This document is a lecture summary that follows the script of the Introduction to Machine Learning lecture at ETH Zurich. The contribution to this is editing and refactoring as well as providing additional material for better understanding. This summary was created during the spring semester 2018. Most graphics are copy & pasted from the slides. If you don't want yours here, please contact me and I will remove them. Otherwise, this work is published as CC BY-NC-SA.

I do not guarantee correctness or completeness, nor is this document endorsed by the lecturers. Feel free to point out any erratas.

Introduction to Machine Learning

Yannick Merkli

1 Supervised Learning

Learn functions $f: X \to Y$ from labeled data (e.g. X: E-mail messages, Y: "spam" or "not spam").

Supervised Learning Pipeline: Train model on labeled training data, then predict, using the obtained model, on the unlabeled test data.

Representation: We want to represent data in standard manner -> use Vectors e.g. $[1,2,0,4,\dots,9] \in \mathbb{R}^d$

Model selection and validation

1.1 Regression

Goal: Predict real valued labels (possibly vectors)

1.1.1 Linear Regression

Goal: Learn real valued mapping $f: \mathbb{R}^d \to \mathbb{R}$

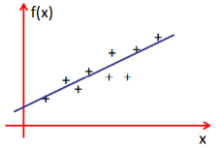
We need to somehow measure **goodness of fit**

$$\to y \approx f(x)$$

$$In\ 1-D: f(x) = ax + b$$

$$In\ 2-D: f(x) = ax_1 + bx_2 + c$$

$$In\ d-D: f(x) = \sum_{i=1}^{d} w_i x_i + b = w^T x + b$$



With $w = [w_1\ w_2 \dots w_d]$, $x = [x_1\ x_2 \dots x_d]$

Homogeneous Representation: $f(x) = w^T x + b = \tilde{w}^T \tilde{x}$

With $\tilde{w} = [w_1\ w_2 \dots w_d\ b] = [w\ b]$, $\tilde{x} = [x_1\ x_2 \dots x_d\ 1] = [x\ 1]$

Quantifying goodness of fit

$$D = \{(\boldsymbol{x_1}, y_1), \dots, (\boldsymbol{x_n}, y_n)\},\ \boldsymbol{x_i} \in \mathbb{R}^{\boldsymbol{d}},\ y_i \in \mathbb{R}$$

Take sum over square residuals:

$$\tilde{R}(\boldsymbol{w}) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - \boldsymbol{w}^T \boldsymbol{x_i})^2$$

The optimal weights minimize the summed squared errors:

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{argmin}\, \tilde{R}(\boldsymbol{w}) = \underset{\boldsymbol{w}}{argmin} \sum_{i=1}^{n} (y_i - \boldsymbol{w}^T \boldsymbol{x_i})^2$$

Two ways of solving this:

- Closed form solution:

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

  Where $X = \begin{pmatrix} -\boldsymbol{x_1}- \\ \dots \\ -\boldsymbol{x_n}- \end{pmatrix} \in \mathbb{R}^{nxd}$, $y = \begin{pmatrix} y_1 \\ \dots \\ y_n \end{pmatrix} \in \mathbb{R}^n$

  But: $X^T X$ might not always be invertible!
- Optimization:

  The objective function $\tilde{R}(\boldsymbol{w})$ is convex!

  ➔ We can optimize through **gradient**

1.1.2 Gradient descent

- Start at an arbitrary $w_0 \in \mathbb{R}^d$
- For t=1,2,… do $w_{t+1} = w_t - \eta_t \nabla_{w_t} \tilde{R}(\boldsymbol{w_t})$

Where η_t is called the learning rate.

Under mild assumptions, if the step size is sufficiently small, gradient descent converges to a stationary point (gradient = 0).

Computing the gradient:

$\tilde{R}(\boldsymbol{w}) = \sum_{i=1}^{n} (y_i - \boldsymbol{w}^T \boldsymbol{x_i})^2$

$$\nabla \tilde{R}(\boldsymbol{w}) = \left[\frac{\partial}{\partial w_1}\tilde{R}(\boldsymbol{w}), \dots, \frac{\partial}{\partial w_n}\tilde{R}(\boldsymbol{w})\right] \in \mathbb{R}^d$$

In 1-D:

$$\nabla \tilde{R}(w) = \frac{\partial}{\partial w}\tilde{R}(w) = \frac{\partial}{\partial w}\sum_{i=1}^{n}(y_i - wx_i)^2 = \sum_{i=1}^{n}\frac{\partial}{\partial w}(y_i - wx_i)^2 =$$

$$= \sum_{i=1}^{n} 2(y_i - wx_i)(-x_i) = -2\sum_{i=1}^{n}(y_i - wx_i)x_i$$

In d-D:

$$\nabla \tilde{R}(\boldsymbol{w}) = -2\sum_{i=1}^{n}\underbrace{(y_i - \boldsymbol{w^T x_i})}_{r_i}\boldsymbol{x_i}$$

➔ Gradient is basically a weighted combination of the data points weighted by the respective errors. If we make zero error ($r_i = 0\ \forall i$) we have zero gradient.

How do we choose the step size η_t? Poor step size can lead to divergence/ oscillations.

Adaptive step size:

- Via line search (optimizing step size every step):

  $$\eta_t = \underset{\eta \in \mathbb{R}^+}{argmin}\tilde{R}(\boldsymbol{w_t} - \boldsymbol{\eta g_t})$$

  $$\boldsymbol{g_t} = \nabla \tilde{R}(\boldsymbol{w_t})$$
- „Bold driver" heuristic:
  - If function decreases, increase step size:

    $$If\ \tilde{R}(\boldsymbol{w_{t+1}}) < \tilde{R}(\boldsymbol{w_t})\ then\ \eta_{t+1} = \eta_t * c_{inc}$$
  - If function increases, decrease step size:

    $$If\ \tilde{R}(\boldsymbol{w_{t+1}}) > \tilde{R}(\boldsymbol{w_t})\ then\ \eta_{t+1} = \eta_t * c_{dec}$$

Gradient descent for nonconvex functions is not guaranteed to converge to a global optimum, just a local optimum.

Gradient descent vs closed form

- Closed form solution:
  - : $X^T X \in \mathbb{R}^{dxd} \to O(d^2)$
  - Solving $(X^T X)^{-1} X^T y \to O(d^3)$
- Gradient descent:
  - One iteration costs $O(nd)$

    -> One iteration must compute residuals $r_i \to O(d)$ and sum up n vectors $\to O(n)$

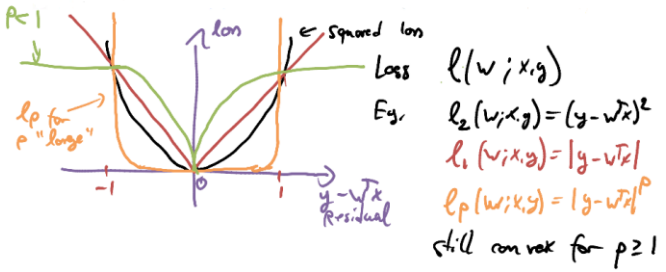
Things to consider:

- Computational complexity
- May not need an optimal solution
- Many problems don't admit closed form solution

1.1.3 Loss functions

So far: Measure goodness of fit via squared error.

Many other loss functions possible (and sensible!)



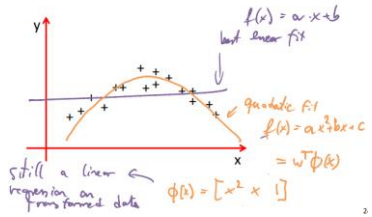
For large values of p, we are very sensitive to outliers, i.e. points with large error (for $p \to \infty$ we look at the largest error: $l_\infty = \max(y - w^T x)$)

For small values of p, we have less sensitivity (more resistivity) to outliers and we treat points close to the origin differently.

1.1.4 Fitting nonlinear functions

How about functions like this:

We can fit **nonlinear functions** via linear regression, using nonlinear features of our data (basis functions)



$$\boxed{f(x) = \sum_{i=1}^{d} w_i \phi_i(x)}\ ,with\ \phi(x) = [\phi_1(x), \phi_2(x), \dots, \phi_d(x)]$$

In 1-D: $\phi(\boldsymbol{x}) = [1, x, x^2, \dots, x^d]$

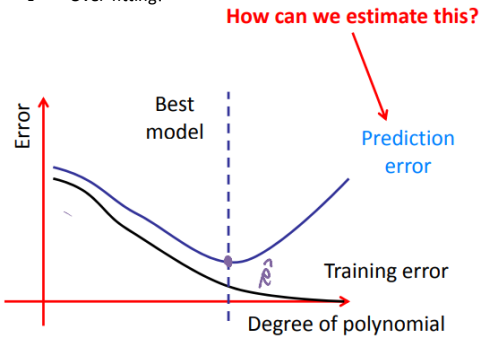
In 2-D: $\phi(\boldsymbol{x}) = [1, x_1, x_2, x_1^2, x_2^2, x_1 x_2 \dots]$

In k-D: $\phi(\boldsymbol{x}) = all\ monomials\ of\ degree\ at\ most\ k$

1.1.5 Model selection for linear regression with polynomials

Given a dataset, with a large enough polynomial, we can fit the data perfectly. However, we then rely too much on the data. If the data is noisy, we fit our model to the noise and not to the real data, which may lead to poor prediction capacity

➔ Over-fitting!



1.1.6 Regularization

Encourage small weights via penalty functions (regularizers).

1.1.6.1 Ridge regression

The regularized optimization problem is:

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

With λ as the regularization parameter.

Note: Now the scale of **x** matters! -> see 1.2.5 Standardization

We can solve this problem using gradient descent or still find an analytical solution:

- Analytical solution:

$$\boldsymbol{w}^* = \underbrace{(\boldsymbol{X^T X} + \lambda \boldsymbol{I})^{-1}}_{\substack{always\ invertible \\ even\ if\ n<d}} \boldsymbol{X^T y}$$

With $I = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix} \in \mathbb{R}^{dxd}$

- Gradient descent:

$$\nabla_w \left(\hat{R}(\boldsymbol{w}) + \lambda ||\boldsymbol{w}||_2^2 \right) = \nabla_w \hat{R}(\boldsymbol{w}) + \nabla_w \lambda ||\boldsymbol{w}||_2^2 =^* \nabla_w \hat{R}(\boldsymbol{w}) + 2\lambda w$$

$$* \Rightarrow \nabla_w \lambda ||\boldsymbol{w}||_2^2 = \lambda \nabla_w \boldsymbol{w^T w} = \lambda \nabla_w (w_1^2 + \cdots + w_d^2) = 2\lambda \boldsymbol{w}$$

Gradient descent formula then is:

$$\boldsymbol{w_{t+1}} = \boldsymbol{w_t} - \eta_t * \nabla_w \hat{R}(\boldsymbol{w}) - \eta_t * 2\lambda \boldsymbol{w_t} = \boldsymbol{w_t}(1 - 2\lambda\eta_t) - \eta_t \nabla_w \hat{R}(\boldsymbol{w})$$

-> Regularized risk minimization

Note: If we set λ very large, the minimization problem basically comes down to setting the first part as close to zero as possible -> set weights to zero (small weights) in order to counterpart the large λ in the $\lambda ||\boldsymbol{w}||_2^2$ term.

How to choose a regularization parameter? Cross-validation

1.1.6.2 Lasso Regression/ Sparse regression

Basically the same as Ridge regression, but replacing the $||\boldsymbol{w}||_2^2$ by $||\boldsymbol{w}||_1$.

The optimization problem then is:

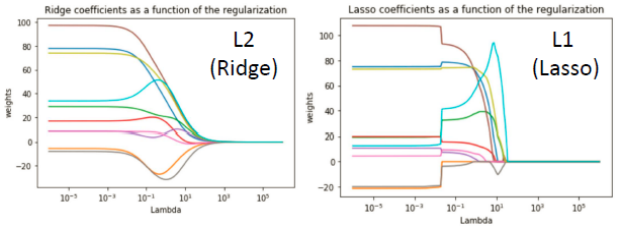
$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - \boldsymbol{w}^T \boldsymbol{x_i})^2 + \lambda ||\boldsymbol{w}||_1$$

This alternative penalty encourages coefficients to be **exactly 0** (not just close to zero as with ridge regression)**!** We force a hard decision whether to pick a coefficient or not.

➔ **Automatic feature selection**

How to choose a regularization parameter? Cross-validation

Illustration:



1.1.7 Renormalizing data: Standardization

Ensure that each feature has zero mean and unit variance:

$$\tilde{x}_{i,j} = (x_{i,j} - \hat{\mu}_j)/\hat{\sigma}_j$$

Hereby $x_{i,j}$ is the value of the j-th feature of the i-th data point and we estimate the mean and variance of feature j by:

$$\hat{\mu}_j = \frac{1}{n} \sum_{i=1}^{n} x_{i,j} \quad , \quad \hat{\sigma}_j = \frac{1}{n} \sum_{i=1}^{n} (x_{i,j} - \hat{\mu}_j)^2$$

➔ $\boldsymbol{X} = \begin{bmatrix} x_{11} & \cdots & x_{1d} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nd} \end{bmatrix}$

1.1.8 Fundamental tradeoff in ML

Need to trade loss (goodness of fit) and simplicity. A lot of supervised learning problems can be written in this way:

$$\min_{\boldsymbol{w}} \hat{R}(\boldsymbol{w}) + \lambda C(\boldsymbol{w})$$

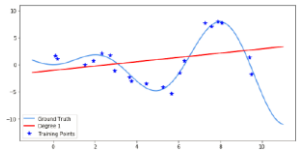
where $C(\boldsymbol{w})$ measures the complexity of the model.

Can control complexity by varying regularization parameter.

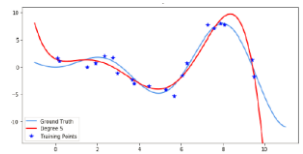
1.2 Model validation and selection

In Machine Learning, we basically want to fit functions. There are three general cases of fit:

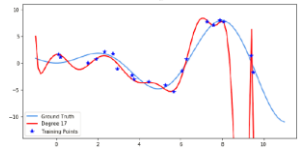
- Underfitting: Too simple model, model not complex enough
  - ➔ Increase model complexity



- Good fit: we fit the model to the **generating** function



- Overfitting: Too complex model, we fit the noise and not the generating function



We thus don't want to work with the training error (error on training data) **but** the prediction error.

We need to make a fundamental assumption:

Our data set is generated **independently and identically distributed (iid)**

$$(\boldsymbol{x_i}, y) \sim P(\boldsymbol{X}, Y) \quad \forall i$$

If our training data were drawn from a different distribution than the real data, the whole approach would break down.

Our goal is to minimize the expected error (**true risk**) under P:

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

➔ This is the expected square residual, over all possible data points (integrate over all x, y)

But: we can't calculate this in reality -> estimate it (**empirical risk**)

1.2.1 Empirical risk

Estimate the true risk by the empirical risk on a sample data set D:

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

ˆ because it's an estimator.

By the law of large numbers: $\hat{R}(\boldsymbol{w}) \to R(\boldsymbol{w})$ for any fixed $\boldsymbol{w}$ almost surely as $|D| \to \infty$

What happens if we optimize on training data?

Suppose we are given training data $D_{train} \sim P(X, Y)$

- Empirical Risk Minimization: $\hat{w}_{D_{train}} = \underset{\boldsymbol{w}}{argmin}\, \hat{R}(\boldsymbol{w})$
- Ideally, we wish to solve: $w = \underset{\boldsymbol{w}}{argmin}\, R(\boldsymbol{w})$

We generally want to move $\hat{w}$ as close to w as possible. However, we don't always have uniform convergence! It is possible that $\hat{w}$ moves away from w while the difference between $\hat{R}$ and R gets smaller.

What if we evaluate performance on training data?

$\hat{w} = \underset{\boldsymbol{w}}{argmin}\, \hat{R}(\boldsymbol{w}) \qquad w = \underset{\boldsymbol{w}}{argmin}\, R(\boldsymbol{w})$

In general, the following will hold:

$$E_D[\hat{R}_{train}(\hat{\boldsymbol{w}}_{\boldsymbol{train}})] \ll E[R(\hat{\boldsymbol{w}}_{\boldsymbol{D}})]$$

Thus we generally obtain an overly optimistic estimate! This is best seen for overfitting: for very large polynomial order, we can get 0 training error but huge prediction error -> we are overly optimistic.

How do we solve this?

-> Use separate test set from the same distribution P. Obtain training and test data D_{train} and D_{test}. Then:

- Optimize w on training set:

$$\hat{w} = \underset{\boldsymbol{w}}{argmin}\, \hat{R}_{train}(\boldsymbol{w})$$

- Evaluate on test set:

$$\hat{R}_{test}(\hat{\boldsymbol{w}}) = \frac{1}{|D_{test}|} \sum_{(\boldsymbol{x},y) \in D_{test}} (y - \hat{\boldsymbol{w}}^T \boldsymbol{x})^2$$

Then: $E[\hat{R}_{test}(\hat{\boldsymbol{w}})] = E[R(\hat{\boldsymbol{w}})]$

Since in this case, $\hat{R}_{test}(\hat{\boldsymbol{w}})$ is an unbiased estimator of the prediction error.

Proof: $D\ train\ set, V\ test\ set, D, V \sim P(X, Y)$

$$E_{D,V}[\hat{R}_V(\hat{\boldsymbol{w}}_{\boldsymbol{D}})] = E_D\left[E_V[\hat{R}_V(\hat{\boldsymbol{w}}_{\boldsymbol{D}})]\right] = E_D\left[E_V\left[\frac{1}{|V|} \sum_{(\boldsymbol{x},y) \in V} \left(y - \hat{\boldsymbol{w}}_{\boldsymbol{D}}{}^T \boldsymbol{x}\right)^2\right]\right]$$

$$= E_D\left[\frac{1}{|V|} \sum_{(\boldsymbol{x},y) \in V} E\left[\left(y - \hat{\boldsymbol{w}}_{\boldsymbol{D}}{}^T \boldsymbol{x}\right)^2\right]\right] = E_D\left[\frac{1}{|V|} \sum_{(\boldsymbol{x},y) \in V} R(\hat{\boldsymbol{w}}_D)\right] = E_D[R(\hat{\boldsymbol{w}}_D]$$

We can now evaluate different model complexities using the test error and therefore choose the best model. Problem: test set itself is random and thus fluctuates. We can thus overfit to the test set.
Solution: use multiple test sets to decrease the variance. However, data is expensive -> "reuse data"

1.2.2 Evaluation for model selection

For each **candidate model m** (e.g., polynomial degree repeat the following procedure for i = 1:k

- Split the same data set into training and validation set:
$$D = D_{train}^{(i)} \uplus D_{val}^{(i)} \quad (disjoint\ union)$$
- Train model:
$$\hat{w}_i = \underset{\boldsymbol{w}}{argmin}\, \hat{R}_{train}^{(i)}(\boldsymbol{w})$$
- Estimate error: $\hat{R}_m^{(i)} = \hat{R}_{val}^{(i)}(\hat{\boldsymbol{w}}_i)$

Finally: Select model
$$\hat{m} = \underset{\boldsymbol{m}}{argmin} \frac{1}{k} \sum_{i=1}^{k} \hat{R}_m^{(i)}$$
i.e. pick the model $\hat{m}$ with the smallest averaged validation error.

1.2.3 Cross validation

How should we do the splitting?

- Randomly (Monte Carlo cross-validation)
  - Pick training set of given size uniformly at random
  - Validate on remaining points
  - Estimate prediction error by averaging the validation error over multiple random trials
- k-fold cross-validation (-> default choice)
  - Partition the data into k „folds"
  - Train on (k-1) folds, evaluating on remaining fold
  - Estimate prediction error by averaging the validation error obtained while varying the validation fold

Cross-validation error estimate is very nearly unbiased for large enough k! How large should we pick k?

- Too small: Risk of overfitting to test set, using too little data for training, risk of underfitting to training set
- Too large: In general, better performance! k=n is totally fine (called leave-one-out cross validation LOOCV). But: computational complexity

In practice, k=5 or k=10 works well.

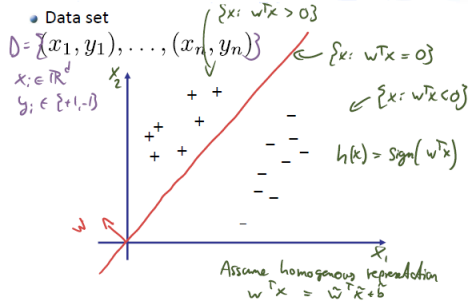
Note: For polynomial models, model complexity is easily controlled through the degree of the polynomial. In general, there may not be an ordering of the features that aligns with complexity. E.g. for nonlinear transformations, it is hard to classify complexity:
$$x \mapsto \phi(x) = [1, x, x^2, \sin(x), \cos(x) * x, \dots]$$

Note on overfitting: If we overfit a model, we get increasingly large weights. Having large weights is an indicator for overfitting -> idea: try to punish large weights.

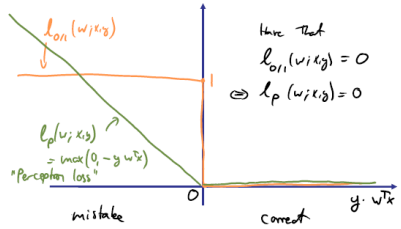
1.3 Linear Classification

Classification: Instance of supervised learning where Y is discrete (categorical). We want to assign data points X to a label Y.

1.3.1 Linear Classification



We want to somehow predict a label (for now binary) -> look at the sign of $\boldsymbol{w^T x_i}$ (we thus don't take the (linear) function as estimator but as a decision boundary). Then seek $\boldsymbol{w}$ that minimizes the number of mistakes:
$$\hat{R}(\boldsymbol{w}) = \sum_{i=1}^{n} [y_i \neq sign(\boldsymbol{w^T x_i})]$$
where $[y_i \neq sign(\boldsymbol{w^T x_i})] = l_{0/1}(\boldsymbol{w}; \boldsymbol{x_i}; \boldsymbol{y_i}) = \begin{cases} 1, y_i \neq sign(\boldsymbol{w^T x_i}) \\ 0, else \end{cases}$ is the 0/1-loss.
Problem: The optimization problem
$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{argmin} \sum_{i=1}^{n} [y_i \neq sign(\boldsymbol{w^T x_i})] = \sum_{i=1}^{n} l_{0/1}(\boldsymbol{w}; \boldsymbol{x_i}; \boldsymbol{y_i})$$
is now no longer nice to solve (NP-hard) since $l_{0/1}$ is not convex, not continuous and its gradient is all 0.
Solution: Use a surrogate loss function: the perceptron loss, which **is** convex!



1.3.2 Perceptron

The perceptron loss: $\boxed{l_p(\boldsymbol{w}; \boldsymbol{x}; y) = max(0, -y\boldsymbol{w^T x})}$

Perceptron optimization problem:

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{argmin}\, \hat{R}_p(\boldsymbol{w}) = \underset{\boldsymbol{w}}{argmin} \sum_{i=1}^{n} l_p(\boldsymbol{w}; \boldsymbol{x_i}; \boldsymbol{y_i}) = \\ = \underset{\boldsymbol{w}}{argmin} \sum_{i=1}^{n} max(0, -y_i \boldsymbol{w^T x_i})$$

Gradient descent: $\hat{R}_p(\boldsymbol{w}) = \sum_{i=1}^{n} max(0, -y_i \boldsymbol{w^T x_i})$
$$\nabla_w \hat{R}_p(\boldsymbol{w}) = \sum_{i=1}^{n} \nabla_w max(0, -y_i \boldsymbol{w^T x_i})$$
$$\rightarrow \nabla_w max(0, -y_i \boldsymbol{w^T x_i}) = \begin{cases} 0, if\ y_i \boldsymbol{w^T x_i} \geq 0 \\ -y_i \boldsymbol{x_i}, if\ y_i \boldsymbol{w^T x_i} < 0 \end{cases}$$
We get:
$$\nabla_w \hat{R}_p(\boldsymbol{w}) = - \sum_{i: y_i \neq sign(\boldsymbol{w^T x_i})} y_i \boldsymbol{x_i}$$
Thus, only wrongly classified points contribute to the gradient.

1.3.3 Perceptron Algorithm

This is just stochastic gradient descent (SGD) on the perceptron loss function:
$$\nabla_w l_p(\boldsymbol{w}; \boldsymbol{x}; y) = \begin{cases} 0, if\ y_i \boldsymbol{w^T x_i} \geq 0 \\ -y_i \boldsymbol{x_i}, if\ y_i \boldsymbol{w^T x_i} < 0 \end{cases}$$
Algorithm:
Initialize weights somehow (e.g. $\boldsymbol{w_1} = 0$)
For $t = 1,2, \dots$ do
Pick $i_t \sim Unif\{1, \dots, n\}$
If $y_{i_t} \neq sign(\boldsymbol{w_t^T x_{i_t}})$
$\boldsymbol{w_{t+1}} = \boldsymbol{w_t} + \eta_t y_{i_t} \boldsymbol{x_{i_t}}$
Else
$\boldsymbol{w_{t+1}} = \boldsymbol{w_t}$
Remarks: Whenever the algorithm encounters a wrongly classified point, the weights $\boldsymbol{w}$ are updated. This algorithm obviously needs much more iterations than normal gradient descent since we only look at one point per iteration.

1.3.3.1 Perceptron analysis

Theorem: If the data is linearly separable, the Perceptron will obtain a linear separator.
Proof: $l_p(\boldsymbol{w}; \boldsymbol{x}; y) = 0 \Leftrightarrow y\boldsymbol{w^T x} > 0 \Leftrightarrow y = sign(\boldsymbol{w^T x})$ i.e. no mistake
$$\Rightarrow \hat{R}_p(\boldsymbol{w}; \boldsymbol{x}, \boldsymbol{y}) = 0 \Leftrightarrow \boldsymbol{w}\ makes\ no\ mistakes$$
By assumption, $\exists \tilde{\boldsymbol{w}}$ that linsep. the data. $\Rightarrow \hat{R}_p(\tilde{\boldsymbol{w}}) = 0$
Since SGD on $\hat{R}_p$ converges to $\tilde{\boldsymbol{w}} \in \underset{\boldsymbol{w}}{argmin}\, \hat{R}_p(\boldsymbol{w})$, it must hold that $\tilde{\boldsymbol{w}}$ linearly separates the data.

1.3.4 Key concept: Surrogate losses

Replace intractable cost function that we care about (e.g., 0/1-loss) by tractable loss function (e.g. Perceptron loss) for sake of optimization/model Fitting. When evaluating a model (e.g., via cross-validation), use original cost/ performance function.

1.3.5 Support Vector Machines (SVMs)

The Perceptron doesn't care about margins. It only cares about whether we classify correctly or not. It doesn't care about how far a point is from the decision boundary. However, we want equidistant decision regions because this makes the classifier more robust towards noisy data.
Solution: Hinge loss

Support Vector Machine (SVM):

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{argmin} \sum_{i=1}^{n} \underbrace{max(0,1 - y_i \boldsymbol{w}^T \boldsymbol{x_i})}_{l_H} + \lambda ||\boldsymbol{w}||_2^2$$

SVMs are widely used, very effective linear classifiers.
SVMs are almost like Perceptron:

- Optimize slightly different, shifted loss (hinge loss)
- Regularize weights (like ridge regression)

Choosing the regularization parameter
We can pick the regularization parameter through cross-validation. Note that instead of using the hinge loss for validation, we use the target performance metric (e.g. # of mistakes: 0/1 loss)

We can also use L1 regularization for SVMs:

1.3.5.1 L1-SVM

By applying the sparsity trick: replacing $||\boldsymbol{w}||_2^2$ by $||\boldsymbol{w}||_1$, we get the L1-SVM. This SVM encourages weights to be either nonzero or exactly zero, i.e. ignore these features.

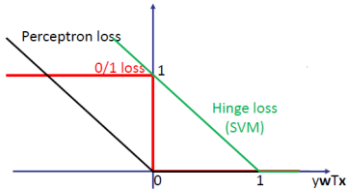
$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{argmin} \sum_{i=1}^{n} \underbrace{max(0,1 - y_i \boldsymbol{w}^T \boldsymbol{x_i})}_{l_H} + \lambda ||\boldsymbol{w}||_1$$

Solving L1 regularized problems:

- L1-norm is convex
- Combined with convex losses, obtain convex optimization problems (e.g., Lasso, l1-SVM, …)
- Can in principle solve using (stochastic) gradient descent
- However, convergence usually slow and will rarely get „exact 0“ entries

1.3.5.2 Hinge loss

We want a loss, that punishes small margins from the decision region. We can encourage this behavior by shifting the Perceptron loss into the region where $y\boldsymbol{w}^T\boldsymbol{x} > 0$ (i.e. correctly classified). We thus now get a loss (punishment) if we only classify correctly by a small margin. It's no longer enough to just barely get it right.
Note: The Hinge loss is an upper bound to the 0/1 loss -> it upper bounds the # of mistakes in our training data.



Hinge loss: $l_H(\boldsymbol{w}; \boldsymbol{x}, y) = max(0, 1 - y\boldsymbol{w}^T\boldsymbol{x})$
Note: The margin is here 1 (for normalization), but this doesn't really Matter, because we also have to care about the weights.

1.3.5.3 SGD for SVMs

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{argmin} \overbrace{\frac{1}{n}\sum_{i=1}^{n} \big(max(0,1 - y_i \boldsymbol{w}^T \boldsymbol{x_i})\big) + \lambda ||\boldsymbol{w}||_2^2}^{G(\boldsymbol{w})} =$$

$$= \underset{\boldsymbol{w}}{argmin} \frac{1}{n}\sum_{i=1}^{n} \underbrace{\big(max(0,1 - y_i \boldsymbol{w}^T \boldsymbol{x_i}) + \lambda ||\boldsymbol{w}||_2^2\big)}_{g_i(\boldsymbol{w})}$$

$$\nabla_{\boldsymbol{w}} G(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{w}} g_i(\boldsymbol{w})$$

$$\nabla_{\boldsymbol{w}} g_i(\boldsymbol{w}) = \underbrace{\nabla_{\boldsymbol{w}} max(0,1 - y_i \boldsymbol{w}^T \boldsymbol{x_i})}_{=\begin{cases} 0, if\ y_i \boldsymbol{w}^T \boldsymbol{x_i} \geq 1 \\ -y_i \boldsymbol{x_i}, else \end{cases}} + \underbrace{\nabla_{\boldsymbol{w}} \lambda ||\boldsymbol{w}||_2^2}_{=2\lambda \boldsymbol{w}}$$

The update rule then is

$$\boldsymbol{w_{t+1}} = (1 - 2\lambda\eta_t)\boldsymbol{w_t} + \frac{1}{n}\eta_t \sum_{i=1}^{n} y_i \boldsymbol{x_i} 1[y_i \boldsymbol{w_t} \boldsymbol{x_i} < 1]$$

1.4 Multiclass classification

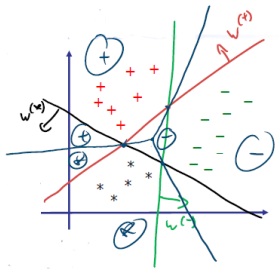
We now have not just two classes but multiple classes. In practice, this is often the case.
Problem setup:
$Given: \mathcal{D} = \{(\boldsymbol{x_1}, y_1), \dots, (\boldsymbol{x_n}, y_n)\}$, $y_i \in \mathcal{Y} = \{1, \dots, c\}$
$Want: f: \mathcal{X} \rightarrow \mathcal{Y}$

1.4.1 One-vs-all

Idea: We train c binary classifiers -> we classify one particular class i against all other data with any class except for i

- Solve c binary classifiers, one for each class
  - Positive examples: all points from class i
  - Negative examples: all other points
- Classify using the classifier with largest confidence

Each classifier i attaches a confidence $f^{(i)}(\boldsymbol{x}) = \boldsymbol{w^{(i)}}^T \boldsymbol{x}$
We then predict: $\hat{y} = \underset{i \in \{1,\dots,c\}}{\text{argmax}} \boldsymbol{w^{(i)}}^T \boldsymbol{x}$
____: decision bound for + against all other data
____: decision bound for * against all other data
____: decision bound for – against all other data
____: final decision bound (without normalized weights)

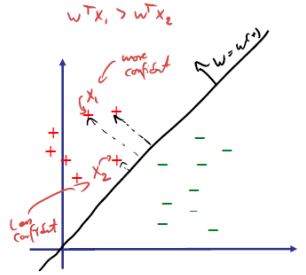


1.4.1.1 Confidence in classification

With this approach, we have normalization issues. $\boldsymbol{w}$ **and** $\alpha\boldsymbol{w}$ implement the same classifier but with very different confidence scores!
➔ $\boldsymbol{w}^T\boldsymbol{x}$ measures how far away we are from the decision boundary
One possible solution:
Normalize weights

$$\boldsymbol{w}^{(i)} = \frac{\boldsymbol{w}^{(i)}}{||\boldsymbol{w}^{(i)}||_2}$$

In fact, regularization (e.g. SVMs) take care of scale.



