

Intro to Machine Learning, and basics on Gaussian Process Regression

Simon Scheidegger
simon.scheidegger@unil.ch

Rochester, November 19th, 2019

Today's Roadmap

I. Detour: What is Machine Learning? (Demystifying ML)

- Supervised Machine Learning
- Unsupervised Machine Learning
- Machine Learning in Dynamic Economic Models?

II. Introduction to Gaussian Process Regression

- Recall Gaussian Distributions
- Making Predictions using noise-free observations
- Making Predictions using noisy observations

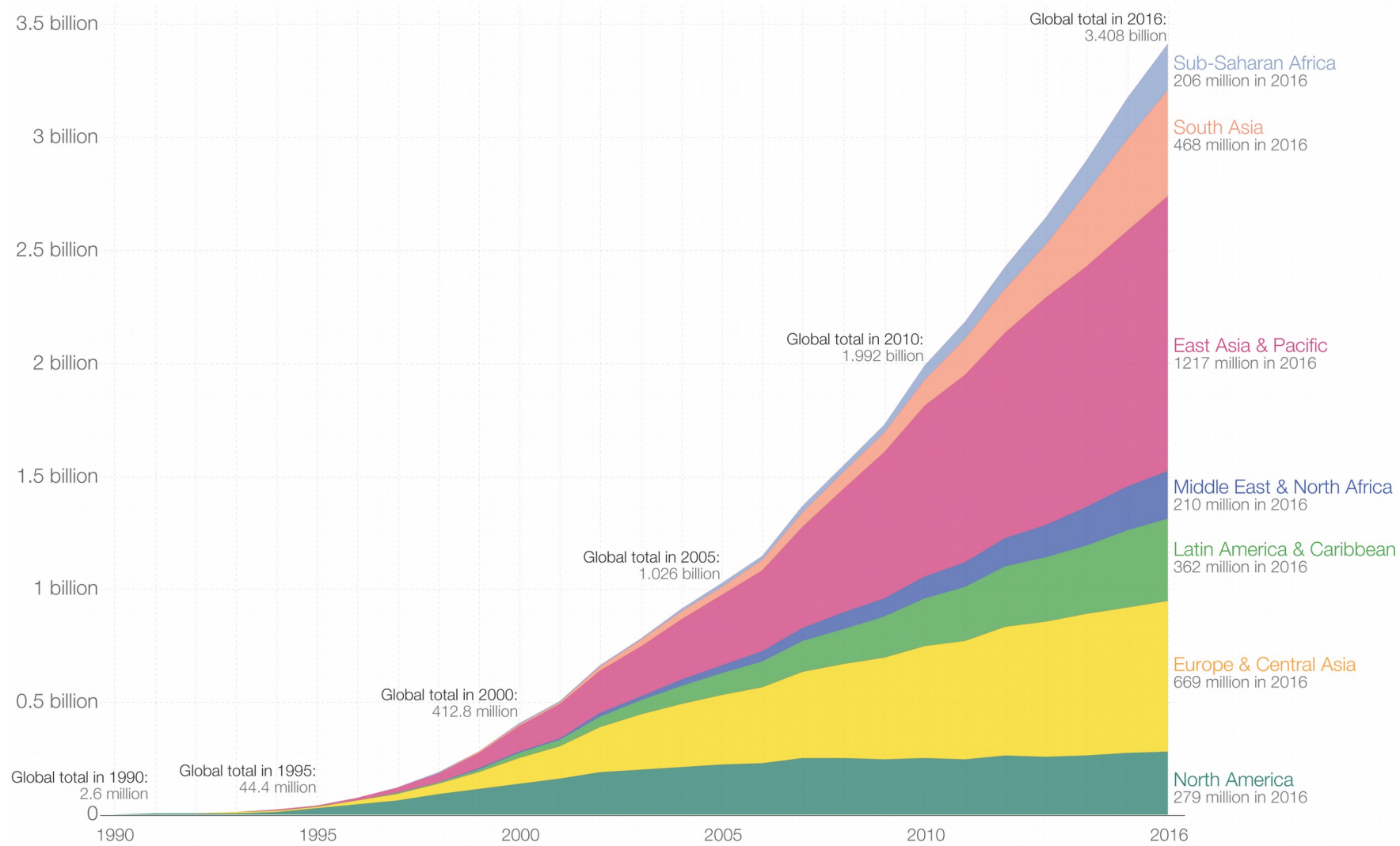
We are drowning in information and starving for knowledge.

—John Naisbitt

Big Data and it's availability

<https://ourworldindata.org/internet>

Internet users by world region since 1990



Data source: Based on data from the World Bank and data from the International Telecommunications Union. Internet users are people with access to the worldwide network. The interactive data visualization is available at [OurWorldinData.org](https://ourworldindata.org). There you find the raw data and more visualizations on this topic. Licensed under CC-BY-SA by the author Max Roser.

Big Data and it's availability (cont'd)

<http://www.live-counter.com/how-big-is-the-internet>

- Size of the internet as we speak: **14.469.600 Petabytes (~14.5 ZB)**
- **It is doubling in size every two years!**

1 Gigabyte	~ 1000 MB
1 Terabyte	~ 1000 GB
1 Petabyte	~ 1000 TB
1 Exabyte	~ 1000 PB
1 Zettabyte	~ 1000 EB

1 Gigabyte: An author takes **50 years for writing every week a book** with about 190 pages, more specifically, with 383,561 characters. This would be a billion letters or bytes.

1 Exabyte: 212 million DVDs weighing 3,404 tons.

1 Zettabyte: 1,000,000,000,000,000,000.000 bytes or characters. Printed on graph paper (with one in letter in each mm² square) would be a paper measuring a billion km. The entire surface of the earth (510 million km²) would be **covered by a layer of paper almost twice**.

Other sources of “Big Data”

- ♦ **Scientific experiments**

- Cern (e.g., LHC)
generates ~ **25 petabytes** per year (2012).
- LIGO
generates ~ **1 Petabyte** per year.



<https://home.cern/>

- ♦ **Numerical computations**

- ♦ ...



<https://www.olcf.ornl.gov/summit>



<https://www.ligo.caltech.edu>

Why Machine Learning?

- Machine learning aims at **gaining insights from data** and **making predictions** based on it.
- **Build a model that is a good and useful approximation to the data.**
- Machine learning methods have been investigated for more than 60 years, but became mainstream only recently due to **more data being available** and advances in computing power ("**Moore's Law**").
- There is no need to "learn" to calculate for example the payroll.
- Learning is used when:
 - **Human expertise does not exist** (navigating on Mars).
 - **Humans are unable to explain their expertise** (speech recognition).
 - **Solution changes in time** (routing on a computer network).
 - **Solution needs to be adapted to particular cases** (user biometrics).
- You're relying on machine learning every day, maybe without being aware of it! → you certainly use a Smart Phone?

Some terminology

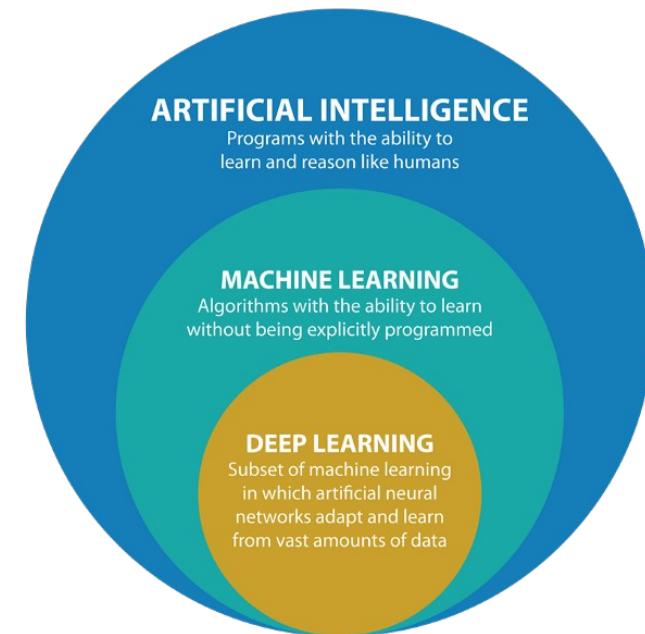
Figs. from F. Chollet (2018) – ML with Python

Artificial intelligence (AI)

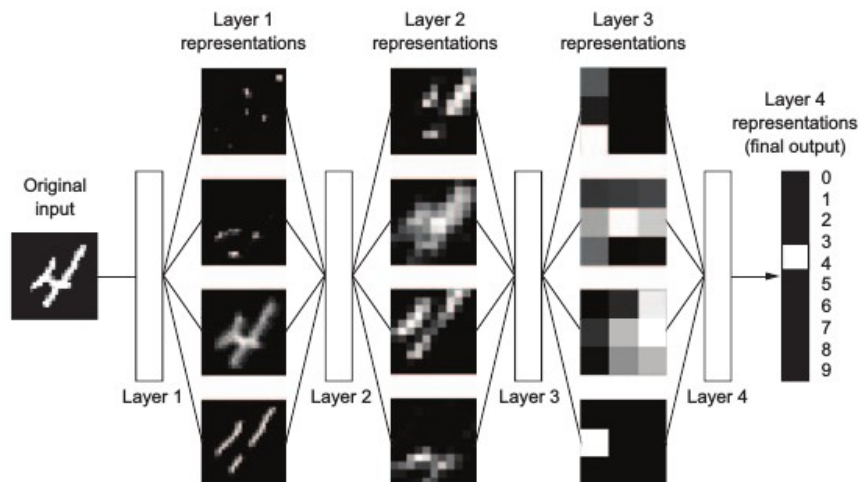
Can computers be made to “think” ?—a question whose ramifications we’re still exploring today.

→ A concise definition of the AI field would be as follows:
the effort to automate intellectual tasks normally performed by humans.

Machine learning (ML)

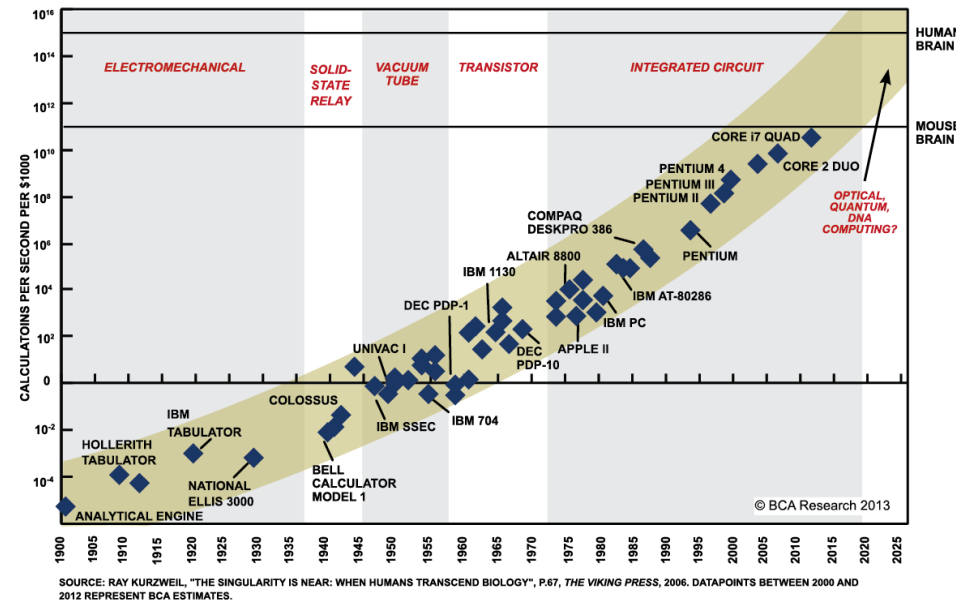
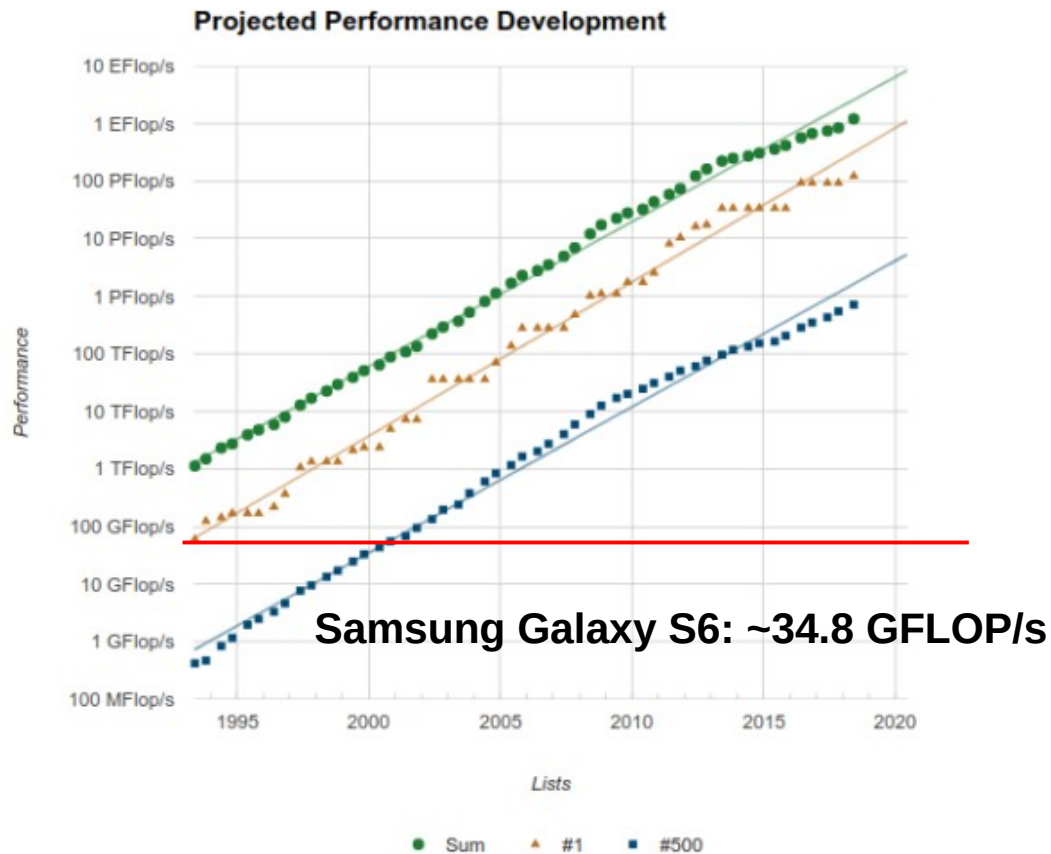


Deep Learning as a particular example of an ML technique



Moore's Law

<http://www.extremetech.com/extreme/210872-extremetech-explains-what-is-moores-law>
<https://www.top500.org>



→ Exponential growth in compute power allows us to move away from stylized models towards “realistically-sized” problems.

Applications of ML

- ♦ **Supervised Learning**

assume that training data is available from which we can **learn to predict** a target feature based on other features (e.g., monthly rent based on area).

- **Classification**

- **Regression**

- ♦ **Unsupervised Learning**

take a given dataset and aim at gaining insights by identifying patterns, e.g., by grouping similar data points.

