MaMa Summary

February 2, 2024

Contents

1 Classification Learning

1.1 vocabulary

- domain set X: set of possible inputs
- classes or label set Y: target of classification
- data points or samples: $x \in X$
- features or attributes: entries of x
- training set $S \subseteq X$: pairs (x, y) of data points x and labels y
- classifier h: function $X \to Y$

1.2 loss and error

Let h be a classifier $h : X \to Y$.

- loss function: $l(y, y') \geq 0$ for $y, y' \in Y$.
- zero-one loss:
$$l_{0-1}(y, y') = \begin{cases} 0, & \text{if } y = y' \\ 1, & \text{if } y \neq y' \end{cases}$$
- training error:
$$L_S(h) = \frac{1}{|S|} \sum_{(x,y) \in S} l(y, h(x))$$

Example 1.2.1. *Let $X = \mathbb{R}^2$ and $Y = \{-1, 1\}$. The easiest function is a linear classifier that splits X in two parts and characterizes data points accordingly. For the general case of $X = \mathbb{R}^n$, $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$ we have:*

$$h_{w,b} = sgn(w^T x + b)$$

You can get rid of the bias b by adapting the training set a little:

$$\overline{S} = (\begin{pmatrix} x \\ 1 \end{pmatrix}, y) : (x, y) \in S$$

where the new linear classifier is defined by

$$\overline{h}_{\overline{w}}(x) = sgn(\overline{w}^T \overline{x})$$

with $\overline{w} = \binom{w}{b}$.

When is it possible to achieve training error 0? (wrt zero-one-loss)
This is precisely then the case, when $\exists w \in \mathbb{R}^n$ s.t. $\forall (x,y) \in S$

- if $y = 1$ then $w^T x \geq 0 \leftrightarrow$ if $y = 1$ then $w^T x > 0$
- if $y = -1$ then $w^T x < 0$

Both can be generalized by saying

$$y \cdot w^T x > 0$$

or

$$\exists w : y \cdot w^T x \geq 1 \ \forall (x,y) \in S$$

A training set with 0 training error is called separable.

1.3 logistic regression

Logistic regression computes a linear classifier.
direct way:
Look for a $w \in \mathbb{R}^n$ such that the training error is minimised, i.e.

$$\min_{w \in \mathbb{R}^n} \frac{1}{|S|} \sum_{(x,y) \in S} h_{0-1}(y, h_w(x))$$

Definition 1.3.1 (logistic function)**.** The logistic function $\phi_{sig} : \mathbb{R} \to [0,1]$ is defined by

$$z \mapsto \frac{1}{1 + e^{-z}}$$

We then solve

$$\min_{w \in \mathbb{R}^n} \frac{1}{|S|} \sum_{(x,y) \in S} -\log_2(\phi_{sig}(y w^T x))$$

Lemma 1.3.2. *For all training sets $S \subseteq \mathbb{R}^n \times \{-1, 1\}$ it holds that*

$$\frac{1}{|S|} \sum_{(x,y) \in S} h_{0-1}(y, h_w(x)) \leq \frac{1}{|S|} \sum_{(x,y) \in S} -log_2(\phi_{sig}(y w^T x))$$

1.4 training error and real life

A classifier is only good, if it performs good on new data. To test your classifier you need to evaluate the classifier on data it hasn't seen during training. This is called the test error. The question now is, how the data should be split into test and training data. This is a difficult question and should be evaluated for each use case separately. A part from the training set should also be used for validation if there are degrees of freedom in the chosen classifier.

1.5 Quadratic classifier

This is another binary classifier. In this case wa search for a matrix $U \in \mathbb{R}^{n \times n}$ of weights, a vector $w \in \mathbb{R}^n$ and a bias $b \in \mathbb{R}$. The classifier is then defined by

$$h(x) = sgn\left(\sum_{i,j=1}^{n} u_{i,j} x_i x_j + \sum_{i=1}^{n} w_i x_i + b\right) = sgn(x^T u x + w^T x + b)$$

To simplify things we redefine the training set S by $\overline{S}$ containing all samples (x, y) with $x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$ and y analogously defined. Redefine x by

$$\overline{x} = (x_1^2, x_1 x_2, x_1 x_3, .., x_n^2, .., x_n, 1)$$

Doing this will transform the quadratic classifier into a linear one.

1.6 nearest neighbour classifier

The training set again is defined by $S \subseteq X \times Y$. When adding a new data-point, we check what class the nearest data point belongs to and add the new point to the same class. In the case of k-nearest-neighbour we check the classes of the k nearest points. The problem is that the training set must be in memory the entire time. This makes the learning potentially very slow.

1.7 decision tree classifier

The classifier of a decision tree works as follows:

1. Set $v = r$ where r is the tree's root
2. while v is not a leaf do
3. Let (i, j) be the decision rule of v
4. If $x_i \leq t$ set $v = v_L$ else set $v = v_R$
5. end while

6. Output the class $c(v)$ of v

The following now describes how a tree can be learned.
The idea is to start with a single node r where the class $c(r)$ is just the majority class in S.
Iteratively decide for each leaf v, if it is beneficial to split it into two child nodes. Let S_v be the path of the training set that, following the existing decisions, ends up in v. The split into S_L and S_R would then be defined by a feature i and a threshold t. We define the gain of the split by

$$gain(v,i,t) = \gamma(S_v) - \left(\frac{|S_L|}{|S_R|}\gamma(S_L) + \frac{|S_R|}{|S_L|}\gamma(S_R)\right)$$

where γ is a inhomogeneity measure. This can be defined in multiple ways.

Definition 1.7.1. The inhomogeneity can be defined by the training error

$$\gamma(S_v) = 1 - \max_{y\in Y} p(y, S_v)$$

where

$$p(y, S_v) = \frac{|\{(x,y') \in S_v :\ y' = y\}|}{|S_v|}$$

Definition 1.7.2 (gini impurity)**.** This is used in scikit-learn.

$$\gamma(S_v) = 1 - \sum_{y\in Y} p(y, S_v)^2$$

Definition 1.7.3 (entropy)**.**

$$\gamma(S_v) = -\sum_{y\in Y} p(y, S_v) \cdot \log_2(y, S_v)$$

1.8 loss functions

So far, we only looked at the zero-one-loss $l_{0-1}(y,y') = \begin{cases} 1, \ y \neq y' \\ 0, \ y = y' \end{cases}$ where y is the true class and y' is the prediction.
Consider the use case of a spam filter. This is a binary classifier that checks whether a new mail is spam or not. This admits two kind of errors:

- false-positive: good mail is classified as spam
- false-negative: spam mail is classified as good

The case of false-positive is more serious in this example. This needs to be applied to the loss function.

$$l(y,y') = \begin{cases} 0, \ y = y' \\ 10, \ \text{if } y \text{ is good and } y' \text{ spam} \\ 1, \ \text{if } y \text{ is spam and } y' \text{ is good} \end{cases}$$

