AML

Nicolas Muntwyler

Autumn 2020

Contents

1 Math

$$f1 = \frac{1}{\frac{1}{TPR} + \frac{1}{Precision}}$$

$$\text{TPR} = \frac{TP}{TP + FP}$$

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

$$\text{Precision} = \frac{TP}{TN + FP}$$

$$1 + x \leq 1 + e^x$$

$$\begin{bmatrix} a & b \end{bmatrix} \begin{bmatrix} c & d \\ e & f \end{bmatrix} \begin{bmatrix} g \\ h \end{bmatrix} = ach + adi + beh + bfi$$

$$p\left(\begin{bmatrix} a_1 \\ a_2 \end{bmatrix}\right) = N\left(\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \middle| \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_2 \end{bmatrix}\right)$$

$$\implies p(a_1|a_2) = N\left(u_1 + \Sigma_{12}\Sigma_2^{-1}(a_2 - u_2), \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}\right)$$

Product rule $\quad P(A, B, C) = P(A) \cdot P(B|A) \cdot P(C|A, B)$

$P(A|B) = \dfrac{P(A, B)}{P(B)}$ $\quad$ (leave me alone, I forget things)

2 Introduction

Digitisation = Control by Algorithms
Algorithms model posterior probabilities of solutions given data.
Learning requires validation of algorithms
We have to control the algorithms

3 Representations, measurements, data types

Overfitting
To reduce overfitting we should minimize the expected classification error (Expected Risk).
However this is not possible since we can't calculate the expected classification error (Expected Risk), since we would have to know the underlying distribution for that.
Instead we minimize the empirical classification error (Empirical Risk).

Expected Risk, Empirical Risk
At the start you have a problem you want to solve. For example predicting the class label of a sample. Now in order to solve that we humans create a risk (expected risk) function. We hope that if this risk function is low, we then have a model that solves our problem well. You will see that the only human made thing in the risk function is the loss function. So in the end we humans create a loss function. So how good the model can actually solve the Humans create a Risk function (Expected Risk):

$$R(f) = E_X[R(f,X)] = \int_X R(f,X)P(x)dX = \int_Y \int_X Q(Y,f(X))P(Y,X)dXdY$$

Here f is our model. For example a neural network, svm, GP, etc.
And Q(Y,f(X)) is the loss function. (Notice that the Risk is always regarding a loss function. Which loss function you choose is the art)

⇒ We want to minimize this Risk. So we would like to find the model f, that minimizes this risk function. Now calculating what f must be to minimize this risk is not possible. First because its probably not analytically solvable with our methods but more importantly, we don't know the P(X,Y) distribution.
However we can try to estimate (written with ˆ) this Risk. We estimate it with the Empirical Risk:

$$\hat{R}(\hat{f}, D_{train}) = \frac{1}{n}\sum_{i=1}^{n} Q(Y_i, f(X_i))$$

$\hat{f}$ is our model that should approximate the best model.
D_{train} is the dataset (= A sample drawn from the distribution P(X,Y)

⇒ We know the X_i's and the Y_i's since it is our training data (Now they are fixed numbers and not random variables anymore). Now we try to find $\hat{f}$, such that it minimizes this empirical risk.
So instead of optimizing over the expected risk (which we can't) we now optimize over the empirical risk. But why should this be any good? Well the empirical risk tries to estimate the true Risk. (This is like the Covariance matrix and the empirical covariance matrix). So if we have infinitely many data sample we should expect that minimizing the empirical risk is equal to minimizing the expected Risk.
However they are never exactly the same since we don't have infinitely many samples.
Now to train our model ($\hat{f}$ we are using the D_{train} Dataset. In order to see how good we are in actually minimizing the expected Risk, we can't use the empirical Risk of the training Data, since we are minimizing over it in the training. So of course it will be low. Therefore we need another Dataset D_{test} for which we can calculate the empirical Risk. Now if we would again use infinitely many datapoints this empirical risk would be equal to the expected risk. However this will never be done in practise. Because first of we don't have infinitely many samples and secondly you want to use as many samples as possible to use for training. So in the end you have a tradeoff in how many samples you can use for training (more = better) and how many you can use for testing (more = more accurate estimation of the expected Risk).

Taxonomy of Data
Our given Data lives in an object space. Examples:

- Monadic data (Objects without reference to other configurations) z.B Temperature
- Byadic data z.B. users-websites (user1-website1, user1-website2, user2-website1,user2-website2)
- Polyadic data z.B. test persons-behaviour-traits

Scales

- Categorical Scale (or Nominal scale)
- Ordinal scale (the ordering is the important thing). z.B. ranking of different marathon races

- Quantitative scale (Interval scale, ratio scale, absolute scale)

Data whitening = Normalize the data

4 Density estimation

Prior: $P(model)$
Likelihood: $P(data|model)$
Posterior: $P(model|data)$
Evidence: $P(data)$

$$P(model|data) = \frac{P(data|model)P(model)}{P(data)}$$

The posterior mean is the weighted sum of the prior mean and the sample mean

4.1 Estimators

Rao-Cramer bound:
For every unbiased estimator $\hat{\theta}$ of θ_0:

$$E[\hat{\theta}^2 - \theta_0^2] \geq \frac{1}{I_n(\theta_0)}$$

Where $I_n(\theta_0)$ is the Fisher Information.

But which estimator $\hat{\theta}$ is best?
If we have infinite samples one can show:

$$E[\hat{\theta_{ML}}^2 - \theta_0^2] = \frac{1}{I_n(\theta_0)}$$

This is as good as we can get. ($\hat{\theta_{ML}}$ is the maximum likelihood estimator)
But for finite samples we have in some cases (e.g. multivariate gaussian $N(\theta_0, \sigma^2 I)$):

$$E[\hat{\theta_{ML}}^2 - \theta_0^2] \geq E[\hat{\theta stei}^2 - \theta_0^2]$$

Meaning that for some cases the maximum likelihood estimator is not the best choice. (Since the stein estimator is better)
However the maximum likelihood estimator is:

- consistent
- asymptotically efficient
- equivariant
- asymptotically normal

4.2 Some Definitions

Bias

$$bias(\theta) = \theta_0 - E[\hat{\theta}]$$

Consitent

$$\forall_{\epsilon>0} P(|\theta_0 - \hat{\theta}| > \epsilon) \rightarrow_{n\rightarrow\infty} 0$$

Asymptotically efficient

$$\text{For } n \rightarrow \infty \hat{\theta} \text{ minimizes } E[|\theta_0 - \hat{\theta}|]$$

Equivariance

$$\text{If } \hat{\theta} \text{ is MLE of } \theta_0 \text{ then } g(\hat{\theta}) \text{ is MLE of } g(\theta_0)$$

Asymptotically normal

$$\sqrt{n}(\theta_0 - \hat{\theta}) \approx N(0, J^{-1} I J^{-1})$$

5 Regression, bias-variance tradeoff

We define linear Regression as the following problem:

$$argmin_f E[(Y - f(X))^2]$$

The optimal solution is proven to be:

$$f^* = E[Y|X = x]$$

But how can we find what f^* is. The problem is that we don't know $P(Y|X)$ and neither $P(X)$.

Parametric (maximum likelihood) Approach:
We assume that $P(Y|X) = N(f(X), \sigma^2 I)$.
Solve: $argmax_f \sum_i^n \log P(Y = y_i|X = x_i, \sigma^2)$

Statistical learning theory
Minimize directly the empirical risk:
Solve: $argmin_f \sum_i^n (f(x_i) - y_i)^2$

Both of these approaches lead to the same solution.

Remember:

- If you can differentiate (loss function. Remember loss function sometimes includes regularization. Therefore choosing a lasso as regularization results in that the loss function can not be differentiated) and solve the 0=ableitung to the parameter you are very happy and have a closed form solution
- If you can only differentiate but not solve for the parameters you use gradient descent
- If you can't differentiate you use an other optimization technique like LARS.

If f(x) is a linear regression, we have that $Y = \beta X$ (since you want to also have a bias $+\beta_0$ you extended X to (1,X) with n+1 dimension). Therefore our risk is:

$$\sum_{i=1}^{n} (y_i - x_i^T \beta)^2 = (y - X\beta)^T (y - X\beta)$$

One can see that we would have the first case. Meaning we can take the derivative and solve for β. This gives us:

$$\beta = (X^T X)^{-1} X^T y$$

Some points:

- The least squares estimator of β (what we did) is unbiased
- Of all unbiased estimators for β has the least squares estimator the smallest variance. (Gauss Markov Theorem)
- This is all in the sense of MSE. So smallest variance for MSE while being unbiased. But we would like that for the generalization error.

Idea: Trade bias for smaller variance.

Bias-variance Tradeoff

$$E[(y - \hat{f}(x))^2] = \underbrace{(E[\hat{f}] - \hat{f})^2}_{Bias(\hat{f})} + \underbrace{Var[\hat{f}]}_{variance} + \underbrace{Var[\epsilon]}_{Noise}$$

You often make a tradeoff between bias and variance.
Small dataset and large Hypothesis class (complex models) $\Longrightarrow$ Variance large, bias small
Large dataset and small Hypothesis class (basic models) $\Longrightarrow$ Variance small, bias large

Solutions to overfitting

- Regularization
- overfitting
- ensembles

5.1 Regularization

Ridge Regression

Cost function: $RSS(\beta;\lambda) = (\mathbf{y}-\mathbf{X}\beta)^{\mathsf{T}}(\mathbf{y}-\mathbf{X}\beta)+\lambda\beta^{\mathsf{T}}\beta.$

Bayesian view: $Y|(X,\beta)\sim\mathcal{N}\left(x^{\mathsf{T}}\beta,\sigma^2\mathbf{I}\right)$, prior on β: $\beta\sim\mathcal{N}(0,\sigma^2/\lambda\mathbb{I})$.

Solution: $\hat{\beta}^{\text{ridge}} = (\mathbf{X}^{\mathsf{T}}\mathbf{X}+\lambda\mathbf{I})^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{y}$

Tikhonov regularization $R(\beta)=\lambda\beta^{\mathsf{T}}\beta$ is also called weight decay in Neural Networks Literature.

LASSO

Cost function: $RSS(\beta;\lambda) = (\mathbf{y}-\mathbf{X}\beta)^{\mathsf{T}}(\mathbf{y}-\mathbf{X}\beta)+\lambda\|\beta\|_1.$

Bayesian view: $Y|(X,\beta)\sim\mathcal{N}\left(x^{\mathsf{T}}\beta,\sigma^2\mathbf{I}\right)$, prior on β_i: Laplace: $p(\beta_i)=\frac{\lambda}{4\sigma^2}\exp(-|\beta|\frac{\lambda}{2\sigma^2})$.

Solution: By efficient optimization techniques (e.g. LARS). Note: $\|\beta\|_1=\sum_{j=0}^{d}|\beta_j|$ is not differentiable.

For each Matrix you can do an SVD:

$$X = UDV^T$$

Now if you do this for the ridge regression you get that:

$$X\beta_{ridge} = \sum_{j=1}^{d} u_j \frac{d_j^2}{d_j^2+\lambda} h_j^T y$$

Since d_j are the singularvalues (squared eigenvalues) you can derive that:
Ridge regression suppresses contribution by small eigenvalues.

LASSO
= (Least absolute shrinkage and selection operator)

$$\hat{\beta_{Lasso}} = argmin_\beta \sum_{i=1}^{n}(y_i-\beta_0-\sum_{i=1}^{n}x_{i,j}\beta_j)^2$$

$$\text{subject to}\sum_{i=1}^{n}|\beta_i|\leq s$$

Lasso is known to be sparse estimators, because the LSE surface often hits the corner of the constraint surface.

Generalized Ridge Regression

$$\hat{\beta} = argmin_\beta \sum_{i=1}^{n}(y_i-\beta_0-\sum_{j=1}^{d}x_{i,j}\beta_j)^2+\lambda\sum_{i=1}^{d}|\beta_j|^q$$

5.2 Nonlinear Regression

Idea: Instead of

$$y_i = \sum_{j=1}^{d}\beta_j x_{i,j}$$

you can do:

$$y_i = \sum_{j=m}^{M}\beta_m h_m(x_i)$$

Where $h_m(x_i) : R^d \to R$
Lets say we use polynomial of degree 2 and d=3 (z.b. x_i = (2, 0.3, -3)
Then $h_1(x_{i,1},x_{i,2},x_{i,3})$=$x_{i,1}$ and $h_2(x_{i,1},x_{i,2},x_{i,3})$=$x_{i,2}$ etc.
Also $h_4(x_{i,1},x_{i,2},x_{i,3})$=$x_{i,1}^2$ etc.
So we have: $h(x_{i,1},x_{i,2},x_{i,3}) = [x_{i,1},x_{i,2},...,x_{i,2}^2,...]$

6 Gaussian Processes

To understand gaussian processes we first need to understand Bayesian linear regression:

6.1 Bayesian linear regression

Idea: Have a prior on the weights. And then a posterior of the weights.

