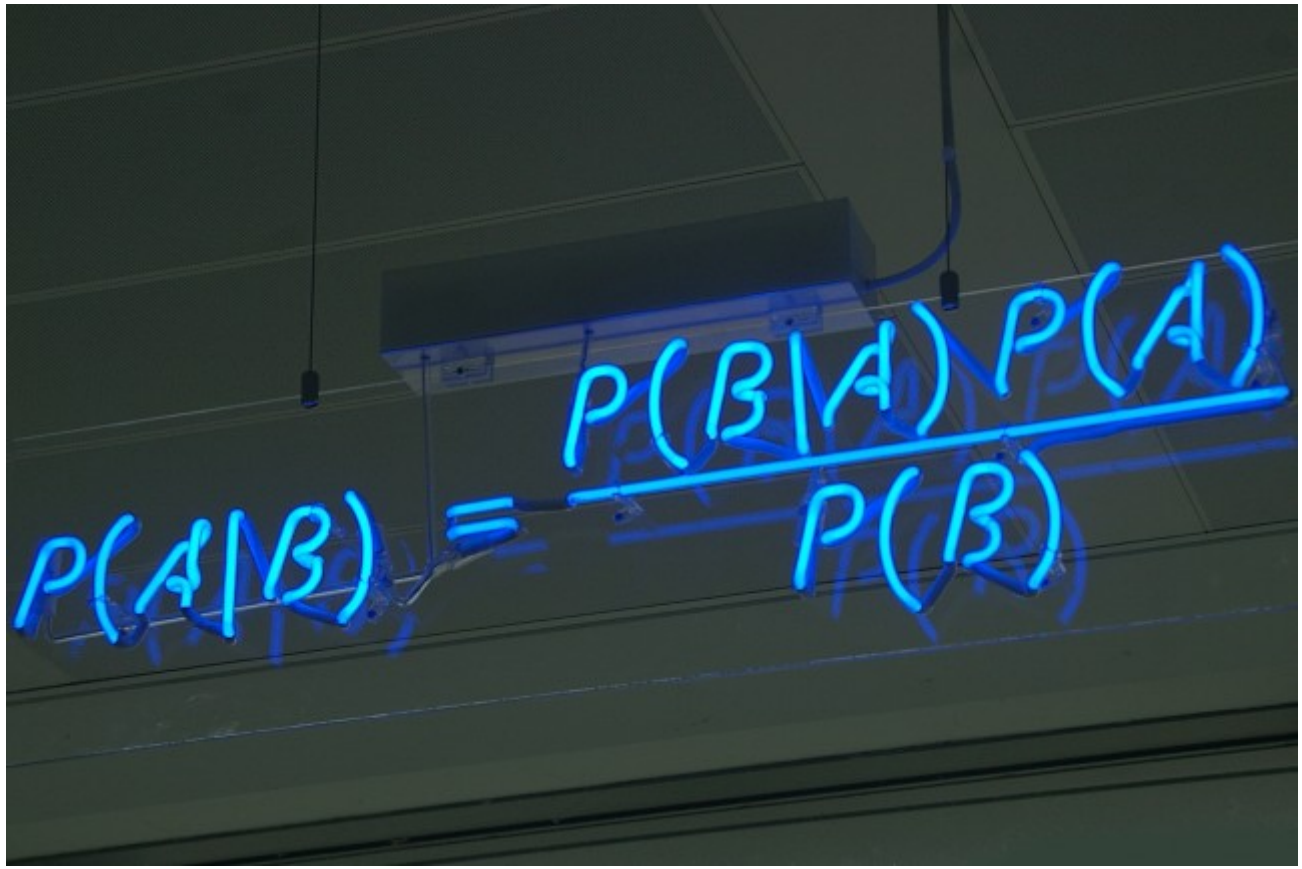
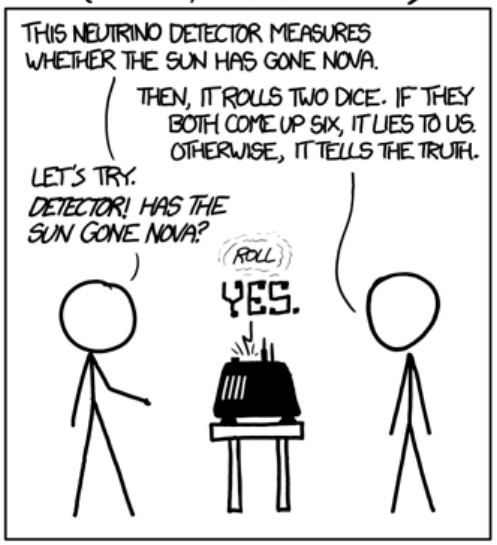
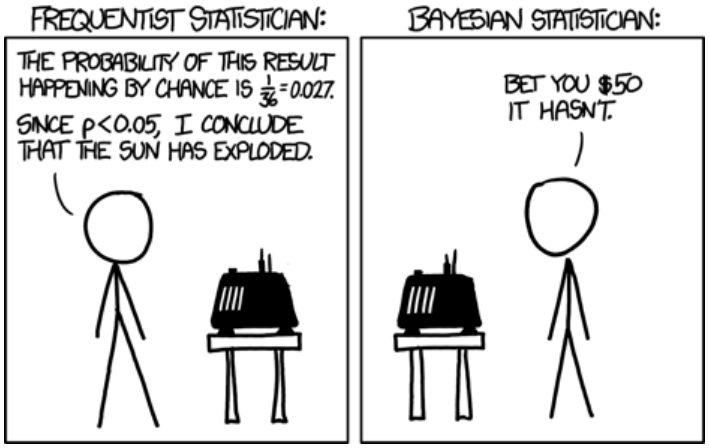
Bayesian Learning



DID THE SUN JUST EXPLODE?

(IT'S NIGHT, SO WE'RE NOT SURE.)





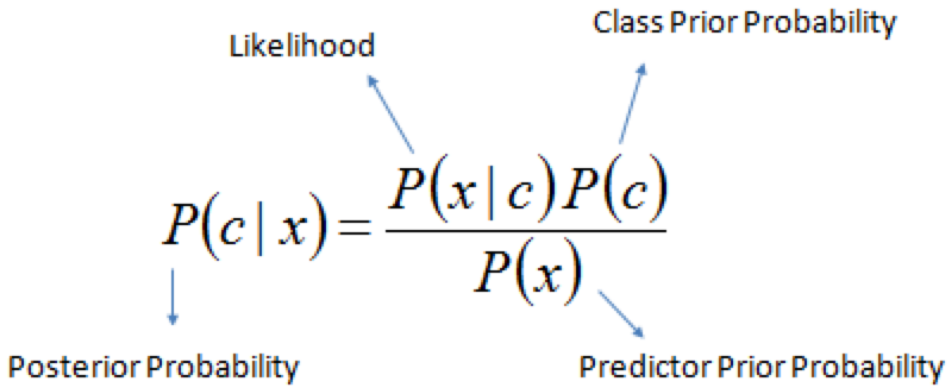
Naive Bayes

Predict the probability that a point belongs to a certain class, using Bayes' Theorem, assuming that the features are independent from each other.

Very fast: only needs to extract statistics from each feature.

Naive Bayes Classifier

A Naive Bayes classifier learns the joint probability $P(x, c) = P(x|c)P(c)$ of the data, and predicts the class of each sample using Bayes' rule:



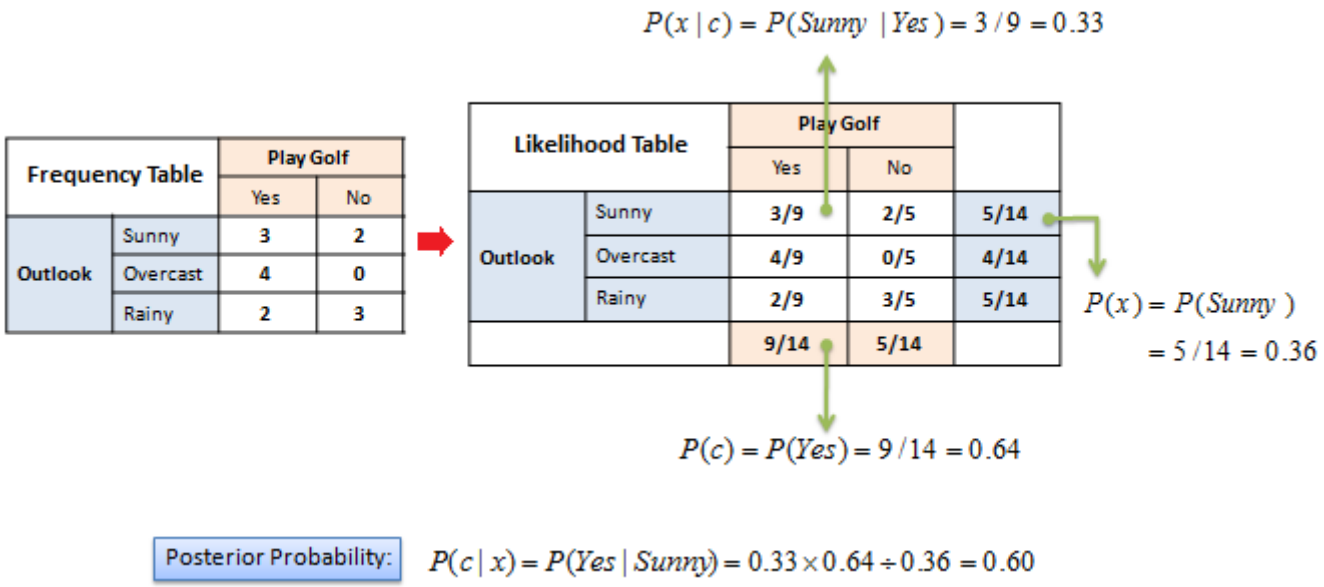
$P(c|x)$ is the posterior probability of class (target) given predictor (attribute).
$P(c)$ is the *prior* probability of class: what you believed before you saw the evidence x
$P(x|c)$ is the *likelihood* of seeing that evidence if your class is correct
$P(x)$ is the prior probability of predictor (*marginal likelihood*): the likelihood of the evidence x under any circumstance

Naive Bayes assumes that all features are conditionally independent from each other, in which case:
$P(\mathbf{x}|c) = P(x_1|c) \times P(x_2|c) \times \ldots \times P(x_n|c)$

Bayesian Networks are another family of algorithms in which the conditional dependencies between features is modelled explictly. We can't discuss them right now, though.

Since Bayesian models model the entire joint distribution, they can generate new (likely) points: *generative model*

Example. True or not? Players will play if weather is sunny.