Example 1.8.1 (loss functions in regressions)**.** *Consider a predictor $h : \mathbb{R}^n \to \mathbb{R}$. The (mean)square-loss is defined by*

$$l(y,y') = (y-y')^2$$

and the (mean) absolute loss

$$l(y,y') = |y-y'|$$

1.9 A statistical model

Let X and Y be sets as above. We define a probability distribution D on $X \times Y$ with the following assumptions

- D is unknown
- if D were defined on X only, then $\mathbb{P}[p] = \begin{cases} \text{large if picture shows cat or dog} \\ \text{small if not} \end{cases}$
  where p is a picture. Instead D is defined on $X \times Y$ because there might be some uncertainty in X
- iid: data points of the training set are drawn from D independently
- D is fixed

A classifier $h^* : X \to Y$ works well on new data if

$$L_D(h^*) = \mathbb{E}_{(x,y)\sim D}[l(y, h^*(x))]$$

is small. This is called true risk or generalization error. For classification using the zero-one-loss this simplifies to

$$L_D(h^*) = \mathbb{P}_{(x,y)\sim D}[h^*(x) \neq y]$$

1.10 Bayes error and classifier

The Bayes error is the smallest error that any classifier can achieve.

$$\varepsilon_{bayes} = \inf_h L_D(h)$$

A Bayes classifier h^* such that $L_D(h^*) = \varepsilon_{bayes}$ is called a Bayes classifier.

Theorem 1.10.1. *For all classifiers $h : X \to Y$ it holds that*

$$L_D(h) \geq L_D(h_{bayes})$$

2 PAC learning

Let H be a set of classifiers. Draw a training set S from D. The goal is to minimize $h_S = \arg\min_{h \in H} L_S(h)$. This is called the method of empirical risk minimization.

2.1 empirical risk minimisation

Lemma 2.1.1 (Hoeffding's inequality)**.** *Given independent random variables $X_1, ..., X_m : \Omega \to [a, b]$ with $\mathbb{E}[X_i] = \mu$ for all $i \in [n]$. Then*

$$\mathbb{P}\left[\left|\frac{1}{m}\sum_{i=1}^{m} X_i - \mu\right|\right] \leq 2 \cdot \exp\left(-2\frac{m\varepsilon^2}{(b-a)^2}\right)$$

We now want to figure out the connection between $L_S(h)$ and $L_D(H)$. Let $|S| = m$ and $S = \{(x_1, y_1), ...(x_m, y_m)\}$. Define $X_i = l(y_i, h(x_i))$. Then the training error is given by

$$\frac{1}{m} = \sum_{i=1}^{m} X_i = L_S(h)$$

Furthermore, $\mathbb{E}[X_i] = \mathbb{E}_{(x,y)\sim D}[l(y, h(x))] = L_D(h)$. Assuming zero-one-loss, Hoeffing implies

$$\mathbb{P}[|L_S(h) - L_D(h)| \geq \varepsilon] \leq 2 \cdot \exp(-2m\varepsilon^2) := \delta$$

Then $\varepsilon = \sqrt{\frac{\ln(\frac{2}{\delta})}{2m}}$

Theorem 2.1.2. *With probability $\geq 1 - \delta$ it holds that*

$$|L_S(h) - L_D(h)| \leq \sqrt{\frac{\ln(\frac{2}{\delta})}{2m}}$$

This does not however imply that with high probability $L_S(h_S) \approx L_D(h_S)$. It is however true for the test-set: with probability $1-\delta$ it holds that $|L_T(h_S) - L_D(h_S)| \leq \sqrt{\frac{\ln(\frac{2}{\delta})}{2|T|}}$.

2.2 error decomposition

Assume we have a training set S and a test set T and the training error h_S has been minimised by empirical risk minimisation. We are interested in the generalisation error $L_D(h_s)$. For example assume a training error of 9% and a test error of 12%. We can decompose the unknown generalisation error as follows:

$$L_D(h_S) = \underbrace{(L_D(h_S) - L_T(h_S))}_{\text{small if } T \text{ reasonably large}} + \underbrace{(L_T(h_S) - L_S(h_S))}_{\text{generalisation gap,12\%}-9\%} + \underbrace{L_S(h_S)}_{9\%}$$

A large generalisation gap means that the classifier learns the training set and not the underlying distribution D. That means the classifier is overfitting.

2.3 finitely many classifiers

Lemma 2.3.1. *Let H be a set of classifiers. Assume that for every $\varepsilon, \delta > 0$, there is an m s.t. when drawing a sample S of size at least m then, with a probability at least $1-\delta$ it holds that*

$$\sup_{h \in H} \left| L_D(h) - LS_(H) \right| \leq \frac{\varepsilon}{2}$$

Then with probability at least $1-\delta$ it holds that

$$L_D(h_S) \leq \inf_{h \in H} L_D(h) + \varepsilon$$

These assumptions are called the uniform convergence property.

Theorem 2.3.2. *Let $\varepsilon, \delta > 0$ and H be a finite class of classifiers. Let the training set S have size $m \geq \frac{2}{\varepsilon^2} \ln \frac{2|H|}{\delta}$. Then with probability at least $1-\delta$ it holds that*

$$L_D(h_S) \leq \min_{h \in H} L_D(h) + \varepsilon$$

This proves that empirical risk minimisation works.

2.4 probably approximately correct (pac)

Let H be a set of classifiers. H is agnostically PAC learnable if
(informal) there is a learning algorithm that provided the training set is large enough, returns an almost optimal classifier.

(formal) $\exists m_H : (0,1)^2 \to \mathbb{Z}_+$ and a learning algorithm A (think risk minimisation) s.t. $\forall \varepsilon, \delta \in (0,1)$ and $\forall$ probability distributions D on $X \times Y$ if S with $|S| \geq m_H(\varepsilon, \delta)$ is iid drawn from D then with probability $\geq 1 - \delta$ it holds that

$$L_D(A(S)) \leq \inf_{h \in H} L_D(h) + \varepsilon$$

Classes H that are finite are PAC-learnable. Also classes that satisfy the uniform convergence property are PAC-learnable as well.

2.5 VC-dimension

Very informally this dimension gives an idea of how powerful a class of classifiers can be. The VC-dimension is only defined for binary classifiers, say with classes 0 and 1. Let H be a set of binary classifiers (for example axis-parallel rectangle classifiers, H $= \{I_R : R$ axis parallel rectangle$\}$, so $I_R(x) = \begin{cases} 1, x \in R \\ 0, x \notin R \end{cases}$). Formal definition: We say that H shatters a set $C \subset X$ if

$$\{h|_C : h \in H\} = H|_C \overset{!}{=} \{f : C \to \{0,1\}\} \Leftrightarrow |H|_C| = 2^{|C|}$$

The VC-dimension of H is then the largest d s.t.

$$\exists C \subseteq X \text{ of } |C| = d \text{ s.t. } C \text{ is shattered by } H$$

The dimension of H is infinite if $\forall k \in \mathbb{Z}_+ \; \exists C \subseteq X$ of $|C| = X$ that is shattered by H.
It is easy to see that the VC-dimension of the axis parallel rectangle classifier is at least 4 and in fact it is exactly for as we will see below.
Consider any set of 5 points in $\mathbb{R}^2$. Let p_1 be the point with the largest y-coordinate and p_2 be the point with the smallest y-coordinate. p_3 has the largest x coordinate and p_4 has the smallest. Notice that each pair of points may be identical. Finally, there is a fifth point q that is somewhere in the rectangle spanned by those four points. If we give points p_1 to p_4 the class 1 and q class 0, then we can't shatter any set of five points implying that VC-dim ≤ 4.

