PRML

August 7, 2022

1. Introduction

Table of Contents

[1]:
```
import math
import numpy as np
import matplotlib.pyplot as plt

from prml.datasets import generate_toy_data
from prml.preprocessing import PolynomialFeature
from prml.linear.linear_regression import LinearRegression
from prml.linear.ridge_regression import RidgeRegression
from prml.distribution import (
    Gaussian,
    MultivariateGaussian
)

# Set random seed to make deterministic
np.random.seed(0)

# Ignore zero divisions and computation involving NaN values.
np.seterr(divide = 'ignore', invalid='ignore')

# Enable higher resolution plots
%config InlineBackend.figure_format = 'retina'
```

1.1. Example: Polynomial Curve Fitting

For presentation purposes, consider a synthetically generated example dataset. The data were generated from the function $\sin(2\pi x)$ by adding random Gaussian noise having standard deviation

$\sigma = 0.3$.

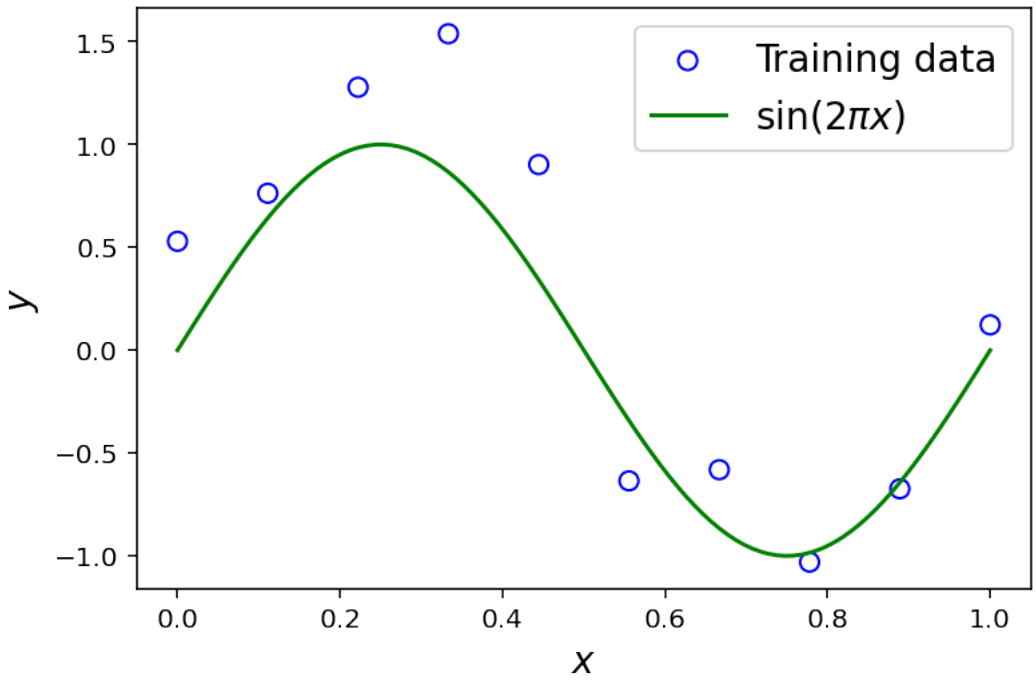
We generated $N = 10$ observations spaced uniformly in range $[0, 1]$. These observations comprise the input data vector,

$$\mathsf{x} = (x_1, \ldots, x_N)^{\mathrm{T}}$$

For each generated observation x we obtained its corresponding value of the function $\sin(2\pi x)$ and then adding the random noise to capture the real-life situation of missing information.

$$\mathsf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$$

[2]:
```
# Sine function
def sin(x: np.ndarray) -> np.ndarray:
    return np.sin(2 * np.pi * x)

# Generate a train data set
x_train, y_train = generate_toy_data(sin, 10, 0.3)

# Generate a test data set
x_test = np.linspace(0, 1, 100)
y_test = sin(x_test)

plt.scatter(x_train, y_train, facecolor="none", edgecolor="b", s=50,␣
 ↪label="Training data")
plt.plot(x_test, y_test, color="g", label="$\sin(2\pi x)$")
plt.xlabel('$x$', fontsize=14); plt.ylabel('$y$', fontsize=14)
plt.legend(fontsize=14); plt.show()
```



The generated training dataset of $N = 10$ points is shown as blue circles, each comprising an observation of the input variable x along with the corresponding target variable t. The green curve shows the function $\sin(2\pi x)$ used to generate the data.

Polynomial Linear Model

Our goal is to predict the value of $\hat{t}$ for some **new** value of $\hat{x}$, in the absence of any knowledge for the green curve. To that end, we consider a simple approach based on curve fitting. In particular, we shall try to fit the data using a polynomial function of the form

$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

where M is the *order* of the polynomial. Functions, such as $y(x, \mathbf{w})$, that are linear functions of the unknown coefficients $\mathbf{w}$, are called *linear models*.

Error Function

Next, we need to determine the values of the coefficients $\mathbf{w}$ by fitting the polynomial to the training data. This can be done by minimizing an *error function* that measures the misfit between the function $y(x, \mathbf{w})$, for a given value of $\mathbf{w}$, and the training data points.

One simple error function is the *sum of squares of the errors* between $y(x, \mathbf{w})$ and the corresponding target values t_n

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y(x,\mathbf{w}) - t_n)^2 \geq 0$$

where the function becomes zero if, and only if, the function $y(x, \mathbf{w})$ were to pass exactly through each training data point.

We can solve the curve fitting problem by choosing the value of $\mathbf{w}$ for which $E(\mathbf{w})$ is as small as possible. Because the error function is quadratic, its derivatives are linear, and so the minimization of the function has a unique closed from solution, denoted by $\mathbf{w}^\star$. To minimize the error function we should derive the gradient vector, set it equal to zero and solve for $\mathbf{w}^\star$ as follows,

$$\nabla E(\mathbf{w}^\star) = \mathbf{0}$$

First, we have to substitute the polynomial into the error function

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\Big(\sum_{j=0}^{M} w_j x_n^j - t_n\Big)^2$$

Note that each of the N data points from the generated training set has 1 dimension, that is $x \in \mathbb{R}$. However, the polynomial function populates M features for each input x, essentially transforming x into a M-dimensional vector. Thus, the training set x can be written as a $N \times M$ matrix $\mathbf{X}$ where $\mathbf{X}_{nj}$ represents x_n^j, that is, the nth input value raised in the power of j.

To find the gradient vector, we take the partial derivative of E with respect to an arbitrary w_k. Differentiating the sum, term by term, we get

$$\begin{aligned}
\nabla E(\mathbf{w}^\star)_k &= \frac{\partial}{\partial w_k}(\mathbf{w}) \\
&= \frac{1}{2}\sum_{n=1}^{N} 2(\sum_{j=0}^{M} w_j x_n^j - t_n)x_n^k = \sum_{n=1}^{N}(\sum_{j=0}^{M} w_j x_n^j - t_n)x_n^k \\
&= \sum_{n=1}^{N}(\mathbf{X}\mathbf{w} - \mathsf{t})_n \mathbf{X}_{nk} = \sum_{n=1}^{N}\mathbf{X}^{\mathrm{T}}_{kn}(\mathbf{X}\mathbf{w} - \mathsf{t})_n \\
&= \left(\mathbf{X}^{\mathrm{T}}(\mathbf{X}\mathbf{w} - \mathsf{t})\right)_k
\end{aligned}$$

Using the partial derivative for one component, we compute the gradient vector by dropping the k subscript. Thus, the minimizer $\mathbf{w}^\star$ must satisfy

$$\nabla E(\mathbf{w}^\star) = \mathbf{X}^{\mathrm{T}}(\mathbf{X}\mathbf{w}^\star - \mathsf{t}) = \mathbf{0}$$

Solving for $\mathbf{w}^\star$ gives the unique solution of the curve fitting problem

$$\mathbf{X}^{\mathrm{T}}(\mathbf{X}\mathbf{w}^\star - \mathsf{t}) = \mathbf{0} \Leftrightarrow \mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{w}^\star = \mathbf{X}^{\mathrm{T}}\mathsf{t} \Leftrightarrow \mathbf{w}^\star = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathsf{t}$$

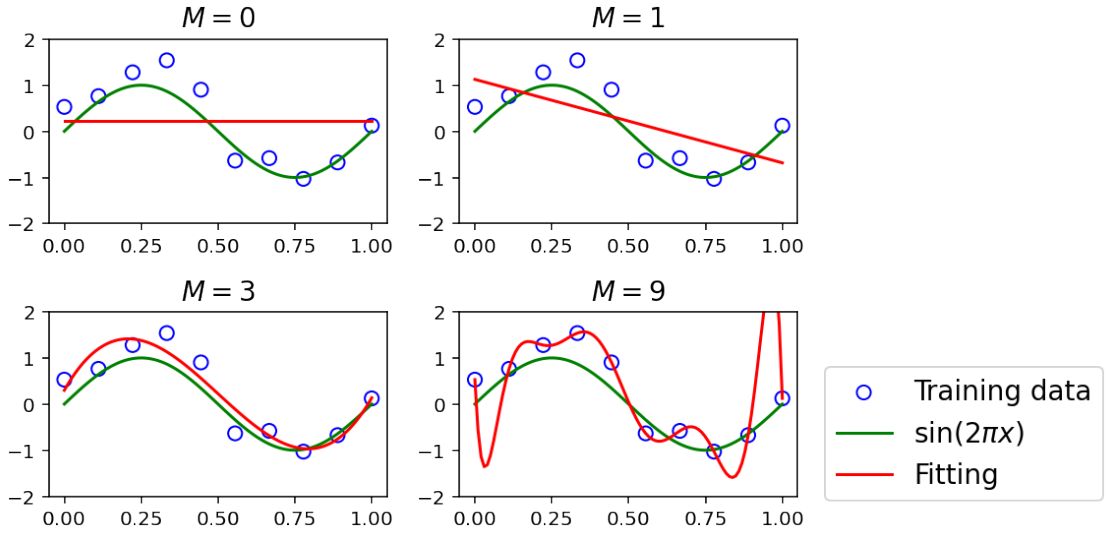
The resulting polynomial is given by the function $y(x, \mathbf{w}^\star)$.

Model Selection

There remains the problem of choosing the order M of the polynomial, which is an example of the important concept called *model selection*.

In order to study the effect of different M values, we plot the result of fitting polynomials having orders $M = 0, 1, 3, 9$ to the data set.

[3]:
```
for i, degree in enumerate([0, 1, 3, 9]):
    plt.subplot(2, 2, i + 1)
    plt.tight_layout()

    feature = PolynomialFeature(degree)
    x_train_features = feature.transform(x_train)
    x_test_features = feature.transform(x_test)

    model = LinearRegression()
    model.fit(x_train_features, y_train)
    y, _ = model.predict(x_test_features)

    plt.scatter(x_train, y_train, facecolor="none", edgecolor="b", s=50,␣
 ↪label="Training data")
    plt.plot(x_test, y_test, color="g", label="$\sin(2\pi x)$")
    plt.plot(x_test, y, color="r", label="Fitting")
    plt.ylim(-2, 2)
    plt.title("$M={}$".format(degree), fontsize=14)

plt.legend(bbox_to_anchor=(1, 0.85), loc=2, borderaxespad=1, fontsize=14)
plt.show()
```



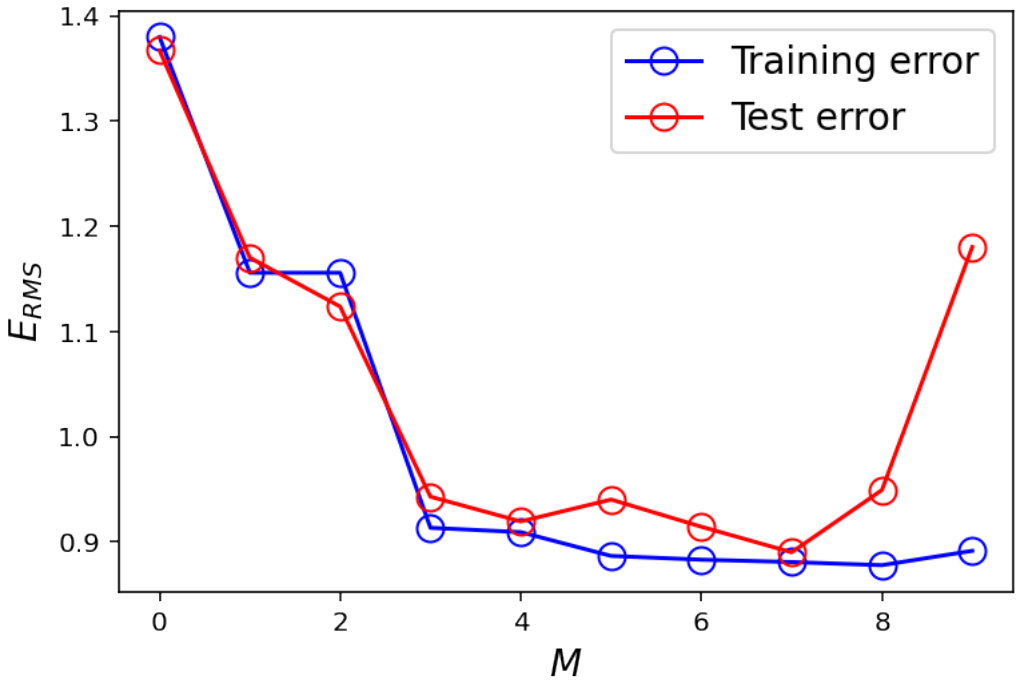
Note that the constant ($M = 0$) and first order ($M = 1$) polynomials give rather poor fits to the data. The third order ($M = 3$) polynomial seems to give the best fit, while the higher order one ($M = 9$) achieves an excellent fit to the data, that is, $E(\mathbf{w}^\star) = \mathbf{0}$. However, the fitted curve gives a poor representation of the underlying function $\sin(2\pi x)$. This phenomenon is known as *over-fitting*.

A more quantitative insight into the generalization performance on M can be obtained by using the *root-mean-square* (RMS) error defined as

$$E_{RMS} = \sqrt{2\frac{E(\mathbf{w}^\star)}{N}}$$

The RMS error on both training and test data points for each value of M is shown in the following figure:

[4]:
```
def rms_error(a, b):
    return np.sqrt(2 * np.mean(np.square(a - b)))

training_errors = []
test_errors = []

for i in range(10):

    feature = PolynomialFeature(i)
    x_train_features = feature.transform(x_train)
    x_test_features = feature.transform(x_test)

    model = LinearRegression()
    model.fit(x_train_features, y_train)
    y, _ = model.predict(x_test_features)
    training_errors.append(rms_error(model.predict(x_train_features), y_train))
    test_errors.append(rms_error(model.predict(x_test_features), y_test + np.
 ↪random.normal(scale=0.3, size=len(y_test))))

plt.plot(training_errors, 'o-', mfc="none", mec="b", ms=10, c="b",␣
 ↪label="Training error")
plt.plot(test_errors, 'o-', mfc="none", mec="r", ms=10, c="r", label="Test␣
 ↪error")
plt.xlabel("$M$", fontsize=14); plt.ylabel("$E_{RMS}$", fontsize=14)
plt.legend(fontsize=14); plt.show()
```

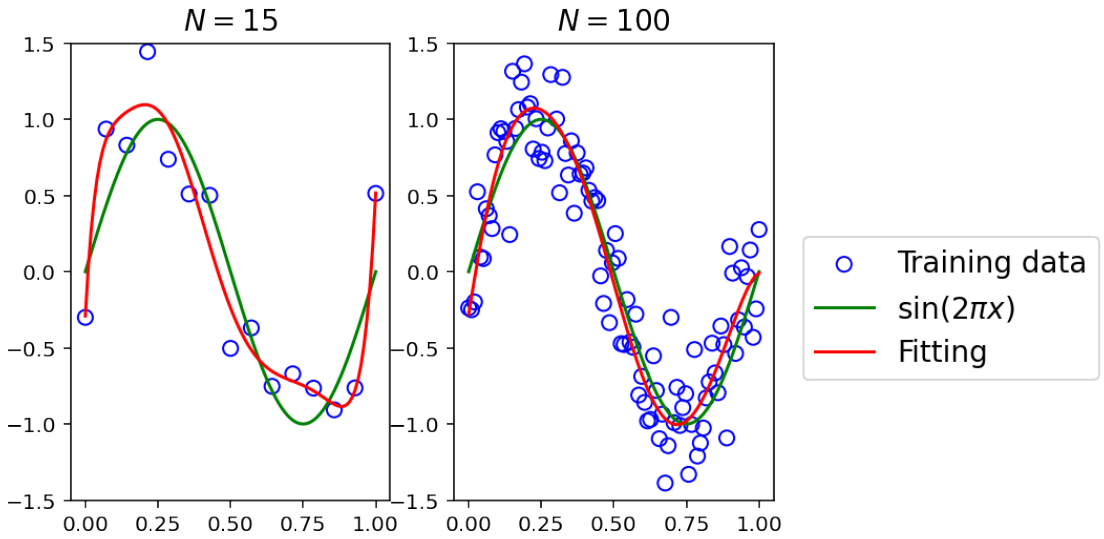


The test set error is measuring how well we are doing in predicting the values of t for new data observations of x. For $M = 9$, the training set error goes to zero, because the polynomial contains 10 degrees of freedom and so it can be tuned exactly to the 10 data points in the training set.

It is also interesting to examine the behavior of the model as the size of the data increases. The following figure depicts the result of fitting the $M = 9$ polynomial for $N = 15$ and $N = 100$ data points.