- ♦ **Reinforcement Learning**

Supervised Regression – an example

- Regression aims at predicting a numerical target feature based on one or multiple other (numerical) features.
- Example: Price of a used car.

x : car attributes

y : price

$y = h(x | \theta)$

$h()$: model

θ : parameters

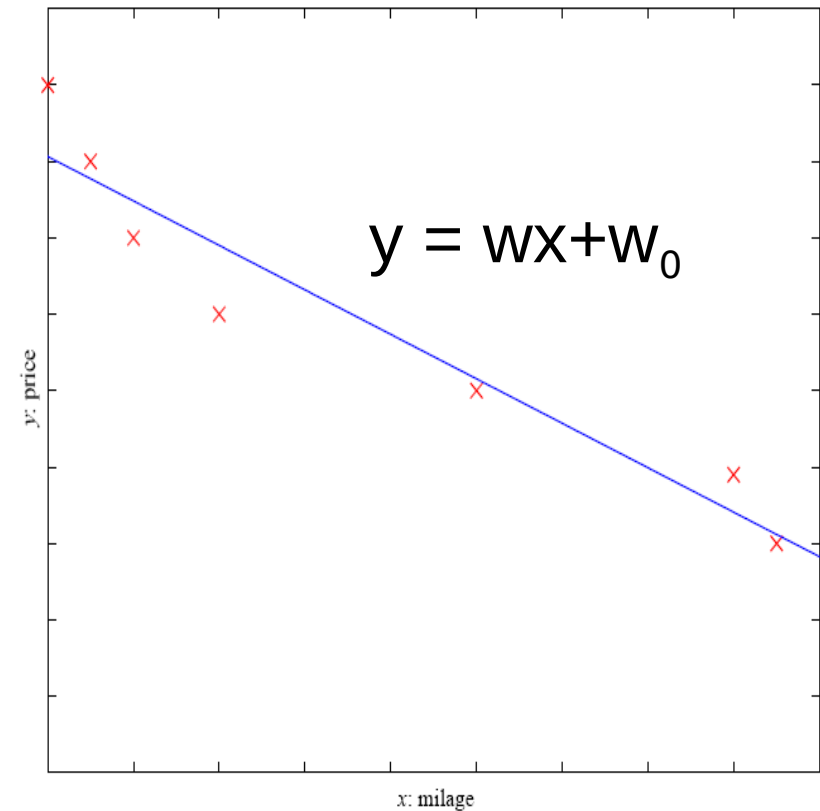


Fig. from Alpaydin (2014)

Supervised Classification – an example

Example 1: Spam Classification

- Decide which emails are Spam and which are not.
- Goal: Use emails seen so far to produce a good prediction rule for future data.



Example 2: Credit Scoring

Differentiating between low-risk and high-risk customers from their income and savings.

Discriminant: IF income $> \theta_1$ AND savings $> \theta_2$
THEN low-risk ELSE high-risk

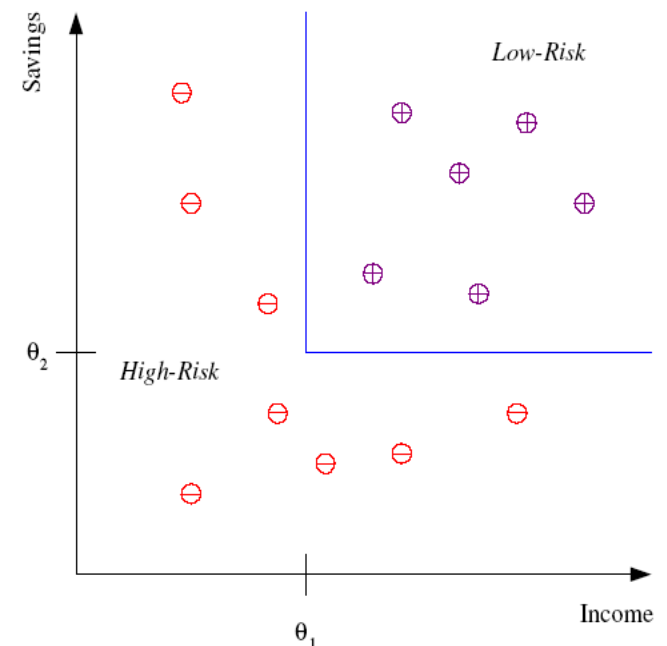


Fig. from Alpaydin (2014)

Handwritten Digit Classification

Movie – from the early 90ies

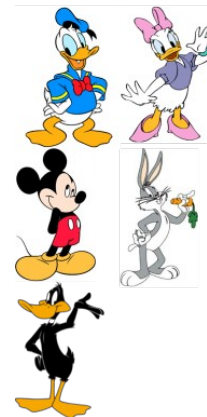
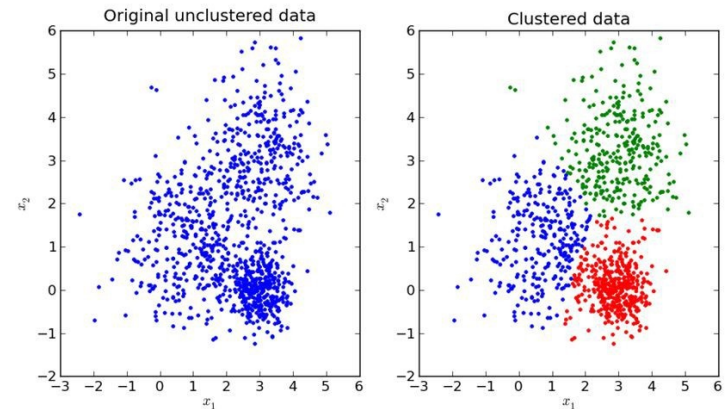
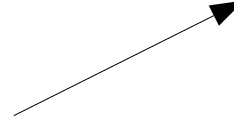
We have come a long way since then...

→ Handwritten Digit Classification – by Yann Lecun

<https://www.youtube.com/watch?v=yxuRnBEczUU>

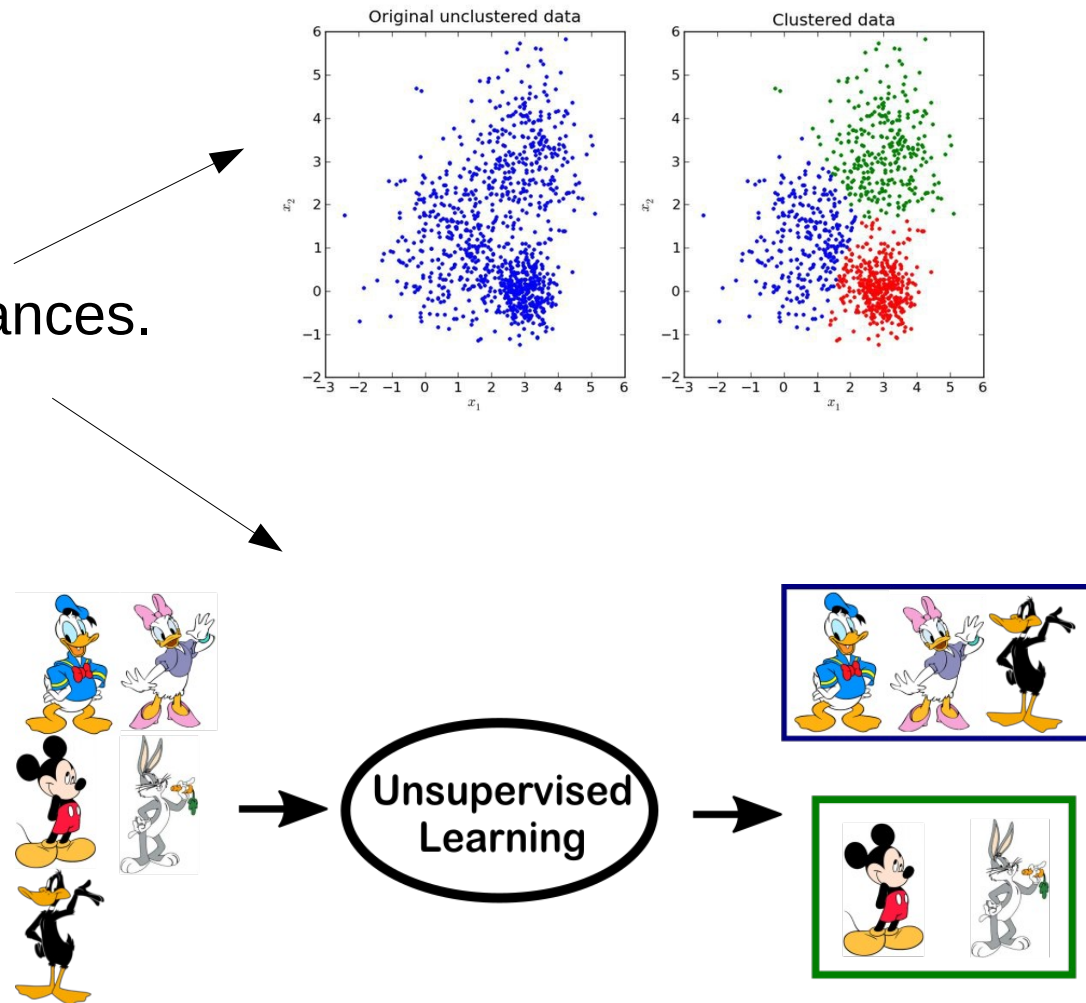
Unsupervised Learning

- Learning “what normally happens”.
- No output.
- Clustering: Grouping similar instances.
- Example applications:
 - Customer segmentation.
 - Image compression: Color quantization.
 - Bio-informatics: Learning motifs.



Unsupervised Learning

- Learning “what normally happens”.
- No output.
- Clustering: Grouping similar instances.
- Example applications:
 - Customer segmentation.
 - Image compression: Color quantization.
 - Bio-informatics: Learning motifs.



Self-driving Cars

Carnegie Mellon University – 1990ies

ALVINN: Autonomous Land Vehicle In a Neural Network

Waymo / Google – 2017

<https://www.youtube.com/watch?v=B8R148hFxPw>



- Classification
- Regression
- Reinforcement learning

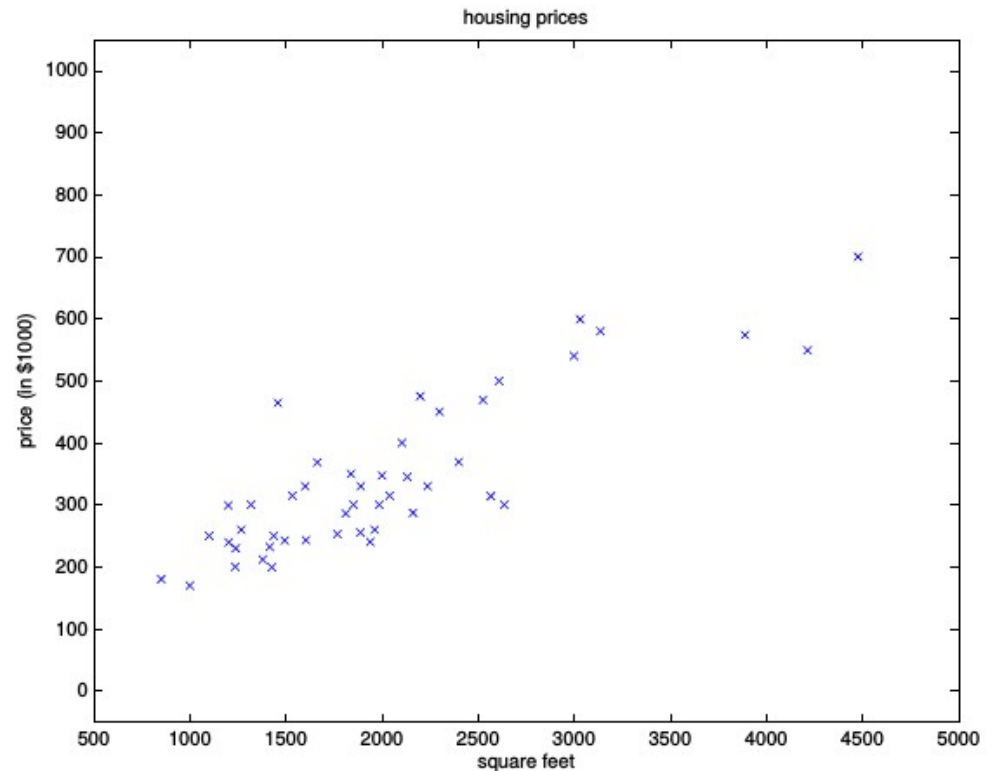


Building an ML Algorithm

- ♦ **Optimize a performance criterion** using example data or past experience.
- ♦ **Role of Statistics**: Inference from a sample.
- ♦ **Role of Computer science**: Efficient algorithms to
 - Solve the optimization problem.
 - Representing and evaluating the model for inference.

Building an ML Algorithm (II)

Living area (feet ²)	Price (1000\$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮



→ Given data like this, how can we learn to predict the prices of other houses as a function of the size of their living areas?

Building an ML Algorithm (III)

cf. Andrew Ng's ML course <http://cs229.stanford.edu/>

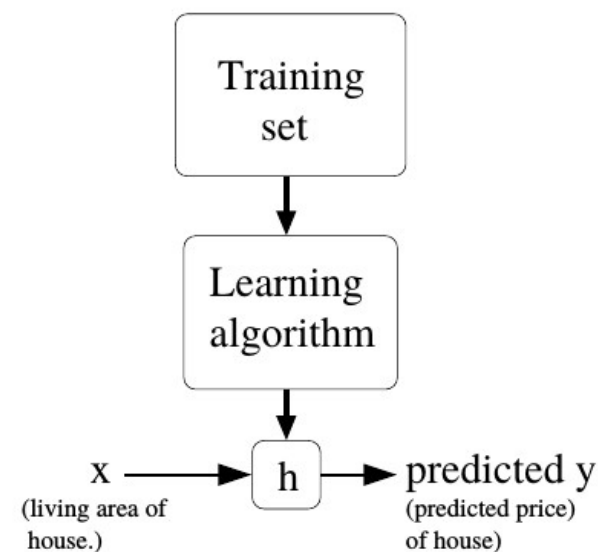
$x(i)$: “input” variables (living area in this example),
also called input features

$y(i)$: “output” / target variable that we are trying to predict (price).

Training example:
a pair $(x(i), y(i))$.

Training set:
a list of m training examples $\{(x(i), y(i)); i = 1, \dots, m\}$

→ To perform supervised learning, we must decide
how we're going to represent
functions/hypotheses h in a computer.



Building an ML Algorithm (IV)

Model / Hypothesis:

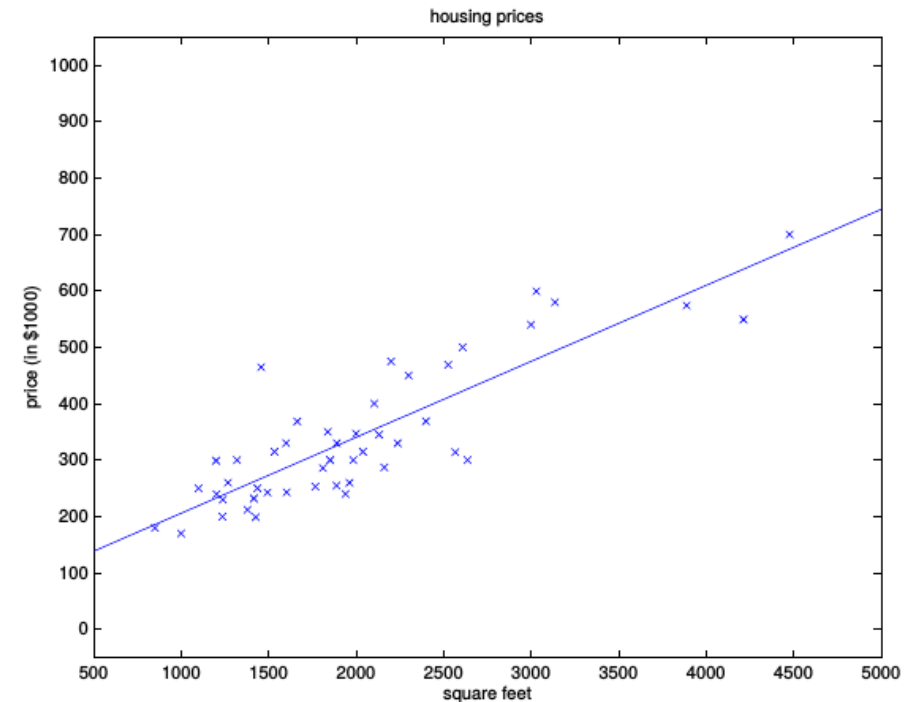
$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

θ_i 's: parameters

Cost Function:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

→ Minimize $J(\theta)$ in order to obtain the coefficients θ .



Building an ML Algorithm (V)

All Models are wrong, but some models are useful – (Box & Draper (1987), p.424)

In General: Machine learning (e.g., supervised) in 3 Steps:

- Choose a **model** $h(x|\theta)$.
- Define a **cost function** $J(\theta|x)$.
- **Optimization procedure** to find θ^* that minimizes $J(\theta)$.

Computationally, we need:
data, linear algebra, statistics tools, and optimization routines.