1.4.1.2 Challenges with one-vs-all

- Only works well if classifiers produce confidence scores on the „same scale“
- Individual binary classifiers see imbalanced data, even if the whole data set is balanced (e.g. 1000 classes, one single classifiers sees 1 vs 999)
- One class might not be linearly separable from all other classes -> here OVA would fail, but polynomial classification would solve this.

1.4.2 One-vs-one

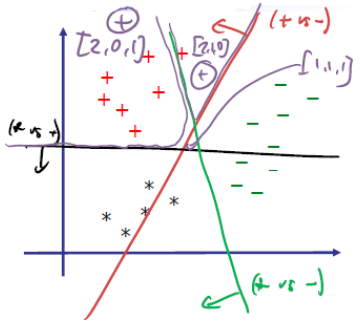
Idea: Train $\frac{c(c-1)}{2}$ binary classifiers, one for each pair of classes (i, j)

- Positive examples: all points from class i
- Negative examples: all points from class j

Then: Apply voting scheme

- Class with highest number of positive prediction wins

In the example (right) we use the following voting vector:
$[+, -, *]$
Vector $[2,0,1]$ means: 2 votes for + (red and green classifier), 0 votes for – and 1 vote for * (black classifier)



Method	One-vs-all	One-vs-one
Advantage	Only c classifiers needed (faster!)	No confidence needed
Disadvantage	Requires confidence in prediction / leads to class imbalance	Slower (need to train c*(c-1)/2 models)

1.4.3 Alternative methods

- Other encodings: e.g. error correcting output codes
- Explicit multi-class models:
  - E.g. multi-class perceptron/ SVM etc.
  - Some models are naturally multi-class (e.g., nearest neighbor, generative probabilistic models)

1.4.4 Class endcoding

How many binary classifiers do we need?
-> OVA: c -> OVO: c*(c-1)/2
We can get away with less! -> Use binary encoding of classes

Class	encoding
0	0…00
1	0…01
…	…
c	1…11

-> i bits -> i binary classifiers, task of classifier j is to predict the j-th bit.
For c classes, we need $i = \lceil \log_2 c \rceil$

1.4.4.1 Multi-class vs. coding

We can in principal view multi-classification as "decoding" the class label. Each classifier predicts one bit.
We might be able to get away with $O(\log c)$ classifiers! We can use ideas from coding theory to do multi-class classification and further improve prediction performance
➔ E.g. use Hamming codes to recognize errors!

1.4.5 Multi-class SVMs

Key idea: Maintain c weight vectors, one for each class

$$\boldsymbol{w}^{(1)}, \dots, \boldsymbol{w}^{(c)}$$

Then somehow train these weights simultaneously
Predict:

$$\hat{y} = \underset{i \in \{1,\dots,c\}}{\operatorname{argmax}} \boldsymbol{w}^{(i)T}\boldsymbol{x}$$

Given each data point $(\boldsymbol{x}, y)$ we want to achieve that

$$\underbrace{\boldsymbol{w}^{(y)T}\boldsymbol{x}}_{\substack{score\ for \\ class\ y}} > \underbrace{\max_{i \neq y} \boldsymbol{w}^{(i)T}\boldsymbol{x}}_{\substack{score\ for\ any \\ other\ class}} + \underbrace{1}_{margin}$$

i.e. score of the correct class should be higher than all other class scores including some margin.
Multi-class Hinge loss

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

ℓ_{MC-H} is 0 ⇔ condition (4) from previous slide is met

$$\nabla_{\mathbf{w}^{(i)}} \ell_{MC-H}(\mathbf{w}^{(1,\dots,c)}; x, y) = \begin{cases} 0 & \text{if } i \neq y \text{ or } i \notin \{y, \hat{y}\} \\ -x & \text{if } i = y \text{ and } i \neq \hat{y} \\ x & \text{if } i \neq y \text{ and } i = \hat{y} \end{cases}$$

Advantage of actual multi-class classification: We now actually train over all classes, not just over classes independently -> expect better results

Note: For multi-class problems, one often considers confusion matrices

Predicted label \ True label	Cat	Dog	Elefant
Cat	5	2	0
Dog	3	7	0
Elefant	1	0	6

1.5 Feature Selection

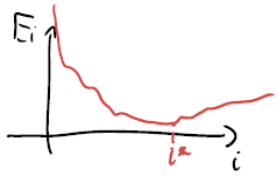
In many high-dimensional problems, we may prefer not to work with all potentially available features.
Why? Interpretability, Generalization, Storage / computation / cost
Naïve approach: Try all possible subsets of features and pick the best via cross-validation. However, this is very expensive.
Formulation:
Set of all features: $V = \{1, \dots, d\}$
Subset of features: $S \subset V, S = \{j_1, \dots, j_k\}$
Define cost function for scoring subsets S of V:
$\hat{L}(S)$ is cross-validation error using features in S only
$x \mapsto x_S = [x_{j_1}, \dots, x_{j_k}]$
$x_i \mapsto x_{i,S}$
$\hat{w}_S = \underset{\boldsymbol{w}}{\operatorname{argmin}} \left(\sum_{i=1}^{n} (y_i - \boldsymbol{w}^T \boldsymbol{x}_{i,S})^2 + \lambda ||\boldsymbol{w}||_2^2\right)$
$\hat{L}(S)$ = cross-validation score of $\hat{w}_S$

1.5.1 Greedy feature selection

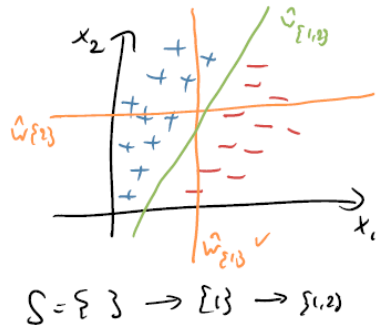
Approach: Greedily add (or remove) features to maximize. I.e. at each iteration find the best feature we haven't added yet and add it. Use cross-validation to predict accuracy. If at some point the error gets larger again, we break. This approach can be used for any method (not only linear regression/ classification)

1.5.1.1 Greedy forward selection

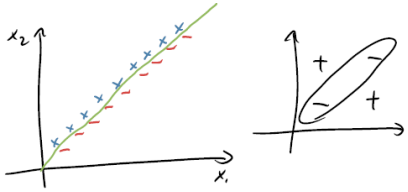
Algorithm:
Start with $S = \emptyset$ and $E_0 = \infty$
For i=1:d
- Find best element to add: $s_i = \underset{j \in V \setminus S}{\operatorname{argmin}} \hat{L}(S \cup \{j\})$
- Compute error: $E_i = \hat{L}(S \cup \{s_i\})$
- If $E_i > E_{i-1}$: break
- Else: set $S = S \cup \{s_i\}$



Example:
Each decision boundary stands for a different feature selection:
- $\hat{w}_{\{1\}}$ just selects the x_1 feature -> we basically project all points to the x_1 axis and find a boundary based on the projected points
- Same for $\hat{w}_{\{2\}}$
- For $\hat{w}_{\{1,2\}}$ we now have all features -> can select a good boundary



Problems with greedy forward selection:
- Greedy forward selection might get stuck in a local optimum
- Projecting points into a lower dimensional space might render our data useless:
  Projecting the data points to a single dimension leads to overlapping + and – points. From the perspective of a linear model, every **single feature** is completely uninformative and taken together, they are informative.



Advantage of FW selection:
Usually faster (if few relevant features)

1.5.1.2 Greedy backward selection

Instead of starting with nothing and selecting, we can also start with everything and start dropping features (backward selection).
Algorithm:
Start with $S = V$ and $E_{d+1} = \infty$
For i=d:-1:1
- Find best element to remove: $s_i = \underset{j \in S}{\operatorname{argmin}} \hat{L}(S \setminus \{j\})$
- Compute error: $E_i = \hat{L}(S \setminus \{s_i\})$
- If $E_i > E_{i-1}$: break
- Else: set $S = S \setminus \{s_i\}$

Advantage of BW selection:
Can handle "dependent" features

1.5.1.3 Problems with greedy feature selection

- Computational cost (need to retrain models many times for different feature combinations)
- Can be suboptimal (get stuck on local optimum)

Can we solve the learning & feature selection problem simultaneously via a single optimization?

➔ Discrete problem: feature selection
Continuous problem: learning (weights)

Can we somehow combine these? For linear models, we actually can!

➔ Selecting features is equivalent to setting non-selected features to zero. Thus, **feature selection is equivalent to finding sparse linear models**.

1.5.2 Joint feature selection and training

With feature selction so far, we just selected a subset of features:

$$\boldsymbol{x} = [x_1, \dots, x_d] \Longrightarrow \boldsymbol{x_S} = [x_{i_1}, \dots, x_{i_k}]$$

Then optimize over the coefficients: $\boldsymbol{w_S} = [w_{i_1}, \dots, w_{i_k}]$

$$\widehat{\boldsymbol{w}}_S = \underset{\boldsymbol{w_S}}{\operatorname{argmin}} \sum_{i=1}^{n} \left(y_i - \boldsymbol{w_S^T x_{i,S}}\right)^2$$

This is equivalent to constraining w to be sparse! (i.e. contain at most k non-zero entries)

➔ Find a linear regressor that sets weights to zero!

We want to solve the problem:

$$\widehat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - \boldsymbol{w^T x_i})^2 \ , s.t. \left|\left|\boldsymbol{w}\right|\right|_0 \leq k$$

Where $\left|\left|\boldsymbol{w}\right|\right|_0$ is the number of non-zeros in $\boldsymbol{w}$.

Alternatively, we can penalize the number of nonzero entries:

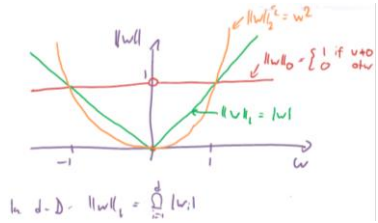
$$\widehat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - \boldsymbol{w^T x_i})^2 + \lambda \left|\left|\boldsymbol{w}\right|\right|_0$$

Problem: this is a difficult combinatorial optimization problem ($\left|\left|\boldsymbol{w}\right|\right|_0$ is not continuous).

Idea: Replace $\left|\left|\boldsymbol{w}\right|\right|_0$ by a more traceable term

➔ L1 as surrogate for L0

➔ **The sparsity trick:** $\left|\left|\boldsymbol{w}\right|\right|_0 \Rightarrow \left|\left|\boldsymbol{w}\right|\right|_1$



We can then use Lasso regression (see 1.1.6.2):

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - \boldsymbol{w}^T \boldsymbol{x_i})^2 + \lambda \left|\left|\boldsymbol{w}\right|\right|_1$$

Since we force coefficients to be either exactly 0 or nonzero with Lasso regression, this is basically feature selection (setting some weights to 0).

Solving L1 regularized problems:

- L1-norm is convex
- Combined with convex losses, obtain convex optimization problems (e.g., Lasso, l1-SVM, …)
- Can in principle solve using (stochastic) gradient descent
- However, convergence usually slow and will rarely get „exact 0“ entries

1.5.3 Comparison: Greedy selection vs. L1-Regularization

Method	Greedy (FW/BW)	L1-Regularization
Advantage	Applies to any prediction method	Faster (training and feature selection happen jointly)
Disadvantage	Slower (need to train many models)	Only works for linear models

1.6 Non-linear prediction with Kernels

General idea: Use linear models to fit nonlinear functions by considering nonlinear transformations into some potentially high dimensional space but then work in this high dimensional space only implicitly via the inner product in that space -> kernel.

In practice, we most often encounter non-linear problems. We have already seen non-linear feature transformations

$$\phi(\boldsymbol{x}) = [\phi_1(\boldsymbol{x}), \phi_2(\boldsymbol{x}), \dots, \phi_d(\boldsymbol{x})]$$

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

where we basically create a new feature set based on functions of the original features and then use linear regression/ classification.

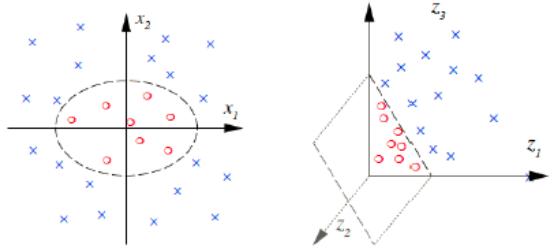
Example: polynomials (in 1-D)

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

Now assume we still have polynomials but higher dimensional data (e.g. 2-D). We then have to consider all possible monomials of the features. For 2-D this would be:

$$\phi: \mathbb{R}^2 \longrightarrow \mathbb{R}^3$$

$$(x_1, x_2) \longmapsto (z_1, z_2, z_3) = (x_1^2, \sqrt{2} x_1 x_2, x_2^2)$$



However, this is not feasible for high dimensions -> need $O(d^k)$ dimensions to represent (multivariate) polynomials of degree k on d features.

Example: $d = 10'000\,, k = 2 \rightarrow d^k \sim 100M\ dimensions$

How do we solve this? Kernels!

1.6.1 Fundamental insight

The optimal hyperplane lives in the span of the data -> optimal weight vector is a linear combination of the data points.

$$\widehat{\boldsymbol{w}} = \sum_{i=1}^{n} \alpha_i \, y_i \boldsymbol{x_i}$$

Note: y_i is there for classification, but this doesn't matter anyway since we can just absorb it into the coefficients α_i.

Proof: This can be shown through looking at (stochastic) gradient descent.

(S)GD creates such a representation. (S)GD starting from 0 constructs such a representation.

SVM: $\boldsymbol{w_{t+1}} = (1 - 2\lambda\eta_t)\boldsymbol{w_t} + \frac{1}{n}\eta_t \sum_{i=1}^{n} y_i \boldsymbol{x_i} 1[y_i \boldsymbol{w_t} \boldsymbol{x_i} < 1]$

Perceptron: $\boldsymbol{w_{t+1}} = \boldsymbol{w_t} + \eta_t y_{i_t} \boldsymbol{x_{i_t}}$

1.6.2 Kernels = efficient inner products

Often, $k(\boldsymbol{x_i}, \boldsymbol{x_j})$ can be computed much more efficiently than $\phi(\boldsymbol{x})^T \phi(\boldsymbol{x'})$

Simple example: Polynomial kernel in degree 2

$$\boldsymbol{x} = (x_1, x_2) \longmapsto \phi(\boldsymbol{x}) = (z_1, z_2, z_3) = (x_1^2, \sqrt{2} x_1 x_2, x_2^2)$$

$$\phi(\boldsymbol{x})^T \phi(\boldsymbol{x'}) = \underbrace{x_1^2 x_1'^2 + x_2^2 x_2'^2 + 2 x_1 x_2 x_1' x_2}_{10\ mult.\ 2\ add.} = \underbrace{(x_1 x_1' + x_2 x_2')^2}_{3\ mult.\ 1\ add.} = (\boldsymbol{x^T x'})^2$$

1.6.3 Example: Polynomial kernel (degree 2)

Suppose $\boldsymbol{x} = [x_1, \dots, x_d]^T$ and $\boldsymbol{x'} = [x_1', \dots, x_d']^T$

Then: $(\boldsymbol{x^T x'})^2 = \underbrace{\left(\sum_{i=1}^{d} x_i x_i'\right)^2}_{\substack{d+1\ mult.\ d\ add. \\ Total: 2d+1 \rightarrow O(d)}} = \phi(\boldsymbol{x})^T \phi(\boldsymbol{x'})$

$$\phi(\boldsymbol{x}) = \underbrace{[x_1^2, x_2^2, \dots, x_d^2, x_1 x_2, x_1 x_3, \dots, x_{d-1} x_d]}_{O(d^2)}$$

1.6.4 Kernelizing the Perceptron

Remember: Perceptron

Algorithm:

Initialize weights somehow (e.g. $\boldsymbol{w_1} = 0$)
For $t = 1,2, \dots$ do
Pick $i_t \sim Unif\{1, \dots, n\}$
If $y_{i_t} \neq sign(\boldsymbol{w_t^T x_{i_t}})$
$\boldsymbol{w_{t+1}} = \boldsymbol{w_t} + \eta_t y_{i_t} \boldsymbol{x_{i_t}}$
Else
$\boldsymbol{w_{t+1}} = \boldsymbol{w_t}$

Using the ansatz: $\widehat{\boldsymbol{w}} = \sum_{i=1}^{n} \alpha_i \, y_i \boldsymbol{x_i}$ we can rewrite the perceptron

Algorithm:

$\alpha = 0$
For $t = 1,2, \dots$ do
Pick $j \sim Unif\{1, \dots, n\}$
If $y_j \neq sign(\underbrace{(\sum_{i=1}^{n} \alpha_i \, y_i \boldsymbol{x_i})^T \boldsymbol{x_j}}_{(**)})$
$\underbrace{\alpha_j = a_j + \eta_t}_{(***)}$
Else
$\alpha_j = a_j$

(**)

$$\left(\sum_{i=1}^{n} \alpha_i \, y_i \boldsymbol{x_i}\right)^T \boldsymbol{x_j} = \sum_{i=1}^{n} \alpha_i \, y_i (\boldsymbol{x_i^T x_j})$$

(***)

$$\boldsymbol{w} = \sum_{i=1}^{n} \alpha_i \, y_i \boldsymbol{x_i} = \sum_{\substack{i=1 \\ i \neq j}}^{n} \alpha_i \, y_i \boldsymbol{x_i} + \alpha_j y_j \boldsymbol{x_j}$$

$$\Longrightarrow \boldsymbol{w} + \eta_t y_j \boldsymbol{x_j} = \sum_{\substack{i=1 \\ i \neq j}}^{n} \alpha_i \, y_i \boldsymbol{x_i} + \alpha_j y_j \boldsymbol{x_j} + \eta_t y_j \boldsymbol{x_j} =$$

$$= \sum_{\substack{i=1 \\ i \neq j}}^{n} \alpha_i \, y_i \boldsymbol{x_i} + \underbrace{(\alpha_j + \eta_t)}_{=: \alpha_j} y_j \boldsymbol{x_j}$$

We see:

- Now the algorithm works in terms of $\alpha's$. We don't work with weights $\boldsymbol{w}$ anymore
- $\boldsymbol{x_i}'\boldsymbol{s}$ only ever appear in pairs (i.e. as inner products) -> we can thus implicitly work in high-dimensional spaces, as long as we can do inner products efficiently.

Notation:

$$\boldsymbol{x} \mapsto \phi(\boldsymbol{x})$$
$$\boldsymbol{x}^T\boldsymbol{x}' \mapsto \phi(\boldsymbol{x})^T\phi(\boldsymbol{x}') =: k(\boldsymbol{x}, \boldsymbol{x}')$$

The perceptron optimization then is:

$$\hat{R}(\alpha) = \min_{\alpha_{1:n}} \sum_{i=1}^{n} max(0, -y_i \boldsymbol{w}^T \boldsymbol{x}_i) =$$
$$= \min_{\alpha_{1:n}} \sum_{i=1}^{n} max\left(0, -\sum_{j=1}^{n} \alpha_j y_i y_j \underbrace{\boldsymbol{x}_i^T \boldsymbol{x}_j}_{\substack{replace \\ by\ kernel}}\right) =$$
$$= \min_{\alpha_{1:n}} \sum_{i=1}^{n} max(0, -y_i \boldsymbol{\alpha}^T \boldsymbol{k}_i)$$

with $\boldsymbol{k_i} = [y_1 k(\boldsymbol{x_i}, \boldsymbol{x_1}), \dots, y_n k(\boldsymbol{x_i}, \boldsymbol{x_n})]$, $\boldsymbol{\alpha} = [\alpha_1, \dots, \alpha_n]^T$

The algorithm then is:

- Initialize $\alpha_1 = \cdots = \alpha_n = 0$
- For *t=1,2,...*
  - Pick data point $(\boldsymbol{x_i}, y_i)$ uniformly at random
  - Predict
  $$\hat{y} = \text{sign}\left(\sum_{j=1}^{n} \alpha_j y_j k(\mathbf{x}_j, \mathbf{x}_i)\right)$$
  - If $\hat{y} \neq y_i$ set $\alpha_i \leftarrow \alpha_i + \eta_t$

Prediction then works as:

- For new point x, predict
$$\hat{y} = \text{sign}\left(\sum_{j=1}^{n} \alpha_j y_j k(\mathbf{x}_j, \mathbf{x})\right)$$

Kernelized Perceptron with RBF (Gaussian) Kernel:

Everytime a new coefficient α_j is set (from 0), a new "Gaussian bump" is added to our predictor function $f(\boldsymbol{x}) = \sum_{j=1}^{n} \alpha_j y_j k(\boldsymbol{x_j}, \boldsymbol{x})$, thus the decision boundary changes.

1.6.5 Polynomial Kernels: Fixed degree

The kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T \boldsymbol{x}')^m$$

implicitly represents all monomials of degree m.

These are all products of *exactly* m input variables

$$\phi(\boldsymbol{x}) = [x_1^m, \dots, x_d^m, x_1^{m-1} x_2, \dots]$$

1.6.6 Polynomial Kernels

The polynomial kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = (1 + \boldsymbol{x}^T \boldsymbol{x}')^m$$

implicitly represents all monomials of up to degree m.

Representing the monomials (and computing inner product explicitly) is exponential in m!!

1.6.7 The Kernel Trick

- Express problems s.t. it only depends on inner products
- Replace inner products by kernels

$$\boldsymbol{x_i^T x_j} \Rightarrow k(\boldsymbol{x_i}, \boldsymbol{x_j})$$

This trick is very widely applicable!

Example: Kernelized Perceptron

$$\min_{\alpha_{1:n}} \sum_{i=1}^{n} max\left(0, -\sum_{j=1}^{n} \alpha_j y_i y_j \boldsymbol{x_i^T x_j}\right)$$
$$\Rightarrow \min_{\alpha_{1:n}} \sum_{i=1}^{n} max\left(0, -\sum_{j=1}^{n} \alpha_j y_i y_j k(\boldsymbol{x_i}, \boldsymbol{x_j})\right)$$

1.6.8 Properties of kernel functions

Given a dataspace X, a kernel function is a function: $k: X \times X \to \mathbb{R}$ where:

- k must be an inner product in a suitable space
- k must be symmetric

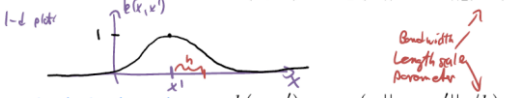
Definition: Kernel functions

- Data space X
- A **kernel** is a function $k: X \times X \to \mathbb{R}$ satisfzing:
  - **Symmetry**: For any $\boldsymbol{x}, \boldsymbol{x}' \in X$ it must hold that
  $$k(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x}', \boldsymbol{x})$$
  - **Positive semi-definiteness:** For any n, any set $S = \{\boldsymbol{x_1}, \dots, \boldsymbol{x_n}\} \subseteq X$, the kernel (Gram) matrix
  $$\boldsymbol{K} = \begin{pmatrix} k(\boldsymbol{x_1}, \boldsymbol{x_1}) & \cdots & k(\boldsymbol{x_1}, \boldsymbol{x_n}) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{x_n}, \boldsymbol{x_1}) & \cdots & k(\boldsymbol{x_n}, \boldsymbol{x_n}) \end{pmatrix}$$
  Must be positive semi-definite.

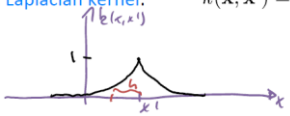
Using these properties, we can now prove whether a given function is a kernel or not.

1.6.9 Examples of kernels on $\mathbb{R}^d$

- Linear kernel: $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}'$
- Polynomial kernel: $k(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T \mathbf{x}' + 1)^d$
- Gaussian (RBF, squared exp. kernel): $k(\mathbf{x}, \mathbf{x}') = \exp(-||\mathbf{x} - \mathbf{x}'||_2^2 / h^2)$



- Laplacian kernel: $k(\mathbf{x}, \mathbf{x}') = \exp(-||\mathbf{x} - \mathbf{x}'||_1 / h)$



Remarks on the Gaussian kernel:

- Very widely used
- Feature map: this kernel maps the finite dim. Subspace to an infinite dim. space

1.6.10 Effect of kernel on function class

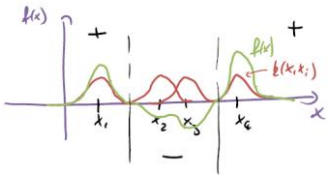
Given kernel k, predictors (for kernelized classification) have the form

$$\hat{y} = sign\left(\underbrace{\sum_{j=1}^{n} \alpha_j y_j k(\boldsymbol{x_j}, \boldsymbol{x})}_{f(\boldsymbol{x})}\right)$$

$f(\boldsymbol{x})$ is generally highly nonlinear -> in order for us to find nonlinear predictors, we want to turn $f(\boldsymbol{x})$ into a discrete decision by applying the sign.

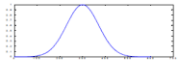
Example: Gaussian Kernel

Each kernel function $k(\boldsymbol{x}, \boldsymbol{x_i})$ is a "Gaussian bump". Each datapoint $\boldsymbol{x_i}$ has a gaussian bump around itself. The parameter h in the Gaussian kernel sets the width of the gaussian bump. $f(\boldsymbol{x})$ is then the sum of all these Gaussian bumps, each bump scaled by some factor.

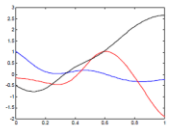


On bandwidth: Gaussian Kernel

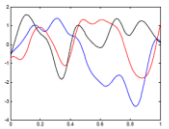
$$k(\mathbf{x}, \mathbf{x}') = \exp(-||\mathbf{x} - \mathbf{x}'||_2^2 / h^2)$$



$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x})$$



Bandwidth h=.3



Bandwidth h=.1

Note: We can define kernels on far more spaces than just $\mathbb{R}^d$!

1.6.11 Kernel engineering (composition rules)

- Suppose we have two kernels

$$k_1 : \mathcal{X} \times \mathcal{X} \to \mathbb{R} \qquad k_2 : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$$

defined on data space X

- Then the following functions are valid kernels:

$$k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}')$$
$$k(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}')\, k_2(\mathbf{x}, \mathbf{x}')$$
$$k(\mathbf{x}, \mathbf{x}') = c\, k_1(\mathbf{x}, \mathbf{x}') \text{ for } c > 0$$
$$k(\mathbf{x}, \mathbf{x}') = f(k_1(\mathbf{x}, \mathbf{x}'))$$

where f is a polynomial with positive coefficients or the exponential function

1.6.12 Example: ANOVA kernel

$$\boldsymbol{X} = \mathbb{R}^d, k_i: \mathbb{R} \times \mathbb{R} \to \mathbb{R}, \boldsymbol{x} = [x^{(1)}, \dots, x^{(d)}]$$

$k(\boldsymbol{x}, \boldsymbol{x}') = \underbrace{\sum_{i=1}^{d} k_i(x^{(i)}, x'^{(i)})}_{\text{each coordinate gets a different kernel}}$ -> rule (1) from 1.5.11: this is a valid kernel

Predictor: $f(\boldsymbol{x}) = \sum_{j=1}^{n} \alpha_j y_j k(\boldsymbol{x_j}, \boldsymbol{x}) = \sum_{j=1}^{n} \alpha_j y_j \sum_{i=1}^{d} k_i(x_j^{(i)}, x^{(i)}) =$

$$= \sum_{i=1}^{d} \left(\sum_{j=1}^{n} \alpha_j y_j k_i(x_j^{(i)}, x^{(i)}) \right) = \sum_{i=1}^{d} f_i(x^{(i)})$$

➔ If we work with this kernel, we work with functions $f(\boldsymbol{x})$ that decompose additively over the features.

1.6.13 Modeling pairwise data

We might want to use kernels to model pairwise data. We could for example just multiply two kernels together (gives a new valid kernel) where each single kernel looks at only one feature:
E.g.:

$$k([x^{(1)}, x^{(2)}], [x'^{(1)}, x'^{(2)}]) = k_1(x^{(1)}, x'^{(1)}) * k_2(x^{(2)}, x'^{(2)})$$

The Gaussian Kernel already has this feature kind of built-in: multiplying two Gaussian Kernels just gives us a new, scaled Gaussian Kernel.

1.6.14 Kernels as *similarity* functions (RBF)

When we predict for a new point $\tilde{\boldsymbol{x}}$, we look through all (training) data points and look at the contribution of each respective Gaussian bump.

- If our new point $\tilde{\boldsymbol{x}}$ is far away from a training point $\boldsymbol{x_i}$, the value of the RBF kernel will be $k(\boldsymbol{x_i}, \tilde{\boldsymbol{x}}) \approx 0$ (since $k(\boldsymbol{x_i}, \boldsymbol{x})$ is centered at $\boldsymbol{x_i}$ and decays fast).
  Thus, in this case the point $\boldsymbol{x_i}$ does not contribute anything to $\boldsymbol{f}(\tilde{\boldsymbol{x}})$.
- If however $\tilde{\boldsymbol{x}}$ is very close to $\boldsymbol{x}$, we have $k(\boldsymbol{x_i}, \tilde{\boldsymbol{x}}) \approx 1$.

