Learning from Examples II

Julius, Hinze (366880)
Ferit, Tohidi Far (366667)

August 29, 2018

Contents

List of Figures

List of Tables

Abstract

Machine learning is currently used by many companies world wide and its potential is enormous. Many problems that cannot be solved efficiently using ordinary algorithms can be solved efficiently using artificial intelligence. This papers objective is to present a brief overview of some of the different machine learning techniques. These are discussed in Section I - III, equipped with example datasets and pseudocode. Section IV covers the comparison of different models in the fields efficiency, learning convergence and reliability.

The contents presented in this paper are the results of literature review with added examples. This paper is published as part of a series of papers in the course "Proseminar: Artificial Intelligence" at RWTH-Aachen university. Some technical terms that are used might have been explained in a previous paper.

Sourcecodes and datasets can be found in our github repository: supermuesli/ProKI2018.

1 Regression and Classification with Linear Models

Imagine you have a dataset which consists of exactly one feature and one output, for instance the amount of hours someone studied for an exam (feature) and the score that that person achieved in that exam (output). If there is a correlation, ideally a linear one, between the output and its feature - and if there is enough data (a set of features and their respective outputs) - we can use a model called "linear regression" to predict output-values for feature-values that we do not have.
The hypothesis space, a set of all possible functions that can be learned by our model, is set as the class of linear functions that take continuous-valued inputs.

1.1 Linear regression

Univariate linear regression is better known as "fitting a straight line" of the form

$$h_w(x) = w_1 x + w_0$$

(where w stands for weight) to a dataset. The goal is to minimize the empirical loss function

$$L_2(expected, guess) = (expected - guess)^2$$

where

$$L_q(expected, guess) = (expected - guess)^q$$

for all training samples to obtain "good weights" w_0 and w_1. This process is called linear regression. Let $N = n-1$ where n is the length of the dataset, then the summed Loss function can be written as:

$$Loss(h_w) = \sum_{j=0}^{N} L_2(y_j, h_w(x_j)) = \sum_{j=0}^{N} (y_j - h_w(x_j))^2 = \sum_{j=0}^{N} (y_j - (w_1 x_j + w_0))^2$$

A quadratic function gets to a minimum if the first partial derivatives (with respect to w_0 and w_1) are zero:

$$\frac{\delta}{\Delta w_0} \sum_{j=0}^{N} (y_j - (w_1 x_j + w_0))^2 = 0 \text{ and } \frac{\delta}{\Delta w_1} \sum_{j=0}^{N} (y_j - (w_1 x_j + w_0))^2 = 0$$

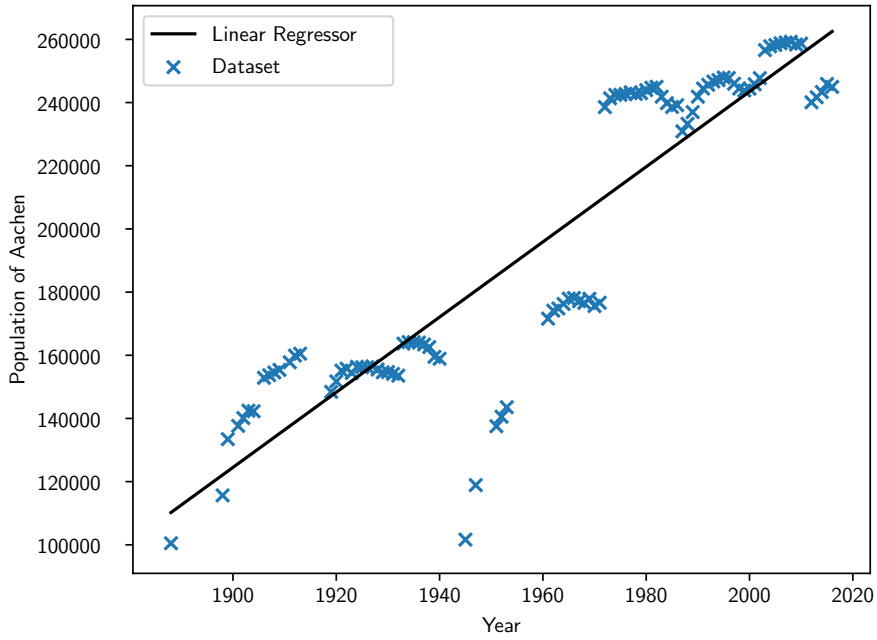

Figure 1: Population of the German city Aachen from 1888 to 2016 (Source Wikipedia)

The unique closed-form solutions to those equations are:

$$w_1 = \frac{N(\sum_{j=0}^{N} x_j y_j) - (\sum_{j=0}^{N} x_j)(\sum_{j=0}^{N} y_j)}{N(\sum_{j=0}^{N} x_j^2) - (\sum_{j=0}^{N} x_j)^2} \text{ and } w_0 = \frac{(\sum_{j=0}^{N} y_j - w_1(\sum_{j=0}^{N} x_j))}{N}$$

Figure 1 shows the population of Aachen (crosses) and the linear function that minimizes $Loss(h_w)$ ($w_0 = -2135977.29$ and $w_1 = 1189.73$). Notice the drop at 1940 due to the Second World War?

1.2 Gradient descent

The closed-form solutions for w_0 and w_1 above only work for univariate datasets (datasets with exactly one feature and one output). For multivariate datasets (multiple features and one output) there is also a closed-form solution. Adjusting our hypothesis space leaves us with the set of functions of the form:

$$h_w(x_j) = w_0 + \sum_{i=1}^{N} w_i x_{j,i}$$

By introducing a dummy input attribute $x_{j,0}$ which is always set equal to 1, the hypothesis space can now be brought into a more convenient form:

$$h_w(x_j) = w \cdot x_j = w^T x_j = \sum_{i=0}^{N} w_i x_{j,i}$$

with the summed loss function:

$$Loss(h_w(x)) = \sum_{j=0}^{N}(L_2(h_w(x)))^2 = \sum_{j=0}^{N}(y_j - h_w(x_j))^2 = \sum_{j=0}^{N}(y_j - w^T x_j) = \sum_{j=0}^{N}(y_j - \sum_{i=0}^{N} w_i x_{j,i})$$

Let X be the data-matrix where X_i is the $i'th$ feature-vector of the dataset and let y be the vector of outputs. The following equation (ordinary least squares) minimizes $Loss(h_w)$:

$$w^* = (X^T X)^{-1} X^T y$$

The time complexity of the ordinary least squares method is $O(n^3)$, where n is the length of the dataset. There is a quicker way to compute the best fitting weights - especially if the dataset is very large - by approaching the weights numerically using gradient descent [DL06].

We start by assigning random values to our weights. Essentially we keep adjusting the weights by subtracting their respective partial derivatives of the summed loss function $Loss(h_w)$ multiplied by a learning rate α to them.
To provoke (or speed up) convergence it is critical to pick a learning rate $0 < \alpha < 1$. If α is too big it is possible that the algorithm might overshoot the minimum and if α is too small it will take longer to converge. The value of α is unknown at the beginning, which is why it has to be guessed at first. This can be done by starting with a large value like 1 (which will almost certainly cause the algorithm to diverge) and steadily decreasing it until the algorithm stops diverging. As soon as convergence seems to take place it should be good to use [TS13].
Figure 2 gives us an understanding of how different learning rates affect the summed loss:

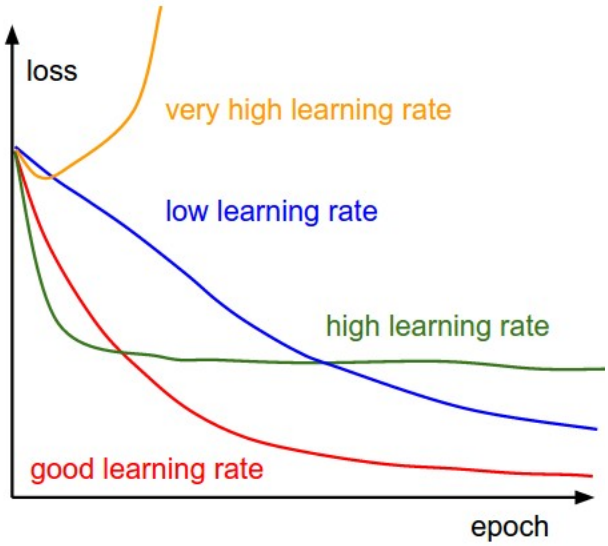

Figure 2: A cartoon that demonstrates the effect of learning rates on the summed loss (Source cs231n.github.io)