[5]:
```
for i, size in enumerate([15, 100]):
    plt.subplot(1, 2, i + 1)

    # Generate a train set
    x_train_100, y_train_100 = generate_toy_data(sin, size, 0.3)

    # Generate a test set
    x_test_100 = np.linspace(0, 1, 100)
    y_test_100 = sin(x_test)

    feature = PolynomialFeature(9)
    x_train_100_features = feature.transform(x_train_100)
    x_test_100_features = feature.transform(x_test_100)

    model = LinearRegression()
    model.fit(x_train_100_features, y_train_100)
    y, _ = model.predict(x_test_100_features)
```

```
    plt.scatter(x_train_100, y_train_100, facecolor="none", edgecolor="b",␣
 ↪s=50, label="Training data")
    plt.plot(x_test_100, y_test_100, color="g", label="$\sin(2\pi x)$")
    plt.plot(x_test_100, y, color="r", label="Fitting")
    plt.ylim(-1.5, 1.5)
    plt.title("$N={}$".format(size), fontsize=14)

plt.legend(bbox_to_anchor=(1, 0.64), loc=2, borderaxespad=1, fontsize=14)
plt.show()
```



Note that the over-fitting problem becomes less severe as the size of the data set increases. In other words, the larger the data set, the more complex the model that we can afford to fit to the data.

Regularization

One technique that is often used to control the over-fitting phenomenon is that of *regularization*, which adds a penalty term to the error function in order to discourage the coefficients from reaching large values. The simplest such penalty term is the sum of squares of all of the coefficients, leading to a modified error function of the following form

$$\tilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y(x,\mathbf{w}) - t_n)^2 + \lambda||\mathbf{w}||_2^2$$

Such techniques are known as *shrinkage* methods because they reduce the value of the coefficients. The particular case of the quadratic regularization is called *ridge regression*. In neural networks, this approach is also known as *weight decay*.

Similar to the previous case, the ridge error function can be minimized exactly in closed form as follows,

$$\begin{aligned}
\nabla E(\mathbf{w}^\star)_k &= \frac{\partial}{\partial w_k}(\mathbf{w}) \\
&= \frac{1}{2}\sum_{n=1}^{N} 2(\sum_{j=0}^{M} w_j x_n^j - t_n)x_n^k + \frac{1}{2}\lambda 2 w_k \\
&= \sum_{n=1}^{N}(\sum_{j=0}^{M} w_j x_n^j - t_n)x_n^k + \lambda w_k \\
&= \sum_{n=1}^{N}(\mathbf{X}\mathbf{w} - \mathsf{t})_n \mathbf{X}_{nk} + \lambda w_k = \sum_{n=1}^{N} \mathbf{X}_{kn}^{\mathrm{T}}(\mathbf{X}\mathbf{w} - \mathsf{t})_n + \lambda w_k \\
&= \left(\mathbf{X}^{\mathrm{T}}(\mathbf{X}\mathbf{w} - \mathsf{t})\right)_k + \lambda w_k
\end{aligned}$$

Using the partial derivative for one component, we compute the gradient vector by dropping the k subscript. Then, the minimizer $\mathbf{w}^\star$ must satisfy

$$\nabla E(\mathbf{w}^\star) = \mathbf{X}^T(\mathbf{X}\mathbf{w}^\star - \mathsf{t}) + \lambda \mathbf{w}^\star \mathbf{I} = \mathbf{0}$$

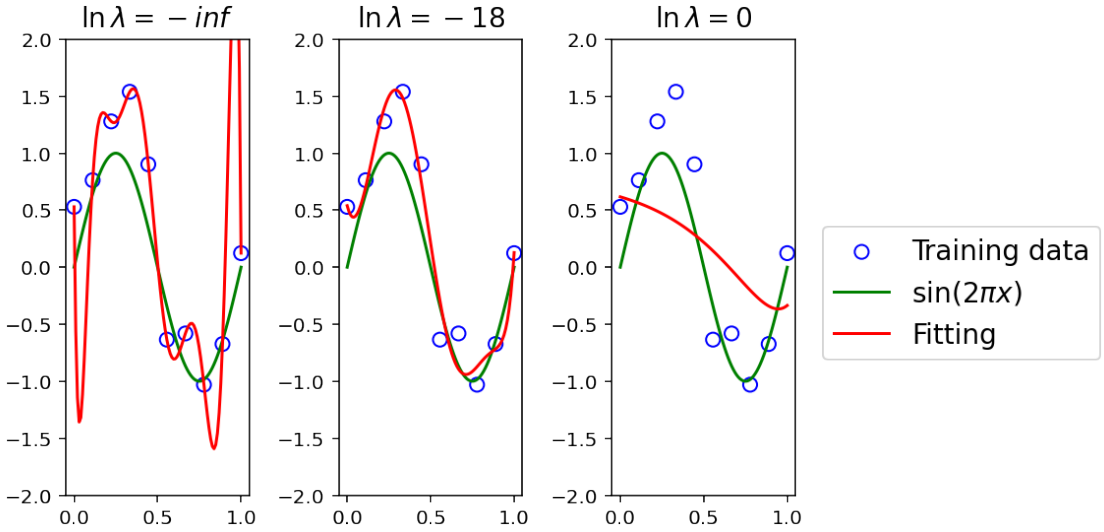
Solving for $\mathbf{w}^\star$ gives the unique solution that minimizes the ridge error

$$\begin{aligned}
&\mathbf{X}^{\mathrm{T}}(\mathbf{X}\mathbf{w}^\star - \mathsf{t}) + \lambda \mathbf{w}^\star \mathbf{I} = \mathbf{0} \Leftrightarrow \\
&\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{w}^\star - \mathbf{X}^{\mathrm{T}}\mathsf{t} + \lambda \mathbf{w}^\star \mathbf{I} = \mathbf{0} \Leftrightarrow \\
&\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{w}^\star + \lambda \mathbf{w}^* \mathbf{I} = \mathbf{X}^{\mathrm{T}}\mathsf{t} \Leftrightarrow \\
&\mathbf{w}^\star(\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda \mathbf{I}) = \mathbf{X}^{\mathrm{T}}\mathsf{t} \Leftrightarrow \\
&\mathbf{w}^\star = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda \mathbf{I})^{-1}\mathbf{X}^{\mathrm{T}}\mathsf{t}
\end{aligned}$$

The following figures depict the results of fitting the polynomial of order $M = 9$ to the same data set as before but this time using the regularized error function. We see that, for a value of $\ln \lambda = -18$, the over-fitting has been suppressed and we obtain a much closer representation of the underlying function $\sin(2\pi x)$. If, however, we use too large a value for λ then we again obtain a poor fit.

[6]:
```
feature = PolynomialFeature(9)
x_train_features = feature.transform(x_train)
x_test_features = feature.transform(x_test)

for i, ln_lambda in enumerate([float('-inf'), -18, 0]):
    plt.subplot(1, 3, i + 1)
    plt.tight_layout()

    model = RidgeRegression(alpha=math.exp(ln_lambda))
    model.fit(x_train_features, y_train)
    y, _ = model.predict(x_test_features)
```

```
    plt.scatter(x_train, y_train, facecolor="none", edgecolor="b", s=50,␣
 ↪label="Training data")
    plt.plot(x_test, y_test, color="g", label="$\sin(2\pi x)$")
    plt.plot(x_test, y, color="r", label="Fitting")
    plt.ylim(-2, 2)
    plt.title("$\ln\lambda={}$".format(ln_lambda), fontsize=14)

plt.legend(bbox_to_anchor=(1, 0.65), loc=2, borderaxespad=1, fontsize=14)
plt.show()
```

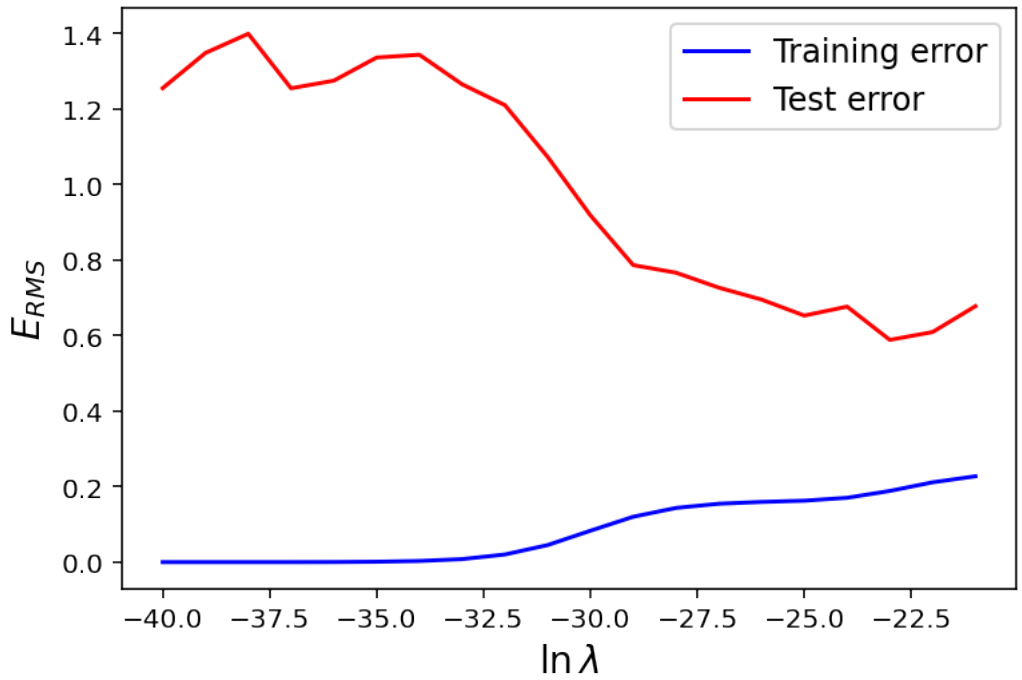


A sample of coefficients from the fitted polynomials is presented in the table below, showing that regularization has the desired effect of reducing the magnitude of the coefficients.

$\ln\lambda = -\infty$	$\ln\lambda = -18$	$\ln\lambda = 0$
0.39	0.38	0.36
−135.04	−2.14	−0.46
3206.76	81, 88	−0.39
−29215.92	−390.51	−0.22
139594.34	578.12	−0.05
−388863.80	−31.48	0.07
652373.24	−49.12	0.17
−648124.69	−28.57	0.25
350721.94	540.17	0.31
−79556.29	−255.65	0.35

The impact of the regularization term on the generalization error can be seen by plotting the value of the RMS error for both training and test sets against $\ln\lambda$, as shown in the next figure. We see that λ controls the effective complexity of the model and hence determines the degree of over-fitting.

[7]:
```
training_errors = []
test_errors = []

feature = PolynomialFeature(9)
x_train_features = feature.transform(x_train)
x_test_features = feature.transform(x_test)

ln_lambda_values = range(-40, -20, 1)
for ln_lambda in ln_lambda_values:

    model = RidgeRegression(alpha=math.exp(ln_lambda))
    model.fit(x_train_features, y_train)
    training_errors.append(rms_error(model.predict(x_train_features)[0],␣
 ↪y_train))
    test_errors.append(
        rms_error(model.predict(x_test_features)[0], y_test + np.random.
 ↪normal(scale=0.3, size=len(y_test)))
    )

plt.plot(ln_lambda_values, training_errors, mfc="none", mec="b", ms=10, c="b",␣
 ↪label="Training error")
plt.plot(ln_lambda_values, test_errors, mfc="none", mec="r", ms=10, c="r",␣
 ↪label="Test error")
plt.xlabel("$\ln\lambda$", fontsize=14); plt.ylabel("$E_{RMS}$", fontsize=14)
plt.legend(fontsize=12); plt.show()
```



1.2 Probability Theory

The probability $p(A)$ of an event A is always a non-negative number, i.e.,

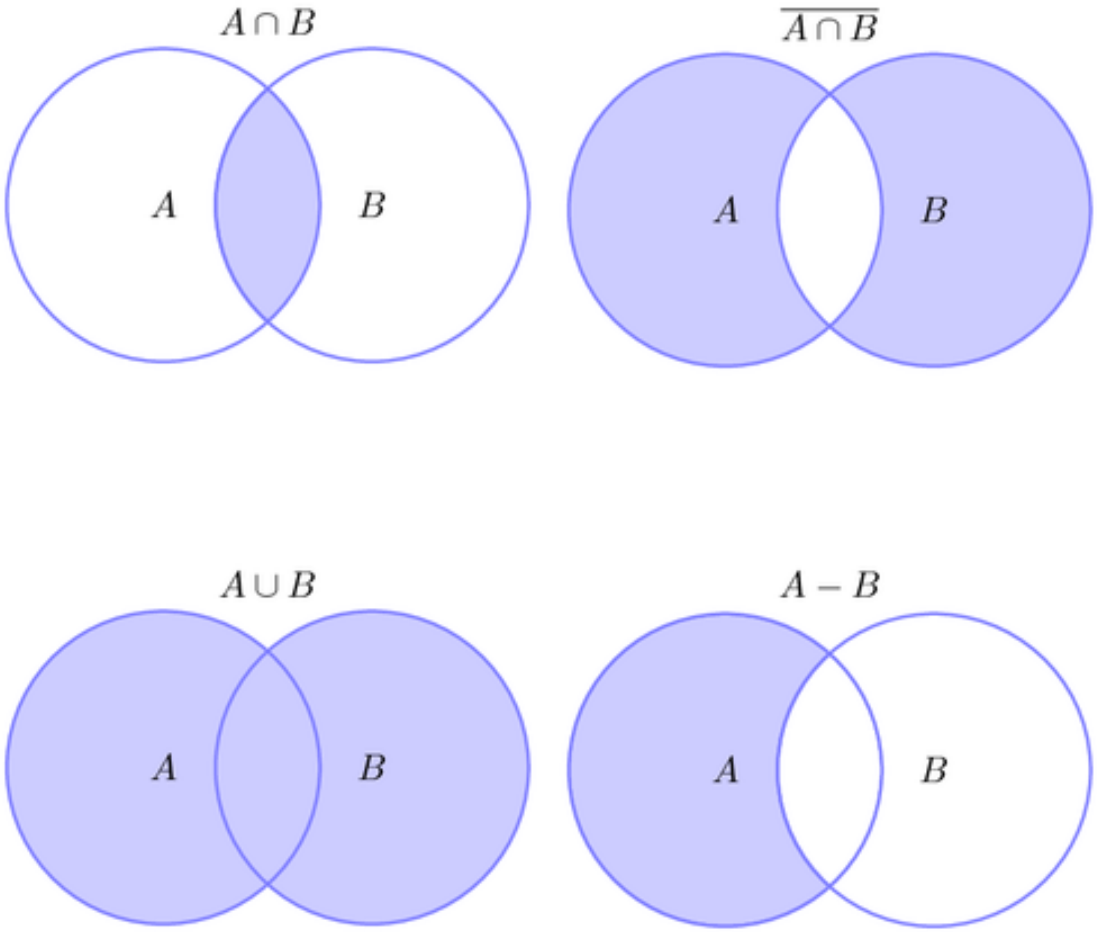
$$p(A) \geq 0$$

The probability $p(B)$ of an event B which is certain to occur is always equal to one, i.e.,

$$p(B) = 1$$

In case two events A and B are **mutually exclusive**, that is, they cannot occur simultaneously $p(A \cap B) = 0$, then the probability of occurrence of either A or B is denoted as $A \cup B$ and is given by

$$p(A \cup B) = p(A) + p(B) - 2p(A \cap B) = p(A) + p(B)$$



Rules of Probability

Consider the slightly more general example involving two random variables X and Y instead of just two events (which are essentially binary random variables). Suppose that X can take any of the values x_i, and Y can take the values y_j. Moreover, consider a total of N trials, and let the number of such trials in which $X = x_i$ and $Y = y_j$ be n_{ij}. Also, let the number of trials in which X takes the value x_i (irrespective of the value that Y takes) be denoted by c_i, and similarly let the number of trials in which Y takes the value y_j be denoted by r_j.

Then the **marginal**, **conditional** and **joint probabilities** are given by

$$p(X = x_i) = \frac{c_i}{N} \quad p(Y = y_j) = \frac{r_j}{N} \quad p(X = x_i, Y = y_j) = \frac{n_{ij}}{N} \quad p(Y = y_j | X = x_i) = \frac{r_j}{c_i}$$

Note that the joint probability $p(X = x_i, Y = y_j)$ is short notation for $p(X = x_i \cap Y = y_j)$.

Sum rule:

$$p(X) = \sum_Y p(X, Y) = \int_Y p(X, Y) dY$$

Applying the sum rule as above is called "marginalizing out Y".

Product rule:

$$p(X, Y) = p(Y|X)p(X)$$

Computing $p(Y|X)$ is called "conditioning on X". The product rule is generalized as follows

$$p(X_1, X_2, \dots, X_K) = p(X_K|X_{K-1}, \dots, X_1)p(X_{K-1}, \dots, X_1), \dots$$

Note that if the *joint distribution* of two random variables factorizes into the product of their marginals, so that $p(X, Y) = p(X)p(Y)$, then X and Y are said to be *statistically independent*. In such case, the product rule becomes $p(Y|X) = p(Y)$.

Bayes Theorem

From the *product rule*, we can immediately obtain the *Bayes' theorem*, using the symmetry property $p(X, Y) = p(Y, X)$ as follows

$$\begin{aligned} p(X, Y) &= p(Y|X)p(X) \Leftrightarrow \\ p(Y|X) &= \frac{p(X, Y)}{p(X)} \Leftrightarrow \\ p(Y|X) &= \frac{p(Y, X)}{p(X)} \Leftrightarrow \\ p(Y|X) &= \frac{p(X|Y)p(Y)}{p(X)} \end{aligned}$$

Using the *sum* and *product rules*, the marginal probability $p(X)$ in the denominator can be expressed in terms of the quantities in the numerator

$$p(Y|X) = \frac{p(X|Y)p(Y)}{\sum_Y p(X, Y)} = \frac{p(X|Y)p(Y)}{\sum_Y p(X|Y)p(Y)}$$

An interpretation of the Bayes theorem is that if we had been asked which is the most probable value of Y, *before* we observe any value for X, then the most complete information we have available is provided by the *prior probability* $p(Y)$. *After* we observe the value of X, we can use the Bayes theorem to compute the the *posterior* probability $p(Y|X)$, which represents our updated knowledge after incorporating the evidence provided by the observed data.

