Machine Learning
Lecture Notes

January 2021

Contents

Change log

Version V2
Corrected a typo on page 15 ($h_\theta < 0.5$)
Corrected a typo in the first sentence of section 3.5
Corrected a typo in equation 3.9

Chapter 1

Introduction

1.1 Machine learning

The *Hello World!* of machine learning is the classification of handwritten digits. You are given a set of images containing handwritten digits (see figure 1.1) and the task is to determine what digit (0 - 9) corresponds to the image. For a human this is a relatively simple task, for a computer this is not so easy. If you would like to hard-code an algorithm it will be extremely difficult to handle all the different ways the digits are written. However with the advance of machine learning, even these complicated tasks can be solved by the computer. The concept is that you have an algorithm that *learns* from data you provide.

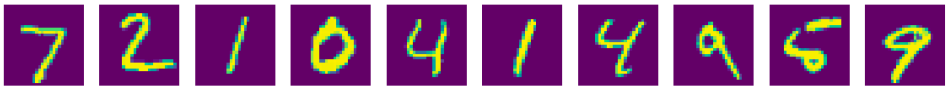

Figure 1.1: Images of handwritten digits from the MNIST data set. Classifying these images is the *Hello World!* of machine learning.

There are basically three different ways a computer can learn:

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning

In *supervised learning* you provide the algorithm with data. This data can for example be (in the case of classifying handwritten digits) the intensity values of the pixels that make up the image. In the images in the figure there are $28 \times 28 = 784$ pixels and thus each sample (or instance) of the dataset is said to have 784 *features*. The algorithm predicts the corresponding digit and compares is to what it should be. To be able to do that, each instance of the data not only has a set of features, but also a *label* that characterizes the instance. In this case the label can be either one of the 10 possible digits (0 - 9). The algorithm tries to minimize the deviation between prediction and the correct label by tuning *weight* parameters. By feeding the algorithm many instances it can *learn* to correctly predict new instances, that the algorithm has never seen before. Examples of supervised learning are: linear and logistic regression, support vector machines and neural networks.

In *unsupervised learning* the algorithm only has knowledge on the features of the instances and no prior knowledge on the corresponding labels. In unsupervised learning the algorithm tries to find some structure in the data by "itself". Examples are Kmeans clustering and dimension reduction such as principle component analysis.

In *reinforcement learning* you provide the algorithm a *state* (e.g. a chess board with all the positions of the pieces). The algorithm (in reinforcement learning often referred to as the *agent*) produces an *action* (i.e. a move on the chessboard) and gets feedback trough a *reward*. In the case of chess this reward will be large if the *agent* wins. By playing many games the *agent* tries to maximize the *reward*.

1.2 Notation

1.2.1 Math

First, we will use the following convention to write scalars, vectors and matrices:

- Scalars are written in lowercase (η) or uppercase (L)
- Vectors are written in bold lowercase ($\mathbf{b}$)
- Elements of vectors are in lowercase with subscript (b_i)
- The norm of a vector is written as $\|\mathbf{b}\|$
- Matrices are written in bold uppercase ($\mathbf{W}$)
- Elements of matrices are in uppercase with subscript (W_{ij}, with i the row and j the column)

1.2.2 Data

The following notation conventions are used for the data.

- M: The number of samples in the dataset. E.g if your dataset contains 1000 images of handwritten digits then $M = 1000$
- N: The number of features excluding a *bias*, so for the handwritten digits $N = 784$. Including a *bias* the feature vector has a length of $N + 1$
- $y^{(m)}$: The label of one (the m-th) instances of the dataset. In the example of classifying handwritten digits, each instance has a label that can be either one of the values 0,1,2..9.
- $\mathbf{x}^{(m)}$: The feature vector. Each sample can have one or more features that characterize the sample. In the example of classification handwritten digits the feature vector is made up of the pixel values of the images of the digits. The feature vector is a column vector, with the first item the *bias* which is always 1[1]:

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ .. \\ x_n \\ .. \\ x_N \end{bmatrix} = \begin{bmatrix} 1 \\ x_1 \\ .. \\ x_n \\ .. \\ x_N \end{bmatrix} \tag{1.1}$$

[1] In some algorithms like neural networks the bias is treated separately.

Chapter 2

Linear Regression

2.1 Introduction

Assume we have a data set of M data points characterized by label $y^{(m)}$ and feature vector $\mathbf{x}^{(m)}$. An example of a dataset is plotted in figure 2.1. In this case the label is just a scalar and the feature vector has in addition to the *bias* only one element ($N = 1$). In Linear regression we try to define a (linear) model based on the data at hand such that we can make predictions on the value of the label given some new feature vector.

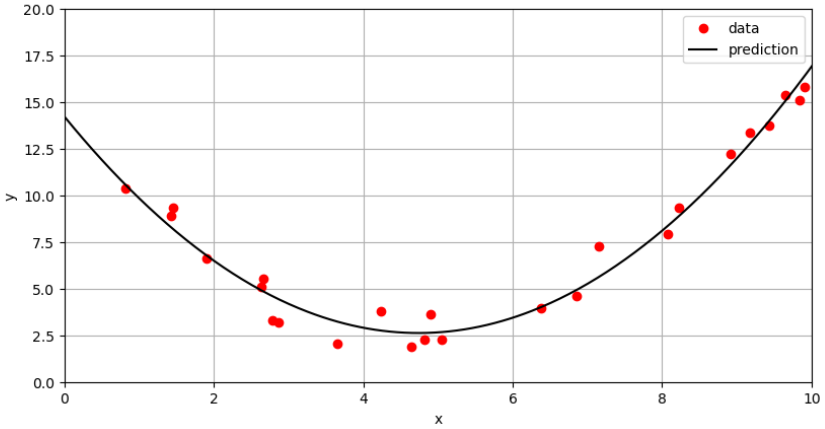

Figure 2.1: A data set of $M = 25$ points and a prediction based on a linear regression model.

2.2 Modeling

We represent the whole dataset as follows.

$$\mathbf{y} = \begin{bmatrix} y^{(0)} \\ y^{(1)} \\ .. \\ y^{(m)} \\ .. \\ y^{(M-1)} \end{bmatrix}, \mathbf{X} = \begin{bmatrix} \left(\mathbf{x}^{(0)}\right)^T \\ \left(\mathbf{x}^{(1)}\right)^T \\ .. \\ \left(\mathbf{x}^{(m)}\right)^T \\ .. \\ \left(\mathbf{x}^{(M-1)}\right)^T \end{bmatrix} \tag{2.1}$$

, so in the matrix $\mathbf{X}$ each row contains the features of one sample from the dataset, and in the vector $\mathbf{y}$ contains the corresponding label. In linear regression we try to find a linear relation between

the features and the label. However, the label can have a non-linear dependence on the feature(s). For example in figure 2.1 the relation between the label (y) and the feature (x) is clearly not linear. To solve that problem we artificially add features computed from the feature vector $\mathbf{x}$. In the example, we could add a feature that is the square of x. So if originally the first feature vector was (note that the first item is always 1, the *bias*):

$$\mathbf{x}^{(0)} = \begin{bmatrix} 1 \\ 0.7 \end{bmatrix} \tag{2.2}$$

, we artificially expanded this to:

$$\mathbf{x}^{(0)} = \begin{bmatrix} 1,0 \\ 0.7 \\ 0.49 \end{bmatrix} \tag{2.3}$$

Now it is possible to predict the corresponding label using a linear combination:

$$\begin{aligned} \hat{y}^{(0)} &= \theta_0 x_0^{(0)} + \theta_1 x_1^{(0)} + \theta_2 x_2^{(0)} \\ \hat{y}^{(0)} &= \theta_0 \times 1 + \theta_1 \times 0.7 + \theta_2 \times 0.49 \end{aligned} \tag{2.4}$$

, where the parameters θ are the weights. From now one we assume that the feature vectors $\mathbf{x}^{(m)}$ contain all features: both original and any derived feature.

For a given data point $(y^{(m)}, \mathbf{x}^{(m)})$ we can then make a prediction of the value of $y^{(m)}$, which we denote by $\hat{y}^{(m)}$:

$$\begin{aligned} \hat{y}^{(m)} &= \theta_0 x_0^{(m)} + \theta_1 x_1^{(m)} + \cdots + \theta_n x_n^{(m)} + \cdots + \theta_{N-1} x_{N-1}^{(m)} \\ \hat{y}^{(m)} &= \sum_{n=0}^{N} X_{mn} \theta_n, \qquad \text{with } X_{mn} = x_n^{(m)} \end{aligned} \tag{2.5}$$

Note that the term *linear regression* does by no means mean that the relation between y and $\mathbf{x}$ is linear. The computed extra features can as we have seen be non-linear (e.g. polynomial). It is called *linear* regression only because there is a *linear* dependence of the model on the weights.

Because we have M data points there are M equations that we can conveniently write as:

$$\hat{\mathbf{y}} = \mathbf{h}_\theta(\mathbf{X}) = \mathbf{X}\boldsymbol{\theta} \tag{2.6}$$

, where the function $\mathbf{h}_\theta$ is the *hypothesis* function that depends on the weights θ, and:

$$\mathbf{X} = \begin{bmatrix} x_0^{(0)} & \cdot & x_n^{(0)} & \cdot & x_N^{(0)} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ x_0^{(m)} & \cdot & x_n^{(m)} & \cdot & x_N^{(m)} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ x_0^{(M-1)} & \cdot & x_n^{(M-1)} & \cdot & x_N^{(M-1)} \end{bmatrix}, \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \cdot \\ \theta_n \\ \cdot \\ \theta_N \end{bmatrix} \tag{2.7}$$

Note that the elements of $\mathbf{X}$ can be denoted by X_{mn}, with $X_{mn} = x_n^{(m)}$.

To optimize the weight parameter vector θ we need to minimize the difference between data $\hat{\mathbf{y}}$ and model $\mathbf{h}_\theta$. We define an error vector ϵ as the difference:

$$\boldsymbol{\epsilon} = \mathbf{h}_\theta(\mathbf{X}) - \mathbf{y} \tag{2.8}$$

To this end we define a loss function that we then try to minimize.

2.3 The loss function

A widely used loss function (also referred to as the *cost* function) is the quadratic loss function which is just the normalized norm of the vector ϵ:

$$\begin{aligned} L &= \frac{1}{M}\frac{1}{2}\left\|\boldsymbol{\epsilon}\right\|^2 \\ L &= \frac{1}{M}\frac{1}{2}\left\|\mathbf{X}\boldsymbol{\theta}-\mathbf{y}\right\|^2 \end{aligned} \tag{2.9}$$

Note that there is an additional factor $\frac{1}{2}$ in the loss function. This is by convention any has no influence on the result. In fact, we can multiply the loss function by any scalar we like without changing the training. The goal is now to choose the weights such that the loss is minimal.

2.4 Minimizing the loss function

Because we need to minimize the loss function by tweaking the weight parameters we need to derive the derivatives of the loss function with respect to the weights. These derivatives can be computed using:

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = \frac{1}{M}\mathbf{X}^T\left(\mathbf{X}\boldsymbol{\theta}-\mathbf{y}\right) \tag{2.10}$$

To derive the above equation we consider the component form of the loss function:

$$L = \frac{1}{M}\frac{1}{2}\sum_{m=0}^{M-1}\left(-y_m+\sum_{n=0}^{N}X_{mn}\theta_n\right)^2 \tag{2.11}$$

Next we take the derivative with respect to the weight θ_n:

$$\begin{aligned} \frac{\mathrm{d}L}{\mathrm{d}\theta_n} &= \frac{\mathrm{d}}{\mathrm{d}\theta_n}\left\{\frac{1}{M}\frac{1}{2}\sum_{m=0}^{M-1}\left(-y_m+\sum_{n=0}^{N}X_{mn}\theta_n\right)^2\right\} \\ &= \frac{1}{M}\frac{1}{2}\sum_{m=0}^{M-1}\frac{\mathrm{d}}{\mathrm{d}\theta_n}\left(-y_m+\sum_{n=0}^{N}X_{mn}\theta_n\right)^2 \\ &= \frac{1}{M}\sum_{m=0}^{M-1}\left(-y_m+\sum_{n=0}^{N}X_{mn}\theta_n\right)\frac{\mathrm{d}}{\mathrm{d}\theta_n}\left(\sum_{n=0}^{N}X_{mn}\theta_n\right) \\ &= \frac{1}{M}\sum_{m=0}^{M-1}\left(-y_m+\sum_{n=0}^{N}X_{mn}\theta_n\right)X_{mn} \\ &= \frac{1}{M}\sum_{m=0}^{M-1}\epsilon_m X_{mn} \\ &= \frac{1}{M}\sum_{m=0}^{M-1}X^T_{nm}\epsilon_m \end{aligned} \tag{2.12}$$

which is the component form of equation 2.10. To be clear, X^T_{nm} refers to the component (n,m) of the transpose of matrix $\mathbf{X}$.

