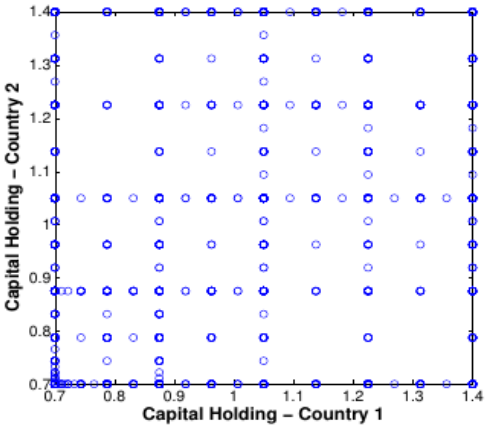
UNIL | Université de Lausanne

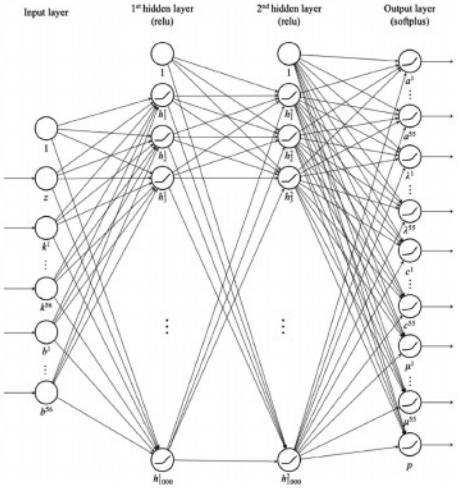
Advanced Methods in Computational Economics

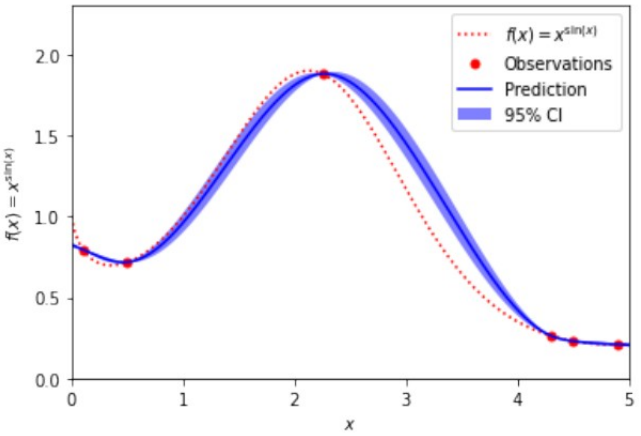
Day 3, Nov 21st, 2022

Simon Scheidegger
simon.scheidegger@unil.ch
November 14th- 21st, 2022
CREST

https://github.com/sischei/crest_comp_econ







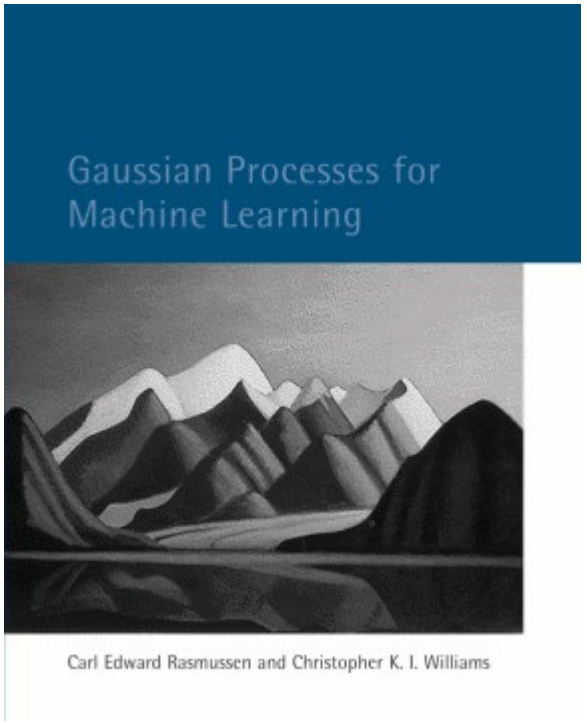
Today's Roadmap

- Lecture – part 1 (ca 90 min; examples in /lecture_3/code_part1):
  - Basics of Gaussian Process Regression
  - Noise-free kernels
  - Kernels with noise

- Lecture – part 2 (ca 90 min; examples in /lecture_3/code_part2):
  - The curse of dimensionality and how to deal with it (e.g., active subspaces)
  - Gaussian Mixture Models (unsupervised ML)
  - Bayesian active learning
  - Dynamic Programming/optimal control with GPs
  - An outlook to frontier topics of GPs (Limitations of GPs and "big data" Scalable GPs)

- Throughout lectures – hands-on:
  - Basics on GPs
  - *Option-pricing examples (self-study)
  - *Exercises on the day's topics (self-study)
  - Basics on Gaussian mixture models
  - A growth model solved with GPs and dynamic programming
  - Exercises on the day's topics

Gaussian Process Regression



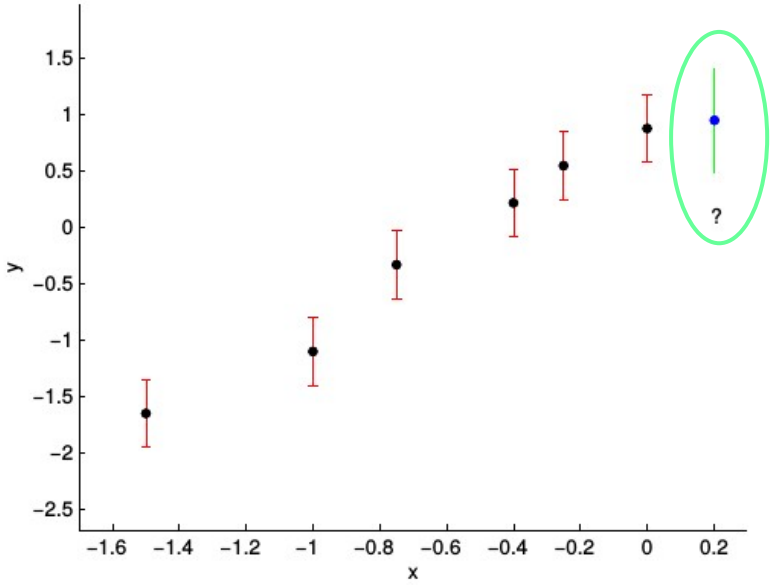
http://www.gaussianprocess.org/gpml/

Recall: 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

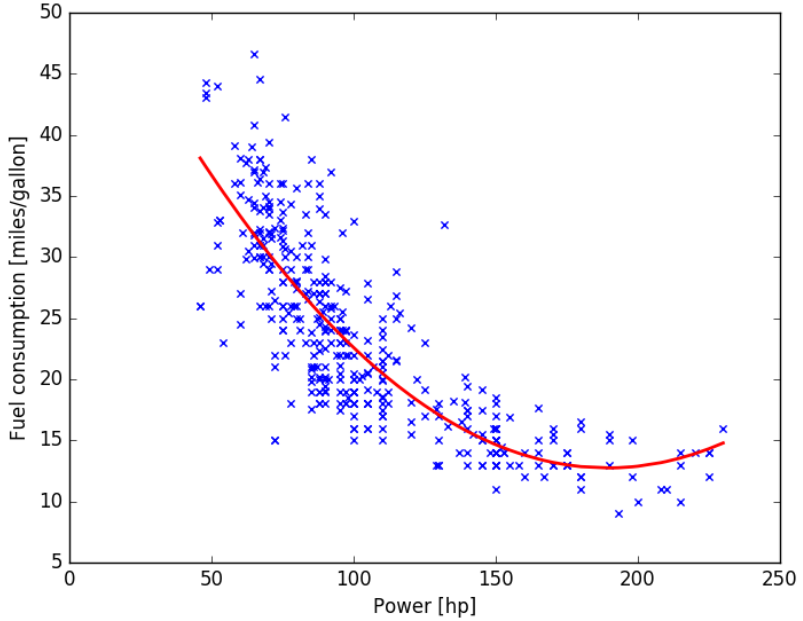
- If we expect the underlying function f(x) to be linear, and can make some assumptions about the input data, we might use a least-squares method to fit a straight line (linear regression).
- Moreover, if we suspect f(x) may also be quadratic, cubic, or even non-polynomial, we can use the principles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

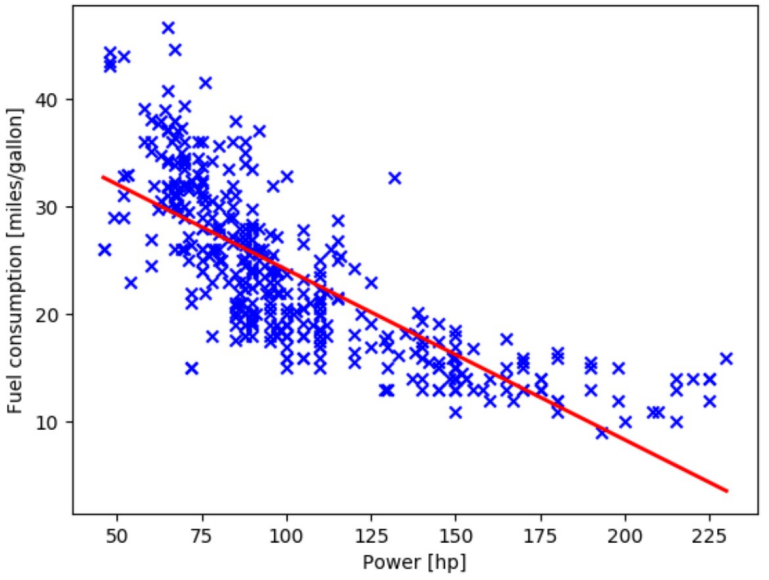
Model Selection

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

See: poly_reg.ipynb

```
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()
```

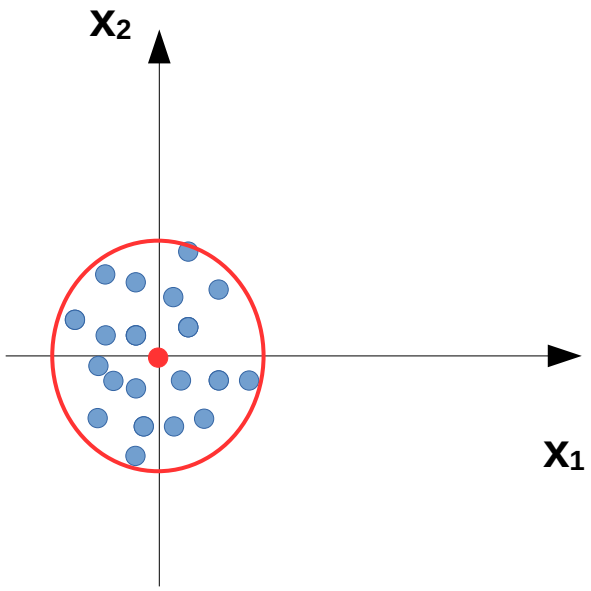
Degree of regression
1: linear
2: quadratic
...

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

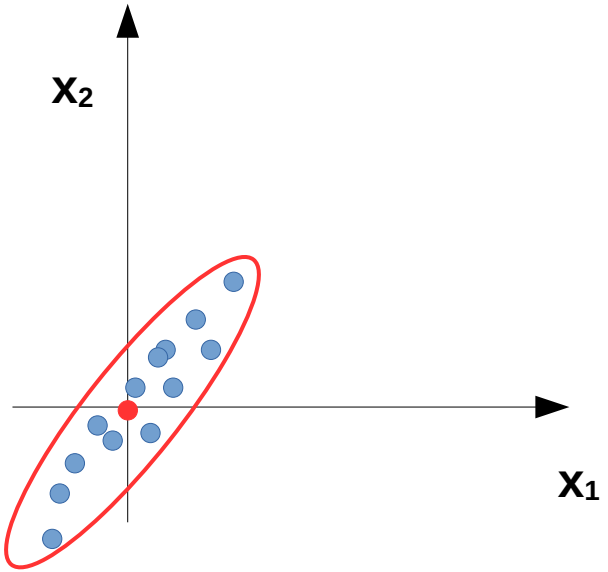


Say you measure two variables, e.g.,

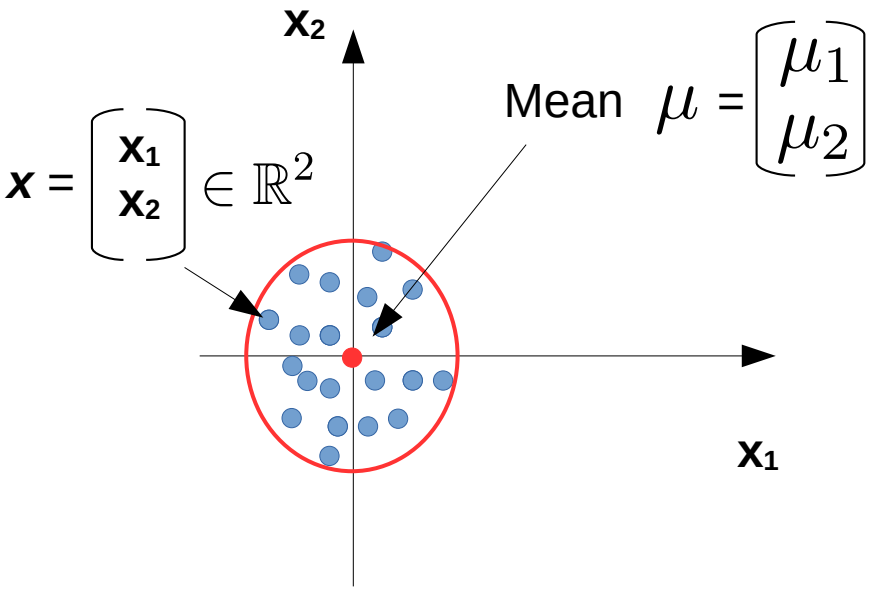
- x_1:height
- x_2: weight

→ plot

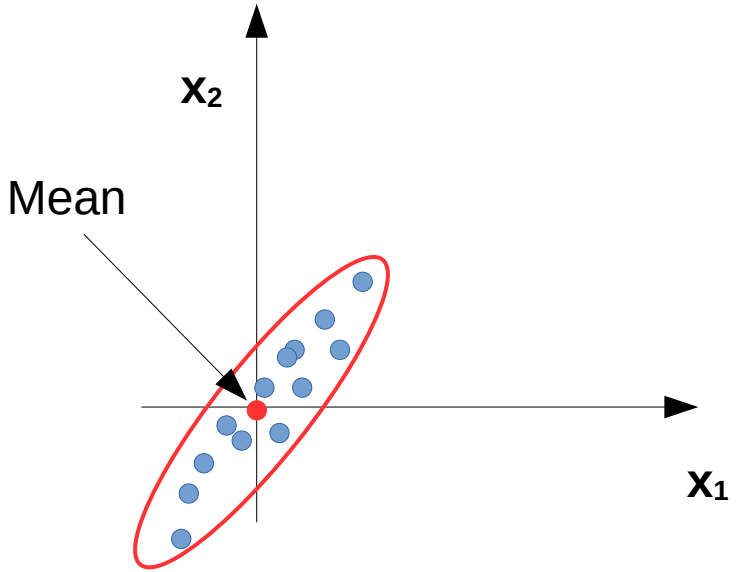
→ we want to fit a Gaussian to these points.



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 $\mathbf{x_1}$ increase $\mathbf{x_2}$?)

  → 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 $\mathbf{X} = (X_1, X_2, \ldots, X_n)^{\mathrm{T}}$ are random variables, each with finite variance and expected value, then the covariance matrix $\mathbf{K_{xx}}$ is the matrix whose (i ,j) entry is the covariance.

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

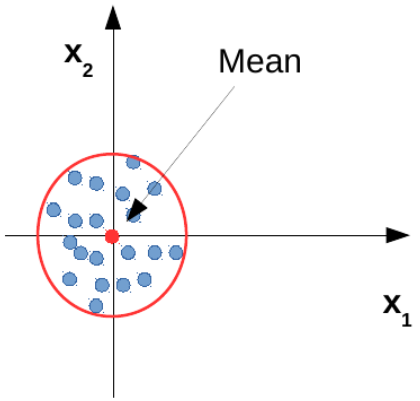
*Covariance matrix: positive definite (Matrix symmetric, its Eigenvalues positive).

Multivariate Gaussian (IV)

- Assume for a moment that **E**[ ] = 0 → Covariance **E**[x_1x_2] is a **"dot" product**.
- Assume two points [1 0] $\begin{bmatrix}1\\0\end{bmatrix}$ = 1 → **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 x_1 does not provide any information about x_2 as they are uncorrelated.



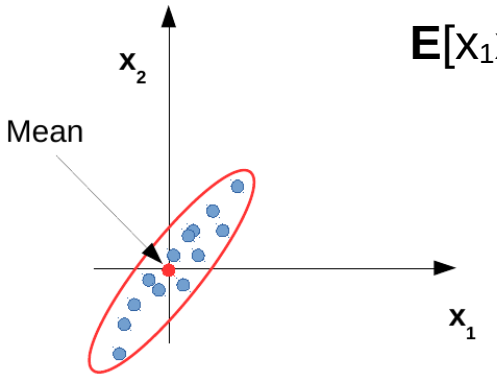
E[x_1x_2] = 0

Multivariate Gaussians (V)

- Assume for a moment that **E**[ ] = 0 → Covariance **E**[x_1x_2] is a "dot" product.
- Assume two points [1 0] $\begin{bmatrix}1\\0\end{bmatrix}$ = 1 → Covariance measure similarity.

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

E[x_1x_2] ≠ 0



→ Knowing about x_1 DOES provide information about x_2 .
→ if $x_1$is positive, x_2 is with great probability.
→ knowing something about x_1 allows us to know something about x_2.

Joint Gaussian distributions

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

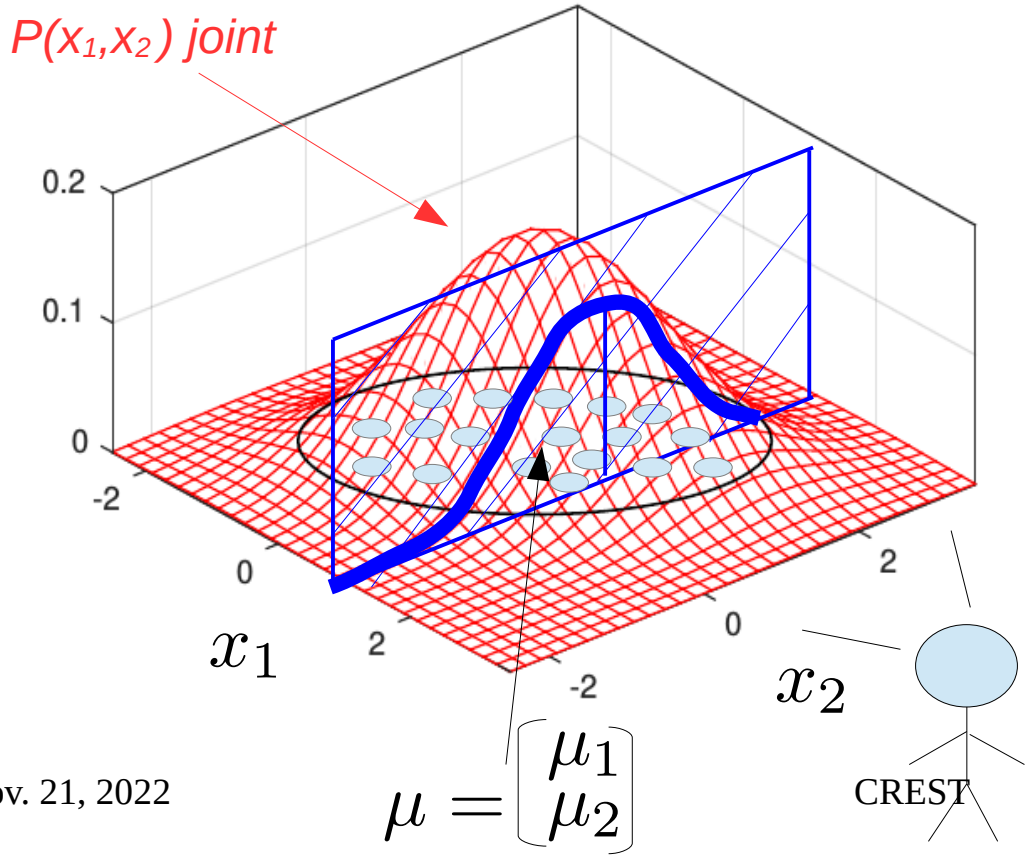
$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim N\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right)$$

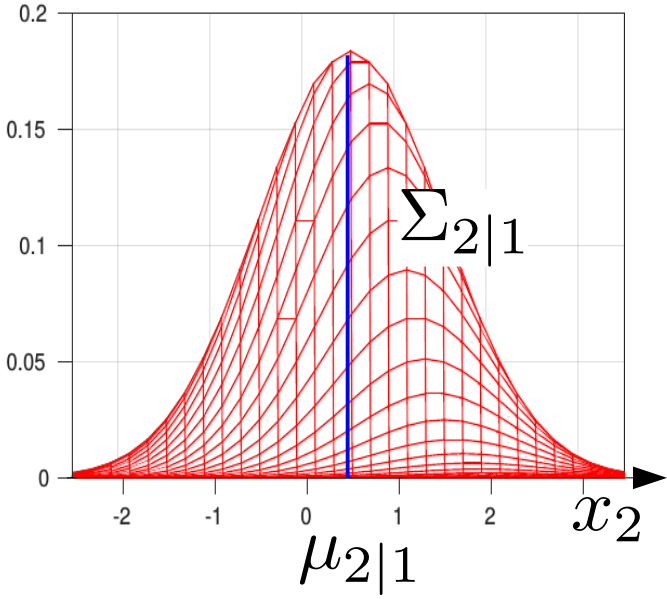
Mean

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*

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

Two "blocks" of vectors

Then the marginals are given by

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

and the posterior conditional is given by

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

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

Producing Data from Gaussians

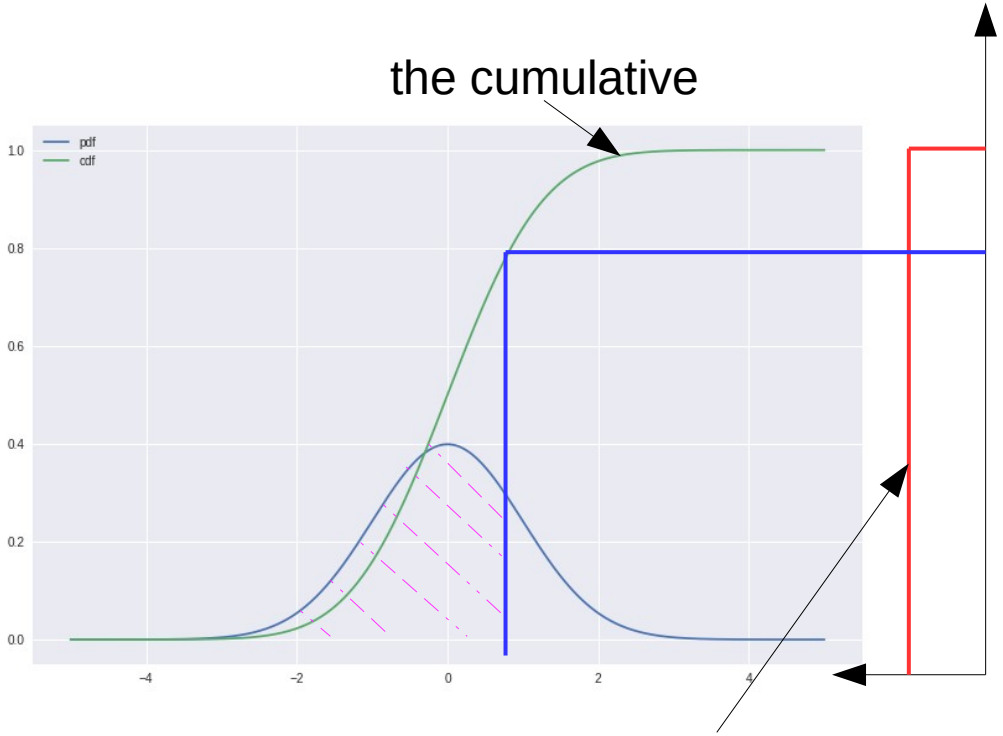
$$x_i \sim \mathcal{N}(0,1)$$

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

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: $\Sigma = LL^T$

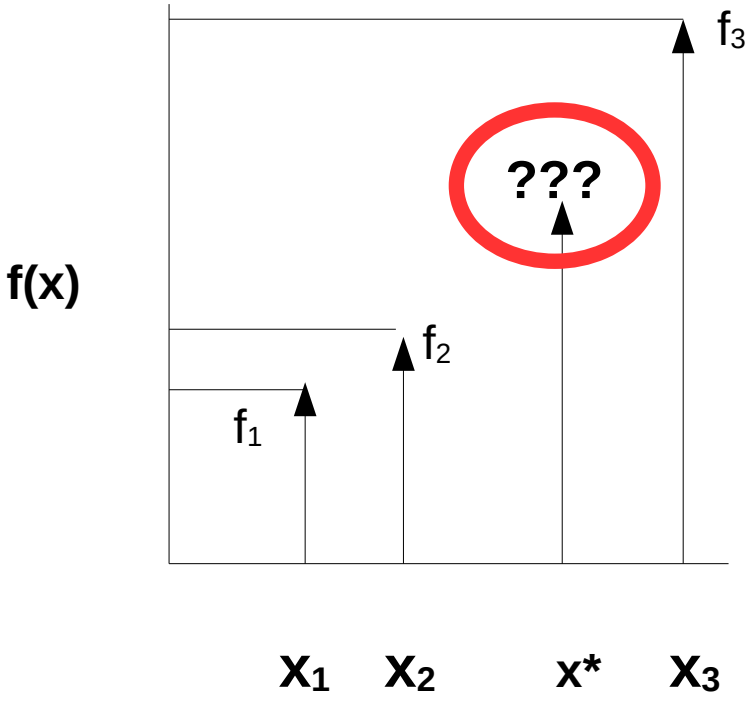


$$\rightarrow \quad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim N\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right) \quad \rightarrow \quad x \sim \mu + L\mathcal{N}(0,I)$$

$\mathbf{x}$ μ Σ

Recall Eq. (4.68) from the previous slide,

Observations → Interpolation



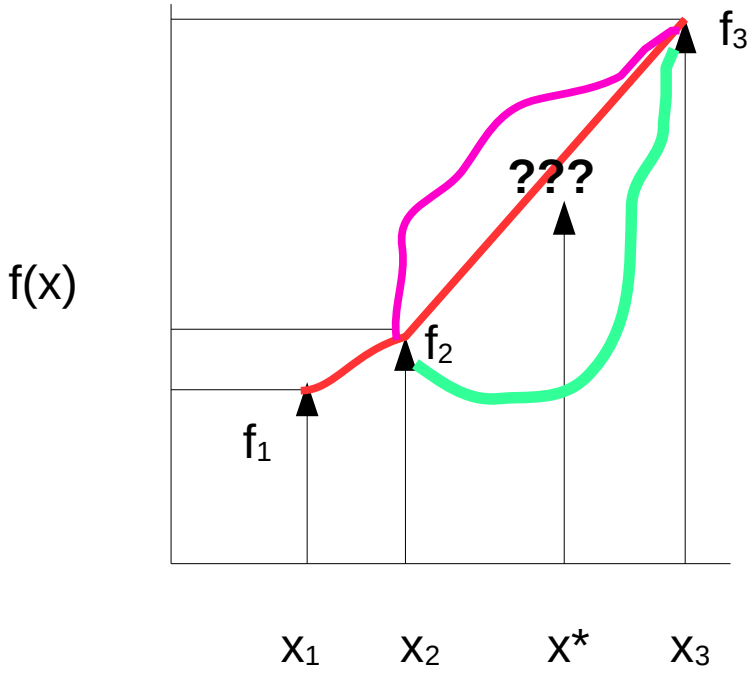
We have 3 observations at x_i for $f(x_i)$

→ **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 → Interpolation (II)

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



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

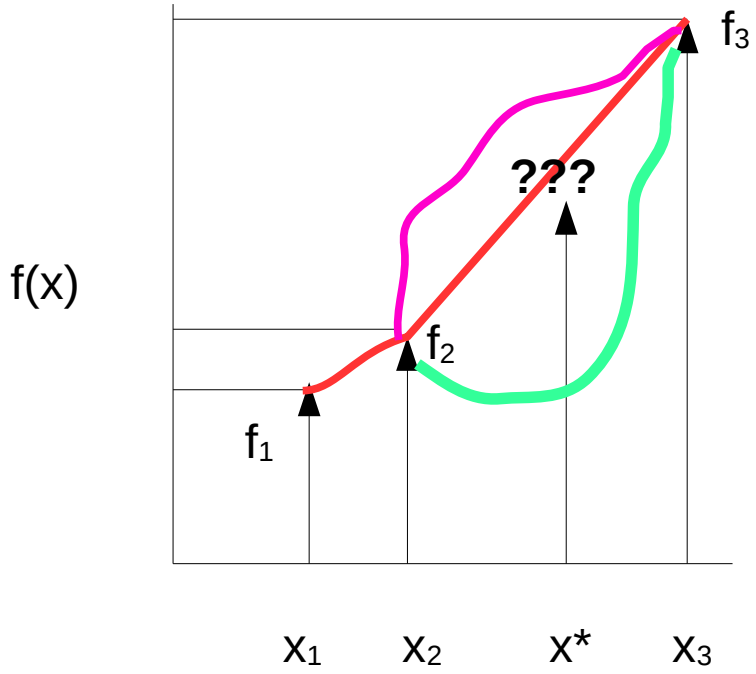
Note: f_1 and f_2 should probably be more correlated, as they are nearby (compared to f_1 and f_3).