“Easy-to-use” Open Source Libraries

<https://www.tensorflow.org/>



<https://keras.io/>



<https://scikit-learn.org/stable/>

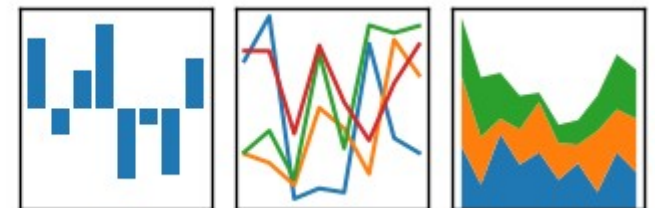


<https://pandas.pydata.org/>

...

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$

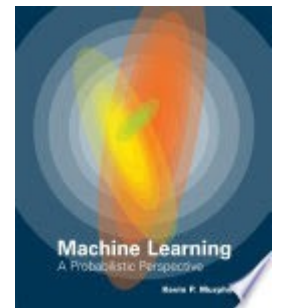


Some source materials

Machine Learning: a Probabilistic Perspective

K. Murphy, MIT Press, 2012

<https://www.cs.ubc.ca/~murphyk/MLbook/index.html>

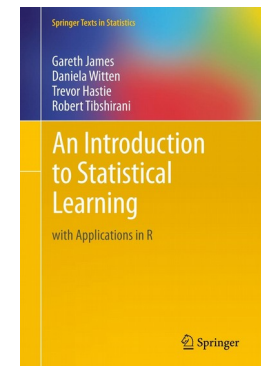


An Introduction to Statistical Learning

Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani

Springer, 8th edition, 2017

<https://www-bcf.usc.edu/~gareth/ISL/>

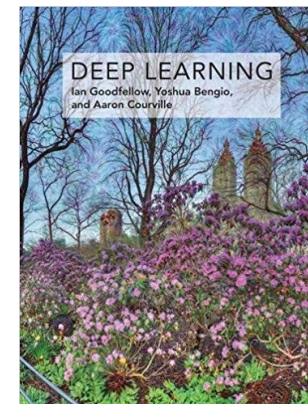


Deep Learning

Ian Goodfellow and Yoshua Bengio and Aaron Courville

MIT Press 2016

<http://www.deeplearningbook.org/>



Machine Learning and dynamic models?

Recall e.g., Dynamic Programming

The solution is approached in the limit as $j \rightarrow \infty$ by iterations on at every coordinate of the discretized grid.

$$\underline{V_{j+1}(x)} = \max_u \{ r(x, u) + \beta \underline{V_j(\tilde{x})} \}$$

s.t.

$$\tilde{x} = g(x, u)$$

x: grid point, describes your system.
State-space potentially **high-dimensional, and irregularly shaped, i.e., non-hypercubic.**

'old solution':

high-dimensional function,
approximated by **SOME**
Interpolation method on which we
Interpolate → **PREDICT**

One obvious way to use ML methods in dynamic models

Machine Learning to solve dynamic models?

Supervised ML:

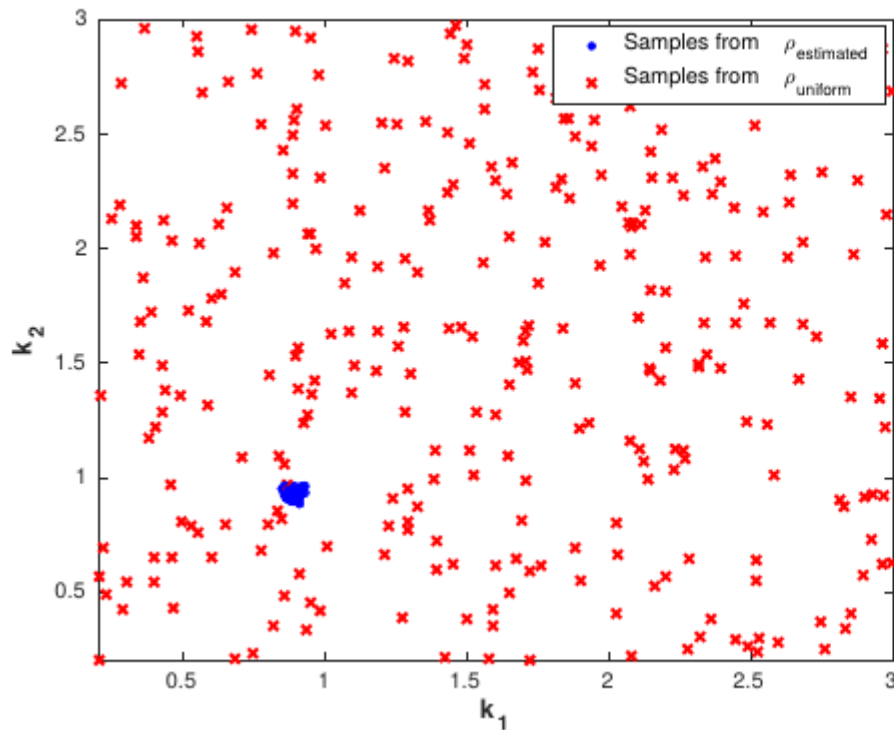
- Regression Methods → Approximate Functions.
- We are (most of the time) in control of the data-generating process.

Unsupervised ML:

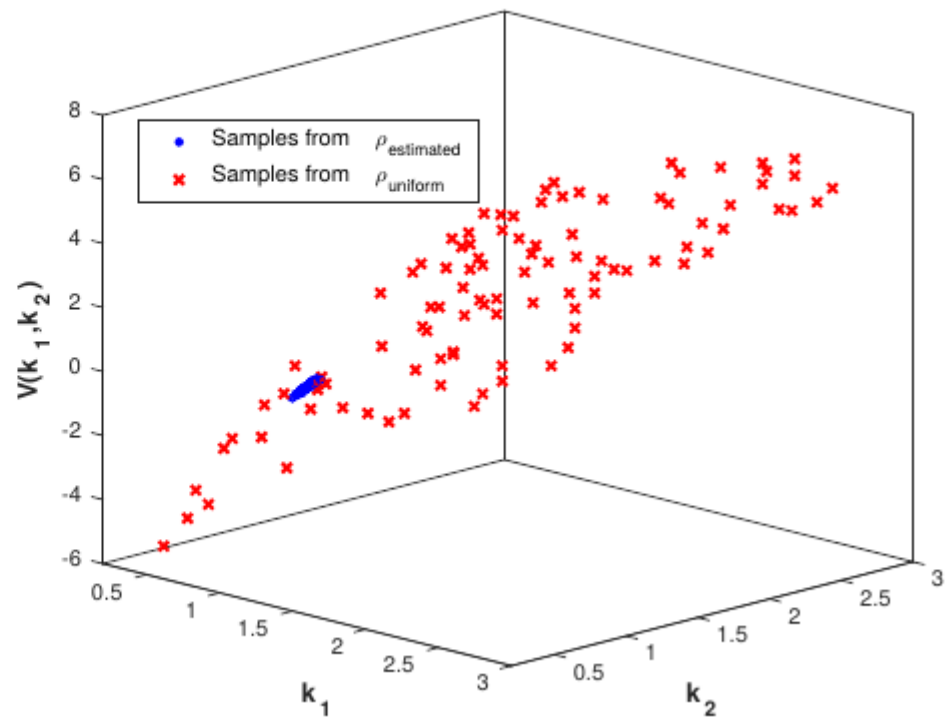
- Describe Ergodic & Feasible sets and sample from within them → Clustering Methods.
- Dimension-reduction → Deal with the Curse of Dimensionality.

Want to solve e.g. Dynamic Models on high-dimensional, irregularly-shaped state-spaces

cf. Scheidegger & Bilionis (2019)

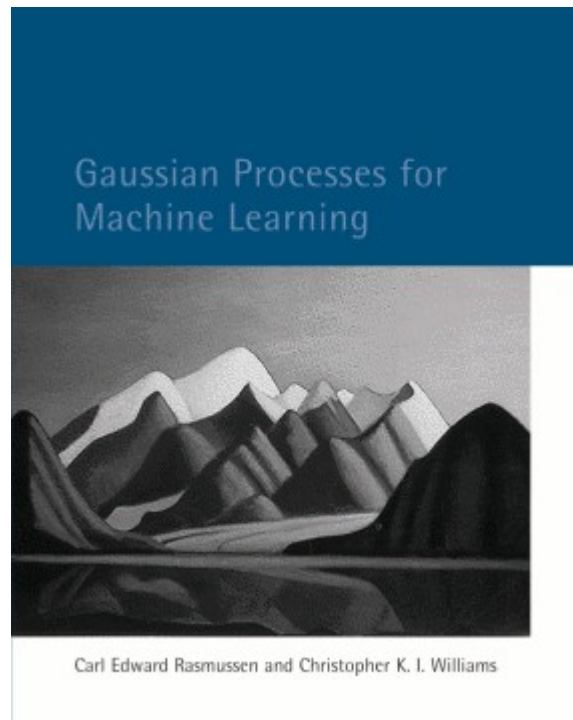


Blue: ergodic set.
Red: Computational domain



Blue: Value function, evaluated on ergodic set
Red: Value function, evaluated on the entire comp. domain

II. Introduction to Gaussian Process Regression



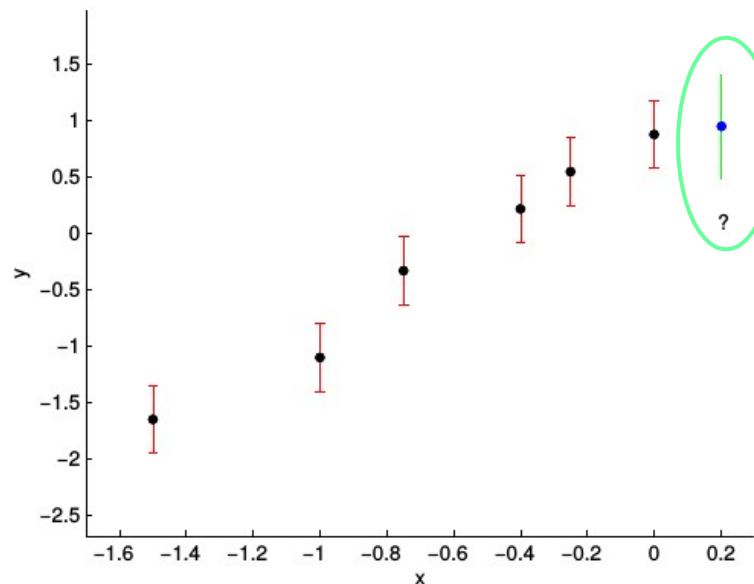
<http://www.gaussianprocess.org/gpml/>

Aim of Regression

- Given some (potential) noisy **observations** of a dependent variable at certain values of the **independent variable x** , what is our **best estimate of the dependent variable y at a new value, x_*** ?
- Let **f** denote an (unknown) function which maps inputs x to outputs

$$f: X \rightarrow Y$$

- Modeling a function **f** means **mathematically representing the relation between inputs and outputs**.
- Often times, the **shape of the underlying function might be unknown**, the function can be hard to evaluate, or other requirements might complicate the process of information acquisition.



Choosing a model

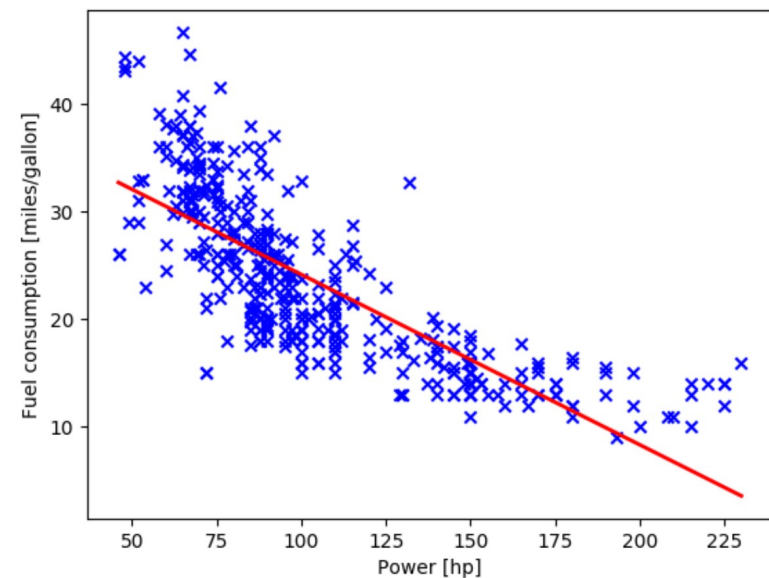
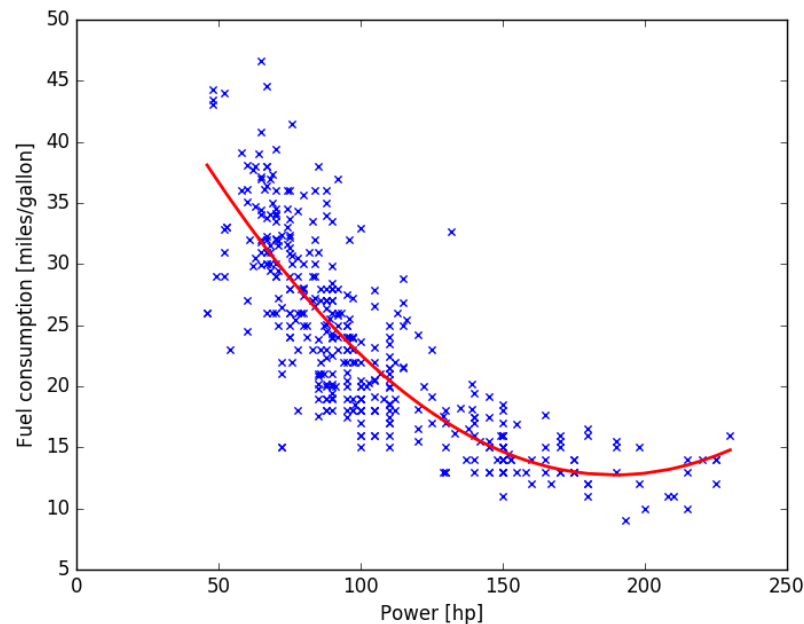
- If we expect the underlying function $f(x)$ to be linear, and can make some assumptions about the input data, we might use a least-squares method to fit a straight line (linear regression).
- Moreover, if we suspect $f(x)$ may also be quadratic, cubic, or even non-polynomial, we can use the principles of model selection to choose among the various possibilities.

Model Selection

Example data set by <https://archive.ics.uci.edu/ml/datasets/Auto+MPG>

One common approach to reliably assess the quality of a machine learning model and avoid over-fitting is to **randomly split the available data** into

- **training data** (~70% of the data) is used for determining optimal coefficients.
- **validation data** (~20% of the data) is used for **model selection** (e.g., fixing degree of polynomial, selecting a subset of features, etc.)
- **test data** (~10% of the data) is used to measure the quality that is reported.