The loss is minimized if the derivative of the loss w.r.t. the weights equals zero:

$$\begin{aligned} \frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} &= 0 \\ \mathbf{X}^T\left(\mathbf{X}\boldsymbol{\theta}-\mathbf{y}\right) &= 0 \end{aligned} \tag{2.13}$$

Now it might be tempting to set $\mathbf{X}\boldsymbol{\theta} = \mathbf{y}$ and then to compute the weights as $\boldsymbol{\theta} = \mathbf{X}^{-1}\mathbf{y}$. However, this does not work since $\mathbf{X}$ is not necessarily square and therefore the inverse does not exist. Therefore we procede as follows:

$$\begin{aligned} \frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} &= 0 \\ \mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) &= 0 \\ \mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y} &= 0 \\ \mathbf{X}^T\mathbf{X}\boldsymbol{\theta} &= \mathbf{X}^T\mathbf{y} \\ \boldsymbol{\theta} &= \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y} \end{aligned} \tag{2.14}$$

The term $\left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T$ is called the *Moore-Penrose* inverse of $\mathbf{X}$. The expression derived above allows to compute the optimal weights such that the loss is minimal. Note that, since we chose the *quadratic* loss function, there exists only one θ that minimises the loss. Alternatively a gradient descent method can be used, which will be discussed next.

2.5 Gradient descent

If we start by choosing some initial values for the weights, then we can update the weights in an iterative procedure according to:

$$\boldsymbol{\theta} \rightarrow \boldsymbol{\theta} - \eta\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} \tag{2.15}$$

, where η is a parameter that determines how fast we move towards the minimum. The value of η is usually chosen in the range $0 < \eta < 1$. A careful choice is important as a too small value will result in slow convergence and a too large value will never result in finding the minimum (see figure 2.3). A strategy can also be to start with a relative large value of η and too slowly lower the value every so many iterations. Furthermore, it is common practice to keep track of the loss during the gradient descent. Plotting the loss versus the iteration step (or every so many iterations if the number of iterations if very large) gains much insight in the overall performance of the method and helps to tune parameters like the training rate η.

When dealing with large data sets gradient descent can be computational expensive (or worse, it may not fit the in the computer memory). A solution is to use *stochastic gradient descent*. With this method not all data is used in each step of the gradient descent algorithm, but only single data points or *batches*. The data is subdivided in (mini) batches and the gradients descent step is performed for each batch. Going through all the training data is called to an *epoch*. Usually many *epochs* are required to find the minimum loss.

2.6 Under- and over-fitting

In machine learning in general the performance of the model may suffer from under and over fitting the data. To illustrate this we fitted the data of figure 2.1 using different numbers of polynomial features. When we take a degree of 1, we basically have a linear relationship between y and feature x. Obviously the data does not satisfy such a simple linear relation. This is an example of under fitting. When we add a quadratic term as a feature the prediction becomes much better. If we choose the degree of polynomial features too large, the model can fit almost any data point in the data and as a result the *loss* will probably become very small. But the model probably does not generalize very well. If we make predictions with this model on new data it will likely perform very poor. In this case the data is over fitted.

To detect under- and over-fitting, and to optimize the model, the data is usually divided in three:

1. **training set**. This data is used to train the model (i.e. determine the weights of the model that minimize the loss). Usually this is the largest part of the total data set.

2. **validation set**. This data set is used to test the performance of different models and to tune *hyperparameters* of the model (here for example the polynomial degree) and in this way address

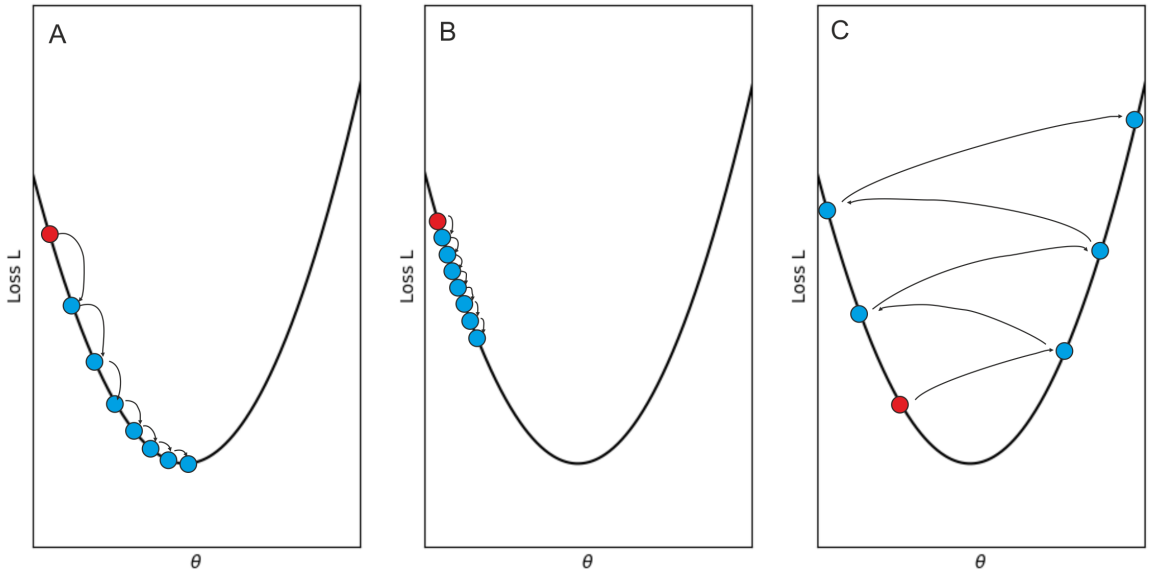

Figure 2.2: In gradient descent you start with an initial guess of the weights (red marker). Depending on the value of the rate η you either quickly find the minimum of the loss in an iterative way (panel A). If you choose η too small it takes many iterations to end up in the minimum (panel B). Even worse, if you choose η too large you can even move away from the minimum. To keeps things simple only a single weight θ is shown. In practice we have N weights (one for each feature) and therefore we are descending towards the minimum on a N-dimensional surface.

not only the performance off the model on data that was not seen during training, but also choose the best model and settings.

3. **test set**. This data is used to test the final model. Note that the validation set cannot be used as this data is used to tune the model in thus not independent anymore.

The terms under-fitting and over-fitting are related to the terms *bias* and *variance*. In machine learning, a high *bias* means there is a larger error between the model predictions and the data (high loss). In this case we are under-fitting the data. In contrast, a high *variance* means that the predictions vary a lot between different subsets of data. The question then arises is how to determine the optimal model? In the example: what is the proper polynomial degree? Plotting the loss L as function of the polynomial degree for both the training data and the validation data can give some insight. If the polynomial degree is low, the loss for both training and validation data will be more or less equal and high. Increasing the polynomial degree will result in lower loss. However, further increasing the polynomial degree beyond some optimum value will still lower the loss on the training set, but the loss for the validation set will start to increase. It is exactly at this point where we have the best trade-off between *bias* and *variance*.

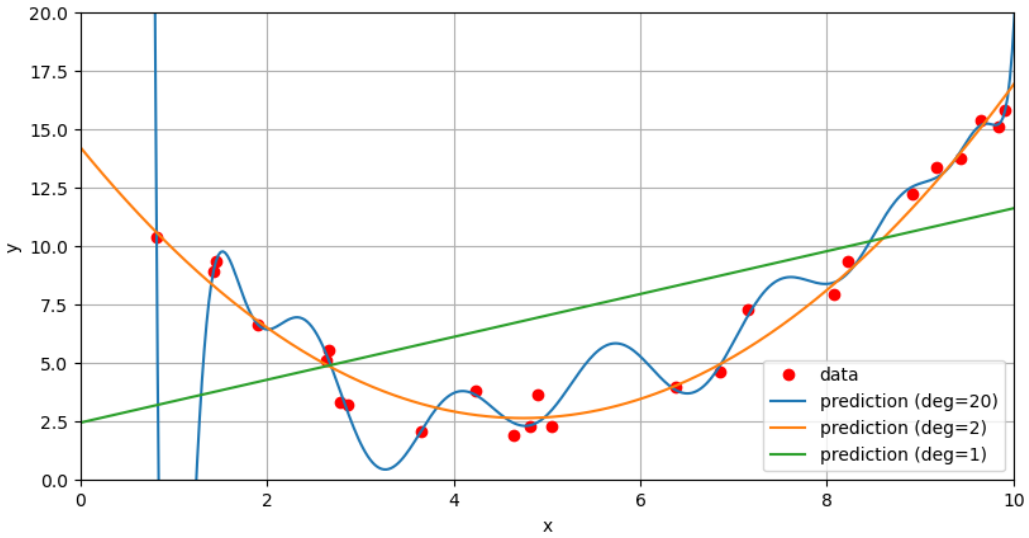

Figure 2.3: Predictions based on polynomial functions using different degrees of order. Clearly visible is that a polynomial degree of 1 under fits the data, while a degree of 20 over fits the data.

Chapter 3

Logistic Regression

3.1 Introduction

Logistic regression deals with classification. As an example we take some data from the widely used *iris* data set. This data contains features on different type of irises (the flowers). In figure 3.1 we plot two features x_1 and x_2 (the petal length and the petal width) on the x- and y-axis for two type of irises. Clearly two clusters appear and this allows to predict what flower it is based on the two features.

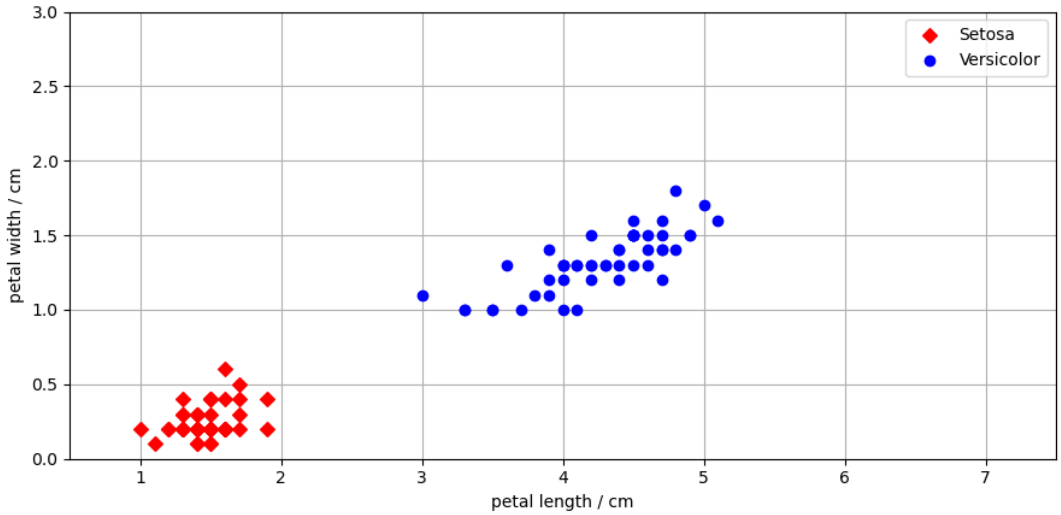

Figure 3.1: The dataset *Iris* from `sklearn`. The goal is to classify the type of flower based on petal length and width.