But why do we take the derivative of the summed loss function? Figure 3 provides an intuition of what is actually happening in gradient descent:

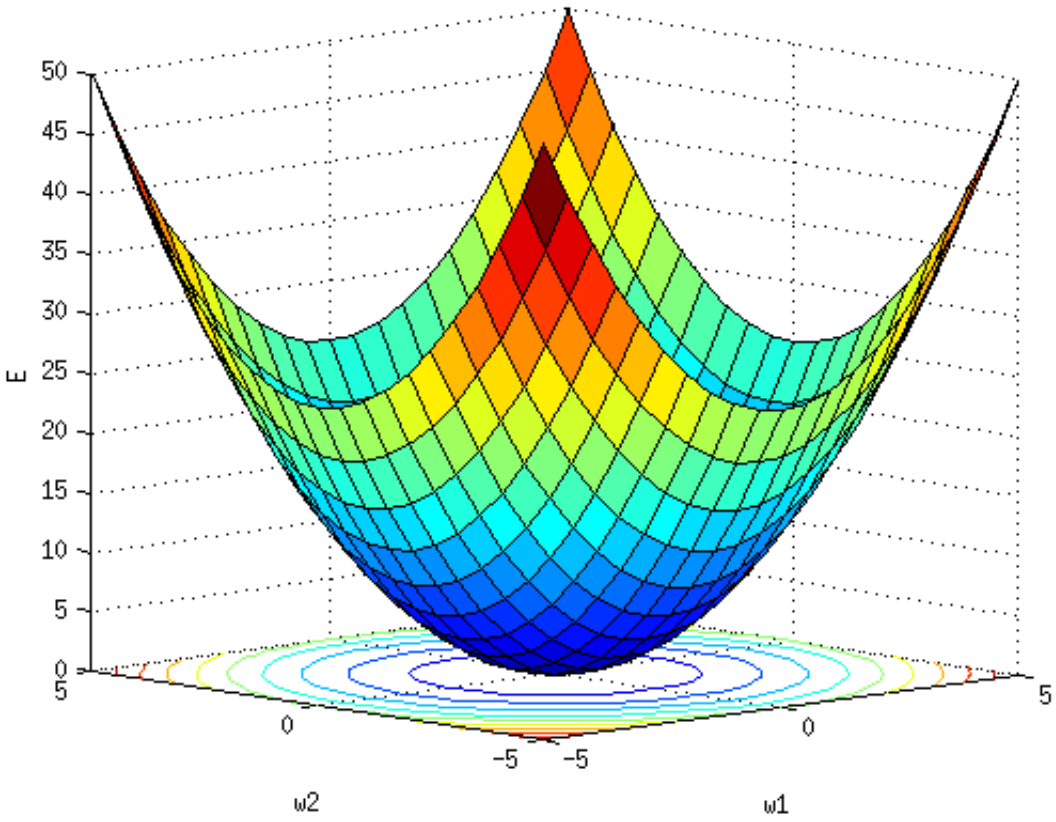

Figure 3: Error Surface of Linear Neuron with two Input Weights (Source Wikipedia)

In our notation the axes would be labeled as "E" = "$Loss(h_w)$", "w_1" = "w_0" and "w_2" = "w_1". The desired weights, in this case for univariate datasets, are exactly at the lowest point of the error slope. The only way to get there is by following the negative gradient of the slope, which is why we subtract the partial derivatives of the summed loss function. They basically "point" at the direction that needs to be followed to minimize the summed loss.

Note that in the following calculations we summarize all occuring factors within α for the sake of readability. Repeating the following until convergence minimizes $Loss(h_w)$ for both univariate and multivariate datasets (using their respective hypothesis spaces $h_w(x)$):

$$for \quad each \quad weight \quad w_i :$$
$$w_i \leftarrow w_i + \alpha \frac{\partial}{\partial w_i} Loss(h_w)$$

As an example, let us determine the gradients of w_0 and w_1 of $Loss(h_w)$ for any univariate dataset:

$$\frac{\partial}{\partial w_0} Loss(h_w) = \frac{\partial}{\partial w_0} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = \sum_{j=1}^{N} -2(y_j - (w_1 x_j + w_0))$$
$$\frac{\partial}{\partial w_1} Loss(h_w) = \frac{\partial}{\partial w_0} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = \sum_{j=1}^{N} -2(y_j - (w_1 x_j + w_0)) \cdot x_j$$

(∗) Now we adjust w_0 and w_1 by adding their gradients multiplied by our learning rate α:

$$w_0 \leftarrow w_0 + \alpha(\sum_{j=0}^{N} -2(y_j - (w_1 x_j + w_0)))$$
$$w_1 \leftarrow w_1 + \alpha(\sum_{j=0}^{N} -2(y_j - (w_1 x_j + w_0))x_j)$$

Repeating (∗) until convergence will deliver the desired weights w_0 and w_1 that minimize $Loss(h_w)$ for any univariate dataset.

The above weight update rule is called "batch gradient descent" because the weights are updated all at once. Another type of gradient descent is "stochastic gradient descent", where weights are changed one training-data at a time. Additionally, the order of introduction of the training-data is now random and continuously changing randomly, which helps prevent overfitting in multivariate datasets. However, convergence is not guaranteed in the latter. Picking a decaying learning rate fixes this problem, this will become clear later on.

1.3 Overfitting

In supervised learning models with multivariate datasets it is possible that some feature appears by chance to be relevant. Thus, using gradient descent will often times deliver weights that appear to do the job for the given training-dataset but fail to perform well on foreign datasets. This type of error is called "overfitting". To prevent this there are multiple things that we can do. In multivariate linear regression we can use regularization.

We aim to minimize the total cost of the hypothesis by taking into account its empirical loss (which is the mean of the summed L_2 loss function) as well as its complexity:

$$Cost(h_w) = EmpLoss(h_w) + \lambda \cdot Complexity(h_w)$$

where

$$Complexity(h_w) = L_q(w) = \sum_i |w_i|^q$$

is a set of regularization functions.

Since we now have to minimize $Cost(h)$ we are interested in its gradient with respect to w_i. As an example, for L_1 regularization we have:

$$\begin{aligned}
\frac{\partial}{\partial w_i} Cost(h_w) &= \frac{\partial}{\partial w_i} EmpLoss(h) + \lambda \cdot Complexity(h_w) \\
&= \frac{\partial}{\partial w_i} \frac{1}{n} \sum_{j=0} (y_j - h_w(x_j))^2 + \lambda L_1(w) \\
&= \frac{\partial}{\partial w_i} \frac{1}{n} \sum_{j=0} (y_j - h_w(x_j))^2 + \lambda \sum_{i=0} |w_i| \\
&= \frac{2}{n} \sum_{j=0} (y_j - h_w(x_j)) x_{j,i} + \lambda
\end{aligned}$$

Thus, our new weight update rule becomes:

$$w_i \leftarrow w_i + \alpha \sum_{j=0}^{N} ((y_j - h_w(x_j))x_{j,i}) + \lambda$$

Regularization can be seen as a penalty for dramatically large weights, because the larger the weights become, the larger L_q becomes, the larger $Cost(h)$ becomes. By forcing the sum of the weights to stay small we lose the ability to perfectly fit our hypothesis to our training-data, but in exchange we drastically reduce the empirical loss (mean of the summed L_2 loss).

The question might arise: how do we pick λ?. Just like we did for the learning rate α, we guess. We first look for an α for which gradient descent is sure to converge. After having found an α, we can start looking for λ. Again, we start out with a large value and steadily decrease it until the model seems to converge. Once it starts converging, we just keep decreasing it until the summed loss does not get any smaller.

