Introduction to Machine Learning

Marc Toussaint

July 11, 2019

This is a direct concatenation and reformatting of all lecture slides and exercises from the Machine Learning *course (summer term 2019, U Stuttgart), including indexing to help prepare for exams.*

*Double-starred** sections and slides are not relevant for the exam.*

Contents

1 Introduction

What is Machine Learning?

1) A long list of methods/algorithms for different data anlysis problems
   – in sciences
   – in commerce

2) Frameworks to develop your own learning algorithm/method

3) Machine Learning = model formulation + optimization

1:1

What is Machine Learning?

- Pedro Domingos: *A Few Useful Things to Know about Machine Learning*

learning = representation + evaluation + optimization

- "Representation": Choice of model, choice of hypothesis space
- "Evaluation": Choice of objective function, optimality principle

  Notes: The *prior* is both, a choice of representation and, usually, a part of the objective function.

  In Bayesian settings, the choice of model often directly implies also the "objective function"
- "Optimization": The algorithm to compute/approximate the best model

1:2

Pedro Domingos: *A Few Useful Things to Know about Machine Learning*

- It's generalization that counts
  – Data alone is not enough
  – Overfitting has many faces
  – Intuition fails in high dimensions
  – Theoretical guarantees are not what they seem
- Feature engineering is the key
- More data beats a cleverer algorithm
- Learn many models, not just one
- Simplicity does not imply accuracy
- Representable does not imply learnable

- Correlation does not imply causation

1:3

What is Machine Learning?

- In large parts, ML is: (let's call this ML^0)
  Fitting a function $f : x \mapsto y$ to given data $D = \{(x_i, y_i)\}_{i=1}^n$

- Why is function fitting so omnipresent?
  - Dynamics, behavior, decisions, control, predictions – are all about functions
  - Thinking? (i.e., planning, optimization, (logical) inference, CSP solving, etc?)
  - In the latter case, algorithms often provide the scaffolding, ML the heuristics to accelerate/scale them. (E.g., an evaluation function within a MCTS planning algorithm.)

1:4

ML^0 objective: Empirical Risk Minimization

- We have a hypothesis space $\mathcal{H}$ of functions $f : x \mapsto y$
  In a standard parameteric case $\mathcal{H} = \{f_\theta \mid \theta \in \mathbb{R}^n\}$ are functions $f_\theta : x \mapsto y$ that are described by n parameters $\theta \in \mathbb{R}^n$

- Given data $D = \{(x_i, y_i)\}_{i=1}^n$, the standard objective is to minimize the "error" on the data

$$f^* \operatorname*{argmin}_{f \in \mathcal{H}} \sum_{i=1}^n \ell(f(x_i), y_i) \ ,$$

  where $\ell(\hat y, y) > 0$ penalizes a discrepancy between a model output $\hat y$ and the data y.
  - Squared error $\ell(\hat y, y) = (\hat y - y)^2$
  - Classification error $\ell(\hat y, y) = [\hat y \neq y]$
  - neg-log likelihood $\ell(\hat y, y) = -\log p(y \mid \hat y)$
  - etc

1:5

What is Machine Learning beyond ML^0?

- Fitting more structured models to data, which includes
  - Time series, recurrent processes
  - Graphical Models
  - Unsupervised learning (semi-supervised learning)

  ...but in all these cases, the scenario is still not interactive, the data D is static, the decision is about picking a single model f from a hypothesis space, and the objective is a loss based on f and D only.

- Active Learning, where the "ML agent" makes decisions about what data label to query next
- Bandits, Reinforcement Learning, manipulating the domain (and thereby data source)

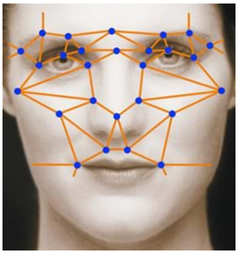
1:6

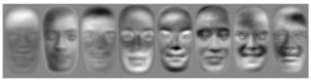
Machine Learning is everywhere

NSA, Amazon, Google, Zalando, Trading, ...
Chemistry, Biology, Physics, ...
Control, Operations Reserach, Scheduling, ...

1:7

Face recognition


keypoints


eigenfaces

(e.g., Viola & Jones)

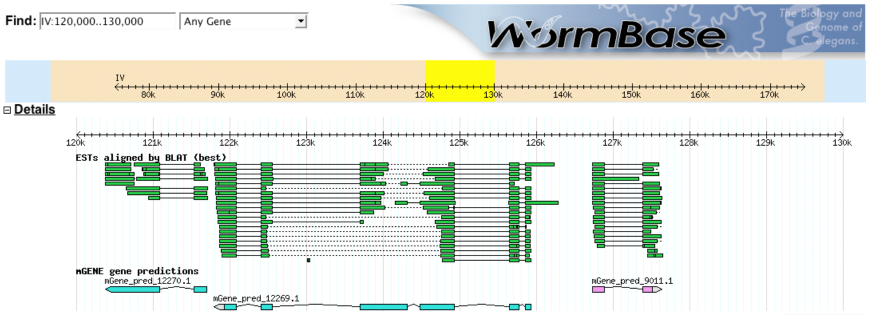
1:8

Hand-written digit recognition (US postal data)

(e.g., Yann LeCun)

1:9

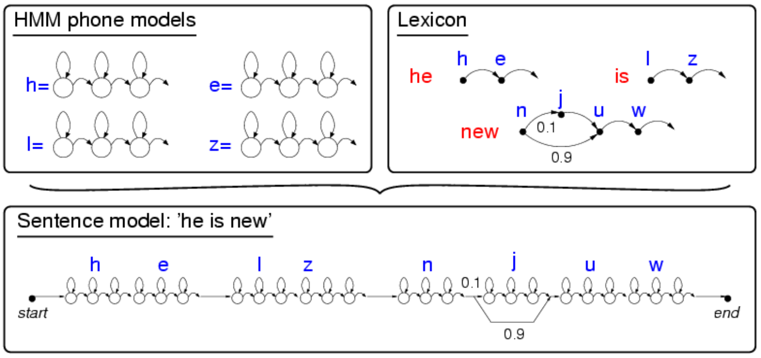
Gene annotation



(e.g., Gunnar Rätsch, mGene Project)

1:10

Speech recognition



1:11

Spam filters

	george	you	your	hp	free	hpl	!	our	re	edu	remove
spam	0.00	2.26	1.38	0.02	0.52	0.01	0.51	0.51	0.13	0.01	0.28
email	1.27	1.27	0.44	0.90	0.07	0.43	0.11	0.18	0.42	0.29	0.01

1:12

Machine Learning became an important technology in science as well

(Stuttgart Cluster of Excellence "Data-integrated Simulation Science (SimTech)")

1:13

Organization of this lecture

See TOC of last year's slide collection

- Part 1: The Core: Regression & Classification
- Part 2: The Breadth of ML methods
- Part 3: Bayesian Methods

1:14

Is this a theoretical or practical course?

Neither alone.

- The goal is to teach how to design good learning algorithms

$$\begin{tikzcd} \text{data} \arrow[d] \\ \text{modelling [requires theory \& practise]} \arrow[d] \\ \text{algorithms [requires practise \& theory]} \arrow[d] \\ \text{testing, problem identification, restart} \end{tikzcd}$$

1:15

How much math do you need?

- Let $L(x) = \|y - Ax\|^2$. What is

$$\operatorname*{argmin}_x L(x)$$

- Find

$$\min_x \|y - Ax\|^2 \quad \text{s.t.} \quad x_i \le 1$$

- Given a discriminative function $f(x, y)$ we define

$$p(y \,|\, x) = \frac{e^{f(y,x)}}{\sum_{y'} e^{f(y',x)}}$$

- Let A be the covariance matrix of a Gaussian. What does the Singular Value Decomposition $A = VDV^\top$ tell us?

1:16

How much coding do you need?

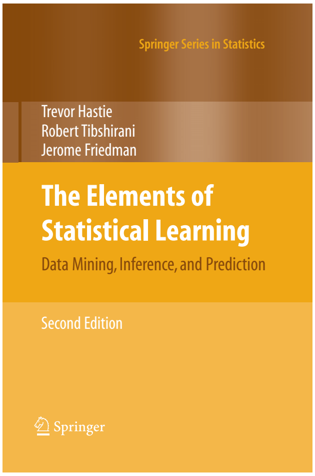
- A core subject of this lecture: learning to go from principles (math) to code

- Many exercises will implement algorithms we derived in the lecture and collect experience on small data sets

- Choice of language is fully free. I support C++; tutors might prefer Python; Octave/Matlab or R is also good choice.

1:17

Books

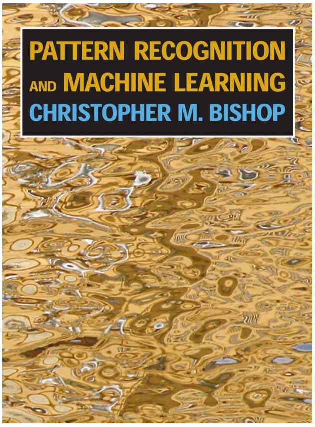

Springer Series in Statistics
Trevor Hastie
Robert Tibshirani
Jerome Friedman
The Elements of Statistical Learning
Data Mining, Inference, and Prediction
Second Edition
Springer

Trevor Hastie, Robert Tibshirani and Jerome Friedman: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* Springer, Second Edition, 2009.
`http://www-stat.stanford.edu/~tibs/ElemStatLearn/`
(recommended: read introductory chapter)

(this course will not go to the full depth in math of Hastie et al.)

1:18

Books



Bishop, C. M.: *Pattern Recognition and Machine Learning*.
Springer, 2006
http://research.microsoft.com/en-us/um/people/cmbishop/prml/
(some chapters are fully online)

1:19

Books & Readings

- more recently:
  - David Barber: Bayesian Reasoning and Machine Learning
  - Kevin Murphy: Machine learning: a Probabilistic Perspective

- See the readings at the bottom of:

http://ipvs.informatik.uni-stuttgart.de/mlr/marc/teaching/index.html#readings

1:20

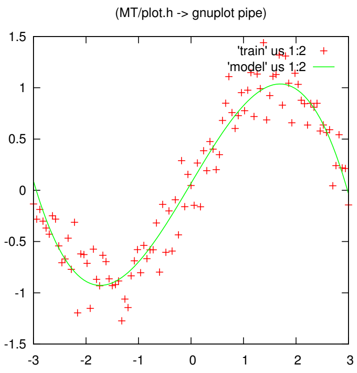
Organization

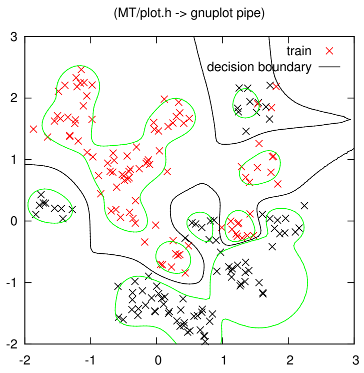
- Course Webpage:

http://ipvs.informatik.uni-stuttgart.de/mlr/marc/teaching/19-MachineLearning/
  - Slides, Exercises & Software (C++)
  - Links to books and other resources
- Admin things, please first ask:
  Carola Stahl, Carola.Stahl@ipvs.uni-stuttgart.de, Raum 2.217

- Rules for the tutorials:
  - Doing the exercises is crucial!
  - **Nur Votieraufgaben.** At the beginning of each tutorial:
    – sign into a list
    – vote on exercises you have (successfully) worked on
  - Students are randomly selected to present their solutions
  - You need 50% of completed exercises to be allowed to the exam
  - Please check 2 weeks before the end of the term, if you can take the exam

2 Regression





- Are these linear models? Linear in *what*?
  – Input: No.
  – Parameters, features: Yes!

2:1

Linear Modelling is more powerful than it might seem at first!

- Linear Regression on non-linear features $\rightarrow$ very powerful (polynomials, piece-wise, spline basis, kernels)
- Regularization (Ridge, Lasso) & cross-validation for proper generalization to test data
- Gaussian Processes and SVMs are closely related (linear in kernel features, but with different optimality criteria)
- Liquid/Echo State Machines, Extreme Learning, are examples of linear modelling on many (sort of random) non-linear features
- Basic insights in model complexity (effective degrees of freedom)
- Input relevance estimation (z-score) and feature selection (Lasso)
- Linear regression $\rightarrow$ linear classification (logistic regression: outputs are likelihood ratios)

$\Rightarrow$ Linear modelling is a core of ML
(We roughly follow Hastie, Tibshirani, Friedman: *Elements of Statistical Learning*)

2:2

Linear Regression

- Notation:
  – input vector $x \in \mathbb{R}^d$
  – output value $y \in \mathbb{R}$
  – parameters $\beta = (\beta_0, \beta_1, .., \beta_d)^\top \in \mathbb{R}^{d+1}$
  – linear model

$$f(x) = \beta_0 + \sum_{j=1}^d \beta_j x_j$$

- Given training data $D = \{(x_i, y_i)\}_{i=1}^n$ we define the *least squares* cost (or "loss")

$$L^{\text{ls}}(\beta) = \sum_{i=1}^n (y_i - f(x_i))^2$$

2:3

Optimal parameters β

- Augment input vector with a 1 in front: $\bar x = (1, x) = (1, x_1, .., x_d)^\top \in \mathbb{R}^{d+1}$
  $\beta = (\beta_0, \beta_1, .., \beta_d)^\top \in \mathbb{R}^{d+1}$

$$f(x) = \beta_0 + \sum_{j=1}^n \beta_j x_j = \bar x^\top \beta$$

- Rewrite sum of squares as:

$$L^{\text{ls}}(\beta) = \sum_{i=1}^n (y_i - \bar x_i^\top \beta)^2 = \|y - X\beta\|^2$$

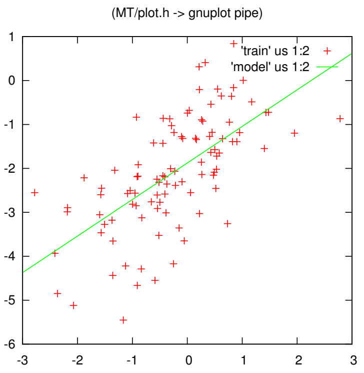
$$X = \begin{pmatrix} \bar x_1^\top \\ \vdots \\ \bar x_n^\top \end{pmatrix} = \begin{pmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,d} \\ \vdots & & & & \vdots \\ 1 & x_{n,1} & x_{n,2} & \cdots & x_{n,d} \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

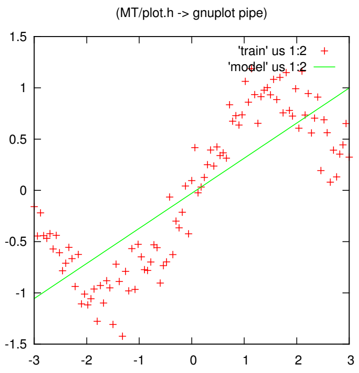
- Optimum:

$$\mathbf{0}_d^\top = \frac{\partial L^{\text{ls}}(\beta)}{\partial \beta} = -2(y - X\beta)^\top X \iff \mathbf{0}_d = X^\top X \beta - X^\top y$$

$$\hat\beta^{\text{ls}} = (X^\top X)^{-1} X^\top y$$

2:4





```
./x.exe -mode 1 -dataFeatureType 1 -modelFeatureType 1
```

2:5

Non-linear features

- Replace the inputs $x_i \in \mathbb{R}^d$ by some non-linear features $\phi(x_i) \in \mathbb{R}^k$

$$f(x) = \sum_{j=1}^k \phi_j(x)\ \beta_j = \phi(x)^\top \beta$$

- The optimal β is the same

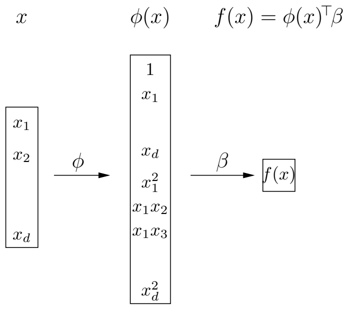
$$\hat\beta^{\text{ls}} = (X^\top X)^{\text{-}1} X^\top y \quad \text{but with} \quad X = \begin{pmatrix} \phi(x_1)^\top \\ \vdots \\ \phi(x_n)^\top \end{pmatrix} \in \mathbb{R}^{n \times k}$$

- What are "features"?
  a) Features are an arbitrary set of basis functions
  b) Any function *linear in* β can be written as $f(x) = \phi(x)^\top \beta$
  for some ϕ, which we denote as "features"

2:6

Example: Polynomial features

- Linear: $\phi(x) = (1, x_1, .., x_d) \ \in \mathbb{R}^{1+d}$
- Quadratic: $\phi(x) = (1, x_1, .., x_d, x_1^2, x_1 x_2, x_1 x_3, .., x_d^2) \ \in \mathbb{R}^{1+d+\frac{d(d+1)}{2}}$
- Cubic: $\phi(x) = (.., x_1^3, x_1^2 x_2, x_1^2 x_3, .., x_d^3) \ \in \mathbb{R}^{1+d+\frac{d(d+1)}{2}+\frac{d(d+1)(d+2)}{6}}$

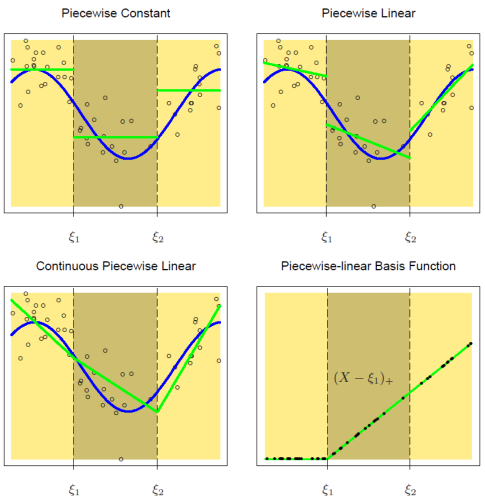


```
./x.exe -mode 1 -dataFeatureType 1 -modelFeatureType 1
```

2:7

Example: Piece-wise features (in 1D)

- Piece-wise constant: $\phi_j(x) = [\xi_j < x \le \xi_{j+1}]$
- Piece-wise linear: $\phi_j(x) = (1, x)^\top\, [\xi_j < x \le \xi_{j+1}]$
- Continuous piece-wise linear: $\phi_j(x) = [x - \xi_j]_+ \quad$ (and $\phi_0(x) = x$)



2:8

Example: Radial Basis Functions (RBF)

- Given a set of centers $\{c_j\}_{j=1}^k$, define

$$\phi_j(x) = b(x, c_j) = e^{-\frac{1}{2}\|x - c_j\|^2} \in [0, 1]$$

Each $\phi_j(x)$ measures similarity with the center c_j

- Special case:

use all training inputs $\{x_i\}_{i=1}^n$ *as centers*

$$\phi(x) = \begin{pmatrix} 1 \\ b(x, x_1) \\ \vdots \\ b(x, x_n) \end{pmatrix} \quad (n+1 \text{ dim})$$

This is related to "kernel methods" and GPs, but not quite the same—we'll discuss this later.

2:9

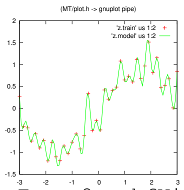
Features

- Polynomial
- Piece-wise
- Radial basis functions (RBF)
- Splines (see Hastie Ch. 5)

- Linear regression on top of rich features is extreme

2:10

The need for regularization



Noisy sin data fitted with radial basis functions

```
./x.exe -mode 1 -n 40 -modelFeatureType 4 -dataType 2 -rbfWidth .1 -sigma
.5 -lambda 1e-10
```

- **Overfitting & generalization:**
  The model overfits to the data—and generalizes badly

- **Estimator variance:**
  When you repeat the experiment (keeping the underlying function fixed), the regression always returns a different model estimate

2:11

Estimator variance

- Assumption:
  – The data was noisy with variance $\text{Var}\{y\} = \sigma^2 \mathbf{I}_n$

- We computed parameters $\hat\beta = (X^\top X)^{-1} X^\top y$, therefore

$$\text{Var}\{\hat\beta\} = (X^\top X)^{-1} \sigma^2$$

– high data noise $\sigma \to$ high estimator variance
– more data $\to$ less estimator variance: $\text{Var}\{\hat\beta\} \propto \frac{1}{n}$

- In practise we don't know σ, but we can estimate it based on the deviation from the learnt model: (with $k = \dim(\beta) = \dim(\phi)$)

$$\hat\sigma^2 = \frac{1}{n-k} \sum_{i=1}^n (y_i - f(x_i))^2$$

2:12

Estimator variance

- "Overfitting"
  – picking one specific data set $y \sim \mathcal{N}(y_\text{mean}, \sigma^2 \mathbf{I}_n)$
  $\leftrightarrow$ picking one specific $\hat b \sim \mathcal{N}(\beta_\text{mean}, (X^\top X)^{\text{-}1} \sigma^2)$

- If we could reduce the variance of the estimator, we could reduce overfitting—and increase generalization.

2:13

Hastie's section on shrinkage methods is great! Describes several ideas on reducing estimator variance by reducing model complexity. We focus on regularization.

2:14

Ridge regression: L_2-regularization

- We add a *regularization* to the cost:

$$L^\text{ridge}(\beta) = \textstyle\sum_{i=1}^n (y_i - \phi(x_i)^\top \beta)^2 + \lambda \sum_{j=2}^k \beta_j^2$$

NOTE: β_1 is usually *not* regularized!

- Optimum:

$$\hat\beta^\text{ridge} = (X^\top X + \lambda I)^{\text{-}1} X^\top y$$

(where $I = \mathbf{I}_k$, or with $I_{1,1} = 0$ if β_1 is not regularized)

2:15

- The objective is now composed of two "potentials": The loss, which depends on the data and jumps around (introduces variance), and the regularization penalty (sitting steadily at zero). Both are "pulling" at the optimal β $\to$ the regularization reduces variance.

- The estimator variance becomes less: $\text{Var}\{\hat\beta\} = (X^\top X + \lambda I)^{\text{-}1}\sigma^2$

- The ridge effectively reduces the complexity of the model:

$$\text{df}(\lambda) = \sum_{j=1}^d \frac{d_j^2}{d_j^2+\lambda}$$

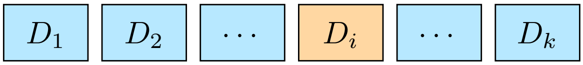
  where d_j^2 is the eigenvalue of $X^\top X = VD^2V^\top$
  (details: Hastie 3.4.1)

2:16

Choosing λ: generalization error & cross validation

- $\lambda = 0$ will always have a lower *training* data error
  We need to estimate the *generalization* error on test data

- **k-fold cross-validation:**



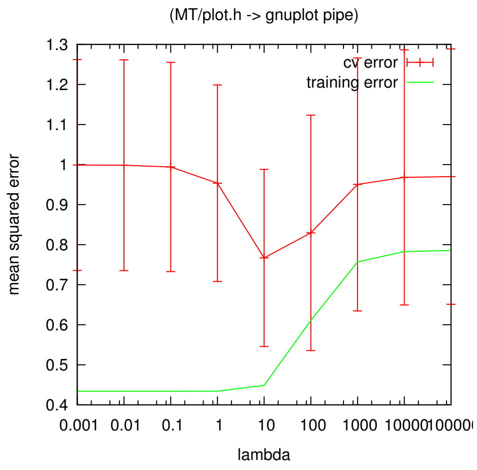
```
1: Partition data $D$ in $k$ equal sized subsets $D = \{D_1, .., D_k\}$
2: for $i = 1, .., k$ do
3:     compute $\hat\beta_i$ on the training data $D \setminus D_i$ leaving out $D_i$
4:     compute the error $\ell_i = L^{\text{ls}}(\hat\beta_i, D_i)/|D_i|$ on the validation data $D_i$
5: end for
6: report mean squared error $\hat\ell = 1/k \sum_i \ell_i$ and variance $1/(k\text{-}1)[(\sum_i \ell_i^2) - k\hat\ell^2]$
```

- Choose λ for which $\hat\ell$ is smallest

2:17

---

quadratic features on sinus data:



```
./x.exe -mode 4 -n 10 -modelFeatureType 2 -dataType 2 -sigma .1
./x.exe -mode 1 -n 10 -modelFeatureType 2 -dataType 2 -sigma .1
```

2:18

Lasso: L_1-regularization

- We add a L_1 regularization to the cost:

$$L^{\text{lasso}}(\beta) = \sum_{i=1}^n (y_i - \phi(x_i)^\top \beta)^2 + \lambda \sum_{j=2}^k |\beta_j|$$

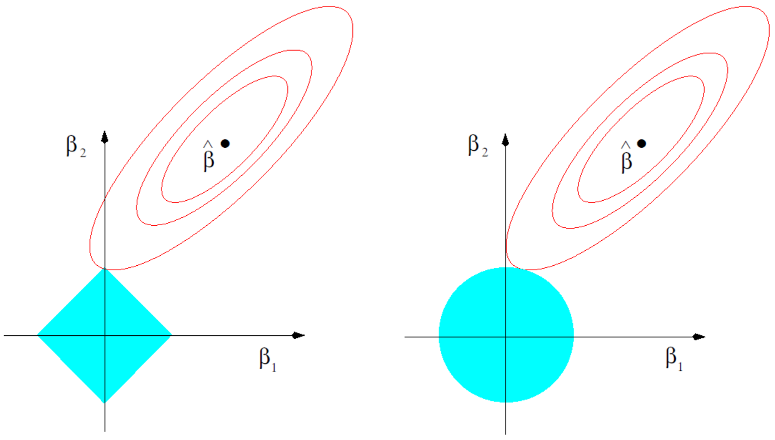
NOTE: β_1 is usually not regularized!

- Has no closed form expression for optimum

(Optimum can be found by solving a quadratic program; see appendix.)

2:19

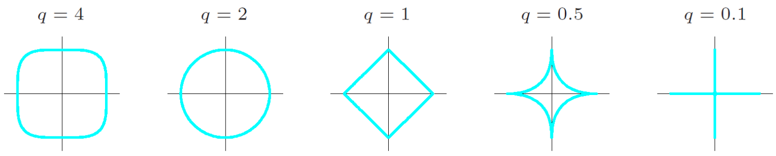
Lasso vs. Ridge:



- Lasso $\rightarrow$ sparsity! feature selection!

2:20

$$L^q(\beta) = \textstyle\sum_{i=1}^n (y_i - \phi(x_i)^\top\beta)^2 + \lambda \sum_{j=2}^k |\beta_j|^q$$



- *Subset selection:* $q = 0$ (counting the number of $\beta_j \neq 0$)

2:21

Summary

- **Representation:** choice of features

$$f(x) = \phi(x)^\top\beta$$

- **Objective:** squared error + Ridge/Lasso regularization

$$L^{\text{ridge}}(\beta) = \sum_{i=1}^n (y_i - \phi(x_i)^\top\beta)^2 + \lambda\|\beta\|_I^2$$

- **Solver:** analytical (or quadratic program for Lasso)

$$\hat\beta^{\text{ridge}} = (X^\top X + \lambda I)^{\text{-1}} X^\top y$$

2:22

Summary

- **Linear models** on non-linear features—extremely powerful

linear polynomial piece-wise linear RBF kernel	Ridge Lasso	regression classification*

*logistic regression

- Generalization $\leftrightarrow$ **Regularization** $\leftrightarrow$ complexity/DoF penalty

- **Cross validation** to estimate generalization empirically $\to$ use to choose regularization parameters

2:23

Appendix: Dual formulation of Ridge

- The standard way to write the Ridge regularization:
$L^{\text{ridge}}(\beta) = \sum_{i=1}^n (y_i - \phi(x_i)^\top\beta)^2 + \lambda \sum_{j=2}^k \beta_j^2$

- Finding $\hat\beta^{\text{ridge}} = \text{argmin}_\beta\, L^{\text{ridge}}(\beta)$ is equivalent to solving

$$\hat\beta^{\text{ridge}} = \underset{\beta}{\text{argmin}} \sum_{i=1}^n (y_i - \phi(x_i)^\top\beta)^2$$
$$\text{subject to } \sum_{j=2}^k \beta_j^2 \le t$$

λ is the Lagrange multiplier for the inequality constraint

2:24

Appendix: Dual formulation of Lasso

- The standard way to write the Lasso regularization:
$L^{\text{lasso}}(\beta) = \sum_{i=1}^n (y_i - \phi(x_i)^\top\beta)^2 + \lambda \sum_{j=2}^k |\beta_j|$

- Equivalent formulation (via KKT):

$$\hat\beta^{\text{lasso}} = \operatorname*{argmin}_\beta \sum_{i=1}^n (y_i - \phi(x_i)^\top \beta)^2$$
$$\text{subject to } \sum_{j=2}^k |\beta_j| \le t$$

- Decreasing t is called "shrinkage": The space of allowed β shrinks. Some β will become zero $\to$ feature selection

2:25

3 Classification & Structured Output

Structured Output & Structured Input

- regression:

$$\mathbb{R}^d \to \mathbb{R}$$

- structured output:

$$\begin{aligned}
\mathbb{R}^d &\to \text{binary class label } \{0, 1\} \\
\mathbb{R}^d &\to \text{integer class label } \{1, 2, .., M\} \\
\mathbb{R}^d &\to \text{sequence labelling } y_{1:T} \\
\mathbb{R}^d &\to \text{image labelling } y_{1:W,1:H} \\
\mathbb{R}^d &\to \text{graph labelling } y_{1:N}
\end{aligned}$$

- structured input:

$$\begin{aligned}
\text{relational database} &\to \mathbb{R} \\
\text{labelled graph/sequence} &\to \mathbb{R}
\end{aligned}$$

3:1

3.1 The discriminative function

3:2

Discriminative Function

- Represent a discrete-valued function $F : \mathbb{R}^d \to Y$ via a **discriminative function**

$$f : \mathbb{R}^d \times Y \to \mathbb{R}$$

  such that

$$\boxed{F : x \mapsto \text{argmax}_y f(x, y)}$$

  That is, a discriminative function $f(x, y)$ maps an input x to an output

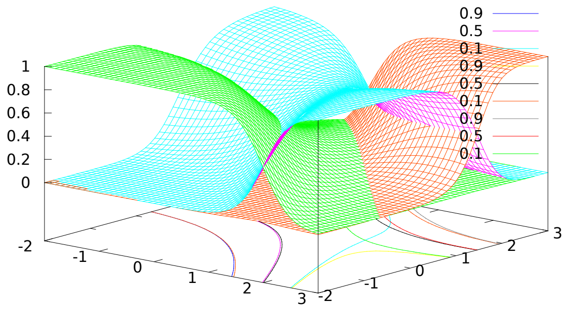
$$\hat{y}(x) = \underset{y}{\text{argmax}} f(x, y)$$

- A discriminative function $f(x, y)$ has high value if y is a correct answer to x; and low value if y is a false answer
- In that way a discriminative function discriminates correct labelling from wrong ones

3:3

Example Discriminative Function

- Input: $x \in \mathbb{R}^2$; output $y \in \{1,2,3\}$
  displayed are $p(y\!=\!1|x)$, $p(y\!=\!2|x)$, $p(y\!=\!3|x)$



(here already "scaled" to the interval [0,1]... explained later)

- You can think of $f(x,y)$ as M separate functions, one for each class $y \in \{1,..,M\}$. The highest one determines the class prediction $\hat y$
- More examples: `plot[-3:3] -x-2,0,x-2` `splot[-3:3][-3:3] -x-y-2,0,x+y-2`

3:4

How could we parameterize a discriminative function?

- Linear in features!
  - Same features, different parameters for each output: $f(x,y) = \phi(x)^\top \beta_y$
  - More general input-output features: $f(x,y) = \phi(x,y)^\top \beta$

- Example for joint features: Let $x \in \mathbb{R}$ and $y \in \{1,2,3\}$, might be

$$\phi(x,y) = \begin{pmatrix} 1 & [y=1] \\ x & [y=1] \\ x^2 & [y=1] \\ 1 & [y=2] \\ x & [y=2] \\ x^2 & [y=2] \\ 1 & [y=3] \\ x & [y=3] \\ x^2 & [y=3] \end{pmatrix}, \quad \text{which is equivalent to } f(x,y) = \begin{pmatrix} 1 \\ x \\ x^2 \end{pmatrix}^\top \beta_y$$

- Example when both $x,y \in \{0,1\}$ are discrete:

$$\phi(x,y) = \begin{pmatrix} 1 \\ [x=0][y=0] \\ [x=0][y=1] \\ [x=1][y=0] \\ [x=1][y=1] \end{pmatrix}$$

3:5

Notes on features

- Features "connect" input and output. Each $\phi_j(x,y)$ allows f to capture a certain dependence between x and y
- If both x and y are discrete, a feature $\phi_j(x,y)$ is typically a joint indicator function (logical function), indicating a certain "event"
- Each weight β_j mirrors how important/frequent/infrequent a certain dependence described by $\phi_j(x,y)$ is
- $-f(x,y)$ is also called energy, and the is also called **energy-based modelling**, esp. in neural modelling

3:6

3.2 Loss functions for classification

3:7

What is a good objective to train a classifier?

