IML

Nicolas Muntwyler

Spring 2020

Contents

1 Introduction

1.1 Infos

Übungsstunde: Mittwoch 15.00 - 18.00
Project: 30% final grade. https://project.las.ethz.ch

1.2 Notation

Every variable with a hat comes from Data.

1.3 Recap

$$
\begin{aligned}
(\mathbf{AB})^{-1} &= \mathbf{B}^{-1}\mathbf{A}^{-1} && (1)\\
(\mathbf{ABC}...)^{-1} &= ...\mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1} && (2)\\
(\mathbf{A}^T)^{-1} &= (\mathbf{A}^{-1})^T && (3)\\
(\mathbf{A}+\mathbf{B})^T &= \mathbf{A}^T+\mathbf{B}^T && (4)\\
(\mathbf{AB})^T &= \mathbf{B}^T\mathbf{A}^T && (5)\\
(\mathbf{ABC}...)^T &= ...\mathbf{C}^T\mathbf{B}^T\mathbf{A}^T && (6)\\
(\mathbf{A}^H)^{-1} &= (\mathbf{A}^{-1})^H && (7)\\
(\mathbf{A}+\mathbf{B})^H &= \mathbf{A}^H+\mathbf{B}^H && (8)\\
(\mathbf{AB})^H &= \mathbf{B}^H\mathbf{A}^H && (9)\\
(\mathbf{ABC}...)^H &= ...\mathbf{C}^H\mathbf{B}^H\mathbf{A}^H && (10)
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial \mathbf{x}^T\mathbf{a}}{\partial \mathbf{x}} &= \frac{\partial \mathbf{a}^T\mathbf{x}}{\partial \mathbf{x}} = \mathbf{a} && (69)\\
\frac{\partial \mathbf{a}^T\mathbf{X}\mathbf{b}}{\partial \mathbf{X}} &= \mathbf{a}\mathbf{b}^T && (70)\\
\frac{\partial \mathbf{a}^T\mathbf{X}^T\mathbf{b}}{\partial \mathbf{X}} &= \mathbf{b}\mathbf{a}^T && (71)\\
\frac{\partial \mathbf{a}^T\mathbf{X}\mathbf{a}}{\partial \mathbf{X}} &= \frac{\partial \mathbf{a}^T\mathbf{X}^T\mathbf{a}}{\partial \mathbf{X}} = \mathbf{a}\mathbf{a}^T && (72)\\
\frac{\partial \mathbf{X}}{\partial X_{ij}} &= \mathbf{J}^{ij} && (73)\\
\frac{\partial (\mathbf{X}\mathbf{A})_{ij}}{\partial X_{mn}} &= \delta_{im}(\mathbf{A})_{nj} = (\mathbf{J}^{mn}\mathbf{A})_{ij} && (74)\\
\frac{\partial (\mathbf{X}^T\mathbf{A})_{ij}}{\partial X_{mn}} &= \delta_{in}(\mathbf{A})_{mj} = (\mathbf{J}^{nm}\mathbf{A})_{ij} && (75)
\end{aligned}
$$

- Hessian of $(y_i - w^T x_i)$ is: $x_i x_i^T$
- Hessian of $\lambda w^T w$ is: λI
- $\dfrac{\partial w^T X^T X w}{\partial w} = 2X^T X w$
- $\dfrac{\partial -2w^T X^T y}{\partial w} = -2X^T y$
- $\dfrac{\partial \lambda w^T w}{\partial w} = 2\lambda w$
- $\dfrac{\partial f(v)}{\partial v} = \dfrac{\partial \frac{1}{1+e^{-v}}}{\partial v} = f(v)(1-f(v))$

2 Nici Stupid

- $\|Xw - y\|_2 =< Xw - y, Xw - y >$
- $< Xw - y, Xw - y >= w^T X^T XW - 2w^T X^T y + y^T y$
- $X^T X$ is symmetric and positive semi-defnite
- positive semi-definite have non-negative eigenvalues
- Determinante is the product of all algebraic eigenvalues
- Invertible iff determinante is nonzero
- $f_{X|Y}(x|y) = \frac{f_{X,Y}(x,y)}{f_Y(y)}$
- $E_{X,Y}[f(x,y)] = E_X[E_Y[f(x,y)|X = x]]$

3 Matrices

- Orthogonal matrices:
-
  - $A^{-1} = A^T$
  - $AA^T = I$ (only orthonormal)
  - $A^T A = I$
- The eigenvectors of the Eigenvalue decomposition of the Covariance Matrix of (de-meaned) datapoints in R^d is a basis for R^d
- Covarince-Matrix: $\sum = \frac{1}{n} \sum_{i=1}^{n} x_i x_i^T$
- Eigenvalue Decompositioin is: $\sum_{i=1}^{n} \lambda_i v_i v_i^T$
- Covariance Matrix $\sum = X^T X$ where X = DataMatrix
- SVD decomposition: X = $V \Sigma U^T$, where V,U are orthogonal and singular values: $\sigma \geq 0$
- Big singular value = Big Variance component (corresponding to that singular value)
- Small singular value = Small Variance component (corresponding to that singular value)

4 Supervised Learning

$f : X \to Y$
X: Input (E-Mail)
Y: Label (Spam, Nonspam)
Aim: How to find this function.

You always define 5 things:

Representation/features	Linear hypothesis Nonlinear hypotheses through feature transformation Kernels
Model/objective	Loss-function (Squared loss, l_p-loss) + Regularization (l_2-norm, l_1-norm, l_0-penalty) 0/1 loss Perceptron loss Hinge loss cost sensitive loss multi-class hinge loss reconstruction error
Method	Exact Solution Gradient Descent (mini-batch) SGD Reductions Lloyd's heuristic
Evaluation metric	Mean squared error Accuracy F1 score AUC Confusion matrices compression performance
Model Selection	K-fold Cross-Validation Monte Carlo CV

Representation:
Bag of Words in R^d vector
Problem: Some words like "the", "a" will have many appearances. Length of the document should not matter. Some words are more "important" than others. Order is ignored.

Key: Trading goodness of fit and model complexity.

4.1 Linear Regression

Part of supervised learning.

Find f(x) = y
If we choose f to be linear we have: $f(x1, x2, x3) = W : 1x_1 + W_2x_2 + W_3x_3 + w_0$
We just write this homogeneously: $f(x) = w^T x$

Goodness of fit: Least Squares (vertically).
Error = Cost = $\hat{R}(w)$

We made 2 choices for how we define **Goodness of fit:**

- Quantify error for point via squared residual
- We sum over all points

So the weights for the minimal LSE-error is: $\hat{w} = (x^T x)^{-1} X^T y$.
X := Every data point is a row.

Def. 4.1. Convex
$f : R^d \to R$ is convex iff:
$\forall x, x', \lambda$ with $\lambda \in [0,1]$ if holds that: $f(\lambda x + (1-\lambda)x') \leq \lambda f(x) + (1-\lambda) f(x')$

LSE objective function: $\hat{R}(w) = \sum (y_i - w^T x_i)^2$. Which is convex :)
$\Longrightarrow$ We just have to walk downhill to get the best w. ($\Longrightarrow$**Gradient Descent**)

Gradient Descent
Gradient (∇): Direction in which the function increases the most. (Hence we walk in the negative direction)

Def. 4.2. $\nabla \hat{R}(w) = [\frac{\delta}{\delta_{w_1}} \hat{R(w)}, ..., \frac{\delta}{\delta_{w_y}} \hat{R(w)}] = [-2 \sum_{i=1}^n r_i x_i]$

$$w_{t+1} = w_t - \mu_t \nabla \hat{R}(w_t)$$

w_0 is just an arbitrary $w_0 \in R^d$
μ_t is the learning rate = (step size)

We choose a step size of $\frac{1}{2}$ and we get linear convergence.