To prevent overfitting caused by irrelevant features it is conventional to use L_1 regularization, because it often sets weights to zero, rendering them useless. To demonstrate this, we implemented a multivariate linear regression model without regularization in python. Our dataset contains 1500 entries with the following information:

1. Frequency (in Hertzs)
2. Angle of attack (in degrees)
3. Chord length (in meters)
4. Free-stream velocity (in meters per second)
5. Suction side displacement thickness (in meters)
6. Scaled sound pressure level (in decibels)

where items 1-5 are features and item 6 is the output. What these values mean is of no interest to us, we solely want to inspect our models performance. Further information about the used dataset and the sourcecode of our implementation is provided in our github repository.

When inspecting a couple of samples, we note that the relationship of the features is not linear, which leads us to believe that the model will deliver imperfect performance when it is done training. In figure 4 we can see the summed loss after 1000 iterations, starting with all weights set to 0. Our model achieves a mean summed loss of 8653. The weights that came out at the end were:

$$
\begin{aligned}
w_0 &= 0.0006930449991439476 \\
w_1 &= 0.018092900052509107 \\
w_2 &= 0.00514462606296179 \\
w_3 &= 0.00010200223803102637 \\
w_4 &= 0.033865004926948813 \\
w_5 &= 7.819526052013582 \cdot 10^{-6}
\end{aligned}
$$

Figure 4 shows the summed loss after 1000 iterations after applying L_1 regularization, again, starting with all weights set to 0. This model achieves a mean summed loss of 2539. As

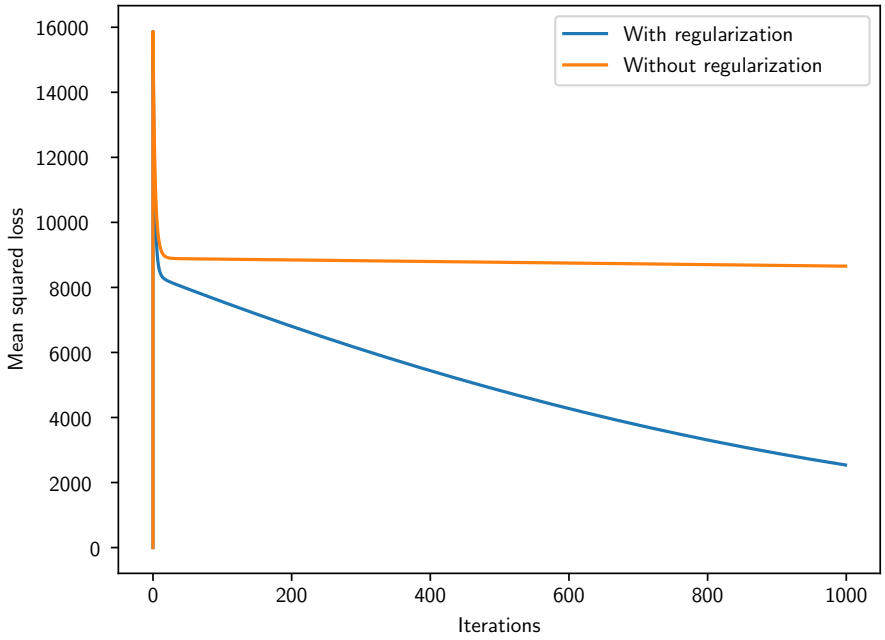

Figure 4: Summed error after 1000 iterations with and without regularization

predicted, the loss got considerably smaller. The weights that came out at the end were:

$$
\begin{aligned}
w_0 &= 1.000386469038675 \\
w_1 &= 0.014372872686644802 \\
w_2 &= 1.0030966344631689 \\
w_3 &= 1.000058142636903 \\
w_4 &= 1.0170036581920823 \\
w_5 &= 1.0000047464723236
\end{aligned}
$$

For the mean summed loss we applied cross-validation, which basically splits the given training-data into two parts in order to use one part for training and one part for validation. This will be explained indepth later on.

1.4 Linear classifiers

Now we want to classify data. This is done by first finding the boundary line (or surface in higher dimensions) that best seperates the different classes of data and then mapping points that are seperated by that line (or surface) to their respective classes. The graph in figure 5 plots features x_1 and x_2 of classes red and blue. The boundary line that seperates these two classes of data is a line of the form:

$$0 = w_2x_2 + w_1x_1 + w_0$$

The points under the boundary line are of class red and the points above the boundary line are of class blue. This means that data points of class blue are points for which $w_2x_2 + w_1x_1 + w_0 > 0$ and data points of class red are points for which $w_2x_2 + w_1x_1 + w_0 < 0$.

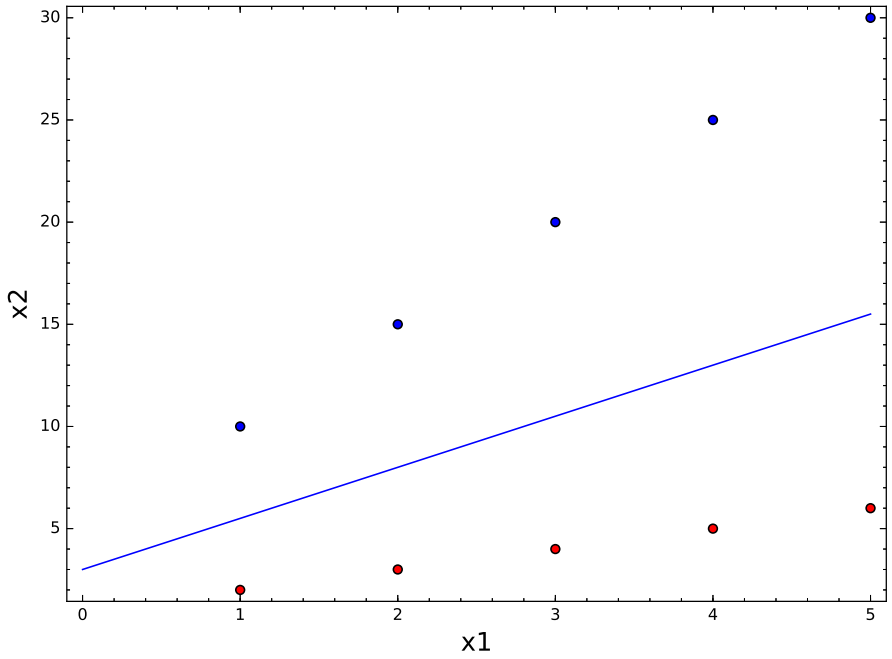

Figure 5: Feature points of classes red and blue are seperated by a linear boundary line

Like previously, by using a dummy input $x_0 = 1$ the classification hypothesis can be written as

$$h_w(x) = 1 \quad \text{if} \quad w^T x \geq 0 \quad \text{else} \quad 0$$

where 1 is interpreted as blue and 0 as red. The classification hypothesis passes $w^T x$ to a threshold function, so generalizing it for any threshold function leaves us with

$$h_w(x) = Threshold(w^T x)$$

where the hard threshold is defined by

$$Threshold(z) = 1 \quad \text{if} \quad z \geq 0 \quad \text{else} \quad 0$$

Now we want to find the boundary line that minimizes the summed loss. We cannot do that by using gradient descent because the gradients of the hard threshold function are 0 at nearly every point in weightspace and at some points not even differentiable. If the given data is linearly separable, then the perceptron learning rule, which is also used for neural networks, will deliver the weights that best seperate the classes.

The rule will be applied one training-data at a time, preferrably at random order like in stochastic gradient descent (this will speed up convergence). Let y be the corresponding class for the current feature-vector x. Repeating the following until convergence will deliver the desired weights:

$$\begin{aligned} &for \quad each \quad weight \quad w_i : \\ &\quad w_i \leftarrow w_i + \alpha(y - h_w(x))x_i \end{aligned}$$

In this case we used and derived from the summed L_1 loss function, because our only outputs are 1's and 0's. If the outputs were realvalued like before then we would have used the

summed L_2 function. Also, if the dataset is not linearly separable, the perceptron learning rule does not guarantee convergence. However, by picking a learning rate that is decaying over time, the algorithm is sure to converge again as long as the training-data is presented at random order and the decay of α is in $O(t)$ where t is the number of iterations that have been performed. Since the perceptron learning rule is applied one training-data at a time, it is possible that good weights are replaced by bad ones, especially if the dataset is not linearly separable. We can expand the rule by memorizing the weights that caused the least summed loss and only keeping the weight updates if they were benefitial. This procedure is called "pocket algorithm" and is more likely to deliver the best weights, at a faster pace even [Mus97].