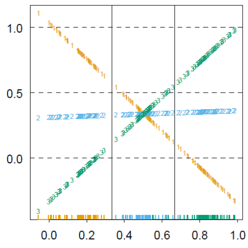
- **Accuracy, Precision & Recall:**
  For data size n, *false positives* (FP), *true positives* (TP), we define:
  - accuracy = $\frac{\text{TP+TN}}{n}$
  - precision = $\frac{\text{TP}}{\text{TP+FP}}$ (TP+FP = classifier positives)
  - recall (TP-rate) = $\frac{\text{TP}}{\text{TP+FN}}$ (TP+FN = data positives)
  - FP-rate = $\frac{\text{FP}}{\text{FP+TN}}$ (FP+TN = data negatives)
- Such metrics be our actual objective. But they are not differentiable.
  For the purpose of ML, we need to define a "proxy" objective that is nice to optimize.
- Bad idea: Squared error regression

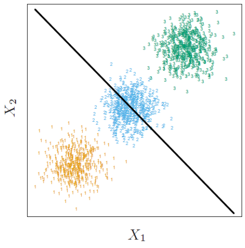
3:8

Bad idea: Squared error regression of class indicators

- Train $f(x,y)$ to be the indicator function for class y
  that is, $\forall y$: train $f(x,y)$ on the regression data $D = \{(x_i, I(y\!=\!y_i))\}_{i=1}^n$:
  train $f(x,1)$ on value 1 for all x_i with $y_i = 1$ and on 0 otherwise
  train $f(x,2)$ on value 1 for all x_i with $y_i = 2$ and on 0 otherwise
  train $f(x,3)$ on value 1 for all x_i with $y_i = 3$ and on 0 otherwise
  ...

- This typically fails: (see also Hastie 4.2)





Although the optimal separating boundaries are linear and linear discriminating functions could represent them, the linear functions trained on class indicators fail to discriminate.

$\to$ *squared error regression on class indicators is the "wrong objective"*

3:9

Log-Likelihood

- The discriminative function $f(y,x)$ not only defines the class prediction $F(x)$; we can additionally also define probabilities,

$$p(y\,|\,x) = \frac{e^{f(x,y)}}{\sum_{y'} e^{f(x,y')}}$$

- Maximizing Log-Likelihood: (minimize neg-log-likelihood, nll)

$$L^{\text{nll}}(\beta) = -\textstyle\sum_{i=1}^n \log p(y_i\,|\,x_i)$$

3:10

Cross Entropy

- This is the same as log-likelihood for categorical data, just a notational trick, really.
- The categorical data $y_i \in \{1,..,M\}$ are class labels. But assume they are encoded in a **one-hot-vector**

$$\hat y_i = \boldsymbol{e}_{y_i} = (0,..,0,1,0,...,0)\ , \quad \hat y_{iz} = [y_i = z]$$

Then we can write the neg-log-likelihood as

$$L^{\text{nll}}(\beta) = -\sum_{i=1}^n \sum_{z=1}^M \hat y_{iz} \log p(z\,|\,x_i) = \sum_{i=1}^n H(\hat y_i,\ p(\cdot,x_i))$$

where $H(p,q) = -\sum_z p(z) \log q(z)$ is the so-called cross entropy between two normalized multinomial distributions p and q.

- As a side note, the cross entropy measure would also work if the target $\hat y_i$ are probabilities instead of one-hot-vectors.

3:11

Hinge loss

- For a data point (x, y^*), the **one-vs-all hinge loss** "wants" that $f(y^*, x)$ is larger than any other $f(y, x)$, $y \neq y^*$, by a margin of 1.
  In other terms, it penalizes when $f(y^*, x) < f(y, x) + 1$, $y \neq y^*$.
- It penalizes linearly, therefore the one-vs-all hinge loss is defined as

$$L^{\text{hinge}}(f) = \sum_{y \neq y^*} [1 - (f(y^*, x) - f(y, x))]_+$$

- This is related to **Support Vector Machines** (only data points inside the margin induce an error and gradient), and also to the **Perceptron Algorithm**

3:12

3.3 Logistic regression

3:13

Logistic regression: Multi-class case

- Data $D = \{(x_i, y_i)\}_{i=1}^n$ with $x_i \in \mathbb{R}^d$ and $y_i \in \{1, .., M\}$
- We choose $f(x, y) = \phi(x)^\top \beta_y$ with separate parameters β_y for each y
- Conditional class probabilties

$$p(y \,|\, x) = \frac{e^{f(x,y)}}{\sum_{y'} e^{f(x,y')}} \qquad \leftrightarrow \qquad f(x, y) - f(x, z) = \log \frac{p(y \,|\, x)}{p(z \,|\, x)}$$

(the discriminative functions model "*log-ratios*")

- Given data $D = \{(x_i, y_i)\}_{i=1}^n$, we minimize the regularized neg-log-likelihood

$$L^{\text{logistic}}(\beta) = -\textstyle\sum_{i=1}^n \log p(y_i \,|\, x_i) + \lambda \|\beta\|^2$$

  Written as cross entropy (with one-hot encoding $\hat{y}_{iz} = [y_i = z]$):

$$L^{\text{logistic}}(\beta) = -\sum_{i=1}^n \sum_{z=1}^M [y_i = z] \log p(z \,|\, x_i) + \lambda \|\beta\|^2$$

3:14

Optimal parameters β

- Gradient:

$$\frac{\partial L^{\text{logistic}}(\beta)}{\partial \beta_c}^\top = \sum_{i=1}^n (p_{ic} - y_{ic}) \phi(x_i) + 2\lambda I \beta_c = X^\top (p_c - y_c) + 2\lambda I \beta_c$$

where $p_{ic} = p(y{=}c \,|\, x_i)$

which is non-linear in β $\Rightarrow$ $\partial_\beta L = 0$ does not have an analytic solution

- Hessian:

$$H = \frac{\partial^2 L^{\text{logistic}}(\beta)}{\partial\beta_c \partial\beta_d} = X^\top W_{cd} X + 2[c=d]\,\lambda I$$

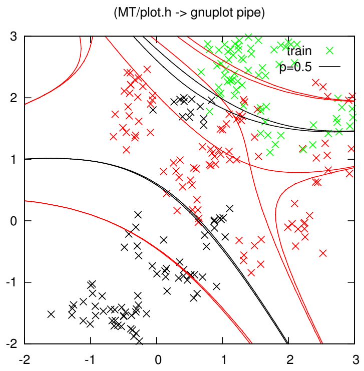
where W_{cd} is diagonal with $W_{cd,ii} = p_{ic}([c=d] - p_{id})$

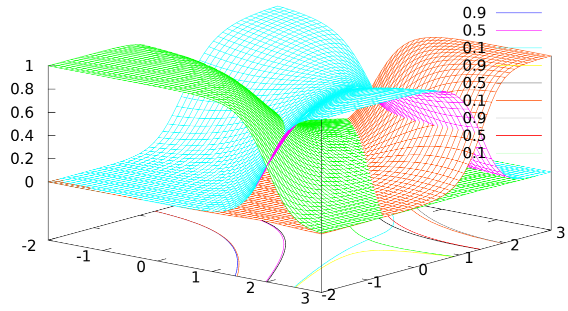
- Newton algorithm: iterate

$$\beta \leftarrow \beta - H^{-1}\, \frac{\partial L^{\text{logistic}}(\beta)}{\partial \beta}^\top$$

3:15

polynomial (quadratic) ridge 3-class logistic regression:





```
./x.exe -mode 3 -d 2 -n 200 -modelFeatureType 3 -lambda 1e+1
```

3:16

- Note, if we have M discriminative functions $f(x,y)$, w.l.o.g., we can always choose one of them to be constantly zero. E.g.,

$$f(x,M) \equiv 0 \text{ or } \beta_M \equiv 0$$

The other functions then have to be greater/less relative to this baseline.

- This is usually not done in the multi-class case, *but almost always in the binary case.*

3:17

Logistic regression: Binary case

- In the binary case, we have "two functions" $f(x,0)$ and $f(x,1)$. **W.l.o.g. we may fix $f(x,0)=0$ to zero.** Therefore we choose features

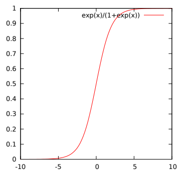
$$\phi(x,y) = \phi(x)\,[y=1]$$

  with arbitrary input features $\phi(x) \in \mathbb{R}^k$

- We have

$$f(x,1) = \phi(x)^\top \beta \ , \quad \hat y = \underset{y}{\text{argmax}}\, f(x,y) = \begin{cases} 0 & \text{else} \\ 1 & \text{if } \phi(x)^\top \beta > 0 \end{cases}$$

- and conditional class probabilities



$$p(1\,|\,x) = \frac{e^{f(x,1)}}{e^{f(x,0)} + e^{f(x,1)}} = \sigma(f(x,1))$$

  with the *logistic sigmoid function* $\sigma(z) = \frac{e^z}{1+e^z} = \frac{1}{e^{-z}+1}$.

- Given data $D = \{(x_i,y_i)\}_{i=1}^n$, we minimize the regularized neg-log-likelihood

$$L^{\text{logistic}}(\beta) = -\textstyle\sum_{i=1}^n \log p(y_i\,|\,x_i) + \lambda \|\beta\|^2$$

$$= -\textstyle\sum_{i=1}^n \Big[y_i \log p(1\,|\,x_i) + (1-y_i)\log[1-p(1\,|\,x_i)] \Big] + \lambda\|\beta\|^2$$

3:18

Optimal parameters β

- Gradient (see exercises):

$$\frac{\partial L^{\text{logistic}}(\beta)}{\partial \beta}^\top = \sum_{i=1}^n (p_i - y_i)\phi(x_i) + 2\lambda I \beta = X^\top (p-y) + 2\lambda I \beta$$

$$\text{where} \quad p_i := p(y\!=\!1\,|\,x_i)\ , \quad X = \begin{pmatrix} \phi(x_1)^\top \\ \vdots \\ \phi(x_n)^\top \end{pmatrix} \in \mathbb{R}^{n\times k}$$

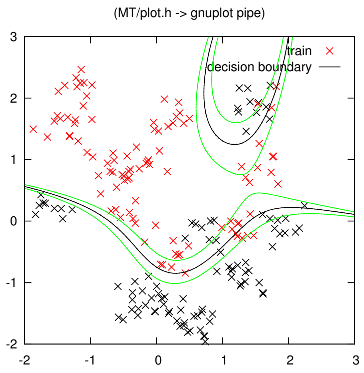
- Hessian $H = \frac{\partial^2 L^{\text{logistic}}(\beta)}{\partial \beta^2} = X^\top W X + 2\lambda I$
  $W = \text{diag}(p \circ (1-p))$, that is, diagonal with $W_{ii} = p_i(1-p_i)$

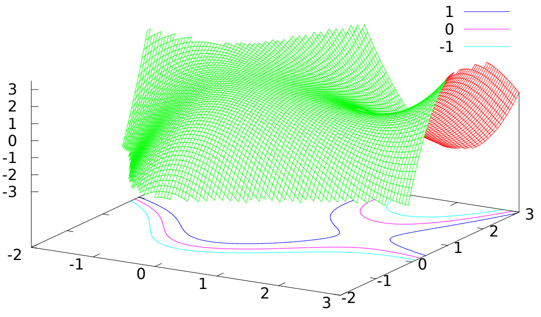
- Newton algorithm: iterate

$$\beta \leftarrow \beta - H^{\text{-1}} \ \frac{\partial L^{\text{logistic}}(\beta)}{\partial \beta}^\top$$

3:19

polynomial (cubic) ridge logistic regression:

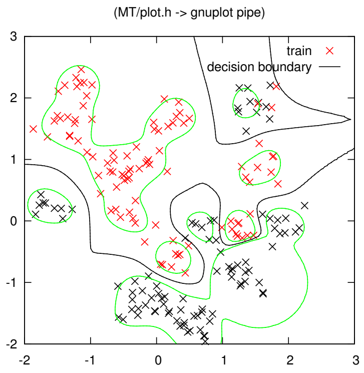


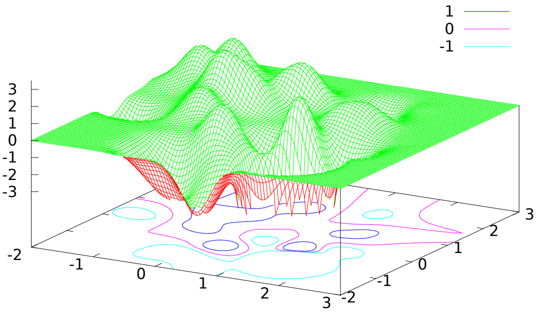


```
./x.exe -mode 2 -d 2 -n 200 -modelFeatureType 3 -lambda 1e+0
```

3:20

RBF ridge logistic regression:





```
./x.exe -mode 2 -d 2 -n 200 -modelFeatureType 4 -lambda 1e+0 -rbfBias
0 -rbfWidth .2
```

3:21

Recap

	ridge regression	logistic regression
REPRESENTATION	$f(x) = \phi(x)^\top\beta$	$f(x,y) = \phi(x,y)^\top\beta$
OBJECTIVE	$L^{\text{ls}}(\beta) = \sum_{i=1}^n (y_i - f(x_i))^2 + \lambda\|\beta\|_I^2$	$L^{\text{logistic}}(\beta) = -\sum_{i=1}^n \log p(y_i \,\vert\, x_i) + \lambda\|\beta\|_I^2$ $p(y \,\vert\, x) \propto e^{f(x,y)}$
SOLVER	$\hat\beta^{\text{ridge}} = (X^\top X + \lambda I)^{\text{-1}} X^\top y$	binary case: $\beta \leftarrow \beta - (X^\top W X + 2\lambda I)^{\text{-1}} (X^\top(p-y) + 2\lambda I\beta)$

3:22

3.4 Conditional Random Fields**

3:23

Examples for Structured Output

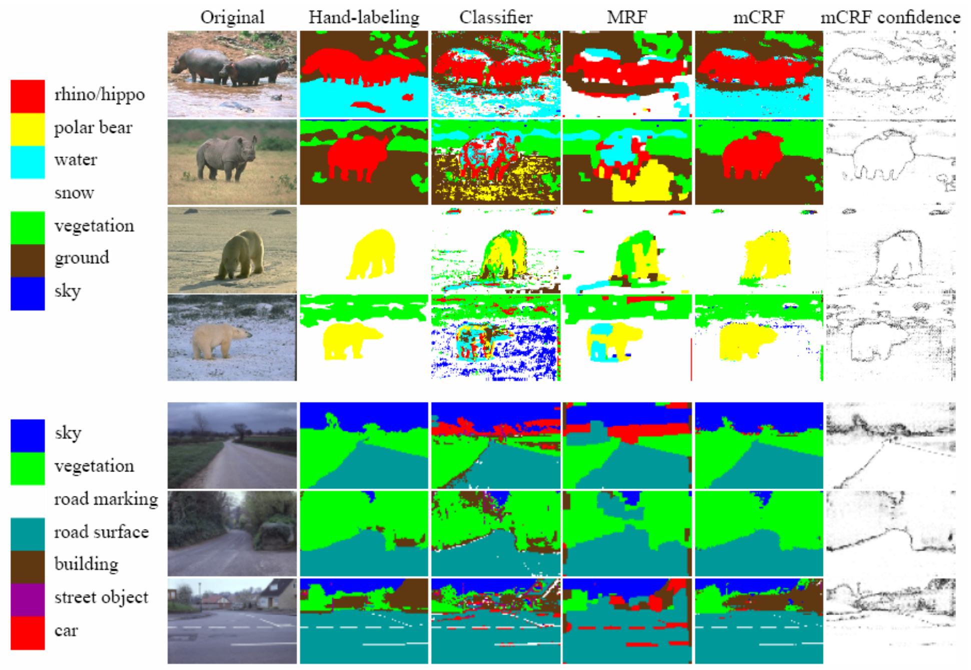
- Text tagging
  X = sentence
  Y = tagging of each word
  http://sourceforge.net/projects/crftagger

- Image segmentation
  X = image
  Y = labelling of each pixel
  http://scholar.google.com/scholar?cluster=13447702299042713582

- Depth estimation
  X = single image
  Y = depth map
  http://make3d.cs.cornell.edu/

3:24

CRFs in image processing



3:25

CRFs in image processing

- Google "conditional random field image"
  - Multiscale Conditional Random Fields for Image Labeling (CVPR 2004)
  - Scale-Invariant Contour Completion Using Conditional Random Fields (ICCV 2005)
  - Conditional Random Fields for Object Recognition (NIPS 2004)
  - Image Modeling using Tree Structured Conditional Random Fields (IJCAI 2007)
  - A Conditional Random Field Model for Video Super-resolution (ICPR 2006)

3:26

Conditional Random Fields (CRFs)

- CRFs are a generalization of logistic binary and multi-class classification
- The output y may be an arbitrary (usually discrete) thing (e.g., sequence/image/grap labelling)
- Hopefully we can maximize efficiently

$$\operatorname*{argmax}_y f(x, y)$$

  over the output!

$\to f(x, y)$ should be *structured* in y so this optimization is efficient.

- The name CRF describes that $p(y|x) \propto e^{f(x,y)}$ defines a probability distribution (a.k.a. random field) over the output y conditional to the input x. The word "field" usually means that this distribution is structured (a graphical model; see later part of lecture).

3:27

CRFs: the structure is in the features

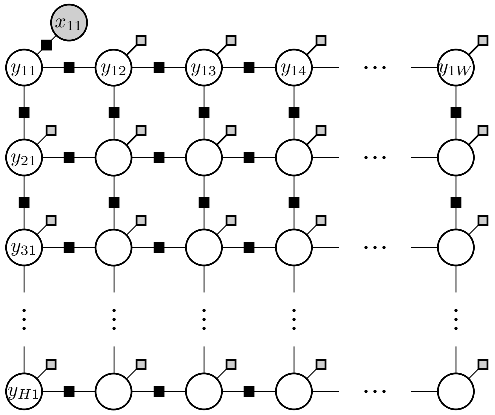
- Assume $y = (y_1, .., y_l)$ is a tuple of individual (local) discrete labels
- We can assume that $f(x, y)$ is linear in features

$$f(x, y) = \sum_{j=1}^{k} \phi_j(x, y_{\partial j})\beta_j = \phi(x, y)^\top \beta$$

where **each feature** $\phi_j(x, y_{\partial j})$ **depends only on a subset** $y_{\partial j}$ **of labels**. $\phi_j(x, y_{\partial j})$ effectively couples the labels $y_{\partial j}$. Then $e^{f(x,y)}$ is a **factor graph**.

3:28

Example: pair-wise coupled pixel labels



- Each black box corresponds to features $\phi_j(y_{\partial j})$ which couple neighboring pixel labels $y_{\partial j}$
- Each gray box corresponds to features $\phi_j(x_j, y_j)$ which couple a local pixel observation x_j with a pixel label y_j

3:29

CRFs: Core equations

$$f(x, y) = \phi(x, y)^\top \beta$$

$$
\begin{aligned}
p(y|x) &= \frac{e^{f(x,y)}}{\sum_{y'} e^{f(x,y')}} = e^{f(x,y)-Z(x,\beta)} \\
Z(x,\beta) &= \log \sum_{y'} e^{f(x,y')} \quad \text{(log partition function)} \\
L(\beta) &= -\sum_i \log p(y_i|x_i) = -\sum_i [f(x_i,y_i) - Z(x_i,\beta)] \\
\nabla Z(x,\beta) &= \sum_y p(y|x)\ \nabla f(x,y) \\
\nabla^2 Z(x,\beta) &= \sum_y p(y|x)\ \nabla f(x,y)\ \nabla f(x,y)^\top - \nabla Z\ \nabla Z^\top
\end{aligned}
$$

- This gives the neg-log-likelihood $L(\beta)$, its gradient and Hessian

3:30

Training CRFs

- Maximize conditional likelihood

  But Hessian is typically too large (Images: ∼10 000 pixels, ∼50 000 features)

  If $f(x,y)$ has a chain structure over y, the Hessian is usually banded $\to$ computation time linear in chain length

  Alternative: Efficient gradient method, e.g.:

  Vishwanathan et al.: Accelerated Training of Conditional Random Fields with Stochastic Gradient Methods

- Other loss variants, e.g., hinge loss as with Support Vector Machines

  ("Structured output SVMs")

- Perceptron algorithm: Minimizes hinge loss using a gradient method

3:31

4 Neural Networks

Outline

- Model, Objective, Solver:
  - How do NNs represent a function $f(x)$, or discriminative function $f(y, x)$?
  - What are objectives? (standard objectives, different regularizations)
  - How are they trained? (Initialization, SGD)
- Computation Graphs & Chain Rules
- Images & Sequences
  - CNNs
  - LSTMs & GRUs
  - Complex architectures (e.g. Mask-RCNN, dense pose prediction, etc)

4:1

Neural Network models

- NNs are a parameterized function $f_\beta : \mathbb{R}^d \mapsto \mathbb{R}^M$
  - β are called weights
  - Given a data set $D = \{(x_i, y_i)\}_{i=1}^n$, we minimize some loss

$$\beta^* = \underset{\beta}{\text{argmin}} \sum_{i=1}^n \ell(f_\beta(x_i), y_i) + \text{ regularization}$$

- In that sense, they just replace our previous model assumption $f(x) = \phi(x)^\top \beta$, the reset is "in principle" the same

4:2

Neural Network models

- A (fwd-forward) NN $\mathbb{R}^{h_0} \mapsto \mathbb{R}^{h_L}$ with L layers, each h_l-dimensional, defines the function

$$\begin{array}{lll} \text{1-layer: } f_\beta(x) = W_1 x + b_1, & & W_1 \in \mathbb{R}^{h_1 \times h_0}, b_1 \in \mathbb{R}^{h_1} \\ \text{2-layer: } f_\beta(x) = W_2 \sigma(W_1 x + b_1) + b_2, & & W_l \in \mathbb{R}^{h_l \times h_{l\text{-}1}}, b_l \in \mathbb{R}^{h_l} \\ L\text{-layer: } f_\beta(x) = W_L \sigma(\cdots \sigma(W_1 x + b_1) \cdots) + b_L & & \end{array}$$

- The parameter $\beta = (W_{1:L}, b_{1:L})$ is the collection of all **weights** $W_l \in \mathbb{R}^{h_l \times h_{l\text{-}1}}$ and **biases** $b_l \in \mathbb{R}^{h_l}$
- To describe the mapping as an iteration, we introduce notation for the intermediate values:
  - the **input** to layer l is $z_l = W_l x_{l\text{-}1} + b_l \in \mathbb{R}^{h_l}$
  - the **activation** of layer l is $x_l = \sigma(z_l) \in \mathbb{R}^{h_l}$

  Then the L-layer NN model can be computed using the **forward propagation:**

$$\forall_{l=1,..,L\text{-}1} : \ z_l = W_l x_{l\text{-}1} + b_l \ , \quad x_l = \sigma(z_l)$$

  where $x_0 \equiv x$ is the input, and $f_\beta(x) \equiv z_L$ the output

4:3

Neural Network models

- The **activation function** $\sigma(z)$ is applied *element-wise*,

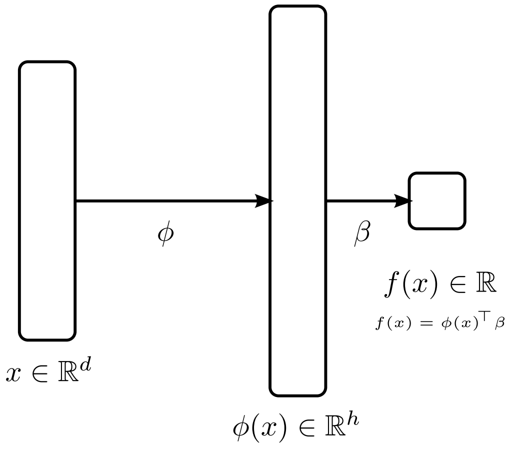
rectified linear unit (ReLU)	$\sigma(z) = [z]_+ = \max\{0, z\} = z[z \ge 0]$
leaky ReLU	$\sigma(z) = \max\{0.01z, z\} = \begin{cases} 0.01z & z < 0 \\ z & z \ge 0 \end{cases}$
sigmoid, logistic	$\sigma(z) = 1/(1 + e^{-z})$
tanh	$\sigma(z) = \tanh(z)$

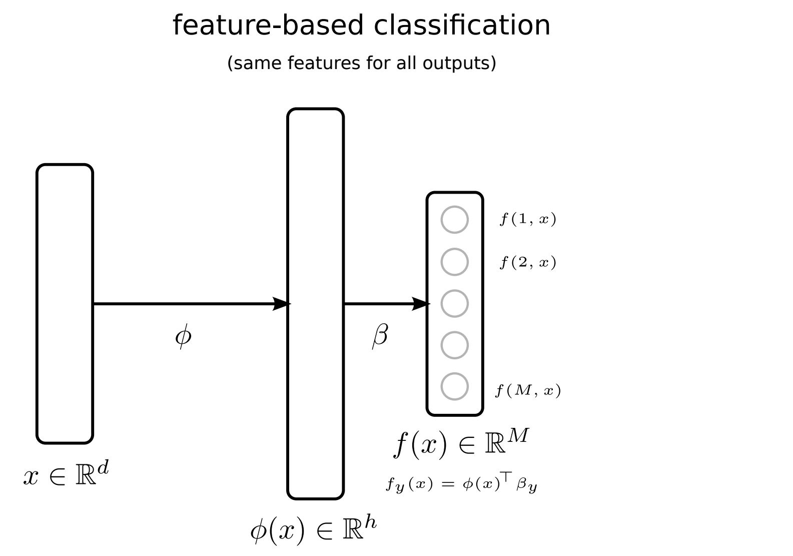
- L-layer means $L - 1$ hidden layers plus 1 output layer. (The input x_0 is not counted.)

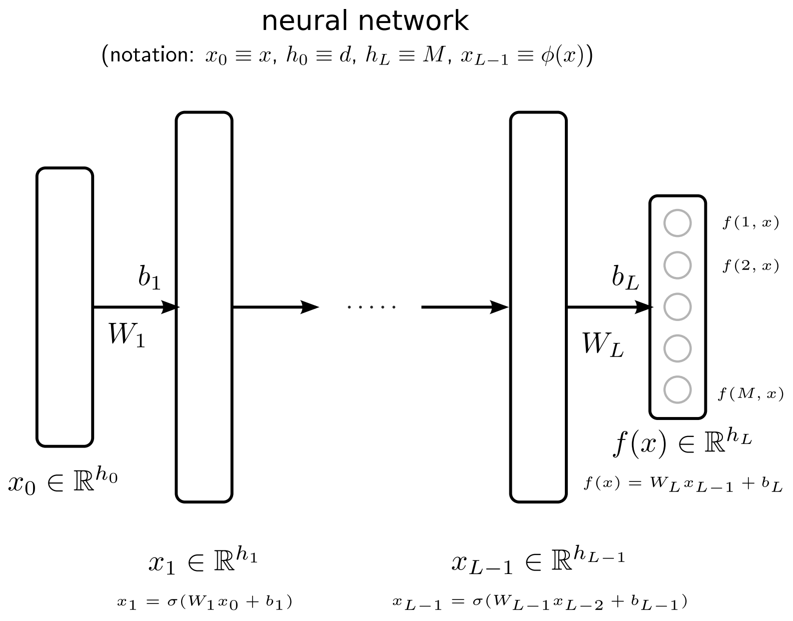
- The forward propagation therefore iterates applying
  - a linear transformation $x_{l\text{-}1} \mapsto z_l$, highly parameterized with W_l, b_l
  - a non-linear transformation $z_l \mapsto x_l$, element-wise and without parameters

4:4

feature-based regression




feature-based classification
(same features for all outputs)
ϕ
β
$f(1,x)$
$f(2,x)$
$f(M,x)$
$x \in \mathbb{R}^d$
$\phi(x) \in \mathbb{R}^h$
$f(x) \in \mathbb{R}^M$
$f_y(x) = \phi(x)^\top \beta_y$


neural network
(notation: $x_0 \equiv x$, $h_0 \equiv d$, $h_L \equiv M$, $x_{L-1} \equiv \phi(x)$)
b_1
W_1
b_L
W_L
$f(1,x)$
$f(2,x)$
$f(M,x)$
$x_0 \in \mathbb{R}^{h_0}$
$x_1 \in \mathbb{R}^{h_1}$
$x_1 = \sigma(W_1 x_0 + b_1)$
$x_{L-1} \in \mathbb{R}^{h_{L-1}}$
$x_{L-1} = \sigma(W_{L-1} x_{L-2} + b_{L-1})$
$f(x) \in \mathbb{R}^{h_L}$
$f(x) = W_L x_{L-1} + b_L$

4:5

Neural Network models

- We can think of the second-to-last layer $x_{L\text{-}1}$ as a feature vector

$$\phi_\beta(x) = x_{L\text{-}1}$$

- This aligns NNs models with what we discussed so far. But the crucial difference is:

  In NNs, the features $\phi_\beta(x)$ are also parameterized and trained!

  While in previous lectures, we had to fix $\phi(x)$ by hand, NNs allow us to learn features and **intermediate representations**

- Note: It is a common approach to train NNs as usual, but after training fix the trained features $\phi(x)$ ("remove the head (=output layer) and fix the remaining body of the NN") and use these trained features for similar problems or other kinds of ML on top.

4:6

NNs as unversal function approximators

- A 1-layer NN is linear in the input
- Already a 2-layer NN with $h_1 \to \infty$ hidden neurons is a universal function approximator
  – Corresponds to $k \to \infty$ features $\phi(x) \in \mathbb{R}^k$ that are well tuned

4:7

Objectives to train NNs

– loss functions
– regularization

4:8

Loss functions as usual

- Squared error regression, for h_L-dimensional output:
  – for a single data point (x, y^*), $\ell(f(x), y^*) = (f(x) - y^*)^2$
  – the loss gradient is $\frac{\partial \ell}{\partial f} = 2(f - y^*)^\top$

- For multi-class classification we have $h_L = M$ outputs, and $f_\beta(x) \in \mathbb{R}^M$ represents the discriminative function
- Neg-log-likelihood or cross entropy loss:
  – for a single data point (x, y^*), $\ell(f(x), y^*) = -\log p(y^*|x)$
  – the loss gradient at output y is $\frac{\partial \ell}{\partial f_y} = p(y|x) - [y = y^*]$
- One-vs-all hinge loss:
  – for a single data point (x, y^*), $\ell(f(x), y^*) = \sum_{y \neq y^*} [1 - (f_{y^*} - f_y)]_+$,
  – the loss gradient at non-target outputs $y \neq y^*$ is $\frac{\partial \ell}{\partial f_y} = [f_{y^*} < f_y + 1]$
  – the loss gradient at the target output y^* is $\frac{\partial \ell}{\partial f_{y^*}} = -\sum_{y \neq y^*} [f_{y^*} < f_y + 1]$

4:9

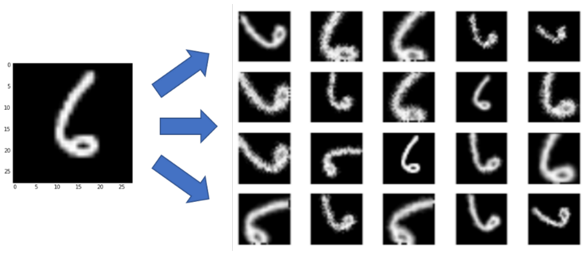
New types of regularization

- Conventional, add a L_2 or L_1 regularization.
  - adds a penalty $\lambda W_{l,ij}^2$ (Ridge) or $\lambda |W_{l,ij}|$ (Lasso) for every weight
  - In practise, compute the unregularized gradient as usual, then add $\lambda W_{l,ij}$ (for L_2), or $\lambda \,\text{sign}\, W_{l,ij}$ (for L_1) to the gradient
  - Historically, this is called **weight decay**, as the additional gradient (executed after the unregularized weight update) just decays weights
- Dropout
  - *Srivastava et al: Dropout: a simple way to prevent neural networks from overfitting, JMLR 2014.*
  - "a way of approximately combining exponentially many different neural network architectures efficiently"
  - "p can simply be set at 0.5, which seems to be close to optimal for a wide range ofnetworks and tasks"
  - on test/prediction time, take true averages
- Others:
  - Data Augmentation
  - Training ensembles, bagging (averaging bootstrapped models)
  - Additional embedding objectives (e.g. semi-supervised embedding)
  - Early stopping

4:10

Data Augmentation

- A very interesting form of regularization is to modify the data!
- Generate more data by applying invariances to the given data. The model then learns to generalize as described by these invariances.
- This is a form of regularization that directly incorporates expert knowledge



4:11

Optimization

4:12

Computing the gradient

- Recall **forward propagation** in an L-layer NN:

$$\forall_{l=1,..,L\text{-}1} : \; z_l = W_l x_{l\text{-}1} + b_l \; , \quad x_l = \sigma(z_l)$$

- For a single data point (x, y^*), assume we have a loss $\ell(f(x), y^*)$
  We define $\delta_L \triangleq \frac{d\ell}{df} = \frac{d\ell}{dz_L} \in \mathbb{R}^{1\times M}$ as the gradient (as row vector) w.r.t. output values z_L.
- **Backpropagation:** We can recursivly compute the gradient $\frac{d\ell}{dz_l} \in \mathbb{R}^{1\times h_l}$ w.r.t. all other layers z_l as:

$$\forall_{l=L\text{-}1,..,1} : \; \delta_l \triangleq \frac{d\ell}{dz_l} = \frac{d\ell}{dz_{l+1}} \frac{\partial z_{l+1}}{\partial x_l} \frac{\partial x_l}{\partial z_l} = [\delta_{l+1}\, W_{l+1}] \circ [\sigma'(z_l)]^\top$$

  where $\circ$ is an *element-wise product*. The gradient w.r.t. parameters:

$$\frac{d\ell}{dW_{l,ij}} = \frac{d\ell}{dz_{l,i}} \frac{\partial z_{l,i}}{\partial W_{l,ij}} = \delta_{l,i}\, x_{l\text{-}1,j} \quad \text{or} \quad \frac{d\ell}{dW_l} = \delta_l^\top x_{l\text{-}1}^\top \; , \quad \frac{d\ell}{db_l} = \delta_l^\top$$

4:13

- This forward and backward computations are done for each data point (x_i, y_i).
- Since the total loss is the sum $L(\beta) = \sum_i \ell(f_\beta(x_i), y_i)$, the total gradient is the sum of gradients per data point.
- Efficient implementations send multiple data points (tensors) simultaneously through the network (fwd and bwd), which speeds up computations.