Let $\mathbf{w}$ be parameters and $\mathcal{D}$ be data. Bayes theorem is given by

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})p(\mathbf{w})}{p(\mathcal{D})} \Leftrightarrow \text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}}$$

The frequentist paradigm generally quantifies the properties of data driven quantities in the light of the fixed model parameters, while the Bayesian paradigm generally quantifies the properties of unknown model parameters in light of observed data.

1.2.1 Probability densities

If the probability of a real-valued variable x falling in the interval $(u, u+\delta)$ is given by $p_x(u)\delta$, then $p_x(x)$ is called the probability density function over x.

$$p(u \leq x \leq u+\delta) = \int_u^{u+\delta} p(x)dx = P_x(u+\delta) - P_x(u) = \frac{P_x(u+\delta) - P_x(u)}{\delta}\delta$$

Then we have that

$$\lim_{\delta \to 0} \frac{P_x(u+\delta) - P_x(u)}{\delta}\delta = \frac{dP_x(u)}{dx}\delta = p_x(x)\delta$$

Therefore, the probability that x will lie in an interval (a, b) is then given by

$$p(a \leq x \leq b) = \int_a^b p_x(x)dx$$

and it must satisfy the following two conditions

$$p_x(x) \geq 0$$
$$\int_{-\infty}^{\infty} p_x(x)dx = 1$$

The probability that x lies in the interval $(-\infty, z)$ is given by the *cumulative distribution function* (CDF) given by

$$P_x(z) = \int_{-\infty}^{z} p_x(x)dx$$

where

$$p(x) = \frac{dP_x(x)}{dx}$$

If x is a discrete variable, then $p_x(x)$ is sometimes called a *probability mass function* (PMF) because it can be regarded as a set of *probability masses* concentrated at the allowed values of x.

1.2.2 Expectations and covariances

The average value of some function $f(x)$ under a probability distribution $p_x(x)$ is called the *expectation* of $f(x)$ and is denoted by $\mathbb{E}[f]$. The average is weighted by the relative probabilities of the different values of x as follows

$$\mathbb{E}[f] = \sum_x p_x(x)f(x) = \int p_x(x)f(x)dx$$

For a finite number of N points drawn from the probability distribution, then the expectation can be approximated as a finite sum over these points

$$\mathbb{E}[f] \approx \frac{1}{N}\sum_{n=1}^{N} f(x_n)$$

Note that the expectations of functions of several variables, may use a subscript to indicate which variable is being averaged, i.e., $\mathbb{E}_x[f(x,y)]$.

Whereas the expectation provides a measure of centrality, the variance of a random variable quantifies the spread of that random variable's distribution. Thus, the variance provides a measure of how much variability there is in $f(x)$ around its mean value $\mathbb{E}[f(x)]$ and is defined as follows

$$\begin{aligned}
\text{var}[f] &= \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2] \\
&= \mathbb{E}[f(x)^2 - 2f(x)\mathbb{E}[f(x)] + \mathbb{E}[f(x)]^2] \\
&= \mathbb{E}[f(x)^2] - \mathbb{E}[2f(x)\mathbb{E}[f(x)]] + \mathbb{E}[\mathbb{E}[f(x)]^2] \\
&= \mathbb{E}[f(x)^2] - 2\mathbb{E}[f(x)]\mathbb{E}[f(x)] + \mathbb{E}[f(x)]^2 \\
&= \mathbb{E}[f(x)^2] - \mathbb{E}[f(x)]^2
\end{aligned}$$

The covariance expresses the extent to which x and y vary together and is given by

$$\text{cov}[x,y] = \mathbb{E}_{x,y}[(x - \mathbb{E}[x])(y - \mathbb{E}[y])] = \mathbb{E}_{x,y}[xy] - \mathbb{E}[x]\mathbb{E}[y]$$

A covariance matrix has entries σ_{ij} corresponding to the covariance of variables i and j. If two variables are independent, then their covariance vanishes, e.g., $= \mathbf{I}$.

1.2.4 The Gaussian distribution

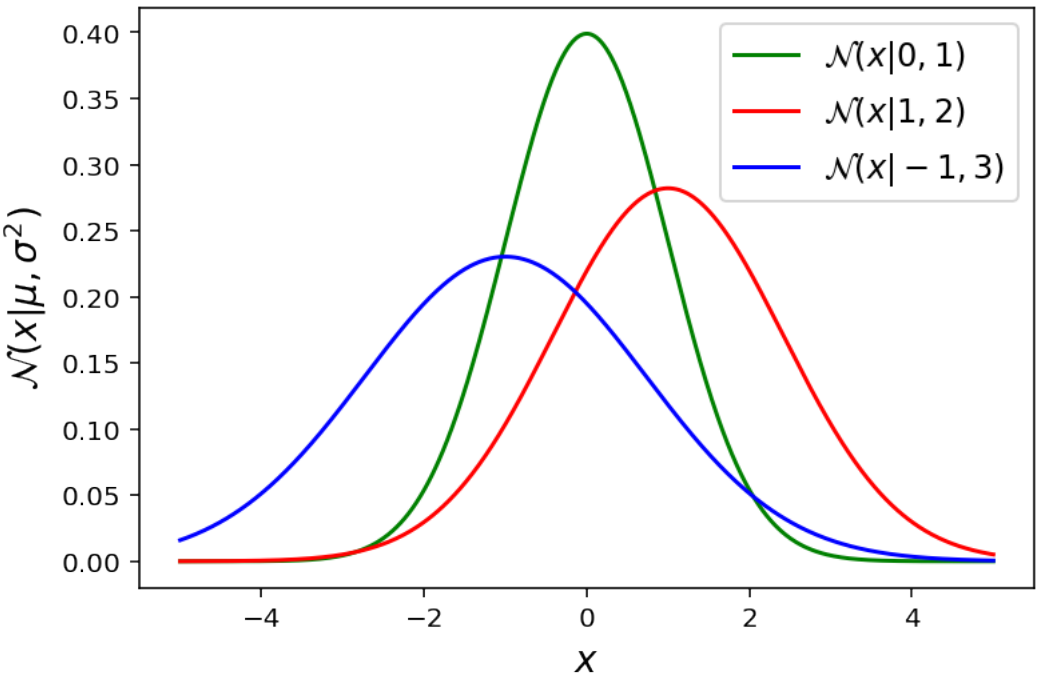
The *Normal* or *Gaussian* distribution is one of the most important probability distributions for continuous variables. For the case of a single real-valued variable x, the distribution is defined as follows

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

and is governed by the parameters μ, called the *mean*, and σ^2, called the *variance*. The square root of the variance σ is called *standard deviation*. An alternative way to represent a Gaussian distribution is by considering a *precision* term $\beta = \frac{1}{\sigma^2}$, denoted by

$$\mathcal{N}(x|\mu,\beta^{-1}) = \frac{\beta^{1/2}}{\sqrt{2\pi\sigma^2}} \exp\Big\{-\frac{\beta}{2}(x-\mu)^2\Big\}$$

[8]:
```
x_space = np.linspace(-5, 5, 1000)

for mean, var, c in [(0, 1, "g"), (1, 2, "r"), (-1, 3, "b")]:

    N_distribution = Gaussian(mean, var)
    y = [N_distribution.pdf(x) for x in x_space]

    plt.plot(x_space, y, color=c, label="$\mathcal{N}$" + "$(x|{},{})$".
 ↪format(mean, var))

plt.xlabel("$x$", fontsize=14); plt.ylabel("$\mathcal{N}(x|\mu,\sigma^2)$",␣
 ↪fontsize=14)
plt.legend(fontsize=12); plt.show()
```



Note that the probability density function is *not an actual probability*, therefore it can take values $\mathcal{N}(x|\mu,\sigma^2) > 1$. We can see that the Gaussian distribution satisfies

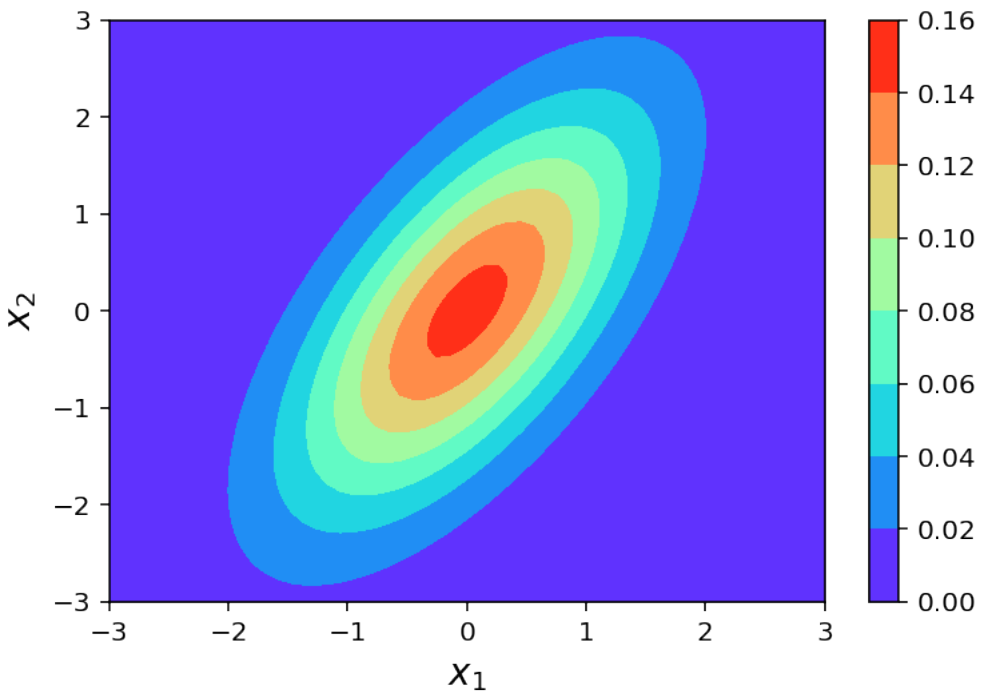
$$\mathcal{N}(x|\mu,\sigma^2) > 0$$
$$\int \mathcal{N}(x|\mu,\sigma^2)dx = 1$$

The Gaussian distribution can be also defined over a D-dimensional vector $\mathbf{x}$ of continuous variables as follows:

$$\mathcal{N}(\mathbf{x}|\ ,\ ) = \frac{1}{(2\pi)^{D/2}|\ |^{1/2}} \exp\left\{ -\frac{1}{2}(\mathbf{x} -\ )^{\mathrm{T}}\ ^{-1}(\mathbf{x} -\ )\right\}$$

where the D-dimentional vector holds the mean of each dimension, while the $D \times D$ matrix is the covariance.

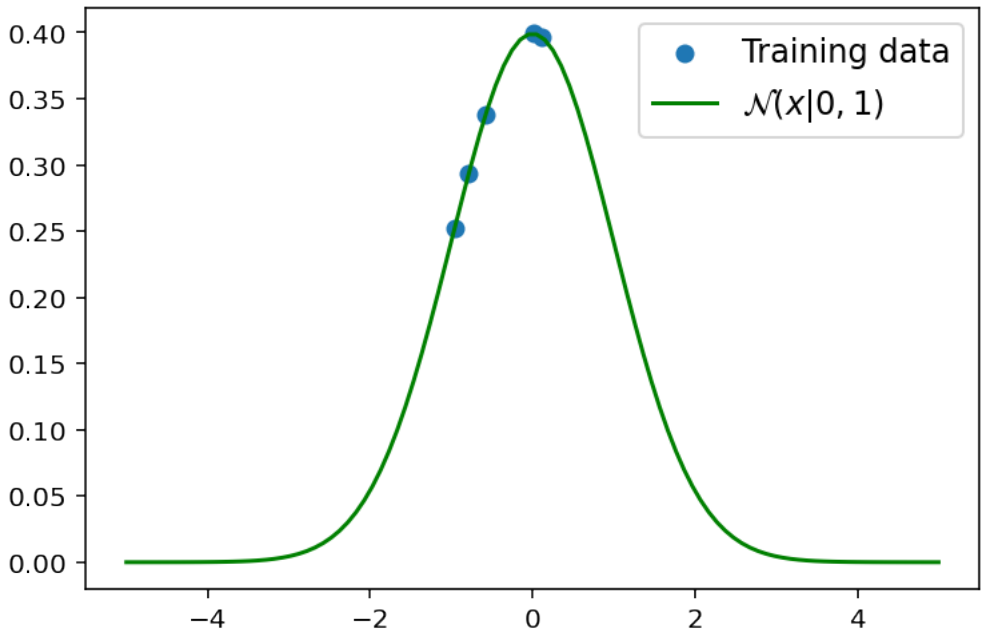
[9]:
```
mean = np.array([0, 0])
sigma = np.array(
    [[1.0, 0.92],
     [0.92, 2.0]]
)

N_distribution = MultivariateGaussian(mean, sigma)

N = 100
x1, x2 = np.meshgrid(np.linspace(-5, 5, N), np.linspace(-5, 5, N))
p = np.zeros((N, N))
for i in range(N):
    for j in range(N):
        p[i, j] = N_distribution.pdf(np.array([x1[i, j], x2[i, j]]))

cp = plt.contourf(x1, x2, p, cmap='rainbow')
plt.colorbar(cp)
plt.xlabel('$x_1$', fontsize=14); plt.ylabel('$x_2$', fontsize=14)
plt.axis([-3, 3, -3, 3])
plt.show()
```



Maximum likelihood

Consider we have a data set of observations $\mathbf{x} = (x_1, \ldots, x_N)^{\mathrm{T}}$ drawn independently from a Gaussian distribution. Data points that are drawn independently from the same distribution are said to be *independent and identically distributed*.

[10]:
```
N_distribution = Gaussian(0, 1)
x_sample_data = N_distribution.draw(5)
y_sample_data = [N_distribution.pdf(x) for x in x_sample_data]

x_space = np.linspace(-5, 5, 100)
y = [N_distribution.pdf(x) for x in x_space]

plt.scatter(x_sample_data, y_sample_data, label="Training data")
plt.plot(x_space, y, color="g", label="$\mathcal{N}(x|0,1)$")
plt.legend(fontsize=12); plt.show()
```



Because the data are independent, the likelihood function of the univariate Gaussian is as follows,

$$p(\mathrm{x}|\mu, \sigma^2) = \prod_{n=1}^{N} \mathcal{N}(x_n|\mu, \sigma^2)$$

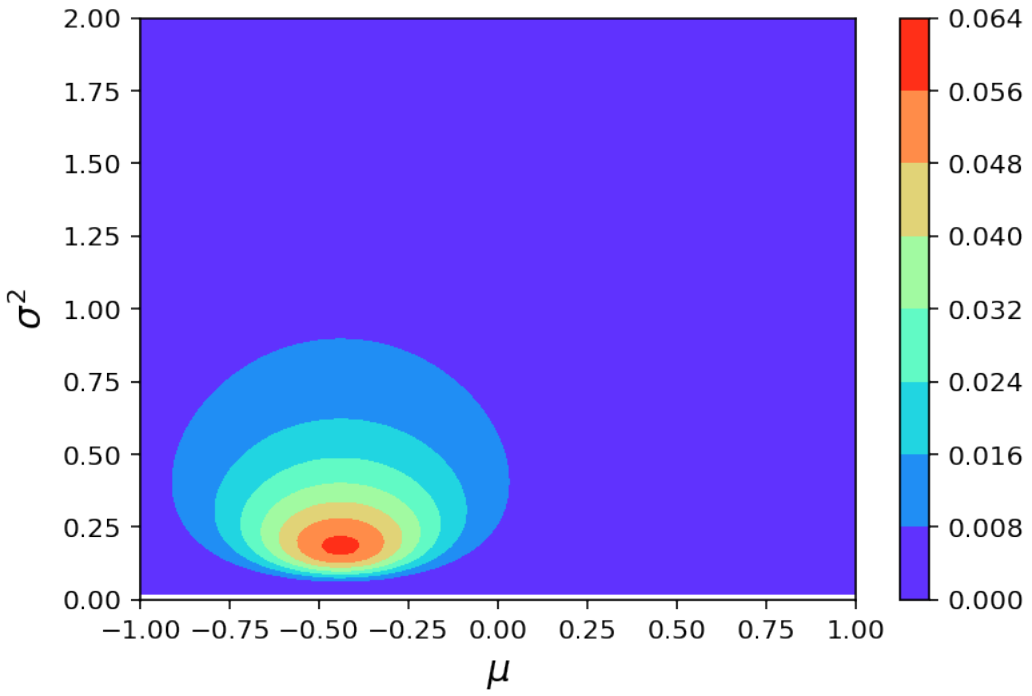
which corresponds to the product of the blue points in the figure above. Therefore, in order to find the unknown parameters μ and σ^2 is to use the observed data set and find the parameter values that maximize the likelihood function.