Def. 4.3. Linear Convergence
$\forall t \geq t_0 \exists \alpha \leq 1$
$\underbrace{(\hat{R}(w_{t+1}) - \hat{R}(\hat{w})}_{\epsilon} \leq \alpha(\hat{R}(w_t) - \hat{R}(\hat{w}))$

If we have linear convergence we can find an ϵ-optimal solution in $O(\ln \frac{1}{\epsilon})$ iterations.

How to choose stepsize (Adaptive Stepsize)

- Line Search
- "bold driver" heuristic

Bold Driver: Increase stepsize if we get a smaller error in the next iteration and decrease stepsize if we get a bigger error.

Complexity
Closed Form : $O(d^3 + nd^2)$
Gradient descent: $O(nd) \cdot O(\frac{1}{\epsilon})$

Other loss functions
As a measure of goodness of fit we covered LSE, but we can have other loss functions.
LSE is $l_2(r) = r^2$ (r:= residual). We could use $l_1(r) = |r|$, or $l_p(r)$ with p»1 (give value to big residuals) or p ≤ 1 (Give value to small residuals).

4.2 Linear regression for polynomials

We fit a non-linear function:

$$f(x) = \sum_{i=1}^{d} w_i \phi_i(x)$$

$\phi(x)$ is the vector of all monomials in $x_1, ..., x_p$.

We would like to know what our expected prediction error will be. (True Risk)

Def. 4.4. True Risk (= Expected Error)

$$R(w) = \int P(x,y)(y - w^T x)^2 dx dy = E_{x,y}[(y - w^T x)^2]$$

P(x,y) := Unknown Distribution (Our data set is iid $\sim$ P(x,y))

However since we don't know this distribution we would like to estimate our prediction error with:

Def. 4.5. Empirical Risk

$$\hat{R}_D(w) = \frac{1}{|D|} \sum_{(x,y)\in D} (y - w^T x)^2$$

We choose w such that the Empirical Risk is minimized (=**ERM**).
But careful: After we trained to receive a w such that $E_D[\hat{R}_D(w)]$ is minimized, we don't have $E_D[\hat{R}_D(w)] = E_D[R_D(w)]$ (even if the law of large numbers suggest it) because it also depends on what we trained on. So of course we have a low $E_D[\hat{R}_D(w)]$ if we specifically train our network to minimize that. But if we apply it on other data that comes from the same unknown distribution (P(x,y)) we would get a higher Risk, i.e. $E_D[R_D(w)]$ is higher.
Mathematically you get:

$$E_D[\hat{R}_D(w)] \leq E_D[R_D(w)]$$

Therefore we have a Training Set of Data and a separate set of test data. With that we get:

$$E_{D_{train},D_{test}}[\hat{R}_{D_{test}}(\hat{w}_{D_{train}})] = E_{D_{train}}[R(\hat{w}_{D_{train}}]$$

$\Longrightarrow$ Use an independent test set

How to choose degree (m) of polynomial
We would like to:

1.) $\hat{w}_m = \underbrace{argmin}_{w:degree(w)\leq m} \quad R_{train}(\hat{w}_m)$

2.) $\hat{m} = \underbrace{argmin}_{m} \hat{R}_{test}(\hat{w}_m)$

3.) Train with degree $\hat{m}$

However we need the test set to not influence our decision of w. Therefore we split our training data D into D_{train} and D_{test}. There are two methods of how we could do that. Both use **cross-validation:**

Randomly (Monte-Carlos cross-validation)

- Pick training set of given size u.a.r.
- Validate on remaining points
- Estimate prediction error by averaging the validation error over multiple random trials

k-fold cross-validation (Default)

- Partition the data into k "folds"
- Train of (k-1) folds, evaluating on remaining fold
- Estimate prediction error by averaging the validation error obtained while varying the validation fold.

With that we have the following algorithm (For a k-fold cross-validation):

1.) For $1 \leq i \leq k$ do:

2.) Split the Data D into $D^{(i)}_{train} \uplus D^{(i)}_{val}$

3.) $\hat{w}_{i,m} =$ argmin $\hat{R}^{(i)}_{train}(w_m)$

4.) $\hat{R}^{(i)}_m = \hat{R}^{(i)}_{val}(\hat{w}_{i,m})$

5.) $\hat{m} =$ argmin $\frac{1}{k}\sum_{i=1}^{k} \hat{R}^{(i)}_m$

Regularization
Overfitting results in large weights. Therefore we would like to encourage small weights via penalty functions. We get Ridge regression:

$$\min_w \frac{1}{n}\sum_{i=1}^{n}(y_i - w^T x_i)^2 + \lambda \|w\|_2^2$$

With gradient descent we get: $w_{t+1} = (1 - 2\lambda\mu_t)w_t - \mu_t \nabla \hat{R}(w_t)$
With closed form we get: $\hat{w} = (X^T X + \lambda I)^{-1} X^T y$

!! Scaling matters. Therefore we often normalize our data beforehand !!

How do we choose λ? Again with $\lambda \in \{10^{-6}, ..., 10^{6}\}$ and then cross-validation.

4.3 Linear Classification

Instead of predicting a number we now want to predict a discrete label (spam (+) /not-spam (-))
We usually denote a binary classification with label + and label - (so the label is the sign($w^T x$)) So for linear Classification we want to find a function:

$$h(x) = sign(w^T x)$$

So the decision boundary is where $w^T x = 0$

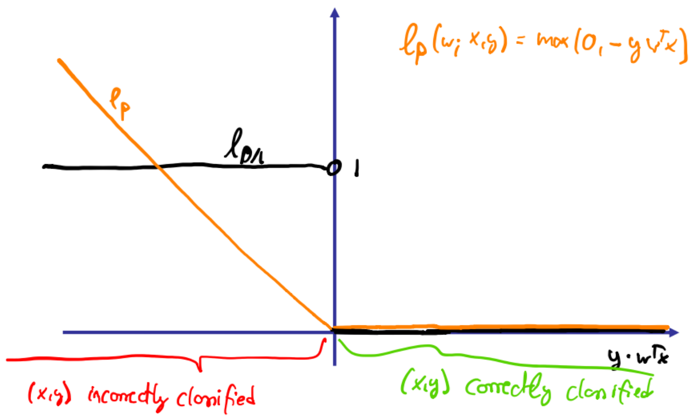
4.3.1 How to find the weights (i.e. h(x))

Idea: Minimize number of mistakes.

$$\hat{w} = \underbrace{argmin}_{w} \frac{1}{n}\sum_{i=1}^{n} l_{0/1}(w; x_i, y_i)$$

where $l_{0/1} = \begin{cases} 1 & sign(w^T x_i) \neq y_i \\ 0 & o/w \end{cases}$ Problem: This is NP-hard (not convex, not differentiable)

Therefore we use a **Surrogate loss function** (z.B, $l_P(w; x, y) = max(0, -yw^T x)$ = **Perceptron loss**):



So now we want to solve:

$$\hat{w} = argmin_w \frac{1}{n}\sum_{i=1}^{n} l_P(w; x_i, y_i)$$

Note: This is only piecewise differentiable. But this is fine for GD.
How? With Gradient descent, or better stochastic gradient descent.

4.3.2 Stochastic Gradient Descent

1.) Start at an arbitrary $w \in R^d$

2.) For t = 1,2,.. do:

3.) Pick data point (x',y') $\in D$ from training set uar. (with replacement), and set:
$w_{t+1} = w_t - \eta_t \nabla l(w_t; x', y')$

With: $\sum_t \eta_t = \infty$ and $\sum_t \eta_t^2 < \infty$ (z.B. $\eta_t = \frac{1}{t}$) However instead of just using one data point we most often use a batch of data points.