There is a better way to do linear classification where predictions are probabilities instead of absolute discrete class values simply by using a different threshold function. Instead of using a hard threshold function like before, we now need a soft threshold function, which means that the function is required to be continuous and differentiable at every point in weightspace. Furthermore, it has to be able to map every real number to $\{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$, which is exactly the set of numbers that respresents probabilistic values. A popular and widely used function that has these properties is the sigmoid function:

$$Sigmoid(z) = \frac{1}{1 - e^{-z}}$$

Using a soft threshold enables us to use gradient descent on our previously defined classification hypothesis space, so we can now minimize the summed loss. This process is called "logistic regression". Our classification hypothesis space is now

$$h_w(x) = Sigmoid(w^T x)$$

with the summed loss function L_2. For the weight update rule it helps to know that the sigmoid function, which will henceforth be reffered to as g with g' being its derivative, has an additional property:

$$g'(z) = g(z)(1 - g(z))$$

Thus, the weight update rule for gradient descent is:

$$\begin{aligned} &for \quad each \quad weight \quad w_i : \\ &\qquad w_i \leftarrow w_i + \alpha(y - h_w(x))h_w(x)(1 - h_w(x))x_i \end{aligned}$$

until convergence.

When comparing the performance of linear classification with a hard threshold and logistic regression for linearly separable datasets, logistic regression converges at a slower pace but produces more reliable outputs. On the other hand, for non-linearly separable datasets, linear classification tends to perform poorly while logistic regression converges way quicker and reliably, even if the training-data is noisy. This implies that logistic regression is a better model for classification, which is backed up by the fact that it is widely used among serious data scientists.

This sums up linear regression. As a general rule of thumb, if the outputs of a dataset are continuous we apply linear regression and if the outputs are discrete we apply linear classification. In the following section we will learn about a model which can be thought of as a model for non-linear regression and classification. It is by far the most popular machine learning model as of right now.

2 Artificial Neural Networks

Lets step back from artificial intelligence and discuss human intelligence and the human brain. Brain activity was found to be electrochemical signals that get emitted and received by **neurons**. Even though neurons and the brain are not fully understood until now, scientists already have a great understanding of how they work. **Artificial Neural Networks** try to imitate the human brain by implementing a similar structure to neurons and their connections.

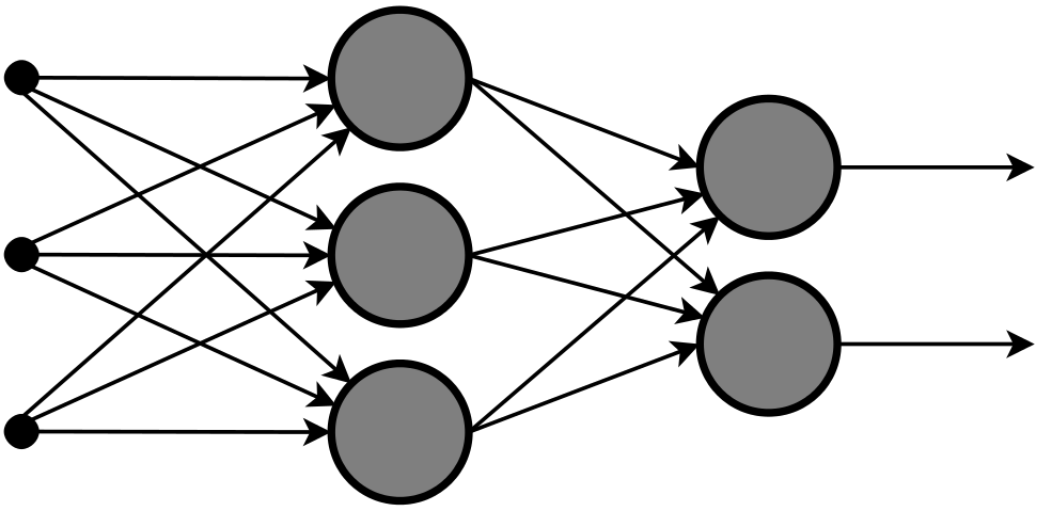

Figure 6: Neural network with three inputs, one hidden layer with three nodes and two outputs

2.1 Neural Network Structures

A neuron (in its simplified form) can be described in three parts. Other neurons pass their signals to into the inputs of the next neuron, this is called the input. This input is then processed by an **activation function** g which calculates the output of the neuron. By linking the in- and outputs of different neurons, this artificial "brain" can solve different problems depending on topology and structure of the network. Mathematically speaking these cells are usually called **nodes** or units which are joined together using directed links which pass an activation a_i from node i to node j to form a network. A link from node i to j has an associated weight $w_{i,j}$ to damp or increase the incoming or outgoing signal. The first weight $a_0 = 1$ is called a dummy input (just like in linear regression). It has the associated weight $w_{0,j}$. Each neuron in the networks calculates the weighted sum in_j of the incoming signals

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i.$$

The node then applies the activation function g to calculate its output:

$$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j} a_i).$$

The function g can either be a **hard threshold** or a **logistic function** which are both non linear functions. This non linear property of the activation function leads to the fact that neural networks can represent non linear functions which is a big improvement compared to

linear regression.

There are two different ways of linking the different nodes together, but we will mainly focus on feed-forward networks. Feed-forward networks, as the name implies, just hand information in one direction starting from an input. This neural network has no internal "memory" aside of the weights associated which each link.

Algorithm 1: Backpropagation in FFNN

Input: *examples*, the example dataset with input vector x and output y
network, a multilayer network with L layers, weights $w_{i,j}$, activation function g
Output: a trained neural network
repeat
  foreach *weight* $w_{i,j}$ *in network* **do**
    $w_{i,j} \leftarrow$ a small random number
  end
  foreach *example* (x, y) *in examples* **do**
    `/* Propagate the inputs forward to compute the outputs */`
    foreach *node* i *in the input layer* **do**
      $a_i \leftarrow x_i$
    end
    for $l = 2$ *to* L **do**
      foreach *node* j *in layer* l **do**
        $in_j \leftarrow \sum_i w_{i,j} a_i \; a_j \leftarrow g(in_j)$
      end
    end
    `/* Propagate deltas backward from output layer to input layer */`
    foreach *node* j *in output layer* **do**
      $\Delta_j \leftarrow g'(in_i) \cdot (y_j - a_j)$
    end
    for $l = L - 1$ *to* 1 **do**
      foreach *node* i *in layer* l **do**
        $\Delta_i \leftarrow g'(in_i) \sum_j w_{i,j} \Delta_j$
      end
    end
    `/* Update every weight in network using deltas */`
    foreach *weight* $w_{i,j}$ *in network* **do**
      $w_{i,j} \leftarrow w_{i,j} + \alpha \cdot a_i \cdot \Delta_j$
    end
  end
until *some stopping criterion is satisfied*;

2.2 Feed-forward neural networks

Arranging neurons in a feed-forward neural network (FFNN) is usually done in **layers**. Each of those layers receives inputs from the previous layers and passes them to the next one. The first layer is called the **input layer** and the last one **output layer**. In between those layers are the so called **hidden layers**. The hidden layers may have arbitrary size. In figure 6 you can see an example of a neural network featuring three inputs, one hidden layer with three nodes and two outputs. Every FFNN calculates a non linear function $h_w(x)$ which depends

only on the input x and the weights w. Because of this property a FFNN can be used to do non linear regression: every node calculates a soft-threshholded linear combination of its inputs [RN02].

