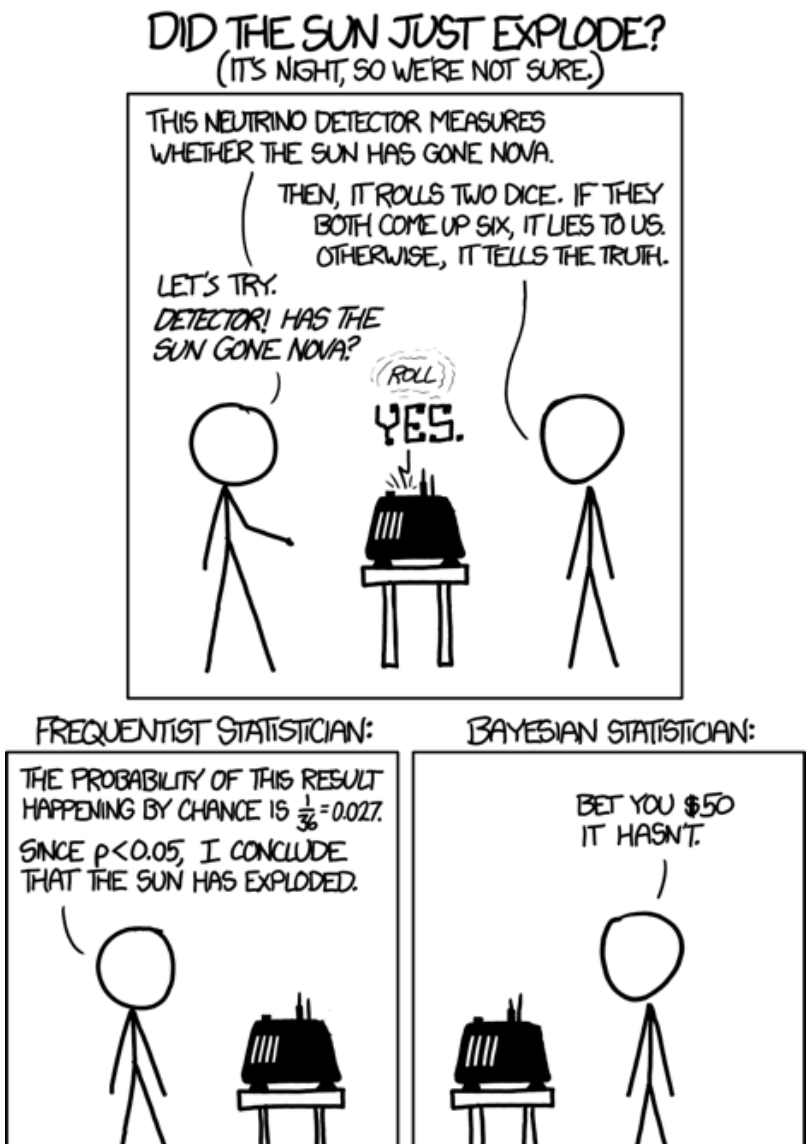
Lecture 7. Bayesian Learning

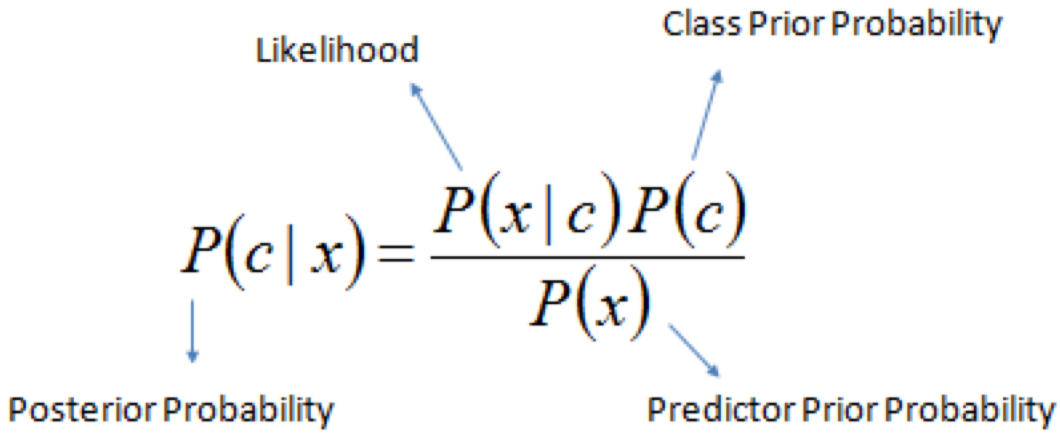
Dealing with uncertainty

Joaquin Vanschoren


DID THE SUN JUST EXPLODE?
(IT'S NIGHT, SO WE'RE NOT SURE.)
THIS NEUTRINO DETECTOR MEASURES WHETHER THE SUN HAS GONE NOVA.
THEN, IT ROLLS TWO DICE. IF THEY BOTH COME UP SIX, IT LIES TO US. OTHERWISE, IT TELLS THE TRUTH.
LET'S TRY.
DETECTOR! HAS THE SUN GONE NOVA?
ROLL
YES.
FREQUENTIST STATISTICIAN:
THE PROBABILITY OF THIS RESULT HAPPENING BY CHANCE IS 1/36 = 0.027.
SINCE p<0.05, I CONCLUDE THAT THE SUN HAS EXPLODED.
BAYESIAN STATISTICIAN:
BET YOU $50 IT HASN'T.

Bayes' rule

Rule for updating the probability of a hypothesis c given data x



$P(c|x)$ is the posterior probability of class c given data x.
$P(c)$ is the *prior* probability of class c: what you believed before you saw the data x
$P(x|c)$ is the *likelihood* of seeing data x if you know that the class is c
$P(x)$ is the prior probability of the data (*marginal likelihood*): the likelihood of the data x under any circumstance (no matter what the class is)

Example

- Let's compute the probability that the sun has exploded
- Prior $P(exploded)$: the sun has an estimated lifespan of 10 billion years, $P(exploded) = \frac{1}{4.38x10^{13}}$
- Likelihood that detector lies: $P(lie) = \frac{1}{36}$

$$
\begin{aligned}
P(exploded|yes) &= \frac{P(yes|exploded)P(exploded)}{P(yes)} \\
&= \frac{(1 - P(lie))P(exploded)}{P(exploded)(1 - P(lie)) + P(lie)(1 - P(explode} \\
&= \frac{1}{1.25226x10^{12}}
\end{aligned}
$$

- The one positive data point of the detector increases the probability, but only a tiny bit.

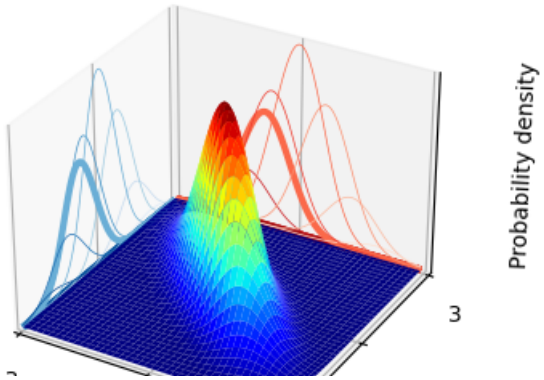
Example 2

- What is the probability of having COVID-19 if a 96% accurate test returns positive? Assume a false positive rate of 4%
- Prior $P(C) : 0.015$ (117M cases, 7.9B people)
- $P(TP) = P(pos|C) = 0.96$, and $P(FP) = (pos|notC) = 0.04$

$$
\begin{aligned}
P(C|pos) &= \frac{P(pos|C)P(C)}{P(pos)} \\
&= \frac{P(pos|C)P(C)}{P(pos|C)P(C) + P(pos|notC)(1 - P(C))} \\
&= \frac{0.96 * 0.015}{0.96 * 0.015 + 0.04 * 0.985} \\
&= 0.268
\end{aligned}
$$

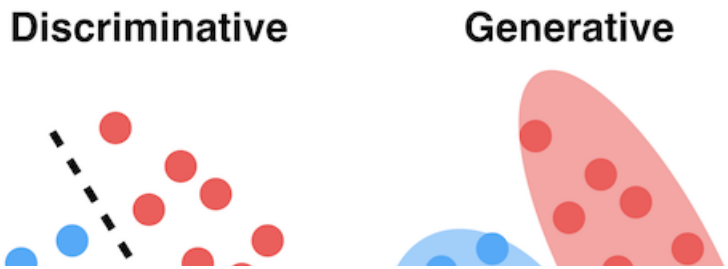
Bayesian models

- Are easily updatable with new data points ('turning the crank')
    - Previous posterior $P(c|x)$ becomes new prior $P(c)$, then apply Bayes' rule on new data
- They learn the joint distribution $P(x, y) = P(x|y)P(y)$.
    - With every input x you get a mean and uncertainty interval for y (blue)
    - For every desired output y you can see how to sample x (red)



Bayesian models are generative

- The joint distribution represents the training data for a particular output (e.g. a class)
- You can sample a *new* point $\mathbf{x}$ with high predicted likelihood $P(x, c)$: that new point will be very similar to the training points
- Hence, you can generate new (likely) points according to the same distribution: *generative model*
    - You can generate examples that are *fake* but very to other examples of the same class
    - Generative neural networks (e.g. GANs) can do this very accurately for text, images, ...



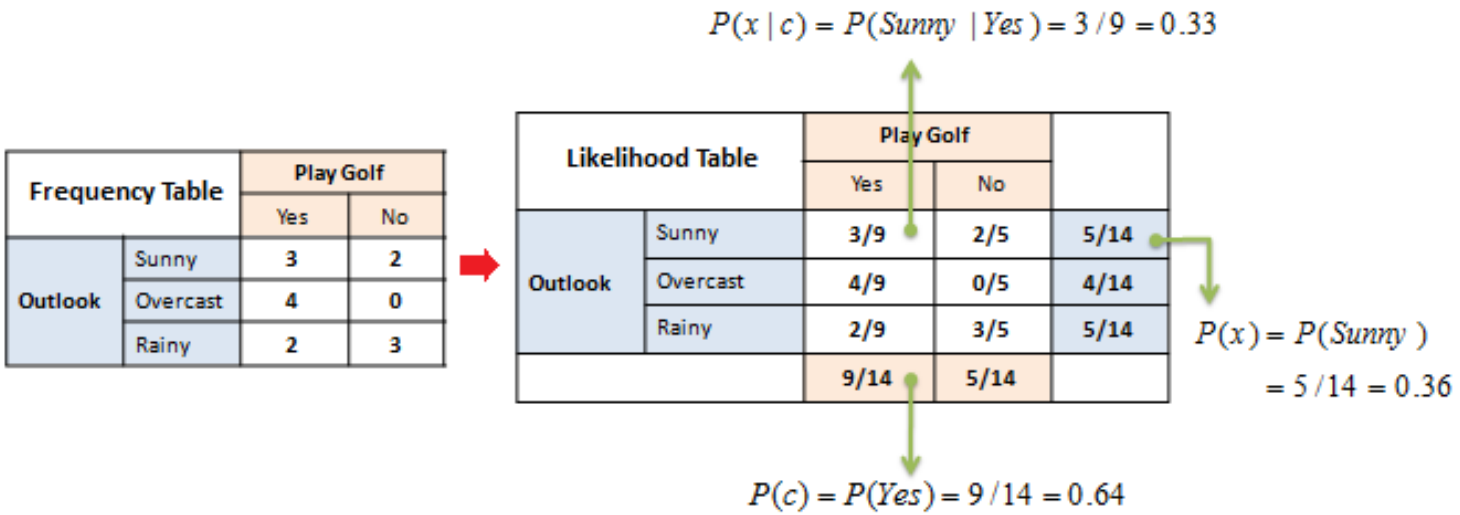
Naive Bayes Classifier

- Predict the probability that a point belongs to a certain class, using Bayes' Theorem

$$P(c|\mathbf{x}) = \frac{P(\mathbf{x}|c)P(c)}{P(\mathbf{x})}$$

- Problem: since $\mathbf{x}$ is a vector, $P(\mathbf{x}|c)$ can be very complex
- Naively assume that all features are conditionally independent from each other, in which case:
  $P(\mathbf{x}|c) = P(x_1|c) \times P(x_2|c) \times \ldots \times P(x_n|c)$
- Very fast: only needs to extract statistics from each feature.