For completeness: Polynomial Regression in Python

Code and data set here: [Lecture_2/code/polyreg.py](#)

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model, preprocessing

# read cars dataset, a clean data set
cars = pd.read_csv('auto-mpg.data.txt', header=None, sep='\s+')

# extract mpg values
y = cars.iloc[:,0].values

# extract horsepower values
X = cars.iloc[:,3].values
X = X.reshape(X.size, 1)

# precompute polynomial features
poly = preprocessing.PolynomialFeatures(2)
Xp = poly.fit_transform(X)

# fit linear regression model
reg = linear_model.LinearRegression()
reg.fit(Xp,y)

# coefficients
reg.intercept_ # 56.900099702112925
reg.coef_ # [-0.46618963, 0.00123054]

# compute correlation coefficient
np.corrcoef(reg.predict(Xp),y) # 0.82919179 (from 0.77842678)

# compute mean squared error (MSE)
sum((reg.predict(Xp) - y)**2) / len(y) # 18.984768907617223 ( from 23.943662938603104)

### plot

hp = cars.iloc[:,3].values
mpg = cars.iloc[:,0].values

hps = np.array(sorted(hp))
hps = hps.reshape(hps.size, 1)
hpsp = poly.fit_transform(hps)

plt.scatter(hp, mpg, color='blue', marker='x')
plt.plot(hps, reg.predict(hpsp), color='red', lw=2)
plt.xlabel('Power [hp]')
plt.ylabel('Fuel consumption [miles/gallon]')
plt.show()
```

Degree of regression
1: linear
2: quadratic
...

Why Gaussian Process Regression?

- There are **many projections possible**.
- We have to choose one either a priori or by model comparison with a set of possible projections.
- Especially if the problem is to **explore and exploit a completely unknown function**, this approach will not be beneficial as there is little guidance to which projections we should try.

Gaussian process regression offers a principled solution to this problem in which projections are chosen implicitly, effectively leading “**the data decide**” on the complexity of the function.

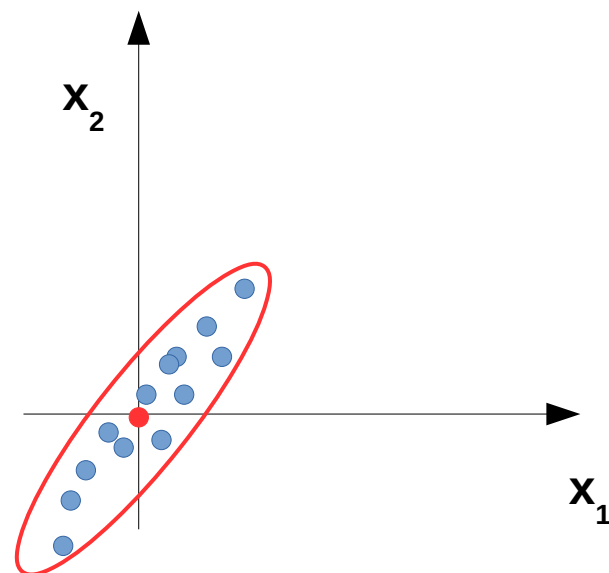
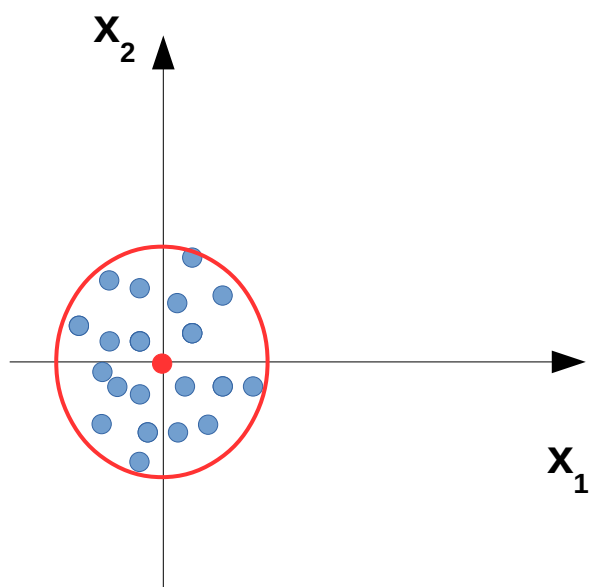
Recall Multivariate Gaussians

Say you measure two variables, e.g.,

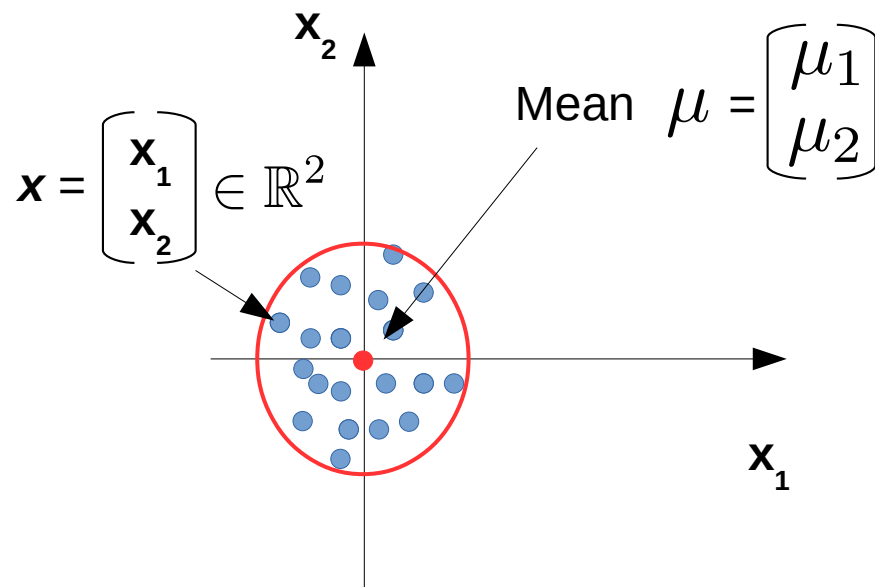
- x_1 : height
- x_2 : weight

→ plot

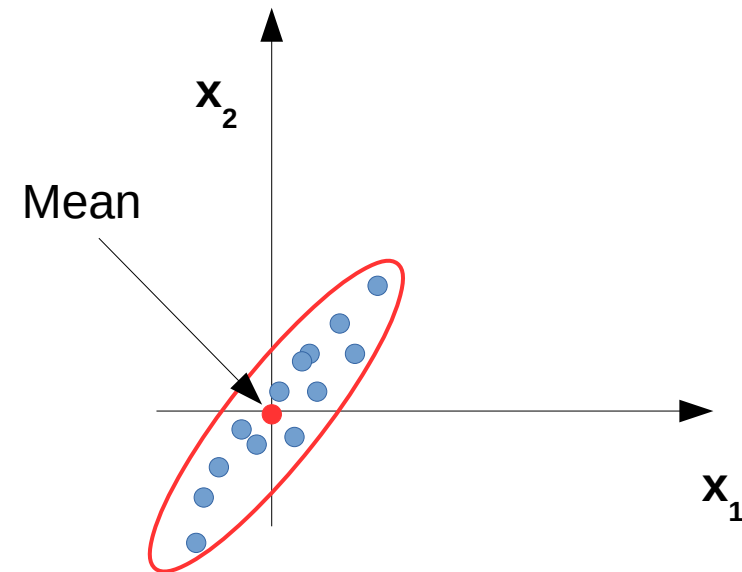
→ we want to fit a Gaussian to these points.



Recall Multivariate Gaussians (II)



Fit a Gaussian that with a covariance that is **circular**.



Fit a Gaussian that with a covariance that is an **ellipse**.

Multivariate Gaussians (III)

- Assume the points are Gaussian distributed (this is our “model”).
- **How do points relate to each other?** (“how does increasing x_1 increase x_2 ?”)
→ The variable to describe this is called “Covariance*” (cov)

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim N \left[\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right]$$

Mean
Covariance

- If the entries in the column vector $\mathbf{X} = (X_1, X_2, \dots, X_n)^T$ are **random variables**, each with **finite variance and expected value**, then the covariance matrix $\mathbf{K}_{\mathbf{xx}}$ is the matrix whose (i, j) entry is the covariance.

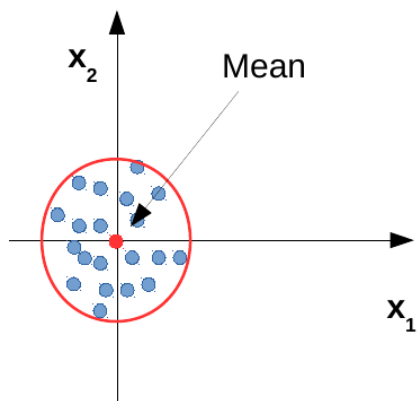
$$\mathbf{K}_{X_i X_j} = \text{cov}[X_i, X_j] = E[(X_i - E[X_i])(X_j - E[X_j])]$$

Multivariate Gaussian (IV)

- Assume for a moment that $\mathbf{E}[\cdot] = 0 \rightarrow$ Covariance $\mathbf{E}[\mathbf{x}_1 \mathbf{x}_2]$ is a “dot” product.
- Assume two points $[1 \ 0] \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1 \rightarrow$ **Covariance is a measure similarity.**

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim N \left[\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right]$$

Knowing about \mathbf{x}_1 does not provide any information about \mathbf{x}_2 as they are uncorrelated.



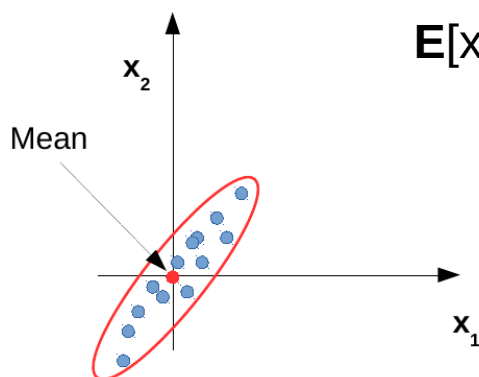
$$\mathbf{E}[x_1 x_2] = 0$$

Multivariate Gaussians (V)

- Assume for a moment that $\mathbf{E}[\cdot] = 0 \rightarrow$ Covariance $\mathbf{E}[\mathbf{x}_1 \mathbf{x}_2]$ is a “dot” product.
- Assume two points $[1 \ 0] \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1 \rightarrow$ Covariance measure similarity.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim N \left[\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} 1 & 0.6 \\ 0.6 & 1 \end{bmatrix} \right]$$

$$\mathbf{E}[x_1 x_2] \neq 0$$



\rightarrow Knowing about \mathbf{x}_1 DOES provide information about \mathbf{x}_2 .

\rightarrow if \mathbf{x}_1 is positive, \mathbf{x}_2 is with great probability.

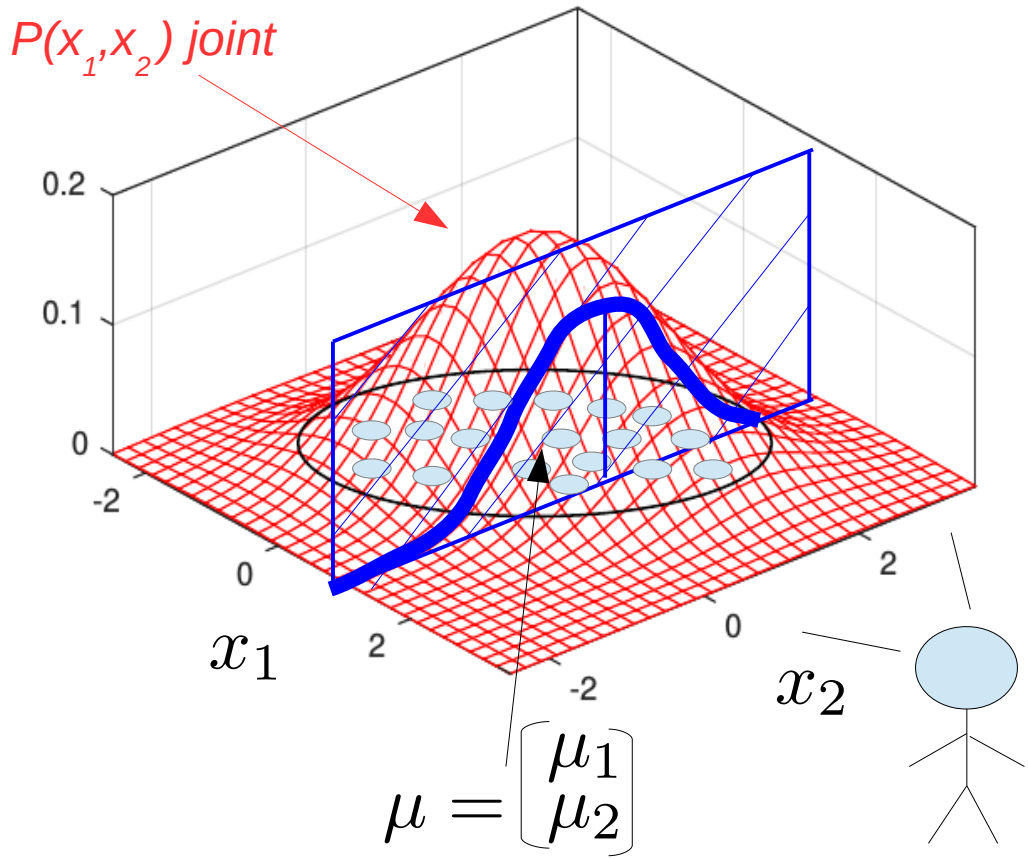
\rightarrow knowing something about \mathbf{x}_1 allows us to know something about \mathbf{x}_2 .

Joint Gaussian distributions

see, e.g., Rasmussen et al. (2005), Murphy (2012)

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim N \left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right)$$

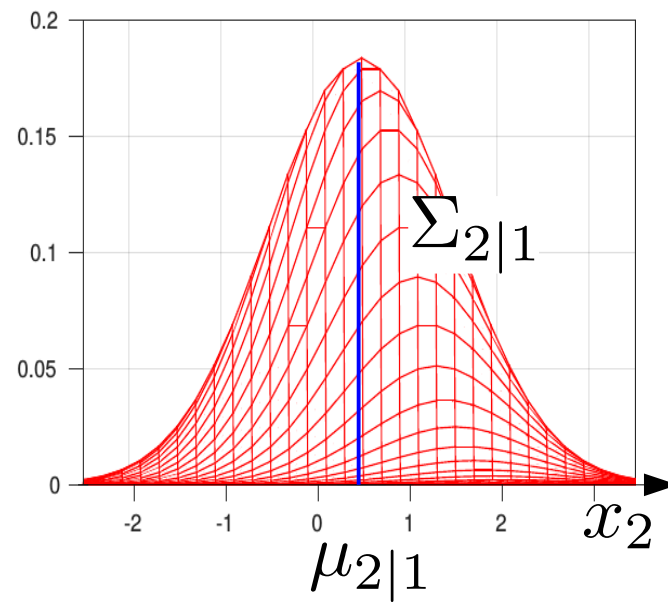
Mean Covariance



Conditional distribution

↓

$$P(x_2 | X_1 = x_1)$$



From Joint to Conditional distributions

see, e.g., Murphy (2012), chapter 4.

Theorem 4.3.1 (Marginals and conditionals of an MVN). Suppose $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2)$ is jointly Gaussian with parameters

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{pmatrix}, \quad \boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1} = \begin{pmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{pmatrix} \quad (4.67)$$

Then the marginals are given by

$$\begin{aligned} p(\mathbf{x}_1) &= \mathcal{N}(\mathbf{x}_1 | \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{11}) \\ p(\mathbf{x}_2) &= \mathcal{N}(\mathbf{x}_2 | \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22}) \end{aligned} \quad (4.68)$$

and the posterior conditional is given by

$$\begin{aligned} p(\mathbf{x}_1 | \mathbf{x}_2) &= \mathcal{N}(\mathbf{x}_1 | \boldsymbol{\mu}_{1|2}, \boldsymbol{\Sigma}_{1|2}) \\ \boldsymbol{\mu}_{1|2} &= \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12} \boldsymbol{\Sigma}_{22}^{-1} (\mathbf{x}_2 - \boldsymbol{\mu}_2) \\ &= \boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{11}^{-1} \boldsymbol{\Lambda}_{12} (\mathbf{x}_2 - \boldsymbol{\mu}_2) \\ &= \boldsymbol{\Sigma}_{1|2} (\boldsymbol{\Lambda}_{11} \boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{12} (\mathbf{x}_2 - \boldsymbol{\mu}_2)) \\ \boldsymbol{\Sigma}_{1|2} &= \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12} \boldsymbol{\Sigma}_{22}^{-1} \boldsymbol{\Sigma}_{21} = \boldsymbol{\Lambda}_{11}^{-1} \end{aligned} \quad (4.69)$$