6.1.1 Prior: $N(0, I)$

Prior: $p(w) = N(0, I)$
Likelihood: $p(y|w, y, \sigma_n) = N(y; w^T x, \sigma_n^2)$
Posterior: $p(w|X, y) = N(w; \mu, \Sigma)$ $\mu = (X^T X + \sigma_n^2 I)^{-1} X^T y$ $(= \sigma_n^{-2} \Sigma X^T y?)$
$\Sigma = (\sigma_n^{-2} X^T X + I)^{-1}$
Inference: $p(f^*|X, y, x^*) = N(\mu^T x^*, x^{*T} \Sigma x^*)$
Inference: $p(y^*|X, y, x^*) = N(\mu^T x^*, x^{*T} \Sigma x^* + \sigma_n^2)$

Note: σ_n^2 comes from the noise which we assume is distributed as: $N(0, \sigma_n^2)$ Note: $f^* = x^T x^*$ (so without the noise)

6.1.2 Prior: $N(0, \sigma_w I)$

Prior: $p(w) = N(0, \sigma_w I)$
(Likelihood: $p(y|w, y\sigma_n) = N(y; w^T x, \sigma_n^2)$?
Posterior: $p(w|X, y) = N(w; \mu, \Sigma)$ $\mu = \sigma_n^{-2} \Sigma X^T y$
$\Sigma = (\sigma_n^{-2} X^T X + \sigma_w^{-2} I)^{-1}$
Inference: $p(y^*|X, y, x^*) = N(\mu^T x^*, x^{*T} \Sigma x^* + \sigma_n^2)$

Note: σ_n^2 comes from the noise which we assume is distributed as: $N(0, \sigma_n^2)$

6.1.3 Explanation

The advantage is that you have a distribution over your model parameters. So you know how sure you are about your model. Because lets say you want to predict cancer A or B. Now your model says to 0.95 its A and to 0.05 its B. The model is very confident. But are you sure that your model is really correct? Bayesian view does that. So instead of 1 model that says 0.95 and 0.05 you now have infinite models (a distribution over your model parameters). Now you could sample from that. Lets say you sample 1000 times (so you would have 1000 models) and each model predicts A with high certainty. Then you are aurally convinced that your model is correct and take its answer as valid. Notice that in order to be sure that it is A you want that your sampled models predict the same AND that the models themselves are confident in their answer.

6.2 Gaussian Processes

In BLR we fit linear functions. Now we want to extend this idea to nonlinear functions.
So we could introduce nonlinearity with a function $h(x)$ as before. However this is computationally heavy. To solve this issue we can use the kernel Trick.

Without the kernel trick:
When we calculate the posterior for Gaussian noise and a gaussian prior $w \approx N(0, I)$ (Remember $f = Xw$ and $y = f + \text{Noise}$) we have: $f \approx N(0, XX^T)$
Now if we extended X to have more dimensions (for example with h(x) = $[x^1, x^2, ..., x^3]$) we have a huge X and since we need to multiply X with X^T this will be extremely computationally expensive.

With kernel trick:
We realize that XX^T is like a kernel. So we get: $f \approx N(0, K)$, hence data points only enter as inner products (Key advantage. Which we use in the kernel trick. See questions).

So we have that $XX^T = K = \begin{pmatrix} k(x_1, x_1) & k(x_1, x_2) & \dots & k(x_1, x_m) \\ \vdots & & & \vdots \\ k(x_m, x_1) & k(x_m, x_2) & \dots & k(x_m, x_m) \end{pmatrix}$

6.2.1 Prediction with GP

$$p\left(\begin{bmatrix} y \\ y_{n+1} \end{bmatrix} | x_{n+1}, X, \sigma\right) = N\left(\begin{bmatrix} y \\ y_{n+1} \end{bmatrix} \middle| 0, \begin{bmatrix} C_n & k \\ k^T & c \end{bmatrix}\right)$$

$C_n = K + \sigma^2 I$
$c = k(x_{n+1}, x_{n+1}) + \sigma^2$
$k = k(x_{n+1}, X)$
$K = k(X, X)$

We can now use the conditional Gaussian distribution trick:

$$p\left(\begin{bmatrix} a_1 \\ a_2 \end{bmatrix}\right) = N\left(\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \middle| \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_2 \end{bmatrix}\right)$$

$$\implies p(a_2|a_1 = z) = N\left(a_2|u_2 + \Sigma_{21}\Sigma_{11}^{-1}(z - u_1), \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}\right)$$

With that we get:

$$p(y_{n+1}|k^T C_n^{-1} y, c - k^T C_n^{-1} k)$$

6.3 Questions

- "kernel function encoder our assumptions about the function which we wish to learn" So for the 1d case. If we have a gaussian kernel we indirectly assume that the true regression curve is smooth and polynomial like, while when we use the exponential kernel we assume that the true regression function is more wavelet like, much more spikier.

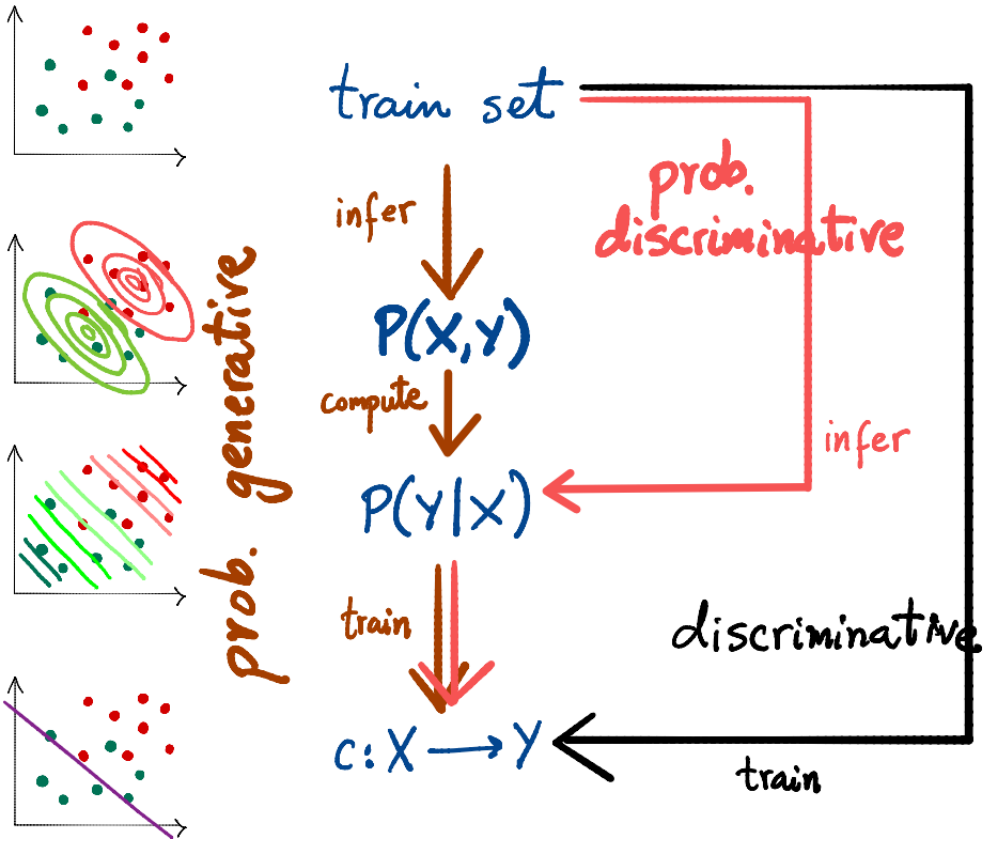
- How does the kernel trick help to reduce the computational complexity?
  $\implies$ Guess: Before we did not use kernels. With using kernels it is low. A kernel reduces the complexity. Lets say we have k(x,x') and $h = (x, x^2, x^3, .., x^{100})$ then we would have to first calculate h(x) and then h(x') and then multiply them together. However the kernel saves us (we don't have to do that) but instead we already know what the result of h(x)*h(x') is. For example: $(x^T x' + 1)^{100}$. You can see that if this 100 becomes a 1000000000 we still just have to calculate $(x^T x' + 1)^{1000000000}$ but without the kernel we have to calculate $h = (x, x^2, x^3, .., x^{1000000000})$ and the same for h(x') and then still multiply them together.

7 Linear discriminant functions

There exist three types of models:

- Probabilistic Generative Models
- Probabilistic Discriminant Models
- Discriminant Models

We explain the differences by an example:
Problem: We have two features: (cough duration, temperature) and want to guess if this person has pneumonia (0,1)
Solution: Train a linear classifier



Probabilistic Generative Model: tries to infer $p(X, Y)$
Probabilistic discriminant Model: tries to directly infer $p(Y|X)$
Discriminant Model: tries to directly predict Y

If you don't assume anything you are in the case of: discriminant models (e.g. SVM).

7.1 Probabilistic Generative Model

Since we try to model the underlying distribution $p(X, Y)$ we have:
Advantages:

- better understanding
- Can generate new samples
- Outlier Detection
- Degree of belief

Disadvantages:

- Hard to do
- Strong bias (since we need to assume many things)

7.1.1 Infer $p(X, Y)$

Notice we only infer this. Most often we don't calculate $p(X, Y)$ (We can though? And in some cases you do so you can generate new samples or detect outliers etc.) Its just that we model $p(y)$ and $p(x|y)$ which is enough information to calculate $p(x, y)$, but we will only use it to model $p(y|x)$ in the next subsection.

- Guess a family of parametric probabilistic models
- Infer parameters (for example with MLE)

So often you choose a distribution of $p(Y)$ and a distribution $p(X|Y)$. Because given these two we can construct $p(X, Y)$ with bayes: $p(x, y) = p(x|y)p(y)$

Example
We model:

$$p(Y) \sim Bernoulli(\beta)$$
$$p(X|Y = 1) \sim N(\mu_1, \Sigma_1)$$
$$p(X|Y = 0) \sim N(\mu_0, \Sigma_0)$$

As our next step we need to infer the parameters of this family of probability distributions. One way would be MLE (Maximum likelihood estimation)
If you do the calculations you get for this examples:
$\beta = \frac{n_1}{n}$, $\mu_1 = \frac{1}{n_1}\sum_{i=1}^{n} x_i$, and $_1 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_1)(x_i - \mu_1)^T$

7.1.2 Calculate $p(Y|X)$

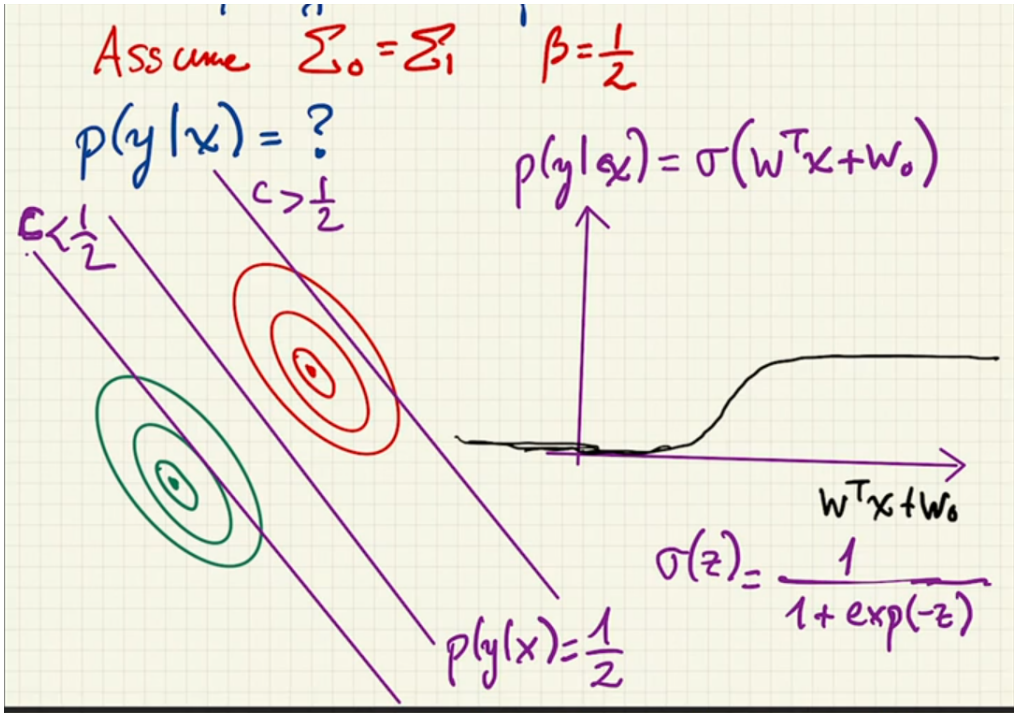
Meaning we need to calculate the posterior.
We do that with the information from before:

$$p(y = 1|x) = \frac{p(x, y = 1)}{p(x)} = \frac{p(x|y = 1)p(y = 1)}{p(x)} = \frac{p(x|y = 1)p(y = 1)}{p(x|y = 1)p(y = 1) + p(x|y = 1)p(y = 1)}$$

If we calculate this we get:

$$p(y|x) = \underbrace{\sigma(w^T x + w_0)}_{gen.lin.model}$$

Why is there sigmoid in the endformula? Here is a graphical intuition: Back in our example we assumed that we have two Gaussian's fro $p(X|Y)$. Now remember that we have a linear classifier (violet line). Now observe how the probability of $p(y|x)$ changes if we move the violet line. In order to get a probability distribution (so we have values between 0 and 1) we can apply the sigmoid function.



