7.5.2 Effect of Number of Features

As discussed, overfitting is sensitive to the number and choice of features, which makes feature selection critical in supervised learning. When describing molecules, one can use physical descriptors, create linear combinations, apply nonlinear functions, or use data from experiments, simulations, and quantum calculations, leading to an unlimited number of possible descriptors.

For simple models, three regimes can usually be distinguished:

1. **Underfitting:** Too few features fail to describe the generating process adequately, resulting in underfitting. Nonlinear combinations of input features might help as they better describe the problem.

2. **Optimal Fit:** Increasing the number of features improves the model up to a point, providing a good balance between bias and variance.

3. **Overfitting:** Further increasing the number of features leads to overfitting. Linear methods become unstable when the number of features approaches or exceeds the number of data points.

The risk of overfitting decreases as the size of the data set increases because noise effects diminish with more data. As a general rule of thumb, to double the number of features, you should quadruple the number of data points to reduce the risk of overfitting. Thus, there is a strong relationship between the complexity of the model, the achievable accuracy, the required data, and the noise of the labels.

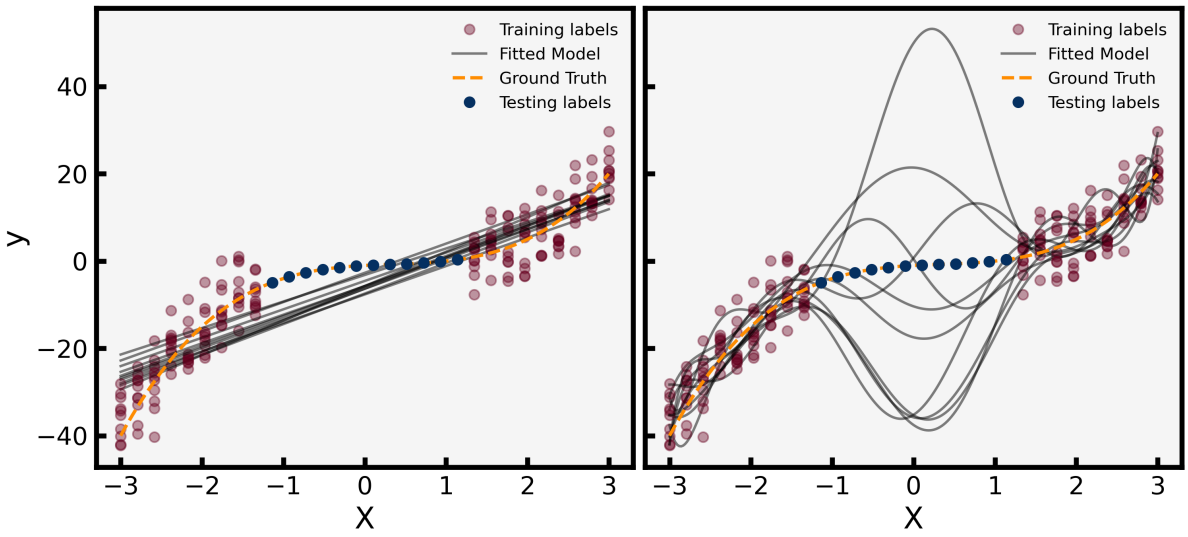

Figure 7.4: Bias-Variance trade off. The generating, cubic model is shown in orange. The training points are are noisy, each model used a different subset of the training points. Left: High bias models, as they are linear and by construction do not allow to fully fit the cubic generating model. Right: High variance models, high-order polynomials are fitted to the same subsets of training points.

7.6 Bias-variance tradeoff

In machine learning, the **bias-variance trade-off** explains the relationship between a model's complexity, its prediction accuracy, and its ability to generalize to unseen data. Generally, as we increase the number of tunable parameters in a model, it becomes more flexible and fits the training data, resulting in a lower bias. However, this increased flexibility also leads to greater variability in the model's performance when trained on different datasets, indicating a higher variance in the model's estimated parameters (see Fig. 7.4).

The bias-variance dilemma or problem arises from the challenge of minimizing the two aforementioned sources of error that hinder supervised learning algorithms from generalizing beyond their training set:

- **Bias error** stems from incorrect assumptions in the learning algorithm. High bias can lead to underfitting, where the algorithm fails to capture the relevant relationships between features and target outputs.
- **Variance error** arises from the sensitivity of the algorithm to small fluctuations in the training set. High variance can result in overfitting, where the algorithm models the random noise in the training data.

We use bias-variance decomposition to break down the error into three components: bias, variance, and an irreducible error due to the inherent noise in the problem itself.

$$\begin{aligned}\mathbb{E}_{S_n}\left[(y-\hat{f}(\mathbf{x};S_n))^2\right] &= \mathbb{E}_{S_n}\left[(f(\mathbf{x})+\epsilon-\hat{f}(\mathbf{x};S_n))^2\right] \\ &= \underbrace{\mathbb{E}_{S_n}\left[f(\mathbf{x})-\hat{f}(\mathbf{x};S_n)\right]^2}_{\text{Bias}} + \underbrace{\mathbb{E}_{S_n}\left[\left(\mathbb{E}_{S_n}\left[\hat{f}(\mathbf{x};S_n)\right]-\hat{f}(\mathbf{x};S_n)\right)^2\right]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{irreducible noise}}\end{aligned} \tag{7.21}$$

The equations above are expectations over the training set S_n, i.e., in practice one would split the data set any possible way, compute these values and take the mean. The first term (Bias) describes how well the model can possibly fit the generating function over any possible training set. The variance shows how much the fit changes based on the training set. The last term is the irreducible noise inherent in the data that we cannot remove.

7.7 Strategies to Assess Models

We will now discuss more ways to assess model performance. What will we report as the certainty in a new point? Intuitively, the test error is a measure of the expected error on an unseen data point, but it might be nice to even report some form of confidence interval.

7.7.1 k-fold Cross-Validation

The bias–variance decomposition introduced above shows that our testing error $\mathbb{E}\left[(y-\hat{f}(\mathbf{x}))^2\right]$ is sensitive to the training data chosen. Usually, we have only a single value for the MSE, as we only perform a single data split. One way to better estimate the value on unseen data is to repeat the process of splitting the data into training and testing multiple times. This is called k-fold cross-validation, where k is the number of times you repeat the process. k-fold cross-validation (CV) is useful because certain high-variance model choices can give different testing errors depending on the train/test split. k-fold also provides multiple samples so that you can estimate the uncertainty in testing error. Typically, k-fold cross-validation is not done for very large data sets, because label noise dominates and training+testing a model k times can be time consuming and yields very similar results.

In k-fold CV, we split our data into k segments. Then we train on $k-1$ segments and test on the last segment. Since 80:20 is a typical ratio that balances having enough data for good training and enough to robustly assess how well your model performs, 5-fold cross-validation is very common.

The test statistics should not be very sensitive to k. This is good, since the choice of k is somewhat arbitrary. Larger k means more samples, but each test data is smaller, so these two effects should balance out. Only for small data sets, we expect some dependence on k as it will matter more how large the training set is compared to the over all data set.

7.7.2 Leave-one-out CV

In leave-one-out cross-validation (LOOCV), we use the largest possible k, namely $k = n$. In this case, every datum is its own segment. Here we create n different models, one for each left-out data point, so it is only used for small datasets. Thus, the advantage of LOOCV is that it maximizes the training data but maximizes the number of times the model needs to be trained.

7.7.3 Bootstrap Resampling

To estimate quantiles, we need to have a series of observations of predictions. For example, we could do 5-fold cross-validation and have 5 estimates of $\hat{f}_k(\mathbf{x})$ and could estimate the quantiles using a t-statistic.

Here, we will introduce a method called bootstrap resampling instead, which removes the restriction that we can only use $(k-1)/k$ of the data set. Bootstrap resampling is a general process for estimating uncertainty for empirical statistics without assuming a certain probability distribution (i.e., nonparametric). In bootstrap resampling, we create as many new training sets as desired that are the same size as the original by sampling with replacement from the original set. That means our new data set has fewer unique examples than the original, but makes up the difference with duplicates. This means that we can potentially generate 2^n new datasets.

In this way we create N training sets of size n and train a model for each of them, then we can use the N models to perform prediction for the same unseen point $\mathbf{x}'$. Using the 25%, 50%, and 75% quantiles, we can report median (50% quantile) and confidence interval $((q_{75}-q_{25})/2)$ thanks to bootstrap resampling.

There are a few disadvantages, though. i) we need to produce and keep N models, one for each bootstrap resample. For good statistics, one needs to choose N of the size of 100. ii) this process does not give a reportable test error, since we only use a single prediction point. These prediction intervals also underestimate the model bias, because it has no estimate of the test error. It only accounts for variation due to the training data. Using the language above, it only accounts for model variance, but not model bias.

In summary, bootstrap resampling is an excellent technique that is often used to estimate uncertainties. However, it is not a great choice for estimating model error on unseen points.

7.7.4 Jacknife

An alternative technique that considers model variance similar to the bootstrap method and model bias akin to the k-fold cross-validation method is termed Jacknife. Instead of the usual practice, we employ LOOCV to generate an ensemble of n models (a subsample can be used if n is excessively large) and determine the models' test error on the excluded test instance. The resulting quantile estimates integrate the variance from the diversity of models (model variance) and each model's individual test error (model bias). Specifically, we calculate the residual:

$$R_i = \left| y^{(i)} - \hat{f}(\mathbf{x}^{(i)}; S_n \setminus \mathbf{x}^{(i)}) \right| \tag{7.22}$$

We then go on to compute the two sets of adjusted predictions

$$S_{\text{low}} = \{\hat{f}(\mathbf{x}'; S_n \setminus \mathbf{x}^{(i)}) - R_i\}_{i=1,\ldots,n}$$
$$S_{\text{high}} = \{\hat{f}(\mathbf{x}'; S_n \setminus \mathbf{x}^{(i)}) + R_i\}_{i=1,\ldots,n}$$

we then report the median of $\hat{f}(\mathbf{x}'; S_n \setminus \mathbf{x}^{(i)})$ and the confidence interval as $(q_{95}^{\text{high}} - q_5^{\text{low}})/2$ (the difference of the 95% quantile of S_{high} and the 5% quantile of S_{low}). This method combines the ensemble of prediction models given by bootstrap resampling with the error estimates from LOOCV.

The uncertainty is often significantly greater, which probably reflects real-world scenarios more accurately. For small datasets (n ranging from 1 to 1000) where models can be trained swiftly, the Jackknife method is recommended. To reduce computational expense, instead of performing exhaustive LOOCV, you can use a randomized approach by conducting only a few iterations of LOOCV, thus minimizing the number of models to compute.

7.8 Shift in data distribution

We often assume that the testing data comes from the same probability distribution as our training data. This is true when randomly split the data, but can be violated when we actually get new data to make predictions with, which is totally different from anything in the training and/or test set.

A **covariate shift** happens when the distribution of features changes. Covariate is in mathematics another word for the features. An example might be that the molecular weights of your molecules are larger in your testing data. The relationship between features and labels $f(\mathbf{x})$ remains the same, but the distribution of features is different. In contrast, a **label shift** means that the distribution of labels has changed. Perhaps our training data were all very soluble molecules, but at test time, we examined mostly insoluble molecules. Again, the fundamental relationship that we try to estimate with our model still holds (in this case, the physical laws that govern solubility).

There are two common reasons unseen data can be out of the training data distribution. i) you are extrapolating to new regions of chemical space. For example, you have training data of drug activities. You make a model that can predict activity and try to find the most active drug molecule. However, this molecule is likely to be unusual and is not in your training data, as otherwise you would already be done. Thus, you will be pushing your model to regions outside of your training data. ii) Another reason you can be out of training data is that the way you generated training data differs from how the model is used. For example, you trained on molecules that do not contain fluorine. Then, you try your model on molecules that contain fluorine. Your features will be different from what you observed in training.

In general, the result of leaving your training data distribution is that your test error increases and the estimates you provide will be too low. In order to get a grasp on how well your model generalizes (it is not affected by these shifts), one can split the data set not randomly but by domain; e.g., during training one splits the molecules into those with and without fluorine, or soluble and insoluble. This is often a good way to test the robustness of your model prior to application to completely new data.

8 Non-Linear Models I: Kernel Methods

8.1 Introduction

Let us begin by recalling how linear classifiers can be used to make nonlinear predictions. One straightforward method is to map every example $\mathbf{x} \in \mathbb{R}^d$ to corresponding feature vectors $\boldsymbol{\phi}(\mathbf{x}) \in \mathbb{R}^p$, where p is typically much larger than d. We then apply a linear classifier to these new (higher-dimensional) feature vectors, as if they were the original inputs. In doing so, any linear classifiers we have learned previously remain valid, yet they yield nonlinear classifiers in the original coordinate space.

There are many ways to create such feature vectors. For example, we can build $\boldsymbol{\phi}(\mathbf{x})$ by concatenating polynomial terms of the original coordinates, which in 2D might give us a five-dimensional feature vector.

$$\boldsymbol{\phi}(\mathbf{x}) = \left(x_1, x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2\right)^\top$$

We can then train a "linear" classifier (linear in the new coordinates). In other words, our training set is now $S_n = \{(\boldsymbol{\phi}(\mathbf{x}^{(i)}), y^{(i)}) | i = 1, \ldots, n\}$. The estimates of the resulting parameters $\hat{\mathbf{w}}$ and $\hat{b}$ define a linear decision boundary in the ϕ-coordinates but a nonlinear boundary in the original x-coordinates.

The primary issue with the aforementioned method is that the feature vectors $\boldsymbol{\phi}(\mathbf{x})$ can become very high-dimensional. For example, if we begin with $\mathbf{x} \in \mathbb{R}^d$, where $d = 1000$, forming $\boldsymbol{\phi}(\mathbf{x})$ by concatenating polynomial terms to the second order would result in a dimension of $d + d(d+1)/2$ or roughly 500,000. Using higher-order polynomial terms would render $\boldsymbol{\phi}(\mathbf{x})$ impractical. However, it is still feasible to use such feature vectors implicitly. If our training and prediction problems can be framed solely in terms of inner products between the examples, then the computation we are interested in is $\left\langle \boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}') \right\rangle$. Depending on how $\boldsymbol{\phi}(\mathbf{x})$ is defined, this computation might be carried out efficiently even if it is not using $\boldsymbol{\phi}(\mathbf{x})$ explicitly. For example, when $\boldsymbol{\phi}(\mathbf{x}) = \left(x_1, x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2\right)^\top$, we see that

$$\left\langle \boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}') \right\rangle = \left(\mathbf{x} \cdot \mathbf{x}'\right) + \left(\mathbf{x} \cdot \mathbf{x}'\right)^2$$

So the inner product is easily calculated on the basis of the original input vectors without having to explicitly represent $\boldsymbol{\phi}(\mathbf{x})$. We will try to turn our linear classifiers into a form that relies only on the inner products between the examples.

8.2 Kernel Methods

We have discussed several linear prediction methods such as the perceptron algorithm, support vector machine, and linear regression (Ridge/Lasso/Elastic Net). As mentioned in the Introduction, all these methods can be transformed into non-linear methods through the feature vectors $\boldsymbol{\phi}(\mathbf{x})$.

$$\text{Preceptron: } \hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b)$$
$$\text{Linear Regression: } \hat{y} = \mathbf{w} \cdot \mathbf{x} + b$$

differ from each other based on how they are trained in response to (expanded) training examples. This means that the parameters for the models would be different, even if the data were the same (as we have seen with the logits, we can also use regression for binary labels).

8.2.1 Kernel perceptron

We can run the perceptron algorithm (until convergence) when the training examples are linearly separable in the given feature representation. Recall the stepwise training algorithm, Algorithm 2.

Based on this algorithm, we can re-write the parameters as

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y^{(i)} \boldsymbol{\phi}(\mathbf{x}^{(i)}) \tag{8.1}$$

$$b = \sum_{i=1}^{n} \alpha_i y^{(i)} \tag{8.2}$$

where α_i is the number of times that we have made a mistake on the corresponding training example $(\boldsymbol{\phi}(\mathbf{x}^{(i)}), y^{(i)})$. Our goal here is to rewrite the algorithm so that we simply update the α_i,never explicitly constructing $\mathbf{w}$ which may be high-dimensional. It is important to realize that the scalar product of weights and features is

$$\mathbf{w} \cdot \boldsymbol{\phi}(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i y^{(i)} \boldsymbol{\phi}(\mathbf{x}^{(i)}) \cdot \boldsymbol{\phi}(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i y^{(i)} k(\mathbf{x}^{(i)}, \mathbf{x}) \tag{8.3}$$

where the inner product $k(\mathbf{x}^{(i)}, \mathbf{x}) = \left\langle \boldsymbol{\phi}(\mathbf{x}^{(i)}), \boldsymbol{\phi}(\mathbf{x}) \right\rangle$ is known as the **kernel function**. It is a function of two arguments and is always defined as the inner product of feature vectors corresponding to the input arguments. Indeed, we say that a kernel function is valid if there is some feature vector $\boldsymbol{\phi}(\mathbf{x})$ (possibly infinite-dimensional!) such that $k(\mathbf{x}, \mathbf{x}')$ is defined and can be written as an inner product of some feature vectors for all $\mathbf{x}$ and $\mathbf{x}'$.

We want the perceptron algorithm to use only kernel values rather than feature vectors directly. To achieve this,we will write the discriminant function $\mathbf{w} \cdot \boldsymbol{\phi}(\mathbf{x}) + b$ solely in terms of the kernel function and α's

$$\mathbf{w} \cdot \boldsymbol{\phi}(\mathbf{x}) + b = \sum_{i=1}^{n} \alpha_i y^{(i)} \boldsymbol{\phi}(\mathbf{x}^{(i)}) + \sum_{i=1}^{n} \alpha_i y^{(i)} = \sum_{i=1}^{n} \alpha_i y^{(i)} \left[k(\mathbf{x}^{(i)}, \mathbf{x}) + 1 \right] \tag{8.4}$$

This is all that we need for prediction or for assessing whether there was a mistake ona particular training example $(\boldsymbol{\phi}(\mathbf{x}^{(i)}), y^{(i)})$. We can therefore write the algorithm only in terms of α's, updating them in response to each mistake. The resulting *kernel perceptron* algorithm is given by algortihm 7.

Note that the algorithm can be run with any valid kernel function $k(\mathbf{x}, \mathbf{x}')$. Typically, only a few of the counts α_i will be non-zero. This means that only a few of the training examples are relevant for finding a solution separating the examples, the rest of the counts α_i remain exactly at zero. So, just as with support vector machines, the solution can be quite sparse.