4:14

Optimization

- For small data size:
  We can compute the loss and its gradient $\sum_{i=1}^n \nabla_\beta \ell(f_\beta(x_i), y_i)$.
  - Use classical gradient-based optimization methods
  - default: L-BFGS, oldish but efficient: Rprop
  - Called **batch learning** (in contrast to online learning)

- For large data size: The $\sum_{i=1}^n$ is highly inefficient!
  - Adapt weights based on much smaller data subsets, **mini batches**

4:15

Stochastic Gradient Descent

- Compute the loss and gradient for a mini batch $\hat D \subset D$ of fixed size k.

$$\begin{aligned} L(\beta, \hat D) &= \sum_{i \in \hat D} \ell(f_\beta(x_i), y_i) \\ \nabla_\beta L(\beta, \hat D) &= \sum_{i \in \hat D} \nabla_\beta \ell(f_\beta(x_i), y_i) \end{aligned}$$

- Naive Stochastic Gradient Descent, iterate

$$\beta \leftarrow \beta - \eta \nabla_\beta L(\beta, \hat D)$$

  - Choice of learning rate η is crucial for convergence!
  - Exponential cooling: $\eta = \eta_0^t$

4:16

Stochastic Gradient Descent

- SGD with momentum:

$$\begin{aligned} \Delta\beta &\leftarrow \alpha \Delta\beta - \eta \nabla_\beta L(\beta, \hat D) \\ \beta &\leftarrow \beta + \Delta\beta \end{aligned}$$

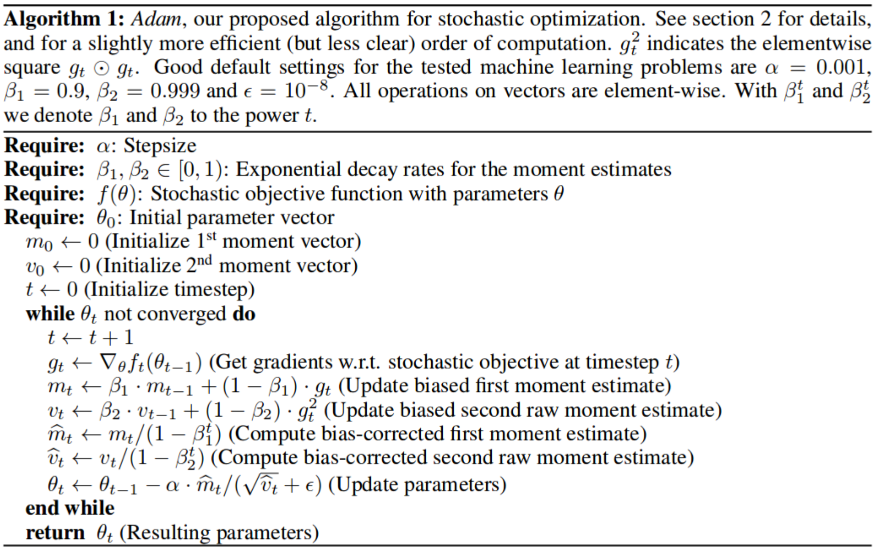
- Nesterov Accelerated Gradient (“Nesterov Momentum”):

$$\begin{aligned} \Delta\beta &\leftarrow \alpha \Delta\beta - \eta \nabla_\beta L(\beta + \Delta\beta, \hat D) \\ \beta &\leftarrow \beta + \Delta\beta \end{aligned}$$

Yurii Nesterov (1983): *A method for solving the convex programming problm with convergence rate* $O(1/k^2)$

4:17

Adam



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

arXiv:1412.6980

(all operations interpreted element-wise)

4:18

Adam & Nadam

- Adam interpretations (everything element-wise!):
  - $m_t \approx \langle g \rangle$ the mean gradient in the recent iterations
  - $v_t \approx \langle g^2 \rangle$ the mean gradient-square in the recent iterations
  - $\hat m_t, \hat v_t$ are bias corrected (check: in first iteration, $t = 1$, we have $\hat m_t = g_t$, unbiased, as desired)
  - $\Delta\theta \approx -\alpha \frac{\langle g \rangle}{\langle g^2 \rangle}$ *would* be a Newton step if $\langle g^2 \rangle$ *were* the Hessian...

- Incorporate Nesterov into Adam: Replace parameter update by

$$\theta_t \leftarrow \theta_{t\text{-}1} - \alpha/(\sqrt{\hat v_t} + \epsilon) \cdot (\beta_1 \hat m_t + \frac{(1-\beta_1) g_t}{1 - \beta_1^t})$$

Dozat: *Incorporating Nesterov Momentum into Adam*, ICLR'16

4:19

Initialization

- The Initialization of weights is important! Heuristics:

- Choose random weights that don't grow or vanish the gradient:

  - E.g., initialize weight vectors in $W_{l,i\cdot}$ with standard deviation 1, i.e., each entry with sdv $\frac{1}{\sqrt{h_{l\text{-}1}}}$
  - Roughly: If each element of z_l has standard deviation ϵ, the same should be true for z_{l+1}.

- Choose each weight vector $W_{l,i\cdot}$ to point in a uniform random direction $\to$ same as above

- Choose biases $b_{l,i}$ randomly so that the ReLU hinges cover the input well (think of distributing hinge features for continuous piece-wise linear regression)

4:20

Brief Discussion

4:21

Historical discussion

(This is completely subjective.)

- Early (from 40ies):
  - McCulloch Pitts, Hebbian learning, Rosenblatt, Werbos (backpropagation)
- 80ies:
  - Start of connectionism, NIPS
  - ML wants to distinguish itself from pure statistics (“machines”, “agents”)
- ’90-’10:
  - More theory, better grounded, Statistical Learning theory
  - Good ML is pure statistics (again) (Frequentists, SVM)
  - ...or pure Bayesian (Graphical Models, Bayesian X)
  - sample-efficiency, great generalization, guarantees, theory
  - Great successes, in applications across disciplines; supervised, unsupervised, structured
- ’10-:
  - Big Data. NNs. Size matters. GPUs.
  - Disproportionate focus on images
  - Software engineering becomes central

4:22

- NNs did not become “better” than they were 20y ago. What changed is the metrics by which they’re are evaluated:
- Old:
  - Sample efficiency & generalization; get the most from little data

  - Guarantees (both, w.r.t. generalization and optimization)
  - **generalize** much better than nearest neighbor
- New:
  - Ability to cope with billions of samples
  - $\to$ no batch processing, but **stochastic** optimization (Adam) without monotone convergence
  - $\to$ nearest neighbor methods infeasible, **compress** data into high-capacity NNs

4:23

NNs vs. nearest neighbor

- Imagine an autonomous car. Instead of carrying a neural net, it carries 1 Petabyte of data (500 hard drives, several billion pictures). In every split second it records an image from a camera and wants to query the database to returen the 100 most similar pictures. Perhaps with a non-trivial similarity metric. That's not reasonable!
- In that sense, NNs are much better than nearest neighbor. They store/compress/memorize huge amounts of data. Sample efficiency and the precise generalization behavior beome less relevant.
- That's how the metrics changed from '90-'10 to nowadays

4:24

4.1 Computation Graphs

- A great collateral benefit of NN research!
- Perhaps a new paradigm to design large scale systems, beyond what software engineering teaches classically
- [see section 3.2 in "Maths" lecture]

4:25

Example

- Three real-valued quantities x, g and f which depend on each other:

$$f(x,g) = 3x + 2g \quad \text{and} \quad g(x) = 2x \; .$$

  What is $\frac{\partial}{\partial x} f(x,g)$ and what is $\frac{d}{dx} f(x,g)$?

- The *partial* derivative only considers a single function $f(a,b,c,..)$ and asks how the output of this single function varies with one of its arguments. (Not caring that the arguments might be functions of yet something else).
- The *total* derivative considers full networks of dependencies between quantities and asks how one quantity varies with some other.

4:26

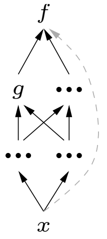
Computation Graphs

- A **function network** or **computation graph** is a DAG of n quantities x_i where each quantity is a deterministic function of a set of parents $\pi(i) \subset \{1, .., n\}$, that is
$$x_i = f_i(x_{\pi(i)})$$
where $x_{\pi(i)} = (x_j)_{j \in \pi(i)}$ is the tuple of parent values
- (This could also be called *deterministic Bayes net*.)

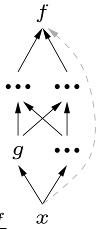
- Total derivative: Given a variation dx of some quantity, how would all child quantities (down the DAG) vary?

4:27

Chain rules



- Forward-version: (I use in robotics)
$$\frac{df}{dx} = \sum_{g \in \pi(f)} \frac{\partial f}{\partial g} \frac{dg}{dx}$$
Why "forward"? You've computed $\frac{dg}{dx}$ already, now you move forward to $\frac{df}{dx}$.
Note: If $x \in \pi(f)$ is also a direct argument to f, the sum includes the term $\frac{\partial f}{\partial x} \frac{dx}{dx} \equiv \frac{\partial f}{\partial x}$. To emphasize this, one could also write $\frac{df}{dx} = \frac{\partial f}{\partial x} + \sum_{\substack{g \in \pi(f) \\ g \neq x}} \frac{\partial f}{\partial g} \frac{dg}{dx}$.



- Backward-version: (used in NNs!)
$$\frac{df}{dx} = \sum_{g : x \in \pi(g)} \frac{df}{dg} \frac{\partial g}{\partial x}$$
Why "backward"? You've computed $\frac{df}{dg}$ already, now you move backward to $\frac{df}{dx}$.
Note: If $f \in \pi(g)$, the sum includes $\frac{df}{df} \frac{\partial f}{\partial x} \equiv \frac{\partial f}{\partial x}$. We could also write $\frac{df}{dx} = \frac{\partial f}{\partial x} + \sum_{\substack{g : x \in \pi(g) \\ g \neq f}} \frac{df}{dg} \frac{\partial g}{\partial x}$.

4:28

4.2 Images & Time Series

4:29

Images & Time Series

- My guess: 90% of the recent success of NNs is in the areas of images or time series
- For images, convolutional NNs (CNNs) impose a very sensible prior; the representations that emerge in CNNs are in fact similar to representations in the visual area of our brain.

- For time series, long-short term memory (LSTM) networks represent long-term dependencies in a way that is well trainable – something that is hard to do with other model structures.
- Both these structural priors, combined with huge data and capacity, make these methods very strong.

4:30

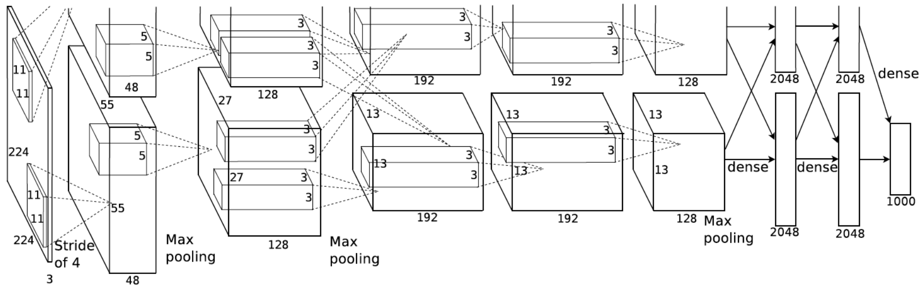
Convolutional NNs

- Standard fully connected layer: full matrix W_i has $h_i h_{i+1}$ parameters
- Convolutional: Each neuron (entry of z_{i+1}) receives input from a square receptive field, with $k \times k$ parameters. All neurons *share* these parameters $\to$ *translation invariance*. The whole layer only has k^2 parameters.
- There are often multiple neurons with the same receitive field ("depth" of the layer), to represent different "filters". Stride leads to downsampling. Padding at borders.

- Pooling applies a predefined operation on the receptive field (no parameters): max or average. Typically for downsampling.

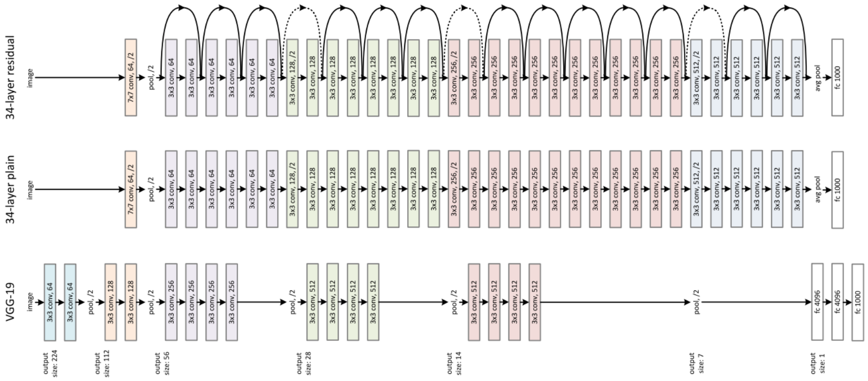
4:31

Learning to read these diagrams...



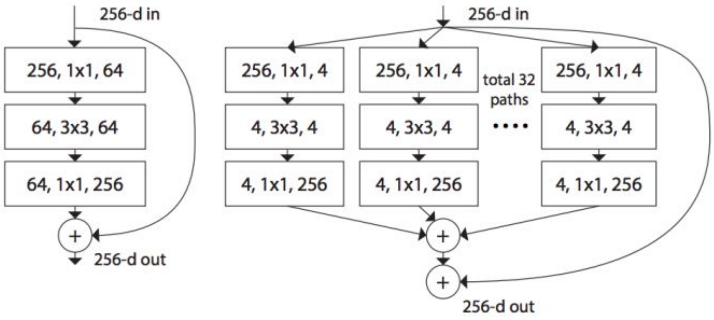
AlexNet

4:32



ResNet

4:33



ResNeXt

4:34

Pretrained networks

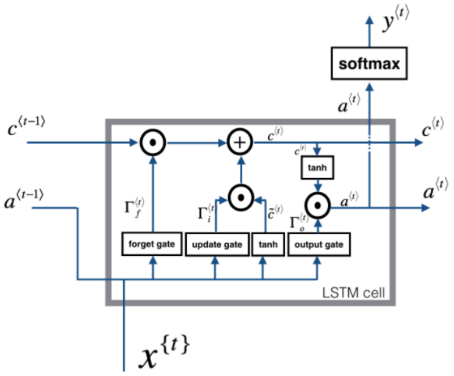
- ImageNet5k, AlexNet, VGG, ResNet, ResNeXt

4:35

LSTMs

2 - Long Short-Term Memory (LSTM) network

This following figure shows the operations of an LSTM-cell.



. . .

$$\Gamma_f^{\langle t\rangle} = \sigma(W_f[a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_f)$$
$$\Gamma_u^{\langle t\rangle} = \sigma(W_u[a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_u)$$
$$\tilde{c}^{\langle t\rangle} = \tanh(W_C[a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_C)$$
$$c^{\langle t\rangle} = \Gamma_f^{\langle t\rangle} \circ c^{\langle t-1\rangle} + \Gamma_u^{\langle t\rangle} \circ \tilde{c}^{\langle t\rangle}$$
$$\Gamma_o^{\langle t\rangle} = \sigma(W_o[a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_o)$$
$$a^{\langle t\rangle} = \Gamma_o^{\langle t\rangle} \circ \tanh(c^{\langle t\rangle})$$

Figure 4: LSTM-cell. This tracks and updates a "cell state" or memory variable $c^{\langle t\rangle}$ at every time-step, which can be different from $a^{\langle t\rangle}$.

4:36

LSTM

- c is a memory signal, that is multiplied with a sigmoid signal Γ_f. If that is saturated ($\Gamma_f \approx 1$), the memory is preserved; and backpropagation copies gradients back
- If Γ_i is close to 1, a new signal $\tilde{c}$ is written into memory
- If Γ_o is close to 1, the memory contributes to the normal neural activations a

4:37

Gated Recurrent Units

- Cleaner and more modern: Gated Recurrent Units
  but perhaps just similar performance
- Gated Feedback RNNs

4:38

Deep RL

- Value Network
- Advantage Network
- Action Network
- Experience Replay (prioritized)
- Fixed Q-targets
- etc, etc

4:39

Conclusions

- Conventional feed-forward neural networks are by no means magic. They're a parameterized function, which is fit to data.
- Convolutional NNs do make strong and good assumptions about how information processing on images should be structured. The results are great and related to some degree to human visual representations. A large part of the success of deep learning is on images.

  Also LSTMs make good assumptions about how memory signals help represent time series.

  The flexibility of "clicking together" network structures and general differentiable computation graphs is great.

  All these are innovations w.r.t. *formulating structured models* for ML
- The major strength of NNs is in their capacity and that, using massive parallelized computation, they can be trained on tons of data. Maybe they don't even need to be better than nearest neighbor lookup, but they can be queried much faster.

4:40

5 Kernelization

Kernel Ridge Regression—the "Kernel Trick"

- Reconsider solution of Ridge regression (using the *Woodbury* identity):
$$\hat\beta^{\text{ridge}} = (X^\top X + \lambda \mathbf{I}_k)^{\text{-}1} X^\top y = X^\top (X X^\top + \lambda \mathbf{I}_n)^{\text{-}1} y$$

- Recall $X^\top = (\phi(x_1), .., \phi(x_n)) \in \mathbb{R}^{k\times n}$, then:
$$f^{\text{ridge}}(x) = \phi(x)^\top \beta^{\text{ridge}} = \underbrace{\phi(x)^\top X^\top}_{\kappa(x)^\top} (\underbrace{X X^\top}_{K} + \lambda I)^{\text{-}1} y$$

  K is called *kernel matrix* and has elements
$$K_{ij} = k(x_i, x_j) := \phi(x_i)^\top \phi(x_j)$$

  κ is the vector: $\kappa(x)^\top = \phi(x)^\top X^\top = k(x, x_{1:n})$

The kernel function $k(x, x')$ calculates the scalar product in feature space.

5:1

The Kernel Trick

- We can rewrite kernel ridge regression as:
$$\begin{aligned} f^{\text{rigde}}(x) &= \kappa(x)^\top (K + \lambda I)^{\text{-}1} y \\ \text{with } K_{ij} &= k(x_i, x_j) \\ \kappa_i(x) &= k(x, x_i) \end{aligned}$$

  $\to$ at no place we actually need to compute the parameters $\hat\beta$
  $\to$ at no place we actually need to compute the features $\phi(x_i)$
  $\to$ we only need to be able to compute $k(x, x')$ for any x, x'

- This rewriting is called *kernel trick.*
- It has great implications:
  - Instead of inventing funny non-linear features, we may directly invent funny kernels
  - Inventing a kernel is intuitive: $k(x, x')$ expresses how correlated y and y' should be: it is a meassure of similarity, it compares x and x'. Specifying how 'comparable' x and x' are is often more intuitive than defining "features that might work".

5:2

- Every choice of features implies a kernel.
- But, does every choice of kernel correspond to a specific choice of features?

5:3

Reproducing Kernel Hilbert Space

- Let's define a vector space $\mathcal{H}_k$, spanned by infinitely many basis elements
$$\{\phi_x = k(\cdot, x) \ : \ x \in \mathbb{R}^d\}$$
Vectors in this space are linear combinations of such basis elements, e.g.,
$$f = \sum_i \alpha_i \phi_{x_i} \ , \quad f(x) = \sum_i \alpha_i k(x, x_i)$$
- Let's define a scalar product in this space. Assuming $k(\cdot,\cdot)$ is positive definite, we first define the scalar product for every basis element,
$$\langle \phi_x, \phi_y \rangle := k(x, y)$$
Then it follows
$$\langle \phi_x, f \rangle = \sum_i \alpha_i \langle \phi_x, \phi_{x_i} \rangle = \sum_i \alpha_i k(x, x_i) = f(x)$$
- The $\phi_x = k(\cdot, x)$ is the 'feature' we associate with x. Note that this is a function and infinite dimensional. Choosing $\alpha = (K + \lambda I)^{\text{-}1} y$ represents $f^{\text{ridge}}(x) = \sum_{i=1}^n \alpha_i k(x, x_i) = \kappa(x)^\top \alpha$, and shows that ridge regression has a **finite-dimensional solution** in the basis elements $\{\phi_{x_i}\}$. A more general version of this insight is called **representer theorem**.

5:4

Representer Theorem

- For
$$f^* = \underset{f \in \mathcal{H}_k}{\text{argmin}}\, L(f(x_1), .., f(x_n)) + \Omega(\|f\|^2_{\mathcal{H}_k})$$
where L is an arbitrary loss function, and Ω a monotone regularization, it holds
$$f^* = \sum_{i=1}^n \alpha_i k(\cdot, x_i)$$
- Proof:

  decompose $f = f_s + f_\perp, \ f_s \in \text{span}\{\phi_{x_i} : x_i \in D\}$

  $f(x_i) = \langle f, \phi_{x_i} \rangle = \langle f_s + f_\perp, \phi_{x_i} \rangle = \langle f_s, \phi_{x_i} \rangle = f_s(x_i)$

  $L(f(x_1), .., f(x_n)) = L(f_s(x_1), .., f_s(x_n))$

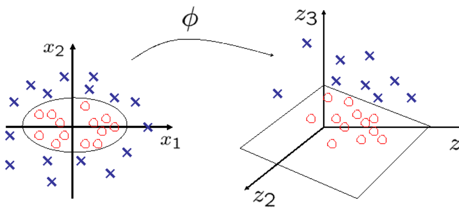
  $\Omega(\|f_s + f_\perp\|^2_{\mathcal{H}_k}) \geq \Omega(\|f_s\|^2_{\mathcal{H}_k})$

5:5

Example Kernels

- Kernel functions need to be positive definite: $\forall_{z:|z|>0} : k(z,z') > 0$
  $\to K$ is a positive definite matrix
- Examples:
  - Polynomial: $k(x,x') = (x^\top x' + c)^d$
    Let's verify for $d=2$, $\phi(x) = \big(1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2\big)^\top$:

$$k(x,x') = ((x_1,x_2)\begin{pmatrix}x'_1\\x'_2\end{pmatrix} + 1)^2$$
$$= (x_1x'_1 + x_2x'_2 + 1)^2$$
$$= x_1^2{x'_1}^2 + 2x_1x_2x'_1x'_2 + x_2^2{x'_2}^2 + 2x_1x'_1 + 2x_2x'_2 + 1$$
$$= \phi(x)^\top\phi(x')$$



  - Squared exponential (radial basis function): $k(x,x') = \exp(-\gamma\,|x-x'|^2)$

5:6

Example Kernels

- Bag-of-words kernels: let $\phi_w(x)$ be the count of word w in document x; define $k(x,y) = \langle\phi(x),\phi(y)\rangle$
- Graph kernels (Vishwanathan et al: Graph kernels, JMLR 2010)
  - Random walk graph kernels

- Gaussian Process regression will explain that $k(x,x')$ has the semantics of an (apriori) *correlatedness* of the yet unknown underlying function values $f(x)$ and $f(x')$
  - $k(x,x')$ should be high if you believe that $f(x)$ and $f(x')$ might be similar
  - $k(x,x')$ should be zero if $f(x)$ and $f(x')$ might be fully unrelated

5:7

Kernel Logistic Regression*

For logistic regression we compute β using the Newton iterates

$$\beta \leftarrow \beta - (X^\top WX + 2\lambda I)^{-1}\,[X^\top(p-y) + 2\lambda\beta] \tag{1}$$
$$= -(X^\top WX + 2\lambda I)^{-1}\,X^\top[(p-y) - WX\beta] \tag{2}$$

Using the Woodbury identity we can rewrite this as

$$(X^\top WX + A)^{-1}X^\top W = A^{-1}X^\top(XA^{-1}X^\top + W^{-1})^{-1} \tag{3}$$
$$\beta \leftarrow -\frac{1}{2\lambda}X^\top(X\frac{1}{2\lambda}X^\top + W^{-1})^{-1}\,W^{-1}[(p-y) - WX\beta] \tag{4}$$
$$= X^\top(XX^\top + 2\lambda W^{-1})^{-1}\,\Big[X\beta - W^{-1}(p-y)\Big]. \tag{5}$$

We can now compute the discriminative function values $f_X = X\beta \in \mathbb{R}^n$ at the training points by iterating over those instead of β:

$$f_X \leftarrow XX^\top(XX^\top + 2\lambda W^{\text{-}1})^{\text{-}1} \Big[X\beta - W^{\text{-}1}(p - y)\Big] \tag{6}$$

$$= K(K + 2\lambda W^{\text{-}1})^{\text{-}1} \Big[f_X - W^{\text{-}1}(p_X - y)\Big] \tag{7}$$

Note, that p_X on the RHS also depends on f_X. Given f_X we can compute the discriminative function values $f_Z = Z\beta \in \mathbb{R}^m$ for a set of m query points Z using

$$f_Z \leftarrow \kappa^\top(K + 2\lambda W^{\text{-}1})^{\text{-}1} \Big[f_X - W^{\text{-}1}(p_X - y)\Big]\ , \quad \kappa^\top = ZX^\top \tag{8}$$

6 Unsupervised Learning

Unsupervised learning

- What does that mean? Generally: modelling $P(x)$
- Instances:
  - Finding lower-dimensional spaces
  - Clustering
  - Density estimation
  - Fitting a graphical model
- "Supervised Learning as special case"...

6:1

6.1 PCA and Embeddings

6:2

Principle Component Analysis (PCA)

- Assume we have data $D = \{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$.

  Intuitively: "We believe that there is an **underlying lower-dimensional space** explaining this data".

- How can we formalize this?

6:3

PCA: minimizing projection error

- For each $x_i \in \mathbb{R}^d$ we postulate a lower-dimensional latent variable $z_i \in \mathbb{R}^p$

$$x_i \approx V_p z_i + \mu$$

- Optimality:
  Find V_p, μ and values z_i that minimize $\sum_{i=1}^n \|x_i - (V_p z_i + \mu)\|^2$

6:4

Optimal V_p

$$\hat\mu, \hat z_{1:n} = \underset{\mu, z_{1:n}}{\text{argmin}} \sum_{i=1}^n \|x_i - V_p z_i - \mu\|^2$$

$\Rightarrow \hat\mu = \langle x_i \rangle = \frac{1}{n}\sum_{i=1}^n x_i$, $\quad \hat z_i = V_p^\top (x_i - \mu)$

- Center the data $\tilde{x}_i = x_i - \hat{\mu}$. Then

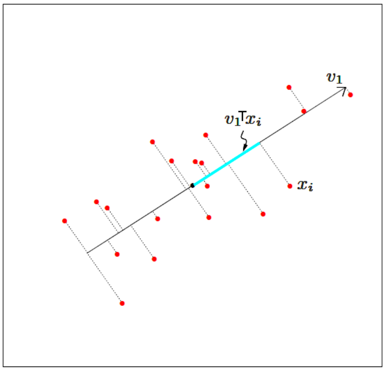
$$\hat{V}_p = \underset{V_p}{\text{argmin}} \sum_{i=1}^n \|\tilde{x}_i - V_p V_p^\top \tilde{x}_i\|^2$$

- Solution via Singular Value Decomposition
  – Let $X \in \mathbb{R}^{n\times d}$ be the centered data matrix containing all $\tilde{x}_i$
  – We compute a sorted Singular Value Decomposition $X^\top X = VDV^\top$
  D is diagonal with sorted singular values $\lambda_1 \ge \lambda_2 \ge \dots \ge \lambda_d$
  $V = (v_1\ v_2\ \cdots\ v_d)$ contains largest eigenvectors v_i as columns

$$V_p := V_{1:d,1:p} = (v_1\ v_2\ \cdots\ v_p)$$

6:5

Principle Component Analysis (PCA)



$V_p^\top$ is the matrix that projects to the largest variance directions of $X^\top X$

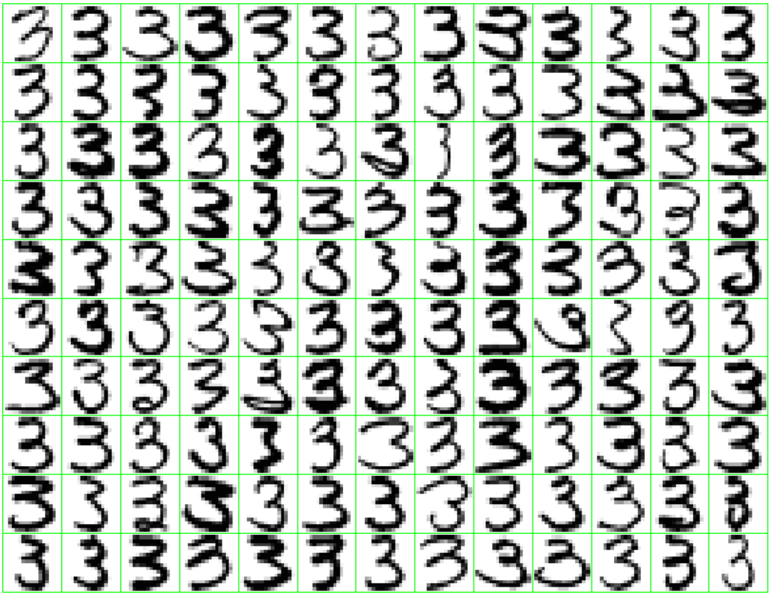
$$z_i = V_p^\top (x_i - \mu)\ , \quad Z = XV_p$$

- In non-centered case: Compute SVD of variance

$$A = \text{Var}\{x\} = \langle xx^\top \rangle - \mu\mu^\top = \frac{1}{n} X^\top X - \mu\mu^\top$$

6:6

Example: Digits



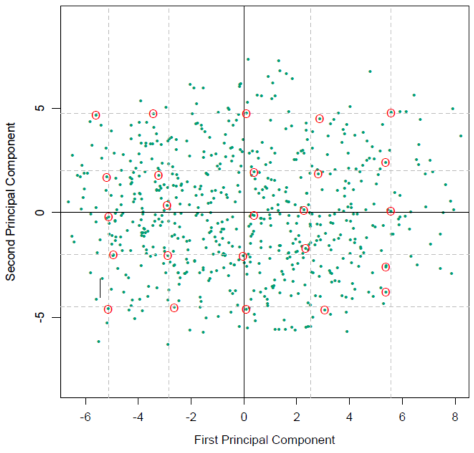
6:7

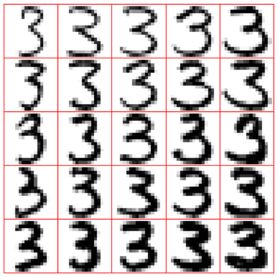
Example: Digits

- The "basis vectors" in V_p are also **eigenvectors**
  Every data point can be expressed in these eigenvectors

$$\begin{aligned} x &\approx \mu + V_p z \\ &= \mu + z_1 v_1 + z_2 v_2 + \dots \\ &= \end{aligned}$$

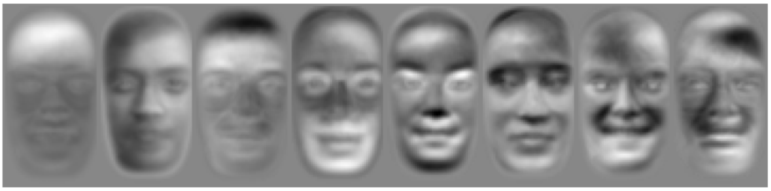
$+ z_1 \cdot$ $+ z_2 \cdot$ $+ \cdots$





6:8

Example: Eigenfaces

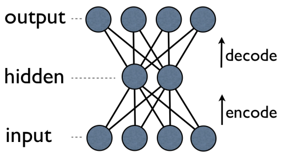


(Viola & Jones)

6:9

Non-linear Autoencoders

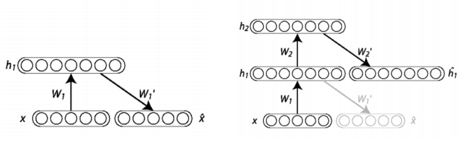
- PCA given the "optimal linear autoencode"
- We can relax the encoding (V_p) and decoding ($V_p^\top$) to be non-linear mappings, e.g., represented as a neural network



A NN which is trained to reproduce the input: $\min_i \|y(x_i) - x_i\|^2$

The hidden layer ("bottleneck") needs to find a good representation/compression.