In the previous chapter the label $y^{(m)}$ could have any (scalar) value. In classification problems this is not the case. In classification we have a distinct number of *categories*. If we want e.g. to classify the flowers there are only two distinct categories *Versicolor* and *Setosa*. For the corresponding labels we use "1" for the *Versicolor* iris and "0" for the *Setosa* iris and thus we have only two options for the label:

$$y^{(m)} \in \{0, 1\} \tag{3.1}$$

In order to model/predict the label given the features it makes sense to use an hypothesis function $\mathbf{h}_\theta(\mathbf{X})$ that only gives values between 0 and 1. In that case any value larger than 0.5 gives the prediction "1" and any value smaller than 0.5 gives the prediction "0". A common choice is to use the logistic function for this.

3.2 Logistic function

The Logistic function is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.2}$$

A plot of this function is given in figure 3.2. The function has the desired properties. It can take values from 0 to 1 and it is s-shaped (also referred so as a sigmoid function). If the value of z is larger than 0 then $\sigma(z) > 0.5$, likewise if $z < 0$ we find $\sigma(z) < 0.5$. As we need it later, we give here also the derivative of the sigmoid function, which takes the relative simple form:

$$\frac{\mathrm{d}\sigma(z)}{\mathrm{d}z} = \sigma(z)\,(1 - \sigma(z)) \tag{3.3}$$

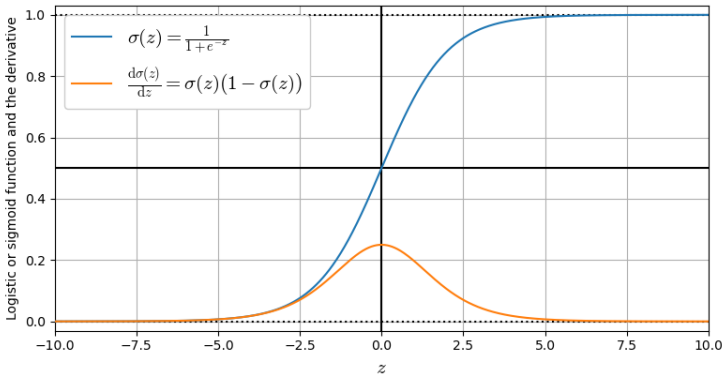

Figure 3.2: The logistic or sigmoid function and it's derivative.

If we apply the logistic function, our hypothesis function becomes:

$$\begin{aligned} \mathbf{h}_\theta(\mathbf{X}) &= \sigma(\mathbf{X}\boldsymbol{\theta}) \\ &= \frac{1}{1 + e^{-\mathbf{X}\boldsymbol{\theta}}} \end{aligned} \tag{3.4}$$

For each sample the hypothesis functions generates a value between 0 and 1 on which we can decide to which category the sample belongs.

3.3 The decision boundary

As we have shown, classification relies on the value of the hypothesis function. A value larger than 0.5 predicts e.g. a label "1" (or the *Versicolor* iris) and a value smaller than 0.5 predicts a label "0" (referring to the *Setosa* iris). Take a single sample with three features (including the *bias* of 1):

$$x_0 = 1; x_1 = \text{length}; x_2 = \text{width}$$

Then we find

$$\mathbf{x}^T\boldsymbol{\theta} = \theta_0 \times 1 + \theta_1 \times \text{length} + \theta_2 \times \text{width} \tag{3.5}$$

We can define/compute a decision boundary as the values of length and width that give $\mathbf{x}^T\theta = 0$ as that yields a value of the hypothesis function of 0.5. The decision boundary is thus given by:

$$0 = \theta_0 \times 1 + \theta_1 \times \text{length} + \theta_2 \times \text{width} \tag{3.6}$$

Let's say we determined the weights as $\theta_0 = 2.1$, $\theta_1 = -0.5$ and $\theta_2 = -1.0$ than we can write the following equation which just describes a line:

$$\text{width} = -0.5 \times \text{length} + 2,1 \tag{3.7}$$

This line is called the decision boundary and is plotted in figure 3.3. Below the decision boundary (in this case) $h_\theta < 0.5$ and all feature vectors in this area will classify as "0", whereas all feature vectors above the decision boundary classify as "1". Clearly, the decision boundary is not always a line. If we would have three features then the decision boundary is described by a plane. Also it is not always linear. By introducing new features that are non-linear functions of the feature vector $\mathbf{x}$ non-linear decision boundaries will emerge as we will see later.

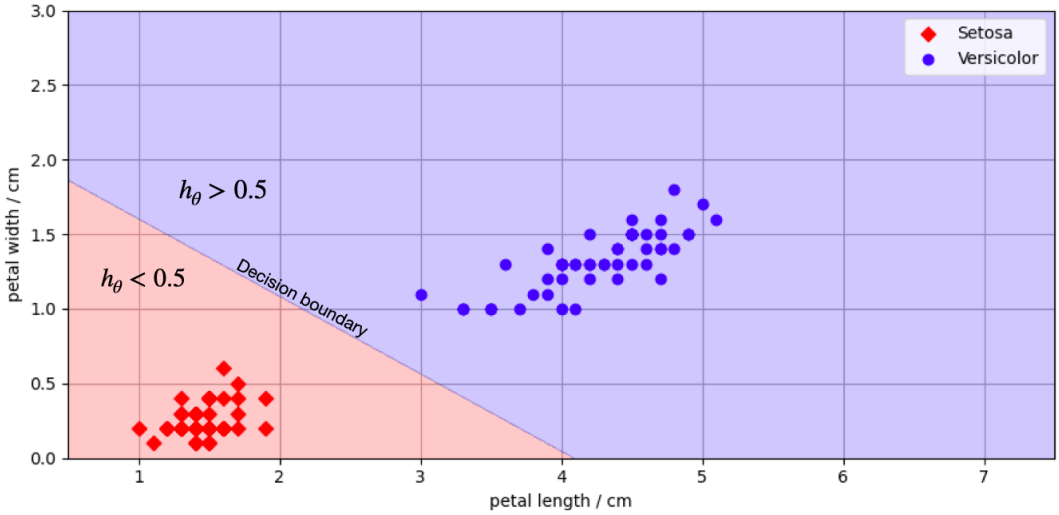

Figure 3.3: A linear decision boundary separating the two categories. The red shaded area indicate all points for which the hypothesis function $h_\theta < 0.5$, whereas the blue shaded area indicate all points for which $h_\theta > 0.5$. The decision boundary is at the interface ($h_\theta = 0.5$).

3.4 The loss function

In finding the best weights a loss function is needed just as with linear regression. However, the quadratic loss function in combination with a logistic hypothesis function does not result in a convex function. This means that the loss function can exhibit many local minima which poses a problem if we want to use a gradient descent approach to find the global minimum of the loss function. We can easily be trapped in a local minimum and as a result determine wrong weights. The solution lies in choosing another loss function. Without going into the details, the *log loss* function is a convex function and is given by:

$$L = -\frac{1}{M}\sum_{m=0}^{M-1} y^{(m)} \ln\left[h_\theta(\mathbf{x}^{(m)})\right] + \left(1 - y^{(m)}\right) \ln\left[1 - h_\theta(\mathbf{x}^{(m)})\right] \tag{3.8}$$

This expression can be understood by realizing that the label $y^{(m)}$ can have only two values "0" or "1". If the label $y^{(m)} = 1$ the above equation reduces just to a sum of $-\ln\left[h_\theta(\mathbf{x}^{(m)})\right]$ which approaches 0 if $h_\theta(\mathbf{x}^{(m)})$ equals 1 (so a correct prediction) and approaches ∞ if $h_\theta(\mathbf{x}^{(m)})$ equals 0 (a wrong prediction). If on the other hand the label $y^{(m)} = 0$ only terms $\ln\left[1 - h_\theta(\mathbf{x}^{(m)})\right]$ are left, which are infinity if $h_\theta(\mathbf{x}^{(m)})$ equals 1 and approaches 0 if $h_\theta(\mathbf{x}^{(m)})$ equals 0. So this loss function forces the hypothesis function to produce values that correspond to the labels (see also figure 3.4).

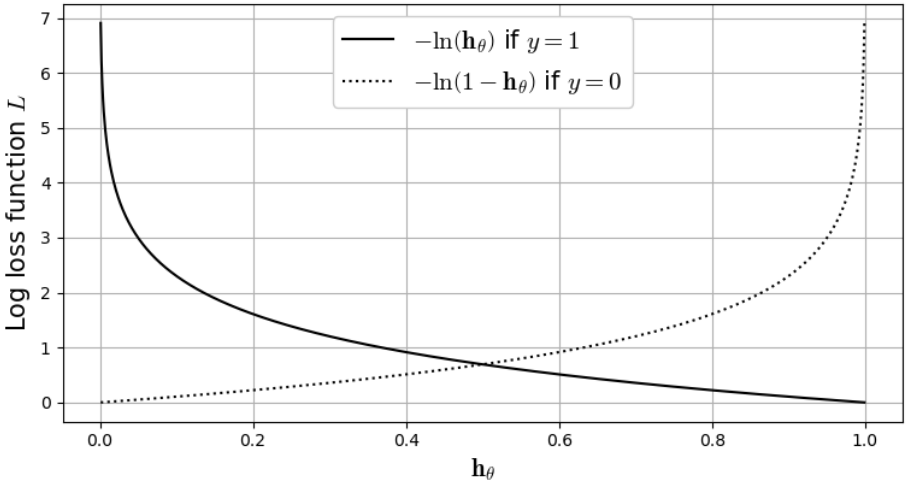

Figure 3.4: The *log loss* function for the two distinct cases of the label y. The loss approaches 0 if the prediction $\mathbf{h}_\theta$ corresponds to the label and grows to very large values otherwise.

Extending the above equation of the loss to the full data set we have:

$$L = -\frac{1}{M}\left(\mathbf{y}^T \ln\left[\sigma\left(\mathbf{X}\boldsymbol{\theta}\right)\right] + \left(\mathbf{1}-\mathbf{y}\right)^T \ln\left[\mathbf{1}-\sigma\left(\mathbf{X}\boldsymbol{\theta}\right)\right]\right) \tag{3.9}$$

3.5 Gradient descent

In order to apply the gradient descent method we have to determine the derivatives of the loss function w.r.t. to the weights. These derivatives are computed by:

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = \frac{1}{M}\mathbf{X}^T\left(\sigma\left(\mathbf{X}\boldsymbol{\theta}\right)-\mathbf{y}\right) \tag{3.10}$$

, which is similar to the derivatives we derived for linear regression.