Algorithm 7 Kernel Perceptron Algorithm

procedure KERNELPERCEPTRON($\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1,\dots,n}, T$)
  $\boldsymbol{\alpha} = 0$ (vector of length n)
  for $t = 1, \dots, T$ **do**
    for $k = 1, \dots, n$ **do**
      if $y^{(k)} \sum_{i=1}^{n} \alpha_i y^{(i)} \left[k(\mathbf{x}^{(i)}, \mathbf{x}) + 1 \right] \leq 0$ **then**
        $\alpha_k \leftarrow \alpha_k + 1$
      end if
    end for
  end for
  return $\boldsymbol{\alpha}$
end procedure

8.2.2 Kernel ridge regression

For simplicity, let us again use the notation with $\boldsymbol{\theta} = \left(w_1, \dots, w_p, b\right)$, i.e., that the regression function is written as if it did not have a bias.

$$J_{n,\lambda}(\boldsymbol{\theta}) = \frac{1}{2} \sum_{i=1}^{n} \left(y^{(i)} - \boldsymbol{\theta} \cdot \boldsymbol{\phi}(\mathbf{x}^{(i)}) \right)^2 + \frac{\lambda}{2} ||\boldsymbol{\theta}||^2 \tag{8.5}$$

This is a convex function (bowl-shaped), and thus the minimum is obtained at the point, where the gradient is zero.

$$\nabla_\theta J_{n,\lambda}(\boldsymbol{\theta}) = -\sum_{i=1}^{n} \underbrace{\left(y^{(i)} - \boldsymbol{\theta} \cdot \boldsymbol{\phi}(\mathbf{x}^{(i)}) \right)}_{?} \boldsymbol{\phi}(\mathbf{x}^{(i)}) + \lambda \boldsymbol{\theta} \stackrel{!}{=} 0 \tag{8.6}$$

This gradient evaluates to zero where the expression with the underbrace is equal to $\lambda \alpha_i$.

$$\begin{aligned} \lambda \alpha_t &= y^{(t)} - \boldsymbol{\theta} \cdot \boldsymbol{\phi}(\mathbf{x}^{(t)}) \\ &= y^{(t)} - \sum_{i=1}^{n} \alpha_i \boldsymbol{\phi}(\mathbf{x}^{(i)}) \cdot \boldsymbol{\phi}(\mathbf{x}^{(t)}) \\ &= y^{(t)} - \sum_{i=1}^{n} \alpha_i k(\mathbf{x}^{(i)}, \mathbf{x}^{(t)}) \end{aligned} \tag{8.7}$$

where we have used $\boldsymbol{\theta} = \sum_{i=1}^{n} \alpha_i \boldsymbol{\phi}(\mathbf{x}^{(i)})$. The above has to hold for all $t = 1, \dots, n$. Let's now rewrite the sum in eq. (8.7) using vector notation

$$\sum_{i=1}^{n} k(\mathbf{x}^{(i)}, \mathbf{x}^{(t)}) \alpha_i = [\mathbf{K}\boldsymbol{\alpha}]_t \tag{8.8}$$

where $\boldsymbol{\alpha}$ is the vector of weights α_i and $\mathbf{K}$ is the $n \times n$ kernel or gram matrix with $K_{ij} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$. Now we can recast eq. (8.7) for all indices

$$\lambda \boldsymbol{\alpha} = \mathbf{y} - \mathbf{K}\boldsymbol{\alpha} \tag{8.9}$$

$$(\lambda \mathbf{I} + \mathbf{K})\, \boldsymbol{\alpha} = \mathbf{y} \tag{8.10}$$

where $\mathbf{I}$ is the identity matrix. The solution is simply

$$\boldsymbol{\alpha} = (\lambda \mathbf{I} + \mathbf{K})^{-1} \mathbf{y} \tag{8.11}$$

luckily the matrix is always invertible for $\lambda > 0$. In other words, estimated coefficients $\hat{\alpha}_t$ can be computed only in terms of the kernel function and target responses, without having to explicitly construct feature vectors $\boldsymbol{\phi}(\mathbf{x})$. Once we have the coefficients, prediction for a new point $\mathbf{x}$ is similarly easy

$$\hat{\boldsymbol{\theta}} \cdot \boldsymbol{\phi}(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i \boldsymbol{\phi}(\mathbf{x}^{(i)}) \cdot \boldsymbol{\phi}(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i k(\mathbf{x}^{(i)}, \mathbf{x}) \tag{8.12}$$

8.3 Kernel Functions

All of the methods discussed above can be run with any valid kernel function $k(\mathbf{x}, \mathbf{x}')$. A kernel function is valid if and only if there exists some feature mapping $\boldsymbol{\phi}(\mathbf{x})$ such that $k(\mathbf{x}, \mathbf{x}') = \left\langle \boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}') \right\rangle$. We don't necessarily need to know what$\boldsymbol{\phi}(\mathbf{x})$ is, only that one exists. We can build many common kernel functions based only on the following four rules:

1. $k(\mathbf{x}, \mathbf{x}') = 1$ is a valid kernel.
2. If $k(\mathbf{x}, \mathbf{x}')$ is a valid kernel function, then so is $\tilde{k}(\mathbf{x}, \mathbf{x}') = f(\mathbf{x}) k(\mathbf{x}, \mathbf{x}') f(\mathbf{x}')$ with $f : \mathbb{R}^d \to \mathbb{R}$.
3. If $k_1(\mathbf{x}, \mathbf{x}')$ and If $k_2(\mathbf{x}, \mathbf{x}')$ are valid kernels, then so is their sum.
4. If $k_1(\mathbf{x}, \mathbf{x}')$ and If $k_2(\mathbf{x}, \mathbf{x}')$ are valid kernels, then so is their product.

In addition, we know that the kernel must be symmetric $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$, as it represents an inner product.

8.3.1 Linear Kernel

The linear kernel is the most straightforward one. It is useful for linearly separable data. Using this kernel is identical to normal linear regression

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \mathbf{x}' \tag{8.13}$$

8.3.2 Polynomial Kernel

The polynomial kernel of degree d is the simplest, non-linear kernel

$$k(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\top \mathbf{x}' + c)^d \ , \tag{8.14}$$

where c is a constant. One can write down the features that are equivalent to this kernel, even if they are much longer vectors.

8.3.3 Gaussian (RBF) Kernel

The radial basis function or Gaussian kernel is defined as:

$$k(\mathbf{x}, \mathbf{x}') = e^{-\frac{||\mathbf{x}-\mathbf{x}'||_2^2}{2\sigma^2}} \tag{8.15}$$

σ is a parameter that defines the width of the Gaussian. When using kernels, people also use $\gamma = 1/2\sigma^2$. It is very general applicable, and thus useful when there is little prior knowledge about the system.

Using the rules above, one can realize that $k(\mathbf{x}, \mathbf{x}') = e^{\mathbf{x}\cdot\mathbf{x}'}$ is also a kernel. Since the expansion is an infinite sum, the resulting feature representation is infinite dimensional! This is also why the radial basis function (rbf) kernel has an infinite-dimensional feature representation. Specifically,

$$k(\mathbf{x}, \mathbf{x}') = e^{-\frac{||\mathbf{x}-\mathbf{x}'||_2^2}{22}} \tag{8.16}$$

$$= e^{\frac{||\mathbf{x}||_2^2}{2}}\ e^{\mathbf{x}\cdot\mathbf{x}'}\ e^{\frac{||\mathbf{x}'||_2^2}{2}}$$

$$= f(\mathbf{x})\ e^{\mathbf{x}\cdot\mathbf{x}'}\ f(\mathbf{x}') \tag{8.17}$$

where $f(\mathbf{x}) = e^{\frac{||\mathbf{x}||_2^2}{2}}$. The radial basis kernel is special in many ways. For example, any distinct set of training examples is always perfectly separable by the radial basis kernel. In other words, running the perceptron algorithm with the radial basis kernel will always converge to a separable solution, provided that the training examples are all distinct.

8.3.4 Laplacian Kernel

The Laplacian kernel is similar to the Gaussian one, but uses the L1 norm instead

$$k(\mathbf{x}, \mathbf{x}') = e^{-\frac{||\mathbf{x}-\mathbf{x}'||_1}{\sigma}} = e^{-\gamma||\mathbf{x}-\mathbf{x}'||_1} \tag{8.18}$$

8.3.5 Exponential Kernel

The exponential kernel is similar to the Laplace kernel, but uses the L2 norm like the Gaussian kernel

$$k(\mathbf{x}, \mathbf{x}') = e^{-\frac{||\mathbf{x}-\mathbf{x}'||_2}{\sigma}} \tag{8.19}$$

8.3.6 Matérn Kernel

The Matérn kernel is a popular choice in machine learning, particularly in Gaussian Processes, due to its flexibility in controlling the smoothness of the function it models. The Matérn kernel is defined as:

$$k(\mathbf{x}, \mathbf{y}') = \frac{2^{1-\nu}}{\Gamma(\nu)}\left(\sqrt{2\nu}\frac{||\mathbf{x}-\mathbf{x}'||_2}{\ell}\right)^{\nu} K_\nu\left(\sqrt{2\nu}\frac{||\mathbf{x}-\mathbf{x}'||_2}{\ell}\right) \tag{8.20}$$

ν controls the smoothness, ℓ is the length scale, K_ν is a modified Bessel function and $\Gamma(\nu)$ the Gamma function. For $\nu \to \infty$ the kernel converges to the Gaussian kernel (or RBF kernel), $\nu = 0.5$ is equivalent to the exponential kernel. Therefore, this kernel is very flexible as it can model a very smooth or very rough function (depending on the choice of ν). This gives it a better generalization ability than the RBF kernel; however, it needs one more hyperparameter to be adjusted.

8.3.7 Sigmoid Kernel

The sigmoid or hyperbolic tangent kernel

$$k(\mathbf{x}, \mathbf{x}') = \tanh(\alpha \mathbf{x}^\top \mathbf{x}' + c) \tag{8.21}$$

can be used to model neural network-like behaviour. It maps everything into the range $(-1, 1)$.

8.4 Support Vector Regression

Support Vector Regression (SVR) extends Support Vector Machine (SVM) to handle regression problems. Although SVMs are used mainly for classification purposes, SVR modifies this approach to forecast continuous outputs. As demonstrated later, SVR can employ the kernel trick, enabling it to model non-linear relationships. Consequently, the very first kernel method you encountered in this course was SVM classification. Using the kernel trick, SVMs can transcend straightforward linear classification.

For SVR, the key idea is to find a function that has a maximum deviation ϵ from the actual target values for all training data, while being as flat as possible.

8.4.1 Formulation

The function $f(\mathbf{x})$ in SVR is expressed as a simple linear function:

$$f(\mathbf{x}; \mathbf{w}, b) = \mathbf{w}^\top \mathbf{x} + b$$

where $\mathbf{w}$ and b are the parameters to be learned. The objective is to minimize the following cost function:

$$J_{n,C}(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{w}\|^2 + C \sum_{i=1}^{n} L_\epsilon(y_i, f(\mathbf{x}_i)) \tag{8.22}$$

Here, $L_\epsilon(y_i, f(\mathbf{x}_i))$ is the ϵ-insensitive loss function defined as:

$$L_\epsilon(y_i, f(\mathbf{x}_i)) = \begin{cases} 0 & \text{if } |y_i - f(\mathbf{x}_i)| \leq \epsilon \\ |y_i - f(\mathbf{x}_i)| - \epsilon & \text{otherwise} \end{cases} \tag{8.23}$$

The parameter $C > 0$ determines the trade-off between the flatness of $f(\mathbf{x})$ and the amount up to which deviations larger than ϵ are tolerated.

8.4.2 Dual Problem and Kernel Trick

To handle non-linear relationships, SVR can be formulated in its dual problem, which allows the use of kernel functions to map the input space into a higher-dimensional feature space. The dual problem of SVR is given by the objective function:

$$J_{n,C}(\boldsymbol{\alpha}, \boldsymbol{\alpha}^*) = \frac{1}{2} \sum_{i,j=1}^{n} (\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*) k(\mathbf{x}_i, \mathbf{x}_j) + \epsilon \sum_{i=1}^{n} (\alpha_i + \alpha_i^*) - \sum_{i=1}^{n} y_i (\alpha_i - \alpha_i^*) \tag{8.24}$$

subject to:

$$\sum_{i=1}^{n} (\alpha_i - \alpha_i^*) = 0 \quad \text{and} \quad 0 \leq \alpha_i, \alpha_i^* \leq C \tag{8.25}$$

Once the dual problem is solved, the function $f(\mathbf{x})$ is expressed as:

$$f(\mathbf{x}) = \sum_{i=1}^{n} (\alpha_i - \alpha_i^*) k(\mathbf{x}_i, \mathbf{x}) + b \tag{8.26}$$

where one could fuse the Lagrange multipliers $\alpha' = \alpha_i - \alpha_i^*$ making this more similar to other kernel expressions. We can drop any training sample for which $\alpha' = 0$. The remaining examples are called **support vectors**.

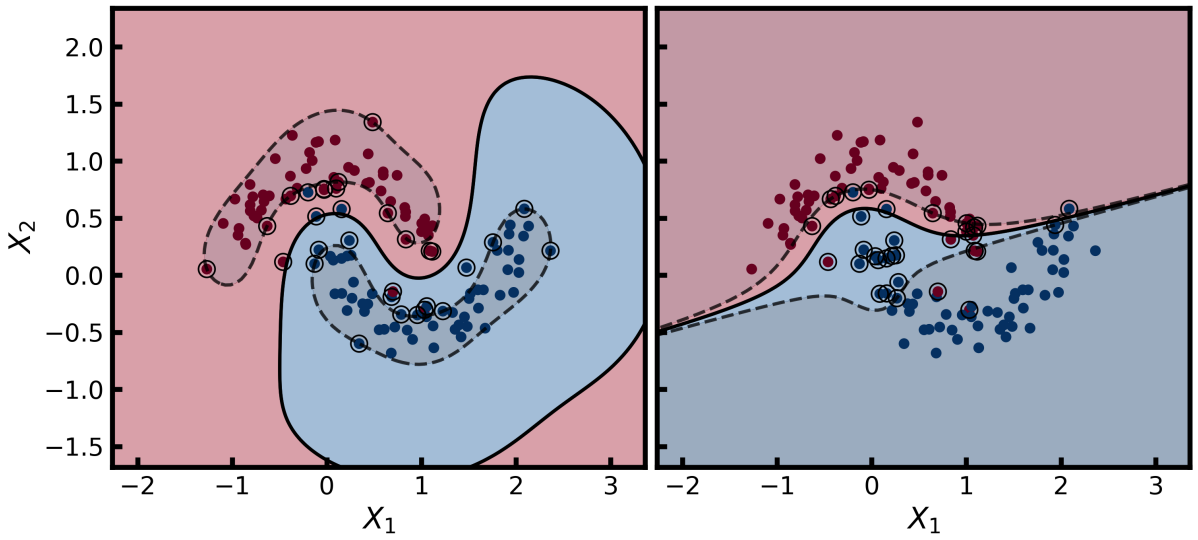

Figure 8.1: Support vector classification using two different kernels. The lightly shaded areas the within the margin (dashed lines) around the decision boundary (solid line). Circled points are the support vectors, the samples with non-zero Lagrange multipliers. Left: Infinitely dimensional radial basis function kernel. Right: Polynomial kernel of power 3.

8.4.3 Algorithm

Unlike the previous kernel regression methods, support vector regression can be solved by gradient descent. A general procedure is given in Algorithm 8. If we use gradient descent, that is, use the entire data set per update step, then T is the number of updates. When using stochastic gradient descent, then T is the number of epochs and each epoch contains an internal for loop over the smaller random batches. The projection of the parameters on the feasible region is done at every subupdate step. Once, can pre-compute the kernel matrix $\mathbf{K}$ to save time. However, this can be costly in terms of memory if the data set is large. One has to consider a balance of memory costs and time wasted on recomputing the kernel value.

8.4.4 Summary

Key Characteristics

SVR seeks to reduce the error within a tolerance margin ϵ, thus making it *resilient* to minor errors and noise in the training data set. Only data points outside the ϵ tube influence the final model,

Algorithm 8 Support Vector Regression Algorithm

procedure SVR($\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1,\ldots,n}, C, k(\mathbf{x}, \mathbf{x}'), \epsilon, \eta, T$)
  $\alpha_i = 0$ and α_i^* for $i = 1, \ldots, n$
  for $t = 1, \ldots, T$ **do**
    # Compute gradients (either over the entire data set or loop over batches):
    $\frac{\partial L}{\partial \alpha_i} = \sum_{j=1}^{n} (\alpha_j - \alpha_j^*)\mathbf{K}_{ij} - y_i + \epsilon$
    $\frac{\partial L}{\partial \alpha_i^*} = \sum_{j=1}^{n} (\alpha_j - \alpha_j^*)\mathbf{K}_{ij} - y_i - \epsilon$
    $\alpha_i \leftarrow \alpha_i - \eta \frac{\partial L}{\partial \alpha_i}$
    $\alpha_i^* \leftarrow \alpha_i^* - \eta \frac{\partial L}{\partial \alpha_i^*}$
    # Project α_i and α_i^* onto the feasible region ($0 \leq \alpha_i, \alpha_i^* \leq C$):
    $\alpha_i \leftarrow \max(0, \min(C, \alpha_i))$, $\alpha_i^* \leftarrow \max(0, \min(C, \alpha_i^*))$
  end for
  return $\boldsymbol{\alpha}$, $\boldsymbol{\alpha}^*$
end procedure

leading to a sparse solution. Similarly to other kernel methods, by utilizing various kernel functions, such as linear, polynomial, and radial basis functions (RBFs), SVR can capture complex nonlinear patterns.

Comparison with Kernel Ridge Regression

Whereas KRR utilizes a quadratic loss function, SVR leverages an ϵ-insensitive loss function which disregards errors within the ϵ margin. Both approaches incorporate a regularization term, with SVR offering greater flexibility in adjusting the balance between model simplicity and error tolerance via the ϵ parameter. SVR inherently produces a sparse model, since only the support vectors influence the decision function, in contrast to KRR, which generally employs all training data points. The training complexity for both methodologies scales as $O(n^3)$, although SVR may offer faster inference due to its sparsity.

9 Unsupervised Learning

Up until now, we considered supervised learning scenarios, where we have access to both examples and the corresponding target labels or responses: $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1,\dots,n}$. The goal was to learn a mapping from examples to labels that would work generalize well to unseen examples. In contrast, here we have only examples $S_n = \{(\mathbf{x}^{(i)})\}_{i=1,\dots,n}$. What is the learning task now? The goal of unsupervised learning is to uncover useful structure in the data S_n such as identify groups or clusters of similar examples. Another important application of unsupervised learning is dimensionality reduction, i.e., identify those dimensions of the feature space that are sufficient to distinguish the examples in S_n.

9.1 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a widely used technique for dimensionality reduction and feature extraction. It transforms the data into a new coordinate system such that the greatest variance by any projection of the data lies on the first principal component, the second greatest variance on the second principal component, and so on.

PCA is commonly used in exploratory data analysis, image compression, and especially relevant to this course as a preprocessing step for other machine learning algorithms to reduce the dimensionality of the data.

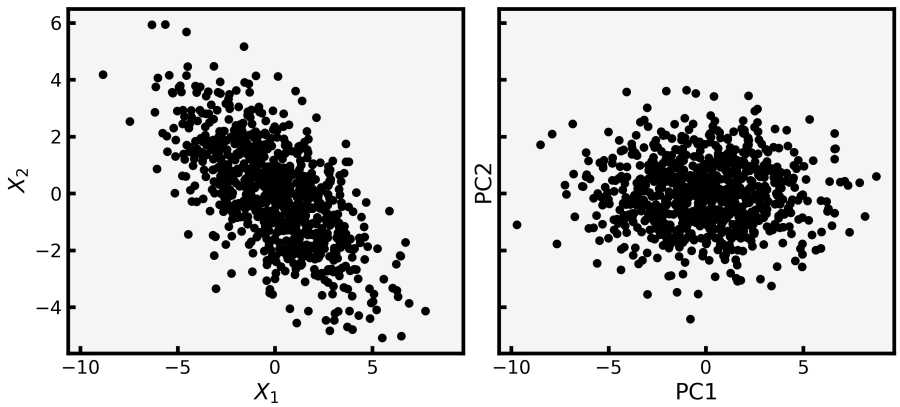

Figure 9.1: PCA performs a rotation of the data such that the new axes are sorted by variance. Note that PC1 is now the direction of the highest variance of the original data. PC2 is then simply the remaining orthogonal direction.

9.1.1 Algorithm

PCA effectively rotates your coordinate system such, that the new system consists of linear combinations of the old features such that they are sorted by maximal variance and orthogonal to each

other. The steps of the algorithm are:

1. Standardize the data: Ensure that each feature has zero mean and in case that the features have different units, also normalize the variance.

2. Compute the covariance matrix

$$\mathbf{\Sigma} = \frac{1}{n-1}\sum_{i=1}^{n}(\mathbf{x}^{(i)} - \bar{\mathbf{x}})(\mathbf{x}^{(i)} - \bar{\mathbf{x}})^\top = \frac{1}{n-1}\sum_{i=1}^{n}\mathbf{x}^{(i)}\,(\mathbf{x}^{(i)})^\top = \frac{1}{n-1}\mathbf{X}^\top\mathbf{X} \tag{9.1}$$

   where the first equality follows from the fact that the data is mean-free and $\mathbf{X}$ is the $n \times d$ matrix of all stacked examples. The prefactor $1/(n-1)$ is sometimes also given as simply $1/n$, both give yield the same eigenvalues and eigenvectors.

3. Compute the eigenvalues and eigenvectors of the covariance matrix.

4. Sort the eigenvalues and their corresponding eigenvectors in descending order.

5. To reduce the dimensionality: Select the top k eigenvectors to form the projection matrix $\mathbf{W}$ (where each column is an eigenvector) and transform the original data

$$\mathbf{Z} = \mathbf{X}\mathbf{W}$$

   where $\mathbf{Z}$ has dimension $n \times k$.

The top k are usually chosen to encompass a certain amount of the total (cumulative) variance in the data, typical choices are 80 % or 90%. In many applications this can reduce the number of features from thousands to a few dozen.

9.2 Kernel Principal Component Analysis (Kernel PCA)

Kernel Principal Component Analysis (Kernel PCA) is a non-linear extension of PCA that uses kernel methods to project the data into a higher-dimensional space where it is linearly separable.

Kernel PCA is used for non-linear dimensionality reduction, denoising, and feature extraction. It can capture complex structures in the data that linear PCA cannot.

9.2.1 Algorithm

The algorithm is similar to normal PCA, it just has to be updated with regards to the kernel function.

1. Choose a kernel function $k(\mathbf{x}, \mathbf{x}')$ to compute the kernel (Gram) matrix $\mathbf{K}$:

$$K_{ij} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$$

2. Center the kernel matrix:

$$\mathbf{K}' = \mathbf{K} - \mathbf{1}_n\mathbf{K} - \mathbf{K}\mathbf{1}_n + \mathbf{1}_n\mathbf{K}\mathbf{1}_n$$

   where $\mathbf{1}_n$ is an $n \times n$ matrix where each element is $1/n$.

3. Compute the eigenvalues and eigenvectors of the centered kernel matrix.

4. To reduce the features: Select the top k eigenvectors to form the projection matrix $\mathbf{W}$, then transform the kernel matrix as:

$$\mathbf{Z} = \mathbf{KW}$$

If one chooses the linear kernel $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \mathbf{x}'$ then the above algorithm is identical to the linear PCA introduced in the previous section.