We could thus rewrite the formula for $\hat{y}$ by just summing over points close to the to be predicted point $\tilde{\boldsymbol{x}}$:

$$\hat{y} = sign\left(\sum_{i=1}^{n} \alpha_i y_i k(\boldsymbol{x_i}, \boldsymbol{x}) \right) \approx sign\left(\sum_{\substack{i=1 \\ ||x_i - x|| \le 2h}}^{n} \alpha_i y_i \right)$$

1.6.15 K-NN vs Kernelized Perceptron

k-NN: Does not do any training, just looks at distance in between points. Prediction works as follows: For data point $\boldsymbol{x}$, predict majority label of k nearest neighbors:

$$y = sign\left(\sum_{i=1}^{n} y_i 1[\boldsymbol{x_i} \text{ among } k \text{ nearest neighbors of } \boldsymbol{x}] \right)$$

How to choose k? -> Cross-validation

k-NN vs Kernel Perceptron

- K-NN:

$$y = sign\left(\sum_{i=1}^{n} y_i 1[\boldsymbol{x_i} \text{ among } k \text{ nearest neighbors of } \boldsymbol{x}] \right)$$

- Kernel Perceptron:

$$y = sign\left(\sum_{i=1}^{n} \alpha_i y_i * \underbrace{k(\boldsymbol{x_i}, \boldsymbol{x})}_{\approx \begin{cases} 1 \text{ if } x_i \text{ close to } x \\ 0 \text{ if } x_i \text{ far from } x \end{cases}} \right)$$

➔ $k(\boldsymbol{x_i}, \boldsymbol{x})$ filters points $\boldsymbol{x_i}$ that are far from $\boldsymbol{x}$.

Method	k-NN	Kernelized Perceptron
Advantage	No training necessary	Optimized weights can lead to improved performance. Can capture „global trends“ with suitable kernels. Depends on „wrongly classified“ examples only (only points that were wrongly classified have $\alpha_i \neq 0$)
Disadvantage	Depends on all data -> inefficient k-NN is very simple -> might not be able be suitable for complex dataset	Training requires optimization

1.6.16 Parametric vs nonparametric learning

- Parametric models have finite set of parameters
  - Example: Linear regression, linear Perceptron, ...
- Nonparametric models grow in complexity with the size of the data
  - Potentially much more expressive
  - But also more computationally complex – Why?
  - Example: Kernelized Perceptron, k-NN, ...

Kernels provide a principled way of deriving nonparametric models from parametric ones.

1.6.17 Kernelized SVM

Remember: The Support Vector Machine

$$\boldsymbol{w}^* = \underset{w}{\operatorname{argmin}} \left(\sum_{i=1}^{n} \max(0, 1 - y_i \boldsymbol{w^T x_i}) + \lambda ||\boldsymbol{w}||_2^2 \right)$$

This can also be kernelized. (SVM often work much better than Perceptron)

How to kernelize SVM: see slides

Kernelized SVM:

$$\min_{\boldsymbol{\alpha_{1:n}}} \sum_{i=1}^{n} max(0, 1 - y_i \boldsymbol{\alpha^T k_i}) + \lambda \boldsymbol{\alpha^T D_y K D_y \alpha}$$

with $\boldsymbol{k_i} = [y_1 k(\boldsymbol{x_i}, \boldsymbol{x_1}), \dots, y_n k(\boldsymbol{x_i}, \boldsymbol{x_n})]$

Prediction:

$$y = sign\left(\sum_{i=1}^{n} \alpha_i y_i k(\boldsymbol{x_i}, \boldsymbol{x}) \right)$$

Note on regularization parameter:

- If we choose λ small, we want to predict all points as correctly as possible an don't care about margins so much
- If we choose λ large, we care more about margins than about predicting everything correctly.

Kernelized SVM with Gaussian Kernel

- Large bandwidth h: puts wide bumps around each datapoint -> we get a very smooth decision boundaries (almost linear)
- Small bandwidth h: puts narrow bumps around each datapoint -> we sum up all those bumps -> highly nonlinear/ non-smooth decision boundary
  - ➔ Danger if overfitting: due to the non-smooth decision boundary we may fit the data points (i.e. the noise) and not the real data

Kernelized SVM with Laplacian Kernel

- We get a piecewise linear decision boundary (straight line segments)

1.6.18 Kernelized Linear Regression

We can also kernelize linear regression! The predictor has the form

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x_i}, \boldsymbol{x})$$

Remember: The original (parametric) linear optimization problem is

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}} \left(\underbrace{\sum_{i=1}^{n} (y_i - \boldsymbol{w}^T \boldsymbol{x_i})^2}_{\hat{R}(w)} + \lambda ||\boldsymbol{w}||_2^2 \right)$$

As with the perceptron, the optimal $\boldsymbol{w}^*$ lies in the span of the data:

$$\boldsymbol{w}^* = \sum_{i=1}^{n} \alpha_i \boldsymbol{x_i}$$

Kernelizing:

- $\hat{R}(w) = \sum_{i=1}^{n} (y_i - \boldsymbol{w}^T \boldsymbol{x_i})^2 = \sum_{i=1}^{n} \left(y_i - \left(\sum_{j=1}^{n} \alpha_j \boldsymbol{x_j} \right)^T \boldsymbol{x_i} \right)^2 =$
  $= \sum_{i=1}^{n} \left(\sum_{i=1}^{n} \alpha_j \boldsymbol{x_j}^T \boldsymbol{x_i} - y_i \right)^2 \Rightarrow \sum_{i=1}^{n} \left(\sum_{i=1}^{n} \alpha_j k(\boldsymbol{x_j}, \boldsymbol{x_i}) - y_i \right)^2$
- $||\boldsymbol{w}||_2^2 = \boldsymbol{w^T w} = \left(\sum_{i=1}^{n} \alpha_i \boldsymbol{x_i} \right)^T \left(\sum_{j=1}^{n} \alpha_j \boldsymbol{x_j} \right) = \sum_{\boldsymbol{i,j=1}}^{\boldsymbol{n}} \alpha_i \alpha_j \left(\boldsymbol{x_i^T x_j} \right) =$
  $= \sum_{\boldsymbol{i,j=1}}^{\boldsymbol{n}} \alpha_i \alpha_j k(\boldsymbol{x_j}, \boldsymbol{x_i})$

Using the kernel matrix we can write this more compactly

$$\boldsymbol{K} = \begin{pmatrix} k(\boldsymbol{x_1}, \boldsymbol{x_1}) & \cdots & k(\boldsymbol{x_1}, \boldsymbol{x_n}) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{x_n}, \boldsymbol{x_1}) & \cdots & k(\boldsymbol{x_n}, \boldsymbol{x_n}) \end{pmatrix} = \begin{pmatrix} | & \dots & | \\ \boldsymbol{k_1} & \dots & \boldsymbol{k_n} \\ | & \dots & | \end{pmatrix}$$

- $\hat{R}(w) = \sum_{i=1}^{n} \left(\sum_{i=1}^{n} \alpha_j k(\boldsymbol{x_j}, \boldsymbol{x_i}) - y_i \right)^2 = \sum_{i=1}^{n} (\boldsymbol{\alpha^T k_i} - y_i)^2 =$
  $= ||\boldsymbol{\alpha^T K} - \boldsymbol{y^T}||_2^2$
- $||\boldsymbol{w}||_2^2 = \boldsymbol{\alpha^T K \alpha}$

With $\boldsymbol{\alpha} = [\alpha_1, \dots, \alpha_n]^T, \boldsymbol{y} = [y_1, \dots, y_n]^T$

We get:

Learning & Predicting with KLR

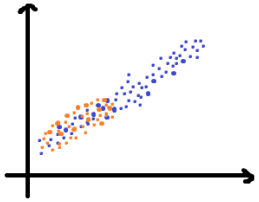
$$\min_{\boldsymbol{\alpha}} ||\boldsymbol{\alpha^T K} - \boldsymbol{y^T}||_2^2 + \lambda \boldsymbol{\alpha^T K \alpha}$$

➔ **Closed form solution:** $\boldsymbol{\alpha}^* = (\boldsymbol{K} + \lambda \boldsymbol{I})^{-1} \boldsymbol{y}$

Prediction:

$$\hat{y} = f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x_i}, \boldsymbol{x})$$

Example: KLR with different Kernels
We train on the following data:
Train data , Test data



- Linear Kernel: This is just linear regression -> we put a line through the orange data points -> we capture the correct linear trend
  Sanity check: $k(\boldsymbol{x_i}, \boldsymbol{x}) = \boldsymbol{x_i^T x}$

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x_i}, \boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i \boldsymbol{x_i^T x} = \left(\sum_{i=1}^{n} \alpha_i \boldsymbol{x_i}\right)^T \boldsymbol{x} = \boldsymbol{w^T x}$$

- Polynomial Kernel: The higher the degree of the linear kernel, the worse the fit (we over fit) -> we try to fit some complicated curve although the data generating function is just a linear function
- Gaussian Kernel: Whatever is outside the training data (i.e. where the gaussian bumps have eventually decayed to zero)
  (Remember: function $f(x)$ is a sum of gaussian bumps. Each gaussian bump lies around a training data point -> where no training data, there no gaussian bumps since they decay quite fast)
  - If we have a larger bandwidth, we get slower decay and smooth function. The curve starts to resemble the polynomial kernel
  - If we have a small bandwidth, we get kind of delta impulses around each training data point and zero in between and outside of the training data.
  - Large regularization parameter: we push the α's down to zero and thus get a more flat curve
- Laplacian kernel: we get a very "spicky" curve with similar effects as for the Gaussian kernel. However for large bandwidth, the Laplacian Kernel still tries to interpolate the training data since less smooth.

We see, every kernel has its drawbacks:
- Gaussian: very localized, might fail to catch global trends
- Parametric model (e.g.linear or polynomial) very global model, very rigid -> might impose too rigid and strong assumptions on data

-> Why not combine them?!

1.6.18.1 Semi-parametric KLR

Often, parametric models are too „rigid", and nonparametric models fail to extrapolate. Solution: Use additive combination of linear and nonlinear kernel function:

$$k(\boldsymbol{x}, \boldsymbol{x'}) = c_1 \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{x'}||_2^2}{h^2}\right) + c_2 \boldsymbol{x^T x'}$$

According to kernel engineering, this is a valid kernel (sum of two kernels). Then:

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x_i}, \boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i \left(c_1 \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{x_i}||_2^2}{h^2}\right) + c_2 \boldsymbol{x^T x_i}\right) =$$

$$= c_1 \sum_{i=1}^{n} \alpha_i \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{x_i}||_2^2}{h^2}\right) + c_2 \left(\sum_{i=1}^{n} \alpha_i \boldsymbol{x_i}\right)^T \boldsymbol{x}$$

If we use this model, we get a linear model plus a bunch of gaussian bumps.

Periodic Kernel:

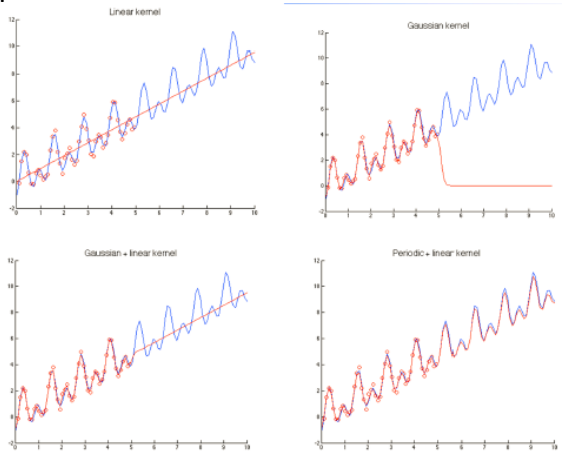
$$k(x, x') = \exp\left(-\frac{||sin(x\gamma) - sin(x'\gamma)||_2^2}{h^2}\right)$$

Why is this a valid kernel? We just apply some feature transformation to our data and then plug the transformed data into the normal Gaussian Kernel.

$$f(x) = \sum_{i=1}^{n} \alpha_i k(x_i, x) = \sum_{i=1}^{n} \alpha_i \exp\left(-\frac{||sin(x\gamma) - sin(x_i\gamma)||_2^2}{h^2}\right)$$

Periodic because $f(x) = f(x + \frac{2\pi}{\gamma})$

Example:



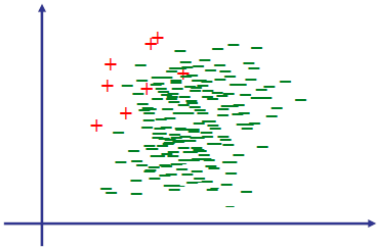
How do we pick kernels/ parameters?
- Every kernel has its own parameters -> use cross-validation to choose good parameters
- How to find the right kernels -> might have to try out some different ones. Use domain knowledge and cross validation. In some cases: brute force

1.6.19 What about overfitting?

Kernels map to very high dimensional feature spaces. Why do we hope to be able to learn?
- Typically, we have $\#parameters \ll \#of\ dimensions$
  -> we don't actually work in these high dimensional feature spaces since we have $\#parameters = \#of\ training\ data\ points$
- However, overfitting can still happen (if we choose poor parameters)
- We can combat overfitting by regularization
  - This is already built into kernelized linear regression and kernelized SVMs, but not the kernelized perceptron

2 Class Imbalance

Problem setup:
We are dealing with a classification problem where one class is much more prominent than the other one. Sources for these kinds of data could be: fraud detection, spam filtering, medical diagnosis, etc.



Issues with imbalanced data:
- Accuracy is not a good metric: May prefer certain mistakes over others (trade false positives and false negatives)
- Minority class instances contribute little to the empirical risk -> may be ignored during optimization!

$$\hat{R}(w) = \underbrace{\sum_{i:\, y_i = +} \ell(w; x_i, y_i)}_{\hat{R}_+(w)} + \underbrace{\sum_{i:\, y_i = -} \ell(w; x_i, y_i)}_{\hat{R}_-(w)}$$

$\hat{R}_+(w) \ll \hat{R}_-(w)$

- If we use SGD, we might never or very rarely pick a point from the minority class as a data point for the algorithm.

2.1 Solutions

2.1.1 Subsampling/ Downsampling

Remove training examples from the majority class (e.g., uniformly at random) such that the resulting data set is balanced.

2.1.2 Upsampling

Repeat data points from minority class (possibly with small random perturbation) to obtain balanced data set.

Method	Subsampling/ Downsampling	Upsampling
Advantage	Smaller data set -> faster	Makes use of all data
Disadvantage	Available data is wasted. May lose information about the majority class	Slower (data set up to twice as large). Adding perturbations requires arbitrary choices.

2.1.3 Cost-sensitive classification

Modify Perceptron / SVM to take class balance into account:
Only difference: Use cost-sensitive loss function. Replace loss by

$$l_{CS}(\boldsymbol{w}; \boldsymbol{x}, y) = \underbrace{c_y}_{\geq 0}\, l(\boldsymbol{w}; \boldsymbol{x}, y)$$

Where c_y depends on the class y. This can be applied to all seen loss functions.
Example:
- Perceptron: $l_{CS-P}(\boldsymbol{w}; \boldsymbol{x}, y) = c_y \max(0, -y\boldsymbol{w^T x})$
- SVM: $l_{CS-H}(\boldsymbol{w}; \boldsymbol{x}, y) = c_y \max(0, 1 - y\boldsymbol{w^T x})$

With parameters c_+, c_- (for binary classification) controlling tradeoff.

However, for SGD, we still have the problem that we only rarely pick the minority class. Solution: use non-uniform sampling.

The empirical risk now is:

$$\hat{R}(\boldsymbol{w}; c_+, c_-) = \sum_{i:y_i=+} c_+ l(\boldsymbol{w}; \boldsymbol{x}, y) + \sum_{i:y_i=-} c_- l(\boldsymbol{w}; \boldsymbol{x}, y)$$

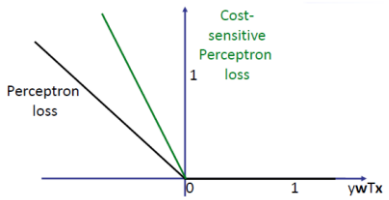
w.l.o.g., we can fix $c_- = 1$ -> we don't really need two parameters.

Observe: $\hat{R}(\boldsymbol{w}; c_+, c_-) = \frac{1}{\alpha}\hat{R}(\boldsymbol{w}; \alpha c_+, \alpha c_-) \underbrace{=}_{\alpha = c_-} c_- \hat{R}\left(\boldsymbol{w}; \frac{c_+}{c_-}, 1\right)$

➔ We now have a single tuning parameter $\frac{c_+}{c_-}$

2.1.3.1 Cost-sensitive Perceptron loss

c_y basically just changes the slope of the line for the different classes.
The perceptron algorithm stays ecactly the same, we just use a different cost function.



$$\ell(\mathbf{w}; \mathbf{x}, y) = c_y \max(0, -y\mathbf{w}^T\mathbf{x})$$

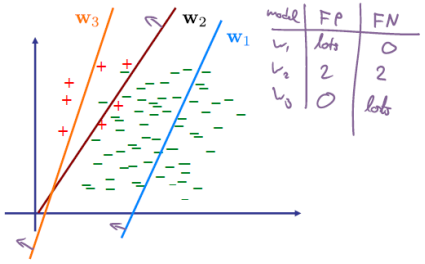
2.1.4 Accuracy for imbalanced data

convention: + is rare class

		True label: Positive	True label: Negative
Predicted label	Positive	TP	FP
	Negative	FN	TN

$\Sigma = p_+$ (row Positive), $\Sigma = p_-$ (row Negative), $\Sigma = n_+$ (col Positive), $\Sigma = n_-$ (col Negative), $n = n_+ + n_- = p_+ + p_-$

p_+: all data we **predict** to be positive
p_-: all data we **predict** to be negative
n_+: total # of positive data
n_-: total # of negative data



model	FP	FN
w_1	lots	0
w_2	2	2
w_3	0	lots

2.1.5 Metrics for imbalanced data

- Accuracy:
$$\frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{n} = \frac{\#correctly\ classified\ data}{\#all\ data}$$
- Precision:
$$\frac{TP}{TP + FP} = \frac{TP}{p_+} \in [0,1]$$
Among everything we predict positive, what is the fraction we predict correctly as positive.
- Recall:
$$\frac{TP}{TP + FN} = \frac{TP}{n_+} \in [0,1]$$
Among all positive data, how many do we predict correctly (how many do we recall).

Ideally, we'd want precision and recall $= 1$ (means we make no error at all).
However, mostly there's a tradeoff.

- F1 score:
$$\frac{2TP}{2TP + FP + FN} = \frac{2}{\frac{TP + FP}{TP} + \frac{TP + FN}{TP}} = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$

If we have a good F+ score, both Precision and Recall are good.

How do we obtain a tradeoff?

- Use cost-sensitive classifier and vary tradeoff parameter $\frac{c_+}{c_-}$
- Find a single classifier and vary the classification threshold τ
➔ $y = sign(\boldsymbol{w}^T y - \tau)$

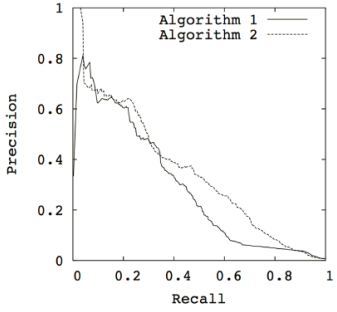
- True positive rate (TPR) = Recall!
$$\frac{TP}{TP + FN} = \frac{TP}{n_+} \in [0,1]$$
- False positive rate (FPR)
$$\frac{FP}{TN + FP} = \frac{FP}{n_-} \in [0,1]$$

Several other metrics are used. E.g. expected TPR and FPR:

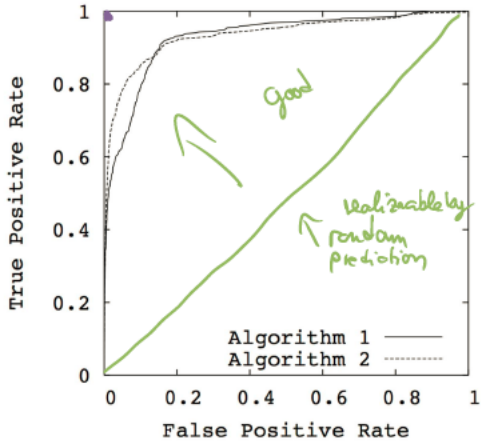
$$E[TPR] = \frac{pn_+}{n_+} = p\,, E[FPR] = \frac{pn_-}{n_-} = p$$

2.1.6 Precision recall curve

Plots the precision and recall while varying a parameter (e.g. tradeoff parameter $\frac{c_+}{c_-}$ (alg. 1) or the classification threshold τ)
Using this curve, we can now optimize, i.e. fix some amount of precision and find the best possible recall.

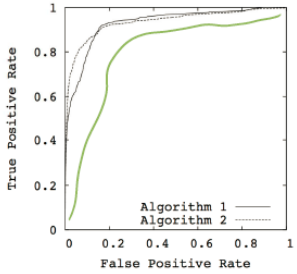


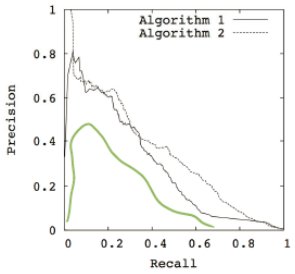
2.1.7 Receiver Operator Characteristic (ROC) Curve



In green: performance of a random classifier

2.1.8 Comparison of the curves

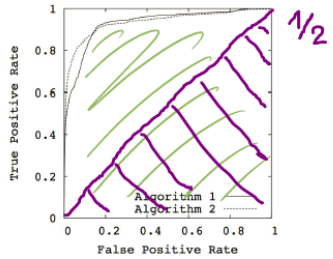




Theorem [Davis & Goadrich '06]: Alg 1 dominates Alg 2 in terms of ROC Curve ⇔ Alg 1 dominates Alg 2 in terms of Precision Recall curves

2.1.9 Area under the curve

Often want to compare the ability of classifiers to provide imbalanced classification. Can compute Area under the ROC or Precision Recall curves.
In green: dynamic range of this area under the curve.
A random classifier has area under curve $= 1/2$



3 Unsupervised Learning

- Unsupervised Learning: we don't have any labels (a y) in the training data
- Typically useful for exploratory data analysis ("find patterns", visualization,…)
- Most common methods:
  - Clustering (unsupervised classification)
  - Dimension reduction (unsupervised regression)

3.1 Clustering

What is clustering?

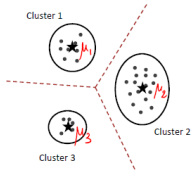
- Given data points, group into clusters such that
  - Similar points are in the same cluster
  - Dissimilar points are in different clusters
- Points are typically represented either
  - in (high-dimensional) Euclidean space
  - with distances specified by a metric or kernel
- Related: Anomaly / outlier detection – Identification of points that "don't fit well in any of the clusters"

Standard approaches to clustering:

- Hierarchical clustering
  - Build a tree (bottom-up or top-down), representing distances among data points
  - Example: single-, average- linkage clustering
- Partitional approaches
  - Define and optimize a notion of "cost" defined over partitions
  - Example: Spectral clustering, graph-cut based approaches
- Model-based approaches
  - Maintain cluster "models" and infer cluster membership (e.g., assign each point to closest center)
  - Example: k-means, Gaussian mixture models, …

3.1.1 K-Means clustering

- Represent each cluster by single point (center), k = # of clusters
- Assign points to closest center
- Induces Voronoi partition



3.1.1.1 The k-means problem

- Assume points are in Euclidian space: $\boldsymbol{x_i} \in \mathbb{R}^d$
- Represent clusters as centers: $\boldsymbol{\mu_j} \in \mathbb{R}^d$
- Each point is assigned to closest center

Goal: Pick centers to minimize average squared distance

$$\hat{R}(\mu) = \hat{R}(\mu_1, \dots, \mu_k) = \sum_{i=1}^{n} \min_{j \in \{1,\dots,k\}} ||\mathbf{x}_i - \mu_j||_2^2$$

$$\hat{\mu} = \arg\min_{\mu} \hat{R}(\mu)$$

$d(x_i, \mu)$ — loss function for k-means

Problem:

- Non-convex optimization
- NP-hard -> can't solve optimally in gensrl

3.1.1.2 K-means algorithm (Llyod's heuristic)

- Initialize cluster centers: $\mu^{(0)} = [\mu_1^{(0)}, \dots, \mu_k^{(0)}]$
  (Note: since the problem is non-convex, initialization matters)
- While not converged (t iterator):
  - Assign each point $\boldsymbol{x_i}$ to closest center:

$$z_i = \underset{j \in \{1,\dots,k\}}{\operatorname{argmin}} \left|\left|\boldsymbol{x_i} - \boldsymbol{\mu_j^{(t-1)}}\right|\right|_2^2$$

    (This is the Voronoi partition)
  - Update center as mean of assigned data points

$$\boldsymbol{\mu_j^t} = \frac{1}{n_j} \sum_{i:z_i=j} \boldsymbol{x_i}$$

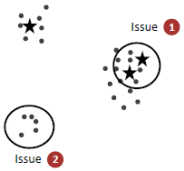
3.1.1.3 Properties of k-means

- Guaranteed to monotonically decrease average squared distance in each iteration (eventhough it's **not** a gradient based algorithm)

$$\hat{R}(\mu^{(t)}) = \sum_{i=1}^{n} \min_{j \in \{1,\dots,k\}} \left|\left|\boldsymbol{x_i} - \boldsymbol{\mu_j^{(t)}}\right|\right|_2^2$$

$$\forall t: \hat{R}(\mu^{(t)}) \geq \hat{R}(\mu^{(t+1)}) \geq 0$$

- Converges to a local optimum
- Complexity: $O(n \cdot k \cdot d)$ per iteration



3.1.1.4 Initializing k-Means

- Lloyd's heuristic does not generally converge to the optimal solution
- Performance heavily depends on the initialization
- Approaches towards initialization: Multiple random restarts, Farthest points heuristic (often works well, but prone to outliers), Seeding with k-Means++

How about random seeding initialization?

If we initialize centers uniformly from data points and we have one large group of points and other very small groups of points:

- For uniform random initialization, we'll likely have several initial centers in the same (the large) group. -> we initialize close to another.
- If this happens, since we only have k centers, we'll have groups with no initialized center!

3.1.1.5 K-Means++

- Start with random point as center for cluster 1:

$$i_1 \sim Unif(\{1, \dots, n\})$$

$$\mu_1 = x_{i_1}$$

  (i_1: index for data point which becomes center for cluster 1)
- Add centers 2 to k randomly, proportionally to squared distance to closest selected center:
  - For $j = 2$ *to* k
    - i_j sampled with propability $P(i_j = i) = \frac{1}{Z} \underbrace{\min_{1 \leq l < j} \left|\left|\boldsymbol{x_i} - \boldsymbol{\mu_l}\right|\right|_2^2}_{\text{to which already assigned center is } x_i \text{closest to}}$

      Where $Z = \sum_i \min_{1 \leq l < j} \left|\left|\boldsymbol{x_i} - \boldsymbol{\mu_l}\right|\right|_2^2$
    - $\boldsymbol{\mu_j} = \boldsymbol{x_{i_j}}$

Now, points far away from current centers are more likely to be picked. However, if we have a nearby group with **lots** of points, a center in this group is likely too since lots of data points are contributing.

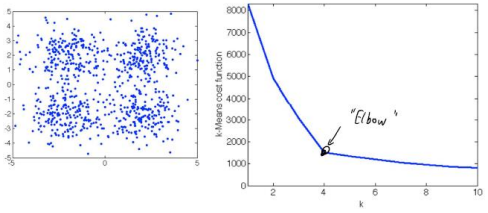
Expected cost is $O(\log k)$ times that of optimal k-Means solution

3.1.2 Model selection in clustering

- In general, model selection (e.g., determining the number of clusters) is very difficult
- Approaches:
  - Heuristic quality measures
  - Regularization (favor „simple" models with few parameters by penalizing complex models)
  - Information theoretic basis (tradeoff between robustness (stability) and informativeness)

3.1.2.1 Heuristic for determining k

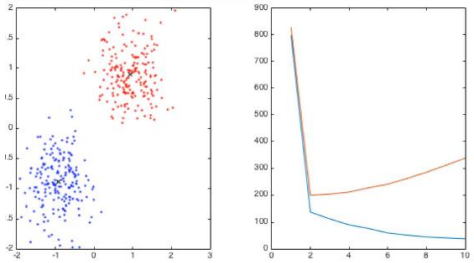
„Diminishing returns" in the loss function. Pick k so that increasing k leads to negligible decrease in loss:



3.1.2.2 Regularization

We can add a penalty term that qualifies the model complexity:

$$\min_{k} \min_{\mu_{1:k}} \sum_{i=1}^{n} \min_{1\le l\le k} \left|\left|x_i - \mu_l\right|\right|_2^2 + \lambda \cdot k$$



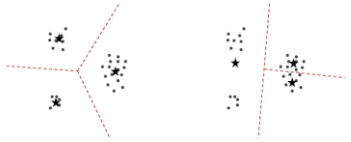
3.1.2.3 Generalization

The test loss generally keeps decreasing with the number of clusters (extreme case: we just have a cluster for every single training data point)

➔ We can't use cross-validation to determine the # of clusters!

3.1.3 Challenges with k-Means

- Generally only converges to local optimum
  - Performance strongly dependent on initialization!



  ➔ Practically not a big issue (with right initialization)
- Number of iterations required can be exponential (even in the plane!)
  - In practice however often converges quickly
  ➔ Practically not a big issue
- Determining the number of clusters k is difficult
- Cannot well model clusters of arbitrary shape
  K-Means is only able to do linear boundaries. It thus works well on isolated "bubble" groups. Cicrular groups, moon-shaped groups etc. aren't clustered correctly with k-Means.
  - (although Kernel-k-Means can be used to fix this)
- Determining the number of clusters k is difficult
  ➔ Major (unsolved) practical problem!

3.2 Dimension reduction

Basic challenge:
Given data set $\mathcal{D} = \{x_1, \dots, x_n\}, x_i \in \mathbb{R}^d$, obtain "embedding" (low-dimensional representation) $z_1, \dots, z_n \in \mathbb{R}^k$

Motivation:

- Visualization (k=1,2,3)
- Regularization (model selection)
- Unsupervised feature discovery (i.e., determine features from data!)