Compute the posterior for every class and predict the class with highest probability

On numeric data

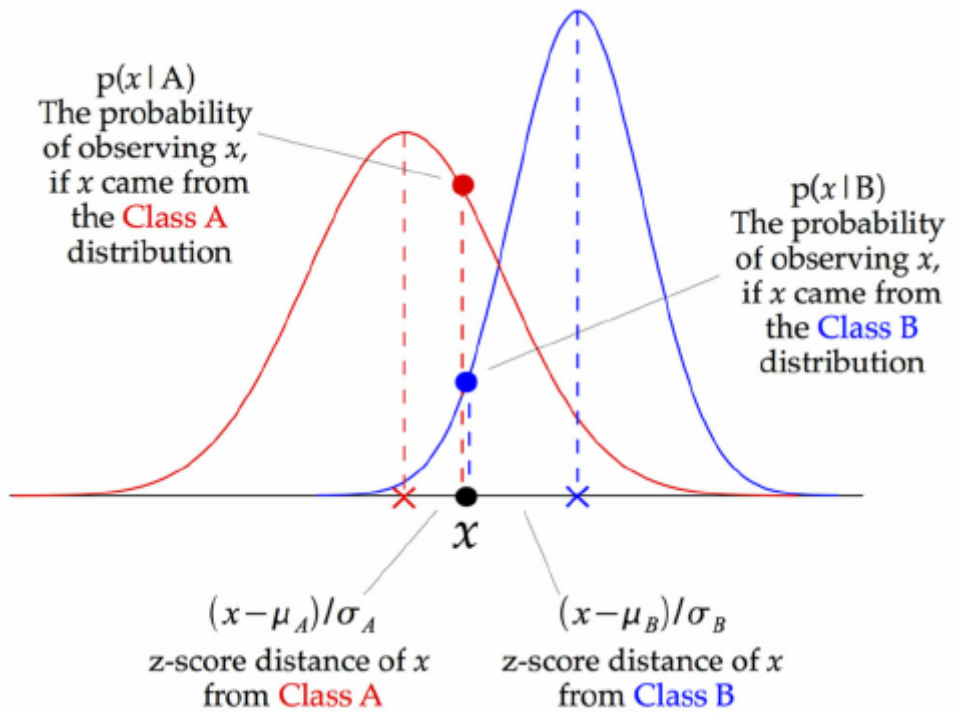
GaussianNB:

- Computes mean μ_c and standard deviation σ_c of the feature values per class
- It then fits a Gaussian distribution around the mean

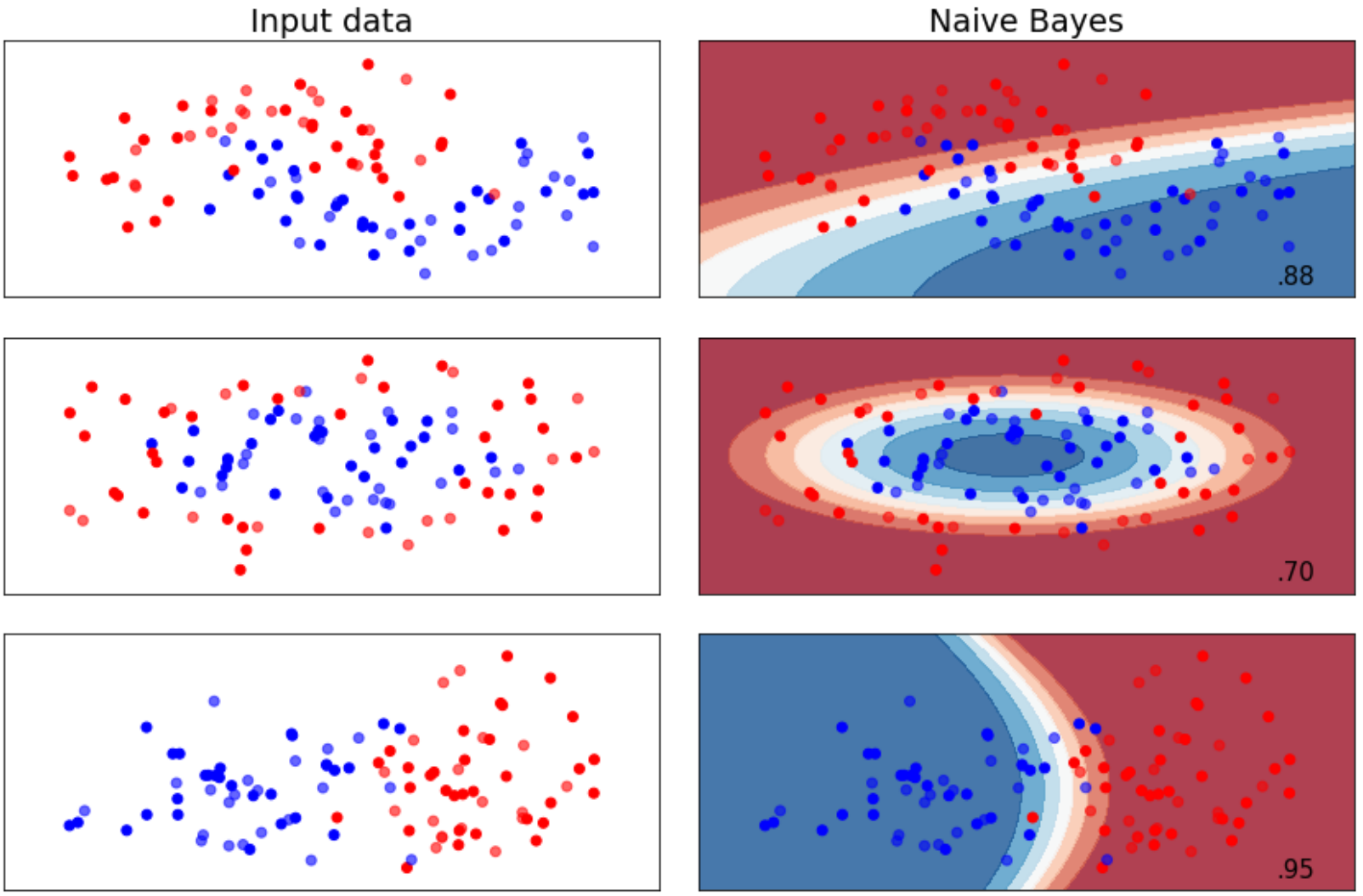
$$p(x = v \mid c) = \frac{1}{\sqrt{2\pi\sigma_c^2}}\, e^{-\frac{(v-\mu_c)^2}{2\sigma_c^2}}$$

- Prediction are made using Bayes' theorem, by computing the joint probability given all features, and calculating the posterior:

$$p(c \mid \mathbf{x}) = \frac{p(c)\, p(\mathbf{x}|c)}{p(\mathbf{x})}$$


p(x | A)
The probability of observing x, if x came from the Class A distribution
p(x | B)
The probability of observing x, if x came from the Class B distribution
x
$(x-\mu_A)/\sigma_A$
z-score distance of x from Class A
$(x-\mu_B)/\sigma_B$
z-score distance of x from Class B

Visualizing Naive Bayes



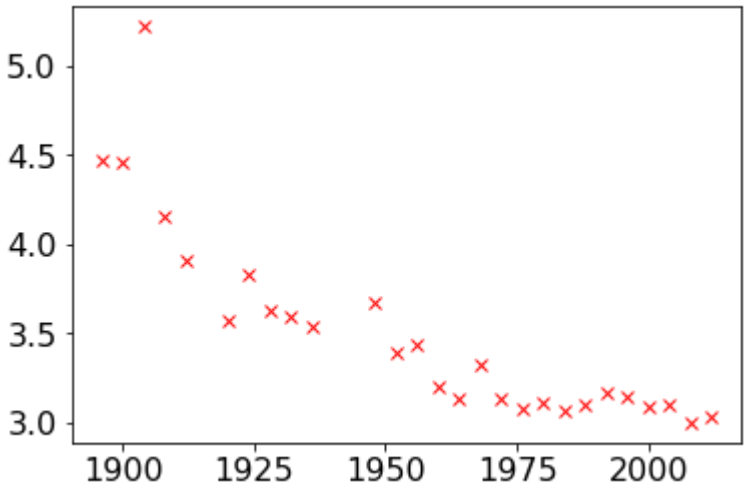
Other Naive Bayes classifiers:

- 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)

Probabilistic interpretation of regression

Let's look at a the following regression problem



Let's first try to fit a linear model

$$y = f(\mathbf{x}_i) = \mathbf{x}_i^\top \mathbf{w}$$

We can solve this via linear algebra by making a design matrix of the data, which includes the $x_0 = 1$ column, to represent the bias. Hence, each vector $\mathbf{x}_i$ is given by appending a 1 onto the original vector

$$\mathbf{x}_i = \begin{bmatrix} 1 \\ x_i \end{bmatrix}$$

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

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

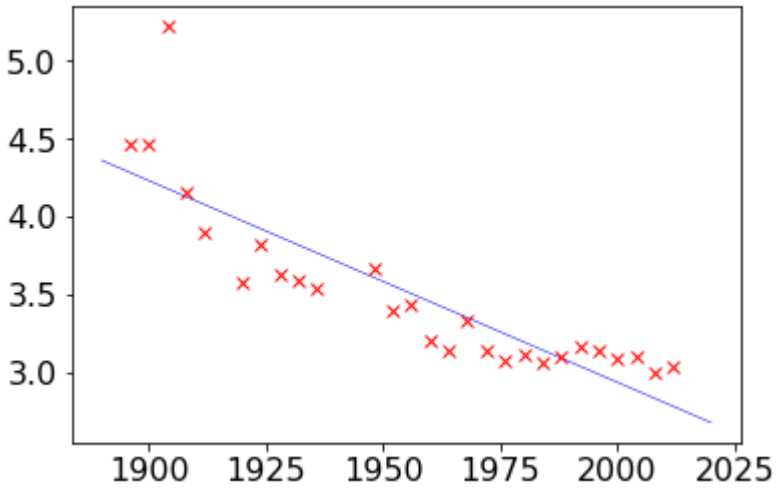
In numpy this is done and solved with the following commands:

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

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

With $w = [w_0, w_1]$, we can now fit the function

$$y = w_1 x + w_0 = -0.013x + 28.895$$

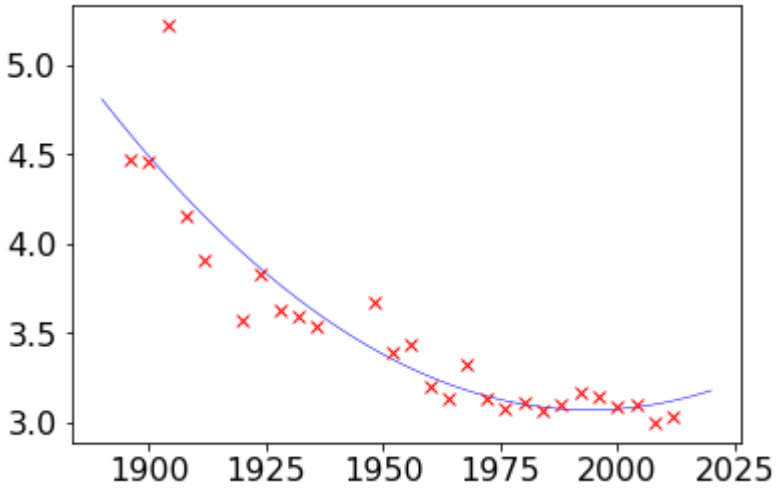


The linear model isn't a very good fit, let's try fitting a 2nd degree polynomial (quadratic model). This can be done by adding more *basis functions*.
Firstly, we need to create a new design matrix that contains the quadratic basis,