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

→ 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).

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

σ_f^2 – controls vertical variation.

ℓ – controls horizontal length scale.

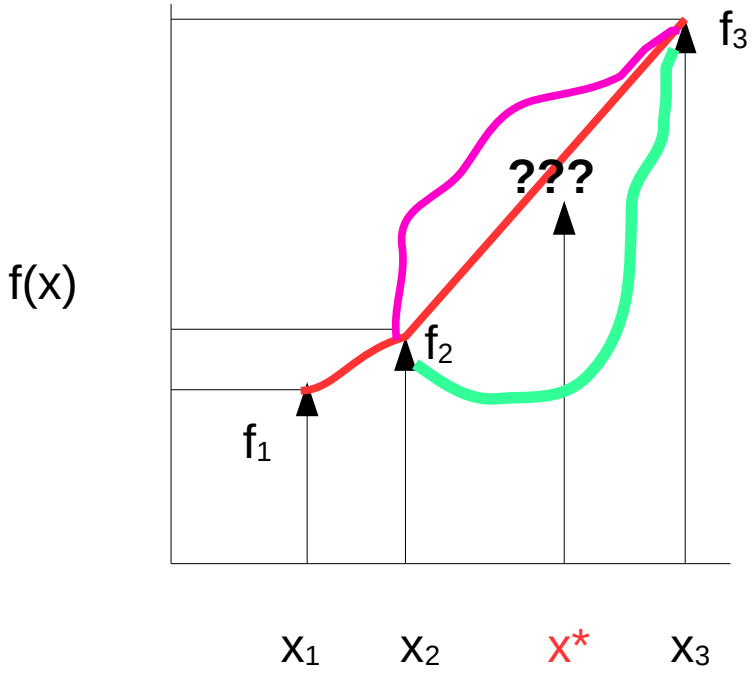
Observations → Interpolation (III)

Given data D = { (x_1,f_1), (x_2,f_2), (x_3,f_3)} → **$f(x^*) = f_*$** ?

→ Assume f ~ N(0 , K(·,·))

→ Assume f(x*) ~ 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**.

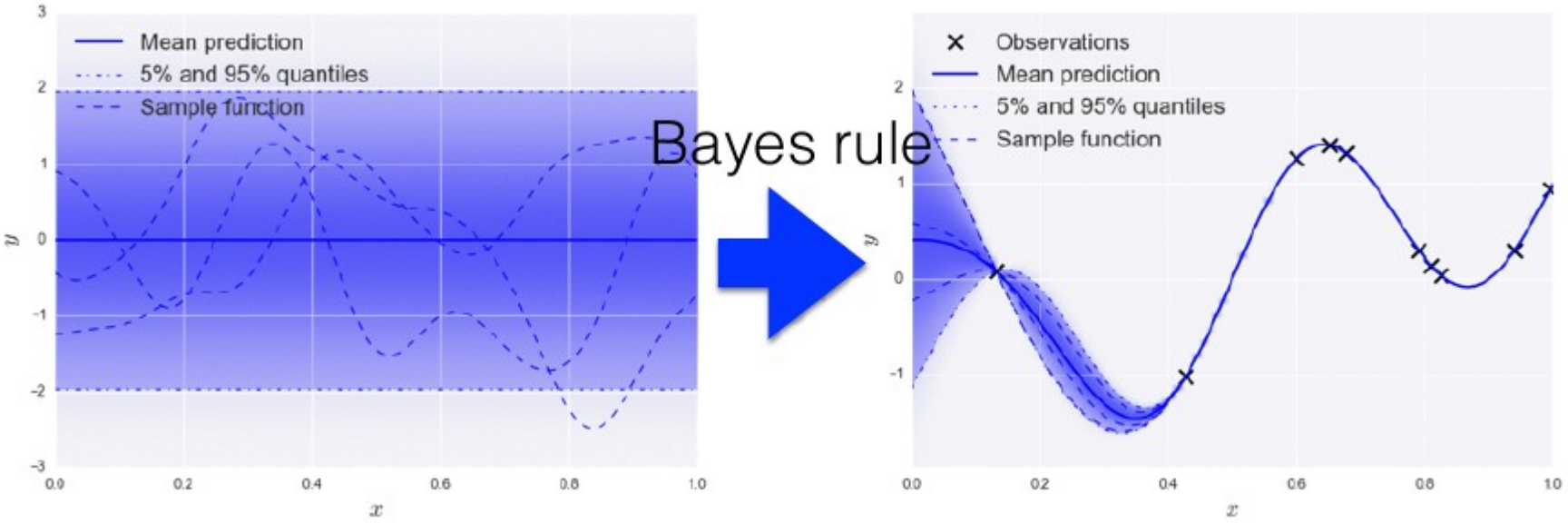
→ In this example, we "cut" in 3 dimensions.

→ 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)



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

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

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

Test point = interpolation at 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 × 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 $\boldsymbol{a} = (a_1, \ldots, a_N) = \left(\mathbf{K} + s_n^2 \mathbf{I}_N\right)^{-1} (\mathbf{t} - \mathbf{m})$
and ***t*** being the *N* observations.

→ We can think of the GP posterior mean as an approximation of f(·) 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.

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

GP – a distribution over functions

demo/1d_gp_example.ipynb

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

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

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

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

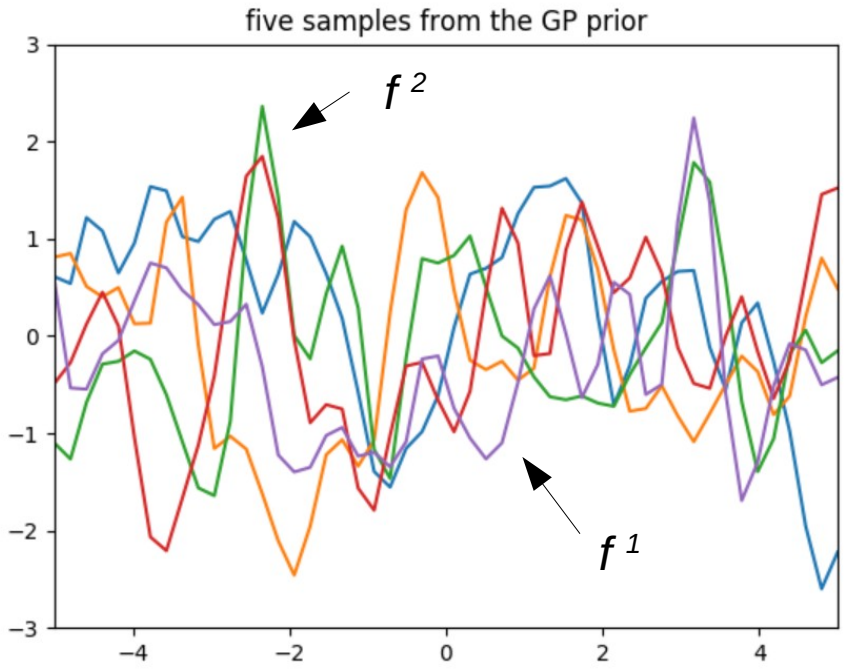
Procedure

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 demo/1d_gp_example.ipynb and Murphy, Chapter 15

```python
import numpy as np
import matplotlib.pyplot as pl

""" 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)
```

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

$$K_y = LL^T$$

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

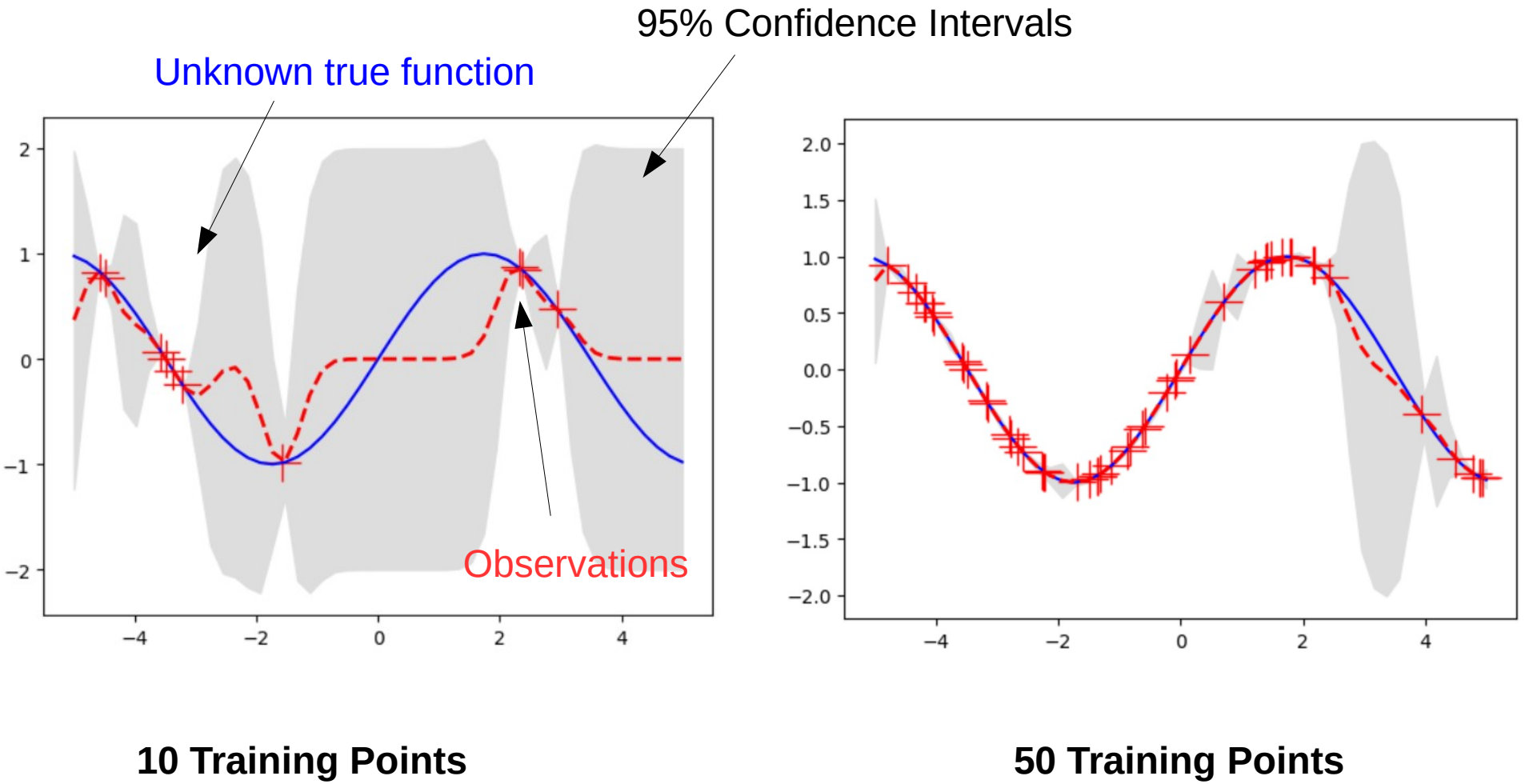
Algorithm 15.1: GP regression

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

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.

Non-hypercubic domain: GPR versus ASG

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

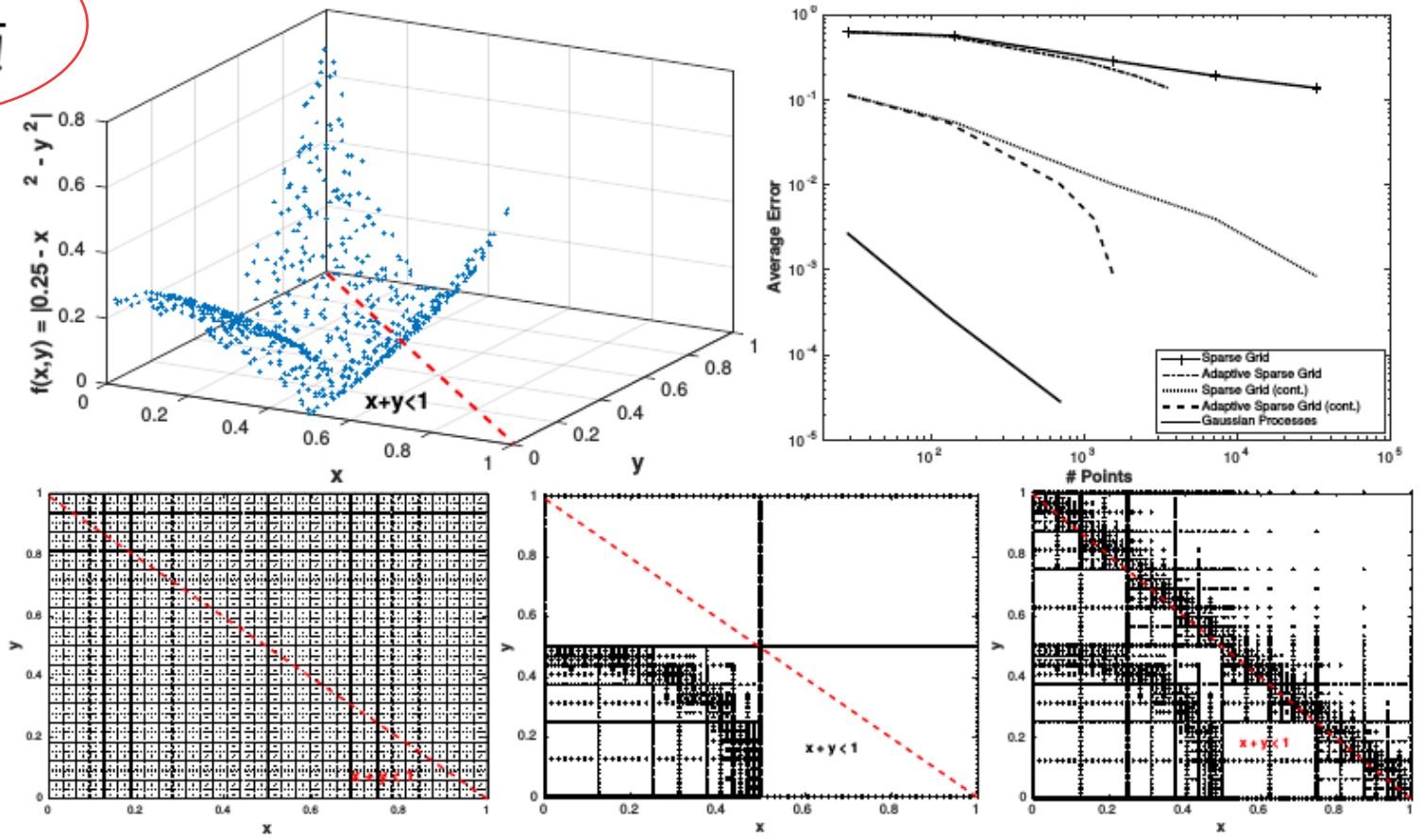


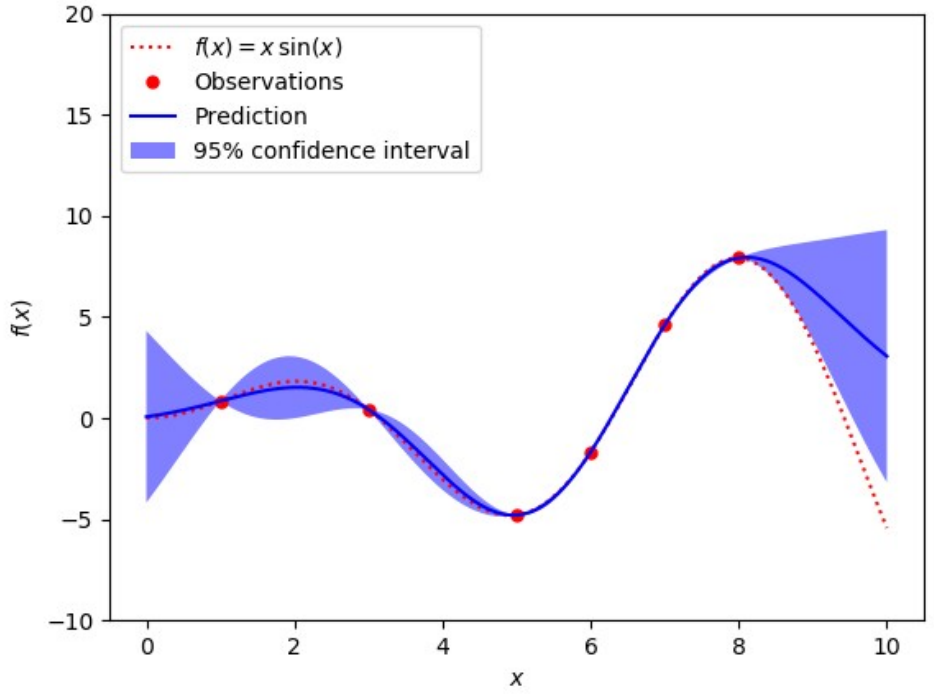
Figure: The upper left panel shows the analytical test function evaluated at random test points on the simplex. The upper right panel displays a comparison of the interpolation error for GPs, sparse grids, and adaptive sparse grids of varying resolution and constructed under the assumption that a continuation 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 demo/1d_GPR.ipynb



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)

Illustration of noiseless GPR prediction (II)

- We use a squared exponential kernel, aka Gaussian kernel or RBF kernel.
- In *1d*, this is given by $$\kappa(x, x') = \sigma_f^2 \exp(-\frac{1}{2\ell^2}(x - x')^2)$$
- Here *l* controls the horizontal length scale over which the function varies, and σ_f controls the vertical variation.
- We usually show predictions from the posterior, $p(f_* | X_*, X, f)$.
- We see (NEXT SLIDE) 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

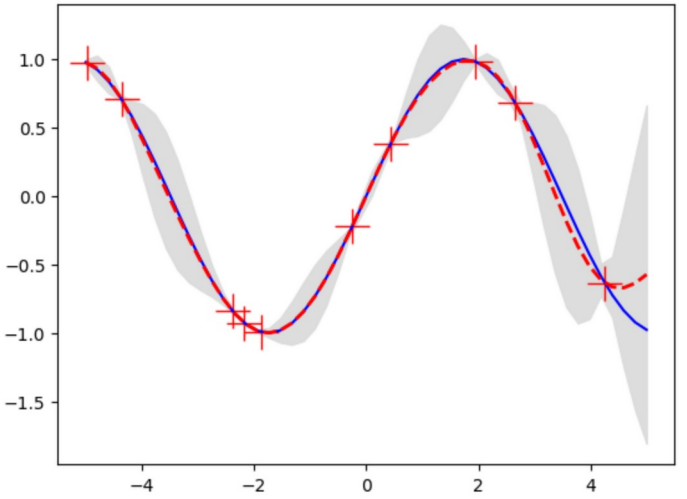
cf. demo/1d_gp_example.ipynb

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

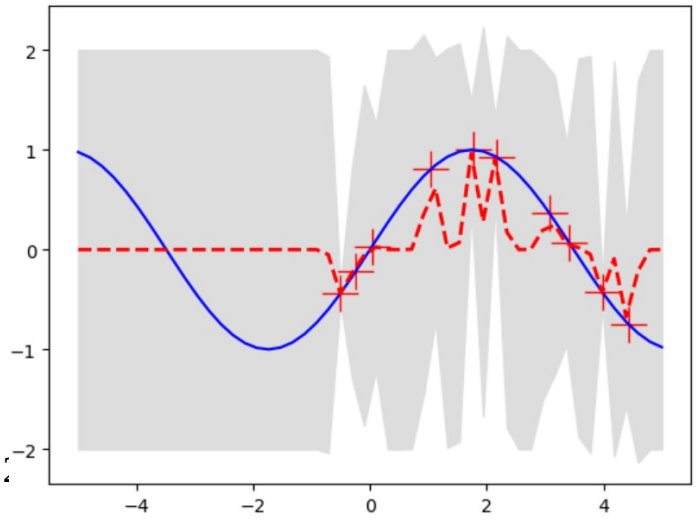
Let $\sigma_f^2 = 1$

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

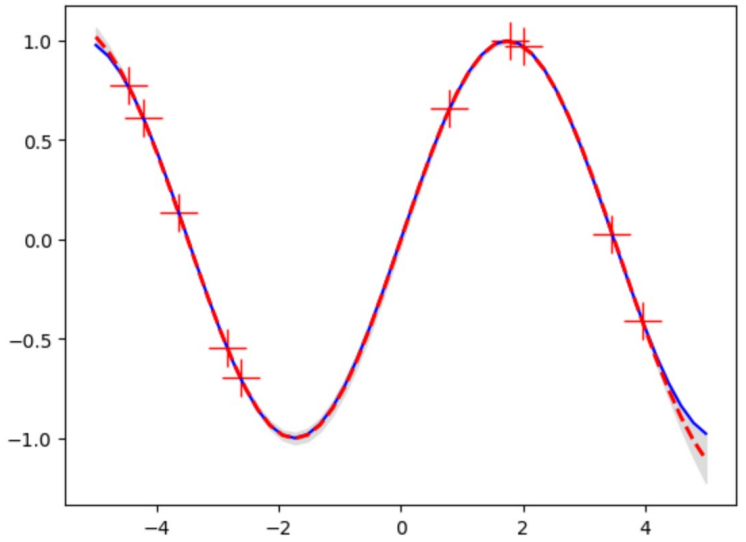
l^2 = 1.0



l^2 = 0.01



l^2 = 10.0



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

$$\mathrm{cov}\left[y_p, y_q\right] = \kappa(\mathbf{x}_p, \mathbf{x}_q) + \sigma_y^2 \delta_{pq} \quad \text{where} \quad \delta_{pq} = \mathbb{I}(p = q)$$

- 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

Noise in the diagonal, rest is as before.

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

Latent function.

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

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

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

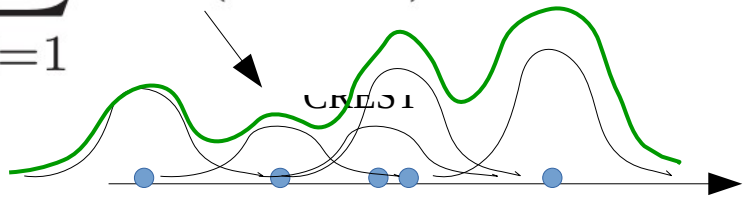
1 x N

N x 1

$$\overline{f}_* = \mathbf{k}_*^T \mathbf{K}_y^{-1}\mathbf{y} = \sum_{i=1}^{N} \alpha_i \kappa(\mathbf{x}_i, \mathbf{x}_*) \quad \text{where } \boldsymbol{\alpha} = \mathbf{K}_y^{-1}\mathbf{y}$$

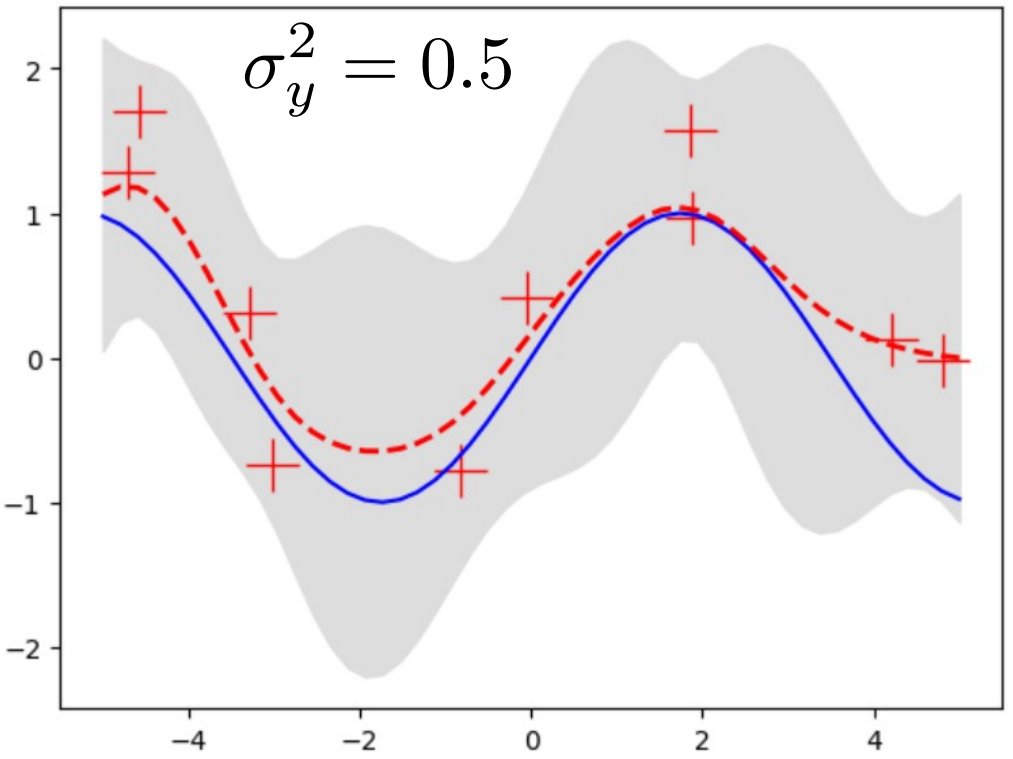
N x N

from training data



Some Plots

cf. demo/1d_gp_example.ipynb



- 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.

“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 $\boldsymbol{\theta}$

- One can show that the following holds.

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

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

Computational complexity:

- It takes *O(N³)* time to compute $\mathbf{K}_y^{-1}$

- *O(N²)* time per hyper-parameter to compute the gradient.

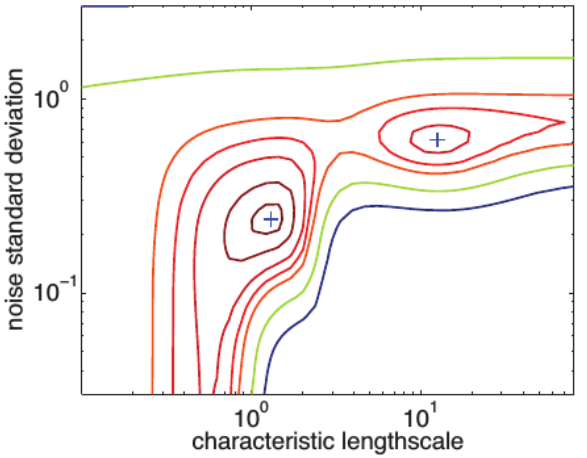
Careful: Different optima correspond to different interpretations/believes

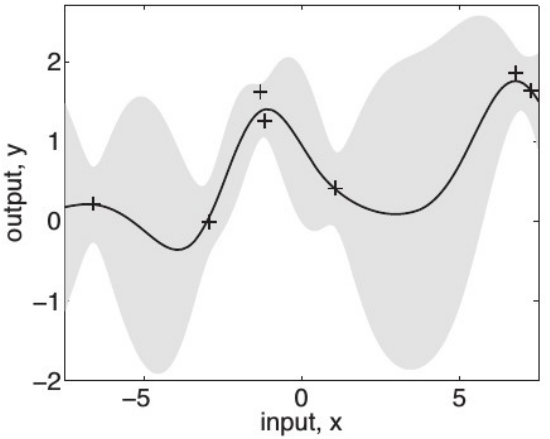
Fig. 5.5 of (Rasmussen and Williams 2006).