Two “blocks” of vectors

This Theorem allows you to go from joint to conditional distributions.

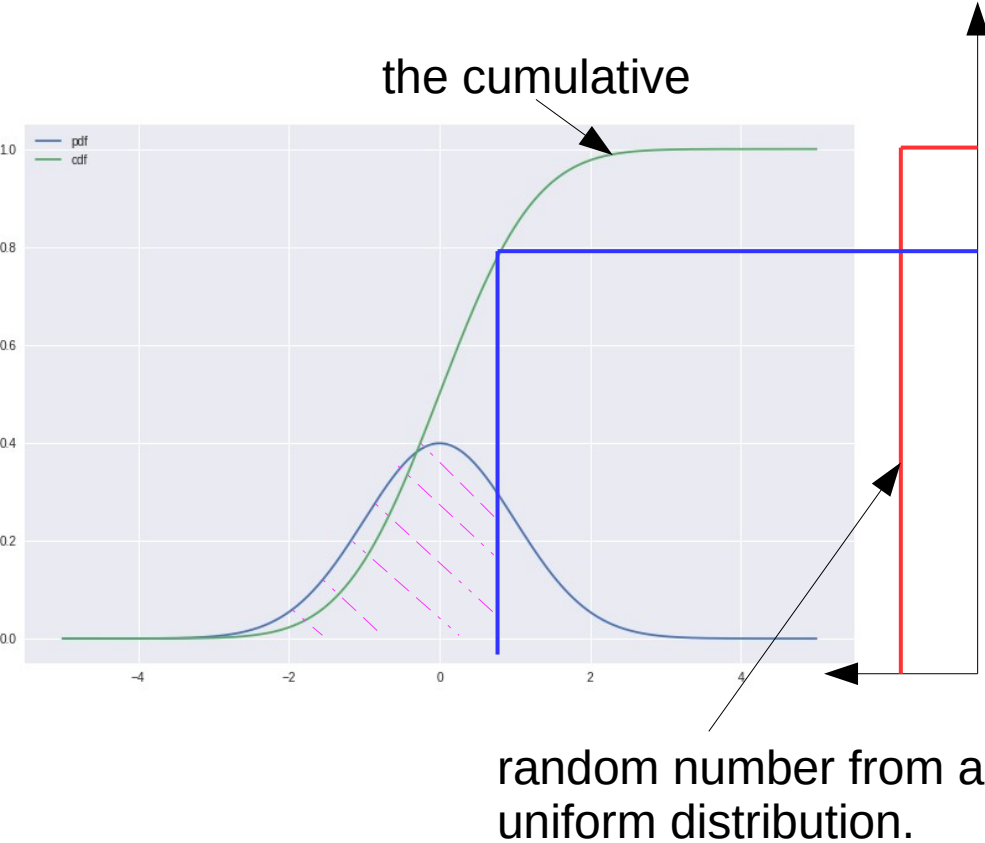
Producing Data from Gaussians

$$x_i \sim \mathcal{N}(0, 1)$$

$$x_i \sim \mathcal{N}(\mu, \sigma^2) \sim \mu + \sigma \mathcal{N}(0, 1)$$

As we have the capability of drawing 1-dim random numbers from a Gaussian, we can also do this in a multivariate case.

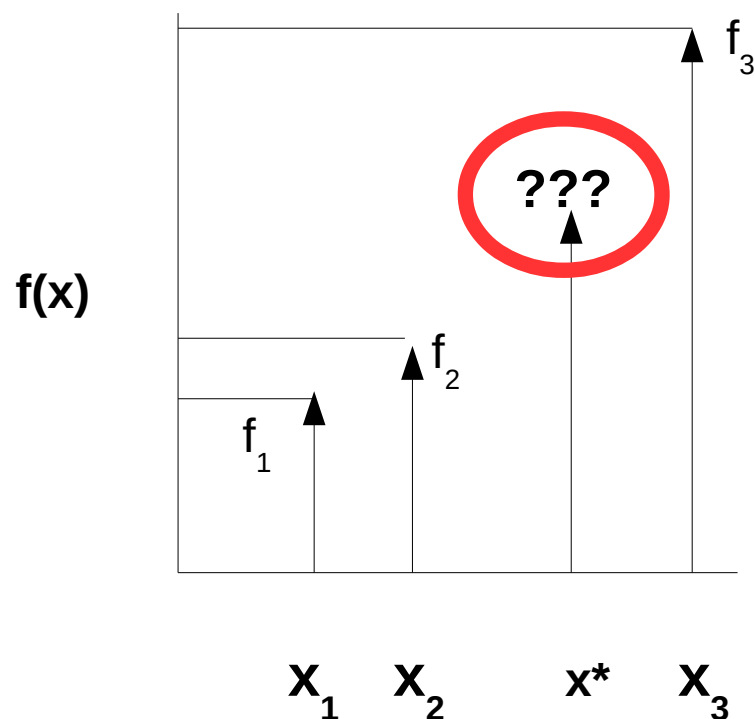
- We need a way to take “square roots” from matrices.
- **Cholesky** decomposition: $\Sigma = LL^T$



$$\begin{array}{c} \longrightarrow \begin{matrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim N \left[\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right] \\ \uparrow \quad \quad \uparrow \quad \quad \uparrow \\ \mathbf{x} \quad \quad \mu \quad \quad \Sigma \end{matrix} \longrightarrow x \sim \mu + L\mathcal{N}(0, I) \end{array}$$

Recall Eq. (4.68) from the previous slide,

Observations → Interpolation

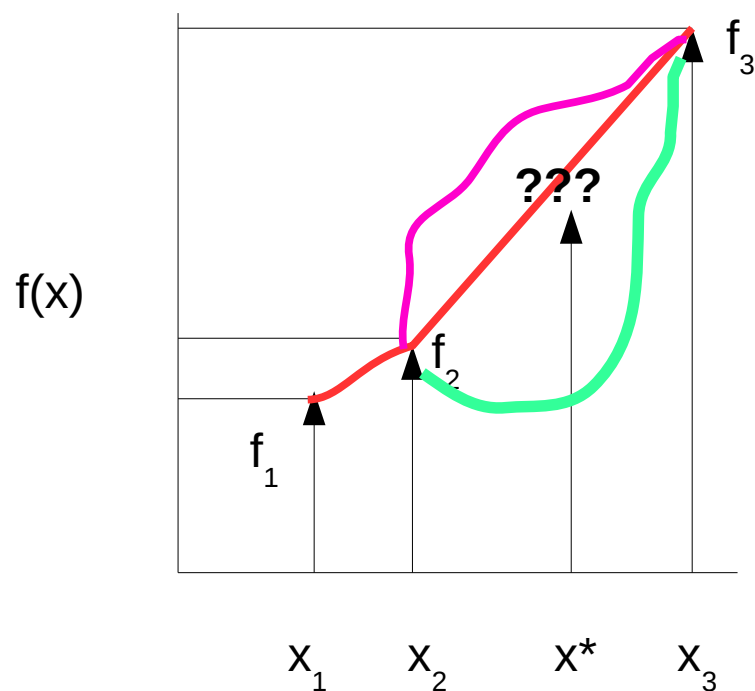


We have 3 observations at x_i for $f(x_i)$

- Given the data pairs
 $D = \{ (x_1, f_1), (x_2, f_2), (x_3, f_3) \}$
- **want to find/learn the function that describes the data, i.e.,**
for a “new” x^* , we want to know what $f(x^*)$ would be!

Observations \rightarrow Interpolation (II)

We assume that **f's (the height) are Gaussian distributed**, with **zero – mean** and some **covariance matrix K**.



$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \sim N \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \right)$$

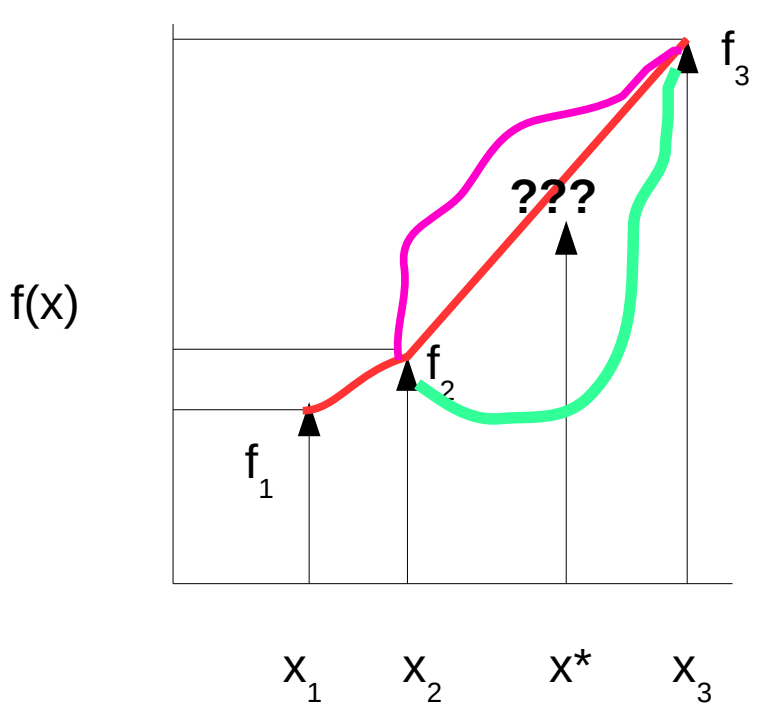
Note: f_1 and f_2 should probably be more correlated, as they are nearby (compared to f_1 and f_3).

\rightarrow The prior mean function μ reflects the expected function value at input x : $\mu(x) = \mathbb{E}(f(x))$

\rightarrow It is often set to 0.

Observations → Interpolation (II)

We assume that **f's (the height) are Gaussian distributed**, with **zero – mean** and some **covariance matrix K**.



$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \sim N \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \right)$$

Note: f_1 and f_2 should probably be more correlated, as they are nearby (compared to f_1 and f_3), e.g.,

$$\sim N \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & 0.7 & 0.2 \\ 0.7 & 1 & 0.6 \\ 0.2 & 0.6 & 1 \end{bmatrix} \right)$$

Covariance matrix **constructed** by some “**measure of similarity**”, i.e., a **kernel function** (parametric ansatz), such as “squared exponential”. Parameters can be obtained e.g. via MLE (later).

$$\kappa(x, x') = \sigma_f^2 \exp\left(-\frac{1}{2\ell^2}(x - x')^2\right)$$

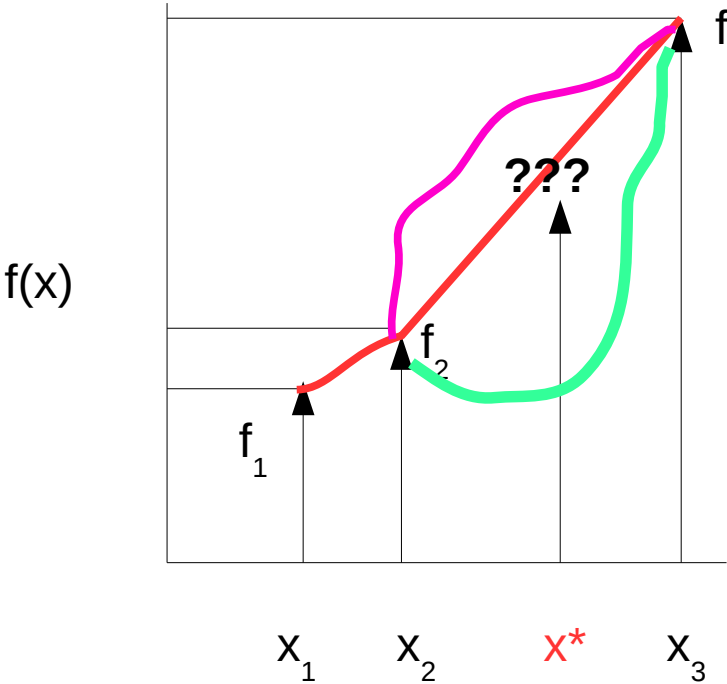
σ_f^2 – controls vertical variation.

ℓ – controls horizontal length scale.

Observations → Interpolation (III)

Given data $D = \{ (x_1, f_1), (x_2, f_2), (x_3, f_3) \} \rightarrow f(x^*) = f_* ?$
→ Assume $f \sim N(0, K(\cdot, \cdot))$
→ Assume $f(x^*) \sim N(0, K(x^*, x^*))$

3d-Covariance K from the training data



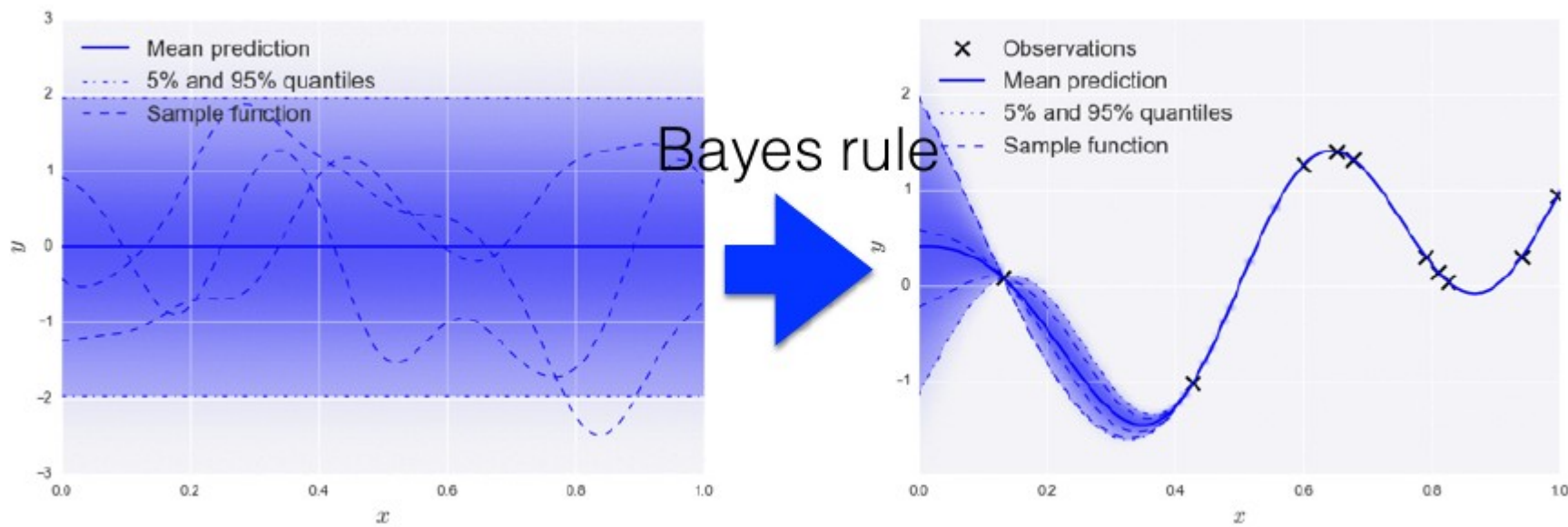
$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_* \end{pmatrix} \sim N \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} K_{11} & K_{12} & K_{13} & K_{1*} \\ K_{21} & K_{22} & K_{23} & K_{2*} \\ K_{31} & K_{32} & K_{33} & K_{3*} \\ K_{*1} & K_{*2} & K_{*3} & K_{**} \end{bmatrix} \right)$$

- Joint distribution over f and f_* .
- We need the conditional of f_* given f .
- In this example, we “cut” in 3 dimensions.
- What is left is a 1-dimensional Gaussian, i.e., the Gaussian for f_*

$$K(x_1, x_*) = K_{1*}$$

Interpolation → Noiseless GPR

(see, e.g., Rasmussen & Williams (2006), with references therein)



Prior GP

Posterior GP

Training set: $D = \{(\mathbf{x}_i, y_i) | i = 1, \dots, n\}$

$$\begin{pmatrix} \mathbf{f} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N} \left(\begin{pmatrix} \boldsymbol{\mu} \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{pmatrix} \right)$$

$$p(\mathbf{f}_* | \mathbf{X}_*, \mathbf{X}, \mathbf{f}) = \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$$

$$\boldsymbol{\mu}_* = \boldsymbol{\mu}(\mathbf{X}_*) + \mathbf{K}_*^T \mathbf{K}^{-1} (\mathbf{f} - \boldsymbol{\mu}(\mathbf{X}))$$
$$\boldsymbol{\Sigma}_* = \mathbf{K}_{**} - \mathbf{K}_*^T \mathbf{K}^{-1} \mathbf{K}_*$$

Test point = interpolation at \mathbf{X}^*

- **predictive mean** $\mu_* = \mathbb{E}(f_*)$
- **Confidence Intervals!**

Where we have data, we have high confidence in our predictions.
Where we do not have data, we cannot be to confident about our predictions.

Algorithmic Complexity!

- The central computational operation in using Gaussian processes will involve the **inversion of a matrix of size $N \times N$** , for which standard methods require **$O(N^3)$** computations.
- The matrix inversion must be performed **once for the given training set**.
 - For large training data sets, however, **the direct application of Gaussian process methods can become infeasible**.
 - Sparse Methods (cf. Rasmussen et. al (2006) and references therein).

A note on the predictive mean

Note that the predictive mean can (in general) also be written as

$$\mu = m(x) + \sum_{i=1}^N a_i k(x_i, x)$$

where $\mathbf{a} = (a_1, \dots, a_N) = (\mathbf{K} + s_n^2 \mathbf{I}_N)^{-1} (\mathbf{t} - \mathbf{m})$
and \mathbf{t} being the N observations.

→ We can think of the **GP posterior mean** as an **approximation of $f(\cdot)$** using **N symmetric basis functions** centered at each observed input.

→ by choosing a **covariance function that vanishes when x and x' are separated** by a lot, for example the squared exponential covariance function, we see that **an observed input-output will only affect the approximation locally**.

$$k_{\text{SE}}(\mathbf{x}, \mathbf{x}'; \boldsymbol{\theta}) = s^2 \exp \left\{ -\frac{1}{2} \sum_{i=1}^D \frac{(x_i - x'_i)^2}{\ell_i^2} \right\}$$

GP – a distribution over functions

see [Lecture_2/code/1d_gp_example.py](#)

$$f(x) \sim GP(\mu(x), k(x, x'))$$

$$\mu(x) = \mathbb{E}[f(x)]$$

$$k(x, x') = \mathbb{E}[(f(x) - \mu(x))(f(x') - \mu(x'))^T]$$

$$k(x, x') = \exp\left(-\frac{1}{2}(x - x')^2\right)$$

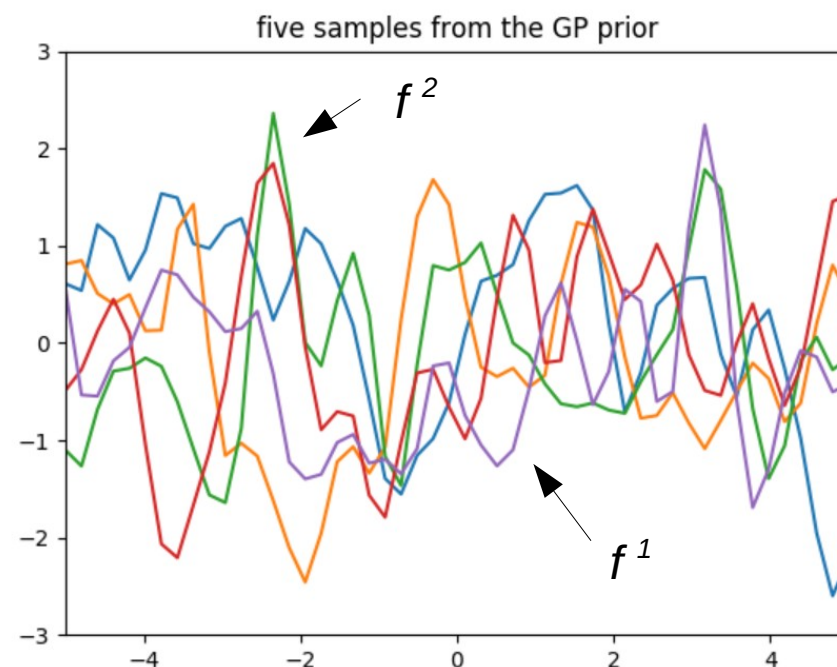
Procedure

Create vector $X_{1:N}$

$$\mu = 0, K_{N \times N}$$

$$K = LL^T$$

$$f^i \sim N(0, K) \sim L N(0, I)$$



GPR Example code & numerical stability

Look at [Lecture_2/code/1d_gp_example.py](#) and Murphy, Chapter 15