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

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

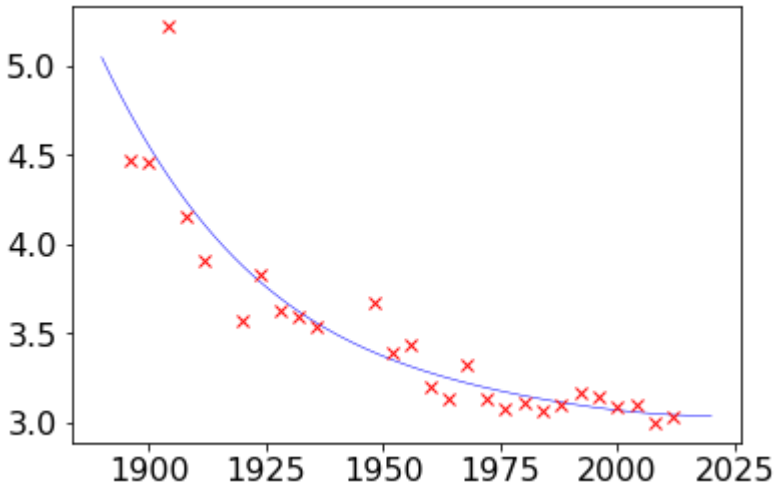
```
w: [[643.642  -0.643   0.   ]]
```



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: [[63919.611   -88.864     0.027     0.       -0.       -0.        0.
]]
```

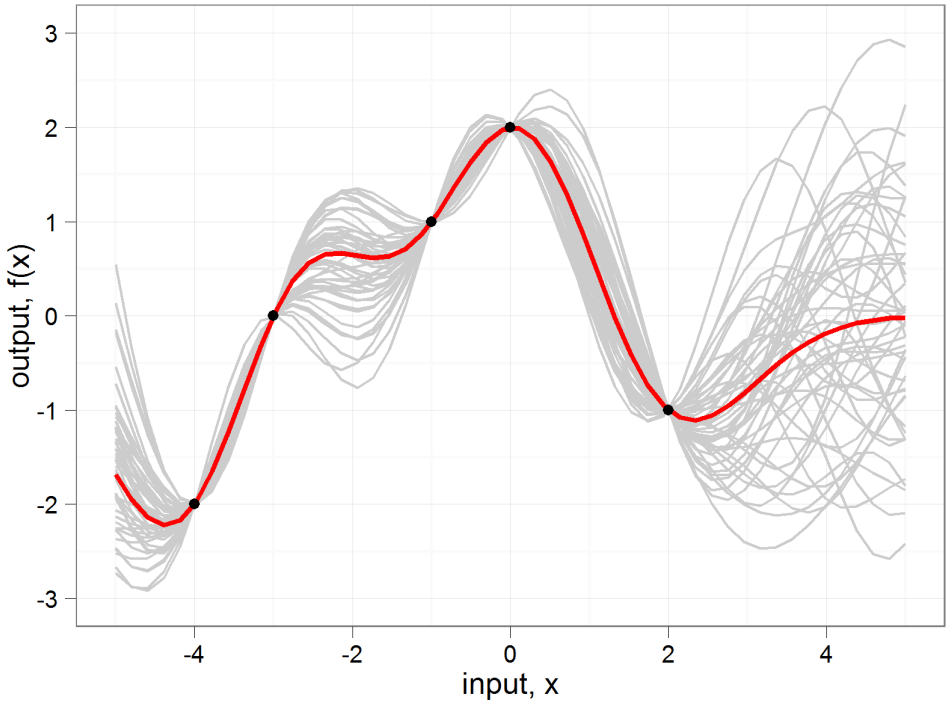


There must be a better way

- 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?
  - Ideally, I don't want to specify this up front, and consider *every possible function* that matches out data, with however many parameters are involved.
  - This is called a non-parametric solution (note: it's not that there aren't parameters, but there are infinitely many parameters)
  - A *Gaussian process* learns a *distribution* on this set of functions (which we can then sample from)

Gaussian processes

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



Probabilistic interpretation of regression

When there are more observations than unknowns (overdetermined systems), we cannot perfectly fit

$$y = w_1 x + w_0$$

This issue can be solved by assuming that the data in inherently uncertain, and model it explictly by introducing a type of slack variable (http://en.wikipedia.org/wiki/Slack_variable), ϵ_i, known as noise.

For each observation we now have the equation

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

The slack variable represented the difference between our actual prediction and the true observation. This is also known as the *residual*.

We now have an additional n variables to estimate, one for each data point, $\{\epsilon_i\}$. With the original w_1 and w_0 we now have $n + 2$ parameters to be estimated from n observations (underdetermined system).

We can however make assumptions about the noise distribution, i.e. that the slack variables are distributed according to a probability density. One often assumes Gaussian noise:

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

with zero mean and variance σ^2.

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, each element of the parameters vector, w_i, was drawn from a Gaussian density with variance α

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

Gaussian process model parameters:

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

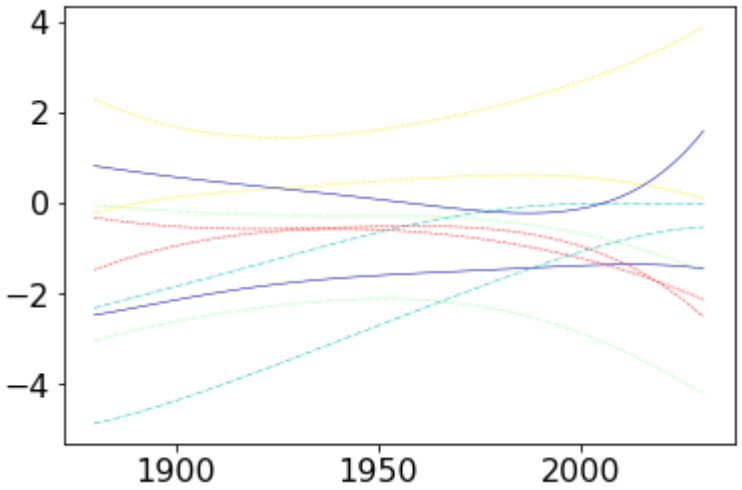
```
alpha = 4.
degree = 5
sigma2 = 0.01
```

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