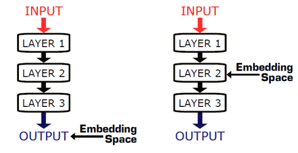
- Stacking autoencoders:



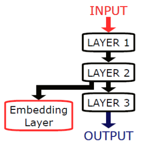
6:10

Augmenting NN training with semi-supervised embedding objectives

- Weston et al. (ICML, 2008)



(a) Output (b) Internal



(c) Auxiliary

Mnist1h dataset, deep NNs of 2, 6, 8, 10 and 15 layers; each hidden layer 50 hidden units

	2	4	6	8	10	15
NN	26.0	26.1	27.2	28.3	34.2	47.7
$Embed^{O}$NN	19.7	15.1	15.1	15.0	13.7	11.8
$Embed^{ALL}$NN	18.2	12.6	7.9	8.5	6.3	9.3

6:11

What are good representations?

- Reproducing/autoencoding data, maintaining maximal information
- Disentangling correlations (e.g., ICA)
- those that are most correlated with desired *outputs* (PLS, NNs)
- those that maintain the clustering
- those that maintain relative distances (MDS)

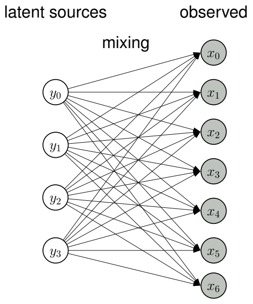
...

- those that enable efficient reasoning, decision making & learning in the real world
- How do we represent our 3D environment, enabeling physical & geometric reasoning?
- How do we represent things to enable us inventing novel things, machines, technology, science?

6:12

Independent Component Analysis**

- Assume we have data $D = \{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$.
  PCA: $P(x_i \,|\, z_i) = \mathcal{N}(x_i \,|\, W z_i + \mu, \mathbf{I})$, $\quad P(z_i) = \mathcal{N}(z_i \,|\, 0, \mathbf{I})$
  Factor Analysis: $P(x_i \,|\, z_i) = \mathcal{N}(x_i \,|\, W z_i + \mu, \Sigma)$, $\quad P(z_i) = \mathcal{N}(z_i \,|\, 0, \mathbf{I})$
  ICA: $P(x_i \,|\, z_i) = \mathcal{N}(x_i \,|\, W z_i + \mu, \epsilon\mathbf{I})$, $\quad P(z_i) = \prod_{j=1}^d P(z_{ij})$



- In ICA
  1) We have (usually) as many latent variables as observed $\dim(x_i) = \dim(z_i)$
  2) We require all latent variables to be **independent**
  3) We allow for latent variables to be **non-Gaussian**

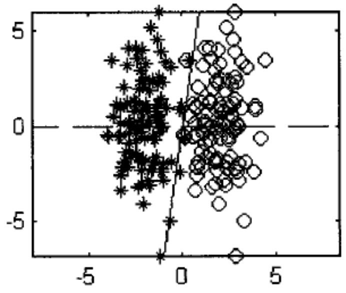
  Note: without point (3) ICA would be without sense!

6:13

Partial least squares (PLS)**

- Is it really a good idea to just pick the p-higest variance components??

  Why should that be a good idea?



6:14

PLS*

- Idea: The first dimension to pick should be the one **most correlated with the OUTPUT**, not with itself!

```
Input: data $X \in \mathbb{R}^{n\times d}$, $y \in \mathbb{R}^n$
Output: predictions $\hat y \in \mathbb{R}^n$
1: initialize the predicted output: $\hat y = \langle y \rangle\, 1_n$
2: initialize the remaining input dimensions: $\hat X = X$
3: for $i = 1, .., p$ do
4:     $i$-th input 'basis vector': $z_i = \hat X \hat X^\top y$
5:     update prediction: $\hat y \leftarrow \hat y + Z_i y$   where $Z_i = \frac{z_i z_i^\top}{z_i^\top z_i}$
6:     remove "used" input dimensions: $\hat X \leftarrow \hat X(\mathbf{I} - Z_i)$
7: end for
```

(Hastie, page 81)

Line 4 identifies a new input "coordinate" via maximal correlation between the remaning input dimensions and y.

Line 5 updates the prediction to include the project of y onto z_i

Line 6 removes the projection of input data $\hat X$ along z_i. All z_i will be orthogonal.

6:15

PLS for classification*

- Not obvious.
- We'll try to invent one in the exercises :-)

6:16

- back to linear autoencoding, i.e., PCA - but now linear in RKHS

6:17

"Feature PCA" & Kernel PCA**

- The *feature* trick: $X = \begin{pmatrix} \phi(x_1)^\top \\ \vdots \\ \phi(x_n)^\top \end{pmatrix} \in \mathbb{R}^{n\times k}$
- The *kernel* trick: rewrite all necessary equations such that they only involve scalar products $\phi(x)^\top\phi(x') = k(x,x')$:

We want to compute eigenvectors of $X^\top X = \sum_i \phi(x_i)\phi(x_i)^\top$. We can rewrite this as

$$X^\top X v_j = \lambda v_j$$

$$\underbrace{XX^\top}_{K} \underbrace{Xv_j}_{K\alpha_j} = \lambda \underbrace{Xv_j}_{K\alpha_j} \;, \quad v_j = \sum_i \alpha_{ji}\phi(x_i)$$

$$K\alpha_j = \lambda\alpha_j$$

Where $K = XX^\top$ with entries $K_{ij} = \phi(x_i)^\top\phi(x_j)$.

$\to$ We compute SVD of the kernel matrix K $\to$ gives eigenvectors $\alpha_j \in \mathbb{R}^n$.

Projection: $x \mapsto z = V_p^\top \phi(x) = \sum_i \alpha_{1:p,i} \phi(x_i)^\top \phi(x) = A\kappa(x)$

(with matrix $A \in \mathbb{R}^{p\times n}$, $A_{ji} = \alpha_{ji}$ and vector $\kappa(x) \in \mathbb{R}^n$, $\kappa_i(x) = k(x_i, x)$)

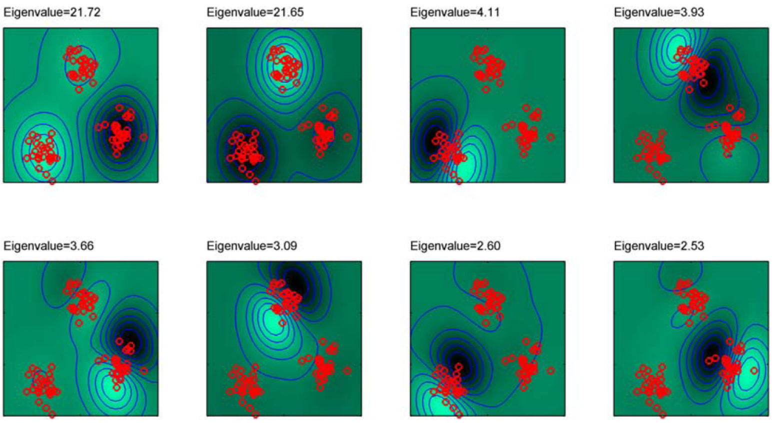
Since we cannot *center the features* $\phi(x)$ we actually need "the double centered kernel matrix" $\widetilde{K} = (\mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}^\top)K(\mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}^\top)$, where $K_{ij} = \phi(x_i)^\top\phi(x_j)$ is uncentered.

6:18

Kernel PCA

red points: data

green shading: eigenvector $\boldsymbol{\alpha}_j$ represented as functions $\sum_i \alpha_{ji} k(x_j, x)$



Kernel PCA "coordinates" allow us to discriminate clusters!

6:19

Kernel PCA

- Kernel PCA uncovers quite surprising structure:

  While PCA "merely" picks high-variance dimensions

  Kernel PCA picks high variance *features*—where features correspond to basis functions (RKHS elements) over x

- Kernel PCA may map data x_i to latent coordinates z_i where *clustering* is much easier

- All of the following can be represented as kernel PCA:
  – Local Linear Embedding
  – Metric Multidimensional Scaling

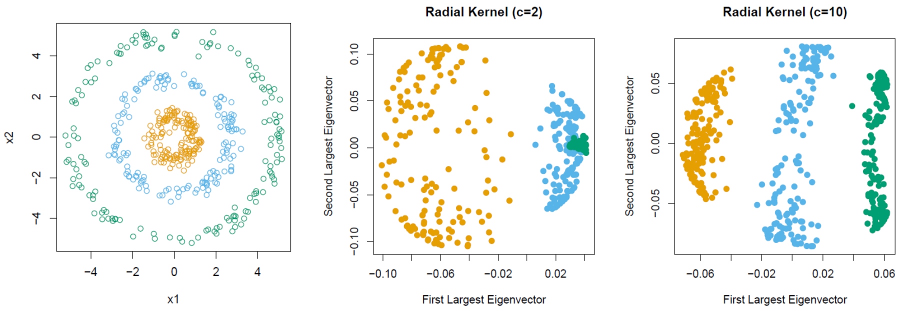
– Laplacian Eigenmaps (Spectral Clustering)

see "Dimensionality Reduction: A Short Tutorial" by Ali Ghodsi

6:20

Kernel PCA clustering

- Using a kernel function $k(x, x') = e^{-\|x-x'\|^2/c}$:



- Gaussian mixtures or k-means will easily cluster this

6:21

Spectral Clustering**

Spectral Clustering is very similar to kernel PCA:

- Instead of the kernel matrix K with entries $k_{ij} = k(x_i, x_j)$ we construct a weighted *adjacency matrix*, e.g.,

$$w_{ij} = \begin{cases} 0 & \text{if } x_i \text{ are not a } k\text{NN of } x_j \\ e^{-\|x_i-x_j\|^2/c} & \text{otherwise} \end{cases}$$

w_{ij} is the weight of the *edge* between data point x_i and x_j.

- Instead of computing *maximal* eigenvectors of $\widetilde{K}$, compute *minimal* eigenvectors of

$$L = \mathbf{I} - \widetilde{W} \ , \quad \widetilde{W} = \text{diag}(\textstyle\sum_j w_{ij})^{-1} W$$

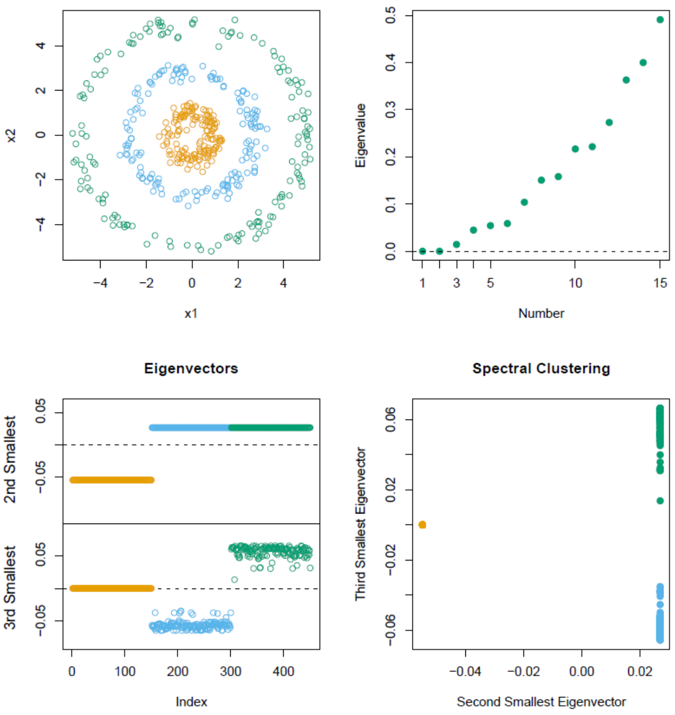
($\sum_j w_{ij}$ is called *degree of node i*, $\widetilde{W}$ is the normalized weighted adjacency matrix)

6:22

- Given $L = UDV^\top$, we pick the p smallest eigenvectors $V_p = V_{1:n,1:p}$ (perhaps exclude the trivial smallest eigenvector)

- The latent coordinates for x_i are $z_i = V_{i,1:p}$
- Spectral Clustering provides a method to compute latent low-dimensional coordinates $z_i = V_{i,1:p}$ for each high-dimensional $x_i \in \mathbb{R}^d$ input.
- This is then followed by a standard clustering, e.g., Gaussian Mixture or k-means

6:23



6:24

- Spectral Clustering is similar to kernel PCA:
  – The kernel matrix K usually represents similarity
    The weighted adjacency matrix W represents proximity & similarity
  – High Eigenvectors of K are similar to low EV of $L = \mathbf{I} - W$
- Original interpretation of Spectral Clustering:
  – $L = \mathbf{I} - W$ (weighted graph Laplacian) describes a diffusion process:
    The diffusion rate W_{ij} is high if i and j are close and similar

– Eigenvectors of L correspond to stationary solutions

- The Graph Laplacian L: For some vector $f \in \mathbb{R}^n$, note the following identities:

$$\begin{aligned}(Lf)_i &= (\sum_j w_{ij}) f_i - \sum_j w_{ij} f_j = \sum_j w_{ij}(f_i - f_j) \\ f^\top L f &= \sum_i f_i \sum_j w_{ij}(f_i - f_j) = \sum_{ij} w_{ij}(f_i^2 - f_i f_j) \\ &= \sum_{ij} w_{ij}(\frac{1}{2} f_i^2 + \frac{1}{2} f_j^2 - f_i f_j) = \frac{1}{2}\sum_{ij} w_{ij}(f_i - f_j)^2\end{aligned}$$

where the second-to-last = holds if $w_{ij} = w_{ji}$ is symmetric.

6:25

Metric Multidimensional Scaling**

- Assume we have data $D = \{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$.
  As before we want to indentify latent lower-dimensional representations $z_i \in \mathbb{R}^p$ for this data.

- A simple idea: Minimize the stress

$$S_C(z_{1:n}) = \sum_{i \neq j}(d_{ij}^2 - \|z_i - z_j\|^2)^2$$

We want distances in high-dimensional space to be equal to distances in low-dimensional space.

6:26

Metric Multidimensional Scaling = (kernel) PCA

- Note the relation:

$$d_{ij}^2 = \|x_i - x_j\|^2 = \|x_i - \bar{x}\|^2 + \|x_j - \bar{x}\|^2 - 2(x_i - \bar{x})^\top (x_j - \bar{x})$$

This translates a distance into a (centered) scalar product

- If may we define

$$\widetilde{K} = (\mathbf{I} - \tfrac{1}{n}\mathbf{1}\mathbf{1}^\top) D (\mathbf{I} - \tfrac{1}{n}\mathbf{1}\mathbf{1}^\top) \ , \quad D_{ij} = -d_{ij}^2/2$$

then $\widetilde{K_{ij}} = (x_i - \bar{x})^\top (x_j - \bar{x})$ is the normal covariance matrix and MDS is equivalent to kernel PCA

6:27

Non-metric Multidimensional Scaling

- We can do this for any data (also non-vectorial or not $\in \mathbb{R}^d$) as long as we have a data set of comparative dissimilarities d_{ij}

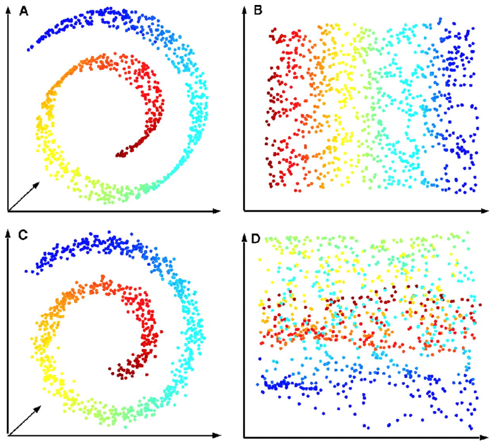
$$S(z_{1:n}) = \sum_{i \neq j}(d_{ij}^2 - |z_i - z_j|^2)^2$$

- Minimize $S(z_{1:n})$ w.r.t. $z_{1:n}$ *without any further constraints*!

6:28

Example for Non-Metric MDS: ISOMAP**

- Construct kNN graph and label edges with Euclidean distance
  – Between any two x_i and x_j, compute "geodescic" distance d_{ij} (shortest path along the graph)
  – Then apply MDS



by Tenenbaum et al.

6:29

The zoo of dimensionality reduction methods

- PCA family:
  – kernel PCA, non-neg. Matrix Factorization, Factor Analysis

- All of the following can be represented as kernel PCA:
  – Local Linear Embedding
  – Metric Multidimensional Scaling
  – Laplacian Eigenmaps (Spectral Clustering)

They all use different notions of distance/correlation as input to kernel PCA

see "Dimensionality Reduction: A Short Tutorial" by Ali Ghodsi

6:30

PCA variants*

6:31

PCA variant: Non-negative Matrix Factorization**

- Assume we have data $D = \{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$.
  As for PCA (where we had $x_i \approx V_p z_i + \mu$) we search for a lower-dimensional space with linear relation to x_i

- In NMF we require everything is **non-negative**: the data x_i, the projection W, and latent variables z_i
  Find $W \in \mathbb{R}^{p\times d}$ (the tansposed projection) and $Z \in \mathbb{R}^{n\times p}$ (the latent variables z_i) such that

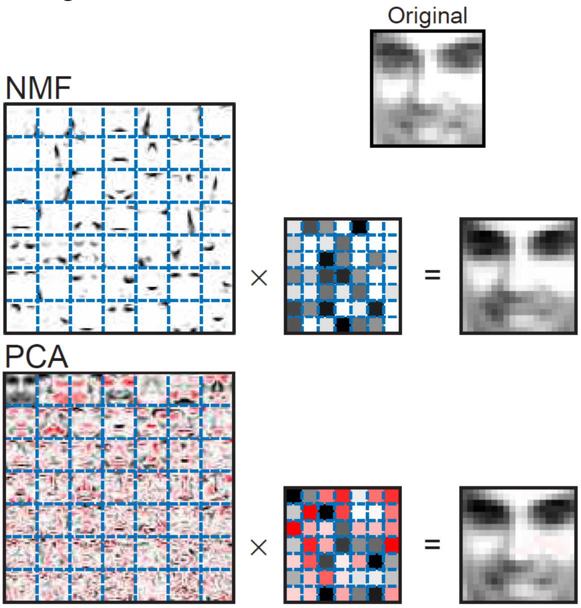
$$X \approx ZW$$

- Iterative solution: (E-step and M-step like...)

$$z_{ik} \leftarrow z_{ik} \frac{\sum_{j=1}^d w_{kj} x_{ij}/(ZW)_{ij}}{\sum_{j=1}^d w_{kj}}$$
$$w_{kj} \leftarrow w_{kj} \frac{\sum_{i=1}^N z_{ik} x_{ij}/(ZW)_{ij}}{\sum_{i=1}^N z_{ik}}$$

6:32

PCA variant: Non-negative Matrix Factorization*



(from Hastie 14.6)

6:33

PCA variant: Factor Analysis**

Another variant of PCA: (Bishop 12.64)

Allows for different noise in each dimension $P(x_i \,|\, z_i) = \mathcal{N}(x_i \,|\, V_p z_i + \mu, \Sigma)$ (with Σ diagonal)

6:34

6.2 Clustering

6:35

Clustering

- Clustering often involves two steps:
- First map the data to some embedding that emphasizes clusters
  - (Feature) PCA
  - Spectral Clustering
  - Kernel PCA
  - ISOMAP
- Then explicitly analyze clusters
  - k-means clustering
  - Gaussian Mixture Model
  - Agglomerative Clustering

6:36

k-means Clustering

- Given data $D = \{x_i\}_{i=1}^n$, find K centers μ_k, and a data assignment $c : i \mapsto k$ to minimize

$$\min_{c,\mu} \sum_i (x_i - \mu_{c(i)})^2$$

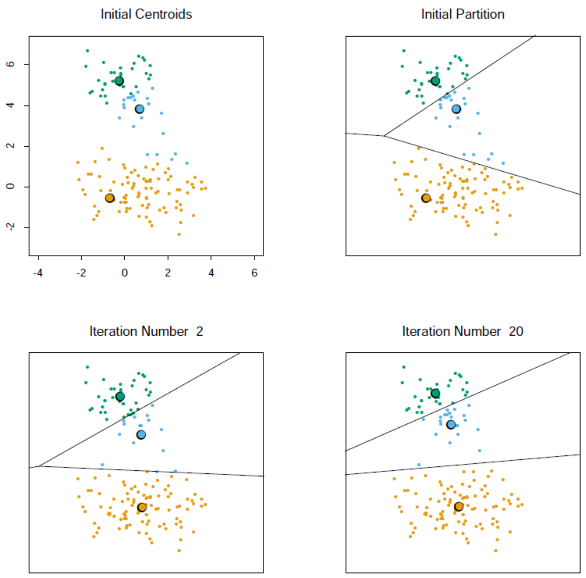
- k-means clustering:
  - Pick K data points randomly to initialize the centers μ_k
  - Iterate adapting the assignments $c(i)$ and the centers μ_k:

$$\forall_i: \; c(i) \leftarrow \operatorname*{argmin}_{c(i)} \sum_j (x_j - \mu_{c(j)})^2 = \operatorname*{argmin}_k (x_i - \mu_k)^2$$

$$\forall_k: \; \mu_k \leftarrow \operatorname*{argmin}_{\mu_k} \sum_i (x_i - \mu_{c(i)})^2 = \frac{1}{|c^{-1}(k)|} \sum_{i \in c^{-1}(k)} x_i$$

6:37

k-means Clustering



from Hastie

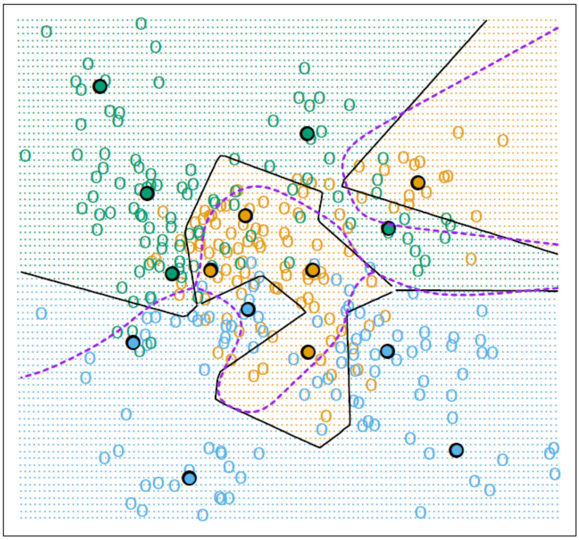
6:38

k-means Clustering

- Converges to local minimum $\rightarrow$ **many restarts**
- Choosing k? Plot $L(k) = \min_{c,\mu} \sum_i (x_i - \mu_{c(i)})^2$ for different k – choose a trade-off between model complexity (large k) and data fit (small loss $L(k)$)

6:39

k-means Clustering for Classification



from Hastie

6:40

Gaussian Mixture Model for Clustering

- GMMs can/should be introduced as *generative* probabilistic model of the data:
  - K different Gaussians with parmameters μ_k, Σ_k
  - Assignment RANDOM VARIABLE $c_i \in \{1, .., K\}$ with $P(c_i = k) = \pi_k$
  - The observed data point x_i with $P(x_i \mid c_i = k; \mu_k, \Sigma_k) = \mathcal{N}(x_i \mid \mu_k, \Sigma_k)$
- EM-Algorithm described as a kind of soft-assignment version of k-means
  - Initialize the centers $\mu_{1:K}$ randomly from the data; all covariances $\Sigma_{1:K}$ to unit and all π_k uniformly.
  - **E-step:** (probabilistic/soft assignment) Compute

$$q(c_i = k) = P(c_i = k \mid x_i, \mu_{1:K}, \Sigma_{1:K}) \propto \mathcal{N}(x_i \mid \mu_k, \Sigma_k)\ \pi_k$$

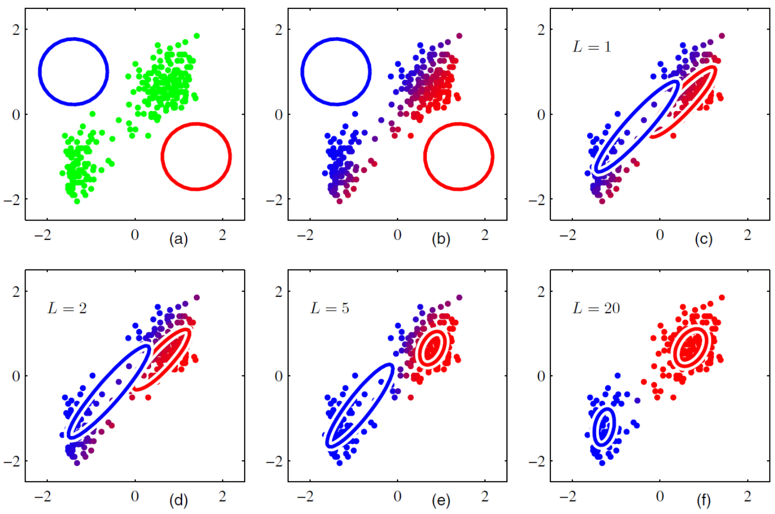
  - **M-step:** Update parameters (centers AND covariances)

$$\begin{aligned}
\pi_k &= \frac{1}{n} \textstyle\sum_i q(c_i = k) \\
\mu_k &= \frac{1}{n\pi_k}\ \textstyle\sum_i q(c_i = k)\ x_i \\
\Sigma_k &= \frac{1}{n\pi_k}\ \textstyle\sum_i q(c_i = k)\ x_i x_i^\top - \mu_k \mu_k^\top
\end{aligned}$$

6:41

Gaussian Mixture Model

EM iterations for Gaussian Mixture model:



from Bishop

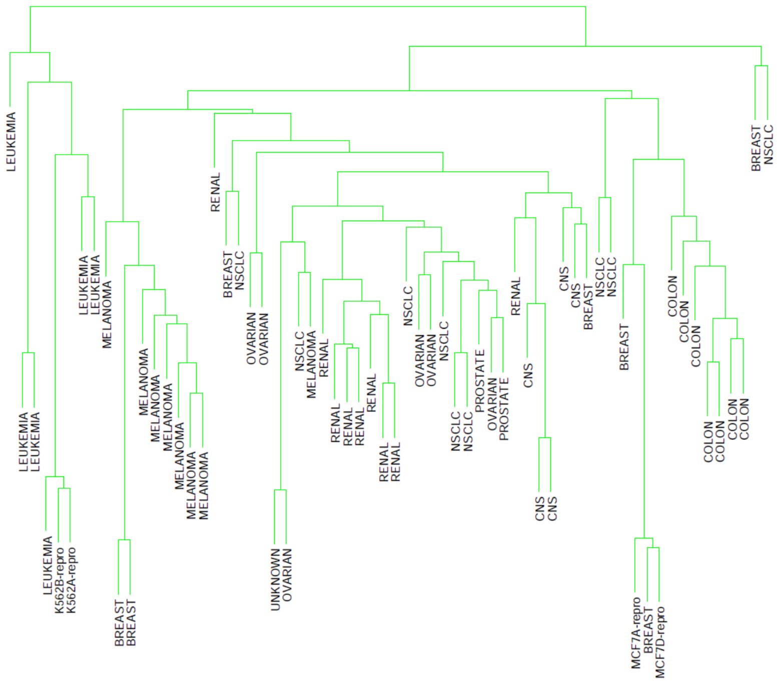
6:42

Agglomerative Hierarchical Clustering

- *agglomerative* = bottom-up, *divisive* = top-down
- Merge the two groups with the smallest intergroup dissimilarity
- Dissimilarity of two groups G, H can be measures as
  – Nearest Neighbor (or "single linkage"): $d(G, H) = \min_{i\in G, j\in H} d(x_i, x_j)$
  – Furthest Neighbor (or "complete linkage"): $d(G, H) = \max_{i\in G, j\in H} d(x_i, x_j)$
  – Group Average: $d(G, H) = \frac{1}{|G||H|} \sum_{i\in G} \sum_{j\in H} d(x_i, x_j)$

6:43

Agglomerative Hierarchical Clustering



Appendix: Centering & Whitening

- Some prefer to *center* (shift to zero mean) the data before applying methods:

$$x \leftarrow x - \langle x \rangle \ , \quad y \leftarrow y - \langle y \rangle$$

  this spares augmenting the bias feature 1 to the data.

- More interesting: The loss and the best choice of λ depends on the *scaling* of the data. If we always scale the data in the same range, we may have better priors about choice of λ and interpretation of the loss

$$x \leftarrow \frac{1}{\sqrt{\text{Var}\{x\}}}\, x \ , \quad y \leftarrow \frac{1}{\sqrt{\text{Var}\{y\}}}\, y$$

- **Whitening:** Transform the data to remove all correlations and variances.
  Let $A = \text{Var}\{x\} = \frac{1}{n} X^\top X - \mu\mu^\top$ with Cholesky decomposition $A = MM^\top$.

$$x \leftarrow M^{\text{-}1} x \ , \quad \text{with } \text{Var}\{M^{\text{-}1} x\} = \mathbf{I}_d$$

7 Local Learning & Emsemble Learning

Local learning & Ensemble learning

- "Simpler is Better"
  - We've learned about [kernel] ridge — logistic regression
  - We've learned about high-capacity NN training
  - Sometimes one should consider also much simpler methods as baseline

- Content:
  - Local learners
    - local & lazy learning, kNN, Smoothing Kernel, kd-trees
  - Combining weak or randomized learners
    - Bootstrap, bagging, and model averaging
    - Boosting
    - (Boosted) decision trees & stumps, random forests

7:1

7.1 Local & lazy learning

7:2

Local & lazy learning

- Idea of local (or "lazy") learning:
  Do not try to build one global model $f(x)$ from the data. Instead, whenever we have a query point x^*, we build a specific local model in the neighborhood of x^*.

- Typical approach:
  – Given a query point x^*, find all kNN in the data $D = \{(x_i, y_i)\}_{i=1}^N$
  – Fit a local model f_{x^*} only to these kNNs, perhaps weighted
  – Use the local model f_{x^*} to predict $x^* \mapsto \hat y_0$

- **Weighted local least squares:**

$$L^{\text{local}}(\beta, x^*) = \textstyle\sum_{i=1}^n K(x^*, x_i)(y_i - f(x_i))^2 + \lambda \|\beta\|^2$$

  where $K(x^*, x)$ is called **smoothing kernel**. The optimum is:

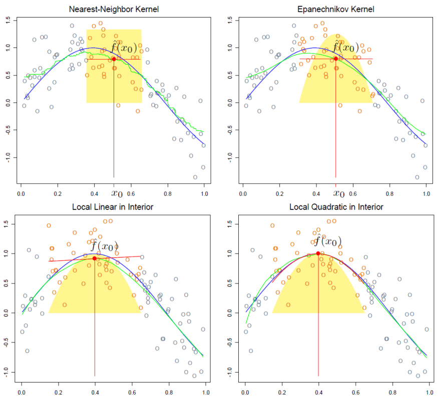
$$\hat\beta = (X^\top W X + \lambda I)^{\text{-1}} X^\top W y \;, \quad W = \text{diag}(K(x^*, x_{1:n}))$$

7:3

Regression example

kNN smoothing kernel: $K(x^*, x_i) = \begin{cases} 1 & \text{if } x_i \in \text{kNN}(x^*) \\ 0 & \text{otherwise} \end{cases}$

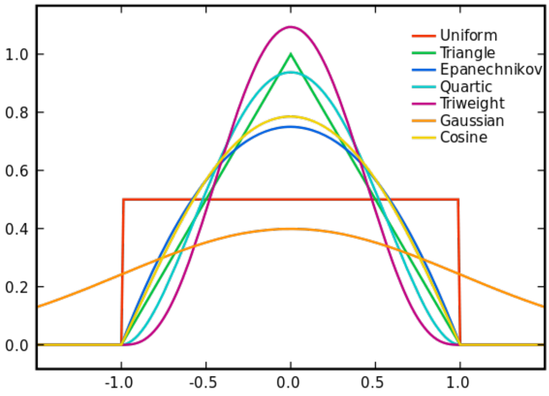
Epanechnikov quadratic smoothing kernel: $K_\lambda(x^*, x) = D(|x^* - x|/\lambda)$, $D(s) = \begin{cases} \frac{3}{4}(1-s^2) & \text{if } s \le 1 \\ 0 & \text{otherwise} \end{cases}$



(Hastie, Sec 6.3)

7:4

Smoothing Kernels



from Wikipedia

7:5

Which metric to use for NN?