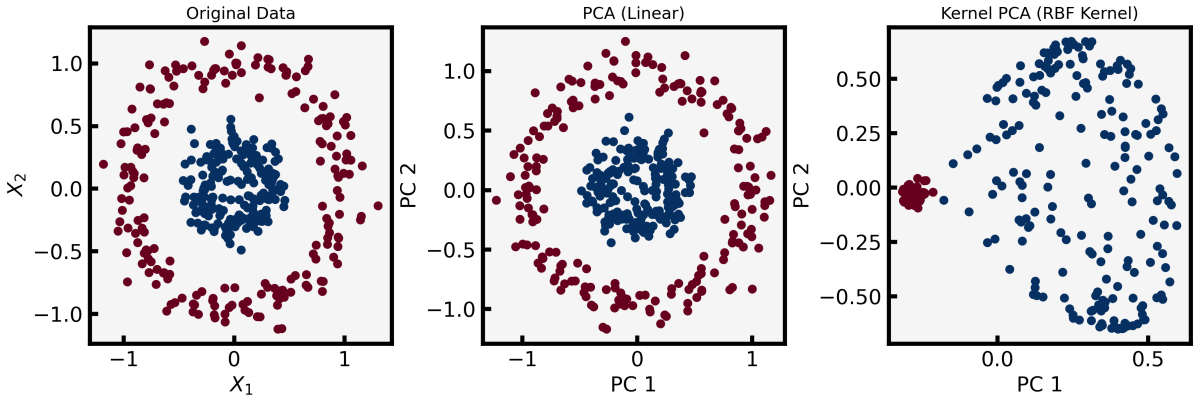

Figure 9.2: Comparison of (linear) PCA with kernel PCA. The normal PCA tries to rotate the data, but due to the symmetry cannot separate the two clusters much. The radial basis function kernel helps to separate the two circles such that they now could be distinguished with a simple, linear classifier.

9.3 Manifold Learning

Manifold learning aims to uncover low-dimensional structures embedded in high-dimensional data. Manifold learning techniques are used for data visualization, noise reduction, feature extraction, and as preprocessing steps for other machine learning algorithms. They are particularly effective for uncovering hidden structures in high-dimensional data and for reducing dimensionality while preserving important data characteristics.

There are many manifold learning techniques (e.g., Multi Dimensional Scaling, Isomap, Locally Linear Embedding and its variants, Spectral Embedding). We will only discuss one here, namely t-SNE, as it is frequently used as a preprocessing step.

9.3.1 t-Distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE is a non-linear dimensionality reduction technique that is particularly well-suited for embedding high-dimensional data for visualization in a low-dimensional space of **two or three** dimensions. To perform such a drastic reduction of dimensions (assuming that before the number of features was much higher than 3), it is recommendable to use t-SNE after having used linear PCA first.

Algorithm

1. Compute pairwise similarities between the high-dimensional points using Gaussian distributions:

$$p_{j|i} = \frac{\exp(-\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|_2^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|\mathbf{x}^{(i)} - \mathbf{x}^{(k)}\|_2^2 / 2\sigma_i^2)}$$

   and symmetrize:

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$$

2. Compute pairwise similarities in the low-dimensional space using Student's t-distribution:

$$q_{ij} = \frac{(1 + \|\mathbf{z}^{(i)} - \mathbf{z}^{(j)}\|_2^2)^{-1}}{\sum_{k \neq l} (1 + \|\mathbf{z}^{(k)} - \mathbf{z}^{(l)}\|_2^2)^{-1}}$$

3. Minimize the asymmetrical Kullback-Leibler divergence between the two distributions:

$$\mathrm{KL}(P||Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

   using gradient descent.

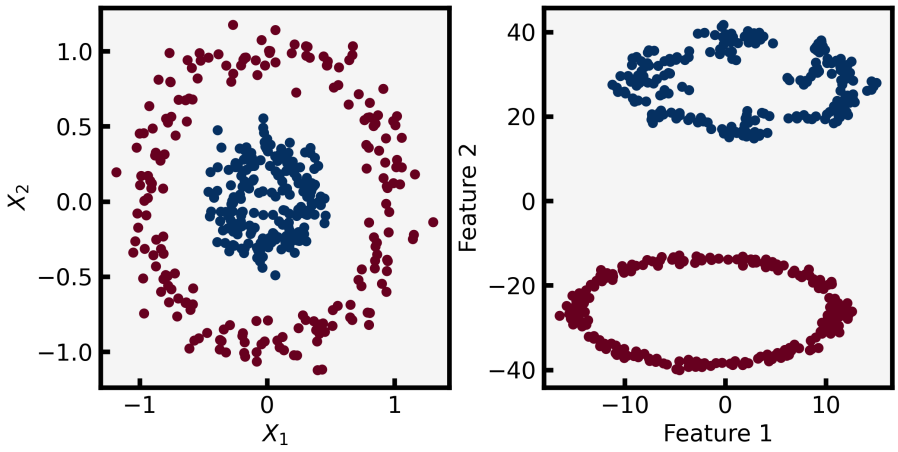

Figure 9.3: The transformation performed by t-SNE is strictly non-linear. After the transformation the groups of points are well separated by retain the spherical shape unlike after the Kenrel PCA in Fig. 9.2.

Strengths and Weaknesses

t-SNE transforms Gaussian probabilities from the original space into Student's t-distributions. This allows t-SNE to be particularly sensitive to local structure and brings a few advantages over existing techniques. It has a reduced tendency to maps all points together into a single heap and it can operate at different scales at the same time. While other methods focus on finding a single, continuous manifold, t-SNE tends to extract clustered local groups of samples. The ability to group samples

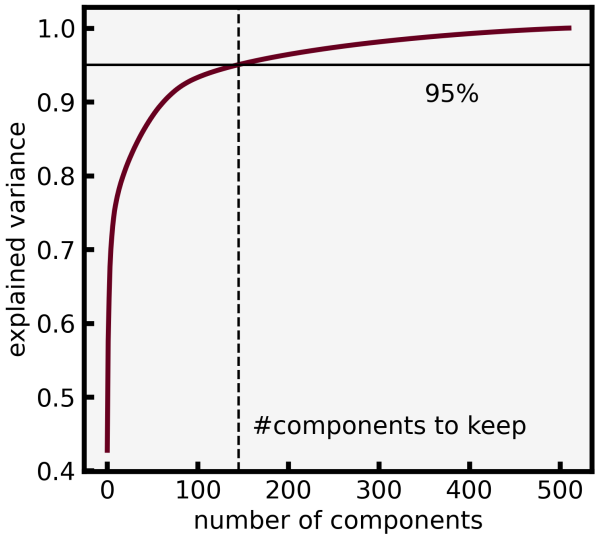

Figure 9.4: When performing a PCA with many dimensions, one can use the ratio of explained variance in each PC to determine how many of the top PCs to keep, as they include the most variance in the data set. A similar analysis can be performed with the eigenvalues of the kernel PCA.

based on the local structure can be beneficial to visually disentangle a data set, but might create dissimilarities where there are none, just because of missing data.

The KL-divergence of the joint probabilities is minimized by gradient descent, but the asymmetric KL-divergence is not convex, i.e. multiple restarts with different parameter initializations will end up in local minima. Thus, one might try different random seeds and select the embedding with the lowest KL divergence.

There are several problems with t-SNE. One is that it can be computationally expensive, and can take hours on datasets with millions of examples. As just mentioned, the algorithm is stochastic and different seeds can yield different embeddings, which can yield very arbitrary results, though it is in principle perfectly legitimate to pick the embedding with the least error. It is important to keep in mind that the global structure is not preserved, i.e., distances in the t-SNE embedding are not meaningful. t-SNE tends to move samples into clusters, but it might form some in non-structured data, meaning it creates clusters where there should not be any. One should therefore not over-interpret the size/density of clusters. Lastly, as t-SNE uses the Euclidean distance, it suffers from the curse of dimensionality in the sense that in high-dimensional spaces the Euclidean distance becomes meaningless (all pairs of points have very similar values). Hence, it will fail for very high-dimensional data.

9.4 Analyzing PCA Results and Deciding How Many Features to Keep

9.4.1 Introduction to Dimensionality Reduction

Dimensionality reduction is a key preprocessing step in machine learning, particularly when dealing with high-dimensional datasets. Reducing the number of features can improve the performance of a

subsequent model, and also speed up computations, and reduce storage requirements. However, it is not always obvious how many features to retain.

9.4.2 Principal Component Analysis (PCA)

PCA is a linear technique that projects data onto a new coordinate system where the axes, the *principal components*, are ordered by the amount of variance they capture. The goal of PCA is to reduce the dimensionality of the data while retaining as much variance (information) as possible.

Explained Variance and Cumulative Explained Variance

After applying PCA, each principal component explains a portion of the total variance in the data. The *explained variance* for each principal component quantifies this contribution, it is the ratio of the PC's eigenvalue and the sum of all eigenvalues. The *cumulative explained variance* is the sum of the explained variances up to a certain number of principal components, thus it is in a range of 0 to 1.

To determine how many components to retain, it is common to examine the cumulative explained variance. Typically, we aim to retain enough components to explain a large percentage of the variance (e.g., 95%). This ensures that the dimensionality is reduced while still preserving most of the original data's information. Other common thresholds are 99% if one wants to make sure to retain almost all information, and 90% or less, if the reduction of dimensions/noise is the main objective.

Scree Plot

Alternatively, than a specific numerical cutoff, the number of dimensions to keep can be determined visually (see Fig. 9.4). For this one plots the cumulative explained variance versus the number components kept, this is called a "scree plot". By analyzing the scree plot, you can choose the number of components before the explained variance begins to plateau (also known as the "elbow point"). Look for the point where the curve flattens out, which indicates that adding more components yields diminishing returns in terms of explained variance.

Another heuristic is to retain components with eigenvalues greater than 1, as they contribute more variance than an average original dimension.

9.4.3 Kernel PCA

Unlike standard PCA, Kernel PCA does not directly provide explained variance ratios because the transformation is nonlinear. However, you can still decide the number of components to keep using the following approaches:

Eigenvalue Magnitude:

The eigenvalues in Kernel PCA reflect the importance of each component. You can select components based on the magnitude of their corresponding eigenvalues. A common approach is to create a scree plot, where on displays the eigenvalues against the component number. Look for the "elbow point", where the eigenvalues start to drop off sharply.

Cross-Validation:

Another practical method is to use cross-validation to evaluate how the number of components affects the performance of your model on a downstream task (e.g., classification accuracy). Select the number of components that optimizes the model performance.

9.5 Clustering the Idea

Clustering is one of the key problems in exploratory data analysis. Examples of clustering applications include mining customer purchase patterns, modeling language families, or grouping search results according to topics. We have yet to specify any criterion for selecting clusters or the example representing the set of all members in a cluster. To this end, we must be able to compare pairs of points to determine whether they are indeed similar (should be in the same cluster) or not (should be in a different cluster). The comparison can be either in terms of similarity such as **cosine similarity** or dissimilarity as in **Euclidean distance**. Cosine similarity is simply the angle between two vectors (elements):

$$\cos(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \frac{\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}}{||\mathbf{x}^{(i)}||_2 ||\mathbf{x}^{(j)}||_2} = \frac{\sum_{\ell=1}^{d} x_\ell^{(i)} x_\ell^{(j)}}{\sqrt{\sum_{\ell=1}^{d} (x_\ell^{(i)})^2} \sqrt{\sum_{\ell=1}^{d} (x_\ell^{(j)})^2}} \tag{9.2}$$

Alternatively, we can focus on dissimilarity as in pairwise distance. Here, we will primarily use squared Euclidian distance

$$d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = ||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_2^2 = \sum_{\ell=1}^{d} \left(x_\ell^{(i)} - x_\ell^{(j)} \right)^2 \tag{9.3}$$

but there are many alternatives. For example, you may often encounter L_1 distance.

The choice of which distance metric to use is important as it will determine the type of clusters you will find. A reasonable metric or similarity is often easy to find based on the application. Another issue with the metric is that the available clustering algorithms such as k-means (discussed below) may rely on a particular metric. Once we have the distance metric, we can specify an objective function for clustering. In other words, we specify the cost of choosing any particular set of clusters or their representatives (a.k.a. centroids). The "optimal" clustering is then obtained by minimizing the cost, which is often cast in terms of distortion associated with individual clusters. For instance, the cost associated with cluster C could be the sum of pairwise distances within the points in C, or the diameter of the cluster (largest pairwise distance).

9.6 Gaussian Mixture Models (GMM)

Gaussian Mixture Models (GMM) are probabilistic models that assume the data is generated from a mixture of several Gaussian distributions with unknown parameters. Each component in the mixture model represents a cluster.

GMMs are used in clustering, density estimation, and as generative models for classification tasks.

9.6.1 Mathematical Formulation

A GMM can be defined as:

$$p(\mathbf{x}|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{9.4}$$

where π_k are the mixing coefficients, $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ are the mean (vector) and covariance (matrix) of the k-th Gaussian component, and $\mathcal{N}$ is the Gaussian distribution. Generally, one uses multi-variate Gaussians

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-d/2} \det(\boldsymbol{\Sigma})^{-1/2} e^{-\frac{(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})}{2}} \tag{9.5}$$

However, under the assumption (constraint) that $\boldsymbol{\Sigma}$ is diagonal, this simplifies to the product of one-dimensional Gaussians, one for each of the d dimensions.

9.6.2 Expectation-Maximization Algorithm

Algorithm 9 Gaussian Mixture Model

procedure GMM($\{\mathbf{x}^{(i)}\}_{i=1,\ldots,n}$, K)
  Initialize the parameters for the K Gaussians π_k, $\boldsymbol{\mu}_k$, and $\boldsymbol{\Sigma}_k$.
  repeat
    Expectation step (E-step):
    Compute the responsibilities (the probability that $\mathbf{x}^{(i)}$ belongs to Gaussian k)

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(\mathbf{x}^{(i)}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}^{(i)}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

    Maximization step (M-step):
    Update the parameters

$$\pi_k = \frac{1}{n} \sum_{i=1}^{n} \gamma_{ik}$$

$$\boldsymbol{\mu}_k = \frac{\sum_{i=1}^{n} \gamma_{ik} \mathbf{x}^{(i)}}{\sum_{i=1}^{n} \gamma_{ik}}$$

$$\boldsymbol{\Sigma}_k = \frac{\sum_{i=1}^{n} \gamma_{ik} (\mathbf{x}^{(i)} - \boldsymbol{\mu}_k)(\mathbf{x}^{(i)} - \boldsymbol{\mu}_k)^\top}{\sum_{i=1}^{n} \gamma_{ik}}$$

  until convergence
end procedure

9.7 k-Means Clustering

We will define the distortion here as the sum of (squared) distances from each point in the cluster to the corresponding cluster representative $\mathbf{z}$. For cluster C with centroid $\mathbf{z}$, the distortion is defined as $\sum_{i \in C} ||\mathbf{x}^{(i)} - \mathbf{z}||_2^2$. The cost of clustering $C_1, C2, \ldots, C_k$,is simply the sum of costs of individual

clusters:

$$\text{cost}(C_1, \ldots, C_k, \mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(k)}) = \sum_{j=1}^{k} \sum_{i \in C_j} ||\mathbf{x}^{(i)} - \mathbf{z}^{(j)}||_2^2 \tag{9.6}$$

Our goal is to find a clustering that minimizes this cost. Note that the cost here depends on both the clusters and how the representatives (centroids) are chosen for each cluster. It seems unnecessary to have to specify both clusters and centroids and, indeed, one usually implies the other. Note also that we only consider valid clusterings, i.e., each point must belong to one and only one cluster.

For some cost functions these two representations (cluster sets or representative example) are interchangeable: knowing the representatives, we can compute the corresponding clusters and vice versa. In fact, this statement holds for the cost function introduced above. We can define clusters by their representatives:

$$C_j = \{\mathbf{x}^{(i)} \mid \mathbf{z}^{(j)} = \arg \min_{\ell=1,\ldots,k} d(\mathbf{x}^{(i)}, \mathbf{z}^{(\ell)})\}$$

These clusters define an optimal clustering with respect to our cost function for a fixed setting of the representatives $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(k)}$. In other words,

$$\text{cost}(\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(k)}) = \sum_{i=1}^{n} \min_{j=1,\ldots,k} ||\mathbf{x}^{(i)} - \mathbf{z}^{(j)}||_2^2 \tag{9.7}$$

where in the last expression we are simply assigning each point to its closest representative (as we should). Geometrically, the partition induced by the centroids can be visualized as a Voronoi partition of $\mathbb{R}^d$.

The solution for this partitioning can be found iteratively. The new centroid is simply the geometrical average of current cluster

$$\mathbf{z}'^{(j)} = \frac{1}{|C_j|} \sum_{i \in C_j} \mathbf{x}^{(i)} \tag{9.8}$$

9.7.1 Algorithm

The k-means algorithm aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean.

Algorithm 10 k-Means Clustering

1: **procedure** k-MEANS($\{\mathbf{x}^{(i)}\}_{i=1,\ldots,n}$, k)
2: Initialize k cluster centroids randomly.
3: **repeat**
4: Assign each data point to the nearest centroid.
5: Recalculate the centroids as the mean of the data points assigned to each cluster and the cost.
6: **until** convergence (no change in cost)
7: **end procedure**

9.7.2 Initialization

Initialization can significantly affect the performance of k-means. Common initialization methods include:

- Random Initialization: Randomly choose k data points as initial centroids.
- K-means++: An improved method that spreads out the initial cluster centers.

9.7.3 Convergence

k-means is guaranteed to converge, but it may only reach a local optimum. The time complexity is $O(n \cdot k \cdot d \cdot t)$, where n is the number of data points, k is the number of clusters, d is the number of dimensions, and t is the number of iterations.

9.7.4 Improvement: k-Means++

k-means++ improves the initialization of centroids in k-means to avoid poor clusterings.

Algorithm 11 k-Means++ Initialization

1: Choose one center uniformly at random from the data points.
2: **repeat**
3: For each data point x, compute its distance to the nearest center already chosen.
4: Choose a new data point as a center with a probability proportional to the square of the distance from the nearest center.
5: **until** k centers have been chosen

k-means++ provides a way to choose initial cluster centers that are more likely to lead to better results, thereby improving the convergence of the clustering algorithm.

9.8 k-Medoids Clustering

We had previously defined the cost function for the k-means algorithm in terms of squared Euclidean distance of each point to the closest cluster representative. For any given cluster, the best representative to choose is the mean of the points in thecluster. The resulting cluster mean typically does not correspond to any point in the original dataset. The k-medoids algorithm operates exactly like k-means, but instead of choosing the cluster mean as a representative, it chooses one of the original points as a representative, now called an exemplar. Selecting exemplars rather than cluster means as representatives can be important in applications. Take, for example, Google News, where a single article is used to represent a news cluster. Blending articles together to evaluate the "mean" would not make sense in this context. Another advantage of k-medoids is that we can easily use other distance measures, other than the squared Euclidian distance.

9.9 Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

DBSCAN is a density-based clustering algorithm that can find arbitrarily shaped clusters and handle noise. In this algorithm clusters are areas of high density separated by areas of low density. Due to this general approach, clusters found by DBSCAN can be any shape, as opposed to k-means which assumes that clusters are convex shaped (spherical). In density-based clustering algorithms one

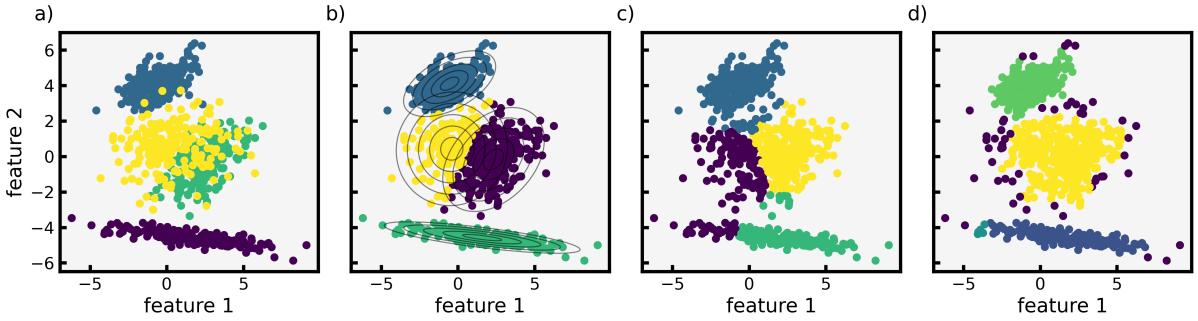

Figure 9.5: a) Synthetic, labeled data, sampled from 4 multivariate Gaussians. Only the number of unique labels is passed on. b) GMM clustering with 4 Gaussians. The Gaussian prior is important here. Elliptical lines indicate the positioning of the fitted Gaussians. c) k-means clustering with 4 clusters. d) DBSCAN result, it finds 4 clusters and identifies many points as noise (dark purple).

does not chose the number of clusters, instead one has to select a density cutoff, to determine cluster membership.

9.9.1 Algorithm

The central component is the concept of core samples, which are those in areas of high density. Hence, a cluster is a set of core samples, each close to each other (measured by some distance measure) and a set of non-core samples that are close to a core sample (but are not themselves core samples). There are two parameters to the algorithm, the minimum number of samples N_{min} and the cutoff ϵ.

A core sample is defined as a sample in the dataset such that there exist at least N_{min} other samples within a distance of ϵ, which are defined as neighbors of the core sample. This means that the core sample is in a dense area. A cluster is then a set of core samples that can be built by recursively taking a core sample, finding all of its neighbors that are core samples, finding all of their neighbors that are core samples, and so on. A cluster also has a set of non-core samples, which are samples that are neighbors of a core sample in the cluster but are not themselves core samples. Intuitively, these samples are on the fringes of a cluster.