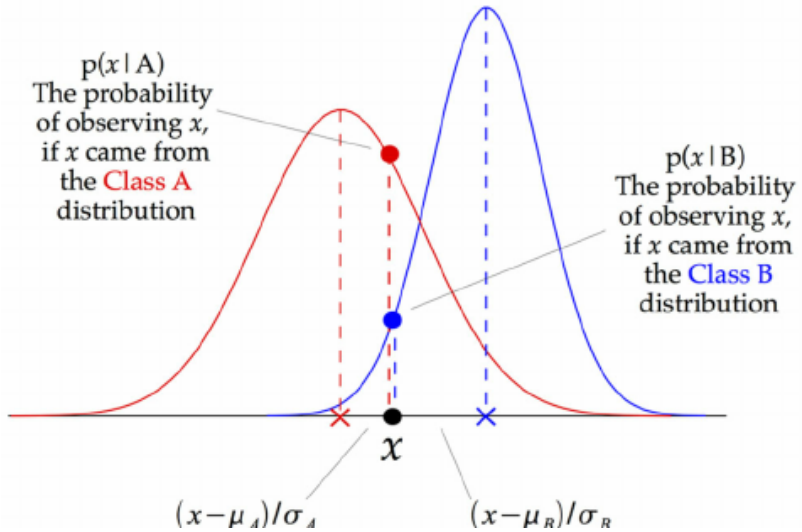
Example. What's the probability that your friend will play golf if the weather is sunny.



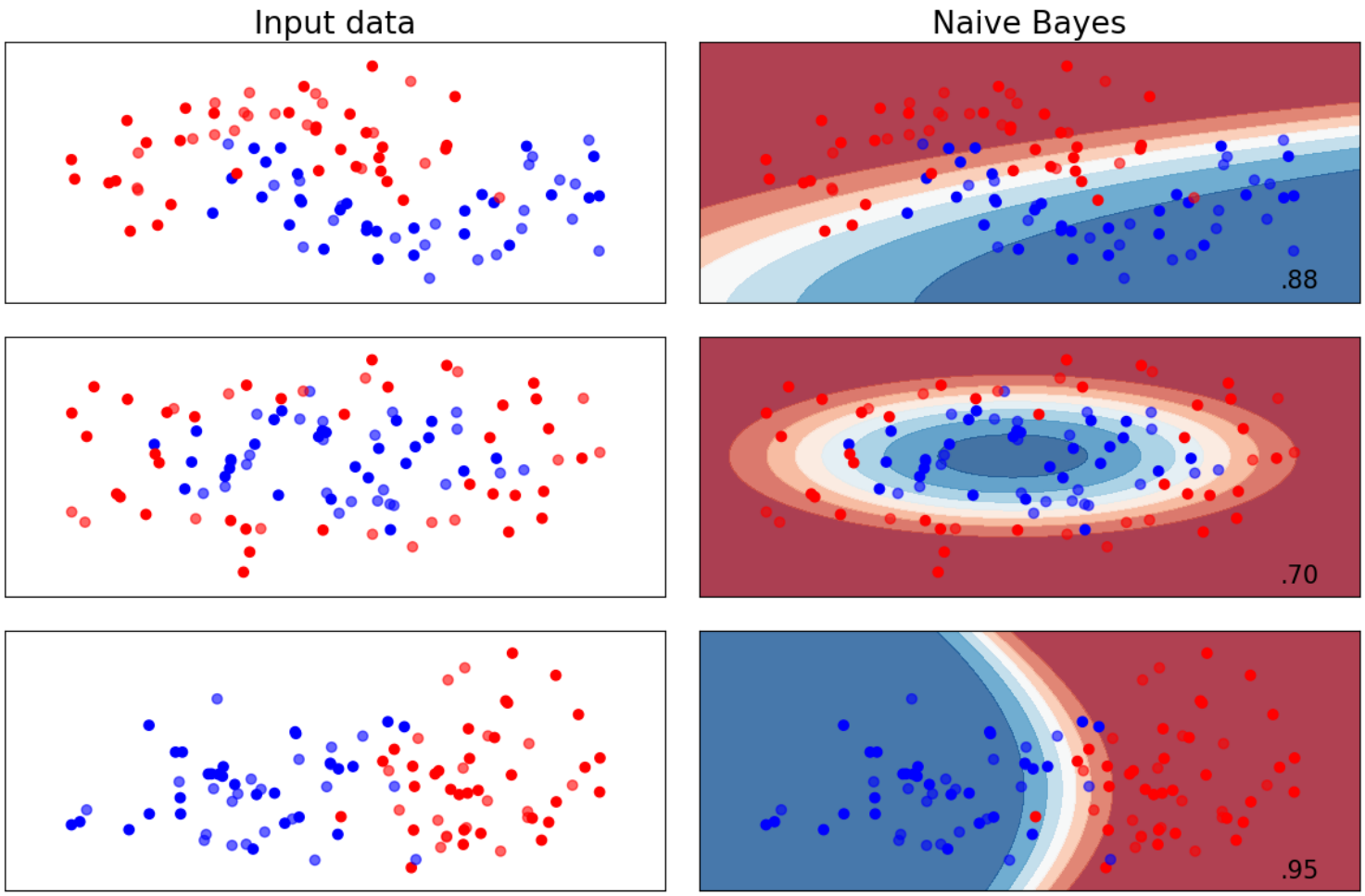
Posterior Probability: $P(c \mid x) = P(Yes \mid Sunny) = 0.33 \times 0.64 \div 0.36 = 0.60$

On numeric data

- We need to fit a distribution (e.g. Gaussian) over the data points
- GaussianNB: Computes mean μ_c and standard deviation σ_c of the feature values per class: $p(x = v \mid c) = \frac{1}{\sqrt{2\pi\sigma_c^2}} e^{-\frac{(v-\mu_c)^2}{2\sigma_c^2}}$
- Prediction are made using Bayes' theorem: $p(c \mid \mathbf{x}) = \frac{p(c)\, p(\mathbf{x}|c)}{p(\mathbf{x})}$



- What do the predictions of Gaussian Naive Bayes look like?

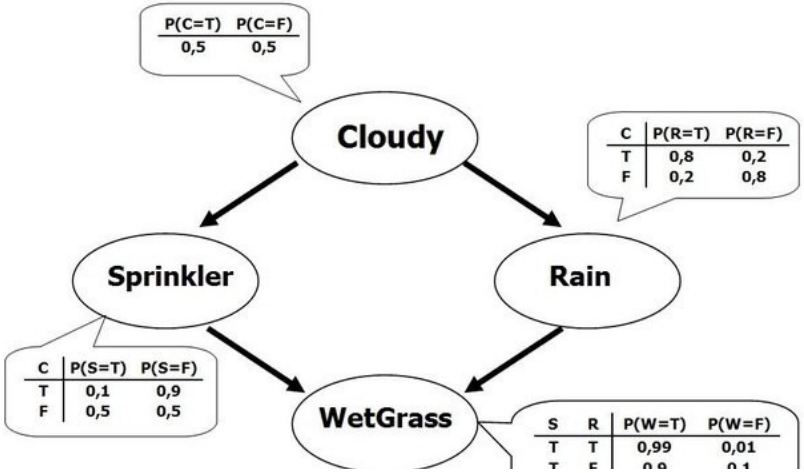


Other Naive Bayes classifiers:

- BernoulliNB
    - Assumes binary data
    - Feature statistics: Number of non-zero entries per class
- MultinomialNB
    - Assumes count data
    - Feature statistics: Average value per class
    - Mostly used for text classification (bag-of-words data)

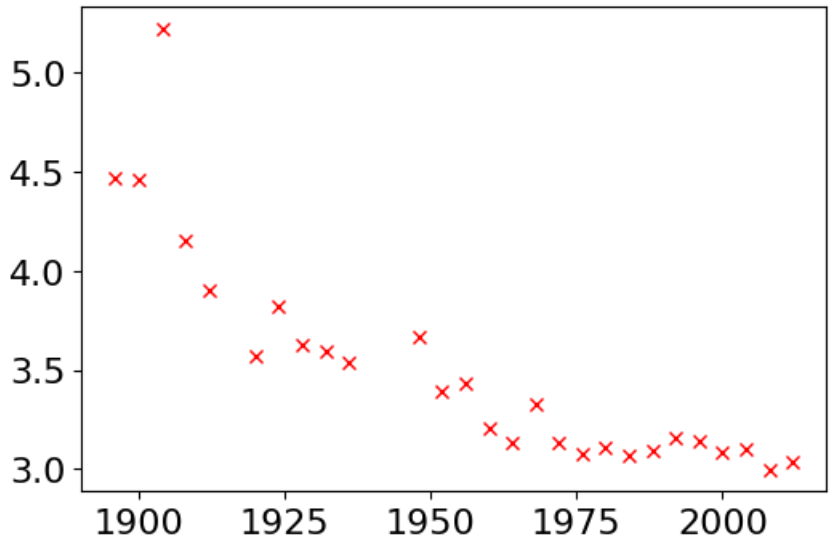
Bayesian Networks

- What if we know that some variables are not independent?
- A *Bayesian Network* is a directed acyclic graph representing variables as nodes and conditional dependencies as edges.
- If an edge (A, B) connects random variables A and B, then $P(B|A)$ is a factor in the joint probability distribution
    - We must know $P(B|A)$ for all values of B and A
- The graph structure can be designed manually or learned (hard!)



Linear regression and basis expansions (recap)

Let's look at a the following regression problem



Let's first try to fit a linear model

$$y = f(\mathbf{x}_i) = \mathbf{x}_i \mathbf{w} + b$$

In this case (with one input feature):

$$y = w_1 \cdot x_1 + b \cdot 1$$

We can solve this via linear algebra (closed form solution). To obtain a matrix we add a $x_0 = 1$ column to represent the bias b. Hence, each vector $\mathbf{x}_i = [1, x_i]$.

We can do this for the entire data set to form a design matrix $\mathbf{X}$,

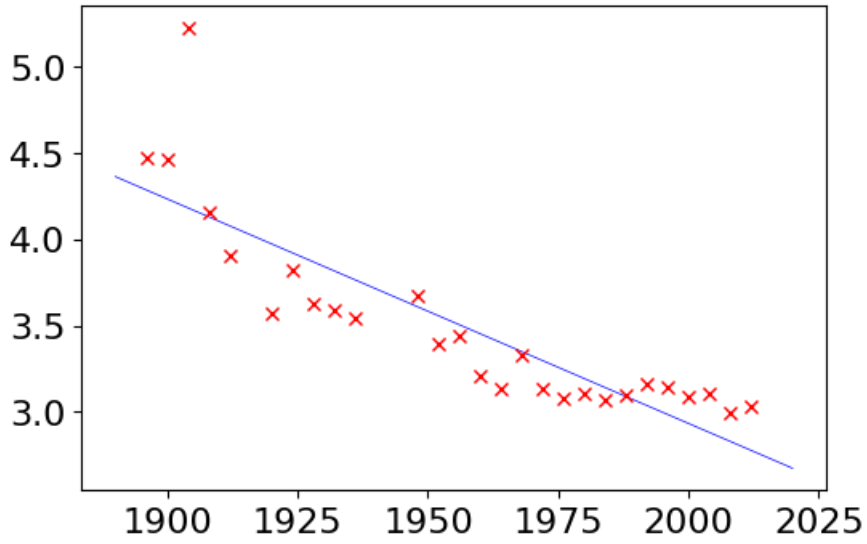
$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_N^\top \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix},$$

We can solve this with `numpy`

```
w = np.linalg.solve(np.dot(X.T, X), np.dot(X.T, y))
```

```
w: [[28.895 -0.013]]
```