To derive this equation we again take the component form and treat the two cases $y_m = 1$ and $y_m = 0$ separately:

$$\frac{\partial L}{\partial \theta_n} = \begin{cases} -\frac{1}{M}\sum_{m=0}^{M-1}\frac{\partial}{\partial\theta_n}\ln(h_m) = -\frac{1}{M}\sum_{m=0}^{M-1}\frac{1}{h_m}\frac{\partial h_m}{\partial\theta_n} & \text{for } y_m = 1 \\ -\frac{1}{M}\sum_{m=0}^{M-1}\frac{\partial}{\partial\theta_n}\ln(1-h_m) = -\frac{1}{M}\sum_{m=0}^{M-1}\frac{-1}{1-h_m}\frac{\partial h_m}{\partial\theta_n} & \text{for } y_m = 0 \end{cases} \tag{3.11}$$

, where we conveniently used h_m to abbreviate $h_\theta(\mathbf{x}^{(m)})$, the hypothesis of the m-th sample. To compute the derivative of h_m w.r.t. θ_m we make use of the known derivative of the sigmoid function:

$$\begin{aligned} \frac{\partial h_m}{\partial\theta_n} &= \frac{\partial\sigma(z_m)}{\partial\theta_n}\text{ , with } z_m = \sum_{n=0}^{N} X_{mn}\theta_n \\ &= \sigma(z_m)(1-\sigma(z_m))\frac{\partial z_m}{\partial\theta_n} \\ &= \sigma(z_m)(1-\sigma(z_m))\frac{\partial}{\partial\theta_n}\sum_{n=0}^{N} X_{mn}\theta_n \\ &= h_m(1-h_m)X_{mn} \end{aligned} \tag{3.12}$$

Substituting in the previous equations yields:

$$\frac{\partial L}{\partial \theta_n} = \begin{cases} -\frac{1}{M}\sum_{m=0}^{M-1}(1-h_m)X_{mn} & \text{for } y_m = 1 \\ -\frac{1}{M}\sum_{m=0}^{M-1} -h_m X_{mn} & \text{for } y_m = 0 \end{cases} \tag{3.13}$$

These expressions can be combined into a single equation as:

$$\begin{aligned} \frac{\partial L}{\partial \theta_n} &= \frac{1}{M}\sum_{m=0}^{M-1}(h_m - y_m)X_{mn} \\ &= \frac{1}{M}\sum_{m=0}^{M-1} X^T_{nm}(h_m - y_m) \end{aligned} \tag{3.14}$$

, which is the component form of equation 3.10

3.6 Feature scaling

Because the logistic or sigmoid function varies most for input values around zero (see fig 3.2) it is important that, when using gradient descent, the values of the different features are also in that range. If not, the gradient will be zero (or at least very small) and the optimization will take a very long time. Obviously this is not always the case and therefore it is common practice to scale the features. One approach is the following:

1. For each feature, determine the mean and standard deviation across the samples.
2. Scale all features by subtracting the mean and dividing by the standard deviation.

In this way all features will have a mean of zero and a standard deviation of 1.

3.7 Polynomial features

It is not always possible to classify correctly all data points using only a linear decision boundary. When we add another iris species (see figure 3.5) and we want to discriminate against that species as well, using a linear decision boundary will not suffice.

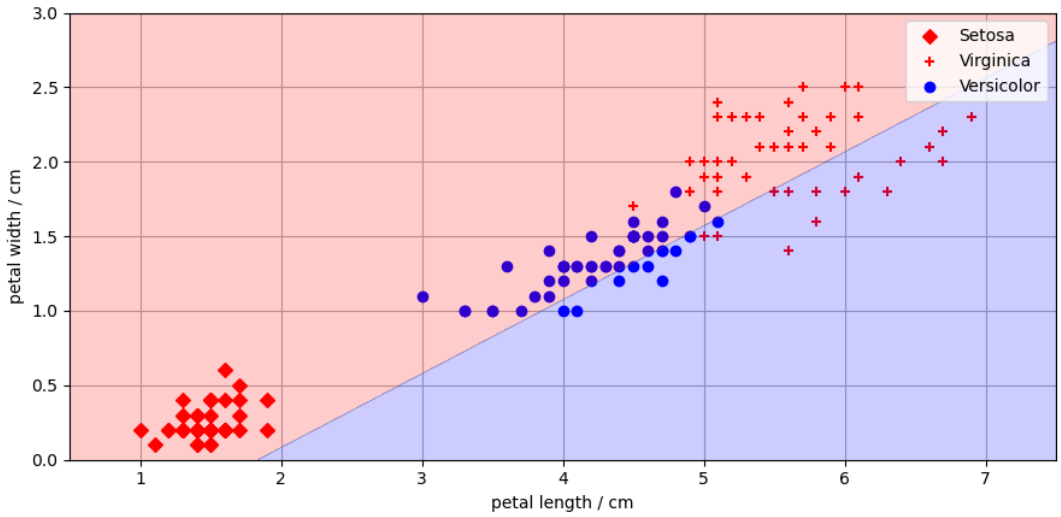

Figure 3.5: More complicated classification tasks cannot be done by using linear decision boundaries. In this example the *Versicolor* iris cannot successfully be discriminated against the others.

Just as with Linear regression, additional features can be added/computed. For example polynomial features can be added. If we have two original features (x_1 and x_2) than we can compute polynomial features as:

$$1, x_1, x_2, x_1^2, x_1x_2, x_2^2, x_1^3, x_1^2x_2, x_1x_2^2, x_2^3 \tag{3.15}$$

, where we listed all features up to a polynomial degree of 3 and included the bias (1) and the original features. In figure 3.6 these new features were applied and the result shows non-linear decision boundaries results in better classification. But it is also clear this new result suffers from over fitting. A sample with a petal length of 1 and a petal width of 0.7 would classify as a *Versicolor* which would in reality be very unlikely. To solve this problem a technique called *regularisation* can be applied.

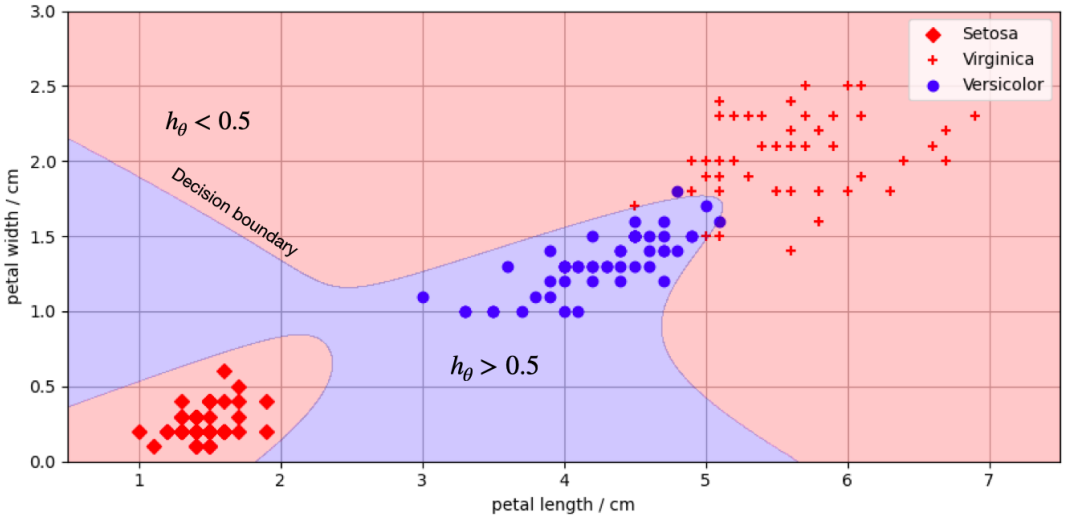

Figure 3.6: Adding polynomial features to the data allows non-linear decision boundaries. Here the polynomial degree is 2.

3.8 Regularisation

To prevent over fitting in logistic regression *regularisation* can be used [1]. The idea is to penalize weights that are large. This forces weights to be small and will in general result in more weights being so small that these features will have negligible contribution to the model. Less features will result in *simpler* models. Penalizing weights can be done by adding the squares of the weights to the loss function [2]. In order to adjust how much regularisation we want, a parameter $\lambda_{reg} > 0$ is used. Furthermore, the bias (x_0) and the corresponding weight θ_0 is not regularised. We add the following term to the loss:

$$\frac{\lambda_{reg}}{2M} \sum_{n=1}^{N} \theta_n^2 \tag{3.16}$$

, or in vector form:

$$\frac{\lambda_{reg}}{2M} \boldsymbol{\theta}^T \mathbf{A} \boldsymbol{\theta} \tag{3.17}$$

, with $\mathbf{A}_{N+1}$ the $(N+1) \times (N+1)$ identity matrix with the first element $A_{00} = 0$ in order to exclude the bias. The total loss function then becomes:

$$L = -\frac{1}{M} \left(\mathbf{y}^T \ln \left[\sigma \left(\mathbf{X}\boldsymbol{\theta} \right) \right] + \left(\mathbf{1} - \mathbf{y} \right)^T \left(\mathbf{1} - \ln \left[\sigma \left(\mathbf{X}\boldsymbol{\theta} \right) \right] \right) \right) + \frac{1}{2M} \lambda_{reg} \boldsymbol{\theta}^T \mathbf{A} \boldsymbol{\theta} \tag{3.18}$$

[1] The technique of regularisation is also applicable to many other regression techniques such as linear regression.

[2] Adding the squares of the weights is referred to as *ridge regression* and is just one of many possibilities. For example *lasso regressions* adds the absolute values of the weights to the loss function.

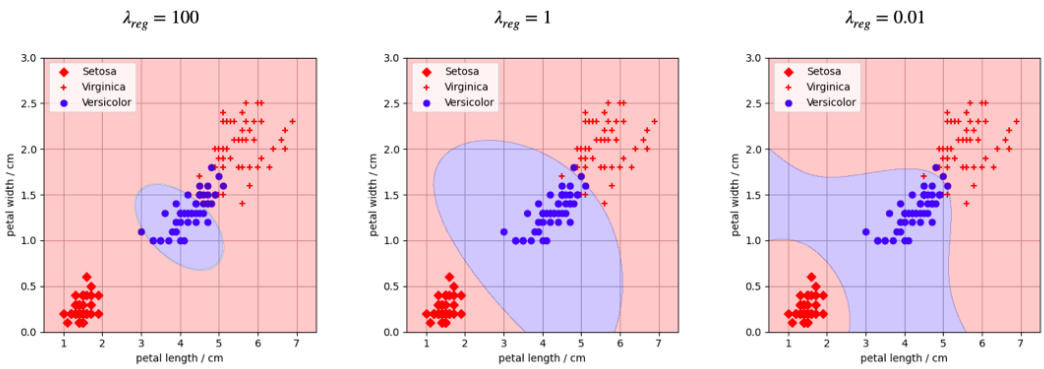

Figure 3.7: Effect of regularisation hyper-parameter choice on the decision boundary. Polynomial degree=3. Too high values result in under-fitting (left panel), and too small values results in over-fitting (right panel).

and it's derivative:

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = \frac{1}{M}\left(\mathbf{X}^T\left(\sigma\left(\mathbf{X}\boldsymbol{\theta}\right) - \mathbf{y}\right) + \lambda_{reg}\mathbf{A}\boldsymbol{\theta}\right) \tag{3.19}$$

In figure 3.7 the iris data set is modelled using three different values of λ_{reg}. For high values of λ_{reg} the decision boundary becomes "simpler" and is thus less prone to over fitting. However, as is apparent from the first panel of figure 3.7 if the hyperparameter λ_{reg} is chosen too large we get under fitting. Tuning the hyperparameters (e.g. λ_{reg} and the learning rate η) is one of the challenges of machine learning.

Chapter 4

Support Vector Machines

4.1 Introduction

Support vector machines (SVM) are used for classification. In figure 4.1 the concept of a SVM is illustrated. Two categories labeled -1 and 1 are separated with a decision boundary. In the left and right panel two possible linear decision boundaries are drawn that both correctly separate the two categories. However, in the left panel, the *margin* (the orthogonal distance between the decision boundary and the closest data point) is relatively small. In the right panel the boundary is chosen such that the *margin* is maximised. The strategy of an SVM, is to choose a decision boundary such that the *margin* is as large as possible. Obviously there are always 2 or more data points that limit the size of the margin. These data points (represented by feature vectors $\mathbf{x}$) are referred to as the *support vectors*. In the right panel of figure 4.1 the support vectors are indicated with yellow. It is important to realize that only these (in this case three) datapoints determine the decision boundary.

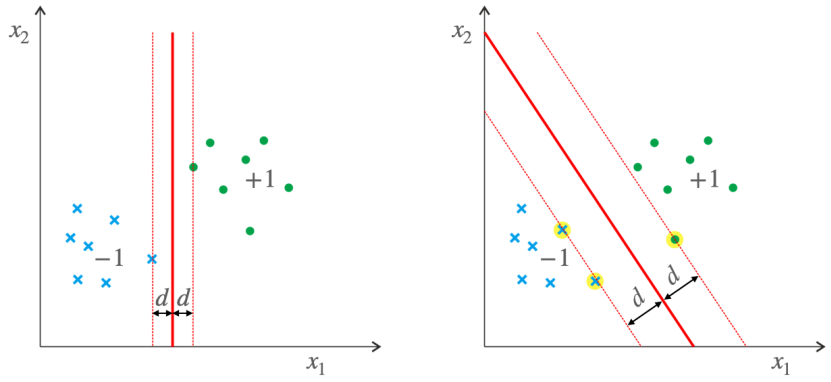

Figure 4.1: The concept of a SVM is to set the *margin* between the decision boundary (solid red) and the data points as large as possible. In the left panel this margin is relatively small, whereas in the right panel the margin is maximized. The data points that limit the *margin* are the *support vectors* and are indicated in yellow.