```
x_pred = np.linspace(1880, 2030, 100)[:, None] # sample
Phi_pred = polynomial(x_pred, degree=degree, loc=loc, scale=scale) # pre
dict
```

Weight Space View

We assume that the parameters are drawn independently from a Gaussian density $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \alpha\mathbf{I})$, then our prediction function $f(\mathbf{x})$ becomes
$f(\mathbf{x}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x})$.
We can now sample from the prior density to obtain a vector $\mathbf{w}$ using the function `np.random.normal` and combine these parameters with our basis to create some samples of what $f(\mathbf{x})$ looks like. This is a sample from the *space* of possible models.



Function space view

We can also generate examples of f directly.

We know that if $\mathbf{w}$ is sampled from a multivariate Gaussian with covariance $\alpha\mathbf{I}$ and zero mean, then assuming that $\mathbf{\Phi}$ is a deterministic matrix (i.e. it is not sampled from a probability density) then the vector $\mathbf{f}$ will also be distributed according to a zero mean multivariate normal as follows,

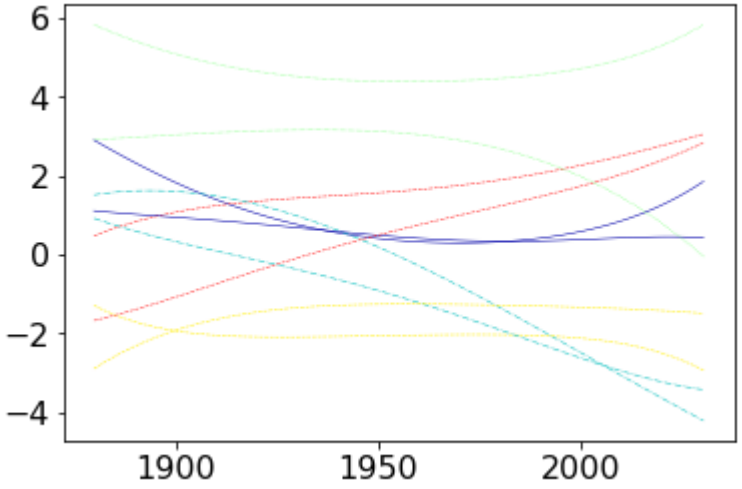
$$\mathbf{f} \sim \mathcal{N}(\mathbf{0}, \alpha\mathbf{\Phi}\mathbf{\Phi}^\top).$$

What happens if we sample $\mathbf{f}$ directly from this density, rather than first sampling $\mathbf{w}$ and then multiplying by $\mathbf{\Phi}$.

Let's try this. First of all we define the covariance (joined variability between 2 variables) as

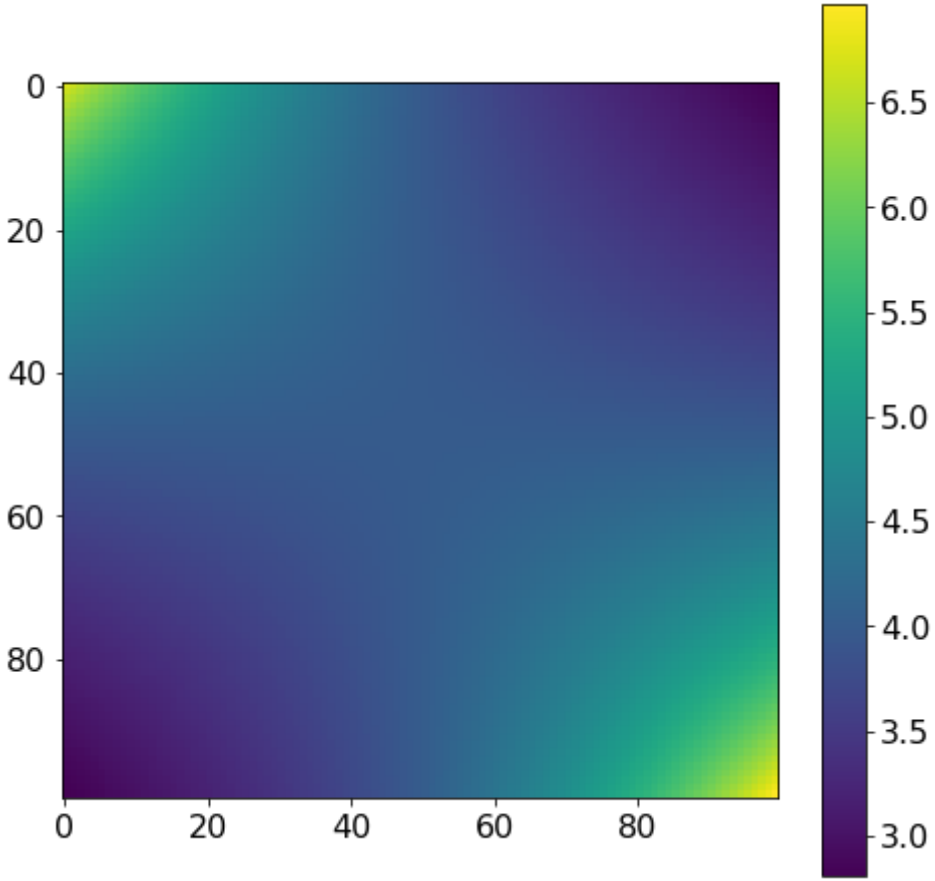
$$\mathbf{K} = \alpha\mathbf{\Phi}\mathbf{\Phi}^\top.$$

We can use `np.random.multivariate_normal` for sampling from a multivariate normal with covariance given by $\mathbf{K}$ and zero mean,



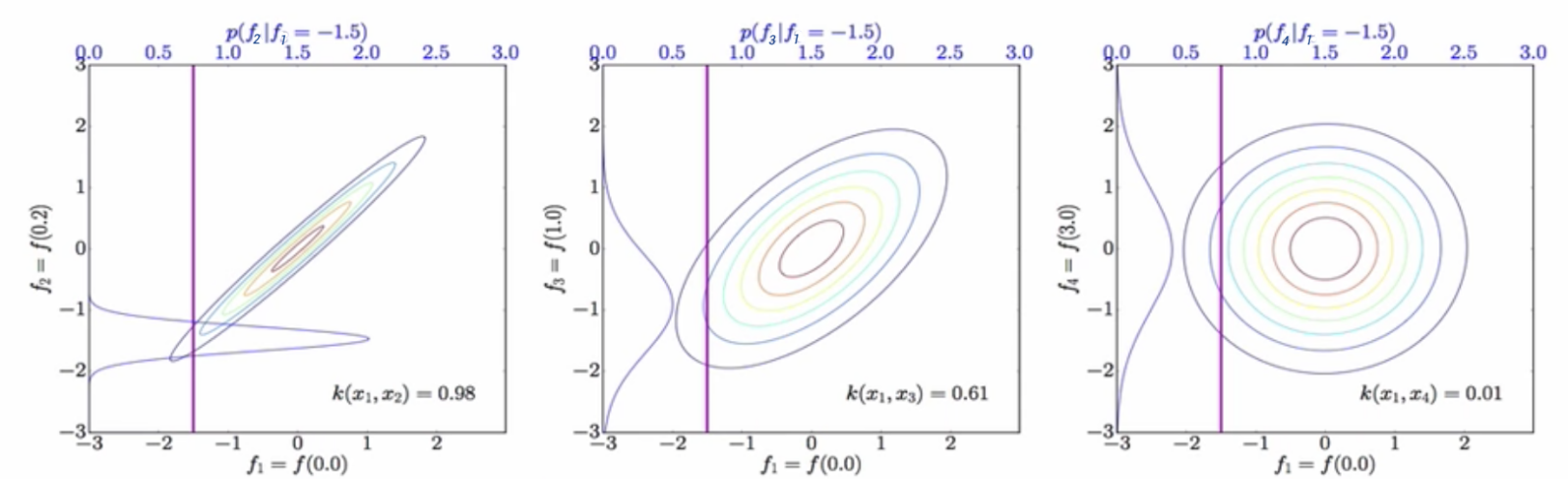
These look very similar! Indeed, they are effectively drawn from the same multivariate normal density.

We can also visualize the covariance matrix $\mathbf{K}$ for polynomial functions:



Many more examples of covariance matrices: https://pymc3-testing.readthedocs.io/en/rtd-docs/notebooks/GP-covariances.html (https://pymc3-testing.readthedocs.io/en/rtd-docs/notebooks/GP-covariances.html)

Understanding covariances



Left: If two points are similar, they covariate strongly. Knowing about f_1 tells us a lot about f_2

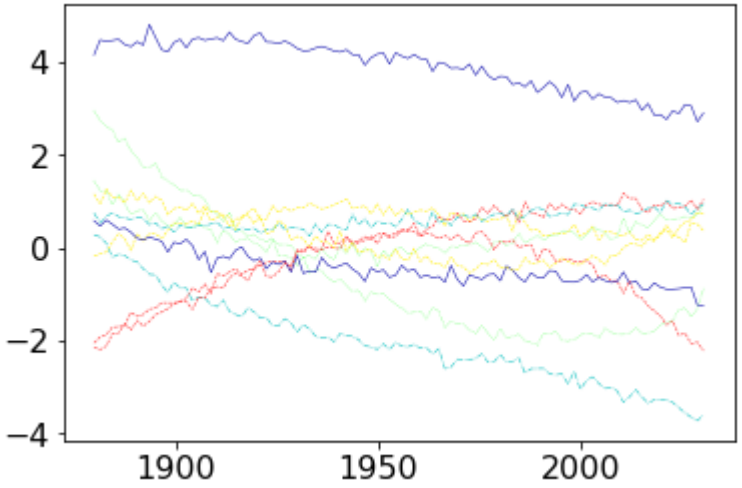