The log likelihood function can be written as follows

$$
\begin{aligned}
\ln p(\mathrm{x}|\mu,\sigma^2) &= \ln\left[\prod_{n=1}^{N}\mathcal{N}(x_n|\mu,\sigma^2)\right] \\
&= \sum_{n=1}^{N}\ln\mathcal{N}(x_n|\mu,\sigma^2) \\
&\overset{(1.46)}{=} \sum_{n=1}^{N}\ln\left(\frac{1}{(2\pi\sigma^2)^{1/2}}\exp\left\{-\frac{1}{2\sigma^2}(x_n-\mu)^2\right\}\right) \\
&= \sum_{n=1}^{N}\ln\left(\frac{1}{(2\pi\sigma^2)^{1/2}}\right) + \sum_{n=1}^{N}\ln\left(\exp\left\{-\frac{1}{2\sigma^2}(x_n-\mu)^2\right\}\right) \\
&= N\ln\left(\frac{1}{(2\pi\sigma^2)^{1/2}}\right) - \sum_{n=1}^{N}\frac{1}{2\sigma^2}(x_n-\mu)^2 \\
&= N\ln 1 - N\ln(2\pi\sigma^2)^{1/2} - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2 \\
&\overset{\ln 1=0}{=} -N\ln(2\pi\sigma^2)^{1/2} - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2 \\
&\overset{\ln x^y = y\ln x}{=} -\frac{N}{2}\ln 2\pi\sigma^2 - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2 \\
&= -\frac{N}{2}\ln 2\pi - \frac{N}{2}\ln\sigma^2 - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2
\end{aligned}
$$

In the machine learning literature, the negative log of the likelihood function is called an error function. Since the negative logarithm is a monotonically decreasing function, maximizing the likelihood is equivalent to minimizing the error. The following figure depicts the likelihood of μ against σ^2.

[11]:
```
mu_space = np.linspace(-1, 1, 100)
var_space = np.linspace(0, 2, 100)
mu_mesh, var_mesh = np.meshgrid(mu_space, var_space)
ll = np.zeros((mu_space.size, var_space.size))

for i, mu  in enumerate(mu_space):
    for j, var  in enumerate(var_space):
        ll[i, j] = Gaussian(mu, var).likelihood_iid(x_sample_data)

cp = plt.contourf(mu_mesh, var_mesh, ll.T, cmap='rainbow')
plt.colorbar(cp)
plt.xlim(-1, 1); plt.ylim(0, 2)
plt.xlabel("$\mu$", fontsize=14); plt.ylabel("$\sigma^2$", fontsize=14)
plt.show()
```



Then by maximizing with respect to μ, we obtain the following solution

$$
\begin{aligned}
&\frac{\partial \ln p(\mathsf{x}|\mu,\sigma^2)}{\partial \mu} = 0 \Leftrightarrow \\
&\frac{\partial}{\partial \mu}\left(\frac{N}{2}\ln 2\pi - \frac{N}{2}\ln\sigma^2 - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2\right) = 0 \Leftrightarrow \\
&\frac{\partial}{\partial \mu}\left(\frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2\right) = 0 \Leftrightarrow \\
&\frac{\partial}{\partial \mu}\sum_{n=1}^{N}(x_n-\mu)^2 = 0 \Leftrightarrow \\
&\sum_{n=1}^{N}(2\mu - 2x_n) = 0 \Leftrightarrow 2\sum_{n=1}^{N}(\mu - x_n) = 0 \Leftrightarrow \\
&N\mu - \sum_{n=1}^{N}x_n = 0 \Leftrightarrow N\mu = \sum_{n=1}^{N}x_n \Leftrightarrow \\
&\mu_{ML} = \frac{1}{N}\sum_{n=1}^{N}x_n
\end{aligned}
$$

which is the *sample mean* of the observed values x. In a similar manner, maximizing with respect to σ^2, we obtain the maximum likelihood solution for the variance as follows

$$
\begin{aligned}
&\frac{\partial \ln p(\mathsf{x}|\mu,\sigma^2)}{\partial \sigma^2} = 0 \Leftrightarrow \\
&\frac{\partial}{\partial \sigma^2}\left(\frac{N}{2}\ln 2\pi - \frac{N}{2}\ln\sigma^2 - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2\right) = 0 \Leftrightarrow \\
&\frac{\partial}{\partial \sigma^2}\left(-\frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2 - \frac{N}{2}\ln\sigma^2\right) = 0 \Leftrightarrow \\
&\frac{\partial}{\partial \sigma^2}\left(-\frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2\right) + \frac{\partial}{\partial \sigma^2}\left(-\frac{N}{2}\ln\sigma^2\right) = 0 \Leftrightarrow \\
&\frac{\partial}{\partial \sigma^2}\left(-(2\sigma^2)^{-1}\sum_{n=1}^{N}(x_n-\mu)^2\right) + -\frac{N}{2\sigma^2} = 0 \Leftrightarrow \\
&\frac{1}{4\sigma^4}\sum_{n=1}^{N}(x_n-\mu)^2 - \frac{N}{2\sigma^2} = 0 \Leftrightarrow \\
&\frac{1}{4\sigma^4}\sum_{n=1}^{N}(x_n-\mu)^2 = \frac{N}{2\sigma^2} \Leftrightarrow \\
&\sigma^2_{ML} = \frac{1}{N}\sum_{n=1}^{N}(x_n-\mu_{ML})^2
\end{aligned}
$$

which is the *sample variance* measured with respect to the sample mean μ_{ML}.

A problem that arises in the context of our solutions for the maximum likelihood approach is that it systematically underestimates the variance of the Gaussian distribution. This is an example of a phenomenon called bias and is related to the problem of over-fitting encountered in the context of polynomial curve fitting. First note that the maximum likelihood solutions μ_{ML} and σ_{ML} are functions of the data set values x_n. Now, consider the expectations of these quantities with respect to the data set values, which themselves come from a Gaussian distribution with parameters μ and σ^2, given by

$$
\mathbb{E}[\mu_{ML}] = \mathbb{E}\left[\frac{1}{N}\sum_{n=1}^{N}x_n\right] = \frac{1}{N}\sum_{n=1}^{N}\mathbb{E}[x_n] = \frac{1}{N}N\mu = \mu
$$

and

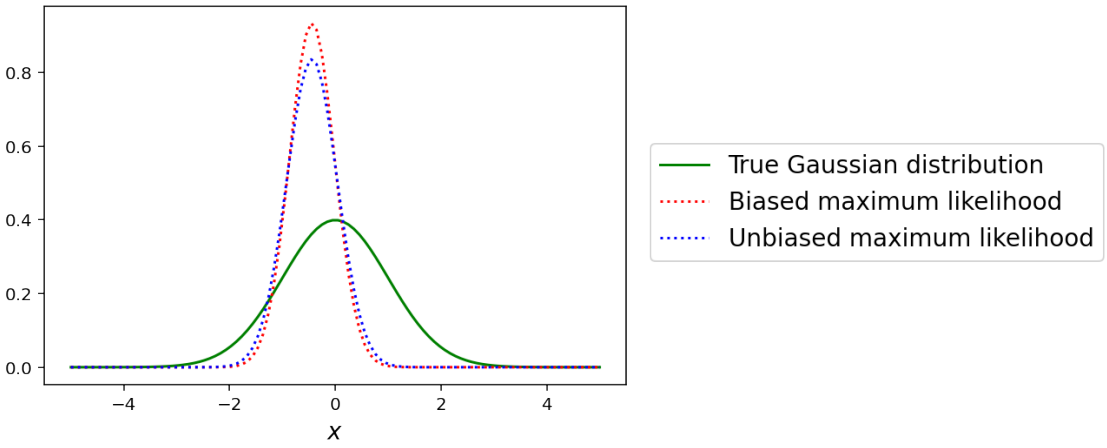
$$
\begin{aligned}
\mathbb{E}[\sigma^2_{ML}] &= \mathbb{E}\left[\frac{1}{N}\sum_{n=1}^{N}(x_n - \mu_{ML})^2\right] \\
&= \frac{1}{N}\mathbb{E}\left[\sum_{n=1}^{N}(x_n - \mu_{ML})^2\right] \\
&= \frac{1}{N}\mathbb{E}\left[\sum_{n=1}^{N}(x_n^2 - 2x_n\mu_{ML} + \mu^2_{ML})\right] \\
&= \frac{1}{N}\left(\mathbb{E}\left[\sum_{n=1}^{N}x_n^2\right] - \mathbb{E}\left[\sum_{n=1}^{N}2x_n\mu_{ML}\right] + \mathbb{E}\left[\sum_{n=1}^{N}\mu^2_{ML}\right]\right) \\
&\overset{(1.50)}{=} \mu^2 + \sigma^2 - \frac{1}{N}\left(\mathbb{E}\left[\sum_{n=1}^{N}2x_n\mu_{ML}\right] + \mathbb{E}\left[\sum_{n=1}^{N}\mu^2_{ML}\right]\right) \\
&\overset{(1.55)}{=} \mu^2 + \sigma^2 - \frac{2}{N}\mathbb{E}\left[\sum_{n=1}^{N}x_n\left(\frac{1}{N}\sum_{n=1}^{N}x_n\right)\right] + \mathbb{E}\left[\left(\frac{1}{N}\sum_{n=1}^{N}x_n\right)^2\right] \\
&= \mu^2 + \sigma^2 - \frac{2}{N^2}\mathbb{E}\left[\sum_{n=1}^{N}x_n\left(\sum_{n=1}^{N}x_n\right)\right] + \frac{1}{N^2}\mathbb{E}\left[\left(\sum_{n=1}^{N}x_n\right)^2\right] \\
&= \mu^2 + \sigma^2 - \frac{2}{N^2}\mathbb{E}\left[\left(\sum_{n=1}^{N}x_n\right)^2\right] + \frac{1}{N^2}\mathbb{E}\left[\left(\sum_{n=1}^{N}x_n\right)^2\right] \\
&= \mu^2 + \sigma^2 - \frac{1}{N^2}\mathbb{E}\left[\left(\sum_{n=1}^{N}x_n\right)^2\right] \\
&= \mu^2 + \sigma^2 - \frac{1}{N^2}(N^2\mu^2 + N\sigma^2)) \\
&= \mu^2 + \sigma^2 - \mu^2 + \frac{1}{N}\sigma^2 \\
&= \frac{N-1}{N}\sigma^2
\end{aligned}
$$

Therefore, on average the maximum likelihood estimate obtains the correct mean but it underestimates the true variance by a factor $(N-1)/N$. The estimate for the unbiased variance parameter is given by

$$
\tilde{\sigma}^2 = \frac{N}{N-1}\sigma^2_{ML} = \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \mu_{ML})^2
$$

[12]:
```
N_distribution.ml(x_sample_data, unbiased=False)
y_biased = [N_distribution.pdf(x) for x in x_space]
N_distribution.ml(x_sample_data, unbiased=True)
y_unbiased = [N_distribution.pdf(x) for x in x_space]

plt.plot(x_space, y, color="g", label="True Gaussian distribution")
plt.plot(x_space, y_biased, color="r", ls=":", label="Biased maximum␣
 ↪likelihood")
```

```
plt.plot(x_space, y_unbiased, color="b", ls=":", label="Unbiased maximum␣
↪likelihood")
plt.xlabel("$x$", fontsize=14)
plt.legend(bbox_to_anchor=(1, 0.7), loc=2, borderaxespad=1, fontsize=14); plt.
↪show()
```



1.2.5 Curve fitting re-visited

Now, lets return to the curve fitting example and examine it from a probabilistic perspective, thereby gaining some insights into error functions and regularization, as well as, taking us towards a full Bayesian treatment. We shall assume that, the target variable t has a Gaussian distribution having mean equal to $y(x, \mathbf{w})$ of the polynomial curve, given by

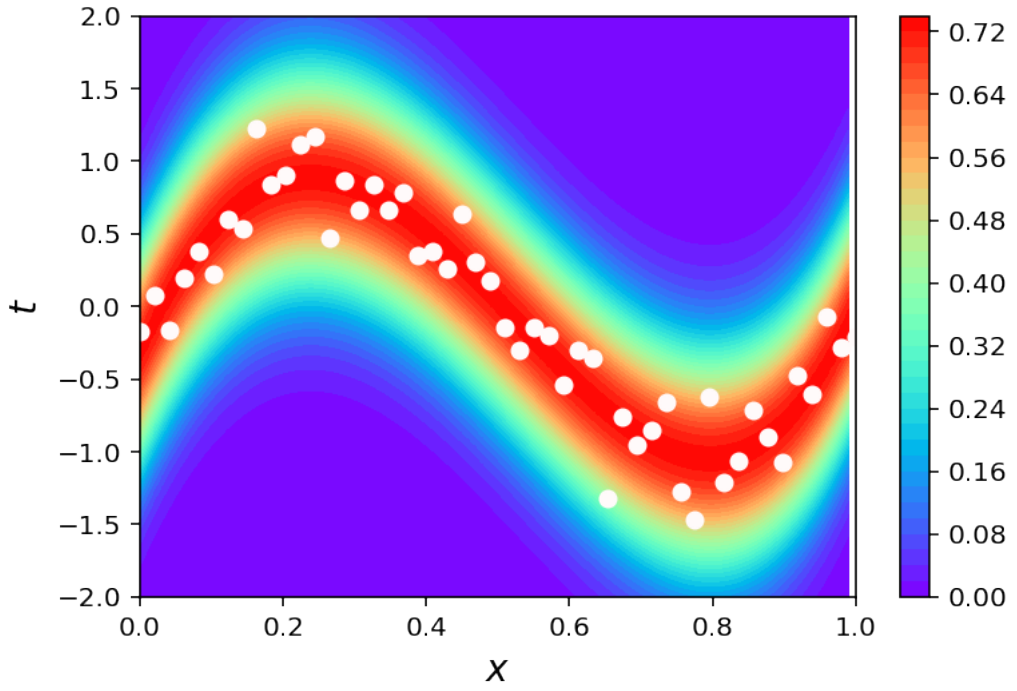
$$p(t|x, \mathbf{w}, \beta) = \mathcal{N}(t|y(x, \mathbf{w}), \beta^{-1})$$

where we have defined β to be the precision parameter, corresponding to the inverse variance.

Each input point x defines a Gaussian distribution located into prediction of $y(x, \mathbf{w})$. We plot the Gaussian distribution heatmap of the values of x against t.

[13]:
```
x_sample, t_sample = generate_toy_data(sin, 50, 0.3)

feature = PolynomialFeature(4)
x_features = feature.transform(x_sample)
model = LinearRegression()
model.fit(x_features, t_sample)

x_space = np.arange(0, 1, 0.01)
t_space = np.arange(-2, 2, 0.01)
X, Y = np.meshgrid(x_space, t_space)
Z = np.zeros((x_space.size, t_space.size))
```

```
for i, x in enumerate(x_space):
    # create the Gaussian distribution for a given input value
    predicted_mu, _ = model.predict(feature.transform(x))
    g = Gaussian(predicted_mu[0], 0.3)
    for j, t in enumerate(t_space):
        # compute the value of the Gaussian for each target value
        Z[i, j] = g.pdf(t)

cp = plt.contourf(X, Y, Z.T.reshape(X.shape), cmap="rainbow", levels=40)
plt.scatter(x_sample, t_sample, color='snow')
plt.xlim(0, 1); plt.ylim(-2, 2)
plt.xlabel("$x$", fontsize=14); plt.ylabel("$t$", fontsize=14)
plt.colorbar(cp)
plt.show()
```



Then, we can use the training data x, t to determine the values of the unknown parameters $\mathbf{w}, \beta$ by maximum likelihood. Assuming the data are i.i.d, the likelihood function is given by

$$p(\mathsf{t}|\mathsf{x}, \mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(t_n|y(x, \mathbf{w}), \beta^{-1})$$

Similar to the Gaussian distribution earlier, we maximize the logarithm of the likelihood function in the form

$$\ln p(\mathsf{t}|\mathsf{x}, \mathbf{w}, \beta) = -\frac{\beta}{2}\sum_{n=1}^{N}(y(x, \mathbf{w}) - t_n)^2 + \frac{N}{2}\ln\beta - \frac{N}{2}\ln 2\pi$$

For determining the maximum likelihood solution for the polynomial coefficients $\mathbf{w}_{ML}$, we can omit the terms that do not depend on $\mathbf{w}$. Since the β does not alter the location of the function it can be replaced by 1. Therefore, the maximization is equivalent to minimizing the *sum of squares error function*, as presented the polynomial curve fitting example. The sum of squares error function has arisen as a consequence of maximizing the likelihood under the assumption of a Gaussian noise distribution!

As a final step, we can also use maximum likelihood to determine the precision parameter β, as follows:

$$\begin{aligned}
&\frac{\partial}{\partial\beta}\ln p(\mathsf{t}|\mathsf{x}, \mathbf{w}, \beta) = 0 \Leftrightarrow \\
&\frac{\partial}{\partial\beta}\Big[-\frac{\beta}{2}\sum_{n=1}^{N}(y(x, \mathbf{w}) - t_n)^2 + \frac{N}{2}\ln\beta - \frac{N}{2}\ln 2\pi\Big] = 0 \Leftrightarrow \\
&\frac{\partial}{\partial\beta}\Big[-\frac{\beta}{2}\sum_{n=1}^{N}(y(x, \mathbf{w}) - t_n)^2\Big] + \frac{\partial}{\partial\beta}\Big[\frac{N}{2}\ln\beta\Big] = 0 \Leftrightarrow \\
&-\frac{1}{2}\sum_{n=1}^{N}(y(x, \mathbf{w}) - t_n)^2 + \frac{N}{2}\frac{1}{\beta} = 0 \Leftrightarrow \\
&\sum_{n=1}^{N}(y(x, \mathbf{w}) - t_n)^2 = \frac{N}{\beta} \Leftrightarrow \\
&\frac{1}{\beta_{ML}} = \frac{1}{N}\sum_{n=1}^{N}(y(x, \mathbf{w}) - t_n)^2
\end{aligned}$$

Since we have a probabilistc model, we can use the predictive distribution that gives the probability distribution over t, rather than a simple point estimate.

$$p(t|x, \mathbf{w}_{ML}, \beta_{ML}) = \mathcal{N}(t|y(x, \mathbf{w}_{ML}), \beta_{ML}^{-1})$$

We can further introduce a prior distribution over the polynomial coefficients $\mathbf{w}$, in order to take a step towards a more Bayesian approach. Consider for instance a Gaussian prior of the form

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I}) = \Big(\frac{\alpha}{2\pi}^{(M+1)/2}\Big)\exp\Big\{-\frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}\Big\}$$

where α is the precision of the multivariate Gaussian, and M is the order of the polynomial, that is, the dimension of the paremeter vector $\mathbf{w}$. Then from the Bayes theorem we have that

$$p(\mathbf{w}|\mathsf{x}, \mathsf{t}, \alpha, \beta) \propto p(\mathsf{t}|\mathsf{x}, \mathbf{w}, \beta)p(\mathbf{w}|\alpha)$$

Therefore, we can determine $\mathbf{w}$, by maximizing the posterior distribution. This technique is known as *maximum posterior* or MAP inference. The maximum of the posterior is given by the minimum of the negative logarithm, which it can be proved to be

$$\frac{\beta}{2}\sum_{n=1}^{N}(y(x,\mathbf{w})-t_n)^2+\frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$$

Thus we see that maximizing the posterior distribution is equivalent to minimizing the regularized sum of squared error function encountered in Regularization, where $\lambda=\alpha/\beta$.