Note: When calculation $p(y|x)$ and uses the assumptions. So for example we have $p(x|y=1) = |2\pi\Sigma_1|^{-\frac{1}{2}} exp(-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1))$. Also observe that the result has a sigmoid. This comes from some reformulations where we use log and exp to cancel each other out.

Note: The process of assuming that the covariance matrices are the same, and we have a mixture of Gaussian's (two) and then fitting a linear classifier with a probabilistic generative model is called :
linear discriminant analysis

Note: For this example we assumed that the covariance matrices are the same.
If we don't assume that we would get:

$$p(y|x) = \underbrace{\sigma(w^T W x + x^T w + w_0)}_{gen.quadr.model}$$

In this case we call this process **quadratic discriminant analysis**

Note: We assumed a bernoulli distribution for Y and two Gaussian's for the $X|Y$. For this case we could calculate the posterior analytically. But this does not hold for every assumptions. For most of them you can't analytically derive the posterior.

7.1.3 Train this classifier

Here we uses Bayes decision theory

- Define a loss function
- Use the function from before to do: $min_c E_{X,Y}[L(Y, c(X))]$

Notice that:

$$min_c E_{X,Y}[L(Y, c(X))] = \sum_{x,y} p(x,y)L(y, c(x)) = \sum_x p(x) \sum_y p(y|x)L(y, c(x))$$

So our classifier is:

$$c^*(x) = argmin_a \sum_y p(y|x)L(y, a)$$

In order to find this minimum, you can use various different techniques. If its analytically tractable you can just use analysis. Otherwise you may have to use gradient descent etc. (But in this case i believe its analytically tractable)

Note: Since we have a cost(loss) function we are now not guaranteed anymore to have a linear classifier.
Note2: Loss function needs to be bounded by below

7.2 Probabilistic discriminant Model

We directly try to model $p(Y|X)$
So could assume:

$$p(y|x) = \sigma(w^T x + w_0)$$

So the number of parameters is $O(d+1)$. In the generative approach we modeled the covariance matrix etc. where we had $O(d^2)$ parameters. This can be an advantage for the probabilistic discriminant model.

As the second step we estimate the parameters:

$$argmax_w \text{log-likelihood of our training set}$$

$$argmax_w \log \prod_{i=1}^{n} p(x_i, y_i|w)$$

$$argmax_w \log \prod_{i=1}^{n} p(y_i|x_i, w) \underbrace{p(x_i|w)}_{p(x_i)}$$

In order to find this maximum we need to use gradient descent (It is not analytically tractable). We can do gradient descent because it is differentiable.

7.2.1 Whats a good learning rate, Newtons method

An optimal learning rate is:

$$\eta = argmin_\eta NL(w^{(k+1)}$$

Sadly (as we know before) $NL^{(k+1)}$ is analytically intractable. But there exist multiple ideas to solve the argmin from above. For example use a taylor approximation of NL at w^{k+1}. Doing a second order taylor expansion we derive that the optimal learning rate at point $w^{(k)}$ is:

$$\eta(k) = \frac{\|\nabla NL(w^{(k)})\|}{\nabla NL^T(w^{(k)} H_{NL}(w^{(k)} \nabla NL(w^{(k)}}$$

One could also argue instead of finding the best learning rate at that point, we could try and directly find the next best estimate/point/update. If we follow that idea we get **Newton's method**. And the result is:

$$w^{(k+1)} = w^{(k)} - H_{NL}^{-1}(w^{(k)} \nabla(w^{(k)})$$

The problem with the newtons method is that you need to calculate the inverse Hessian. The hessian depends on the number of parameters. In a Deep neural network where you have millions of parameters this is computationally undoable. So in practise you just do gradient descent. Also in practise you don't even use the best (calculated) learning rate, since practise has shown that this most often does not translate to the fastest convergence. Therefore we use other methods to find a good learning rate.

7.3 Discriminant model

We forget about distribution, and we directly try to find a classifier that fits best our training set. To do a purely discriminant approach a common technique is **statistical learning theory**.

- Start with a loss function $L(y, x)$
- This gives us then a expected risk (generalization error) $E_{X,Y}[L(Y, c(X))]$
- Since we don't know the $p(x, y)$ distribution we can't directly minimize the expected risk. Therefore we try to approximate it with the empirical risk. $\frac{1}{n}\sum_{i=1}^{n} L(y_i, c(x_i))$
- Use an algorithm that minimizes the empirical risk

Some examples of discriminant models:

- Perceptron
- Fishers linear discriminant
- SVM's

7.3.1 Perceptron

A perceptron has the following form:

$$c(x) = sign(w^T x + w_0)$$

We first need to find a loss(cost)-function:

$$L(y, c(x)) = \begin{cases} 0 & \text{if } c(x) = y \\ |w^T x| & \text{if } c(x) \neq y \end{cases}$$

If the labels are +1,-1 then you can simplify this to:

$$L(y, c(x)) = \begin{cases} 0 & \text{if } yw^T x > y \\ -yw^T x & \text{if } yw^T x < y \end{cases}$$

So in the end we have:

$$L(w) = \sum_{i=1}^{n} L(y_i, c(x_i))$$

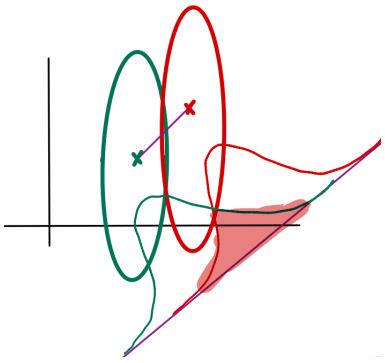
Since this is differentiable we can now use gradient descent to minimize this Loss.
We only have to update when we classify wrong. This gives rise to the **variable increment perceptron**. You can then argue that this algorithm is an instance of the robbins monro algorithm, for which you can prove convergence (under some conditions about the learning rate and that the training set is linearly separable).
Notice that the solution of the is not unique (multiple classifiers exist, that converged).

7.3.2 Fishers linear discriminant

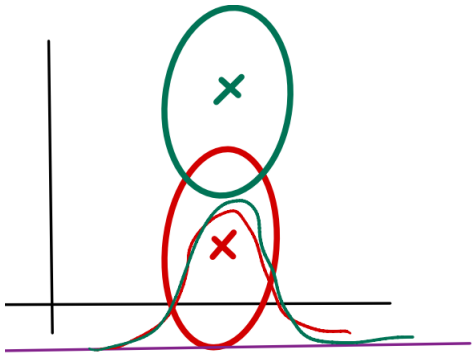
The easiest way of understanding FLD is to look at it graphically. You can see it as trying to fit a line such that when we project the traininset (or only the means) on that line it should for example maximize the variance or maximize the distance between the means. After you have this projection, you can try to fit a mixture of gaussian
Fisher combines two ideas:

- Maximize the distance of the means (projected onto the line)
- Minimize the variance of trainingset (projected onto the line)

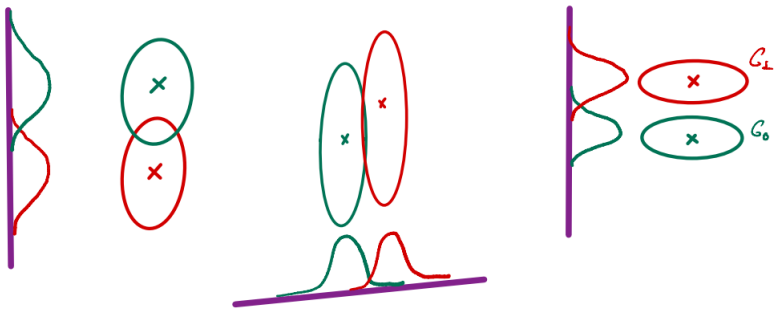
Doing only one of them is not good.
Only maximizing the distance of the means could have the following result:



And only minimizing the variance could have the following result:



But if we combine the two we get desired results:



Mathematically we do:

$$max_w(w^t(\bar{x_0} - \bar{x_1}))^2$$

$$min_w \quad \underbrace{Var(w^T C_0)}_{\frac{1}{n}\sum_{i \leq n}(w^T x_i - w^T \bar{x_0})^2} + Var(w^T C_1)$$

But for convenience we minimize the variance but don't divide by the number of samples, so we get:

$$max_w(w^t(\bar{x_0} - \bar{x_1}))^2$$

$$min_w \sum_{i \leq n} (w^T x_i - w^T \bar{x_0})^2 + \sum_{i \leq n} (w^T x_i - w^T \bar{x_1})^2$$

So how do we combine the two? We could maximize the fraction:

$$max \frac{(w^t(\bar{x_0} - \bar{x_1}))^2}{\sum_{i \leq n}(w^T x_i - w^T \bar{x_0})^2 + \sum_{i \leq n}(w^T x_i - w^T \bar{x_1})^2}$$

You can then do math to prove that this simplifies to:

$$w \propto (S_W)^{-1}(\bar{x_0} - \bar{x_1})$$

C_0:= Training samples we classified by 0

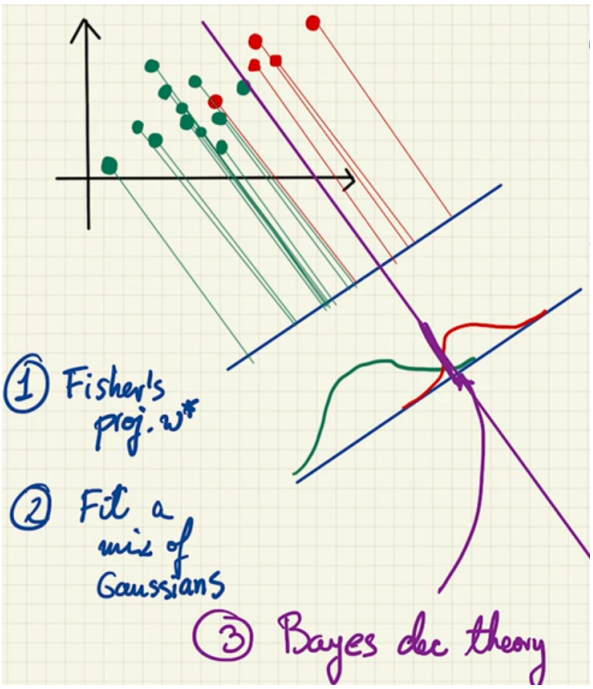
$$S_w = \hat{Cov}(C_0) + \hat{Cov}(C_1) = \sum_{k \in \{1,2\}} \sum_{i \in C_k})(x_i - \bar{x_k}(x_i - \bar{x_k})^T$$

$$x_k = \frac{1}{|C_k|} \sum_{i \in C_k} x_i$$

Note: This can also work with multiple classes, but the math is more complicated.

We have now found the projection line. But how can we now to classification?
An idea is to fit a **mixture of Gaussian's**. (Important: Then it becomes a probabilistic generative model, since we assume something about the probability distribution of the projection to be a mixture of gaussian).

So we have now found out how we would set our gaussians (mean, variance). But we still dont know how we would then classify. Therefore lastly we can use **Bayes decision theory** to set a threshold at which point we decide 1 or 0:



Note: Fishers linear discriminant is only the projection part. So you can also look at FLD as a dimensionality reduction technique.

7.3.3 SVM

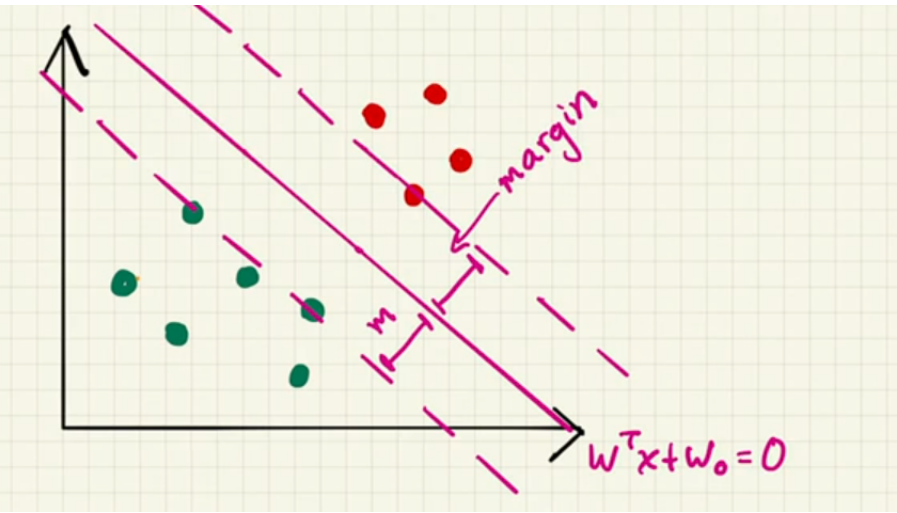
We cover SVM's extensively in the next part.