Any core sample is part of a cluster, by definition. In contrast, any sample that is not a core sample, and is ϵ or further away from any core sample, is considered an outlier by the algorithm.

While N_{min} primarily controls the algorithm's tolerance towards noise (on noisy and large data sets it may be desirable to increase this parameter), the parameter ϵ is absolutely crucial to choose appropriately for the specific distance function and data set, it usually cannot be left at the default value. ϵ controls the local neighborhood of the points. When chosen too small, most data will be labeled as noise. When chosen too large, it causes close clusters to be merged into one cluster, and eventually the entire data set to be returned as a single cluster. Some heuristics for choosing this parameter have been discussed in the literature, for example based on a knee in the nearest neighbor distances plot similar to the elbow-method for k-means.

Algorithm 12 DBSCAN

for each data point p **do**
    Find all points within distance ϵ of p (neighbors).
    if the number of neighbors is greater than or equal to N_{min} **then**
        p is a core point.
        Form a cluster with p and its neighbors.
        Recursively find all density-connected points and add them to the cluster.
    else

    end if
end for
Any point marked as noise that is found to be within ϵ of a core point is added to that core point's cluster.

9.9.2 Comparison with k-Means

The clear advantage of density based clustering is that it can find clusters of arbitrary shape. The user also does not require the number of clusters to be specified. And lastly, it can handle noise and outliers and does not have to assign them to any cluster.

But just as with k-means or -mediods, the choice of the hyperparameters has a very large impact on the clustering result. Especially the choice of ϵ can be difficult and requires domain knowledge. Further, DBSCAN can struggle with clusters of varying densities. The extension, HDBSCAN alleviates the assumption of homogeneous densities and explores all possible density scales by building an alternative representation of the clustering problem.

Just like the previously discussed clustering algorithms, DBSCAN does not have a fixed number of iterations. It runs in $O(n \log n)$ time, but can be $O(n^2)$ in the worst case. It always converges, because each point is visited a finite number of times.

9.10 Heuristics for Analyzing Clustering Results

After applying a clustering algorithm, it is essential to evaluate the quality of the clustering results. Since clustering is unsupervised, we often rely on internal and external validation measures to assess how well the clustering has performed.

9.10.1 Silhouette Score

The **Silhouette Score** is an internal validation metric used to measure the quality of a clustering. It quantifies how well each point is clustered, taking into account both *cohesion* (how close a point is to other points in the same cluster) and *separation* (how far a point is from points in other clusters).

For each data point i, the Silhouette coefficient $s(i)$ is defined as:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \tag{9.9}$$

where:

- $a(i)$ is the average distance between point i and all other points in the same cluster (intra-cluster distance).

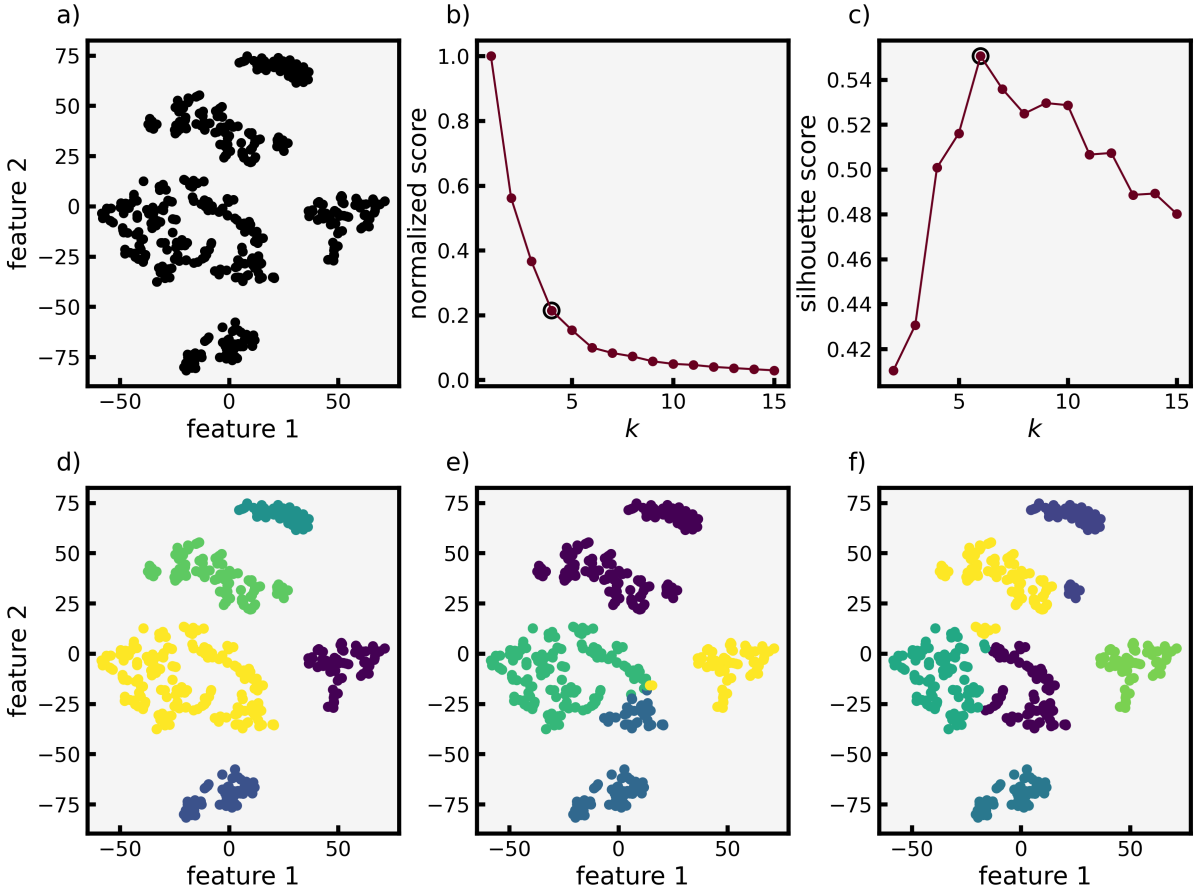

Figure 9.6: a) unlabeled group of points. Visually one might tend to guess 4 clusters, but it could be that sub-clusters are important. b) normalized WCSS for varying k. Kneedle method determind the optimum to be $k = 4$. c) silhouette score for verying number of clusters. Maximum silhouette score suggests $k = 6$. d) DBSCAN clustering of the data. e) k-means clustering with 4 clusters. f) k-means clustering with 6 clusters.

- $b(i)$ is the minimum average distance from point i to all points in the nearest different cluster (inter-cluster distance).

The Silhouette Score ranges from -1 to 1:

- $s(i) \approx 1$: The point is well-clustered, with its cluster being dense and well-separated from others.
- $s(i) \approx 0$: The point is on or near the boundary between two clusters.
- $s(i) < 0$: The point may have been assigned to the wrong cluster.

The overall Silhouette Score for the clustering is the mean of the Silhouette coefficients of all points. A higher Silhouette Score indicates better-defined clusters.

9.10.2 Normalized Mutual Information (NMI)

The **Normalized Mutual Information (NMI)** is an external validation metric that measures the similarity between two clustering results. It is particularly useful when you have ground truth labels or another clustering result to compare against.

NMI quantifies the amount of information shared between the predicted clustering U and the true clustering V. It is given by:

$$\text{NMI}(U,V) = \frac{2 \cdot I(U,V)}{H(U) + H(V)} \tag{9.10}$$

where:

- $I(U,V)$ is the mutual information between clusterings U and V.
- $H(U)$ and $H(V)$ are the entropies of the clusterings U and V, respectively.

NMI ranges from 0 to 1:

- NMI $= 1$ indicates perfect agreement between the two clusterings.
- NMI $= 0$ indicates no mutual information, meaning the clusterings are independent.

9.10.3 Other Heuristics for Clustering Analysis

- **Adjusted Rand Index (ARI):** Measures the similarity between two clusterings, adjusted for chance. ARI ranges from -1 to 1, with 1 indicating perfect agreement and 0 indicating random clustering.
- **Davies-Bouldin Index (DBI):** Evaluates the average similarity ratio of each cluster with its most similar cluster. Lower DBI values indicate better clustering.
- **Calinski-Harabasz Index (CH Index):** Also known as the Variance Ratio Criterion, it evaluates the ratio of the sum of between-cluster dispersion to within-cluster dispersion. A higher CH Index indicates better clustering.

9.11 Choosing k

The selection of k greatly impacts the quality of your k-means clustering solution. In some applications, the desired k is intuitively clear based on the problem definition. For instance, if we have prior knowledge of how many clusters exist or how many we need when tasked to divide things into a fixed number of groups.. In most problems, however, the optimal k is not given and we have to select it automatically.

Let's start by understanding the connection between the number of clusters k and the cost function (the within-cluster sum of squares). If every point belongs to its own cluster, then the cost is equal to zero. At the other extreme, if all the points belong to a single cluster with a single center $\mathbf{z}$, then the cost is the maximum possible value. It is informative to plot the cost as a function of k. On can often observe a sharp decrease in cost for the the first few, small values of k. While the decrease continues for larger k, it levels off. This observation motivates one of the commonly used heuristics for selecting k, called the **elbow method**. This method suggests that we should choose the value of

k that results in the highest relative drop in the cost (which corresponds to an "elbow" in the graph capturing as a function of k), see Fig. 9.6 for a comparison of using the WCSS and the silhouette score. Instead of just visually, one can use the “kneedle”-method to determine the elbow. However, it may be hard to identify (or justify) such a sharp point, as in many real-life scenarios the cost-vs-k graph decreases gradually.

10 Non-Linear Models II: More Methods

10.1 K-Nearest Neighbor (KNN) Regression and Classification

K-Nearest Neighbor (KNN) is a versatile, non-parametric, and instance-based learning algorithm widely used for both regression and classification tasks. KNN works by identifying the k training samples closest in distance to a new input, and then predicting the output based on the values of these neighbors. It is particularly effective for non-linear problems as it does not assume any underlying data distribution, hence non-parametric.

KNN methods are simple yet powerful methods for solving non-linear problems. Their ability to adapt to the structure of the data without assuming any specific model form makes them highly flexible. However, they can be computationally intensive, especially with large datasets, and their performance depends on the choice of k and the employed distance metric.

10.1.1 KNN Regression

KNN regression predicts the target value for a new data point based on the average of the target values of its k-nearest neighbors. This method can capture non-linear relationships in the data, because it relies on local patterns rather than a global model. It relies on the assumption that the nearest k-neighbors are close enough so that an interpolation is sensible.

Algorithm

Given the training set $S_n = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$, the KNN regression algorithm can be summarized as follows:

Algorithm 13 KNN Regression Algorithm

Require: Training set $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$, number of neighbors k, distance metric $d(\mathbf{x}, \mathbf{z})$
Ensure: Predicted value $\hat{y}$ for a new input $\mathbf{x}$

1: Compute distances $d(\mathbf{x}, \mathbf{x}_i)$ for all $i = 1, 2, \ldots, n$
2: Select the k nearest neighbors $\mathcal{N}_k(\mathbf{x})$ with the smallest distances
3: Compute the predicted value $\hat{y}$ as the average of the target values of the k nearest neighbors:

$$\hat{y} = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x})} y^{(i)}$$

10.1.2 KNN Classification

KNN classification assigns a class to a new data point based on the majority class among its k-nearest neighbors. This method is particularly useful for capturing complex decision boundaries in non-linear classification problems.

Algorithm

Given training set S_n, where $\mathbf{x}^{(i)} \in \mathbb{R}^d$ are the feature vectors and $y^{(i)} \in \{1, 2, \ldots, C\}$ are the class labels, the KNN classification algorithm can be summarized as follows:

Algorithm 14 KNN Classification Algorithm

Require: Training set $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$, number of neighbors k, distance metric $d(\mathbf{x}, \mathbf{z})$
Ensure: Predicted class $\hat{y}$ for a new input $\mathbf{x}$

1: Compute distances $d(\mathbf{x}, \mathbf{x}^{(i)})$ for all $i = 1, 2, \ldots, n$
2: Select the k nearest neighbors $\mathcal{N}_k(\mathbf{x})$ with the smallest distances
3: Determine the class $\hat{y}$ by majority vote among the k nearest neighbors:

$$\hat{y} = \arg\max_c \sum_{i \in \mathcal{N}_k(\mathbf{x})} \mathbb{I}(y^{(i)} = c)$$

where $\mathbb{I}$ is the indicator function.

Example

Again, consider a dataset with input-output pairs and a new input $\mathbf{x}$. If $k = 3$ and the three nearest neighbors have class labels $y_1 = 1$, $y_2 = 2$, and $y_3 = 1$, the predicted class is:

$$\hat{y} = 1$$

since class 1 appears more frequently among the nearest neighbors.

10.1.3 Distance Metrics

The choice of distance metric $d(\mathbf{x}, \mathbf{z})$ is crucial for the performance of KNN algorithms. The general metric is the **Minkowski Distance**, which you know as the L^p norm.

$$d(\mathbf{x}, \mathbf{z}) = \left(\sum_{j=1}^d |x_j - z_j|^p \right)^{\frac{1}{p}}$$

The most common choices are $p = 2$ (Euclidean distance) and $p = 1$ (Manhatten distance). However, in principle any metric could be used to determine the distance between two points.

10.1.4 Choosing the Number of Neighbors

The parameter k (number of neighbors) significantly impacts the KNN algorithm's performance. A small k can make the model sensitive to noise, while a large k may smooth out the decision boundaries too much, potentially leading to underfitting. Cross-validation is commonly used to select an optimal value for k.

10.1.5 Algorithm to Determine the k Nearest Neighbors

Unfortunately, there is no special algorithm to find the nearest neighbors. For a given input $\mathbf{x}$ one has to calculate the distance between $\mathbf{x}$ and every point in the training set, sort these distances, and select the k smallest distances. Hence, with increasing data set size this algorithm will become slower.

10.2 Decision Trees

Decision Trees are fundamental building blocks in many machine learning algorithms, including Random Forest and XGBoost. They are intuitive, easy to interpret, and powerful for both classification and regression tasks. A Decision Tree models a sequence of decisions and their possible consequences.

10.2.1 Structure of a Decision Tree

A Decision Tree is composed of nodes, branches, and leaves:

- **Root Node**: The top node representing the entire dataset. It is the starting point for the decision-making process.
- **Internal Nodes**: Nodes that represent decisions based on feature values. Each internal node splits the data into subsets based on a specific feature and a threshold or criterion.
- **Branches**: The connections between nodes, representing the outcome of a decision. Each branch corresponds to one of the possible values of the feature used at the internal node.
- **Leaf Nodes**: Terminal nodes that represent the final outcome, either a class label in classification or a continuous value in regression.

10.2.2 Decision Tree Algorithm

The construction of a Decision Tree involves recursively splitting the data based on the feature that results in the most significant reduction in impurity (for classification) or variance (for regression).

Initialize the root node with the entire dataset. Then, choose a feature and a threshold that best splits the data at the current node. Split the data into subsets based on the chosen feature and threshold. Create child nodes for each subset. For each child node, make the node a leaf, if the subset is pure (all instances have the same class or value) or a stopping criterion is met (e.g., maximum depth). Otherwise, keep repeating the process recursively for the subset at this child node.

Splitting Criteria

The choice of the feature and threshold for splitting is critical. Common criteria are:

- **Gini Impurity**:

$$\text{Gini}(D) = 1 - \sum_{c=1}^{C} p_c^2$$

  where p_c is the proportion of class c in dataset D, and C is the total number of classes.

- **Entropy (Information Gain)**:

$$\text{Entropy}(D) = -\sum_{c=1}^{C} p_c \log_2(p_c)$$

  The information gain from a split is calculated as the difference in entropy before and after the split.

- **Variance Reduction** (for regression):

$$\text{Variance}(D) = \frac{1}{|D|} \sum_{i=1}^{|D|} (y_i - \bar{y})^2$$

  where $\bar{y}$ is the mean of the target values in dataset D, and $|D|$ is the number of instances in D.

Stopping Criteria

The recursive splitting continues until one of the following stopping criteria is met:

- Maximum tree depth is reached.
- Minimum number of samples required to split a node is not met.
- All instances at a node belong to the same class (pure node).
- The reduction in impurity or variance is below a certain threshold.

10.2.3 Advantages and Disadvantages

Advantages

Decision Trees are easy to interpret and visualize, making them useful for explaining decisions to non-experts. They can capture non-linear relationships between features and the target variable without requiring feature transformation, as the decision rules do not have to be linear. Decision Trees make no assumptions about the distribution of the data, making them flexible for various types of data.

Disadvantages

Decision Trees can easily overfit the training data, especially when the tree is deep and complex. It is extremely heard to compare trees, as small changes in the data can lead to different splits, resulting in a completely new tree. When classes are imbalanced, Decision Trees may be biased towards the majority/dominant class.

10.2.4 Practical Considerations

Pruning

Pruning is a technique used to reduce the size of the tree by removing nodes that provide little power in classifying instances. This helps to combat overfitting and improves the tree's generalization to new data.

Feature Importance

Decision Trees provide a measure of feature importance by analyzing the reduction in impurity each feature provides across all the splits in the tree. Features that contribute to many high-quality splits are considered more important.

10.3 Random Forest

Random Forest is an ensemble learning method that builds upon the strengths of Decision Trees while mitigating their weaknesses, particularly the tendency to overfit. It combines multiple Decision Trees to create a "forest" of trees, hence the name. Each tree in the forest is trained on a different subset of the data, and the final prediction is made by aggregating the predictions of all the trees. This method leverages the power of multiple models to improve accuracy and robustness.

10.3.1 Overview of Random Forest

Random Forest operates by constructing a multitude of Decision Trees during training time and outputting the mode of the classes (for classification) or the mean prediction (for regression) of the individual trees. It is a type of ensemble method known as **bagging** (Bootstrap Aggregating).

10.3.2 Algorithm

The Random Forest algorithm can be broken down into the following steps:

Algorithm 15 Random Forest Algorithm

Require: Training set $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$, number of trees T, number of features m to consider at each split, and a base Decision Tree algorithm

1: **for** each tree $t = 1, 2, \ldots, T$ **do**

2: Draw a bootstrap sample D_t by randomly sampling n instances with replacement from the training set.

3: Grow a Decision Tree on D_t by recursively repeating the following steps for each node:

1. Randomly select m features from the total d features.
2. Choose the best feature and threshold for the split based on these m features.
3. Split the node into two child nodes based on the selected feature and threshold.

4: The tree is grown to its full depth (or until other stopping criteria are met) without pruning.

5: **end for**

6: For a new input $\mathbf{x}$, aggregate the predictions from all T trees:

- **For classification**: Predict the class by majority vote:

$$\hat{y} = \arg\max_{c} \sum_{t=1}^{T} \mathbb{I}(y_t = c)$$

- **For regression**: Predict the output by averaging the predictions:

$$\hat{y} = \frac{1}{T} \sum_{t=1}^{T} \hat{y}_t$$

10.3.3 Key Concepts in Random Forest

Bagging

Bagging, short for Bootstrap Aggregating, is the primary mechanism behind Random Forest. By creating multiple subsets of the original data set through random sampling with replacement, each Decision Tree is trained on a slightly different data set. This introduces diversity among the trees, which helps reduce variance and prevent overfitting.

Feature Randomness

In addition to bagging, Random Forest introduces an additional layer of randomness by selecting a random subset of features at each split in the tree. This ensures that the trees in the forest are less correlated with each other, further reducing variance.

Out-of-Bag Error

Since each tree in the Random Forest is trained on a bootstrap sample, approximately one-third of the data is not used in any given bootstrap sample. These unused data points are called **Out-of-Bag (OOB)** samples. The OOB samples can be used to estimate the model's prediction error without the need for a separate validation set. The OOB error is calculated as the average error for each observation using only the trees that did not use that observation in their bootstrap sample.

10.3.4 Advantages of Random Forest

Random Forest offers several advantages over a single Decision Tree:

- **Reduced Overfitting**: By averaging the predictions of multiple trees, Random Forest reduces the risk of overfitting, which is a common problem with individual Decision Trees.
- **Improved Accuracy**: The ensemble of multiple trees typically yields better predictive performance than a single tree, as it captures more complex patterns in the data.
- **Robustness to Noise**: The randomness in both the data samples and the feature selection process makes Random Forest robust to noise and less sensitive to overfitting.
- **Feature Importance**: Random Forest provides a natural way to assess the importance of each feature by measuring how much each feature decreases the impurity in the trees.
- **Handles Large Datasets**: Random Forest is scalable and can handle large datasets with high dimensionality.

10.3.5 Disadvantages of Random Forest

Despite its many advantages, Random Forest has some limitations:

- **Less Interpretability**: While Decision Trees are easily interpretable, the aggregation of many trees in a Random Forest makes the model more complex and harder to interpret.
- **Computationally Intensive**: Training a large number of trees can be computationally expensive and time-consuming, especially with large datasets.

- **Memory Usage**: Random Forest requires storing multiple trees, which can lead to high memory usage.