Middle: If two points are far apart, their covariance is small.

Right: If covariance is 0, the conditional and marginal distributions are the same.

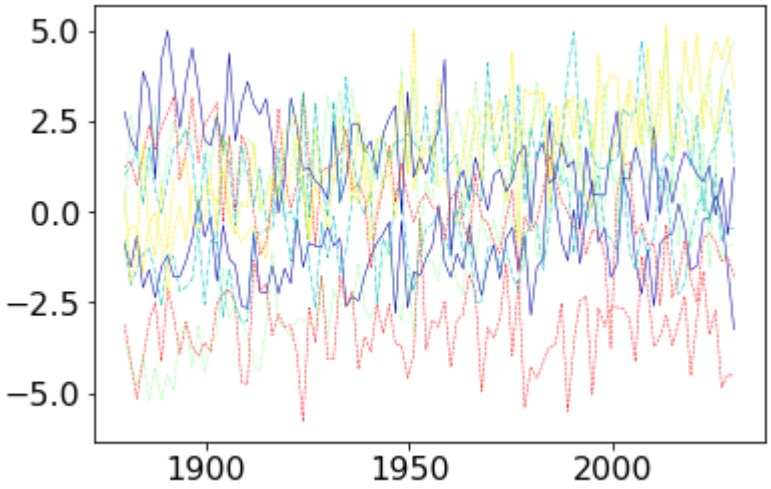
Noisy functions

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.

In a Gaussian process we can do this by specifying the *covariance function* directly, rather than *implicitly* through a basis matrix and a prior over parameters.

Gaussian processes have the additional advantage that they can be *nonparametric*: they can have *infinite* basis functions.

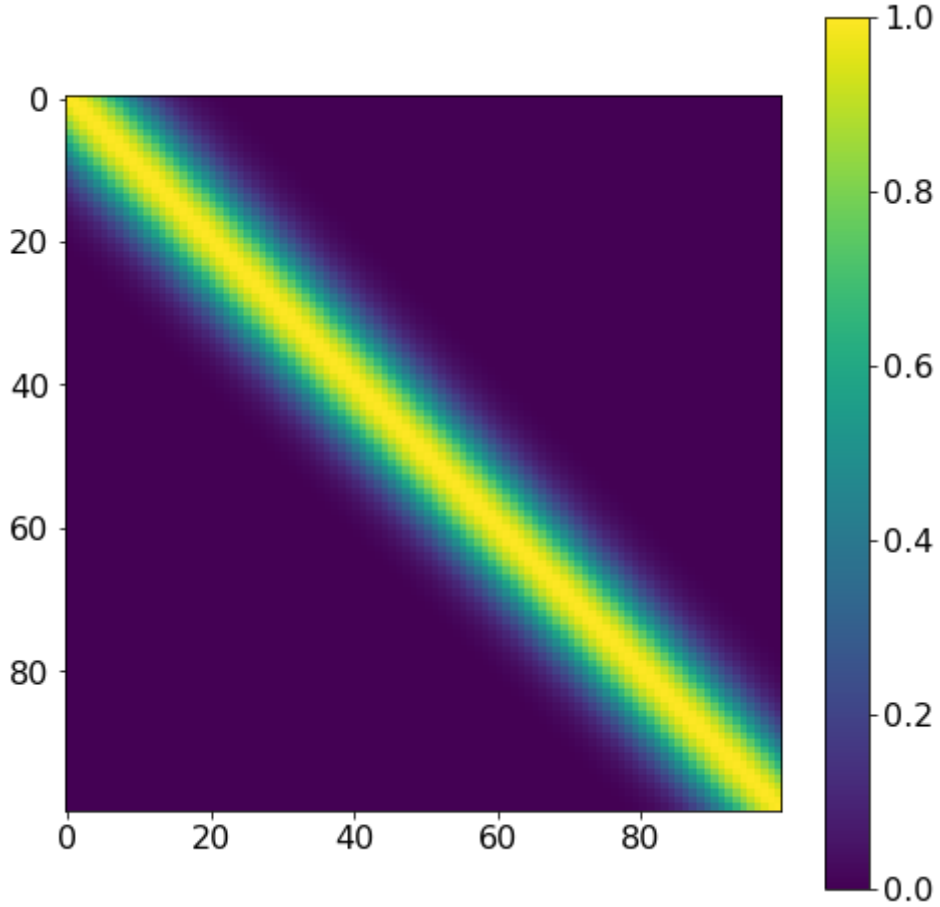
The RBF (Gaussian) covariance function is specified by

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

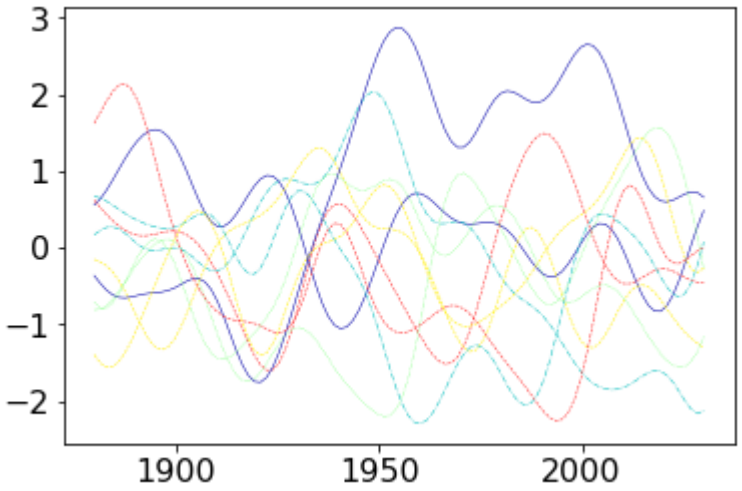
where $\left\|\mathbf{x} - \mathbf{x}'\right\|^2$ is the squared distance between the two input vectors

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

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

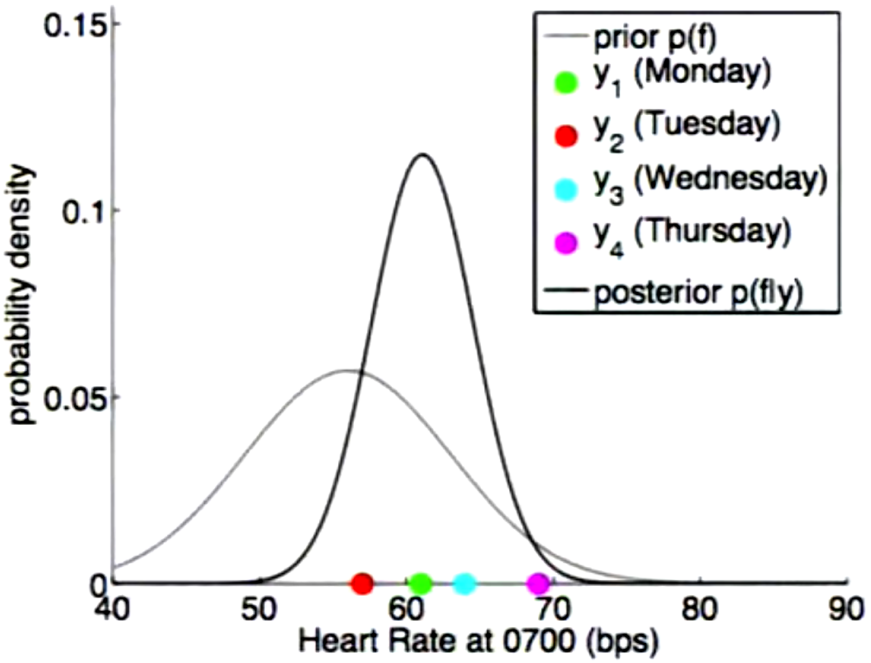


Finally, we can sample functions with this kernel (covariance matrix)

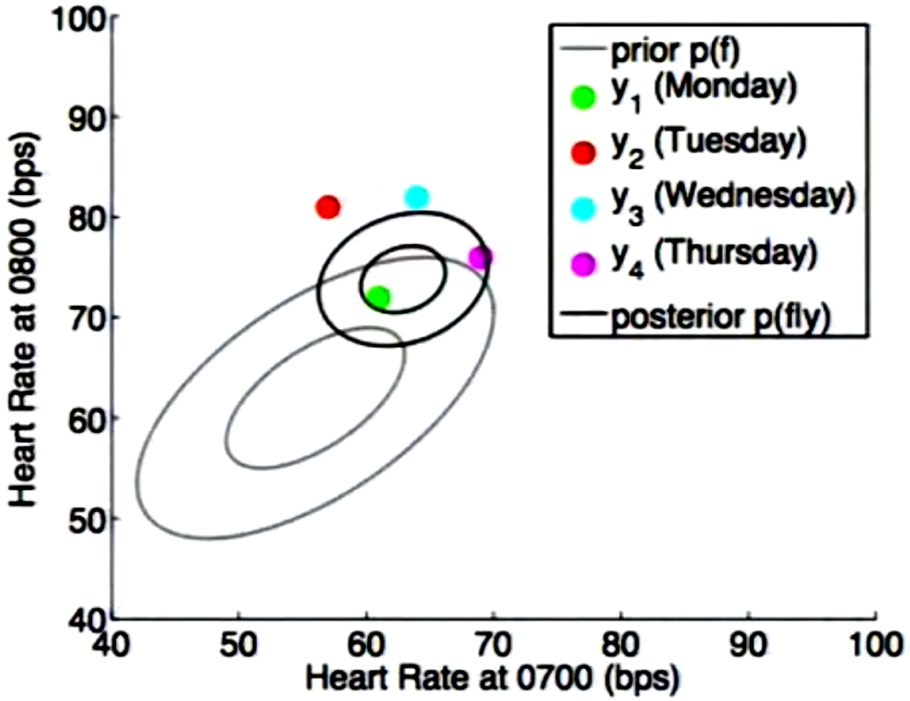


Gaussian process intuition

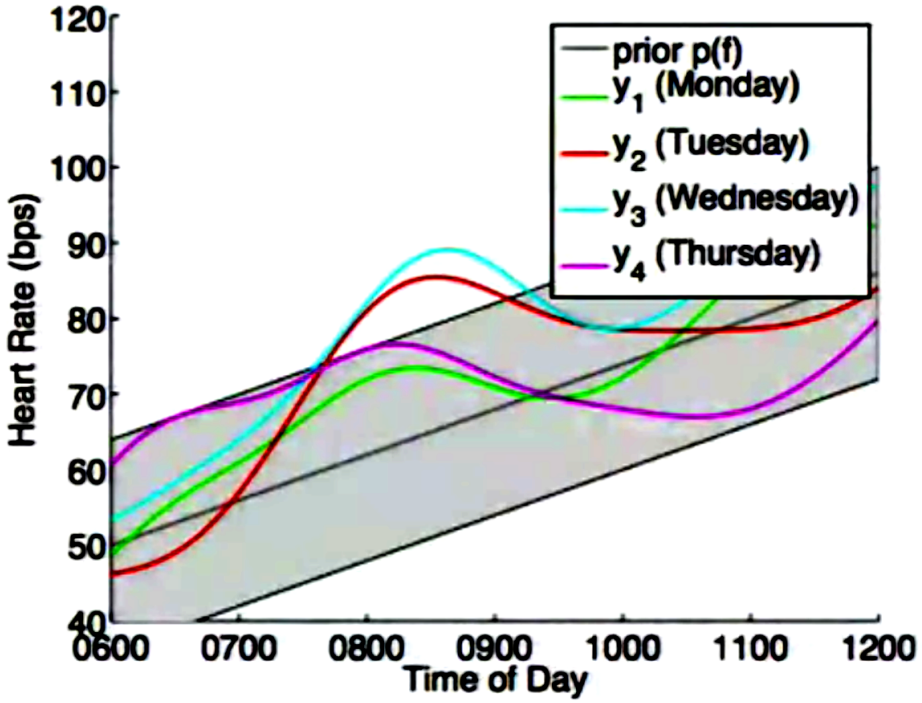
Univariate Gaussians: distributions over real-valued variables



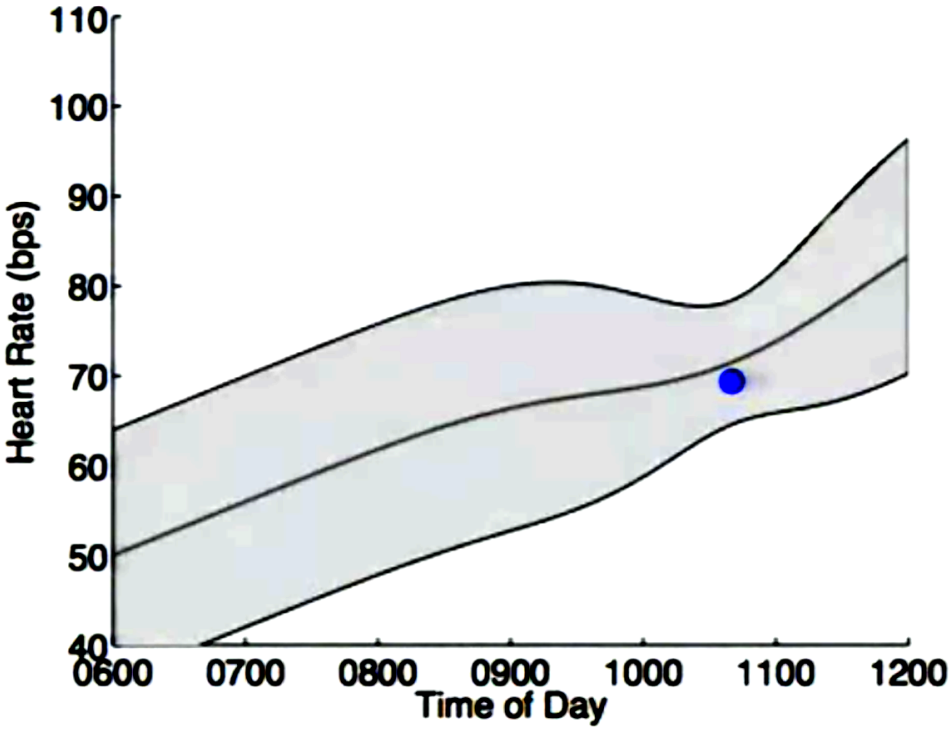
Multi-variate Gaussians: pairs (triplets,...) of real valued variables



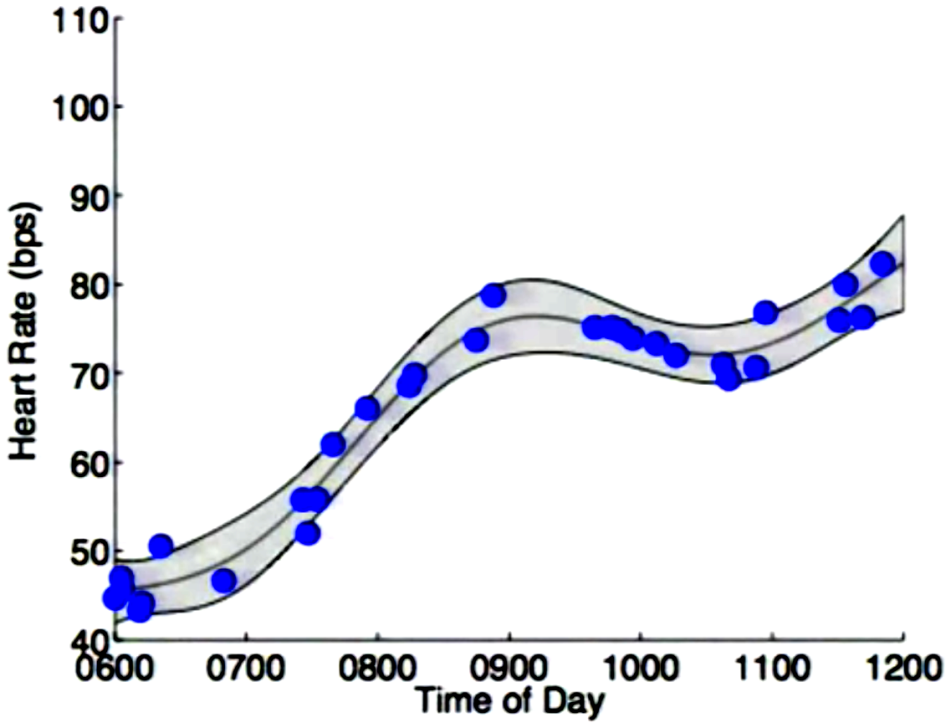
Gaussian processes: functions of infinite numbers of real values variables



Posterior probability after observation 1



Posterior probability after many observations



Gaussian process optimization

The Gaussian process perspective takes the marginal likelihood of the data to be a joint Gaussian density with a covariance given by $\mathbf{K}$.

The model likelihood is of the form,

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

where the input data, $\mathbf{X}$, influences the density through the covariance matrix, $\mathbf{K}$ whose elements are computed through the covariance function, $k(\mathbf{x}, \mathbf{x}')$.

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

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

where the *parameters* of the model are also embedded in the covariance function, they include the parameters of the kernel (such as lengthscale and variance), and the noise variance, σ^2.