4.3.3 Perceptron Algorithm

The Perceptron Algorithm us just SGD on the Perceptron loss function l_P with learning rate 1.

Satz 4.1. If the data is linearly separable, the Perceptron will obtain a linear separator.

4.3.4 Support Vector Machines (SVM)

The Perceptron Algorithm had some "problems".

- Doesn't settle in scenarios where there is high noise and we can't completely separate.
- Perfect solution could be w=0.

So in order to counter that we use the hinge-loss:

$$\hat{w} = \underbrace{argmin}_{w} \frac{1}{n} \sum_{i=1}^{n} max\{0, 1 - y_i w^T x_i\} + \lambda \|w\|_2^2$$

We solve that with SGD with learning rate $\eta_t = \frac{1}{\lambda t}$:

$$w_{t+1} = w_t(1 - \frac{2\lambda}{n}) + [y_i w^T x_i <] \cdot y_i x_i$$

The SVM Algorithm maximized the margin.(margin := Distance from the boundary to the nearest data point)

Note: The maximizing of the margin is achieved by minimizing the weights, because: If $y_i w^T x_i \geq 1$, then margin$\geq \frac{1}{\|w\|}$
Note: We converge in separable and non-separable cases How to choose λ: Again (as in regression) we do cross-validation

4.4 Linear Classification for polynomials

Extend the current feature space and fit there a linear line.
TODO: Include Picture
Other impretation: Don't fit a line, but fit a polynomial as the separation line.
Note: polynomial doesn't has to be positive definite

5 Feature Selection

z.B. We have a function f(x) = 4 + 2x + sin(x) and we now want to find a curve that fits our function.
However we don't know f(x) so we don't know which features to choose. Let's say we choose a constant feature (tries to fit a line f(x) = c) together with a linear feature (tries to fit a line f(x) = c_1x + c_2).
Eventually (by hand) we get that we need a constant, a linear and a sinus feature to get a good result.
Now we want to automatize this:

5.1 Greedy feature selection

5.1.1 Forward greedy

- start with no feature
- from the set of all possible feature we try to fit, choose the one that gives the best results (by cross-validation)
- repeat while we improve (otherwise we stop)

Feature Intuition: First start with only choosing one feature hence only x_1 (git a constant line), then in the next iteration we take the next feature also hence x_1 and x_2 where we then can fit a nice linear line.

Problem: Can't tell difference between informative from irrelevant features.

5.1.2 Backward greedy

Start with all features and drop the feature that we need the least.
Tends to be better but it also not optimal.
Problem: Expensive (z.B. 100000 features)

5.2 Feature Selection while training

We add (like the $\lambda \left\lVert w \right\rVert_2^2$) a "+ $\lambda \left\lVert w \right\rVert_0 \leq k$" to the optimization function.
Here $\left\lVert w \right\rVert_0$:= number of zeros in w
However $\left\lVert w \right\rVert_0$ is not good to do math with so we use $\left\lVert w \right\rVert_1$ instead (**Lasso-Trick**). Turns out this works

With that our weights not just go slowly to zero for big lambdas as with the $\left\lVert w \right\rVert_2$ but actually go to 0 for some weights. Which is exactly what we want. With that many features actually get a weight zero (z.B. x wight for x^2, sin(x) and 3x get weight zero but feature x^3 and 5x get a nonzero weight).
So we more or less done feature selection while training.

Again this method doesn't work always but most often it does.
Note: l_1-Norm is convex
How to solve? SGD (doesn't converge fast), Recent work on proximal mathods works well

Problem: only works for linear models

6 Kernels

Look at non-linear regression/classification.
The important thing is that we avoid the feature explosion:
For example in regression we had a sneak peak at non-linear features with:

$$f(x) = \sum_{i=1}^{d} w_i \phi_i(x)$$

where ϕ is a non-linear transform. Now for higher-dimensions and higher-degree we need to generate all possible monomials. You can easily see that the number of monomials (features) explodes.
However we can use the following:

Satz 6.1. The optimal hyperplane lies int the span of the data:

$$\hat{w} = \sum_{i=1}^{n} \alpha_i y_i x_i$$

In other words: This is an other way to express $\hat{w}$.
We can use this such that in our optimization formula we get a crossproduct, which can be computed with the kernel. (very fast).

6.1 Kernelized Perceptron

$$\hat{w} = argmin_w \frac{1}{n} \sum_{i=1}^{n} max(0, -y_i w^T x_i)$$

we then can replace w with Satz 4.1, to get:

$$\hat{\alpha} = argmin_\alpha \frac{1}{n} \sum_{i=1}^{n} max(0, -y_i (\sum_{j=1}^{n} \alpha_j y_j x_j)^T x_i)$$

$$\hat{\alpha} = argmin_\alpha \frac{1}{n} \sum_{i=1}^{n} max(0, -y_i \sum_{j=1}^{n} \alpha_j y_j (x_j^T x_i))$$

$$\hat{\alpha} = argmin_\alpha \frac{1}{n} \sum_{i=1}^{n} max(0, -y_i \sum_{j=1}^{n} \alpha_j y_j k(x_j, x_i))$$

Prediction: $\hat{y} = sign(\sum_{i=1}^{n} \alpha_i y_i k(x_i, x))$

Now what is $k(x_j, x_i)$? It is a function that fulfills:

Lemma 6.1. Kernels need to be:

- symmetric: $k(x, x') = k(x', x)$
- positive semidefinite:
  For any n, any set $S = \{x_1, ..., x_n\} \subseteq X$ (X = Dataspace) the Kernel matrix K is positive semidefinite

$$K = \begin{bmatrix} k(x_1, x_1) & ... & k(x_1, x_n) \\ \vdots & & \vdots \\ k(x_n, x_1) & ... & k(x_n, x_n) \end{bmatrix}$$

Satz 6.2. A symmetric matrix is positive semidefinite iff:

- $\forall x \in R : x^T M x \geq 0$
- All eigenvalues of M ≥ 0

Now there are different kernels for different feature spaces. Lets say that we wanted to have a linear perceptron. We then would have to choose:

$$k(x, x') = x^T x'$$

So in this case it would not be different.
However lets say we wanted to have all monomials up to degree d as our feature space. Of course we would have a feature explosion. But with the kernelized version we just have to choose the following kernel:

$$k(x, x') = (x^T x' + 1)^d$$

It is proven that this gives us the same result. Only this time we don't have the feature explosion.

We now ask ourself which feature space can be kernelized. Or even which kernel (has to fulfill Lemma 4.1 of course) corresponds to which feature space.

6.2 Kernel Engineering

Here is a list of the most common kernels:
TODO: Import Image of kernels

You can construct further Kernels by combining them:
TODO: Import Image of Slide 5 in kernel 3

6.3 An other Kernel Interpretation

The kernel approach can also be seen as how much weight I want to give each initial feature.
So here in the following example we have a Dataset of 4 points. We then want to predict the label for a new x. So for each Datapoint (x_i, y_i) we calculate $\alpha_{ii} k(x_i, x)$ and then sum up. Since the gaussian kernel will output small values if x is far away from the current datapoint we automatically don't give much weight to that datapoint. An other factor to how much we want to weight that datapoint is the $alpha_i$, which is trained by our learning algorithm.
So in some sort the learning algorithm can scale the magnitude of each gaussian bump

6.4 Kernel as similarity function

Using a gaussian kernel we are very much like a k-nearest-neighbour predictor. (look at k-nearest neighbours of x and dicide on the majority of their labels).
This is because of the fact we discussed above. The gaussian kernel gives almost 0 weight to datapoints that are far away of our x (that we want to predict its label).
TODO: import image with the gaussian bumps