Typical approaches:

- Assume $\mathcal{D} = \{x_1, \dots, x_n\} \subseteq \mathbb{R}^d$
- Obtain mapping $f: \begin{matrix}\mathbb{R}^d \to \mathbb{R}^k \\ x_i \mapsto z_i\end{matrix}$ where $k \ll d$
- We can distinguish:
  - **Linear** dimension reduction $f(x) = Ax$
    -> Just a linear transformation, with $A \in \mathbb{R}^{k\times d}$
  - Nonlinear dimension reduction (parametric or non-parametric)

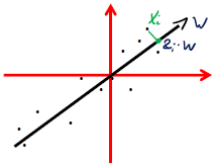
3.2.1 Linear dimension reduction

3.2.1.1 Linear dimension reduction as *compression*

- Motivation: Low-dimensional representation should allow to compress original data (accurate reconstruction)

Example: k=1

- Given data set $\mathcal{D} = \{x_1, \dots, x_n\} \subseteq \mathbb{R}^d$
- Want to represent data as points on a line $w \in \mathbb{R}^d$ with coefficients $z_1, \dots, z_n$
- I.e., want $z_i w \approx x_i$, assuming $\mu = \frac{1}{n}\sum_i x_i = 0$
  We need to assume uniform (centered) distribution of data points here (non-centered data can be centered)



-> Represent each data point by a point on the line
-> $\mathbb{R}^2 \to \mathbb{R}^1$ reduction

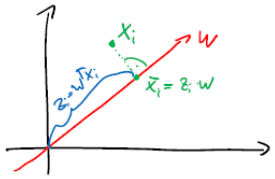
3.2.1.2 Linear dimension reduction for *reconstruction*

- Given data set $\mathcal{D} = \{x_1, \dots, x_n\} \subseteq \mathbb{R}^d$
- Want $z_i w \approx x_i$, e.g. minimizing $\left|\left|z_i w - x_i\right|\right|_2^2$
- To ensure uniqueness, normalize: $\left|\left|w\right|\right|_2 = 1$
  -> $\forall \alpha \neq 0: \underbrace{(\alpha \cdot w) \cdot \left(\frac{1}{\alpha} \cdot z\right) = w \cdot z}_{\substack{\text{scaled } w \text{ and scaled } z \text{ can be} \\ \text{equal to unscaled } w \text{ and } z \\ \to require\ ||w||_2=1}}$
- Optimize over $w, z_1, \dots, z_n$ jointly:

$$(w^*, z^*) = \underset{\substack{||w||_2=1 \\ z_1,\dots,z_n\in\mathbb{R}}}{\operatorname{argmin}} \sum_{i=1}^{n} \left|\left|z_i w - x_i\right|\right|_2^2$$

3.2.1.3 Solving for z given w

Suppose we consider some weight vector w. What can we say about the optimal z?
We want to find the best representation of point x_i on the line $z_i w$
-> the point $\bar{x}_i$ on the line w with the smallest distance from x_i is the point that is the intersection of the perpendicular.



Thus: $z_i^* = w^T x_i$
Thus, we effectively solve a regression problem, interpreting x as features and z as labels.

Inserting $z_i^* = w^T x_i$ into the optimization formula gives us:

$$w^* = \underset{||w||_2=1}{\operatorname{argmin}} \sum_{i=1}^{n} \underbrace{\left|\left|ww^T x_i - x_i\right|\right|_2^2}_{(**)}$$

Now we only need to optimize over w and not over z

$$(**) = (ww^T x_i - x_i)^T (ww^T x_i - x_i) =$$

$$= \left(\overbrace{x_i^T w \underbrace{w^T w}_{=1} w^T x_i}^{\text{same term as } (***)} - 2\overbrace{x_i^T w w^T x_i}^{(***)} + x_i^T x_i \right) = x_i^T x_i - (x_i^T w)(w^T x_i) =$$

$$= \overbrace{||x_i||_2^2}^{\Sigma_i} \underbrace{-(w^T x_i)^2}_{\text{minimize this}}$$

-> $||x_i||_2^2$ doesn't depend on w -> neglect for argmin

-> minimizing $-(w^T x_i)^2$ is equal to **maximizing** $+(w^T x_i)^2$.
Thus, the optimization problem from above is equivalent to

$$w^* = \underset{||w||_2=1}{\operatorname{argmax}} \sum_{i=1}^{n} \underbrace{(w^T x_i)^2}_{(***)}$$

$$(***) = \sum_{i=1}^{n} (w^T x_i)(x_i^T w) = \sum_{i=1}^{n} w^T (x_i x_i^T) w = w^T \underbrace{\left(\sum_{i=1}^{n} x_i x_i^T\right)}_{n\cdot\Sigma} w =$$

$$= w^T n \Sigma w$$

with $\Sigma = \frac{1}{n}\sum_{i=1}^{n} x_i x_i^T$ the empirical covariance matrix (assuming $\mu = 0$)
Thus, the problem $w^* = \underset{||w||_2=1}{\operatorname{argmax}} \sum_{i=1}^{n} (w^T x_i)^2$ is equivalent to:

$$\boxed{w^* = \underset{||w||_2=1}{\operatorname{argmax}}\ w^T \Sigma w}$$

With $\Sigma = \frac{1}{n}\sum_{i=1}^{n} x_i x_i^T$ the empirical covariance matrix, assuming the data is centered $\mu = \sum_{i=1}^{n} x_i = 0$
The optimal solution to $w^* = \underset{||w||_2=1}{\operatorname{argmax}}\ w^T \Sigma w$ is given by the **principal eigenvector** of Σ, i.e. $w^* = v_1$ where

$$\Sigma = \sum_{i=1}^{d} \lambda_i v_i v_i^T \ , \lambda_1 \geq \dots \geq \lambda_d \geq 0$$

$$\left|\left|v_i\right|\right|_2 = 1 \ , v_i v_j^T = \begin{cases} 1 \ , i = j \\ 0 \ , else \end{cases} \to orthonormal\ eigenvectors$$

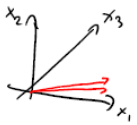
(Proof: sees slides)

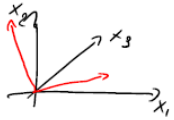
3.2.2 How about k>1

Suppose we wish to project to more than one dimension ($k > 1$). Thus we want:

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

where $W \in \mathbb{R}^{d\times k}$ is orthogonal, $z_1, \dots, z_n \in \mathbb{R}^k$
-> Why W orthogonal? E.g. dimension reduction $\mathbb{R}^3 \to \mathbb{R}^2$, the $x_1 - x_2$ plane can be described by many vectors -> we want the columns of W to be orthogonal!





This is called the **Principal Component Analysis** problem.
Its solution can be obtained in close form even for $k > 1$

3.2.2.1 Principal component analysis (PCA)

Given centered data $\mathcal{D} = \{x_1, \dots, x_n\} \subseteq \mathbb{R}^d, 1 \leq k \leq d$

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} x_i x_i^T \,, \mu = \sum_{i=1}^{n} x_i = 0$$

- The solution to the PCA problem

$$(\mathbf{W}, \mathbf{z}_1, \dots, \mathbf{z}_n) = \arg\min \sum_{i=1}^{n} ||\mathbf{W}\mathbf{z}_i - \mathbf{x}_i||_2^2$$

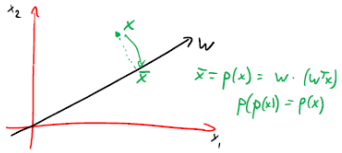
where $\mathbf{W} \in \mathbb{R}^{d\times k}$ is orthogonal, $\mathbf{z}_1, \dots, \mathbf{z}_n \in \mathbb{R}^k$

is given by $\mathbf{W} = (\mathbf{v}_1|\dots|\mathbf{v}_k)$ and $\mathbf{z}_i = \mathbf{W}^T\mathbf{x}_i$
where $\Sigma = \sum_{i=1}^{d} \lambda_i \mathbf{v}_i \mathbf{v}_i^T \qquad \lambda_1 \geq \dots \geq \lambda_d \geq 0$

Where $\boldsymbol{v_1}, \dots, \boldsymbol{v_n}$ are the k principal eigenvectors (i.e. the k eigenvectors with k largest eigenvalues)

PCA is a projection!

The linear mapping $f(\boldsymbol{x}) = \boldsymbol{W^T x}$ obtained from PCA projects vectors $\boldsymbol{x} \in \mathbb{R}^d$ into a k-dimensional subspace.
This projection is chosen to minimize the reconstruction error (measured in Euclidean norm)



Connection to SVD

- Can obtain PCA through Singular-Value Decomposition
- *Recall:* Can represent any $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ as $\boldsymbol{X} = \boldsymbol{USV^T}$ where $\boldsymbol{U} \in \mathbb{R}^{n\times d}$ and $\boldsymbol{V} \in \mathbb{R}^{d\times d}$ are orthogonal, and $\boldsymbol{S} \in \mathbb{R}^{n\times d}$ is diagonal (wlog in decreasing order) ($n \geq d$)
  Its entries are called singular values:

- The top k principal components are exactly the first k columns of $\boldsymbol{V}$.
  PCs are eigenvectors associated with largest eigenvalues of $\Sigma = \frac{1}{n}\boldsymbol{X^T X}$.
  Using $\boldsymbol{X} = \boldsymbol{USV^T}$, we have

$$\boldsymbol{X^T X} = (\boldsymbol{VS^T U^T})(\boldsymbol{USV^T}) = \boldsymbol{VS^T} \underbrace{\boldsymbol{U^T U}}_{=I_n} \boldsymbol{SV^T} = \boldsymbol{V} \underbrace{\boldsymbol{S^T S}}_{\boldsymbol{D}} \boldsymbol{V}$$

Common PCA usecases:

- Visualization (k=1, 2, 3)
- Feature learning
- Compression

3.2.2.2 Choosing k

- For visualization: by inspection ☺
- For feature induction: by cross-validation
- Otherwise: Pick k so that most of the variance is explained (similar to the choice in k-means)

Reconstruction:
The higher dimensional our subspace is (k), the better we can reconstruct
-> concave function -> find a sweet spot

3.2.2.3 PCA vs. k-Means

PCA: we constrain $\boldsymbol{W}$ but don't constrain $\{z_1, \dots, z_n\}$
k-Means: we don't constrain $\boldsymbol{W}$ but we constrain $\{z_1, \dots, z_n\}$
For k-Means, we set the columns of $\boldsymbol{W}$ to the centers of the clusters.
$\{z_1, \dots, z_n\}$ is just the set of unit vectors in $\mathbb{R}^k$.

PCA Problem:

$$(\mathbf{W}, \mathbf{z}_1, \dots, \mathbf{z}_n) = \arg\min \sum_{i=1}^{n} ||\mathbf{W}\mathbf{z}_i - \mathbf{x}_i||_2^2$$

where $\mathbf{W} \in \mathbb{R}^{d\times k}$ is orthogonal, $\mathbf{z}_1, \dots, \mathbf{z}_n \in \mathbb{R}^k$

k-Means problem: (equivalent formulation)

$$(\mathbf{W}, \mathbf{z}_1, \dots, \mathbf{z}_n) = \arg\min \sum_{i=1}^{n} ||\mathbf{W}\mathbf{z}_i - \mathbf{x}_i||_2^2$$

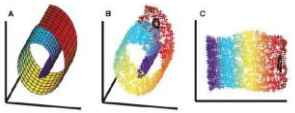
where $\mathbf{W} \in \mathbb{R}^{d\times k}$ *arbitrary* and $\mathbf{z}_1, \dots, \mathbf{z}_n \in E_k$

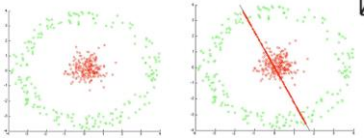
Hereby, $E_k = \{[1,0,\dots,0], \dots, [0,\dots,0,1]\}$ is the set of unit vectors in $\mathbb{R}^k$
-> $e_j = [0,\dots,0,1,0,\dots,0]$ where the 1 is at the j-th position.

- Can think of PCA and k-Means to solve a similar unsupervised learning problem, with different constraints
- Both aim to compress the data with maximum fidelity under constraints on the model complexity
- This insight gives rise to a much broader class of techniques!
  - Matrix factorization

3.2.3 Nonlinear dimension reduction

Linear dimension reduction: think of data as spread out linearly along some lines.
-> What if data is **not** spread out linearly but sort of curled up? E.g.





➔ **Use Kernels!**

- Recall: In supervised learning, kernels allowed us to solve non-linear problems by reducing them to linear ones in high-dimensional (implicitly represented) spaces
- Can take the same approach for unsupervised learning!
- Procedure:
  - First, map to higher dimensional space ($D \gg d$):

$$\underbrace{\boldsymbol{x}}_{\in\mathbb{R}^d} \mapsto \underbrace{\boldsymbol{\phi(x)}}_{\in\mathbb{R}^D}$$

  - Then, do dimension reduction in this higher dimensional space ($k \ll D$):

$$\underbrace{\boldsymbol{\phi(x)}}_{\in\mathbb{R}^D} \mapsto \underbrace{\boldsymbol{z}}_{\in\mathbb{R}^k}$$

  But it could be that $k > d$

Recall PCA for k=1:

Optimal solution to PCA problem solves, for $\Sigma = \frac{1}{n}\boldsymbol{X^T X}$

$$\underset{||w||_2=1}{\operatorname{argmax}} \boldsymbol{w^T X^T X w} = \underset{||w||_2=1}{\operatorname{argmax}} \sum_{i=1}^{n} (\boldsymbol{w^T x_i})^2$$

Applying feature maps, using $\boldsymbol{w} = \sum_{j=1}^{n} \alpha_j \phi(\boldsymbol{x_j})$ and observing that $||\boldsymbol{w}||_2 = \alpha^T \boldsymbol{K} \alpha = 1$

$$\arg\max_{||\mathbf{w}||_2=1} \sum_{i=1}^{n} (\mathbf{w}^T\phi(\mathbf{x}_i))^2 = \arg\max_{\alpha^T\mathbf{K}\alpha=1} \sum_{i=1}^{n} \Big(\sum_{j=1}^{n} \alpha_j \phi(\mathbf{x}_j)^T \phi(\mathbf{x}_i)\Big)^2$$
$$= \arg\max_{\alpha^T\mathbf{K}\alpha=1} \sum_{i=1}^{n} \Big(\sum_{j=1}^{n} \alpha_j k(\mathbf{x}_j, \mathbf{x}_i)\Big)^2 = \arg\max_{\alpha^T\mathbf{K}\alpha=1} \sum_{i=1}^{n} \Big(\alpha^T \mathbf{K}_i\Big)^2$$
$$= \arg\max_{\alpha^T\mathbf{K}\alpha=1} \alpha^T \mathbf{K}^T \mathbf{K} \alpha$$

3.2.3.1 Kernel PCA (k=1)

- The Kernel PCA problem (k=1) requires solving

$$\alpha^* = \underset{\boldsymbol{\alpha^T K \alpha = 1}}{\operatorname{argmax}} \boldsymbol{\alpha^T K^T K \alpha}$$

  With $\boldsymbol{K}$ the kernel matrix.
- The optimal solution is obtained in closed form from the eigenvalue decomposition of $\boldsymbol{K}$:

$$\boxed{\alpha^* = \frac{1}{\sqrt{\lambda_1}} \boldsymbol{v_1}}, \boldsymbol{K} = \sum_{i=1}^{n} \lambda_i \boldsymbol{v_i v_i^T} \,, \lambda_1 \geq \dots \geq \lambda_d \geq 0$$

$$\alpha^T \boldsymbol{K} \alpha = \frac{1}{\sqrt{\lambda_1}} \boldsymbol{v_1^T} \underbrace{\boldsymbol{K v_1}}_{\lambda_1 v_1} \frac{1}{\sqrt{\lambda_1}} = \frac{1}{\lambda_1} v_1^T \lambda_1 v_1 = \underbrace{v_1^T v_1}_{\textit{orthonormal}} = 1$$

3.2.3.2 Kernel PCA (general k)

- For general k > 1, the Kernel Principal Components are given by $\alpha^{(1)}, \dots, \alpha^{(k)} \in \mathbb{R}^n$

where $\alpha^{(i)} = \frac{1}{\sqrt{\lambda_i}} \mathbf{v}_i$

is obtained from: $\mathbf{K} = \sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^T \quad \lambda_1 \geq \cdots \geq \lambda_d \geq 0$

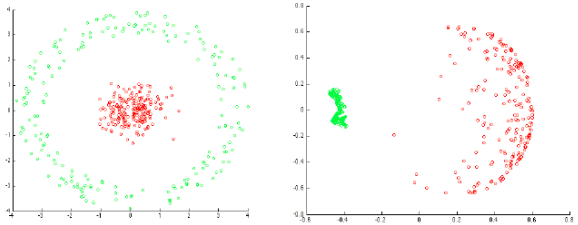
- Given this, a new point $\boldsymbol{x}$ is projected as $\mathbf{z} \in \mathbb{R}^k$

$$z_i = \sum_{j=1}^{n} \alpha_j^{(i)} k(\mathbf{x}, \mathbf{x}_j)$$

Note:

$$z_i = f_i(\boldsymbol{x}) = \boldsymbol{w_i^T} \phi(\boldsymbol{x}) = \left(\sum_{j=1}^{n} \alpha_j^{(i)} \phi(\boldsymbol{x_j}) \right)^{\boldsymbol{T}} \phi(\boldsymbol{x})$$
$$= \sum_{j=1}^{n} \alpha_j^{(i)} \underbrace{\phi(\boldsymbol{x_j})^{\boldsymbol{T}} \phi(\boldsymbol{x})}_{=k(\boldsymbol{x}, \boldsymbol{x_j})}$$

Illustration:



Side note: Applying k-means on kernel-principal components is sometimes called Kernel-k-means or Spectral Clustering.

3.2.3.3 Notes on Kernel PCA

- Kernel-PCA corresponds to applying PCA in the feature space induced by the kernel k
- Can be used to discover non-linear feature maps in closed form
- This can be used as inputs, e.g., to SVMs given "multilayer support vector machines"
- May want to center th kernel: $E = \frac{1}{n}[1, \dots, 1]^T[1, \dots, 1]$

$$\boldsymbol{K' = K - KE - EK + EKE}$$

-> Computing the mean in kernel space might be problematic -> compute the centering directly in kernel)

- Kernel-PCA requires data specified as kernel
  - We need a specific kernel (choice affects representation)
  - Complexity grows with the number of data points
  - Cannot easily „explicitly" embed high-dimensional data (unless we have an appropriate kernel)

3.2.4 Autoencoders

Key idea: Try to learn the identity function!

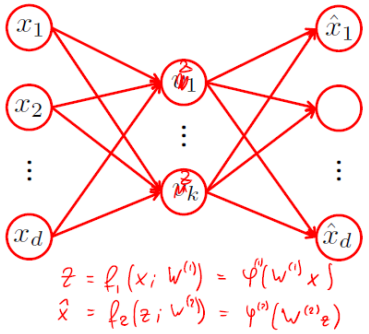
$$\boldsymbol{x} \approx f(\boldsymbol{x}; \boldsymbol{\theta})$$

What function f should we pick?

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = f_2(f_1(\boldsymbol{x}; \boldsymbol{\theta_1}); \boldsymbol{\theta_2}) \approx \boldsymbol{x}$$
$$f_1 \colon \mathbb{R}^d \longrightarrow \mathbb{R}^k \quad "encoder"$$
$$f_2 \colon \mathbb{R}^k \longrightarrow \mathbb{R}^d \quad "decoder"$$

Both these functions f_1 and f_2 have parameters we can optimize over.
-> f_1 needs to somehow preserve information about x, else f_2 won't be able to reconstruct x
We can pick the functions f_1 and f_2 -> we can pick Neural Networks!

3.2.4.1 Neural Network Autoencoders



- Neural network Autoencoders are ANNs where
  - There is one output unit for each of the d input units
  - The number k of hidden units is usually smaller than the number of inputs
- The goal is to optimize the weights such that the output agrees with the input

3.2.4.2 Training autoencoders

- The goal is to optimize the weights such that the output agrees with the input
  - For example, minimize the square loss

$$\min_{\boldsymbol{W}} \sum_{i=1}^{n} \underbrace{||\boldsymbol{x_i} - \boldsymbol{f}(\boldsymbol{x_i}; \boldsymbol{W})||_2^2}_{\sum_{j=1}^{d} (x_{i,j} - f_j(x_i; W))^2}$$

With $\boldsymbol{f}(\boldsymbol{x_i}; \boldsymbol{W}) = \hat{\boldsymbol{x}}_{\boldsymbol{i}} = estimated\ \boldsymbol{x_i}$
We could also use other loss functions.
- Find local minimum via SGD (backpropagation)
- Initialization matters and is challenging

3.2.4.3 Autoencoders vs. PCA

- The internal representation $\boldsymbol{v} = \varphi(\boldsymbol{W^{(1)}x})$ is the „dimensionality reduced" input
- If the activation function is the identity $\varphi(z) = z$ then in fact fitting a NN autoencoder is equivalent to PCA!

Encoder: $f_1(x) = \boldsymbol{W^{(1)^T}x} =: z\,, f_2(z) = \boldsymbol{W^{(2)^T}}z$

Optimal solution $\boldsymbol{W^{(1)}}$ is given by PCA, $\boldsymbol{W^{(2)}} = \boldsymbol{W^{(1)^T}}$

4 Neural Networks/ "feature learning"

Constant question in ML: what are good features?

- Features are crucial in ML. Success in learning crucially depends on the quality of features.
- Hand-designing features requires domain-knowledge
- What about kernel methods?
  - Rich set of feature maps
  - Can fit „any function" with infinite data*
  - Choosing the „right" kernel can be challenging
  - Computational complexity grows with size of data

Can we learn good features from data directly??

4.1 Learning features

Up until now, we solved problems by minimizing some loss function:

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{argmin} \sum_{i=1}^{n} l\left(y_i; \sum_{j=1}^{m} w_j \phi_j(\boldsymbol{x_i})\right)$$

Key idea now: Parametrize the feature maps and optimize over the parameters!

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

➔ We now have free parameters $\boldsymbol{w}$ **and** θ!
ϕ now has free parameters θ_j over which we can optimize
➔ This is the basic idea behind neural networks
We want to include nonlinear functions to open up richer models. A linear function ϕ would just again yield a linear model.

4.2 Parametrizing feature maps

One possibility of parametrizing the feature maps is:

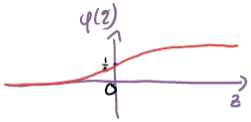
$$\phi(\boldsymbol{x}, \boldsymbol{\theta}) = \varphi(\boldsymbol{\theta^T x})$$

Hereby, $\boldsymbol{\theta} \in \mathbb{R}^{\boldsymbol{d}}$ and $\varphi \colon \mathbb{R} \longrightarrow \mathbb{R}$ is a nonlinear function, called "activation function". $\boldsymbol{\theta}$ is going to be learned, φ is selected by us.

4.2.1 Activation functions

4.2.1.1 Sigmoid activation function

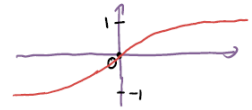
$$\varphi(z) = \frac{1}{1 + \exp(-z)}$$



4.2.1.2 tanh activation function

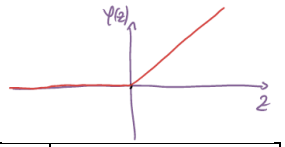
$$\varphi(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

Note: Sigmoid and tanh are almost identical up to some shift



4.2.1.3 Rectified linear units (ReLU)

$$\varphi(z) = \max(z, 0)$$



Activation	Sigmoid/ tanh	ReLU
Advantages	Everywhere differentiable, gradient is non-zero	Gradient is more well-behaved
Disadvantages	Gradient is almost zero everywhere except for a narrow range around $z = 0$	Not differentiable at $z = 0$, gradient = 0 for $z < 0$

In practice, ReLU works often much better than Sigmoid and tanh because its gradient is more well-behaved.

4.2.2 Derivatives of activation function

4.2.2.1 Sigmoid

$$\varphi'(z) = -\frac{1}{(1+e^{-z})^2}e^{-z}(-1) = \frac{e^{-z}}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}} = (1-\varphi(z)) \cdot \varphi(z)$$

Properties:

+ $\varphi(z)$ differentiable (with non-zero gradient) everywhere
- $\varphi(z) \approx 0$ everywhere except for $z \approx 0$

Why do we care whether $\varphi'(z)$ is zero?

If $\varphi'(z)$ Is zero at some point, the whole signal (the gradient) is zero. This zero backpropagates due to the nature of the algorithm -> everything back from $\varphi'(z) = 0$ will be zero too! This is especially bad for deep networks since the probability for hitting a zero at some point is large.

4.2.2.2 ReLU

$$\varphi'(z) = \begin{cases} 1, if\ z > 0 \\ 0, if\ z > 0 \end{cases}, \quad at\ z = 0\ not\ differentiable$$

Properties:

+ very efficient
+ > 0 in $\mathbb{R}_+$ (very important)
- not differentiable at $z = 0$ (in practice set to 0, doesn't really matter)

4.3 Artificial Neural networks (ANNs)

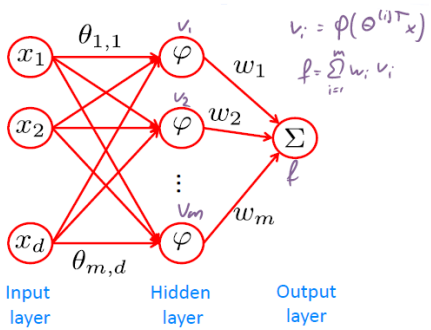
Functions of this form

$$f(\boldsymbol{x}; \boldsymbol{w}, \boldsymbol{\theta}) = \sum_{j=1}^{m} \varphi(\boldsymbol{\theta}_j^T \boldsymbol{x})$$

are examples of artificial Neural Networks (ANNs), also called Multi-layer Perceptron. Basically, what we do is, taking our linear model from before and parametrizing our feature functions by sending a linear function $\boldsymbol{\theta}_j^T \boldsymbol{x}$ through a nonlinear function $\varphi(.)$. The linear parts in the Neural Network are being learned while the nonlinear parts are fixed.
More generally, the term artificial neural network refers to nonlinear functions which are nested compositions of (variable) linear functions composed with (fixed) nonlinearities.

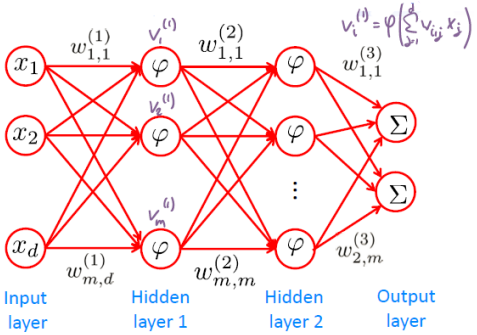
Nested compositions: take $\boldsymbol{x}$ and replace it with another nonlinear function φ that takes another set of inputs -> creates layers of Neural Networks
-> deep learning

Examples:



We can have more than one output (useful e.g. for multi-class classification or multi-output regression). We can also have more than one hidden layer.

Example:

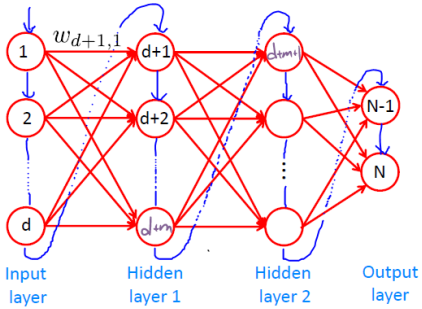


Remarks:

- Different layers don't need the same amount of neurons and can have different kind of nonlinearities
- Notation:

  $w_{i,j}^{(1)}$ = connection between j-th input and i-th output unitm (1) refers to the layer

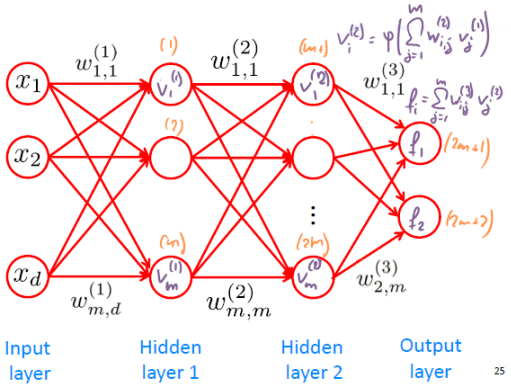
Indexing units



4.4 Forward propagation

Suppose we have learned all parameters $w_{i,j}$. Given an input, how do we make predictions? How do we evaluate the network?

➔ **Forward propagation**



Compute all values $v_i^{(1)}$ on the first hidden layer first:

$$v_i^{(i)} = \varphi\left(\sum_{j=1}^{m} w_{i,j}^{(1)} x_j\right)$$

Then compute all values $v_i^{(2)}$ on hidden layer 2, etc. Finally, compute the values f_i of the output layer:

$$f_i = \sum_{j=1}^{m} w_{i,j}^{(3)} v_j^{(2)}$$

- For each unit j on input layer, set its value $v_j = x_j$
- For each layer $\ell = 1 : L-1$
  - For each unit j on layer ℓ set its value

$$v_j = \varphi\Big(\sum_{i \in \text{Layer}_{\ell-1}} w_{j,i} v_i\Big)$$

- For each unit j on output layer, set its value

$$f_j = \sum_{i \in \text{Layer}_{L-1}} w_{j,i} v_i$$

- Predict $y_j = f_j$ for regression,
  $y_j = \text{sign}(f_j)$ for classification

Using the following notation:

$$\boldsymbol{W}^{(l)} = \begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \dots & w_{1,m^{(l-1)}}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \dots & w_{2,m^{(l-1)}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m^{(l)},1}^{(l)} & w_{m^{(l)},2}^{(l)} & \dots & w_{m^{(l)},m^{(l-1)}}^{(l)} \end{bmatrix} \in \mathbb{R}^{m^{(l)} \times m^{(l-1)}}$$

We get forward propagation in matrix form:

- For input layer: $\mathbf{v}^{(0)} = \mathbf{x}$
- For each hidden layer $\ell = 1 : L-1$

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)} \mathbf{v}^{(\ell-1)} \in \mathbb{R}^{m^{(\ell)}}$$

$$\mathbf{v}^{(\ell)} = \varphi(\mathbf{z}^{(\ell)}) \in \mathbb{R}^{m^{(\ell)}}$$

applied componentwise

- For output layer: $f = \mathbf{W}^{(L)} \mathbf{v}^{(L-1)}$
- Predict: $\mathbf{y} = \mathbf{f}$ (regression) or $\mathbf{y} = \text{sign}(\mathbf{f})$ (class.)

$$z_i^{(l)} = \sum_{j=1}^{m^{(l-1)}} w_{i,j}^{(l)} v_j^{(l-1)}$$

4.5 Training the weights

Given a dataset $\mathcal{D} = \{(\boldsymbol{x_1}, y_1), \dots, (\boldsymbol{x_n}, y_n)\}$, we want to optimize the weights

$$\boldsymbol{W} = [\boldsymbol{W}^{(1)}, \dots, \boldsymbol{W}^{(L)}]$$

where L is the number of layers.

How do we measure and optimize goodness of fit?

- Apply loss function (e.g. perceptron loss, multi-class hinge loss, square loss etc.) to output

$$l(\boldsymbol{W}; \boldsymbol{y}, \boldsymbol{x}) = l(\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{W}))$$

- Then optimize the weights to minimize loss over $\mathcal{D}$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{argmin} \sum_{i=1}^{n} l(\boldsymbol{W}; \boldsymbol{y}_i, \boldsymbol{x}_i)$$

With this, we want to de empirical risk minimization, i.e. jointly optimize over all weights for all layers to minimize loss over the training data. However, this is in general a non-convex optimization problem. Nevertheless, we can still try to find local optima!

➔ Initialization matters!

Side note: Losses for multi-outputs

When predicting multiple outputs at the same time, usually define loss as sum of per-output losses:

$$l(\boldsymbol{W}; \boldsymbol{y}, \boldsymbol{x}) = \sum_{i=1}^{p} l_i(\boldsymbol{W}; y_i, \boldsymbol{x}) \text{ , where } p = \#of \text{ outputs}$$

Note: we could have different losses for different outputs
-> For regression, we might use square loss, for classification, may use the perceptron or hinge loss.