10.3.6 Comparison with a Single Decision Tree

The primary difference between Random Forest and a single Decision Tree lies in the ensemble nature of Random Forest. While a single Decision Tree builds a model based on one set of rules derived from the data, Random Forest creates multiple models and aggregates their predictions to make a final decision. This aggregation typically results in better generalization to new data, as it reduces the variance and the likelihood of overfitting that a single Decision Tree might exhibit. An important difference is also that in a single Decision Tree, all features are considered for each split. In contrast, a Random Forest randomly selects a subset of features at each split, which can lead to better generalization and information about the importance of each feature.

10.3.7 Practical Considerations

Hyperparameters

Random Forest has several hyperparameters that can be tuned to optimize performance: i) the **Number of Trees** (increasing the number of trees generally improves performance but also increases computational cost), ii) the **Number of Features** (smaller values increase the randomness and can lead to better generalization), and iii) the **Maximum Depth of Trees** (limiting the depth of each tree can help prevent overfitting, though this may reduce the model's accuracy).

Feature Importance

One of the valuable outputs of a Random Forest model is the feature importance measure, which indicates how much each feature contributes to the model's predictive power. This is typically computed by looking at the average decrease in impurity across all trees for each feature.

10.4 XGBoost

XGBoost, which stands for **eXtreme Gradient Boosting**, is a highly efficient and scalable implementation of gradient boosting, a powerful ensemble method for both classification and regression tasks. XGBoost has gained popularity for its performance and flexibility, often being a top choice in data science competitions and practical machine learning applications.

10.4.1 Boosting

The first ensemble method that we covered was *Bagging*, which involves training multiple models independently on various data subsets created by sampling with replacement. The purpose of bagging is to generate diversity among the ensemble models. The next ensemble method that we will explore is **Boosting**. This approach forms the foundation of several highly effective, leading ensemble techniques in machine learning, including Adaboost and Gradient Boosted Trees.

In order to understand boosting, we will first examine the cornerstone of boosting algorithms, weak learners.

Weak Learners

Basically all models discussed in this course so far are *Strong Learners*, those are models with the goal of performing as well as possible on the given regression or classification task. In contrast, weak learners are simple models that only perform slightly better than random chance. Boosting algorithms start with a single weak learner. The architecture of choice is usually a tree, but theoretically any model could be employed here.

Steps of Boosting

Boosting works as follows:

1. Train a single weak learner.
2. Determine which examples the weak learner got wrong.
3. Build another weak learner that focuses on the areas the first weak learner did not fit well.
4. Repeat this process until a predetermined stopping condition is met, e.g., until a number of weak learners have been created, or the performance has plateaued.

Thus, every new weak learner is tailored to address the shortcomings of the earlier weak learners. The higher the frequency of a missed example, the higher the probability that the subsequent weak learner will correctly fit that example. Therefore, all weak learners collectively contribute to forming one robust learner.

10.4.2 Gradient Boosting

Gradient boosting is a machine learning method that operates in a functional space, focusing on pseudo-residuals instead of the usual residuals (difference between prediction and label) seen in traditional boosting. This technique produces a prediction model composed of an ensemble of weak learners models, generally simple decision trees that assume little about the data.

Assuming we have a set of training samples $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$ and a differentiable loss function $L(y, F(\mathbf{x}))$, then the algorithm to create a gradient boosting model F over T rounds is:

Algorithm 16 Gradient Boosting

Require: Training set $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$, number of boosting rounds T

1: Initialize the model with a constant prediction:

$$F_0 = \arg\min_{y} \sum_{i=1}^{n} L(y^{(i)}, y)$$

where $L(y_i, \hat{y})$ is the loss function (e.g., squared error for regression, log loss for classification).

2: **for** each boosting round $t = 1, 2, \ldots, T$ **do**

3: Compute the so-called pseudo-residuals:

$$r_t^{(i)} = -\left[\frac{\partial L(y^{(i)}, F(\mathbf{x}^{(i)}))}{\partial F(\mathbf{x}^{(i)})} \right]_{F=F_{t-1}} \quad \text{for } i = 1, \ldots, n$$

4: Fit a weak learner $f_t(x)$ to the pseudo-residuals, i.e., train it using the training set $\{(\mathbf{x}^{(i)}, r_t^{(i)})\}_{i=1}^{n}$.

5: Compute the scaling parameter γ_t by solving the following line search (one-dimensional optimization problem):

$$\gamma_t = \arg\min_{\gamma} \sum_{i=1}^{n} L(y^{(i)}, F_{t-1}(\mathbf{x}^{(i)}) + \gamma f_t(\mathbf{x}^{(i)}))$$

6: Update the model by adding the weak learner:

$$F_t(\mathbf{x}) = F_{t-1}(\mathbf{x}) + \gamma_t f_t(\mathbf{x})$$

7: **end for**

8: Output the final model as the sum of all weak learners:

$$F_t(\mathbf{x}) = F_0 + \sum_{t=1}^{T} \gamma_t f_t(\mathbf{x})$$

10.4.3 Overview of XGBoost

XGBoost is an advanced version of gradient boosting that incorporates additional enhancements to improve both the speed and accuracy of the model. These enhancements include:

- **Regularization**: XGBoost includes like the elastic regression both L_1 (Lasso) and L_2 (Ridge) regularization to control the complexity of the model and prevent overfitting.
- **Weighted Quantile Sketch**: This algorithm enables efficient handling of sparse data and the creation of decision trees that are more robust to imbalanced data.
- **Tree Pruning**: XGBoost uses a method called *max depth pruning* rather than pre-pruning, allowing trees to grow as large as necessary before pruning back branches that do not improve model performance.

10.4.4 XGBoost Algorithm

The XGBoost algorithm can be broken down into the following steps:

Algorithm 17 XGBoost

Require: Training set $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$, number of boosting rounds T, learning rate η, regularization parameters λ and α

Ensure: An ensemble model consisting of T decision trees

1: Initialize the model with a constant prediction:

$$F_0 = \arg\min_{y} \sum_{i=1}^{n} L(y^{(i)}, y)$$

where $L(y^{(i)}, F(\mathbf{x}^{(i)}))$ is the loss function.

2: **for** each boosting round $t = 1, 2, \ldots, T$ **do**

3: Compute the **residuals** (gradients) and **second-order gradients** (Hessians) for each sample:

$$r_t^{(i)} = -\left[\frac{\partial L(y^{(i)}, F_{t-1}(\mathbf{x}^{(i)}))}{\partial F_{t-1}(\mathbf{x}^{(i)})}\right] \quad h_t^{(i)} = \left[\frac{\partial^2 L(y^{(i)}, F_{t-1}(\mathbf{x}^{(i)}))}{\partial \left(F_{t-1}(\mathbf{x}^{(i)})\right)^2}\right] \quad \text{for } i = 1, \ldots, n$$

4: Fit a new slow learner f_t to the residuals using the following objective function and the training set $\{(\mathbf{x}^{(i)}, \frac{r_t^{(i)}}{h_t^{(i)}}\}_{i=1}^{n}$:

$$\mathscr{L}^{(t)} = \sum_{i=1}^{n} \frac{1}{2} h_t^{(i)} \left[\frac{r_t^{(i)}}{h_t^{(i)}} - f_t(\mathbf{x}^{(i)})\right]^2 + \Omega(f_t) \approx \sum_{i=1}^{n} \left[\frac{1}{2} h_t^{(i)} f_t(\mathbf{x}^{(i)})^2 - r_t^{(i)} f_t(\mathbf{x}^{(i)})\right] + \Omega(f_t)$$

where $f_t(\mathbf{x})$ is the output of the t^{th} tree, and $\Omega(f_t)$ is a regularization term:

$$\Omega(f_t) = \alpha B + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

with B being the number of leafs, w_j being the leaf weights, and α and λ the regularization parameter.

5: Update the model by adding the new tree's predictions:

$$F_t = F_{t-1} + \eta f_t(\mathbf{x})$$

6: **end for**

7: Output the final model as the sum of all trees:

$$F(\mathbf{x}) = F_0 + \sum_{t=1}^{T} \eta f_t(\mathbf{x})$$

10.4.5 Key Features of XGBoost

Regularization

XGBoost includes L_1 (Lasso) and L_2 (Ridge) regularization in its objective function to control the complexity of the model. This helps prevent overfitting, especially when dealing with high-dimensional data.

Shrinkage (Learning Rate)

The learning rate η controls the contribution of each tree to the final model. By setting a small learning rate, XGBoost gradually updates the model, allowing more trees to be added without overfitting. This is often referred to as shrinkage.

Column Subsampling

XGBoost supports column subsampling, which means that for each tree (or even for each split within a tree), a random subset of features is selected. This adds diversity among the trees, reducing overfitting and improving generalization, similar to the Random Forest approach.

Handling Missing Data

XGBoost has a built-in capability to handle missing data. During the tree-building process, it automatically learns the best direction (left or right) for missing values, improving its robustness to incomplete datasets.

10.4.6 Advantages of XGBoost

- **High Performance**: XGBoost is known for its exceptional performance in terms of both accuracy and speed, making it suitable for large-scale and complex datasets.
- **Flexibility**: XGBoost supports various loss functions and custom objective functions, providing flexibility for a wide range of applications.
- **Built-in Regularization**: The incorporation of regularization helps control model complexity and prevents overfitting, which is crucial when dealing with high-dimensional data.
- **Efficient Handling of Sparse Data**: XGBoost's optimized algorithms for sparse data allow it to efficiently handle datasets with missing or zero values.
- **Feature Importance**: Like Random Forest, XGBoost can provide insights into feature importance, helping to identify the most influential features in the dataset.

10.4.7 Disadvantages of XGBoost

- **Complexity**: XGBoost's complexity and the large number of hyperparameters can make it challenging to tune and optimize, particularly for those new to the method.
- **Computational Resources**: While XGBoost is efficient, the training process can still be resource-intensive, especially with large datasets or when tuning hyperparameters extensively.

- **Interpretability**: Like other ensemble methods, the final model in XGBoost is less interpretable than a single decision tree, as it consists of an ensemble of many trees.

10.4.8 Practical Considerations

Hyperparameter Tuning

XGBoost has several hyperparameters that need to be carefully tuned to achieve optimal performance: i) the **Learning Rate (η)**, which controls the contribution of each tree (a lower learning rate requires more trees but can lead to better performance), ii) the **Number of Trees (Boosting Rounds)** as more trees generally improve performance, but increase the risk of overfitting, iii) the **Maximum Depth** of a tree (deeper trees capture more complex patterns but are prone to overfitting), iv) the **Subsample Ratio**, meaning the fraction of training samples to use when growing each tree (lower values make the model more robust to overfitting), and v) the **Column Subsample Ratio**, which denotes the fraction of features to use when growing each tree, adding diversity to trees and preventing overfitting.

Early Stopping

XGBoost supports early stopping during training, where the training process is halted if the model's performance on a validation set does not improve after a certain number of rounds. This is in general (and not just for tree related algorithms) a useful technique to prevent overfitting and reduce computational time.

Parallel and Distributed Computing

XGBoost is designed to take advantage of parallel and distributed computing, making it highly scalable. Note, however, that the different trees are by means trained in parallel, it is still a sequential algorithm. Only the training of each single tree is highly parallelized. Hence, XGBoost is optimized such that it can be run on multiple cores or even on a cluster of machines, making it suitable for large-scale data processing tasks.

11 Neural Networks I: First Steps

So far, we have covered two different ways of performing non-linear learning. The first one maps examples $\mathbf{x}$ explicitly into feature vectors $\boldsymbol{\phi}(\mathbf{x})$ which contain non-linear terms of the coordinates of $\mathbf{x}$. The second method translates the same problem into a kernel form so as to reduce the computations involved into comparisons between examples (via the kernel) rather than evaluating the feature vectors explicitly. This approach can be considerably more efficient in cases where the inner product between feature vectors, i.e., the kernel, can be evaluated without ever enumerating the coordinates of the feature vectors.

Here, we will formulate models – neural networks – where the feature representation is learned jointly with the classifier/regressor, both focused on improving the end-to-end performance.

11.1 Feed-Forward Neural Networks

Neural networks consist of a large number of simple computational units/neurons (e.g.,linear classifiers) which, together, specify how the input vector $\mathbf{x}$ is processed towards the final classification decision. Neural networks can do many things, but we will start with classification for continuity. In a simple case, the units in the neural network are arranged in layers, where each layer defines how the input signal is transformed in stages. These are called feed-forward neural networks. They are loosely motivated by how our brain processes many signals at the same time. The layers in our models include:

1. The **input layer**, where the units simply store the coordinates of the input vector (one unit/node assigned to each coordinate). The input units are special in the sense that they do not compute anything.

2. Possible **hidden layers** of units which represent complex transforms of the input signal, from one layer to the next, towards the final output. These units determine their activation by aggregating input from the preceding layer.

3. The **output layer**, for now a single unit, which is a linear classifier taking as its input the activations of the units in the penultimate (hidden or the input) layer. Figure 11.1 shows a simple feed-forward neural network with two hidden layers. The units correspond to the nodes in the graph and the edges specify how the activation of preceedeing units is aggregated. All the units except the input units act in this way. The input units are clamped to the observed coordinates of $\mathbf{x}$.

Such architectures are also called multi-layer perceptrons (MLP), depp neural networks (DNN), and artificial neural networks (ANN). All these acronyms basically mean the same thing (Fig. 11.1).

11.2 The Simplest Neural Network

Let's begin with the simplest neural network (a linear classifier). In this case, we only have d input units corresponding to the coordinates of $\mathbf{x} = [x_1, \ldots, x_d]^\top$ and a single linear output unit producing $F(\mathbf{x}, \boldsymbol{\theta})$ (here we will use a capital letter for a network, as it encompasses many operations chained together). We will use $\boldsymbol{\theta}$ to refer to the set of all parameters in a given network. So the number of parameters varies from one architecture to another. Now, the output unit receives a weighted combination of the inputs plus an overall offset/bias

$$z = \sum_{i=1}^{d} x_i w_i + b = \mathbf{w}^\top \mathbf{x} + b \tag{11.1}$$

$$F(\mathbf{x}, \boldsymbol{\theta}) = \sigma(z) = z \tag{11.2}$$

where $\sigma(\cdot)$ denotes the activation, which in this particular case is simply linear $\sigma(z) = z$. So this would be a standard linear regressor, if we want to classify each $\mathbf{x}$, then we take the sign of the network output. However, we will leave the network output as a real number, so that it can be fed into the Hinge or other such loss function. The parameters for this network are $\boldsymbol{\theta} = \{w_1, \ldots, w_d, b\}$.

Note that we can interpret and draw each unit in a neural network graphically as a linear unit based on the inputs that it receives. The output of the unit may be non-linear in general, evaluated as $\sigma(z)$ where $\sigma(\cdot)$ is typically a monotonically increasing function of z (e.g.,linear as above).

11.3 Hidden Layers

Now we extend the model a bit and consider a neural network with one hidden layer. As before, the input units are simply clamped to the coordinates of the input vector $\mathbf{x}$. In contrast, each of the m hidden units evaluate their output in two steps

$$z_j = \sum_{i=1}^{d} W_{ij} x_i + b_j \qquad \text{(what actually happens: } \mathbf{z} = \mathbf{x}^\top \mathbf{W} + \mathbf{b}\text{)} \tag{11.3}$$

$$\sigma(z_j) = \max\{0, z_j\} \tag{11.4}$$

where we have used so called rectified linear activation function (ReLU) which simply passes any positive input through as is and squashes any negative input to zero. A number of other activation functions are possible (see next section). Here we have used ReLU as an example.

Now, the single output unit no longer sees the input example directly, but only through the activations of the hidden units. In other words, as a unit, it can be written exactly as before but it now takes in each $\sigma(z_j)$ as input instead of the coordinates x_i.

$$z = \sum_{i=1}^{m} x_i w_i + b \tag{11.5}$$

$$F(\mathbf{x}, \boldsymbol{\theta}) = z \tag{11.6}$$

The output unit is again linear and functions as a linear classifier on a new feature representation $[\sigma(z_1), \ldots, \sigma(z_m)]^\top$ of each example. Note that the z_js are always functions of $\mathbf{x}$, but we have suppressed this dependence in the notation. The parameters $\boldsymbol{\theta}$ include in this case the weight matrix and bias vector from the first layer as well as the vector and scalar from the second layer.

How powerful is the neural network model with one hidden layer? It turns out that it is already a universal approximator in the sense that it can approximate any mapping from the input to the output if we increase the number of hidden units sufficiently enough. However, it is not necessarily easy to find the parameters that realize any specific mapping given by the training examples.

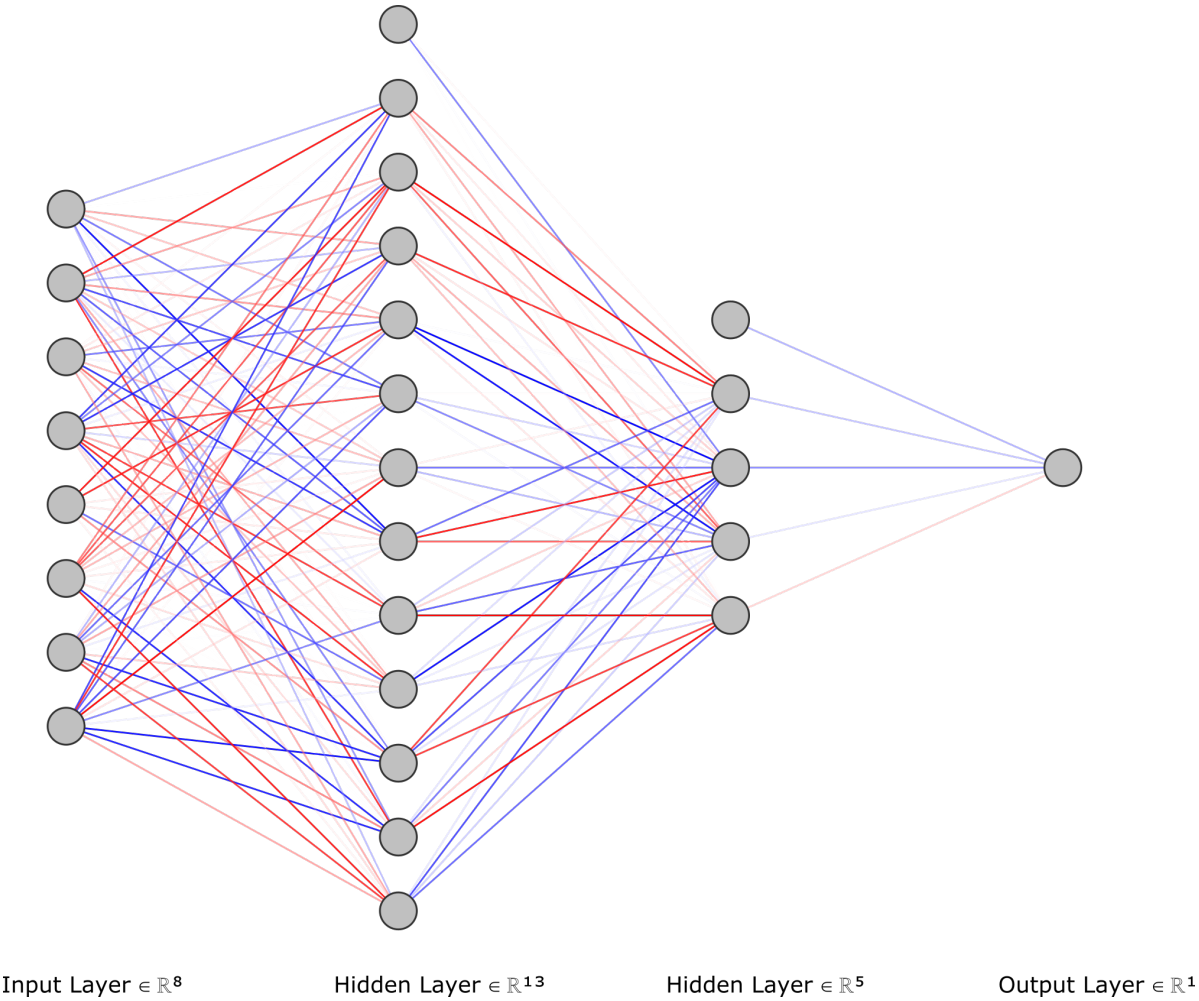

Figure 11.1: Depiction of a dense multilayer perceptron. The color of the weights indicate the sign of the edge weights. The bias are depicted as extra, source-less nodes. The intensity of a line scales with the absolute size of the edge weight.

11.4 Notation

In the context of a multi-layer perceptron (MLP), the notation typically includes indices for layers, nodes (neurons), and weights, as well as activation functions and biases. To standardize the notation of MLPs in this course, we will use the following:

1. **Layers:**
    - Let L be the total number of layers in the MLP, including the input layer but not counting

it towards the indexing. Hence, the input layer is considered layer 0, the first hidden layer is layer 1, and the output layer is layer L.