8 SVM

As in the section before we will again derive expression (e.g. empirical risk with hinge loss) we would like to minimize.
Unfortunately we will see that it is analytically intractable. Also gradient descent can't really help since we have constraints that need to hold (see formula of SVM below). However the convex optimization theory shows that we can formulate a dual (using Lagrange multipliers) to the objective we want to minimize, which is then analytically solvable. We can also show that under some conditions the solution of the dual is the same as for the original objective. The solution of the dual is calculated with quadratic programming.

8.1 SVM as perceptron

We saw before that a perceptron classifier can have multiple solutions. An SVM comes from the idea that we use the peceptron that has the maximum margin.



Maximizing the "margin" is the same as minimizing $\frac{1}{2}\|w\|^2$
So an SVM solves the following optimization problem:

$$min_{w,w_0} \frac{1}{2}\|w\|^2$$

$$\text{s.t. } 1 - y_i(w^T x_i + w_0) \leq 0$$

But how can you solve such a problem?
In Mathematics this problem is a constraint optimizations problem. And in the following we will show how you can solve such a problem. However not all problems can be solved that way since sometimes it is analytically not tractable (to complex to find a solution to the 4 equations). However in the section "Constraint optimization for SVM" we will then show an alternative to solve such a problem.

8.2 Constraint optimization

If you have a problem of the type:

$$min_w f(w)$$

$$\text{s.t. } g_i(w) = 0, h_j(w) \leq 0$$

You have to do the following steps:

1. Check regularity conditions
Most often you use Slaters condition:

$$\text{Is there a } w \text{ s.t. } g_i(w) = 0 \text{ and } h_j(w) < 0$$

2. Solve a system of inequalities
You need to find a w for which holds:

- $\frac{\partial L}{\partial w} = 0$ (L = Lagrangian)

- $g_i(w) = 0$ and $h_j(w) \leq 0$
- $j \geq 0$ (comes from the Lagrangian)
- $a_j h_j(w) = 0$ (complementary slackness)

Lagrangian:

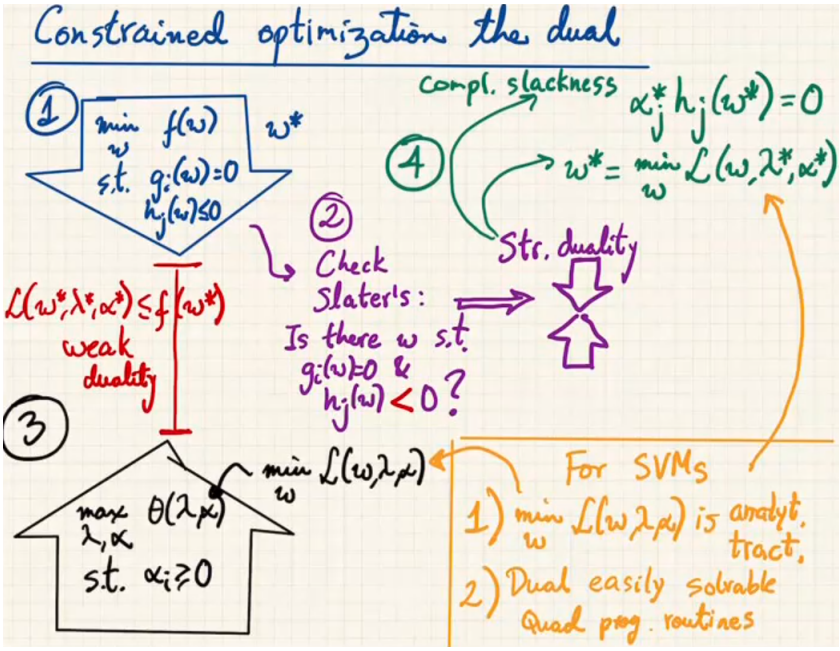
$$L(w, \lambda, \alpha) = f(w) + \sum_i \lambda_i g_i(w) + \sum_j \alpha_j h_j(w)$$

$$a_j \geq 0$$

Note: Why exactly we need to solve with respect to these 4 equations can be intuitively explained with a graphical experiment, where you try to find the point (human) on a plane which minimizes the distance to a quadrocopter. Sadly the human is restricted to stay on the plane (no flying or digging). Additionally the plane is not infinite, but is disk shaped with radius r.
While mathematically deriving this point, you will implicitly solve these 4 equations.

8.3 Constraint optimization for SVM

If we apply the procedure from above on SVM's it won't work. Because solving the 4 equations from above won't be analytically tractable.
However there exists an alternative (use dual) from convex optimization theory.
The idea is to create the dual problem of the original problem. If some conditions (e.g. Slaters condition) we have strong duality, which means that the solution for the dual is the same as for the original problem. To solve the dual we can most often use quadratic programming. However quadratic programming would not be able to solve a maximization of a minimization (as it is now), hence we first need to analytically solve the minimization of the Lagrangian. Luckily this is analytically tractable.
The following will now give an overview how we solve it step by step (In the next section we have the individual steps in detail):



0. Setup
Notice that for SVM we only have only the inequality constraints, meaning our objective looks like this:

$$min_w f(w)$$

$$\text{s.t. } h_j(w) \leq 0$$

1. Check Regularity conditions
We check if Slaters condition hold

2. Create the dual
A dual has the form:

$$max_{\alpha,\lambda}\theta(,\alpha)$$
$$\text{s.t. } \alpha \geq 0$$

And in our SVM case we have that $\theta(\lambda,\alpha) = min_w L(w,\lambda,\alpha)$:

$$max_\alpha min_w L(w,\alpha)$$
$$\text{s.t. } \alpha \geq 0$$

L=Lagrangian.

Notice that we could not solve this with quadratic programming (maximization of minimization), hence we need to first solve the minimization of the Lagrangian analytically. With that we can then rewrite the dual, which is now only a maximization.

3. Solve the dual
With the rewriting of the dual we can use quadratic programming to solve for the α's.

8.3.1 Strong Duality

We can always create the dual. But it generally only holds that the solution of the duality bounds the solution of the real problem, This is called weak duality.
But if Slaters condition hold then we have **strong duality** meaning that the solution of the dual is exactly the same as the solution for the original problem.
Notice that if we have strong duality it gives rise to **complementary slackness**: $\alpha_j^* h_j(w^*) = 0$

8.4 SVM solution for linear separable training sets

For this section we assume that the training set is linearly separable.

0. Setup
We have the following objective (Here the linearly separability is assumed):

$$max_{w,w_0} 2m(w,w_0)$$
$$\text{s.t. } y_i(w^T x_i + w_0) \geq b > 0$$

$m(w,w_0)$:= margin

We can simplify this to:

$$min_{w,w_0} \frac{1}{2}\|w\|^2$$
$$\text{s.t. } y_i(w^T x_i + w_0) \geq 1$$

Note: This is already convex. So why even use the dual? It is for the case where we include transformations $\phi(x_i)$. **1. Check conditions**
The Slaters condition holds (need to prove this, but it is very doable. You start with assuming linear separability)

2. We compute the dual
Remember that we want to solve the dual formulation with quadratic programming. However as it stands now we can no do that. Quadratic programming can't handle a maximization of a minimization. Therefore we need to first analytically solve the inner minimization (of the Lagrangian). Luckily it is analytically tractable. The Lagrangian is:

$$L(w,w_0,\alpha) = \frac{1}{2}\|w\|^2 + \sum_i \alpha_i(1 - y_i(w^T x_i + w_0))$$
$$\text{s.t. } \alpha_i \geq 0$$

This is the Lagrangian we need to minimize. Taking the derivative with respect to w and w_0 gets us two things:

$$\frac{\partial L}{\partial w} = 0 \quad \Longrightarrow w^* = \sum_i \alpha_i y_i x_i$$

$$\frac{\partial L}{\partial w_0} = 0 \quad \Longrightarrow \sum_i \alpha_i y_i = 0$$

We can now use this to rewrite our dual as:

$$max_\alpha \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^T x_j + \sum_i \alpha_i - \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

$$\text{s.t. } \alpha_i \geq 0 \text{ and } \sum_i \alpha_i y_i = 0$$

And lastly we further simplify this to:

$$max_\alpha \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

$$\text{s.t. } \alpha_i \geq 0 \quad \sum_i \alpha_i y_i = 0$$

3. Solve the dual to get w^*
This we can now solve with quadratic programming which gives us the α^*'s. We can use the alphas to finally calculate w^*:

$$w^* = \sum_i \alpha_i y_i x_i$$

Notice that we still need w_0^*.

4. Find w_0^*
1. Find x^+ and x^-

$$x^+ = min_{i \leq n, y_i = +1} w^{*T} x_i \qquad x^- = max_{i \leq n, y_i = -1} w^{*T} x_i$$

2. Do linear algebra
$w^{*T} x^+ + w_0^* = 1$ and $w^{*T} x^- + w_0^* = -1$
From these follows:

$$w_0^* = -\frac{1}{2}(w^{*T} x^+ + w^{*T} x^-)$$

8.4.1 Complementary slackness

Strong duality implies complementary slackness:

$$\alpha_i^* h_i(w^*) = 0$$

Based on that one can derive with some math that: α^* is sparse.
We also have:

$$w^* = \sum_i \alpha_i^* y_i x_i$$

Putting these two fact together gets us that w^* is just a linear combination of a few support vectors

8.5 SVM with neglected examples

We now don't assume linear separability anymore

Setup

$$\min_{w, w_0} \frac{1}{2} \|w\|^2 + C \sum_{i \leq n} \epsilon_i$$

$$s.t. y_i(w^T x_i + w_0) \geq 1 - \epsilon_i$$

$$\epsilon_i \geq 0$$

Remarks:

- ϵ_i lowers the bar for each neglected example

- C: tradeoff hyperparamter.
  Small C: Wide margin, many neglected samples
  Big C: narrow margin, few neglected examples

The dual

$$\max_{\alpha} \sum_{\alpha_i} -\frac{1}{2} \sum_{i,j} \alpha_i, \alpha_j y_i, y_j x_i^T x_j$$
$$s.t. 0 \leq \alpha_i \leq C \quad \sum_i \alpha_i y_i = 0$$

Solve dual to get w^* and ϵ_i
Solving the dual gets us the 's.

$$w^* = \sum_i \alpha_i y_i x_i$$
$$\epsilon_i = \max(0, 1 - y_i(w^{*T} x_i + w_0^*))$$

8.6 Transformations and kernels

Like in IML we again repeat the topic of transformations and kernels.
Idea: A non linearly separable training set might become linearly separable in a transformed space.
Note: Lets assume we have a polynomial transformation and we can then linearly separate in this transformed space. But lets go back to the original space. We are now fitting a more sophisticated classifier, namely we are trying to fit a polynomial.
Note2: All a kernel needs is to be symmetric and its kernel matrix needs to be positive semi definite. So you can define a kernel e.g. on sets $A_1, ..A_n$ with $K(A_i, A_j) = 2^{|A_i \cap A_j|}$. Since this kernel matrix is psd and symmetric, this is an accepted kernel. We just showed that we can now use SVM on discrete sample spaces like sets.

8.6.1 Kernels for SVM

The dual of the SVM is:

$$\max_{\alpha} \sum_{\alpha_i} -\frac{1}{2} \sum_{i,j} \alpha_i, \alpha_j y_i, y_j x_i^T x_j$$
$$s.t. 0 \leq \alpha_i \leq C \quad \sum_i \alpha_i y_i = 0$$

Now we can do the transformation:

$$\max_{\alpha} \sum_{\alpha_i} -\frac{1}{2} \sum_{i,j} \alpha_i, \alpha_j y_i, y_j \phi(x_i)^T \phi(x_j)$$
$$s.t. 0 \leq \alpha_i \leq C \quad \sum_i \alpha_i y_i = 0$$

The prediction is then:

$$w^{*T} \phi(x_{new}) = (\sum_i \alpha_i^* y_i \phi(x_i))^T \phi(x_{new})$$

And now observe that we can again use the kernel trick. (We don't need $\phi(x_i)$ nor $\phi_(x)$ but just the result of the multiplication $\phi(x_i)\phi(x)$).