4.6 Stochastic gradient descent for ANNs

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} \sum_{i=1}^{n} \ell(\mathbf{W}; \mathbf{y}_i, \mathbf{x}_i)$$

- Initialize weights $\mathbf{W} \leftarrow$ how ?
- For t = 1,2,...
  - Pick data point $(\mathbf{x}, \mathbf{y}) \in D$ uniformly at random
  - Take step in negative gradient direction

$$\mathbf{W} \leftarrow \mathbf{W} - \eta_t \nabla_{\mathbf{W}} \ell(\mathbf{W}; \mathbf{y}, \mathbf{x})$$

Note: The larger the amount of neurons on the hidden layers, the less the chance of getting stuck in a local optimum since the weights get initialized at random -> more neurons -> more weights -> better distribution, smaller probability of a bad starting point. In practice, people use massively over-sized networks which leads to higher probability of finding the global optimum.

4.7 Computing the gradient

In order to apply SGD, we need to compute

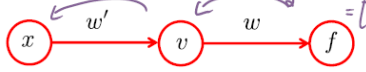
$$\nabla_{\boldsymbol{W}} l(\boldsymbol{W}; \boldsymbol{y}, \boldsymbol{x})$$

I.e. for each weight between any two connected units i and j we need

$$\frac{\partial}{\partial w_{i,j}} l(\boldsymbol{W}; \boldsymbol{y}, \boldsymbol{x})$$

4.7.1 Simple example

- ANN with 1 output, 1 hidden and 1 input unit



We have weights $W = [w, w']$ and error signals δ, δ'
In order to compute the derivative of our cost (the contribution of our data point (x, y) to the loss) we have to proceed **backwards** -> start with output f, compute error signal δ from f, propagate backwards, then proceed with hidden layer and work further back to input layer.

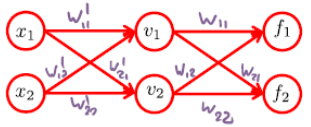
Here, the output is: $f(x, \boldsymbol{W}) = w \overbrace{\varphi(\underbrace{w'x}_{z})}^{v}$

The loss: $L := L(w, w') = l(y, f) = l_y(f)$ (subscript y to show that it depends on y). E.g. $l_y(f) = (y-f)^2 \Rightarrow l_y'(f) = 2(y-f)(-1) = 2(f-y)$
Then:

- $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial w} = \overbrace{l_y'(f)}^{\delta} v$
- $\frac{\partial L}{\partial w'} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial w'} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial v}\frac{\partial v}{\partial w'} = \underbrace{\delta w \varphi'(z)}_{\delta'} \cdot x$
  - $\rightarrow \frac{\partial L}{\partial f} = l_y'(f) =: \delta \quad \frac{\partial f}{\partial w} = \frac{\partial}{\partial w}(w\varphi(w'x)) = \varphi(w'x) = v$
  - $\rightarrow \frac{\partial f}{\partial v} = \frac{\partial}{\partial v}(wv) = w \quad \frac{\partial v}{\partial w'} = \frac{\partial}{\partial w'}\varphi(w'x) = \varphi'(z)\frac{\partial}{\partial w'}(w'x) = \varphi'(z)x$

E.g. $l_y(f) = (y-f)^2 \Rightarrow l_y'(f) = 2(y-f)(-1) = 2(f-y)$

4.7.2 More complicated example



$$L = \sum_i \ell_i\Big(\sum_j w_{i,j}\varphi\Big(\sum_k w'_{j,k}x_k\Big)\Big)$$

$= L(W), W = [w_{ij}, w'_{ij}]_{ij}$

$\nabla_W L = \left[\frac{\partial L}{\partial w_{ij}}, \frac{\partial L}{\partial w'_{ij}}\right]_{ij}$

-> $L = \sum_i l_i\big(\sum_j w_{i,j} v_j\big) = \sum_i l_i(f_i)$, this example: $L = l(f_1) + l(f_2)$

-> $\nabla_W L = \left[\frac{\partial L}{\partial w_{i,j}}, \frac{\partial L}{\partial w'_{i,j}}\right]_{i,j}$ -> calculate for all i,j

Note: $w_{i,j}$ has only an effect on output f_i. Thus has also only an effect on l_i from L, all other parts are constant w.r.t. $w_{i,j}$

➔ $\frac{\partial L}{\partial f_i} = l_i'(f_i) + 0$

$\frac{\partial L}{\partial w_{i,j}} = \frac{\partial L}{\partial f_i}\frac{\partial f_i}{\partial w_{i,j}} = l_i'(f_i)\frac{\partial}{\partial w_{i,j}}\big(\sum_j w_{i,j}\varphi\big(\sum_k w'_{j,k}x_k\big)\big) =$

$= l_i'(f_i)\varphi\big(\sum_k w'_{i,k}x_k\big) = l_i'(f_i)v_j$

$\frac{\partial L}{\partial w'_{j,k}} = \sum_i \underbrace{\frac{\partial l_i}{\partial f_j}}_{\delta_i}\underbrace{\frac{\partial f_i}{\partial v_j}}_{w_{i,j}}\underbrace{\frac{\partial v_j}{\partial w'_{j,k}}}_{\varphi'(z_k)x_k} = \sum_{i=1}^{2} \delta_i w_{i,j}\varphi'(z_j)x_k = \varphi'(z_j)x_k \sum_{i=1}^{2} \delta_i w_{i,j}$

$\frac{\partial f_i}{\partial v_j} = \frac{\partial}{\partial v_j}\left(\sum_j w_{i,j}\underbrace{\varphi\big(\sum_k w'_{j,k}x_k\big)}_{v_j}\right) = w_{i,j}$

$\frac{\partial v_j}{\partial w'_{j,k}} = \frac{\partial}{\partial w'_{j,k}}\Big(\varphi\big(\sum_k w'_{j,k}x_k\big)\Big) = \frac{\partial}{\partial z_j}\varphi(z_j)\frac{\partial}{\partial w'_{j,k}}\big(\sum_k w'_{j,k}x_k\big) = \varphi'(z_k)x_k$

4.8 Backpropagation

- For each unit j on the output layer
  - Compute error signal $\delta_j = \ell_j'(f_j)$
  - For each unit i on layer L, $\frac{\partial}{\partial w_{j,i}} = \delta_j v_i$
- For each unit j on hidden layer $\ell = L-1 : -1 : 1$
  - Compute error signal $\delta_j = \varphi'(z_j) \sum_{i \in \text{Layer}_{\ell+1}} w_{i,j}\delta_i$
  - For each unit i on layer $\ell - 1$, $\frac{\partial}{\partial w_{j,i}} = \delta_j v_i$

With L the last layer.

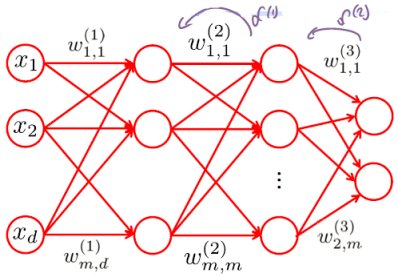
Backpropagation in Matrix form:

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

Again with

$$\boldsymbol{W}^{(l)} = \begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \dots & w_{1,m^{(l-1)}}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \dots & w_{2,m^{(l-1)}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m^{(l)},1}^{(l)} & w_{m^{(l)},2}^{(l)} & \dots & w_{m^{(l)},m^{(l-1)}}^{(l)} \end{bmatrix} \in \mathbb{R}^{m^{(l)} \times m^{(l-1)}}$$

Backpropagation illustration



4.9 Initializing weights

- Since optimization problem is non-convex, initialization matters!
- Usually random initialization works well, e.g.:
  - $w_{i,j} \sim \mathcal{N}(0, 0.1)$
  - $w_{i,j} \sim \mathcal{N}(0, 1/\sqrt{|Layer_{l+1}|})$
    -> adapt the randomness to the amount of layers (no real reason…)
- Might want to repeat training multiple times, to avoid getting stuck in a poor local optimum
- „Incorrect“ initialization can lead to bad results (see both demos)

4.10 Learning rate

To implement SGD rule

$$\boldsymbol{W} = \boldsymbol{W} - \eta_t \nabla_{\boldsymbol{W}} l(\boldsymbol{W}; \boldsymbol{y}, \boldsymbol{x})$$

We need to choose a learning rate η_t
Approaches:

- Usually it’s a good idea to start with a (small) learning rate, and decrease slowly after some iterations, e.g.
  $\eta_t = \min(0.1, 100/t)$
  Idea behind this: At first, we just want to go roughly into the right direction, don’t really care about accuracy. Then later on, we care about accuracy -> decrease learning rate.
- May want to monitor ratio of weight change (gradient) to weight magnitude.
  - If too small, increase learning rate
  - If too large, decrease learning rate
- Make sure step size is appropriate for current weights
  - Large weights -> large η_t
  - Small weights -> small η_t

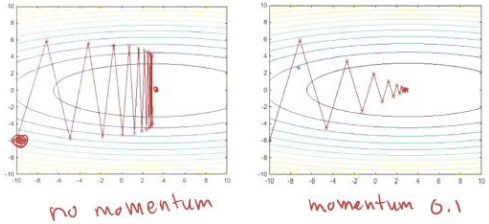
4.10.1 Learning with momentum

Common extension to training with SGD. Can help to escape local minima.
Idea: Move not only into direction of gradient, but also in direction of last weight update.
Updates:

$$a = m \cdot a + \eta_t \nabla_{\boldsymbol{W}} l(\boldsymbol{W}; \boldsymbol{y}, \boldsymbol{x})$$
$$\boldsymbol{W} = \boldsymbol{W} - a$$

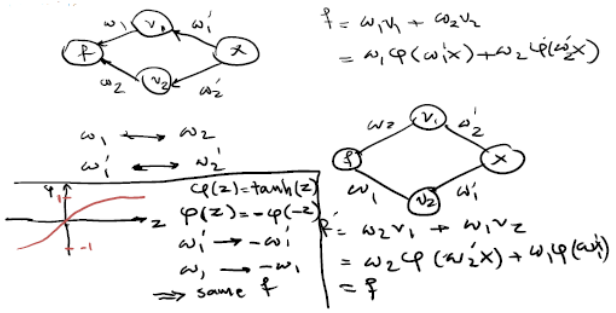
with m: friction parameter, start e.g. with 0.5, later increase it.

Intuition: Ball is pushed down a slope with a local minima. To reach the global minimum, we give the ball an extra push to get over the local minima and reach the global minimum.
Further, learning with momentum is also useful to prevent oscillations, which will lead to finding the minima faster.



4.11 Weight-space symmetry

Multiple distinct weights compute the same predictions.



- The two nets above represent the same function! Thus different nets represent same function -> leads to multiple local minima
- Symmetry of the $tanh$ function leads to equivalent nets to
  - $\tanh(-w_1) = \tanh(w_1)$

-> We might get different sets of weights for the same minima.

4.12 Avoiding overfitting

Neural networks have many parameters -> danger of overfitting

4.12.1 Early stopping

In general, might not want to run training until weights converge
-> Overfitting!
One possibility:

- Monitor prediction performance on validation set;
- stop training once validation error starts to increase

However, we don’t want to stop immediately when the validation increases a bit -> add a patience term.

4.12.2 Regularization

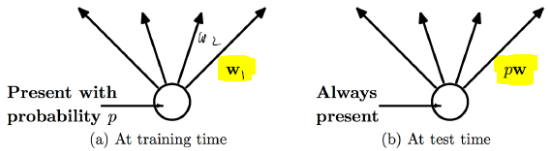
Add penalty term to keep weights small

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{argmin} \sum_{i=1}^{n} l(\boldsymbol{W}; \boldsymbol{y}_i, \boldsymbol{x}_i) + \lambda ||\boldsymbol{W}||_2^2$$

Pick λ through cross-validation. Problem: Training NNs is expensive, CV is thus not very efficient.

4.12.3 Dropout regularization

General idea: We want to make sure that we don’t rely too heavily on a single layer/ unit.
During each iteration we randomly ignore (“drop out”) hidden units and all connected weights to that unit with a certain probability p. After the iteration, the unit (&weights) become available again.
After training, multiply the weights with p to compensate.



(a) At training time (b) At test time

4.13 Invariances

Predictions should be unchanged under some transformations of the data, e.g.:

- Classification of handwritten digits:
  - Be robust against translation, shifts, rotations, scale, etc.
- Speech recognition
  - Be robust against lower or higher pitch, etc.

Invariances can be learned from data*

How to encourage a model to learn specific invariances?

- Implement invariance into structure of ANN (e.g. using convolutional neural networks)

4.14 Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) are ANNs for specialized applications (e.g. image recognition).

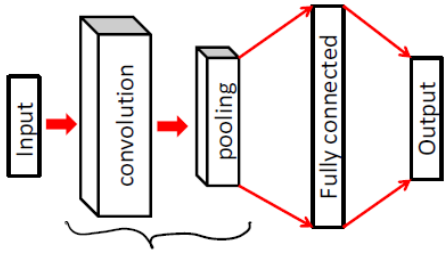
- The hidden layer(s) closest to the input layer shares parameters: Each hidden unit only depends on all „closeby" inputs (e.g., pixels), and weights constrained to be identical across all units on the layer.
- This reduces the number of parameters, and encourages robustness against (small amounts of) translation
- The weights can still be optimized via backpropagation

Example:

Handwritten digits, we want a NN to classifiy it. If digit image has 10^6 pixels and NN with one layer, 10^6 units -> $(10^6)^2$ parameters -> hard to store

Solution: CNN -> each unit in the net corresponds to a patch of the image and the weights are going to apply only to that particular patch, zero everywhere else. All patches must be treated similarly -> drastically decrease # of parameters -> allows deeper networks

CNN architecture:



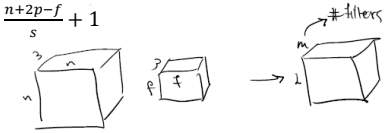
Terminology:

- filters/kernels
- stride (step size)
- padding

4.14.1 Computing output dimensions

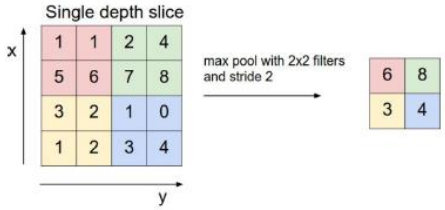
Applying m different fxf filters to a nxn image, Padding p, Stride s

Then: $L = \frac{n+2p-f}{s} + 1$



4.14.2 Pooling layers

In some applications (e.g., image classification), it can make sense to aggregate several units to decrease the width of the network (and hence # of parameters). Usually, one considers either the average or the maximum value.



High-level idea behind max-pooling:

Let's say I want to recognize faces. I apply filters to patches of the image and get responses -> high response = likely a face.

Thus, in a certain area, I only care whether I somewhere captured parts of a face -> only look at the max. value.

4.14.3 Visualization of filters

Filters show us the kind of pattern that are going to be matched (e.g. diagonal shapes, green images, "chess board shapes", etc.

Initial layers in the CNN extract generic information about the image. The deeper we go in the CNN, the filters are trained to pick up more complex patterns. Thus, the deeper we go, the more complicated/ explicit the filters become.

4.14.4 Choice of activation function

- Several possible activation functions available
- Popular in the past: sigmoid / tanh activations
  - Differentiable!
- Currently used extensively: Rectified linear units (ReLUs)
  - Not differentiable ☹
  - Very fast to compute ☺
  - Gradients do not „vanish" (important for deep networks) ☺

4.15 Parameter/ architecture choice

Several parameters have to be chosen:

- Number and width of hidden layers
- Use convolutional/ pooling layers? How many?
- Type of activation functions
- Weight initialization
- Learning rate schedule
- Regularization method

We can use CV to compare models, but this is quite expensive.

4.16 Connection: ANNs vs Kernels

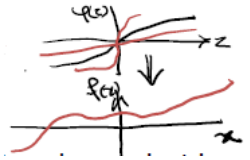
So far, we've looked at 2 approaches for learning nonlinear functions:

- Kernels: using a linear model with feature trafo and then applying the kernel trick, which allows us to solve a linear model in a very high dimensional feature space. Kernels allow us to not having to explicitly calculate the feature trafo.
- NNs: Here we explicitly specify a nonlinearity (activation function)

When using the $tanh$ function, an ANN with a single hidden layer learns functions of the form

$$f(\boldsymbol{x}) = \sum_{i=1} w_i \tanh(\boldsymbol{\theta}_i^T \boldsymbol{x})$$

-> We basically scale& shift the $tanh$ function through $\boldsymbol{\theta}_i$ and $\boldsymbol{x}$ and add all the versions up:



This is exactly the same type of function learned with kernels, when using the tanh kernel (doesn't mean we'll get the same function when training on the same data!)

$$f(\boldsymbol{x}) = \sum_{i=1} \alpha_i \underbrace{\tanh(\boldsymbol{x}_i^T \boldsymbol{x})}_{k(\boldsymbol{x}_i, \boldsymbol{x})}$$

Difference: kernels optimize over α's only -> convex

ANNs optimize over both, w's and θ_i -> non-convex

4.16.1 Example: ANN vs kernels for regression

We try to fit a sine wave.

- No noise:
  - ANN: NN with (1,) layers -> very bad, just a straight line. Increase number of neurons to (50,) -> better fit but not perfect. Add a hidden layer (50,50) -> good fit
  - SVM w/ RBF kernel: kind of OK fit but worse than NN

  -> Without noise, NNs seem to perform better
- With noise:
  - ANN: here we try to fit the noise -> overfitting -> bad model
  - SVM w/ RBF kernel: use a large bandwidth -> smooth model -> pretty good fit

In general: e.g. for very noisy data, kernel methods are much more robust since NNs are likely to model the noise -> NNs aren't always better.

Method	Kernels	ANNs
Advantages	Convex optimization, no local minima. Robust against noise. Models grow with size of data	Flexible nonlinear models, with fixed parameterization Multiple layers discover representations at „multiple levels of abstraction“
Disadvantages	Models grow with size of data ☺ Don't allow „multiple layers“	Many free parameters/ architectural choices that need to be tuned. Often suffer from very noisy data.

5 Probabilistic modeling

So far, all the seen models don't have any statistical interpretation. Often we would like to statistically model the data:

- Quantify uncertainty
- Express prior knowledge / assumptions

5.1 Statistical models for regression

Recall: Goal of supervised learning

Given training data $\mathcal{D} = \{(\boldsymbol{x_1}, y_1), \dots, (\boldsymbol{x_n}, y_n)\}$ we want to identify a Hypothesis $h: \mathcal{X} \longrightarrow \mathcal{Y}$, e.g.

- Linear regression: $h(\boldsymbol{x}) = \boldsymbol{w^T x}$
- Kernel regression: $h(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x_i}, \boldsymbol{x})$
- Neural Networks: $h(\boldsymbol{x}) = \sum_{i=1}^{k} w_i' \varphi(\boldsymbol{w_i^T x})$ (single hidden layer)

5.1.1 Minimizing the generalization error

- Fundamental assumption: Our data set is generated independently and identically distributed (iid)

$$(\boldsymbol{x_i}, y_i) \sim P(\boldsymbol{X}, Y)$$

- Would like to identify a hypothesis that minimizes the prediction error (risk)

$$R(h) = \int P(\boldsymbol{x}, y) \cdot \underbrace{l(y; h(\boldsymbol{x}))}_{\substack{\text{prediction error} \\ \text{depends on loss}}} d\boldsymbol{x}dy = \mathbb{E}_{\boldsymbol{X},Y}\left[l(y; h(\boldsymbol{x}))\right]$$

- Defined in terms of a loss function

5.1.1.1 Least-squares regression

- In least-squares regression, risk is

$$R(h) = \mathbb{E}_{\boldsymbol{X},Y}\left[\left(y - h(\boldsymbol{x})\right)^2\right]$$

- Suppose (unrealistic) we knew the data generating distribution $P(\boldsymbol{X}, Y)$

Which h minimizes the risk then? (we want an optimal hypothesis function h from all possible functions)

$$\min_h \mathbb{E}_{\boldsymbol{X},Y}\left[\left(y - h(\boldsymbol{x})\right)^2\right] = \mathbb{E}_{\boldsymbol{X} \sim \boldsymbol{P_X}} \min_{h(\boldsymbol{x})} \mathbb{E}_Y\left[\left(y - h(\boldsymbol{x})\right)^2 | \boldsymbol{x}\right]$$

-> since we can pick $h(\boldsymbol{x})$ independently from $h(\boldsymbol{x}')$ for $\boldsymbol{x} \neq \boldsymbol{x}'$

-> for the optimal hypothesis, we don't need a linear function, we can pick a h that is different for every $\boldsymbol{x}$

Given $\boldsymbol{x}$, what is the optimal prediction $y^* = h^*(\boldsymbol{x})$ (assuming we know the distribution $P(\boldsymbol{X}, Y)$)

$$y^*(\boldsymbol{x}) = \operatorname*{argmin}_{\hat{y}} \underbrace{\mathbb{E}_Y[(\hat{y} - y)^2 | \boldsymbol{x}]}_{l(\hat{y})}$$

$$l(\hat{y}) = \int (\hat{y} - y)^2 p(y|x) dy$$

$$\frac{\partial}{\partial \hat{y}} l(\hat{y}) = \int \frac{\partial}{\partial \hat{y}} (\hat{y} - y)^2 p(y|x) dy = \int 2(\hat{y} - y) p(y|x) dy = 0$$

$$\Leftrightarrow \hat{y} \underbrace{\int p(y|x) dy}_{=1} = \int y p(y|x) dy \Leftrightarrow \hat{y} = \mathbb{E}[Y|\boldsymbol{X} = \boldsymbol{x}]$$

The hypothesis $h^*(\boldsymbol{x})$ minimizing $R(h)$ is thus given by

$$\boxed{h^*(\boldsymbol{x}) = \mathbb{E}[\boldsymbol{Y}|\boldsymbol{X} = \boldsymbol{x}]}$$

This (in practice unattainable) is called the **Bayes' optimal predictor** for the squared loss.

In practice, we have finite data. Thus, the strategy for estimating a predictor from training data is to estimate the conditional distribution

$$\hat{P}(\boldsymbol{Y}|\boldsymbol{X})$$

and the, for test point **x,** predict label

$$\hat{y} = \hat{\mathbb{E}}[Y|\boldsymbol{X} = \boldsymbol{x}] = \int y \cdot \hat{P}(y|\boldsymbol{X} = \boldsymbol{x}) dy$$

5.1.1.2 Estimating conditional distributions

- Common approach: Parametric estimation
  - Choose a parametric form $\hat{P}(Y|\boldsymbol{X}, \boldsymbol{\theta})$
  - Then, optimize the parameters

-> **Maximum (conditional) Likelihood Estimation**

$$\theta^* = \operatorname*{argmax}_{\theta} \hat{P}(y_1, \dots, y_n | \boldsymbol{x_1}, \dots, \boldsymbol{x_n}, \boldsymbol{\theta}) \overset{iid}{=} \operatorname*{argmax}_{\theta} \prod_{i=1}^{n} \hat{P}(y_i | \boldsymbol{x_i}, \boldsymbol{\theta})$$

-> assume each data point $(\boldsymbol{x_i}, y_i)$ is generated iid (y_i only depends on $\boldsymbol{x_i}$)

$$\theta^* = \operatorname*{argmax}_{\theta} \log \hat{P}(y_{1:n} | \boldsymbol{x_{1:n}}, \boldsymbol{\theta}) \overset{iid}{=} \underbrace{\operatorname*{argmax}_{\theta} \sum_{i=1}^{n} \log \hat{P}(y_i | \boldsymbol{x_i}, \boldsymbol{\theta})}_{\substack{\text{this now looks like something we've} \\ \text{seen before} \rightarrow \text{sum of cost functions}}}$$

5.1.2 MLE for conditional linear Gaussian

Consider linear regression. We make the statistical assumption that the noise is gaussian;

$$y_i \sim \mathcal{N}(\boldsymbol{w^T x_i}, \sigma^2) \rightarrow y_i = \boldsymbol{w^T x_i} + \epsilon_i \,, \epsilon_i \sim \mathcal{N}(0, \sigma^2)$$

The (conditional) likelihood of the data given any candidate model **w** is:

$$-\log \hat{P}(y_{1:n} | x_{1:n}, w) = -\sum_{i=1}^{n} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \boldsymbol{w^T x_i})^2}{2\sigma^2}\right)\right) =$$

$$= \frac{n}{2}\log(2\pi\sigma^2) + \sum_{i=1}^{n} \frac{(y_i - \boldsymbol{w^T x_i})^2}{2\sigma^2}$$

We want to minimize this expression (minimize since negative log likelihood). The first term and the factor $2\sigma^2$ don't matter in minimization

-> $\boxed{\operatorname*{argmax}_{\boldsymbol{w}} P\left(y_1, \dots, y_n \middle| \boldsymbol{x_1}, \dots, \boldsymbol{x_n}, \boldsymbol{w}\right) = \operatorname*{argmin}_{\boldsymbol{w}} \sum_{i=1}^{n} (y_i - \boldsymbol{w^T x_i})^2}$

Thus, under the "conditional linear Gaussian" assumption, <u>maximizing the likelihood is equivalent to least squares estimation!!</u>

5.1.2.1 More generally: MLE for iid Gaussian noise

- Suppose $\mathcal{H} = \{h: \mathcal{X} \longrightarrow \mathbb{R}\}$ is a class of functions
- Assuming that $P(Y = y|\boldsymbol{X} = \boldsymbol{x}) = \mathcal{N}(y|h^*(\boldsymbol{x}), \sigma^2)$ for some function $h^*: \mathcal{X} \longrightarrow \mathbb{R}$ and some $\sigma^2 > 0$ the MLE for data $\mathcal{D} = \{(\boldsymbol{x_1}, y_1), \dots, (\boldsymbol{x_n}, y_n)\}$ in $\mathcal{H}$ is given by

$$\hat{h} = \operatorname*{argmin}_{h \in \mathcal{H}} \sum_{i=1}^{n} \left(y_i - h(\boldsymbol{x_i})\right)^2$$

5.1.2.2 Least-squares regression = Gaussian MLE

- The Maximum Likelihood Estimate (MLE) is given by the least squares solution, assuming that the noise is iid Gaussian with constant variance
- This is useful since MLE satisfies several nice statistical properties (not formally defined here)
  - Consistency (parameter estimate converges to true parameters in probability)
  - Asymptotic efficiency (smallest variance among all „wellbehaved“ estimators for large n)
  - Asymptotic normality
- However, all these properties are asymptotic (hold as $n \rightarrow \infty$). For finite n, we must avoid overfitting!

5.1.3 Bias through Bayesian modeling

- Can introduce bias by expressing assumptions on parameters through a Bayesian prior
- For example, let's assume $\boldsymbol{w} \sim \mathcal{N}(0, \beta^2 \boldsymbol{I}), i.e. w_i \sim \mathcal{N}(0, \beta^2)$
  -> make an a priori assumption on the weights (without any knowledge yet)
- Then, the posteriori distribution of $\boldsymbol{w}$ is given using Bayes' rule by

$$\underbrace{P(w|x_{1:n}; y_{1:n})}_{\substack{\text{joint prob.of } w \\ \textbf{given } x \text{ with } y \\ (y \text{ not given!})}} = \frac{P(w|x_{1:n}) \cdot P(y_{1:n}|x_{1:n}, w)}{P(y_{1:n}|x_{1:n})} = \frac{P(w) \cdot P(y_{1:n}|x_{1:n}, w)}{P(y_{1:n}|x_{1:n})}$$

  Assuming weights are independent from x
- Which parameters **w** are most likely a posteriori?

5.1.3.1 Maximum a posteriori estimate

$$\operatorname*{argmax}_{w} P(w|x_{1:n}; y_{1:n}) = \operatorname*{argmin}_{w} -\log P(w|x_{1:n}; y_{1:n}) =$$

$$= \operatorname*{argmin}_{w} \left(-\log P(w) - \underbrace{\log P(y_{1:n}|x_{1:n}, w)}_{\text{see before}} + \underbrace{\log P(y_{1:n}|x_{1:n})}_{\text{not depend on } w}\right)$$

$$\rightarrow -\log P(w) = -\log\left(\prod_{i=1}^{d} \mathcal{N}_{w_i}(0, \beta^2)\right) = \underbrace{-d \log \frac{1}{\sqrt{2\pi\beta^2}}}_{\text{not depend on } w} + \underbrace{\sum_{i=1}^{d} \frac{w_i^2}{2\beta^2}}_{\frac{1}{2\beta^2}||w||_2^2}$$

We get: $\operatorname*{argmax}_{w} P(w|x_{1:n}; y_{1:n}) = \operatorname*{argmin}_{w} \left(\frac{1}{2\beta^2}||w||_2^2 + \sum_{i=1}^{n} \frac{(y_i - \boldsymbol{w^T x_i})^2}{2\sigma^2}\right) =$

$$\boxed{\operatorname*{argmin}_{w} \left(\lambda||w||_2^2 + \sum_{i=1}^{n} (y_i - \boldsymbol{w^T x_i})^2\right)}$$

With $\lambda = \frac{\sigma^2}{\beta^2}$. -> This is exactly ridge regression!!

5.1.3.2 Ridge regression = MAP estimation

- Ridge regression can be understood as finding the Maximum A Posteriori (MAP) parameter estimate for a linear regression problem, assuming that
  - The noise $P(y|x,w)$ is iid Gaussian and
  - The prior $P(w)$ on the model parameters **w** is Gaussian

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \lambda ||\mathbf{w}||_2^2 \equiv \arg\max_{\mathbf{w}} P(\mathbf{w}) \prod_i P(y_i \mid \mathbf{x}_i, \mathbf{w})$$

-> With regularization term: be Bayesian about weights (for L2)
-> Without regularization: ML estimation

Note: $\lambda = \frac{\sigma^2}{\beta^2}$

➔ λ scales linearly with the variance of the noise -> large noise needs large λ (natural)

➔ λ scales inversely proportional with β^2 -> if we choose weights close together (small β^2) -> need large λ

5.1.4 Regularization vs. MAP inference

- More generally, regularized estimation can often be understood as MAP inference

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \ell(\mathbf{w}^T \mathbf{x}_i; \mathbf{x}_i, y_i) + C(\mathbf{w}) = \arg\max_{\mathbf{w}} \prod_i P(y_i \mid \mathbf{x}_i, \mathbf{w}) P(\mathbf{w})$$
$$= \arg\max_{\mathbf{w}} P(\mathbf{w} \mid D)$$

where $C(\mathbf{w}) = -\log P(\mathbf{w})$
and $\ell(\mathbf{w}^T \mathbf{x}_i; \mathbf{x}_i, y_i) = -\log P(y_i \mid \mathbf{x}_i, \mathbf{w})$

- This perspective allows changing priors (=regularizers) and likelihoods (=loss functions)

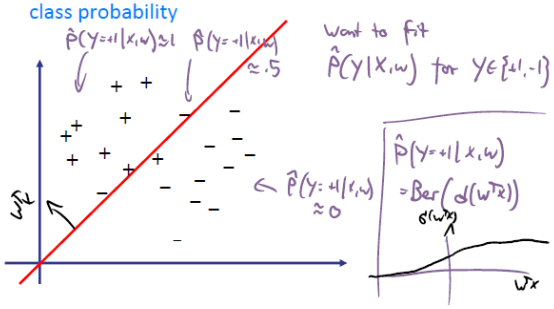
Question: Is there a prior corresponding to L1 regularization (lasso)?
Answer: The Laplace prior

5.2 Statistical models for classification

5.2.1 Logistic regression

Idea: For normal classification, we use regression, train weights w and then look at the sign of $\boldsymbol{w^T x}$. We thus "force" the algorithm to make a hard choice. However, gaussian noise is continuous -> find a new noise model for binary data, where points close to the decision boundary as less certain.