Example 2.5.1. *Let $X = \mathbb{R}$. Define*

$$H = \{h_{[a,b]} : a \leq b\}$$

where

$$h_{[a,b]}(x) = \begin{cases} 1 \; x \in [a,b] \\ 0; \; else \end{cases}$$

Obviously $VC(H) \geq 2$ because there is always an interval that contains two points. Indeed the dimension is exactly 2 because if we have $x_1 < x_2 < x_3$ with classes 1, 0 and 1, there is no classifier that is correct.

The class of homogeneous linear classifiers in $\mathbb{R}^d$

$$h_w : x \mapsto sgn(w^T x)$$

has VC-dimension of d.

2.6 Fundamental Theorem of PAC-learning

Theorem 2.6.1. *Let H be a set of binary classifiers. Then H is PAC-learnable iff the VC dimension of H is finite.*

Theorem 2.6.2. *Let H be a set of binary classifiers with $VC(H) = d < \infty$. Then $\exists c > 0$ s.t. $\forall \varepsilon, \delta > 0$ it holds that with probability at least $1 - \delta$*

$$L_D(h_S) \leq \inf_{h \in H} L_D(h) + \sqrt{c \frac{d + \log(\frac{1}{\delta})}{m}}$$

Definition 2.6.3. Let H be a set of binary classifiers. We define the growth function

$$\tau : \mathbb{N} \to \mathbb{N}$$

with

$$\tau(n) = \max_{C \subseteq X, |C| = m} |H|_C| = \max |\{h|_c : C \to \{0,1\} : h \in H\}|$$

Lemma 2.6.4. *Let H be a set of binary classifiers with $VC(H) = d < \infty$. Then*

$$\tau(m) \leq \sum_{i=0}^{d} \binom{m}{i} \; \forall m \in \mathbb{N}$$

In particular if

$$m \geq d + 1$$

then $\tau(m) \leq (d+1) \cdot m^{d+1}$.

Theorem 2.6.5. *Let H be a set of binary classifiers. Then for every $\delta > 0$ with probability of at least $1 - \delta$ over the choice $S \sim D^m$ of the training set it holds that*

$$\sup_{h \in H} |L_D(h) - L_S(h)| \leq \frac{4 + \sqrt{\log(\tau_H(2 \cdot m))}}{\delta \sqrt{2 \cdot m}}$$

Lemma 2.6.6. *Let H be a set of binary classifiers. Then for every $\delta > 0$ with probability at least $1 - \delta$ over the choice $S \sim D^m$ of the training set it holds that*

$$\sup_{h \in H} |L_D(h) - L_S(h)| \leq \frac{8 + 2\sqrt{\log(\tau_H(2 \cdot m))}}{\delta \sqrt{m}}$$

Note: The lemma is a weaker bound than the one in above theorem and can thus be ignored.

Theorem 2.6.7. *Let H be a set of binary classifiers with the domain set X of VC-dimension $\geq 2m$ and a learning algorithm A. Then there is a distribution D on $X \times \{0, 1\}$ s.t.*

1. *$\exists h^* \in H$ with $L_D(h^*) = 0$*
2. *with probability $\geq \frac{1}{7}$ and a choice of training set S with $|S| = m$ we have:*

$$L_D(A(S)) \geq \frac{1}{8}$$

2.7 VC-dimension of linear classifiers

Theorem 2.7.1. *The class of linear classifiers in $\mathbb{R}^d$*

$$H = \{x \mapsto sgn(w^T x) : \ w \in \mathbb{R}^d\}$$

has VC-dimension d.

2.8 neural networks

For an input vector x, weights w and a bias b the output of an artificial neuron is $w^T x + b$. A classification network is made up of an input layer, the hidden layer where the neurons are and the output layer. In such a case, the weight vector w would be a weight matrix $W^{(1)}$. The each neuron then has a weight to the output neuron which also has a bias $b^{(2)}$ so the final output is then given by $sgn(W^{(2)} h + b^{(2)})$ where h is the output of the hidden layer. One can notice that this is simply an affine function. Hence neurons are augmented by an activation function, e.g. $\text{ReLU}(x) = \max(0, x)$ (=Rectified linear unit).

If the activation function is non-linear one can calculate more difficult stuff than with normal affine functions.
Activation functions $\sigma : \mathbb{R} \to \mathbb{R}$ are applied component-wise. It is important to note, that neurons on the same layer must share the same activation function. The necessity of activation functions becomes inherently apparent, when one tries to compute XOR with a neural network. Even though XOR is not separable, one can compute the XOR function using a neural network, if an activation function like ReLU is used.
Modern neural networks consist of many hidden layers, each equipped with ReLU while the output layer usually uses the logistic function or softmax. A logistic function output does not give a simple class but a confidence level instead. This is a difference to the classical classification task. Quite similarly, one can use a softmax function defined by

$$z \mapsto \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

This gives a distribution over the classes instead.

Theorem 2.8.1. *Consider the class F of neural networks with $L-1$ fully connected hidden layers and one output layer. Each hidden layer uses the ReLU function as an activation function whereas the output has sgn as activation function. Let the number of weights be upper bounded by N. Then*

$$VCdim(F) \in \mathcal{O}(NL \cdot \log N)$$

There is also a lower bound given by

$$VCdim(F) \in \Omega\left(NL \cdot \log\left(\frac{N}{L}\right)\right)$$

Using the tanh *activation function, we find*

$$VCdim(F) \in \Omega(N^2)$$

3 stochastic gradient descent

3.1 convexity

Definition 3.1.1 (convex set)**.** A set M is convex, if $\forall x, y \in M,\ \forall \lambda \in [0,1]$,

$$\lambda x + (1-\lambda) y \in M$$

Definition 3.1.2 (convex function)**.** A function f is convex if $\forall x_1, x_2$ in the convex preimage of f and $\forall \lambda \in [0,1]$ we have

$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda)f(x_2)$$

There is a relation between convex sets and convex functions.

Remark 3.1.3. Let C be a convex set and f be a function on C, then the epigraph $epi(f) = \{(x,y) : x \in C, y \geq f(x)\}$ is a convex set iff f is a convex function.

3.2 convex optimization

Definition 3.2.1. Let $K \subseteq \mathbb{R}^n$ be convex, $f : K \to \mathbb{R}$ convex. A convex optimization problem searches for

$$\inf_{x \in K} f(x)$$

Remark 3.2.2. Any local minimum of a convex function is a global minimum.