4.2 Formulation of the optimizing problem

We start by labeling the data with -1 and 1 to differentiate between the two categories:

$$y^{(m)} \in \{-1, 1\} \tag{4.1}$$

This choice (over the labels 0 and 1) is for convenience only as will become clear later. In case of a linear separable dataset (i.e. a linear decision boundary can correctly classify the data) we can classify the data

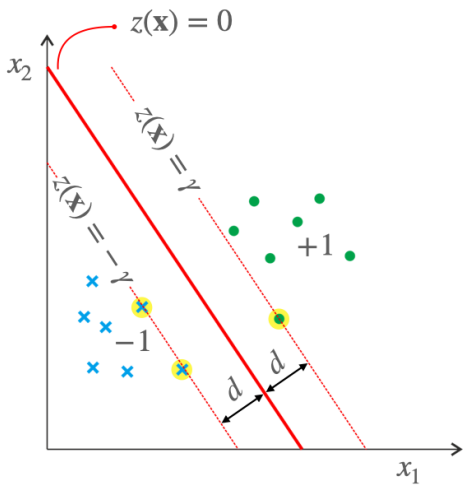

Figure 4.2: Two categories are classified by a linear decision boundary (hyperplane) $\mathbf{z}(\mathbf{x}) = 0$. The two additional hyperplanes $\mathbf{z}(\mathbf{x}) = \pm\gamma$ are parallel to the decision boundary and define the maximum *margin* as limited by the support vectors (indicated in yellow)

by computing (identical to logistic regression) on which side of the decision boundary the datapoint is located. Therefore we start by writing down the expression for the decision boundary:

$$z(\mathbf{x}) = 0 \tag{4.2}$$

where $z(\mathbf{x})$ is defined as a linear combination of the features:

$$z(\mathbf{x}) = \sum_{n=0}^{N} \theta_n x_n = \sum_{n=1}^{N} \theta_n x_n + \theta_0 x_0 = \sum_{n=1}^{N} \theta_n x_n + \theta_0 \tag{4.3}$$

where the form is such that the *bias* θ_0 is made more explicit. Furthermore we adopt the general convention of $\mathbf{w}$ for the weights vector and b for the bias and define the feature vector $\mathbf{x}$ from now on such that it does not include the bias $x_0 = 1$. This makes the decision boundary expressed as:

$$z(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0 \tag{4.4}$$

To further illustrate this we take the example in figure 4.1 where there are only two features x_1 and x_2. Therefore the decision boundary in this case can be written as:

$$w_1 x_1 + w_2 x_2 + b = 0 \tag{4.5}$$

This is an equation of a line (i.e. the decision boundary). If we assume that we have found the correct weights (such that both the data points are correctly classified and that the margin is as large as possible) than in order to predict the category based on some value of x_1 and x_2 we just need to compute on which side of the decision boundary the datapoint lies. In our example this is mathematically formulated as:

$$\begin{cases} \hat{y} = +1 & \text{if } w_1 x_1 + w_2 x_2 + b \geq 0 \\ \hat{y} = -1 & \text{if } w_1 x_1 + w_2 x_2 + b < 0 \end{cases} \tag{4.6}$$

Generalizing this to an arbitrary decision boundary (which in general a *hyperplane*) gives:

$$\begin{cases} \hat{y}^{(m)} = +1 & \text{if } z(\mathbf{x}^{(m)}) \geq 0 \\ \hat{y}^{(m)} = -1 & \text{if } z(\mathbf{x}^{(m)}) < 0 \end{cases} \tag{4.7}$$

The next step is to find a strategy that determines the weights $\mathbf{w}$ vector and the bias b that maximizes the margin (and of course correctly classifies the data). Obviously we require a loss function so we can optimize the weights such that the loss in minimal. As the concept is to make the *margin* (i.e. the orthogonal distance between the decision boundary and the support vectors) as large as possible we start by computing this distance. In figure 4.2 two additional lines (hyperplanes in general) parallel to the

decision boundary are shown. These lines are positioned such that they just go through the support vectors and have a distance d from the decision boundary and are defined as:

$$z(\mathbf{x}) = \pm\gamma \tag{4.8}$$

which we can write as (making use of $y \in \{-1, 1\}$):

$$y\left(\mathbf{w}^T\mathbf{x} + b\right) = \gamma \tag{4.9}$$

, where the value of γ is fully determined by the value of $y\left(\mathbf{w}^T\mathbf{x} + b\right)$ for the data point that is the closest to the decision boundary. So equation 4.9 can be rephrased as:

$$\gamma = \min_m \left\{ y^{(m)} \left(\mathbf{w}^T\mathbf{x}^{(m)} + b \right) \right\} \tag{4.10}$$

The target for a SVM is to find the values for the weights $\mathbf{w}$ and bias b that maximizes the distance d, while correctly classify all data points. The distance between the two hyperplanes defined by equation 4.8 equals:

$$2d = \frac{2\gamma}{\|\mathbf{w}\|} \tag{4.11}$$

And therefore we can formulate the optimization task of the SVM as:

$$\begin{aligned} &\max_{\mathbf{w},b} \left\{ \frac{1}{\|\mathbf{w}\|} \min_m \left\{ y^{(m)} \left(\mathbf{w}^T\mathbf{x}^{(m)} + b \right) \right\} \right\} \\ &\text{subject to } y^{(m)} \left(\mathbf{w}^T\mathbf{x}^{(m)} + b \right) \geq 0 \text{ for } m = 0, 1, 2, \cdots, M-1 \end{aligned} \tag{4.12}$$

Where a constraint is added to enforce all data points to be correctly classified (i.e. on the correct side of the decision boundary).

4.3 Solving

To solve this optimization problem different approaches can be used. The commonly used strategy relies on *quadratic programming*. We take a look at two other methods. First we discuss a *genetic algorithm*.

4.3.1 Genetic algorithm

The objective is to find parameters (the weights and the bias) that maximizes equation **??**. One choice of parameters is referred to as an *individual*, and each individual is composed of *genes* (i.e. the individual parameters $w_1, w_2, w_3, .., b$). In a genetic algorithm you start by randomly creating a set of (say $K = 100$) *individuals*. This set is called a *generation*. The next step involves the evaluation of the objective for each *individual* of the *generation*. Some individuals will perform better than others. Based on the performance a next generation (*offspring*) can be computed. To do this we *cross* individuals and we introduce *mutations*. Two individuals are crossed by randomly taking some genes from the first, and the others from the second individual. This makes a new individual that inherits genes from both "parents". In order to introduce new genes we allow genes of an individuals to mutate. This can be done by randomly selecting some genes form an individual and replace them with a random value. Iterating over generations will converge to the best weights and bias to maximize the objective.
There are many parameters to tweak in this algorithm. You can choose to what extent genes of the *offspring* are inherited from their parents and to what extent you allow mutations to take place. A possibility is to split a generation in parts based on their performance and then treat these parts differently. The best 25% of the generation can be left unchanged, the second 25% can be crossed and mutated, the third 25% can be crossed by random new individuals and the final 25% can be completely new random individuals.

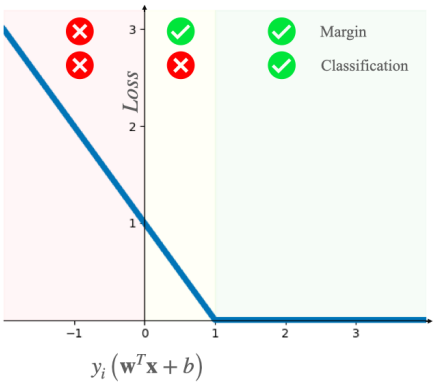

Figure 4.3: The *hinge loss* loss function. When a data point is correctly classified and does not violate the margin, the loss equals zero. The loss increases linearly if the data point is on the wrong side of the margin hyperplane.

4.3.2 Gradient descent

Often the optimization problem is rephrased slightly different. If we (for now) demand that all data points are classified correctly (*hard margin condition*) we have the constraint:

$$y^{(m)}(\mathbf{w}^T\mathbf{x}^{(m)} + b) \geq \gamma \tag{4.13}$$

But we can choose any value for γ we want. If we have a closest data point $\mathbf{x}$ (the support vector) we know that this point is on (one of) the hyperplanes that define the *margin* and thus $y^{(m)}(\mathbf{w}^T\mathbf{x}^{(m)}+b) = \gamma$. But because this equation is linear we can multiply this with any constant (say β) and the relation still holds: $y^{(m)}(\mathbf{w}^T\mathbf{x}^{(m)} + b) = \beta\gamma$ if the values of the weights and biases are just scaled with β in this case. A common choice for the value of γ is 1. Then we have:

$$\max_{\mathbf{w}} \left\{ \frac{1}{\|\mathbf{w}\|} \right\}, \text{ subject to: } y^{(m)}(\mathbf{w}^T\mathbf{x}^{(m)} + b) \geq 1 \tag{4.14}$$

But maximizing $\|\mathbf{w}\|^{-1}$ is more conveniently written as minimizing the square of the norm of the weight vector:

$$\min_{\mathbf{w}} \frac{1}{2}\|\mathbf{w}\|^2, \text{ subject to: } y^{(m)}(\mathbf{w}^T\mathbf{x}^{(m)} + b) \geq 1 \tag{4.15}$$

It is important to realize that so far we assumed that there is always a decision boundary possible that correctly classifies all data points. But this is not always possible. Also results can be poor in the presence of outliers because if there is a hard criteria that *all* data points are outside the margins, an outlier can result in very small margins to satisfy this criteria. To ease that condition we can allow some points to violate the margins or even to be misclassified by introducing a penalty to the minimization target. To do this we introduce the *hinge loss* loss function. This function is shown in figure 4.4. The loss equals 0 if the point does not violate the margin $y^{(m)}(\mathbf{w}^T\mathbf{x}^{(m)} + b) \geq 1$) and increases linearly for smaller values. The loss can be written as:

$$L = \frac{1}{2}\|\mathbf{w}\|^2 + C\frac{1}{M}\sum_{m=0}^{M-1} \max\left(0, 1 - y^{(m)}(\mathbf{w}^T\mathbf{x}^{(m)} + b)\right) \tag{4.16}$$

There are now two conflicting targets. On the one hand the margin needs to be as wide as possible, on the other hand the number of misclassified points should be as small as possible. With the hyperparameter C we can set the balance between the two. If C is very large, margin violations are penalized strongly and thus we have a *hard margin* SVM. Choosing smaller and smaller values of C we allow more and more margin violations and even misclassifications resulting in a *soft margin* SVM.