Hence, we learned: $y = w_1 x + w_0 = -0.013x + 28.895$

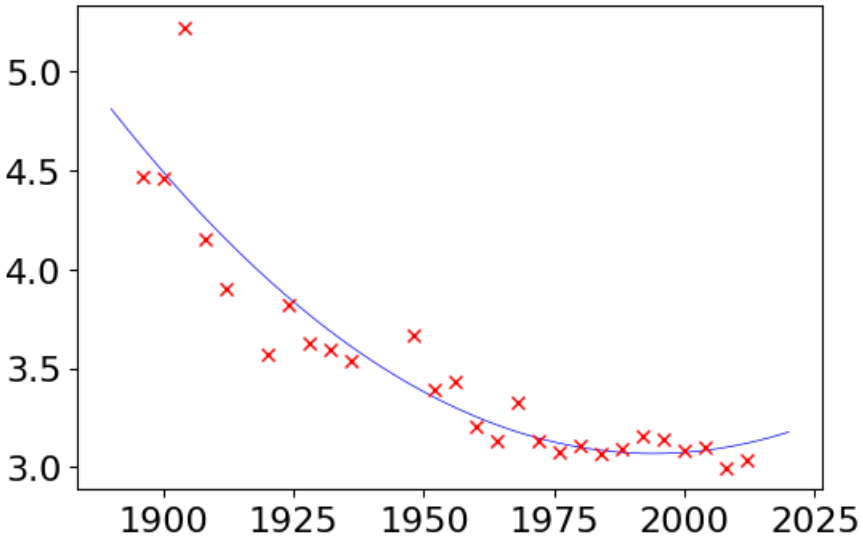


We can fit a 2nd degree polynomial by basis expansion (adding more *basis functions*):

$$\mathbf{\Phi} = \begin{bmatrix} \mathbf{1} & \mathbf{x} & \mathbf{x}^2 \end{bmatrix}$$

```
Phi = np.hstack([np.ones(x.shape), x, x**2])
w = np.linalg.solve(np.dot(Phi.T, Phi), np.dot(Phi.T, y))
```

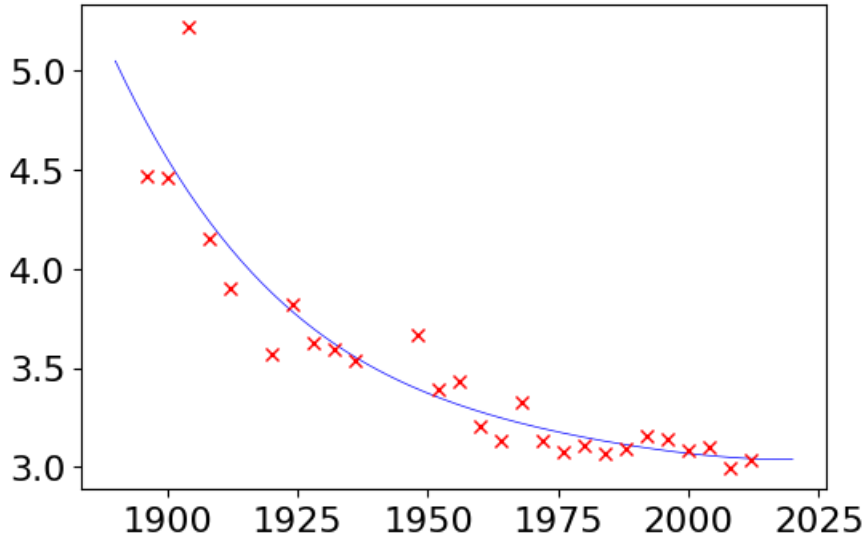
```
w: [[643.642  -0.643   0.   ]]
```



Repeating up to degree 6 gives us:

```
Phi = np.hstack([np.ones(x.shape), x, x**2, x**3, x**4, x**5, x**6])
w = np.linalg.solve(np.dot(Phi.T, Phi), np.dot(Phi.T, y))
```

```
w: [[55403.877   -58.241    -0.018     0.        -0.         0.         0.   ]]
```

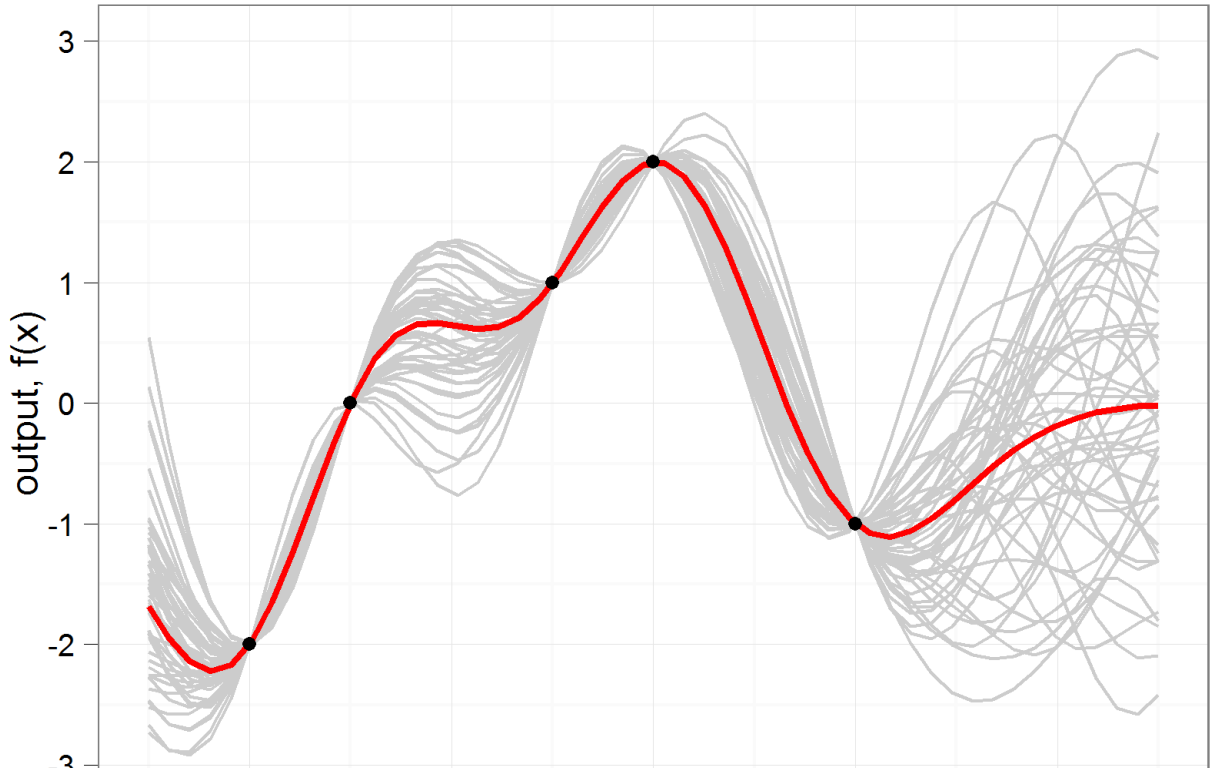


There must be a better way

- We would like a *probability* for predictions instead of just one value
    - We should be more certain about predictions close to the training points
    - We need a probabilistic version of regression
- How do we know the optimal degree?
    - Don't specify this up front
    - Consider *every possible function* that matches our data, with however many parameters are involved (i.e. a non-parametric model).
    - A *Gaussian process* learns a *distribution* on this set of functions

Gaussian processes

Learn a probability distribution of possible base functions, update this distribution based on new data.



Probabilistic interpretation of regression

- Assume that the data is inherently uncertain. This can be modeled explictly by introducing a slack variable, ϵ_i, known as noise.

$$y_i = w_1 x_i + w_0 + \epsilon_i.$$

- Assume that the noise (i.e. the slack variables) is distributed according to a probability density.
    - One often assumes Gaussian noise, with zero mean and variance σ^2.

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2),$$

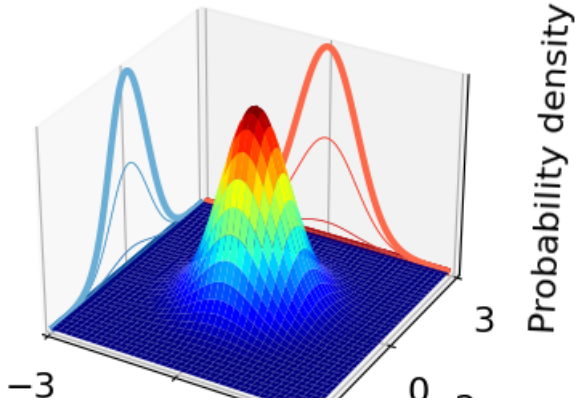
Bayesian prior

In the Bayesian approach, we also assume a *prior distribution* for the parameters, $\mathbf{w}$:

$$\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \alpha\mathbf{I})$$

I.e, w_i is drawn from a Gaussian density with variance α

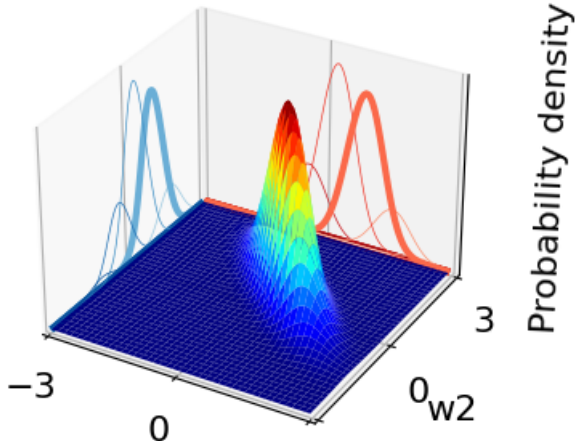
$$w_i \sim \mathcal{N}(0, \alpha)$$



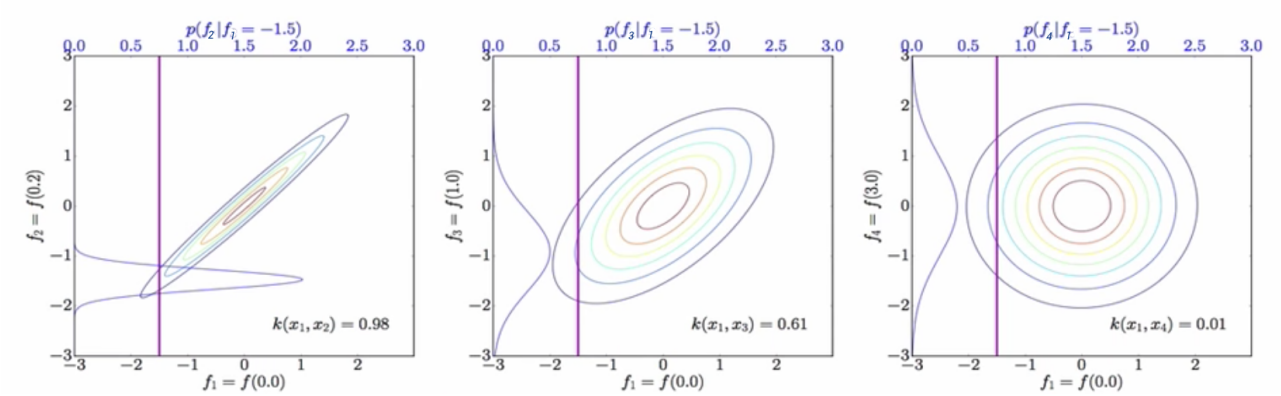
- The prior is typically $\mathcal{N}(\mathbf{0}, \alpha\mathbf{I})$, with mean $\mathbf{0}$ and covariance matrix $\alpha\mathbf{I}$

$$\begin{bmatrix} \alpha & 0 \\ 0 & \alpha \end{bmatrix}$$

- Given data, we will learn the actual distribution of w (the mean and covariance matrix)
- Example with mean $[m, m]$ and covariance $\begin{bmatrix} \alpha & \beta \\ \beta & \alpha \end{bmatrix}$



Understanding covariances



Left: If two variables f_i covariate strongly, knowing about f_1 tells us a lot about f_2
Right: If covariance is 0, knowing f_1 tells us nothing about f_2 (the conditional and marginal distributions are the same)

Gaussian process hyper-parameters

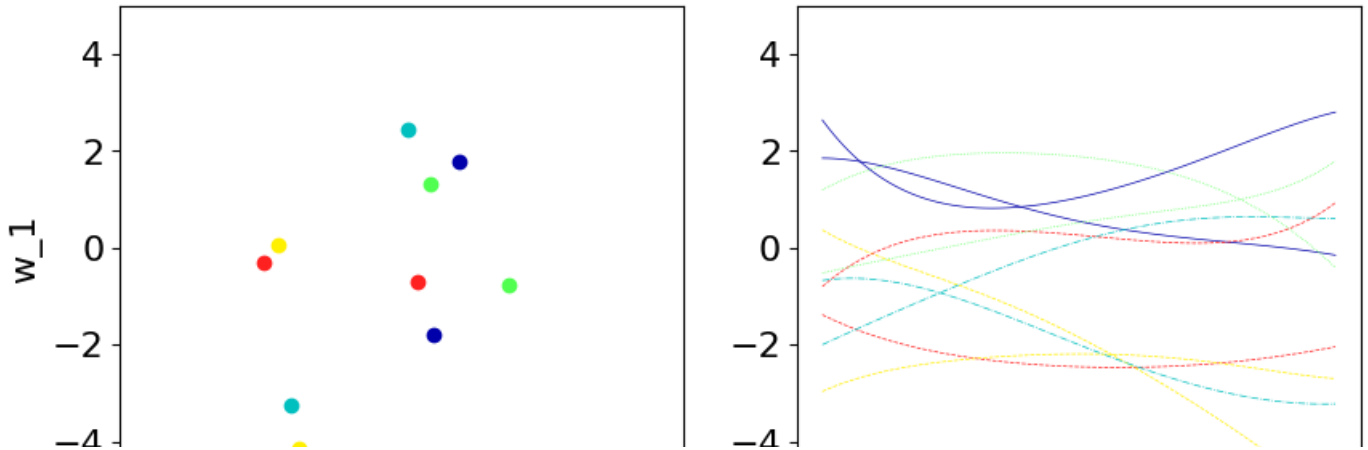
So far:

- breadth of the prior (alpha)
- degree of the basis functions (degree)
- noise level (sigma^2)

Weight Space View

Now we can sample (e.g. 100 points) from the prior distribution to see what form we are imposing on the functions *a priori* (before seeing any data).