6.5 Kernelized SVM

$$\hat{\alpha} = argmin_\alpha \frac{1}{n} \sum_{i=1}^{n} max(0, 1 - y_i \alpha^T k_i) + \lambda \alpha^T D_y K D_y \alpha$$

$$K = \begin{bmatrix} k(x_1, x_1) & ... & k(x_1, x_n) \\ \vdots & & \vdots \\ k(x_n, x_1) & ... & k(x_n, x_n) \end{bmatrix} D_y = \begin{bmatrix} y_1 & & 0 \\ & \ddots & \\ 0 & & y_n \end{bmatrix} k_i = [y_1 k(x_i, x_1), ..., y_n k(x_i, x_n)]$$

Prediction: $\hat{y} = sign(\sum_{i=1}^{n} \alpha_i y_i k(x_i, x))$

6.6 Kernelized linear Regression

$$\hat{\alpha} = argmin_\alpha \frac{1}{n} \|\alpha^T K - y\|_2^2 + \lambda \alpha^T K \alpha$$

Closed form solution for the alphas: $\hat{\alpha} = (K + n\lambda I)^{-1} y$
Prediction: $\hat{y} = \sum_{i=1}^{n} \hat{\alpha}_i k(x_i, x)$

6.7 Kernelized Percpetron (again)

$$\hat{\alpha} = argmin_{\alpha} \frac{1}{n} \sum_{i=1}^{n} max(0, -y_i \alpha^T k_i)$$

Prediction: $\hat{y} = sign(\sum_{i=1}^{n} \alpha_i y_i k(x_i, x))$

6.8 Semi-parametric regression

If the ground truth function is linear + periodic our methods fail. Because till now we either had a linear/polynomial or gaussian kernel, but we would need a combination. This we can actually do and engineer our kernel to do exactly that:

$$k(x, x) = c_1 \cdot exp(\|x - x'\|_2^2 / h^2) + c_2 x^T x'$$

6.9 Choosing Kernels

Cross-Validation

7 Class Imbalance

TODO: import slide 2
Naive Solution:

- Subsampling
- Upsampling

However both ideas have big disadvantages. Therefore we use: **Cost-sensitive classification** methods.

If you have much less datapoints ob label1 (+) than of labell2 (-) you adapt you lossfunction. For label1 you multiply with c_+ and for label2 you multiply with c_-, where $c_+ >> c_-$.
With that we get the for example the following cost-sensitive loss-functions for the Perceptron:
$l_{CS-P}(w; x, y) = c_y l(w; x, y)$

7.1 Evaluating Accuracy

- Accuracy: $\frac{TP+TN}{TP+TN+FP+FN}$
- Precision: $\frac{TP}{TP+FP}$
- Recall: $\frac{TP}{TP+FN}$
- F1-score: $\frac{2TP}{2TP+FP+FN}$

To have a metric you then either use:

- Precision, Recall, F1, etc...
- ROC / Precision Recall curves, Area under Curve (AUC)

Precision Recall Curve
X-Axis:Recall
Y-Axis: Precision

Receiver Operator Characteristics (ROC) Curve
X-Axis: True positive rate (TPR, =recall)
Y-Axis: False positive Rate (FPR

TPR = $\frac{TP}{TP+FN}$ TFR = $\frac{FP}{TN+FP}$

8 Multi-Class Problems

When you need to classify for multiple classes. There are multiple approaches. In the first two you use the binary classifier whilst for the multiclass SVM you do something different.

8.1 One-vs-all

- Solve c binary classifiers, one for each class (with labels: (+)= all from class i, (-)=all others)
- Classify using the classifier with largest confidence

For the confidence you need to be careful because $sign(\alpha w^T x)$ and $sign(w^T x)$ have the same decision boundary but a different confidence. Therefore you normalize the weights to unit length.
(In practise you often don't have to do that when using regularization. Its more or less ok).

8.2 One-vs-one

- Train c(c-1)/2 binary classifiers, one for each pair of classes (i,j) (with labels: (+): all from class i, (-): all from class j)
- Class with highest number of positive prediction wins

8.3 Multi-class SVM

Here the idea is to do it while training. You keep c different wight vectors w_i, where c is the number of classes. The prediction is done like this:

$$\hat{y} = argmax_i w^{(i)T} x$$

In order to do that we change our loss function to :

$$l_{MC-H}(w^{(1)}, ..., w^{(c)}; x, y) = \max\left(0, 1 + \max_{j \in \{1,...,y-1,y+1,...,c} w^{(j)T} x - w^{(y)T} x\right)$$

8.4 Evaluation

Now you may want to use the **Confusion Matrix**

9 Neural Nets

Before we had (normal regression) with m features:

$$w^* = argmin_w \sum_{i=1}^{n} l(y_i; \sum_{j=1}^{m} w_j \phi_j(x_i))$$

Now the Idea is to parameterize the feature maps and optimize over the parameters. (Because choosing the right features is hard, so we also let the machine figure that out)

$$w^* = argmin_{w,\theta} \sum_{i=1}^{n} l(y_i; \sum_{j=1}^{m} w_j \phi(x_i, \theta_j))$$

Often: $\phi(x,\theta) = \varphi(\underbrace{\theta^T x}_{z})$.

We note: $f(x_i, w_\theta) = \sum_{i=1}^{m} w_j \underbrace{\varphi(\theta^T x)}_{v_j}$

Examples for $\varphi(z)$:

- (Sigmoid): $\varphi(z) = \frac{1}{1+exp(-z)}$
- /Tanh): $\varphi(z) = tanh(z) = \frac{exp(z)-exp(-z)}{exp(z)+exp(-z}$
- (Rectified linear units = ReLU): $\varphi(z) = max(z, 0)$

Graphical we have vertically the features and horizontally the layers.

9.1 Prediction

We just use **forward propagation:**

- Input layer: $v^{(0)} = x$
- Hidden layer: for l = 1..(L-1):
  - $z^{(l)} = W^{(l)} v^{(l-1)}$
  - $v^{(l)} = \varphi(z^{(l)})$
- Output layer: $f = W^{(L)} v^{(L-1)}$

$W = (W^{(1)}, .., W^{(L)})$

And then have our prediction:

- Regression: y = f
- Classifier(binary): y = sign(f)
- Classifier: y = $argmax_i = f_i$

9.2 Training

Sadly now its not convex anymore. However in practise first order methods and SGD still work fine. To actually have the gradients we need backpropagation:

- For the output layer:
- - Compute "error": $\delta^{(L)} = l'(f) = [l'(f_1), ..., l'(f_p)]$
  - Gradient: $\nabla_{W^{(L)}} l(W; y, x) = \delta^{(L)} v^{(L-1)T}$
- For each hidden layer l = L-1,..,1:
- - Compute "error": $\delta^{(l)} = \varphi'(z^{(l)}) \odot \left(W^{(l+1)T\delta^{(l+1}} \right)$
  - Gradient: $\nabla_{W^{(l)}} l(W; y, x) = \delta^{(l)} v^{(l-1)T}$

Careful: Almost everything is a vector in this notation And then we just to SGD:

- Initialize weights W
- For t = 1,2,..
-
  - Pick data point (x,y)$\in D$ u.a.r.
  - Take step in negative gradient direction:
    $W \leftarrow W - \eta_t \nabla_w l(W; y, x)$

9.3 Initializing the weights

Idea: Keep variance of weights approximately constant across layers to avoid vanishing and exploring gradients.

n_{in} := number of nodes that have a connection to your input.