- **Idea**: Use (generalized) linear model for the class probability



Link function for logistic regression:

$$\sigma(\boldsymbol{w^T x}) = \frac{1}{1 + \exp(-\boldsymbol{w^T x})}$$

- Logistic regression (a classification method) replaces the assumption of Gaussian noise (squared loss) by iid Bernoulli noise:

$$P(y|x, \boldsymbol{w}) = Ber\big(y; \sigma(\boldsymbol{w^T x})\big)$$

- How can we estimate the parameters w?
  - Maximum Likelihood Estimation / MAP estimation

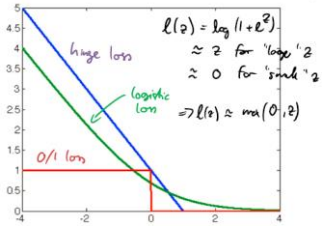
5.2.2 MLE for logistic regression

$$\hat{w} \in \operatorname*{argmax}_w P(y_{1:n}|w, x_{1:n}) = \operatorname*{argmin}_w -\log P(y_{1:n}|w, x_{1:n}) =$$
$$= \operatorname*{argmin}_w -\sum_{i=1}^{n} \underbrace{\log P(y_i|w, x_i)}_{(*)} = (**)$$
$$(*) - \log P(y_i|w, x_i) = -\log \frac{1}{1+\exp(-y_i \boldsymbol{w^T x_i})} = \log(1 + \exp(-y_i \boldsymbol{w^T x_i}))$$
$$(**) = \operatorname*{argmin}_w \underbrace{\sum_{i=1}^{n} \log(1 + \exp(-y_i \boldsymbol{w^T x_i}))}_{\hat{R}(w)}$$

The negative log likelihood (=objective) function is given by

$$\hat{R}(w) = \sum_{i=1}^{n} \log(1 + \exp(-y_i \boldsymbol{w^T x_i}))$$

The logistic loss is convex -> use convex optimization techniques (e.g. SGD)



Note: Logistic regression actually tries to be careful about label probability (tries to model it) -> SVMs don't do this (SVMs only care about making as few mistakes as possible). Logistic regression tries to estimate label noise.

Remarks on logistic regression:

- Can kernelize (kernelized logistic regression)
- Can apply logistic loss function to neural networks, in order to have them output probabilities
- Natural multi-class variants

5.2.3 Gradient for logistic regression

Loss for data point $(\boldsymbol{x}, y)$: $l(\boldsymbol{w}) = \log(1 + \exp(-y\boldsymbol{w^T x}))$
Gradient:

$$\nabla_{\boldsymbol{w}} l(\boldsymbol{w}) = \frac{1}{1 + \exp(-y\boldsymbol{w^T x})} \cdot \exp(-y\boldsymbol{w^T x}) \cdot (-y\boldsymbol{x}) =$$
$$= \frac{\exp(-y\boldsymbol{w^T x})}{1 + \exp(-y\boldsymbol{w^T x})} \cdot (-y\boldsymbol{x}) = \underbrace{\frac{1}{1 + \exp(y\boldsymbol{w^T x})}}_{\hat{P}(Y=-y|w,x)} \cdot (-y\boldsymbol{x}) =$$
$$= \underbrace{\hat{P}(Y = -y|w, x)}_{\text{large if model w is surprised by y}} \cdot (-y\boldsymbol{x})$$

$\hat{P}(Y = -y|w, x)$ is the probability of misclassification on the current model.

5.2.4 SGD for logistic regression

- Initialize **w**
- For t = 1,2,...
  - Pick data point $(\boldsymbol{x}, y)$ uniformly at random from data D
  - Compute probability of misclassification with current model

$$\hat{P}(Y = -y \mid \mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(y\mathbf{w}^T\mathbf{x})}$$

  - Take gradient step

$$\mathbf{w} \leftarrow \mathbf{w} + \eta_t \, y \, \mathbf{x} \, \hat{P}(Y = -y \mid \mathbf{w}, \mathbf{x})$$

5.2.5 Logistic regression and regularization

- Similar to SVMs and linear regression, we want to use **regularizers** to control model complexity
- Thus, instead of solving MLE

$$\min_{\boldsymbol{w}} \sum_{i=1}^{n} \log(1 + \exp(-y_i \boldsymbol{w^T x_i}))$$

estimate MAP/ solve regularized problem.

- **L2 (Gaussian prior):**

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

- **L1 (Laplace prio):**

$$\min_{\boldsymbol{w}} \sum_{i=1}^{n} \log(1 + \exp(-y_i \boldsymbol{w^T x_i})) \, \lambda ||\boldsymbol{w}||_1$$

Regularized logistic regression:

- Learning:
  - Find optimal weights by minimizing logistic loss + regularizer

$$\hat{w} = \operatorname*{argmin}_{\boldsymbol{w}} \sum_{i=1}^{n} \log(1 + \exp(-y_i \boldsymbol{w^T x_i})) + \lambda ||\boldsymbol{w}||_2^2 =$$
$$= \operatorname*{argmin}_{\boldsymbol{w}} P(\boldsymbol{w}|(\boldsymbol{x_1}, \dots, \boldsymbol{x_n}, y_1, \dots, \boldsymbol{y_n}))$$

- Classification:
  - Use conditional distribution

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

  - E.g., predict more likely class label

5.2.6 SGD for L2-regularized logistic regression

- Initialize **w**
- For t = 1,2,...
  - Pick data point $(\boldsymbol{x}, y)$ uniformly at random from data D
  - Compute probability of misclassification with current model

$$\hat{P}(Y = -y \mid \mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(y\mathbf{w}^T\mathbf{x})}$$

  - Take gradient step

$$\mathbf{w} \leftarrow \mathbf{w}(1 - 2\lambda\eta_t) + \eta_t \, y \, \mathbf{x} \, \hat{P}(Y = -y \mid \mathbf{w}, \mathbf{x})$$

5.2.7 Kernelized logistic regression

- Learning:
  - Find optimal weights by minimizing logistic loss + regularizer

$$\hat{\alpha} = \arg\min_{\alpha} \sum_{i=1}^{n} \log\left(1 + \exp\left(-y_i \alpha^T \mathbf{K}_i\right)\right) + \lambda \alpha^T \mathbf{K} \alpha$$

$\alpha \in \mathbb{R}^n$ $\quad K = \left(K_1 \mid \dots \mid K_n\right)$, $K_i = \begin{pmatrix} k(x_1, x_i) \\ \vdots \\ k(x_n, x_i) \end{pmatrix}$

- Classification:
  - Use conditional distribution

$$\hat{P}(y \mid \mathbf{x}, \hat{\alpha}) = \frac{1}{1 + \exp\left(-y \sum_{j=1}^{n} \alpha_j k(\mathbf{x}_j, \mathbf{x})\right)}$$

$\sum_{j=1}^{n} \alpha_j k(\mathbf{x}_j, \mathbf{x}) = f(x)$

  - E.g., predict more likely class label

5.2.8 Multi-class logistic regression

Remember: Multiclass SVMs -> via reduction (multiple binary classification problems) or optimize all classifiers together -> this is used here.

- Can extend logistic regression to multi-class setting
- Maintain one weight vector per class (c classes) and model

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

Observation: The whole thing is shift-invariant!

$$\frac{exp(\boldsymbol{w_i^T x} + \tau)}{\sum_{j=1}^{c} exp(\boldsymbol{w_j^T x} + \tau)} = \frac{exp(\boldsymbol{w_i^T x})}{\sum_{j=1}^{c} exp(\boldsymbol{w_j^T x})}$$

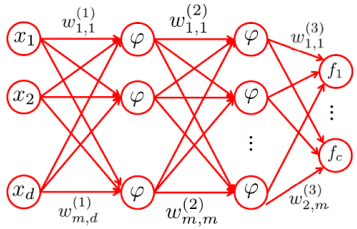
We can choose $\tau = -\boldsymbol{w_c^T x}$ -> the term from the last class c cancels out exp $(\boldsymbol{w_c^T x} - \boldsymbol{w_c^T x}) = exp(0) = 1$ -> thus we can force $\boldsymbol{w_c} = \boldsymbol{0}$ w.l.o.g.!

- The model from above is not unique -> through forcing $\boldsymbol{w_c} = \boldsymbol{0}$ we can force uniqueness!
- Corresponding loss function (cross-entropy loss):

$$l(y; \boldsymbol{x}, \boldsymbol{w_1}, \dots, \boldsymbol{w_c}) = -\log P(Y = y | \boldsymbol{x}, \boldsymbol{w_1}, \dots, \boldsymbol{w_c})$$

5.2.9 Training neural nets for multi-class

We basically have logistic regression, only that we parametrize our features using a NN



Loss: $l(Y = i; f_1, \dots, f_c) = -\log \frac{exp(f_i)}{\sum_{j=1}^{c} exp(f_j)}$

5.2.10 SVM vs. logistic regression

Method	SVM/ Perceptron	Logistic regression
Advantages	Sometimes higher classification accuracy; Sparse sol's	Can obtain class probabilities
Disadvantages	Can't (easily) get class probabilities	Dense solutions

- SVMs: directly optimize accuracy, not probability -> best class label (can lead to higher accuracy, especially for low data)
- Logistic regression: optimize over class probability

5.3 Bayesian decision theory

- So far, we've seen how we can interpret supervised learning as fitting probabilistic models of the data
- Now, we'll use the estimated models to make decisions
- Idea: assume we've estimated the conditional distribution on the labels (conditioned on the observed features) -> $P(y|x)$

5.3.1 Setup

- Given:
  - Conditional distribution over labels: $P(y|x)$, $y \in \mathcal{Y}$
  - Set of actions: $\mathcal{A}$
  - Cost function: $\mathcal{C}: \mathcal{Y} \times \mathcal{A} \longrightarrow \mathbb{R}$
- Bayesian decision theory recommends to pick the action that minimizes the expected cost

$$a^* = \underset{a \in \mathcal{A}}{\operatorname{argmin}} \mathbb{E}_y[C(y, a)|\boldsymbol{x}]$$

-> For classification: $\mathbb{E}_y[C(y, a)|\boldsymbol{x}] = \sum_y P(y|x) C(y, a)$

- If we had access to the true distribution $P(y|\boldsymbol{x})$ this decision implements the Bayesian optimal decision
- In practice, can only estimate it, e.g., (logistic) regression -> try to get close, a good estimate of $P(y|\boldsymbol{x})$

5.3.2 Example: logistic regression

- Est. cond. dist.: $P(Y = y|X = x) = Ber\left(y; \sigma(\boldsymbol{w^T x})\right)$
- Action set: $\mathcal{A} = \{+1, -1\}$
- Cost function: $C(y, a) = [y \neq a] = \begin{cases} 1, y \neq a \\ 0, else \end{cases}$

$$\mathbb{E}_Y[C(Y, a)|X = x] = \sum_{y \in \{-1,+1\}} P(y|x)[y \neq a] = P(y \neq a|x) = \frac{1}{1 + \exp(a \cdot \boldsymbol{w^T x})}$$

$$a^* \in \underset{a \in \{-1,+1\}}{\operatorname{argmin}} \mathbb{E}_Y[C(Y, a)|X = x] = \underset{a \in \{-1,+1\}}{\operatorname{argmax}} 1 + \exp(a \cdot \boldsymbol{w^T x}) = \underset{a \in \{-1,+1\}}{\operatorname{argmax}} \exp(a \cdot \boldsymbol{w^T x}) = sign(\boldsymbol{w^T x}) = a^*$$

-> Just pick the same sign as $\boldsymbol{w^T x}$ for a to maximize (a can only be ± 1)
Thus, the action that minimizes the expected cost

$$a^* = \underset{a \in \mathcal{A}}{\operatorname{argmin}} \mathbb{E}_Y[C(Y, a)|X = x]$$

Is the most likely class

$$a^* = \underset{y}{\operatorname{argmax}} \hat{P}(y|x) = sign(\boldsymbol{w^T x})$$

5.3.3 Asymmetric cost

- Est. cond. dist.: $\hat{P}(y|\boldsymbol{x}) = Ber\left(y; \sigma(\boldsymbol{w^T x})\right)$
- Action set: $\mathcal{A} = \{+1, -1\}$
- Cost function: $C(y, a) = \begin{cases} c_{FP}, if\ y = -1\ and\ a = +1 \\ c_{FN}, if\ y = +1\ and\ a = -1 \\ 0, else \end{cases}$

Define $p = P(y = +1|x)$
Then the action that minimizes the expected cost is

$$c_+ = \mathbb{E}_Y[C(Y, +1)|x] = c_{FP} P(Y = -1|x) = c_{FP} \cdot (1 - p)$$
$$c_- = \mathbb{E}_Y[C(Y, -1)|x] = c_{FN} P(Y = +1|x) = c_{FN} \cdot p$$

Thus, the action that minimizes the expected cost

$$Predict + 1 \Leftrightarrow c_+ < c_- \Leftrightarrow c_{FP} \cdot (1 - p) < c_{FN} \cdot p \Leftrightarrow p > \frac{c_{FP}}{c_{FP} + c_{FN}}$$

5.3.4 "Doubtful" logistic regression

- Est. cond. dist.: $\hat{P}(y|\boldsymbol{x}) = Ber\left(y; \sigma(\boldsymbol{w^T x})\right)$
- Action set: $\mathcal{A} = \{+1, -1, D\}$ -> D: don't know
- Cost function: $C(y, a) = \begin{cases} [y \neq a], if\ a \in \{+1, -1\} \\ c, if\ a = D \end{cases}$

Then the action that minimizes the expected cost

$$a^* = \underset{a \in \mathcal{A}}{\operatorname{argmin}} \mathbb{E}_y[C(y, a)|\boldsymbol{x}]$$

Is given by:

$$a^* = \begin{cases} y, if\ \hat{P}(y|x) \geq 1 - c \\ D, else \end{cases}$$

i.e., pick most likely class only if confident enough.

5.3.5 Example: linear regression

- Est. cond. dist.: $\hat{P}(y|\boldsymbol{x}, \boldsymbol{w}) = \mathcal{N}(y; \boldsymbol{w^T x}, \sigma^2)$
- Action set: $\mathcal{A} = \mathbb{R}$
- Cost function: $C(y, a) = (y - a)^2$ -> squared error

$$\min_a \underbrace{\mathbb{E}_y[C(y, a)|x]}_{g(a)} = \min_a \mathbb{E}_y[(y - a)^2|x]$$

-> find minima -> derive after a and set to zero

$$\frac{d}{da} g(a) = \frac{d}{da} \mathbb{E}_y[(y - a)^2|x] = \mathbb{E}_y\left[\frac{d}{da}(y - a)^2|x\right] = \mathbb{E}_y[-2(y - a)|x]$$
$$= -2\mathbb{E}_y[y|x] + 2\mathbb{E}_y[a|x] = 0$$
$$\Leftrightarrow \mathbb{E}_y[y|x] = \mathbb{E}_y[a|x] = a$$

-> $\mathbb{E}_y[a|x] = a$ since a is a constant w.r.t. x and y
Then the action that minimizes the expected cost

$$a^* = \underset{a \in \mathcal{A}}{\operatorname{argmin}} \mathbb{E}_y[C(y, a)|\boldsymbol{x}]$$

is the most likely class:

$$a^* = \mathbb{E}_y[y|x] = \int \hat{P}(y|\boldsymbol{x}) dy = \hat{\boldsymbol{w}}^T \boldsymbol{x}$$

5.3.6 Example: Asymmetric cost for regression

- Est. cond. dist.: $\hat{P}(y|\boldsymbol{x}) = \mathcal{N}(y; \hat{\boldsymbol{w}}^T\boldsymbol{x}, \sigma^2)$
- Action set: $\mathcal{A} = \mathbb{R}$
- Cost function: $C(y,a) = \underbrace{c_1 \max(y-a,0)}_{underestimation} + \underbrace{c_2 \max(a-y,0)}_{overestimation}$

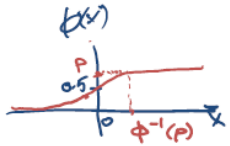
Then the action that minimizes the expected cost

$$a^* = \underset{a\in\mathcal{A}}{\operatorname{argmin}}\, \mathbb{E}_y[C(y,a)|\boldsymbol{x}]$$

is the most likely class:

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

where ϕ^{-1} is the inverse gaussian CDF.



By choosing c_1 and c_2 correctly, we can avoid over- or underestimation.

5.3.7 Active learning

Labels are expensive to get (need to ask experts, e.g. doctor). If we have unlabeled data points, which ones should we get a label for?
-> Pick points we're uncertain about, i.e. points close to the decision boundary.

5.3.7.1 Uncertainty sampling

Simple strategy: Always pick the example that we are most uncertain about
Given: Pool of unlabeled examples $D_X = \{\boldsymbol{x_1}, \dots, \boldsymbol{x_n}\}$
Also maintain labeled data set D, initially empty

- For $t = 1,2,3, \dots$
  - Estimate $\hat{P}(Y_i|\boldsymbol{x_i})$ given current data D (estimate based on labeled data)
  - Pick unlabeled example that we are most **uncertain** about
    $$i_t \in \underset{i}{\operatorname{argmin}}\left[0.5 - \hat{P}(Y_i|\boldsymbol{x_i})\right]$$
  - Query label y_i and set $D = D \cup \{(\boldsymbol{x_{i_t}}, y_{i_t})\}$

Comments:

- Active learning violates i.i.d. assumption!
- Can get stuck with bad models
- More advanced selection criteria available
  - E.g.: query point that reduces uncertainty of other points as much as possible

5.3.8 Deriving decision rules

- Bayesian decision theory provides a principled way to derive decision rules from conditional distributions
- Standard rules arise as special cases:
  - Linear regression: $\hat{\boldsymbol{w}}^T\boldsymbol{x}$
  - Logistic regression: $sign(\hat{\boldsymbol{w}}^T\boldsymbol{x})$
- Can accommodate more complex settings
  - „Doubt" (i.e., requiring sufficient confidence)
  - Asymmetric losses
  - Active learning
  - ...

5.4 Discriminative vs. Generative Modeling

Motivating example: Logistic regression can be overconfident with labels. For points that are very far from the decision boundary, logistic regression will predict ≈ 1 (the further from the decision boundary, the more confident we are). If we see a point x (far away from boundary) for which we haven't really seen any similar point yet, we really shouldn't be confident about this point.

5.4.1 Discriminative modeling

- So far, we have considered learning methods that estimate conditional distributions
  $$P(y|\boldsymbol{x})$$
- Examples: Linear regression, logistic regression, etc.
- Such models do not attempt to model $P(\boldsymbol{x})$
- Thus, they will not be able to detect outliers (i.e., „unusual" points for which P(x) is very small)

Discriminative vs. Generative Modeling

- Discriminative models aim to estimate
  $$P(y|\boldsymbol{x})$$
- Generative models aim to estimate joint distribution
  $$P(y, \boldsymbol{x})$$
- Can derive conditional from joint distribution, but not vice versa!
  $$P(y,x) \Longrightarrow P(y|\boldsymbol{x}) = \frac{P(x,y)}{P(\boldsymbol{x})} = \frac{P(x,y)}{\sum_y P(\boldsymbol{x},y)}$$
  Vice versa ($\Longleftarrow$) is not possible!

5.4.2 Generative modeling

Generative modeling attempts to infer the process, according to which examples are generated. We thus want to model $P(y, \boldsymbol{x})$. However, directly modeling $P(y, \boldsymbol{x})$ is hard.
1. Estimate prior on labels $P(y)$
2. Estimate conditional distribution $P(\boldsymbol{x}|y)$ for each class y
3. Obtain predictive distribution using Bayes' rule:

$$P(y|\boldsymbol{x}) = \frac{1}{Z}P(y)P(\boldsymbol{x}|y) = \frac{P(x,y)}{P(\boldsymbol{x})}$$

with $Z = P(\boldsymbol{x})$
Generative modeling attempts to infer the process, according to which examples are generated.
E.g.: MNIST data set

- $P(y)$: class labels: 0-9 digits -> sample and check distribution
- $P(\boldsymbol{x}|y)$: for each label y (0-9) sample the x's (e.g. pixels)

5.4.3 Example: Naive Bayes model

- Model class label as generated from categorical variable
  $$P(Y = y) = p_y\ , y \in \mathcal{Y} = \{1, \dots, c\}\ , \sum_{y=1}^{c} p_y = 1$$
- Model features as conditionally independent given Y
  $$P(\boldsymbol{X_1} = \boldsymbol{x_1}, \dots, \boldsymbol{X_d} = \boldsymbol{x_d}|Y) = \prod_{i=1}^{d} P(\boldsymbol{X_i} = \boldsymbol{x_i}|Y)$$
- I.e., given class label, each feature is „generated" independently of the other features.
  - E.g.: MNIST data set. Given class (digit) 8, the probability that pixel (10,8) is dark is independent from pixel (3,6) being dark.
    -> in reality, this is obviously **not** true!
- Need to still specify feature distributions $P(X_i|Y)$

5.4.3.1 Example: Gaussian Naive Bayes

- Model class label as generated from categorical variable
  $$P(Y = y) = p_y\ , y \in \mathcal{Y} = \{1, \dots, c\}$$
- Model features by (conditionally) independent Gaussians (we need to make a further assumption: model the conditional features as gaussian)
  $$P(x_i|y) = \mathcal{N}\left(x_i|\mu_{y,i}, \sigma^2_{y,i}\right)$$
  e.g. $\mu_{y,i}$ the mean of i-th pixel of class y (for image classification)
- How do we estimate parameters? $p_y, \mu_{y,i}, \sigma^2_{y,i}$

5.4.4 Maximum Likelihood Estimation for P(y)

Goal: $P(Y = 1) = p\ , P(Y = -1) = 1 - p$ (binary classification)
Data: $D = \{(\boldsymbol{x_1}, y_1), \dots, (\boldsymbol{x_n}, y_n)\}$

$$P(D|p) = \prod_{i=1}^{n} p^{1[y_i=+1]} \cdot (1-p)^{1[y_i=-1]} = p^{n_+} \cdot (1-p)^{n_-}$$

where n_+: # of positive examples in D, n_-: # of negative examples in D
We want to maximize this quantity -> consider log-likelihood

$$l(p) = \log P(D|p) = n_+ \log(p) + n_- \log(1-p)$$

Take derivative and set to zero:

$$\frac{d}{dp}l(p) = \frac{n_+}{p} - \frac{n_-}{1-p} = 0 \Leftrightarrow \boxed{p = \frac{n_+}{n_+ + n_-}}$$

5.4.5 Maximum Likelihood Estimation for P(x|y)

Assumed: $P(\boldsymbol{x_i}|y) = \mathcal{N}\left(\boldsymbol{x_i}; \mu_{y,i}, \sigma^2_{y,i}\right)$
Data: $D = \{(\boldsymbol{x_1}, y_1), \dots, (\boldsymbol{x_n}, y_n)\}$ -> from D we want to estimate $\mu_{i,y}, \sigma_{i,y}$
$\mu_{i,y}, \sigma_{i,y}$: $\underbrace{D_{x_i|y} = \{x\ s.t.\ x_{j,i} = x, Y_j = y\}}_{\substack{narrow\ down\ data\ set\ to\ relevant\\ instances\ where\ label\ y\ and\ narrow\\ down\ to\ i-th\ feature}}$

Then, do MLE: $\hat{\mu}_{y,i} = \frac{1}{n_y}\sum_{x\in D_{x_i|y}} x \qquad \hat{\sigma}^2_{y,i} = \frac{1}{n_y}\sum_{x\in D_{x_i|y}}\left(x - \hat{\mu}_{i,y}\right)^2$

where n_y is the # of data points belonging to class y
-> all instances for given y and x_i are generated IID -> just MLE estimate

5.4.6 Deriving decision rules

- Estimate $\hat{P}(y)$ and $\hat{P}(\boldsymbol{x}|y)$
- In order to predict label y for new point x, use
  $$P(y|\boldsymbol{x}) = \frac{1}{Z}P(y)P(\boldsymbol{x}|y), Z = \sum_y P(y)P(\boldsymbol{x}|y) = P(\boldsymbol{x})$$
- Predict using Bayesian decision theory.
- E.g., in order to minimize misclassification error, predict

$$y = \underset{y'}{\operatorname{argmax}}\, P(y'|\boldsymbol{x}) = \underset{y'}{\operatorname{argmax}} \frac{1}{Z}P(y')P(\boldsymbol{x}|y') \overset{iid}{=}$$
$$= \underset{y'}{\operatorname{argmax}}\, P(y') \prod_{i=1}^{d} P(x_i|y')$$

Take log for argmax (more numerically stable)

$$y = \underset{y'}{\operatorname{argmax}} \log P(y') + \sum_{i=1}^{d} \log P(x_i|y')$$

5.4.7 Gaussian Naive Bayes Classifier

- Learning given data $D = \{(\boldsymbol{x_1}, y_1), \dots, (\boldsymbol{x_n}, y_n)\}$
  - MLE for class prior: $\hat{P}(Y = y) = \hat{p}_y = \frac{Count(Y=y)}{n}$
  - MLE for feature distribution: $\hat{P}(\boldsymbol{x_i}|y) = \mathcal{N}(\boldsymbol{x_i}; \hat{\mu}_{y,i}, \sigma^2_{y,i})$

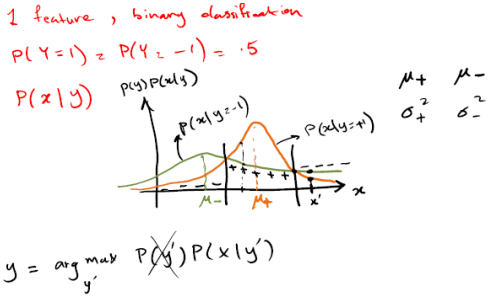
$$\hat{\mu}_{y,i} = \frac{1}{Count(Y=y)} \sum_{j:y_j=y} x_{j,i}$$

$$\hat{\sigma}^2_{y,i} = \frac{1}{Count(Y=y)} \sum_{j:y_j=y} (x_{j,i} - \hat{\mu}_{y,i})^2$$

- Prediction given new point **x**:

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

5.4.7.1 Decision boundaries (1D)



5.4.8 Decision rules for binary classification

- Want to predict $y = \underset{y'}{\operatorname{argmax}}\, P(y'|\boldsymbol{x})$
- For binary tasks (i.e. c=2, $y \in \{+1, -1\}$) this is equivalent to

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

  -> Predcit +1 if: $P(Y=+1|\boldsymbol{x}) > P(Y=-1|\boldsymbol{x})$
  -> Predcit -1 if: $P(Y=+1|\boldsymbol{x}) < P(Y=-1|\boldsymbol{x})$

- The function $f(\boldsymbol{x}) = \log \frac{P(Y=+1|\boldsymbol{x})}{P(Y=-1|\boldsymbol{x})}$ is called **discriminant function**

5.4.9 Gaussian Naive Bayes (c=2), const. variance

Given: $P(Y=1) = 0.5$ and $P(\boldsymbol{x}|y) = \prod_i \mathcal{N}(\boldsymbol{x_i}; \mu_{y,i}, \sigma_i^2)$
(i.e. assume equal class prob., class independent variance)
Why naive? Each feature has the same variance across all classes and the features are not correlated.
Want: $f(\boldsymbol{x}) = \log \frac{P(Y=+1|\boldsymbol{x})}{P(Y=-1|\boldsymbol{x})}$

$$f(\boldsymbol{x}) = \log \frac{P(Y=+1|\boldsymbol{x})}{P(Y=-1|\boldsymbol{x})} = \log \frac{\frac{P(Y=+1)P(\boldsymbol{x}|Y=+1)}{P(\boldsymbol{x})}}{\frac{P(Y=-1)P(\boldsymbol{x}|Y=-1)}{P(\boldsymbol{x})}} =$$

$$= \log \frac{0.5 \prod_i P(\boldsymbol{x_i}|Y=+1)}{0.5 \prod_i P(\boldsymbol{x_i}|Y=-1)} = \log \frac{\prod_i \mathcal{N}(\boldsymbol{x_i}; \mu_{+1,i}, \sigma_i^2)}{\prod_i \mathcal{N}(\boldsymbol{x_i}; \mu_{-1,i}, \sigma_i^2)} =$$

$$= \log \frac{\prod_i \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{1}{2\sigma_i^2}(x_i - \mu_{+1,i})^2\right)}{\prod_i \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{1}{2\sigma_i^2}(x_i - \mu_{-1,i})^2\right)} =$$

$$= -\sum_{i=1}^{d} \frac{1}{2\sigma_i^2}\left((x_i - \mu_{+1,i})^2 - (x_i - \mu_{-1,i})^2\right) =$$

$$= -\sum_{i=1}^{d} \frac{1}{2\sigma_i^2}\left(x_i^2 - 2x_i\mu_{+1,i} + \mu_{+1,i}^2 - x_i^2 + 2x_i\mu_{-1,i} - \mu_{-1,i}^2\right) =$$

$$= \sum_{i=1}^{d} \underbrace{\frac{1}{\sigma_i^2}(\mu_{+1,i} - \mu_{-1,i})}_{w_i} x_i + \underbrace{\sum_{i=1}^{d} \frac{1}{2\sigma_i^2}(\mu_{-1,i}^2 - \mu_{+1,i}^2)}_{=:w_0}$$

-> this is just a linear function!

$$f(\boldsymbol{x}) = \boldsymbol{w^T x} + w_0$$

In general: non-equal class probabilities! -> $P(Y=+1) = \hat{p}_+$
-> non-equal class probabilities just shifts around the decision boundary (w_0 is shifted)
with $w_0 = \log \frac{\hat{p}_+}{1-\hat{p}_+} + \sum_{i=1}^{d} \frac{\mu_{-1,i}^2 - \mu_{+1,i}^2}{2\sigma_i^2}$, $w_i = \frac{\mu_{+1,i} - \mu_{-1,i}}{\sigma_i^2}$

Discriminant function vs. class probability
Define $P(Y=+1|x) =: p(x)$
Then: $f(x) = \log \frac{p(x)}{1-p(x)} \Leftrightarrow \exp(f(x)) = \frac{p(x)}{1-p(x)} \Leftrightarrow p(x) = \frac{\exp(f(x))}{1+\exp(f(x))} \Leftrightarrow$
$P(Y=+1|x) = p(x) = \frac{1}{1+\exp(-f(x))} = \sigma(f(x))$

Note: If model assumptions are met, GNB will make same predictions as Logistic Regression! I.e. if $P(x, y)$ is a gaussian naïve Bayes model of the form we've discussed (with shared class variance) then the conditional distribution $P(y|x)$ is a logistic regression.
Remark: If we don't assume that the variance σ_i^2 is independent from the class y, we'd get a quadratic decision boundary.

5.4.10 Issues with Naive Bayes models

- Conditional independence assumption means that features are generated independently given class label (very naive assumption)
- If there is (conditional) correlation between class labels, then this assumption is violated
- Due to conditional independence assumption, predictions can become overconfident (very close to 1 or 0)
- This might be fine if we care about most likely class only, but not if we want to use probabilities for making decisions (e.g., asymmetric losses etc.)