- Draw w (left) independently from a Gaussian density $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \alpha\mathbf{I})$
- Every sample yields a functions $f(\mathbf{x})$ (right): $f(\mathbf{x}) = \mathbf{w}\boldsymbol{\phi}(\mathbf{x})$.
- Each function is a sample from the *space* of possible functions.



Function space view

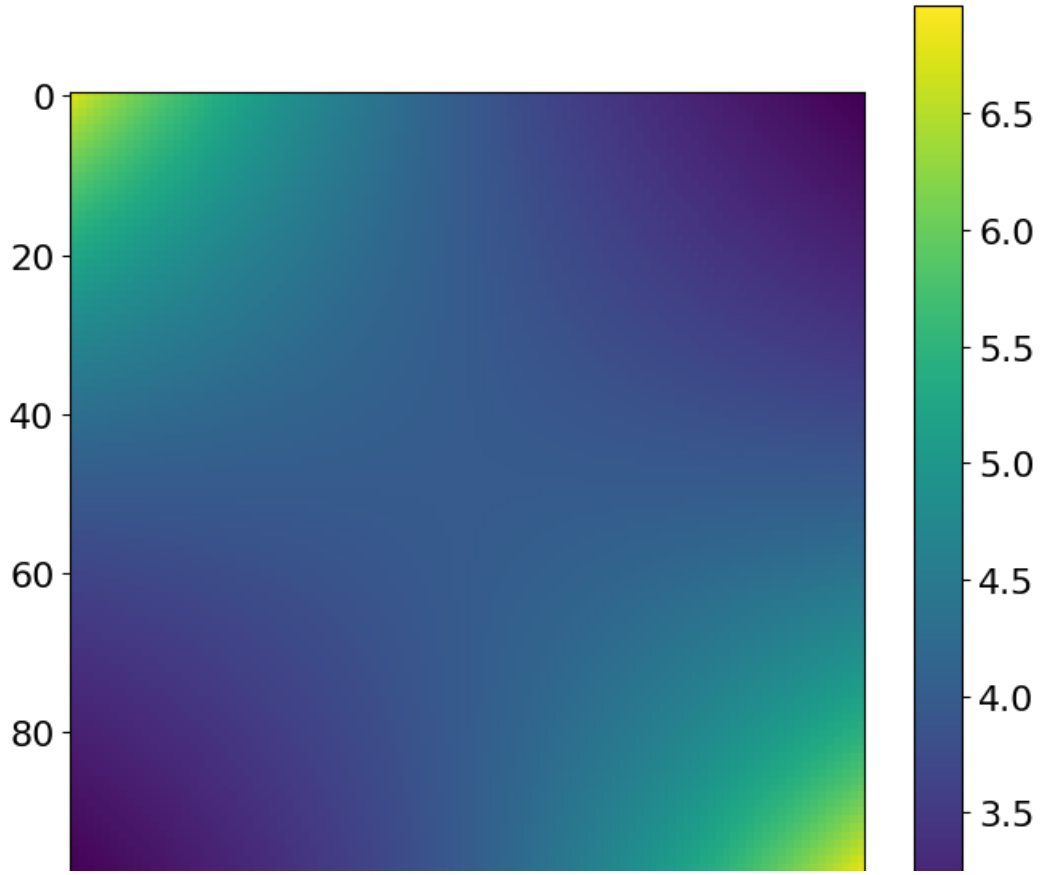
- Instead of sampling $\mathbf{w}$ and then multiplying by $\mathbf{\Phi}$, we can also generate examples of f directly.
- If $\mathbf{w}$ is sampled from a normal distribution with covariance $\alpha\mathbf{I}$ and zero mean, then $\mathbf{f}$ can be sampled from a normal distribution with zero mean and covariance matrix $\mathbf{K} = \alpha\mathbf{\Phi}\mathbf{\Phi}^\top$:

$$\mathbf{f} \sim \mathcal{N}(\mathbf{0}, \mathbf{K})$$

- The sampled functions look very similar! Indeed, they are effectively drawn from the same density.

What does covariance matrix $\mathbf{K} = \alpha \mathbf{\Phi} \mathbf{\Phi}^\top$ look like? For polynomial functions (we had about 100 data points):

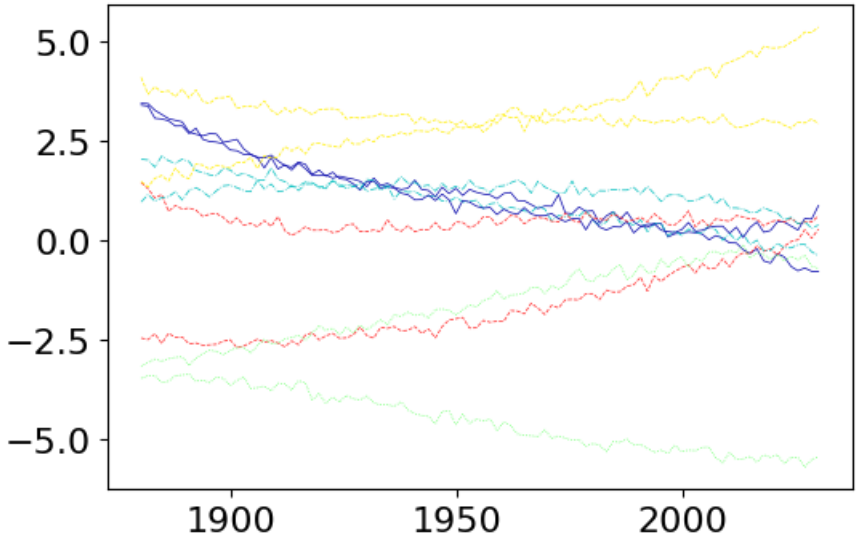
- Two nearby points can have very different values (low covariance)



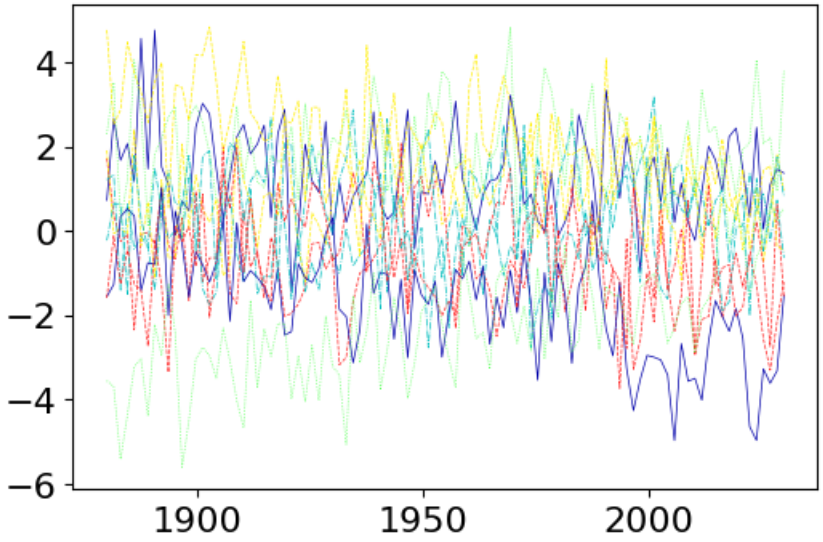
Noisy functions

We normally add Gaussian noise to obtain our observations:

$$\mathbf{y} = \mathbf{f} + \boldsymbol{\epsilon}$$



We can also increase the variance of the noise



Gaussian Process

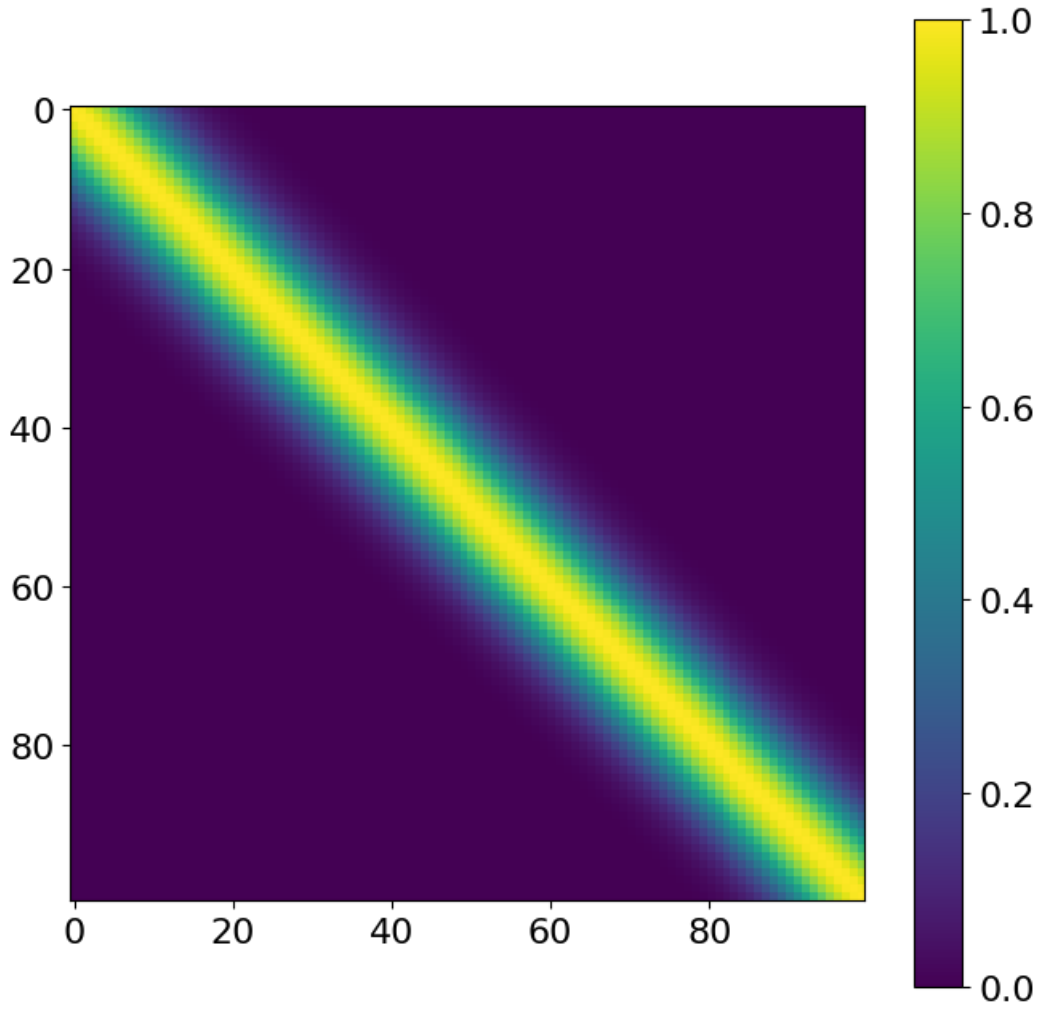
- Usually, we want our functions to be *smooth*: if two points are similar, the predictions should be similar.
    - Hence, we need a similarity measure (a kernel)
- In a Gaussian process we can do this by specifying the *covariance function* directly
    - The covariance matrix is the kernel matrix: $\mathbf{f} \sim \mathcal{N}(\mathbf{0}, \mathbf{K})$
- The RBF (Gaussian) covariance function (or *kernel*) is specified by

$$k(\mathbf{x}, \mathbf{x}') = \alpha \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2} \right).$$

where $\|\mathbf{x} - \mathbf{x}'\|^2$ is the squared distance between the two input vectors