- This is *the* crutial question? The fundamental question of generalization.
  - Given a query x^*, which data points x_i would you consider as being "related", so that the label of x_i is correlated to the correct label of x^*?

- Possible answers beyond naive Euclidean distance $|x^* - x_i|$
  - Some other kernel function $k(x^*, x_i)$
  - First encode x into a "meaningful" latent representation z; then use Euclidean distance there
  - Take some off-the-shelf pretrained image NN, chop of the head, use this internal representation

7:6

---

kd-trees

- For local & lazy learning it is essential to efficiently retrieve the kNN

  Problem: Given data X, a query x^*, identify the kNNs of x^* in X.

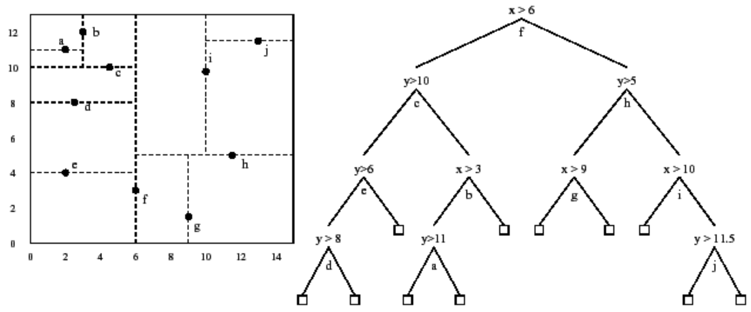
- Linear time (stepping through all of X) is far too slow.

  A kd-tree pre-structures the data into a binary tree, allowing $O(\log n)$ retrieval of kNNs.

7:7

---

kd-trees



(There are "typos" in this figure... Exercise to find them.)

- Every node plays two roles:
  – it defines a hyperplane that separates the data along *one* coordinate
  – it hosts a data point, which lives exactly on the hyperplane (defines the division)

7:8

---

kd-trees

- Simplest (non-efficient) way to construct a kd-tree:
  – hyperplanes divide alternatingly along 1st, 2nd, ... coordinate

– choose random point, use it to define hyperplane, divide data, iterate

- Nearest neighbor search:
  – descent to a leave node and take this as initial nearest point
  – ascent and check at each branching the possibility that a nearer point exists on the other side of the hyperplane

- Approximate Nearest Neighbor (libann on Debian..)

7:9

7.2 Combining weak and randomized learners

7:10

Combining learners

- The general idea is:
  – Given data D, let us learn *various models* $f_1, .., f_M$
  – Our prediction is then some combination of these, e.g.

$$f(x) = \sum_{m=1}^{M} \alpha_m f_m(x)$$

- *"Various models"* could be:

  Model averaging: Fully different types of models (using different (e.g. limited) feature sets; neural nets; decision trees; hyperparameters)

  Bootstrap: Models of same type, trained on randomized versions of D

  Boosting: Models of same type, trained on cleverly designed modifications/reweight of D

- How can we choose the α_m? (You should know that!)

7:11

Bootstrap & Bagging

- **Bootstrap:**
  – Data set D of size n
  – Generate M data sets D_m by resampling D *with replacement*
  – Each D_m is also of size n (some samples doubled or missing)

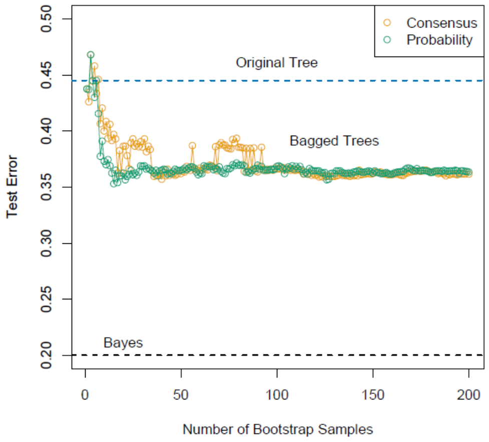
– Distribution over data sets $\leftrightarrow$ distribution over β (compare slide 02:13)
– The ensemble $\{f_1, .., f_M\}$ is similar to cross-validation
– Mean and variance of $\{f_1, .., f_M\}$ can be used for model assessment

- **Bagging:** ("bootstrap aggregation")

$$f(x) = \frac{1}{M} \sum_{m=1}^M f_m(x)$$

7:12

- Bagging has similar effect to regularization:



(Hastie, Sec 8.7)
7:13

Bayesian Model Averaging

- If $f_1, .., f_M$ are very different models
  – Equal weighting would not be clever
  – More confident models (less variance, less parameters, higher likelihood)
  $\rightarrow$ higher weight

- Bayesian Averaging

$$P(y|x) = \sum_{m=1}^M P(y|x, f_m, D)\ P(f_m|D)$$

The term $P(f_m|D)$ is the weighting α_m: it is high, when the model is likely under the data ($\leftrightarrow$ the data is likely under the model & the model has "fewer parameters").

7:14

The basis function view: Models are features!

- Compare model averaging $f(x) = \sum_{m=1}^M \alpha_m f_m(x)$ with regression:

$$f(x) = \sum_{j=1}^k \phi_j(x)\, \beta_j = \phi(x)^\top \beta$$

- We can think of the M models f_m as **features** ϕ_j for linear regression!
  – We know how to find optimal parameters α
  – But beware overfitting!

7:15

Boosting

- In Bagging and Model Averaging, the models are trained on the "same data" (unbiased randomized versions of the same data)

- Boosting tries to be cleverer:
  – It adapts the data for each learner
  – It assigns each learner a differently *weighted* version of the data

- With this, boosing can
  – *Combine many "weak" classifiers to produce a powerful "committee"*
  – A weak learner only needs to be somewhat better than random

7:16

AdaBoost**

(Freund & Schapire, 1997)

(classical Algo; use Gradient Boosting instead in practice)

- Binary classification problem with data $D = \{(x_i, y_i)\}_{i=1}^n$, $y_i \in \{-1, +1\}$
- We know how to train discriminative functions $f(x)$; let

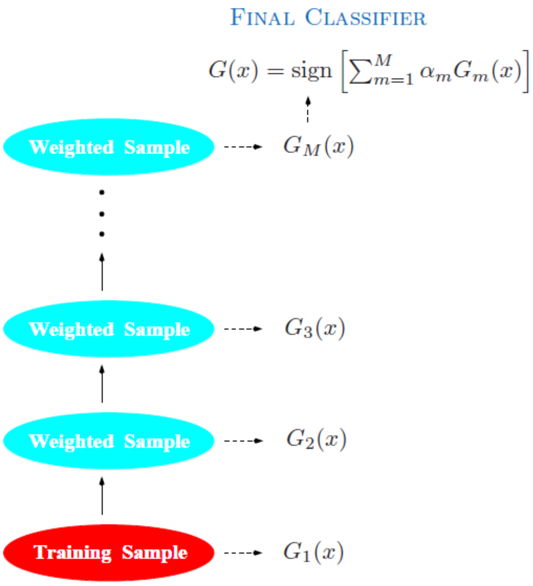
$$G(x) = \text{sign}\, f(x) \quad \in \{-1, +1\}$$

- We will train a sequence of classificers $G_1, .., G_M$, each on differently weighted data, to yield a classifier

$$G(x) = \text{sign} \sum_{m=1}^M \alpha_m G_m(x)$$

7:17

AdaBoost**



(Hastie, Sec 10.1)

7:18

AdaBoost**

```
Input:  data $D = \{(x_i, y_i)\}_{i=1}^n$
Output: family of $M$ classifiers $G_m$ and weights $\alpha_m$
1: initialize $\forall_i : w_i = 1/n$
2: for $m = 1, .., M$ do
3:     Fit classifier $G_m$ to the training data weighted by $w_i$
4:     $\text{err}_m = \frac{\sum_{i=1}^n w_i \, [y_i \neq G_m(x_i)]}{\sum_{i=1}^n w_i}$
5:     $\alpha_m = \log[\frac{1-\text{err}_m}{\text{err}_m}]$
6:     $\forall_i : w_i \leftarrow w_i \, \exp\{\alpha_m \, [y_i \neq G_m(x_i)]\}$
7: end for
```

(Hastie, sec 10.1)

Weights unchanged for correctly classified points
Multiply weights with $\frac{1-\text{err}_m}{\text{err}_m} > 1$ for mis-classified data points

- *Real AdaBoost:* A variant exists that combines probabilistic classifiers $\sigma(f(x)) \in [0,1]$ instead of discrete $G(x) \in \{-1,+1\}$

7:19

The basis function view

- In AdaBoost, each model G_m depends on the data weights w_m
  We could write this as

$$f(x) = \sum_{m=1}^M \alpha_m f_m(x, w_m)$$

  The "features" $f_m(x, w_m)$ now have additional parameters w_m
  We'd like to optimize

$$\min_{\alpha, w_1,..,w_M} L(f)$$

  w.r.t. α and all the feature parameters w_m.

- In general this is hard.
  But assuming $\alpha_{\hat m}$ and $w_{\hat m}$ fixed, optimizing for α_m and w_m is efficient.

- AdaBoost does exactly this, choosing w_m so that the "feature" f_m will best reduce the loss (cf. PLS)
  (Literally, AdaBoost uses exponential loss or neg-log-likelihood; Hastie sec 10.4 & 10.5)

7:20

Gradient Boosting

- AdaBoost generates a series of basis functions by using different data weightings w_m depending on so-far classification errors
- We can also generate a series of basis functions f_m by fitting them to the gradient of the so-far loss

7:21

Gradient Boosting

- Assume we want to miminize some loss function

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

We can solve this using gradient descent

$$f^* = f_0 + \alpha_1 \underbrace{\frac{\partial L(f_0)}{\partial f}}_{\approx f_1} + \alpha_2 \underbrace{\frac{\partial L(f_0 + \alpha_1 f_1)}{\partial f}}_{\approx f_2} + \alpha_3 \underbrace{\frac{\partial L(f_0 + \alpha_1 f_1 + \alpha_2 f_2)}{\partial f}}_{\approx f_3} + \cdots$$

– Each f_m aproximates the so-far loss gradient

– We use linear regression to choose α_m (instead of line search)

- More intuitively: $\frac{\partial L(f)}{\partial f}$ "points into the direction of the error/redisual of f". It shows how f could be improved.

  Gradient boosting uses the next lerner $f_k \approx \frac{\partial L(f_{\text{so far}})}{\partial f}$ to approximate how f can be improved.

  Optimizing α's does the improvement.

7:22

Gradient Boosting

```
Input:  function class $\mathcal{F}$ (e.g., of discriminative functions), data $D = \{(x_i, y_i)\}_{i=1}^n$,
        an arbitrary loss function $L(y, \hat y)$
Output: function $\hat f$ to minimize $\sum_{i=1}^n L(y_i, f(x_i))$
1: Initialize a constant $\hat f = f_0 = \text{argmin}_{f \in \mathcal{F}} \sum_{i=1}^n L(y_i, f(x_i))$
2: for $m = 1 : M$ do
3:     For each data point $i = 1 : n$ compute $r_{im} = -\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\big|_{f=\hat f}$
4:     Fit a regression $f_m \in \mathcal{F}$ to the targets $r_{im}$, minimizing squared error
5:     Find optimal coefficients (e.g., via feature logistic regression)
               $\alpha = \text{argmin}_\alpha \sum_i L(y_i, \sum_{j=0}^m \alpha_m f_m(x_i))$
                        (often: fix $\alpha_{0:m\text{-}1}$ and only optimize over $\alpha_m$)
6:     Update $\hat f = \sum_{j=0}^m \alpha_m f_m$
7: end for
```

- If $\mathcal{F}$ is the set of regression/decision trees, then step 5 usually re-optimizes the terminal constants of all leave nodes of the regression tree f_m. (Step 4 only determines the terminal regions.)

7:23

Gradient boosting is the preferred method

- Hastie's book quite "likes" gradient boosting
  - Can be applied to any loss function
  - No matter if regression or classification
  - Very good performance
  - Simpler, more general, better than AdaBoost

7:24

Classical examples for boosting

7:25

Decision Trees**

- Decision trees are particularly used in Bagging and Boosting contexts
- Decision trees are "linear in features", but the features are the terminal regions of a tree, which are constructed depending on the data
- We'll learn about
  – Boosted decision trees & stumps
  – Random Forests

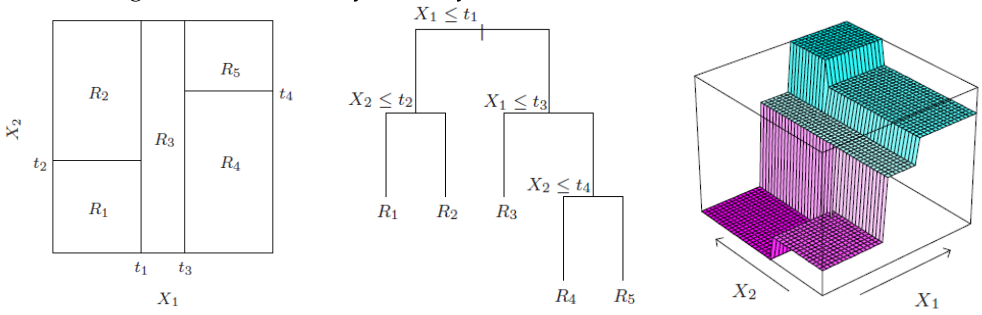
7:26

Decision Trees

- We describe CART (classification and regression tree)
- Decision trees are linear in features:

$$f(x) = \sum_{j=1}^{k} c_j \, [x \in R_j]$$

  where R_j are disjoint rectangular regions and c_j the constant prediction in a region
- The regions are defined by a binary decision tree



7:27

Growing the decision tree

- The constants are the region averages $c_j = \frac{\sum_i y_i \, [x_i \in R_j]}{\sum_i [x_i \in R_j]}$
- Each split $x_a > t$ is defined by a choice of input dimension $a \in \{1, .., d\}$ and a threshold t

- Given a yet unsplit region R_j, we split it by choosing

$$\min_{a,t} \Big[\min_{c_1} \sum_{i:x_i \in R_j \wedge x_a \le t} (y_i - c_1)^2 + \min_{c_2} \sum_{i:x_i \in R_j \wedge x_a > t} (y_i - c_2)^2 \Big]$$

  - Finding the threshold t is really quick (slide along)
  - We do this for every input dimension a

7:28

Deciding on the depth (if not pre-fixed)

- We first grow a very large tree (e.g. until at most 5 data points live in each region)

- Then we rank all nodes using "weakest link pruning":
  Iteratively remove the node that least increases

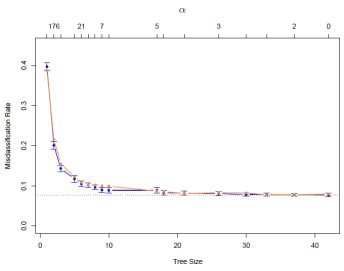
$$\sum_{i=1}^n (y_i - f(x_i))^2$$

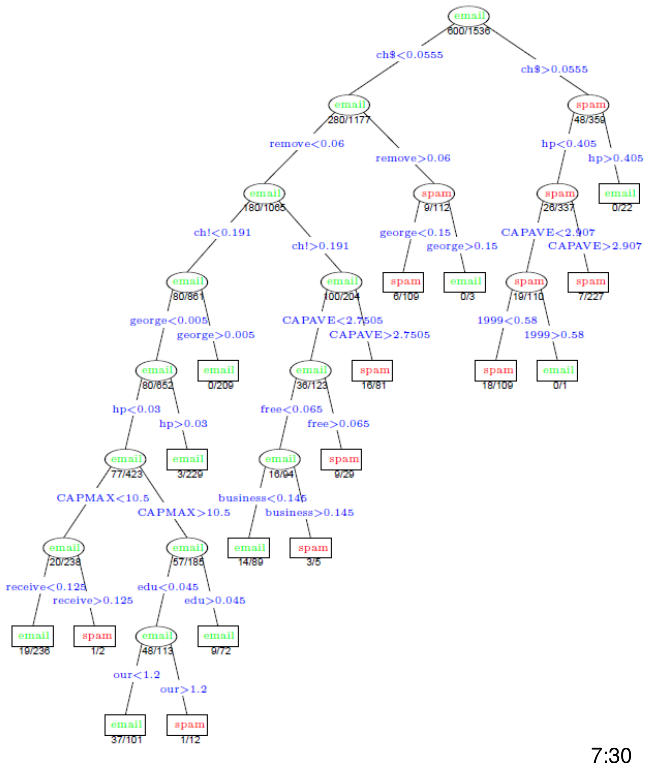
- Use cross-validation to choose the eventual level of pruning

This is equivalent to choosing a regularization parameter λ for
$L(T) = \sum_{i=1}^n (y_i - f(x_i))^2 \; + \; \lambda |T|$
where the regularization $|T|$ is the tree size

7:29

Example:
CART on the Spam data set
(details: Hastie, p 320)





	Predicted	
True	email	spam
email	57.3%	4.0%
spam	5.3%	33.4%

Test error rate: 8.7%

7:30

Boosting trees & stumps**

- A **decision stump** is a decision tree with fixed depth 1 (just one split)
- Gradient boosting of decision trees (of fixed depth J) and stumps is very effective

  Test error rates on Spam data set:

full decision tree	8.7%
boosted decision stumps	4.7%
boosted decision trees with $J = 5$	4.5%

7:31

Random Forests: Bootstrapping & randomized splits**

- Recall that Bagging averages models $f_1, .., f_M$ where each f_m was trained on a bootstrap resample D_m of the data D

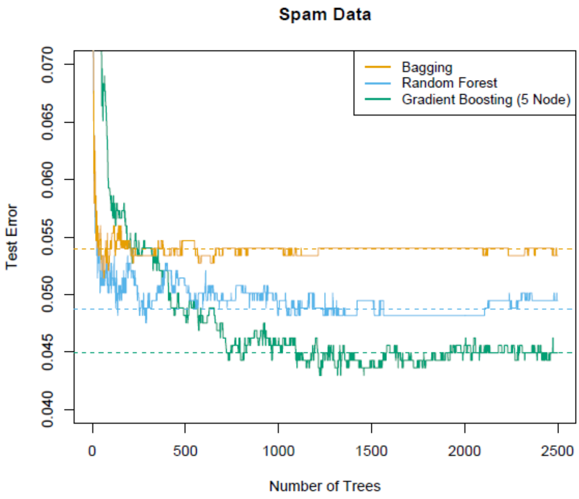
This randomizes the models and avoids over-generalization

- Random Forests do Bagging, but additionally randomize the trees:
  - When growing a new split, choose the input dimension a only from a *random subset* m features
  - m is often very small; even $m = 1$ or $m = 3$

- Random Forests are the prime example for "creating many randomized weak learners from the same data D"

7:32

Random Forests vs. gradient boosted trees



(Hastie, Fig 15.1)

7:33

8 Probabilistic Machine Learning

Learning as Inference

- The parameteric view

$$P(\beta|\text{Data}) = \frac{P(\text{Data}|\beta)\ P(\beta)}{P(\text{Data})}$$

- The function space view

$$P(f|\text{Data}) = \frac{P(\text{Data}|f)\ P(f)}{P(\text{Data})}$$

- Today:
  – Bayesian (Kernel) Ridge Regression $\leftrightarrow$ Gaussian Process (GP)
  – Bayesian (Kernel) Logistic Regression $\leftrightarrow$ GP classification
  – Bayesian Neural Networks (briefly)

8:1

- Beyond learning about specific Bayesian learning methods:

  Understand relations between

$$\text{loss/error} \quad \leftrightarrow \quad \text{neg-log likelihood}$$

$$\text{regularization} \quad \leftrightarrow \quad \text{neg-log prior}$$

$$\text{cost (reg.+loss)} \quad \leftrightarrow \quad \text{neg-log posterior}$$

8:2

8.1 Bayesian [Kernel] Ridge|Logistic Regression & Gaussian Processes

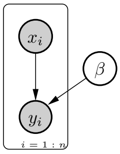
8:3

Gaussian Process = Bayesian (Kernel) Ridge Regression

8:4

Ridge regression as Bayesian inference

- We have random variables $X_{1:n}, Y_{1:n}, \beta$
- We observe data $D = \{(x_i, y_i)\}_{i=1}^n$ and want to compute $P(\beta \,|\, D)$



- Let's assume:
  $P(X)$ is arbitrary
  $P(\beta)$ is Gaussian: $\beta \sim \mathcal{N}(0, \frac{\sigma^2}{\lambda}) \propto e^{-\frac{\lambda}{2\sigma^2}\|\beta\|^2}$
  $P(Y \,|\, X, \beta)$ is Gaussian: $y = x^\top\beta + \epsilon \ , \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$

8:5

Ridge regression as Bayesian inference

- Bayes' Theorem:

$$P(\beta \,|\, D) = \frac{P(D \,|\, \beta)\ P(\beta)}{P(D)}$$

$$P(\beta \,|\, x_{1:n}, y_{1:n}) = \frac{\prod_{i=1}^n P(y_i \,|\, \beta, x_i)\ P(\beta)}{Z}$$

  $P(D \,|\, \beta)$ is a *product* of independent likelihoods for each observation (x_i, y_i)

  Using the Gaussian expressions:

$$P(\beta \,|\, D) = \frac{1}{Z'} \prod_{i=1}^n e^{-\frac{1}{2\sigma^2}(y_i - x_i^\top\beta)^2}\ e^{-\frac{\lambda}{2\sigma^2}\|\beta\|^2}$$

$$-\log P(\beta \,|\, D) = \frac{1}{2\sigma^2}\Big[\sum_{i=1}^n (y_i - x_i^\top\beta)^2 + \lambda\|\beta\|^2\Big] + \log Z'$$

$$-\log P(\beta \,|\, D) \propto L^{\text{ridge}}(\beta)$$

  1st insight: The *neg-log posterior* $P(\beta \,|\, D)$ is proportional to the cost function $L^{\text{ridge}}(\beta)$!

8:6

Ridge regression as Bayesian inference

- Let us compute $P(\beta \,|\, D)$ explicitly:

$$P(\beta \,|\, D) = \frac{1}{Z'} \prod_{i=1}^n e^{-\frac{1}{2\sigma^2}\ (y_i - x_i^\top\beta)^2}\ e^{-\frac{\lambda}{2\sigma^2}\|\beta\|^2}$$

$$
\begin{aligned}
&= \frac{1}{Z'}\; e^{-\frac{1}{2\sigma^2}\;\sum_i (y_i - x_i^\top\beta)^2}\; e^{-\frac{\lambda}{2\sigma^2}\,\|\beta\|^2} \\
&= \frac{1}{Z'}\; e^{-\frac{1}{2\sigma^2}[(y-X\beta)^\top(y-X\beta)+\lambda\beta^\top\beta]} \\
&= \frac{1}{Z'}\; e^{-\frac{1}{2}[\frac{1}{\sigma^2}y^\top y + \frac{1}{\sigma^2}\beta^\top(X^\top X+\lambda\mathbf{I})\beta - \frac{2}{\sigma^2}\beta^\top X^\top y]} \\
&= \mathcal{N}(\beta\,|\,\hat\beta,\Sigma)
\end{aligned}
$$

This is a Gaussian with covariance and mean

$$\Sigma = \sigma^2\;(X^\top X + \lambda\mathbf{I})^{\text{-}1}\;, \quad \hat\beta = \tfrac{1}{\sigma^2}\;\Sigma X^\top y = (X^\top X + \lambda\mathbf{I})^{\text{-}1}X^\top y$$

- **2nd insight:** The mean $\hat\beta$ is exactly the classical $\text{argmin}_\beta\, L^{\text{ridge}}(\beta)$.
- **3rd insight:** The Bayesian approach not only gives a mean/optimal $\hat\beta$, but also a variance Σ of that estimate. **(Cp. slide 02:13!)**

8:7

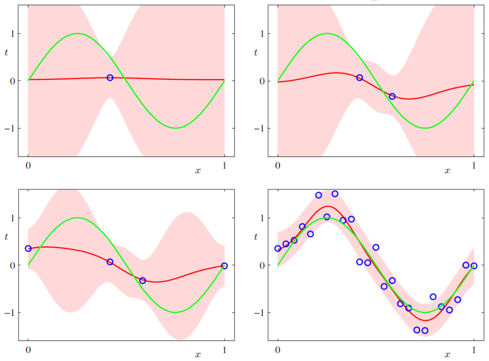
Predicting with an uncertain β

- Suppose we want to make a prediction at x. We can compute the **predictive distribution** over a new observation y^* at x^*:

$$
\begin{aligned}
P(y^*\,|\,x^*,D) &= \textstyle\int_\beta P(y^*\,|\,x^*,\beta)\;P(\beta\,|\,D)\;d\beta \\
&= \textstyle\int_\beta \mathcal{N}(y^*\,|\,\phi(x^*)^\top\beta,\sigma^2)\;\mathcal{N}(\beta\,|\,\hat\beta,\Sigma)\;d\beta \\
&= \mathcal{N}(y^*\,|\,\phi(x^*)^\top\hat\beta,\;\sigma^2 + \phi(x^*)^\top\Sigma\phi(x^*))
\end{aligned}
$$

Note, for $f(x)=\phi(x)^\top\beta$, we have $P(f(x)\,|\,D) = \mathcal{N}(f(x)\,|\,\phi(x)^\top\hat\beta,\;\phi(x)^\top\Sigma\phi(x))$ without the σ^2

- So, y^* is Gaussian distributed around the mean prediction $\phi(x^*)^\top\hat\beta$:



(from Bishop, p176)

8:8

Wrapup of Bayesian Ridge regression

- **1st insight:** The *neg-log posterior* $P(\beta\,|\,D)$ is equal to the cost function $L^{\text{ridge}}(\beta)$.

This is a very very common relation: optimization costs correspond to neg-log probabilities; probabilities correspond to exp-neg costs.

- **2nd insight:** The mean $\hat\beta$ is exactly the classical $\text{argmin}_\beta\, L^{\text{ridge}}(\beta)$

  More generally, the most likely parameter $\text{argmax}_\beta\, P(\beta|D)$ is also the least-cost parameter $\text{argmin}_\beta\, L(\beta)$. In the Gaussian case, most-likely β is also the mean.

- **3rd insight:** The Bayesian inference approach not only gives a mean/optimal $\hat\beta$, but also a variance Σ of that estimate

  This is a core benefit of the Bayesian view: It naturally provides a probability distribution over predictions (*"error bars"*), not only a single prediction.

8:9

Kernel Bayesian Ridge Regression

- As in the classical case, we can consider arbitrary features $\phi(x)$
- .. or directly use a kernel $k(x,x')$:

$$\begin{aligned}
P(f(x)\,|\,D) &= \mathcal{N}(f(x)\,|\,\phi(x)^\top\hat\beta,\ \phi(x)^\top\Sigma\phi(x)) \\
\phi(x)^\top\hat\beta &= \phi(x)^\top X^\top(XX^\top+\lambda\mathbf{I})^{\text{-}1}y \\
&= \kappa(x)(K+\lambda\mathbf{I})^{\text{-}1}y \\
\phi(x)^\top\Sigma\phi(x) &= \phi(x)^\top\sigma^2\ (X^\top X+\lambda\mathbf{I})^{\text{-}1}\phi(x) \\
&= \frac{\sigma^2}{\lambda}\phi(x)^\top\phi(x) - \frac{\sigma^2}{\lambda}\phi(x)^\top X^\top(XX^\top+\lambda\mathbf{I}_k)^{\text{-}1}X\phi(x) \\
&= \frac{\sigma^2}{\lambda}k(x,x) - \frac{\sigma^2}{\lambda}\kappa(x)(K+\lambda\mathbf{I}_n)^{\text{-}1}\kappa(x)^\top
\end{aligned}$$

3rd line: As on slide 05:2

2nd to last line: Woodbury identity $(A+UBV)^{\text{-}1} = A^{\text{-}1} - A^{\text{-}1}U(B^{\text{-}1}+VA^{\text{-}1}U)^{\text{-}1}VA^{\text{-}1}$ with $A=\lambda\mathbf{I}$

- In standard conventions $\lambda=\sigma^2$, i.e. $P(\beta)=\mathcal{N}(\beta|0,1)$
  – Regularization: scale the covariance function (or features)

8:10

Gaussian Processes

are equivalent to Kernelized Bayesian Ridge Regression

(see also Welling: "Kernel Ridge Regression" Lecture Notes; Rasmussen & Williams sections 2.1 & 6.2; Bishop sections 3.3.3 & 6)

- But it is insightful to introduce them again from the "function space view": GPs define a probability distribution over functions; they are the infinite dimensional generalization of Gaussian vectors

8:11

Gaussian Processes – function prior

- The function space view

$$P(f|D) = \frac{P(D|f)\,P(f)}{P(D)}$$

- A Gaussian Processes **prior** $P(f)$ defines a probability distribution over functions:
  - A function is an infinite dimensional thing – how could we define a Gaussian distribution over functions?
  - For every finite set $\{x_1, .., x_M\}$, the function values $f(x_1), .., f(x_M)$ are Gaussian distributed with mean and covariance

$$\mathrm{E}\{f(x_i)\} = \mu(x_i) \qquad \text{(often zero)}$$
$$\mathrm{cov}\{f(x_i), f(x_j)\} = k(x_i, x_j)$$

  Here, $k(\cdot,\cdot)$ is called **covariance function**

- Second, for Gaussian Processes we typically have a Gaussian **data likelihood** $P(D|f)$, namely

$$P(y\,|\,x, f) = \mathcal{N}(y\,|\,f(x), \sigma^2)$$

8:12

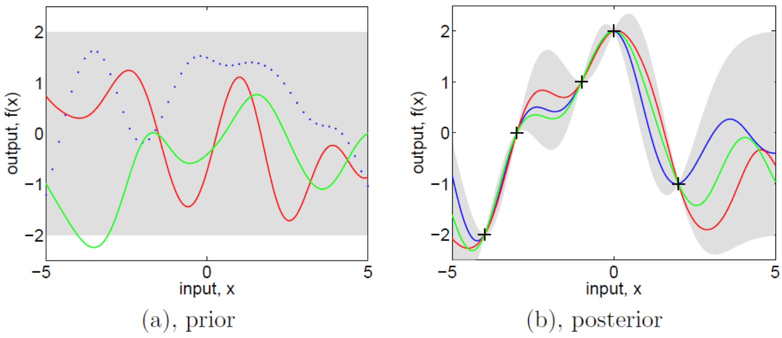
Gaussian Processes – function posterior

- The **posterior** $P(f|D)$ is also a Gaussian Process, with the following mean of $f(x)$, covariance of $f(x)$ and $f(x')$: (based on slide 10 (with $\lambda = \sigma^2$))

$$\mathrm{E}\{f(x)\,|\,D\} = \kappa(x)(K + \lambda\mathbf{I})^{\text{-}1}y \;+\; \mu(x)$$
$$\mathrm{cov}\{f(x), f(x')\,|\,D\} = k(x, x') - \kappa(x')(K + \lambda\mathbf{I}_n)^{\text{-}1}\kappa(x')^\top$$

8:13

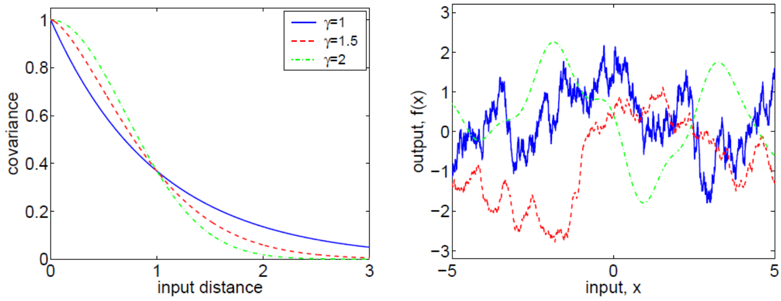
Gaussian Processes



(from Rasmussen & Williams)

8:14

GP: different covariance functions



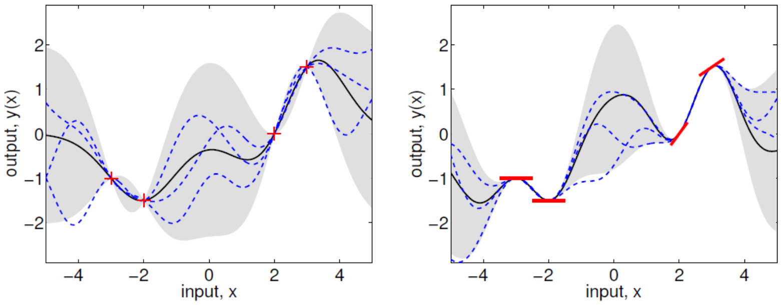
(from Rasmussen & Williams)

- These are examples from the γ-exponential covariance function

$$k(x,x') = \exp\{-|(x-x')/l|^\gamma\}$$

8:15

GP: derivative observations



(from Rasmussen & Williams)

8:16

- Bayesian Kernel Ridge Regression = Gaussian Process

- GPs have become a standard regression method
- If exact GP is not efficient enough, many approximations exist, e.g. sparse and pseudo-input GPs

8:17

GP classification = Bayesian (Kernel) Logistic Regression

8:18

Bayesian Logistic Regression (binary case)

- f now defines a discriminative function:

$$\begin{aligned} P(X) &= \text{arbitrary} \\ P(\beta) &= \mathcal{N}(\beta|0, \frac{2}{\lambda}) \propto \exp\{-\lambda\|\beta\|^2\} \\ P(Y\!=\!1 \,|\, X, \beta) &= \sigma(\beta^\top \phi(x)) \end{aligned}$$

- Recall

$$L^{\text{logistic}}(\beta) = -\sum_{i=1}^n \log p(y_i \,|\, x_i) + \lambda\|\beta\|^2$$

- Again, the parameter posterior is

$$P(\beta|D) \propto P(D \,|\, \beta)\; P(\beta) \propto \exp\{-L^{\text{logistic}}(\beta)\}$$

8:19

Bayesian Logistic Regression

- Use **Laplace approximation** (2nd order Taylor for L) at $\beta^* = \text{argmin}_\beta\, L(\beta)$:

$$L(\beta) \approx L(\beta^*) + \bar\beta^\top \nabla + \frac{1}{2}\bar\beta^\top H \bar\beta\;, \quad \bar\beta = \beta - \beta^*$$

At β^* the gradient $\nabla = 0$ and $L(\beta^*) = \text{const}$. Therefore

$$\begin{aligned} \tilde P(\beta|D) &\propto \exp\{-\frac{1}{2}\bar\beta^\top H \bar\beta\} \\ \Rightarrow\; P(\beta|D) &\approx \mathcal{N}(\beta|\beta^*, H^{\text{-}1}) \end{aligned}$$

- Then the predictive distribution of the *discriminative function* is also Gaussian!

$$\begin{aligned} P(f(x) \,|\, D) &= \textstyle\int_\beta P(f(x) \,|\, \beta)\; P(\beta \,|\, D)\; d\beta \\ &\approx \textstyle\int_\beta \mathcal{N}(f(x) \,|\, \phi(x)^\top\beta, 0)\; \mathcal{N}(\beta \,|\, \beta^*, H^{\text{-}1})\; d\beta \\ &= \mathcal{N}(f(x) \,|\, \phi(x)^\top\beta^*, \phi(x)^\top H^{\text{-}1}\phi(x)) \;=:\; \mathcal{N}(f(x) \,|\, f^*, s^2) \end{aligned}$$

- The predictive distribution over the label $y \in \{0, 1\}$:

$$\begin{aligned} P(y(x)\!=\!1 \,|\, D) &= \textstyle\int_{f(x)} \sigma(f(x))\; P(f(x)|D)\; df \\ &\approx \sigma((1 + s^2\pi/8)^{\text{-}\frac{1}{2}} f^*) \end{aligned}$$

which uses a probit approximation of the convolution.
$\rightarrow$ The variance s^2 pushes the predictive class probabilities towards 0.5.