- Glorat (tanh): $w_{i,j} \sim N(0, \frac{1}{n_{in}}$
- Glorat (tanh): $w_{i,j} \sim N(0, \frac{2}{n_{in}+n_{out}}$
- He (ReLU): $w_{i,j} \sim N(0, \frac{2}{n_{in}}$

9.4 Learning rate

$\eta_t = \min(0.1, \frac{100}{t})$

9.5 Learning with momentum

Sadly the functions are now not convex anymore. Hence with SGD we might get stuck in a local minimum. Therefore we can apply the learning with momentum SGD:

$$a \leftarrow m \cdot a + \eta_t \nabla_w l(W; x, y)$$

$$W \leftarrow W - a$$

9.6 Avoid overfitting

Using a Regularizer

$$\hat{W} = argmin_W \sum_{i=1}^{n} l(W; x_i, y_i) + \lambda \|W\|_F^2$$

Early Stopping Usually overfitting comes over time while training. So we can just stop a little bit earlier. We monitor prediction performance on a validation set and stop as soon as the validation error increases.

Dropout We randomly ignore units(nodes) in the hidden layers each with probability 0.5 during each iteration of SGD.
Then after training we half the weights to compensate.

9.7 Batch normalization

We know, that we have to standardize our datapoints. However after each layer of the neural network the inputs are scaled and shifted through each layer. Therefore we have to use Batch Normalization. It is a widely used technique to normalize the inputs to each layer.

10 Clustering

There are 3 types of approaches:

- Hierarchical clustering (Build tree then cut)
- Partitional approaches (build graph and cut edges)
- Model-based approach

We will look at the Model-based approach (the other two are not connected to ML).

There are different "models" in this approach:

- k-means
- Gaussian mixture

10.1 k-Means clustering

We represent each cluster by a single point (center).
Each point is assigned to its closest center.

- Initialize cluster centers:
  $\mu^{(0)} = [\mu_1^{(0)}, ..., \mu_k^{(0)}]$
- While not converged:
- - Assign each point x_i to closest center:
    $z_i^{(t)} = arg\ j \in \overset{min}{\{1, ..., k\}}\ \|x_i - \mu_j^{(t-1)}\|_2^2$
- Update center as mean of assigned data points:
- - $\mu_j^{(t)} = \frac{1}{n_j} \sum_{i:z_i^{(t)}} x_i$

Where $n_j = |\{i : Z_i^{(t)} = j\}|$.

10.1.1 How to initialize

We use adaptive seeding:

- Start with a random data point as the center
- Add centers 2 to k randomly. proportionally to squared distance to closest selected center

Using this initialization the whole approach is then called: **K-Means++**.

10.1.2 Model selection

:= Determining the number of clusters.
There are three approaches:

Heuristic quality measures
Do the algorithm for k = 2..10 and then look at the error curve and pick the elbow-point.

Regularization Is like the elbow method by introducing a penalty for choosing more clusters:
$min_{k,\mu_{1..k}} R(\hat{\mu}_{:k}) + \lambda \cdot k$

Information theoretic basis This we will learning in Statistical Learning Theory

10.1.3 Problems

We still can't model clusters of arbitrary shape. This we will learn later via **kernel k-means**.
Determining the number of clusters is still an unsolved problem.

11 Dimension Reduction

Analog to regression from supervised learning.

Goal: Find function: $f : R^d \longrightarrow R^k$ where $k << d$
(Here we go from $x_i \in R^n$ to $z_i \in R^k$
Problem: Which function would we prefer?

We want a function that minimizes the reconstruction error:

$$(W, z_1, ..., z_n) = argmin \sum_{i=1}^{n} \|Wz_i - x_i\|_2^2$$

$W \in R^{d*k}$ is orthogonal, $z_1, ..., z_n \in R^k$ and $\|w\|_2 = 1$

Visual example: If you have k=1, we would want to project every datapoint onto a line w. The z_i would represent the scale factor of w such that $z_i w \approx x_i$. Now if we transform x_i into our new (smaller) space, the representation of x_i in the new space (hence in case of k=1, on this line) is defined by z_i. From $z_i w \approx x_i$ we get:

$$z_i = W^T x_i$$

!!! Here z_i = the transformed point from x_i which is just the i-th datapoint !!!

So we just need to know what W is.
We can replcae z_i in our original argmin function and with some derivation one gets to the following solution:

Solution:

$$W = (v_1|...|v_k)$$

Where $\Sigma = \sum_{i=1}^{n} \lambda_i v_i v_i^T$ with $\lambda_1 \geq ... \geq \lambda_n$ (=Eigenvalue decomposition)
and $\Sigma = \frac{1}{n} \sum_{i=1}^{n} x_i x_i^T$ (=Empirical Covariance Matrix).

Note: The data needs to be centered!!
Note2: Choosing k is not clear. Use Elbow-Trick.

=> This is just PCA!!

11.1 Using Kernels

For some datasets we have that the data is not linearly separable (e.g. circle dataset). So we want to apply the kernel trick.
We again assume (or know):

$$w = \sum_{i=1}^{n} \alpha_i x_i (\text{ case: k=1})$$

We use this to apply the kernel trick. We can again prove that this will just result in PCA, but scaled!

Solution: A point x in R^d is projected as a point $z \in R^k$ like this:

$$z_i = \sum_{j=1}^{n} a_j^{(i)} k(x, x_j)$$

Where: $\alpha^{(1)}, ..., \alpha^{(k)} \in R^n$, $\alpha^{(i)} = \frac{1}{\lambda_i} v_i$
$K = \sum_{i=1}^{n} \lambda_i v_i v_i^T$

!!! Here z_i = i-th element of the point z!!!

11.2 Using Neural Nets (Autoencoders)

In supervised learning we used neural nets in order to learn which feature maps to use. Here we can do the exact same.
The idea is to learn an identity function:

$$x \approx f(x;\theta) = \underbrace{f_2(\underbrace{f_1(x;\theta_1)}_{\text{encoder}};\theta_2)}_{\text{decoder}}$$

Because with that we can then use a supervised learning approach:

$$min_W \sum_{i=1}^{n} \|x_i - f(x_i;W)\|_2^2$$

You usually have a Neural Net, which has one hidden layer. The number of nodes in the hidden layer is (=k) and the number of input nodes (=d) as well as the number of output nodes is (=d).

12 Probabilistic Modeling

We always have function h: X $\rightarrow$ Y
Where X are the datapoints and Y the labels/numbers.

So in the past we had h(x) = normal regression/kernel regression/neural nets
But what if we try to find h(x) analytically. So what h(x) would minimize the prediction error (=risk)???

Remember that a prediction error (risk) is always defined over a loss-function (How we define our goodness of fit). From there we have different models. So if we choose a squared-distance as our loss function we have the least-squares Regression.
So given a loss-function we want to find the h(x) that minimizes the corresponding risk:

12.1 Least-squares Regression

In least-squares the Risk is:

$$R(x) = E_{X,Y}[(Y - h(x))^2]$$

Now which h(x) would minimize this risk?

We always assume:

Satz 12.1.
Our data is generated iid from an unknown distribution P(X,Y)

With this assumption and via the definition of the expected value one can show that: $h(x) = E[Y|X = x]$ Now we don't know P(X,Y) however we can make an educated guess.
We guess:

$$\text{y = f(x) + } \epsilon$$

Where $\epsilon \sim N(0, \sigma)$ and f(x)=$w^T x$.

With that we get that:

$$P(y|x, w, \sigma) = N(y; w^T x, \sigma)$$

We also assume that we know σ, so the only unknown would be w.
Now in order to find which w would explain the observed distribution the best we can use **Maximum Likelihood Estimation**.