1.6 Information Theory

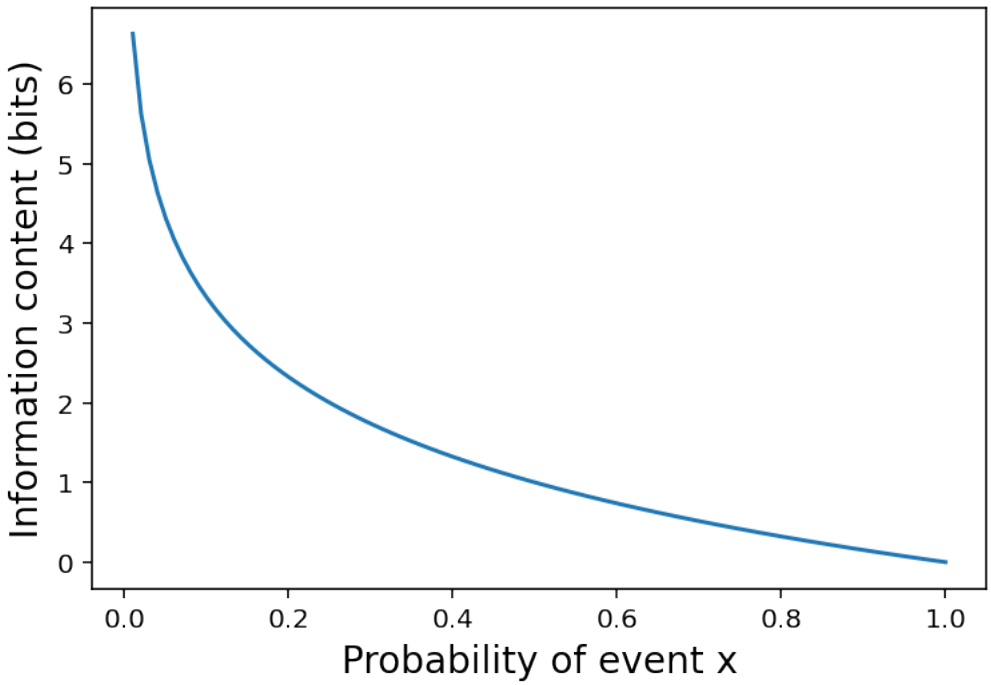
How much information is received when we observe a specific variable?

Consider for instance the event of seeing an alien spaceship appearing in the sky. We have never seen one and we would be extremely surprised if one day such an event occured, since it is unexpected given our current knowledge. Therefore, the amount of information can be viewed as the *degree of surprise* on learning the value of x. The measure of information content is therefore depend on the probability distribution $p(x)$ and in particular is given by the logarithm of $p(x)$ as follows,

$$\mathrm{h}(x)=-\log_2 p(x)$$

where the negative sign ensures that $h(x)\geq 0$. When the base of the logarithm is 2 the units of $h(x)$ are **bits**.

[14]:
```
def h(prob: float) -> float:
    if not 0 <= prob <= 1:
        raise ValueError("Probability should be [0,1].")
    elif prob == 0:
        return float('-inf')
    else:
        return -math.log2(prob)

px_space = np.linspace(0, 1, 100)
y = [h(px) for px in px_space]

plt.plot(px_space, y)
plt.xlabel("Probability of event x", fontsize=14); plt.ylabel("Information␣
 ↪content (bits)", fontsize=14)
plt.show()
```

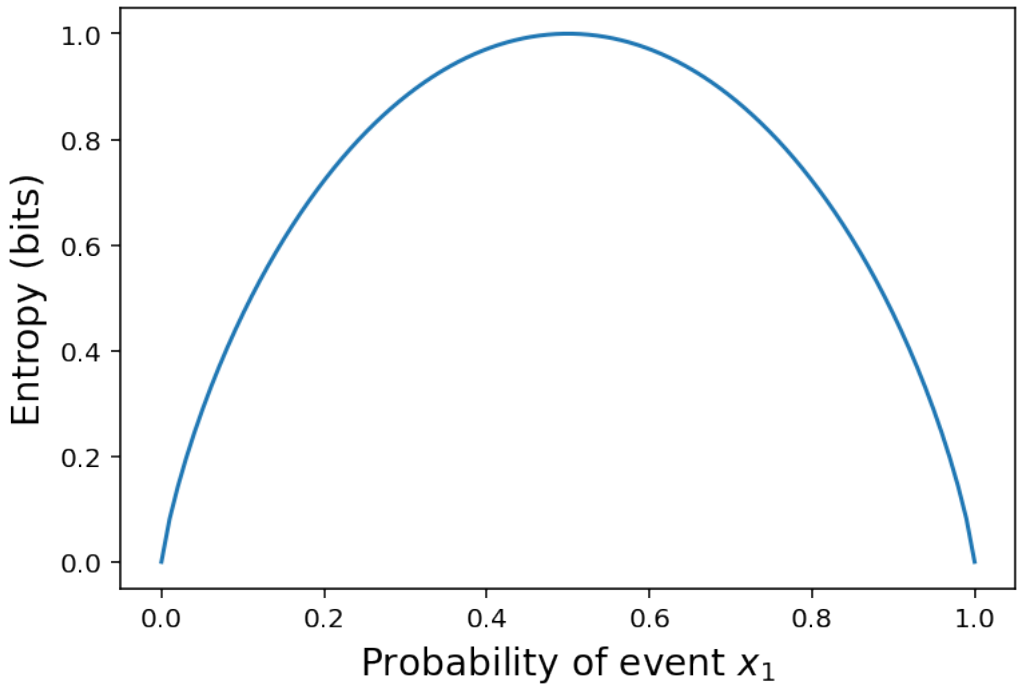


The average amount of information is obtained by taking the expectation over the information content, given by,

$$\mathrm{H}[x] = -\sum_x p(x)h(x) = -\sum_x p(x)\log_2 p(x)$$

This important quantity is called *entropy* of the random variable X. Consider a discrete random variable X having two possible values x_1 and x_2 with probabilities p and $1-p$.

[15]:
```
def entropy(probs):
    if 0 in probs:
        return 0
    else:
        return sum([px * h(px) for px in probs])

px_space = np.linspace(0, 1, 100)
y = [entropy([px, 1-px]) for px in px_space]

plt.plot(px_space, y)
plt.xlabel("Probability of event $x_1$", fontsize=14); plt.ylabel("Entropy␣
 ↪(bits)", fontsize=14)
plt.show()
```



Note that the entropy is maximized when the two events are equiprobable, that is, for uniform distributions. The entory is also defined in a similar way for continous random variables and is called *differential entropy*:

$$\mathrm{H}[\mathbf{x}] = -\int p(\mathbf{x}) \ln p(\mathbf{x}) \mathrm{d}\mathbf{x}$$

In the case of a joint distribution $p(\mathbf{x}, \mathbf{y})$, the average additional information needed to specify $\mathbf{y}$ given that the value of $\mathbf{x}$ is already known, is called *conditional entropy*, and is given by

$$\mathrm{H}[\mathbf{y}|\mathbf{x}] = -\int p(\mathbf{x}, \mathbf{y}) \ln p(\mathbf{y}|\mathbf{x}) \mathrm{d}\mathbf{x} \mathrm{d}\mathbf{y}$$

Moreover it is easily seen, using the product rule, that the conditional entropy satisfies the relation,

$$
\begin{aligned}
\mathrm{H}[\mathbf{x},\mathbf{y}] &= -\int\int p(\mathbf{x},\mathbf{y})\ln p(\mathbf{x},\mathbf{y})\mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\
&= -\int\int p(\mathbf{x},\mathbf{y})\ln\left(p(\mathbf{y}|\mathbf{x})p(\mathbf{x})\right)\mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\
&= -\int\int p(\mathbf{x},\mathbf{y})(\ln p(\mathbf{y}|\mathbf{x}) + \ln p(\mathbf{x}))\mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\
&= -\int\int p(\mathbf{x},\mathbf{y})\ln p(\mathbf{y}|\mathbf{x})\mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} - \int\int p(\mathbf{x},\mathbf{y})\ln p(\mathbf{x})\mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\
&= -\int\int p(\mathbf{x},\mathbf{y})\ln p(\mathbf{y}|\mathbf{x})\mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} - \int\int p(\mathbf{x},\mathbf{y})\mathrm{d}\mathbf{y}\ln p(\mathbf{x})\mathrm{d}\mathbf{x} \\
&= -\int\int p(\mathbf{x},\mathbf{y})\ln p(\mathbf{y}|\mathbf{x})\mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} - \int p(\mathbf{x})\ln p(\mathbf{x})\mathrm{d}\mathbf{x} \\
&= \mathrm{H}[\mathbf{y}|\mathbf{x}] + \mathrm{H}[\mathbf{x}]
\end{aligned}
$$

stating that the information needed to describe $\mathbf{x}$ and $\mathbf{y}$ is given by the information needed to describe $\mathbf{x}$ alone plus the additional information to specify $\mathbf{y}$.

1.6.1 Relative entropy and mutual information

Consider some unknown distribution $p(\mathbf{x})$ that we have modelled using an approximate distribution $q(\mathbf{x})$. Then, the average *additional* information required to specify a value of $\mathbf{x}$ as a result of using $q(\mathbf{x})$ instead of $p(\mathbf{x})$ is given by

$$
\mathrm{KL}(p||q) = -\int p(\mathbf{x})\ln q(\mathbf{x})d\mathbf{x} - \left(-\int p(\mathbf{x})\ln q(\mathbf{x})d\mathbf{x}\right) = -\int p(\mathbf{x})(\ln q(\mathbf{x}) - \ln p(\mathbf{x}))d\mathbf{x} = -\int p(\mathbf{x})\ln\left(\frac{q(\mathbf{x})}{p(\mathbf{x})}\right)d\mathbf{x}.
$$

This is known as the *relative entropy* or *Kullback-Leibler divergence* between distributions $p(\mathbf{x})$ and $q(\mathbf{x})$. Note that it is not a symmetrical quantity, that is to say $\mathrm{KL}(p||q) \neq \mathrm{KL}(q||p)$. Moreover the Kullback-Leibler divergence satisfies $\mathrm{KL}(p||q) \geq 0$, where the equality holds if, and only if, $p(\mathbf{x}) = q(\mathbf{x})$. Thus, we can interpret the Kullback-Leibler divergence as a measure of dissimilarity of the two distributions.

Since the most efficient compression is achieved when we know the true distribution, otherwise additional information is required, there is an important relationship between data compression and density estimation.

Suppose we are trying to approximate the unknown distribution using some parametric distribution $q(\mathbf{x}|\ )$, governed by a set of adjustable parameters , for instance a multivariate Gaussian distribution. One way to determine is to minimize the Kullback-Leibler divergence between $p(\mathbf{x})$ and $q(\mathbf{x}|\ )$. This is impossible since we do not know $p(\mathbf{x})$. However, if we have observed a finite set of training points $\mathbf{x}_n$ drawn from $p(\mathbf{x})$, then the expectation with respect to $p(\mathbf{x})$ can be approximated by a finite sum over these points, using (1.35), so that,

$$
\mathrm{KL}(p||q) \approx \frac{1}{N}\sum_{n=1}^{N}\left(-\ln q(\mathbf{x}_n|\ ) + \ln p(\mathbf{x}_n)\right)
$$

Since the second term is independent of , minimizing this **approximated** Kullback-Leibler divergence is equivalent to minimizing the log likelihood function of $q(\mathbf{x}_n|\ )$ evaluated on the training set.

Mutual information

Consider the joint distribution $p(\mathbf{x}, \mathbf{y})$ between two sets of variables $\mathbf{x}$ and $\mathbf{y}$. If the sets are independent, then $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{x})(\mathbf{y})$. If the variables are not independent, we can measure whether they are *close* to being independent* by considering the Kullback-Leibler divergence between the joint distribution and the product of their marginals, given by

$$\mathrm{I}[\mathbf{x}, \mathbf{y}] = \mathrm{KL}(p(\mathbf{x}, \mathbf{y})||p(\mathbf{x})p(\mathbf{y})) = -\int\int p(\mathbf{x}, \mathbf{y}) \ln \left(\frac{p(\mathbf{x})p(\mathbf{y})}{p(\mathbf{x}, \mathbf{y})} \right) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y}$$

which is called the *mutual information* of variables $\mathbf{x}$ and $\mathbf{y}$. The mutual information satisfies $\mathrm{I}[\mathbf{x}, \mathbf{y}] \geq 0$ where the equality holds if, and only if, $\mathbf{x}$ and $\mathbf{y}$ are independent. Moreover, using the sum and product rules of probability, we can prove that the mutual information is related to the conditional entropy,

$$\begin{aligned}
\mathrm{I}[\mathbf{x}, \mathbf{y}] &= -\int\int p(\mathbf{x}, \mathbf{y}) \ln \left(\frac{p(\mathbf{x})p(\mathbf{y})}{p(\mathbf{x}, \mathbf{y})} \right) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\
&= -\int\int p(\mathbf{x}, \mathbf{y}) \ln \left(\frac{p(\mathbf{x})p(\mathbf{y})}{p(\mathbf{x}|\mathbf{y})p(\mathbf{y})} \right) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\
&= -\int\int p(\mathbf{x}, \mathbf{y}) \ln \left(\frac{p(\mathbf{x})}{p(\mathbf{x}|\mathbf{y})} \right) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\
&= -\int\int p(\mathbf{x}, \mathbf{y}) \Big(\ln p(\mathbf{x}) - \ln p(\mathbf{x}|\mathbf{y}) \Big) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\
&= -\int\int p(\mathbf{x}, \mathbf{y}) \ln p(\mathbf{x}) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} + \int\int p(\mathbf{x}, \mathbf{y}) \ln p(\mathbf{x}|\mathbf{y}) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\
&= -\int\int p(\mathbf{x}, \mathbf{y}) \mathrm{d}\mathbf{y} \ln p(\mathbf{x}) \mathrm{d}\mathbf{x} + \int\int p(\mathbf{x}, \mathbf{y}) \ln p(\mathbf{x}|\mathbf{y}) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\
&= -\int p(\mathbf{x}) \ln p(\mathbf{x}) \mathrm{d}\mathbf{x} + \int\int p(\mathbf{x}, \mathbf{y}) \ln p(\mathbf{x}|\mathbf{y}) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\
&= \mathrm{H}[\mathbf{x}] - \mathrm{H}[\mathbf{x}|\mathbf{y}] = \mathrm{H}[\mathbf{y}] - \mathrm{H}[\mathbf{y}|\mathbf{x}]
\end{aligned}$$

2. Probability Distributions

Table of Contents

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from sklearn.datasets import make_classification
from prml.datasets import load_old_faithful
from prml.distribution import (
    Bernoulli,
    Binomial,
    Beta,
    Categorical,
    Multinomial,
    Dirichlet,
    Gaussian,
    MultivariateGaussian,
    Gamma,
    StudentT
)
from prml.neighbors import (
    NearestNeighborsDensity,
    KNearestNeighborsClassifier
)

# Set random seed to make deterministic
np.random.seed(0)

# Ignore zero divisions and computation involving NaN values.
np.seterr(divide = 'ignore', invalid='ignore')

# Enable higher resolution plots
%config InlineBackend.figure_format = 'retina'
```

2.1 Binary Variables

Consider a single binary random variable $x \in \{0, 1\}$, for instance x might be the outcome of flipping a coin. Then, the probability of heads ($x = 1$) can be denoted by a parameter μ so that,

$$p(x = 1|\mu) = \mu$$

where $0 \leq \mu \leq 1$. The *Bernoulli* probability distribution over x has therefore the form,

$$\text{Bern}(x|\mu) = \mu^x(1 - \mu)^{1-x}$$

It is easily verified that the Bernoulli distribution has mean given by,

$$\mathbb{E}[x] = \sum_{x \in \{0,1\}} x \text{Bern}(x|\mu) = \sum_{x \in \{0,1\}} x \mu^x (1-\mu)^{1-x} = \mu^1 (1-\mu)^0 = \mu$$

and variance given by,

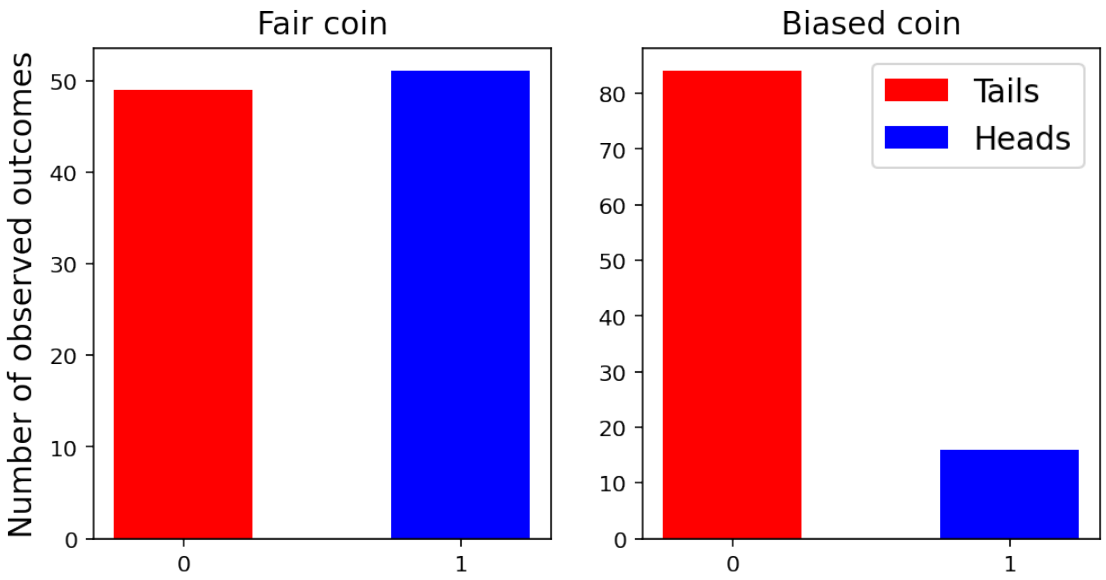
$$\begin{aligned} \text{var}[x] &= \mathbb{E}[x]^2 - \mathbb{E}[x^2] \\ &= \mu^2 - \sum_{x \in \{0,1\}} x^2 \text{Bern}(x|\mu) \\ &= \mu^2 - \sum_{x \in \{0,1\}} x^2 \mu^x (1-\mu)^{1-x} \\ &= \mu^2 - \mu = \mu(\mu - 1) \end{aligned}$$

Now suppose we have given a data set $\mathcal{D} = \{x_1, \ldots, x_N\}$ of observed values sampled from an **unknown** Bernoulli distribution (outcomes of coin tosses), that is, the μ parameter (probability of heads) is unknown.

[2]:
```
# For demonstration purposes we shall create a data set from a fair coin (mu =␣
↪0.5)
# and another data set from a biased coin favoring tails (mu = 0.2).