2. **Nodes (neurons)**
   - $n^{(l)}$ is the number of nodes in layer l, where $l \in \{0, 1, \ldots, L\}$.
   - $a_j^{(l)}$ is the activation of the j-th node in layer l, i.e., the output of the activation function, or for layer 0 the clamped input x_j.
3. **Weights**
   - $\mathbf{W}^{(l)}$ is the weight matrix for connections from layer $l-1$ to layer l. The element $[\mathbf{W}^{(l)}]_{ij} = W_{ij}^{(l)}$ represents the weight connecting the i-th node of layer $l-1$ to the j-th node of layer l.
4. **Biases**
   - $b_j^{(l)}$ is the bias term for the j-th node in layer l.
5. **Inputs and Outputs**
   - x_i is the i-th input feature and thus equal to $a_i^{(0)}$.
   - $\hat{y}_k = a_k^{(L)}$ be the k-th output prediction for architectures that have more than a single output value.
6. **Activation Functions**
   - $\sigma^{(l)}(\cdot)$ is the activation function used on layer l. One is free to mix activation functions from layer to layer There is no $\sigma^{(0)}$ and usually $\sigma^{(L)}(z) = z$.
7. **Forward Propagation**
   - The pre-activation (weighted sum plus bias) of the j-th node in layer l is denoted as $z_j^{(l)}$:

$$z_j^{(l)} = \sum_{i=1}^{n^{(l-1)}} W_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)} \quad \text{or} \quad \mathbf{z}^{(l)} = \mathbf{a}^{(l-1)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}$$

   - The activation of the j-th node in layer l is then:

$$a_j^{(l)} = \sigma^{(l)}(z_j^{(l)}) \quad \text{or} \quad \mathbf{a}^{(l)} = \sigma^{(l)}(\mathbf{z}^{(l)})$$

   where the second equality is an abuse of notation, implicitly applying the activation to every element of $\mathbf{z}^{(l)}$ individually.

11.5 Learning

Given a training set $S_n = \{(\mathbf{x}^{(k)}, y^{(k)}), k = 1, \ldots, n\}$ of samples $\mathbf{x} \in \mathbb{R}^d$ and labels y, we would like to estimate parameters $\boldsymbol{\theta}$ of the chosen neural network model so as to minimize the average loss over the training examples

$$J_n(\boldsymbol{\theta}) = \frac{1}{n} \sum_{k=1}^{n} \text{Loss}(y^{(k)}, F(\mathbf{x}^{(k)}; \boldsymbol{\theta})) \,, \tag{11.7}$$

where we assume that the loss is the Hinge loss $\text{Loss}_h(z) = \max\{0, 1-z\}$.

In order to minimize the average loss, we will resort to a simple stochastic optimization procedure rather than performing gradient descent steps on $J_n(\boldsymbol{\theta})$ directly. The stochastic version, while simpler, is also likely to work better with complex models, providing the means to randomize the exploration of good parameter values. On an abstract level: We sample a training example (or a batch of them) at random, and nudge the parameters towards values that would improve the classification of that particular example. Many such small steps will overall move the parameters in a direction that reduce the average loss. As it was already introduced previously in this course, you know the algorithm as stochastic gradient descent or SGD.

Note that the gradient with regards to $\boldsymbol{\theta}$

$$\nabla_\theta \text{Loss}(y^{(k)}, F(\mathbf{x}^{(k)}; \boldsymbol{\theta})) = \left[\frac{\partial}{\partial \theta_1} \text{Loss}(y^{(k)}, F(\mathbf{x}^{(k)}; \boldsymbol{\theta})), \ldots, \frac{\partial}{\partial \theta_D} \text{Loss}(y^{(k)}, F(\mathbf{x}^{(k)}; \boldsymbol{\theta})) \right] \tag{11.8}$$

has the dimension D, which is the number of all flexible parameters in the model, and it points in the direction of steepest loss increase. We therefore move the parameters in the opposite direction. The learning rate η should decrease slowly with the number of updates. It should be small enough that we don't overshoot (often), i.e., if η is large, the new parameter values might actually increase the loss after the update. The presence of hidden layers makes choosing η considerably more challenging. Another important difference to the algorithms studied up to this point is that it is critical that the parameters are NOT initialized to zero when we have hidden layers.

While our estimation problem may appear daunting with lots of hidden units, it is surprisingly easier if we increase the number of hidden units in each layer. In contrast,adding layers (deeper architectures) are tougher to optimize. The argument for increasing the size of each layer is that high-dimensional intermediate feature spaces yield much room for gradient steps. This is not the case with small models.

11.5.1 Backpropagation

Here we want to make the algorithm more concrete, adapt it to simple example networks, and see back-propagation (chain rule) in action. We will also discuss little adjustments to the algorithm that may help make it work better in practice. Our network generates a real valued output $F(\mathbf{x}^{(i)}; \boldsymbol{\theta})$ in response to any input vector $\mathbf{x} \in \mathbb{R}^d$. This mapping is mediated by parameters $\boldsymbol{\theta}$ (all weights and biases of the model).

Let us look at how to use stochastic gradient descent (SDG) to minimize the average loss over the training examples. The algorithm performs a series of small updates by focusing each time on a randomly chosen loss term. After many small updates, the parameters will have moved in a direction that reduces the overall loss above. More concretely, we iteratively select a training example in random order (we usually iterate over the entire training set in each epoch) and move the parameters in the opposite direction of their partial derivative.

$$\theta_i \leftarrow \theta_i - \eta \frac{\partial}{\partial \theta_i} \text{Loss}(y^{(k)}, F(\mathbf{x}^{(k)}; \boldsymbol{\theta})), \quad i = 1, \ldots, D \tag{11.9}$$

where D is the total number of parameters in the network. To use SGD, we have to specify three things: i) how to evaluate the derivatives, ii) how to initialize the parameters, and iii) how to set the learning rate.

No hidden units

Let us begin with the simplest network without hidden units, which is just a linear classifier ($\boldsymbol{\theta} = [w_1, \ldots, w_d, b]^\top$) and thus SGD can be performed with the Pegasos algorithm. Here, we will go through the calculation in detail to structure it in a manner that generalizes to more complex models.

Given any training example $(\mathbf{x}^{(k)}, y^{(k)})$, our first task is to evaluate

$$\frac{\partial}{\partial w_\ell} \text{Loss}(y^{(k)}, F(\mathbf{x}^{(k)}; \boldsymbol{\theta})) = \frac{\partial}{\partial w_\ell} \text{Loss}_h(y^{(k)} z^{(k)}) \tag{11.10}$$

where $z^{(k)} = \sum_{i=1}^d x_\ell^{(k)} w_\ell + b$ and we have assumed that the output unit is linear. Moreover, we will assume that the loss function is the Hinge loss, i.e., $\text{Loss}_h(z) = \max\{0, 1-z\}$. Then, the effect of w_i on the network output, and consequently the loss, can be evaluated by applying the chain rule

$$\frac{\partial}{\partial w_\ell} \text{Loss}_h(y^{(k)} z^{(k)}) = \left(\frac{\partial z^{(k)}}{\partial w_\ell} \right) \left(\frac{\partial \text{Loss}_h(y^{(k)} z^{(k)})}{\partial z^{(k)}} \right) \tag{11.11}$$

$$= \left(\frac{\partial \sum_{\ell=1}^d x_\ell^{(k)} w_\ell + b}{\partial w_\ell} \right) \left(\frac{\partial \text{Loss}_h(y^{(k)} z^{(k)})}{\partial z^{(k)}} \right) \tag{11.12}$$

$$= \left(x_\ell^{(k)} \right) \begin{pmatrix} -y^{(k)} & \text{if } \text{Loss}_h(y^{(k)} z^{(k)}) > 0 \\ 0 & \text{otherwise} \end{pmatrix} \tag{11.13}$$

The equations for the bias b are very similar. You may notice that the Hinge loss is not actually differentiable at the point where the product $yz = 1$. For our purposes here it suffices to use either left or right derivative. We can now explicitly write down the updates for the one layer neural network, and they look very much like the perceptron or Pegasos algorithm. Indeed, we will only get a non-zero update when if $\text{Loss}_h(y^{(k)} z^{(k)}) > 0$ and in those cases:

$$w_\ell \leftarrow w_\ell + \eta y^{(k)} x_\ell^{(k)} \quad \ell = 1, \ldots, d \tag{11.14}$$

$$b \leftarrow b + \eta y^{(k)} \tag{11.15}$$

One hidden layer

Let us now consider the two layer network discussed previously. Recall that the output is now obtained in stages, activating the hidden units, before evaluating the output. In other words,

$$F(\mathbf{x}^{(k)}; \boldsymbol{\theta}) = \sigma(\mathbf{x}^\top \mathbf{W} + \mathbf{b})\mathbf{w} + b \tag{11.16}$$

The last layer (the output unit) is again simply a linear classifier, but bases its decisions on the transformed input $[\sigma(z_1, \ldots, z_m)]$ rather than the original $\mathbf{x}$. As a result, we can follow the SGD updates we have already derived for the single layer model. Specifically, when the loss is non-zero,

$$w_j \leftarrow w_j + \eta_t y^{(k)} f(z_j^{(k)}), j = 1, \ldots, m \tag{11.17}$$

where here $z_j^{(k)}$ is the input to the j^{th} hidden unit resulting from example $\mathbf{x}^{(k)}$, note this is a deviation from the general notation introduced earlier. Note that now $f(z_j^{(k)})$ serves the same role as the input coordinate $x_j^{(k)}$ did in the single layer network.

In order to update W_{ij}, we must consider how it impacts the network output. Changing W_{ij} will first change z_j, then $f(z_j)$, then the final output. In this case the path of influence of the parameter on the network output is unique. There are typically many such paths in multi-layer networks (and must all be considered). To calculate the derivative of the loss with respect to W_{ij} we again have to apply the chain rule

$$\frac{\partial}{\partial W_{ij}}\text{Loss}_h(y^{(k)}z^{(k)}) = \left(\frac{\partial z_j^{(k)}}{\partial W_{ij}}\right)\left(\frac{\partial f(z_j^{(k)})}{\partial z_j^{(k)}}\right)\left(\frac{\partial z^{(k)}}{\partial f(z_j^{(k)})}\right)\left(\frac{\partial \text{Loss}_h(y^{(k)}z^{(k)})}{\partial z^{(k)}}\right) \tag{11.18}$$

$$= \left(x_i^{(k)}\right)\mathbb{1}[z_j^{(k)} > 0]\left(w_j\right)\begin{pmatrix} -y^{(k)} & \text{if Loss}_h(y^{(k)}z^{(k)}) > 0 \\ 0 & \text{otherwise} \end{pmatrix} \tag{11.19}$$

Note that $f(z_j) = \max\{0, z_j\}$ is not differentiable at $z_j = 0$, but we will simply take the left derivative. When there are multiple layers of units, the gradients can be evaluated efficiently by propagating them backwards from the output back towards the inputs. This is because each previous layer must evaluate how the next layer affects the output as the influence of the associated parameters goes through the next layer. The process of propagating the gradients backwards towards the input layer is called back-propagation. More concretely, let's see how we can cache some computations as we evaluate the derivatives. We proceed backwards, beginning by evaluating and storing how the loss changes relative to the last mathematical operation and so forth. Here an example

$$\delta^{(k)} = \left(\frac{\partial \text{Loss}_h(y^{(k)}z^{(k)})}{\partial z^{(k)}}\right) = \begin{cases} -y^{(k)} & \text{if Loss}_h(y^{(k)}z^{(k)}) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{11.20}$$

The exact value of this derivative depends on the input $\mathbf{x}^{(k)}$. Now we can go one operation further backwards and evaluate the next derivative

$$\delta_j^{(k)} = \left(\frac{\partial \text{Loss}_h(y^{(k)}z_j^{(k)})}{\partial z^{(k)}}\right) = \mathbb{1}[z_j^{(k)} > 0]\left(w_j\right)\delta^{(k)}, j = 1, \ldots, m \tag{11.21}$$

Note that we can now use the previously computed partial derivative. Finally, the derivative with respect to the most inner operation (the matrix multiplication) is

$$\left(\frac{\partial \text{Loss}_h(y^{(k)}z^{(k)})}{\partial W_{ij}}\right) = x_i^{(k)}\delta_j^{(k)} \tag{11.22}$$

So, put another way, by calculating δ's backwards, layer by layer, we can then easily evaluate the derivatives with respect to specific parameters on the basis of these cached values.

The further back the parameters are, the more multiplications we have gone through. This has the effect that the δ's may easily either explode or vanish, precluding effective learning (known as exploding or vanishing gradient problem). The issue of learning rate is therefore quite important for these parameters (as discussed before). Other "tricks" include clipping the gradients as they propagate backwards so that, numerically, they do not explode.

Properly initializing the parameters is much more important in networks with two or more layers. For example, if we set all the parameters (W_{ij}'s and w_j's) to zero, then also each activation $a_j^{(l)}$ is 0 and so is their gradient (at least when ReLU is used). Thus, no update occurs and the SGD would remain at the all-zero state.

Since all hidden units have (in this example) the same functional form, we must use the initialization process to break symmetries. This is normally achieved by initializing the parameters randomly, sampling each parameter value from a Gaussian distribution with zero mean and variance σ^2, where the variance depends on the layer (the number of units feeding to each hidden unit). For example, unit z_j receives d inputs in our two-layer model. We would like to set the variance of the parameters such that the overall input to the unit (after randomization) does not strongly depend on d (the number of input units feeding to the hidden unit). In this sense, the unit would be initialized in a way that does not depend on the network architecture it is part of. To achieve this, we could sample each W_{ij} from a zero-mean Gaussian distribution with variance $1/d$. As a result, the pre-activation output $z_j = \sum_{i=1}^{d} x_i W_{ij} + 0$ to each hidden unit (with zero offset) corresponds to a different random realization of W_{ij}, whose variance is independent of the input size d.

11.5.2 Regularization

Network models can be quite complex (have a lot of power to overfit to the trainingdata) as we increase the number of hidden units or add more layers (deeper architecture). Very deep architectures often have more parameters than there exist training examples, which is something one would normally avoid at any cost. There are many ways to regularize the models. The simplest way would be to add a squared penalty on the parameter values to the training objective, as we have done previously. This additional term would push the parameters towards zero and this term remains even when the loss is zero.

There are better ways to regularize large network models. In the two layer model, we could imagine increasing m, the size of the hidden layer, to create an arbitrarily powerful model. How could we regularize this model such that it would not overfit to noise in the training data? In order to extract complex patterns from the input example, each hidden node must be able to rely on the behavior of its neighbors so to complement each other. We can make this co-adaption harder by randomly turning off hidden nodes. In other words, with some probability we set each hidden nodes to have output zero, i.e., the unit is simply dropped out (this is called **Monte Carlo Dropout**). This randomization is done anew for each training example. When a node is turned off, the input/output weights coming in/going out will not be updated either. In a sense, we have dropped those nodes from the model for the current training sample. This process makes is much harder for units to co-adapt. Instead, units can rely on the signal from their neighbors only if a larger numbers of them support it.

At test time we can include all of the nodes as we would without randomization, but a slight modification is needed. Since during training each hidden node was present one a fraction of the time, the signal to the nodes ahead is different form what it would be if all the hidden nodes were present. As a result, we can simply multiply the outgoing weights from each hidden nodes by the dropout probability to compensate. This works well in practice. Theoretical justification comes from thinking about the randomization during training as a way of creating large ensembles of networks (different configurations due to dropouts). This reduction of outgoing weights is a fast approximation to the ensemble (a collection of independent models) output, which is more expensive to compute.

11.6 Optimization Algorithms

With complex architectures, we have to not only change the way we initialize the parameters but also think carefully how to choose the learning rate. We could just use a decreasing sequence of values, as we did with the Pegasos algorithm, $\eta_t = \eta_0/(t+1)$. For more complex architectures, this is not optimal any longer.

In SGD, the magnitude of the update is directly proportional to the gradient. When the gradient (slope) is small, so is the update. In contrast, if the objective function varies sharply with $\mathbf{w}\theta$, the gradient-based update would be very large. However, this behavior is exactly the wrong way around. When the objective function varies little, we can safely take larger steps to get to the minimum faster. Conversely, if the objective varies sharply, the update should be smaller to avoid overshooting. A fixed and/or decreasing learning rate is oblivious to such concerns.

11.6.1 AdaGrad (Adaptive Gradient Algorithm)

Instead of a fixed or continuously decreasing learning rate we can adaptively set the step-size based on the gradients (AdaGrad):

$$g_i \leftarrow \frac{\partial}{\partial \theta_i} \text{Loss}(y^{(k)}, F(\mathbf{x}^{(k)}; \boldsymbol{\theta})) \tag{11.23}$$

$$G_i \leftarrow G_i + g_i^2 \quad \text{(cumulative squared gradient)} \tag{11.24}$$

$$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sqrt{G_i + \epsilon}} g_i \quad \text{(adaptive gradient update)} \tag{11.25}$$

where ϵ is a small constant to avoid division by zero. The updates are performed vectorized for all the parameters in one go. With this algorithm, η can be set to a fixed value, since $\sqrt{G_i}$ reflects both the magnitude of the gradients as well as the number of updates performed. Note that we have some freedom in terms of how to bundle the adaptive scaling of learning rates. In the above example, the learning rate is adjusted separately for each parameter. Alternatively, we could use a common scaling per node in the network such that all the incoming weights to a node are updated with the same learning rate, adjusted by the cumulative squared gradient, which is now a sum of the individual squared derivatives.

The **AdaGrad** algorithm adjusts the learning rate for each parameter based on the magnitudes of the gradient. This means that parameters with large gradients have their effective learning rate reduced, while parameters with smaller gradients have their learning rates increased, relatively speaking. AdaGrad is particularly well-suited for dealing with sparse data, as it helps make larger updates to infrequent features.

One limitation of AdaGrad is that the accumulated squared gradients in G_i can lead to the effective learning rate becoming infinitesimally small, which may cause the model to stop learning too early.

11.6.2 RMSProp (Root Mean Square Propagation)

To address the decaying learning rate issue in AdaGrad, **RMSProp** was introduced. RMSProp modifies AdaGrad by keeping an exponentially decaying average of past squared gradients, which helps to avoid the drastic decay in learning rates. This allows RMSProp to perform better in non-convex settings, such as training deep neural networks.

The RMSProp update rule is:

$$g_i \leftarrow \frac{\partial}{\partial \theta_i} \text{Loss}(y^{(k)}, F(\mathbf{x}^{(k)}; \boldsymbol{\theta}))$$

$$G_i \leftarrow \alpha G_i + (1-\alpha) g_i^2 \tag{11.26}$$

$$\theta_i \leftarrow \theta_i - \frac{\eta}{\sqrt{G_i + \epsilon}} \cdot g_i \tag{11.27}$$

Here, α is the decay rate, typically set to a value like 0.9. It ensures that the sum of squared gradients G_i does not keep increasing, but instead becomes the exponentially weighted moving average of squared gradients (i.e., the contribution of gradients to the sum decreases exponentially).

RMSProp has proven to be very effective in training deep neural networks and is widely used in practice. It mitigates AdaGrad's diminishing learning rates and performs robustly in a variety of optimization problems.

11.6.3 Stochastic Gradient Descent with Momentum

Another problem of the "vanilla" form of SGD is that the updates can be noisy, especially when the loss landscape has ravines (i.e., steep and narrow valleys). The optimization process can oscillate, resulting in slow convergence.

To alleviate this problem, **momentum** is introduced into the optimization process. Momentum helps smooth the updates by accumulating a running average of past gradients, which allows the optimization process to maintain a consistent direction in the parameter space, thus speeding up convergence and reducing oscillations.

The update rule for SGD with momentum is given as

$$g_i \leftarrow \frac{\partial}{\partial \theta_i} \text{Loss}(y^{(k)}, F(\mathbf{x}^{(k)}; \boldsymbol{\theta}))$$

$$m_i \leftarrow \gamma m_i + \eta g_i \quad \text{(accumulation of past gradients)} \tag{11.28}$$

$$\theta_i \leftarrow \theta_i - m_i \quad \text{(update with exponential average of gradients)} \tag{11.29}$$

where m_i is the momentum term, which accumulates the moving average of past gradients and γ is the momentum coefficient (typically set to a value like 0.9), which controls how much of the previous velocity is retained.

In contrast to RMSProp, where we modify the learning rate with an exponentially weighted average of past gradient norms, here we modify the direction of the update (no square) by using an exponential average of prior gradients.

Understanding Momentum

The core idea behind **momentum** is inspired by the physical concept of inertia. Just as an object in motion tends to remain in motion in the same direction unless acted upon by an external force, in optimization, momentum helps carry the optimization process forward in the direction of the gradient, even if the current gradient suggests a slight deviation.

Without momentum, stochastic gradient descent can struggle in regions of the loss surface that have high curvature, noisy gradients, or local minima. Momentum dampens these oscillations by smoothing out the trajectory of the updates. Specifically:

- In regions where gradients consistently point in the same direction, momentum amplifies the gradient updates, allowing faster convergence.
- In areas where gradients oscillate (such as in the case of deep valleys), momentum helps to average out these oscillations, leading to smoother, more consistent progress.

Nesterov Accelerated Gradient (NAG)

An improvement to traditional momentum-based optimization is **Nesterov Accelerated Gradient (NAG)**. In NAG, instead of calculating the gradient at the current position, the gradient is calculated after a step is made in the direction of the current momentum. This anticipatory behavior leads to better adjustments of the updates and improves the convergence speed.

The update rule for NAG is as follows:

$$\mathbf{m}_t \leftarrow \gamma \mathbf{m}_{t-1} + \eta \nabla_\theta J(\boldsymbol{\theta}_{t-1} - \gamma \mathbf{m}_{t-1}) \tag{11.30}$$

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \mathbf{m}_t \tag{11.31}$$

In this formulation, we first make a "look-ahead" step by moving in the direction of the previous momentum, and then the gradient is evaluated. This allows the optimizer to make more informed updates, especially in complex loss landscapes.