To compute the gradient we have to distinguish between two cases:

$$\frac{\partial L}{\partial \mathbf{w}} = \begin{cases} \mathbf{w} - C\frac{1}{M}\sum_{m=0}^{M-1} y^{(m)}\mathbf{x}^{(m)} & \text{if } y^{(m)}(\mathbf{w}^T\mathbf{x}^{(m)} + b) < 1 \\ \mathbf{w} & \text{if } y^{(m)}(\mathbf{w}^T\mathbf{x}^{(m)} + b) \geq 1 \end{cases} \tag{4.17}$$

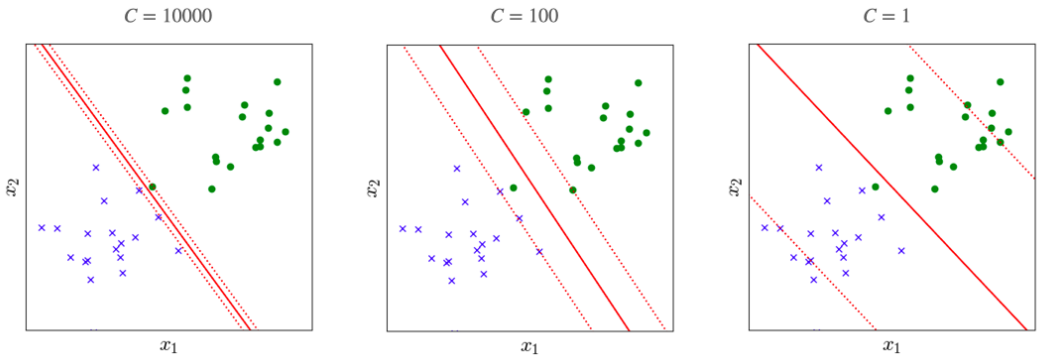

Figure 4.4: Tuning the balance between *hard margin* and *soft margin* using the hyperparameter C. A large value of C results in a *hard margin*, while smaller values allows the margin to be wider at the expense of margin violations. Note that although for the case $C = 1$ many data points are violating the margins, there is only one data point that is misclassified (i.e. is on the wrong side of the decision boundary.)

$$\frac{\partial L}{\partial b} = \begin{cases} -C\frac{1}{M}\sum_{m=0}^{M-1} y^{(m)} & \text{if } y^{(m)}(\mathbf{w}^T\mathbf{x}^{(m)} + b) < 1 \\ 0 & \text{if } y^{(m)}(\mathbf{w}^T\mathbf{x}^{(m)} + b) \geq 1 \end{cases} \tag{4.18}$$

Using these gradients we can apply the (stochastic) gradient descent method.

4.4 Kernels

Often the data is not linearly separable (see e.g. the left panel in figure 4.5). That means that it is not possible to properly classify the data by a hyperplane. To solve that we can compute different features using the existing ones as we have seen before. Let's say we have a feature vector:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{4.19}$$

We can map this into a different feature *space* by e.g. computing all polynomial terms of degree 2 using a function $\phi(\mathbf{x})$:

$$\phi(\mathbf{x}) = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix} \tag{4.20}$$

Inspecting the right panel of figure 4.5 learns that applying this feature mapping on the example data does allow to separate the data with a plane.

A kernel does exactly this, but has a special feature. The inner product of two vectors $\phi(\mathbf{x}^1)$ and $\phi(\mathbf{x}^2)$ can be expressed in terms of the inner product of the vectors in the original feature space. Using the above feature mapping we get e.g.:

$$\begin{aligned}\phi(\mathbf{x}^{(1)}) \cdot \phi(\mathbf{x}^{(2)}) &= \begin{bmatrix} (x_1^{(1)})^2 \\ \sqrt{2}x_1^{(1)}x_2^{(1)} \\ (x_2^{(1)})^2 \end{bmatrix} \cdot \begin{bmatrix} (x_1^{(2)})^2 \\ \sqrt{2}x_1^{(2)}x_2^{(2)} \\ (x_2^{(2)})^2 \end{bmatrix} \\ &= (x_1^{(1)}x_1^{(2)})^2 + 2(x_1^{(1)}x_1^{(2)})(x_2^{(1)}x_2^{(2)}) + (x_2^{(1)}x_2^{(2)})^2 \\ &= (x_1^{(1)}x_1^{(2)} + x_2^{(1)}x_2^{(2)})^2 \\ &= (\mathbf{x}^{(1)} \cdot \mathbf{x}^{(2)})^2\end{aligned} \tag{4.21}$$

So we have a kernel for the second degree polynomial $\mathbf{K}$:

$$\mathbf{K}\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \left((\mathbf{x}^{(i)})^T \mathbf{x}^{(j)}\right)^2 \tag{4.22}$$

Why is this useful? It appears that, for an SVM, it is possible to express the loss function in terms of inner products of the feature vectors. This means that we can use feature mapping without ever computing the features and makes this technique very computational efficient[1].

Different type of kernels are available. The polynomial was already shown above, another widely used is the *Gaussian* kernel.

In a Gaussian kernel (or radial basis function, rbf) you compute (well actually you don't if you use the kernel trick) the following feature vector:

$$\phi(\mathbf{x}^{(i)}) = \begin{bmatrix} \exp\left(-\gamma \left\|\mathbf{x}^{(i)} - \mathbf{x}^{(1)}\right\|^2\right) \\ \exp\left(-\gamma \left\|\mathbf{x}^{(i)} - \mathbf{x}^{(2)}\right\|^2\right) \\ \cdot \\ 1 \\ \cdot \\ \exp\left(-\gamma \left\|\mathbf{x}^{(i)} - \mathbf{x}^{(M)}\right\|^2\right) \end{bmatrix} \tag{4.23}$$

The items in this feature vector $\mathbf{x}^{(i)}$ are a measure of the Euclidean distance of that data point to all other data points. The (Gaussian) function exp() equals zero if the distance is large, and equals one if this distance is zero. Because the distance of the data point to itself equals zero, the i-th element of the feature vector equals one.

When trained with a Gaussian kernel, classification of new data points are based on the distance to the support vectors. If the *hyperparameter* γ is small the value of exp() drops slowly with increasing distance, whereas if γ is large, the value of exp() drops very quickly. This means that taking γ too large can result in over fitting, and taking γ too small can result in under fitting (see figure 4.6).

In total there are M data points, so the length of the feature vector is also M. With large data sets this can easily become very computational expensive. Fortunately this feature mapping has a kernel function that defines the inner product in terms of the original features:

$$\mathbf{K}\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \exp\left(-\gamma \left\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\right\|^2\right) \tag{4.24}$$

In order to apply the kernels you could use techniques like gradient descent or use a genetic algorithm. This is possible if the number of features that you add remains limited. However, computing the new features using a Gaussian kernel quickly becomes computational heavy as the number of features is equal the number of datapoints. Therefore usually a different method is used in SVM implementations. This method makes use of the *kernel trick* that makes it much more efficient. A full explanation is beyond the scope of these notes.

[1] This is called the *kernel trick*

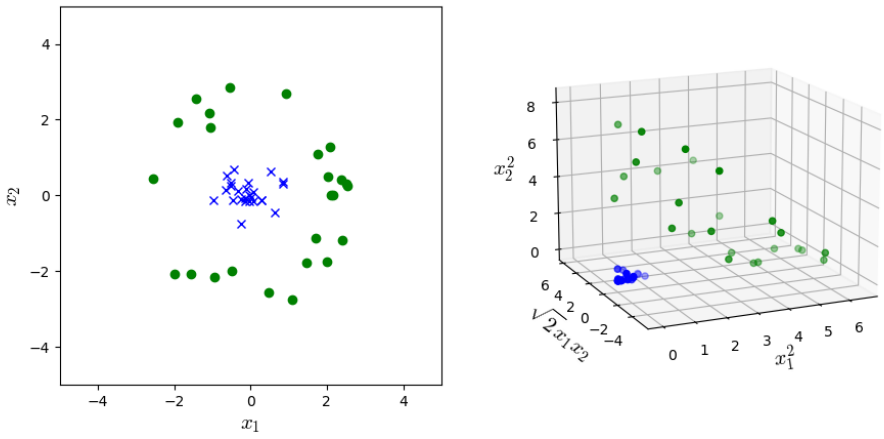

Figure 4.5: Left: the categories are not linearly separable using features x_1 and x_2. Right: computing new features, in this case second-order polynomials, maps the data in a 3D feature space in which the two categories are separable with a 2D hyperplane.

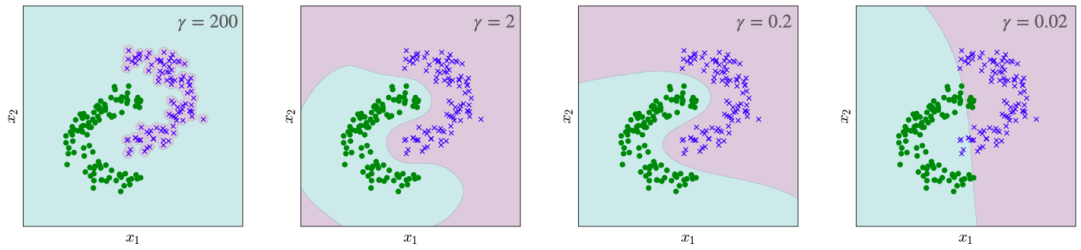

Figure 4.6: Performance of the Gaussian kernel for different *hyperparameter* γ settings. Large values result in over fitting, small values in under fitting.

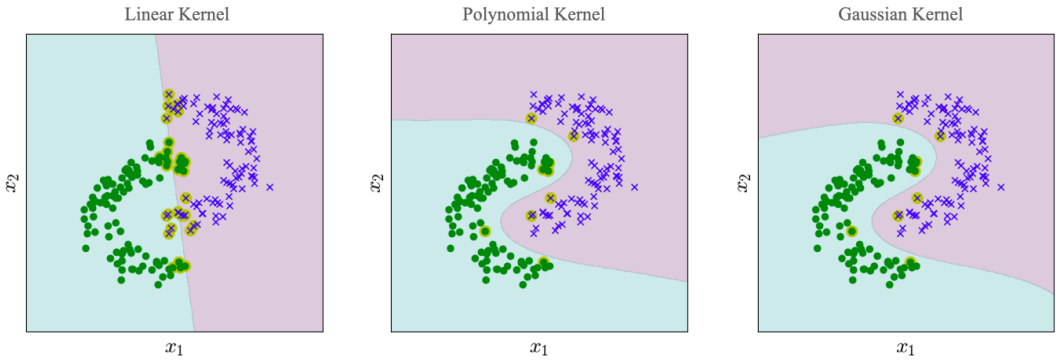

Figure 4.7: Different kernels applied to a data set. Left panel: a linear kernel does not successfully classify this data set. Middle panel: a third degree polynomial kernel allows for much better classification. Right panel: the Gaussian kernel result. Support vectors are indicated with yellow.

Chapter 5

K-means

Kmeans belongs to the family of *unsupervised learning*. It is an algorithm that tries to cluster data using only knowledge on the features and no knowledge on the labels. So far we have used only supervised training. For each feature vector a label is known (e.g. to which category the data point belongs) and this label is used during the learning. Sometimes however there is no label available and supervised learning is not applicable. One example is clustering. In unsupervised clustering learning we try to find some *structure* in the data without any prior knowledge.

As usual we have M data points $\mathbf{x}^{(m)}$ of N features[1]. Next, we define K number of clusters. We can choose K from 1 (just one cluster containing all data points) up to M (each cluster contains just 1 data point). Each cluster S_k contains the indices of the data points that belong to that cluster. E.g. if cluster S_0 contains data point $m = 0$, $m = 4$ and $m = 5$ and cluster S_1 has data points $m = 1$, $m = 2$ and $m = 3$, we have:

$$S_0 = \{0, 4, 5\}$$
$$S_1 = \{1, 2, 3\}$$

In order to determine what the best distribution of the data points over the clusters is we need a loss function. The loss function that is used by Kmeans is quite intuitive. It is the sum of all distances of the data points to the center $\mathbf{c}^{(k)}$ of the cluster it belongs to:

$$L = \sum_{k=0}^{K-1} \sum_{m \in S_k} \left\| \mathbf{x}^{(m)} - \mathbf{c}^{(k)} \right\|^2 \tag{5.1}$$

, where the center of the cluster is computed as the center-of-mass of all data points that belong to the cluster.

To minimize the loss function we need to know (i) the centers of the clusters $\mathbf{c}^{(k)}$ and (ii) which data point belongs to which cluster S_k. To find $\mathbf{c}^{(k)}$ and S_k that minimizes the loss we use an iterative algorithm:

[1] In Kmeans we do not include a *bias* in the feature vector $\mathbf{x}^{(m)}$

1. **Initialization.** Determine K starting centers $\mathbf{c}^{(0)}, \mathbf{c}^{(1)}, .., \mathbf{c}^{(k)}, ..\mathbf{c}^{(K-1)}$. E.g take K random (but unique) data points as the starting centers.