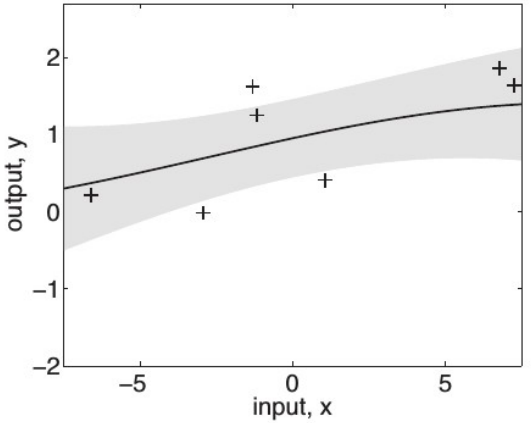
- We use the SE kernel

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

- with $\sigma_f^2 = 1$
- We plot $\log p(\mathbf{y}|\mathbf{X}, \ell, \sigma_y^2)$ (where X and 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
demo/GPR_scikit_noise.ipynb

```
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'$f(x) = x\,\sin(x)$')
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

demo/scikit_multi-d.ipynb

```
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)

demo/scikit_multi-d.ipynb

```
#-----------------------------------------------------------------------

# Important -- save the model to a file
with open('2d_model.pcl', '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.pcl', '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.

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

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 σ²** 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

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

- 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

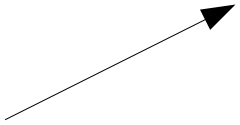
- The **Matérn kernel** is a stationary kernel and a generalization of the RBF kernel.
- It has an additional parameter ν which controls the smoothness of the resulting function. It is parameterized by a length-scale parameter $\ell > 0$, which can either be a scalar (isotropic variant of the kernel) 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 $\nu \to 0$, the Matérn kernel converges to the RBF kernel.
- When $\nu = 1/2$, the Matérn kernel becomes identical to the absolute exponential kernel.

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



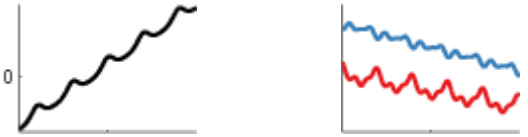
- This are popular choices for learning functions that are not infinitely differentiable (as assumed by the RBF kernel) but at least once ($\nu = 3/2$) (if $\nu = 5/2$, the kernel is twice differentiable).

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**



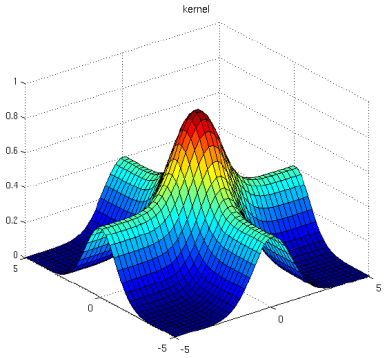
  - A linear kernel plus a periodic results in functions which are periodic with increasing mean as we move away from the origin.

- **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

Pricing options with GPs (BS model)

- see finance_demo/GP-BS-Pricing_01.ipynb

Greeks

- see finance_demo/GP-BS-Pricing_02.ipynb

- The GP provides analytic derivatives with respect to the input variables

$$\partial_{X_*}\mathbb{E}[\mathbf{f}_*|X, Y, X_*] = \partial_{X_*}\boldsymbol{\mu}_{X_*} + \partial_{X_*}K_{X_*,X}\alpha$$

$$\partial_{X_*}K_{X_*,X} = \frac{1}{\ell^2}(X - X_*)K_{X_*,X}$$

$$\alpha = [K_{X,X} + \sigma_n^2 I]^{-1}\mathbf{y}$$

- Second-order sensitivities → diff. wrt. X $_*$

Pricing options with GPs (Heston)

- see finance_demo/GP-Heston-Pricing_03.ipynb

- You need to install first Junyan Xu's Python Heston Option Pricing Library
- pip install git+https://github.com/junyanxu/Python-Heston-Option-Pricer

Recall: The Heston Model

- see finance_demo/GP-Heston-Pricing_03.ipynb
- In the following example, the portfolio holds a long position in both a European call and a put option struck on the same underlying, with K = 100.
- We assume that the underlying follows Heston dynamics:

$$\frac{dS_t}{S_t} = \mu dt + \sqrt{V_t} dW_t^1,$$

$$dV_t = \kappa(\theta - V_t)dt + \sigma\sqrt{V_t} dW_t^2,$$

$$d\langle W^1, W^2 \rangle_t = \rho dt,$$

Recall: The Heston Model

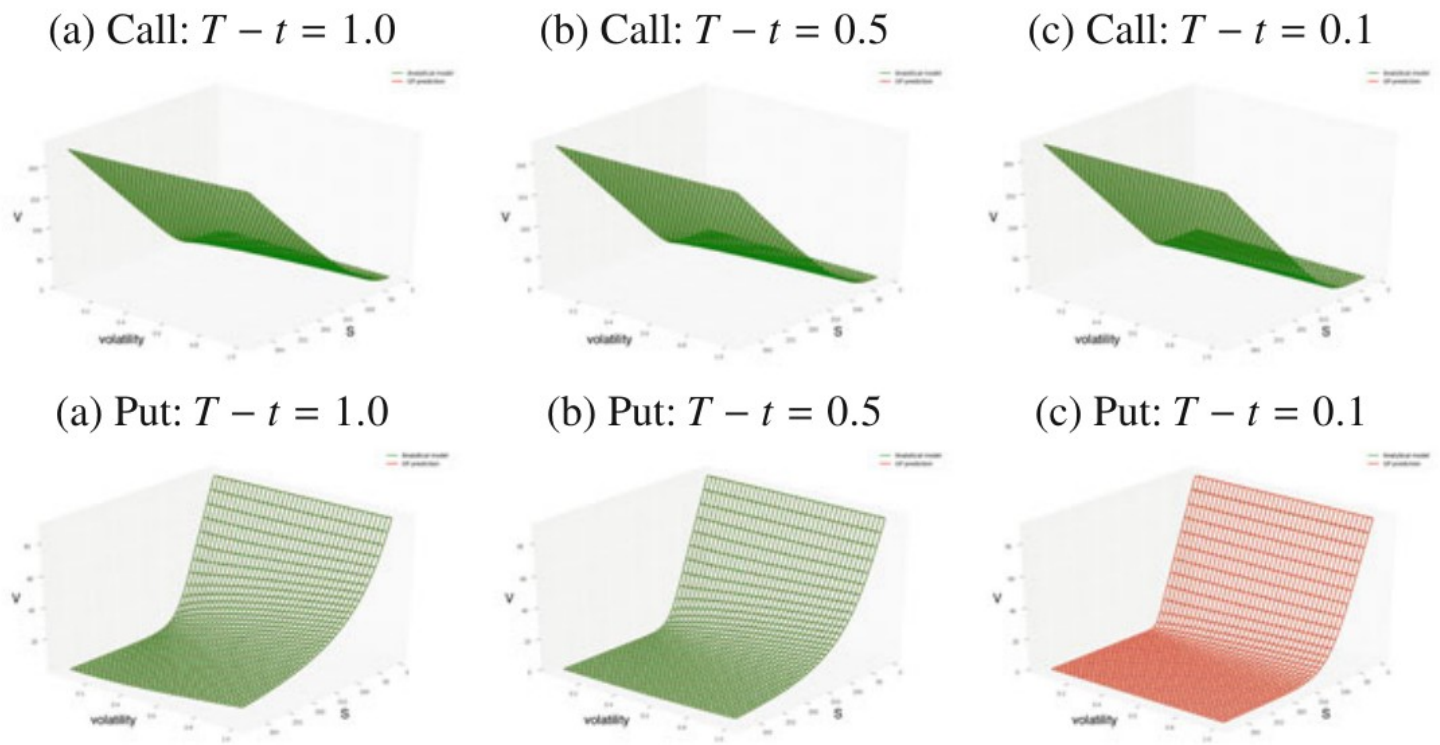
- This table shows the values of the parameters for the Heston dynamics and terms of the European Call and Put option contracts.

Parameter description	Symbol	Value
Mean reversion rate	κ	0.1
Mean reversion level	θ	0.15
Vol. of Vol.	σ	0.1
Risk-free rate	r_0	0.002
Strike	K	100
Maturity	T	2.0
Correlation	ρ	-0.9

- This table shows the values for the Euler time stepper used for market risk factor simulation.

Parameter description	Symbol	Value
Number of simulation	M	1000
Number of time steps	n_s	100
Initial stock price	S_0	100
Initial variance	V_0	0.1

Some plots



Comparison of the gridded Heston model call (top) and put (bottom) price surfaces at various time to maturities, with the GP estimate.
The GP estimate is observed to be practically identical (slightly below in the first five panels and slightly above in the last one).
Within each column in the figure, the same GP model has been simultaneously fitted to both the Heston model call and put price surfaces over a 30 × 30 grid of prices and volatilities, fixing the time to maturity.
Across each column, corresponding to different time to maturities, a different GP model has been fitted.
The GP is then evaluated out-of-sample over a 40 × 40 grid, so that many of the test samples are new to the model. This is repeated over various time to maturities.

Gaussian Process Classification

- Either go over this notebook, or read the slides first:

  see demo/GPC_classification_step_by_step.ipynb

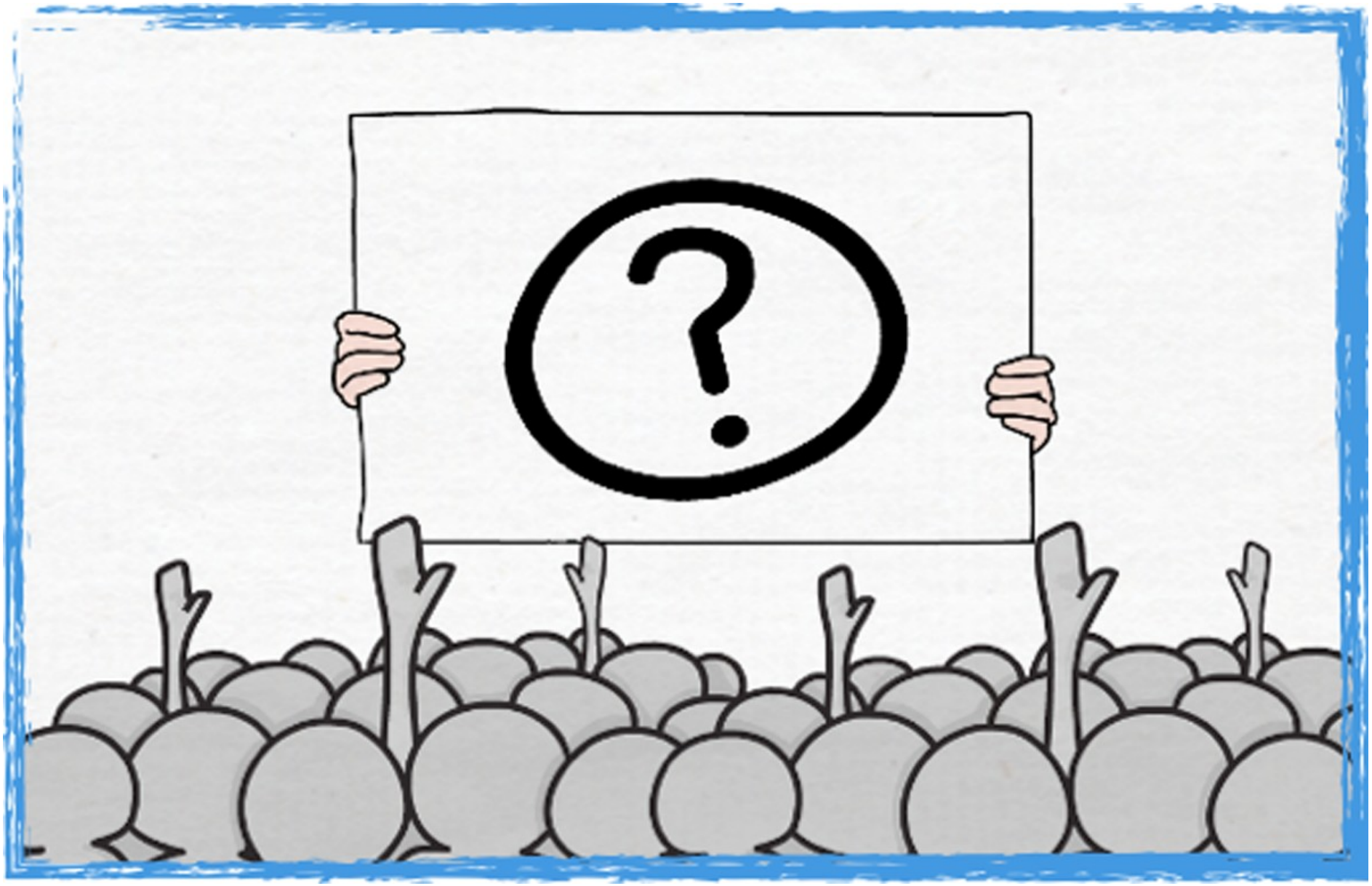
Gaussian Process Classification

- While it is possible to perform multi-class classification with a Gaussian process, we will consider only two classes, labelled as +1 and -1.
- The task of the process is then to model the probability that input x belongs to class 1, which means that the output should be a value between 0 and 1 (inclusive) like all good probabilities.
- We will arrange this by squashing it using the logistic function

$$P(t^* = 1|a) = \sigma(a) = 1/(1+\exp(-a))$$

Summary on GPs

- For a fairly simple idea, Gaussian processes do tend to work very well on a wide range of topics.
- The way that the covariance function explicitly encodes the correlations that can be seen in the data means that the user has a lot of control.
- Even in the simple treatment here we have put quite a lot of effort into making the computations numerically stable and relatively fast.
- However, there is much more that can be done, including methods for approximation to speed things up significantly.



Part 2

- Lecture (60 min; examples in /lecture_3/code_part1):
  - The curse of dimensionality and how to deal with it (e.g., active subspaces)
  - Gaussian Mixture Models (unsupervised ML)
  - Bayesian active learning
  - Dynamic Programming/optimal control with GPs
  - An outlook to frontier topics of GPs (Limitations of GPs and “big data” Scalable GPs)

- Throughout lectures – hands-on:
  - Basics on Gaussian mixture models
  - A growth model solved with GPs and dynamic programming
  - Exercises on the day’s topics

Motivation – the “curse of dimensionality”

- Modern dynamic economic models such (e.g. DSGE) are extremely **rich** to capture **all the effects of interest**:

- large stochastic shocks that lead to highly non-linear policies (e.g., rare disaster shocks).
- many agents that lead to a high-dimensional state space.
- ...

- Standard solution techniques such as log-linearization **often fail to deliver reliable results** across the entire domain of interest.

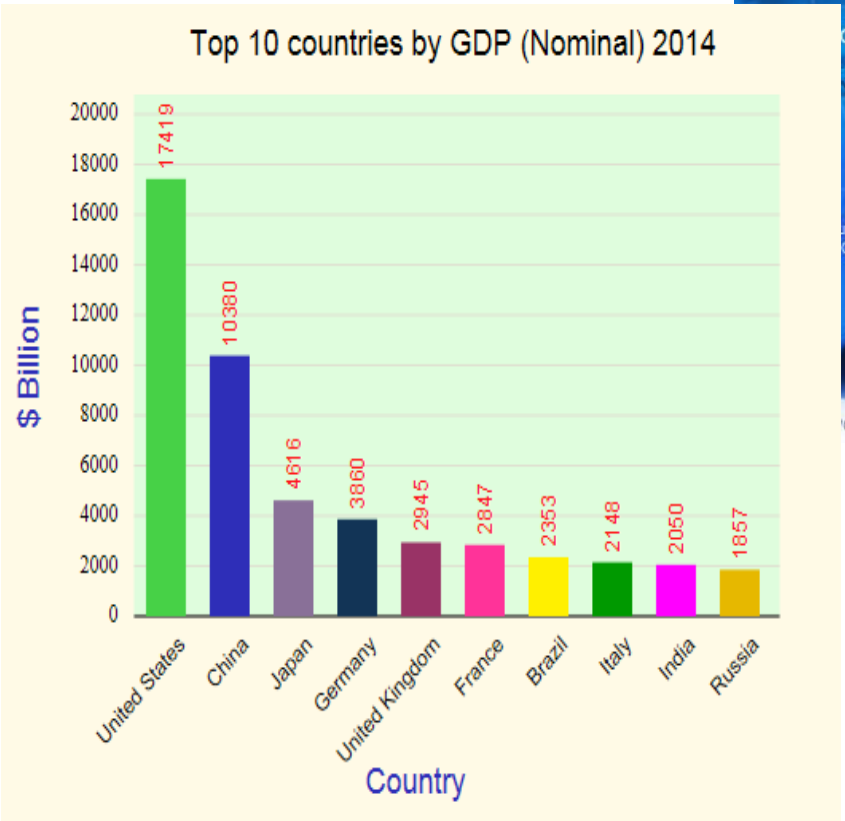
- The latter method may be useful to describe an economy that operates in normal times, but **fails** in the presence of nonlinearities such as occasionally binding constraints, among other types of salient features of the economic reality that the modelers would like to capture appropriately in their models.

Example – Heterogeneity in IRBC models

- Model trade imbalance

- FX rates

-...



- How many regions does a minimal model have?
- Are policy functions smooth? (borrowing constraints)

→ **Model heterogeneous & high-dimensional**

Example – Heterogeneity in OLG* models

*Overlapping generation models

To model e.g. social security:

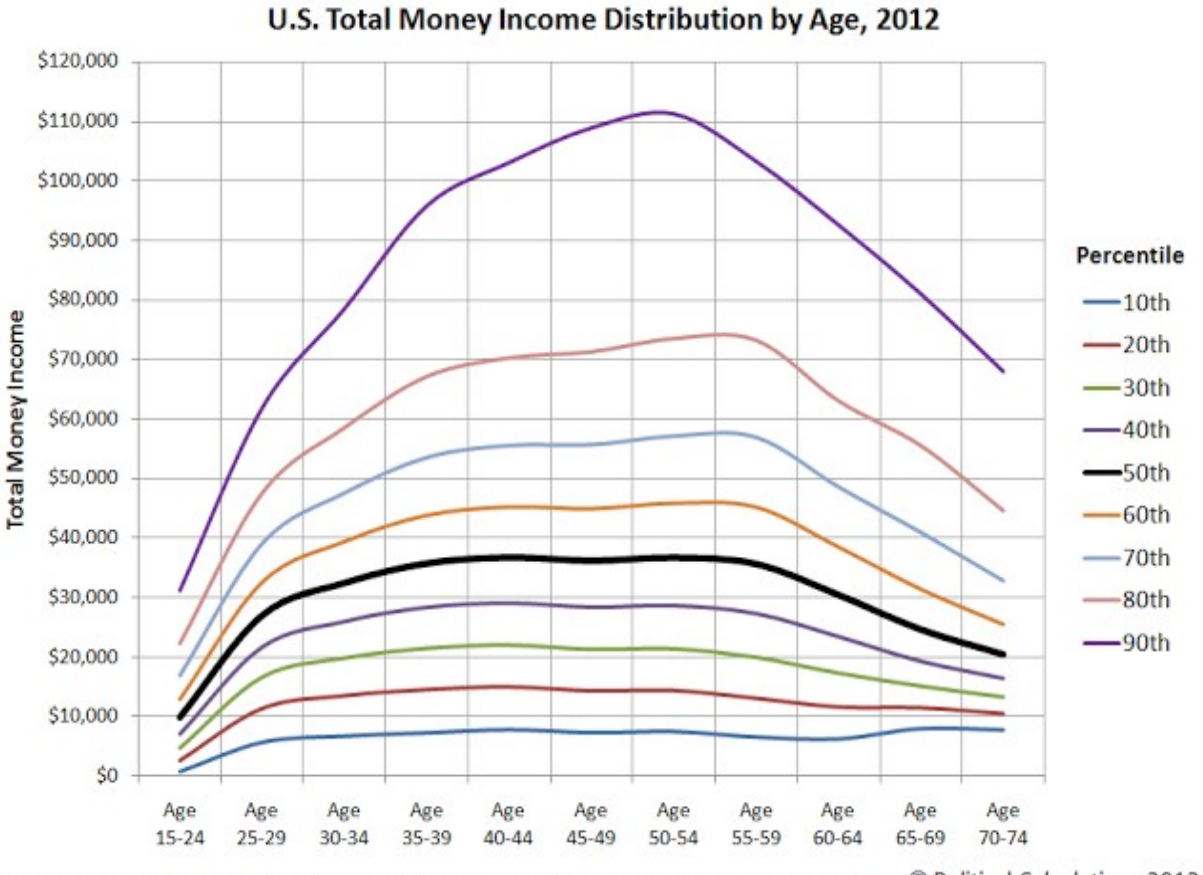
- How many age groups?

- borrowing constraints?

- aggregate shocks?

- …

→ Model: heterogeneous & high-dimensional



Financial markets: non-Gaussian returns

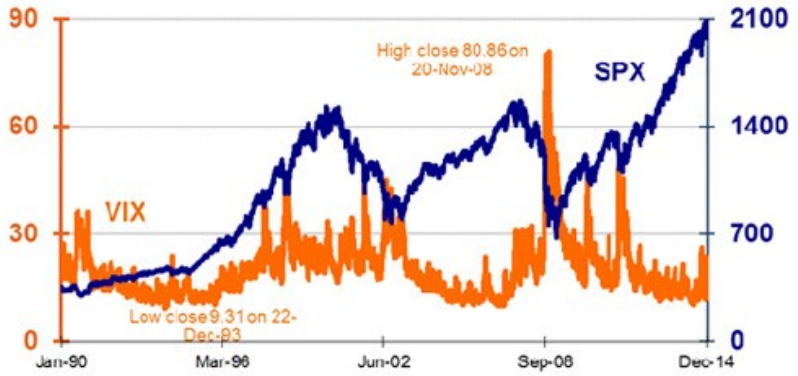
- Derivative contracts giving a right to buy or sell an underlying security.
  - *European* if exercise at expiration only.
  - ***American*** if exercise any time until expiration.

- American options are extremely challenging:
  → Dynamic optimization problem.

- Basic models do not describe dynamics accurately (e.g., Hull (2011)).

- Financial returns are often not Gaussian.

- Realistic models are hard to deal with, as they need many factors.
  → Curse of dimensionality.



Dynamic Programming/Value Function Iteration

e.g. Stokey, Lucas & Prescott (1989), Judd (1998), ...

Dynamic programming seeks a time-invariant policy function ***p*** mapping a state $\mathbf{x}_t$ into the control $\mathbf{u}_t$ such that for all $t \in \mathbb{N}$ $u_t = p(x_t)$
The solution is approached in the limit as $j \to \infty$ by iterations on:

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

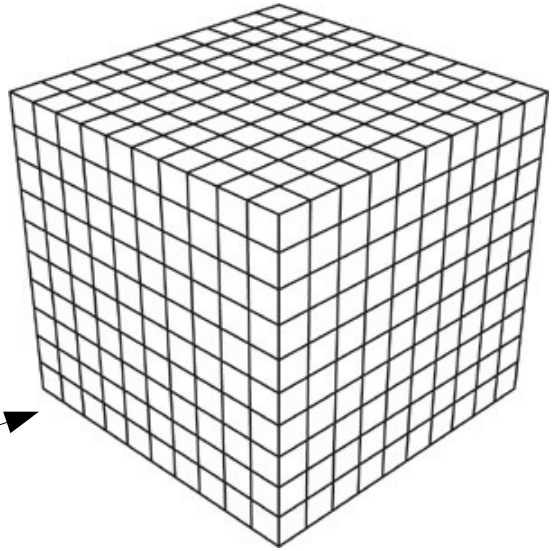
s.t.

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

x: grid point, describes your system.
State-space potentially **high-dimensional**.

`old solution':
high-dimensional function on which **we interpolate**.

→ **N^d** points in ordinary discretization schemes.



→ Use-case for Gaussian Process regression.

How many is dimensions is high dimensions?

?

How many is dimensions is high dimensions?

Number of parameters (the dimension)	**Number of model runs** (at 10 points per dimension)	**Time for parameter study** (at 1 second per run)
1	10	10 sec
2	100	~ 1.6 min
3	1,000	~ 16 min
4	10,000	~ 2.7 hours
5	100,000	~ 1.1 days
6	1,000,000	~ 1.6 weeks
…	…	…
20	1e20	3 trillion years (240x age of the universe)

How many is dimensions is high dimensions?

Number of parameters (the dimension)	Number of model runs (at 10 points per dimension)	Time for parameter study (at 1 second per run)
1	10	10 sec
2	100	~ 1.6 min
3	1,000	~ 16 min
4	10,000	~ 2.7 hours
5	100,000	~ 1.1 days
6	1,000,000	~ 1.6 weeks
…	…	…
20	1e20	3 trillion years (240x age of the universe)

Dimension reduction
Exploit symmetries, e.g., via the active subspace method

Deal with #Points
Adaptive Sparse Grids

High-performance computing
Reduces time to solution, but not the problem size

The Curse of Dimensionality

- Why is it hard (impossible) to learn functions in high dimensions?
- Methods relying on **local similarity measures** (e.g., kernels in the context of GPR) have severe problems.
- "The curse of highly variable functions for local kernel machines", Bengio et al. (2006)).

→ The reason is that the **Euclidean distance** is not a good "closeness" measure in high-dimensions.

Most of the volume of a high-dimensional orange is in the skin, not in the pulp. If a constant number of examples is distributed uniformly in a high-dimensional hypercube, beyond some dimensionality most examples are closer to a face of the hypercube than to their nearest neighbors.*