In figure 6 the output layer depends on all previous nodes. Training this network should take into account all the errors that occur and back propagate the new weights accordingly. The gradient of the loss function L_2 for a single layer network can be written as

$$\frac{\partial}{\partial w} Loss(w) = \frac{\partial}{\partial w} |y - h_w(x)|^2 = \frac{\partial}{\partial w} \sum_k (y_k - a_k)^2 = \sum_k \frac{\partial}{\partial w} (y_k - a_k)^2$$

where k iterates over all output layers and y is the expected output. We can see that it is easy to decompose the m-output learning problem into m 1-output learning problems. Things get complicated if the FFNN consist of more than one hidden layer: the error $y - h_w$ at the hidden layers can not be represented by the previous loss function. In order to successfully bring the FFNN to converge we need to back-propagate the errors through all hidden layers. Let Err_k be the kth component of the error vector $y - h_w$ and $\Delta_k = Err_k \cdot g'(in_k)$ a modified error. The weight update rule now becomes

$$w_{i,j} \leftarrow w_{i,j} + \alpha \cdot a_i \cdot \Delta_k$$

We assume that every hidden node j is responsible for a fraction of the error Δ_k in all output nodes to which node j is linked. The back-propagation algorithm calculates a **weighted error** Δ_j based on the strength of the link between the node j and all output nodes:

$$\Delta_j = g(in_j) \sum_k w_{j,k} \Delta_k$$

The new update rule is quite similar to the previous update rule:

$$w_{i,j} \leftarrow w_{i,j} + \alpha \cdot a_i \cdot \Delta_j$$

The pseudocode for back-propagation is taken directly from [RN02]:

To put it in simple terms, this is what we do:

1. using the observed error, we compute the Δ values for the output units
2. starting with the output layer, we repeat the following for each layer in the network, until we reach the first layer:
   (a) we propagate the Δ values back to the previous layer
   (b) we update the weights between the two layers

To demonstrate this, we derive the back propagation update rules by hand. We have to compute the gradient for $Loss_k = (y_k - a_k)^2$ at the k'th output. Except for weights $w_{j,k}$ that connect to the k'th output unit, the gradients of the mentioned loss function with respect to the weights connecting the hidden layer to the output layer will be 0. We have

$$\begin{aligned}
\frac{\partial Loss_k}{\partial w_{j,k}} &= -2(y_k - a_k) \frac{\partial a_k}{\partial w_{j,k}} = -2(y_k - a_k) \frac{\partial g(in_k)}{\partial w_{j,k}} \\
&= -2(y_k - a_k) g'(in_k) \frac{\partial in_k}{\partial w_{j,k}} = -2(y_k - a_k) g'(in_k) \frac{\partial}{\partial w_{j,k}} (\sum_j w_{j,k} a_j) \\
&= -2(y_k - a_k) g'(in_k) a_j = -a_j \Delta_k
\end{aligned}$$

Deriving the gradients with respect to the $w_{i,j}$ weights connecting the input layer to the hidden layer by expanding out the activations a_j we get

$$\begin{aligned}
\frac{\partial Loss_k}{\partial w_{i,j}} &= -2(y_k - a_k)\frac{\partial a_k}{\partial w_{i,j}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{i,j}} \\
&= -2(y_k - a_k)g'(in_k)\frac{\partial in_k}{\partial w_{i,j}} = -2\Delta_k \frac{\partial}{\partial w_{i,j}}(\sum_j w_{i,ja_j}) \\
&= -2\Delta_k w_{j,k}\frac{\partial a_j}{\partial w_{i,j}} = -2\Delta_k w_{j,k}\frac{\partial g(in_j)}{\partial w_{i,j}} \\
&= -2\Delta_k w_{j,k}g'(in_j)\frac{\partial in_j}{\partial w_{i,j}} \\
&= -2\Delta_k w_{j,k}g'(in_j)\frac{\partial}{\partial w_{i,j}}(\sum_i w_{i,j}a_i) \\
&= -2\Delta_k w_{j,k}g'(in_j)a_i = -a_i\Delta_j
\end{aligned}$$

The same applies for networks with more than one hidden layer if continued in this fashion. This essentially sums up learning in neural networks.

3 Support Vector Machines (SVM)

Support Vector Machines (SVMs) are supervised learning models which classify new examples into categories. This is done by finding one (or multiple) Hyperplanes which separate the n dimensional space into smaller subspaces. A SVM may use different learning techniques. Which one to use depends on many constraints such as the separability of the features or the general domain of the problem. There even exists a model to use SVMs in regression problems [CST00]. Note that in the following paragraphs $\bullet$ denotes the dot product between the two vectors ($\vec{x} \bullet \vec{y} = \vec{x}^T\vec{y}$).

3.1 Maximum margin hyperplane

A SVM maps examples as points in a space divided by a hyperplane such that the space is now split in multiple parts representing a category each. Those points should have a maximum distance to this hyperplane so noisy input will be categorized correctly. New examples will get mapped into the same space and predicted to belong to a category based on which side of the hyperplane the points fall. The points closest to the hyperplane are called **support vectors**. The name "support vector" is chosen because these vectors hold up the hyperplane.

Let $T = \{\vec{x_i}, y_i\}$ a linear separable set with y values from -1 to 1, a hyperplane can then be written as

$$\vec{w} \bullet \vec{x} + b = 0$$

where $\vec{w}$ is the normal on the hyperplane and b the distance to the origin. Furthermore, a decision function

$$g(\vec{x}) = \vec{w} \bullet \vec{x_i} + b$$

returns the functional distance from the hyperplane. To simplify the calculations let us normalize $\vec{w}$ and $\vec{b}$. Let d^+ and d^- be the shortest distance of the hyperplane to a support vector on either side. Two conditions must hold for all $\vec{x_i} \in T$: $\vec{w} \bullet \vec{x_i} + b \geq 1$ for $y_i = 1$ and $\vec{w} \bullet \vec{x_i} + b \leq -1$ for $y_i = -1$. Or combined:

$$y_i(\vec{w} \bullet \vec{x_i} + b) - 1 \geq 0 \forall i.$$

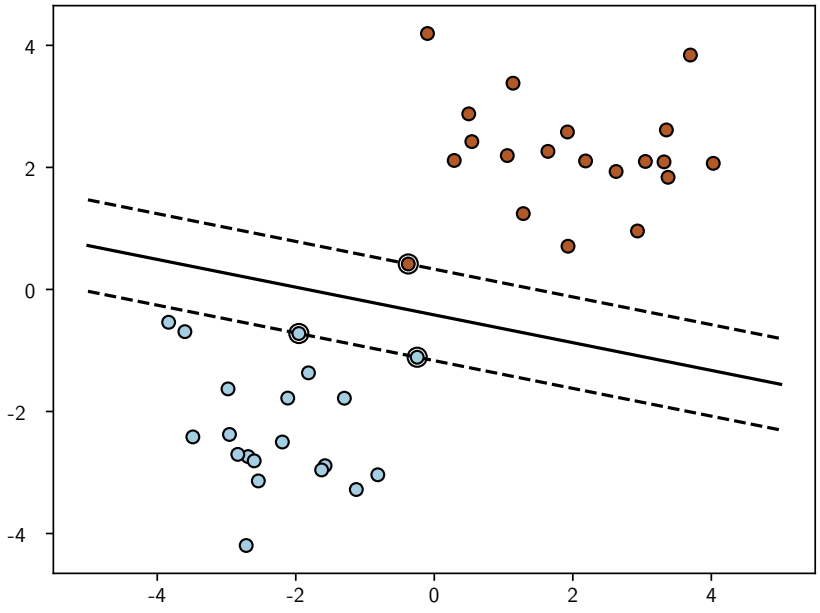

Figure 7: Linear SVM classifier with hyperplane, margins and support vectors

This equation implies that $d^+ = d^- = \|\vec{w}\|$ (a margin of $\frac{2}{\|\vec{w}\|}$) and that no point lays closer to the hyperplane that that margin. One can now simply maximize $\frac{2}{\|\vec{w}\|}$ and gather the resulting hyperplane. The next subsection covers this part in detail.

Lets stick with a two dimensional problem. The linear hyperplane now becomes a line which separates the space into two parts. Lets have a look at an example: figure 7 shows some random points with two features and two classes (blue and brown). The solid line indicates the separating hyperplane, the dotted lines represent the maximum margin. Circles with a double border are **support vectors**, clearly one can see that this is the perfect split because the point colors do not mix in either of the sides. The margin is relatively small so a kind of **soft margin** could be introduced to allow for some errors in order to gain less noise in the classification process.
In cases where it is just not possible to find a linear correlation between features and the corresponding class one can use a **kernel** "trick" to increase the dimension of the problem by merging two (or more) features into one new.