```
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 inter
est
        ## NOTE: Not the correct *numerical* way to compute this! For ea
se 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.sh
ape[0]))
        # The matrix inner product of the inverse covariance
        self.Kinvy = np.dot(self.Kinv, self.y)
        self.yKinvy = (self.y*self.Kinvy).sum()


    def log_likelihood(self):
        # use the pre-computes to return the likelihood
        return -0.5*(self.K.shape[0]*np.log(2*np.pi) + self.logdetK + se
lf.yKinvy)

    def objective(self):
```

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

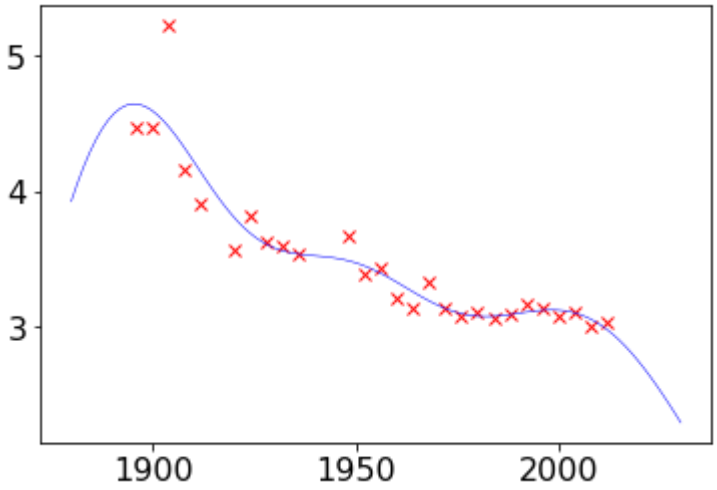
$$\mathbf{C}_f = \mathbf{K}_{*,*} - \mathbf{K}_*^\top \left[\mathbf{K} + \sigma^2\mathbf{I}\right]^{-1}\mathbf{K}_*.$$

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

We can now get the mean and covariance:

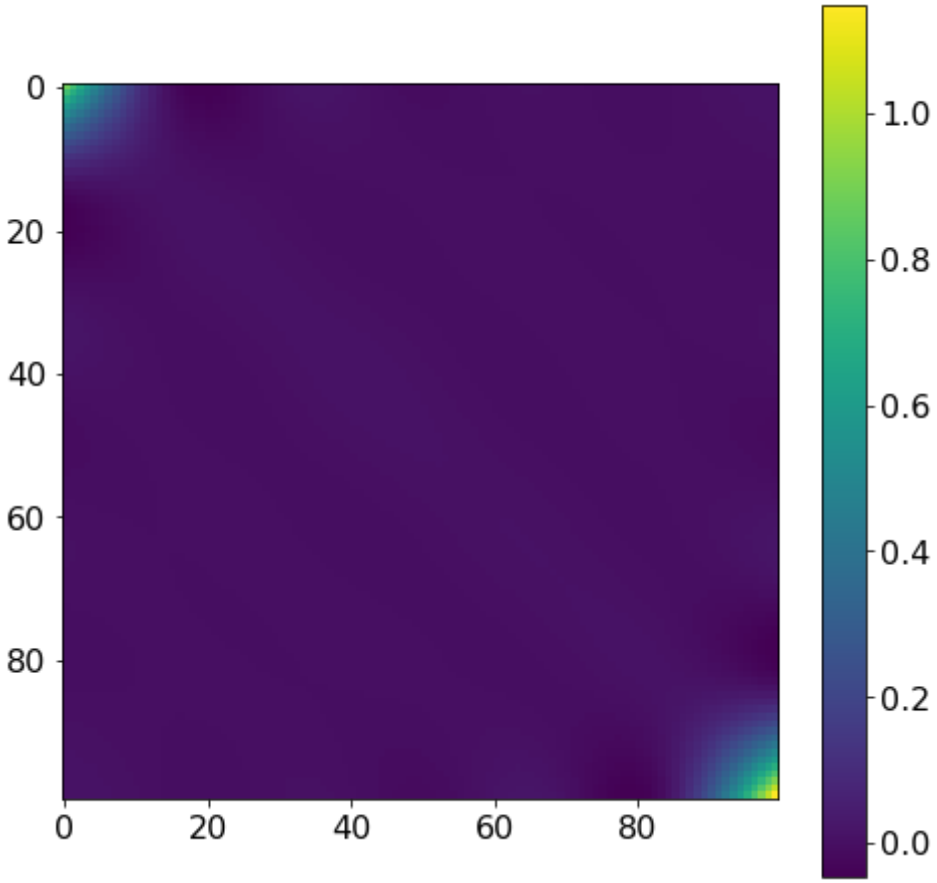
```
model = GP(x, y, sigma2, exponentiated_quadratic, variance=16.0, lengths
cale=32)
mu_f, C_f = model.posterior_f(x_pred)
```

Plot the mean:



The covariance looks like this:

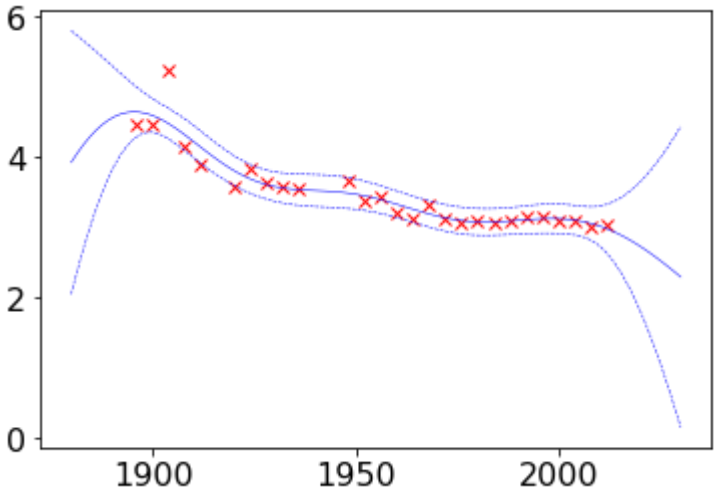
- Look along the diagonal
- High variance at the beginning and the end



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

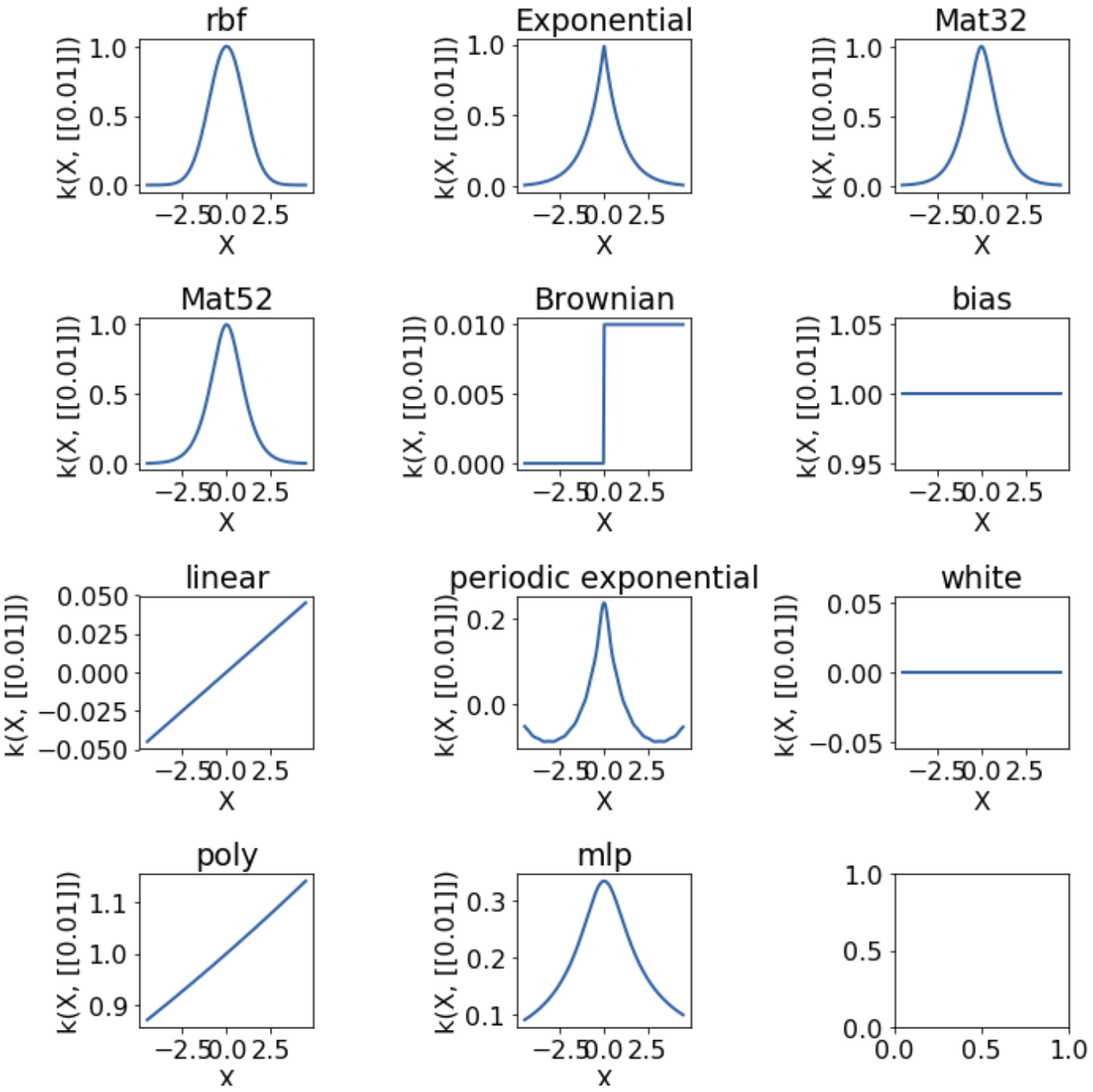
    ```
    kernel = GPy.kern.RBF(input_dim=1, variance=1., lengths
    cale=1.)
    ```

  - Other kernels:

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

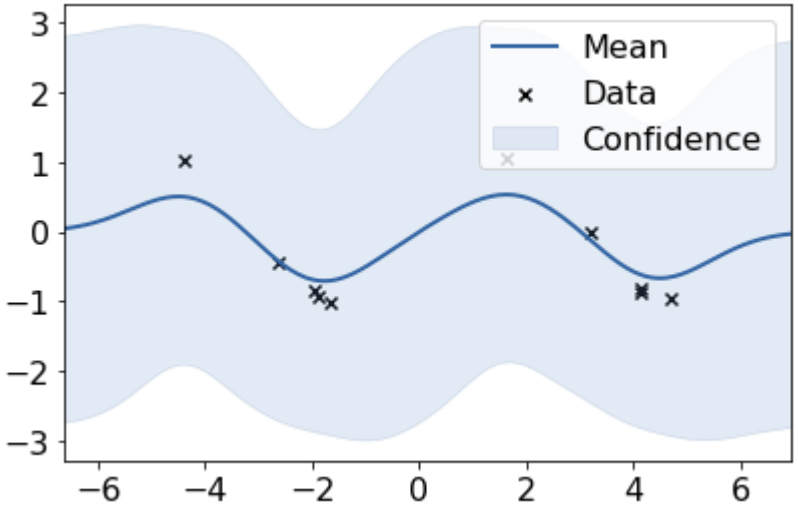
- Build model:

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


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

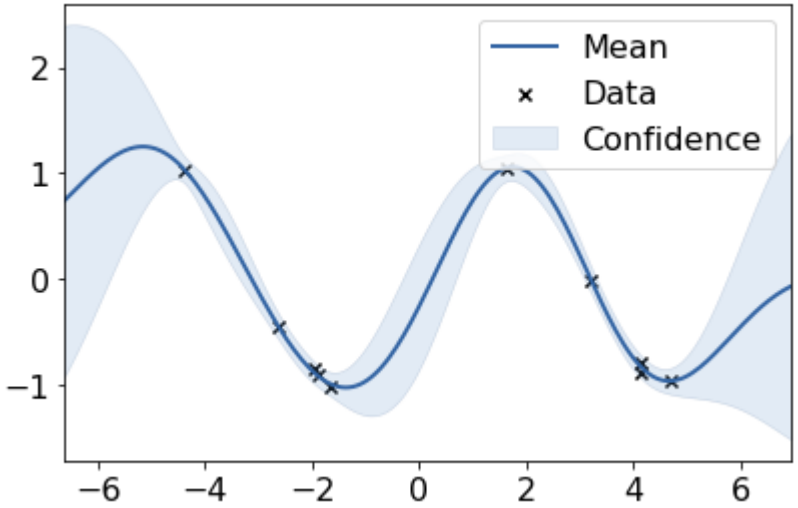
`Matern` is a generalized RBF kernel that can scale between RBF and Exponential

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