```
import numpy as np
import matplotlib.pyplot as plt

""" A code to illustrate the workings of GP regression in 1d.
    We assume a zero mean GP Prior """

# This is the true unknown, one-dimensional function we are trying to approximate
f = lambda x: np.sin(0.9*x).flatten()

# This is the kernel function
def kernel_function(a, b):
    """ GP squared exponential kernel function """
    kernelParameter = 0.1
    sqdist = np.sum(a**2,1).reshape(-1,1) + np.sum(b**2,1) - 2*np.dot(a, b.T)
    return np.exp(-.5 * (1/kernelParameter) * sqdist)

# Here we run the test
N = 10          # number of training points.
n = 50          # number of test points.
s = 0.00005     # noise variance.

# Sample some input points and noisy versions of the function evaluated at
# these points.
X = np.random.uniform(-5, 5, size=(N,1))
y = f(X) + s*np.random.randn(N) #add some noise

K = kernel_function(X, X)
L = np.linalg.cholesky(K + s*np.eye(N))

# points we're going to make predictions at.
TestPoint = np.linspace(-5, 5, n).reshape(-1,1)

# compute the mean at our test points.
Lk = np.linalg.solve(L, kernel_function(X, TestPoint))
mu = np.dot(Lk.T, np.linalg.solve(L, y))