3.2 Optimal hyperplane (quadratic optimization)

One must maximise the margin but maximization is not easy to achieve and detecting convergence is much harder. We could achieve the same result by minimizing the Euclidean norm of $\vec{w}$. The problem can then be formulated as

$$\Phi(\vec{w}) = \frac{1}{2}\vec{w} \cdot \vec{w}$$

which is not only easier to calculate but also assures that training data only occurs in form of dot products between vectors. The kernel trick in the next subsection will benefit from

this property. Note that this cost function Φ is quadratic and convex and the constraints are linear so it can be solved by introducing l Lagrange multipliers $\alpha_i \geq 0$. The Duality Theorem [Hay98] states that this optimization problem has a unique optimal solution (a saddle point can always be found). The mentioned Lagrangian is formed by multiplying the positive Lagrange multipliers and then subtracting them from the cost function:

$$L_P(\vec{w}, b, \vec{\alpha}) = \frac{1}{2}\vec{w} \bullet \vec{w} - \sum_{i=1}^{l} \alpha_i \cdot (y_i \cdot (\vec{w} \bullet \vec{x_i} + b) - 1).$$

Solve L_P by calculating the derivatives of L_P with respect to $\vec{w}$ and b and finding their solution when equaled to zero:

$$\frac{\partial L_P(\vec{w}, b, \vec{\alpha})}{\partial \vec{w}} = 0 \quad \text{and} \quad \frac{\partial L_P(\vec{w}, b, \vec{\alpha})}{\partial b} = 0.$$

Plugging the first and second condition into L_P yields

$$\vec{w} = \sum_{i=1}^{l} \alpha_i y_i \vec{x_i} \quad \text{and} \quad \sum_{i=1}^{l} \alpha_i y_i = 0.$$

One can now expand L_P with the two optimality conditions and get

$$\begin{aligned} L_P(\vec{w}, b, \vec{x}) &= \frac{1}{2}\vec{w} \bullet \vec{w} - \sum_{i=1}^{l} \alpha_i \cdot (y_i \cdot (\vec{w} \bullet \vec{x_i} + b) - 1) \\ &= \frac{1}{2}\vec{w} \bullet \vec{w} - \sum_{i=1}^{l} \alpha_i y_i \vec{w} \bullet \vec{x_i} - b \sum_{i=1}^{l} \alpha_i y_i + \sum_{i=1}^{l} \alpha_i \end{aligned}$$

Note that the term $b \sum_{i=1}^{l} \alpha_i y_i$ is zero due to the second condition. With more rearranging described in [Hof06] we get to the Lagrange dual problem

$$L_D(\vec{\alpha}) = \sum_{i=1}^{l} \alpha_i - \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j \vec{x_i} \bullet \vec{x_j}$$

which can be solved by finding the optimal Lagrange multipliers α_i.

When provided with all the optimal $\alpha_{i,o}$ one can calculate the optimal $\vec{w}$ using

$$\vec{w_o} = \sum_{i=1}^{l} \alpha_{i,o} y_i \vec{x_i}.$$

Lets put this research into the hyperplane equation from the previous subsection:

$$\begin{aligned} \vec{w_o}^T \vec{x} + b_o &= (\sum_{i=1}^{l} \alpha_{i,o} y_i \vec{x_i}) \vec{x} + b_o \\ &= \sum_{i=1}^{l} \alpha_{i,o} y_i \vec{x_i} \bullet \vec{x} + b_o = 0 \end{aligned}$$

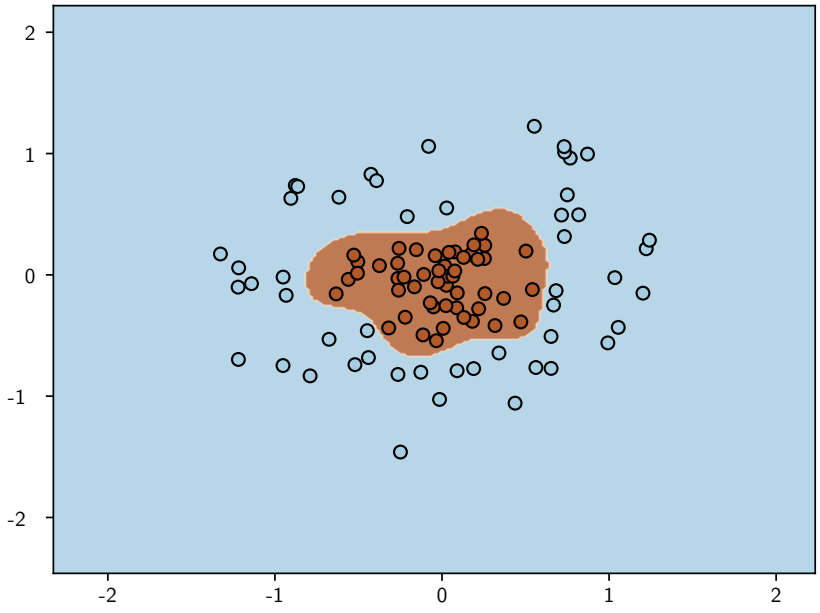

Figure 8: Example RBF SVM classifier

3.3 Kernel trick (non linear)

Only in very few real-world applications the set is linearly separable, as you can see in figure 8 a circular shape is required to decide both classes of points correctly. Introducing a kernel can solve this issue by raising the dimension of the problem. Consider a non linear mapping function $\Phi : \mathbb{R}^2 \to \mathbb{R}^3$ which maps a two dimensional feature space into a three dimensional one. Lets set $\Phi(\vec{x}) = (x_1^2, \sqrt{2} \cdot x_1 x_2, x_2^2)$. Note that Φ is now an elliptic function. If we plug this function into the hyperplane equation $\vec{w}\vec{x} + b = 0$ we get

$$\vec{w} \cdot \Phi(\vec{x}) = w_1 x_1^2 + w_2 \sqrt{2} \cdot x_1 x_2 + w_3 x_2^2 = 0$$

which is obviously a linear function. We see that it is easy to get a linear classifier if provided with a appropriate Φ. It is hard to find such kernel by hand, but there are general purpose kernels that work well for most common scenarios.
The kernel used in the example in figure 8 is the **Gaussian radial basis function** or RBF which is given by the function

$$k(x_i, x_j) = \exp(-\gamma \left\| x_i - x_j \right\|^2).$$

Note that γ is a so called hyper-parameter and is not learnt by the model but has to be assigned by yourself. The blue and brown areas mark the fitted decision function of the SVM, mapped back to $\mathbb{R}^2$. Examples of general purpose kernels are shown in table 1 [Hay98]:

Type of kernel	Inner product	Comments
Polynomial	$k(x_i, x_j) = (x_i x_j + \theta)^d$	d is degree of the polynomial, threshold θ
Gaussian	$k(x_i, x_j) = \exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})$	$2\sigma^2$ specified a priori
Gaussian RBF	$k(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$	
Laplace RBF	$k(x_i, x_j) = \exp(-\frac{\|x_i - x_j\|}{\sigma})$	
Sigmoid	$k(x_i, x_j) = \tanh(\eta x_i x_j + \theta)$	

Table 1: Different general purpose kernels

Furthermore, the **Kernel of sets** can be computed as

$$k(X, Y) = \sum_i = 1 N_X \sum_j = 1 N_Y k(x_i, x_j)$$

where k is a kernel function. This means that any of these kernels can be called to add more and more dimensions to problem, if no linear hyperplane could be found. Note that a function needs to satisfy **Mercer's Theorem** to be a valid candidate for a kernel function.

If after increasing the dimension through a kernel trick (or many kernel tricks) did not achieve the needed linear separability for the algorithm to work we can introduce a **soft margin** classifier which allows for some errors but assigns every sample that gets mapped to the wrong class a penalty proportional to the distance to the hyperplane. In the last section of the paper you can see the influence of the penalty parameter on the learning accuracy of a SVM without a kernel and a soft margin. Speaking in natural language the penalty determines how hard the SVM tries to avoid misclassifications. If the penalty parameter C is big, the SVM tries to find a smaller margin if it helps to reducing errors. Conversely, if C is small the margin gets bigger but the SVM allows for more errors. If C gets **very** small the SVM misclassifies a lot of the training data (even though it might be linearly separable).