8.6.2 RBF Kernel

$$K(x, y) = exp(-\gamma \|x - y\|)$$

This kernel has the following $\phi_(x)^T \phi(\hat{x})$ (Same result as above but other math):

$$\sum_{j=0}^{\infty} \sum_{n_1 + ... + n_d = j} \left(exp(-\frac{1}{2}\|x\|^2) \frac{x_1^{n_1} \cdots x_d^{n_d}}{\sqrt{n_1! \cdots n_d!}} \right) \left(exp(-\frac{1}{2}\|\hat{x}\|^2) \frac{\hat{x}_1^{n_1} \cdots \hat{x}_d^{n_d}}{\sqrt{n_1! \cdots n_d!}} \right)$$

The γ is a hyperparameter. A big gamma (for example 10) gets you many support vectors (so you are maybe overfitting) hence you decision boundary is not very smooth. A small gamma (for example 0.1) gives you very few support vectors and a very smooth decision boundary.
(https://jgreitemann.github.io/svm-demo)

8.6.3 Constructing kernels

There are 3 ways:

- Via composition of kernels
- Defining $\phi(x)$ and analytically (by hand) derive what $\phi(x)^T\phi(y)$ is.
- Via Mercer's theorem. (symmetric and positive semi definite kernel matrix)

8.7 Structural SVM's

A structural SVM: When the input (x_i) is not a vector of numbers (same for output). For example the input is: "The cat chases the dog" and the output should be a syntax tree.
Idea: We are doing classification so we have a number of output classes (every syntax tree would be an output class). We would then compute a **compatibility score** between the input x and the output class. We do this for each class. Then we choose the class that hat the highest score:

$$c(x) =_k (f_k(x))$$

Problem1: We have too many classes (We would need to train a weight vector for each class (like one versus rest multiclass)
Problem2: not interclass learning ("The dog eats", "The dog eats meat", different class but share some information)
Solution: Joint feature maps

8.7.1 Joint feature maps

$$\phi : X \times Y \longrightarrow R^m$$

$$\phi(\text{"The dog chases the cat"}, S - NP - VP - D - N - V - VP - D - N)$$

=(some numerical representation of the sentence (z.b. bag of word), some numerical representation of the syntax tree (z.b. bag of stamps))

So the result could be:

$$((0,0,0,1,1,0,0,1),(0,0,0,0,1,0,2,1))$$

The advantage is that we only need to train one weight vector for all classes:

$$w^T\phi(x,y) = \text{compat. score between x and y}$$

And then we have: $c(x) = argmax_y w^T\phi(x,y)$ The challenge is that we need a good joint feature map ϕ.

8.7.2 SVM formulation

But lets say we found a good joint feature map, what would be the objective (svm formulation). We still need to define that. An example could be:

$$min_w \frac{1}{2}\|w\|^2$$

$$\text{s.t. } w^T\phi(x_i,y_i) \geq 1 + w^T\phi(x_i,y') \quad \text{for any } y' \neq y_i \text{ and } i \leq n$$

With that formulation we fail to capture the gravity of a mistake. For example classifying S-T-NP as S-T-NV is not as grave as classifying S-T-NP as S-T-N-S-VP-VP-N. Therefore we introduce a similarity measure $\triangle(y_i, y')$ between two syntax trees. We also call it our lossfunction.
Secondly the trainingset might not be linearly separable hence we need to introduce slack variables (as already seen). With that we get the following objective:

$$min_{w,\xi} \quad \frac{1}{2}\|w\|^2 + \frac{C}{n}\sum_{i\leq n}\xi_i$$

$$\text{s.t. } w^T\phi(x_i,y_i) \geq \triangle(y_i,y') + w^T\phi(x_i,y') - \xi_i \quad \text{for any } y' \neq y_i \text{ and } i \leq n$$

$$\xi_i \geq 0$$

8.7.3 Relation to statistical learning theory

Theorem: If w^*, ξ^* are optimal, then the empirical risk of w^* w.r.t $\triangle$ is $\leq \frac{1}{n}\sum_i \xi_i^*$

In statistical learning we don't assume anything about the underlying distribution but try to approximate the generalization error by directly minimizing the empirical risk.
We started with SVM by just trying to maximize the margin. But how do we know that this will then generalize well? This theorem tells us that we can bound the empirical error by above with $\sum_i \xi_i^*$. (And within some reason the empirical risk approximates the expected risk).

8.7.4 Training algorithm

We have a training set:

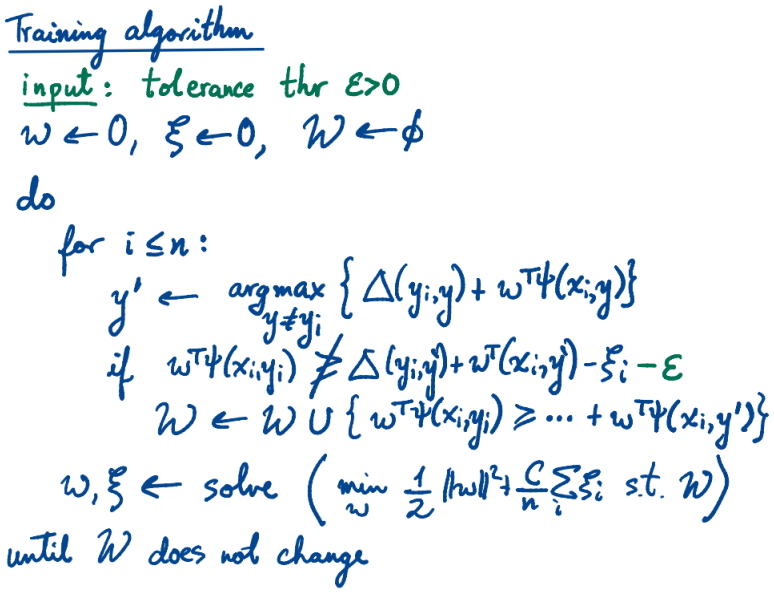
$$Z = \{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$$

So we also have a set of mislabelings:

$$\{(x_1, y_2), (x_1, y_3), (x_2, y_1), (x_2, y_3), (x_3, y_1), (x_3, y_2)\}$$

We need that the compatibility score for the mislabelings is lower than for correct examples. So we are back to a normal classification task. However in practise you have a huge number of such constraints (because there exists many syntax trees and hence a big number of mislabelings). So the training algorithm is a little more intelligent.

Idea: Take some constraints and solve for those using quadratic programming. Then check all other constraints to know which don't hold and which do. We then add the most violated constraint to our list of constraints we want to solve for. We then again do quadratic programming on this new (bigger) set of constraints. We repeat this process until the weight vector doesn't change anymore. Additionally we add a slack variable ϵ so we have an easier time, otherwise even with quadratic programming it would be too expensive.



8.7.5 Prediction

Remember that the prediction is done as:

$$c(x) = argmax_y w^T \phi(x, y)$$

It seems that one needs to go through all possible classes y, hence one needs to try all possible syntax trees. However there exist some techniques from NLP that lets you do dynmaicy programming to find the best y without having to brootforce calculate all classes.

8.7.6 Other example: Page ranking

example (set of docs, subset of docs)

Training set is built defining for each query a set of topics and then using algos that compute a subset of docs covering these topics.

$$\phi(x, y) = \sum_{\text{(doc } \in \text{ y)}} \sum_{\text{(word in doc)}} \text{rel. frequency of word in x}$$

$$\triangle(y, y') = \text{normalized set difference between y and y'}$$

The prediction is again not done by brootforce trying all different y but use a clever method called "submodular optimization".

8.8 Some Points

- Maximizing the margin of an SVM is inducing bias but lowers variance
- You want #Datapoints >> #Dimensions (Otherwise the separating hyperplane is very unstable. high variance)

9 Ensemble Methods

Idea: Reduce variance term in the generalization error by aggregating diverse models (often weak classifiers by themselves)

9.1 Bagging

Idea: Multiple estimators used together will be more accurate. (Guessing the weight of the Ox example. Wisdom of Crowds)
For this to work you need diverse estimators which are also independent of each other.

Algorithm of Bagging:

- Create M Bootstrap sets $Z_1, ..., Z_M$
- Train M base models $b^{(1)}, ..., b^{(M)}$
- aggregate $\hbar^{(M)}$

The bootstrap sets are subsets of the original data set. However datasamples can appear multiple times in order that the boostrap set has the same amount of samples as the original dataset. For Regression the aggregation is just the mean. For classification it is the majority.

Instead of using bootstrap sets we could also choose to train on a subset of features as our different sets. This will then be a **Random Forest**

9.1.1 Random Forest

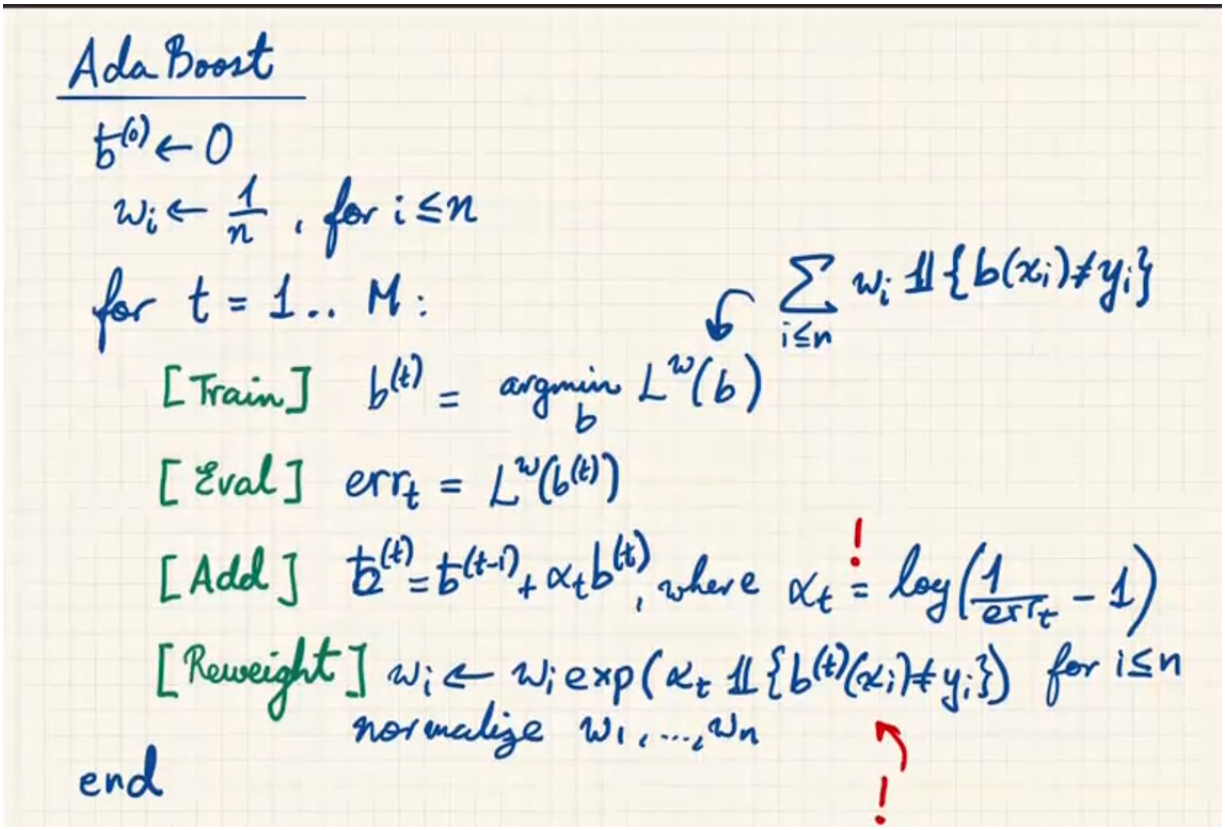
+ Easy and fast
+ Built-in validation
+ Useful for imbalanced dataset (classweights=balanced or bootstrapped samples of equal amount of sampled of class 1 and 0)
– Not good for high-dimensional data
– Less interpretable than decision trees
– Bad for complex relationships between features

9.2 Ada Boosting

You do a cycle of 4 steps:

- Train (Train your favorite model on the whole training set, that minimizes the weighted 0/1 Loss)
- Evaluate (Compute 0/1 Loss err_t on training set)
- Aggregation (Add the prediction to the others with weight $\frac{1}{err\,t} - 1$)
- Reweight (Samples falsely classified get weight $\alpha_t w_i$, otherwise w_i) + Normalize weights

0/1-Loss = Sum of all Missclassifications



The **Prediction** is: $\text{sign}(\sum_i \alpha_i b^{(i)}(x))$

Keypoints:

- Ada Boost is the same as doing statistical learning with the exponential loss function
- Ada Boost doesn't overfit. You can train many many base models and do the aggregation.
- Ada Boost trains a "maximum-margin" (= Difference of: $\sum_{i:b^{(t)}(x)=-1} a_i$ and $\sum_{i:b^{(t)}(x)=+1} a_i$) classifier