Lemma 3.2.3. *Let f be a doubly differentiable function with compact preimage. The following statements are equivalent*

- *f is convex*
- *f' is monotonically non-decreasing*
- *f'' is non-negative*

Lemma 3.2.4. *Let $g : \mathbb{R} \to \mathbb{R}$ be convex and let $x \to w^T x + b$. Then $f : \mathbb{R}^n \to \mathbb{R}$ defined by $f(x) = g(w^T x + b)$ is convex as well.*

As a consequence from above lemma, $w \mapsto \log(1 + e^{-yw^t x})$ is convex.

Lemma 3.2.5. *Let $C \subset \mathbb{R}^n$ be a convex set, $f_1, ..., f_n$ are convex functions from C to $\mathbb{R}$ and weights $w_1, ..., w_n \geq 0$, then*

$$f = \sum_{i=1}^{n} w_i f_i$$

is convex.

As a consequence from this lemma, the logistic loss function is convex.

Lemma 3.2.6. *Let I be some index set and C be a convex set. Let $f_i : C \to \mathbb{R}$ be a family of convex functions. Then*

$$f : x \mapsto \sup_{i \in I} f_i(x)$$

is a convex function.

3.3 Strong convexity

Let $f : K \to \mathbb{R}$ where K is convex, $\mu > 0$ f is μ-strongly convex if

$$\frac{\mu}{2}\lambda(1-\lambda)||x-y||_2^2 + f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)\ \forall x, y \in K,\ \lambda \in [0,1]$$

Lemma 3.3.1. *The map $x \mapsto ||x||_2^2$ is 2-strongly convex.*

Lemma 3.3.2. *Let $f : K \to \mathbb{R}$ be convex and $g : K \mapsto \mathbb{R}$ μ-strongly convex then*

1. *$f + g$ is also strongly convex*
2. *$c \cdot g$ is $c \cdot \mu$-strongly convex for some $c > 0$.*

Lemma 3.3.3. *Let $f : K \to \mathbb{R}$ be differentiable, $K \subseteq \mathbb{R}^n$ convex and open. Then f is (strongly) convex iff*

$$f(y) \geq f(x) + \nabla f(x)^T (y-x)(+\frac{\mu}{2}||x-y||_2^2)$$

3.4 gradient descent

Suppose $\overline{x}$ is the output of the gradient descent to minimize a convex function f where the global minimum is located at x^*. Then

$$f(\overline{x}) - f(x^*) = \varepsilon$$

where $\varepsilon \sim e^{-t}$. However in Machine Learning, where large datasets are regularly used, computing one gradient can take a very large amount of resources, hence we need another solution.
To compute the direction in which to iterate, pick $(x,y) \in S$ at random and use $\nabla L_{(x,y)}(w)$ which is in expectation the same value as the gradient in the normal gradient descent algorithm. This is called the stochastic gradient descent. Instead of computing

$$\frac{1}{|S|}\sum_{(x,y)\in S} \nabla L_{(x,y)}(w)$$

as in the classical gradient descent, we compute

$$\frac{1}{|M|}\sum_{(x,y)\in M} \nabla L_{(x,y)}(w)$$

for a small subset M of S.

3.5 convergence of stochastic gradient descent

We make the following assumptions:

- the loss function is differentiable
- the loss function is μ-strongly convex
- $\sup_w ||\nabla L_{(x,y)}(w)||^2 \leq B$ for all $(x, y) \in S$.

The learning rate is defined by

$$\eta_1 \geq \eta_2 \geq ... > 0 \text{ s.t. } \sum_{t=1}^{\infty} \eta_t = \infty \text{ but } \sum_{t=1}^{\infty} \eta_t^2 < \infty$$

Theorem 3.5.1. *Under these assumptions*

$$\mathbb{E}_{1,\dots,t}[||w^{(t+1)} - w^*||_2^2] \to 0 \textit{ as } t \to \infty$$

where w^* is the global minimum.

Lemma 3.5.2. *Define $\varepsilon = \mathbb{E}_{1,\dots,t-1}[||w^{(t)} - w^*||_2^2]$. Then*

$$\varepsilon_t \leq \varepsilon_{t-1}(1 - \eta_{t-1}\mu) + \eta_{t-1}^2 B$$

Lemma 3.5.3.

$$\varepsilon_{T'+k+1} \leq \varepsilon_{T'} \prod_{t=T'}^{T'+k} (1 - \eta_t \mu) + \sum_{t=T'}^{T'+k} \eta_t^2 B$$

4 neural networks

4.1 back propagation

We now introduce a method to train a neural network with SGD.
For that we define a matrix W which collects all weights and biases. Assume we have K layers. At the input we have $x \in \mathbb{R}^{n_0}$. The first layer produces

$$g^{(1)}(x) = W^{(1)}x + b^{(1)}$$

but the output of the first hidden layer is

$$f^{(1)}(x) = \sigma_1(g^{(1)}(x))$$

where σ is the activation function, which is ReLU for the hidden layer and the logistic loss function for the output. This is easily generalized for the l-th hidden layer. We generally prefer the notation $g^{(k)} = w^{(k)} f^{(k-1)} + b^{(k)}$ because the x-term is always the same, so we omit it. We generally define the dimension of the k-th layer by n_k. In particular $n_K = 1$. We need to compute the values of

$$\frac{\partial L_{(x,y)}}{\partial w_{i,h}^{(k)}} \text{ and } \frac{\partial L_{(x,y)}}{\partial b_j^{(k)}}$$

Starting at the output, we have

$$f_1^{(K)} = \sigma_K(g_1^{(K)}) = \sigma_K(W^{(K)} f^{(K-1)} + b_1^K)$$

Also, $L_{(x,y)} = l(y, f_1^{(K)})$. We find

$$\frac{\partial L_{(x,y)}}{\partial b_1^{(k)}} = \frac{\partial L_{(x,y)}}{\partial f_1^{(K)}} \cdot \frac{\partial f_1^{(K)}}{\partial g_1^{(K)}} \cdot \frac{\partial g_1^{(K)}}{\partial b_1^{(k)}}$$

and

$$\frac{\partial L_{(x,y)}}{\partial w_{1,h}^{(k)}} = \frac{\partial L_{(x,y)}}{\partial f_1^{(K)}} \cdot \frac{\partial f_1^{(K)}}{\partial g_1^{(K)}} \cdot \frac{\partial g_1^{(K)}}{\partial w_{1,h}^{(k)}}$$

where the two common terms in the calculations are defined as $\delta_1^{(K)}$. In particular, we find

$$\frac{\partial L_{(x,y)}}{\partial b_1^{(k)}} = \delta_1^{(K)}$$

and

$$\frac{\partial L_{(x,y)}}{\partial w_{1,h}^{(k)}} = \delta_1^{(K)} \cdot f_h^{(K-1)}$$

We know concentrate on the k-th layer for an arbitrary k. Define

$$\delta^{(k)} = \nabla_{g^{(k)}} L_{(x,y)} = \left(\frac{\partial L_{(x,y)}}{\partial g_1^{(k)}}, \dots, \frac{\partial L_{(x,y)}}{\partial g_{n_k}^{(k)}} \right)$$