# Number of coin tosses per data set
N = 100

# Fair coin
mu_fair = 0.5
true_distribution_fair = Bernoulli(mu_fair)
D_fair = true_distribution_fair.draw(N)

print("Fair coin   -- ", "Heads:", sum(D_fair == 1), "Tails:", sum(D_fair == 0))

# Biased coin
mu_biased = 0.2
true_distribution_biased = Bernoulli(mu_biased)
D_biased = true_distribution_biased.draw(N)

print("Biased coin -- ", "Heads:", sum(D_biased == 1), "Tails:", sum(D_biased␣
↪== 0))

plt.figure(figsize=(8, 4))
plt.subplot(1,2,1)
b = plt.bar([0,1], [sum(D_fair == 0), sum(D_fair == 1)], color=['red','blue'],␣
↪width=0.5)
b[0].set_label("Tails")
b[1].set_label("Heads")
```

```
plt.ylabel("Number of observed outcomes", fontsize=14)
plt.xticks([0, 1])
plt.title("Fair coin", fontsize=14)

plt.subplot(1,2,2)
b = plt.bar([0,1], [sum(D_biased == 0), sum(D_biased == 1)],␣
 ↪color=['red','blue'], width=0.5)
b[0].set_label("Tails")
b[1].set_label("Heads")
plt.xticks([0, 1])
plt.title("Biased coin", fontsize=14)
plt.legend(fontsize=14)

plt.show()
```

```
Fair coin   --  Heads: 51 Tails: 49
Biased coin --  Heads: 16 Tails: 84
```



We can use the likelihood function on the assumption that the observations are drawn independently from $\text{Bern}(x|\mu)$, so that,

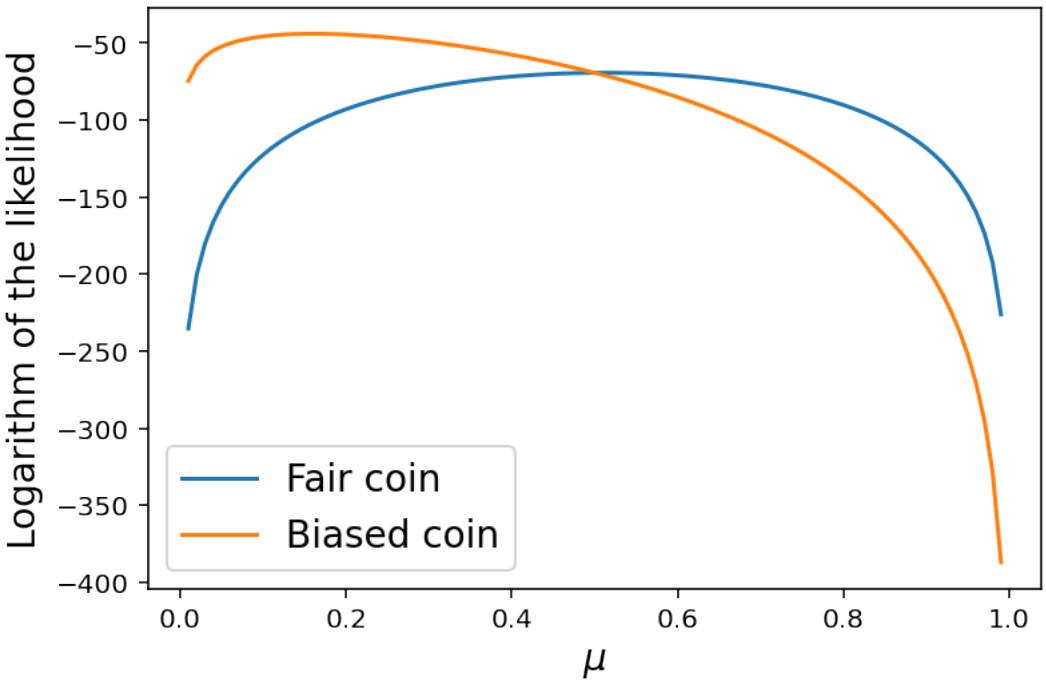
$$p(\mathcal{D}|\mu) = \prod_{n=1}^{N} \text{Bern}(x_n|\mu) = \prod_{n=1}^{N} \mu^{x_n}(1-\mu)^{1-x_n}$$

Then, following the frequentist approach, we can estimate the value of μ by maximizing the logarithm of the likelihood given by

$$
\begin{aligned}
\ln p(\mathcal{D}|\mu) &= \ln \left(\prod_{n=1}^{N} \mu^{x_n} (1-\mu)^{1-x_n} \right) \\
&= \sum_{n=1}^{N} \ln \left(\mu^{x_n} (1-\mu)^{1-x_n} \right) \\
&= \sum_{n=1}^{N} \left(\ln \left(\mu^{x_n} \right) + \ln \left((1-\mu)^{1-x_n} \right) \right) \\
&= \sum_{n=1}^{N} \left(x_n \ln \mu + (1-x_n) \ln(1-\mu) \right)
\end{aligned}
$$

Given an observed data set $\mathcal{D}$, we can plot the value of the log-likelihood against the values of $\mu \in [0, 1]$.

[3]:
```
mu_space = np.arange(0.01, 1, 0.01)
ll = np.zeros(mu_space.shape)

for i, mu_i  in enumerate(mu_space):
    ll[i] = Bernoulli(mu_i).log_likelihood_iid(D_fair)

plt.plot(mu_space, ll, label="Fair coin")

for i, mu_i  in enumerate(mu_space):
    ll[i] = Bernoulli(mu_i).log_likelihood_iid(D_biased)

plt.plot(mu_space, ll, label="Biased coin")

plt.xlabel("$\mu$", fontsize=14); plt.ylabel("Logarithm of the likelihood",␣
 ↪fontsize=14)
plt.legend(fontsize=14); plt.show()
```



Note that the **maximum** of the likelihood is achieved around $\mu = 0.5$ for the fair coin and $\mu = 0.2$ for the biased coin. Therefore, using calculus, if we set the derivative of $\ln p(\mathcal{D}|\mu)$ with respect to μ equal to zero, we obtain

$$
\begin{aligned}
&\frac{\partial}{\partial \mu} \ln p(\mathcal{D}|\mu) = 0 \Leftrightarrow \\
&\frac{\partial}{\partial \mu} \sum_{n=1}^{N} \Big(x_n \ln \mu + (1 - x_n) \ln(1 - \mu) \Big) = 0 \Leftrightarrow \\
&\sum_{n=1}^{N} \frac{\partial}{\partial \mu} \Big(x_n \ln \mu + (1 - x_n) \ln(1 - \mu) \Big) = 0 \Leftrightarrow \\
&\sum_{n=1}^{N} \Big(\frac{1}{\mu} x_n - \frac{1}{1 - \mu} (1 - x_n) \Big) = 0 \Leftrightarrow \\
&\sum_{n=1}^{N} \Big(\frac{1}{\mu} x_n - \frac{1}{1 - \mu} + \frac{1}{1 - \mu} x_n \Big) = 0 \Leftrightarrow \\
&\sum_{n=1}^{N} \Big(\frac{1}{\mu} x_n + \frac{1}{1 - \mu} x_n \Big) = \frac{N}{1 - \mu} \Leftrightarrow \\
&\sum_{n=1}^{N} \Big(\frac{1 - \mu}{\mu} x_n + x_n \Big) = N \Leftrightarrow \\
&\sum_{n=1}^{N} \frac{1}{\mu} x_n = N \Leftrightarrow \\
&\mu_{ML} = \frac{1}{N} \sum_{n=1}^{N} x_n
\end{aligned}
$$

which is known as the *sample mean*. Thus, the probability of landing heads (μ), according to the maximum likelihood estimator, is given by the fraction of observations of heads in the data set $\mathcal{D}$. Indeed caclulating the sample mean gives us a very accurate estimation of the true values for μ.

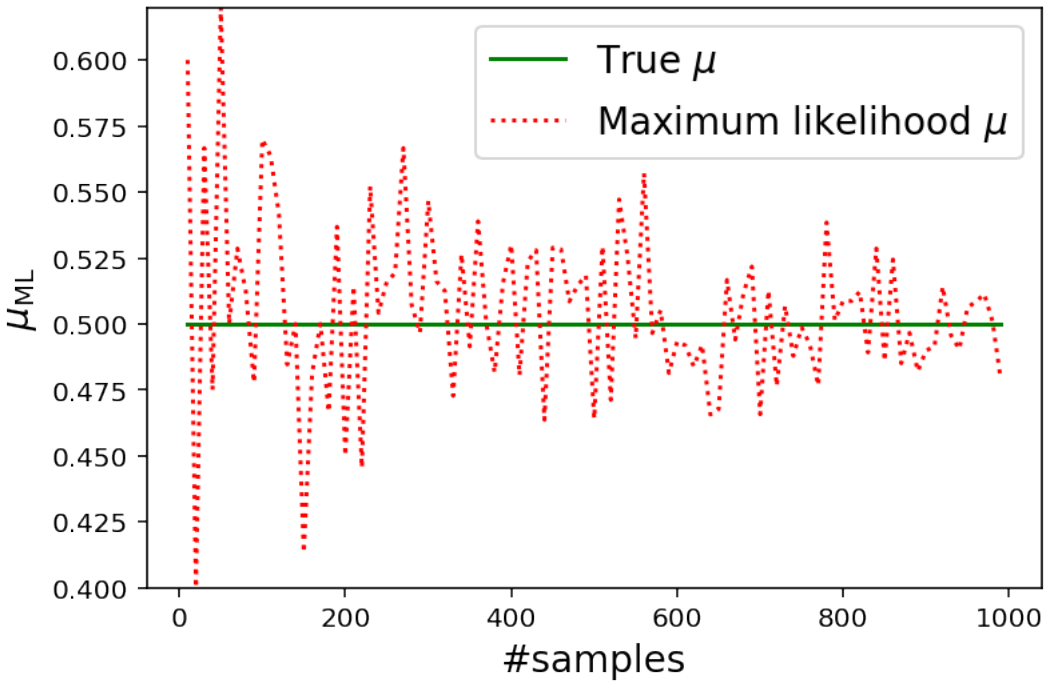
[4]:
```
print("Maximum likelihood  for the fair coin is", sum(D_fair == 1) / N)
print("Maximum likelihood  for the biased coin is", sum(D_biased == 1) / N)
```

```
Maximum likelihood  for the fair coin is 0.51
Maximum likelihood  for the biased coin is 0.16
```

However, A problem that arises from this result is that for small data sets the estimation can be unreasonable. For instance, in the following plot, note that for small data sets $N < 100$, the estimated μ_{ML} can deviate significantly from the true μ leading to incorrect probabilities for the future observations.

[5]:
```
ml_model = Bernoulli()
samples = list(range(10, 1000, 10))
mu_estimations = []

for N in samples:
    D = true_distribution_fair.draw(N)
    ml_model.ml(D)
    mu_estimations.append(ml_model.mu)
```

```
plt.plot(samples, [mu_fair for N in samples], color="g", label="True $\mu$")
plt.plot(samples, mu_estimations, color="r", linestyle=":", label="Maximum␣
↪likelihood $\mu$")
plt.xlabel("#samples", fontsize=14); plt.ylabel("$\mu_{\mathrm{ML}}$",␣
↪fontsize=14)
plt.ylim([min(mu_estimations), max(mu_estimations)])
plt.legend(fontsize=14); plt.show()
```



Now, if we toss a coin 5 times, then, the probability of getting 3 heads and then 2 tails is

$$p(x_1 = 1)p(x_2 = 1)p(x_3 = 1)p(x_4 = 0)p(x_5 = 0) = \mu \times \mu \times \mu \times (1 - \mu) \times (1 - \mu)$$

or more general for N times and m heads,

$$\mu^m (1 - \mu)^{N-m}$$

However, this is just one way to get m heads, there are many other sequences of N trials that would give us m heads. How many exactly?

The distribution describing the number of m observations of $x = 1$ in a sample dataset of size N is called the *Binomial* distribution, and is given by

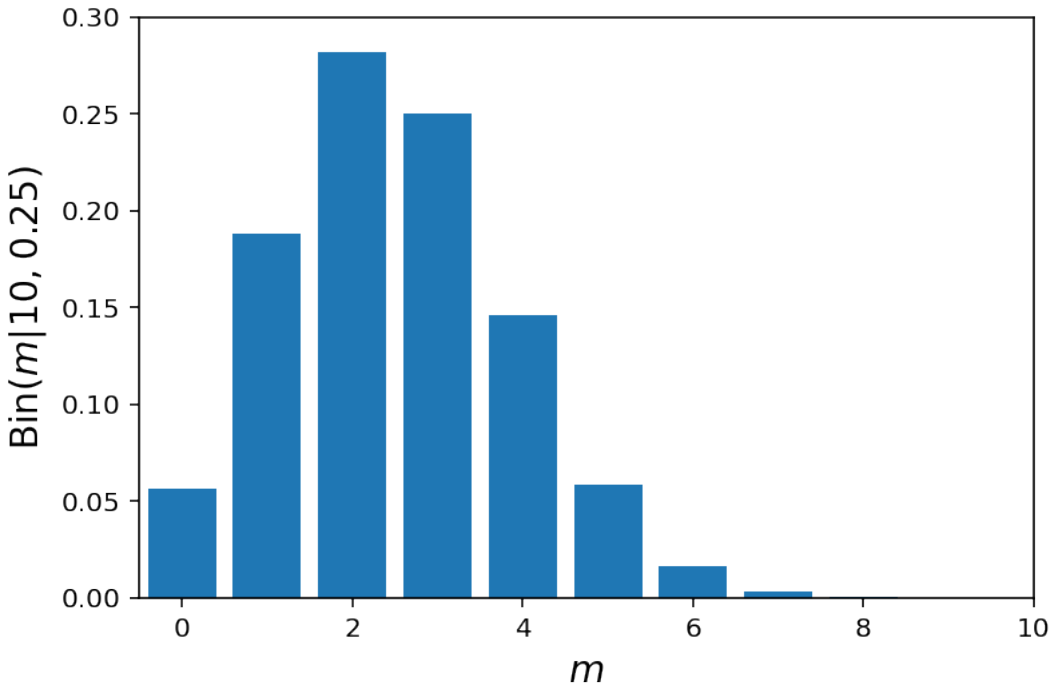
$$\mathrm{Bin}(m|N, \mu) = \begin{pmatrix} N \\ m \end{pmatrix} \mu^m (1 - \mu)^{N-m}$$

where

$$\binom{N}{m} = \frac{N!}{(N-m)!m!}$$

is the numbers of ways of choosing m objects out of a total of N identical objects, and is called the binomial coefficient.

[6]:
```
model = Binomial(10, 0.25)
y = model.pdf(np.arange(0, 10))