* "A few useful things to know about machine learning", Pedro Domingos, (2012)

Volumes in high dimensions

- A lot of probability theory uses the idea of volumes.
- For example, the probability and expectation are usually defined as:

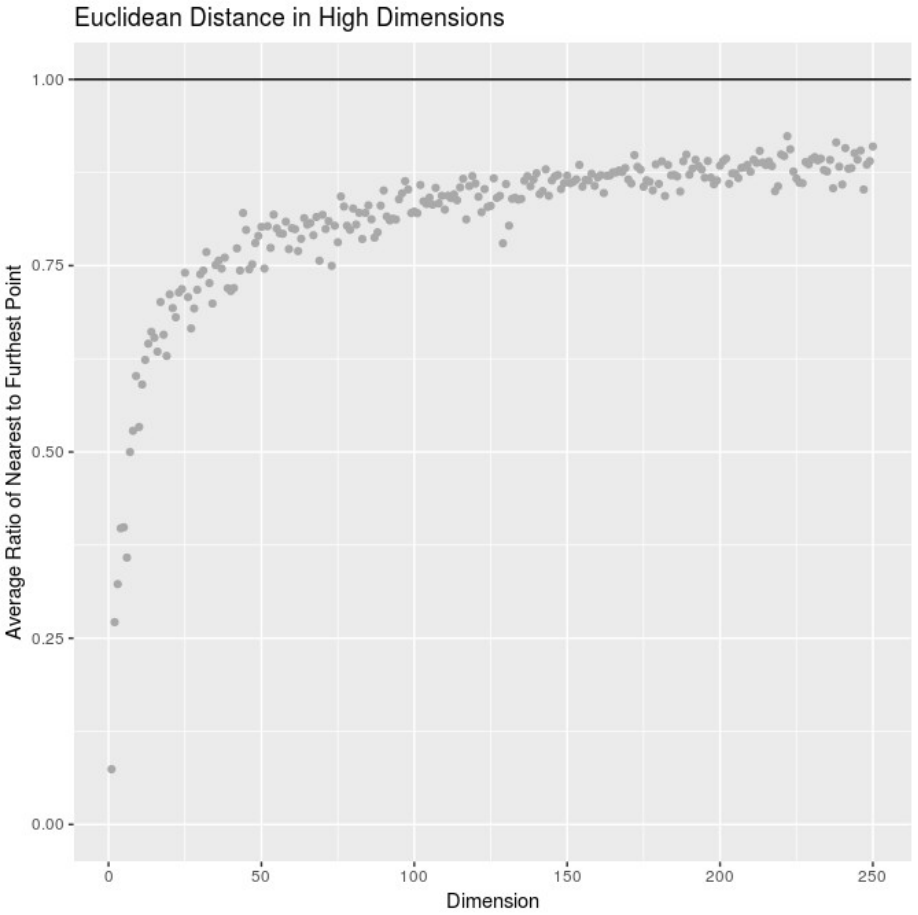
$$\mathbb{P}_\mu(\mathcal{A}) = \int_A d\mu \quad ; \quad \mathbb{E}_\mu(f(X)) = \int_\chi f(x) d\mu(x)$$

- where the differential element is usually a volume, $d_\mu(x)=p(x)dx_1 \ldots dx_d$, and p(x) is a density, but volumes are really weird in higher dimensions.
- The main idea is that our ideas of distance no longer make sense.

Volumes in high dimensions (II)

- Consider the following:
  Let's simulate a bunch of random normal points in many dimensions and plot the average ratio of the distance to nearest point to the distance to farthest point.

$$D = \sqrt{\sum_{i=1}^{d} (x_i)}$$



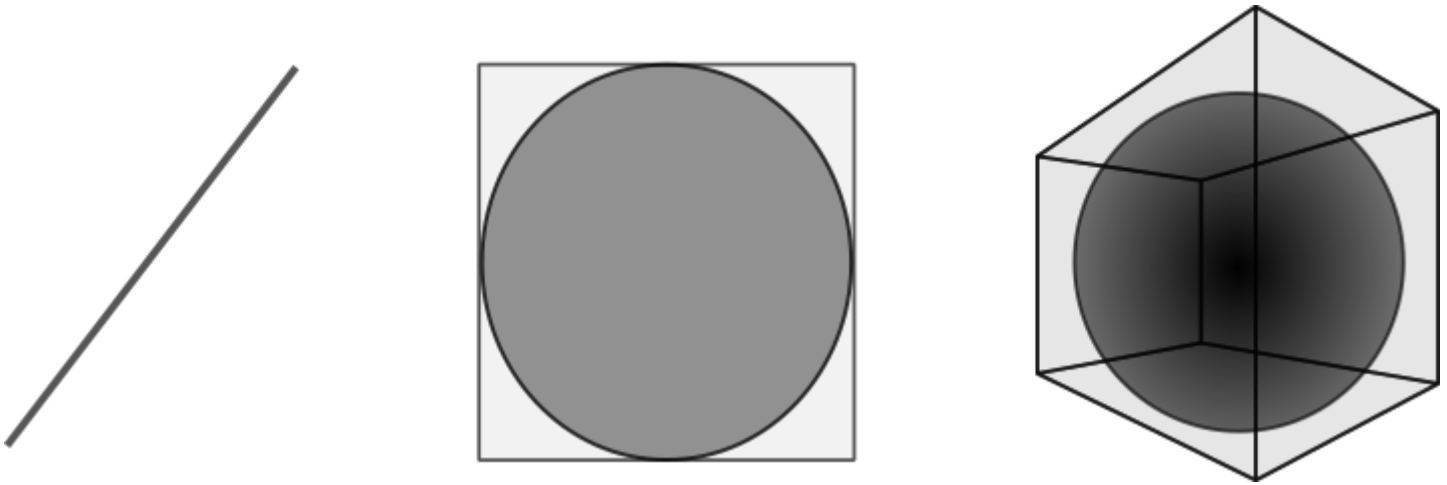
- This ratio tends to one, all points "look alike".

  → No good notion of "locality".

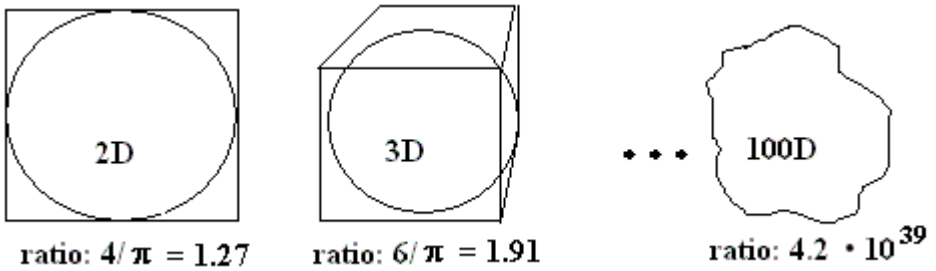
  → Problems, e.g., for GPR.

Volumes in high dimensions (III)

- On the same subject, consider a cube of unit lengths containing a sphere of unit radius in higher dimensions:

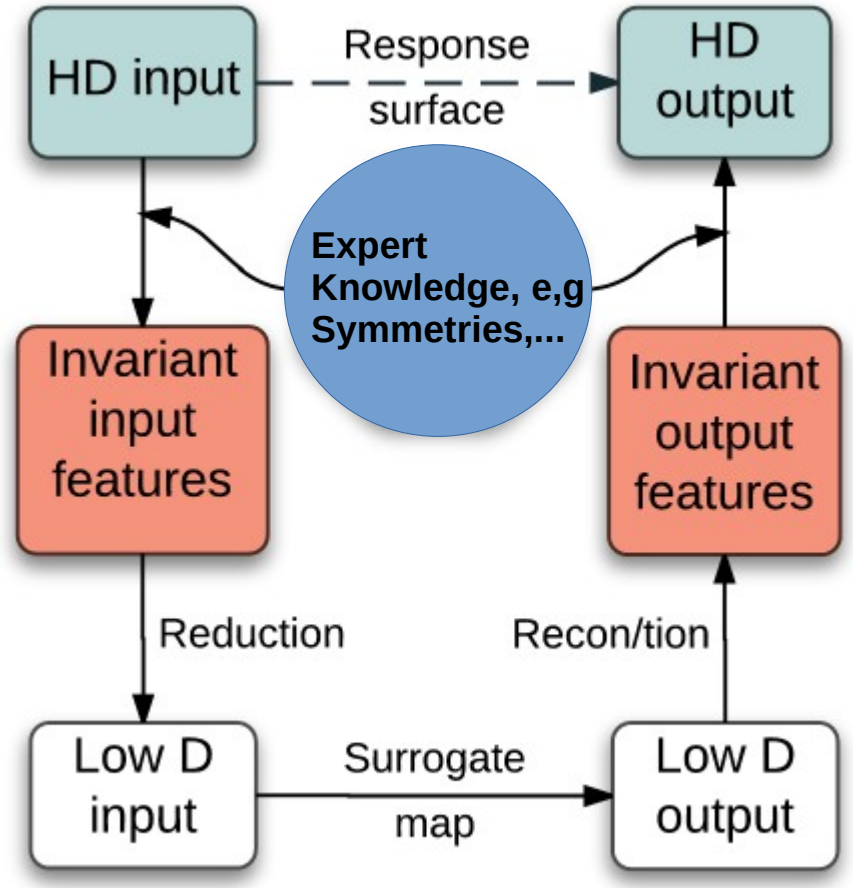


- The edges of the cube keep more and more volume away from the sphere (**cf. Sparse Grids!!!**).



Dealing with high dimensions

- Generic term approach:
- Exploit invariances and symmetries Induced by knowledge about the economics problem.
- Discover and exploit (non)-linear manifolds over which the response exhibits maximal variability.
- Here, we focus on:
  - HD input (already satisfying symmetries).
  - Discover and exploit linear manifold of maximal variability.

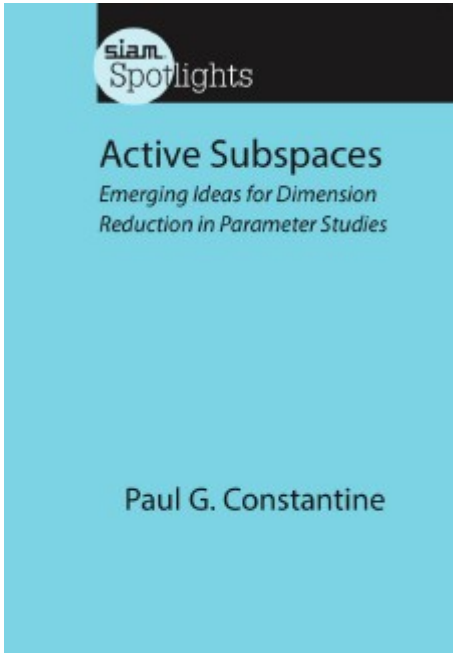


Abstract Objectives of Dim. Reduction

- data compression
- data visualization
- reduce the computational costs of the algorithms
- remove noise
- make the dataset easier to work with
- make the results easier to understand and interpret

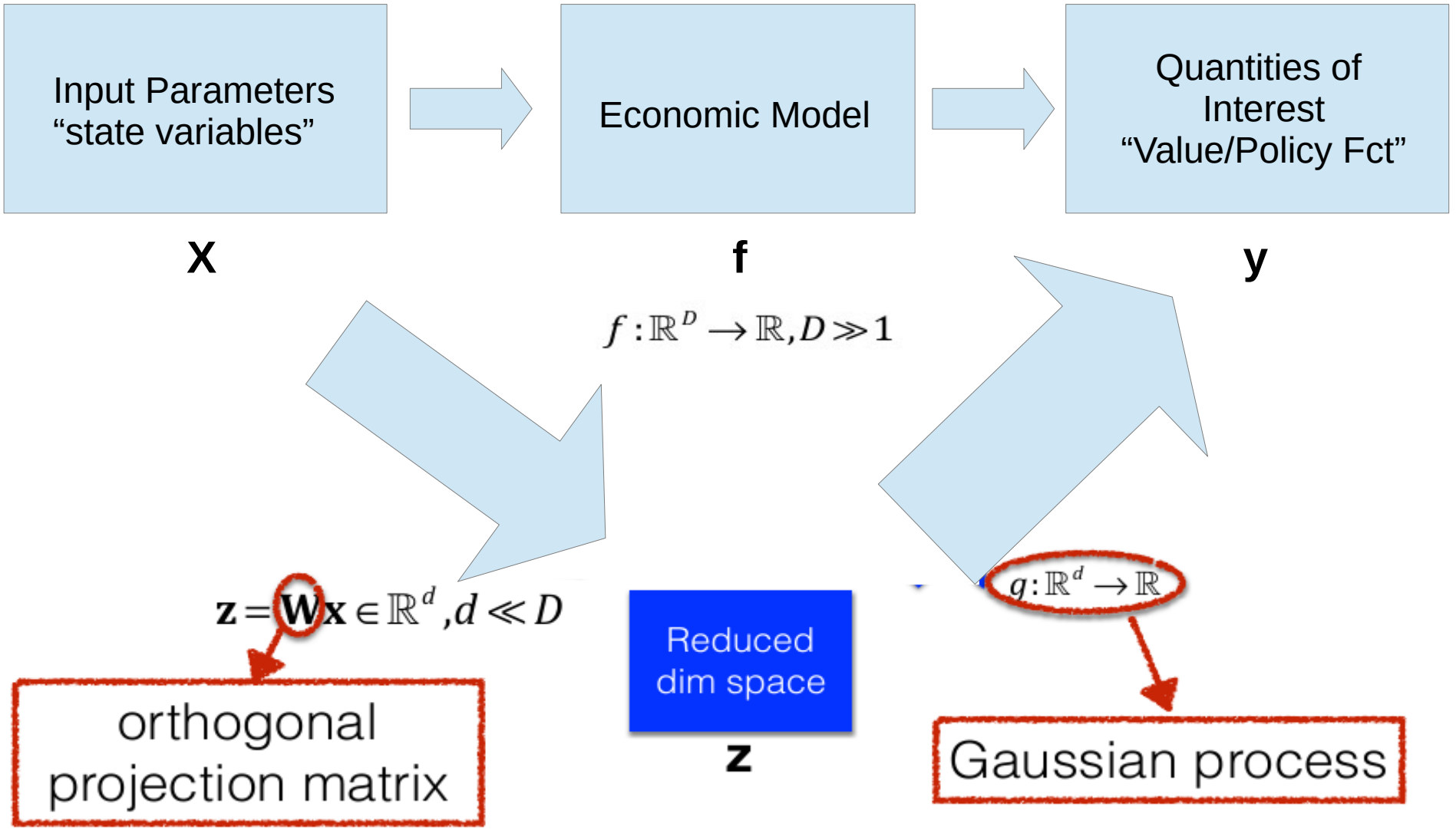
Active Subspaces

http://bookstore.siam.org/sl02/

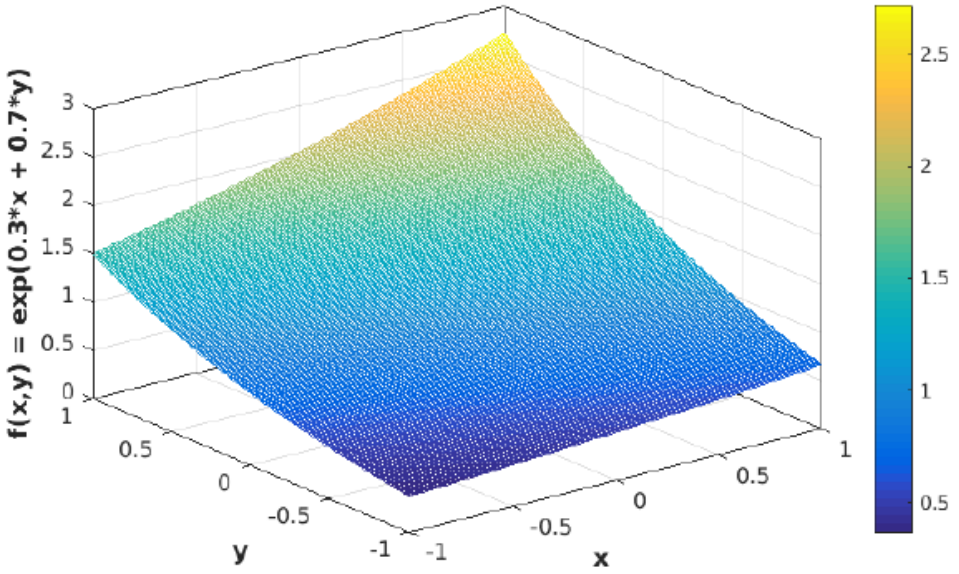


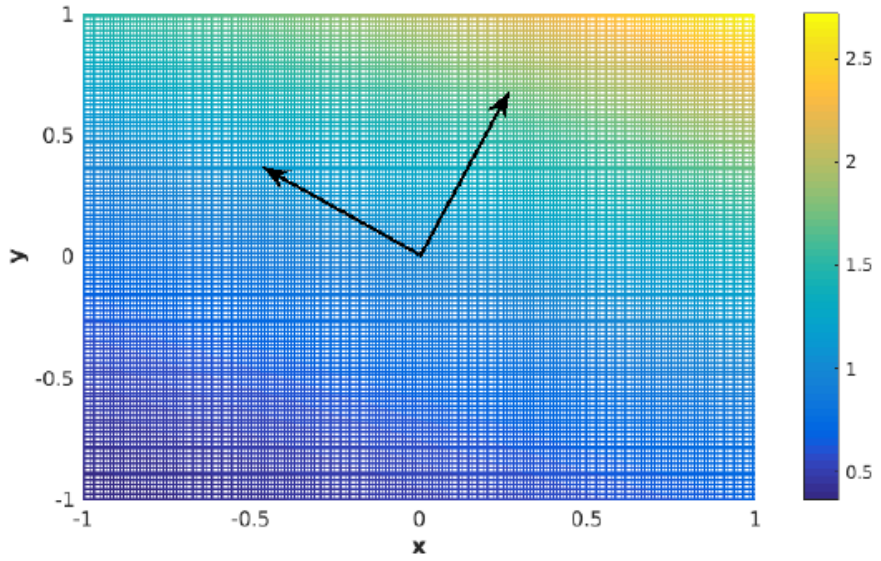
High Dimensions → Active Subspaces

(see, e.g., Constantine et. al (2013); Constantine (2015), with references therein)



Active Subspaces – intuitive example





Function varies most along [0.3, 0.7], and is constant in the orthogonal direction.

Discover active subspaces

(see, e.g., Constantine (2015), with references therein)

Step 1: Find **W**

$$\mathbf{C} = \mathbb{E}\left[\nabla_x f(\mathbf{x}) \nabla_x f(\mathbf{x})^T\right] \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_x f\left(\mathbf{x}^{(i)}\right) \nabla_x f\left(\mathbf{x}^{(i)}\right)^T$$

"Mean-square directional derivative"

Note: there are also derivative-free ways of constructing **W**.

$$\mathbf{C} = \mathbf{W} \Lambda \mathbf{W}^T$$

Gradient info

Partition the eigendecomposition

→ **Compute Eigenvalues, order them.**

→ **Look for "gaps.**

$$\Lambda = \begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix}, \qquad \mathbf{W} = \begin{bmatrix} \mathbf{W}_1 & \mathbf{W}_2 \end{bmatrix}, \qquad \mathbf{W}_1 \in \mathbb{R}^{m \times n}$$

Create a rotated coordinate system

$$\mathbf{x} = \mathbf{W}\mathbf{W}^T\mathbf{x} = \mathbf{W}_1\mathbf{W}_1^T\mathbf{x} + \mathbf{W}_2\mathbf{W}_2^T\mathbf{x} = \mathbf{W}_1\mathbf{q} + \mathbf{W}_2\mathbf{y}$$

GPs in active subspace

Step 2: Regression → $f(\mathbf{x}) \approx g(\boldsymbol{W}_1^T \mathbf{x})$.

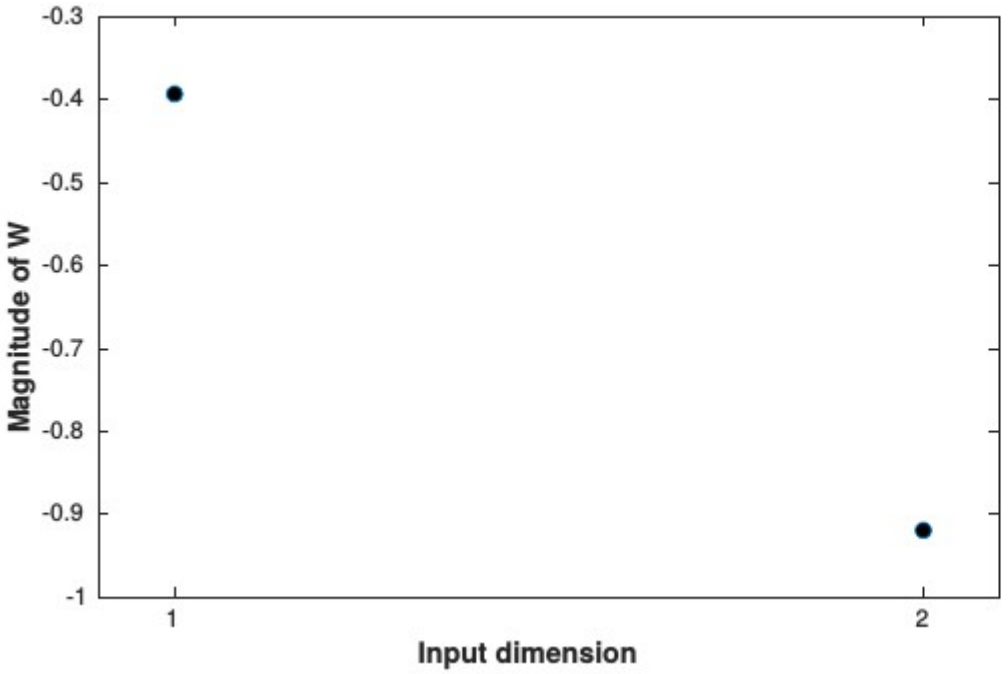
$$\mathcal{D}_{\text{projected}} = \left\{\left(\mathbf{z}^{(i)} = \mathbf{W}\mathbf{x}^{(i)}, y^{(i)} = f(\mathbf{x}^{(i)})\right)\right\}_{i=1}^{N}$$

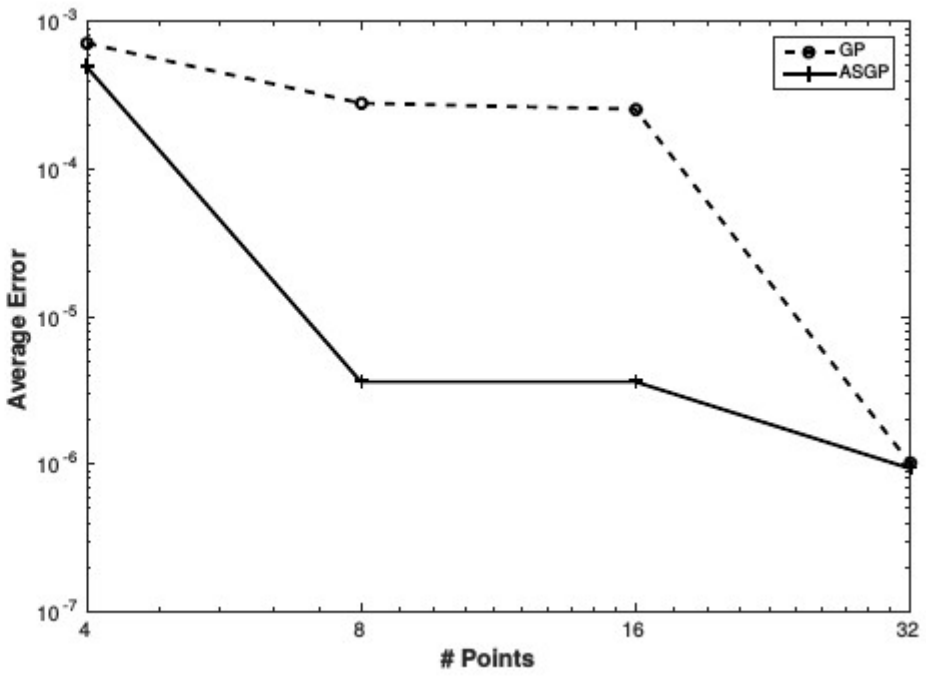
$$g(\cdot) \sim \text{GP}\left(g(\cdot) \mid m(\cdot), k_0(\cdot,\cdot)\right).$$

$$g(\cdot) \mid \mathcal{D}_{\text{projected}} \sim \text{GP}\left(g(\cdot) \mid m^*(\cdot), k_0^*(\cdot,\cdot)\right)$$

Analytical example in 2d

cf. demo/AS_ex1.py





$$f(x, y) = \exp(0.3x + 0.7y)$$

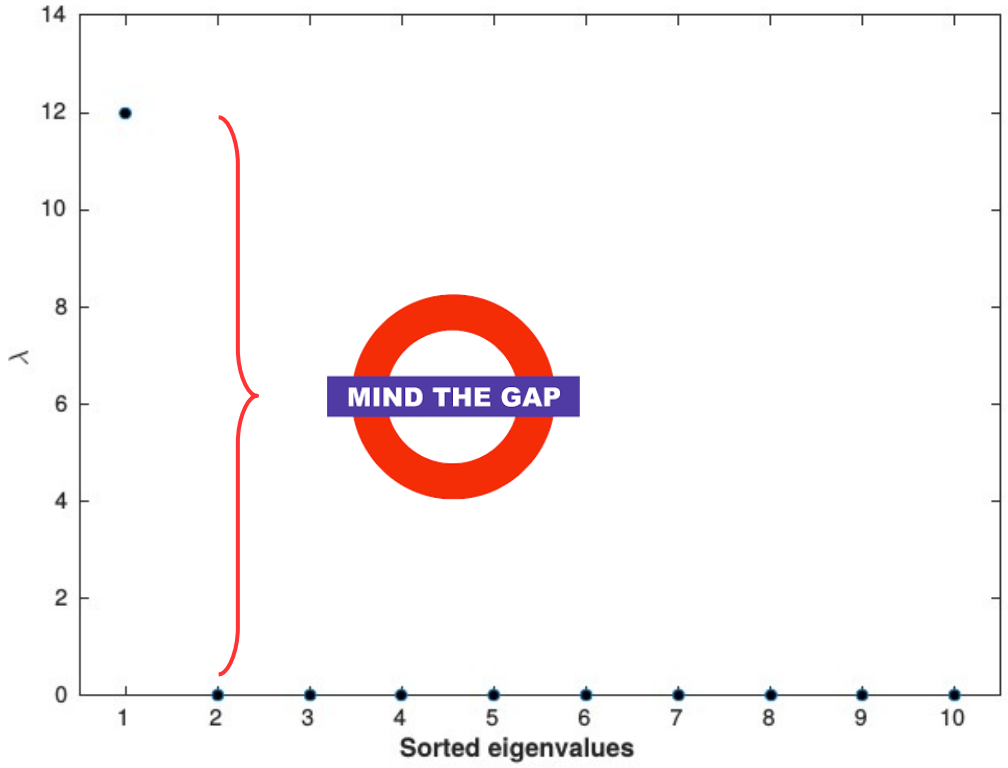
The left panel shows the projection matrix W of the 1-dimensional active subspace.
The right panel displays a comparison of the interpolation error for 2-dimensional GPs and an 1-dimensional AS of varying resolution, respectively.

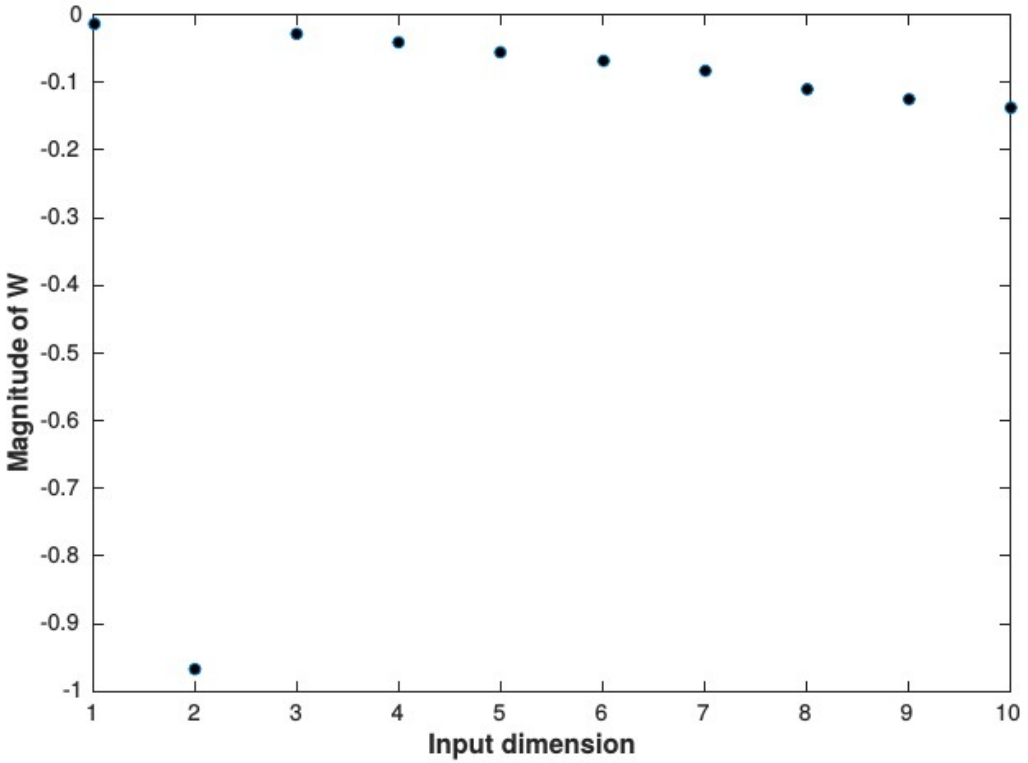
10*d* analytical example

cf. demo/AS_ex2.py

$$f : [-1, 1]^{10} \rightarrow \mathbb{R}$$

$$f(x_1, ..., x_{10}) = \exp(0.01x_1 + 0.7x_2 + 0.02x_3 + 0.03x_4 + 0.04x_5 + 0.05x_6 + 0.06x_7 + 0.08x_8 + 0.09x_9 + 0.1x_{10})$$





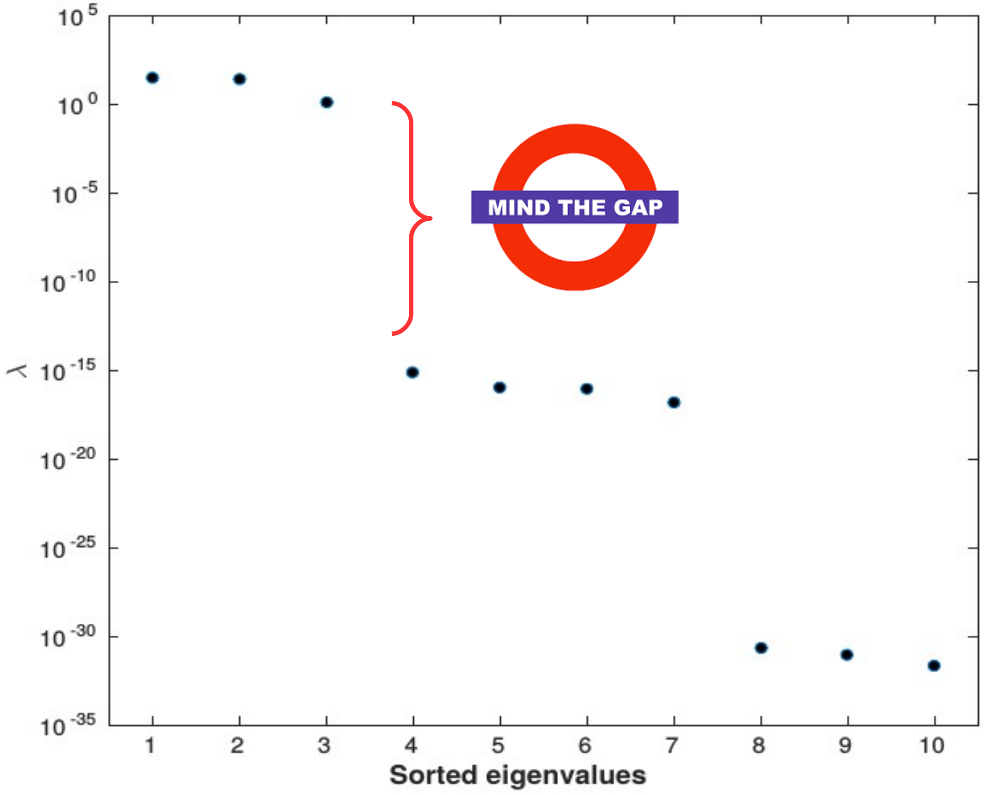
10*d* analytical example: Gap

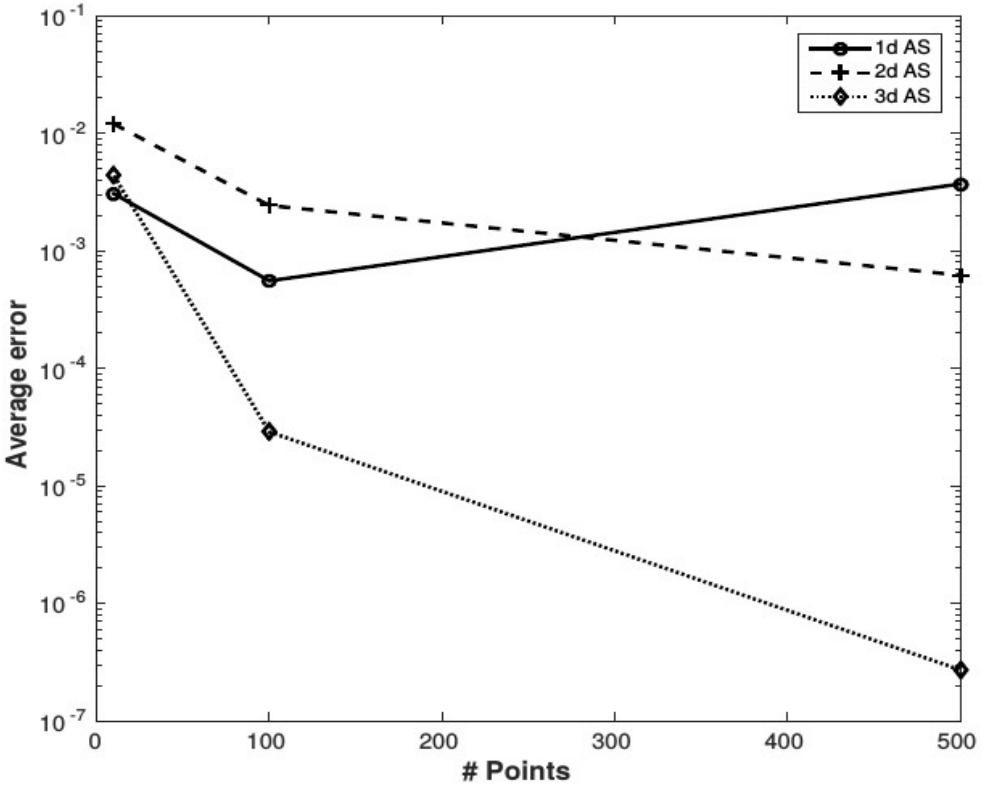
cf. demo/code/AS_ex3.py

$$f : [-1, 1]^{10} \to \mathbb{R}$$

$$f(x_1, ..., x_{10}) = x_2 \cdot x_3 \cdot \exp(0.01x_1 + 0.7x_2 + 0.02x_3 + 0.03x_4 + 0.04x_5 + 0.05x_6 + 0.06x_7 + 0.08x_8 + 0.09x_9 + 0.1x_{10}).$$

ASGP surrogates of dimension $d = \{1, 2, 3\}$





Action Required

- Implement the function

$$f(x,y) = x*\sin(y)$$

- x from [0,1]
- y from [0,1]

- Use the Active subspace method.

- What are the Eigenvalues?
- How does the simplified approximation perform?

Active Subspaces versus PCA

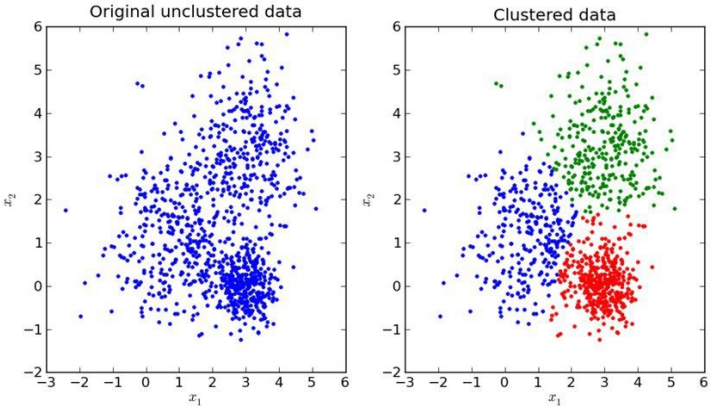
- Some comment on the similarities and didiscrepancies between PCA and AS:
- PCA identifies a projection of the input space.
- The goal of this projection, however, is not the same as in AS.
- PCA picks the projection that minimizes the mean square reconstruction error of the input **x** that is, it minimizes **$E[||x - W(W^T x)||]$**, where the expectation is with respect to the input-generating distribution.
- AS has an objective that is very different to that of PCA:
  AS focuses on finding a W that allows us to approximate f (**x**) with a function of the form h(**Wx**) as well as possible.
- Even though AS has not (yet) been formulated to directly minimize the mean square error **$E[(f(x) - h(Wx))^2]$**, it was shown by Constantine that the mean square error is bounded by a term proportional to the sum of the neglected eigenvalues of C .

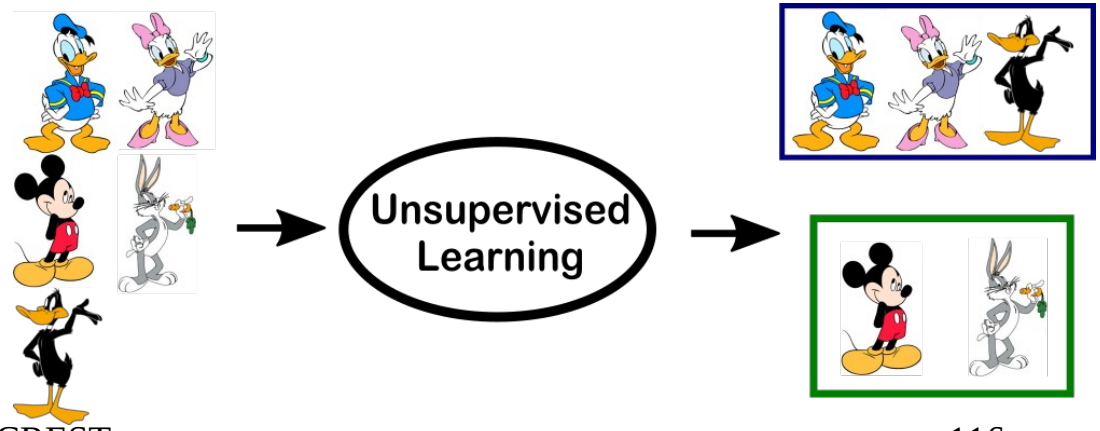
→ PCA focuses on finding the best linear projection that allows the reconstruction of the input, whereas **AS focuses on the search for the best linear projection that enables the reconstruction of the response surface f (x)**.

Gaussian Mixture Models (GMM)

Recall: Unsupervised Machine Learning

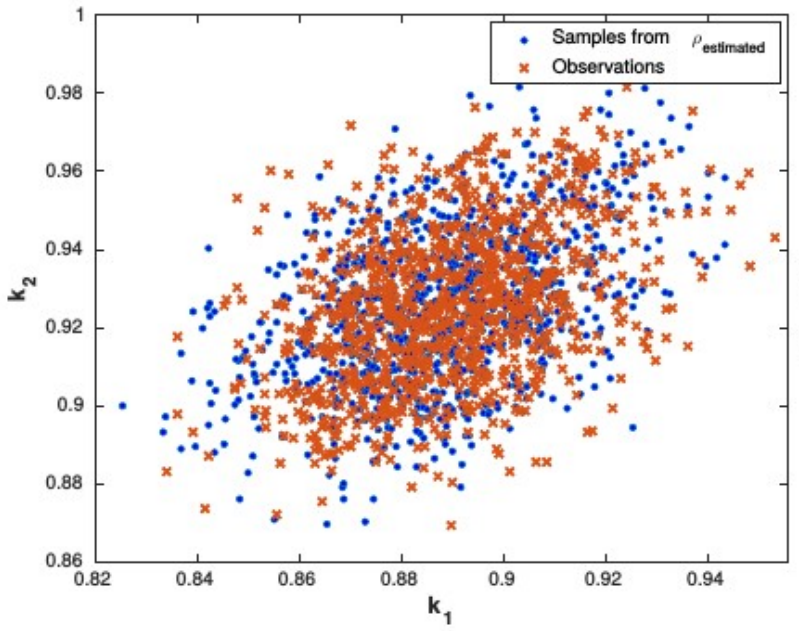
- Learning “what normally happens”.
- No output.
- Clustering: Grouping similar instances.
- Example applications:
  - Customer segmentation.
  - Image compression: Color quantization.
  - Bioinformatics: Learning motifs.

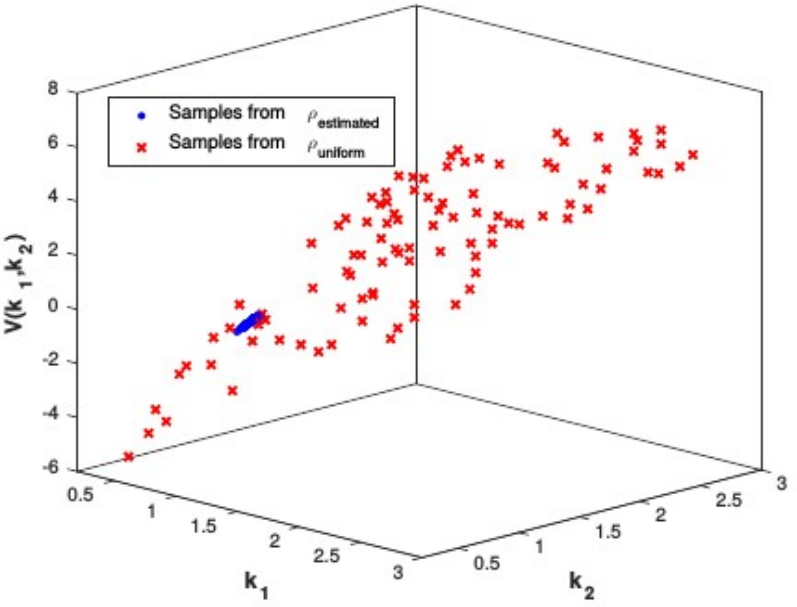




Motivation – Ergodic Sets

Den Haan and Marcet (1994), Judd et al. (2010, and Maliar and Maliar (2015), Scheidegger & Bilionis (2017)





$$V(\mathbf{k}) = \max_{\mathbf{I},\mathbf{c},\mathbf{l}} \left(u(\mathbf{c},\mathbf{l}) + \beta \mathbb{E}\left\{ V_{next}(\mathbf{k}^+) \right\} \right),$$

$$s.t.$$

$$k_j^+ = (1-\delta)\cdot k_j + I_j + \varepsilon_j, \quad j = 1,...,D$$

$$\Gamma_j = \frac{\zeta}{2} k_j \left(\frac{I_j}{k_j} - \delta \right)^2, \quad j = 1,...,D$$

$$\sum_{j=1}^{D} \left\{ c_j + I_j - \delta \cdot k_j \right\} = \sum_{j=1}^{D} \left\{ f(k_j, l_j) - \Gamma_j \right\},$$

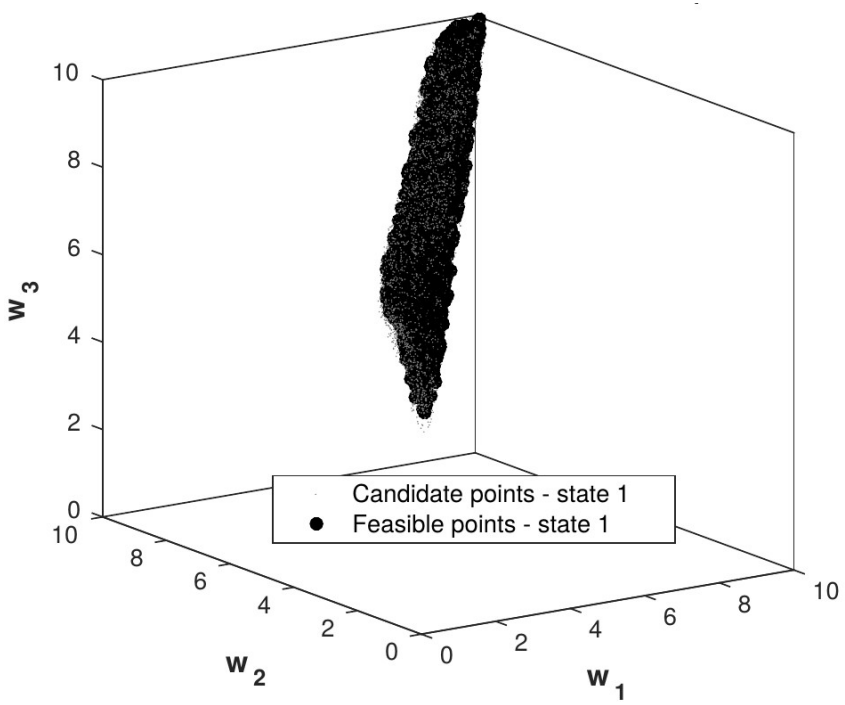
→ Simulate the Model

Motivation – Feasible Sets

see, e.g., Abreu et al. (1986/1990), Fernandes & Phelan (2000)

See, e.g., dynamic incentive problems:

- Domain of interest is high-dimensional.
- Irregularly-shaped.
- You need to approximate Value- and Policy functions on such a domain.
- In the multi-d setting, if you use grid-based methods, you spend way more than 99% of your resources in vain.



→ Q: **How can we represent such sets?**

→ Q: **How can we generate "observations" from within such sets?**

→ One way to do so are **Mixture of Gaussians** (see, e.g., Bishop (2006)).

→ Approximate the set in a probabilistic fashion.

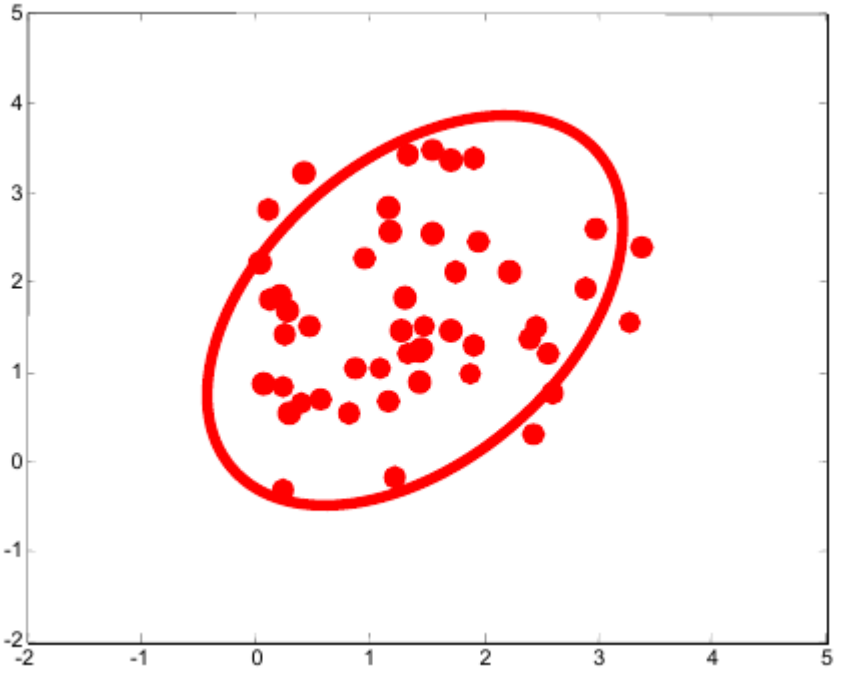
Recall Multivariate Gaussians

$$\mathcal{N}(x\ ;\ \mu, \Sigma) = \frac{1}{(2\pi)^{d/2}} |\Sigma|^{-1/2} \exp\left\{ -\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu) \right\}$$

Maximum Likelihood estimates given by:

$$\hat{\mu} = \frac{1}{N} \sum_i x^{(i)}$$

$$\hat{\Sigma} = \frac{1}{N} \sum_i (x^{(i)} - \hat{\mu})^T (x^{(i)} - \hat{\mu})$$



Mixture of Gaussians – the basic idea

Consult e.g. the books by Bishop (2006) or Murphy (2012) for more details.

- Figure: plots of the 'old faithful' data.

- The blue curves show contours of constant probability density.

- On the left is a single Gaussian distribution which has been fitted to the data using maximum likelihood.

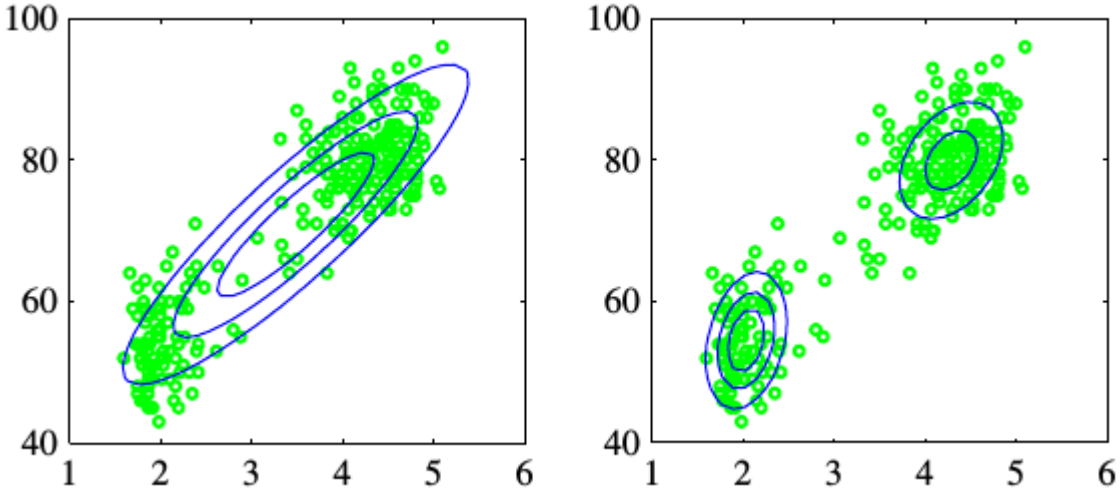
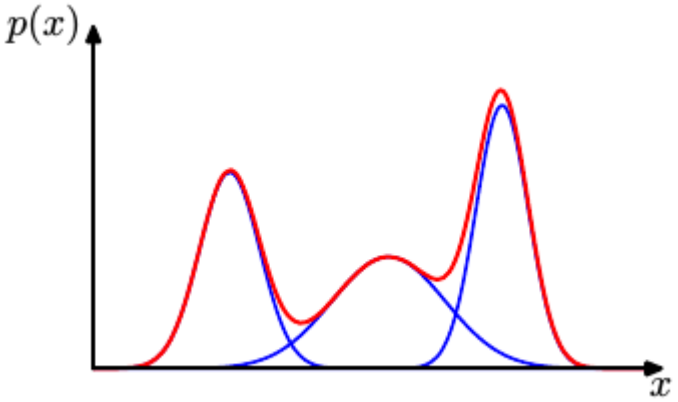


Fig. From Bishop (2006)

→ Note that this distribution fails to capture the two clumps in the data and indeed places much of its probability mass in the central region between the clumps where the data are relatively sparse.

- On the right the distribution is given by a linear combination of two Gaussians which has been fitted to the data, and which gives a better representation of the data.

GMM basics



- Example of a Gaussian mixture distribution p(x) in one dimension showing three Gaussians (each scaled by a coefficient) in blue and their sum in red.

→ Linear combinations of Gaussians can give rise to very complex densities.

→ By using a sufficient number of Gaussians and adjusting their means covariances as well as the coefficients in the linear combination, almost and density can be approximated with arbitrary accuracy.

→ k latent variables.

Superposition of Gaussians

- Formally, a GMM is: $$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \qquad \textbf{(A)}$$
- Each Gaussian density $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ is called a component of the mixture.
- It has its own mean μ_k and covariance Σ_k.
- π_k : mixing coefficients.

→ If we integrate both sides of **(A)** with respect to x, and note that both p(x) and the individual Gaussian components are normalized, one obtains:

$$\sum_{k=1}^{K} \pi_k = 1. \qquad 0 \leqslant \pi_k \leqslant 1.$$

$$p(\mathbf{x}) \geqslant 0 \quad \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \geqslant 0$$

Superposition of Gaussians (II)

- We therefore see that the mixing coefficients satisfy the requirements to be probabilities.

- From the sum and product rules, the marginal density is given by

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(k)p(\mathbf{x}|k) \quad \longleftrightarrow \quad \pi_k = p(k)$$