11.6.4 Adam (Adaptive Moment Estimation)

The **Adam** optimizer combines the advantages of both AdaGrad and RMSProp. It computes individual adaptive learning rates for each parameter and also incorporates momentum, which helps, as we just discussed, to smooth out parameter updates. Momentum in optimization refers to the use of the moving average of past gradients to stabilize the direction of updates.

Adam maintains two moving averages: one for the gradients (first moment) and one for the squared gradients (second moment).

$$\mathbf{g}_t \leftarrow \frac{\partial}{\partial \boldsymbol{\theta}} \text{Loss}(y^{(k)}, F(\mathbf{x}^{(k)}; \boldsymbol{\theta})) \quad \text{(gradient at time step } t\text{)}$$

$$\mathbf{m}_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)\mathbf{g}_t \quad \text{(Update biased first moment estimate)} \tag{11.32}$$

$$G_t \leftarrow \beta_2 G_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \quad \text{(Update biased second raw moment estimate)} \tag{11.33}$$

$$\hat{\mathbf{m}}_t \leftarrow \frac{\mathbf{m}_t}{1-\beta_1^t}, \quad \hat{G}_t = \frac{G_t}{1-\beta_2^t} \quad \text{(Compute bias-correction)} \tag{11.34}$$

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \frac{\eta}{\sqrt{\hat{G}_t} + \epsilon} \cdot \hat{\mathbf{m}}_t \tag{11.35}$$

where $\mathbf{m}_t$ is the exponentially weighted moving average of the gradients (first moment) and G_t the analog for the squared gradients (second moment). β_1 and β_2 are hyperparameters controlling the decay rates of these moving averages (commonly set to 0.9 and 0.999, respectively). $\hat{m}_t$ and $\hat{G}_t$ are bias-corrected estimates of m_t and v_t to account for initialization at zero.

Adam probably the most popular and effective optimization algorithms in machine learning today due to its adaptive learning rate mechanism and momentum incorporation. It typically works well across a wide range of problems and requires little tuning.

11.7 Automatic Differentiation

You may have asked yourself, how all these step-wise differentiations can be performed. Building these rules for every different MLP architecture by hand would be a lot of work. This is where programming packages that can automatically compute derivatives become extremely important. Without them little of the progress with regards to ML models over the last years would have been possible.

Automatic differentiation (autodiff) is a computational technique for efficiently and accurately evaluating derivatives of functions. Unlike symbolic differentiation, which computes the derivative symbolically, and numerical differentiation, which approximates the derivative using finite differences, autodiff leverages the fact that all functions can be broken down into a sequence of elementary operations (addition, multiplication, etc.) for which the derivatives are known.

11.7.1 Basics of Automatic Differentiation

The core idea of autodiff is to apply the chain rule of calculus systematically to these elementary operations to compute derivatives. This is achieved in two primary modes: forward mode and reverse mode.

If we consider a function $y : \mathbb{R}^n \to \mathbb{R}^m$ with input $\mathbf{x} \in \mathbb{R}^n$, which we can break down into elementary steps, such as

$$\mathbf{y} = f(g(h(\mathbf{x}))) \quad \text{with} \quad h(\mathbf{x}) = \mathbf{v}_1, g(\mathbf{v}_1) = \mathbf{v}_2, f(\mathbf{v}_2) = \mathbf{y} \tag{11.36}$$

Then the chain rule gives

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{v}_2} \frac{\partial \mathbf{v}_2}{\partial \mathbf{v}_1} \frac{\partial \mathbf{v}_1}{\partial \mathbf{x}} \tag{11.37}$$

Forward Mode

In forward (or bottom-up or tangent) mode autodiff, we compute the derivative of a function with respect to one of its inputs by propagating derivatives from the input to the output. It is more efficient if the number of outputs is much larger the number of inputs $m >> n$. The forward mode can be visualized as:

$$\frac{\partial \mathbf{v}_i}{\partial \mathbf{x}} = \frac{\partial \mathbf{v}_i}{\partial \mathbf{v}_{i-1}} \frac{\partial \mathbf{v}_{i-1}}{\partial \mathbf{x}}$$

Reverse Mode

Reverse (or top-down or adjoint) mode autodiff is particularly efficient for functions where the number of inputs n is much larger than the number of outputs m ($n >> m$), making it ideal for training machine learning models. In reverse mode, we propagate derivatives from the output back to the input. Using the chain rule, we compute:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{v}_i} = \frac{\partial \mathbf{y}}{\partial \mathbf{v}_{i+1}} \frac{\partial \mathbf{v}_{i+1}}{\partial \mathbf{v}_i}$$

11.7.2 Applications

Automatic differentiation is widely used in various fields due to its efficiency and accuracy. Some notable applications include:

- **Machine Learning**: Autodiff is a key component in training neural networks. Frameworks like JAX and PyTorch rely heavily on reverse mode autodiff for backpropagation.
- **Optimization**: Many optimization algorithms require gradients, which can be efficiently computed using autodiff.
- **Scientific Computing**: Autodiff enables accurate sensitivity analysis and parameter estimation in complex simulations.

11.7.3 Advantages

Using autodiff to compute gradients automatically has several advantages over approaches like finite differences, reducing the need for manual differentiation. Autodiff computes derivatives to machine precision, avoiding the errors introduced by numerical differentiation. By leveraging the chain rule, autodiff efficiently propagates derivatives through a computational graph.

11.7.4 Example: Autodiff in Practice

Consider the function $f(x, y) = x^2 y + \sin(x)$. To compute the derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ using autodiff, we proceed as follows:

PyTorch

```python
import torch

def f(x, y):
return x**2 * y + torch.sin(x)

x = torch.tensor(1.0, requires_grad=True)
y = torch.tensor(2.0, requires_grad=True)

z = f(x, y)

# Compute gradients
z.backward()

# Extract gradients
grad_f_x = x.grad
grad_f_y = y.grad

print("df/dx =", grad_f_x.item())
print("df/dy =", grad_f_y.item())
```

In this example, we use the `requires_grad=True` flag to enable automatic differentiation for the tensors x and y. The `backward()` method computes the gradients, which are then accessed via the `grad` attribute of each tensor.

JAX

```
import jax
import jax.numpy as jnp

def f(x, y):
return x**2 * y + jnp.sin(x)

# Compute gradients
grad_f_x = jax.grad(f, argnums=0)
grad_f_y = jax.grad(f, argnums=1)

x, y = 1.0, 2.0
print("df/dx =", grad_f_x(x, y))
print("df/dy =", grad_f_y(x, y))
```

In this example, we use the `jax.grad` function to compute the partial derivatives of f with respect to x and y. The `argnums` argument specifies which argument to differentiate with respect to. The difference between JAX and PyTorch is that JAX computes a derivative function which can be evaluated at different points, whereas PyTorch only computes the gradient at a specific position.

11.8 Activation Functions

As already briefly discussed, activation functions introduce non-linearity into the neural network, enabling it to learn complex patterns. Here, we discuss some of the most common activation functions used in practice.

11.8.1 Sigmoid Activation Function

The sigmoid function maps any real-valued number into the range (0, 1). It is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{11.38}$$

The sigmoid function is often used in the output layer of binary classification problems. However, it has some drawbacks such as vanishing gradients, which can slow down training.

11.8.2 Hyperbolic Tangent (tanh) Activation Function

The tanh function is similar to the sigmoid function, but maps inputs to the range (-1, 1). It is defined as:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{11.39}$$

The tanh function is zero-centered, which generally makes it a better choice than the sigmoid function. However, it also suffers from the vanishing gradient problem.

11.8.3 Rectified Linear Unit (ReLU)

The ReLU function is one of the most popular activation functions in deep learning due to its simplicity and effectiveness. It is defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{11.40}$$

ReLU helps mitigate the vanishing gradient problem and is computationally efficient. However, it can suffer from the "dying ReLU" problem, where neurons can become inactive and only output zero.

11.8.4 Leaky ReLU

Leaky ReLU is a variation of ReLU designed to address the "dying ReLU" problem. It allows a small, non-zero gradient when the unit is not active. It is defined as:

$$\text{Leaky ReLU}(x;\alpha) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{11.41}$$

where α is a small constant (typically 0.01).

11.8.5 Parametric ReLU (PReLU)

PReLU is a generalization of Leaky ReLU where the parameter α is learned during training. It is defined as:

$$\text{PReLU}(x;\alpha) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{11.42}$$

PReLU can adaptively learn the negative slope, improving model performance.

11.8.6 Exponential Linear Unit (ELU)

ELU is another variation of ReLU that tends to converge faster and produce more accurate results. It is defined as:

$$\text{ELU}(x;\alpha) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases} \tag{11.43}$$

where α is a positive constant.

11.8.7 Swish and SiLU

Swish is yet another variation of ReLU that is nowadays used much commonly that ReLU. It is defined as:

$$\text{Swish}(x;\beta) = x\sigma(\beta x) = \frac{x}{1 + e^{-\beta x}} \tag{11.44}$$

where β is either a trainable parameter or a constant. The most common choice is $\beta = 1$, and the activation function is then called Sigmoid Linear Unit (SiLU).

11.8.8 Gaussian Error Linear Unit (GELU)

GELU is the last ReLU variant that we will present here. It is defined as:

$$\text{GELU}(x) = \frac{x}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right] \approx \frac{x}{1 + e^{-1.702x}} \tag{11.45}$$

its shape is very similar to that of Swish, as highlighted by the approximative expression.

11.8.9 Softmax Function

The softmax function is used in the output layer of multi-class classification problems. It converts logits into probabilities. It is defined as:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{11.46}$$

where x_i are the input values.

11.8.10 Summary

Choosing the right activation function can significantly impact the performance of a neural network. The sigmoid and tanh functions are often used in the output layers of binary and multi-class classification problems, respectively. ReLU and its variants (Leaky ReLU, PReLU, ELU) are commonly used in the hidden layers of deep neural networks due to their effectiveness in mitigating the vanishing gradient problem and improving training speed.

Table 11.1: Summary of common activation functions

Name	**Range**	**Formula**	**Pros/Cons**
Sigmoid	$(0, 1)$	$\sigma(x) = \frac{1}{1+e^{-x}}$	Vanishing gradient problem
Tanh	(-1, 1)	$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	Zero-centered, vanishing gradient
ReLU	$[0, \infty)$	$\text{ReLU}(x) = \max(0, x)$	Mitigates vanishing gradient, dying ReLU problem
Leaky ReLU	$(-\infty, \infty)$	$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$	Prevents dying ReLU
PReLU	$(-\infty, \infty)$	$\text{PReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$	Learns α, improves performance
ELU	$(-\alpha, \infty)$	$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^{x} - 1) & \text{if } x \leq 0 \end{cases}$	Fast convergence, accurate results
SiLU	$(\approx -0.27847, \infty)$	$\text{SiLU}(x) = \frac{x}{1+e^{-x}}$	Fast convergence, accurate results
GELU	$(\approx -0.17, \infty)$	$\text{GELU}(x) = \frac{x}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$	Accurate results, convergence can be faster than SiLU
Softmax	$(0, 1)$	$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$	Used in multi-class classification

12 Neural Networks II: Architectures

We will now see an overview of the large diversity in deep learning layers, though we will limit ourselves to the most common layers. Almost all the layers listed below were initially designed for a specific task. Hence, the nomenclature around that layer is targeted towards a specific kinds of data.

12.1 Convolutional Neural Networks (CNNs)

The first architecture that we will discuss is specifically tailored to images. If we view a digital image as a matrix of pixels, we can vectorize it, use it as input to a feed-forward neural network. Thus we designate an input unit to hold the value of each pixel in the image. If the image has millions of pixels, and we use layers with a few hundred hidden nodes, then the first layer in the feed-forward architecture already has over 10^8 parameters. While the number is large, and one would assume that we will need lots of labeled images to learn the associated parameters, the main problem is the question what the weights are meant to do. For example, it is often useful to first identify small features of an image such as edges, and later combine such features to detect certain objects. But in our very general architecture, we would have to separately learn the weights for recognizing the same simple feature in each possible location in the image (nothing carries over from one part of the image to another). Another issue is accounting for scale. Features are sometimes small, sometimes large, as are objects in the image, i.e., we have to account for resolution/zoom. Moreover, the objects should be classified correctly regardless of where they appear in the image, i.e., our classifier should be (at least partly) translation invariant. All of this would be a daunting task with a simple feed-forward architecture.

A convolutional neural network (CNN, Fig. 12.1) is a feed-forward neural network with a special structure. It consists of interspersed layers of **convolution** and **pooling** operations). A convolution layer applies simple local image filters across the image, producing featuremaps where the "pixels" in the feature map represent how much certain features were present in the corresponding location. Think of this process as breaking the image into overlapping little image patches, and applying a linear classifier to each patch. The size of the patches, and how much they overlap, can vary. The pooling layer, on the other hand, abstracts away from the location where the features are, only storing the maximum activation within each local area, capturing the *what* rather then the *where*.

12.1.1 Convolution

Usually, images are rank 3 tensors, as they have a height, a width, a several color channels (usually 3 for green, red, and blue). However, we will assume that our image is 1-dimensional with d pixels $x_1, \ldots, x_d$. Now we apply a filter of size k (odd number) which we move over our image with a certain stride (how many pixels we move before applying the filter again). For example, if we set the stride one and choose $k = 3$, then the i^{th} coordinate value of the feature map is obtained as

$$a_i^1 = \text{ReLU}\left(\sum_{j=1}^{k} x_{i+j-\lfloor k/2 \rfloor - 1} w_j\right) \tag{12.1}$$

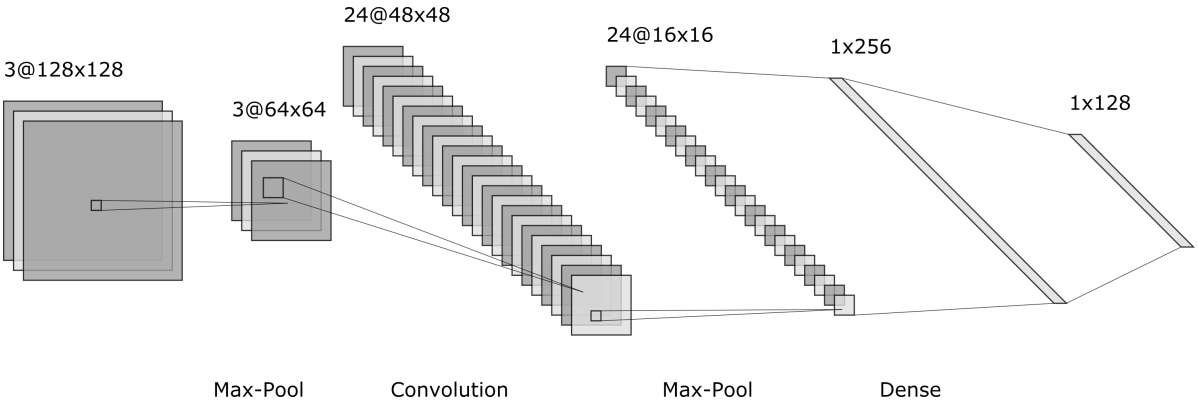

Figure 12.1: Depiction of a convolutional neural network. Architecture includes a convolutional and a pooling layer followed by 2 dense layers.

where $i = 1, \ldots, d$ is the same size as the original image. Further, we assume that the image is padded with zero when the filter extends over the boundaries of the image, e.g., $x_{-1} = 0$. If we increase the stride beyond one, the resulting feature map will be of size d/stride. Note that we only have to learn k weights for the convolutional filter instead of $d \times d/\text{stride}$. To this point, if viewed as a feed-forward layer, the weights in this layer are shared in the sense that one unit in the feature map layer uses the exact same weights to calculate its net input as any other unit in that layer. Thus the only parameters that need to be estimated are $[w_1, \ldots, w_k]$, i.e., the parameters of the filter. This makes the mapping very compact with regards to the number of parameters and it is independent of the image size. Nevertheless, the filter can be adjusted to look for different features and the resulting feature map identifies where the chosen feature is active in the image. We typically introduce a number of different feature maps, so called channels, to account for different patterns and the parameters in the associated filters are estimated jointly in the overall CNN model.

12.1.2 Pooling

Another key operation in CNNs is pooling. There are many different types of pooling operations, but the simplest one is just max-pooling. This is similar to convolution in terms of the filter size, and how it is applied across the image. However, there are no parameters to be learned, and the weighted average that the convolution produces is just replaced by the maximum. In other words, if the pooling filter size is k with stride one, then

$$a_i^{(l+1)} = \max_{j \in \{1,\ldots,k\}} a_{i+j-\lfloor k/2 \rfloor - 1}^{(l)} \,. \tag{12.2}$$

Max-pooling is typically applied with a relatively small k (can be an even number), but with a larger stride (e.g., 2) so as to reduce the resulting feature map size.

12.2 Invariances and Equivariances in Machine Learning

In machine learning, invariances and equivariances are properties that describe how a model's predictions change (or remain unchanged) under certain transformations of the input data. Making use of these properties can significantly improve the performance and generalization of models,

particularly in tasks where specific symmetries are present. By designing models that inherently respect such symmetries, we can achieve better performance and reduce the need for extensive data augmentation. Otherwise we would have to teach the model about these properties, e.g., the energy of a molecule in vacuum is the same irrespective of how I rotate its coordinates before I present it to the NN. Whether dealing with images, sets, point clouds, or other structured data, these properties play a fundamental role in modern machine learning approaches and are at the core as to why modern architectures outperform older ones, especially with respect to the amount of data needed for training them.

12.2.1 Translational Invariance

Translational invariance refers to the property where a model's predictions remain unchanged if the input data is translated (shifted) in space. This is a crucial property in many applications, such as image processing, where objects can appear at different locations within an image, i.e., it doesn't matter where the dog is in the picture as long as we can recognize it.

Mathematically, consider a function $f : \mathbb{R}^n \to \mathbb{R}$ that operates on an input vector $\mathbf{x}$. The function f is translationally invariant if:

$$f(\mathbf{x}) = f(\mathbf{x} + \mathbf{t}) \tag{12.3}$$

for any translation vector $\mathbf{t}$.

In practice, convolutional neural networks (CNNs) are designed to be translationally invariant by using convolutional layers that apply the same filters across different spatial locations.

12.2.2 Permutational Invariance

Permutational invariance is a property where a model's predictions are unchanged under any permutation of the input elements. This is particularly important in tasks involving sets or unordered collections of objects, such as point clouds or molecules.

Mathematically, let $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ be an input set. A function f is permutationally invariant if:

$$f(\mathbf{x}) = f(\mathbf{P}\mathbf{x}) \tag{12.4}$$

for any permutation matrix $\mathbf{P}$.

Models designed to handle sets, such as Deep Sets and PointNet, incorporate permutational invariance by using symmetric functions (e.g., sum, max) to aggregate information from individual elements.

12.2.3 Equivariance

Equivariance is a property where a model's predictions transform in a specific way when the input data is transformed. This is different from invariance, where the predictions remain unchanged. Equivariance is useful in applications where the structure of the input data is important, such as 3D rotations in computer graphics.

Formally, a function $f : \mathbb{R}^n \to \mathbb{R}^m$ is equivariant to a transformation T if there exists a transformation T' such that:

$$f(T(\mathbf{x})) = T'(f(\mathbf{x})) \tag{12.5}$$

A common example is rotational equivariance in image processing, where rotating an image leads to a corresponding rotation in the feature maps. Group convolutional neural networks (Group

CNNs) extend conventional CNNs to handle equivariances to more complex transformations, such as rotations and reflections.

For machine learned interatomic potentials (MLIPs) equivariance is in so far important, as the predicted atomic forces have to rotate with the molecules orientation in space.

12.2.4 Applications of Invariances and Equivariances

Incorporating invariances and equivariances into machine learning models can lead to significant improvements in performance and generalization. Here are some examples:

- **Image Classification:** Translational invariance in CNNs allows for accurate object recognition regardless of object position within an image.
- **Point Cloud Analysis:** Permutational invariance in models like PointNet enables robust classification and segmentation of 3D point clouds.
- **Molecular Modeling:** Permutational invariance in molecular graphs ensures that the model's predictions are consistent regardless of atom ordering.
- **Robotics and Vision:** Rotational and reflectional equivariance in Group CNNs facilitates tasks that involve understanding and manipulating objects from different angles.

12.3 Creating an Embedding

Another important type of input layers are embeddings. Embeddings convert non-machine readable things into vectors. They are typically used to convert characters or words. Characters or words are first converted into tokens separately as a pre-processing step and then the input to the embedding layer is the indices of the token, e.g., we might tokenize characters in the alphabet. There are 26 tokens (letters) in the alphabet (dictionary of tokens) and we could convert the word "hello" into $[7, 4, 11, 11, 14]$ (starting at 0), where 7 means the 8th letter of the alphabet.

After converting into indices, an embedding layer converts these indices into dense vectors of a chosen dimension. The rationale behind embeddings is to go from a large discrete space (e.g., all words in the English language) into a much smaller space of real numbers (e.g., vectors of size 5). For ML force fields, we need to convert the molecular structure (atomic numbers, xyz coordinates, etc.) into an embedding. This is often done over several layers, before feeding it into a dense architecture.