plt.bar(np.arange(0, 10), y)
plt.xlim([-0.5, 10]); plt.ylim([0, 0.3])
plt.xlabel("$m$", fontsize=14); plt.ylabel("$\mathrm{Bin}(m|10,0.25)$",␣
↪fontsize=14)
plt.show()
```



2.1.1 The beta distribution

We have seen that the maximum likelihood estimator for the parameter μ in the Bernoulli and binomial distributions, is given by the fraction of the observations in the data having $x = 1$, which can lead to over-fitted results for small data sets. In order to arrive to more sensible results, we develop a **Bayesian treatment** for the problem by introducing a prior distribution $p(\mu)$ over the parameter μ.

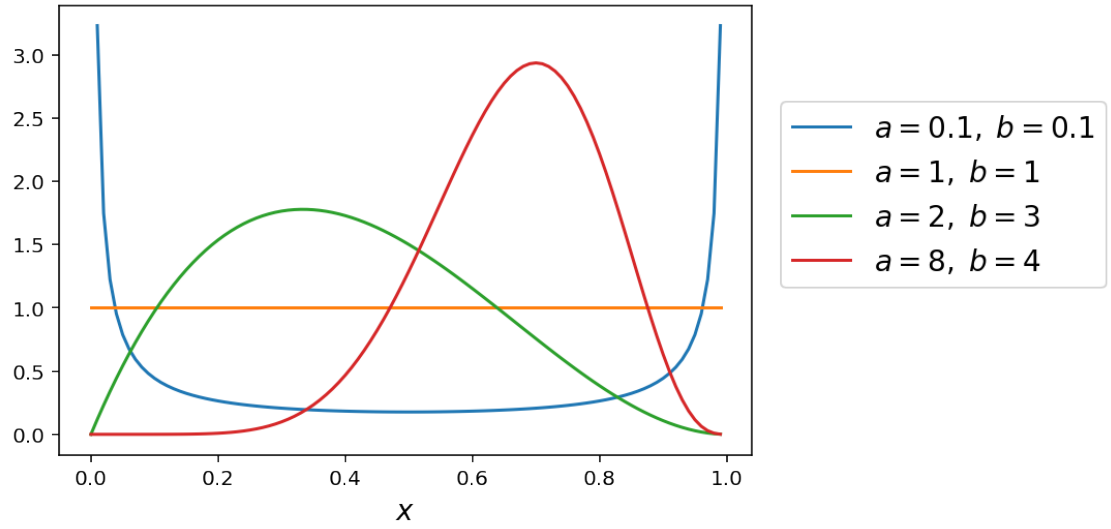
Note that the likelihood function takes the form of the product of factors of the form $\mu^x(1-\mu)^{(1-x)}$. We would like to choose a prior proportional to powers of μ and $(1-\mu)$ in order for the posterior to have the same functional form as the prior, since the posterior is proportional to the product of the prior and the likelihood. This important property is called **conjugacy**.

We therefore we choose a prior, called the **beta** distribution, given by

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

where a and b are often called hyperparameters because they control the distribution of the parameter μ. In order to give an intuition about the hyperparameteres, we plot the beta distribution for various values of a and b.

[7]:
```
x_space = np.arange(0, 1, 0.01)

for a, b in [(0.1, 0.1), (1, 1), (2, 3), (8, 4)]:
    model = Beta(a, b)
    plt.plot(x_space, model.pdf(x_space), label="$a={},\; b={}$".format(a, b))

plt.xlabel("$x$", fontsize=14)
plt.legend(bbox_to_anchor=(1, 0.85), loc=2, borderaxespad=1, fontsize=14); plt.
↪show()
```



The fraction of $\Gamma(x)$ functions ensures that the beta distribution is normalized, so that

$$\int_0^1 \text{Beta}(\mu|a,b)d\mu = 1$$

The mean is given by

$$
\begin{aligned}
\mathbb{E}[\mu] &= \int_0^1 \mu \text{Beta}(\mu|a,b)d\mu \\
&= \int_0^1 \mu \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{a-1}(1-\mu)^{b-1}d\mu \\
&= \int_0^1 \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{a}(1-\mu)^{b-1}d\mu \\
&= \int_0^1 \frac{a\Gamma(a+b+1)}{(a+b)\Gamma(a+1)\Gamma(b)}\mu^{a}(1-\mu)^{b-1}d\mu \\
&= \frac{a}{a+b}\int_0^1 \frac{\Gamma(a+b+1)}{\Gamma(a+1)\Gamma(b)}\mu^{a}(1-\mu)^{b-1}d\mu \\
&= \frac{a}{a+b}\int_0^1 \text{Beta}(\mu|a+1,b)d\mu \\
&= \frac{a}{a+b}
\end{aligned}
$$

where we have taken advantage of the property $\Gamma(x+1) = x\Gamma(x)$. The variance is given by

$$
\text{var}[\mu] = \frac{ab}{(a+b)^2(a+b+1)}
$$

The posterior distribution over μ can be obtained by multiplying the beta prior by the binomial likelihood function and **normalizing**,

$$
p(\mu|m,l,a,b) = \frac{\Gamma(m+a+l+b)}{\Gamma(m+a)\Gamma(l+b)}\mu^{m+a-1}(1-\mu)^{l+b-1}
$$

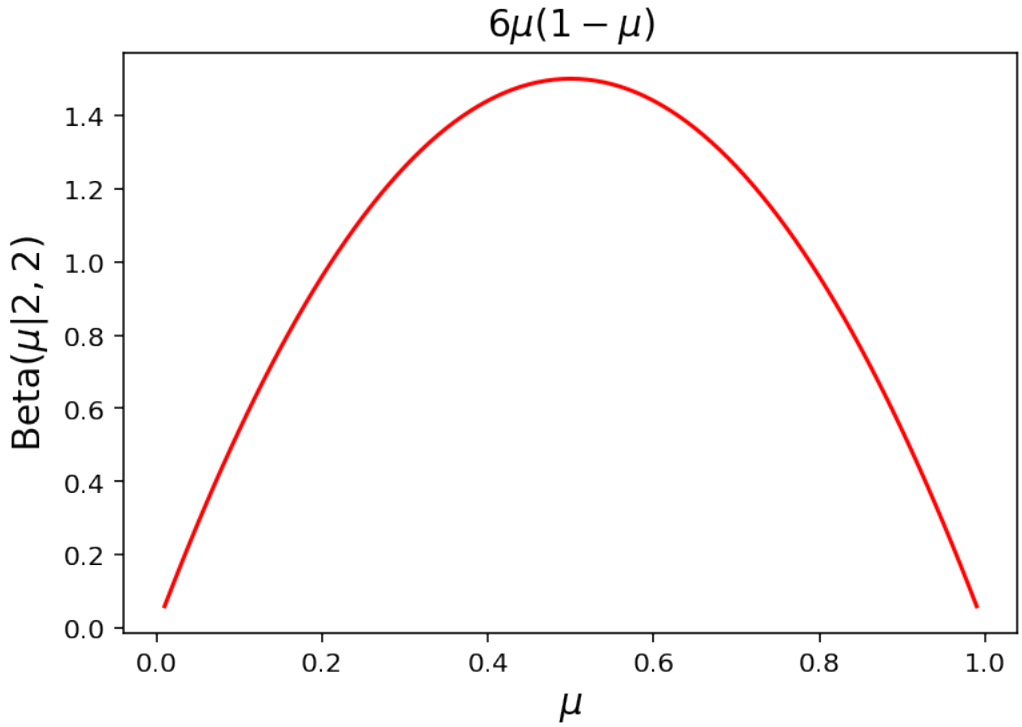
where $l = N - m$, and corresponds to the number of times $x = 0$.

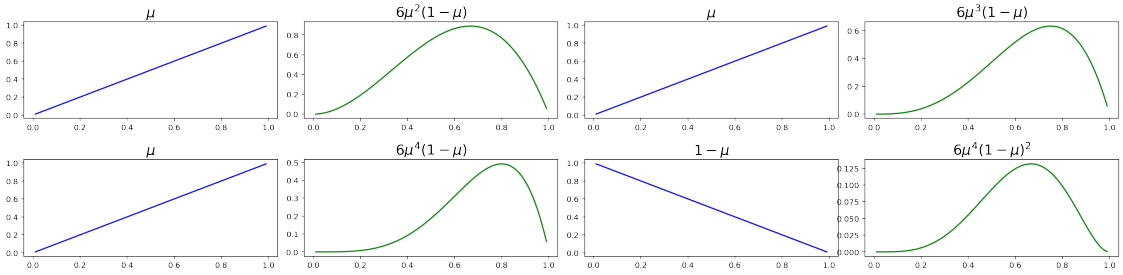
Note that the effect of observing a data set of m observations of $x = 1$ and l observations of $x = 0$ is to increase the value of a by m, and the value of b by l, from the prior to the posterior distribution. Thus, the hyperparameters a and b in the prior represent the number of observations of $x = 1$ and $x = 0$, respectively. Furthermore, the posterior distribution can act as the prior if subsequent observations arrive. Imagine taking one observation at a time and after each observation updating the current posterior distribution by multiplying by the likelihood of the incoming observation. At each stage, the posterior is a beta distribution incorporating some number of (prior and actual) observed values for $x = 1$ and $x = 0$ given by the parameters a and b. Incorporation an additional observation of $x = 1$ corresponds to incrementing the value of a by 1, whereas for $x = 0$ increment b by 1.

We present a sequence of such Bayesian inference steps, where three observations of $x = 1$ arrive before a single observation of $x = 0$. Note how the prior revises the form of the **unnormalized** posterior on each update step according to the likelihood. Apart from the first step, where the prior is a Beta distribution having paramater $a = 2$ and $b = 2$, in each subsequent step the prior is the **unnormalized** posterior of the previous step.

[8]:
```
# The Beta conjugate prior for the Binomial distribution (starting parameter␣
 ↪a=2, b=2)
prior = Beta(a=2, b=2)
plt.plot(mu_space, prior.pdf(mu_space), color='red')
plt.xlabel('$\mu$', fontsize=14); plt.ylabel('$\mathrm{Beta}(\mu|2,2)$',␣
 ↪fontsize=14)
plt.title(prior.change_notation({'x': 'mu'}).to_latex, fontsize=14)
plt.show()

# Apply a sequence of Bayesian inference steps, as more observations arrive in␣
 ↪the form of a likelihood
plt.figure(figsize=(20, 5))

# Row 1
plt.subplot(2,4,1)
plt.tight_layout()
likelihood = Binomial(n=1).pdf(1)
plt.plot(mu_space, likelihood.pdf(mu=mu_space), color='blue')
plt.title(likelihood.to_latex, fontsize=18)

plt.subplot(2,4,2)
plt.tight_layout()
posterior = prior.change_notation({'x': 'mu'}) * likelihood
plt.plot(mu_space, posterior.pdf(mu=mu_space), 'g')
plt.title(posterior.to_latex, fontsize=18)

plt.subplot(2,4,3)
plt.tight_layout()
likelihood = Binomial(n=1).pdf(1)
plt.plot(mu_space, likelihood.pdf(mu=mu_space), color='blue')
plt.title(likelihood.to_latex, fontsize=18)

plt.subplot(2,4,4)
plt.tight_layout()
posterior = posterior * likelihood
plt.plot(mu_space, posterior.pdf(mu=mu_space), 'g')
plt.title(posterior.to_latex, fontsize=18)

# Row 2
plt.subplot(2,4,5)
plt.tight_layout()
likelihood = Binomial(n=1).pdf(1)
plt.plot(mu_space, likelihood.pdf(mu=mu_space), color='blue')
plt.title(likelihood.to_latex, fontsize=18)

plt.subplot(2,4,6)
plt.tight_layout()
```

```
posterior = posterior * likelihood
plt.plot(mu_space, posterior.pdf(mu=mu_space), 'g')
plt.title(posterior.to_latex, fontsize=18)

plt.subplot(2,4,7)
plt.tight_layout()
likelihood = Binomial(n=1).pdf(0)
plt.plot(mu_space, likelihood.pdf(mu=mu_space), color='blue')
plt.title(likelihood.to_latex, fontsize=18)

plt.subplot(2,4,8)
plt.tight_layout()
posterior = posterior * likelihood
plt.plot(mu_space, posterior.pdf(mu=mu_space), 'g')
plt.title(posterior.to_latex, fontsize=18)

plt.show()
```





Thus, the **sequential** approach to learning arises naturally when we adopt a Bayesian viewpoint. It is **independent** of the choice of prior and of the likelihood function and **depends only** on the assumption of i.i.d. data. Sequential methods make use of observations one at a time, or in small batches, and then discard them before the next observations are used. They can be used, for example, in real-time learning scenarios where a steady stream of data is arriving, and predictions must be made before all of the data is seen. Maximum likelihood methods can also be cast into a sequential framework.

If our goal is to predict the outcome fo the next trial, then we must evaluate the predictive distribution of x given the observed data $\mathcal{D}$ given by,

$$p(x = 1|\mathcal{D}) = \frac{m + a}{m + a + l + b}$$

which essentially represents the fraction of observations (actual and prior) that corresponds to $x = 1$. Note that in the limit of an infinitely large data set $m, l \to \infty$, $p(x = 1|\mathcal{D})$ reduces to the maximum likelihood estimator $\frac{m}{N}$.

It is a very **general property** that the Bayesian and maximum likelihood results are identical in the limit of infinitely large data sets. For finite data set, the posterior mean for μ always lies between the prior mean and the maximum likelihood estimate for μ, corresponding to the sample mean.

2.2 Multinomial Variables

A binary variable is good for representing a coin toss, but generally we would like to have more states, such as the number rolled on a die. Such discrete random variables can take on one of K possible mutually exclusive states of the form

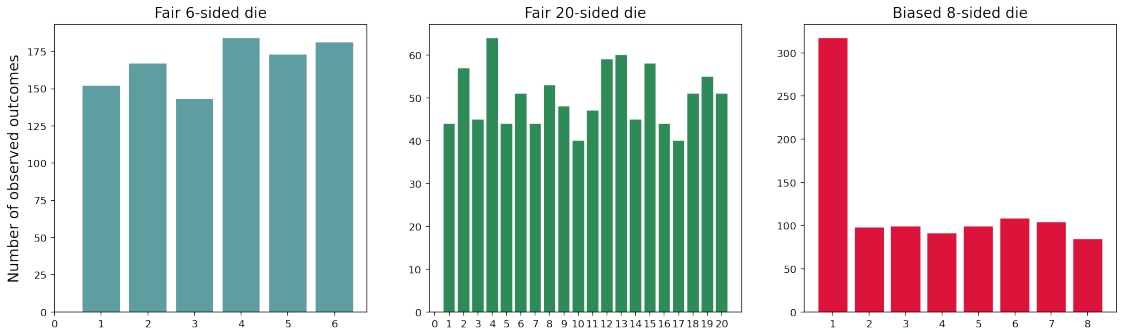
$$\mathbf{x} = (0, 0, 1, 0, 0, 0)^{\mathrm{T}}$$

where each state k is represented by the k^{th} element being 1 and all other elements being 0. If we denote the probability of $x_k = 1$ by the parameter μ_k, then the distribution over $\mathbf{x}$ can be regarded as a generalization of the Bernoulli distribution, called the *categorical distribution*, and is given by

$$p(\mathbf{x}|\mu) = \prod_{k=1}^{K} \mu_k^{x_k}$$

where the parameters $\mathbf{x} = (\mu_1, \ldots \mu_K)^{\mathrm{T}}$ must satisfy $\mu_k \geq 0$ and $\sum_k \mu_k = 1$ because they represent probabilities. Note that $\mu_k^{x_k}$ becomes 1 for every element except the $x_k = 1$, and so the product picks the probability μ_k for the state represented by $\mathbf{x}$.

[9]:
```
plt.figure(figsize=(18, 5))

# Lets roll 1000 fair 6-sided dice
plt.subplot(1,3,1)
fair_rolls_6 = Categorical(mu=np.ones(6) / 6).draw(1000)
plt.bar(np.arange(1,7), sum(fair_rolls_6), color="cadetblue")
plt.ylabel("Number of observed outcomes", fontsize=14)
plt.xticks(np.arange(6+1))
plt.title("Fair 6-sided die", fontsize=14)

# then 1000 fair 20-sided dice
plt.subplot(1,3,2)
fair_rolls_20 = Categorical(mu=np.ones(20) / 20).draw(1000)
plt.bar(np.arange(1,21), sum(fair_rolls_20), color="seagreen")
plt.xticks(np.arange(20+1))
plt.title("Fair 20-sided die", fontsize=14)