- This is equivalent say that $\pi_k = p(k)$ as the prior probability of picking the k-th component.

- The density $N(x|\mu_k, \Sigma_k) = p(x|k)$ can be viewed as the probability of x conditioned on k.

- One can think of p(x|k) as the pdf of cluster k (latent class labels).

Responsibilities: Posterior Distribution of Cluster Labels

- The posterior probabilities *p(k|x)* are also known as *responsibilities*

$$\begin{aligned}\gamma_k(\mathbf{x}) &\equiv p(k|\mathbf{x}) \\ &= \frac{p(k)p(\mathbf{x}|k)}{\sum_l p(l)p(\mathbf{x}|l)} \\ &= \frac{\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_l \pi_l \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_l, \boldsymbol{\Sigma}_l)}\end{aligned}$$

- This represents the posterior probability that point i belongs to cluster k.
- This is sometimes known as soft clustering.
- We can represent the amount of uncertainty in the cluster assignment by using $1 - \max_K \gamma_k(\mathbf{x})$.

The likelihood

- The form of the Gaussian mixture distribution is governed by the parameters **π, μ** and **Σ**, where we have used the notation **π** ≡ {π_1 , . . . , π_k }, **μ** ≡ {μ_1, . . . , μ_k}, **Σ** ≡ {Σ_1 , . . . Σ_k }.

- One way to set the values of these parameters is to use **maximum likelihood**.

- The log of the likelihood function is given by

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}$$

where X = {x_1, . . . , x_N}.

Problems with optimizing the likelihood

- The situation is now much more complex than with a single Gaussian, due to the presence of the summation over k inside the logarithm.
- As a result, the maximum likelihood solution for the parameters no longer has a closed-form analytical solution.
- One approach to maximizing the likelihood function is to use iterative numerical optimization techniques.
- Gradient methods could be used but are painful to implement.

→ Non-convex optimization problem! (multiple optima possible)

Examplein 1d

Observations $x_1 \ldots x_n$

- K=2 Gaussians with unknown μ, σ^2
- Estimation trivial if we know the source of each observation

$$\mu_b = \frac{x_1 + x_2 + \ldots + x_{n_b}}{n_b}$$

$$\sigma_b^2 = \frac{(x_1 - \mu_1)^2 + \ldots + (x_n - \mu_n)^2}{n_b}$$

Example: Expectation Maximization in 1d

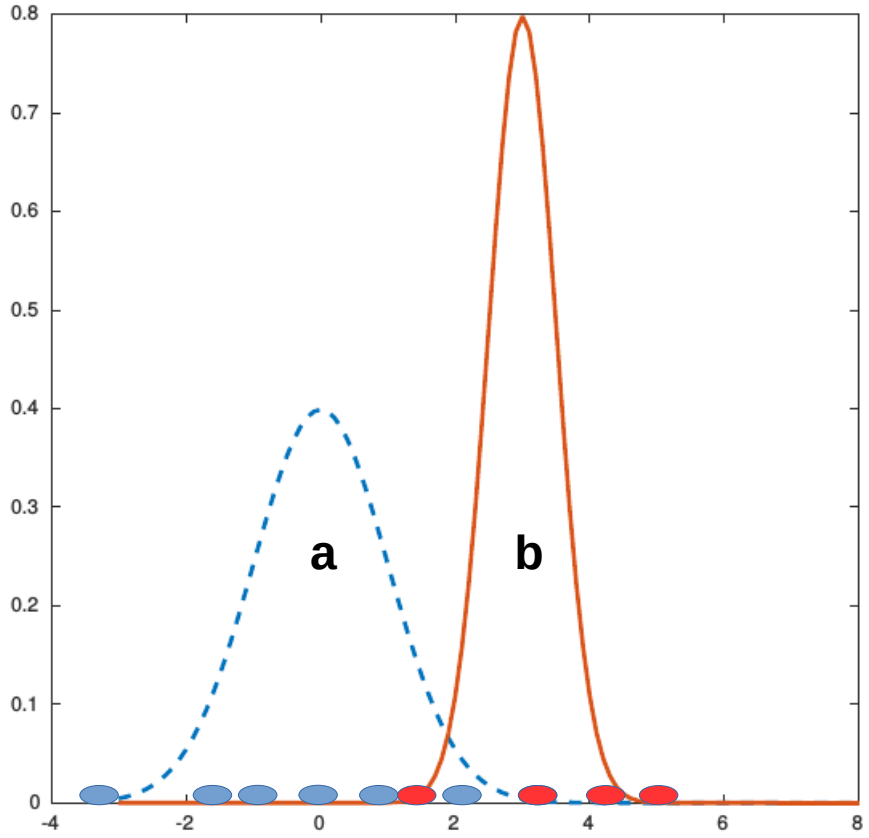
see, e..g., Dempster et al. (1977)

Observations $x_1 \ldots x_n$

- K=2 Gaussians with unknown μ, σ^2
- Estimation trivial if we know the source of each observation

$$\mu_b = \frac{x_1 + x_2 + \ldots + x_{n_b}}{n_b}$$

$$\sigma_b^2 = \frac{(x_1 - \mu_1)^2 + \ldots + (x_n - \mu_n)^2}{n_b}$$



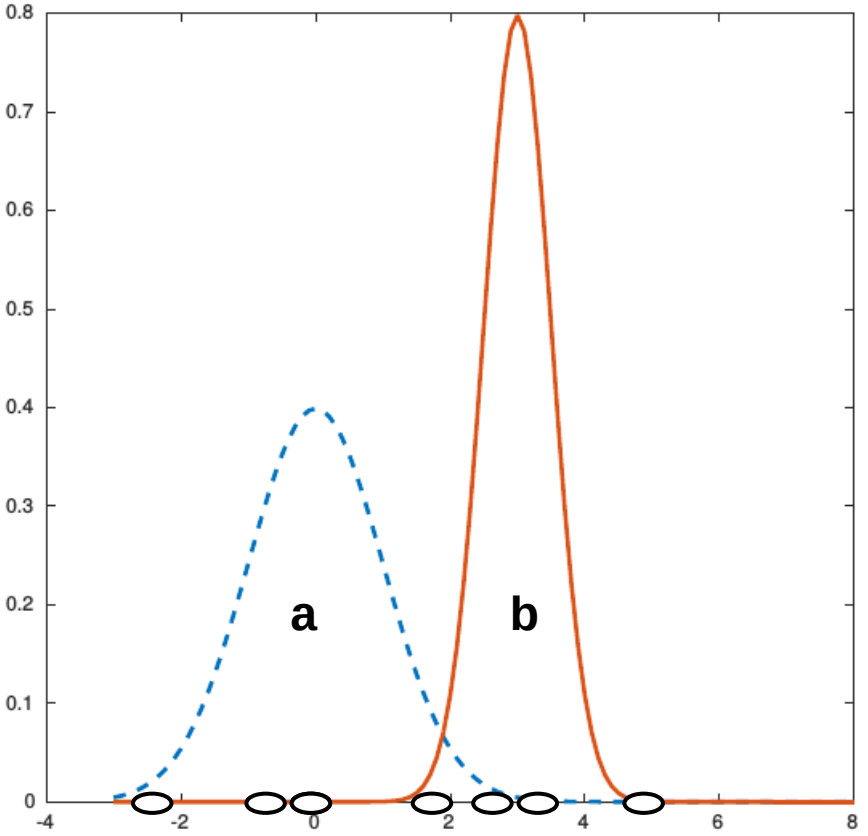
Example: Expectation Maximization in 1d (II)

- What if we don’t know the source?
- If we knew parameters of the Gaussians (μ, σ^2)

Example: Expectation Maximization in 1d (II)

- What if we don't know the source?
- If we knew parameters of the Gaussians (μ, σ²)

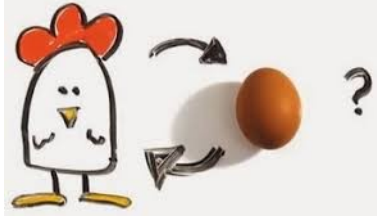
→ can guess whether point is more likely to be a or b.



$$P(b \mid x_i) = \frac{P(x_i \mid b)P(b)}{P(x_i \mid b)P(b) + P(x_i \mid a)P(a)}$$

$$P(x_i \mid b) = \frac{1}{\sqrt{2\pi\sigma_b^2}} \exp\left(-\frac{(x_i - \mu_b)^2}{2\sigma_b^2} \right)$$

EM Algorithm (in 1d)



see, e..g., Dempster et al. (1977), Bishop (2006), Murphy (2012) and references therein for details.

A fundamental problem:

→ we need (μ_a, σ_a^2) and (μ_b,σ_b^2) to guess the source of the points.

→ we need to know the source to estimate (μ_a,σ_a^2) and (μ_b,σ_b^2).

EM algorithm

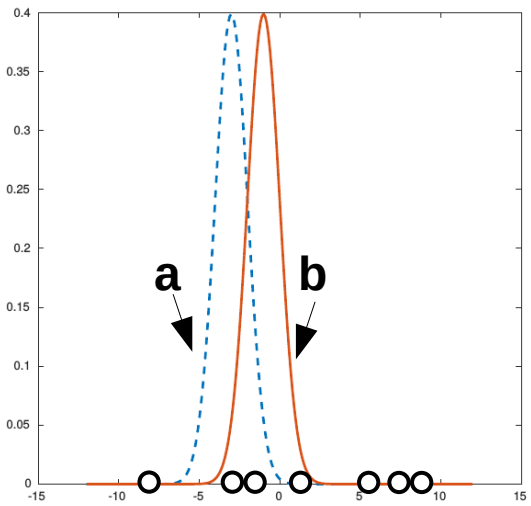
1. **start** with two randomly placed Gaussians (μ_a, σ_a^2) and (μ_b,σ_b^2).

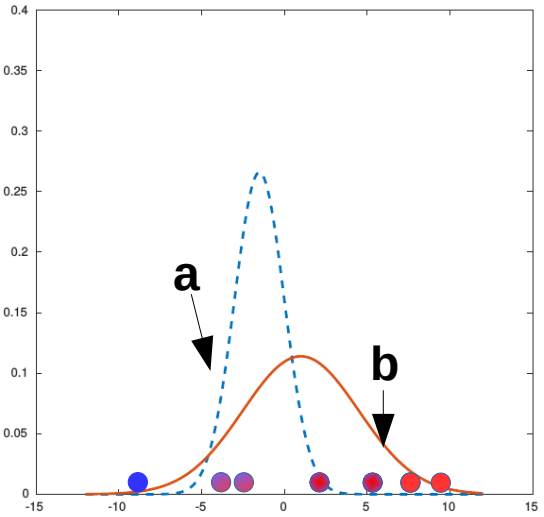
2. **E(xpectation)-step**:
   - for each point: $P(b|x_i)$ = does it look like it came from b?

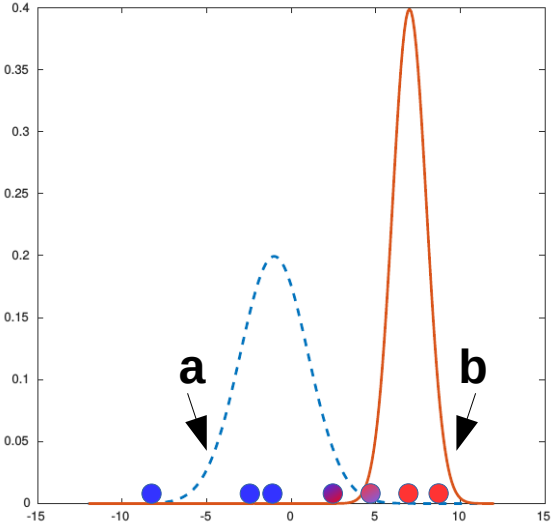
3. **M(maximization)-step**:
   - adjust (μ_a,σ_a^2) and (μ_b,σ_b^2) to fit points assigned to them.

4. **Iterate until convergence.**

EM in 1d







$$P(x_i \mid b) = \frac{1}{\sqrt{2\pi\sigma_b^2}} \exp\left(-\frac{(x_i - \mu_b)^2}{2\sigma_b^2}\right)$$

$$b_i = P(b \mid x_i) = \frac{P(x_i \mid b)P(b)}{P(x_i \mid b)P(b) + P(x_i \mid a)P(a)}$$

$$a_i = P(a \mid x_i) = 1 - b_i$$

$$\mu_b = \frac{b_1 x_1 + b_2 x_2 + \ldots + b_n x_{n_b}}{b_1 + b_2 + \ldots + b_n}$$

$$\sigma_b^2 = \frac{b_1 (x_1 - \mu_1)^2 + \ldots + b_n (x_n - \mu_n)^2}{b_1 + b_2 + \ldots + b_n}$$

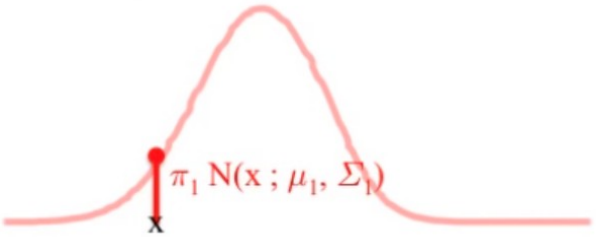
$$\mu_a = \frac{a_1 x_1 + a_2 x_2 + \ldots + a_n x_{n_b}}{a_1 + a_2 + \ldots + a_n}$$

$$\sigma_a^2 = \frac{a_1 (x_1 - \mu_1)^2 + \ldots + a_n (x_n - \mu_n)^2}{a_1 + a_2 + \ldots + a_n}$$

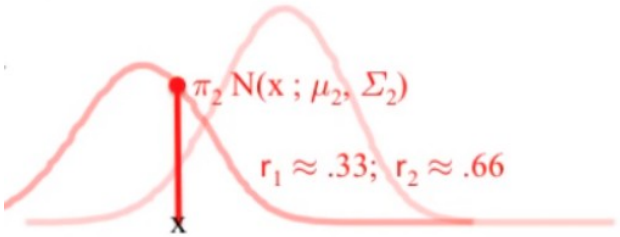
→ We could also estimate priors:
P(b) = (b_1 + b_2 + ... b_n) / n
P(a) = 1 – P(b)

EM in the multidimensional case

- Start with parameters describing each cluster
- Mean μ_c, Covariance Σ_c, "size" π_c
- **E-step ("Expectation"):**
  - For **each observation/point x_i**
  - Compute "r_{ic}", the probability that it belongs to cluster c.
    - Compute its probability under model c.
    - Normalize to sum to one (over clusters c).



$$r_{ic} = \frac{\pi_c \mathcal{N}(x_i \; ; \; \mu_c, \Sigma_c)}{\sum_{c'} \pi_{c'} \mathcal{N}(x_i \; ; \; \mu_{c'}, \Sigma_{c'})}$$



- If x_i is very likely under the c-th Gaussian, it gets high weight.
- Denominator just makes r's sum to one.

EM in the multidimensional case

- **M-step ("Maximization step"):**
  - For each cluster (Gaussian) z=c
  - Update its parameters using the (weighted) data points

$$N_c = \sum_i r_{ic}$$

Total responsibility allocated to cluster c

$$\pi_c = \frac{N_c}{N}$$

Fraction of total assigned to cluster c

$$\mu_c = \frac{1}{N_c} \sum_i r_{ic} x_i$$

Weighted mean of assigned data

$$\Sigma_c = \frac{1}{N_c} \sum_i r_{ic} (x_i - \mu_c)^T (x_i - \mu_c)$$

Weighted covariance of assigned data (use new weighted means here)

Expectation-Maximization: Summary

- Likelihood of the data

$$P(x_1, ..., x_N) = \Pi_{i=1}^{N} \sum_{k=1}^{K} P(x_i|k)P(k)$$

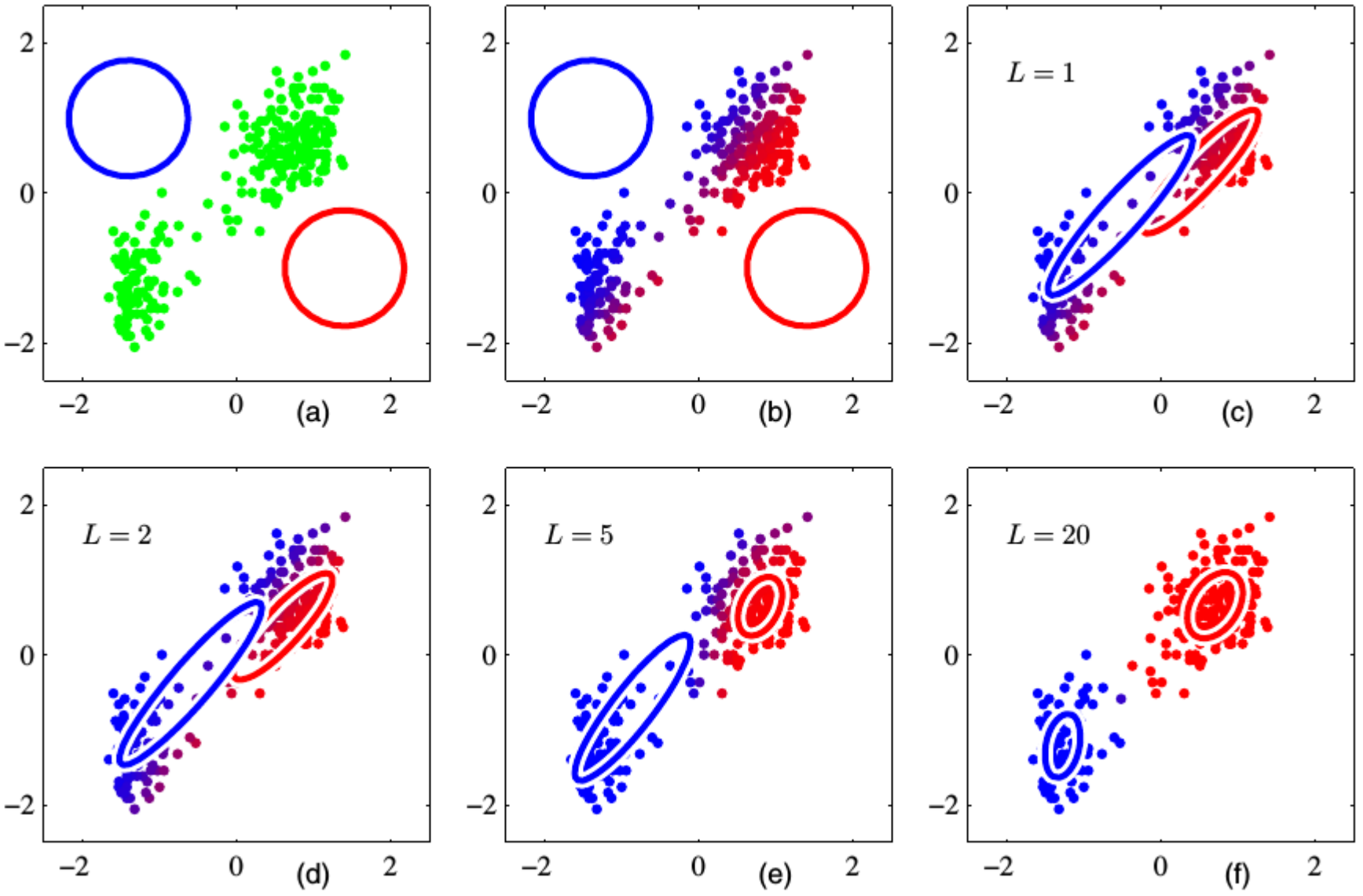
- Each step increases the log-likelihood of our model

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}$$

- Iterate until convergence
  - Convergence guaranteed – another ascent method.
- Cannot discover k.

Gaussian mixture models:d>1

See Bishop (2006) for details



Bayesian Information Criterion (BIC)

See, e.g., Alpaydın (2014), MIT Press

- How to pick k?
- Probabilistic model: L = $\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}$
  - Tries to “fit” the data (maximize likelihood)
- Choose k that makes L as large as possible?
  - K = n: each data point has its own “source”
  - may not work well for new data points
- Split points into training set **T** and validation set **V**
  - for each k: fit parameters of **T**
  - measure likelihood of **V**
  - sometimes still best when k = n
- “Occam’s razor”:
  - Pick the “simplest” of all models that fits the data.
  - Assess, e.g., via Bayes Information Criterion (BIC): **$\max_p$ { *L* – 1/2 p*log(n) }**
  - *L*: Likelihood; p: # Parameters in the model – how simple is the model.

About the EM Algorithm

Some good things about EM:

- no learning rate (step-size) parameter.
- automatically enforces parameter constraints.
- very fast for low dimensions.
- each iteration guaranteed to improve likelihood.

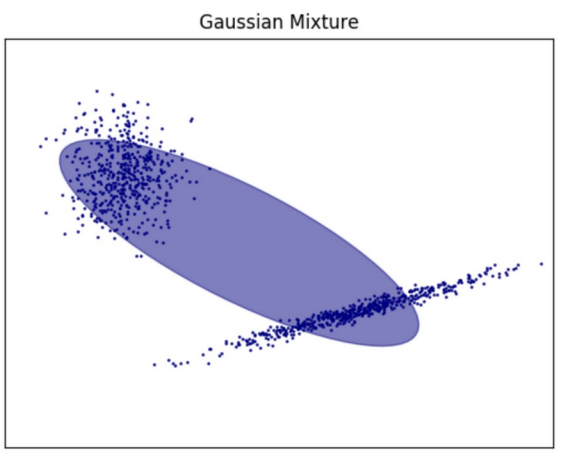
Some bad things about EM:

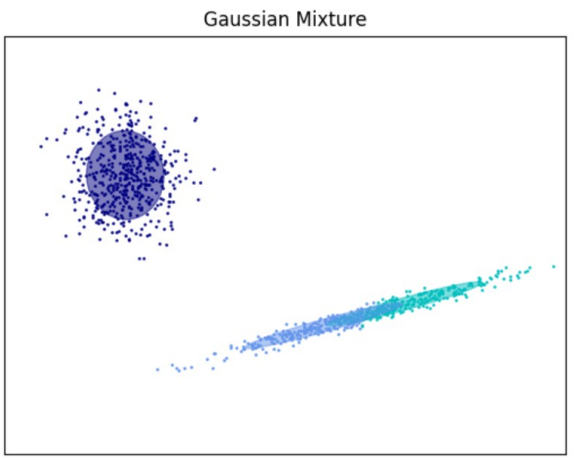
- can get stuck in local minima.
- can be slower than conjugate gradient (especially near convergence).
- requires expensive inference step.
- is a maximum likelihood/MAP (maximum a posterior) method.

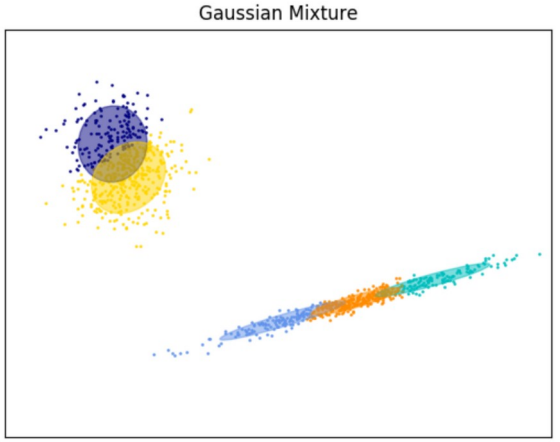
Hands-on example 1

https://scikit-learn.org/stable/modules/mixture.html
code/GMM_scikit_example.py

- Plot the confidence ellipsoids of a mixture of two Gaussians obtained with Expectation Maximization (GaussianMixture class)
- The model has access to 1, 3, and 5 components with which to fit the data. Note that the Expectation Maximization model will necessarily use ALL components
- In the 5-component example, we can see that the Expectation Maximization model splits some components arbitrarily, because it is trying to fit too many components.





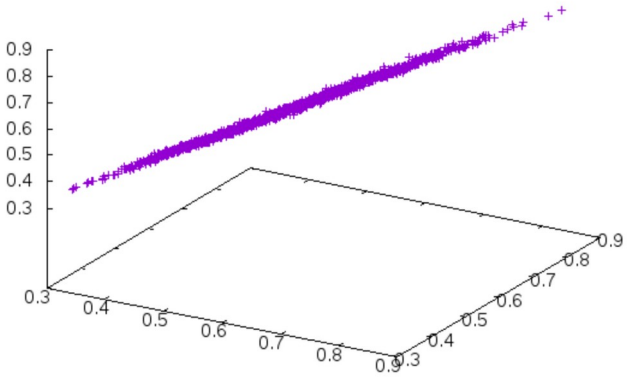


N

Hands-on example 2

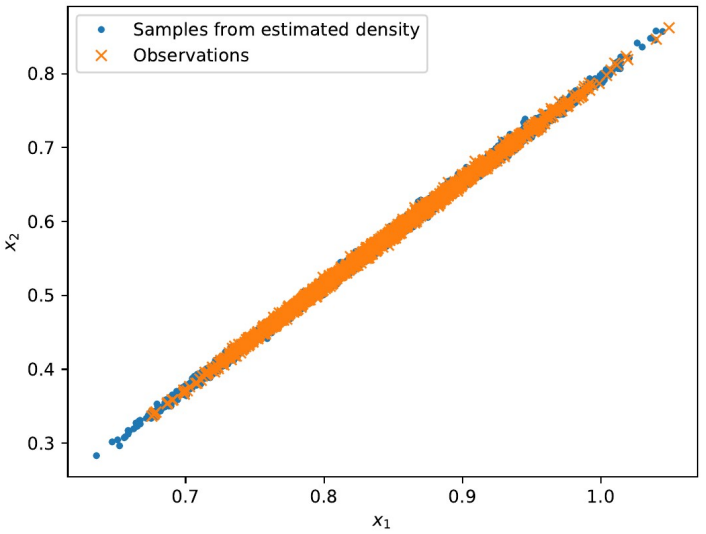
code/code/BGMM_data
cf. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3282487

- We simulate a bunch of data (e.g., an ergodic set).
  - Its in a text file (**ergodic_data.txt – 3 dimensions**)



- We apply GMM (**build_density.py**)

- We can sample data from the fitted GMM model (**sample.py**)



Bayesian Active Learning

- With the recent explosion of available data, you have can have millions of examples with a high cost to obtain labels.
- For instance, when trying to predict the sentiment of tweets, obtaining a training set can require immense manual labor
- But worry not, active learning comes to the rescue!
- In general, AL is a framework allowing you to increase classification performance by intelligently querying you to label the most informative instances.

Bayesian Active Learning

- With the recent explosion of available data, you have can have millions of examples with a high cost to obtain labels.
- For instance, when trying to predict the sentiment of tweets, obtaining a training set can require immense manual labour.
- But worry not, active learning comes to the rescue!
- In general, active learning is a framework allowing you to increase classification performance by intelligently querying you to label the most informative instances.
- code/BAL.ipynb

Reinforcement Learning

-Computing globally accurate optimal policies is a challenging task.

- Thus far, we have placed the observation points to train the Gaussian processes randomly inside the relevant part of the state space (simplex)

- This strategy can be highly inefficient

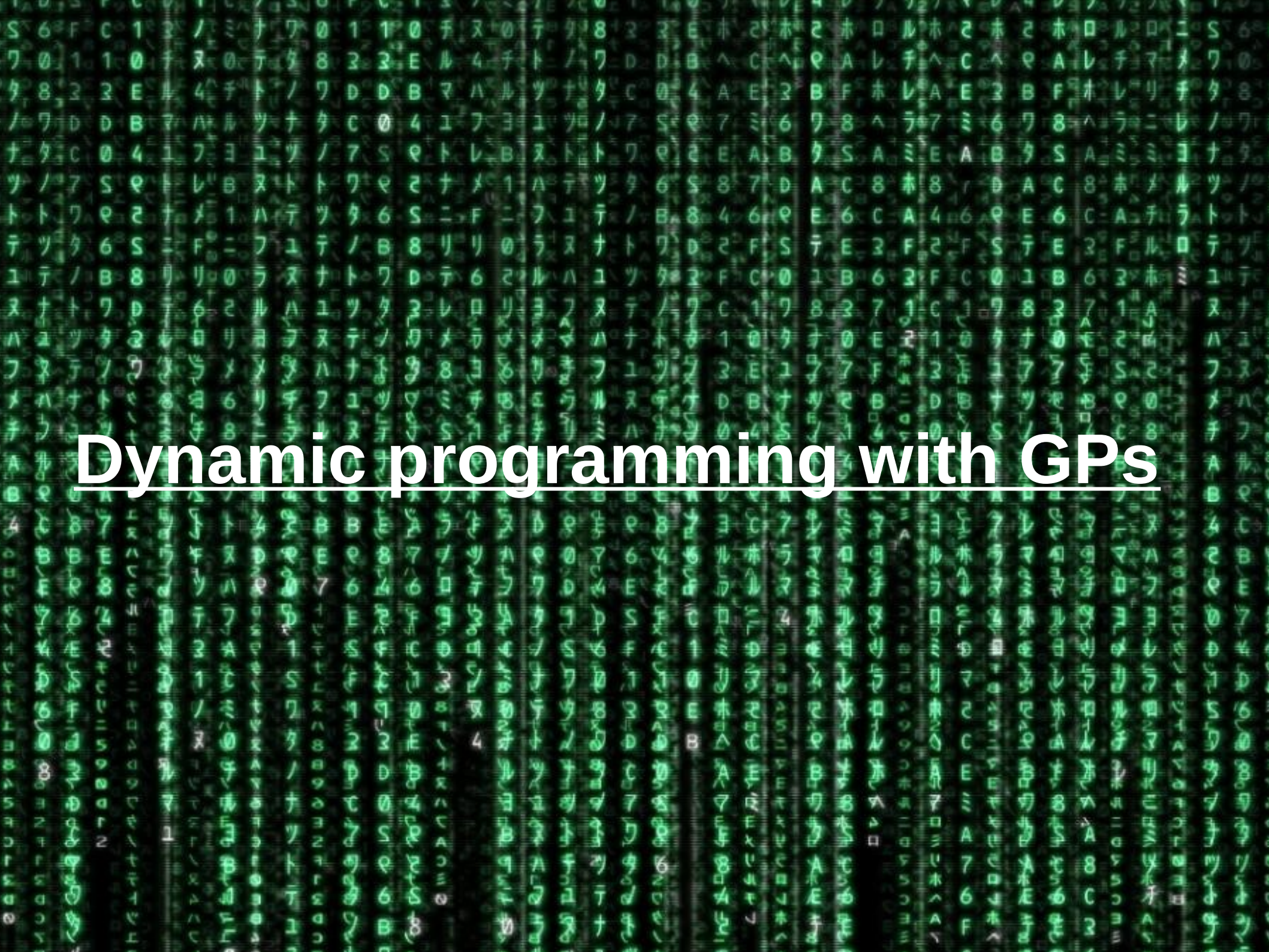
→ **Bayesian Active Learning** (see, e.g., Deisenroth et al. (2009))

- technique from the reinforcement learning to automatically place observations in regions of the state space where they improve most on the quality of the approximator.

- Score Function

$$U(\tilde{x}) = \sigma_m \mathbb{E}\left[V^\tau(\tilde{x})|\mathbf{X}\right] + \frac{\sigma_v}{2}\log\left(\mathrm{var}\left[V^\tau(\tilde{x})|\mathbf{X}\right]\right)$$

→ code/BAL.py


Dynamic programming with GPs

Recall: Infinite-Horizon Dynamic Programming

e.g. Stokey, Lucas & Prescott (1989), Judd (1998), ...

Want to choose an infinite sequence of "controls" $\{u_s\}_{s=0}^{\infty}$ to maximize

$$\sum_{t=0}^{\infty} \beta^t r\left(x_t, u_t\right) \qquad \text{s.t.} \qquad x_{t+1} = g(x_t, u_t) \qquad \beta \in (0,1)$$