This the shows

$$\frac{\partial L_{(x,y)}}{\partial b_h^{(k)}} = \frac{\partial L_{(x,y)}}{\partial g_h^{(k)}} \cdot \frac{\partial g_h^{(k)}}{\partial b_h^{(k)}} = \delta_h^{(k)}$$

and

$$\frac{\partial L(x,y)}{\partial w_{h,i}^{(k)}} = \frac{\partial L_{(x,y)}}{\partial w g_h^{(k)}} \cdot \frac{\partial g_h^{(k)}}{\partial w_{h,i}^{(k)}} = \delta_h^{(k)} f_i^{(k-1)}$$

We hence want to figure out how we can compute the δ.

$$\delta_h^{(k)} = \frac{\partial L_{(x,y)}}{\partial g_h^{(k)}} = \sum_{i=1}^{n_{k+1}} \frac{\partial L_{(x,y)}}{\partial g_i^{(k+1)}} \cdot \frac{\partial g_i^{(k+1)}}{\partial g_h^{(k)}} = \sum_{i=1}^{n_{k+1}} \delta_i^{(k+1)} \cdot \frac{\partial g_i^{(k+1)}}{\partial g_h^{(k)}}$$

But now $g_i^{(k+1)} = \left(W^{(k+1)} f^{(k)}\right)_i + b_i^{(k+1)}$ and $f^{(k)} = \sigma_k(g^{(k)})$. For the calculation of δ we need

$$\frac{\partial g_i^{(k+1)}}{\partial g_h^{(k)}} = \frac{\partial g_i^{(k+1)}}{\partial f_h^{(k)}} \frac{\partial f_h^{(k)}}{\partial g_h^{(k)}} = w_{i,h}^{(k+1)} \cdot \sigma_k(f_h^{(k)})$$

which implies that

$$\delta_h^{(k)} = \sum_{i=1}^{n_{k+1}} \delta_i^{(k+1)} \cdot w_{i,h}^{(k+1)} \cdot \sigma_k'(f_h^{(k)}) = \left(W^{(k+1)T} \cdot \delta^{(k+1)}\right)_h \cdot \sigma_k'(f_h^{(k)})$$

Generally, we can write

$$\delta^{(k)} = \left(W^{(k+1)T} \cdot \delta^{(k+1)}\right) \odot \sigma_k^*(f^{(k)})$$

where $\odot$ is the Hadamard matrix product defined by

$$(A \odot B)_{ij} = A_{ij} \cdot B_{ij}$$

4.2 Loss functions for neural networks

The following loss function are options

- square loss: $L = \frac{1}{|S|} \sum_{(x,y)\in S} \left| y - f_1^{(K)}(x) \right|^2$
- cross entropy loss: $L = -\frac{1}{|S|} \sum_{(x,y)\in S} y \cdot \log(f_1^{(K)}(x)) + (1-y)\log(1 - f_1^{(K)}(x))$

Both are non-negative. One can show, that the second function is better for fast learning using the back propagation algorithm.

Definition 4.2.1 (soft max)**.** Soft max is an activation function for the output layer that does not return a fixed class but rather a probability distribution over all possible classes. It is defined by the mapping of $z = (z_1, ..., z_k) \in \mathbb{R}^k$ to $[0,1]^k$ under

$$z \mapsto \left(\frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \right)$$

Usually one should use log-likelihood loss

$$L = -\frac{1}{|S|} \sum_{(x,y)\in S} \sum_{j=1}^{n_j} \log f_j^{(K)}(x)$$

<u>Benefits of Cross Entropy loss</u>

This is generally defined by

$$-\frac{1}{|S|} \sum_{(x,y)\in S} (y \cdot \log h(x) + (1-y)\log(1-h(x)))$$

Now, let p, q be probability distributions. We define

$$H(p,q) = -\mathbb{E}_p[\log q] = -\sum_{\omega\in\Omega} p(\omega)\log q(\omega)$$

Now let h be a classifier. $h(x)$ is the confidence that x is class 1. We define

$$q(h(x)) = \begin{cases} h(x), y = 1 \\ 1 - h(x), y = 0 \end{cases}$$

The hidden distribution that generates (x, y) is now denoted by p. Then the marginal probability is

$$p(x) = p((x,0)) + p((x,1))$$

This allows us to rewrite the cross-entropy-loss

$$L = -\frac{1}{|S|} \sum_{(x,y)\in S} \log q(y|x)$$

Consider

$$\mathbb{E}_{S\sim p^m}[L] = \mathbb{E}_{(x_1,y_1)\sim p} ... \mathbb{E}_{(x_m,y_m)\sim p}[L] = -\frac{1}{|S|} \sum_{i=1}^{m} \sum_{(x_i,y_i)\sim p} [\log q(y_i|x_i)] = -\mathbb{E}_{(x,y)\sim p}[\log q(y|x)]$$

This can now be pulled apart along the marginal distribution and one gets the cross entropy again. This is the intuition for the following

$$\min_h L \to \min_h \mathbb{E}_x[H(p(\cdot|x), q(\cdot|x))]$$

This is minimal iff

$$H(p(\cdot|x), q(\cdot|x))$$

is minimal for all $x \in X$. This can be further seen to be equivalent to minimizing the so-called Kullback-Leibler-divergence which is a measure of how similar the two distributions p and q are. This moves the cross-entropy towards the Bayes-classifier.

Definition 4.2.2 (Kullback-Leibler)**.** $D_{KL}(p||q) = \mathbb{E}_p[\log(p/q)] = \mathbb{E}_p[\log p] - \mathbb{E}_p[\log q] = \sum_{\omega\in\Omega} p(\omega)\log(\frac{p(\omega)}{q(\omega))})$. If $q(\omega) = 0$, then $p(\omega) = 0$ and we define this value to be 0.

It can be seen that D_{KL} is non-negative and if $D_{KL}(p||q) = 0$, then $p = q$ almost everywhere. Also $D_{KL}(p||q) \neq D_{KL}(q||p)$.

Theorem 4.2.3. *Let p, q be two discrete probability distributions. Then*

- $D_{KL}(p||q) \geq 0$
- *if $D_{KL}(p||q) = 0$ then $p = q$*
- *typically $D_{KL}(p||q) \neq D_{KL}(q||p)$*

4.3 local minima

We now need to make sure the SGD actually works to train neural networks. We know that it works well for differentiable and convex functions but the loss of a neural network is neither of those.
Differentiability is no large concern. We only need a one-sided derivative which exists for functions such as ReLU. It is very unlikely that the not differentiable point is hit exactly.
But why doesn't SGD get trapped in a local minimum?
As a matter of fact, that actually can happen. But we can empirically see that a over-parameterised NN only rarely gets trapped. This is not mathematically proven and still is a field of research.
For the following we need three different kinds of matrix product.

Definition 4.3.1.

- The Hadamard product. This has been defined above and is simply the component-wise multiplication.
- The Kronecker-product. Two matrices A and B of not necessarily the same dimension are multiplied like in the tensor-product $A \otimes B$.
- The Khatari-Rao product. Let A and B have the same number of columns. Then

$$A \circ B = (A_1 \otimes B_1, A_2 \otimes B_2 ... A_n \otimes B_n)$$

  where A_i is a simpler notation for the i-th column of A.