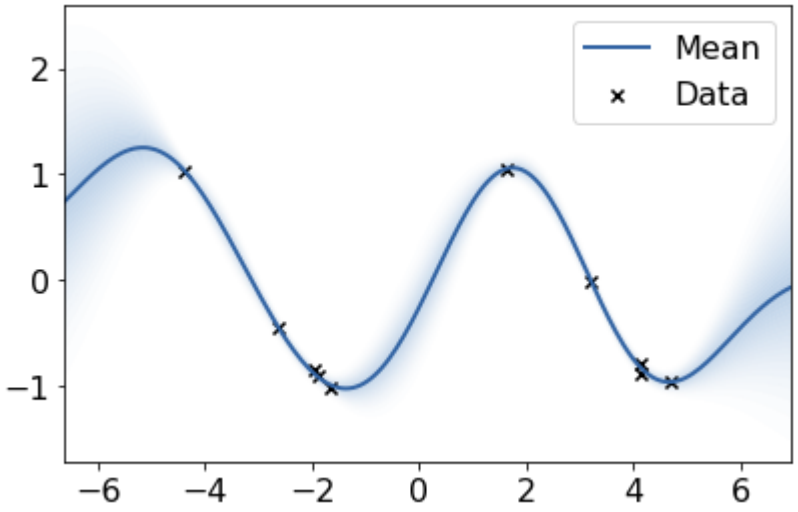


Train the model (optimize the parameters): maximize the likelihood of the data. Best to optimize with a few restarts: the optimizer may converges to the high-noise solution. The optimizer is then restarted with a few random initialization of the parameter values.

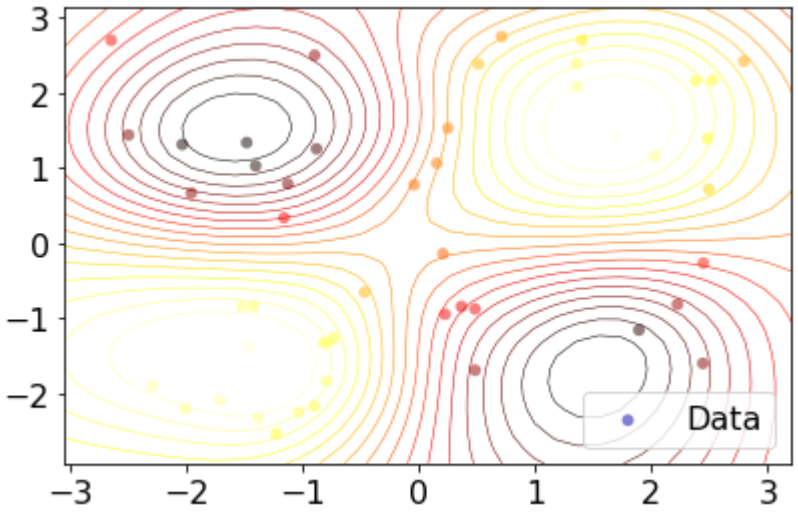
```
Optimization restart 1/10, f = 1.8757893249508362
Optimization restart 2/10, f = 1.8757893249521596
Optimization restart 3/10, f = 1.8757893249516862
Optimization restart 4/10, f = 1.8757893249545017
Optimization restart 5/10, f = 1.875789324951051
Optimization restart 6/10, f = 1.8757893249508353
Optimization restart 7/10, f = 1.8757893249552042
Optimization restart 8/10, f = 1.8757893249513211
Optimization restart 9/10, f = 1.8757893249507047
Optimization restart 10/10, f = 1.8757893249699453
```



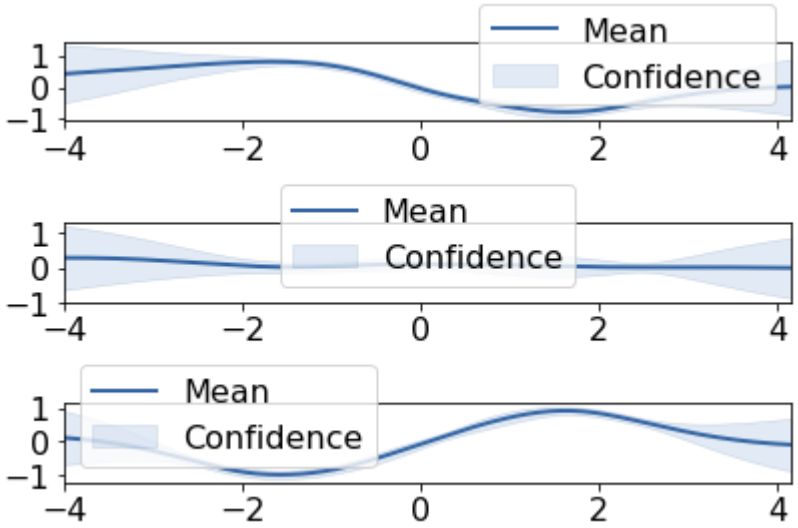
You can also plot densities



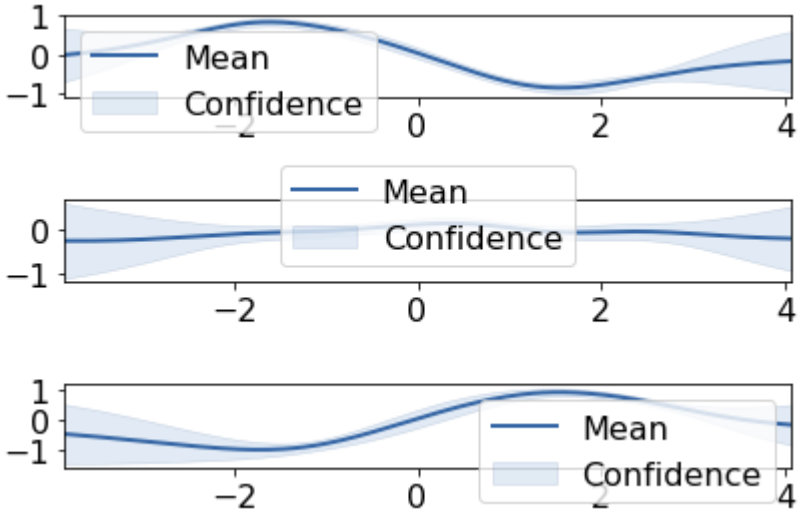
You can also show results in 2D



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

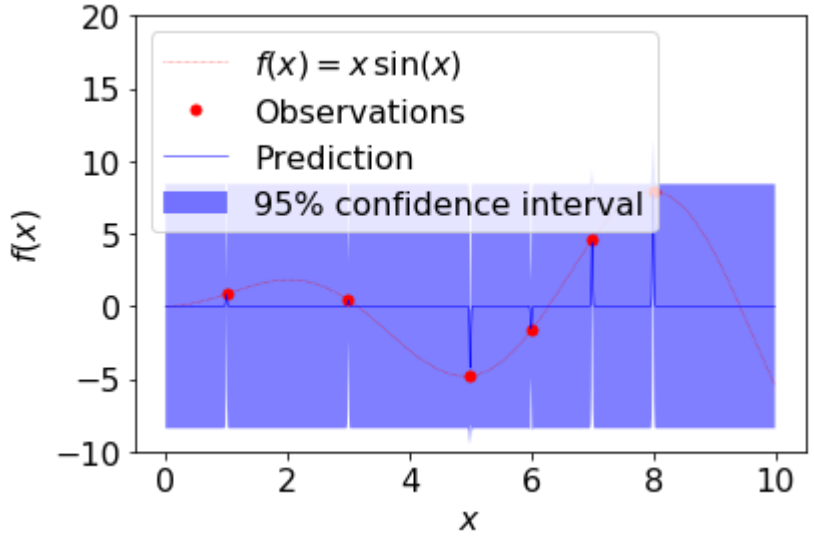
- `GaussianProcessRegressor`
- Hyperparameters:
  - `kernel`: kernel specifying the covariance function of the GP
    - Default: "1.0 * RBF(1.0)"
    - Typically leave at default. Will be optimized during fitting
  - `alpha`: regularization parameter
    - Tikhonov regularization of the assumed covariance 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 fitting
    - Default: 1e-10
  - `n_restarts_optimizer`: number of restarts of the optimizer
    - Default: 0. Best to do at least a few iterations.
    - Optimizer finds the kernel's parameters which

maximize the log-marginal likelihood

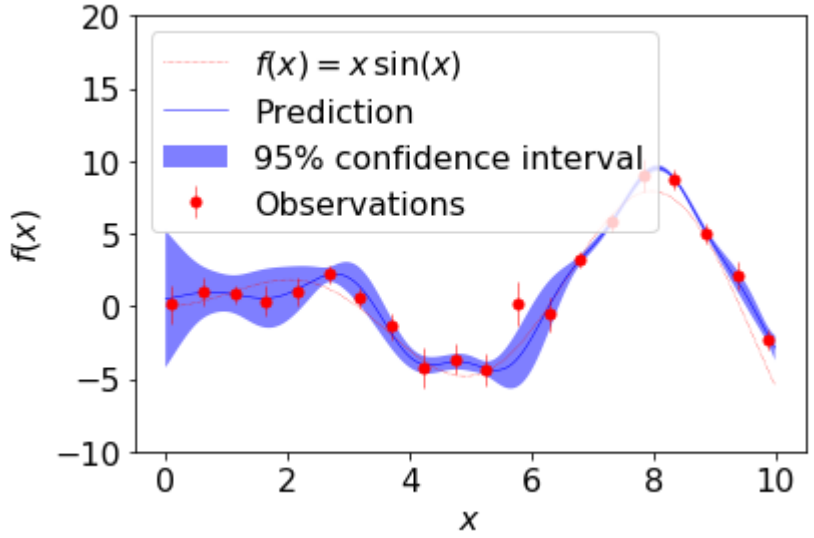
- Retrieve predictions and confidence interval after fitting:

```
y_pred, sigma = gp.predict(x, return_std=True)
```

Example


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

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:

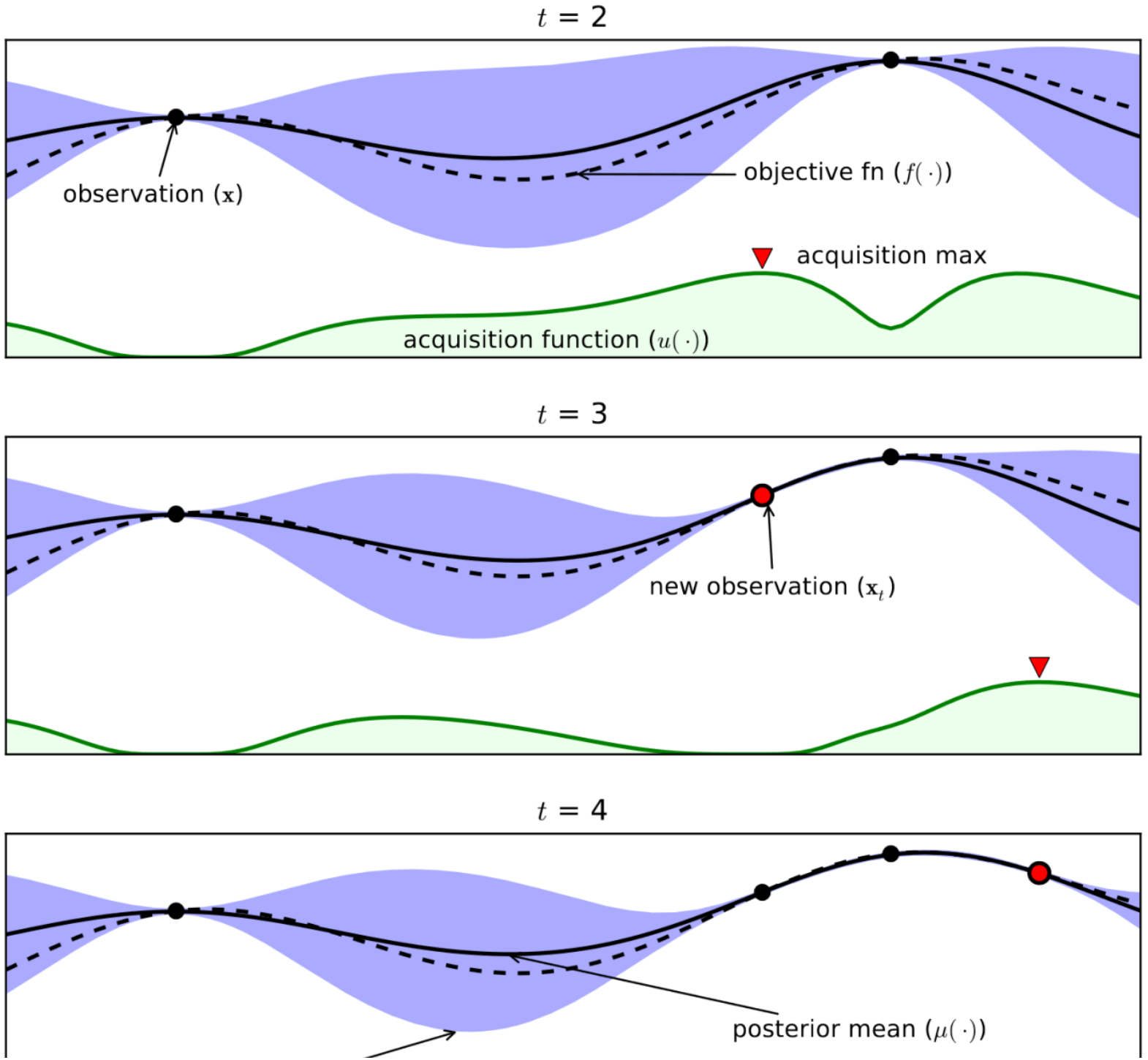
- They are not sparse, i.e., they use the whole samples/features information to perform the prediction.
- They lose efficiency in high dimensional spaces – namely when the number of features exceeds a few dozens.

Bayesian optimization

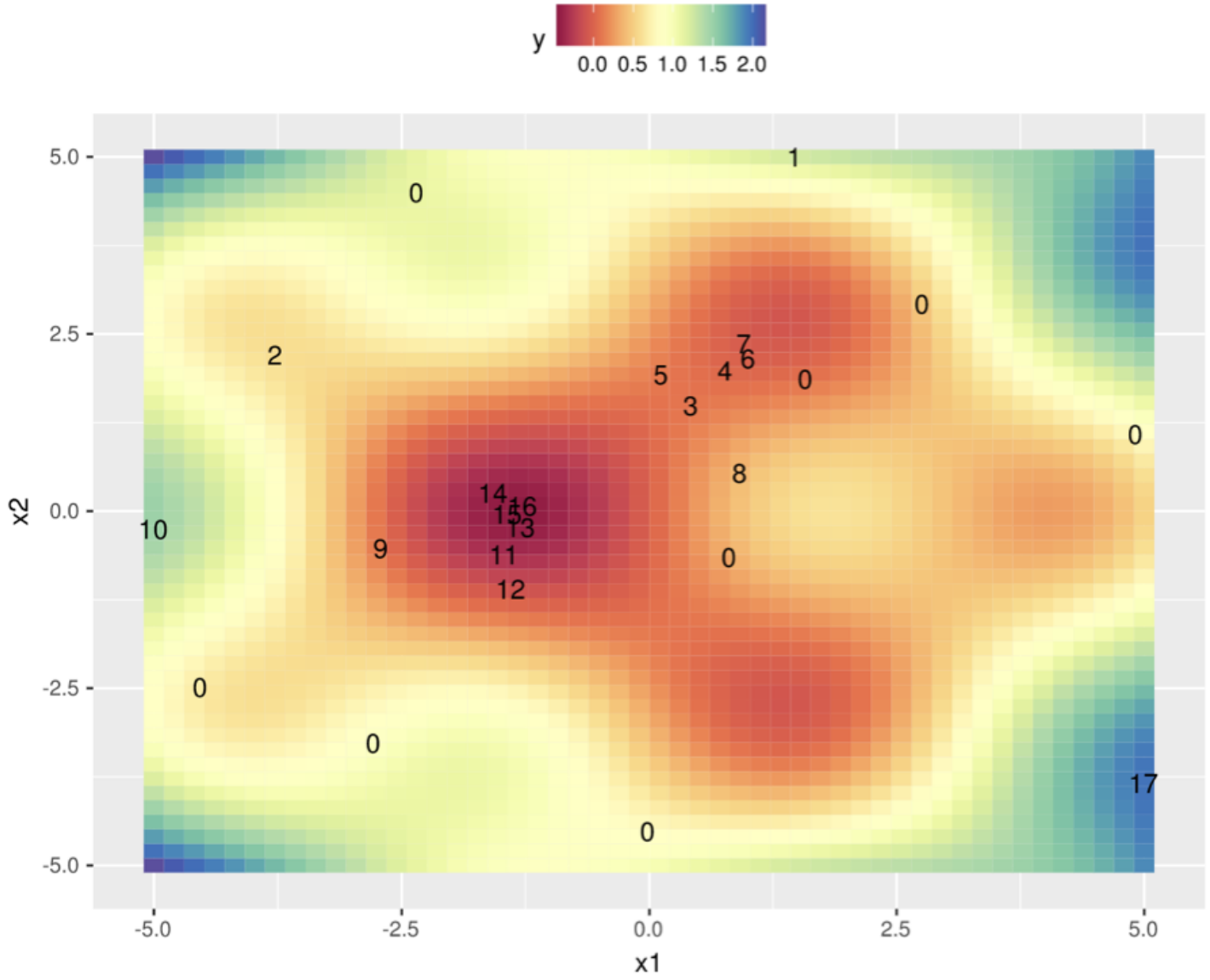
- The incremental updates you can do with Bayesian models allow a more effective way to optimize functions
  - E.g. to optimize the hyperparameter settings of a machine learning algorithm/pipeline
- After a number of random search iterations we know more about the performance of hyperparameter settings 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 ($\mu(\cdot)$)

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 $\mathbf{X}^+$:

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

  - Computing the expected performance requires an integration over the posterior distribution, but has a closed form solution (http://ash-aldujaili.github.io/blog/2018/02/01/ei/).

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