# compute the variance at our test points.
K_ = kernel_function(TestPoint, TestPoint)
s2_ = np.diag(K_) - np.sum(Lk**2, axis=0)
s_ = np.sqrt(s2_)
```

$$\mathbb{E}[f(x_*)] = \mu = k_*^T K_y^{-1} y$$

$$K_y = LL^T$$

$$\alpha = K_y^{-1} y = L^{-T} L^{-1} y$$

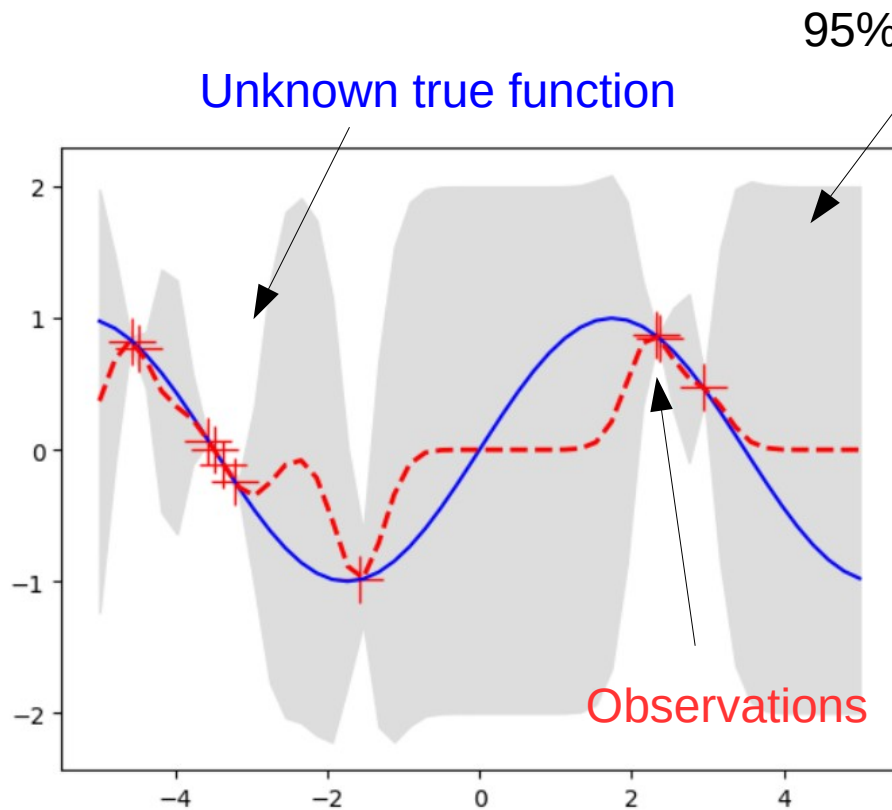
Algorithm 15.1: GP regression

- 1 $L = \text{cholesky}(K + \sigma_y^2 I);$
 - 2 $\alpha = L^T \setminus (L \setminus y);$
 - 3 $\mathbb{E}[f_*] = k_*^T \alpha;$
 - 4 $v = L \setminus k_*;$
 - 5 $\text{var}[f_*] = \kappa(x_*, x_*) - v^T v;$
 - 6 $\log p(y|X) = -\frac{1}{2} y^T \alpha - \sum_i \log L_{ii} - \frac{N}{2} \log(2\pi)$
-

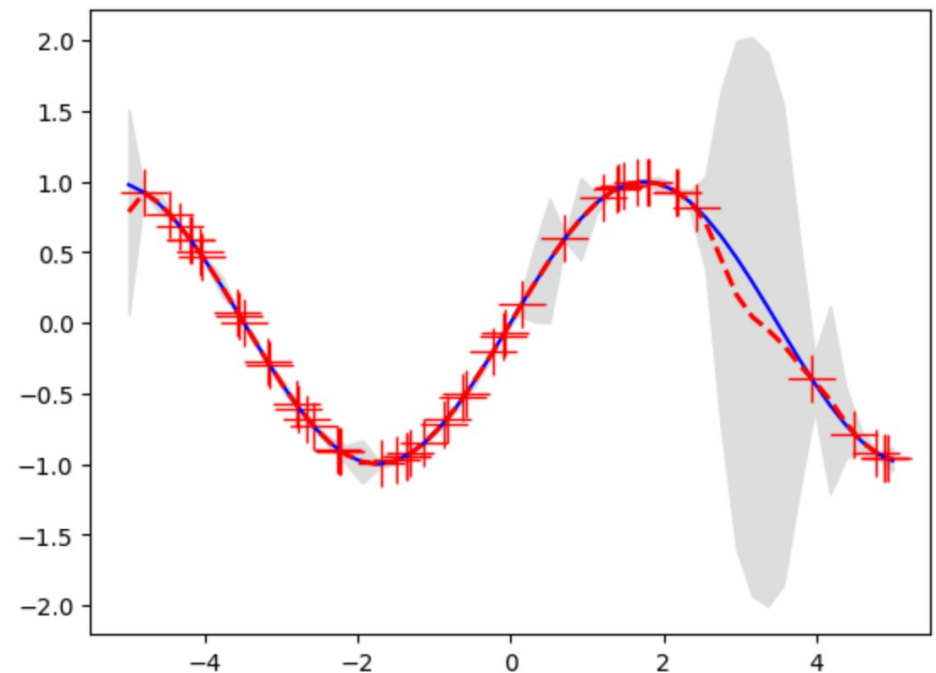
Alg. 15: Numerical more stable.

Prediction & Confidence Intervals

Note: if you don't fix the seed, these pictures vary every time you run of the code.



10 Training Points



50 Training Points

We see that the model perfectly interpolates the training data, and that the predictive uncertainty increases as we move further away from the observed data.

Pin-down a 1d VF by GPs

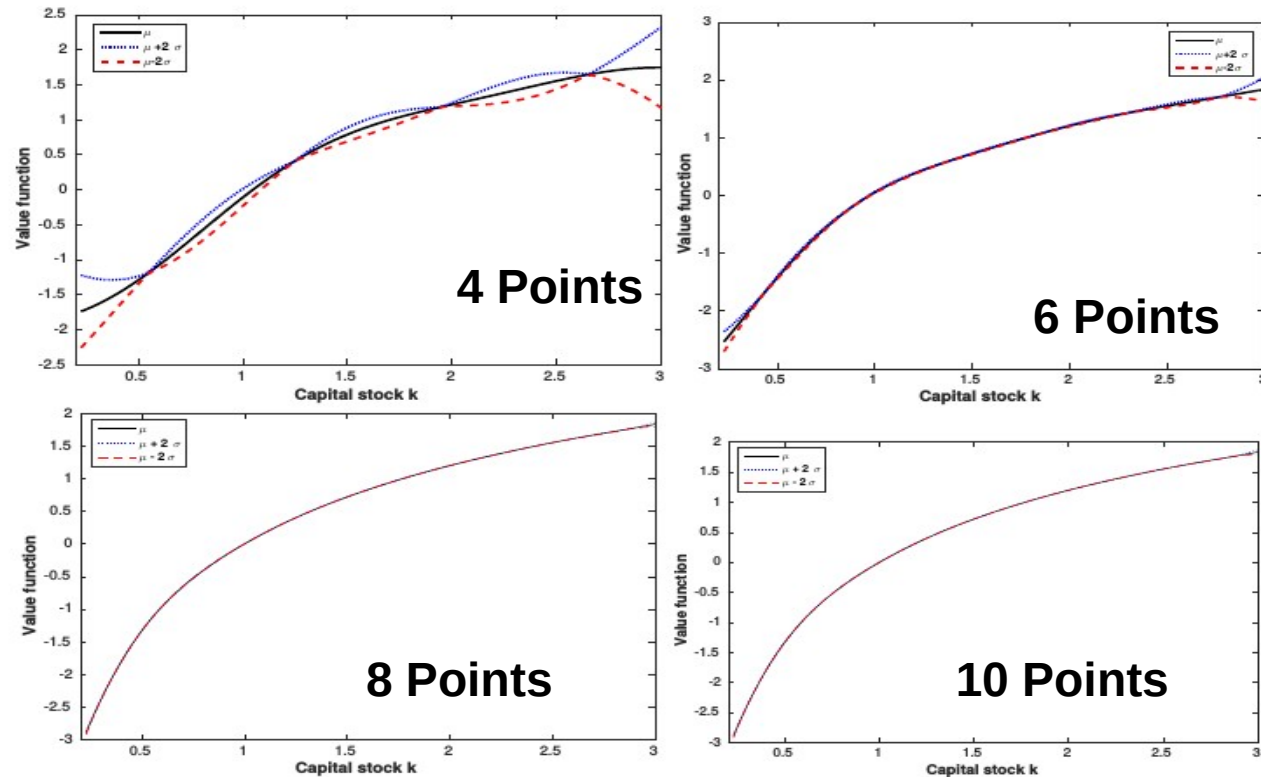


Figure 5: Above's four panels display the predictive mean value function from a 1-dimensional growth model at convergence, and the 95% confidence interval. The upper left figure was constructed based on only 4 sample points in the state space, the upper right on 6 sample points. The lower left was a result of 8 sample points, and the lower right is based on 10 sample points that pin down the function.

Non-hypercubic domain: GPR versus ASG

More on this in Lecture 6!

$$\text{Vol}_{\Delta} = \frac{1}{D!}$$

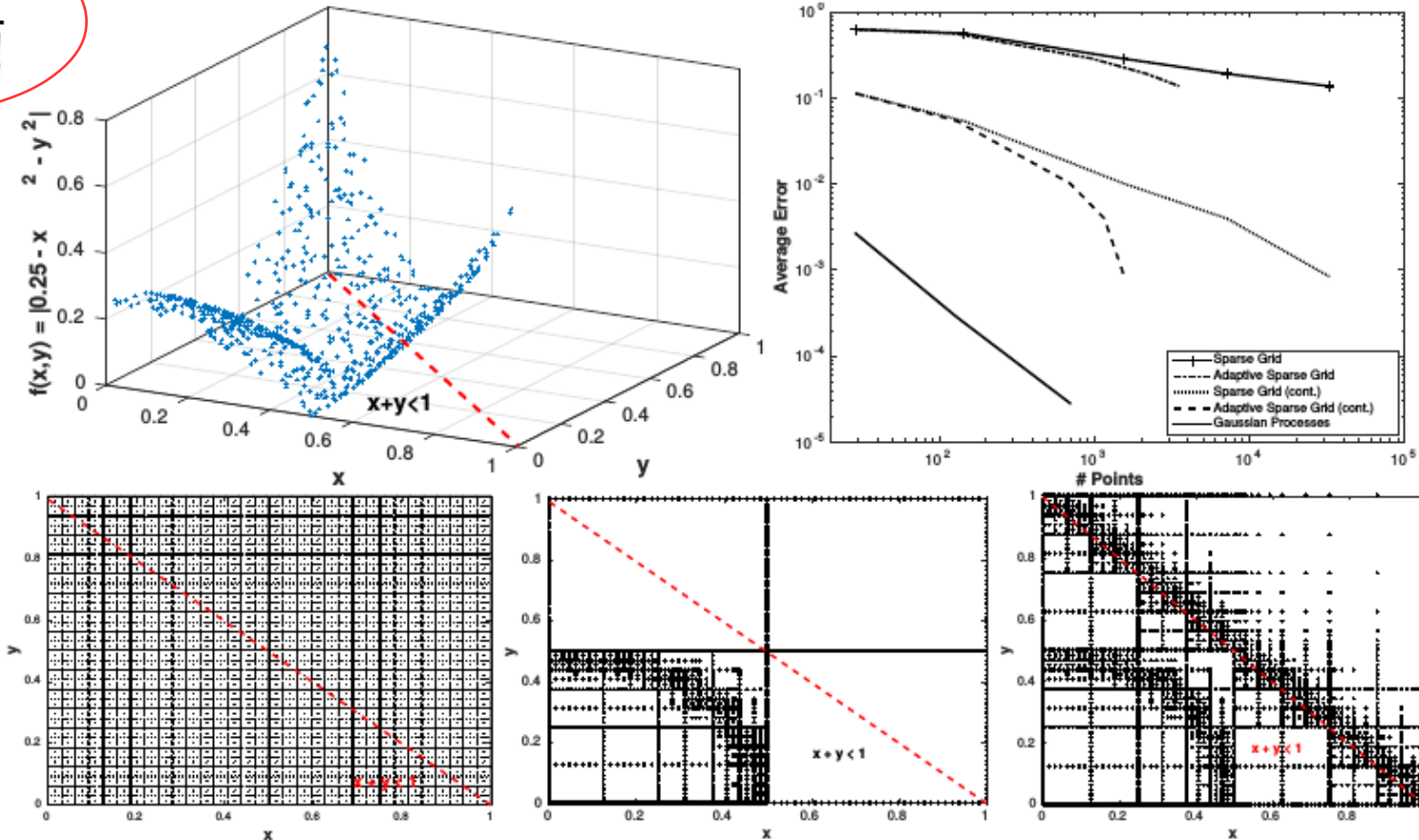


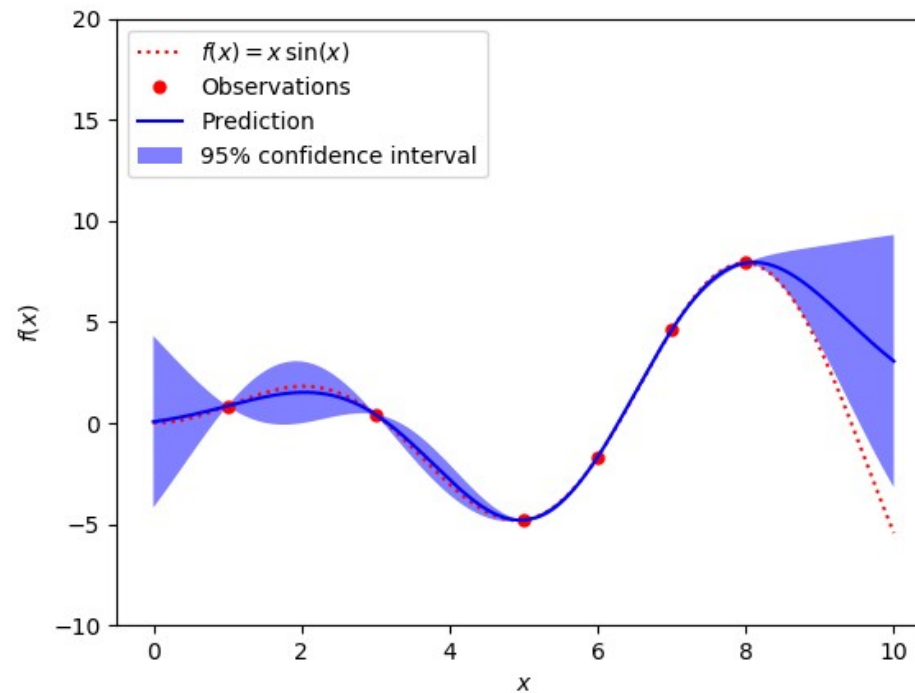
Figure: The upper left panel shows the analytical test function evaluated at random test points on the simplex. The upper right panel displays a comparison of the interpolation error for GPs, sparse grids, and adaptive sparse grids of varying resolution and constructed under the assumption that a continuation value exists, (denoted by "cont"), or that there is no continuation value. The lower left panel displays a sparse grid consisting of 32,769 points. The lower middle panel shows an adaptive sparse grid (cont) that consists of 1,563 points, whereas the lower right panel shows an adaptive sparse grid, constructed with 3,524 points and under the assumption that the function outside Δ is not known.

GPR in scikit-learn.org

https://scikit-learn.org/stable/modules/gaussian_process.html

https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_noisy_targets.html#sphx-glr-auto-examples-gaussian-process-plot-gpr-noisy-targets-py

Look at [Lecture_2/1d_GPR.py](#)



More GP packages (next to scikit-learn.org)

Name	Algorithm	Language	Author
GPML	GP Toolbox	Matlab	Rasmussen and Nickisch (2010)
SFO	Submodular Optimization	Matlab	Krause (2010)
GPy	GP Toolbox	Python	Sheffield ML group (since 2012)
tgp	Tree GPs, GP regression	R	Gramacy et al. (2007)

Get the code e.g. from here

<http://sheffieldml.github.io/GPy/>

\$ pip install GPy

GPR with noisy data

- In empirical setups, **measurement noise may arise** from our **inability to control all the influential factors** or from **irreducible (aleatory) uncertainties**.
- In **computer simulations**, measurement uncertainty may stem from quasi-random stochasticity, or chaotic behavior.
- Now let us consider the case where what we observe is a noisy version of the underlying function

$$y = f(x) + \epsilon, \text{ where } \epsilon \sim \mathcal{N}(0, \sigma_y^2)$$

GPR with noisy data (II)

- In this case (presence of noise), the model is not required to interpolate the data, **but it must come “close”** to the observed data.
- The covariance of the observed noisy responses is

$$\text{cov}[y_p, y_q] = \kappa(\mathbf{x}_p, \mathbf{x}_q) + \sigma_y^2 \delta_{pq}$$

where $\delta_{pq} = \mathbb{I}(p = q)$

- The second matrix is **diagonal** because we assumed the **noise terms were independently added to each observation**.

The GPR with noisy data (III)

- The joint density of the observed data and the latent, noise-free function on the test points is given by

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N} \left(\mathbf{0}, \begin{pmatrix} \mathbf{K}_y & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{pmatrix} \right)$$

Latent function. Noise in the diagonal, rest is as before.

- where we are assuming the mean is zero, for notational simplicity.
- Hence the posterior predictive density is

$$\begin{aligned} p(\mathbf{f}_* | \mathbf{X}_*, \mathbf{X}, \mathbf{y}) &= \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) \\ \boldsymbol{\mu}_* &= \mathbf{K}_*^T \mathbf{K}_y^{-1} \mathbf{y} \\ \boldsymbol{\Sigma}_* &= \mathbf{K}_{**} - \mathbf{K}_*^T \mathbf{K}_y^{-1} \mathbf{K}_* \end{aligned}$$

Prediction at a single test point

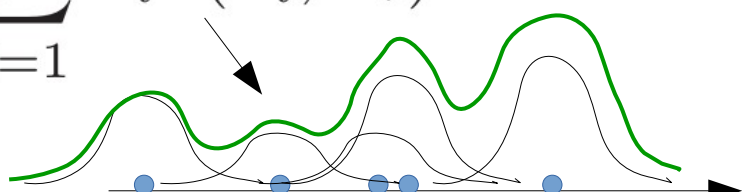
In the case of a single test input, this simplifies as follows

$$p(f_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}) = \mathcal{N}(f_* | \mathbf{k}_*^T \mathbf{K}_y^{-1} \mathbf{y}, k_{**} - \mathbf{k}_*^T \mathbf{K}_y^{-1} \mathbf{k}_*)$$

where $\mathbf{k}_* = [\kappa(\mathbf{x}_*, \mathbf{x}_1), \dots, \kappa(\mathbf{x}_*, \mathbf{x}_N)]$

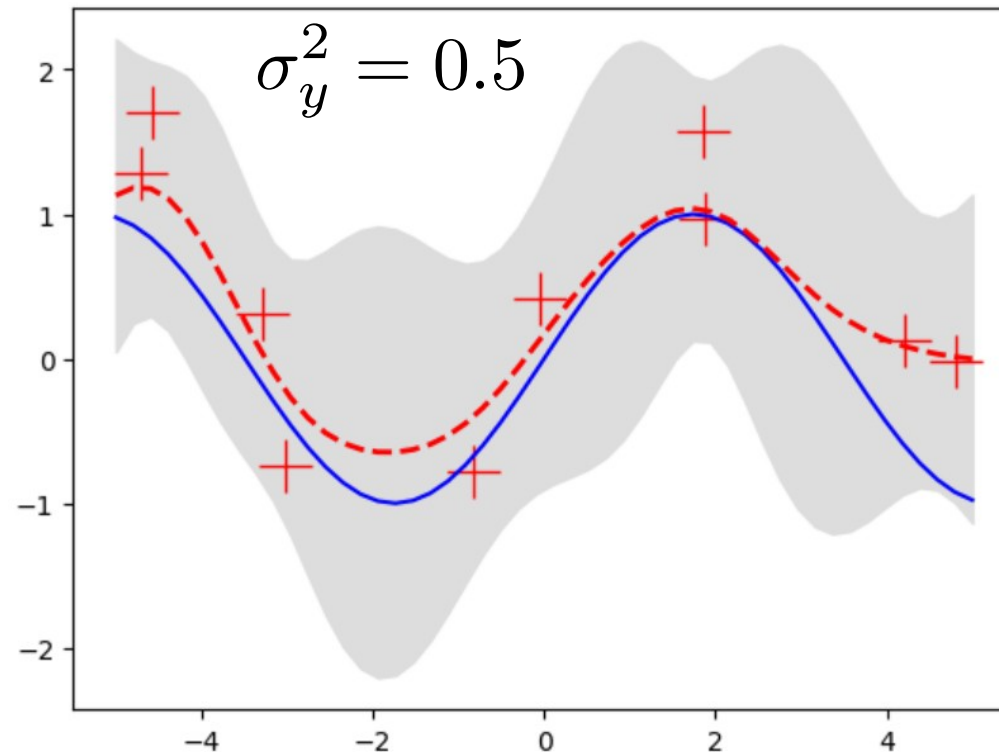
and where $k_{**} = \kappa(\mathbf{x}_*, \mathbf{x}_*)$ (=1)

Again, we can write the **posterior mean** as **expansion of basis functions**

$$\bar{f}_* = \overset{1 \times N}{\mathbf{k}_*^T} \overset{N \times N}{\mathbf{K}_y^{-1}} \overset{N \times 1}{\mathbf{y}} = \sum_{i=1}^N \alpha_i \kappa(\mathbf{x}_i, \mathbf{x}_*) \quad \text{where } \alpha = \underbrace{\mathbf{K}_y^{-1} \mathbf{y}}_{\text{from training data}}$$


Some Plots

cf. Lecture_2/code/1d_gp_example.py



- Even in the regions where you have data, there is still uncertainty.
- In the noise-free version of GPR, the uncertainty is 0 at observation points.
- But we still have the same properties as before: where we have data, we are more certain compared to the case where we have no data.

Noise improves numerical stability

- It is common to use small noise even if there is not any in the data.
- Cholesky fails when covariance is close to being semi-positive definite.
- Adding a small noise improves numerical stability.
- It is known as the “jitter” or as the “nugget” in this case.

Illustration of noiseless GPR prediction (II)

- We use a squared exponential kernel, aka **Gaussian kernel or RBF kernel**.
- In $1d$, this is given by
$$\kappa(x, x') = \sigma_f^2 \exp\left(-\frac{1}{2\ell^2}(x - x')^2\right)$$
- Here **ℓ controls the horizontal length scale** over which the function varies, and **σ_f controls the vertical variation**.
- On the right panel, we showed predictions from the posterior, $p(f_* | X_*, X, f)$.
- **We see that the model perfectly interpolates the training data, and that the predictive uncertainty increases as we move further away from the observed data.**

The parameters in the Kernel

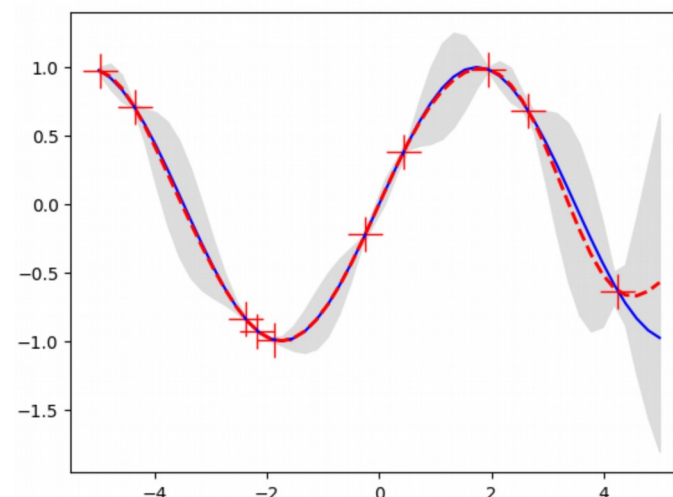
cf. Lecture_2/code/1d_gp_example.py

$$\kappa(x, x') = \sigma_f^2 \exp\left(-\frac{1}{2\ell^2}(x - x')^2\right)$$

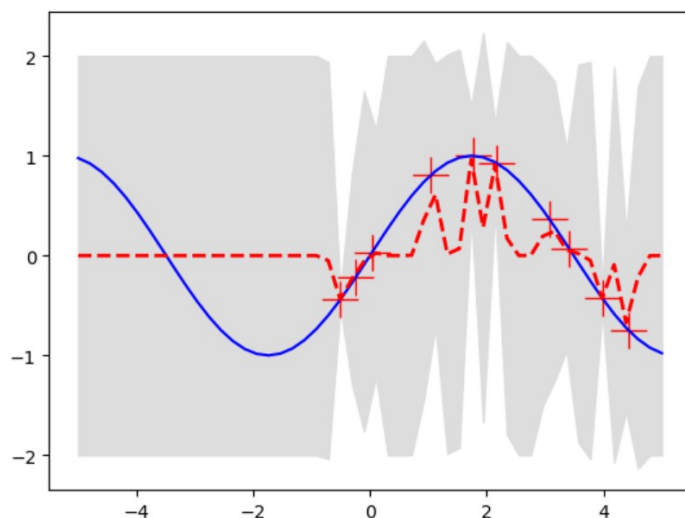
Let $\sigma_f^2 = 1$

→ **Tuning the parameters by hand is not a good idea in general (in particular in high-dimensional settings).**

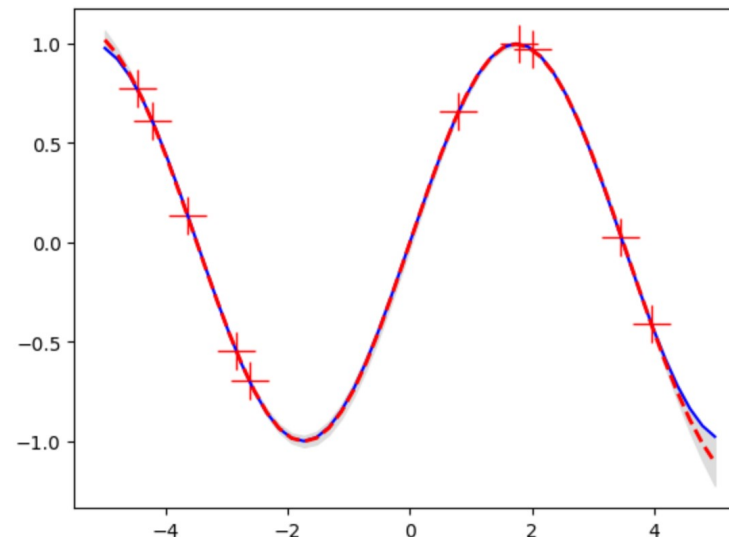
$\ell^2 = 1.0$



$\ell^2 = 0.01$



$\ell^2 = 10.0$



“Learning” the kernel parameters

- ♦ To **estimate the kernel parameters**, we could use **exhaustive search over a discrete grid of values**, with validation loss as an objective, but this can be quite slow.
- ♦ Here we consider an empirical Bayes approach, which will allow us to **use continuous optimization methods**, which are much faster.
- ♦ In particular, we will **maximize the marginal likelihood**.

“Learning” the kernel parameters (II)

Marginal likelihood $p(\mathbf{y}|\mathbf{X}) = \int p(\mathbf{y}|\mathbf{f}, \mathbf{X})p(\mathbf{f}|\mathbf{X})d\mathbf{f}$

Since $p(\mathbf{f}|\mathbf{X}) = \mathcal{N}(\mathbf{f}|\mathbf{0}, \mathbf{K})$

and $p(\mathbf{y}|\mathbf{f}) = \prod_i \mathcal{N}(y_i|f_i, \sigma_y^2)$

the (log-) marginal likelihood is given by

$$\log p(\mathbf{y}|\mathbf{X}) = \log \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}_y) = -\frac{1}{2}\mathbf{y}\mathbf{K}_y^{-1}\mathbf{y} - \frac{1}{2}\log |\mathbf{K}_y| - \frac{N}{2}\log(2\pi)$$

1st term: data fit term

2nd term: a model complexity term

3rd term: a constant.

Maximizing the the marginal likelihood

- Let the kernel parameters (also called **hyper-parameters**) be denoted by **θ**
- One can show that the following holds.

$$\begin{aligned}\frac{\partial}{\partial \theta_j} \log p(\mathbf{y}|\mathbf{X}) &= \frac{1}{2} \mathbf{y}^T \mathbf{K}_y^{-1} \frac{\partial \mathbf{K}_y}{\partial \theta_j} \mathbf{K}_y^{-1} \mathbf{y} - \frac{1}{2} \text{tr}(\mathbf{K}_y^{-1} \frac{\partial \mathbf{K}_y}{\partial \theta_j}) \\ &= \frac{1}{2} \text{tr} \left((\boldsymbol{\alpha} \boldsymbol{\alpha}^T - \mathbf{K}_y^{-1}) \frac{\partial \mathbf{K}_y}{\partial \theta_j} \right)\end{aligned}$$

where $\boldsymbol{\alpha} = \mathbf{K}_y^{-1} \mathbf{y}$

Computational complexity:

- It takes $O(N^3)$ time to compute \mathbf{K}_y^{-1}
- $O(N^2)$ time per hyper-parameter to compute the gradient.

Careful: Different optima correspond to different interpretations/beliefs

Fig. 5.5 of (Rasmussen and Williams 2006).

- We use the SE kernel

$$\kappa_y(x_p, x_q) = \sigma_f^2 \exp\left(-\frac{1}{2\ell^2}(x_p - x_q)^2\right) + \sigma_y^2 \delta_{pq}$$

- with $\sigma_f^2 = 1$

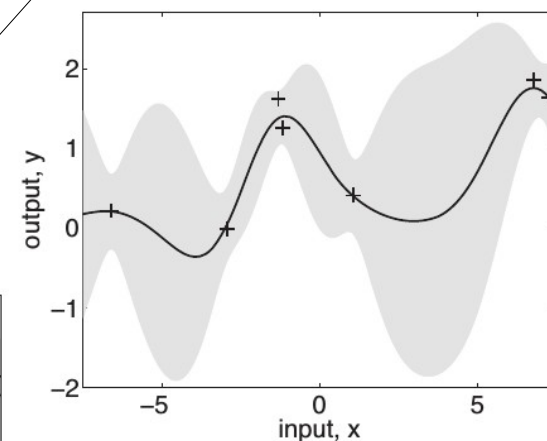
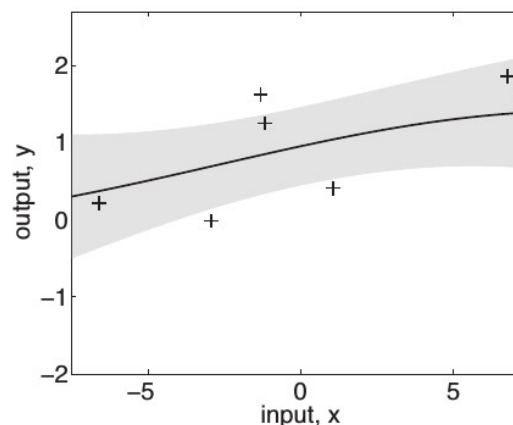
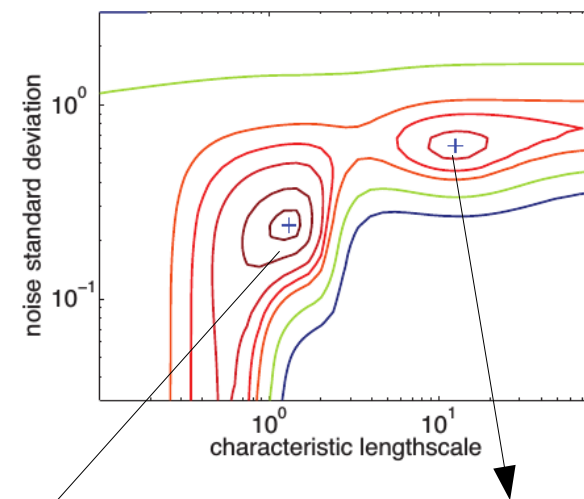
- We plot $\log p(\mathbf{y}|\mathbf{X}, \ell, \sigma_y^2)$ (where \mathbf{X} and \mathbf{y} are the 7 data points shown) as we vary ℓ and σ_y^2 .

- The two local optima are indicated by +.

- The bottom left optimum corresponds to a **low-noise, short-length scale solution**.

- The top right optimum corresponds to a **high-noise, long-length scale solution**.

- With only 7 data points, there is not enough evidence to confidently decide which is more reasonable.



An example: noise and optimization

https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_noisy_targets.html

Lecture_2/code/GPR_scikit_noise.py