Def. 12.1. Maximum Likelihood Estimation We have a probability distribution $P(x_1, ..., x_n; \theta)$ that is dependent on θ, where all x_i are iid from that distribution. Because all x_i are independent we can write:

$P(x_1, ..., x_n; \theta) = \prod_{i=1}^{n} P(x_i; \theta)$

With that we can write the Maximum likelihood as:

$\theta_{ML} = argmax_w \prod_{i=1}^{n} P_\theta(x_i)$

In our case we have $\theta = w$
So we would have the following Most-likelihood estimator:

$$w_{ML} = argmax_w P(y_1, ..., y_n|x_1, ..., x_n, w) = argmax_w N(y; w^T x, \sigma)$$

If we now calculate this maximum likelihood for w, we get:

$$w_{ML} = argmax_w - \sum_{i=1}^{n} (y_i - w^T x_i)^2 = argmin_w \sum_{i=1}^{n} (y_i - w^T x_i)^2$$

And this is exactly what we have as our Least squares regression.

Takeaway: If the unknown distribution function actually is equal to our educated guess, we found with ML the w that fits best with the observed datapoints (=has minimal Bias). Meaning that we found the most probable w

that could explain the datapoints assuming the distribution function of the datapoints is equal to our educated guess.
Moreover we showed that this w is captured by our Least squares regression.
So the least squared regression finds the w with minimal bias.

12.2 Ridge Regression

Def. 12.2. Prediction error = Bias^2 + Variance + Noise

Bias := Excess risk of best model considered compared to minimal achievable risk knowing P(X,Y) (i.e., given infinite data)
Variance := Risk incurred due to estimating model from limited data
Noise := Risk incurred by optimal model (i.e., irreducible error)

Remember that we want to minimize the Risk. So even if we found the w for the optimal low bias we could have that the variance is very high (due to overfitting for example).
So we need to find a Tradeoff between variance and Bias.

We ask ourself what function h(x) has minimal Variance.

So we need a way to capture this variance of the iid drawn datapoints from P(X,Y). Because for each draw of the datapoints we would get a slightly different w from our least-squared regression.

So we can again make an educated guess on how the weights would be distributed:

$$w \sim N(0, \beta)$$

Then we can then calculate which w is most likely a posteriori. (= Which w is most likely given the datapoints AND the corresponding lables, assuming that w is distributed as our educated guess).

Def. 12.3. Maximum a posteriori (=MAP)
$argmx_w P(w|x_1, ..., x_n, y_1, ..., y_n) = argmax_w \frac{P(w|x_1,...,x_n)P(y_1,...,y_n|x_1,...,x_n,w)}{P(y_1,...,y_n|x_1,...,x_n)}$

Now we can further simplify this by assuming that the w is independent of the $x_1, ..., x_n$
$\implies P(w|x_1, ..., x_n) = P(w)$.
And as we did in MLE we can again just take the log() and get:

$$w_{MAP} = argmax_w P(w|x_1, ..., x_n, y_1, ..., y_n)$$
$$= argmin_w \underbrace{-logP(w)}_{A} \underbrace{-logP(y_1, ..., y_n|x_1, ..., x_n, w)}_{B} \underbrace{+logP(y_1, ..., y_n|x_1, ..., x_n)}_{C}$$

Now we can calculate A and get:

$$A = const + \frac{1}{2\beta}\|w\|_2^2$$

And since C is independent of w we can just leave it out.

Now for B is exactly what we did before with MLE. So this would be:

$$B = argmin_w \sum_{i=1}^{n}(y_i - w^T x_i)^2$$

If we now put everything together we have:

$$argmin_w \frac{1}{2\beta}\|w\|_2^2 + \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - w^T x_i)^2$$

Which is the same as:

$$argmin_w \frac{\sigma^2}{\beta}\|w\|_2^2 + \sum_{i=1}^{n}(y_i - w^T x_i)^2$$

Which is exactly our ridge regression.

So if we assume that the noise is Gaussian distributed and the prior P(w) is also gaussian distributed we have the the MAP estimation is exactly our ridge regression.

So in order to achieve minimum bias and variance (under the assumption about the noise and the weights) we used MLE and MAP to show that the best possible solution (that minimizes the risk) would be the exact same as ridge regression.

However if we would assume that we have a different distribution of our prior P(w) for example a laplacian distribution we would get a different regularizer. In fact one can show that we would then get the l_1 normalizer.

Similarly one can use a different educated guess for our likelihood, and from that we get a different loss-function that would be optimal to minimize the bias.

12.3 Logistic Regression

So the last two examples were about how one can calculate a good regularizer and loss function (under given assumptions) for regression.
But what about classification?
So again we need a measure of goodness of fit in order to define our risk.
In classification one often defines this as:

$$R(h) = E_{X,Y}[Y \neq h(X)]$$

And we now want again to fine the function h(x) that minimizes this risk.

One can show that this is given by (**Bayes optimal predictor**):

$$h^*(x) = argmax_y P(Y = y|X = x)$$

However in practise we dont't know P(Y|X).
Therefore we try to make an educated guess.
Namely that y = h(x) + ϵ (as we did before). However this time we cant really say that the noise is gaussian distributed (this does not make sense in a classification task). Therefore we make the assumption that the noise is bernoulli distributed.
Now bernoulli is from [0,1] but our labels are -1,1. Therefore we apply a function $\sigma(z) = \frac{1}{1+expt(-z)}$ which maps the numbers to the following interval:

TODO: Insert slide 15 from logistic regression

So in summary we make the educated guess:

$$P(Y|X, w) = Ber(y; \frac{1}{1 + exp(-w^T x)})$$

Again everything is known except w. So we can again use MLE an get:

$w_{ML} = argmax_w - \sum_{i=1}^{n} log(1 + exp(-y_i w^T x_i) = argmin_w = \sum_{i=1}^{n} log(1 + exp(-y_i w^T x_i)$
So this is our newely figured out loss-function.

As before we can again assume that our w is gaussian (or laplacian) distributed. Hence we would get either the $\|w\|_2^2$ or $\|w\|_1$ as the normalizer.

So in total we would get (with l2-normalizer):

$$min_w \sum_{i=1}^{n} log(1 + exp(-y_i w^T x_i) + \lambda \|w\|_2^2$$

So we have now find a new classifier.
The prediction is done this way:

$$P(y|x, w) = \frac{1}{1 + exp(-y w^T x)}$$

12.3.1 Kernelized logistic regression

Of course we can kernelize this classifier:

$$\alpha^* = argmin_\alpha \sum_{i=1}^{n} log(1 + exp(-y_i \alpha_i^K)) + \lambda \alpha^T K \alpha$$

And the prediction is done this way:

$$P(y|x, \alpha) = \frac{1}{1 + exp(-y \sum_{j=1}^{n} \alpha_j k(x_j, x))}$$

12.3.2 Multi-class logistic regression

$P(Y = i|x, w_1, ..., w_c) = \frac{exp(w_i^T x)}{\sum_{j=1}^{c} exp(w_j^T x)}$

13 Decision Theory

In the case of classification we so far always had the same cost-function. If you misspredict you have a cost of 1 and otherwise 0.
Furthermore we always predicted one of the labels.

We now introduce on what to do if you have:

- Multiple Actions A (z.b. A ={Spam. No-spam, Ask-client})
- Different cost function C (z.b. C(y,a) = [y $\neq$ a]

So we want to find $\alpha \in A$ that minimizes the cost:

$$\hat{\alpha} = \operatorname*{argmin}_{\alpha \in A} \mathbb{E}_y[C(y,a)|x] \tag{1}$$

13.1 Doubtful logistic regression

$\hat{P}(y|x) = Ber(y; \sigma(w^T x))$
A = {+1,-1,D}
$C(y,a) = \begin{cases} [y \neq a] if a \in \{+1,-1\} \\ c \qquad\qquad\qquad if a = D \end{cases}$

Then the action that minimizes (1) is:

$$\alpha^* = \begin{cases} y & if \hat{P}(y|x) \geq 1 - c \\ D & \text{otherwise} \end{cases}$$

13.2 Least Squares regression

$\hat{P}(y|x) = N(y; w^T x, \sigma^2)$
A = $\mathbb{R}$
C(y,a) = $(y-a)^2$

Then the action that minimizes (1) is:

$$\alpha^* = \hat{w}^T x$$

13.3 Least Squares regression with asymmetric cost

$\hat{P}(y|x) = N(y; w^T x, \sigma^2)$
A = $\mathbb{R}$
C(y,a) = $c_1 \max(y-a, 0) + c_2 \max(a-y, 0)$

Then the action that minimizes (1) is:

$$\alpha^* = \hat{w}^T x + \sigma \cdot \phi^{-1}(\frac{c_1}{c_1 + c_2})$$

13.4 Active Learning

The idea is, that it is expensive to know the true labels of our training data points. Therefore we only want to ask for some of the datatpoints what the true label.