(Discrete time) Dynamic programming seeks a **time-invariant policy function *h*** mapping the state x_t into the control u_t, such that the sequence $\{u_s\}_{s=0}^{\infty}$ generated by iterating

$$u_t = h\left(x_t\right)$$

$$x_{t+1} = g\left(x_t, u_t\right)$$

starting from an initial condition solves the original problem.

r in the economic context: often a so-called `utility function'.
r concave: reflects the notion "more is better"; marginal benefit tends to zero.

Recall: Infinite-Horizon Dynamic Programming

To find the policy function *h*, we need to know another function (`**Value Function**') that expresses optimal value of the original problem

$$V(x_0) = \max_{\{u_s\}_{s=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t r(x_t, u_t)$$

→ Task: solve jointly for $V(x), h(x)$ that are linked by the **Bellman equation**

$$V(x) = \max_u \{r(x,u) + \beta V[g(x,u)]\} \qquad \text{(A)}$$

→ The maximizer of (A) is a policy function *h(x)* that satisfies

$$V(x) = r[x, h(x)] + \beta V\{g[x, h(x)]\}$$

Value Function Iteration

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

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

s.t.

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

x: grid point, describes your system. State-space potentially **high-dimensional.**

`old solution': high-dimensional function, approximated by sparse grid Interpolation method on which we Interpolate.

Use-case for GPs

Example: Infinite-Horizon Stochastic DP

If uncertainty is present, previous Bellman equation can be re-written as

$$V(x) = \max_u \{ r(x,u) + \beta E[V[g(x,u,\epsilon)]|x] \}$$

s.t.

$$x_{t+1} = g(x_t, u_t, \epsilon_{t+1})$$

The solution is approached by iterations on

$$V_{j+1}(x) = \max_u \{ r(x,u) + \beta E[V_j[g(x,u,\epsilon)]|x] \}$$

Note: If we have discrete shocks, we may have to carry around multiple sparse grids that need to be updated!

Growth Model & Dynamic Programming & ASG

To demonstrate the capabilities of sparse grids, we consider an **infinite-horizon discrete-time multi-dimensional optimal growth model**
(see, e.g., Scheidegger & Bilionis (2017), and references therein).

The model has few parameters and is relatively easy to explain, whereas the dimensionality of the problem can be scaled up in a straightforward but meaningful way.

→ state-space depends linearly on the number of ***D* sectors** considered.

→ there are D sectors with **capital** $\mathbf{k}_t = (k_{t,1}, ..., k_{t,D})$

and elastic **labour supply** $\mathbf{l}_t = (l_{t,1}, ..., l_{t,D})$

Growth model

The production function of sector ***i*** at time ***t*** is $f(k_{t,i}, l_{t,i}),\ \text{for}\ i = 1, ..., D.$

Consumption: $\mathbf{c}_t = (c_{t,1}, ..., c_{t,D})$

Investment of the sectors at time *t:* $\mathbf{I}_t = (I_{t,1}, ..., I_{t,D})$

→ The goal now is to find **optimal consumption** and **labour supply decisions** such that expected total utility over an infinite time horizon is maximized.

Model

$$V_0(\mathbf{k}_0) = \max_{\mathbf{k}_t, \mathbf{I}_t, \mathbf{c}_t, \mathbf{l}_t, \mathbf{\Gamma}_t} \left\{ \sum_{t=0}^{\infty} \beta^t \cdot u(\mathbf{c}_t, \mathbf{l}_t) \right\},$$

$$s.t.$$

$$k_{t+1,j} = (1-\delta) \cdot k_{t,j} + I_{t,j} \qquad j = 1, ..., D$$

$$\Gamma_{t,j} = \frac{\zeta}{2} k_{t,j} \left(\frac{I_{t,j}}{k_{t,j}} - \delta \right)^2, \quad j = 1, ..., D$$

$$\sum_{j=1}^{D} (c_{t,j} + I_{t,j} - \delta \cdot k_{t,j}) = \sum_{j=1}^{D} (f(k_{t,j}, l_{t,j}) - \Gamma_{t,j})$$

Model (II)

Convex adjustment cost of sector *j*: $\mathbf{\Gamma}_t = (\Gamma_{t,1},,...,\Gamma_{t,D})$

Capital depreciation: δ

Discount factor: β

Recursive formulation

$$V(\mathbf{k}) = \max_{\mathbf{I},\mathbf{c},\mathbf{l}} \left(u(c,l) + \beta \quad \left\{ V_{next}(k^+) \right\} \right),$$

$$s.t.$$

$$k_j^+ = (1-\delta)\cdot k_j + I_j \qquad j = 1, ..., D$$

$$\Gamma_j = \frac{\zeta}{2} k_j \left(\frac{I_j}{k_j} - \delta \right)^2, \quad j = 1, ..., D$$

$$\sum_{j=1}^{D} (c_j + I_j - \delta \cdot k_j) = \sum_{j=1}^{D} (f(k_j, l_j) - \Gamma_j)$$

where we indicate the next period's variables with a superscript "+". $\mathbf{k} = (k_1, ..., k_D)$ represents the state vector, $\mathbf{l} = (l_1, ..., l_D)$, $\mathbf{c} = (c_1, ..., c_D)$, and $\mathbf{I} = (I_1, ..., I_D)$ are $3D$ control variables. $\mathbf{k}^+ = \left(k_1^+, ..., k_D^+\right)$ is the vector of next period's variables. Today's and tomorrow's states are restricted to the finite range $[\underline{\mathbf{k}}, \overline{\mathbf{k}}]^D$, where the lower edge of the computational domain is given by $\underline{\mathbf{k}} = (\underline{k_1}, ..., \underline{k_D})$, and the upper bound is given by $\overline{\mathbf{k}} = (\overline{k_1}, ..., \overline{k_D})$. Moreover, $\mathbf{c} > 0$ and $\mathbf{l} > 0$ holds component-wise.

Utility function etc.

Productivity: $$f(k_j, l_j) \quad = \quad A \cdot k_i^{\psi} \cdot l_i^{1-\psi}$$

Utility: $$u\left(\mathbf{c}, \mathbf{l}\right) = \sum_{i=1}^{d} \left[\frac{\left(c_i / A\right)^{1-\gamma} - 1}{1-\gamma} - (1-\psi) \frac{l_i^{1+\eta} - 1}{1+\eta} \right]$$

Terminal Value function: $$V^{\infty}\left(\mathbf{k}\right) = u\left(f(k, \mathbf{e}), \mathbf{e}\right) / (\mathbf{1} - \beta)$$

where $\mathbf{e}$ is the unit vector

Parametrization

Parameter	Value
β	0.8
δ	0.025
ζ	0.5
$\left[\underline{\mathbf{k}}, \overline{\mathbf{k}}\right]^D$	$[0.2, 3.0]^D$
ψ	0.36
A	$(1-\beta)/(\psi \cdot \beta)$
γ	2
η	1

Value function iteration

$$V(\mathbf{k}) = \max_{\mathbf{I},\mathbf{c},\mathbf{l}} \left(u(c,l) + \beta \quad \left\{ V_{next}(k^+) \right\} \right),$$

$$s.t.$$

$$k_j^+ = (1-\delta)\cdot k_j + I_j \qquad , \quad j = 1, ..., D$$

$$\Gamma_j = \frac{\zeta}{2} k_j \left(\frac{I_j}{k_j} - \delta \right)^2, \quad j = 1, ..., D$$

$$\sum_{j=1}^{D} (c_j + I_j - \delta \cdot k_j) = \sum_{j=1}^{D} (f(k_j, l_j) - \Gamma_j)$$

State k: sparse grid coordinates

V_{next} : sparse grid interpolator from the previous iteration step

Solve this optimization problem at every sample point.

Attention: Take care of the econ domain /sampling domain

Convergence measures (due to contraction mapping)

Average error: $$e^s = \frac{1}{N} \sum_{i=1}^{N} |V^s(\mathbf{x^i}) - V^{s-1}(\mathbf{x^i})|$$

Max. error: $$a^s = \max_{i=1,N} |V^s(\mathbf{x^i}) - V^{s-1}(\mathbf{x^i})|$$

The formal multi-*d* Growth Model

See, e.g. Cai & Judd (2014) and references therein

$$V(\mathbf{k}) = \max_{\mathbf{I},\mathbf{c},\mathbf{l}} \left(u(\mathbf{c},\mathbf{l}) + \beta \mathbb{E}\left\{ V_{next}(\mathbf{k}^+) \right\} \right),$$

$$s.t.$$

$$k_j^+ = (1-\delta)\cdot k_j + I_j + \epsilon_j, \quad j = 1,...,D$$

$$\Gamma_j = \frac{\zeta}{2} k_j \left(\frac{I_j}{k_j} - \delta \right)^2, \quad j = 1,...,D$$

$$\sum_{j=1}^{D} (c_j + I_j - \delta \cdot k_j) = \sum_{j=1}^{D} (f(k_j, l_j) - \Gamma_j),$$

Parameter	Value
β	0.8
δ	0.025
ζ	0.5
$[\underline{\mathbf{k}}, \overline{\mathbf{k}}]^D$	$[0.2, 3.0]^D$
ψ	0.36
A	$(1-\beta)/(\psi \cdot \beta)$
γ	2
η	1
σ	0.01
D	{1,...,500}

k: continuous states

$$u(c,l) = \sum_{i=1}^{d} \left[\frac{(c_i/A)^{1-\gamma} - 1}{1-\gamma} - (1-\psi) \frac{l_i^{1+\eta} - 1}{1+\eta} \right]$$

$$f(k_i, l_i) = A \cdot k_i^{\psi} \cdot l_i^{1-\psi}$$

$$V^{\infty}(\mathbf{k}) = u(f(k, \mathbf{e}, \mathbf{e}), \mathbf{e})/(\mathbf{1}-\beta)$$

$$\epsilon_{t,j} \sim \mathcal{N}(0, \sigma^2)$$

Value function iteration with GPR

Data: Initial guess V_{next} for the next period's value function. Approximation accuracy $\bar{\eta}$.

Result: The (approximate) equilibrium $3D$ policy functions $\xi^* = \{\xi_1^*, ..., \xi_{3D}^*\}$ and the corresponding value function V^*.

Set iteration step $s = 1$.

while $\eta > \bar{\eta}$ **do**

Generate n training inputs $\mathbf{X} = \{\mathbf{k_i^s} : \mathbf{1} \leq \mathbf{i} \leq \mathbf{n}\} \in [\underline{\mathbf{k}}, \bar{\mathbf{k}}]^{\mathbf{D}}$

for $\mathbf{k}_i^s \in \mathbf{X}$ **do**

Compute the maximization problem

$$V(\mathbf{k}_i^s) = \max_{\mathbf{I},\mathbf{c},\mathbf{l}} \left(u(\mathbf{c}, \mathbf{l}) + \beta \mathbb{E}\left\{ V_{next}(\mathbf{k}_i^+) \right\} \right), \text{ s.t.}$$

$$k_j^+ = (1 - \delta) \cdot k_j + I_j + \epsilon_j, \quad j = 1, ..., D$$

$$\Gamma_j = \frac{\zeta}{2} k_j \left(\frac{I_j}{k_j} - \delta \right)^2, \quad j = 1, ..., D$$

$$\sum_{j=1}^{D} (c_j + I_j - \delta \cdot k_j) = \sum_{j=1}^{D} (f(k_j, l_j) - \Gamma_j)$$

given the next period's value function V_{next}.

Set the training targets for the value function: $t_i = V(\mathbf{k_i^s})$.

If required, set the training targets to learn the policy function $\xi_{j_i}(\mathbf{k_i^s}) \in \arg\max_{p_j} V(\mathbf{k_i^s})$.

end

Set $\mathbf{t} = \{t_i : 1 \leq i \leq n\}$.

Given $\{\mathbf{X}, \mathbf{t}\}$, learn a surrogate $V_{surrogate}$ of V with ASGP (or GP).

Set $\xi_\mathbf{j} = \{\xi_{j_i} : 1 \leq i \leq n\}$.

Given $\{\mathbf{X}, \xi_\mathbf{j}\}$, learn a surrogate of the policy ξ_j with ASGP (or GP).

Calculate (an approximation for) the error, e.g., $\eta = \|V_{surrogate} - V_{next}\|_\infty$.

Set $V_{next} = V_{surrogate}$.

Set $s = s + 1$.

end

$V^* = V_{surrogate}$.

$\xi^* = \{\xi_1, \cdots, \xi_{3D}\}$.

"Hybrid parallel"
Implementation
(Shared & distributed memory parallelism)

Algorithmic Complexity of GPR-DP

N: number of observations.

The computational cost of standard GPR is dominated by the need to construct the Cholesky decomposition of the N×N covariance matrix at each step of the likelihood optimization – that is, ***O(N³)***.

The computational cost of **AS** arises from a single SVD of an N × N matrix, plus the cost of a standard GP regression that is, it is also ***O(N³)***.

In both cases, the right number of observations *N* depends on the function smoothness and the input dimensionality (rule of thumb: **N ~ 2-10D observations**):

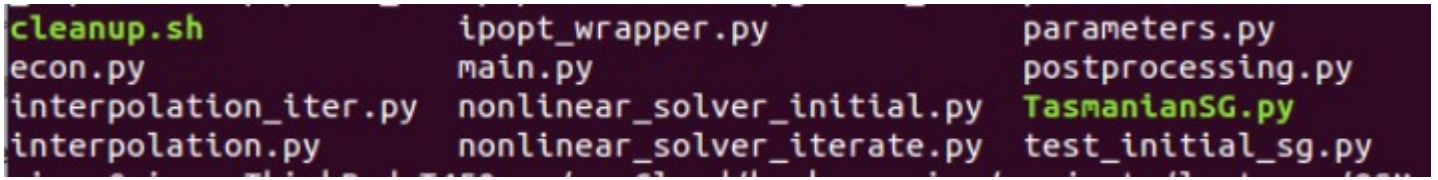
D for the GP regression and **d** for the ASGP regression.

The complete details of this relationship are not entirely understood theoretically and are well beyond the scope of the present lecture. However, we observe in numerical experiments that the number of samples required by GP regression, is GPR much larger than the number of samples required by AS-GPR,

$$N_{\text{GPR}} \gg N_{\text{AS-GPR}} \text{ when } D \gg d.$$

Setup of Code

code/growth_model_GPR



main.py: driver routine

econ.py: contains production function, utility,...

nonlinear_solver_initial/iterate.py: interface GPR ↔ IPOPT.

ipopt_wrapper.py: specifies the optimization problem (objective function,...).

interpolation.py: interface value function iteration ↔ sparse grid.

postprocessing.py: auxiliary routines, e.g., to compute the error.

Code snippet – main.py