5.4.11 Gaussian Bayes classifier

Gaussian Bayes classifier are much more general than Gaussian Naive Bayes classifiers.

- Model class label as generated from categorical variable
  $P(Y=y) = p_y, y \in \mathcal{Y} = \{1, \dots, c\}$
- Model features as generated by multivariate Gaussian
  $P(\boldsymbol{x}|y) = \mathcal{N}(\boldsymbol{x}; \mu_y, \Sigma_y)$

Note: For GNB we have class independent variance.
-> For GNB: $\Sigma_y = \begin{pmatrix} \sigma_{y,1}^2 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_{y,d}^2 \end{pmatrix}$

How do we estimate parameters?

5.4.11.1 MLE for Gaussian Bayes classifier

- Given data set $D = \{(\boldsymbol{x_1}, y_1), \dots, (\boldsymbol{x_n}, y_n)\}$
- MLE for class label distribution $\hat{P}(Y=y) = \hat{p}_y = \frac{Count(Y=y)}{n}$
- MLE for feature distribution $\hat{P}(\boldsymbol{x}|y) = \mathcal{N}(\boldsymbol{x}; \hat{\boldsymbol{\mu}}_y, \hat{\boldsymbol{\Sigma}}_y)$
  - $\hat{\boldsymbol{\mu}}_y = \frac{1}{Count(Y=y)} \sum_{j:y_j=y} \boldsymbol{x_j}$
  - $\hat{\boldsymbol{\Sigma}}_y = \frac{1}{Count(Y=y)} \sum_{j:y_j=y} (\boldsymbol{x_j} - \hat{\boldsymbol{\mu}}_y)(\boldsymbol{x_j} - \hat{\boldsymbol{\mu}}_y)^T$
    (-> empirical covariance matrix)

5.4.11.2 Discriminant functions for GBCs

- Given $P(Y=1) = p$ and $P(\boldsymbol{x}|y) = \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu_y}, \boldsymbol{\Sigma_y})$
- Want: $f(\boldsymbol{x}) = \log \frac{P(Y=+1|\boldsymbol{x})}{P(Y=-1|\boldsymbol{x})}$
- The discriminant function is given by:

$$f(\mathbf{x}) = \log \frac{p}{1-p} + \frac{1}{2}\left[\log \frac{|\hat{\Sigma}_-|}{|\hat{\Sigma}_+|} + \left((\mathbf{x} - \hat{\mu}_-)^T \hat{\Sigma}_-^{-1} (\mathbf{x} - \hat{\mu}_-)\right) - \left((\mathbf{x} - \hat{\mu}_+)^T \hat{\Sigma}_+^{-1} (\mathbf{x} - \hat{\mu}_+)\right)\right]$$

Now, we can capture dependencies between features since we no longer require the $\boldsymbol{\Sigma_y}$'s to be diagonal. However, we could still assume that all $\boldsymbol{\Sigma_y}$ are equal for each class (e.g. for binary classification: $\boldsymbol{\Sigma_+} = \boldsymbol{\Sigma_-}$)
➔ This leads to **LDA**

5.4.12 Fisher's linear discriminant analysis LDA (c=2)

- Suppose we fix $p = 0.5$
- Further, assume covariances are equal: $\hat{\boldsymbol{\Sigma}}_+ = \hat{\boldsymbol{\Sigma}}_- = \hat{\boldsymbol{\Sigma}}$
- Then the discriminant function:

$$f(\mathbf{x}) = \log \frac{p}{1-p} + \frac{1}{2}\left[\log \frac{|\hat{\Sigma}_-|}{|\hat{\Sigma}_+|} + \left((\mathbf{x} - \hat{\mu}_-)^T \hat{\Sigma}_-^{-1} (\mathbf{x} - \hat{\mu}_-)\right) - \left((\mathbf{x} - \hat{\mu}_+)^T \hat{\Sigma}_+^{-1} (\mathbf{x} - \hat{\mu}_+)\right)\right]$$

$p = .5$, $\hat{\Sigma}_- = \hat{\Sigma}_+$

$$= \frac{1}{2}\left[x^T\hat{\Sigma}_-^{-1}x - 2x^T\hat{\Sigma}_-^{-1}\hat{\mu}_- + \hat{\mu}_-^T\hat{\Sigma}_-^{-1}\mu_- - x^T\hat{\Sigma}_+^{-1}x + 2x^T\hat{\Sigma}_+^{-1}\hat{\mu}_+ - \hat{\mu}_+^T\hat{\Sigma}_+^{-1}\hat{\mu}_+\right]$$

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

Thus the GBC discriminant simplifies to:

$$f(\mathbf{x}) = \mathbf{x}^T\hat{\Sigma}^{-1}(\hat{\mu}_+ - \hat{\mu}_-) + \frac{1}{2}(\hat{\mu}_-^T\Sigma^{-1}\hat{\mu}_- - \hat{\mu}_+^T\Sigma^{-1}\hat{\mu}_+)$$

Which is just a linear function! $f(\boldsymbol{x}) = \boldsymbol{w^T x} + w_0$
Under these assumptions we predict

$$y = sign(f(\boldsymbol{x})) = sign(\boldsymbol{w^T x} + w_0)$$

with $\boldsymbol{w} = \hat{\boldsymbol{\Sigma}}^{-1}(\hat{\boldsymbol{\mu}}_+ - \hat{\boldsymbol{\mu}}_-)$ and $w_0 = \frac{1}{2}(\hat{\boldsymbol{\mu}}_-^T\hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{\mu}}_- - \hat{\boldsymbol{\mu}}_+^T\hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{\mu}}_+)$
This linear classifier is called Fisher's linear discriminant analysis (LDA)
We can again relax the condition of equal class probabilities -> this again just shifts around the decision boundary.

5.4.12.1 Fisher's LDA vs. Logistic regression

- Fisher's LDA uses the discriminant function $f(\boldsymbol{x}) = \log\frac{p(Y=+1|\boldsymbol{x})}{p(Y=-1|\boldsymbol{x})}$
- Again, we can derive the class distribution from $f(\boldsymbol{x})$:

$$P(Y=1|\boldsymbol{x}) = \frac{1}{1+\exp(-f(\boldsymbol{x}))} = \sigma(\boldsymbol{w}^T\boldsymbol{x} + w_0)$$

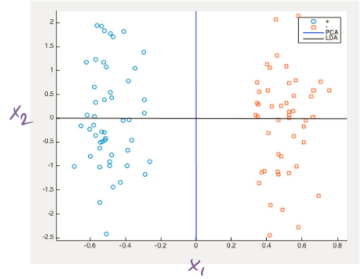
- This is the same form as logistic regression!
- If model assumptions are met, LDA will make same predictions as Logistic Regression!

5.4.12.2 Fisher's LDA vs. PCA

- LDA can be viewed as a projection to a 1-dim. subspace that maximizes ratio of between-class and within-class variances
- In constrast, PCA (k=1) maximizes the variance of the resulting 1-dim. projection

Remember PCA: The problem of minimizing the reconstruction error (PCA) is the same as finding the eigendirection in which the data has maximal variance. The principal component is the direction (the line) in $\mathbb{R}^d$ that is maximally aligned with the data.
-> LDA can be viewed in a similar manner (supervised dimension reduction)
-> the projection we pick is the one that maximizes the ratio of between-class and in-class variances.

Example:
Both classes are alongated along the x_2 axis. Classes are spread apart. PCA (doesn't know about labels) just finds the direction on which the data has maximal variance -> blue line -> PCA projects to blue line -> all points are on top of each other -> we lose all information for classification.



LDA gives us the black line (LDA maximizes the variance **between** classes and at the same time minimizes the variance **within** classes)

➔ LDA is a way of supervised PCA

5.4.13 Quadratic discriminant analysis

In the general case, covariance matrices are different for classes (unlike the LDA assumption). We then get:

$$f(\mathbf{x}) = \log\frac{p}{1-p} + \frac{1}{2}\left[\log\frac{|\hat{\Sigma}_-|}{|\hat{\Sigma}_+|} + \left((\mathbf{x}-\hat{\mu}_-)^T\hat{\Sigma}_-^{-1}(\mathbf{x}-\hat{\mu}_-)\right) - \left((\mathbf{x}-\hat{\mu}_+)^T\hat{\Sigma}_+^{-1}(\mathbf{x}-\hat{\mu}_+)\right)\right]$$

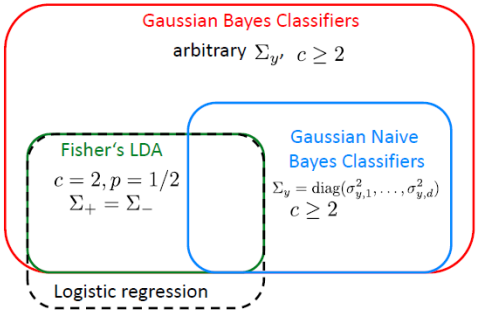
And we predict $y = sign(f(\boldsymbol{x}))$

5.4.14 Outlier detection

- As generative model, GBCs model $P(\boldsymbol{X}, Y)$
- Can identify when data points $\boldsymbol{X}$ are outliers (very unlikely) by $P(\boldsymbol{x}) \geq \tau$
- How do we get P(**x**) from $P(\boldsymbol{x}, y)$?

$$P(\boldsymbol{x}) = \sum_{y=1}^{c} P(y)\cdot P(\boldsymbol{x}|y) \overset{for\ GBC}{=} \sum_{y} \hat{p}_y \mathcal{N}(\boldsymbol{x}|\hat{\mu}_y, \hat{\Sigma}_y)$$

5.4.15 Big picture



5.4.15.1 Fisher's LDA vs. Logistic regression

- Fisher's LDA
  - Generative model, i.e., models $P(\boldsymbol{X}, Y)$
  - Can be used to detect outliers: $P(\boldsymbol{X}) < t$
  - Assumes normality of $\boldsymbol{X}$
  - not very robust against violation of this assumption
- Logistic regression
  - Discriminative model, i.e., models $P(Y|\boldsymbol{X})$ only
  - Cannot detect outliers
  - Makes no assumptions on $\boldsymbol{X}$
  - More robust

5.4.15.2 Gaussian Naive Bayes vs. General GBCs

- Gaussian Naive Bayes models
  - Conditional independence assumption may lead to overconfidence
  - Predictions might still be useful
  - #parameters = $O(c \cdot d)$
  - Complexity (memory + inference) linear in d
- General Gaussian Bayes models
  - Captures correlations among features
  - Avoids overconfidence
  - #parameters = $O(c \cdot d^2)$
  - Complexity quadratic in d

5.4.16 Discrete features

- So far, we've assumed $\boldsymbol{x} \in \mathbb{R}^d$
- Suppose some $\boldsymbol{X_i}$ take discrete values
  - Gender, Nationality, etc.

  -> might not make sense to model these features gaussian
- Generative models allow to easily swap different distributions. E.g., model $P(\boldsymbol{X_i}|Y)$ as
  - Bernoulli, Categorial, Multinomial

5.4.17 Categorical Naive Bayes classifier

- Model class label as generated from categorical variable

$$P(Y=y) = p_y, y \in \mathcal{Y} = \{1, \ldots, c\}$$

- Model features by (conditionally) independent categorical random variables

$$P(\boldsymbol{X_i} = \boldsymbol{x}|Y=y) = \theta_{\boldsymbol{x}|y}^{(i)}$$

  Fulfilling $\forall i, x, y$: $\theta_{\boldsymbol{x}|y}^{(i)} \geq 0$ and $\forall i, x, y$: $\sum_{x=1}^{c} \theta_{\boldsymbol{x}|y}^{(i)} = 1$

5.4.18 MLE for Categorical Naive Bayes classifier

- Given data set $D = \{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_n}, y_n)\}$
- MLE for class label distribution $\hat{P}(Y=y) = \hat{p}_y = \frac{Count(Y=y)}{n}$
- MLE for feature distribution $\hat{P}(\boldsymbol{X_i} = \boldsymbol{x}|y) = \theta_{\boldsymbol{x}|y}^{(i)}$

$$\theta_{\boldsymbol{x}|y}^{(i)} = \frac{Count(\boldsymbol{X_i} = \boldsymbol{x}, Y=y)}{Count(Y=y)}$$

- Prediction given new point $\boldsymbol{x}$:

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

$$= \underset{y'}{\mathrm{argmax}} \log\hat{p}_y + \sum_{i=1}^{d} \log\theta_{\boldsymbol{x}|y}^{(i)}$$

Beyond categorical Naive Bayes:

- We could in principle lift the Naïve Bayes assumption by modeling joint (conditional) distribution of the features: $P(X_1, \ldots, X_d|Y)$
- Issue: We'd need to specify a probability for each combinatorial combination of the vector $(Y, X_1, X_2, \ldots, X_d)$ -> $O(2^{d+1})$
- This would be
  - Computationally intractable
  - Fantastic way to overfit

5.4.19 Discrete and continuous features together?!

- The (Naive) Bayes classifier does not require each feature to follow the same type of conditional distribution
- For example, can model some features as Gaussian, and some others as categorical
- Training (MLE) and prediction remains the same!
- E.g.: data vector $\boldsymbol{X} = (X_1, \ldots, X_{20})$
  $\boldsymbol{X_{1:10}}$ discrete: $for\ 1 \leq i \leq 10$: $P(X_i|Y) = Categorical(X_i|Y, \theta)$
  $\boldsymbol{X_{11:20}}$ continuous: $for\ 11 \leq i \leq 20$: $P(X_i|Y;\theta) = \mathcal{N}(x_i; \mu_{i|y}, \sigma_{i|y}^2)$

$$P(\boldsymbol{X_{1:20}}|Y) = \prod_{i=1}^{10} Categorical(X_i|Y, \theta) \prod_{i=11}^{20} \mathcal{N}(x_i; \mu_{i|y}, \sigma_{i|y}^2)$$

5.4.20 Avoiding overfitting

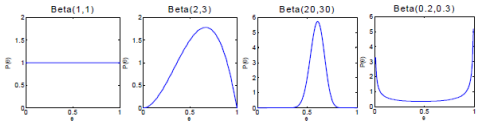
- So far we always used Maximum Likelihood Estimation -- didn't we say that's prone to overfitting?
- Can avoid overfitting by
  - Restricting model class (e.g., assumptions on covariance structure, e.g., Gaussian Naive Bayes) -> fewer parameters
  - Using priors -> „smaller" parameters

5.4.20.1 Prior over parameters (c=2)

- As prior for our class probabilities, have assumed $P(Y=1)=\theta$
- Maximum likelihood estimate: $\hat{\theta}=\frac{Count(Y=1)}{n}$
- What happens in the extreme case $n=1$ (i.e. one training data point)
  $$\mathcal{D}=\{(y_1)\}, y_1=1$$
  Then: $\hat{\theta}=\frac{Count(Y=1)}{n}=\frac{1}{1}=1$
  Thus the model estimates: $P(Y=1)=\hat{\theta}_1=1$
  $P(Y=-1)=\hat{\theta}_{-1}=1-\hat{\theta}_1=0$
  -> This is a very bad model!
- We may want to put a **prior distribution $P(\theta)$** and compute posteriori distribution
  $$P(\theta|y_1,\dots,y_n)=\frac{1}{Z}P(\theta)P(y_{1:n}|\theta), Z=\int P(\theta)P(y_{1:n}|\theta)d\theta$$

5.4.20.2 Beta prior over parameters

$$\text{Beta}(\theta;\alpha_+,\alpha_-)=\frac{1}{B(\alpha_+,\alpha_-)}\theta^{\alpha_+-1}(1-\theta)^{\alpha_--1}$$



Assume obs data D with n_+ times $y=+1$, n_- times $y=-1$

$P(\theta|D)=\frac{1}{Z}P(\theta)P(D|\theta)=\frac{1}{Z'}\theta^{\alpha_+-1}(1-\theta)^{\alpha_--1}\cdot\theta^{n_+}(1-\theta)^{n_-}$

$=\frac{1}{Z'}\theta^{\alpha_++n_+-1}(1-\theta)^{\alpha_-+n_--1}=\text{Beta}(\theta;\alpha_++n_+,\alpha_-+n_-)$

α_+&α_- are the regularization parameters

5.4.20.3 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
- Example: Beta priors and Binomial likelihood
  - Prior: $Beta(\theta;\alpha_+,\alpha_-)$
  - Observations: Suppose we observe n_+ positive and n_- negative labels
- Posterior: $Beta(\theta;\alpha_++n_+,\alpha_-+n_-)$
- Thus α_+,α_- act as „pseudo-counts“
- **MAP estimate:**

$$\hat{\theta}=\underset{\theta}{\text{argmax}}\,P(\theta|y_1,\dots,y_n;\alpha_+,\alpha_-)=\frac{\alpha_++n_+-1}{\alpha_++n_++\alpha_-+n_--2}$$

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

- Can use conjugate priors as regularizers
- (Almost) no computational cost
- How to choose hyperparameters??
  - crossvalidation ☺

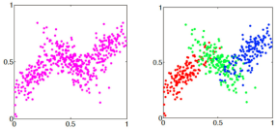
5.5 Dealing with missing data

If we want to do discriminative modeling -> $P(y|x)$
If we don't have an $\boldsymbol{x}$ (or incomplete $\boldsymbol{x}$), we can't do discriminative modeling straightforwardly. However, if we model $P(x,y)$, we could try to fill in the blanks -> generative models can work better.

Possible missing data setups:

- Certain features of certain data points are not available (N/A)
- Certain data points don't have a label (missing label)

We'll focus on the **missing label case!**



Missing labels: We see data like that:
How can we make interferences about the labels? -> What if we assume $P(X|Y)$ is Gaussian?
-> With clustering, we represent clusters with centroids. Now we want to do this with a probabilistic approach
-> make assumptions on structure of model (i.e. on $P(X|Y)$)

Note: Training a Gaussian Bayes Classifier *without labels*
Assume $P(X,Y)$ is a GB classifier: $P(Y=y)=p_y$, $P(\boldsymbol{x}|y)=\mathcal{N}(\boldsymbol{x},\mu_y,\Sigma_y)$
-> Gaussian distribution of each **x** for each class y -> i.e. model each cluster
We only get to see $\boldsymbol{x}=[x_1,\dots,x_d]$. What is $P(\boldsymbol{x})$?

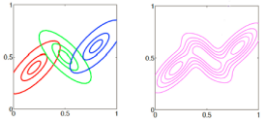
$$P(\boldsymbol{x})=\sum_y P(\boldsymbol{x},y)=\sum_y P(y)P(\boldsymbol{x}|y)=\sum_y p_y\mathcal{N}(\boldsymbol{x},\mu_y,\Sigma_y)$$

5.5.1 Gaussian Mixtures

Convex-combination of Gaussian distributions

$$\boxed{P(\boldsymbol{x}|\theta)=P(\boldsymbol{x}|\mu,\Sigma,\boldsymbol{w})=\sum_i w_i\mathcal{N}(\boldsymbol{x};\mu_i,\Sigma_i)}$$

Where $w_i\geq 0$ and $\sum_i w_i=1$
The level sets of the single Gaussians and the following Gaussian mixture are shown:



5.5.2 Mixture modeling

- Model **each cluster** as a probability distribution
  $$P(\boldsymbol{x}|\theta_j)$$
- Assuming data is sampled iid., likelihood of data is (assuming all data points are independent -> can just multiply all distribution $P(\boldsymbol{x}|\theta)$)
  $$P(D|\theta)=\prod_{i=1}^{n}\underbrace{\sum_{j=1}^{k}w_jP(\boldsymbol{x_i}|\theta_j)}_{P(\boldsymbol{x_i}|\boldsymbol{\theta})}$$
  n: number of data points, k: number of classes
- Choose parameters to minimize negative log likelihood
  $$L(D;\theta)=-\sum_i\log\left(\sum_j w_jP(\boldsymbol{x_i}|\theta_j)\right)$$

5.5.3 Fitting a mixture model

1. We have to estimate how many classes we want to use -> k
2. For each class $y\in\{1,\dots,k\}$ we have to estimate a weight w_j (class probability $P(Y=j)=p_j=w_j$). We also have to estimate the mean μ_j and the covariance matrix Σ_j for each class.

In total have:

- k weights w_j
- k mean vectors μ_j
- k covariance matrices Σ_j

-> these fully specify the model
The optimization problem then is:

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

Challenge in optimization: The likelihood is difficult to optimize directly due to the sum inside the logarithm.

MLE for Gaussian mixtures:
We would like to minimize

$$L(w_{1:k},\mu_{1:k},\Sigma_{1:k})=-\sum_{i=1}^{n}\log\sum_{j=1}^{k}w_j\mathcal{N}(\boldsymbol{x_i}|\boldsymbol{\mu_j},\boldsymbol{\Sigma_j})$$

$\boldsymbol{\mu_j}$ are completely free parameters, Σ_j must be positive definite,
w_j have constraints: $w_j\geq 0$ and $\sum_j w_j=1$
This optimization is nonconvex. We could still apply SGD. However, we must fulfill the constraints -> might be difficult with SGD

➔ We'll look at another possibility

Recall: GMMs as Generative model
To generate a data point i

- Generate cluster index z_i s.t. $P(z_i=j)=w_j$
- Then generate $\boldsymbol{x_i}$ from $\mathcal{N}(\boldsymbol{x_i}|\boldsymbol{\mu_{z_i}},\boldsymbol{\Sigma_{z_i}})$

5.5.4 GMMs vs. Gaussian Bayes Classifiers

- The joint distribution $P(z,\boldsymbol{x})=w_z\mathcal{N}(\boldsymbol{x}|\mu_z,\Sigma_z)$ of (cluster_index, features) is identical to the generative model used by the Gaussian Bayes Classifier
- **Main difference**: In contrast to GBCs, in GMMs the (label) variable z is unobserved!
- Fitting a GMM = Training a GBC without labels
- Clustering = latent variable modeling
- If we could just get the labels, could compute MLE in closed form!

5.5.5 Note: Constrained GMMs

We can constrain GMMs by putting constrains on Σ

- Spherical (level sets are spheres and σ_j give the radius of the sphere)
  $$\Sigma_j = \sigma_j^2$$
  -> # of parameters = k (very strong restriction)
- Diagonal (each feature has its own variance but we assume all features to be conditionally independent)
  $$\Sigma_j = \begin{pmatrix} \sigma_{j,1}^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_{j,d}^2 \end{pmatrix}$$
  -> # of parameters = $k * d$
  -> this is exactly Gaussian Naive Bayes
- Tied (all Σ_j's are equal but otherwise completely free)
  $$\Sigma_1 = \Sigma_2 = \dots = \Sigma_k$$
  -> # of parameters = $\frac{d(d+1)}{2}$ (matrices must be symmetric)
  -> for $k = 2$ this is Fisher's LDA
- Full (No constraints on $\Sigma_1, \Sigma_2, \dots, \Sigma_k$
  -> # of parameters = $k\frac{d(d+1)}{2}$

This is done by tying together parameters in MLE.

5.5.6 Hard-EM

Why hard? -> do a hard (deterministic) assignment for each data point (to which cluster does it belong)

Algorithm:

- Initialize the parameters $\theta^{(0)}$
- For $t = 1,2, \dots$
  - **E-step**: Predict most likely class for each data point.
    $$z_i^{(t)} = \underset{z}{\operatorname{argmax}} P\left(z|\boldsymbol{x_i}, \theta^{(t-1)}\right)$$
    $$\overset{(*)}{=} \underset{z}{\operatorname{argmax}} P\left(z|\theta^{(t-1)}\right) P\left(\boldsymbol{x_i}|z, \theta^{(t-1)}\right)$$
  - Now we got complete data!
    $$D^{(t)} = \left\{\left(\boldsymbol{x_1}, z_1^{(t)}\right), \dots, \left(\boldsymbol{x_n}, z_n^{(t)}\right)\right\}$$
  - **M-step**: Compute MLE as for the Gaussian Bayes classifier
    $$\theta^{(t)} = \underset{\theta}{\operatorname{argmax}} P\left(D^{(t)}|\theta\right)$$

(*) $P\left(z|\boldsymbol{x_i}, \theta^{(t-1)}\right) = \frac{P\left(z|\theta^{(t-1)}\right) P\left(\boldsymbol{x_i}|z, \theta^{(t-1)}\right)}{\underbrace{P\left(\boldsymbol{x_i}|\theta^{(t-1)}\right)}_{doesn't\ depend\ on\ z}}$

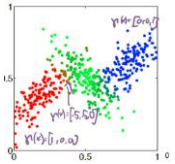
This template can be widely used for different models (once we have labels, we can do lots of computations/ estimations).

Problems with hard EM:

- Points are assigned a fixed label, even though the model is uncertain (e.g. data point in overlapping region of two distributions -> point may want to belong to both clusters with equal probability -> this can't really be modeled with hard EM)
- Intuitively, this tries to extract «too much information» from a single point
- In practice, this may work poorly if clusters are overlapping (more later)

5.5.7 Posterioir probabilities

- Suppose we are given a model $P(z|\theta), P(\boldsymbol{x}|z, \theta)$
- Then, for each data point, we can compute a posteriori distribution over cluster membership
  ➔ This means inferring distributions over latent (hidden) variables z
- Conditional probability of point **x** belonging to class j:
  $$\gamma_j(\boldsymbol{x}) = P(Z = j|\boldsymbol{x}, \Sigma, \mu, \boldsymbol{w}) = \frac{w_j P(\boldsymbol{x}|\Sigma_j, \mu_j)}{\sum_l w_l P(\boldsymbol{x}|\Sigma_l, \mu_l)}$$
  Where Σ_l, μ_l are the found model parameters for class l.
  The γ_j's are sometimes called **responsibilities** and need to fulfill:
  $$\sum_j \gamma_j(\boldsymbol{x}) = 1$$



This approach leads to probabilistic class priors. We no longer make a hard decision on whether a certain point belongs to one class or another. Rather, we assign each point c probabilities (one for each class), where $\gamma_j(\boldsymbol{x})$ is the probability of **x** belonging to j.
Each point **x** then has a class probability vector. Here (example to the right) the vector is $p(\boldsymbol{x}) = [\gamma_R(\boldsymbol{x}), \gamma_G(\boldsymbol{x}), \gamma_B(\boldsymbol{x})]$ where R: red, G: green, B: blue

5.5.8 Maximum Likelihood Estimation

We can show that at MLE:

$$(\mu^*, \Sigma^*, w^*) = \operatorname{argmin} -\sum_i \log \sum_{j=1}^k w_j \mathcal{N}\left(\boldsymbol{x_i}|\mu_j, \Sigma_j\right)$$

It must hold that (must hold for global optimum)

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

$$\Sigma_j^* = \frac{\sum_{i=1}^n \gamma_j(\boldsymbol{x_i})(\boldsymbol{x_i} - \mu_j^*)(\boldsymbol{x_i} - \mu_j^*)^T}{\sum_{i=1}^n \gamma_j(\boldsymbol{x_i})}$$

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

These equations are coupled -> difficult to solve!

5.5.9 Expectation-Maximization (Soft-EM)

We basically solve a complete data MLE problem with the only difference that our points are now weighted fractionally:

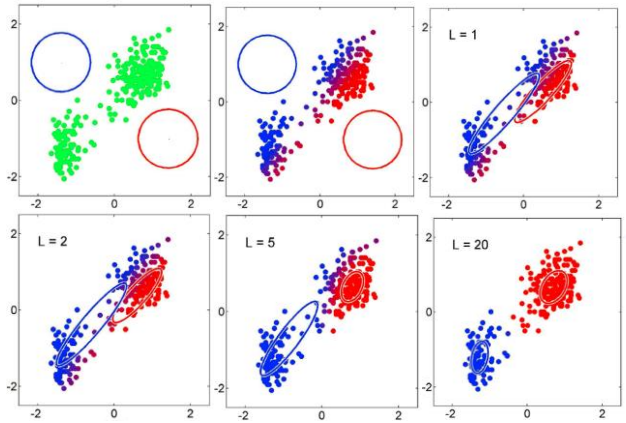
Algorithm:

- While not converged
  - **E-step:** calculate cluster membership weights ("Expected sufficient statistics") for each point (aka "responsibilities"):
    Calculate $\gamma_j^{(t)}(\boldsymbol{x_i})$ for each i and j given estimates of $\mu^{(t-1)}, \Sigma^{(t-1)}, \boldsymbol{w}^{(t-1)}$ from previous iteration.
  - **M-step:** Fit clusters to weighted data points (closed form Maximum likelihood solution!!)

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

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

5.5.9.1 Example: Soft-EM iterations



We see that during the first iteration, we just have some gaussian distributions -> initial guess. Then, after the E-step, we calculated the responsibilities and we know for each point, how likely it is to belong to either of the two (initial estimate) classes.
We then re-estimate the two gaussian distributions (M-step) using the given formulas. This is repeated.

5.5.10 Soft EM vs Hard EM

- The EM algorithm we discussed uses soft assignments to clusters, and is hence called «Soft EM»
- The intuitive first attempt we discussed first uses hard assignments, and is called «Hard EM»
- In general, Soft EM will typically result in higher likelihood values -> Soft-EM generally works better
- Reason: It can deal better with «overlapping clusters»
- The term EM alone typically refers to Soft EM

5.5.11 K-Means vs EM for GMM

Consider Hard EM with uniform weights and spherical covariance
-> Suppose $w_1 = w_2 = \dots = w_k$, $\Sigma_1 = \dots = \Sigma_k = \sigma^2 I$
Then, consider the Hard-EM algorithm:

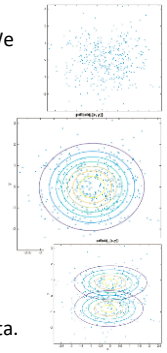
- **E-step:** In iteration t, for point i
  $$z_i^{(t)} = \underset{z}{\operatorname{argmax}} P\left(z|\theta^{(t-1)}\right) = \underset{z}{\operatorname{argmax}}\, w_z \underbrace{\mathcal{N}\left(\boldsymbol{x_i}; \boldsymbol{\mu_z^{(t-1)}}, \sigma^2 I\right)}_{\propto \exp\left(-||x_i - \mu_z^{(t-1)}||/2\sigma^2\right)} =$$
  $$= \underbrace{\underset{z}{\operatorname{argmin}} ||\boldsymbol{x_i} - \boldsymbol{\mu_z^{(t-1)}}||}_{(**)}$$
- **M-step:** for MLE
  $$\underbrace{\mu_j^{(t)} = \frac{1}{n_j}\sum_{i:z_i^{(t)}=j} \boldsymbol{x_i}}_{(**)}$$

(**) we estimate the class of point i to be the class to which center (here expectation vector) it's closest to. We then update the center of class j (expectation vectors) as the average of all points belonging to class j.
-> This is exactly k-Means!