Lemma 4.3.2. *Let $k, l, N \in \mathbb{N}$ and $kl \geq N$. Then for almost all (A, B) with $A \in \mathbb{R}^{k \times N}$ and $B \in \mathbb{R}^{l \times N}$ it holds for the Khatari-Rao product that rank$(A \circ B) = N$.*

We now introduce a matrix of perturbations called ε which can be applied to any step of a neural network. Using X as a training set, a neural network with leaky ReLU on the single hidden layer, no acctivation function on the output layer and MSE as a loss function, the following theorem holds.

Theorem 4.3.3. *For almost all (X, ε), all differentiable local minima are global minima (i.e. MSE $= 0$).*

4.4 ReLU networks and piece-wise affine functions

A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is called piece-wise affine if there are finitely many polyhedra $Q_1, ..., Q_s$ s.t. $\mathbb{R}^n = \bigcup_{i=1}^{s} Q_i$ and $f|_{Q_i}$ is affine. The smallest such s is called the piece-number of f.

Theorem 4.4.1. *The function computed by a (leaky) ReLU network with linear or (leaky) ReLU output layer is a piece-wise affine function.*

Lemma 4.4.2. *Let f, g be piece-wise affine functions with piece-numbers k and l respectively. Then*

- *$f + g$ has piece-number at most $k + l$*
- *$f \circ g$ has piece-number at most kl*

Theorem 4.4.3. *Let N be a (leaky) ReLU neural network with one input, one output and $L - 1$ hidden layers s.t. layer l has n_l nodes. Set $\overline{n} = \sum_{l=1}^{L} n_l$. The N computes a piece-wise affine function with piece-number at most*

$$2^L \prod_{l=1}^{L} n_l \leq \left(\frac{2\overline{n}}{L}\right)^L$$

Theorem 4.4.4. *A function f is continuous piece-wise affine iff there are affine functions $g_i, h_i,\ i = 1, ..., N$ s.t.*

$$f(x) = \max_i g_i(x) - \max_j h_j(x)\ \forall x \in \mathbb{R}^n$$

Theorem 4.4.5. *For all functions f such as in the last theorem, there exists a ReLU network with linear activation at the output node of $2N$ layers and s.t. every layer has at most $6N + 4$ that realises f.*

Definition 4.4.6. The following uses the terms below

- ReLU network: a neural network that uses ReLU activation in every hidden layer except the output where there is no activation
- depth number of layers except the input
- width: largest number of neurons in any layer

Lemma 4.4.7 (1). *The function* $\binom{x_1}{x_2} \mapsto \max(x_1, x_2)$ *can be computed by a ReLU network of depth 2 and width 3.*

Lemma 4.4.8 (2). *Let* N_1 *be a ReLU network of depth* d_1 *and width* $w_1 \geq 2$ *and* N_2 *be a ReLU network of depth* d_2 *and width* $w_2 \geq 2$ *such that both have only one output node. Then we can compute* $N_1 + N_2$ *with a ReLU network of depth* $\leq \max(d_1, d_2)$ *and width* $\leq w_1 + w_2$*.*

Lemma 4.4.9 (3). *Let* N_1 *be a ReLU network of depth* $d_1 \geq 1$ *and width* $w_1 \geq 2$ *and* N_2 *be a ReLU network of depth* $d_2 \geq 1$ *and* $w_2 \geq 2$ *and* $N_1 : \mathbb{R}^n \to \mathbb{R}^k$ *and* $N_2 : \mathbb{R}^k \to \mathbb{R}^l$*. Then* $N_2 \circ N_1$ *can be computed by a ReLU network of depth* $d_1 + d_2 - 1$ *and width* $\max(w_1, w_2)$*.*

Lemma 4.4.10 (4). *Let* $g_1, ..., g_N : \mathbb{R}^n \to \mathbb{R}$ *be functions that can be computed by ReLU networks of depth* $\leq d$ *and width* $\leq w$*. Then*

$$x \mapsto \max_i g_i(x)$$

can be computed by a ReLU network of depth $\leq d$ *and width* $\leq w + 2n$*.*

The collection of these lemmas allows for an easy proof of the above theorem.

4.5 Universal approximators

Theorem 4.5.1. *For every continuous function* $f : \mathbb{R}^n \to \mathbb{R}$ *and* $\varepsilon > 0$ *There is a ReLU network* N *of width* $\leq 6n + 4$ *but perhaps large depth s.t.*

$$\sup_{x \in [0,1]^n} |f(x) - N(x)| \leq \varepsilon$$

Theorem 4.5.2. *Let* $f : \mathbb{R}^n \to \mathbb{R}$ *be Lebesgue-measurable. Then there is a ReLU network* n *of width* $\leq n + 4$ *and perhaps large depth s.t.*

$$\int_{\mathbb{R}} |f(x) - N(x)| \, d\lambda(x) \leq \varepsilon$$

Theorem 4.5.3 (Hanin). *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuous piecewise affine function. If*

$$f(x) = \max_i g_i(x) - \max_j h_j(x)$$

with $i, j \in \{1, ..., N\}$ then there is a ReLU network F with linear output layer of depth at most $3N$ and width $2n + 8$ such that $F(x) = f(x)$.

4.6 Deep neural networks vs. shallow neural networks

In the following we consider the saw-tooth function. Define $\Delta : \mathbb{R} \to \mathbb{R}$ by

$$\Delta : x \mapsto \begin{cases} 2x, & x \in [0, 1/2] \\ 2 - 2x & x \in (1/2, 1] \\ 0 & x \notin [0, 1] \end{cases}$$

For any $n \in \mathbb{N}$ we can easily see, that Δ^n is an iteration of spikes. Δ is a piece-wise affine function with piece number 4.

Lemma 4.6.1. *For $l \geq 1$ we have that*

$$\Delta^l(x) = \begin{cases} 2^l x - k + 1, & x \in [(k-1)\frac{1}{2^l}, k\frac{1}{2^l}] \text{ for odd } k \in \{1, ..., 2^l\} \\ -2^l x + k, & x \in [(k-1)\frac{1}{2^l}, k\frac{1}{2^l}] \text{ for even } k \in \{1, ..., 2^l\} \\ 0, & x \notin [0, 1] \end{cases}$$

It is a piecewise affine functions with piece-number $2 + 2^l$.

We now try to calculate Δ via a ReLU neural network.

Lemma 4.6.2. $\Delta(x) = 2(ReLU(x) - 2ReLU(x - \frac{1}{2}) + ReLU(x - 1))$.