Given: $D_x = x_1, ..., x_n$ (= unlabeled datapoints)
For t=1,2,3,...

- Estimate $\hat{P}(Y_i|x_i)$ given current data D
- Pick unlabeled example that we are most uncertain about

$$i_t \in \operatorname*{argmin}_i |0.5 - \hat{P}(Y_i|x_i)|$$

- Query label y_{i_t} and set $D \leftarrow D\{(x_{i_t}, y_{i_t})\}$

14 Generative Modeling

So far we only tries to model (**Discriminative model**):

$$P(Y|X)$$

However we then never model:

$$P(X)$$

Because of that we can't detect outliers etc. Therefore we now want to model the joint distribution (**Generative model**):

$$P(Y,X)$$

Typical approach to generative modeling:

- Estimate prior on labels p(Y)
- Estimate conditional distribution $P(x|y)$ for each class y.
- Obtain predictive distribution using Bayes' rule:

$$P(y|x) = \underbrace{\frac{1}{P(x)}}_{:=Z} P(x,y) = \frac{1}{Z} P(y)P(x|y)$$

Now in order to model $P(x,y)$ we need to make some assumptions.

14.1 Naive Gaussian Bayes

We assume:

1.) Model class label as generated from categorical variable:

$$P(Y=y) = p_y \qquad \text{for all } y \in \{1,..,c\}$$

2. Model features are conditionally independent for a given Y:

$$P(X_1,...,X_d|Y) = \prod_{i=1}^{d} P(X_i|Y)$$

3.) Model features are Gaussian:

$$P(x_i|y) = N(x_i|\mu_{y,i}, \sigma_{y,i}^2)$$

In other words we assume: "Given a class label each feature (x_i) is generated independently of the other features. Moreover the a feature is gaussian distributed".

We now have $p_y, \mu_{y,i}$ and $\sigma_{y,i}$ as unknown variables. How do we estimate these parameters?
We use MLE!!!

$$\hat{p_y} = \frac{\text{Count(Y = y)}}{n}$$

$$\hat{\mu_{y,i}} = \frac{1}{\text{Count(Y = y)}} \sum_{j:y_j=y} x_{j,i}$$

$$\hat{\sigma_{y,i}^2} = \frac{1}{\text{Count(Y = y)}} \sum_{j:y_j=y} (x_{j,i} - \hat{\mu_{y,i}})^2$$

And then to make the prediction we in general want:

$$y = \underset{y'}{\text{argmax}}\, \hat{P}(y',x) = \underset{y'}{\text{argmax}}\, \hat{P}(y') \prod_{i=1}^{d} \hat{P}(x_i|y')$$

For the case of a binary classifier this is equivalent to:

$$y = \text{sign}(\underbrace{\log \frac{P(Y=1|x}{P(Y=-1|x}}_{f(x)})$$

<u>Note:</u> The function f(x) is called the **discriminant function**

So now for our specific case this would result in:

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

$wi = \frac{\mu_{+,i} - \mu_{-}}{\sigma_i^2}$, $w_0 = \log \frac{\hat{p}_+}{1-\hat{p}_+} + \sum_{i=1}^{d} \frac{\hat{mu}_{-,i}^2 - hat\mu_{+,i}^2}{2\hat{\sigma}_i^2}$ <u>Note:</u> If the model assumption are met, this makes the same predictions as logistic regression.

14.2 Gaussian Bayes

Here we drop the assumption that the features of the same class are all independent. However we still assume that a feature is gaussian.
Hence we assume:

1.) Model class label as generated from categorical variable:

$$P(Y = y) = p_y \qquad \text{for all } y \in \{1, .., c\}$$

2.) Model features are multivariant Gaussian:

$$P(x|y) = N(x|\mu_y, \Sigma_y^2)$$

<u>Note:</u> is the Covariance matrix. In Naive Bayes we just assumed that the Covariance matrix would be only a diagonal.

We now have the unknown variables p_y, μ_y and Σ_y which we again estimate with MLE:

$$\hat{p_y} = \frac{\text{Count(Y = y)}}{n}$$

$$\hat{\mu_y} = \frac{1}{\text{Count(Y = y)}} \sum_{i:y_i=y} x_i$$

$$\hat{\Sigma_y} = \frac{1}{\text{Count(Y = y)}} \sum_{i:y_i=y} (x_i - \hat{\mu_y})(x_i - \hat{y}_y)^T$$

And for the prediction we get:

$$y = sign(\log \frac{p}{1-p} + \frac{1}{2}\left[\log \frac{|\hat{\Sigma_-}|}{|\hat{\Sigma_+}|} + \Big((x - \hat{\mu_-})^T \hat{\Sigma_-}^{-1}(x - \hat{\mu_-})\Big) - \Big((x - \hat{\mu_+})^T \hat{\Sigma_+}^{-1}(x - \hat{\mu_+})\Big)\right]$$

Where $p = P(Y = 1)$

14.3 Fisher's linear discriminant analysis (LDA)

Here we use the Gaussian Bayes model but assume further that:

- c=2
- Covariances are equal:

$$\hat{\Sigma_-} = \hat{\Sigma_+} = \hat{}$$

Then the discriminant function simplifies and we get the following prediction:

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

$w = \hat{\Sigma}^{-1}(\hat{\mu}_+ - \hat{\mu}_-)$ and $w_0 = \frac{1}{2}(\hat{\mu}^T \Sigma^{-1} \hat{\mu}_- - \hat{\mu}_+^T \Sigma^{-1} \hat{\mu}_+)$

<u>Note:</u> If model assumption are met, LDA will make same predictions as Logistic Regression
<u>Note2:</u> LDA can be viewed as a projection to a 1-dim subspace that maximizes ratio of between-class and within-class variances
<u>Note3:</u> We can use LDA for outlier detection: P(x) < t.

14.4 Categorical Naive Bayes

Till now we always had $x_i \in R^d$ but what if we have discrete values?
Because then it doesn't really make sense to model x_i as a Gaussian.

The idea is that we have a categorical random variable for each feature.
Hence we assume in this model:

1.) Model class label as generated from categorical variable:

$$P(Y = y) = p_y \qquad \text{for all } y \in \{1, .., c\}$$

2.) Model features are independent categorical random variables:

$$P(X_i = x|Y = y) = \theta^{(i)}_{x|y}$$

We now have the unknown variables p_y and $\theta^{(i)}_{x|y}$ which we again estimate with MLE:

$$\hat{p_y} = \frac{\text{Count(Y = y)}}{n}$$
$$\theta^{(i)}_{x|y} = \frac{\text{Count}(X_i = c, Y = y)}{\text{Count}(Y = y)}$$

And for the prediction we get:

$$y = \underset{y'}{\text{argmax}}\, \hat{P}(y'|x) = \underset{y'}{\text{argmax}}\, \hat{P}(y') \prod_{i=1}^{d} \hat{P}(x_i|y')$$

Since we assumed independence. This can the be rewritten (with log-trick) to:

$$y = \underset{y'}{\text{argmax}} \log \hat{P}(y') + \sum_{i=1}^{d} \log \theta^{(i)}_{x|y'}$$

14.5 Overfitting

So far we always used MLE, however this is prone to overfitting. How can we avoid overfitting?