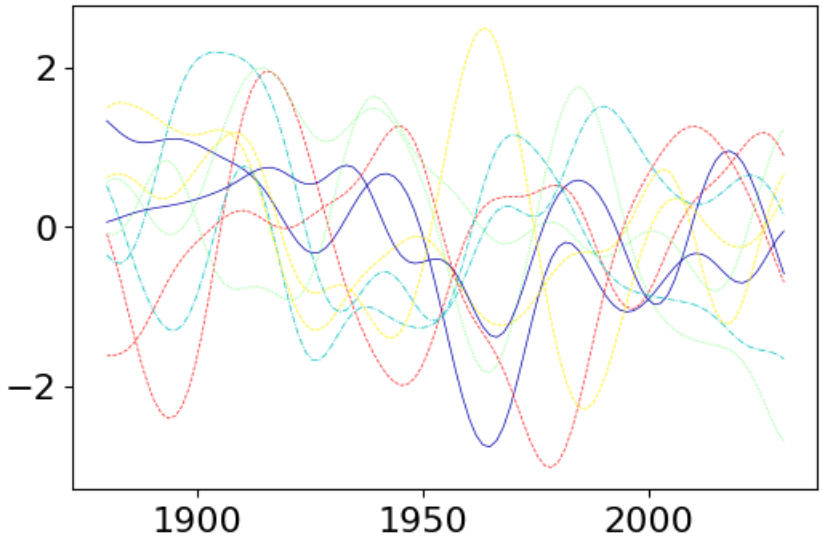
$$\left\|\mathbf{x} - \mathbf{x}'\right\|^2 = (\mathbf{x} - \mathbf{x}')^\top (\mathbf{x} - \mathbf{x}')$$

Let's build the covariance matrix for the RBF function:



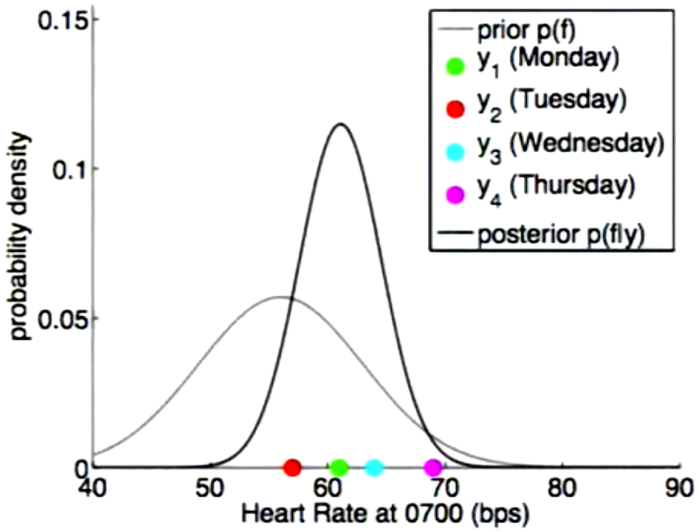
Finally, we can sample functions with this kernel (covariance matrix)

- Note that the mean is 0
- These are our priors. We now need to learn a (posterior) distribution, given data X

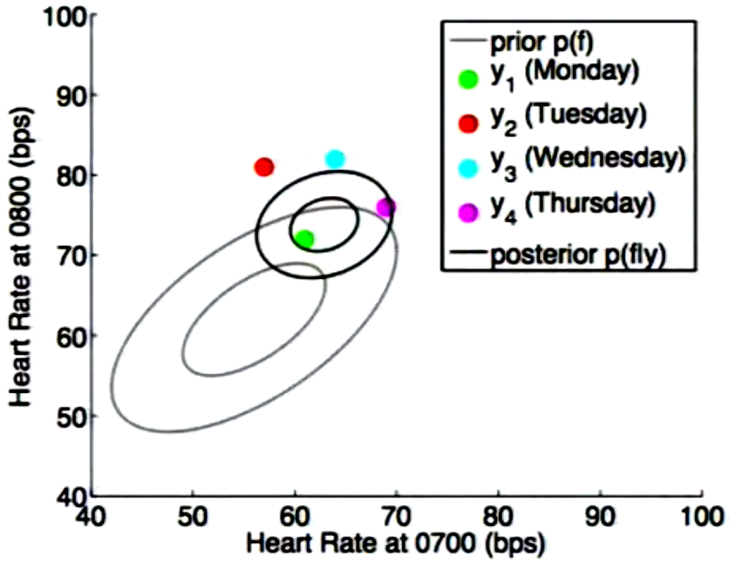


Gaussian process intuition

- Imagine you want to predict your heartrate throughout the day
- Start with predicting just your waking heart rate (at 7am)
    - We assumed a prior around 55 bpm
    - After seeing some data points, we can update this

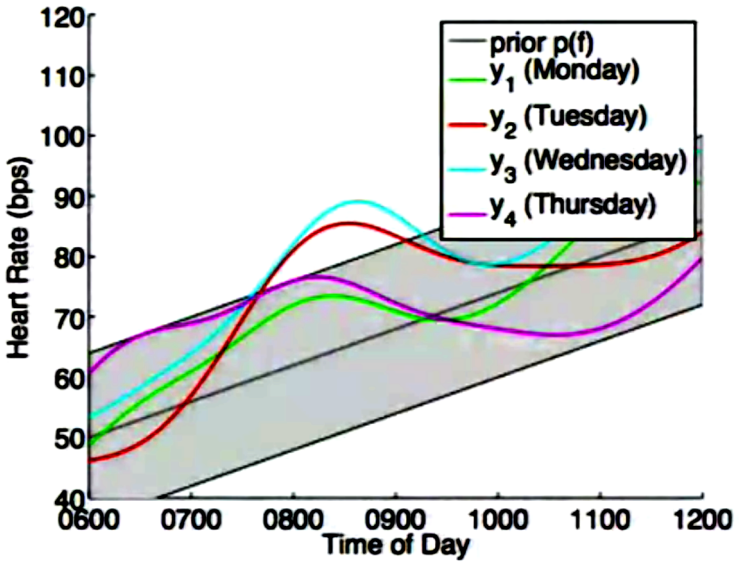


The same, but for 2 variables (heart rate at 7am and 8am)



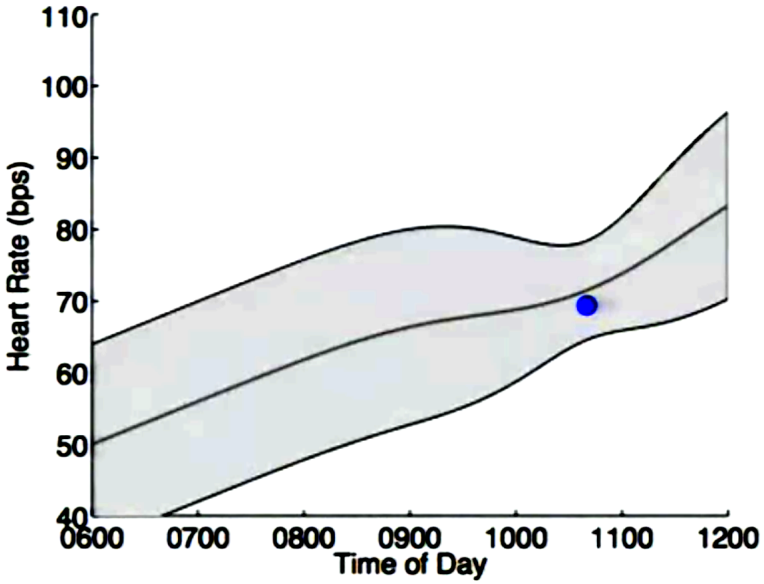
Gaussian processes: functions of infinite numbers of real valued variables

- We again assume a prior (slow increase throughout the day)
- Initially, any sampled function is equally likely

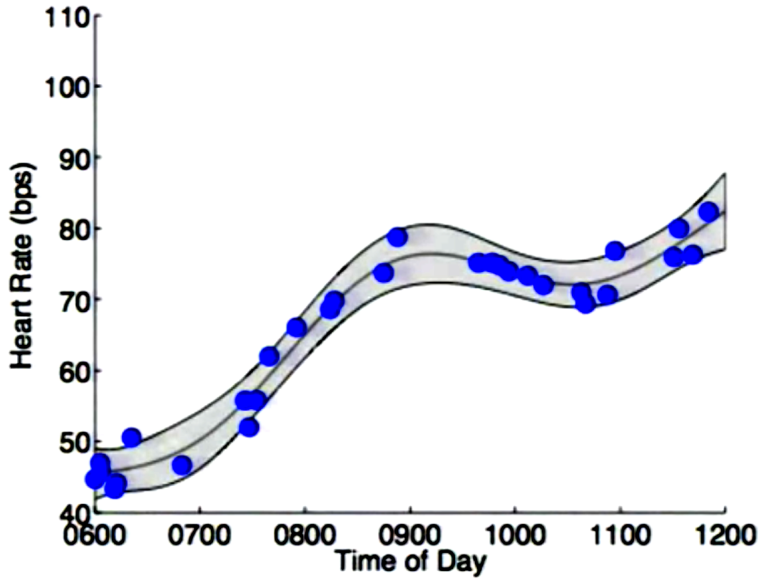


After 1 observation, we can update out posterior

* Heartrates at nearby points must be similar
* We now have increased certainty for nearby points



After many observations we have a posterior probability much much more certainty



Computing the posterior

- Assuming that $P(X)$ is a Gaussian density with a covariance given by kernel matrix $\mathbf{K}$, the model likelihood becomes:

$$p(\mathbf{y}|\mathbf{X}) = \frac{p(y)\, p(\mathbf{X} \mid y)}{p(\mathbf{X})} = \frac{1}{(2\pi)^{\frac{n}{2}}|\mathbf{K}|^{\frac{1}{2}}}\exp\left(-\frac{1}{2}\mathbf{y}^\top\left(\mathbf{K}+\sigma^2\mathbf{I}\right)^{-1}\mathbf{y}\right)$$

- Hence, the negative log likelihood (the objective function) is given by:

$$E(\boldsymbol{\theta}) = \frac{1}{2}\log|\mathbf{K}| + \frac{1}{2}\mathbf{y}^\top\left(\mathbf{K}+\sigma^2\mathbf{I}\right)^{-1}\mathbf{y}$$

- The model parameters (e.g. noise variance σ^2) and the kernel parameters (e.g. lengthscale, variance) can be embedded in the covariance function and learned from data.

- Good news: This loss function can be optimized using linear algebra (Cholesky Decomposition)

```
class GP():
    def __init__(self, X, y, sigma2, kernel, **kwargs):
        self.K = compute_kernel(X, X, kernel, **kwargs)
        self.X = X
        self.y = y
        self.sigma2 = sigma2
        self.kernel = kernel
        self.kernel_args = kwargs
        self.update_inverse()

    def update_inverse(self):
        # Precompute the inverse covariance and some quantities of
interest
        ## NOTE: Not the correct *numerical* way to compute this!
For ease of use.
        self.Kinv =
np.linalg.inv(self.K+self.sigma2*np.eye(self.K.shape[0]))
        # the log determinant of the covariance matrix.
        self.logdetK =
np.linalg.det(self.K+self.sigma2*np.eye(self.K.shape[0]))
        # The matrix inner product of the inverse covariance
        self.Kinvy = np.dot(self.Kinv, self.y)
        self.yKinvy = (self.y*self.Kinvy).sum()
```

Making predictions

The model makes predictions for $\mathbf{f}$ that are unaffected by future values of $\mathbf{f}^*$. If we think of $\mathbf{f}^*$ as test points, we can still write down a joint probability density over the training observations, $\mathbf{f}$ and the test observations, $\mathbf{f}^*$.