- Can understand k-Means Algorithm (Lloyd's heuristic) as special case of Hard-EM for GMM:
  - Uniform weights over mixture components
  - Assuming identical, spherical covariance matrices
- Can also understand k-Means Algorithm as limiting case of Soft-EM for GMM:
  - Assumptions same as above, with additionally variances tending to 0

    -> Why? As $\sigma^2 \to 0$ it holds that $\gamma_j(x) = \begin{cases} 1, if\ \mu_j\ is\ closest\ to\ x \\ 0, otherwise \end{cases}$

5.5.12 Practical issues with Hard-EM

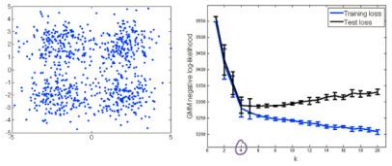


Suppose we sampled a data set from a gaussian mixture with a single gaussian (i.e. just a gaussian distribution). We now use a GMM with more than one clusters, e.g. k=2 clusters. In theory, every GMM is able to model a single gaussian (just set all weights except for one to zero).
We now fit a GMM with k=2 using Hard and Soft EM:

- Soft-EM: Soft-EM recognizes that there is only one genrating gaussian distribution!
- Hard-EM: With Hard-EM, we get two gaussian distributions. The reason for this is the "pushing away" effect -> each data point has to decide hard to which cluster to belong, thus we get a separation of the data.

Selecting k

With GMMs, we thus have a similar problem as with Clustering: we need to select the number of gaussian k in the mixture model.
We can generally use the same techniques as in clustering. However, in contrast to k-Means, cross-validation works fairly well for GMMs.
-> If the data was truly generated from a GMM, cross-validation works well can can be used to select k:



5.5.13 Degeneracy of GMMs

- Suppose we are given a single data point. What is the optimal log-likelihood that can be achieved? (1D)

$$-\log P(x|\mu,\sigma) = \underbrace{\frac{1}{2}\log(2\pi\sigma^2)}_{\to -\infty\ as\ \sigma^2 \to 0} + \frac{1}{2\sigma^2}\underbrace{(x-\mu)^2}_{=0\ if\ x=\mu}$$

  Loss converges to $-\infty$ as $\mu = x$ and $\sigma^2 \to 0$
- Thus, an optimal GMM chooses $k = n$ and puts one gaussian around each training data point with variance tending to 0
  ➔ Overfitting! This explains poor test set log-likelihood

-> just fit a single gaussian to this single data point. To minimize, we pick:
- pick mean $\mu = x$ (intuitive)
- $P(x|\mu,\sigma)$ keeps increasing as $\sigma \to 0$ -> pick $\sigma = 0$

This is then how overfitting happens! Fitting lots of gaussians for few data points
Overfitted model: unless a data point is very close to an existing training point, the model will attach ≈ 0 density to it, hence $-log$ is large -> large error

Avoiding degeneracy in GMMs

We can avoid variances tending to 0 by simply adding a small term to the diagonal of the MLE:

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

This pushes the variances away from 0! Can be motivated from a Bayesian standpoint: This is equivalent to placing a (conjugate) Wishart prior on the covariance matrix, and computing the MAP instead of MLE to regularize. Can choose ν by cross-validation.

5.5.14 Why are mixture models useful?

- Can encode assumptions about "shape" of clusters
  - E.g., fit ellipses instead of points
- Can be part of more complex statistical models
  - E.g., classifiers (or more general probabilistic models)
- Probabilistic models can output likelihood P(x) of a point x
  - Useful for anomaly/outlier detection
- Can be naturally used for semi-supervised learning!

5.5.15 Gaussian-Mixture Bayes classifier

Often times, data points of class aren't close by, maybe they're separated by points belonging to different classes.
Solution: just model **each** class as a GMM!

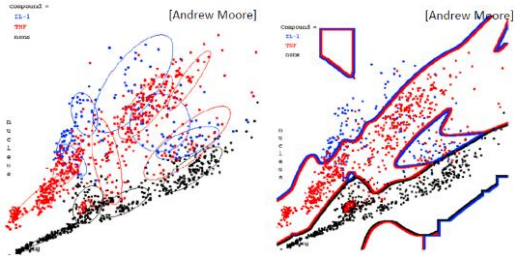
- Given labeled data set $\mathcal{D} = \{(\boldsymbol{x_1}, y_1), \dots, (\boldsymbol{x_n}, y_n)\}$
  - Label $y_i \in \{1, \dots, m\}$
  - Estimate class prior $P(y)$
  - Estimate conditional distribution for each class

$$P(\boldsymbol{x}|y) = \sum_{j=1}^{k_y} w_j^{(y)} \mathcal{N}(\boldsymbol{x}; \mu_j^{(y)}, \Sigma_j^{(y)})$$

    as Gaussian mixture model.
- For classification, we then have (using Bayes rule):

$$P(y|\boldsymbol{x}) = \frac{1}{P(\boldsymbol{x})} P(y) \sum_{j=1}^{k_y} w_j^{(y)} \mathcal{N}(\boldsymbol{x}; \mu_j^{(y)}, \Sigma_j^{(y)})$$

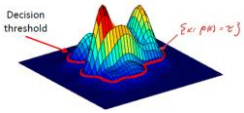
Example:



5.5.16 GMMs for density estimation

- We may be interested in fitting a Gaussian Mixture Model not for clustering but for density estimation
- E.g.,
  - Model P(x) as Gaussian mixture
  - Model P(y|x) using logistic regression, neural network, etc.
- Combines the advantage of accurate predictions / robustness from discriminative model, with the ability to detect outliers!

-> **Anomaly detection:**



Can determine outliers by comparing the estimated density of a data point against a threshold -> how to pick threshold?
If we have no examples, this is hard. If we do have examples, we vary the threshold trades FPs and FNs, use precision-recall / ROC curves as evaluation criterion, e.g. maximize F1 score. This then allows to optimize the threshold e.g. via cross-validation.

5.5.17 Theory behind the EM algorithm

Can show that the EM algorithm is equivalent to the following procedure:

- E-Step: Calculate the expected complete data loglikelihood (= function of θ)
  $$Q(\theta;\theta^{(t-1)}) = \mathbb{E}_{\mathbf{z}_{1:n}}[\log P(\mathbf{x}_{1:n},\mathbf{z}_{1:n}|\theta)|\mathbf{x}_{1:n},\theta^{(t-1)}]$$
  -> since we don't have $P(\mathbf{x}_{1:n},\mathbf{z}_{1:n}|\theta)$, we look at all possible estimations
- M-Step: Maximize $\theta^{(t)} = \underset{\theta}{\operatorname{argmax}}\, Q(\theta;\theta^{(t-1)})$

Details:

- EM objective function
  $$Q(\theta;\theta^{(t-1)}) = \mathbb{E}_{\mathbf{z}_{1:n}}[\log P(\mathbf{x}_{1:n},\mathbf{z}_{1:n}|\theta)|\mathbf{x}_{1:n},\theta^{(t-1)}]$$
  can be simplified: (for GMMs!!)

$$Q(\theta;\theta^{(t-1)}) \overset{iid}{=} \mathbb{E}_{\mathbf{z}_{1:n}}\left[\sum_{i=1}^{n}\log P(\mathbf{x}_i,\mathbf{z}_i|\theta)\,|\mathbf{x}_{1:n},\theta^{(t-1)}\right] \overset{\text{linearity of expectation}}{=}$$

$$\overset{\text{only } z_i \text{ appears in expectation}}{=} \sum_{i=1}^{n}\mathbb{E}_{\mathbf{z}_{1:n}}[\log P(\mathbf{x}_i,\mathbf{z}_i|\theta)\,|\mathbf{x}_{1:n},\theta^{(t-1)}] \overset{=}{=}$$

$$= \sum_{i=1}^{n}\mathbb{E}_{\mathbf{z}_i}[\log P(\mathbf{x}_i,\mathbf{z}_i|\theta)\,|\mathbf{x}_i,\theta^{(t-1)}] =$$

$$= \sum_{i=1}^{n}\sum_{j=1}^{k}\underbrace{P(z_i=j|\mathbf{x}_i,\theta^{(t-1)})}_{\gamma_j(\mathbf{x}_i)}\log\underbrace{P(\mathbf{x}_i,z_i=j|\theta)}_{p_{z_i=j}\mathcal{N}(x_i,\mu_j,\Sigma_j)\ \text{for GMM}}$$

Thus, the EM objective function is equivalent to:

$$Q(\theta;\theta^{(t-1)}) = \sum_{i=1}^{n}\sum_{j=1}^{k}\gamma_{z_i}(\mathbf{x}_i)\log P(\mathbf{x}_i,z_i|\theta)$$

where $\gamma_z(\mathbf{x}_i) = P(z|\mathbf{x}_i,\theta^{(t-1)})$

Thus, the E-step is equivalent to computing $\gamma_z(\mathbf{x}_i)$ (called expected sufficient statistics).

Details for M-step:

- In M-step, we need to compute
  $$\theta^{(t)} = \underset{\theta}{\operatorname{argmax}}\, Q(\theta;\theta^{(t-1)})$$
- Note similarity to MLE in Gaussian Bayes Classifiers:
  $$\theta^{*} = \underset{\theta}{\operatorname{argmax}}\sum_{i=1}^{n}\log P(\mathbf{x}_i,z_i|\theta)$$
- Thus, each iteration in the M-step is equivalent to training a GBC with weighted data!
- Closed form solution (given in EM pseudo-code)

5.5.18 Convergence of EM algorithm

- Can prove that the EM Algorithm monotonically increases the likelihood $\log P(\mathbf{x}_{1:n}|\theta^{(t)}) \geq \log P(\mathbf{x}_{1:n}|\theta^{(t-1)})$
- For Gaussian mixtures, EM is guaranteed to converge to a local maximum*
- Quality of solution highly depends on initialization! (as in k-Means)
- Common strategy: Rerun algorithm multiple times, and use the solution with largest likelihood

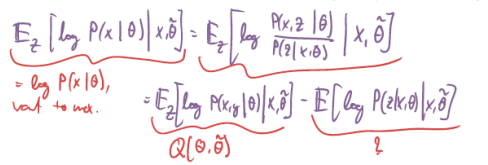
5.5.19 Initialization

How should we initialize the parameters?

- For weights: Typically use uniform distribution $w_i^0 = \frac{1}{k}\ \forall i$
- For means:
  - Randomly initialize
  - Use k-Means++ (initializes means to picked data points, makes sure to not pick all data points from the same clusters)
- For variances:
  - E.g., initialize according to empirical covariance of the data (perhaps restrict to spherical)
    E.g. $\Sigma_1 = \cdots = \Sigma_k = \frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})(x_i-\bar{x})^T$

5.5.20 EM convergence proof

- **Goal**: Maximize $\theta^* = \arg\max_\theta P(\mathbf{x}_{1:n}\mid\theta)$
- Can rewrite: $P(\mathbf{x}_{1:n}\mid\theta) = \frac{P(\mathbf{x}_{1:n},\mathbf{z}_{1:n}\mid\theta)}{P(\mathbf{z}_{1:n}\mid\mathbf{x}_{1:n},\theta)}$
- Take logs, and expectation w.r.t. $P(\mathbf{z}_{1:n}\mid\mathbf{x}_{1:n},\theta^{(t-1)})$



- So far, we have

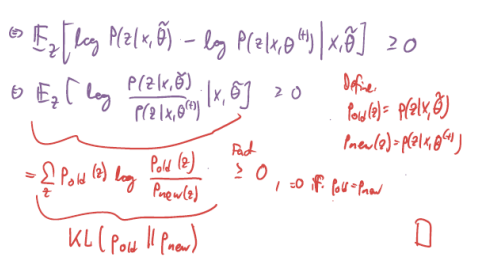
$$\log P(\mathbf{x}\mid\theta) = Q(\theta,\theta^{(t-1)}) - \mathbb{E}_{\mathbf{z}}[\log P(\mathbf{z}\mid\mathbf{x},\theta)\mid\mathbf{x},\theta^{(t-1)}]$$

- Want to show $\log P(\mathbf{x}\mid\theta^{(t)}) \geq \log P(\mathbf{x}\mid\theta^{(t-1)})$
- Compare term by term:
  - EM-Algorithm guarantees (strict inequality unless converged):
    $$Q(\theta^{(t)},\theta^{(t-1)}) \geq Q(\theta^{(t-1)},\theta^{(t-1)})$$
  - Remains to show:

$$\mathbb{E}_{\mathbf{z}}[\log P(\mathbf{z}\mid\mathbf{x},\theta^{(t-1)})\mid\mathbf{x},\theta^{(t-1)}] \geq \mathbb{E}_{\mathbf{z}}[\log P(\mathbf{z}\mid\mathbf{x},\theta^{(t)})\mid\mathbf{x},\theta^{(t-1)}]$$

Want to show

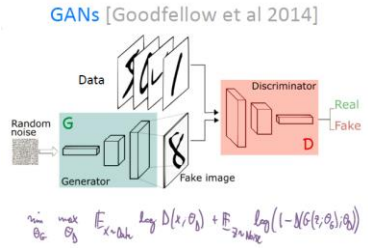
$$\mathbb{E}_{\mathbf{z}}[\log P(\mathbf{z}\mid\mathbf{x},\theta^{(t-1)})\mid\mathbf{x},\theta^{(t-1)}] \geq \mathbb{E}_{\mathbf{z}}[\log P(\mathbf{z}\mid\mathbf{x},\theta^{(t)})\mid\mathbf{x},\theta^{(t-1)}]$$



How about Hard EM? Can show that Hard EM performs alternating optimization on the complete data likelihood. The algorithm converges to a local optimum of $\max_{z_{1:n},\theta} P(\mathbf{x}_{1:n},z_{1:n}|\theta)$

5.5.21 Outlook: Implicit generative models

- Given sample of (unlabeled) points $\mathbf{x}_1,\ldots,\mathbf{x}_n$
- Goal: Learn model $\mathbf{X} = f(\mathbf{Z};\mathbf{w})$
  where Z is a "simple" distribution (e.g., lowdimensional Gaussian), and f some flexible nonlinear function (neural net)
- Approach: Optimize parameters w to make samples from model hard to distinguish from data sample
- Variants
  - Variational autoencoder (VAE)
  - Generative adversarial networks (GAN)

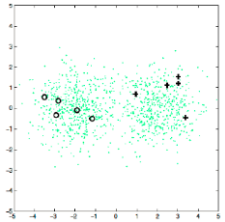


5.6 Semi-supervised learning

Often, it is easy to get unlabeled data and hard (or expensive) to get labeled data (e.g. medical application: lots of unlabeled data available, but we need to ask a doctor for labeled data -> expensive).
-> Why not combine labeled & unlabeled data!
Semi-supervised learning = learning from (large amounts) of unlabeled and (small amounts of) labeled data.
Possibilities:

- Use unlabeled data to discover lower dimensional representations using dimension reduction, PCA, kernel PCA, autoencoders etc. to get a lower dim. feature vector and then train on that labeled data.

Example:
Labeled data points (+,o) don't provide lots of information. However, with unlabeled data, we can extract quite a lot of information
-> using labeled & unlabeled data, estimate model parameters using EM



Semi-supervised learning is easy to do with GMMs

5.6.1 Semi-supervised learning with GMMs

Recall, for GMMs, at the MLE it holds that

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

$$\Sigma_j^* = \frac{\sum_{i=1}^n \gamma_j(\boldsymbol{x_i})(\boldsymbol{x_i}-\mu_j^*)(\boldsymbol{x_i}-\mu_j^*)^T}{\sum_{i=1}^n \gamma_j(\boldsymbol{x_i})}$$

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

where $\gamma_j(\boldsymbol{x}) = P(z = j|\boldsymbol{x}, \Sigma^*, \mu^*, \boldsymbol{w}^*)$
In SSL, for instance **x** with observed labels y, must hold:

$$\gamma_j(\boldsymbol{x_i}) = [j = y_i] = \begin{cases} 1, j = y_i \\ 0, else \end{cases}$$

5.6.1.1 EM for semi-supervised learning with GMMs

- While not converged
  - **E-step:**
    - For unlabeled points:
      $$\gamma_j^{(t)}(\boldsymbol{x_i}) = P\left(z = j|\boldsymbol{x_i}, \Sigma^{(t-1)}, \mu^{(t-1)}, \boldsymbol{w}^{(t-1)}\right)$$
    - For points with label y_i:
      $$\gamma_j^{(t)}(\boldsymbol{x_i}) = [j = y_i]$$
  - **M-step:** Fit clusters to weighted data points (closed form Maximum likelihood solution!!)

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

$$\Sigma_j^{(t)} = \frac{\sum_{i=1}^n \gamma_j^{(t)}(\boldsymbol{x_i})\left(\boldsymbol{x_i}-\mu_j^{(t)}\right)\left(\boldsymbol{x_i}-\mu_j^{(t)}\right)^T}{\sum_{i=1}^n \gamma_j^{(t)}(\boldsymbol{x_i})}$$

5.7 Bias Variance Tradeoff

$$\boxed{Prediction\ error = Bias^2 + Variance + Noise}$$

- **Bias**: Excess risk of best model considered compared to minimal achievable risk knowing $P(X, Y)$ (i.e., given infinite data)
  Bias is the contribution to the prediction error that I encure using a restrictive family of models (all models from family make error -> bias)
  -> Bias depends on the choice of family (linearm polynomial, ANN,…)
- **Variance:** Risk incurred due to estimating model from limited data
  -> data has random fluctuation -> different data has different fluctuation
- **Noise:** Risk error incurred by optimal model (i.e., irreducible error)

5.7.1 Bias in estimation

- MLE solution depends on training data $\mathcal{D}$

$$\hat{h} = \hat{h}_D = \underset{h\in\mathcal{H}}{\operatorname{argmin}} \sum_{(x,y)\in\mathcal{D}} \left(y_i - h(\boldsymbol{x})\right)^2$$

- But training data $\mathcal{D}$ is itself random (drawn iid from $P(\boldsymbol{X}, Y)$)
- We might want to choose H to have small bias (i.e. have small squared error on average)

$$\mathbb{E}_X\left[\underbrace{\mathbb{E}_D[\hat{h}_D(\boldsymbol{X})]}_{avg.prediciton} - \underbrace{h^*(\boldsymbol{X})}_{opt.hypothesis}\right]^2 \geq 0$$

- **Note:** MLE is an unbiased estimator -> $\mathbb{E}_D\left[\hat{h}_D(\boldsymbol{X})\right] - h^*(\boldsymbol{X})$ will be zero as long as our hypothesis class H includes h^* (e.g. data generated by degree 5 polynomial -> if $\mathbb{P}^5 \in H$ we have an unbiased estimate for MLE)

5.7.2 Variance in estimation

- MLE solution depends on training data $\mathcal{D}$

$$\hat{h} = \hat{h}_D = \underset{h\in\mathcal{H}}{\operatorname{argmin}} \sum_{(x,y)\in\mathcal{D}} \left(y_i - h(\boldsymbol{x})\right)^2$$

- This estimator is itself random, and has some variance

$$\mathbb{E}_X Var_D\left[\hat{h}_D(\boldsymbol{X})\right]^2 = \mathbb{E}_X\mathbb{E}_D\left[\hat{h}_D(\boldsymbol{X}) - \mathbb{E}_{D'}\hat{h}_{D'}(\boldsymbol{X})\right]^2$$

5.7.3 Noise in estimation

- Even if we know the Bayes' optimal hypothesis h*, we'd still incur some error due to noise

$$\mathbb{E}_{X,Y}\left[\left(Y - h^*(\boldsymbol{X})\right)^2\right]$$

  -> does not depend on learning algorithm, just comes with nature of the model
- This error is irreducible, i.e., independent of choice of the hypothesis class

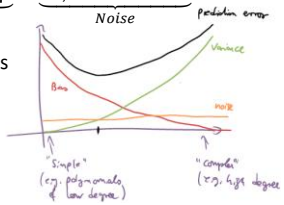
5.7.4 Bias-variance tradeoff

- For least-squares estimation the following holds

$$\mathbb{E}_D\mathbb{E}_{\boldsymbol{X},Y}\left[\left(Y - \hat{h}_D(\boldsymbol{X})\right)^2\right] = \mathbb{E}_{\boldsymbol{X}}\underbrace{\left[\overbrace{\mathbb{E}_D\hat{h}_D(\boldsymbol{X}) - h^*(\boldsymbol{X})}^{bias}\right]^2}_{Bias^2} +$$

$$+\underbrace{\mathbb{E}_{\boldsymbol{X}}\mathbb{E}_D\left[\hat{h}_D(\boldsymbol{X}) - \mathbb{E}_{D'}\hat{h}_{D'}(\boldsymbol{X})\right]^2}_{Variance} + \underbrace{\mathbb{E}_{\boldsymbol{X},Y}[Y - h^*(\boldsymbol{X})]^2}_{Noise}$$

➔ Ideally wish to find an estimator that simultaneously minimizes bias and variance



5.7.5 Bias and variance in regression

- The maximum likelihood estimate (= least-squares fit) for linear regression is unbiased (if h* in class H)
- Furthermore, it is the minimum variance estimator among all unbiased estimators (Gauss-Markov Theorem, not explained further here)
- However, we have already seen that the least-squares solution can overfit
  - Thus, trade (a little bit of) bias for a (potentially dramatic) reduction in variance
  - Regularization (e.g., ridge regression, Lasso, ...)

6 Optimization algorithms

6.1 Gradient descent

- Start at an arbitrary $w_0 \in \mathbb{R}^d$
- For t=1,2,… do $w_{t+1} = w_t - \eta_t \nabla_{w_t} \tilde{R}(\boldsymbol{w_t})$

Where η_t is called the learning rate.

Under mild assumptions, if the step size is sufficiently small, gradient descent converges to a stationary point (gradient = 0).

6.2 Stochastic Gradient Descent

- Computing the gradient requires summing over all data. For large data sets, this is inefficient.
- Moreover, our initial estimates are likely very wrong, and we can get a good (unbiased) gradient estimate by evaluating the gradient on few points.
  - ➔ If we are very far away from the optimum, we don't really care about the exact direction (gradient), we just want to go more or less in the right direction -> gain computational advantage by not computing full gradient (not on all data points)
- We could thus only calculate the gradient on some data points:
  Choose m data points at random: $i_1, \dots, i_m \sim Unif\{1, \dots, n\}$
  Then: Compute gradient over these points: $\sum_{j=1}^{m} \nabla_{\boldsymbol{w}} l\left(\boldsymbol{w}; \boldsymbol{x_{i_j}}; \boldsymbol{y_{i_j}}\right)$
  This gradient can be estimated through:
  $$\nabla \hat{R}(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} l(\boldsymbol{w}; \boldsymbol{x_i}; \boldsymbol{y_i}) = E_{I\sim Unif\{1,\dots,n\}}[\nabla_{\boldsymbol{w}} l(\boldsymbol{w}; \boldsymbol{x_I}; \boldsymbol{y_I})]$$
  And by the law of large numbers, this converges.
- Extreme case: Evaluate on only one randomly chosen point $m = 1$!
  ➔ This is stochastic gradient descent

Algorithm:

- Start at an arbitrary $\mathbf{w}_0 \in \mathbb{R}^d$
- For t=1,2,… Do
  - Pick data point $(\mathbf{x}', y') \in D$ from training set uniformly at random (with replacement), and set

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla \ell(\mathbf{w}_t; \mathbf{x}', y')$$

η_t is called learning rate.

Note: We have the same update rule as with normal gradient descent **but** we only look at gradient of one single point.

This is guaranteed to converge under mild conditions, if

$$\sum_t \eta_t = \infty \ \ and \ \ \sum_t \eta_t^2 < \infty$$

6.2.1 Checking convergence

For normal gradient descent, we can stop once:

- Gradient is small enough
- difference in objective between subsequent iterations is small enough

For SGD, we only have unbiased estimate of gradient / objective value. We cannot use this as stopping condition directly (bad behavior, batch can vary, large variance, etc.). Solution:

- Fix the number of iterations
- May also directly monitor error on separate validation set
- Compute the full objective value and compare its change periodically (e.g. every n iterations, compute the whole cost function and compare them every n iteration)

6.3 Mini-batch SGD

Using single points might have large variance in the gradient estimate, and hence lead to slow convergence. We can reduce variance by averaging over the gradients w.r.t. multiple randomly selected points (mini-batches).

7 Appendix

7.1 Convex

A function f is convex on $\mathbb{R}^d$ iff $\forall \lambda \in [0,1]$ it holds that

$$\boxed{f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)}$$

If $f: \mathbb{R}^d \longrightarrow \mathbb{R}$ is convex, then:

- $\forall x_1, \dots, x_n \in \mathbb{R}^d, \forall \lambda_1, \dots, \lambda_n \in [0,1], \sum_{i=1}^{n} \lambda_i = 1$
  $$f(\lambda_1 x_1 + \dots + \lambda_n x_n) \leq \lambda f(x_1) + \dots + \lambda_n f(x_n)$$
- If X is a R.V.
  $$f(E[X]) \leq E[f(X)]$$
- $\forall x, y \in \mathbb{R}^d$
  $$f(y) - f(x) \geq \nabla f(x)^T (y - x)$$
- Let A, b be such that $\forall z \in \mathbb{R}^n \ Az + b \in \mathbb{R}^d$
  Then: $g(z) \coloneqq f(Az + b)$ is convex in $z \in \mathbb{R}^n$
- If we have $x^* \in \mathbb{R}^d$ such that $\nabla f(x^*) = 0$ -> x^* is a global minimum, i.e. $f(x^*) \leq f(x) \forall x \in \mathbb{R}^d$

Further properties:

- Convex functions are closed under addition
- A twice differentiable function is convex if and only if its Hessian $\boldsymbol{H}$ satisfies $\boldsymbol{w^T H w} \geq 0$

7.2 Probability Theory

7.2.1 Expectation

Given the RV X:

$$E_x[X] = \begin{cases} \int x * p(x) dx \,, for\ X\ real\ RV \\ \sum_x x * p(x) \,, for\ X\ discrete\ RV \end{cases}$$

$$E[f(X)] = \begin{cases} \int f(x) * p(x) dx \,, for\ X\ real\ RV \\ \sum_x f(x) * p(x) \,, for\ X\ discrete\ RV \end{cases}$$

Linearity of expectation:

Suppose X, Y are RVs (can be independent or not). $a, b \in \mathbb{R}$

$$E_{x,y}[aX + bY] = aE_x[X] + bE_y[Y]$$

7.2.2 Variance

$$Var[X] = E[X^2] - E[X]^2$$

7.2.3 Multivariate Gaussian

Let $x_1, x_2, \dots, x_n$ realizations oft he gaussian RVs $X_1, X_2, \dots, X_n$. We can then write the observation vector $\boldsymbol{x} = [x_1 \quad x_2 \quad \dots \quad x_n]^T$ and its distribution:

$$\boxed{f_{X_1,X_2,\dots,X_n}(\boldsymbol{x}) = \frac{1}{(2\pi)^{\frac{n}{2}} \cdot |\Sigma|^{\frac{1}{2}}} \cdot \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\boldsymbol{x} - \boldsymbol{\mu})\right)}$$

$$\mu^T = [\mu_X(t_1), \dots, \mu_X(t_n)]^T$$

With covariance matrix Σ, whose entries are:

$$\Sigma_{ij} = \mathrm{cov}(X_i, X_j)$$

2D case:

$$f_{X,Y}(x,y) = \frac{1}{2\pi\sqrt{|\Sigma|}} \cdot e^{-\frac{1}{2} \cdot \left(\begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} \mu_X \\ \mu_Y \end{bmatrix}\right)^T \Sigma^{-1} \left(\begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} \mu_X \\ \mu_Y \end{bmatrix}\right)}$$

with $\Sigma = \begin{bmatrix} \mathrm{cov}(X,X) & \mathrm{cov}(X,Y) \\ \mathrm{cov}(Y,X) & \mathrm{cov}(Y,Y) \end{bmatrix}$

7.2.4 KL divergence

- For discrete random variables:
  $$D_{KL}(P||Q) = -\sum_i P(i) \log \frac{Q(i)}{P(i)}$$
- For continuous random variables:
  $$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx$$

Properties:

- $D_{KL}(P||Q) \geq 0$

7.3 Integrals

- Gaussian Integral
  $$\int_{-\infty}^{\infty} e^{-a(x+b)^2} dx = \sqrt{\frac{\pi}{a}}$$
  $$\int_{-\infty}^{\infty} e^{-ax^2+bx+c} dx = \sqrt{\frac{\pi}{a}} e^{\frac{b^2}{4a}+c}$$

7.4 Linear Algebra

7.4.1 Positive semi-definite matrices

A symmetric matrix $M \in \mathbb{R}^{nxn}$ is positive semi-definite iff

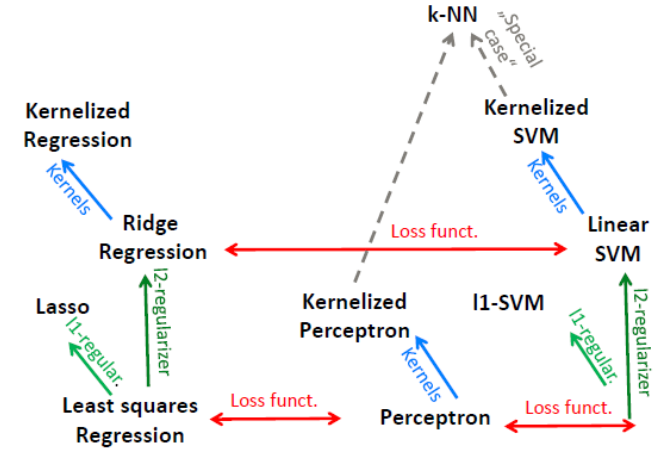
- $\forall \alpha \in \mathbb{R}^n: \alpha^T M \alpha \geq 0$

or ⇔

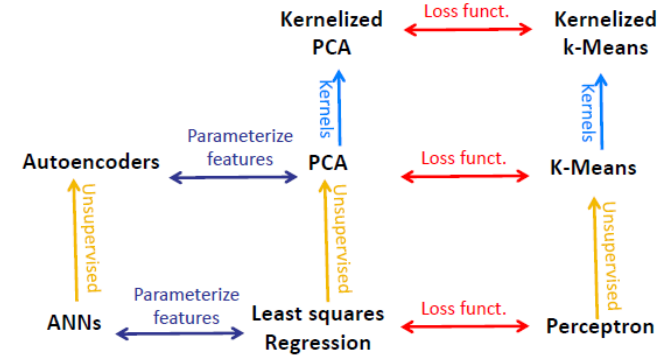
- All eigenvalues of M are ≥ 0

8 Overview

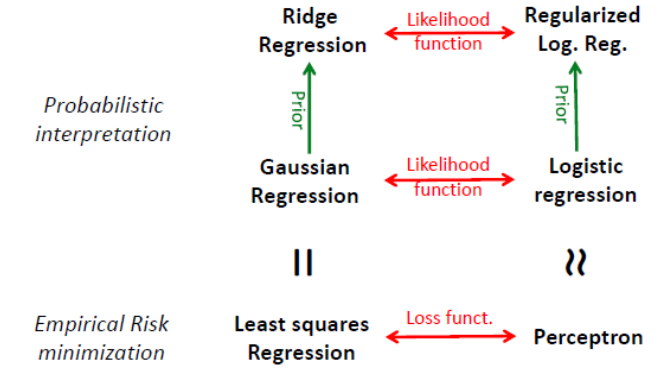
Supervised learning via risk minimization



Supervised vs unsupervised learning



Discriminative probabilistic modeling



Generative vs discriminative modeling