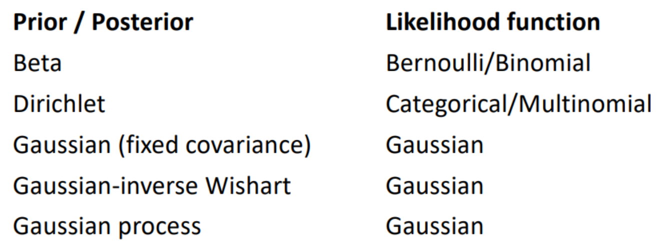
- Restrict model class (assumptions on covariance structure (e.g. Gaussian Naive Bayes)
- Using priors

14.6 Priors

The idea is to have a prior distribution for our parameter θ. That way we don't easily overfit.
Now if we choose the right prior for our likelihood function (how we model $p(\theta|D)$) we have a chance that the posterior distribution remains in the same family as the prior.

Def. 14.1. Conjugate distributions A pair of prior distributions and likelihood functions is called conjugate if the posterior distribution remains in the same family as the prior

And here is a list of which prior to choose to which likelihood function such that we have a conjugate distribution:



Prior / Posterior	**Likelihood function**
Beta	Bernoulli/Binomial
Dirichlet	Categorical/Multinomial
Gaussian (fixed covariance)	Gaussian
Gaussian-inverse Wishart	Gaussian
Gaussian process	Gaussian

But how can we choose the hyperparameters in the prior distribution?
$\Longrightarrow$ Cross validation

14.7 Gaussian Mixtures

!!! Let's assume that the labels are unobserved!!!

Meaning that we don't know the labels. Hence our task is: clustering!

Now what if our data distribution namely P(X|Y) is from multiple Gaussian's?
Then we want to model it as a mixture of multiple Gaussian's.

Reminder: If we would know the labels we could use Gaussian Bayes classifier, which works for $c \geq 2$, however we don't know the labels even while training

So we assume:

$$P(x|\theta) = P(x|\mu, \Sigma, w) = \sum_{i=1}^{c} w_i N(x; \mu_{i,i})$$

Where $w_i \geq 0$ and $\sum_i w_i = 1$
We would like to find:

$$(\mu^*, \Sigma^*, w^*) = \text{argmin} - \sum_i \log \sum_{i=1}^{k} w_j N(x_i|\mu_j, \Sigma_j)$$

However this is not convex and we therefore have to use a different algorithm:.
In a first approach we use the HARD-EM Algorithm:

Algorithm 14.1 (HARD-EM).

```
$\theta^t = \{w_{1..c}^{(t)}, \mu_{1..c}^{(t)}, \Sigma_{1..c}^{(t)}\}$
  1.) Initialize the parameters $\theta_0$ (See SOFT-EM)
  2.) For t=1,2,...
      – E-step: Predict most likely class for each data point:
        $z_i^{(t)} = \underset{z}{\text{argmax}}\, P(z|x_i, \theta^{(t-1)})$
        $= \underset{z}{\text{argmax}}\, \underbrace{P(z|\theta'(t-1)}_{w_z^{(t-1)}} \underbrace{P(x_i|z, \theta^{(t-1)})}_{N(x_i|\mu_z^{(t-1)}, \Sigma_z^{(t-1)}}$    (2)
      – We now got complete data!
        $D^{(t)} = \{(x_1, z_1^{(t)}), .., (x_n, z_n^{(t)}\}$
      – M-step: compute MLE as for the Gaussian Bayes classifier:
        $\theta^{(t)} = \underset{\theta}{\text{argmax}}\, P(D^{(t)}|\theta)$
```

In this algorithm we assign a label to a datapoint even if we are uncertain.

Therefore we want to to assign a probability of how sure we are that this datapoints is corresponds to which label:

$$\gamma_j(x) = P(Z = j|x, , \mu, w)$$

For that we use the following algorithm:

Algorithm 14.2 (SOFT-EM).

Initialization:

- weights: uniform distribution
- means: Randomized or like k-means++
- variances: Initialize as spherical according to empirical variance in the data

While not converged:

- E-step: Calculate cluster membership weights fore each datapoint (from $\mu^{(t-1)}, \Sigma^{(t-1)}, w^{(t-1)}$):

$$\gamma_j(x) = P(Z = j|x, \Sigma, \mu, w) = \frac{w_j P(x|\Sigma_j, \mu_j)}{\sum_l w_l P(x|\Sigma_l, \mu_l)}$$

- M-step: Update (closed from Maximum Likelihood solution):

$$w_j^{(t)} = \frac{1}{n} \sum_{i=1}^{n} \gamma_j^{(t)}(x_i)$$

$$\mu_j^{(t)} = \frac{\sum_{i=1}^{n} \gamma_j^{(t)}(x_i) x_i}{\sum_{i=1}^{n} \gamma_j^{(t)}(x_i)}$$

$$\Sigma_j^{(t)} = \frac{\sum_{i=1}^{n} \gamma_j^{(t)}(x_i)(x_i - \mu_j^{(t)}(x_i - \mu_j^{(t)})^T}{\sum_{i=1}^{n} \gamma_j^{(t)}(x_i)} + v^2 I$$

Note: We know: $P(z|\theta)$, $P(x|z, \theta)$ (We need this in the E-step)
Note2: The $v^2 I$ in the Sigma-update is to avoid degeneracy (overfitting). We choose v^2 with cross-validation.

Of course selecting the number of classes is again difficult. We normally just look for an "elbow".

Relation to k-Means
If we enforce uniform weights ($w_1, ..., w_c = \frac{1}{c}$ and spherical covariances ($\Sigma_{1..c} = I \cdot \sigma^2$), the HARD-EM algorithm results in k-means.

Discussion
Euses ziel isch clustering. Dh mer wend d clusterzugehörigkeit vo pünkt usefinde. (clusterparameters z ha und dass mer e probability distribution hend zu welem cluster en punkt ghört und nöd eifach es hard assignment, wär eifach no en Bonus. Aber wichtig wäri d clusterzugehörigkeit). Jetzt wie findemer d clusterzugehörigkeit use? Mir wend p(y| x). Das isch de posterior. Zum de usrechen benutzemer: p(y|x) = p(y)p(x|y)/p(x). Und das entspricht 2*1/3 vom Bild obe.dh jetzt: Zum p(y=z|x) uszrechne bruchemer p(y=z) und p(x|y=z) sowie p(x|y=1)...p(x|y=k). Und a dem punkt mümmer jetzt assumptions treffe wie die probability distributions usgehnd. Mir assumed jetzt dasss p(x|y=i) für i=1..k alles gaussians sind, und p(y) isch multinomial. Jetzt fähled eus aber d parameter für die funktione (pi, mu, sigma). Die müsstemer jetzt schätze. Zb. mit MLE. Usserdem gilt: $p(x) = sum_{i<=k} p(y = i)p(x|y = i) = sum_{i<=k} pi_i N(x; sigma_i, mu_i)$ Und das isch die formle vo ganz am afang. Durch eusi ahname über p(y=z) und p(x|y=z) chömmemer uf die formle vom afang. Und da mer nöd MLE uf das chönt mache (intractable) dümmer de EM algorithm benutze wo eus die parameter git. Und mit dene parameter chömmer denn p(y|x) usrechne. Devo chömmer de MAP neh wo eus d clusterzueteilig git. Als bonus hämmer au no eusi clusterinfromatione (mean, variance) sowie hettemer de volli posterior (also p(y|x)).