This joint probability density will be Gaussian, with a covariance matrix given by our covariance function, $k(\mathbf{x}_i, \mathbf{x}_j)$.

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^\top & \mathbf{K}_{*,*} \end{bmatrix}\right)$$

where $\mathbf{K}$ is the covariance computed between all the training points,
$\mathbf{K}_*$ is the covariance matrix computed between the training points and the test points,
$\mathbf{K}_{*,*}$ is the covariance matrix computed between all the tests points and themselves.

Conditional Density

Just as in naive Bayes, we defined the joint density (although there it was over both the labels and the inputs, $p(\mathbf{y}, \mathbf{X})$ and now we need to define *conditional* distributions that answer particular questions of interest.

We will need the *conditional density* for making predictions.

$$\mathbf{f}^*|\mathbf{y} \sim \mathcal{N}(\boldsymbol{\mu}_f, \mathbf{C}_f)$$

with a mean given by

$$\boldsymbol{\mu}_f = \mathbf{K}_*^\top \left[\mathbf{K} + \sigma^2\mathbf{I}\right]^{-1}\mathbf{y}$$

and a covariance given by

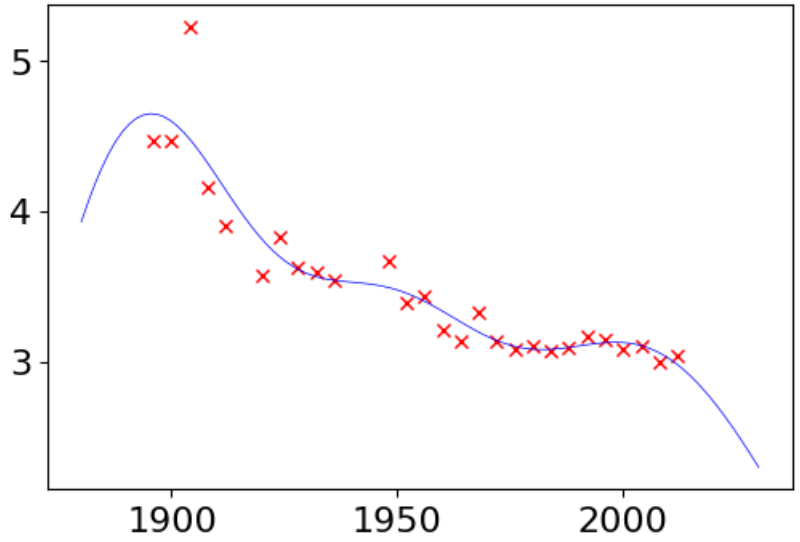
$$\mathbf{C}_f = \mathbf{K}_{*,*} - \mathbf{K}_*^\top \left[\mathbf{K} + \sigma^2\mathbf{I}\right]^{-1}\mathbf{K}_*.$$

Let's compute what those posterior predictions are for the olympic marathon data.

We can now get the mean and covariance:

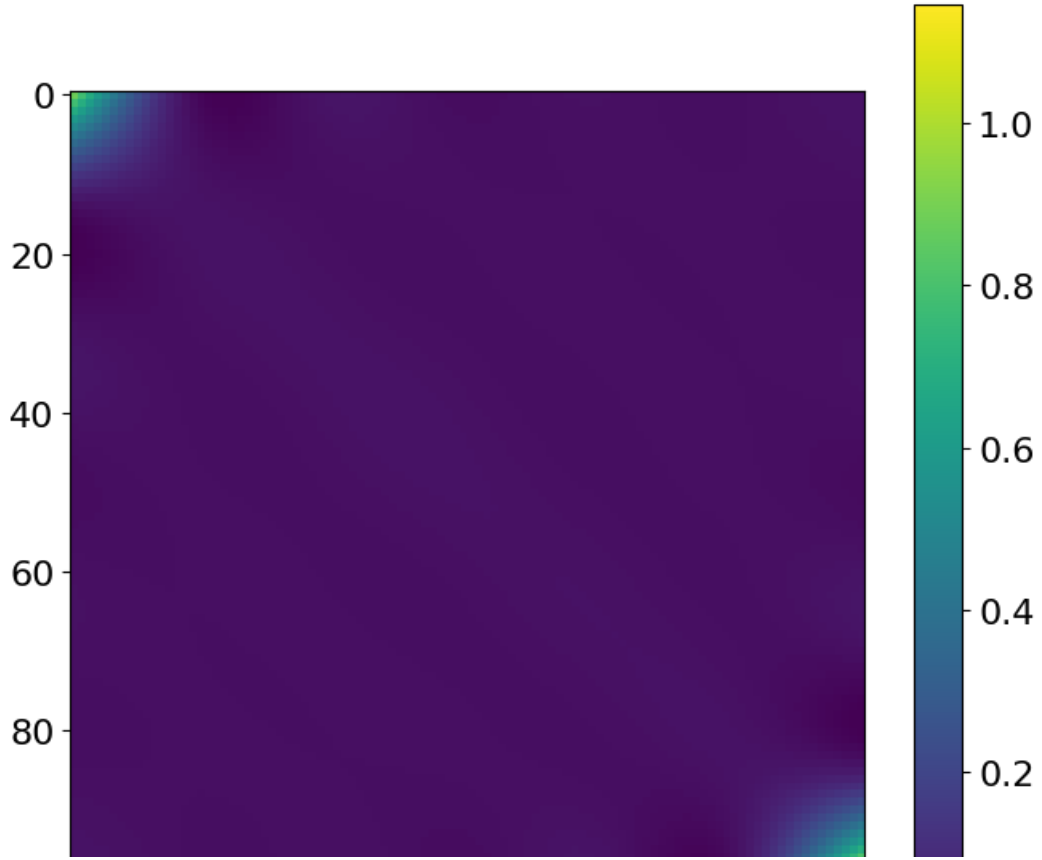
```
model = GP(x, y, sigma2, exponentiated_quadratic, variance=16.0,
lengthscale=32)
mu_f, C_f = model.posterior_f(x_pred)
```

Plot the mean:



The covariance looks like this:

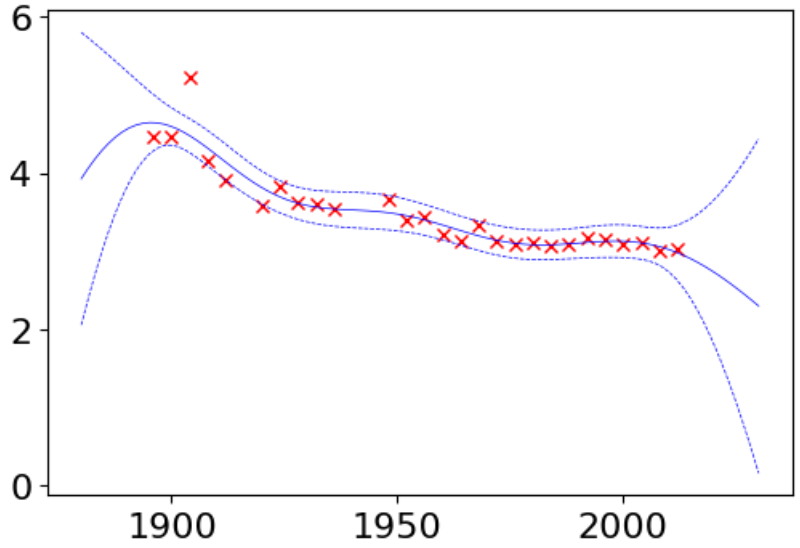
- Look along the diagonal
- High variance at the beginning and the end



Hence, we get:

- High uncertainty and the beginning and the end
- Very low uncertainty in the middle

```
var_f = np.diag(C_f)[:, None]
std_f = np.sqrt(var_f)
```



Gaussian Processes with GPy

- `GPyRegression`
- Generate a kernel first
    - State the dimensionality of your input data
    - Variance and lengthscale are optional, default = 1

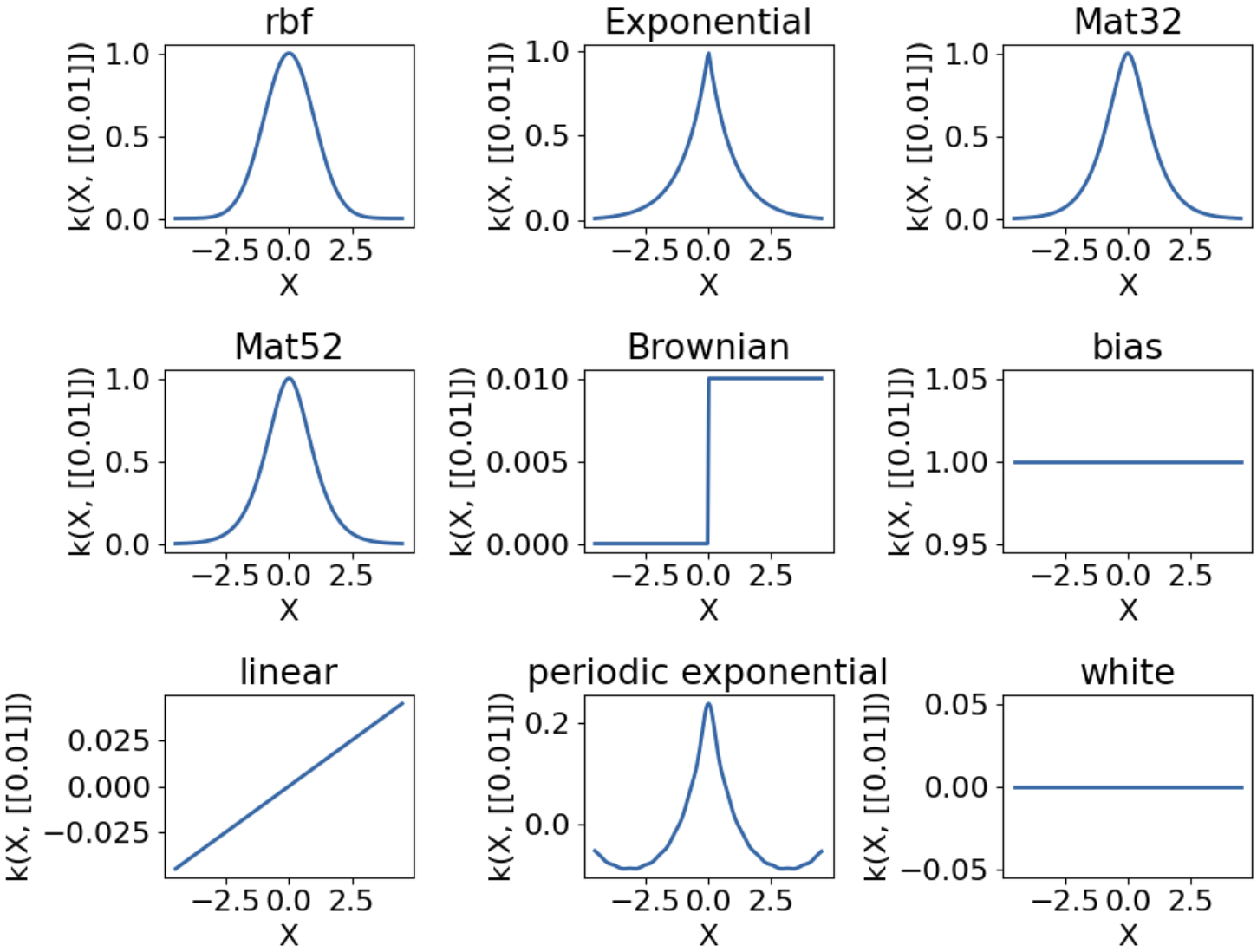
      ```
      kernel = GPy.kern.RBF(input_dim=1, variance=1.,
      lengthscale=1.)
      ```

    - Other kernels:

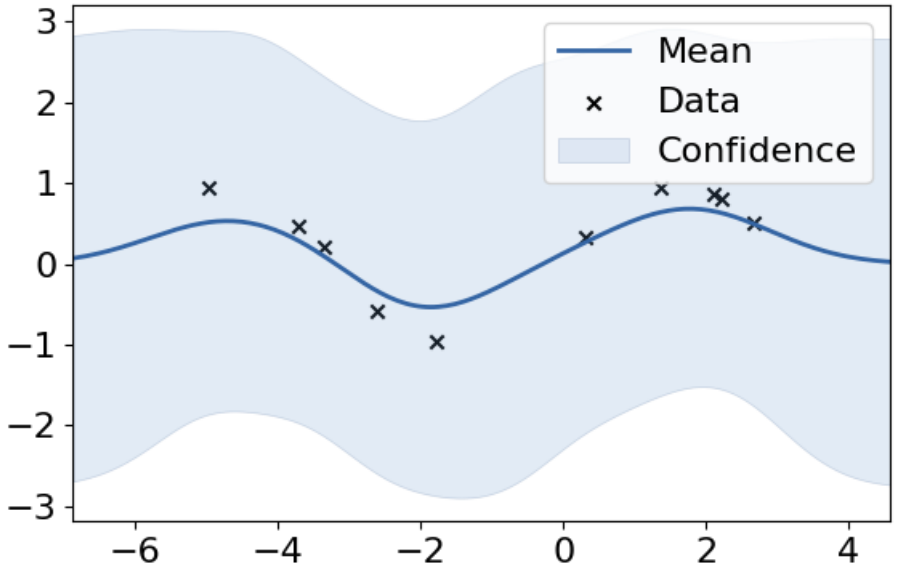
      ```
      GPy.kern.BasisFuncKernel?
      ```

- Build model:

  ```
  m = GPy.models.GPRegression(X,Y,kernel)
  ```

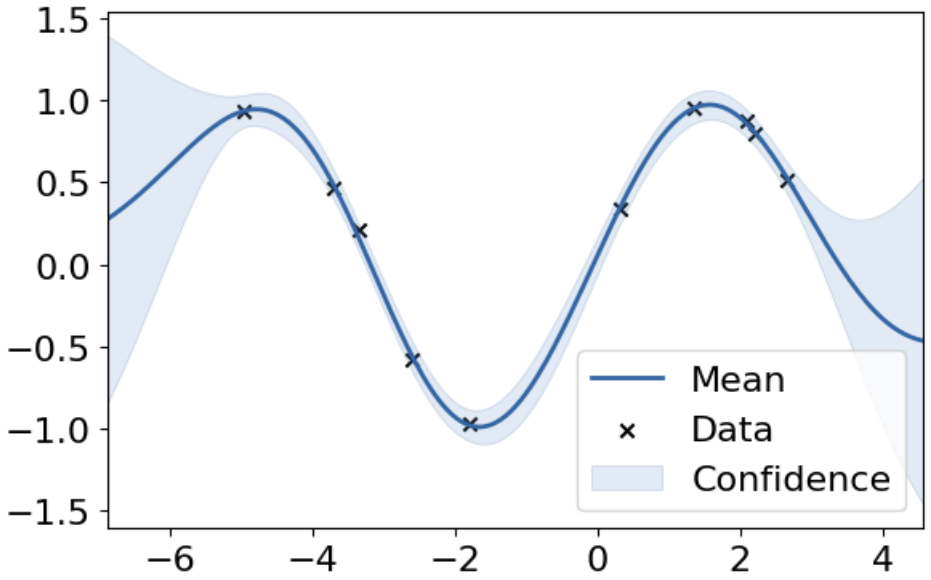

rbf
Exponential
Mat32
Mat52
Brownian
bias
linear
periodic exponential
white
k(X, [[0.01]])
X

Build the untrained GP. The shaded region corresponds to ~95% confidence intervals (i.e. +/- 2 standard deviation)

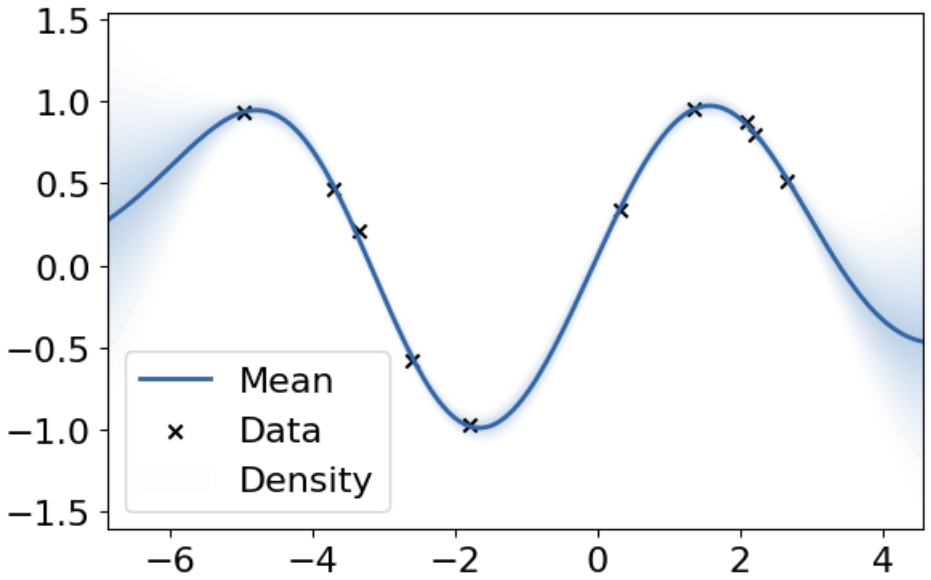


Train the model (optimize the parameters): maximize the likelihood of the data.
Best to optimize with a few restarts: the optimizer may converges to the high-noise solution. The optimizer is then restarted with a few random initialization of the parameter values.

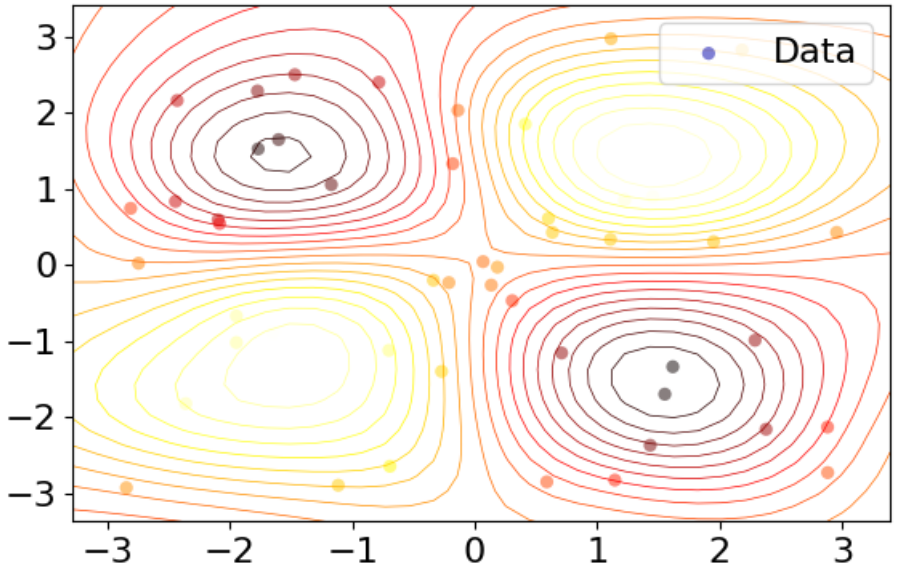
```
Optimization restart 1/3, f = -0.4226226481695612
Optimization restart 2/3, f = -0.42262264817280926
Optimization restart 3/3, f = -0.42262264817276307
```



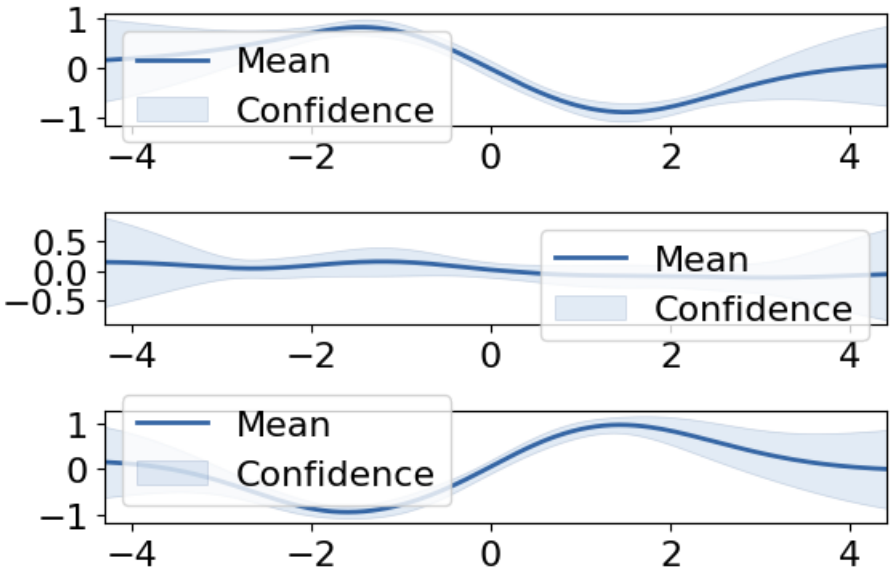
You can also plot densities



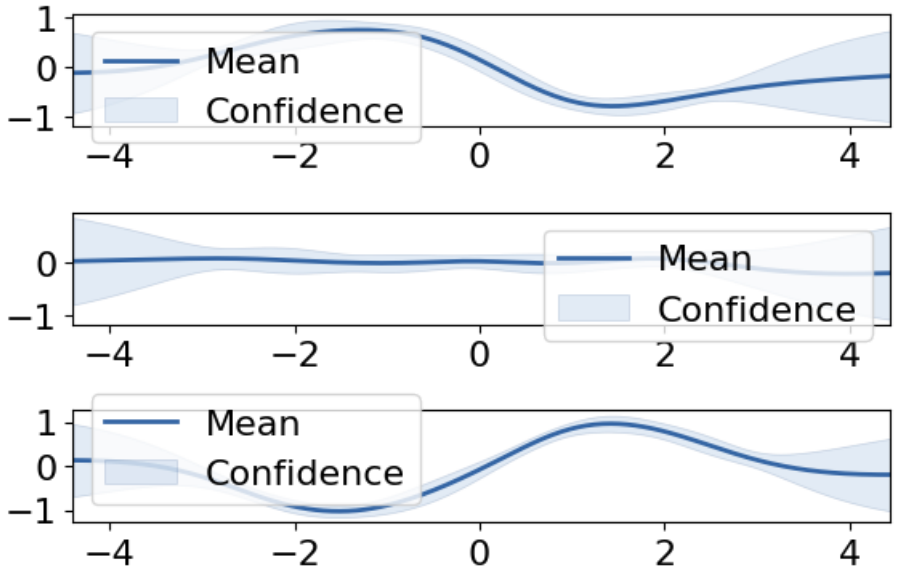
You can also show results in 2D



We can plot 2D slices using the `fixed_inputs` argument to the plot function.
`fixed_inputs` is a list of tuples containing which of the inputs to fix, and to which value.