3.4 Mercer's Theorem

Aside of predefined kernel functions one could be interested in finding an own kernel to fit the needs of the problem more accurately. A kernel function K needs to be symmetric,

$$K(\vec{x}, \vec{z}) = \phi(\vec{x})^T \phi(\vec{z}) = \phi(\vec{z})^T \phi(\vec{x}) = K(\vec{z}, \vec{x}).$$

Furthermore, Mercer provides a necessary and sufficient criteria K needs to fulfill. A kernel can be interpreted as a similarity matrix between its n inputs:

$$K = \begin{pmatrix} \phi(\vec{v_1})^T \phi(\vec{v_1}) & \dots & \phi(\vec{v_1})^T \phi(\vec{v_n}) \\ \vdots & \ddots & \vdots \\ \phi(\vec{v_n})^T \phi(\vec{v_1}) & \dots & \phi(\vec{v_n})^T \phi(\vec{v_n}) \end{pmatrix}$$

K is called the **Gram Matrix** which contains the inner products of the input vectors. As K is a symmetric matrix it is possible to extract its eigenvalues λ_t with corresponding eigenvectors $\vec{v_t} = (v_{ti})_{i=1}^n = (K - \lambda_t \cdot E)$ (columns of K) where E is the identity matrix. If all eigenvalues are greater than zero and ϕ is a feature mapping function with

$$\phi : \vec{x_i} \mapsto (\sqrt{\lambda_i} \cdot v_{ti})_{t=1}^n \in \mathbb{R}^n, i = 1, \dots, n$$

then

$$\phi(\vec{x_i}) \bullet \phi(\vec{x_j}) = \sum_{t=1}^{n} \lambda_t v_{ti} v_{tj} = K_{ij} = K(\vec{x_i}, \vec{x_j}).$$

Note that since K is symmetric and positive definite one can decompose $K = SAS^T$ where A is a diagonal matrix which contains the eigenvalues of K and S contains the respective eigenvectors. Hence a Gram Matrix provides all necessary information for the learning algorithm because K represents "orthogonal input data and therefore self-similarity dominates between-sample similarity" [Hof06]. A Gram matrix provides all necessary information for a learning model: data points and their mapping function which is condensed in the inner product [Hof06].

4 Tuning of Supervised Learning Models

This section covers the main techniques used to evaluate different models and tune their performance to gain better results. In order to compare learning models different type of plots are described. We will further be exploring one simple technique to optimize the choice of hyper-parameters to tune the accuracy of a learning model.

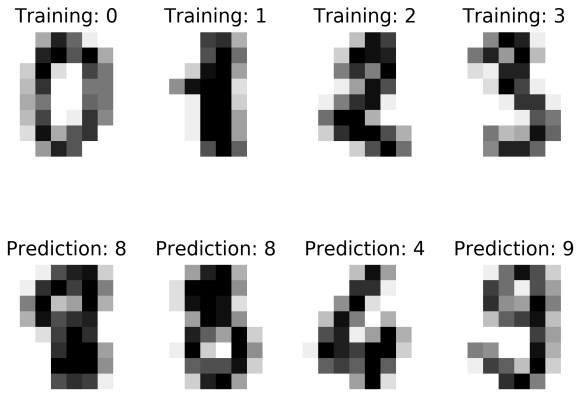

Figure 9: Example of handwritten digits and their predictions. Top row: hand written digits and their labels. Bottom row: hand written digits and their predicted labels

4.1 Train/Test Split and Cross Validation

To determine the accuracy of a model the dataset is being split into two parts where one is used to **train** the model and the other one is used to **test** the model. The test set is used to determine the accuracy and error rates. Note that no test samples where used to train the model, this ensures that the model generalizes correctly. If we did not split the dataset two things might happen. Imagine the training data to be equivalent to the testing data: the model could work perfectly on the training data but behaves bad on unknown training samples (overfitting). In contrast underfitting cannot be compensated by the train/test split method.

However this method has its dangers: if the dataset is not random but sorted in some parameters the test set would not represent the whole data but a fraction. This would result in heavy overfitting. To avoid this issues we can introduce **cross validation**.

The method is very similar to train/test split but it is applied to more subsets. The dataset is split into k subsets and training is only done on the $k-1$th dataset. There exists many different cross validation techniques, one of the most used ones is **k-Folds Cross Validation**.

In K-Folds cross validation the dataset is split in k subsets of equal size. The first $k-1$ subsets are used to train the model, the last fold is used to test the model. This is done k times (rounds) and the results get averaged.

4.2 Learning rate

Now that we know how to successfully evaluate a learning technique and we have talked about several different machine learning models let us take a look at an example: it is a well known problem to classify hand written digits into their ten respective classes. Take a look at figure 9 to get an impression. The input dataset consists of 1797 images which have dimension 8x8 (eight pixels in width and height) and their labels as a number in the range $0-9$ (ten labels). We compare four different models, namely "SVM without a kernel", "SVM with the Gaussian RBF kernel", "FFNN" and "Stochastic gradient descend linear regression (SGDC)". To gather the accuracy we are performing a 10-Fold on the dataset as explained in the previous section to reduce overfitting. Results:

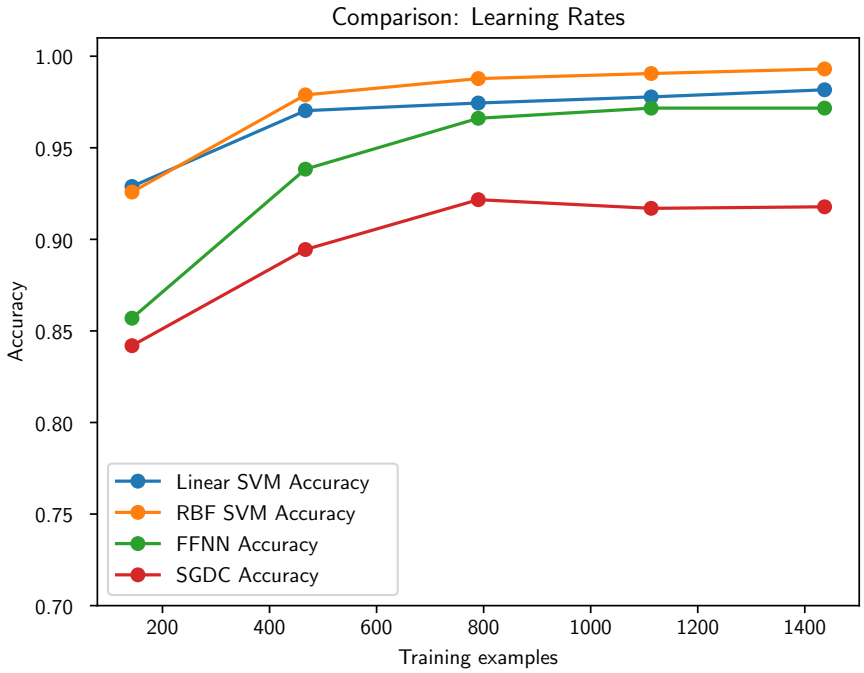

Figure 10: Learning rates of the different machine learning models used.

1) The SVM with no kernel and soft margin penalty $C = 1.0$ reached an accuracy of 98.17%. It converges quite fast and performs significantly better than SGDC

2) The SVM with Gaussian RBF kernel and $\gamma = 0.001$ is the overall "winner" and reaches a final accuracy of 99.30%.

3) The FFNN has the steepest ascend of the four compared models thus converges slower than the both SVM based methods. Neural nets usually need more data to train than a SVM, maybe the dataset of 1797 images too small for the FFNN to converge.

4) SGDC is the worst model in our test with a final accuracy of 91.78%.

4.3 Tuning hyper-parameters of regressors

Hyper-parameters are parameters constant for a machine learning model. That means those will not be learnt by the model but have to be specified by the user (see the γ parameter of the SVM with RBF kernel). Guessing the right parameters can be a tedious task, even if you have a lot of knowledge in this topic. It requires a lot of fine tuning and gut feeling. Luckily there exists some common methods to solve this task such as **grid search**.