2. **Assignment.** Assign each data point $\mathbf{x}^{(m)}$ to one of the K clusters. This is done by choosing the cluster for which the distance to the center is the smallest:

$$k = \text{argmin} \sum_{k=0}^{K-1} \left\| \mathbf{x}^{(m)} - \mathbf{c}^{(k)} \right\| \tag{5.2}$$

3. **Updating.** Next update the cluster centers by computing the center of mass of the cluster.

$$\mathbf{c}^{(k)} = \frac{1}{\|S_k\|} \sum_{m \in S_k} \mathbf{x}^{(m)} \tag{5.3}$$

4. **Repeat.** Repeat step 2 and 3 until the clusters do not change anymore, i.e. a convergence criterion is reached.

As an example this algorithm is applied on the iris dataset that was used in logistic regression. The process is illustrated in figure 5.1

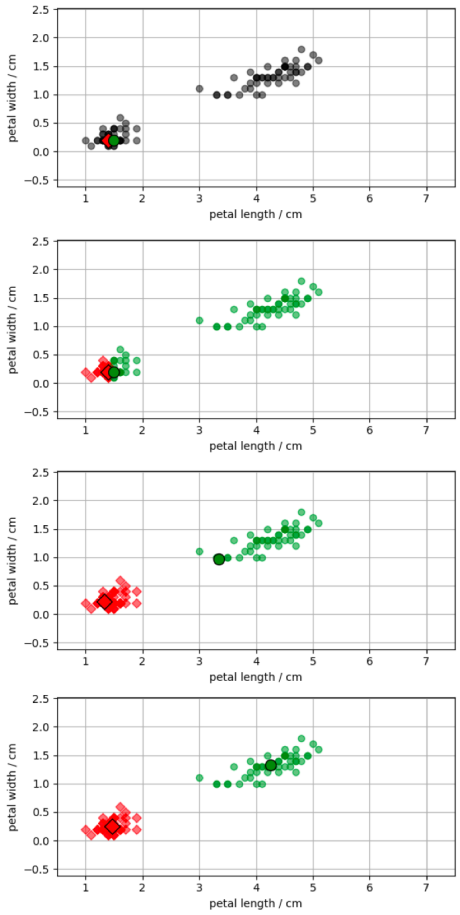

Figure 5.1: The Kmeans algorithm visualised. (Top panel) randomly choose (in this case 2) data points as centers for the two clusters (indicated with a red diamond and a blue circle). (Second panel) determine for each point to which cluster it belongs based on the distance to the centers. (Third panel) compute new centers as the center of mass of all data points in the cluster and re-assign the datapoints to the clusters. (Bottom panel) repeat the procedure until the centers do not "move" anymore.

Clearly clustering is not limited to data with only 2 features. Although it becomes a bit harder to visualize, the algorithm is identical for higher dimensions. In figure 5.2 an example is given for 3 features. In this particular example feature 1 and feature 2 are not sufficient to discriminate the red and green

clusters as they overlap if you project the data points on the plane determined by $x_3 = 0$. However, feature 3 clearly separates the green and red clusters in the x_3 direction/dimension.

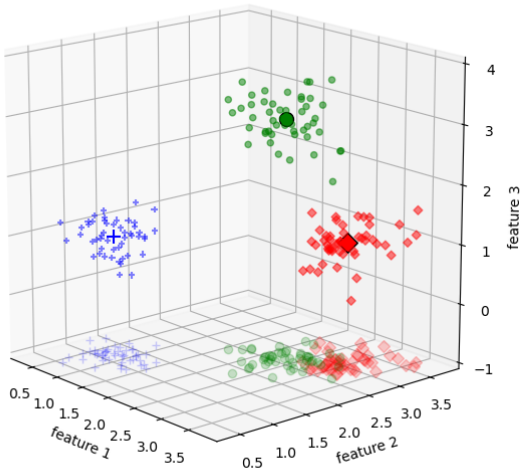

Figure 5.2: Example of Kmeans clustering using three features $N = 3$ and three clusters $K = 3$. Using only feature 1 and feature 2 it would not be possible to separate the green and red clusters as they overlap (visualized here by a 2D projection of the data on the plane $z = -1$).

It is important to realize that the Kmeans algorithm can be sensitive to the choice of initial cluster centers. In figure 5.3 the same clustering is done, but with other initial center positions. The clusters are wrongly assigned. It is therefore wise to try (many) initial center locations and select the one(s) that result in the lowest loss.

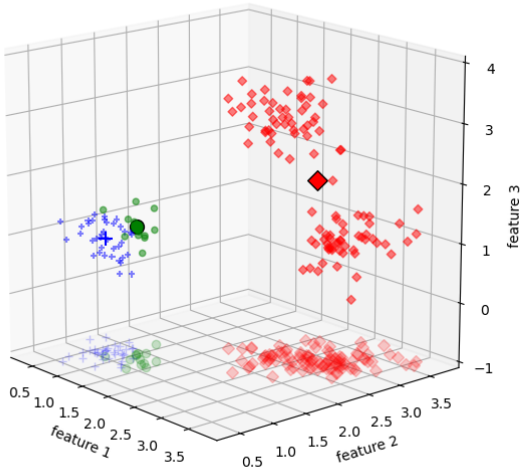

Figure 5.3: Kmeans can be sensitive to initial cluster centers. This figure shows that the algorithm does not result in the proper clusters.

Finally some thoughts on the choice of the number of clusters K. As it is unsupervised learning, often the number of clusters in not known. One approach is to try different values for K and plot the loss as a function of K. If the loss does not significantly changes anymore with increasing K you have reached some optimum value for K (see figure 5.4).

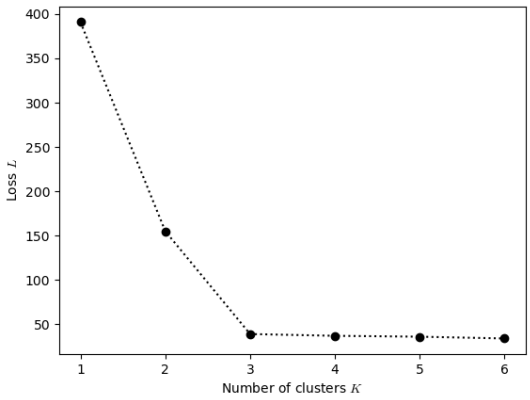

Figure 5.4: Plotting the obtained loss as function of the number of clusters can give insight in the optimal number of clusters. Here the loss for the data of figure5.2 is analysed. The optimal number of clusters is $K = 3$ as for larger number of clusters the loss does not significantly improve.

Chapter 6

Neural Networks

6.1 Introduction

Watch the following Youtube video of *3Blue1Brown* on neural networks to get the general idea of how a neural network works (There are 4 episodes, watch at least the first episode):

`https://www.youtube.com/watch?v=aircAruvnKk`

6.2 Definitions

As you have seen in the video a NN consist of layers in which each layer consists of one or more neurons. Furthermore layers are connected through weights and biases. In order to mathematically formulate the network and to be able to implement it, it is important to clearly define the different parameters etc.

The layer index is denoted with l. Our NN has an input layer ($l = 0$), an output layer ($l = L$) and hidden layers ($1 \leq l \leq L - 1$).

For each layer l we have:

- A number of nodes: n_l
- An output (column) vector: $\mathbf{a}^l \in \mathbb{R}^{n_l \times 1}$
- A weight matrix: $\mathbf{W}^l \in \mathbb{R}^{n_l \times n_{l-1}}$
- A bias (column) vector: $\mathbf{b}^l \in \mathbb{R}^{n_l \times 1}$
- An activation function: σ^l

The input layer actually does not do much, it just passes the sample data to the output which then acts as input for the first hidden layer. So the input layer does not have a weight, bias and activation function.

6.3 Forward Pass

When the data is fed into the NN the aim in the forward pass is to compute the output of the NN given all the weights and biases of the NN. If we feed one sample $\mathbf{x}$ to the NN (later we will expand this in Python to compute the output of the NN for multiple samples simultaneously), then for the input layer ($l = 0$) we have as output:

$$\mathbf{a}^0 = \mathbf{x} \tag{6.1}$$

, with $\mathbf{x} \in \mathbb{R}^{n_0 \times 1}$ a column vector containing the features of the sample.

For the hidden layers and the output layer ($1 \leq l \leq L$) we have:

$$\begin{aligned} \mathbf{z}^l &= \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l \\ \mathbf{a}^l &= \sigma^l\left(\mathbf{z}^l\right) \end{aligned} \tag{6.2}$$

In other words, we multiply the output of the previous layer with the weight matrix, add the bias and feed that to the activation function to obtain the output. Note that we defined an intermediate variable $\mathbf{z}^l$, which will become useful in back propagation step.

The above equations can also be written in component form as:

$$\begin{aligned} z_i^l &= \sum_{j=1}^{n_{l-1}} W_{ij}^l a_j^{l-1} + b_i^l \\ a_i^l &= \sigma^l\left(z_i^l\right) \text{ see Note 1} \end{aligned} \tag{6.3}$$

Note 1 This component wise form does not hold for the softmax activation function (see next section) as for this activation function the output of one neuron depends on the inputs of *all* neurons of the layer. In that case we have $a_i^l = \sigma^l\left(z_1^l, z_2^l, .., z_i^l, .., z_{n_l}^l\right)$

6.4 Activation functions

There are a few commonly used activation functions:

$$\begin{aligned} \text{Sigmoid} : \sigma_i(\mathbf{z}) &= \frac{1}{1+\exp(-\mathbf{z}_i)} \\ \text{ReLu} : \sigma_i(\mathbf{z}) &= \max(0, \mathbf{z}_i) \\ \text{Tanh} : \sigma_i(\mathbf{z}) &= \tanh(\mathbf{z}_i) \\ \text{Softmax} : \sigma_i(\mathbf{z}) &= \frac{\exp(\mathbf{z}_i)}{\sum_{i=1}^{n_L} \exp(\mathbf{z}_i)} \end{aligned} \tag{6.4}$$

The first three activation functions are computed for each node separately (they depend only on the input of the node). The shape is shown in the figure below.

The softmax activation function is often used in the output layer of a NN in classification tasks. The function computes a normalized output such that the values are between 0 and 1 and the sum is equal to unity. Therefore this can be regarded as a probability. The i-th component of the output is large when the sample belongs to the i-th category. The physicists and chemist among you might recognise similarity with the Boltzmann distribution: $p_i = e^{-\epsilon_i/k_B T} / \sum_i e^{-\epsilon_i/k_B T}$

6.5 The data labels: one-hot encoding

When using the NN for classification (e.g. using a softmax activation function in the output layer of the NN) the labels of the data are usually represented as a vector as a so-called *one-hot-encoding* vector. In the example of handwritten digit classification for example there are 10 categories that need to be distinguished. Now you could try to use a scalar label $y \in 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ in combination with a single neuron in the output layer. It appears that this does not work very well. A better choice is to have 10 neurons in the output layer in which each neuron corresponds to one of the 10 categories. To compare the NN output $\mathbf{a}^L$ to the label it is convenient to use the *one-hot-encoding* format. In this format all components of the vector are zero except at the index corresponding to the label. So the label for the digit 0 is written as $\mathbf{y} = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]^T$ and the label for digit 1 as $\mathbf{y} = [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]^T$, etc.