Theorem1
With probability $\geq 1 - \delta$:
Gen. error $\leq$ "fraction of examples with margin $\leq \mu$" + $O(\ldots \frac{1}{n} \ldots \frac{1}{\mu^2} \ldots \log(|H|))$

Theorem 2
For AdaBoost "fraction of examples with margin $\leq \mu$ decreases exponentially with M

Theorem 3
Random forests and AdaBoost are self-averaging algorithms that train spiky interpolating classifiers
Self-averaging := An algorithm is self-averaging if the trained model is an average of diverse models

Benefits of spiky interpolating models:

- Mostly smooth decision boundary (expect the spikes)
- Allows localized and sharp modifications to prevent influence from noise
- Allows fitting complex signals

Benefits of averaging spiky interpolation models:

- localize noise effect
- even spikier models

Final remarks:

- You want rather complex base models. (Diversity and high dimensionality aspect)
- In a classifier you want a low and high dimensionality aspect. (You need to keep both in check to not underfit/overfit)
- Booting tries to make the best of the bias-variance tradeoff

9.3 Points

- Bagging: Use different bootstrap samples (to train you classifier) "parallel"
- Boosting: Trained on same data but reweighted "sequential"
- Bagging: Average multiple weak learners
- Boosting: Make a weak learner a strong learner
- Bagging: Always same weight of prediction of each base model
- Booting: Weight the prediction of a base model based on its train accuracy.
- Bagging: Varies dataset for each base model.
- Boosting: Always the same dataset for each base model
- Use AdaBoost to localize outliers. (samples with high weight)
- In AdaBoost you do reweighting with samples. In gradient boosting you use gradients.
- AdaBoost is the same as forward stagewise additive modeling with an exponential loss function. (You can see a connection also to gradient boosting, where your next additive base model minimizes the negative gradient. Here is FSAM we minimize the loss function regarding $\alpha_t b^t(x)$)
- AdaBoost is exactly the same as FSAM with exponential loss function
- Gradient boosting approximates FSAM (This also means that gradient boosting with an exponential loss function approximates FSAM, hence Gradient boosting with an exponential loss function approximates Ada boosting)
- FSAM is the general concept of minimizing a lossfunction by adding another base classifier which tries to minimizes the residuals.
- Gradient boosting is a generalization of AdaBoost (can choose different loss functions)
- Gradient boosting is a specific implementation of FSAM (approximates it)
- Gradient boosting with an exponential loss function approximates FSAM with an exponential loss function, hence it approximates AdaBoost.
- Gradient boosting and FSAM have different approaches to doing the same thing. (Same result?)

10 Deep Learning

Theorem 1
If $f : [0, 1] \to R$ is continuous, $\epsilon > 0$
then there is a NN s.t.:
1.) $NN(x) = \sum_{i \leq m} \alpha_i \sigma(w_i^T x + b_i)$
2.) $max|f(x) - NN(x)| < \epsilon$

This NN has two layers. One with the sigmoid activation function and one with the identity as the activation function.

10.1 Training neural networks

The goal is to minimize the empirical risk:

$$min_\theta \sum_{i \leq n} L(y_i, NN_\theta(x_i)) = \min_{theta} L(\theta)$$

With $\theta = (w_1^{(1)}; b_1^{(1)}; w_1^{(2)}; b_1^{(2)}; w_1^{(3)}; b_1^{(3)}; ...)$ (However we will leave the $b_i^{(j)}$'s out for short notation.)
$w_i^{(j)}$:= The weights for the i-th neuron in the (j+1)-th layer.
Meaning that $w^{(j)}$ is between the j-th and (j+1)-th layer

To train we then do Gradient Descent.

Algorithm 1: Gradient Descent

```
k = 0;
w_r^(l) = unif([-sqrt(1/n), sqrt(1/n)]) (n = units of previous layer);
b^(l) = 0;
repeat
    w_r^(l) = w_r^(l) - η(k) ∇_{w_r^(l)} L(θ);
    k = k + 1;
until ?;
```

? = Early stopping, ...

How to compute $\nabla_{w_r^{(l)}}$
We need to take the derivative of the neural net with respect to $w_r^{(l)}$. (For all layers)
Doing so naively will take up $O(l^2)$ time. However if you do dynamic programming you can find that you can store the results you have already calculated and use them in your next calculation. This is due to the nature of the Chain rule. Explicitly for layer l you can use the result of layer l+1
This leads to the approach to do first Backpropagation. With this we have $O(l)$ time.
Notice that you only have to take the derivative (=Calculating the Jacobian) of the activation function and the layer. The rest is just multiplying the jacobian matrices (chain rule). This is very easy for most activation function and a simple linear layer.
Since the Joacobian matrices are sometimes of simple form you can use clever linear algebra to speed up the computation.

Note that you need to evaluate the Jacobian matrices (Set the value). For example you need the output of the previous layer to set the values of the jacobian. To get the output from the previous layer you again could calculate it separately for every layer, however this would again take $O(l^2)$. Alternatively you again save the previous results. This is called the Forward Pass with $O(l)$ time complexity.
Lastly note that the backward pass is dependent on the forward pass. This means you first do the forward pass and then the backward pass.

Algorithm 2: Gradient Descent (Detailed

k = 0;
$w_r^{(l)}$ = unif($[-\sqrt{\frac{1}{n}}, \sqrt{\frac{1}{n}}]$) (n = units of previous layer);
$b^{(l)} = 0$;
repeat
  Do forward pass and backward pass to compute:;
  $\left.\frac{\partial NN}{\partial w_r^{(l)}}\right|_{\theta, x_i}$ for each l,r and i;
  $\frac{L}{\partial w_r^{(l)}} = \sum_{i \leq n} \frac{\partial L(y_i, NN_\theta(x_i))}{\partial w_r^{(l)}}$;
  $w_r^{(l)} = w_r^{(l)} - \eta(k) \frac{\partial L}{\partial w_r^{(l)}}$;
  k = k + 1;
until *?*;

? = Early Stopping,...

Observe that this algorithm requires to take all training samples for every iteration. However you can do Stochastic Gradient descent to approximate it but with significant speed up.

Remember our goal:

$$min_\theta \sum_{i \leq n} L(y_i, NN_\theta(x_i))$$

Since this is the same as taking the derivative and setting it to 0, one can rewrite this term (with assumption that we have infinitely many training samples) as:

$$E_{X,Y}[\nabla_\theta L(Y, NN_\theta(X))] = 0$$

On this you can apply the Robins-Monro algorithm:

Algorithm 3: Robins-Monro algorithm

Goal: Compute θ s.t. $E_Z[f(z;\theta)] = 0$;
Input: Learning rate function η and Samples $z_1, ..., z_n$;
repeat
  $\theta^{(k)} = \theta^{(k-1)} - (k) f(z_k;^{(k)}$
until *?*;

.

This algorithm is guaranteed to converge if:

- Regularity condition
- $\eta(k) \geq 0$
- $\sum_k \eta(k) = \infty$
- $\sum_k \eta^2(k) \leq \infty$

You can see that stochastic gradient descent is a version of the Robins-monro algorithm.

Keynotes on (Batch GD,Mini-batch GD = Stochastic GD):

- Batch GD = Gradient descent that uses the whole training set
- Stochastic GD = Mini-batch GD, uses a subsample of training samples in an iteration
- Batch GD is more precise
- Stochastic GD can handle larger training sets
- Stochastic GD you have faster improvements
- Stochastic GD is better at escaping local minimums
- Stochastic GD has a lower generalization error

- You might find a better minimum with the Batch GD but it wont generalize well). Because the best minimum might be a NN that grossly overfits. Hence you almost want to have noise etc (Therefore you introduce sometimes dropout layers) to not find this best minimum that would overfit. But rather find a good local minimum that doesn't overfit. This can be seen as regularizing the network. Introducing Stochasticity can help overfitting.

10.2 Variational autoencoders

This is an unsupervised learning task

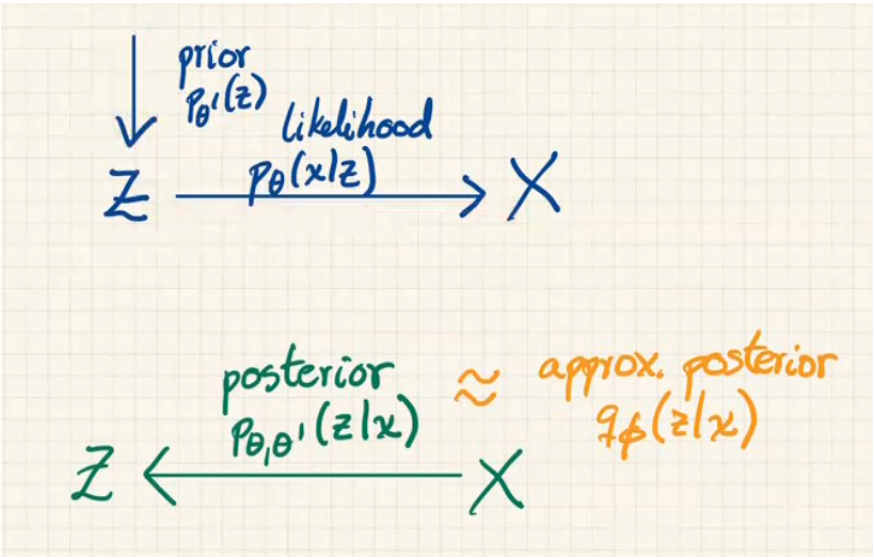
Goal: Learn a meaningful representation Z of data X without supervision
A good representation should be: informative, disentangled and robust.

informative:= Given a representation it should be easy to guess the measurement. (Given the features you as a human after some training should be able to say if those features represent a cat or a dog)
disentangled: Every component in the representation is associated with a distinguished feature (in the input/measurement).
entangled features:= If you change a feature than you change multiple attributes of the input. For example the smile and the eyebrows. You would like a disentangled feature. Hence when you change the entry of this features you would only change one attribute of the input. For example the smile.
Robust: Noisy perturbations in the measurement(input) should not substantially affect the representation and vice versa.

Infomax principle
Goal: Z = enc_θ maximizes $I(X;Z)$
$\Rightarrow argmax_\theta I(X;Z) = argmax_\theta E_{X,Z}[\log(\frac{p(X,Z}{p(x)p(z)})] \approx argmax_\theta \sum_{i\leq n} E_{Z|X_i}[\log p(X_i|Z)]$

Maximizing with the Infomax principle will give us informative features but not disentangles and robust features.
To fight this we propose:

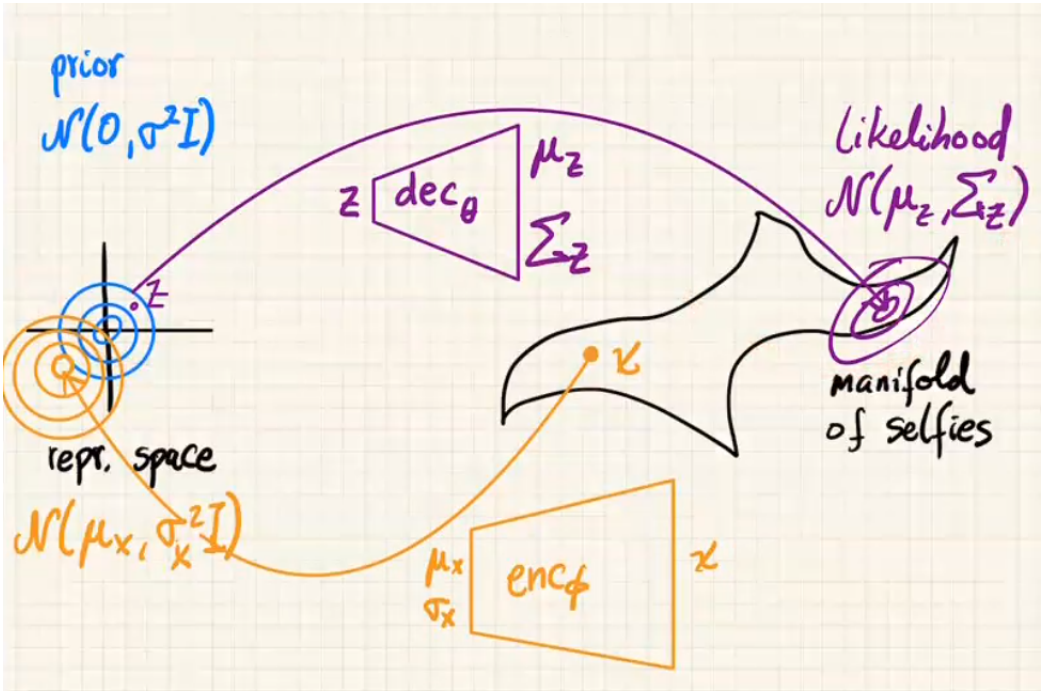
Variational autoencoder
Tries to fit a probabilistic generative model to the training set where the representation are latent variables.