Lemma 4.6.3. *Δ^l can be realised by a ReLU network with $2l$ layers and $4n$ neurons.*

Theorem 4.6.4 (Telgarsky). *Let $L \geq 2$ and $l = L^2 + 4$.*

1. *Δ^l can be computed by a ReLU network with $\leq 2L^2 + 8$ layers and $\leq 4L^2 + 16$ neurons.*

2. *for every ReLU network N with at most L layers and at most 2^L neurons it holds that*

$$\int_{[0,1]} \left|N(x) - \Delta^l(x)\right| dx \geq \frac{1}{32}$$

4.7 Overconfident neural networks

Theorem 4.7.1. *Let N be a ReLU network with softmax output layer and let*

$$f : \mathbb{R}^n \to \mathbb{R}^K$$

be the piecewise affine function such that N computes the function $soft \circ f$, where $soft$ is the softmax function. Let $Q_1, ..., Q_s$ be the pieces of f and let $f|_{Q_t}$ be described by the affine function $x \mapsto A^{(t)}x + b^{(t)}$. If for all $t = 1, ..., s$ the matrix $A^{(t)}$ does not have identical rows, then for almost every $x \in \mathbb{R}^n$ there exists a class $k \in \{1, ..., K\}$ such that the confidence of N that αx lies in k tends to 1 as $\alpha \to \infty$, i.e.

$$\lim_{\alpha \to \infty} \frac{e^{f_k(\alpha x)}}{\sum_{l=1}^{K} e^{f_l(\alpha x)}} = 1$$

Lemma 4.7.2. *Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a piecewise affine function with pieces $Q_1, ..., Q_s$. For every $x \in \mathbb{R}^n$ there is an $\alpha \geq 1$ and a piece Q_t such that $\beta x \in Q_t$ fpr all $\beta \geq \alpha$.*

4.8 loss functions for binary classifications ($Y = \{-1, 1\}$)

The key metric is the 0-1-loss

$$l_{0-1}(y, y') = \begin{cases} 0, & y = y' \\ 1, & y \neq y' \end{cases}$$

The problem is that minimizing this function is quite hard, which is why we introduce surrogate loss functions.

- logistic loss $l(y, y') = -\log\left(\frac{1}{1+e^{-yy'}}\right)$
- square loss $l(y, y') = (y - y')^2$
- exponential loss $l(y, y') = \exp(-yy')$

Interestingly, all these loss functions may be rewritten (if necessary) such that only the product yy' appears as a variable. Thus $l(y, y') = \phi(yy')$. For example

$$l_{0-1}(y, y') = \begin{cases} 0, & yy' = 1 \\ 1, & yy' = -1 \end{cases} = 1_{yy'=-1}$$

Let's consider a classifier

$$h : X \to \{-1, 1\}$$

in fact we compute,

$$f : X \to \mathbb{R}$$

and h takes the form

$$h(x) = sgn(f(x))$$

We want to concentrate on loss functions of the type $l(y, f(x)) = \phi(yf(x))$.
So now the question is, if the classifier f minimizes the surrogate loss ϕ does this imply that f is a good classifier (i.e. small true risk)?
The Bayes error was defined by

$$\varepsilon_{bayes} = \inf_h L_D(h) = \mathbb{E}_{(x,y)\sim D} l(y, h(x))$$

and the Bayes classifier is an h that achieves ε_{bayes}. For a loss function $\phi : \mathbb{R} \to \mathbb{R}_+$, $f : \mathbb{R} \to \mathbb{R}$ we define

$$L_\phi(f) = \mathbb{E}_{(x,y)\sim D}[\phi(yf(x))]$$

Definition 4.8.1. ϕ is Bayes-consistent if for all $(f_n)_{n\in\mathbb{N}} : X \to \mathbb{R}$ s.t.

$$L_\phi(f_i) \to \inf_{g:X\to\mathbb{R}} L_\phi(g)$$

then also

$$L(sgn \circ f_i) \to \varepsilon_{bayes}$$

Theorem 4.8.2. *Let $\phi : \mathbb{R} \to \mathbb{R}_+$ be continuous, differentiable at 0 with $\phi'(0) < 0$ and convex. Then ϕ is Bayes-consistent.*

4.9 imbalanced classes

It is not always the case that all classes are equally as important. This is for example the case in the spam-filter. I we talk about binary classification tasks, we can define one class as the positive class, while the other will be called the negative class. This gives rise to the following vocabulary:

Definition 4.9.1.

true positive t_p

true negative t_n

false positive f_p

false negative f_n

We define some measures to achieve a value for each of the classes.

Definition 4.9.2. 1. true positive rate (tpr) $= \frac{t_p}{t_p+f_n}$

2. true negative rate (tnr) $= \frac{t_n}{t_n+f_p}$

3. false positive rate (fpr) $= 1 - tnr$

We can also define a precision measure

$$precision = \frac{t_p}{t_p + f_p}$$

4.10 fine-tuning after training

Given a classifier $h : X \to \{-1, 1\}$ we want to make it better somehow. Normally h outputs its confidence level, i.e. $h : X \to [0, 1]$ or it outputs some score in $\mathbb{R}$. In both cases we need to introduce a threshold t and predict 1 if $h(x) \geq t$ and otherwise -1. By adjusting this threshold, we may end up with different fpr and tpr values. The precision will also change but is slightly more unpredictable.
The second approach would be to incorporate the class imbalance/ class importance into the training itself. Going back to the example of spam classification. How can we change the training in such a way that the imbalance in classes is incorporated? The first idea would be to go away from the 0-1-loss and instead use a loss function of the form

$$l(y, y') = \begin{cases} 0, & y = y' \\ 10, & y = ham\ y' = spam \\ 1, & y = spam\ y' = ham \end{cases} = \begin{cases} 0, & y = y' \\ w(y), & y \neq y' \end{cases}$$

where $w : Y \to \mathbb{R}_+$ is a weight function on the classes. It's an easy observation that $l(y, y') = w(y) l_{0-1}(y, y')$ which we will denote by $l = w l_{0-1}$. The goal is to minimize the true loss

$$L_{D,l}(h) = \mathbb{E}_{(x,y) \sim D}[l(y, h(x))]$$

Lemma 4.10.1. *Let D be a distribution on $X \times Y$ and $w : Y \to \mathbb{R}_{>0}$ a class weight function. Then $\exists$ distribution D_2 on $X \times Y$ and $z > 0$ such that for all loss functions l_1, l_2 with $l_1 = w l_2$ it holds that*

$$z L_{D_2, l_2}(h) = L_{D_1, l_1}(h)$$

for every classifier h.

We make one additional observation:

Remark 4.10.2. We note that in general D_2 is defined by

$$\mathbb{P}_{D_2}[(x,y)] = \frac{1}{z}\mathbb{P}_{D_1}[(x,y)]w(y)$$

where z is simply some normalization parameter. This gives rise to the fact that

$$\mathbb{P}_{D_2}[y|x] = \frac{\mathbb{P}_{D_1}[y|x] \cdot w(y)}{\sum_{y' \in Y} \mathbb{P}_{D_1}[y'|x]w(y')}$$

Given a loss function l and a distribution D on $X \times Y$, let's say that a classifier $h^* : X \to Y$ is a Bayes-classifier for (D, l) if

$$L_{D,l}(h^*) = \inf_g L_{D,l}(g)$$

Lemma 4.10.3. *Assume a distribution D on $X \times \{-1, 1\}$ with 0-1 loss. Then we know that the Bayes classifier is for a given $x \in X$*

$$h^*(x) = \begin{cases} 1, & \mathbb{P}_D[1|x] \geq \frac{1}{2} \\ -1, & \textit{else} \end{cases}$$

For a more general $l = w \cdot l_{0-1}$ we can also exactly compute the Bayes classifier.