6.6 Loss function

For training of the NN we have to define a loss (or cost) function L that we need to minimize. Clearly we need to feed (many) different samples to the NN during the training. The different samples will be

labeled with m as $(\mathbf{x}^{(m)}, \mathbf{y}^{(m)})$, for $1 \leq m \leq M$. The (scalar) value of this function depends on the labels $\mathbf{y}^{(m)}$ and the output of the NN $\mathbf{a}^L(\mathbf{x}^{(m)})$, where the output of the NN is written explicitly as a function of the input sample $\mathbf{x}_m$. To actually compute $\mathbf{a}^L(\mathbf{x}^{(m)})$ you have to propagate $\mathbf{x}^{(m)}$ through the whole network. There are many options on how we can define the loss function. One example is the quadratic loss function:

$$L_Q\left(\mathbf{W}^1, \mathbf{W}^2, ..\mathbf{W}^L, \mathbf{b}^1, \mathbf{b}^2, ..\mathbf{b}^L\right) = \frac{1}{M}\sum_{m=1}^{M}\left\|\mathbf{a}^L(\mathbf{x}^{(m)}) - \mathbf{y}^{(m)}\right\|^2 \tag{6.5}$$

Since the data is fixed, the loss function only depends on the weights and biases. Finding the proper values for all weights and biases such that the loss function is minimized is referred to as *training*.

Another widely used loss function, and the one we will use, is the cross-entropy loss function. It is defined as:

$$L_{CE}\left(\mathbf{W}^1, \mathbf{W}^2, ..\mathbf{W}^L, \mathbf{b}^1, \mathbf{b}^2, ..\mathbf{b}^L\right) = -\frac{1}{M}\sum_{m=1}^{M}\mathbf{y}^{(m)} \cdot \ln \mathbf{a}^L(\mathbf{x}^{(m)}) \tag{6.6}$$

The cross entropy loss is usually combined with a softmax activation function in the output layer.

As we will see, there is no need to actually compute this loss function in training the network.

6.7 Gradient descent

The goal of training is to minimize the loss (or cost) function. In the stochastic gradient approach each parameter (all weights and biases) needs to be updated using:

$$\begin{aligned} W_{ij}^l &\to W_{ij}^l - \eta \sum_{batch} \frac{\partial L}{\partial W_{ij}^l} \\ b_i^l &\to b_i^l - \eta \sum_{batch} \frac{\partial L}{\partial b_i^l} \end{aligned} \tag{6.7}$$

, where the parameter η is a (small) constant referred to as the *learning rate*. In order to update the parameters, we repeatedly take a (small) batch of the total training data set. During training, we can take each batch randomly at each iteration, or alternatively, we divide the training data set in a number of fixed batches before training starts.

6.8 Back propagation

To compute the derivatives of the loss function with respect to all the parameters we make use of the *chain rule*. This gives the following 4 equations (a derivation is given afterwards):

$$\frac{\partial L}{\partial \mathbf{z}^L} = \begin{cases} \frac{\mathrm{d}\sigma^L}{\mathrm{d}\mathbf{z}^L} \circ \left(\mathbf{a}^L - \mathbf{y}\right) & \text{see Note 2} \\ \\ \mathbf{a}^L - \mathbf{y} & \text{see Note 3} \end{cases} \tag{6.8}$$

$$\frac{\partial L}{\partial \mathbf{z}^l} = \frac{\mathrm{d}\sigma^l}{\mathrm{d}\mathbf{z}^l} \circ (\mathbf{W}^{l+1})^T \frac{\partial L}{\partial \mathbf{z}^{l+1}} \qquad \text{for } 1 \leq l \leq L-1 \tag{6.9}$$

$$\frac{\partial L}{\partial \mathbf{W}^l} = \frac{\partial L}{\partial \mathbf{z}^l}(\mathbf{a}^{l-1})^T \qquad \text{for } 1 \leq l \leq L \tag{6.10}$$

$$\frac{\partial L}{\partial \mathbf{b}^l} = \frac{\partial L}{\partial \mathbf{z}^l} \qquad \text{for } 1 \leq l \leq L \tag{6.11}$$

Note 2 This expression holds for a quadratic loss function and any activation function for which the output of a neuron *only* depends on the input for that neuron. Therefore this does not hold for the softmax activation function.

Note 3 This expression holds for a cross-entropy loss function in combination with a softmax activation function.

In the above expression the symbol $\circ$ is used to indicate the Hadamard product i.e. the component wise product. To evaluate these expressions, we first compute all derivatives $\partial L/\partial \mathbf{z}^l$. To do this we first need to compute the derivative of the last (output) layer $\partial L/\partial \mathbf{z}^L$ using the first expression. With that result we can compute the derivative for the one-but-last layer using the second expression. Repeated use of the second expression allows to compute the derivatives all the way upto alyer $l = 1$. Because we progress from the last layer back to the first layer, this technique is called *back propagation.*

Equation 1 sub a

The derivative of the loss L with respect to the input $\mathbf{z}^L$ of the activation function of the last layer. For this we write the derivative in component form and work out the derivative using the chain rule:

$$\frac{\partial L}{\partial z_i^L} = \frac{\partial L}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_i^L} = \frac{\partial L}{\partial a_i^L}\frac{\partial \sigma^L}{\partial z_i^L} \tag{6.12}$$

Next we need compute the derivative $\partial L/\partial a_i^L$ given a quadratic loss function. If we consider a single sample $\mathbf{x}, \mathbf{y}$, then the derivative can be written as:

$$\begin{aligned}\frac{\mathrm{d}L}{\mathrm{d}a_i^L} &= \frac{\mathrm{d}}{\mathrm{d}a_i^L}\left[\frac{1}{2}\left\|\mathbf{a}^L - \mathbf{y}\right\|\right] \\ &= \frac{\mathrm{d}}{\mathrm{d}a_i^L}\left[\frac{1}{2}\left((a_1^L - y_1)^2 + .. + (a_i^L - y_i)^2 + .. + (a_{n_L}^L - y_{n_L})^2\right)\right] \\ &= a_i^L - y_i\end{aligned} \tag{6.13}$$

Which makes the final expression:

$$\frac{\partial L}{\partial z_i^L} = \frac{\partial \sigma^L}{\partial z_i^L}(a_i^L - y_i) \tag{6.14}$$

This is the component expression of the first equation.

Equation 1 sub b

$$\frac{\partial L}{\partial z_i^L} = -\frac{\partial}{\partial z_i^L}\left[\sum_{j=1}^{n_L} y_j \ln(a_j^L)\right] = -\sum_{j=1}^{n_L} y_j \frac{\partial}{\partial z_i^L}\ln(a_j^L) \tag{6.15}$$

We first consider the derivative term in the sum:

$$\frac{\partial}{\partial z_i^L}\ln(a_j^L) = \frac{1}{a_j^L}\frac{\partial a_j^L}{\partial z_i^L} \tag{6.16}$$

Substituting the softmax activation function:

$$\frac{\partial a_j^L}{\partial z_i^L} = \frac{\partial}{\partial z_i^L}\left[\frac{\exp(z_j^L)}{\sum_{i=1}^{n_L}\exp(z_i^L)}\right] = \begin{cases} -a_i^L a_j^L \text{ for } i \neq j \\ a_i^L\left(1 - a_j^L\right) \text{ for } i = j \end{cases} \tag{6.17}$$

substituting this into the equations above we find:

$$\begin{aligned}\frac{\partial L}{\partial z_i^L} &= -\sum_{j=1}^{n_L} y_j \frac{\partial}{\partial z_i^L}\ln(a_j^L) \\ &= -\sum_{j=1;j\neq i}^{n_L} y_j \frac{1}{a_j^L}\left(-a_i^L a_j^L\right) - y_i \frac{1}{a_i^L} a_i^L\left(1 - a_i^L\right) \\ &= a_i^L\left(y_i + \sum_{j=1;j\neq i}^{n_L} y_j\right) - y_i \\ &= a_i^L - y_i\end{aligned} \tag{6.18}$$

, which is the component form of the first expression for a cross entropy loss function and a softmax activation function.

Equation 2

For the other layers we can also use the chain rule to find an expression of the loss L w.r.t the input $\mathbf{z}^l$ of the activation function of that particular layer. Again we compute this for the components of $\mathbf{z}^L$ and use the chain rule:

$$\frac{\partial L}{\partial z_i^l} = \sum_{j=1}^{n_l} \frac{\partial L}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_i^l} \tag{6.19}$$

Because each input z_i^l ends up via the weights in every component of $\mathbf{z}^{l+1}$ we need to sum over these components. The first derivative of this sum is already known when using backpropagation, so only the last derivative needs to be worked out a bit further:

$$\begin{aligned} \frac{\partial L}{\partial z_i^l} &= \sum_{j=1}^{n_l} \frac{\partial L}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_i^l} = \sum_{j=1}^{n_l} \frac{\partial L}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial a_i^l} \frac{\partial a_i^l}{\partial z_i^l} = \sum_{j=1}^{n_l} \frac{\partial L}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{a_i^l} \frac{\partial \sigma^l}{\partial z_i^l} = \\ &= \frac{\partial \sigma^l}{\partial z_i^l} \sum_{j=1}^{n_l} \frac{\partial L}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{a_i^l} \end{aligned} \tag{6.20}$$

Since the input z_j^{l+1} depends on the output a_i^l of the previous layer through the weights of layer $l+1$ the last derivative can be expressed explicitly in these weights:

$$\begin{aligned} \frac{\partial L}{\partial z_i^l} &= \frac{\partial \sigma^l}{\partial z_i^l} \sum_{j=1}^{n_l} \frac{\partial L}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{a_i^l} = \frac{\partial \sigma^l}{\partial z_i^l} \sum_{j=1}^{n_l} \frac{\partial L}{\partial z_j^{l+1}} W_{ji}^{l+1} = \\ &= \frac{\partial \sigma^l}{\partial z_i^l} \sum_{j=1}^{n_l} (W^T)_{ij}^{l+1} \frac{\partial L}{\partial z_j^{l+1}} \end{aligned} \tag{6.21}$$

Which is the component form of the second equation.

Equation 3

Next we derive the equations of the derivative of the loss function with respect to the weights. Again in component form:

$$\frac{\partial L}{\partial W_{ij}^l} = \frac{\partial L}{\partial z_i^l} \frac{\partial z_i}{\partial W_{ij}^l} = \frac{\partial L}{\partial z_i^l} a_j^l \tag{6.22}$$

, where we made use of:

$$z_i^l = W_{ij}^l a_j^l + b_i^l \tag{6.23}$$

Equation 4

Finally the fourth equation that express the derivative of the loss function with respect to the components of the bias vector:

$$\frac{\partial L}{\partial b_i^l} = \frac{\partial L}{\partial z_i^l} \frac{\partial z_i}{\partial b_i^l} = \frac{\partial L}{\partial z_i^l} \tag{6.24}$$

, where we made use of:

$$z_i^l = W_{ij}^l a_j^l + b_i^l \tag{6.25}$$

6.9 Training the network

We now have all elements in place to perform the training. To apply stochastic gradient descent it is common practice to divide the data set in three:

1. **A training set**. This is the largest subset of the data and is actually used to train the NN.
2. **A validation set**. After each iteration (called *epoch*) in the training, the validation set is passed through the NN and the loss is computed. If this loss is compared to the loss of the training set it is possible to judge if we have over fitting. If for example, after some number of epochs, the loss of the training set keeps dropping every epoch, but the loss of the validation set does not decrease (or even increases), this is a clear sign of over fitting the data. So the validation set helps to determine when to stop.
3. **A test set**. Finally the test set is used to observe the performance of the trained NN. Note that we cannot use the validation set as this set is used to tune the model.

Now we can start training:

1. **create batches**. Divide the data in a batches of equal size.
2. **epoch**. For each batch do:
   (a) **forward pass**. Calculate the output of the NN ($\mathbf{a}^L$) for each data point in the batch.
   (b) **backward pass**. Compute the derivatives $\partial L/\partial \mathbf{W}^l$ and $\partial L/\partial \mathbf{b}^l$ for all layers and sum over all data points in the batch.
   (c) **update weights**. Compute the new weights and biases.
3. **compute loss**. Compute the loss of the training and validation set.
4. **Repeat**. Repeat steps 2 through 4 until you decide that the performance does not further increase
5. **test**. Evaluate the performance of the trained NN by computing the loss of the the test set.

Finally, the whole procedure can be repeated with different settings such as learning rate, number of layers, activation functions, number of neurons in each layer etc.