```
import numpy as np
from matplotlib import pyplot as plt

from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C

np.random.seed(1)

def f(x):
    """The function to predict."""
    return x * np.cos(x)*np.sin(x)

# -----
# Mesh the input space for evaluations of the real function, the prediction and
# its MSE
x = np.atleast_2d(np.linspace(0, 10, 1000)).T

# Instantiate a Gaussian Process model
kernel = C(1.0, (1e-3, 1e3)) * RBF(10, (1e-2, 1e2))

# now the noisy case
X = np.linspace(0.1, 9.9, 20)
X = np.atleast_2d(X).T

# Observations and noise
y = f(X).ravel()
dy = 0.5 + 1.0 * np.random.random(y.shape)
noise = np.random.normal(0, dy)
y += noise

# Instantiate a Gaussian Process model
gp = GaussianProcessRegressor(kernel=kernel, alpha=dy ** 2,
                              n_restarts_optimizer=10)

# Fit to data using Maximum Likelihood Estimation of the parameters
gp.fit(X, y)

# Make the prediction on the meshed x-axis (ask for MSE as well)
y_pred, sigma = gp.predict(x, return_std=True)

# Plot the function, the prediction and the 95% confidence interval based on
# the MSE
plt.figure()
plt.plot(x, f(x), 'r:', label=u'$f(x) = x \cdot \sin(x)$')
plt.errorbar(X.ravel(), y, dy, fmt='r.', markersize=10, label=u'Observations')
plt.plot(x, y_pred, 'b-', label=u'Prediction')
plt.fill(np.concatenate([x, x[::1]]),
         np.concatenate([y_pred - 1.9600 * sigma,
                         (y_pred + 1.9600 * sigma)[::-1]]),
         alpha=.5, fc='b', ec='None', label='95% confidence interval')
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.ylim(-10, 20)
plt.legend(loc='upper left')

plt.show()
```

A multi-d example

Lecture_2/code/scikit_multi-d.py

```
import numpy as np
from matplotlib import pyplot as plt
import cPickle as pickle
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C

np.random.seed(1)

# Test function
def f(x):
    """The 2d function to predict."""
    return np.sin(x[0]) * np.cos(x[1])

# generate training data
n_sample = 100 #points
dim = 2 #dimensions

X = np.random.uniform(-1., 1., (n_sample, dim))
y = np.sin(X[:, 0:1]) * np.cos(X[:, 1:2]) + np.random.randn(n_sample, 1) * 0.005

# Instantiate a Gaussian Process model
kernel = C(1.0, (1e-3, 1e3)) * RBF(10, (1e-2, 1e2))
gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)

# Fit to data using Maximum Likelihood Estimation of the parameters
gp.fit(X, y)

# Make the prediction on the meshed x-axis / training points
y_pred, sigma = gp.predict(X, return_std=True)

#Compute MSE
mse = 0.0
n_sample_test=50
Xtest1 = np.random.uniform(-1., 1., (n_sample_test, dim))
y_pred1, sigma = gp.predict(Xtest1, return_std=True)
for g in range(len(Xtest1)):
    delta = abs(y_pred1[g] - f(Xtest1[g]))
    mse += delta

mse = mse/len(y_pred)
print(".....")
print(" The MSE is ", mse[0])
print(".....")
```


A multi-d example (II)

Lecture_2/code/scikit_multi-d.py

```
#-----  
# Important -- save the model to a file  
with open('2d_model.pkl', 'wb') as fd:  
    pickle.dump(gp, fd, protocol=pickle.HIGHEST_PROTOCOL)  
    print("data written to disk")  
  
# Load the model and do predictions  
with open('2d_model.pkl', 'rb') as fd:  
    gm = pickle.load(fd)  
    print("data loaded from disk")  
  
# generate training data  
n_test = 50  
dim = 2  
Xtest = np.random.uniform(-1., 1., (n_test, dim))  
y_pred_test, sigma_test = gm.predict(Xtest, return_std=True)  
  
MSE2 = 0  
for a in range(len(Xtest)):  
    delta = abs(y_pred_test[a] - f(Xtest[a]))  
    MSE2 += delta  
  
MSE2 = MSE2/len(Xtest)  
print(".....")  
print(" The MSE 2 is ", MSE2[0])  
print(".....")  
#-----
```

More on Kernels

<http://www.gaussianprocess.org/gpml/chapters/RW4.pdf>

See also e.g. "The Kernel Cookbook": <https://www.cs.toronto.edu/~duvenaud/cookbook/>

https://scikit-learn.org/stable/modules/gaussian_process.html#kernels-for-gaussian-processes

- Our **prior beliefs** about the response are **encoded** in our choice of the mean and covariance functions.
- The choice of an appropriate kernel is **based on assumptions** such as **smoothness** and **likely patterns** to be expected in the data.
- A sensible assumption is usually that **the correlation between two points decays with distance** between the points according to some **power function**.
- The choice of kernel determines almost all the generalization properties of a GP model.
- **You are the expert on your modeling problem - so you're the person best qualified to choose the kernel!**

Stationary versus non-stationary Kernels

Two categories of kernels can be distinguished:

- ♦ **Stationary kernels:**

They depend only on the **distance** of two data points and **not on their absolute values**

$k(x_i, x_j) = k(d(x_i, x_j))$ and are thus invariant to translations in the input space

Stationary kernels can further be subdivided into **isotropic** and **anisotropic** kernels, where isotropic kernels are also invariant to rotations in the input space.

- ♦ **Non-stationary kernels:**

They depend also on the **specific values of the data points**.

Squared Exponential Kernel

The SE kernel has become the **de-facto default kernel** for GPs.

$$k_{\text{SE}}(x, x') = \sigma^2 \exp \left(-\frac{(x - x')^2}{2\ell^2} \right)$$

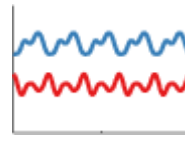
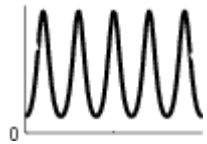
This is probably because it has some nice properties:

- It is **universal**, and **you can integrate it against** most functions that you need to.
- Every function in its prior has **infinitely many derivatives**.
- It also has only two parameters:
 - The **lengthscale ℓ** determines the length of the 'wiggles' in your function. In general, **you won't be able to extrapolate more than ℓ units away from your data**.
 - The output **variance σ^2** determines the average distance of your function away from **its mean**. Every kernel has this parameter out in front; it's just a scale factor.

Pitfalls for the SE kernel

- Most people who set up a GP regression (or classification) model end up using the SE kernel.
- They are a **quick-and-dirty solution** that will probably **work pretty well for interpolating smooth functions when N is a multiple of D** , and when there are **no 'kinks'** in your function.
- If your function happens to have a **discontinuity** or is **discontinuous in its first few derivatives** (for example, the `abs()` function), then either **your length-scale will end up being extremely short**, and your posterior mean will become zero almost everywhere, or your posterior mean will have 'ringing' effects.
- Even if there are no hard discontinuities, the **length-scale will usually end up being determined by the smallest 'wiggle' in your function** - so you might end up **failing to extrapolate in smooth regions** if there is even a **small non-smooth** region in your data.
- **If your data is more than two-dimensional, it may be hard to detect this problem.** One indication is if the length-scale chosen by maximum marginal likelihood never stops becoming smaller as you add more data. This is a classic sign of **model misspecification**.

Periodic Kernel



$$k_{\text{Per}}(x, x') = \sigma^2 \exp \left(-\frac{2 \sin^2(\pi |x - x'|/p)}{\ell^2} \right)$$

- The periodic kernel allows one to model functions which repeat themselves exactly.
- Its parameters are easily interpretable:
 - The period **p** simply determines the **distance between repetitions** of the function.
 - The **length-scale ℓ** determines the length-scale function in the same way as in the SE kernel.

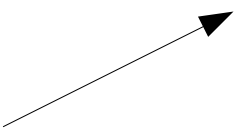
Matérn kernel

- The **Matérn kernel** is a stationary kernel and a **generalization of the RBF kernel**.
- It has an **additional parameter ν which controls the smoothness** of the resulting function. It is parameterized by a **length-scale parameter $\ell > 0$** , which can either be a scalar (isotropic variant of the kernel) or a vector with the same number of dimensions as the inputs (anisotropic variant of the kernel).
- The kernel is given by:

$$k(x_i, x_j) = \sigma^2 \frac{1}{\Gamma(\nu) 2^{\nu-1}} \left(\gamma \sqrt{2\nu} d(x_i/l, x_j/l) \right)^\nu K_\nu \left(\gamma \sqrt{2\nu} d(x_i/l, x_j/l) \right)$$

Matérn kernel (II)

- As $\nu \rightarrow 0$, the Matérn kernel converges to the RBF kernel.
- When $\nu = 1/2$, the Matérn kernel becomes identical to the absolute exponential kernel.

$$k_{\text{mat}}(x, x') = \sigma^2 \left(1 + \sqrt{3} \sum_{i=1}^l \frac{(x_i - x'_i)^2}{\ell_i^2} \right) \exp \left(-\sqrt{3} \sum_{i=1}^l \frac{(x_i - x'_i)^2}{\ell_i^2} \right)$$


- These are popular choices for learning functions that are not infinitely differentiable (as assumed by the RBF kernel) **but at least once ($\nu = 3/2$)** (if **$\nu = 5/2$, the kernel is twice differentiable**).

Combining/Adding Kernels

- Roughly speaking, adding two kernels can be thought of as an **OR** operation.
 → If you add together two kernels, then the resulting kernel will have high value if either of the two base kernels have a high value.

- **Linear plus Periodic Kernel**

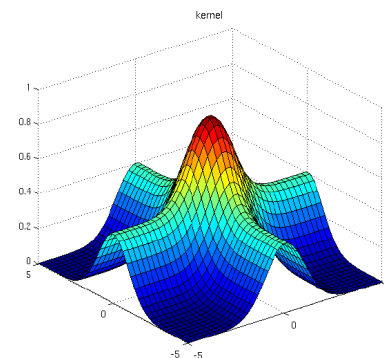


- A linear kernel plus a periodic results in functions which are periodic with increasing mean as we move away from the origin.

- **Adding across dimensions**

- Adding kernels which each depend only on a single input dimension results in a prior over functions which are a sum of one-dimensional functions, one for each dimension. That is, the function $f(x,y)$ is simply a sum of two functions $f_x(x) + f_y(y)$
- These kernels have the form:

$$k_{\text{additive}}(x, y, x', y') = k_x(x, x') + k_y(y, y')$$



Automatically choosing Kernels

- Sometimes, it is not obvious which kernel is appropriate for your problem.
- In fact, you might decide that choosing the kernel is one of the main difficulties in doing inference
- Just as you don't know what the true parameters are, you also don't know what the true kernel is.
- Probably, you should try out a few different kernels at least, and compare their marginal likelihood on your training data.
- Automatic ways:
 - <https://arxiv.org/abs/1302.4922>
(Structure Discovery in Nonparametric Regression through Compositional Kernel Search)
 - <https://github.com/jamesrobertlloyd/gp-structure-search>

Questions?