Indeed we can apply the previous lemma for $D = D_1, l_1 = l, l_2 = l_{0-1}$ and w to see that $\exists z > 0$ and D_2 such that

$$L_{D,l}(h) = zL_{D_2,l_{0-1}}(h)$$

since we know the Bayes classifier on the RHS, the result on the LHS will be a Bayes classifier for the other distribution. That is we still get

$$\mathbb{P}_{D_2}[1|x] \geq \frac{1}{2}$$

This we know how to handle and we end up with

$$\mathbb{P}_D[1|x] \geq \frac{w(-1)}{w(1) + w(-1)}$$

For the following we need to broaden the meaning of Bayes consistency.

Definition 4.10.4. Let l^* be a (class weighted version of the 0-1) loss function and l be a surrogate loss function $l : Y \times Y \to \mathbb{R}_+$. Then we say that l is Bayes-consistent for l^*

if $\forall h_1, h_2, ... : X \to Y$ and all D on $X \times Y$ with

$$\lim_{i \to \infty} L_{D,l}(h_i) = \inf_g L_{D,l}(g)$$

then it follows that

$$\lim_{i \to \infty} L_{D,l^*}(h_i) = \inf_g L_{D,l^*}(g)$$

5 ensemble learning

5.1 wisdom of the crowd

We want to use this concept with an ensemble of classifiers $h_1, ..., h_T$. We want these such that the probability that classifier h_i is correct is $p > \frac{1}{2}$. Lastly, we need these classifiers to be stochastically independent. In general, this is blatantly wrong and we can never assume that. But let's roll with it for now. Then one can observe that the probability that the majority of the classifiers is wrong goes to 0 as $T \to \infty$.

5.2 inequalties

We know Tchebycheff's inequality

$$\mathbb{P}[|X - \mathbb{E}[X]| \geq \lambda] \leq \frac{V[X]}{\lambda^2}$$

Cantelli's inequality gives a slightly different result than the above.

Lemma 5.2.1. *For a random variable X and $\lambda > 0$ it holds that*

1. $\mathbb{P}[X \geq \mathbb{E}[x] + \lambda] \leq \frac{V[X]}{V[X]+\lambda^2}$
2. $\mathbb{P}[X \leq \mathbb{E}[X] - \lambda] \leq \frac{V[X]}{V[X]+\lambda^2}$

To prove this we'll need Markov's inequality.

Lemma 5.2.2. *Let X be a non-negative random variable and $t > 0$. Then*

$$\mathbb{P}[X \geq t] \leq \frac{\mathbb{E}[X]}{t}$$

5.3 dependent classifiers

We again assume an ensemble of classifiers which we will model as random variables $X_1, ..., X_n \in \{0, 1\}$ where 1 means it classifies correctly and 0 not. We want to consult the majority of the classifiers. We hence calculate

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i := \bar{p}$$

and we say that $\bar{X}$ is correct iff $\bar{X} > \frac{1}{2}$. It is no loss of generality to assume that $\mathbb{P}[X_i] > \frac{1}{2}$ because otherwise it would be quite bad. This then is enough to show that

$$\mathbb{E}[\bar{X}] > \frac{1}{2}$$

Similarly, using Cantelli's inequality we can note that

$$\mathbb{P}[\bar{X} > \frac{1}{2}] \to \frac{(\bar{p}-\frac{1}{2})^2}{c\bar{p}(1-\bar{p}) + (\bar{p}-\frac{1}{2})^2}(n \to \infty)$$

where c is some correlation-constant. This holds true for all classifiers even if they are correlated. If one plots this as a function in $\bar{p}$ with different constants c, one can notice, that a small correlation coefficient is more important than good classifiers.

Definition 5.3.1. The following classifier is called the majority classifier.

$$h(x) = \arg\max_{y\in\{-1,1\}} |\{h_i(x) = y\}|$$

We can also introduce a number of random variables

$$X_i(x,y) = \begin{cases} 1, & h_i(x) = y \\ 0, & otherwise \end{cases}$$

5.4 random forests

This is a special kind of majority classifiers (which is also possible to be used as a regression but we won't). The set of classifiers $h_1, ..., h_n$ is a set of decision trees in this case. For each h_i we want

- a small true risk
- only small correlation to the other decision trees

For the second point there are two tricks.

Definition 5.4.1 (bootstrapping)**.** The training set S is used as a basis to construct different training sets S_i for each h_i. If $|S| = m$, we usually want the S_i to be of size δm

for $\delta \in (0,1]$. Then S_i is formed by drawing δm elements from S with replacement (i.e. a point can be drawn multiple times).

Definition 5.4.2 (random feature selection)**.** Usually, when training a decision tree, we would want to maximize over the gain of homogeneity over all features and all thresholds. Now, want to draw a subset F of the features and we only want to maximize the gain over this subset. If there are d features total, in practice one would usually choose $|F| = \sqrt{d}$.

5.5 Adaboost

This is yet another ensemble classifier. It takes a set of comparatively weak base classifiers and distributes them in a weighted linear combination

$$A(x) = sgn\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

For the base classifiers, one would usually choose something like decision stumps. On their own, these are very bad classifiers.
training: First, we train one of the weak classifiers. Afterwards, we figure out the points that are classified wrong and give those some kind of weight. This has the effect, that training the second classifier will probably take care of these wrong samples. The weight is usually introduced by a weighted loss function for the training of the following weak base classifiers.
But how do we fix the α in the formula of Adaboost? And how do we adjust the weights? Let $p^t_{(x,y)}$ be the weight that is given to the tuple (x, y) when training the t-th classifier. Similarly, let

$$\varepsilon_t = \sum_{(x,y)\in S} p^t_{(x,y)} l_{0-1}(y, h_t(x))$$

be the (weighted) error of the t-th classifier. Then we choose

$$\alpha_t = \frac{1}{2}\log\left(\frac{1}{\varepsilon_t} - 1\right)$$

and

$$p^{t+1}_{(x,y)} = \frac{p^t_{(x,y)} e^{-\alpha_t y h_t(x)}}{\sum_{(x',y')\in S} p^t_{(x,y)} e^{-\alpha_t y' h_t(x')}}$$

Note that the denominator of the last fraction is simply a normalization term, that guarantees that the weights sum up to 1.
Now let's figure out the training error of Adaboost. We want that each of the base classifiers adds something to A. Therefore, we make the following assumption: For every

training set S and for every probability distribution p over S it holds that

$$\min_{h \in W} L_{p,S}(h) \leq \frac{1}{2} - \gamma$$

where W is the set of base classifiers. We tacitly assume that we can give weights to the training set.

Theorem 5.5.1. *If the above holds, the training error of the classifier h^* ouput by Adaboost is*

$$L_S(h^*) \leq e^{-2\gamma^2 T}$$

where T is the number of base classifiers.