8:20

Kernelized Bayesian Logistic Regression

- As with Kernel Logistic Regression, the MAP discriminative function f^* can be found iterating the Newton method $\leftrightarrow$ iterating GP estimation on a *re-weighted* data set.
- The rest is as above.

8:21

Kernel Bayesian Logistic Regression

is equivalent to Gaussian Process Classification

- GP classification became a standard classification method, if the prediction needs to be a meaningful probability that takes the *model uncertainty* into account.

8:22

8.2 Bayesian Neural Networks

8:23

Bayesian Neural Networks

- Simple ways to get uncertainty estimates:
  - Train ensembles of networks (e.g. bootstrap ensembles)
  - Treat the output layer fully probabilistic (treat the trained NN body as feature vector $\phi(x)$, and apply Bayesian Ridge/Logistic Regression on top of that)

- Ways to treat NNs inherently Bayesian:
  - Infinite single-layer NN $\to$ GP (classical work in 80/90ies)
  - Putting priors over weights ("Bayesian NNs", Neil, MacKay, 90ies)
  - Dropout (much more recent, see papers below)

- Read

  Gal & Ghahramani: *Dropout as a bayesian approximation: Representing model uncertainty in deep learning* (ICML'16)

  Damianou & Lawrence: *Deep gaussian processes* (AIS 2013)

8:24

Dropout in NNs as Deep GPs

- Deep GPs are essentially a a chaining of Gaussian Processes
  - The mapping from each layer to the next is a GP
  - Each GP could have a different prior (kernel)

- Dropout in NNs
  - Dropout leads to randomized prediction
  - One can estimate the mean prediction from T dropout samples (MC estimate)
  - Or one can estimate the mean prediction by averaging the weights of the network ("standard dropout")
  - Equally one can MC estimate the variance from samples
  - Gal & Ghahramani show, that a Dropout NN is a Deep GP (with very special kernel), and the "correct" predictive variance is this MC estimate plus $\frac{pl^2}{2n\lambda}$ (kernel length scale l, regularization λ, dropout prob p, and n data points)

8:25

8.3 No Free Lunch**

8:26

No Free Lunch

- Averaged over *all* problem instances, any algorithm performs equally. (E.g. equal to random.)
  - "there is no one model that works best for every problem"

  Igel & Toussaint: *On Classes of Functions for which No Free Lunch Results Hold* (Information Processing Letters 2003)

- Rigorous formulations formalize this "average over *all* problem instances". E.g. by assuming a uniform prior over problems
  - In black-box optimization, a uniform distribution over underlying objective functions $f(x)$
  - In machine learning, a uniform distribution over the hidden true function $f(x)$

  ... and NLF always considers *non-repeating queries*.

- But what does *uniform distribution over functions mean?*

- NLF is trivial: when any previous query yields NO information at all about the results of future queries, anything is exactly as good as random guessing

8:27

Conclusions

- Probabilistic inference is a very powerful concept!

  – Inferring about the world given data

  – Learning, decision making, reasoning can view viewed as forms of (probabilistic) inference

- We introduced Bayes' Theorem as the fundamental form of probabilistic inference

- Marrying Bayes with (Kernel) Ridge (Logisic) regression yields
  – Gaussian Processes
  – Gaussian Process classification

- We can estimate uncertainty also for NNs
  – Dropout
  – Probabilistic weights and variational approximations; Deep GPs

- No Free Lunch for ML!

8:28

A Probability Basics

The need for modelling

- Given a real world problem, translating it to a well-defined learning problem is non-trivial
- The "framework" of plain regression/classification is restricted: input x, output y.
- Graphical models (probabilstic models with multiple random variables and dependencies) are a more general framework for modelling "problems"; regression & classification become a special case; Reinforcement Learning, decision making, unsupervised learning, but also language processing, image segmentation, can be represented.

9:1

Outline

- Basic definitions
  - Random variables
  - joint, conditional, marginal distribution
  - Bayes' theorem
- Examples for Bayes
- Probability distributions [skipped, only Gauss]
  - Binomial; Beta
  - Multinomial; Dirichlet
  - Conjugate priors
  - Gauss; Wichart
  - Student-t, Dirak, Particles
- Monte Carlo, MCMC [skipped]

  These are generic slides on probabilities I use throughout my lecture. Only parts are mandatory for the AI course.

9:2

Thomas Bayes (1702-–1761)

~~Rev. T. Bayes~~

"Essay Towards Solving a Problem in the Doctrine of Chances"

- Addresses problem of *inverse probabilities*:
  Knowing the conditional probability of B given A, what is the conditional probability of A given B?

- Example:
  40% Bavarians speak dialect, only 1% of non-Bavarians speak (Bav.) dialect
  Given a random German that speaks non-dialect, is he Bavarian?
  (15% of Germans are Bavarian)

9:3

Inference

- "Inference" = Given some pieces of information (prior, observed variabes) what is the implication (the implied information, the posterior) on a non-observed variable

- **Decision-Making and Learning as Inference:**
  - given pieces of information: about the world/game, collected data, assumed model class, *prior* over model parameters
  - make decisions about actions, classifier, model parameters, etc

9:4

Probability Theory

- Why do we need probabilities?
  - Obvious: to express inherent stochasticity of the world (data)

- But beyond this: (also in a "deterministic world"):
  - lack of knowledge!
  - hidden (latent) variables

  - expressing *uncertainty*
  - expressing *information* (and lack of information)

- Probability Theory: an information calculus

9:5

Probability: Frequentist and Bayesian

- Frequentist probabilities are defined in the limit of an infinite number of trials
  Example: "The probability of a particular coin landing heads up is 0.43"

- Bayesian (subjective) probabilities quantify degrees of belief
  Example: "The probability of rain tomorrow is 0.3" – not possible to repeat "tomorrow"

9:6

A.1 Basic definitions

9:7

Probabilities & Random Variables

- For a random variable X with discrete domain $\text{dom}(X) = \Omega$ we write:
  $\forall_{x\in\Omega} :\ 0 \le P(X{=}x) \le 1$
  $\sum_{x\in\Omega} P(X{=}x) = 1$

  Example: A dice can take values $\Omega = \{1, .., 6\}$.
  X is the random variable of a dice throw.
  $P(X{=}1) \in [0, 1]$ is the probability that X takes value 1.

- A bit more formally: a random variable is a map from a measureable space to a domain (sample space) and thereby introduces a probability measure on the domain ("assigns a probability to each possible value")

9:8

Probabilty Distributions

- $P(X{=}1) \in \mathbb{R}$ denotes a specific probability
  $P(X)$ denotes the probability distribution (function over Ω)

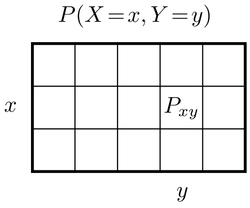
  Example: A dice can take values $\Omega = \{1, 2, 3, 4, 5, 6\}$.
  By $P(X)$ we discribe the full distribution over possible values $\{1, .., 6\}$. These are 6 numbers that sum to one, usually stored in a *table*, e.g.: $[\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}]$

- In implementations we typically represent distributions over discrete random variables as tables (arrays) of numbers
- Notation for summing over a RV:
  In equation we often need to sum over RVs. We then write
  $$\sum_X P(X)\ \cdots$$
  as shorthand for the explicit notation $\sum_{x\in \text{dom}(X)} P(X\!=\!x)\ \cdots$

9:9

Joint distributions

Assume we have *two* random variables X and Y



- Definitions:
  Joint: $P(X,Y)$
  Marginal: $P(X)=\sum_Y P(X,Y)$
  Conditional: $P(X|Y)=\frac{P(X,Y)}{P(Y)}$

  The conditional is normalized: $\forall_Y:\ \sum_X P(X|Y)=1$
- X is *independent* of Y iff: $P(X|Y)=P(X)$
  (table thinking: all columns of $P(X|Y)$ are equal)

9:10

Joint distributions

joint: $P(X,Y)$
marginal: $P(X)=\sum_Y P(X,Y)$
conditional: $P(X|Y)=\frac{P(X,Y)}{P(Y)}$

- Implications of these definitions:
  Product rule: $P(X,Y)=P(X|Y)\ P(Y)=P(Y|X)\ P(X)$

  Bayes' Theorem: $P(X|Y)=\frac{P(Y|X)\ P(X)}{P(Y)}$

9:11

Bayes' Theorem

$$P(X|Y) = \frac{P(Y|X)\ P(X)}{P(Y)}$$

$$\text{posterior} = \frac{\text{likelihood} \cdot \text{prior}}{\text{normalization}}$$

9:12

Multiple RVs:

- Analogously for n random variables $X_{1:n}$ (stored as a rank n tensor)
  Joint: $P(X_{1:n})$
  Marginal: $P(X_1) = \sum_{X_{2:n}} P(X_{1:n})$,
  Conditional: $P(X_1|X_{2:n}) = \frac{P(X_{1:n})}{P(X_{2:n})}$

- X is *conditionally independent* of Y given Z iff:
  $$P(X|Y,Z) = P(X|Z)$$

- Product rule and Bayes' Theorem:

$$P(X_{1:n}) = \prod_{i=1}^n P(X_i|X_{i+1:n})$$

$$P(X_1|X_{2:n}) = \frac{P(X_2|X_1,X_{3:n})\ P(X_1|X_{3:n})}{P(X_2|X_{3:n})}$$

$$P(X,Z,Y) = P(X|Y,Z)\ P(Y|Z)\ P(Z)$$

$$P(X|Y,Z) = \frac{P(Y|X,Z)\ P(X|Z)}{P(Y|Z)}$$

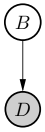
$$P(X,Y|Z) = \frac{P(X,Z|Y)\ P(Y)}{P(Z)}$$

9:13

Example 1: Bavarian dialect

- 40% Bavarians speak dialect, only 1% of non-Bavarians speak (Bav.) dialect
  Given a random German that speaks non-dialect, is he Bavarian?
  (15% of Germans are Bavarian)

$P(D{=}1 \mid B{=}1) = 0.4$
$P(D{=}1 \mid B{=}0) = 0.01$
$P(B{=}1) = 0.15$



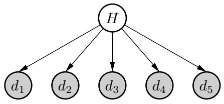
If follows
$P(B{=}1 \mid D{=}0) = \frac{P(D{=}0 \mid B{=}1)\ P(B{=}1)}{P(D{=}0)} = \frac{.6 \cdot .15}{.6 \cdot .15 + 0.99 \cdot .85} \approx 0.097$

9:14

Example 2: Coin flipping

HHTHT

HHHHH



- What process produces these sequences?
- We compare two hypothesis:
  $H=1$: fair coin $P(d_i=\texttt{H}\,|\,H=1)=\frac{1}{2}$
  $H=2$: always heads coin $P(d_i=\texttt{H}\,|\,H=2)=1$
- Bayes' theorem:

$$P(H\,|\,D)=\frac{P(D\,|\,H)P(H)}{P(D)}$$

9:15

Coin flipping

$D=\texttt{HHTHT}$

$P(D\,|\,H=1)=1/2^5$ $\qquad P(H=1)=\frac{999}{1000}$
$P(D\,|\,H=2)=0$ $\qquad P(H=2)=\frac{1}{1000}$

$$\frac{P(H=1\,|\,D)}{P(H=2\,|\,D)}=\frac{P(D\,|\,H=1)}{P(D\,|\,H=2)}\;\frac{P(H=1)}{P(H=2)}=\frac{1/32}{0}\;\frac{999}{1}=\infty$$

9:16

Coin flipping

$D=\texttt{HHHHH}$

$P(D\,|\,H=1)=1/2^5$ $\qquad P(H=1)=\frac{999}{1000}$
$P(D\,|\,H=2)=1$ $\qquad P(H=2)=\frac{1}{1000}$

$$\frac{P(H=1\,|\,D)}{P(H=2\,|\,D)}=\frac{P(D\,|\,H=1)}{P(D\,|\,H=2)}\;\frac{P(H=1)}{P(H=2)}=\frac{1/32}{1}\;\frac{999}{1}\approx 30$$

9:17

Coin flipping

$D = \mathtt{HHHHHHHHHH}$

$$P(D\,|\,H\!=\!1) = 1/2^{10} \qquad P(H\!=\!1) = \tfrac{999}{1000}$$
$$P(D\,|\,H\!=\!2) = 1 \qquad P(H\!=\!2) = \tfrac{1}{1000}$$

$$\frac{P(H\!=\!1\,|\,D)}{P(H\!=\!2\,|\,D)} = \frac{P(D\,|\,H\!=\!1)}{P(D\,|\,H\!=\!2)}\,\frac{P(H\!=\!1)}{P(H\!=\!2)} = \frac{1/1024}{1}\,\frac{999}{1} \approx 1$$

9:18

Learning as Bayesian inference

$$P(\text{World}|\text{Data}) = \frac{P(\text{Data}|\text{World})\;P(\text{World})}{P(\text{Data})}$$

$P(\text{World})$ describes our prior over all possible worlds. Learning means to infer about the world we live in based on the data we have!

- In the context of regression, the "world" is the function $f(x)$

$$P(f|\text{Data}) = \frac{P(\text{Data}|f)\;P(f)}{P(\text{Data})}$$

$P(f)$ describes our prior over possible functions

Regression means to infer the function based on the data we have

9:19

A.2 Probability distributions

recommended reference: Bishop.: *Pattern Recognition and Machine Learning*

9:20

Bernoulli & Binomial

- We have a binary random variable $x \in \{0,1\}$ (i.e. $\text{dom}(x) = \{0,1\}$)
  The *Bernoulli* distribution is parameterized by a single scalar μ,

$$P(x\!=\!1\,|\,\mu) = \mu\;, \quad P(x\!=\!0\,|\,\mu) = 1-\mu$$
$$\text{Bern}(x\,|\,\mu) = \mu^x(1-\mu)^{1-x}$$

- We have a data set of random variables $D = \{x_1, .., x_n\}$, each $x_i \in \{0,1\}$. If each $x_i \sim \text{Bern}(x_i \,|\, \mu)$ we have

$$P(D \,|\, \mu) = \textstyle\prod_{i=1}^n \text{Bern}(x_i \,|\, \mu) = \prod_{i=1}^n \mu^{x_i}(1-\mu)^{1-x_i}$$

$$\underset{\mu}{\text{argmax}}\ \log P(D \,|\, \mu) = \underset{\mu}{\text{argmax}} \sum_{i=1}^n x_i \log \mu + (1-x_i)\log(1-\mu) = \frac{1}{n}\sum_{i=1}^n x_i$$

- The *Binomial distribution* is the distribution over the count $m = \sum_{i=1}^n x_i$

$$\text{Bin}(m \,|\, n, \mu) = \binom{n}{m} \mu^m (1-\mu)^{n-m} \;, \qquad \binom{n}{m} = \frac{n!}{(n-m)!\; m!}$$

9:21

Beta

How to express uncertainty over a Bernoulli parameter μ

- The *Beta* distribution is over the interval $[0,1]$, typically the parameter μ of a Bernoulli:

$$\text{Beta}(\mu \,|\, a,b) = \frac{1}{B(a,b)}\, \mu^{a-1}(1-\mu)^{b-1}$$

with mean $\langle\mu\rangle = \frac{a}{a+b}$ and mode $\mu^* = \frac{a-1}{a+b-2}$ for $a,b>1$

- The crucial point is:
  - Assume we are in a world with a "Bernoulli source" (e.g., binary bandit), but don't know its parameter μ
  - Assume we have a *prior* distribution $P(\mu) = \text{Beta}(\mu \,|\, a,b)$
  - Assume we collected some data $D = \{x_1, .., x_n\}$, $x_i \in \{0,1\}$, with counts $a_D = \sum_i x_i$ of $[x_i\!=\!1]$ and $b_D = \sum_i (1-x_i)$ of $[x_i\!=\!0]$
  - The posterior is

$$\begin{aligned} P(\mu \,|\, D) &= \frac{P(D \,|\, \mu)}{P(D)}\, P(\mu) \propto \text{Bin}(D \,|\, \mu)\, \text{Beta}(\mu \,|\, a,b) \\ &\propto \mu^{a_D}(1-\mu)^{b_D}\ \mu^{a-1}(1-\mu)^{b-1} = \mu^{a-1+a_D}(1-\mu)^{b-1+b_D} \\ &= \text{Beta}(\mu \,|\, a+a_D, b+b_D) \end{aligned}$$

9:22

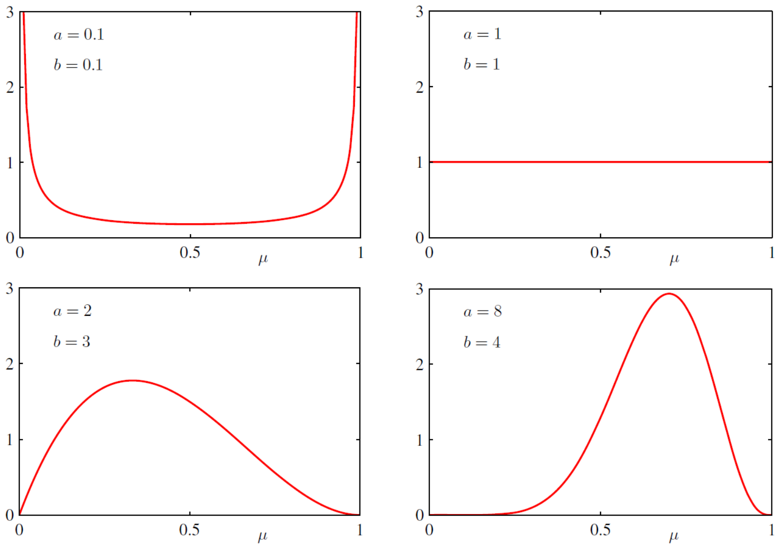
Beta

The prior is $\text{Beta}(\mu \,|\, a,b)$, *the posterior is* $\text{Beta}(\mu \,|\, a+a_D, b+b_D)$

- Conclusions:
  - The semantics of a and b are counts of $[x_i=1]$ and $[x_i=0]$, respectively
  - The Beta distribution is conjugate to the Bernoulli (explained later)
  - With the Beta distribution we can represent beliefs (state of knowledge) about uncertain $\mu \in [0,1]$ and know how to update this belief given data

9:23

Beta



from Bishop

9:24

Multinomial

- We have an integer random variable $x \in \{1,..,K\}$
  The probability of a single x can be parameterized by $\mu = (\mu_1,..,\mu_K)$:

$$P(x=k \mid \mu) = \mu_k$$

  with the constraint $\sum_{k=1}^K \mu_k = 1$ (probabilities need to be normalized)
- We have a data set of random variables $D = \{x_1,..,x_n\}$, each $x_i \in \{1,..,K\}$. If each $x_i \sim P(x_i \mid \mu)$ we have

$$P(D \mid \mu) = \prod_{i=1}^n \mu_{x_i} = \prod_{i=1}^n \prod_{k=1}^K \mu_k^{[x_i=k]} = \prod_{k=1}^K \mu_k^{m_k}$$

  where $m_k = \sum_{i=1}^n [x_i=k]$ is the count of $[x_i=k]$. The ML estimator is

$$\operatorname*{argmax}_\mu \ \log P(D \mid \mu) = \frac{1}{n}(m_1,..,m_K)$$

- The *Multinomial distribution* is this distribution over the counts m_k

$$\text{Mult}(m_1,..,m_K \,|\, n, \mu) \propto \textstyle\prod_{k=1}^K \mu_k^{m_k}$$

9:25

Dirichlet

How to express uncertainty over a Multinomial parameter μ

- The *Dirichlet* distribution is over the K-simplex, that is, over $\mu_1,..,\mu_K \in [0,1]$ subject to the constraint $\sum_{k=1}^K \mu_k = 1$:

$$\text{Dir}(\mu \,|\, \alpha) \propto \textstyle\prod_{k=1}^K \mu_k^{\alpha_k - 1}$$

It is parameterized by $\alpha = (\alpha_1,..,\alpha_K)$, has mean $\langle \mu_i \rangle = \frac{\alpha_i}{\sum_j \alpha_j}$ and mode $\mu_i^* = \frac{\alpha_i - 1}{\sum_j \alpha_j - K}$ for $a_i > 1$.

- The crucial point is:
  - Assume we are in a world with a "Multinomial source" (e.g., an integer bandit), but don't know its parameter μ
  - Assume we have a *prior* distribution $P(\mu) = \text{Dir}(\mu \,|\, \alpha)$
  - Assume we collected some data $D = \{x_1,..,x_n\}$, $x_i \in \{1,..,K\}$, with counts $m_k = \sum_i [x_i = k]$
  - The posterior is

$$\begin{aligned} P(\mu \,|\, D) &= \frac{P(D \,|\, \mu)}{P(D)}\, P(\mu) \propto \text{Mult}(D \,|\, \mu)\, \text{Dir}(\mu \,|\, a, b) \\ &\propto \textstyle\prod_{k=1}^K \mu_k^{m_k}\, \prod_{k=1}^K \mu_k^{\alpha_k - 1} = \prod_{k=1}^K \mu_k^{\alpha_k - 1 + m_k} \\ &= \text{Dir}(\mu \,|\, \alpha + m) \end{aligned}$$

9:26

Dirichlet

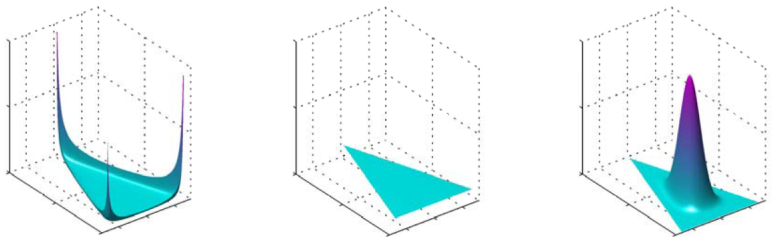
The prior is $\text{Dir}(\mu \,|\, \alpha)$*, the posterior is* $\text{Dir}(\mu \,|\, \alpha + m)$

- Conclusions:
  - The semantics of α is the counts of $[x_i = k]$
  - The Dirichlet distribution is conjugate to the Multinomial
  - With the Dirichlet distribution we can represent beliefs (state of knowledge) about uncertain μ of an integer random variable and know how to update this belief given data

9:27

Dirichlet

Illustrations for $\alpha = (0.1, 0.1, 0.1)$, $\alpha = (1, 1, 1)$ and $\alpha = (10, 10, 10)$:



from Bishop

9:28

Motivation for Beta & Dirichlet distributions

- Bandits:
  - If we have binary [integer] bandits, the Beta [Dirichlet] distribution is a way to represent and update beliefs
  - The belief space becomes discrete: The parameter α of the prior is continuous, but the posterior updates live on a discrete "grid" (adding counts to α)
  - We can in principle do belief planning using this
- Reinforcement Learning:
  - Assume we know that the world is a finite-state MDP, but do not know its transition probability $P(s' \mid s, a)$. For each (s, a), $P(s' \mid s, a)$ is a distribution over the integer s'
  - Having a separate Dirichlet distribution for each (s, a) is a way to represent our belief about the world, that is, our belief about $P(s' \mid s, a)$
  - We can in principle do belief planning using this $\to$ *Bayesian Reinforcement Learning*
- Dirichlet distributions are also used to model texts (word distributions in text), images, or mixture distributions in general

9:29

Conjugate priors

- Assume you have data $D = \{x_1, .., x_n\}$ with likelihood

$$P(D \mid \theta)$$

that depends on an uncertain parameter θ
Assume you have a prior $P(\theta)$

- The prior $P(\theta)$ is **conjugate** to the likelihood $P(D \mid \theta)$ iff the posterior

$$P(\theta \mid D) \propto P(D \mid \theta)\ P(\theta)$$

is in the *same distribution class* as the prior $P(\theta)$

- Having a conjugate prior is very convenient, because then you know how to update the belief given data

9:30

Conjugate priors

likelihood	conjugate
Binomial $\text{Bin}(D \mid \mu)$	Beta $\text{Beta}(\mu \mid a, b)$
Multinomial $\text{Mult}(D \mid \mu)$	Dirichlet $\text{Dir}(\mu \mid \alpha)$
Gauss $\mathcal{N}(x \mid \mu, \Sigma)$	Gauss $\mathcal{N}(\mu \mid \mu_0, A)$
1D Gauss $\mathcal{N}(x \mid \mu, \lambda^{-1})$	Gamma $\text{Gam}(\lambda \mid a, b)$
nD Gauss $\mathcal{N}(x \mid \mu, \Lambda^{-1})$	Wishart $\text{Wish}(\Lambda \mid W, \nu)$
nD Gauss $\mathcal{N}(x \mid \mu, \Lambda^{-1})$	Gauss-Wishart $\mathcal{N}(\mu \mid \mu_0, (\beta\Lambda)^{-1})\ \text{Wish}(\Lambda \mid W, \nu)$

9:31

A.3 Distributions over continuous domain

9:32

Distributions over continuous domain

- Let x be a continuous RV. The **probability density function (pdf)** $p(x) \in [0, \infty)$ defines the probability

$$P(a \le x \le b) = \int_a^b p(x)\ dx\ \in [0, 1]$$

The **(cumulative) probability distribution** $F(y) = P(x \le y) = \int_{-\infty}^y dx\ p(x) \in [0, 1]$ is the cumulative integral with $\lim_{y \to \infty} F(y) = 1$

(In discrete domain: *probability distribution* and *probability mass function* $P(x) \in [0, 1]$ are used synonymously.)

- Two basic examples:

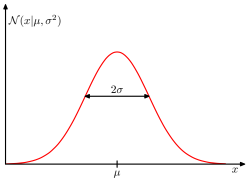
Gaussian: $\mathcal{N}(x \mid \mu, \Sigma) = \frac{1}{|\,2\pi\Sigma\,|^{1/2}}\ e^{-\frac{1}{2}(x-\mu)^\top\ \Sigma^{-1}\ (x-\mu)}$

Dirac or δ ("point particle") $\delta(x) = 0$ except at $x = 0$, $\int \delta(x)\ dx = 1$

$\delta(x) = \frac{\partial}{\partial x} H(x)$ where $H(x) = [x \ge 0] =$ Heaviside step function

9:33

Gaussian distribution



- 1-dim: $\mathcal{N}(x\,|\,\mu,\sigma^2) = \frac{1}{|\,2\pi\sigma^2\,|^{1/2}}\ e^{-\frac{1}{2}(x-\mu)^2/\sigma^2}$
- n-dim Gaussian in *normal form*:

$$\mathcal{N}(x\,|\,\mu,\Sigma) = \frac{1}{|\,2\pi\Sigma\,|^{1/2}}\ \exp\{-\frac{1}{2}(x-\mu)^\top\,\Sigma^{\text{-}1}\,(x-\mu)\}$$

with **mean** μ and **covariance** matrix Σ. In *canonical form*:

$$\mathcal{N}[x\,|\,a,A] = \frac{\exp\{-\frac{1}{2}a^\top A^{\text{-}1}a\}}{|\,2\pi A^{\text{-}1}\,|^{1/2}}\ \exp\{-\frac{1}{2}x^\top\,A\,x + x^\top a\} \tag{9}$$

with **precision** matrix $A=\Sigma^{\text{-}1}$ and coefficient $a=\Sigma^{\text{-}1}\mu$ (and mean $\mu = A^{\text{-}1}a$).
Note: $|\,2\pi\Sigma\,| = \det(2\pi\Sigma) = (2\pi)^n\det(\Sigma)$

- Gaussian identities: see http://ipvs.informatik.uni-stuttgart.de/mlr/marc/notes/gaussians.pdf

9:34

Gaussian identities

Symmetry: $\mathcal{N}(x\,|\,a,A) = \mathcal{N}(a\,|\,x,A) = \mathcal{N}(x-a\,|\,0,A)$

Product:
$\mathcal{N}(x\,|\,a,A)\ \mathcal{N}(x\,|\,b,B) = \mathcal{N}[x\,|\,A^{\text{-}1}a+B^{\text{-}1}b, A^{\text{-}1}+B^{\text{-}1}]\ \mathcal{N}(a\,|\,b,A+B)$
$\mathcal{N}[x\,|\,a,A]\ \mathcal{N}[x\,|\,b,B] = \mathcal{N}[x\,|\,a+b,A+B]\ \mathcal{N}(A^{\text{-}1}a\,|\,B^{\text{-}1}b, A^{\text{-}1}+B^{\text{-}1})$

"Propagation":
$\int_y \mathcal{N}(x\,|\,a+Fy,A)\ \mathcal{N}(y\,|\,b,B)\ dy = \mathcal{N}(x\,|\,a+Fb, A+FBF^\top)$

Transformation:
$\mathcal{N}(Fx+f\,|\,a,A) = \frac{1}{|\,F\,|}\ \mathcal{N}(x\,|\ F^{\text{-}1}(a-f),\ F^{\text{-}1}AF^{\text{-}\top})$

Marginal & conditional:
$\mathcal{N}\left(\begin{matrix} x \\ y\end{matrix}\,\middle|\,\begin{matrix} a \\ b\end{matrix}, \begin{matrix} A & C \\ C^\top & B\end{matrix}\right) = \mathcal{N}(x\,|\,a,A)\cdot\mathcal{N}(y\,|\,b+C^\top A^{\text{-}1}(x\text{-}a),\ B-C^\top A^{\text{-}1}C)$

More Gaussian identities: see http://ipvs.informatik.uni-stuttgart.de/mlr/marc/notes/gaussians.pdf

9:35

Gaussian prior and posterior

- Assume we have data $D = \{x_1, .., x_n\}$, each $x_i \in \mathbb{R}^n$, with likelihood

$$\begin{aligned} P(D \,|\, \mu, \Sigma) &= \textstyle\prod_i \mathcal{N}(x_i \,|\, \mu, \Sigma) \\ \operatorname*{argmax}_\mu \; P(D \,|\, \mu, \Sigma) &= \frac{1}{n} \sum_{i=1}^n x_i \\ \operatorname*{argmax}_\Sigma \; P(D \,|\, \mu, \Sigma) &= \frac{1}{n} \sum_{i=1}^n (x_i - \mu)(x_i - \mu)^\top \end{aligned}$$

- Assume we are initially uncertain about μ (but know Σ). We can express this uncertainty using again a Gaussian $\mathcal{N}[\mu \,|\, a, A]$. Given data we have

$$\begin{aligned} P(\mu \,|\, D) &\propto P(D \,|\, \mu, \Sigma)\; P(\mu) = \textstyle\prod_i \mathcal{N}(x_i \,|\, \mu, \Sigma)\; \mathcal{N}[\mu \,|\, a, A] \\ &= \textstyle\prod_i \mathcal{N}[\mu \,|\, \Sigma^{\text{-}1} x_i, \Sigma^{\text{-}1}]\; \mathcal{N}[\mu \,|\, a, A] \propto \mathcal{N}[\mu \,|\, \Sigma^{\text{-}1} \sum_i x_i,\; n\Sigma^{\text{-}1} + A] \end{aligned}$$

Note: in the limit $A \to 0$ (uninformative prior) this becomes

$$P(\mu \,|\, D) = \mathcal{N}(\mu \,|\, \frac{1}{n} \sum_i x_i, \frac{1}{n}\Sigma)$$

which is consistent with the Maximum Likelihood estimator

9:36

Motivation for Gaussian distributions

- Gaussian Bandits
- Control theory, Stochastic Optimal Control
- State estimation, sensor processing, Gaussian filtering (Kalman filtering)
- Machine Learning
- etc

9:37

Particle Approximation of a Distribution

- We approximate a distribution $p(x)$ over a continuous domain $\mathbb{R}^n$

- A particle distribution $q(x)$ is a weighed set $\mathcal{S} = \{(x^i, w^i)\}_{i=1}^N$ of N particles
  – each particle has a "location" $x^i \in \mathbb{R}^n$ and a weight $w^i \in \mathbb{R}$
  – weights are normalized, $\sum_i w^i = 1$

$$q(x) := \sum_{i=1}^N w^i \; \delta(x - x^i)$$

where $\delta(x - x^i)$ is the δ-distribution.