1.) Draw a representation Z from a prior distribution $p_{\theta'}(Z)$

2.) From this representation Z draw a measurement: $p_\theta(X|Z)$

Task: Find the family of the prior and likelihood function. (Afterwards you just do maximum likelihood estimation to find the parameters θ).



Ideally you would want to find θ and θ' and then analytically calculate the posterior. However the likelihood function will most likely be very complicated and one can not calculate the posterior analytically. Therefor one does variational inference (therefore the name). The idea is to approximated the posterior with $q_\phi(Z|X)$, hence we will also need to optimize ϕ.
You want to optimize over θ, θ', ϕ, however what for example is θ. It is a set of parameters that describe the prior distribution. Hence before we start we would take assumptions about this prior distribution. For example one often assumes that it is a gaussian $N(0, \sigma^2 I)$. In this case we have that $\theta = \sigma$, so we only need to optimize one variable.
The same goes for the likelihood function. Again you often choose a gaussian $N(\mu_Z, \Sigma_Z)$ The parameter of this gaussian is determined using Deep learning (the decoder)

So notice that only now we would just have a NN (the decoder), that maps representation space to a likelihoods. We can give it a sample from the prior distribution (representation space) and it will apply the decoder to get a likelihood. By maximum likelihood you can then make your final prediction, which is a newly generated sample in the measurement space (so a never seen before face could be generated that way). The very important part is that in the representation space we have a continuous subspace for same objects in the measurements space. Meaning that if the task is to generate dog,cat or giraffe we can be sure that if [0.5,1.3] is a dog in the representation space, that [0.5001,1.3001] will also be a dog. (However this will be before seen dog. A newly generated dog).

So far we only defined the decoder and now how we can train it. And here comes the encoder. The aim is to minimize the reconstruction error. Hence we have a sample A that we send though the encoder then through the decoder. After this we should see a very similar image as A was. Now the problem lies in this encoder part. What is it?. Mathematically it should be the posterior of the likelihood and prior distribution. So in theory one could say that the encoder should be a mathematical function. But since the likelihood function is a complicated likelihood function we analytically cant calculated the posterior. So we have a dead end. We need to have the posterior as the encoder, otherwise the whole thing doesn't work. Smart people came then up with the idea of Variational Inference. We simply try to approximated the posterior. This means that we try to find a distribution q_ϕ that approximated the posterior.

How to train the VAE:

$$argmax_{\theta,\theta',\phi} \sum_{i \leq n} \log p_{\theta,\theta'}(x_i)$$

Which is the same as:

$$argmax \underbrace{E\left[\log\left(\frac{p_{\theta,\theta'}(X_i, Z)}{q_\phi(Z|X_i)}\right)\right]}_{elbo_{\theta,\theta',\phi}(x_i)} + KL(q_\phi(.|x_i)|p_{\theta,\theta'}(.|s_i))$$

10.3 Open topics of deep learning

- Architecture choice (How many layers, activation functions, ...)
- Search strategy (Stochastic gradient descent, why drop-out, ...)
- We have no theory for deep learning

Here are some points from Prof. Buhmann:

- Till now we always thought of NN as functions. But one can argue that mathematically we should not see a NN as a function. For example people that atteir to bayesian inference believe it should output not a value but a probability distribution. Hence a NN should be a relation rather than a function.
- You purposely don't get rid of the stochasticity.

10.4 Problems with optimizations

It is not clear what we should optimize. We have our loss function we try to optimize. However we purposely add regularization (penalize large weights), add stochasticity (dropout) and even stop (early stopping) before we would reach our minimum of the loss function.
We pretend we optimize our loss function, but we actually don't want to do that exactly.
Machine learning is not optimization.

10.5 Model selection

We often choose a model by minimizing the validation cost.
Training, relative and Test data: X', X", X"'
Possible set of posteriors:

$$\{p^{(1)}(\theta|X), ..., p^{(k}(\theta|X)\}$$

with costs:

$$\{-\log p^{(1)}(\theta|X), ..., -\log p^{(k)}(\theta|X)\}$$

Then we choose our model as:

$$P^*(.|.) = argmin_{1\leq\alpha\leq k} E_{\theta|X'}[-\log p^{(\alpha)}(\theta|X'')]$$

10.5.1 Sensitivity Analysis

Lets assume that: X' = $\overline{X}$ $+\delta$ and X" = $\overline{X}$ $-\delta$
We can then prove that the our of sample error ($E_{\theta|X}[-\log p(\theta|X'')]$) is linear in δ (1)

Prof. Buhmann is trying to find a new decision criteria, based on the posterior. ($E_{\theta|X}[p(\theta|X'')]$) (2)

We can prove that for $E_{\theta|X}[p(\theta|X'')]$ the linear term in δ vanishes and we are only quadratically dependent on δ.
However the problem lies in that minimizing cost is often convex while optimizing this posterior (maximizing posterior agreement) might not be.

In essence this is the main point:
We would like to minimize the expected Risk (True Risk). However we can't do that, hence we minimize the expected Risk (Empirical risk). Now prof. Buhmann argues that even if you could minimize the expected Risk, this might not be the optimal thing to do. In other words prof. Buhmann is a Bayesian.
The bayesian approach has only a quadratic dependency on δ.

Other points:
You should not trust a single solution (non bayesian) but a solution sampled from an as good as possible approximated posterior (bayesian). (Sensitivity analysis argument).
Other words: If you have a coin with probability p=?. Now what helps more to guess what p is: The sequence which is the most likely (non bayesian) or a typical sequence (bayesian)

Other point:
Right now models are not using these posteriors but minimize the cost (hence non-bayesian). However it works. (So a more or less typical solution will be presented). Why? Because we introduced the randomness. This

stochasticity (dropout etc.) helps to not get the true minimum. Also the early stopping and the regularization help you to not get the true minimum. Since the true minimum is not what you want. The most likely sequence (=True minimum) of a coin (HHHHHH) is not useful, but a typical (THTTH) would be. Hence in order to not get this true minimum we do little tricks. But maybe the overall ground concept is wrong. Why even try to minimize something you don't even want the minimum of. Hence we should refine the objective we want and choose a different approach. Prof. Buhmann suggest this bayesian approach.

10.5.2 Summary

- Minimizing our of sample error is linear in noise (δ)
- Maximizing the expected log posterior agreement is quadratic in noise. Hence more robust, but could yield a non convex optimization problem
- To choose a model you should not take the model with the lowest validation error, but the one which maximizes the log posterior agreement.

10.6 My points

- In SGD: The concept of taking a step into the negative gradient (how far is determined by the learning rate) is the robbins-monro algorithm

11 Mixture Models

We are in the unsupervised case. The task is to do clustering.
Popular Algorithms:

- k-Means
- EM- Algorithm

11.1 K-Means

k-Means Algorithm

Require: INPUT: $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$
init $\boldsymbol{\mu}_c = \mathbf{x}_c$ for $1 \leq c \leq k$

1: **repeat**
2: Keep prototypes $\mathcal{Y}$ fixed and assign sample vectors $\mathbf{x}$ to nearest prototype

$$c(\mathbf{x}) \in \underset{c \in \{1,\ldots,k\}}{\operatorname{argmin}} \|\mathbf{x} - \boldsymbol{\mu}_c\|^2$$

3: Keep assignments $c(\mathbf{x})$ fixed and estimate prototypes

$$\boldsymbol{\mu}_\alpha = \frac{1}{n_\alpha} \sum_{\mathbf{x}: c(\mathbf{x}) = \alpha} \mathbf{x} \quad \text{with} \quad n_\alpha = \#\{\mathbf{x} : c(\mathbf{x}) = \alpha\}$$

4: **until** changes of $c(\mathbf{x}), \mathcal{Y}$ vanish
5: **return** $c(\mathbf{x}), \forall \mathbf{x} \in \mathcal{X}$ and the prototypes $\mathcal{Y}$

Problem: Is minimizing the distance to the mean really the correct measure we want to minimize?

11.2 EM-Algorithm

The details are already described in the IML course. Here the most important things.
Idea: Data is generated from a mixture of components.

$$p(x|\pi_1, ..., \pi_k, \theta_1, ..., \theta_k) = \sum_{c \leq k} \pi_c p(x|\theta_c)$$

The π_c is the prior probability that sample is generated by the mixture component c with parameters θ_c
You need to choose $p(x|\theta_c)$. This can be a gaussian, gamma distribution etc. For example a Gaussian would be:

$$p(x|\mu, \Sigma) = \frac{1}{\sqrt{2\pi}^d} \frac{1}{\sqrt{\Sigma}} exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right)$$

We then need to find the parameters. An often used method is the maximum likelihood.
Find $\hat{\theta}$ such that it maximizes the likelihood:

$$p(X|\pi_1, ..., \pi_k, \theta_1, ..., \theta_k) = \prod_{x \in X} \sum_{c \leq k} \pi_c p(x|\theta_c)$$

This is computationally difficult so we can arrive at the same result by maximizing the log likelihood:

$$p(X|\pi_1, ..., \pi_k, \theta_1, ..., \theta_k) = \sum_{x \in X} \log(\sum_{c \leq k} \pi_c p(x|\theta_c))$$

However this is still intractable. Therefore we use the EM-algorithm.

Summary: EM for Mixture of Gaussians

1: **repeat**

2: **E-Step**: Calculate assignment probabilities

$$\gamma_{\mathbf{x}c} = \frac{P\left(\mathbf{x}\left|c, \boldsymbol{\theta}^{(j)}\right.\right) P\left(c\left|\boldsymbol{\theta}^{(j)}\right.\right)}{P\left(\mathbf{x}\left|\boldsymbol{\theta}^{(j)}\right.\right)}$$

3: **M-Step**: Update the mixture model parameters

$$\begin{aligned}
\boldsymbol{\mu}_c^{(j+1)} &= \frac{\sum_{\mathbf{x}\in\mathcal{X}} \gamma_{\mathbf{x}c}\mathbf{x}}{\sum_{\mathbf{x}\in\mathcal{X}} \gamma_{\mathbf{x}c}} \\
(\sigma_c^2)^{(j+1)} &= \frac{\sum_{\mathbf{x}\in\mathcal{X}} \gamma_{\mathbf{x}c}\left(\mathbf{x}-\boldsymbol{\mu}_c^{(j+1)}\right)^2}{\sum_{\mathbf{x}\in\mathcal{X}} \gamma_{\mathbf{x}c}} \\
\pi_c^{(j+1)} &= \frac{1}{|\mathcal{X}|}\sum_{\mathbf{x}\in\mathcal{X}} \gamma_{\mathbf{x}c}
\end{aligned}$$

We still have problems though:

- Which model for a mixture mode should use
- How many mixture components do we need
- (Still use distance to mean a measure of goodness)

To solve these problems you can do non-parametric Bayesian mixture models

12 Nonparametric Bayesian Methods

Dirichlet Distribution is the multivariate generalization of the beta distribution.
Latent clusters: For a finite number of drawings N , we do not have to realize all K clusters. The unrealized clusters are then called latent clusters.
So the best selection of number of clusters should be K with a probability distribution (bayesian approach). Because there might exist clusters that don't observe a sample, since the probability for a sample to land in that cluster is very low). The idea is to allow infinitely many clusters but then let the observed data decide how many we should actually need.
When we choose K=∞ we have nonparametric Bayes.
"nonparametric" means we have infinitely many parameters.

Nonparametric bayes is a density estimation procedure with arbitrarily many mixture components.