```
import nonlinear_solver_initial as solver     #solves opt. problems for terminal VF
import nonlinear_solver_iterate as solviter   #solves opt. problems during VFI
from parameters import *                      #parameters of model
import interpolation as interpol              #interface to sparse grid library/terminal VF
import interpolation_iter as interpol_iter    #interface to sparse grid library/iteration
import postprocessing as post                 #computes the L2 and Linfinity error of the model
import numpy as np


#=============================================================================
# Start with Value Function Iteration


for i in range(numstart, numits):
# terminal value function
    if (i==1):
        print "start with Value Function Iteration"
        interpol.GPR_init(i)

    else:
        print "Now, we are in Value Function Iteration step", i
        interpol_iter.GPR_iter(i)


#=============================================================================
print "============================================================="
print " "
print " Computation of a growth model of dimension ", n_agents ," finished after ", numits, " steps"
print " "
print "============================================================="
#=============================================================================

# compute errors
avg_err=post.ls_error(n_agents, numstart, numits, No_samples_postprocess)

#=============================================================================
print "============================================================="
print " "
#print " Errors are computed -- see error.txt"
print " "
print "============================================================="
#=============================================================================
```

Code snippet – parameters.py

```
#=====================================================================

# How many training points for GPR
n_agents= 1  # number of continuous dimensions of the model
No_samples = 10*n_agents

# control of iterations
numstart = 1   # which is iteration to start (numstart = 1: start from scratch, number=/0: restart)
numits = 7    # which is the iteration to end

filename = "restart/restart_file_step_"  #folder with the restart/result files

#=====================================================================

# Model Paramters

beta=0.8
rho=0.95
zeta=0.5
psi=0.36
gamma=2.0
delta=0.025
eta=1
big_A=(1.0-beta)/(psi*beta)

# Ranges For States
k_bar=0.2
k_up=3.0
range_cube = k_up - k_bar # range of [0..1]^d in 1D


# Ranges for Controls
c_bar=1e-2
c_up=10.0

l_bar=1e-2
l_up=10.0

inv_bar=1e-2
inv_up=10.0

#=====================================================================

# Number of test points to compute the error in the postprocessing
No_samples_postprocess = 20
```

Code snippet – ipopt_wrapper.py

```
#=======================================================================
#   Objective Function to start VFI (in our case, the value function)

def EV_F(X, k_init, n_agents):

    # Extract Variables
    cons=X[0:n_agents]
    lab=X[n_agents:2*n_agents]
    inv=X[2*n_agents:3*n_agents]

    knext= (1-delta)*k_init + inv

    # Compute Value Function
    VT_sum=utility(cons, lab) + beta*V_INFINITY(knext)

    return VT_sum

# V infinity
def V_INFINITY(k=[]):
    e=np.ones(len(k))
    c=output_f(k,e)
    v_infinity=utility(c,e)/(1-beta)
    return v_infinity

#=======================================================================
#   Objective Function during VFI (note - we need to interpolate on an "old" GPR)

def EV_F_ITER(X, k_init, n_agents, gp_old):

    # Extract Variables
    cons=X[0:n_agents]
    lab=X[n_agents:2*n_agents]
    inv=X[2*n_agents:3*n_agents]

    knext= (1-delta)*k_init + inv

    #transform to comp. domain of the model
    knext_cube = box_to_cube(knext)

    # initialize correct data format for training point
    s = (1,n_agents)
    Xtest = np.zeros(s)
    Xtest[0,:] = knext_cube

    # interpolate the function, and get the point-wise std.
    V_old, sigma_test = gp_old.predict(Xtest, return_std=True)

    VT_sum = utility(cons, lab) + beta*V_old

    return VT_sum
```

Code snippet – interpolate_iter.py

```
def GPR_iter(iteration):


    # Load the model from the previous iteration step
    restart_data = filename + str(iteration-1) + ".pcl"
    with open(restart_data, 'rb') as fd_old:
        gp_old = pickle.load(fd_old)
        print "data from iteration step ", iteration -1 , "loaded from disk"
    fd_old.close()

    ##generate sample aPoints
    np.random.seed(666)   #fix seed
    dim = n_agents
    Xtraining = np.random.uniform(k_bar, k_up, (No_samples, dim))
    y = np.zeros(No_samples, float) # training targets

    # solve bellman equations at training points
    for iI in range(len(Xtraining)):
        y[iI] = solver.iterate(Xtraining[iI], n_agents,gp_old)[0]

    #for iI in range(len(Xtraining)):
        #print Xtraining[iI], y[iI]

    # Instantiate a Gaussian Process model
    #kernel = 1.0 * RBF(length_scale=1.0, length_scale_bounds=(1e-1, 10.0))

    kernel = 1.0 * RBF(length_scale=100.0, length_scale_bounds=(1e-1, 2e2)) \
    + WhiteKernel(noise_level=1, noise_level_bounds=(1e-3, 1e+0))

    #kernel = 1.0 * RBF(length_scale=100.0, length_scale_bounds=(1e-1, 2e2))
    #kernel = 1.0 * Matern(length_scale=1.0, length_scale_bounds=(1e-1, 10.0),nu=1.5)

    gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10)

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

    ##save the model to a file
    output_file = filename + str(iteration) + ".pcl"
    print output_file
    with open(output_file, 'wb') as fd:
        pickle.dump(gp, fd, protocol=pickle.HIGHEST_PROTOCOL)
        print "data of step ", iteration ,"  written to disk"
        print " ----------------------------------------"
    fd.close()
```

Code snippet – IPOPT_wrapper.py

```
#=========================================================================
#   Objective Function during VFI (note - we need to interpolate on an "old" sprase grid)
def EV_F_ITER(X, k_init, theta_init, n_agents, grid_list):

    # Extract Variables
    cons=X[0:n_agents]
    lab=X[n_agents:2*n_agents]
    inv=X[2*n_agents:3*n_agents]

    knext= (1-delta)*k_init + inv

    # Compute E[V(next, theta)]
    exp_v=0.0

    for itheta in range(ntheta):
        theta_next=theta_range[itheta]
        exp_v+=prob(theta_init, theta_next)*grid_list[itheta].evaluate(knext)