- Given weighted particles, we can estimate for any (smooth) f:

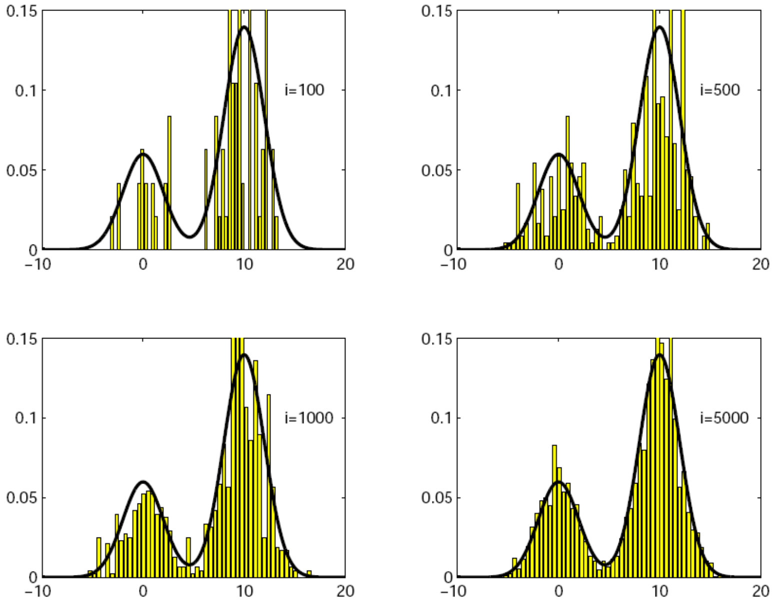
$$\langle f(x) \rangle_p = \int_x f(x) p(x) dx \approx \textstyle\sum_{i=1}^N w^i f(x^i)$$

See *An Introduction to MCMC for Machine Learning* www.cs.ubc.ca/~nando/papers/mlintro.pdf

9:38

Particle Approximation of a Distribution

Histogram of a particle representation:



9:39

Motivation for particle distributions

- Numeric representation of “difficult” distributions
  - Very general and versatile
  - But often needs many samples
- Distributions over games (action sequences), sample based planning, MCTS
- State estimation, particle filters
- etc

9:40

Utilities & Decision Theory

- Given a space of events Ω (e.g., outcomes of a trial, a game, etc) the utility is a function
$$U : \Omega \to \mathbb{R}$$
- The utility represents preferences as a single scalar – which is not always obvious (cf. multi-objective optimization)
- *Decision Theory* making decisions (that determine $p(x)$) that maximize expected utility
$$\text{E}\{U\}_p = \int_x U(x)\, p(x)$$
- Concave utility functions imply risk aversion (and convex, risk-taking)

9:41

Entropy

- The neg-log $(-\log p(x))$ of a distribution reflects something like "error":
  - neg-log of a Guassian $\leftrightarrow$ squared error
  - neg-log likelihood $\leftrightarrow$ prediction error

- The $(-\log p(x))$ is the "optimal" coding length you should assign to a symbol x. This will minimize the expected length of an encoding
$$H(p) = \int_x p(x)[-\log p(x)]$$

- The **entropy** $H(p) = \text{E}_{p(x)}\{-\log p(x)\}$ of a distribution p is a measure of uncertainty, or lack-of-information, we have about x

9:42

Kullback-Leibler divergence

- Assume you use a "wrong" distribution $q(x)$ to decide on the coding length of symbols drawn from $p(x)$. The expected length of a encoding is
$$\int_x p(x)[-\log q(x)] \ge H(p)$$

- The difference
$$D\big(p \,\|\, q\big) = \int_x p(x) \log \frac{p(x)}{q(x)} \ge 0$$
is called Kullback-Leibler divergence

Proof of inequality, using the Jenson inequality:
$$-\int_x p(x) \log \frac{q(x)}{p(x)} \ge -\log \int_x p(x) \frac{q(x)}{p(x)} = 0$$

9:43

A.4 Monte Carlo methods

9:44

Monte Carlo methods

- Generally, a Monte Carlo method is a method to generate a set of (potentially weighted) samples that approximate a distribution $p(x)$.

  In the unweighted case, the samples should be i.i.d. $x_i \sim p(x)$

  In the general (also weighted) case, we want particles that allow to estimate expectations of anything that depends on x, e.g. $f(x)$:

$$\lim_{N\to\infty} \langle f(x) \rangle_q = \lim_{N\to\infty} \sum_{i=1}^N w^i f(x^i) = \int_x f(x)\, p(x)\, dx = \langle f(x) \rangle_p$$

  In this view, Monte Carlo methods approximate an integral.
- Motivation: $p(x)$ itself is too complicated to express analytically or compute $\langle f(x) \rangle_p$ directly

- Example: What is the probability that a solitair would come out successful? (Original story by Stan Ulam.) Instead of trying to analytically compute this, generate many random solitairs and count.
- Naming: The method developed in the 40ies, where computers became faster. Fermi, Ulam and von Neumann initiated the idea. von Neumann called it "Monte Carlo" as a code name.

9:45

Rejection Sampling

- How can we generate i.i.d. samples $x_i \sim p(x)$?
- Assumptions:
  - We can sample $x \sim q(x)$ from a simpler distribution $q(x)$ (e.g., uniform), called **proposal distribution**
  - We can numerically evaluate $p(x)$ for a specific x (even if we don't have an analytic expression of $p(x)$)
  - There exists M such that $\forall_x : p(x) \le Mq(x)$ (which implies q has larger or equal support as p)

- Rejection Sampling:
  - Sample a candiate $x \sim q(x)$
  - With probability $\frac{p(x)}{Mq(x)}$ accept x and add to $\mathcal{S}$; otherwise reject
  - Repeat until $|\mathcal{S}| = n$
- This generates an unweighted sample set $\mathcal{S}$ to approximate $p(x)$

9:46

Importance sampling

- Assumptions:
  - We can sample $x \sim q(x)$ from a simpler distribution $q(x)$ (e.g., uniform)
  - We can numerically evaluate $p(x)$ for a specific x (even if we don't have an analytic expression of $p(x)$)
- Importance Sampling:
  - Sample a candiate $x \sim q(x)$
  - Add the weighted sample $(x, \frac{p(x)}{q(x)})$ to $\mathcal{S}$
  - Repeat n times
- This generates an weighted sample set $\mathcal{S}$ to approximate $p(x)$
  The weights $w_i = \frac{p(x_i)}{q(x_i)}$ are called **importance weights**

- Crucial for efficiency: a good choice of the proposal $q(x)$

9:47

Applications

- MCTS estimates the Q-function at branchings in decision trees or games
- Inference in graphical models (models involving many depending random variables)

9:48

Some more continuous distributions

Gaussian	$\mathcal{N}(x \mid a, A) = \frac{1}{\mid 2\pi A \mid^{1/2}}\ e^{-\frac{1}{2}(x-a)^\top\ A^{\text{-}1}\ (x-a)}$
Dirac or δ	$\delta(x) = \frac{\partial}{\partial x} H(x)$
Student's t (=Gaussian for $\nu \to \infty$, otherwise heavy tails)	$p(x; \nu) \propto [1 + \frac{x^2}{\nu}]^{-\frac{\nu+1}{2}}$
Exponential (distribution over single event time)	$p(x; \lambda) = [x \ge 0]\ \lambda e^{-\lambda x}$
Laplace ("double exponential")	$p(x; \mu, b) = \frac{1}{2b} e^{-\mid x-\mu \mid /b}$
Chi-squared	$p(x; k) \propto [x \ge 0]\ x^{k/2-1} e^{-x/2}$
Gamma	$p(x; k, \theta) \propto [x \ge 0]\ x^{k-1} e^{-x/\theta}$

9:49

9 Exercises

9.1 Exercise 1

There will be no credit points for the first exercise – we'll do them on the fly. (*Präsenzübung*) For those of you that had lectures with me before this is redundant—you're free to skip the tutorial.

9.1.1 Reading: Pedro Domingos

Read at least until section 5 of Pedro Domingos's *A Few Useful Things to Know about Machine Learning* http://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf. Be able to explain roughly what generalization and the bias-variance-tradeoff (Fig. 1) are.

9.1.2 Matrix equations

a) Let X, A be arbitrary matrices, A invertible. Solve for X:

$$XA + A^\top = \mathbf{I}$$

b) Let X, A, B be arbitrary matrices, $(C - 2A^\top)$ invertible. Solve for X:

$$X^\top C = [2A(X + B)]^\top$$

c) Let $x \in \mathbb{R}^n, y \in \mathbb{R}^d, A \in \mathbb{R}^{d\times n}$. A obviously *not* invertible, but let $A^\top A$ be invertible. Solve for x:

$$(Ax - y)^\top A = \mathbf{0}_n^\top$$

d) As above, additionally $B \in \mathbb{R}^{n\times n}$, B positive-definite. Solve for x:

$$(Ax - y)^\top A + x^\top B = \mathbf{0}_n^\top$$

9.1.3 Vector derivatives

Let $x \in \mathbb{R}^n, y \in \mathbb{R}^d, A \in \mathbb{R}^{d\times n}$.

a) What is $\frac{\partial}{\partial x} x$? (Of what type/dimension is this thing?)

b) What is $\frac{\partial}{\partial x}[x^\top x]$?

c) Let B be symmetric (and pos.def.). What is the minimum of $(Ax - y)^\top(Ax - y) + x^\top Bx$ w.r.t. x?

9.1.4 Coding

Future exercises will need you to code some Machine Learning methods. You are free to choose your programming language. If you're new to numerics we recommend Python (SciPy & scikit-learn) or Matlab/Octave. I'll support C++, but recommend it really only to those familiar with C++.

To get started, try to just plot the data set `http://ipvs.informatik.uni-stuttgart.de/mlr/marc/teaching/data/dataQuadReg2D.txt`, e.g. in Octave:

```
D = importdata('dataQuadReg2D.txt');
plot3(D(:,1),D(:,2),D(:,3), 'ro')
```

Or in Python

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

D = np.loadtxt('dataQuadReg2D.txt')

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot(D[:,0],D[:,1],D[:,2], 'ro')
plt.show()
```

Or you can store the grid data in a file and use gnuplot, e.g.:

```
splot 'dataQuadReg2D.txt' with points
```

For those using C++, download and test `https://github.com/MarcToussaint/rai`. In particular, have a look at test/Core/array with many examples on how to use the array class. Report on problems with installation.

9.2 Exercise 2

9.2.1 Getting Started with Ridge Regression (10 Points)

In the appendix you find starting-point implementations of basic linear regression for Python, C++, and Matlab. These include also the plotting of the data and model. Have a look at them, choose a language and understand the code in detail.

On the course webpage there are two simple data sets `dataLinReg2D.txt` and `dataQuadReg2D.txt`. Each line contains a data entry (x, y) with $x \in \mathbb{R}^2$ and $y \in \mathbb{R}$; the last entry in a line refers to y.

a) The examples demonstrate plain linear regression for `dataLinReg2D.txt`. Extend them to include a regularization parameter λ. Report the squared error on the full data set when trained on the full data set. (3 P)

b) Do the same for `dataQuadReg2D.txt` while first computing quadratic features. (4 P)

c) Implement cross-validation (slide 02:17) to evaluate the *prediction error* of the quadratic model for a third, noisy data set `dataQuadReg2D_noisy.txt`. Report 1) the squared error when training on all data (=*training error*), and 2) the mean squared error $\hat{\ell}$ from cross-validation. (3 P)

Repeat this for different Ridge regularization parameters λ. (Ideally, generate a nice bar plot of the generalization error, including deviation, for various λ.)

Python (by Stefan Otte)

```
#!/usr/bin/env python
# encoding: utf-8
"""
NOTE: the operators + - * / are element wise operation. If you want
matrix multiplication use ‘‘dot‘‘ or ‘‘mdot‘‘!
"""
from __future__ import print_function
import numpy as np
from numpy import dot
from numpy.linalg import inv
from numpy.linalg import multi_dot as mdot
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
# 3D plotting
###############################################################################
# Helper functions
def prepend_one(X):
    """prepend a one vector to X."""
    return np.column_stack([np.ones(X.shape[0]), X])

def grid2d(start, end, num=50):
    """Create an 2D array where each row is a 2D coordinate.
    np.meshgrid is pretty annoying!
    """
    dom = np.linspace(start, end, num)
    X0, X1 = np.meshgrid(dom, dom)
    return np.column_stack([X0.flatten(), X1.flatten()])

###############################################################################
# load the data
data = np.loadtxt("dataLinReg2D.txt")
print("data.shape:", data.shape)

# split into features and labels
X, y = data[:, :2], data[:, 2]
print("X.shape:", X.shape)
print("y.shape:", y.shape)

# 3D plotting
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d') # the projection arg is important!
ax.scatter(X[:, 0], X[:, 1], y, color="red")
ax.set_title("raw data")
plt.draw()

# show, use plt.show() for blocking
# prep for linear reg.
X = prepend_one(X)
print("X.shape:", X.shape)

# Fit model/compute optimal parameters beta
beta_ = mdot([inv(dot(X.T, X)), X.T, y])
print("Optimal beta:", beta_)

# prep for prediction
X_grid = prepend_one(grid2d(-3, 3, num=30))
print("X_grid.shape:", X_grid.shape)

# Predict with trained model
y_grid = dot(X_grid, beta_)
print("Y_grid.shape", y_grid.shape)

# vis the result
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d') # the projection part is important
ax.scatter(X_grid[:, 1], X_grid[:, 2], y_grid) # don't use the 1 infront
ax.scatter(X[:, 1], X[:, 2], y, color="red") # also show the real data
ax.set_title("predicted data")
plt.show()
```

C++

(by Marc Toussaint)

```
//install https://github.com/MarcToussaint/rai in $HOME/git and compile 'make -C rai/Core'
//export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/git/rai/lib
//g++ -I$HOME/git/rai/rai -L$HOME/git/rai/lib -fPIC -std=c++0x main.cpp -lCore

#include <Core/array.h>

//===========================================================================

void gettingStarted() {
  //load the data
  arr D = FILE("dataLinReg2D.txt");

  //plot it
  FILE("z.1") <<D;
  gnuplot("splot 'z.1' us 1:2:3 w p", true);

  //decompose in input and output
  uint n = D.d0; //number of data points
  arr Y = D.sub(0,-1,-1,-1).reshape(n);         //pick last column
  arr X = catCol(ones(n,1), D.sub(0,-1,0,-2)); //prepend 1s to inputs
  cout <<"X dim = " <<X.dim() <<endl;
  cout <<"Y dim = " <<Y.dim() <<endl;

  //compute optimal beta
  arr beta = inverse_SymPosDef(~X*X)*~X*Y;
  cout <<"optimal beta=" <<beta <<endl;

  //display the function
  arr X_grid = grid(2, -3, 3, 30);
  X_grid = catCol(ones(X_grid.d0,1), X_grid);
  cout <<"X_grid dim = " <<X_grid.dim() <<endl;

  arr Y_grid = X_grid * beta;
  cout <<"Y_grid dim = " <<Y_grid.dim() <<endl;
  FILE("z.2") <<Y_grid.reshape(31,31);
  gnuplot("splot 'z.1' us 1:2:3 w p, 'z.2' matrix us ($2/5-3):($1/5-3):3 w l", true);

  cout <<"CLICK ON THE PLOT!" <<endl;
}

//===========================================================================

int main(int argc, char *argv[]) {
  rai::initCmdLine(argc,argv);

  gettingStarted();
  return 0;
}
```

Matlab

(by Peter Englert)

```
clear;

% load the date
load('dataLinReg2D.txt');

% plot it
figure(1);clf;hold on;
plot3(dataLinReg2D(:,1),dataLinReg2D(:,2),dataLinReg2D(:,3),'r.');

% decompose in input X and output Y
n = size(dataLinReg2D,1);
X = dataLinReg2D(:,1:2);
Y = dataLinReg2D(:,3);

% prepend 1s to inputs
X = [ones(n,1),X];

% compute optimal beta
beta = inv(X'*X)*X'*Y;

% display the function
[a b] = meshgrid(-2:.1:2,-2:.1:2);
Xgrid = [ones(length(a(:)),1),a(:),b(:)];
Ygrid = Xgrid*beta;
Ygrid = reshape(Ygrid,size(a));
h = surface(a,b,Ygrid);
view(3);
grid on;
```

9.3 Exercise 3

(BSc Data Science students may skip b) parts.)

9.3.1 Hinge-loss gradients (5 Points)

The function $[z]_+ = \max(0, z)$ is called hinge. In ML, a hinge penalizes errors (when $z > 0$) linearly, but raises no costs at all if $z < 0$.

Assume we have a single data point (x, y^*) with class label $y^* \in \{1, .., M\}$, and the discriminative function $f(y, x)$. We penalize the discriminative function with the *one-vs-all* hinge loss

$$L^{\text{hinge}}(f) = \sum_{y \neq y^*} [1 - (f(y^*, x) - f(y, x))]_+$$

a) What is the gradient $\frac{\partial L^{\text{hinge}}(f)}{\partial f(y,x)}$ of the loss w.r.t. the discriminative values. For simplicity, distinguish the cases of taking the derivative w.r.t. $f(y^*,x)$ and w.r.t. $f(y,x)$ for $y \neq y^*$. (3 P)

b) Now assume the parameteric model $f(y,x) = \phi(x)^\top \beta_y$, where for every y we have different parameters $\beta_y \in \mathbb{R}^d$. And we have a full data set $D = \{(x_i,y_i)\}_{i=1}^n$ with class labels $y_i \in \{1,..,M\}$ and loss

$$L^{\text{hinge}}(f) = \sum_{i=1}^n \sum_{y \neq y_i} [1 - (f(y_i,x_i) - f(y,x_i))]_+$$

What is the gradient $\frac{\partial L^{\text{hinge}}(f)}{\partial \beta_y}$? (2 P)

9.3.2 Log-likelihood gradient and Hessian (5 Points)

Consider a binary classification problem with data $D = \{(x_i,y_i)\}_{i=1}^n$, $x_i \in \mathbb{R}^d$ and $y_i \in \{0,1\}$. We define

$$f(x) = \phi(x)^\top \beta \tag{10}$$

$$p(x) = \sigma(f(x)) \;, \quad \sigma(z) = 1/(1+e^{-z}) \tag{11}$$

$$L^{\text{nll}}(\beta) = -\sum_{i=1}^n \Big[y_i \log p(x_i) + (1-y_i) \log[1-p(x_i)] \Big] \tag{12}$$

where $\beta \in \mathbb{R}^d$ is a vector. (NOTE: the $p(x)$ we defined here is a short-hand for $p(y=1|x)$ on slide 03:9.)

a) Compute the derivative $\frac{\partial}{\partial \beta} L(\beta)$. Tip: use the fact $\frac{\partial}{\partial z}\sigma(z) = \sigma(z)(1-\sigma(z))$. (3 P)

b) Compute the 2nd derivative $\frac{\partial^2}{\partial \beta^2} L(\beta)$. (2 P)

9.4 Exercise 4

At the bottom several 'extra' exercises are given. These are not part of the tutorial and only for your interest.

(BSc Data Science students may skip preparing exercise 2.)

(There were questions about an API documentation of the C++ code. See `https://github.com/MarcToussaint/rai-maintenance/blob/master/help/doxygen.md`.)

9.4.1 Logistic Regression (6 Points)

On the course webpage there is a data set `data2Class.txt` for a binary classification problem. Each line contains a data entry (x,y) with $x \in \mathbb{R}^2$ and $y \in \{0,1\}$.

a) Compute the optimal parameters β and the mean neg-log-likelihood $-\frac{1}{n}\log L(\beta)$ for logistic regression using linear features. Plot the probability $P(y=1\,|\,x)$ over a 2D grid of test points. (4 P)

- Recall the objective function, and its gradient and Hessian that we derived in the last exercise:

$$L(\beta) = -\sum_{i=1}^n \log P(y_i\,|\,x_i) + \lambda\|\beta\|^2 \tag{13}$$

$$= -\sum_{i=1}^n \Big[y_i \log p_i + (1-y_i)\log[1-p_i]\Big] + \lambda\|\beta\|^2 \tag{14}$$

$$\nabla L(\beta) = \frac{\partial L(\beta)}{\partial\beta}^\top = \sum_{i=1}^n (p_i - y_i)\,\phi(x_i) + 2\lambda I\beta = X^\top(p-y) + 2\lambda I\beta \tag{15}$$

$$\nabla^2 L(\beta) = \frac{\partial^2 L(\beta)}{\partial\beta^2} = \sum_{i=1}^n p_i(1-p_i)\,\phi(x_i)\,\phi(x_i)^\top + 2\lambda I = X^\top W X + 2\lambda I \tag{16}$$

where $p(x) := P(y{=}1\,|\,x) = \sigma(\phi(x)^\top\beta),\ p_i := p(x_i),\ W := \text{diag}(p\circ(1-p))$ (17)

- Setting the gradient equal to zero can't be done analytically. Instead, optimal parameters can quickly be found by iterating Newton steps: For this, initialize $\beta = 0$ and iterate

$$\beta \leftarrow \beta - (\nabla^2 L(\beta))^{-1}\,\nabla L(\beta)\ . \tag{18}$$

  You usually need to iterate only a few times ($\sim$10) til convergence.

- As you did for regression, plot the discriminative function $f(x) = \phi(x)^\top\beta$ or the class probability function $p(x) = \sigma(f(x))$ over a grid.

Useful gnuplot commands:

```
splot [-2:3][-2:3][-3:3.5] 'model' matrix \
   us ($1/20-2):($2/20-2):3 with lines notitle
plot [-2:3][-2:3] 'data2Class.txt' \
   us 1:2:3 with points pt 2 lc variable title 'train'
```

b) Compute and plot the same for quadratic features. (2 P)

9.4.2 Structured Output (4 Points)

(Warning: This is one of these exercises that do not have "one correct solution".)

Consider data of tuples (x, y_1, y_2) where

- $x \in \mathbb{R}^d$ is the age and some other features of a spotify user
- $y_1 \in \mathbb{R}$ quantifies how much the user likes HipHop
- $y_2 \in \mathbb{R}$ quantifies how much the user likes Classic

Naively one could train separate regressions $x \mapsto y_1$ and $x \mapsto y_2$. However, it seems reasonable that somebody that likes HipHop might like Classic a bit less than average (anti-correlated).

a) Properly define a *representation* and *objective* for modelling the prediction $x \mapsto (y_1, y_2)$. (2 P)

Tip: Discriminative functions $f(y, x)$ can not only be used to define a class prediction $F(x) = \text{argmax}_y f(y, x)$, but equally also continuous predictions where $y \in \mathbb{R}$ or $y \in \mathbb{R}^O$. How can you setup and parameterize a discriminative function for the mapping $x \mapsto (y_1, y_2)$?

b) Assume you would not only like to make a deterministic prediction $x \mapsto (y_1, y_2)$, but map x to a probability distribution $p(y_1, y_2|x)$, where presumably y_1 and y_2 would be anti-correlated. How can this be done? (2 P)

c) Extra: Assume that the data set would contain a third output variable $y_3 \in \{0, 1\}$, e.g. y_3 may indicate whether the user pays for music. How could you setup learning a model $x \mapsto (y_1, y_2, y_3)$?

d) Extra: General discussion: Based on this, how are regression, classification, and conditional random fields related?

9.4.3 Extra: Discriminative Function in Logistic Regression

Logistic Regression (slide 03:19) defines class probabilities as proportional to the exponential of a discriminative function:

$$P(y|x) = \frac{\exp f(x, y)}{\sum_{y'} \exp f(x, y')}$$

Prove that, in the binary classification case, you can assume $f(x, 0) = 0$ without loss of generality.

This results in

$$P(y = 1|x) = \frac{\exp f(x, 1)}{1 + \exp f(x, 1)} = \sigma(f(x, 1)).$$

(Hint: first assume $f(x, y) = \phi(x, y)^\top \beta$, and then define a new discriminative function f' as a function of the old one, such that $f'(x, 0) = 0$ and for which $P(y|x)$ maintains the same expressibility.)

9.4.4 Extra: Special cases of CRFs

Slide 03:31 summarizes the core equations for CRFs.

a) Confirm the given equations for $\nabla_\beta Z(x,\beta)$ and $\nabla^2_\beta Z(x,\beta)$ (i.e., derive them from the definition of $Z(x,\beta)$).

b) Binary logistic regression is clearly a special case of CRFs. Sanity check that the gradient and Hessian given on slide 03:20 can alternatively be derived from the general expressions for $\nabla_\beta L(\beta)$ and $\nabla^2_\beta L(\beta)$ on slide 03:31. (The same is true for the multi-class case .)

c) Proof that ordinary ridge regression is a special case of CRFs if you choose the discriminative function $f(x,y) = -y^2 + 2y\phi(x)^\top\beta$.

9.5 Exercise 5

In these two exercises you'll program a NN from scratch, use neural random features for classification, and train it too. Don't use tensorflow yet, but the same language you used for standard regression & classification. Take slide 04:14 as reference for NN equations.

(DS BSc students may skip 2 b-c, i.e. should at least try to code/draft also the backward pass, but ok if no working solutions.)

9.5.1 Programming your own NN – NN initialization and neural random features (5 Points)

(Such an approach was (once) also known as Extreme Learning.)

A standard NN structure is described by $h_{0:L}$, which describes the dimensionality of the input (h_0), the dimensionality of all hidden layers ($h_{1:L\text{-}1}$), and the dimensionality of the output h_L.

a) Code a routine "forward(x, β)" that computes $f_\beta(x)$, the forward propagation of the network, for a NN with given structure $h_{0:L}$. Note that for each layer $l = 1,..,L$ we have parameters $W_l \in \mathbb{R}^{h_L \times h_{L\text{-}1}}$ and $b_l \in \mathbb{R}^{h_l}$. Use the leaky ReLu activation function. (2 P)

b) Write a method that initializes all weights such that for each neuron, the $z_i = 0$ hyperplane is located randomly, with random orientation and random offset (follow slide 04:21). Namely, choose each $W_{l,i\cdot}$ as Gaussian with sdv $1/\sqrt{h_{l\text{-}1}}$, and choose the biases $b_{l,i} \sim \mathcal{U}(-1,1)$ uniform. (1 P)

c) Consider again the classification data set `data2Class.txt`, which we also used in the previous exercise. In each line it has a two-dimensional input and the output $y_i \in \{0,1\}$.

Use your NN to map each input x to features $\phi(x) = x_{L\text{-}1}$, then use these features as input to logistic regression as done in the previous exercise. (Initialize a separate β and optimize by iterating Newton steps.)

First consider just $L = 2$ (just one hidden layer and $x_{L\text{-}1}$ are the features) and $h_1 = 300$. (2 P)

Extra) How does it perform if we initialize all $b_l = 0$? How would it perform if the input would be rescaled $x \leftarrow 10^5 x$? How does the performance vary with h_1 and with L?

9.5.2 Programming your own NN – Backprop & training (5 Points)

We now also train the network using backpropagation and hinge loss. We test again on `data2Class.txt`. As this is a binary classification problem we only need one output neuron $f_\beta(x)$. If $f_\beta(x) > 0$ we classify 1, otherwise we classify 0.

Reuse the "forward(x, β)" coded above.

a) Code a routine "backward(δ_{L+1}, x, w)", that performs the backpropagation steps and collects the gradients $\frac{d\ell}{dw_l}$.

For this, let us use a hinge loss. In the binary case (when you use only one output neuron), it is simplest to redefine $y \in \{-1, +1\}$, and define the hinge loss as $\ell(f, y) = \max(0, 1 - fy)$, which has the loss gradient $\delta_L = -y[1 - yf > 0]$ at the output neuron.

Run forward and backward propagation for each x, y in the dataset, and sum up the respective gradients. (2 P)

b) Code a routine which optimizes the parameters using gradient descent:

$$\forall_{l=1,..,L} : \; W_l \leftarrow W_l - \alpha \frac{d\ell}{dW_l} \;, \quad b_l \leftarrow b_l - \alpha \frac{d\ell}{db_l}$$

with step size $\alpha = .01$. Run until convergence (should take a few thousand steps). Print out the loss function ℓ at each 100th iteration, to verify that the parameter optimization is indeed decreasing the loss. (2 P)

c) Run for $h = (2, 20, 1)$ and visualize the prediction by plotting $\sigma(f_\beta(x))$ over a 2-dimensional grid. (1 P)

9.6 Exercise 6

(DS BSc students can skip the exercise 2b and 3.)

9.6.1 Getting started with tensorflow (4 Points)

Tensorflow (`https://www.tensorflow.org/`) is one of the state-of-the-art computation graph libraries mostly used for implementing neural networks. We recommend using the Python API for this example. Install the tensorflow library with pip using the command `pip install tensorflow --user` or follow the instructions on the webpage for your platform/language.

For the logistic regression, we used the following objective function:

$$L(\beta) = -\sum_{i=1}^{n} \Big[y_i \log p_i + (1 - y_i) \log[1 - p_i] \Big] \tag{19}$$

$$\text{where} \quad p(x) := P(y{=}1 \,|\, x) = \sigma(x^\top \beta) \;, \quad p_i := p(x_i) \tag{20}$$

a) Implement the loss function by using standard tensor commands like `tf.math.sigmoid()`,`tf.tensordot()`,`tf.math.reduce_sum()`. Define the variables $X \in \mathbb{R}^{100\times 3}$, $y \in \mathbb{R}^{100}$ and $\beta \in \mathbb{R}^3$ as `tf.placeholder`. Store the computation graph and display it in a browser using tensorboard. (2 P)

Hints:

– You can save the computation graph by using the following command:
`writer = tf.summary.FileWriter('logs', sess.graph)`

– You can display it by running tensorboard from the command line:
`tensorboard --logdir logs`

– Then open the given url in a browser.

b) Run a session to compute the loss, gradient and hessian. Feed random values into the input placeholders. Gradient and Hessian can be calculated by `tf.gradients()` and `tf.hessians()`. Compare it to the analytical solution using the same random values. (2 P)

Code calculating the analytical solutions of the loss, the gradient and the hessian in python:

```
def numpy_equations(X, beta, y):
   p = 1. / (1. + np.exp(-np.dot(X, beta)))
   L = -np.sum(y * np.log(p) + ((1. - y) * np.log(1.-p)))
   dL = np.dot(X.T, p - y)
   W = np.identity(X.shape[0]) * p * (1. - p)
   ddL = np.dot(X.T, np.dot(W, X))
   return L, dL, ddL
```

9.6.2 Classification with NNs in tensorflow (6 Points)

Now you will directly use tensorflow commands for creating neural networks.

a) We want to verify our results for classification on the dataset "data2Class.txt" by implementing the NN using tensorflow. Create two dense layers with ReLU activation function and $h_1 = h_2 = 100$. Map to one output neuron (i.e. $h_3 = 1$) without activation function. Display the computation graph as in the previous exercise. Use `tf.losses.hinge_loss()` as a loss function and the Adam Optimizer to train the network. Run the training and plot the final result. (3 P)

Hints:

– There are many tutorials online, also on `www.tensorflow.org`. HOWEVER, most of them use the *keras* conventions to first abstractly declare the model structure, then compile it into actual tensorflow structure (Factory pattern). You can use this, but to really learn tensorflow we recommend using the direct tensorflow methods to create models instead.
– Here is an example that declares an input variable, two hidden layers, an output layer, a target variable, and a loss variable:

```
input = tf.placeholder(shape=[None,2], dtype=tf.float32)
target_output = tf.placeholder('float')

relu_layer_operation = tf.layers.Dense(100,
                     activation=tf.nn.leaky_relu,
                     kernel_initializer=tf.initializers.random_uniform(-.1,.1),
                     bias_initializer=tf.initializers.random_uniform(-1.,1.))

linear_layer_operation = tf.layers.Dense(1,
                     activation=None,
                     kernel_initializer=tf.initializers.random_uniform(-.1,.1),
                     bias_initializer=tf.initializers.random_uniform(-.01,.01))

hidden1 = relu_layer_operation(input)
hidden2 = relu_layer_operation(hidden1)
model_output = linear_layer_operation(hidden2)

loss = tf.reduce_mean(tf.losses.hinge_loss(logits=model_output, labels=target_output))
```

– Use any tutorial to realize the training of such a model.

b) Now we want to use a neural network on real images. Download the BelgiumTS[1] dataset from: https://btsd.ethz.ch/shareddata/BelgiumTSC/BelgiumTSC_Training.zip (Training data) and https://btsd.ethz.ch/shareddBelgiumTSC/BelgiumTSC_Testing.zip (Test data). The dataset consists of traffic signs according to 62 different classes. Create a neural network architecture and train it on the training dataset. You can use any architecture you want but at least use one convolutional layer. Report the classification error on the test set. (3 P)

Hints: Use `tf.layers.Conv2D` to create convolutional layers, and `tf.contrib.layers.flatten` to reshape an image layer into a vector layer (as input to a dense layer). The following code can be used to load data, rescale it and display images:

[1]Belgium traffic sign dataset; Radu Timofte*, Markus Mathias*, Rodrigo Benenson, and Luc Van Gool, Traffic Sign Recognition - How far are we from the solution?, International Joint Conference on Neural Networks (IJCNN 2013), August 2013, Dallas, USA

```
import os
import skimage
from skimage import transform
from skimage.color import rgb2gray
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

def load_data(data_directory):
    directories = [d for d in os.listdir(data_directory)
                   if os.path.isdir(os.path.join(data_directory, d))]
    labels = []
    images = []
    for d in directories:
        label_directory = os.path.join(data_directory, d)
        file_names = [os.path.join(label_directory, f)
                      for f in os.listdir(label_directory)
                      if f.endswith(".ppm")]
        for f in file_names:
            images.append(skimage.data.imread(f))
            labels.append(int(d))
    return np.array(images), np.array(labels)

def plot_data(signs, labels):
    for i in range(len(signs)):
        plt.subplot(4, len(signs)/4 + 1, i+1)
        plt.axis('off')
        plt.title("Label {0}".format(labels[i]))
        plt.imshow(signs[i])
        plt.subplots_adjust(wspace=0.5)
    plt.show()


images, labels = load_data("./Training")

# display 30 random images
randind = np.random.randint(0, len(images), 30)
plot_data(images[randind], labels[randind])

images = rgb2gray(np.array([transform.resize(image, (50, 50)) for image in images]))  # convert to 50x50
```

9.6.3 Bonus: Stochastic Gradient Descent (3 Points)

(Bonus means: The extra 3 points will count to your total, but not to the required points (from which you eventually need 50%).)

We test SDG on a squared function $f(x) = \frac{1}{2}x^\top Hx$. A Newton method would need access to the exact Hessian H and directly step to the optimum $x^* = 0$. But SDG only has access to an estimate of the gradient. Ensuring proper convergence is much more difficult for SGD.

Let $x \in \mathbb{R}^d$ for $d = 1000$. Generate a sparse random matrix $J \in \mathbb{R}^{n\times d}$, for $n = 10^5$ as follows: In each row, fill in 10 random numbers drawn from $\mathcal{N}(0, \sigma^2)$ at random places. Each row either has $\sigma = 1$ or $\sigma = 100$, chosen randomly. We now define $H = J^\top J$. Note that $H = \sum_{i=1}^n J_i^\top J_i$ is a sum of rank-1 matrices.

a) Given this setup, simulate a stochastic gradient descent. (2P)

1. Initialize $x = \mathbf{1}_d$.
2. Choose $K = 32$ random integers $i_k \in \{1, .., n\}$, where $k = 1, .., K$. These indicate which data points we see in this iteration.

3. Compute the stochastic gradient estimate
$$g = \frac{1}{K}\sum_{k=1}^{K} J_{i_k}^\top (J_{i_k} x)$$
where J_{i_k} is the i_kth row of J.
4. For logging: Compute the full error $\ell = \frac{1}{2n} x^\top H x$ and the stochastic mini batch error $\hat\ell = \frac{1}{2K}\sum_{k=1}^K (J_{i_k} x)^2$, and write them to a log file for later plotting.
5. Update x based on g using plain gradient descent with fixed step size, and iterate from (ii).

b) Plot the *learning curves*, i.e., the full and the stochastic error. How well do they match? In what sense does optimization converge? Discuss the *stationary distribution* of the optimum. (1P)

c) Extra: Test variants: exponential cooling of the learning rate, Nesterov momentum, and ADAM.

9.7 Exercise 7

(DS BSc students can skip the exercise 3.)

9.7.1 Kernels and Features (4 Points)

Reconsider the equations for Kernel Ridge regression given on slide 05:3, and – if features are given – the definition of the kernel function and $\kappa(x)$ in terms of the features as given on slide 05:2.

a) Prove that Kernel Ridge regression with the linear kernel function $k(x,x') = 1 + x^\top x'$ is equivalent to Ridge regression with linear features $\phi(x) = \begin{pmatrix}1\\x\end{pmatrix}$. (1 P)

b) In Kernel Ridge regression, the optimal function is of the form $f(x) = \kappa(x)^\top\alpha$ and therefore linear in $\kappa(x)$. In plain ridge regression, the optimal function is of the form $f(x) = \phi(x)^\top\beta$ and linear in $\phi(x)$. Prove that choosing $k(x,x') = (1 + x^\top x')^2$ implies that $f(x) = \kappa(x)^\top\alpha$ is a second order polynomial over x. (2 P)

c) Equally, note that choosing the squared exponential kernel $k(x,x') = \exp(-\gamma\,|\,x - x'\,|\,^2)$ implies that the optimal $f(x)$ is linear in radial basis function (RBF) features. Does this necessarily impliy that Kernel Ridge regression with squared exponential kernel, and plain Ridge regression with RBF features are exactly equivalent? (Equivalent means, have the same optimal function.) Distinguish the cases $\lambda = 0$ (no regularization) and $\lambda > 0$. (1 P)

(Voluntary: Practically test yourself on the regression problem from Exercise 2, whether Kernel Ridge Regression and RBF features are exactly equivalent.)

9.7.2 Kernel Construction (4 Points)

For a non-empty set X, a kernel is a symmetric function $k : X \times X \to \mathbb{R}$. Note that the set X can be arbitrarily structured (real vector space, graphs, images, strings and so on). A very important class of useful kernels for machine learning are positive definite kernels. A kernel is called *positive definite*, if for all arbitrary finite subsets $\{x_i\}_{i=1}^n \subseteq X$ the corresponding *kernel matrix* K with elements $K_{ij} = k(x_i, x_j)$ is positive *semi*-definite,

$$\alpha \in \mathbb{R}^n \;\Rightarrow\; \alpha^\top K \alpha \geq 0\,. \tag{21}$$

Let $k_1, k_2 : X \times X \to \mathbb{R}$ be two positive definite kernels. Often, one wants to construct more complicated kernels out of existing ones. Proof that

1. $k(x, x') = k_1(x, x') + k_2(x, x')$
2. $k(x, x') = c \cdot k_1(x, x') \quad \text{for} \quad c \geq 0$
3. $k(x, x') = k_1(x, x') \cdot k_2(x, x')$
4. $k(x, x') = k_1(f(x), f(x')) \quad \text{for} \quad f : X \to X$

are positive definite kernels.

9.7.3 Kernel logistic regression (2 Points)

The “kernel trick” is generally applicable whenever the “solution” (which may be the predictive function $f^{\text{ridge}}(x)$, or the discriminative function, or principal components...) can be written in a form that only uses the kernel function $k(x, x')$, but never features $\phi(x)$ or parameters β explicitly.

Derive a kernelization of Logistic Regression. That is, think about how you could perform the Newton iterations based only on the kernel function $k(x, x')$.

Tips: Reformulate the Newton iterations

$$\beta \leftarrow \beta - (\boldsymbol{X}^\top W \boldsymbol{X} + 2\lambda I)^{\text{-}1}\, [\boldsymbol{X}^\top(\boldsymbol{p} - \boldsymbol{y}) + 2\lambda I \beta] \tag{22}$$

using the two Woodbury identities

$$(X^\top W X + A)^{\text{-}1} X^\top W = A^{\text{-}1} X^\top (X A^{\text{-}1} X^\top + W^{\text{-}1})^{\text{-}1} \tag{23}$$

$$(X^\top W X + A)^{\text{-}1} = A^{\text{-}1} - A^{\text{-}1} X^\top (X A^{\text{-}1} X^\top + W^{\text{-}1})^{\text{-}1} X A^{\text{-}1} \tag{24}$$

Note that you'll need to handle the $\boldsymbol{X}^\top(\boldsymbol{p} - \boldsymbol{y})$ and $2\lambda I \beta$ differently.

Then think about what is actually been iterated in the kernalized case: surely we cannot iteratively update the optimal parameters, because we want to rewrite equations to never touch β or $\phi(x)$ explicitly.

9.8 Exercise 8

(DS BSc students should nominally achieve 8 Pts on this sheet.)

9.8.1 PCA derived (6 Points)

For data $D = \{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$, we introduced PCA as a method that finds lower-dimensional representations $z_i \in \mathbb{R}^p$ of each data point such that $x_i \approx Vz_i + \mu$. PCA chooses V, μ and z_i to minimize the reproduction error

$$\sum_{i=1}^n \|x_i - (Vz_i + \mu)\|^2 \ .$$

We derive the solution here step by step.

1. Find the optimal latent representation z_i of a data point x_i as a function of V and μ. (1P)
2. Find *an* optimal offset μ. (1P)

   (Hint: there is a whole subspace of solutions to this problem. Verify that your solution is consistent with (i.e. contains) $\mu = \frac{1}{n}\sum_i x_i$).
3. Find optimal projection vectors $\{v_i\}_{i=1}^p$, which make up the projection matrix

   $$V = \begin{bmatrix} | & & | \\ v_1 & \dots & v_p \\ | & & | \end{bmatrix} \tag{25}$$

   (2P)

   Guide:

   – Given a projection V, any vector can be split in orthogonal components which belong to the projected subspace and its complement (which we call W). $x = VV^\top x + WW^\top x$.

   – For simplicity, let us work with the centered datapoints $\tilde{x}_i = x_i - \mu$.

   – The optimal projection V is the one which minimizes the discarded components $WW^\top \tilde{x}_i$.

   $$\hat{V} = \underset{V}{\text{argmin}} \sum_{i=1}^n \|WW^\top \tilde{x}_i\|^2 = \sum_{i=1}^n \|\tilde{x}_i - VV^\top \tilde{x}_i\|^2 \tag{26}$$

   – Don't try to solve computing gradients and setting them to zero. Instead, use the fact that $VV^\top = \sum_{i=1}^p v_i v_i^\top$, and the singular value decomposition of $\sum_{i=1}^n \tilde{x}_i \tilde{x}_i^\top = \tilde{X}^\top \tilde{X} = EDE^T$.
4. In the above, is the orthonormality of V an essential assumption? (1P)
5. Prove that you can compute the V also from the SVD of X (instead of $X^\top X$). (1P)

9.8.2 PCA and reconstruction on the Yale face database (5 Points)

On the webpage find and download the Yale face database `http://ipvs.informatik.uni-stuttgart.de/mlr/marc/teaching/data/yalefaces.tgz`. (Optionally use `yalefaces_c` which is slightly cleaned version of the same dataset). The file contains gif images of 165 faces.

1. Write a routine to load all images into a big data matrix $X \in \mathbb{R}^{165\times 77760}$, where each row contains a gray image.

   In Octave, images can easily be read using `I=imread("subject01.gif");` and `imagesc(I);`. You can loop over files using `files=dir(".");` and `files(:).name`. Python tips:

```
import matplotlib.pyplot as plt
import scipy as sp
plt.imshow(plt.imread(file_name))
```

2. Compute the mean face $\mu = \frac{1}{n}\sum_i x_i$ and center the whole data matrix, $\tilde X = X - \mathbf{1}_n\mu^\top$.

3. Compute the singular value decomposition $\tilde X = UDV^\top$ for the centered data matrix.

   In Octave/Matlab, use `[U, S, V] = svd(X, "econ")`, where the `"econ"` ensures you don't run out of memory.

   In python, use

```
import scipy.sparse.linalg as sla
u, s, vt = sla.svds(X, k=num_eigenvalues)
```

4. Find the p-dimensional representations $Z = \tilde X V_p$, where $V_p \in \mathbb{R}^{77760\times p}$ contains only the first p columns of V (Depending on which language / library you use, verify that the eigenvectors are returned in eigenvalue-descending order, otherwise you'll have to find the correct eigenvectors manually). Assume $p = 60$. The rows of Z represent each face as a p-dimensional vector, instead of a 77760-dimensional image.

5. Reconstruct all faces by computing $X' = \mathbf{1}_n\mu^\top + \boldsymbol{Z}V_p^\top$ and display them; Do they look ok? Report the reconstruction error $\sum_{i=1}^n \|x_i - x_i'\|^2$.

   Repeat for various PCA-dimensions $p = 5, 10, 15, 20 \ldots$.

9.9 Exercise 9

(DS BSc students may skip exercise 2, but still please read about Mixture of Gaussians and the explanations below.)

Introduction

There is no lecture on Thursday. To still make progress, please follow this guide to learn some new material yourself. The subject is k-means clustering and Mixture of Gaussians.

k-means clustering: The method is fully described on slide 06:36 of the lecture. Again, I present the method as derived from an optimality principle. Most other references described k-means clustering just as a procedure. Also wikipedia https://en.wikipedia.org/wiki/K-means_clustering gives a more verbose explaination of this procedure. In my words, this is the procedure:

– We have data $D = \{x_i\}_{i=1}^n$, with $x_i \in \mathbb{R}^d$. We want to cluster the data in K different clusters. K is chosen ad-hoc.
– Each cluster is represented only by its mean (or center) $\mu_k \in \mathbb{R}^d$, for $k = 1, .., K$.
– We initially assign each μ_k to a random data point, $\mu_k \leftarrow x_i$ with $i \sim \mathcal{U}\{1, .., n\}$
– The algorithm also maintains an assignment mapping $c : \{1, .., n\} \to \{1, .., K\}$, which assigns each data point x_i to a cluster $k = c(i)$
– For given centers μ_k, we update all assignments using

$$\forall_i : \; c(i) \leftarrow \operatorname*{argmin}_{c(i)} \sum_j (x_j - \mu_{c(j)})^2 = \operatorname*{argmin}_k (x_i - \mu_k)^2 \;,$$

which means we assign x_i to the cluster with the nearest center μ_k.
– For given assignments $c(i)$, we update all centers using

$$\forall_k : \; \mu_k \leftarrow \operatorname*{argmin}_{\mu_k} \sum_i (x_i - \mu_{c(i)})^2 = \frac{1}{|c^{\text{-}1}(k)|} \sum_{i \in c^{\text{-}1}(k)} x_i \;,$$

that is, we set the centers equal to the mean of the data points assigned to the cluster.
– The last two steps are iterated until the assignment does not vary.

Based on this, solve exercise 1.

Mixture of Gaussians: Mixture of Gaussians (MoG) are very similar to k-means clustering. Its objective (expected likelihood maximization) is based on a probabilistic data model, which we do not go into detail here. Slide 06:40 only gives the relevant equations. A much more complete derivation of MoG as instance of Expectation Maximization is found in https://ipvs.informatik.uni-stuttgart.

de/mlr/marc/teaching/15-MachineLearning/08-graphicalModels-Learni
pdf. Bishop's book https://www.microsoft.com/en-us/research/people/cmbishop/#!prml-book also gives a very good introduction. But we only need the procedural understanding here:

- We have data $D = \{x_i\}_{i=1}^n$, with $x_i \in \mathbb{R}^d$. We want to cluster the data in K different clusters. K is chosen ad-hoc.
- Each cluster is represented by its mean (or center) $\mu_k \in \mathbb{R}^d$ and a covariance matrix Σ_k. This covariance matrix describes an ellipsoidal shape of each cluster.
- We initially assign each μ_k to a random data point, $\mu_k \leftarrow x_i$ with $i \sim \mathcal{U}\{1, .., n\}$, and each covariance matrix to uniform (if the data is roughly uniform).
- The core difference to k-means: The algorithm also maintains a *probabilistic* (or soft) assignment mapping $q_i(k) \in [0, 1]$, such that $\sum_{k=1}^K q_i(k) = 1$. The number $q_i(k)$ is the probability of assigning data x_i to cluster k (or the probability that data x_i originates from cluster k). So, each data index i is mapped to a probability over k, rather than a specific k as in k-means.
- For given cluster parameters μ_k, Σ_k, we update all the probabilistic assignments using

$$\begin{aligned}\forall_{i,k} :\ q_i(k) &\leftarrow \mathcal{N}(x_i \,|\, \mu_k, \Sigma_k) = \frac{1}{|\,2\pi\Sigma_k\,|^{1/2}}\, e^{-\frac{1}{2}(x_i-\mu_k)^\top\,\Sigma_k^{-1}\,(x_i-\mu_k)} \\ \forall_{i,k} :\ q_i(k) &\leftarrow \frac{1}{\sum_{k'} q_i(k')} q_i(k)\end{aligned}$$

  where the second line normalizes the probabilistic assignments to ensure $\sum_{k=1}^K q_i(k) = 1$.
- For given probabilistic assignments $q_i(k)$, we update all cluster parameters using

$$\begin{aligned}\forall_k :\ \mu_k &\leftarrow \frac{1}{\sum_i q_i(k)}\ \textstyle\sum_i q_i(k)\, x_i \\ \forall_k :\ \Sigma_k &\leftarrow \frac{1}{\sum_i q_i(k)}\ \textstyle\sum_i q_i(k)\, x_i x_i^\top - \mu_k \mu_k^\top\ ,\end{aligned}$$

  where μ_k is the weighed mean of the data assigned to cluster k (weighted with $q_i(k)$), and similarly for Σ_k.
- In this description, I skipped another parameter, π_k, which is less important and we can discuss in class.

Based on this, solve exercise 2.

Exercise 3 is extra, meaning, it's a great exercise, but beyond the default work scope.

9.9.1 Clustering the Yale face database (6 Points)

On the webpage find and download the Yale face database http://ipvs.informatik.uni-stuttgart.de/mlr/marc/teaching/data/yalefaces_cropBackground.tgz. The file contains gif images of 136 faces.

We'll cluster the faces using k-means in $K = 4$ clusters.

a) Compute a k-means clustering starting with random initializations of the centers. Repeat k-means clustering 10 times. For each run, report on the clustering error $\min \sum_i (x_i - \mu_{c(i)})^2$ and pick the best clustering. Display the center faces μ_k and perhaps some samples for each cluster. (3 P)

b) Repeat the above for various K and plot the clustering error over K. (1 P)

c) Repeat the above on the first 20 principal components of the data. Discussion in the tutorial: Is PCA the best way to reduce dimensionality as a precursor to k-means clustering? What would be the 'ideal' way to reduce dimensionality as precursor to k-means clustering? (2 P)

9.9.2 Mixture of Gaussians (4 Points)

Download the data set `mixture.txt` from the course webpage, containing $n = 300$ 2-dimensional points. Load it in a data matrix $\boldsymbol{X} \in \mathbb{R}^{n\times 2}$.

a) Implement the EM-algorithm for a Gaussian Mixture on this data set. Choose $K = 3$. Initialize by choosing the three means μ_k to be different randomly selected data points x_i (i random in $\{1, .., n\}$) and the covariances $\Sigma_k = \mathbf{I}$ (a more robust choice would be the covariance of the whole data). Iterate EM starting with the first E-step (computing probabilistic assignments) based on these initializations. Repeat with random restarts—how often does it converge to the optimum? (3 P)

b) Do exactly the same, but this time initialize the posterior $q_i(k)$ randomly (i.e., assign each point to a random cluster: for each point x_i select $k' = rand(1 : K)$ and set $q_i(k) = [k = k']$); then start EM with the first M-step. Is this better or worse than the previous way of initialization? (1 P)

9.9.3 Extra: Graph cut objective function & spectral clustering

One of the central messages of the whole course is: To solve (learning) problems, first formulate an objective function that defines the problem, then derive algorithms to find/approximate the optimal solution. That should also hold for clustering.

k-means finds centers μ_k and assignments $c : i \mapsto k$ to minimize $\min \sum_i (x_i - \mu_{c(i)})^2$.

An alternative class of objective functions for clustering are graph cuts. Consider n data points with similarities w_{ij}, forming a weighted graph. We denote by $W = (w_{ij})$ the symmetric weight matrix, and $D = \text{diag}(d_1, .., d_n)$, with $d_i = \sum_j w_{ij}$, the degree matrix. For simplicitly we consider only 2-cuts, that is, cutting the graph in two disjoint clusters, $C_1 \cup C_2 = \{1, .., n\}$, $C_1 \cap C_2 = \emptyset$. The normalized cut objective is

$$\text{RatioCut}(C_1, C_2) = \Big(1/|C_1| + 1/|C_2|\Big) \sum_{i\in C_1, j\in C_2} w_{ij}$$

a) Let $f_i = \begin{cases} +\sqrt{|C_2|/|C_1|} & \text{for } i \in C_1 \\ -\sqrt{|C_1|/|C_2|} & \text{for } i \in C_2 \end{cases}$ be a kind of indicator function of the clustering. Prove that

$$f^\top(D - W)f = n \,\text{RatioCut}(C_1, C_2)$$

b) Further prove that $\sum_i f_i = 0$ and $\sum_i f_i^2 = n$.

Note (to be discussed in the tutorial in more detail): *Spectral clustering* addresses

$$\min f^\top(D - W)f \quad \text{s.t.} \quad \sum_i f_i = 0 \;, \quad \|f\|_2 = 1$$

by computing eigenvectors f of the graph Laplacian $D - W$ with smallest eigenvalues. This is a relaxation of the above problem that minimizes over continuous functions $f \in \mathbb{R}^n$ instead of discrete clusters C_1, C_2. The resulting eigen functions are "approximate indicator functions of clusters". The algorithms uses k-means clustering in this coordinate system to explicitly decide on the clustering of data points.

9.10 Exercise 10

(DS BSc students please try to complete the full exercise this time.)

9.10.1 Method comparison: kNN regression versus Neural Networks (5 Points)

k-nearest neighbor regression is a very simple lazy learning method: Given a data set $D = \{(x_i, y_i)\}_{i=1}^n$ and query point x^*, first find the k nearest neighbors $K \subset \{1, .., n\}$. In the simplest case, the output $y = \frac{1}{K} \sum_{k \in K} y_k$ is then the average of these k nearest neighbors. In the classification case, the output is the majority vote of the neighbors.

(To make this smoother, one can weigh each nearest neighbor based on the distance $|x^* - x_k|$, and use local linear or polynomial (logistic) regression. But this is not required here.)

On the webpage there is a data set `data2ClassHastie.txt`. Your task is to compare the performance of kNN classification (with basic kNN majority voting) with a neural network classifier. (If you prefer, you can compare kNN against another classifier such as logistic regression with RBF features, instead of neural networks. The class boundaries are non-linear in x.)

As part of this exercise, discuss how a fair and rigorous comparison between two ML methods is done.

9.10.2 Gradient Boosting for classification (5 Points)

Consider the following *weak learner* for classification: Given a data set $D = \{(x_i, y_i)\}_{i=1}^n, y_i$ $\{-1, +1\}$, the weak learner picks a single i^* and defines the discriminative function

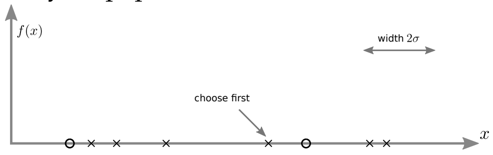
$$f(x) = \alpha e^{-(x - x_{i^*})^2 / 2\sigma^2} ,$$

with fixed width σ and variable parameter α. Therefore, this weak learner is parameterized only by i^* and $\alpha \in \mathbb{R}$, which are chosen to minimize the neg-log-likelihood

$$L^{\text{nll}}(f) = -\sum_{i=1}^n \log \sigma(y_i f(x_i)) .$$

a) Write down an explicit pseudo code for gradient boosting with this weak learner. By "pseudo code" I mean explicit equations for every step that can directly be implemented. This needs to be specific for this particular learner and loss. (3 P)

b) Here is a 1D data set, where $\circ$ are 0-class, and $\times$ 1-class data points. "Simulate" the algorithm graphically on paper. (2 P)



Extra) If we would replace the neg-log-likelihood by a hinge loss, what would be the relation to SVMs?

9.11 Exercise 11

(DS BSc students may skip coding exercise 3, but should be able to draw on the board what the result would look like.)

9.11.1 Sum of 3 dices (3 Points)

You have 3 dices (potentially fake dices where each one has a different probability table over the 6 values). You're given all three probability tables $P(D_1)$, $P(D_2)$, and $P(D_3)$. Write down the equations and an algorithm (in pseudo code) that computes the conditional probability $P(S|D_1)$ of the sum of all three dices conditioned on the value of the first dice.

9.11.2 Product of Gaussians (3 Points)

A Gaussian distribution over $x \in \mathbb{R}^n$ with mean μ and covariance matrix Σ is defined as

$$\mathcal{N}(x \,|\, \mu, \Sigma) = \frac{1}{|\, 2\pi\Sigma \,|^{1/2}} \; e^{-\frac{1}{2}(x-\mu)^\top \, \Sigma^{\text{-}1} \, (x-\mu)}$$

Multiplying probability distributions is a fundamental operation, and multiplying two Gaussians is needed in many models. From the definition of a n-dimensional Gaussian, prove the general rule

$$\mathcal{N}(x \,|\, a, A) \; \mathcal{N}(x \,|\, b, B) \propto \mathcal{N}(x \,|\, (A^{\text{-}1} + B^{\text{-}1})^{\text{-}1}(A^{\text{-}1}a + B^{\text{-}1}b), (A^{\text{-}1} + B^{\text{-}1})^{\text{-}1}) \,.$$

where the proportionality $\propto$ allows you to drop all terms independent of x.

Note: The so-called canonical form of a Gaussian is defined as $\mathcal{N}[x \,|\, \bar a, \bar A] = \mathcal{N}(x \,|\, \bar A^{\text{-}1}\bar a, \bar A^{\text{-}1})$; in this convention the product reads much nicher: $\mathcal{N}[x \,|\, \bar a, \bar A] \; \mathcal{N}[x \,|\, \bar b, \bar B] \propto \mathcal{N}[x \,|\, \bar a + \bar b, \bar A + \bar B]$. You can first prove this before proving the above, if you like.

9.11.3 Gaussian Processes (5 Points)

Consider a Gaussian Process prior $P(f)$ over functions defined by the mean function $\mu(x) = 0$, the γ-exponential covariance function

$$k(x, x') = \exp\{-|(x - x')/l|^\gamma\}$$

and an observation noise $\sigma = 0.1$. We assume $x \in \mathbb{R}$ is 1-dimensional. First consider the standard squared exponential kernel with $\gamma = 2$ and $l = 0.2$.

a) Assume we have two data points $(-0.5, 0.3)$ and $(0.5, -0.1)$. Display the posterior $P(f|D)$. For this, compute the mean posterior function $\hat f(x)$ and the standard deviation function $\hat\sigma(x)$ (on the 100 grid points) exactly as on slide 08:10, using $\lambda = \sigma^2$. Then plot $\hat f$, $\hat f + \hat\sigma$ and $\hat f - \hat\sigma$ to display the posterior mean and standard deviation. (3 P)

b) Now display the posterior $P(y^*|x^*, D)$. This is only a tiny difference from the above (see slide 08:8). The mean is the same, but the variance of y^* includes additionally the obseration noise σ^2. (1 P)

c) Repeat a) & b) for a kernel with $\gamma = 1$. (1 P)

9.12 Exercise 12

All exercises are voluntary, for you to collect extra points.

9.12.1 Autoencoder (7 Points)

On the webpage find and download the Yale face database `http://ipvs.informatik.uni-stuttgart.de/mlr/marc/teaching/data/yalefaces_cropBackground.tgz`. The file contains gif images of 136 faces.

We want to compare two methods (Autoencoder vs PCA) to reduce the dimensionality of this dataset. This means that we want to create and train a neural network to find a lower-dimensional representation of our data. Recall the slides and exercises about dimensionality reduction, neural networks and especially Autoencoders (slide 06:10).

a) Create a neural network using tensorflow (or any other framework, e.g., keras) which takes the images as input, creates a lower-dimensional representation with dimensionality $p = 60$ in the hidden layer (i.e., a layer with 60 neurons) and outputs the reconstructed images. The loss function should compare the original image with the reconstructed one. After having trained the network, reconstruct all faces and display some examples. Report the reconstruction error $\sum_{i=1}^n \|x_i - x'_i\|^2$. (5P)

b) Use PCA to reduce the dimensionality of the dataset to $p = 60$ as well (e.g. use your code from exercise e08:02). Reconstruct all faces using PCA and display some examples. Report the reconstruction error $\sum_{i=1}^n \|x_i - x'_i\|^2$. Compare the reconstructions and the error from PCA with the results from the Autoencoder. Which one works better? (2P)

Extra) Repeat for various dimensions $p = 5, 10, 15, 20 \dots$

9.12.2 Special cases of CRFs (3 Points)

Slide 03:31 summarizes the core equations for CRFs.

a) Confirm the given equations for $\nabla_\beta Z(x, \beta)$ and $\nabla^2_\beta Z(x, \beta)$ (i.e., derive them from the definition of $Z(x, \beta)$). (1P)

b) Binary logistic regression is clearly a special case of CRFs. Sanity check that the gradient and Hessian given on slide 03:20 can alternatively be derived from the general expressions for $\nabla_\beta L(\beta)$ and $\nabla^2_\beta L(\beta)$ on slide 03:31. (The same is true for the multi-class case.) (1P)

c) Proof that ordinary ridge regression is a special case of CRFs if you choose the discriminative function $f(x, y) = -y^2 + 2y\phi(x)^\top\beta$. (1P)

Exam Preparation Tip: These are the headings of all questions that appeared in previous exams – no guarantee that similar ones might appear!

– Gymnastics
– True or false?
– Bayesian reasoning
– Dice rolling
– Coin flipping
– Gaussians & Bayes
– Linear Regression
– Features
– Features & Kernels
– Unusual Kernels
– Empirically Estimating Variance
– Logistic regression
– Logistic regression & log-likelihood gradient
– Discriminative Function
– Principle Component Analysis
– Clustering
– Neural Network
– Bayesian Ridge Regression and Gaussian Processes
– Ridge Regression in the Bayesian view
– Bayesian Predictive Distribution
– Bootstrap & combining learners
– Local Learning
– Boosting
– Joint Clustering & Regression

Not relevant this year (was focus in earlier years)

– Probabilistic independence
– Logistic regression & SVM
– Graphical models & Inference
– Graphical models and factor graphs
– Random forests

Index