12.4 Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) are a class of neural networks designed to operate on graph-structured data. Unlike traditional neural networks that assume the input data is in the form of a fixed-size vector (e.g., images, text), GNNs can directly work with graphs, where data is represented as a set of nodes and edges. This makes GNNs particularly powerful for applications involving relational data, such as social networks, molecular structures, and recommendation systems. The output of GNNs is usually also a graph.

12.4.1 Introduction

A graph $G = (V, E)$ consists of a set of nodes or vertices V and a set of edges E connecting these nodes. Each node $v \in V$ can have associated features $\mathbf{x}_v$, and each edge $(u, v) \in E$ connecting vertices u and v can have associated edge features $\mathbf{e}_{uv}$. The goal of a GNN is to learn meaningful representations of the nodes, edges, or the entire graph that can be used for various tasks such as node classification, link prediction, and graph classification.

12.4.2 Simple Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) are a class of neural networks that operate on graph-structured data. Graphs consist of nodes (vertices) and edges (links), which can represent a wide variety of data structures such as social networks, molecular structures, and knowledge graphs. The key idea behind GNNs is to learn node representations by aggregating information from their neighbors, thereby capturing the underlying graph structure.

Basic Architecture of GNNs

The basic architecture of a GNN consists of multiple layers, where each layer updates the node representations by aggregating information from their neighbors. A simple GNN layer can be defined as follows:

$$\mathbf{a}_i^{(k+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \mathbf{W}^{(k)} \mathbf{a}_j^{(k)} + \mathbf{b}^{(k)}\right) = \sigma\left(\mathbf{W}^{(k)} \sum_{j \in \mathcal{N}(i)} \mathbf{a}_j^{(k)} + \mathbf{b}^{(k)}\right) \tag{12.6}$$

where:

- $\mathbf{a}_i^{(k)}$ is the representation of node i at layer k. Note that my notation is done for consistency, in the literature the symbol $\mathbf{h}_i$ is often used for node embeddings.
- $\mathcal{N}(i)$ is the set of neighbors of node i.
- $\mathbf{W}^{(k)}$ and $\mathbf{b}^{(k)}$ are learnable weight matrix and bias vector at layer k.
- σ is a non-linear activation function (e.g., ReLU), which is vectorized, i.e., it is applied element-wise to its argument.

Permutation Invariance in GNNs

Permutation invariance is an important property for GNNs, especially when dealing with unordered graph data. A GNN is permutation invariant if the node representations are invariant to the ordering of nodes and edges in the graph. This means that if we swap (permute) the nodes or edges, the resulting node representations (and ultimately the output of the GNN) should remain the same.

To achieve permutation invariance, the aggregation function used in the GNN layers must be symmetric. Common symmetric aggregation functions include: i) sum aggregation (show in eq. (12.6)), ii) mean aggregation (the sum of node features is divided by the amount of neighbors), and iii) max aggregation, where instead of a sum we only take the maximum within the neighborhood.

These symmetric functions ensure that the order of neighbors does not affect the aggregated representation, thereby providing permutation invariance.

Non-Permutation Invariant Structures

Not all GNN architectures are permutation invariant. For example, if a GNN uses a non-symmetric aggregation function or incorporates positional encoding that depends on the specific ordering of nodes or edges, it will not be permutation invariant. An example of a non-permutation invariant structure is:

$$\mathbf{a}_i^{(k+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \mathbf{W}_{ij}^{(k)} \mathbf{a}_j^{(k)} + \mathbf{b}^{(k)}\right) \tag{12.7}$$

In this case, the weight matrix $\mathbf{W}_{ij}^{(k)}$ depends on the specific pair of vertices (i, j), and thus the aggregation is not symmetric. This makes the GNN sensitive to the order of nodes and edges, resulting in a loss of permutation invariance.

12.4.3 Graph Convolutions

Graph convolutions are the core building blocks of GNNs, analogous to the convolutional layers in CNNs for image data. The primary idea behind graph convolutions is to aggregate information from a node's neighbors to update its feature representation.

One popular approach is the Graph Convolutional Network (GCN), which defines the convolution operation as follows:

$$\mathbf{a}_v^{(k+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{1}{c_{vu}} \mathbf{W}^{(k)} \mathbf{a}_u^{(k)}\right) \tag{12.8}$$

Here, $\mathbf{a}_v^{(k)}$ represents the feature vector of node v at the k-th layer, $\mathcal{N}(v)$ denotes the set of neighbors of node v, c_{vu} is a normalization constant (often set to $\sqrt{\deg(v)\deg(u)}$, where the degree of a node is the number of other nodes it shares an edge with), $\mathbf{W}^{(k)}$ is a trainable weight matrix, and σ is an activation function such as ReLU.

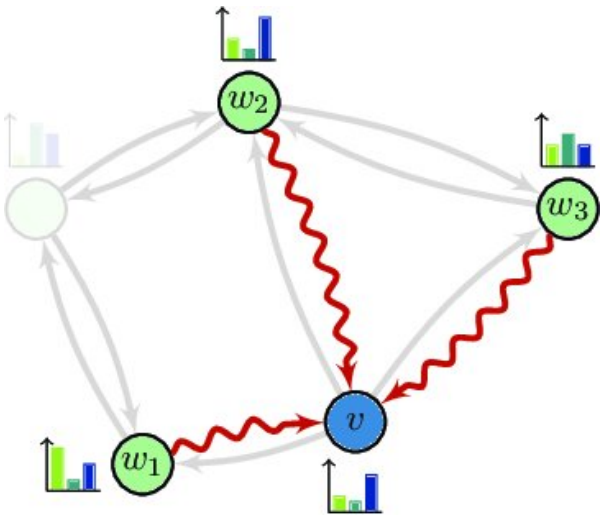

Figure 12.2: Message passing for a single node in a GNN layer. Each direct neighbor of a node passes its message along the given edge. Each node aggregates their incoming messages to update its current node representation. Image taken from [2]

12.4.4 Message Passing

Message passing is a general framework for defining graph convolutions. In this framework, each node v updates its representation by aggregating messages from its neighbors. The message passing process can be described in two phases: message computation and message aggregation.

During the message computation phase, each node u computes a message m_{uv} to send to its neighbor v. This message can depend on the features of both nodes and the edge between them:

$$m_{uv} = \phi(\mathbf{a}_u^{(k)}, \mathbf{a}_v^{(k)}, \mathbf{e}_{uv}) \tag{12.9}$$

Here, ϕ is a function that computes the message, which can be as simple as concatenating the features followed by a linear transformation.

During the message aggregation phase, each node v aggregates the messages from its neighbors to update its feature representation:

$$\mathbf{a}_v^{(k+1)} = \psi\left(\mathbf{a}_v^{(k)}, \bigoplus_{u \in \mathcal{N}(v)} m_{uv}\right) \tag{12.10}$$

Here, $\bigoplus$ is a permutation invariant aggregation function, such as summation or averaging, that can accept an arbitrary number of inputs. ψ then specifies how the node features are updated by the aggregated messages, which is then followed an activation function.

12.4.5 Readout Layers

After multiple layers of graph convolutions and message passing, we obtain node-level representations, which are still organized in a graph. However, in chemistry it is rare that our label itself is a graph. Typically we either have node labels (per-atom) or a molecular label (one label for the entire graph), e.g., the energy or solubility of a molecule.

The process of converting the graph output from the GNN into our predicted node labels or graph label is called the **readout**. If we have node labels, we can simply discard the edges and use the latest node features as embeddings for maybe a few dense feed-forward layers..

On the other hand, if we want to predict a label for the entire graph like the energy of the molecule or the net charge, we need to be careful when converting from node/edge features to a graph label. If we simply put the node features into a dense layer to get to the desired shape graph label, we will lose permutation invariance. The readout we did above in the solubility example was a reduction over the node features to get a graph feature. Then we used this graph feature in dense layers. It turns out this is the only way to obtain a graph-wide readout: we reduce the node embeddings to a node feature and then aggregate those to a graph-readout.

Extensie vs. Intensive Properties

It is important to distinguish whether your labels are intensive or extensive. An intensive label is one whose value is independent of the number of nodes (or atoms), e.g., an excitation energy. The readout for an intensive label should (generally) be independent of the number of a nodes/atoms. Therefore, the reduction in the readout could be a mean or max, but not a sum. In contrast, an extensive label should (generally) use a sum for the reduction in the readout. An example of an extensive molecular property is enthalpy of formation.

12.5 Hyperparameters

When looking back at the possible architectures, we have to choose if we use bias, the activation function, and the output shape. There more complex layers we use the more choices there are. These choices begin to accumulate and in a neural network you may have billions of possible combinations of them. These choices about shape, activation, initialization, and other layer arguments are called **hyperparameters**. They are parameters in the sense that they can be tuned, but they are not trained on our data so we call them hyperparameters to distinguish them from the trainable parameters, the ones we change with SGD.

Choosing these hyperparameters is difficult and we have to rely on experience / published results for ranges of reasonable parameters. In deep learning, we usually are in a space where many different choices yield similar, expressive results. However, optimizing hyperparameters makes training faster and/or require less data. For example, initializing the weights cleverly can be more important than having a very complex architecture. There are ways to scan and optimize hyperparameters, but more often than not we take hyperparameters from previous work as a starting guess and only change them if we believe we really need to.

12.5.1 Train - Validation - Test

The number of hyperparameters is so high that overfitting can actually occur by choosing hyperparameters that minimize error on the test set. This is something we definitely do not want. Therefore, it is inadvisable to use the test data to adjust hyperparameters. Hence, in deep learning we split our data three ways:

1. Training set: Data used for trainable parameters $\boldsymbol{\theta}$.
2. Validation set: Data used to choose hyperparameters and/or to monitor overfitting on the training examples.
3. Test set: The data that is exclusively used to report the estimate of the generalization error.

As a reminder, regardless of whether you split your data set three-ways or use other approaches, generalization error is the error on unseen data.

12.5.2 Tuning

The main answer to the question "How does one tune hyperparameters?" is unfortunately "by hand". Hyperparameters can be continuous (e.g., regularization strength), categorical (e.g., which activation), and discrete (e.g., number of layers). For example, a way to tune hyperparameters is a topic called meta-learning, which aims to learn hyperparameters by looking at multiple related datasets. Another area is auto-machine learning, where optimization strategies that do not require derivatives are used to tune hyperparameters.

13 Probabilistic Models

Probabilistic machine learning methods leverage the principles of probability theory to model uncertainty in predictions and data. This chapter provides an overview of some fundamental probabilistic machine learning approaches, including Naive Bayes classifiers and Gaussian Processes. By the end of this chapter, you should have a basic understanding of these methods.

13.1 Naive Bayes Classifiers

Naive Bayes classifiers are a family of simple yet effective probabilistic classifiers based on applying Bayes' theorem with strong independence assumptions between the features.

13.1.1 Bayes' Theorem

Bayes' theorem is the cornerstone of Bayesian inference and is given by:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{13.1}$$

where $P(A|B)$ is the posterior probability, $P(B|A)$ is the likelihood, $P(A)$ is the prior probability, and $P(B)$ is the marginal likelihood. The notation $P(A|B)$ denotes the probability of A occurring given B.

When using Bayes' equation $P(A)$ includes the prior assumptions about the problem and is chosen by the user, this can greatly bias the outcome of the inference. The marginal likelihood is usually unknown and in practice determined as a normalization constant.

13.1.2 The Naive Bayes Assumption

In the context of classification, let C be the class variable and $\boldsymbol{\phi}(\mathbf{x})$ be the feature vector of dimension d. The naive Bayes assumption is that the features are conditionally independent given the class:

$$P(\boldsymbol{\phi}(\mathbf{x})|C) = \prod_{i=1}^{d} P(\phi(\mathbf{x})_i|C) \tag{13.2}$$

13.1.3 Naive Bayes Classifier Model

Using Bayes' theorem, the posterior probability of class C_k given a feature vector $\boldsymbol{\phi}(\mathbf{x})$ is:

$$P(C_k|\boldsymbol{\phi}(\mathbf{x})) = \frac{P(\boldsymbol{\phi}(\mathbf{x})|C)P(C_k)}{P(\boldsymbol{\phi}(\mathbf{x}))} \tag{13.3}$$

Since $P(\boldsymbol{\phi}(\mathbf{x}))$ is constant for all classes, we can simplify this to:

$$P(C_k|\boldsymbol{\phi}(\mathbf{x})) \propto P(\boldsymbol{\phi}(\mathbf{x})|C_k)P(C_k) \tag{13.4}$$

The naive Bayes classifier assigns the class label $\hat{C}$ that maximizes the posterior probability

$$\hat{C} = \arg\max_{C_k} [P(\boldsymbol{\phi}(\mathbf{x})|C_k)P(C_k)] \tag{13.5}$$

where $P(C_k)$ is the relative frequency of the class label in the training set. Naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(\phi(\mathbf{x})_i|C)$.

Even though they make very simplifying assumptions, naive Bayes classifiers work quite well in many real-world situations, e.g., spam filtering. They are simple and easy to implement. They require a small amount of training data to estimate the necessary parameters, but they are even efficient for large datasets. They also perform well with high-dimensional data. Naive Bayes classifiers can be relatively fast, compared to more complex methods. However, the actual probabilities predicted by naive Bayes should not be interpreted even if the classification works decently. Further, the very strong independence assumptions between features (the naive assumption) has the effect that they may fail for a problem with correlated features.

13.1.4 Gaussian Naive Bayes

For continuous features, a common approach is to assume that the features follow a Gaussian distribution. The likelihood of the feature $\phi(\mathbf{x})_i$ given the class C_k is:

$$P(\phi(\mathbf{x})_i|C_k) = \frac{1}{\sqrt{2\pi\sigma^2_{C_k,i}}} \exp\left(-\frac{(\phi(\mathbf{x})_i - \mu_{C_k,i})^2}{2\sigma^2_{C_k,i}}\right) \tag{13.6}$$

where $\mu_{C_k,i}$ and $\sigma_{C_k,i}$ are the mean and standard deviation of the feature number i for class C_k.

13.2 Gaussian Processes

Gaussian Processes (GPs) are a powerful and flexible approach to non-parametric regression and classification. They provide a probabilistic framework that can model uncertainty in predictions.

13.2.1 Definition of a Gaussian Process

A Gaussian Process is a collection of random variables, any finite number of which have a joint Gaussian distribution. It is defined by a mean function $\mu(\mathbf{x})$ and a covariance function $\text{cov}(\mathbf{x}, \mathbf{x}')$:

$$f(\mathbf{x}) \sim \mathcal{GP}(\mu(\mathbf{x}), \text{cov}(\mathbf{x}, \mathbf{x}')) \tag{13.7}$$

where $\mu(\mathbf{x})$ represents the mean of the process, and $\text{cov}(\mathbf{x}, \mathbf{x}')$ describes the covariance between function values at different points.

13.2.2 Mean and Covariance Functions

The mean function $\mu(\mathbf{x})$ is often assumed to be zero for simplicity:

$$\mu(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})] = 0 \tag{13.8}$$

The covariance function $\text{cov}(\mathbf{x}, \mathbf{x}')$ defines the covariance between the function values at two points $\mathbf{x}$ and $\mathbf{x}'$. A commonly used covariance function is the Radial Basis Function (RBF) kernel:

$$\text{cov}(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2\ell^2}\right) \tag{13.9}$$

where σ_f^2 is the variance parameter and ℓ is the length scale parameter. We need here two parameters, such that the covariance can be larger than 1.

13.2.3 Training and Inference with Gaussian Processes

Given training data $S_n = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$, we want to predict the function value $f(\mathbf{x}^*)$ at a new input $\mathbf{x}^*$.

Training

The training phase basically only involves computing the covariance matrix $\mathbf{\Sigma}$ for all training examples:

$$[\mathbf{\Sigma}(\mathbf{X}, \mathbf{X})]_{ij} = \text{cov}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \tag{13.10}$$

where $\mathbf{X}$ denotes the matrix of the stacked feature vectors in S_n (this is mainly used for notation here). We can choose the covariance function or kernel.

The joint distribution of the training outputs $\mathbf{y}$ and the function value at $\mathbf{x}^*$ is:

$$\begin{pmatrix} \mathbf{y} \\ f(\mathbf{x}^*) \end{pmatrix} \sim \mathcal{N}\left(0, \begin{pmatrix} \mathbf{\Sigma}(\mathbf{X}, \mathbf{X}) + \sigma_n^2 I & \mathbf{\Sigma}(\mathbf{X}, \mathbf{x}^*) \\ \mathbf{\Sigma}(\mathbf{x}^*, \mathbf{X}) & \mathbf{\Sigma}(\mathbf{x}^*, \mathbf{x}^*) \end{pmatrix}\right) \tag{13.11}$$

where σ_n^2 is the noise variance, and $\mathbf{\Sigma}(\mathbf{X}, \mathbf{x}^*)$ is the covariance vector between the training inputs and the new input and $\mathbf{\Sigma}(\mathbf{x}^*, \mathbf{x}^*)$ is the scalar (co)variance of the new input $\mathbf{x}^*$ (the value of the covariance function for two identical inputs).

Inference

To make predictions at a new input $\mathbf{x}^*$, we condition the joint Gaussian distribution on the observed data. The posterior distribution of $f(\mathbf{x}^*)$ is Gaussian with mean and variance given by:

$$\mathbb{E}[f(\mathbf{x}^*)|S_n] = \mathbf{\Sigma}(\mathbf{x}^*, \mathbf{X})\,(\mathbf{\Sigma}(\mathbf{X}, \mathbf{X}) + \sigma_n^2 I)^{-1}\,\mathbf{y} \tag{13.12}$$

$$\text{Var}[f(\mathbf{x}^*)|S_n] = \mathbf{\Sigma}(\mathbf{x}^*, \mathbf{x}^*) - \mathbf{\Sigma}(\mathbf{x}^*, \mathbf{X})\,(\mathbf{\Sigma}(\mathbf{X}, \mathbf{X}) + \sigma_n^2 I)^{-1}\,\mathbf{\Sigma}(\mathbf{X}, \mathbf{x}^*) \tag{13.13}$$

13.2.4 Hyperparameter Optimization

The hyperparameters of the covariance function (e.g., σ_f, ℓ, σ_n) can be optimized by maximizing the log marginal likelihood of the observed data. The log marginal likelihood is given by:

$$\ln p(\mathbf{y}|\mathbf{X}) = -\frac{1}{2}\mathbf{y}^\top(\mathbf{\Sigma}(\mathbf{X}, \mathbf{X}) + \sigma_n^2 I)^{-1}\mathbf{y} - \frac{1}{2}\ln|\mathbf{\Sigma}(\mathbf{X}, \mathbf{X}) + \sigma_n^2 I| - \frac{n}{2}\ln 2\pi \tag{13.14}$$

Maximizing this log marginal likelihood involves finding the hyperparameters that make the observed data most probable under the Gaussian Process model.

13.2.5 Uncertainty Quantification

One of the key advantages of Gaussian Processes is their ability to provide uncertainty estimates for predictions. The posterior variance $\text{Var}[f(\mathbf{x}^*)|S_n]$ quantifies the uncertainty of the prediction at $\mathbf{x}^*$. A high variance indicates high uncertainty, while a low variance indicates high confidence in the prediction.

13.2.6 Acquisition Function

In Bayesian optimization and active learning, an acquisition function is used to decide where to sample next. It balances exploration (sampling in regions of high uncertainty) and exploitation (sampling in regions of high predicted mean). Common acquisition functions include:

Expected Improvement (EI)

The expected improvement at a point $\mathbf{x}$ is given by:

$$\text{EI}(\mathbf{x}) = \mathbb{E}[\max(0, f(\mathbf{x}) - f_{\text{best}})] \tag{13.15}$$

where f_{best} is the best observed function value so far.

Upper Confidence Bound (UCB)

The UCB acquisition function is given by:

$$\text{UCB}(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})] + \kappa\sqrt{\text{Var}[f(\mathbf{x})]} \tag{13.16}$$

where κ is a parameter that controls the trade-off between exploration and exploitation.

Probability of Improvement (PI)

The probability of improvement at a point $\mathbf{x}$ is given by:

$$\text{PI}(\mathbf{x}) = \Phi\left(\frac{\mathbb{E}[f(\mathbf{x})] - f_{\text{best}}}{\sqrt{\text{Var}[f(\mathbf{x})]}}\right) \tag{13.17}$$

where Φ is the cumulative distribution function of the standard normal distribution.

13.3 Comparison of Naive Bayes and Gaussian Processes

Naive Bayes and Gaussian Processes are both probabilistic methods, but they have different strengths and weaknesses. Naive Bayes is simple and fast, making it suitable for large-scale text classification problems. Gaussian Processes, on the other hand, provide a flexible and powerful framework for regression and classification, with the ability to model uncertainty in predictions for complex, non-linear relationships. However, they are computationally expensive and may not scale well to large datasets. And GPs require careful choice of covariance function and hyperparameters

References

1. Zhang, X. *et al.* Polymer-Unit Fingerprint (PUFp): An Accessible Expression of Polymer Organic Semiconductors for Machine Learning. *ACS Applied Materials & Interfaces* **15,** 21537–21548. https://doi.org/10.1021/acsami.3c03298 (May 2023).
2. Piatkowski, N. *et al.* in. Chap. 4 Structured Data (De Gruyter, 2023).