    # Compute Value Function
    VT_sum=utility(cons, lab) + beta*exp_v

    return VT_sum
```

Run the Growth model code

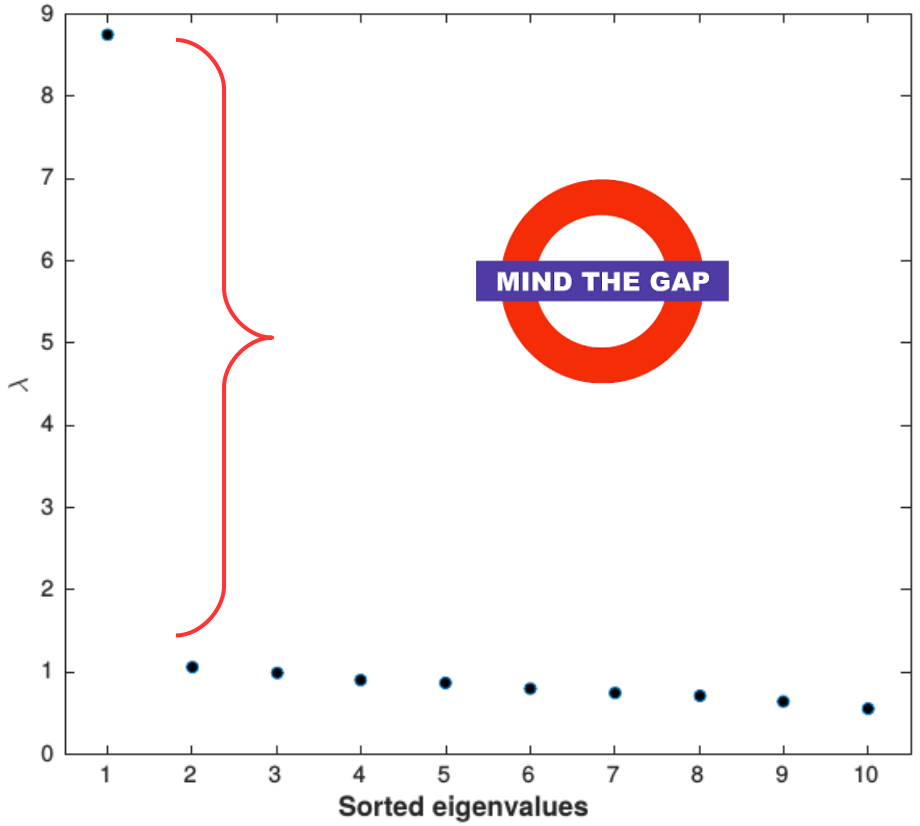
→ Model implemented in Python (scikit-learn)

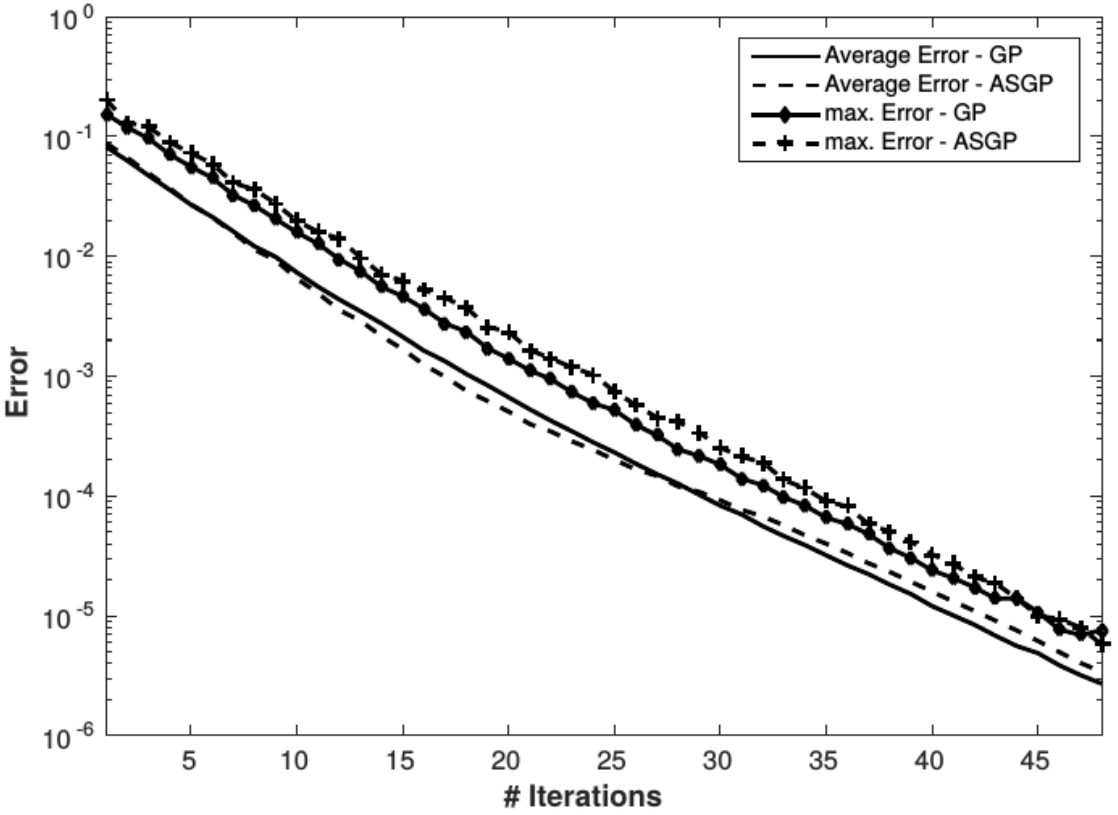
→ Optimizer used: IPOPT & PYIPOPT (python interface)

→ run with

$ python main.py

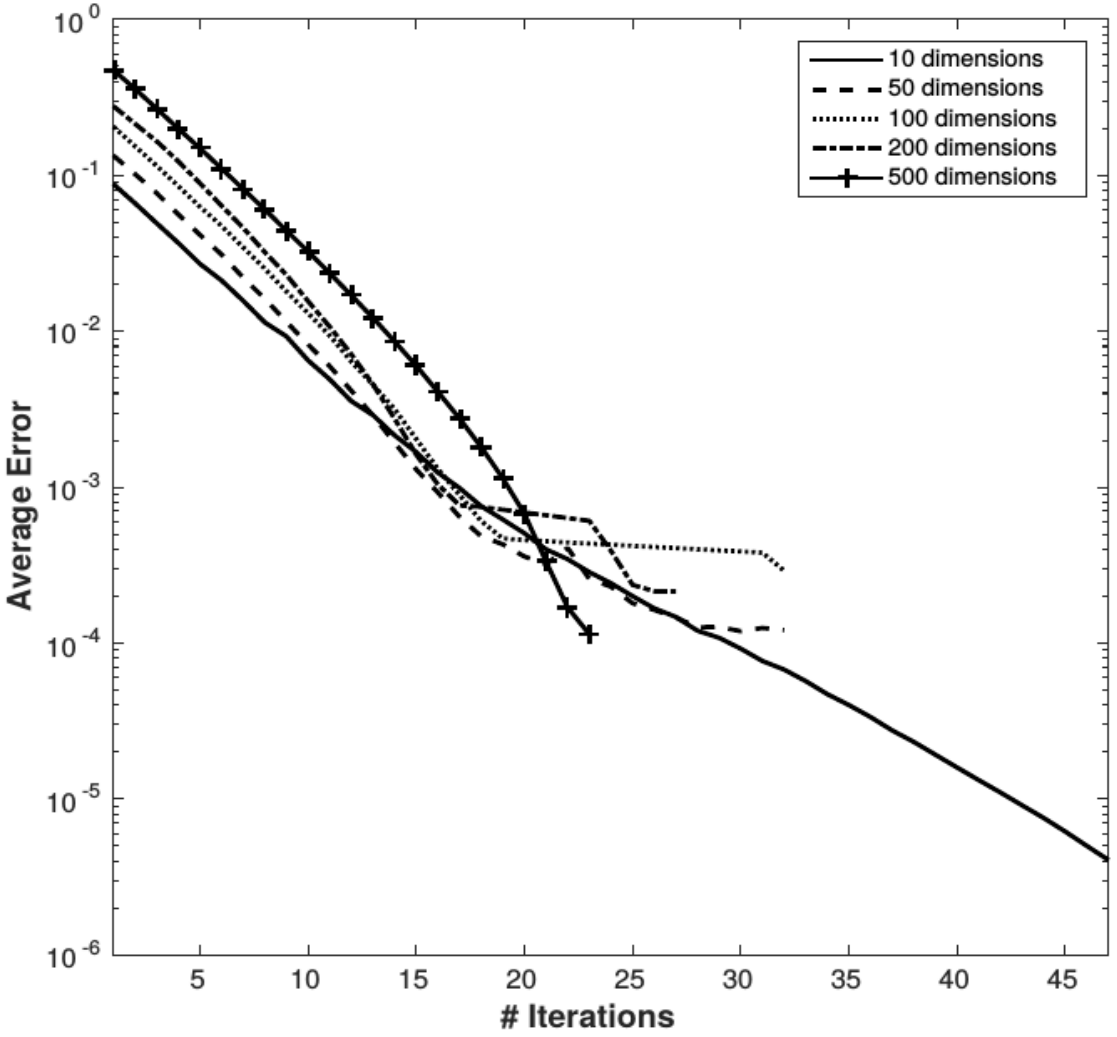
10d Growth Model – converges





Left panel: sorted eigenvalues of of the 10-dimensional OG model.
Right panel: decreasing maximum and average error for 10-D OG model, computed either by GPs or ASGP with an AS of dimension 1, respectively.

Growth model – 1 active subspace



Dynamic Programming on irregularly-shaped geometries

- Natural modelling geometry of economic models often not a hypercube, but rather **a simplex** (e.g., Brumm & Kubler (2014)), or a **hyper-ball** kind of ergodic set (Judd et. al. (2014), Maliar & Maliar (2015)).

→ We need to be able to perform value function iteration or time iteration on "arbitrary/irregularly-shaped" geometries.

→ Want to focus on the region of interest (**limited computational budged**).

Recall – GPs

(e.g. Murphy (2012), Rasmussen & Williams (2006), with references therein)

GPs: grid-free way of constructing interpolators

→ I can add geometry-free observations to *D**!

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

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

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

Test point = interpolation at X*

***this feature is also useful in case the solver does not converge at some point!**

Simulate the economy

Judd et. al. (2014), Maliar & Maliar (2015)

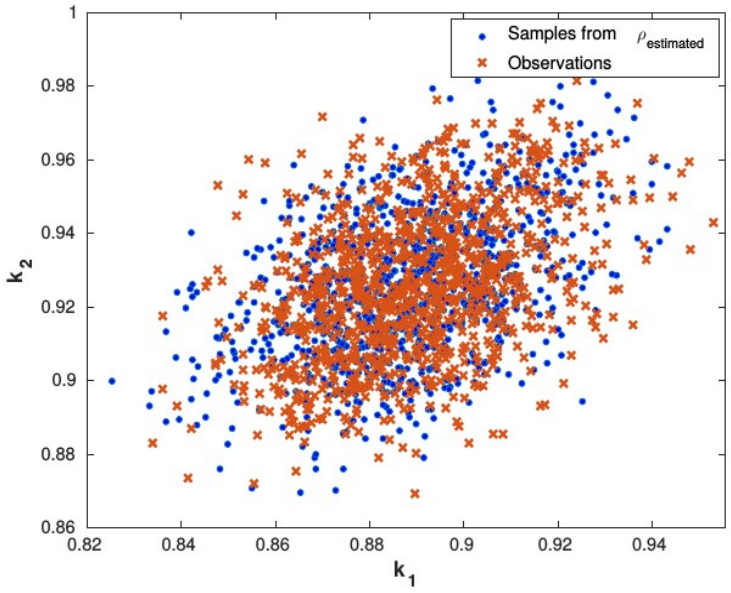
1. Simulate economy at few points, learn the policy

$$k_j^+ = (1-\delta)\cdot k_j + I_j + \epsilon_j, \quad j = 1, ..., D$$

2. Approximate density with mixture of Gaussians
(e.g. Rasmussen (2000), Blei & Jordan (2005))

$$\rho_{estimated}(\mathbf{x}) = \sum_{m=1}^{M} \pi_m \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m)$$

→ **generate training data; plug them into VFI**



DP on ergodic sets

Data: Initial guess V_{next} for the next period's value function. Approximation accuracy $\bar{\eta}$.
Result: The m (approximate) equilibrium policy functions ξ^* and the corresponding value function V^* on $\Omega_{ergodic}$.
Set iteration step $s = 1$.
while $\eta > \bar{\eta}$ **do**
  Determine $\Omega_{ergodic}$ by using (64) and (65).
  Generate n training inputs $\mathbf{X}_{ergodic} = \{\mathbf{k}_i^s : 1 \leq i \leq n\} \in \Omega_{ergodic}$
  for $\mathbf{k}_i^s \in \mathbf{X}_{ergodic}$ **do**
    Evaluate the Bellman operator $TV^s(\mathbf{k}_i^s)$ (see Eq. (5)) given next period's value function V_{next}.
    Set the training targets for the value function: $t_i = TV(\mathbf{k}_i^s)$.
    If required, set the training targets to learn the j-th policy function: $\xi_{j_i}(\mathbf{k}_i^s) \in \arg\max_{p_j} TV(\mathbf{k}_i^s)$.
  end
  Set $\mathbf{t}_{ergodic} = \{t_i : 1 \leq i \leq n\}$.
  Given $\{\mathbf{X}_{ergodic}, \mathbf{t}_{ergodic}\}$, learn a surrogate $V_{surrogate}$ of V.
  Set $\xi_\mathbf{j} = \{\xi_{j_i} : 1 \leq i \leq n\}$.
  Given $\{\mathbf{X}_{ergodic}, \xi_\mathbf{j}\}$, learn a surrogate of the policy ξ_j.
  Calculate (an approximation for) the error, e.g.: $\eta = \|V_{surrogate} - V_{next}\|_\infty$.
  Set $V_{next} = V_{surrogate}$.
  Set $s = s + 1$.
end
$V^* = V_{surrogate}$.
$u^* = \{\xi_1, \cdots, \xi_m\}$.

Algorithm 2: Overview of the critical steps of the VFI algorithm that operates on the ergodic set $\Omega_{ergodic}$.

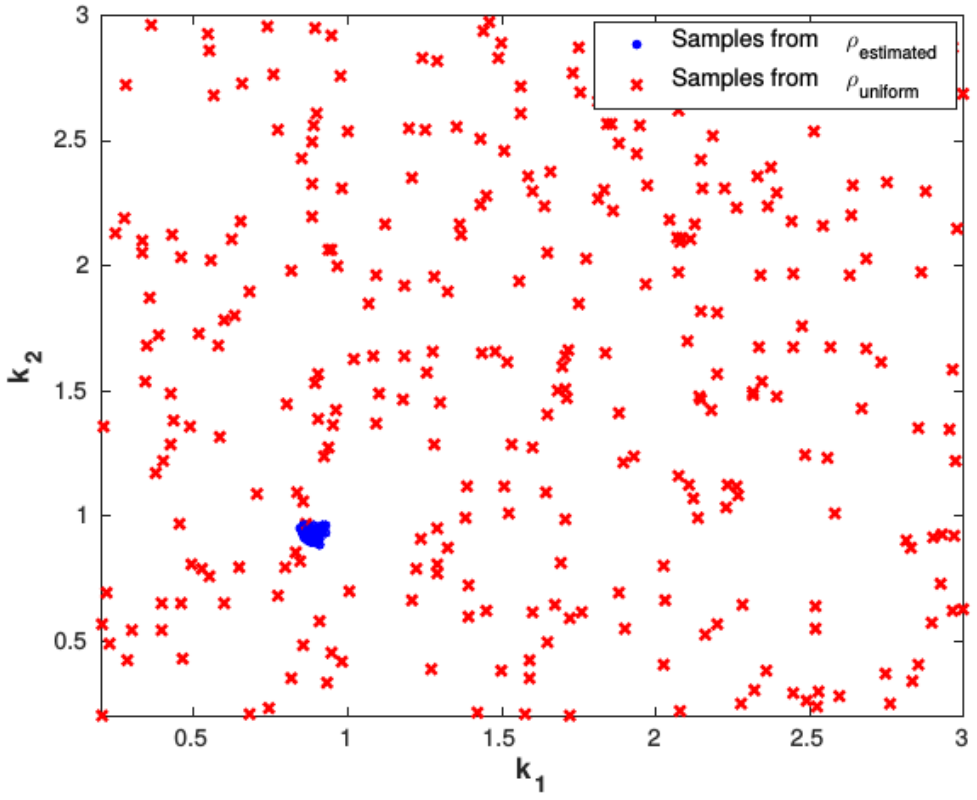
→ **Probably only every 5 to 10 steps...**

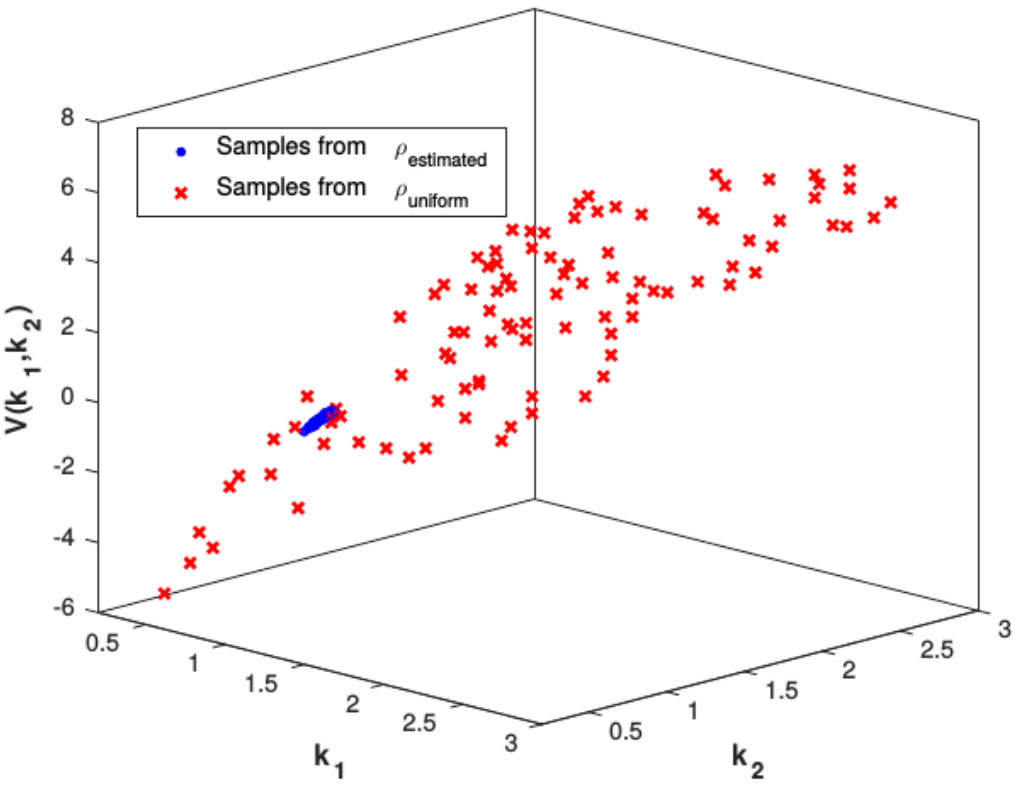
$$\mathbf{X}_{ergodic} = \left\{\mathbf{x}_{ergodic}^{(1)}, \ldots, \mathbf{x}_{ergodic}^{(N)}\right\}$$

$$\mathbf{t}_{ergodic} = \left\{t_{ergodic}^{(1)}, \ldots, t_{ergodic}^{(N)}\right\}$$

$$\{\mathbf{X}_{ergodic}, \mathbf{t}_{ergodic}\} \in \Omega_{ergodic}$$

Focus resources where needed





Uncertainty Quantification (UQ)

→ Uncertainty quantification (UQ) is a field of applied mathematics concerned with the **quantification and propagation of uncertainties** through computational models.

→ In the quantitative economics community, UQ should be of paramount interest, as it can help to address questions such as **which parameters are driving the conclusions derived from an economic model**.

→ Answering them, in turn, can inform the researcher for example **on which parts of the model she needs to focus** on when calibrating it.

There are various sources of uncertainty that can enter economic models, including:

i) **parameter uncertainty** (robustness of results)

ii) **interpolation uncertainty** (more data reduce uncertainty)

Enlarge the state space

- Standard approaches require a **large number of model evaluations**.

- **We get away with a single model evaluation!**

→ Since we can deal with very high-dimensional problems, we simply **enlarge the state space** by the parameters of interest, e.g.:

$$\tilde{S} = (\mathbf{k}_t, \gamma)$$

In the growth model, we simply set it to:

$$[\underline{\mathbf{k}}, \overline{\mathbf{k}}]^D \times [\underline{\gamma}, \overline{\gamma}] = [0.2, 3]^D \times [0.5, 5]$$

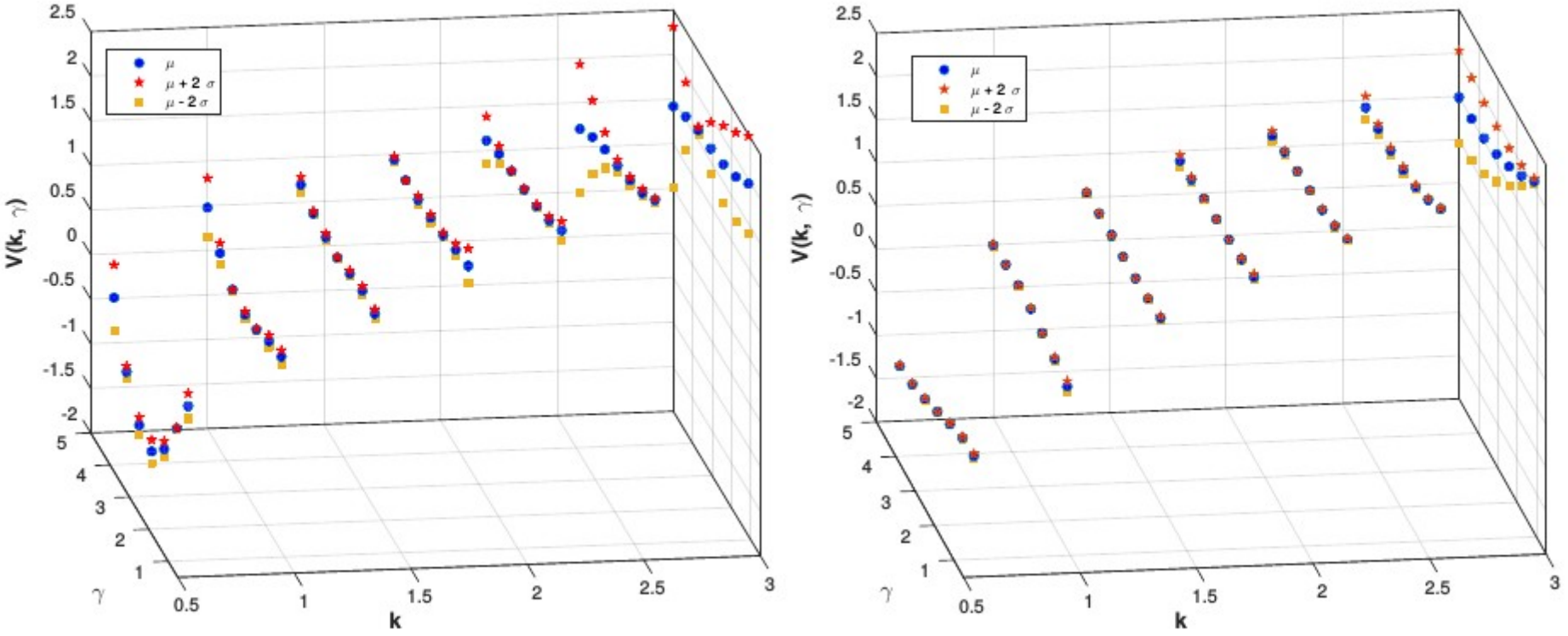
→ no calibration exercise needed!

→ evaluate, e.g., univariate impact of a parameter on the model conclusion.

→ **Bring the model and the data together.**

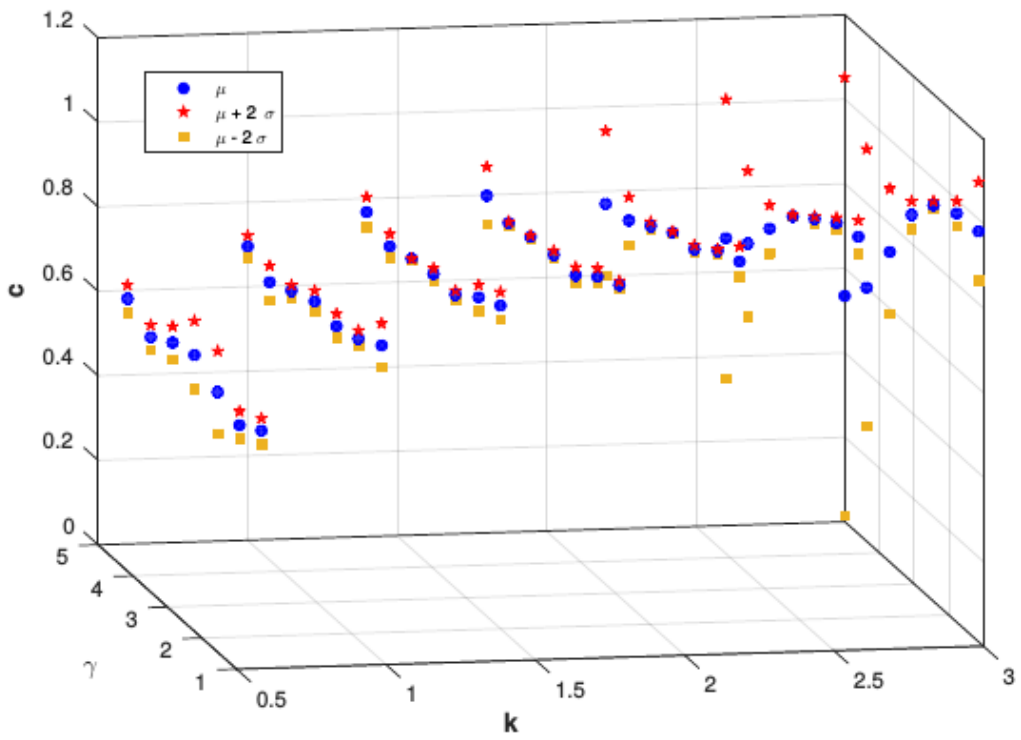
Value function V(**k**,*γ*)

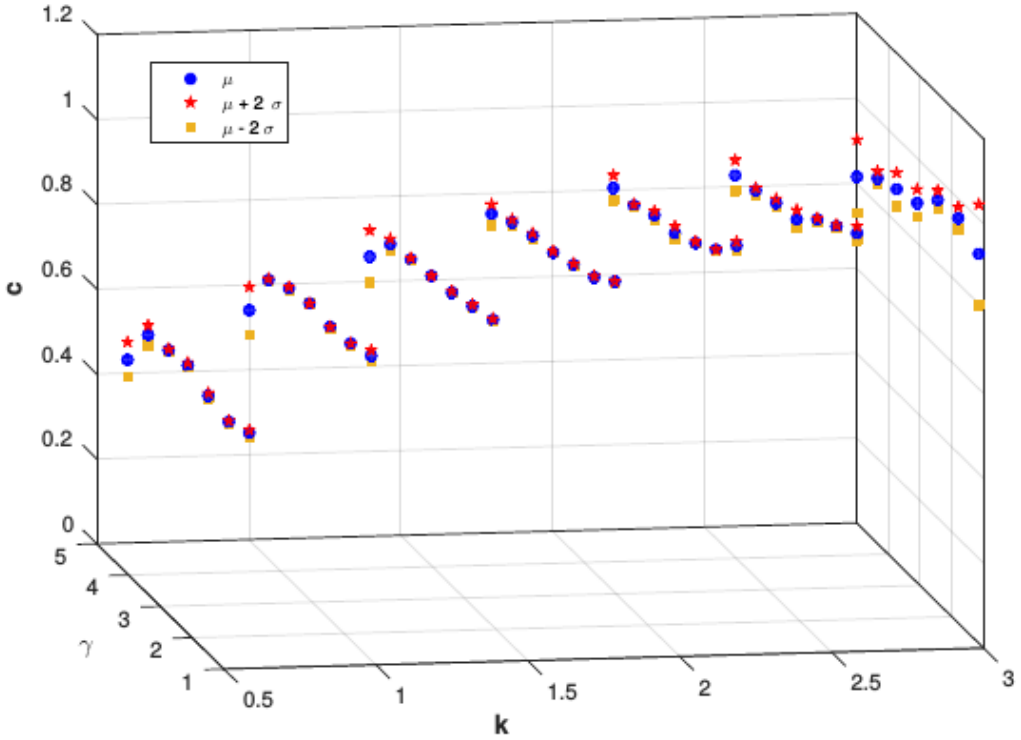
Solve a DP problem with ***γ*** (risk aversion) uniformly sampled from [0.5: 5]
Note: other distributions for parameters possible!



Left: predictive mean and 95% quantile of the value function (20 sample points)
Right: predictive mean and 95% quantile of the value function (40 sample points)

Consumption c(**k**,*y*)





Left: predictive mean and 95% quantile of the value function (20 sample points)
Right: predictive mean and 95% quantile of the value function (40 sample points)

Convergence of ASGDP

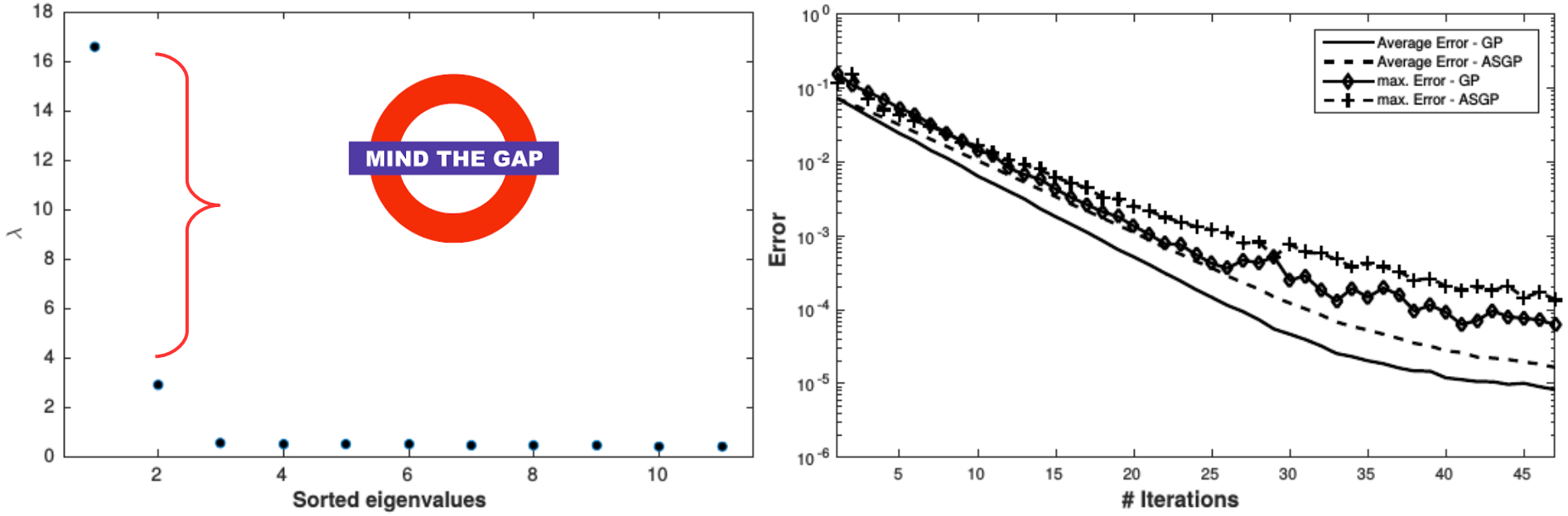


Figure 13: Left panel—Sorted eigenvalues of $\mathbf{C}_N$ (see Eq. (42)) for the 11-dimensional OG problem with continuous states $\tilde{S} = (k_1, ..., k_{10}, \gamma)$. Right panel—Decreasing maximum and average error for the two 11-dimensional OG models, computed either by GPs or by ASGP with an AS of dimension 1, respectively.)

Quantity of interest (QoI) – an example

Jaynes (1982); Harenberg (2017), with references therein

Parameters: $\chi = \{\gamma, \delta, \psi, \zeta, \eta\}$

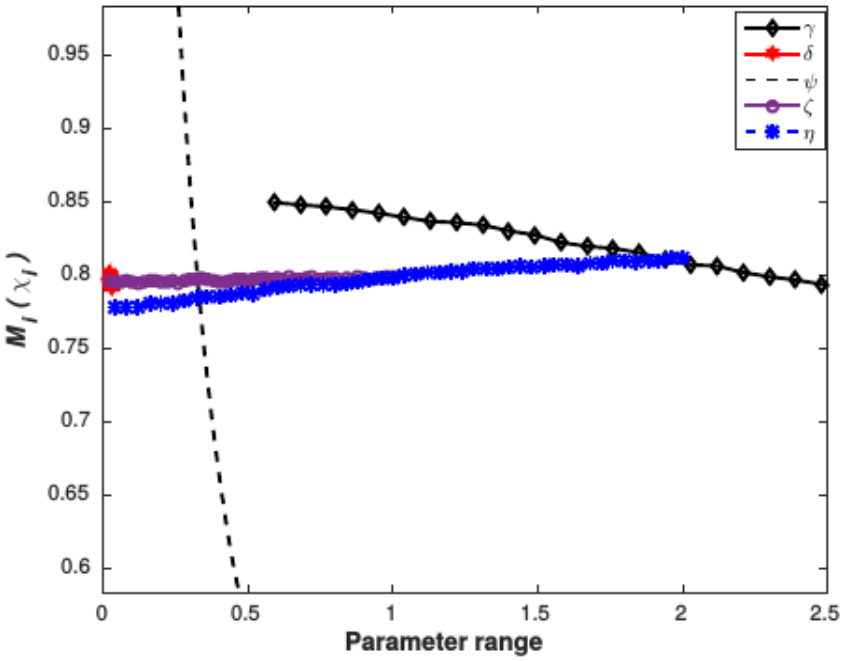
Parameter	Baseline value	lower bound for $\underline{\chi}_i$	upper bound for $\overline{\chi}_i$
γ	2.0	0.5	5.0
δ	0.025	0.02	0.03
ψ	0.36	0.32	0.4
ζ	0.5	0.0	1.0
η	1.0	0.0	2.0

Extended state space: $\tilde{S} = (\mathbf{k}, \chi) \rightarrow [\underline{\mathbf{k}}, \overline{\mathbf{k}}] \times [\underline{\gamma}, \overline{\gamma}] \times [\underline{\delta}, \overline{\delta}] \times [\underline{\psi}, \overline{\psi}] \times [\underline{\zeta}, \overline{\zeta}] \times [\underline{\eta}, \overline{\eta}].$

QoI: **aggregate production**: $\mathcal{M}(\chi) = \mathbb{E}[f]$

Univariate effects:

$$\mathcal{M}_i(\chi_i) = \mathbb{E}[\mathcal{M}(\mathbf{\Theta}|\Theta_i = \chi_i)]$$



→ commonly interpreted as a robust relationship between an input parameter and the QoI.

→ **A global solution method required here.**

Run the Growth model code

→ Model implemented in Python (TASMANIAN)

→ Optimizer used: IPOPT & PYIPOPT (python interface)

→ global_solution_yale19/Lecture_2/SparseGridCode/growth_model/serial_stochastic

→ run with

>python main.py

Advanced Topics (con’d)

http://proceedings.mlr.press/v37/wilson15.pdf

- Ordinary GPs suffer from N^3 scaling.
- Use a sparse kernel matrix.
- KISS-GP (Structured Kernel Interpolation).
- SKI (or KISS-GP) is a great way to scale a GP up to very large datasets (100,000+ data points).
- https://people.orie.cornell.edu/andrew/pattern/#SKI
- Kernel interpolation for scalable structured Gaussian processes (KISS-GP) was introduced in this paper: http://proceedings.mlr.press/v37/wilson15.pdf
- SKI is asymptotically very fast (nearly linear), very precise (error decays cubically).
- See, e.g., https://docs.gpytorch.ai/en/v1.1.1/examples/02_Scalable_Exact_GPs/KISSGP_Regression.html

Advanced Topics

http://proceedings.mlr.press/v37/wilson15.pdf

- Deep Kernel GP
- https://arxiv.org/pdf/1511.02222.pdf
- https://people.orie.cornell.edu/andrew/pattern/
- Combines the structural properties of deep

  learning architectures with the non-parametric flexibility of kernel methods.
- Inference and learning cost O(n) for n training points, and predictions cost O(1) per test point (compared to O(n^3) and O(n^2) for standard GPs).
- Approach to scalability are largely grounded in exploiting algebraic structure in kernel.
- Matrices in combination with local kernel interpolation, for efficient and highly accurate numerical linear algebra operations.