12.1 Gibbs sampling with dirichlet process as prior

Aim: Have number of clusters not be fixed, but found out by the algorithm
In order to have this variable amount of cluster you need a prior that does not enforce a number of clusters. Remember in GMM you had a prior which was: P[x=1]=0.3, P[x=2]=0.5, P[x=3]=0.2. Here x is an arbitrary point. But you had to already assume that we have 3 clusters.
Therefore you choose a different prior. Namely you take a "Dirichlet process" as your prior. Dirichlet process is the whole Chinese Restaurant scenario. But instead of going forward (creating the table setting, aka deciding where sits who, more precisely at which table sit how many people) we go backwards. Meaning that we already have a table setting and now the last guest would come in. With a probability of $\frac{\alpha}{\alpha+n-1}$ he would sit at a new table. For the other tables the probability that he would sit at table i is: $\frac{n_i}{\alpha+n-1}$ where n_i are the amount of people that already sit there. So our prior is the following: P[x=1]= $\frac{n_1}{\alpha+n-1}$, P[x=2]= $\frac{n_2}{\alpha+n-1}$ ... P[x=k] = $\frac{n_k}{\alpha+n-1}$ but with the additional probability of $\frac{\alpha}{\alpha+n-1}$, that this person sits at a new table (opens up a new cluster). (P[x=2] := probability that person x sits at table 2, aka probability that point x is in cluster 2). Initially you need to choose an arbitrary prior. For example you could choose that every point is its own cluster. So we have n clusters.
Having this as a prior we now use Gibbs sampling as our "algorithm" that uses a prior and likelihood to infer a posterior. We already assumed what our prior is (a dirichlet process) but we still need to assume what our likelihood is. Often you choose the likelihood to be a gaussian with the centroid (middle point of the cluster) being the expected value (so just a random initialization) and the variance is a hyperparameter. For example just set it to 1. Now we have our prior and likelihood, and we can begin with our Gipps sampling:
Gibbs sampling iterates over all points and this multiple times until the number of clusters and its centroids are more or less stable. So the algorithm is:
Take an arbitrary point. You know the current initialization of the points (which point corresponds to which cluster. at the start every point is its own cluster). You create your prior from that with the dirichlet process. So P[x=1]= $\frac{n_1}{\alpha+n-1}$, P[x=2]= $\frac{n_2}{\alpha+n-1}$ ... P[x=k] = $\frac{n_k}{\alpha+n-1}$ and lastly a probability of $\frac{\alpha}{\alpha+n-1}$ for the k+1'th cluster. You take this as your prior. Then you take your likelihood function value (Likelyhood[X=x|z=1] is f(X=x|z=1). Since you modeled this to be a gaussian you take the function value for that gaussian. Remember that you have as many Gaussian's as clusters. Therefore you say of which cluster you want it with the "conditioned on z=1". So for that gaussian (expected value is the centroid of this cluster and variance was chosen by you to be e.g. 1) and take its function value. If d=1 you just take $f(x) = \frac{1}{2\pi\sqrt{\sigma}} e^{\frac{(x-\mu)^2}{-2\sigma^2}}$). Now you have your prior and likelihood for that point for each cluster. You then multiply these together. You then have k+1 numbers. Then you normalized these numbers (by dividing with their sum). Voila this is your posterior. You now have the probabilities that this point goes to cluster 1,2,...,k and most importantly k+1. This is the beautiful thing of dirichlet process, that you can theoretically add a cluster. As your last step you now sample from that posterior to get you cluster, which this point should get. So for example you get cluster 2 and you assign this point to that cluster. Notice how you could get k+1 from your sample, which means that you opened a new cluster. But it could also be that you chose cluster 2 and before you were in cluster 3 (and you were the only point that was in this cluster) so you annihilated cluster 3, meaning you have reduces the number of clusters. But having done that you now have a new assignment of the points to the clusters. Hence this is you new state (of the restaurant) for your next points. You repeat this process until the number of clusters and the assignments to the clusters have stabilized.

DP Mixture Model:
Probabilities of clusters (mixture weights): $\rho = (\rho_1, \rho_2, ...) \sim GEM(\alpha)$
Centers of clusters: $\mu_k \sim N(\mu_0, \sigma_0)$ k=1,2,..

Assignments of data point to clusters: $z_i \sim \text{Categorical}(\rho)$, i=1,..,N
Coordinates of data points: $x_i \sim N(\mu_{z_i}, \sigma)$, i=1,..,N

Posterior:

$$p(z_i = k | z_{-i}, x, \alpha, \mu) = \text{Prior} \times \text{Likelihood} = \begin{cases} \frac{N_{k,-i}}{\alpha+N-1} p(x_i | x_{-i,k}, \mu) \\ \frac{\alpha}{\alpha+N-1} p(x_i | \mu) \end{cases}$$

Algorithm 4: Collapsed Gibbs sampler for DP-Mixtures

Start with that every datapoints is its own cluster
for *i=1 to N in random order* **do**
  Remove x_i's sufficient statistics from old cluster z_i
  for *k=1 to K* **do**
    Compute $p_k(x_i) = p_k(x_i | x{-}i, k)$; // likelihood for 1..k
    Set $N_{k,-i} = |x_{-i,k}|$; // How many people at which table (without last guest)
    Compute $p(z_i = k | z_{-i,x}) = \frac{N_{k,-i}}{\alpha+N-1} p_k(x_i | x_{-i,k})$; // Posterior (not normalized)
  end
  Compute $p_*(x_i) = p(x_i | \mu)$ or $(p(x_i | x_{-i})$; // likelihood for k+1
  Compute $p(z_i = * | z_{-i}, x) = \frac{\alpha}{\alpha+N-1} p_*(x_i)$; // Posterior (not normalized)
  Normalize $p(z_i | \cdot)$; // Normalize Posterior (over 1..k+1)
  Sample $z_i \sim p(z_i | \cdot)$
  Add x_i's sufficient statistics to new cluster z_i
  If any cluster is empty, remove it and decrease K
end

12.2 Some points

- Expected number of tables in the Chinese restaurant is: $O(\alpha \log N)$
- GEM distribution := Stick Breaking distribution with infinitely many breaks.

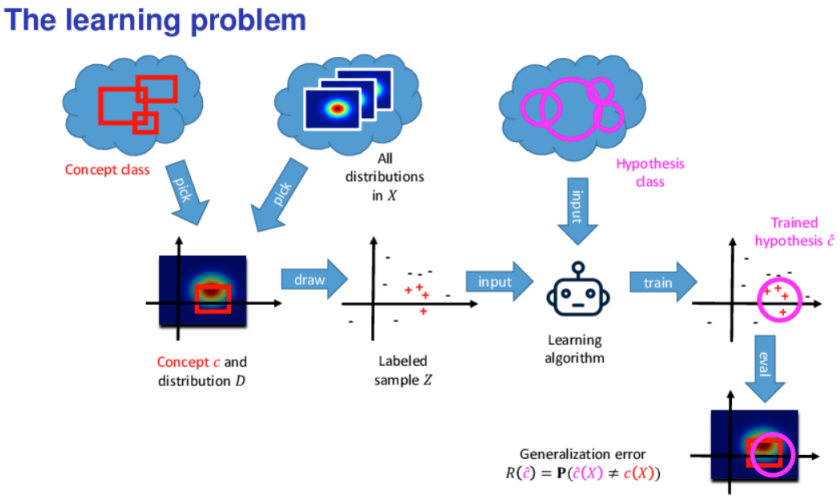
13 PAC learning

PAC = Probably Approximately Correct Learning

13.1 Concepts

PAC learning comes from Statistical learning and tries to answer:

- What is learnable?
- If something is learnable how much can we learn it by empirically minimizing a cost function?



Generalization error (not computable by the learner):

$$R(\hat{c}) := P(\hat{c}(X) \neq c(X))$$

Empirical error (computable by the learner):

$$\hat{R}_n(\hat{c}) := \frac{1}{n} \sum_{i \leq n} 1_{\hat{c}(x_i) \neq c(x_i)}$$

The important thing is that one can show:

$$E_{X, X_1, \ldots, X_n} \left[\hat{R}_n(\hat{c}(X)) \right] = R(\hat{c})$$

13.2 Notation

Instance space X: Think of X as being a set of instances or objects in the learner's world.
Concept: A concept is a subset c of X (we sometimes think of c as a function c: $c : X \longrightarrow \{0, 1\}$).
Concept class: A set of concepts we wish to learn
Hypothesis class: Another set of concepts that we use to learn a target concept from the concept class.

Note: We don't assume anything about the distribution of X.

13.3 The PAC Learning Model

Definition 1
Let H and c be a hypothesis class and a concept. A **learning algorithm** is an algorithm that receives as input a labeled sample $Z = \{(x_1, y_1), ..., (x_n, y_n)\}$ with $\forall_i y_i = c(x_i)$ and outputs a hypothesis $\hat{c} \in H$

Definition 2
A learning algorithm A can learn a concept c if there is a polynomial function $poly(\cdot, \cdot, \cdot)$ such that:

- for any distribution D on X and
- for any $0 < \epsilon < 1/2$ and $0 < \delta < 1/2$,

if A receives as input a sample of size $n \geq poly(\frac{1}{\epsilon}, \frac{1}{\delta}, size(c))$, then A outputs $\hat{c}$ such that:

$$P_{Z \sim D^n}(R(\hat{c}) \leq \epsilon) \geq 1 - \delta$$

Definition 3
A concept class C is **PAC learnable** from a hypothesis class H if there is an algorithm that can learn any concept in C.

Definition 4
If A runs in time polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$ we can say that C is **efficiently PAC learnable**

13.4 Example Rectangle learning

Setting:
$C =$ concept of all axis-aligned rectangles
$H = C$

To prove that this is PAC learnable we now need to define an algorithm A and show that using that algorithm that definition 2 is fulfilled. So we need to show that:

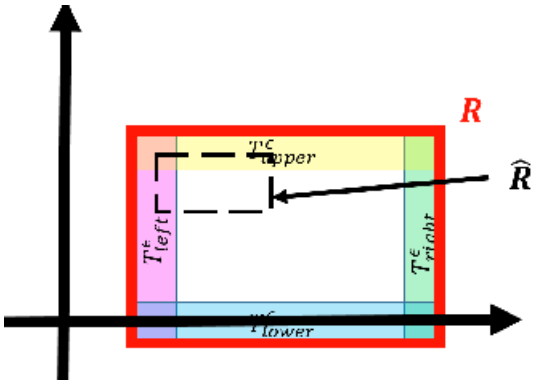
There exists a $poly(\cdot, \cdot, \cdot)$ such that:

- for any rectangle $R \in C$
- for any distribution $DinR^2$
- for any $\epsilon > 0$ and $\delta > 0$

if A receives a sample Z of size $n \geq poly(\frac{1}{\epsilon}, \frac{1}{\delta}, size(R = 4)$, then:

$$P(R(\hat{R}) \leq \epsilon) \geq 1 - \delta$$

Proof Idea:
We define A as: Take the tightest fitting rectangle.
Next we define $\hat{R}IG$ as a rectangle that intersect with all four strips:



We set the strips such that $P($a sample lands in $Strip_i) = \frac{\epsilon}{4}$

Important to note is that we have the 0,1 loss function (corresponding measure is accuracy). Meaning that a loss of ϵ also means an accuracy of ϵ.
Note that ϵ is the error parameter. Also observe that $R(\hat{R})$ is the actual error. In this case the error would be the difference from he true rectangle R and our estimate rectangle R.
We now want to prove (this would conclude the proof):

$$P(R(\hat{R}) \leq \epsilon) \geq P(\hat{R}IG) \geq 1 - 4exp(-\frac{n\epsilon}{4}) \geq 1 - \delta$$

The last inequality holds with:

$$n \geq \frac{4}{\epsilon} \ln(\frac{4}{\delta})$$

The first inequality:
The second inequality:

13.5 Consistent hypothesis and finite hypothesis classes

Assumptions:

- $H = C$
- $|H| \leq \infty$
- $min_{h \in H} R(h) = 0$ (A returns a consistent hypothesis for any concept C and sample Z)

If:

$$n \geq \frac{1}{\epsilon}(\log(|C|) + \log(\frac{1}{\delta})$$

Then it holds:

$$P(R(\hat{c} \leq \epsilon) \geq 1 - \delta \qquad \text{Success probability}$$
$$P(R(\hat{c}) > \epsilon) \leq \delta \qquad \text{Error probability}$$

13.6 General stochastic setting

We will always assume $H = C$ Now we cant assume anymore that the true classifier has zero risk. (For example if we have the same feature for two patients but one is actually sick and the other one isn't). We model this with a distribution D on $X \times \{0, 1\}$. So we draw $Z = \{(x_1, y_1), ..., (x_n, y_n)\}$ from D.
So now our objective will be to minimize the difference of the risk of an optimal classifier with the risk of our classifier:

$$R(\hat{c}) - inf_{c \in C} R(c) \leq \epsilon$$

This gives rise to the General PAC learning model.

13.6.1 General PAC learning model

A learning algorithm A can learn a concept class C from H if there is a polynomial function $poly(\cdot, \cdot, \cdot)$ such that:

- for any distribution D on X and
- for any $0 < \epsilon < 1/2$ and $0 < \delta < 1/2$,

if A receives as input a sample of size $n \geq poly(\frac{1}{\epsilon}, \frac{1}{\delta}, dim(X))$, then A outputs $\hat{c} \in H$ such that:

$$P_{Z \sim D^n}(R(\hat{c}) - inf_{c \in C} R(c) \leq \epsilon) \geq 1 - \delta$$

13.6.2 Error bounds

If $|C| \leq \infty$:

$$P(R(\hat{c_n}^* - inf_{c \in C} R(c) > \epsilon) \leq 2|C| exp(-2n\epsilon^2)$$

If $|H| = \infty$ and $VC_C > 2$:

$$P(R(\hat{c_n}^* - inf_{c \in C} R(c) > \epsilon) \leq 9n^{VC_C} exp(-\frac{n\epsilon^2}{32})$$

Some other useful bounds:

$$R(\hat{c}) - inf_{c \in C} R(c) \leq 2 sup_{cinC} |R(c) - \hat{R}_n(c)|$$