In grid search all possible hyper-parameters, say α and β, get filled with guessed parameters and grouped together. Grid search then evaluates the error function of the model using cross validation and picks the best pair. Usually a range for each parameter is specified so the algorithm just samples random pairs of hyper-parameters until the accuracy of the learning model reaches its maximum.

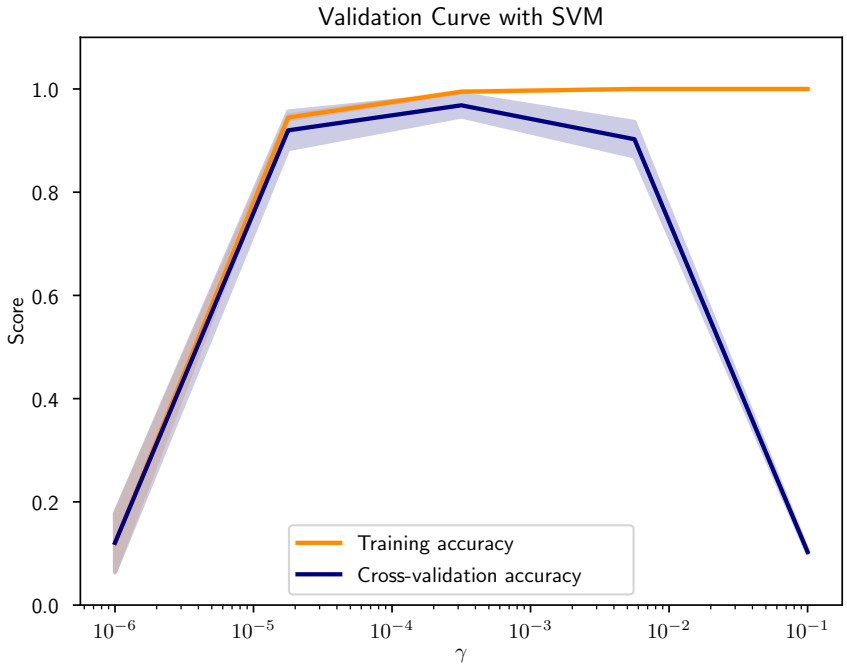

Figure 11: Validation curve on the γ hyper-parameter of the SVM with RBF kernel on the digit dataset.

By using this technique we found out that the SVM with RBF kernel and $\gamma = 0.001$ works best on the digits dataset. It is possible to plot this process of finding the right parameters into a **validation plot**. For this we plot the cross validation- and the **training** accuracy

in respect to the hyper-parameters. The training accuracy is calculated by evaluating the model on the training dataset which the model was trained on. As an example we plotted the influence of the γ parameter on the SVM with RBF kernel and of the on the digits dataset in figure 11. When γ is very small the cross validation accuracy and the training accuracy are very small (under fitting). If the γ parameter is around 10^{-3} and 10^{-4} the training accuracy and the cross validation accuracy are quite high and on one level. You can say the model is performing well. If we choose $\gamma > 10^{-2}$ too high we we see that the model overfits heavily (training accuracy is way above cross validation accuracy). Hence we can conclude that the perfect choice for γ is around $\gamma = 0.001$.

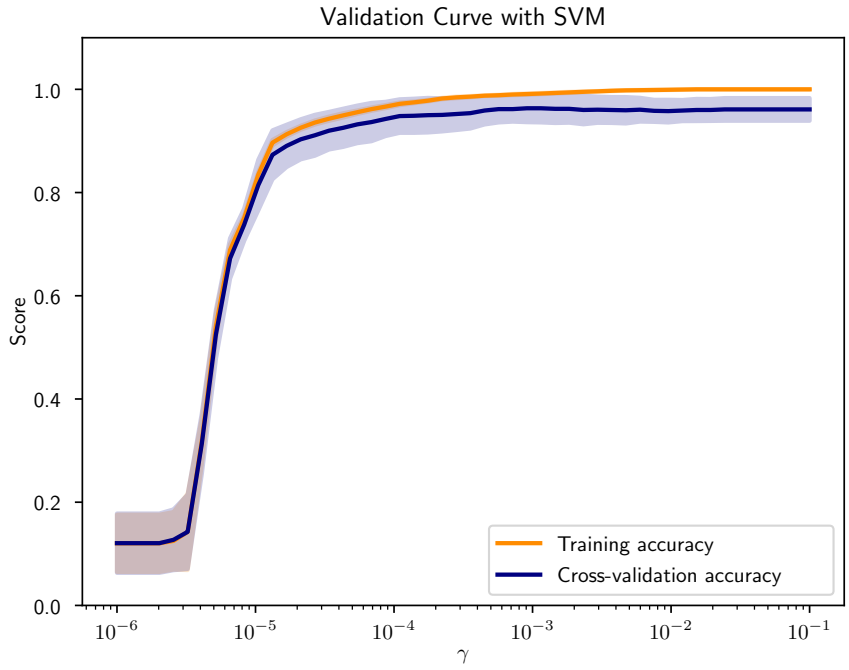

Figure 12: Validation curve on the C hyper-parameter of the SVM without a kernel on the digit dataset.

Lets move on to the SVM without a kernel and a soft margin as seen in figure 12. We plot the accuracy of the model in respect to the penalty C and found out that any $C > 10^{-2}$ would achieve similar good results. You can see this behaviour in figure 12.

Note that if $C < 10^{-5}$ the model fails to classify a lot of the samples: the training- and cross validation accuracy are both very low (underfitting).

4.4 Comparison of the models

The different models in this paper perform differently on certain problems, thus have different use cases. Linear regression for instance is very fast (even in constant time), easy to understand and is not prone to overfitting as much as other models. The simplicity comes with a downside: the model cannot represent complex functions, or non linear relations. For these properties linear regression can be used for a first look at a dataset or for datasets with have a lot of features where other models would take for ever to train.

Neural networks are very powerful as they can model highly complex (nonlinear) correlations. You can create a highly complex neural net without doing much to optimize the network because it is fairly good by default. Downsides are the horrible overfitting and the long training time due to a lot of calculations. Even though the current "state of the art" is using graphic processors the training time is still in a different order of magnitude compared to linear regression. Additionally, a complex neural net might be very hard to comprehend even though it works well (decision trees on the other hand are very easy to understand).

Generally speaking SVMs and FFNNs address the same problem. Though they have major differences: the size of a neural network is fixed because it is a parameterized model. Conversely, kernelized SVMs are non-parametric and the choice of support vectors determines the size of the model. Note that FFNNs are very "data hungry" too.

5 Conclusion

Machine learning and supervised learning models have been around for a long time but the computational power to use them was missing. But as of today, nearly everybody owns a computer or smartphone capable of handling the computational costs needed. The development of programming languages that enables users to prototype their source code rapidly (like Python or R) and the many "beautiful" machine learning libraries available like **sciki-learn**, **pandas**, **keras** lead to a big community of young and old researchers analyzing data everywhere in the world. Even though it is easy to use the libraries it is not trivial to understand the mathematics behind the subject. When writing this research paper we realized that finding good and understandable resources is a tall order. There are also plenty of people who upload bad resources to the internet without actually understanding what they are writing about. We encourage the reader to learn the vocabulary and have an intuition of the mathematics going on behind the scenes. With these two dependencies met we can lead an uncompromising discussion about topics concerning machine learning and artificial intelligence. This paper aims to be a starting point for new machine learning developers or people interested in machine learning.

References

[CST00] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*. Cambridge University Press, New York, NY, USA, 2000.

[DL06] Clara Dismuke and Richard Lindrooth. Ordinary least squares. *Methods and Designs for Outcomes Research*, 93:93–104, 2006.

[Hay98] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[Hof06] Martin Hofmann. Support vector machines — kernels and the kernel trick. 2006.

[Mus97] Marco Muselli. On convergence properties of pocket algorithm. *IEEE Transactions on Neural Networks*, 8(3):623–629, 1997.

[RN02] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (International Edition)*. Pearson US Imports & PHIPEs, November 2002.

[TS13] Yann LeCun Tom Schaul, Sixin Zhang. *No More Pesky Learning Rates*. Feb 2013.