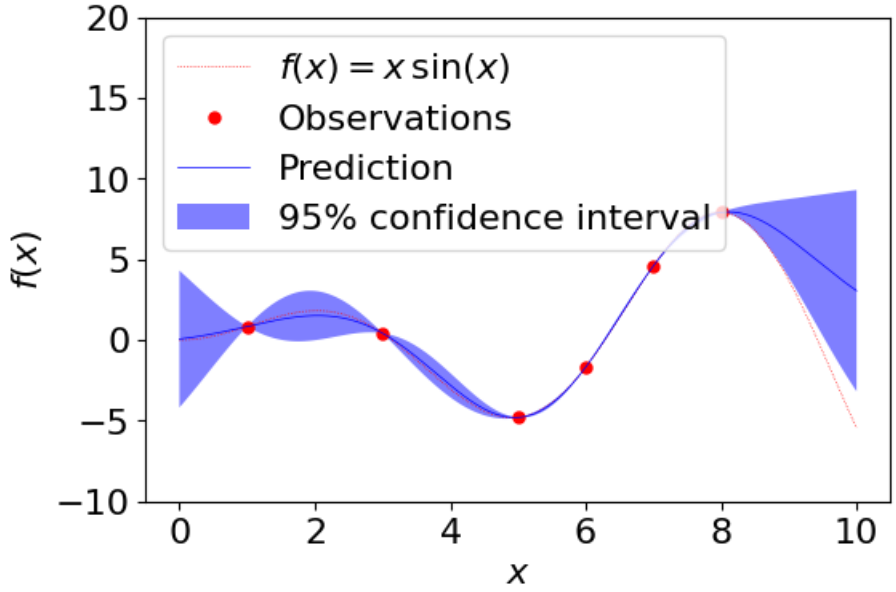
For vertical slices, simply fix the other input: `fixed_inputs=[(0,y)]`



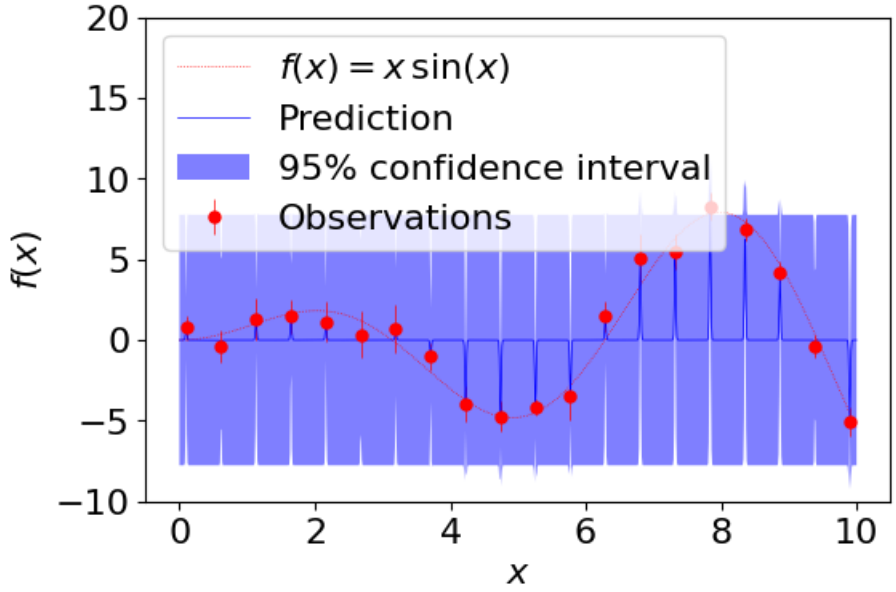
Gaussian Processes with scikit-learn

- `GaussianProcessRegressor`
- Hyperparameters:
    - `kernel`: kernel specifying the covariance function of the GP
        - Default: "1.0 * RBF(1.0)"
        - Typically leave at default. Will be optimized during fitting
    - `alpha`: regularization parameter
        - Tikhonov regularization of the assumed covariance between the training points.
        - Adds a (small) value to the diagonal of the kernel matrix during fitting.
        - Larger values:
            - correspond to increased noise level in the observations
            - also reduce potential numerical issues during

Example


$f(x) = x\sin(x)$
Observations
Prediction
95% confidence interval
$f(x)$
x
20
15
10
5
0
−5
−10
0
2
4
6
8
10

Example with noisy data



Gaussian processes: Conclusions

The advantages of Gaussian processes are:

- The prediction interpolates the observations (at least for regular kernels).
- The prediction is probabilistic (Gaussian) so that one can compute empirical confidence intervals.
- Versatile: different kernels can be specified.

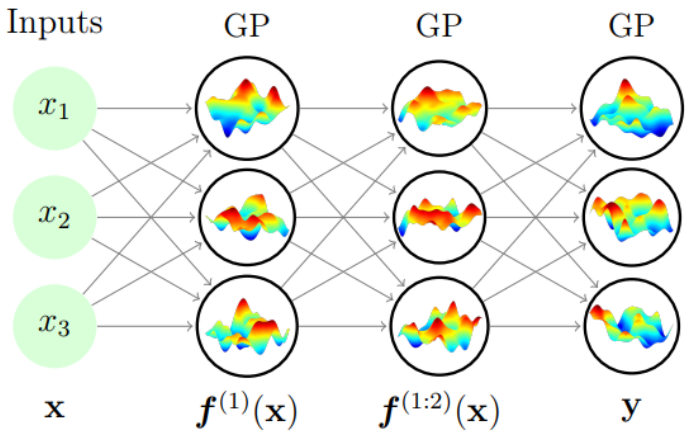
The disadvantages of Gaussian processes include:

- They are not sparse, i.e., they use the whole samples/features information to perform the prediction.
- They lose efficiency in high dimensional spaces – namely when the number of features exceeds a few dozens.

Gaussian processes and neural networks

- You can prove that a Gaussian process is equivalent to a neural network with one layer and an infinite number of nodes
- You can build *deep Gaussian Processes* by constructing layers of GPs

A net with nonparametric activation functions corresponding to a 3-layer deep GP

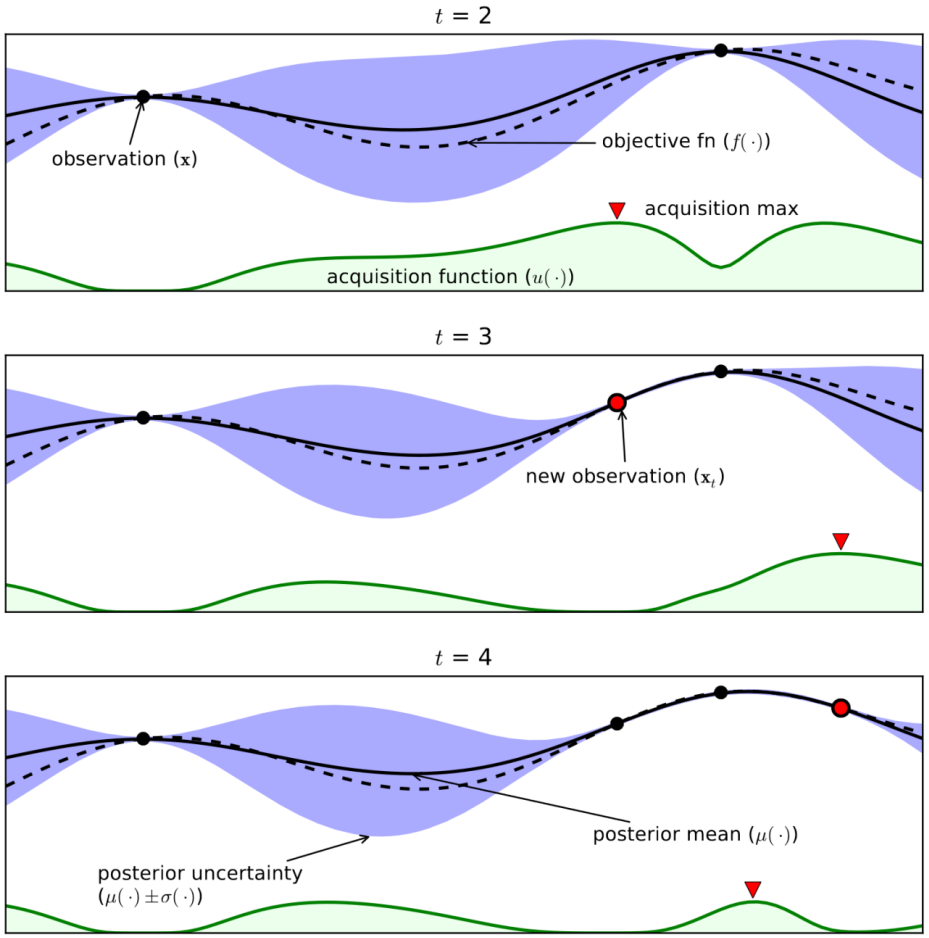


Bayesian optimization

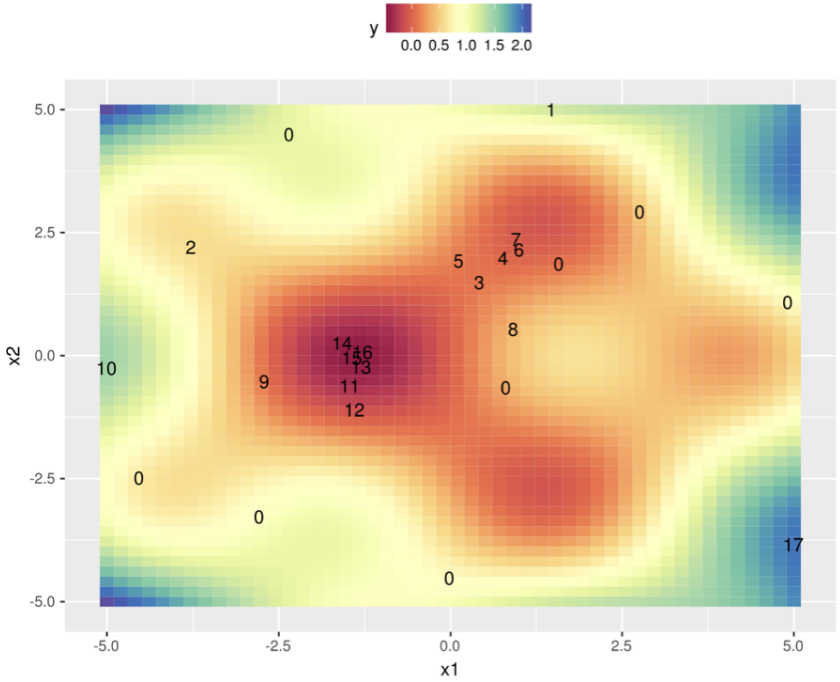
- The incremental updates you can do with Bayesian models allow a more effective way to optimize functions
    - E.g. to optimize the hyperparameter settings of a machine learning algorithm/pipeline
- After a number of random search iterations we know more about the performance of hyperparameter settings on the given dataset
- We can use this data to train a model, and predict which other hyperparameter values might be useful
    - More generally, this is called model-based optimization
    - This model is called a *surrogate model*
- This is often a probabilistic (e.g. Bayesian) model that predicts confidence intervals for all hyperparameter settings
- We use the predictions of this model to choose the next point to evaluate
- With every new evaluation, we update the surrogate model and repeat

Example (see figure):

- Consider only 1 continuous hyperparameter (X-axis)
    - You can also do this for many more hyperparameters
- Y-axis shows cross-validation performance
- Evaluate a number of random hyperparameter settings (black dots)
    - Sometimes an initialization design is used
- Train a model, and predict the expected performance of other (unseen) hyperparameter values
    - Mean value (black line) and distribution (blue band)
- An *acquisition function* (green line) trades off maximal expected performace and maximal uncertainty
    - Exploitation vs exploration
- Optimal value of the asquisition function is the next hyperparameter setting to be evaluated
- Repeat a fixed number of times, or until time budget runs out


t = 2
observation ($\mathbf{x}$)
objective fn ($f(\cdot)$)
acquisition max
acquisition function ($u(\cdot)$)
t = 3
new observation ($\mathbf{x}_t$)
t = 4
posterior mean ($\mu(\cdot)$)
posterior uncertainty
($\mu(\cdot) \pm \sigma(\cdot)$)

In 2 dimensions:



Surrogate models

- Surrogate model can be anything as long as it can do regression and is probabilistic
- Gaussian Processes are commonly used
    - Smooth, good extrapolation, but don't scale well to many hyperparameters (cubic)
    - Sparse GPs: select ‘inducing points’ that minimize info loss, more scalable
    - Multi-task GPs: transfer surrogate models from other tasks
- Random Forests
    - A lot more scalable, but don't extrapolate well
    - Often an interpolation between predictions is used instead of the raw (step-wise) predictions
- Bayesian Neural Networks:
    - Expensive, sensitive to hyperparameters

Acquisition Functions

- When we have trained the surrogate model, we ask it to predict a number of samples
    - Can be simply random sampling
    - Better: *Thompson sampling*
        - fit a Gaussian distribution (a mixture of Gaussians) over the sampled points
        - sample new points close to the means of the fitted Gaussians
- Typical acquisition function: *Expected Improvement*
    - Models the predicted performance as a Gaussian distribution with the predicted mean and standard deviation
    - Computes the *expected* performance improvement over the previous best configuration $\mathbf{X}^+$:

$$EI(X) := \mathbb{E}\left[\max\{0, f(\mathbf{X}^+) - f_{t+1}(\mathbf{X})\}\right]$$

Bayesian Optimization: conclusions

- More efficient way to optimize hyperparameters
- More similar to what humans would do
- Harder to parallellize
- Choice of surrogate model depends on your search space
    - Very active research area
    - For very high-dimensional search spaces, random forests are popular