# finally 1000 biased 8-sided dice
plt.subplot(1,3,3)
biased_rolls_8 = Categorical(mu=np.array([3, 1, 1, 1, 1, 1, 1, 1], float) / 10).
 ↪draw(1000)
plt.bar(np.arange(1,9), sum(biased_rolls_8), color="crimson")
plt.title("Biased 8-sided die", fontsize=14)

plt.show()
```



It is easily seen that the distribution is normalized

$$\begin{aligned}\sum_{\mathbf{x}} p(\mathbf{x}|\mu) &= \sum_{\mathbf{x}} \prod_{k=1}^{K} \mu_k^{x_k} \\ &= \prod_{k=1}^{K} \mu_k^{x_k^1} + \cdots + \prod_{k=1}^{K} \mu_k^{x_k^K} \\ &= \sum_{k=1}^{K} \mu_k = 1\end{aligned}$$

and that

$$\begin{aligned}\mathbb{E}[\mathbf{x}] &= \mathbb{E}[\mathbf{x}|\mu] \\ &= \sum_{\mathbf{x}} \mathbf{x} p(\mathbf{x}|\mu) \\ &= \sum_{\mathbf{x}} \mathbf{x} \prod_{k=1}^{K} \mu_k^{x_k} \\ &= \mathbf{x}^1 \prod_{k=1}^{K} \mu_k^{x_k^1} + \cdots + \mathbf{x}^K \prod_{k=1}^{K} \mu_k^{x_k^K} \\ &= (\mu_1, \ldots, \mu_K)^{\mathrm{T}}\end{aligned}$$

Consider a data set $\mathcal{D}$ of N independent observations $\mathbf{x}_1, \ldots, \mathbf{x}_N$, the corresponding likelihood function is given by

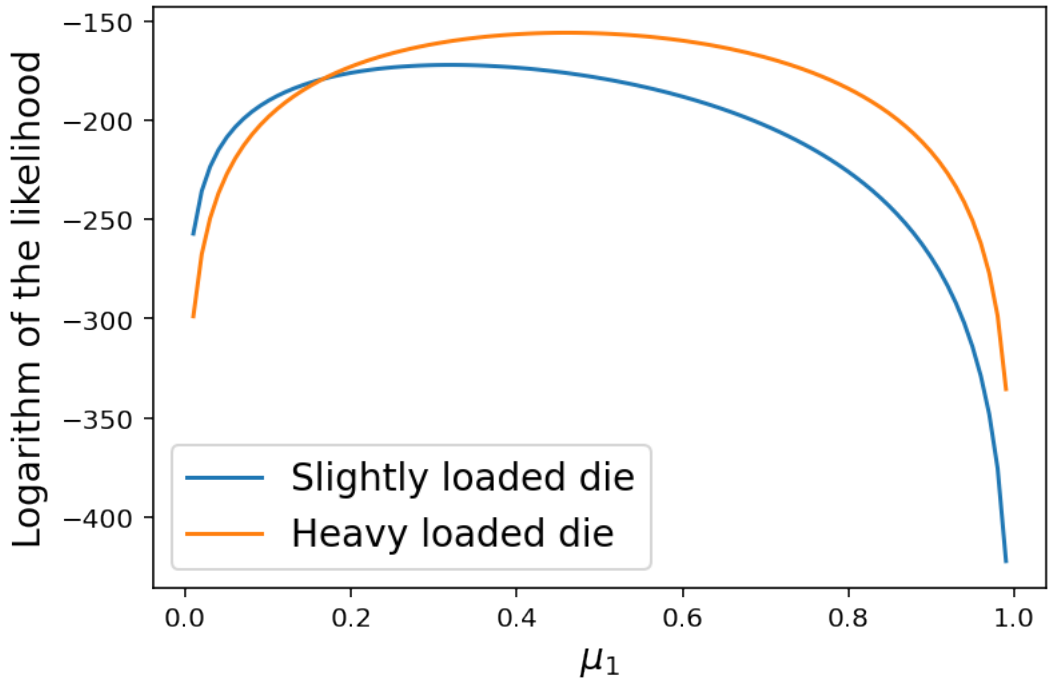
$$\begin{aligned}p(\mathcal{D}|\mu) &= \prod_{n=1}^{N} \prod_{k=1}^{K} \mu_k^{x_{nk}} \\ &= \prod_{k=1}^{K} \mu_k^{(\sum_{n=1}^{N} x_{nk})} \\ &= \prod_{k=1}^{K} \mu_k^{m_k}\end{aligned}$$

Therefore, given some observed dice rolls, we can plot, for instance, how likely it is that the number one in the die that produced these observations is biased in a certain way

[10]:
```
N = 100

def likelihood(mu_1, data):
    mus = (np.ones(6) * (1 - mu_1)) / 5
    mus[0] = mu_1
    return Categorical(mu=mus).log_likelihood_iid(data)

mu_space = np.linspace(0, 1, 100)

nums = np.array([2,1,1,1,1,1], float)
print(nums / sum(nums))
```

```
slightly_loaded_data = Categorical(mu=nums / sum(nums)).draw(N)
plt.plot(mu_space, [likelihood(mu_1, slightly_loaded_data) for mu_1 in␣
 ↪mu_space], label="Slightly loaded die")

nums = np.array([5,1,1,1,1,1], float)
print(nums / sum(nums))
heavy_loaded_data = Categorical(mu=nums / sum(nums)).draw(N)
plt.plot(mu_space, [likelihood(mu_1, heavy_loaded_data) for mu_1 in mu_space],␣
 ↪label="Heavy loaded die")

plt.xlabel("$\mu_1$", fontsize=14); plt.ylabel("Logarithm of the likelihood",␣
 ↪fontsize=14)
plt.legend(fontsize=14); plt.show()
```

```
[0.28571429 0.14285714 0.14285714 0.14285714 0.14285714 0.14285714]
[0.5 0.1 0.1 0.1 0.1 0.1]
```



In order to maximize $\ln p(\mathcal{D}|\mu)$, taking account of the constraint $c(\mu) = \sum_k \mu_k - 1$, we use a Lagrange multiplier λ and maximizing

$$L(\mu, \lambda) = \ln p(\mathcal{D}|\mu) + \lambda c(\mu)$$

By setting the gradient to zero, we obtain the maximum likelihood solution in the form

$$\mu_k = \frac{m_k}{N}$$

which is the fraction of the N observations for which $x_k = 1$. Thus, the maximum likelihood solution, similar to the Bernoulli distribution, is the *sample mean* on each dimension.

[11]:
```
print("Maximum likelihood of   for the slightly loaded die is", np.
 ↪mean(slightly_loaded_data, axis=0))
print("Maximum likelihood of   for the heavy loaded die is", np.
 ↪mean(heavy_loaded_data, axis=0))
```

```
Maximum likelihood of   for the slightly loaded die is [0.32 0.13 0.14 0.12 0.13
0.16]
Maximum likelihood of   for the heavy loaded die is [0.46 0.11 0.11 0.12 0.09
0.11]
```

The joint distribution of the quantities $m_1, \ldots, m_K$, conditioned on the parameters μ and the total number N of observations, is known as the *multinomial* distribution. It would give us the number of $1, 2, 3, \ldots$ rolled in a sample dataset of size N, the same way the binomial distribution gave us the number of heads. The multinomial distribution is given by

$$\text{Mult}(m_1, \ldots, m_K | \mu, N) = \begin{pmatrix} N \\ m_1 m_2 \ldots m_K \end{pmatrix} \prod_{k=1}^{K} \mu_k^{m_k}$$

where

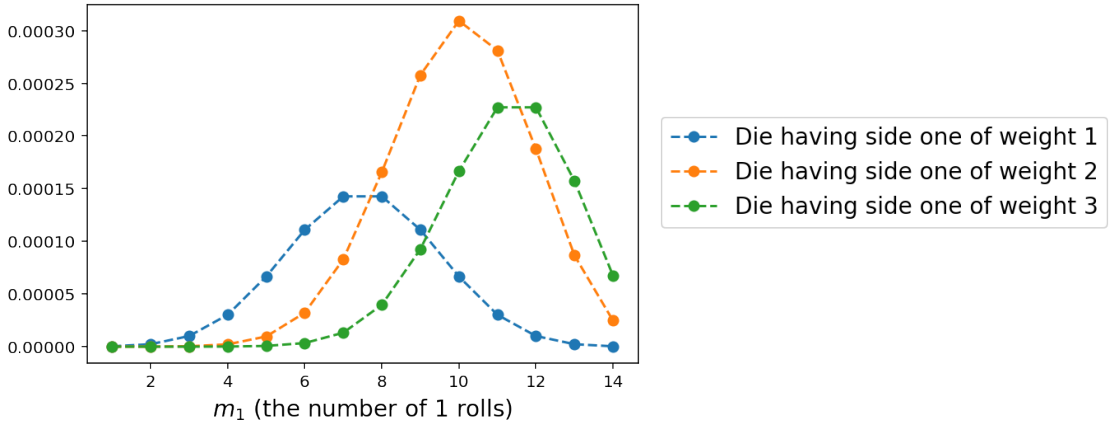
$$\begin{pmatrix} N \\ m_1 m_2 \ldots m_K \end{pmatrix} = \frac{N!}{m_1! m_2! \ldots m_K!}$$

is the number of ways to partition N objects into K groups of size $m_1, \ldots, m_K$, and is called the multinomial coefficient.

We cannot really plot the multinomial distribution directly, because of the many dimensions of $\mathbf{m}$. However, we can show the distribution of the number of ones rolled (constraining the other numbers), using various weighted dice.

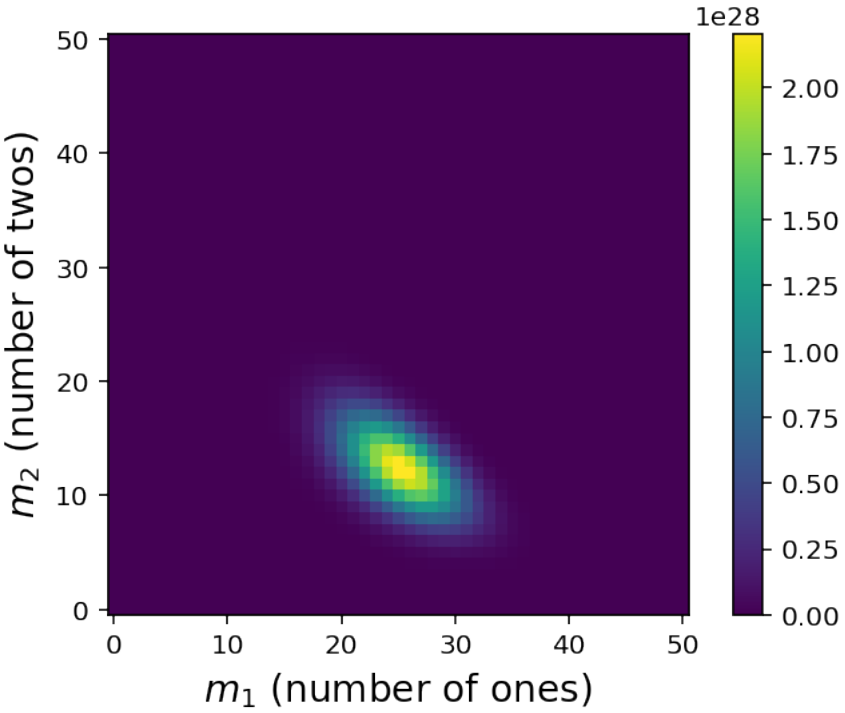
[12]:
```
# Consider N tosses of a 6-sided die.
N = 35
m_space = np.arange(1, 15)

# The weight parameter determines how biased is side 1 of the die
for w in [1, 2, 3]:
    nums = np.array([w, 1, 1, 1, 1, 1])
    distribution = Multinomial(N, mu=nums / sum(nums))
    plt.plot(m_space, [distribution.pdf(np.array([m1, 5, 5, 5, 5, 15 - m1]).T)␣
 ↪for m1 in m_space],
             'o--', label='Die having side one of weight %s' % w)
```

```
plt.xlabel("$m_1$ (the number of $1$ rolls)" , fontsize=14)
plt.legend(bbox_to_anchor=(1, 0.75), loc=2, borderaxespad=1, fontsize=14); plt.
↪show()
```



And we can plot the distribution over a 3 sided die in using a colormap. Note that the plot only depicts two variables on the axes, since the third can be inferred as they sum to N.

[13]:
```
N = 50
m_space = np.arange(N + 1)
distribution = Multinomial(N, mu=np.array([2, 1, 1]).T)

o = np.ones((N + 1, N + 1))
for m1 in m_space:
    for m2 in np.arange((N - m1) + 1):
        m3 = N - m1 - m2
        o[m2, m1] = distribution.pdf(np.array([m1, m2, m3]).T)

plt.imshow(o, origin="lower")
plt.xlabel("$m_1$ (number of ones)", fontsize=14); plt.ylabel("$m_2$ (number of␣
↪twos)", fontsize=14)
plt.colorbar(); plt.show()
```



2.2.1 The Dirichlet distribution

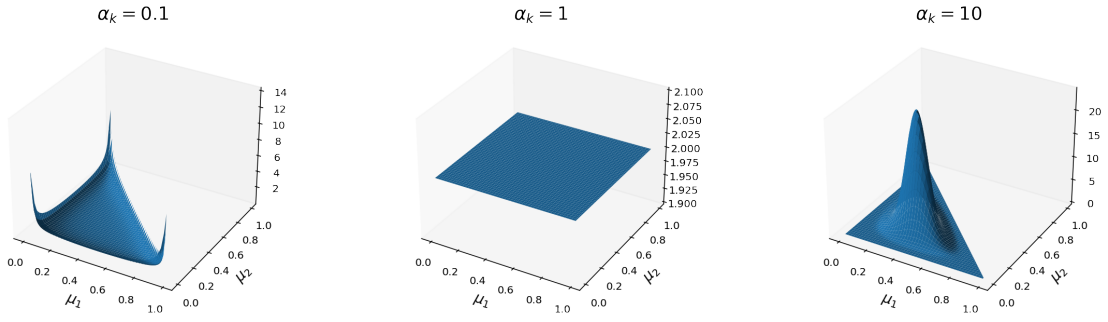
Similar to the Beta distribution, consider a family of prior distributions for the parameters μ of the multinomial distribution, called the *Dirichlet* distribution

$$\mathrm{Dir}(\mu|\alpha) = \frac{\Gamma(\alpha_0)}{\Gamma(\alpha_1)\cdots\Gamma(\alpha_K)} \prod_{k=1}^{K} \mu_k^{\alpha_k - 1}$$

In order to give an intuition we plot the Dirichlet distribution over three variables. Two of them are shown in horizontal axes are coordinates in the plane of the simplex, while the vertical axis corresponds to the value of the density.

[14]:
```
fig = plt.figure(figsize=(20, 5))

for i, alpha in enumerate([0.1, 1, 10]):

    distribution = Dirichlet(np.array([alpha, alpha, alpha]))
    X1, X2 = np.meshgrid(np.arange(0, 1, 0.01), np.arange(0, 1, 0.01))
    X3 = 1 - (X1 + X2)

    X = np.array([np.ravel(X1), np.ravel(X2), [float('nan') if x < 0 else x for␣
 ↪x in np.ravel(X3)]]).T
    Z = distribution.pdf(X).reshape(X1.shape)
```

```
    ax = fig.add_subplot(1, 3, i + 1, projection='3d'); ax.grid(visible=None)
    ax.plot_surface(X1, X2, Z)
    ax.set_xlabel("$\mu_1$", fontsize=14); ax.set_ylabel("$\mu_2$", fontsize=14)
    ax.set_title("$\\alpha_k={}$".format(str(alpha)), fontsize=18)

plt.show()
```



Multiplying the Dirichlet prior by the multinomial likelihood function and **normalizing**, we obtain the posterior distribution for the parameters μ in the form

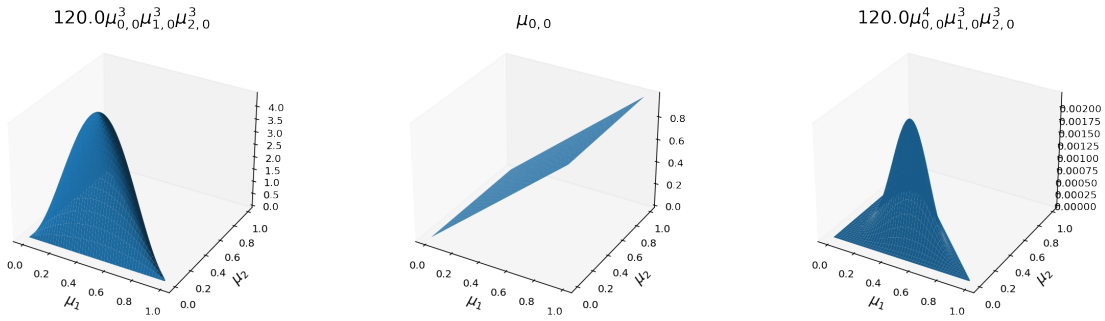
$$p(\mu|\mathcal{D},\alpha) = \mathrm{Dir}(\mu|\alpha+\mathbf{m}) = \frac{\Gamma(\alpha_0+N)}{\Gamma(\alpha_1+m_1)\cdots\Gamma(\alpha_K+m_K)}\prod_{k=1}^{K}\mu_k^{\alpha_k+m_k-1}$$

Note that the posterior distribution again takes the form of a Dirichlet distribution, confirming that the Dirichlet is indeed a **conjugate prior** for the multinomial distribution. As in the case of the binomial distribution and the beta prior, the parameters α_k of the Dirichlet prior can be interpreted as an *effective number of observations* of $x_k = 1$. Thus, the effect of observing a data set of $\mathbf{m}$ observations for the K states of $\mathbf{x}$ is to increase the values of α by $\mathbf{m}$ from the prior to the posterior distribution.

To that end, we present a Bayesian inference step, where an observations of $\mathbf{x} = (1,0,0)^{\mathrm{T}}$ arrives. Given a Dirichlet prior having parameters $\alpha = (2,2,2)^{\mathrm{T}}$, note how the prior revises the form of the **unnormalized** posterior according to the likelihood.

[15]:
```
# Apply a sequence of Bayesian inference steps, as more observations arrive in␣
 ↪the form of a likelihood
fig = plt.figure(figsize=(20, 5))

# The Dirichlet conjugate prior for the multinomial distribution (starting␣
 ↪parameters a=[2, 2, 2])
prior = Dirichlet(alpha=np.array([2, 2, 2]).T)
Z = prior.pdf(X).reshape(X1.shape)
ax = fig.add_subplot(1, 3, 1, projection='3d'); ax.grid(visible=None)
ax.plot_surface(X1, X2, Z)
```

```
ax.set_xlabel("$\mu_1$", fontsize=14)
ax.set_ylabel("$\mu_2$", fontsize=14)
ax.set_title(prior.change_notation({'x': 'mu'}).to_latex, fontsize=18)

# The multinomial likelihood
ax = fig.add_subplot(1, 3, 2, projection='3d'); ax.grid(visible=None)
likelihood = Multinomial(n=1, dim=3).pdf(np.array([1, 0, 0]).T)
Z = likelihood.pdf(mu=X).reshape(X1.shape)
ax.plot_surface(X1, X2, Z)
ax.set_xlabel("$\mu_1$", fontsize=14)
ax.set_ylabel("$\mu_2$", fontsize=14)
ax.set_title(likelihood.to_latex, fontsize=18)

# The resulting posterior
ax = fig.add_subplot(1, 3, 3, projection='3d'); ax.grid(visible=None)
posterior = prior.change_notation({'x': 'mu'}) * likelihood
Z = posterior.pdf(mu=X).reshape(X1.shape)
ax.plot_surface(X1, X2, Z)
ax.set_xlabel("$\mu_1$", fontsize=14)
ax.set_ylabel("$\mu_2$", fontsize=14)
ax.set_title(posterior.to_latex, fontsize=18)

plt.show()
```



2.3 Gaussian Distribution

The *Gaussian*, also known as the normal distribution, is a model for the distribution of continuous random variables. In the case of single variable x, the Gaussian distribution is given by

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

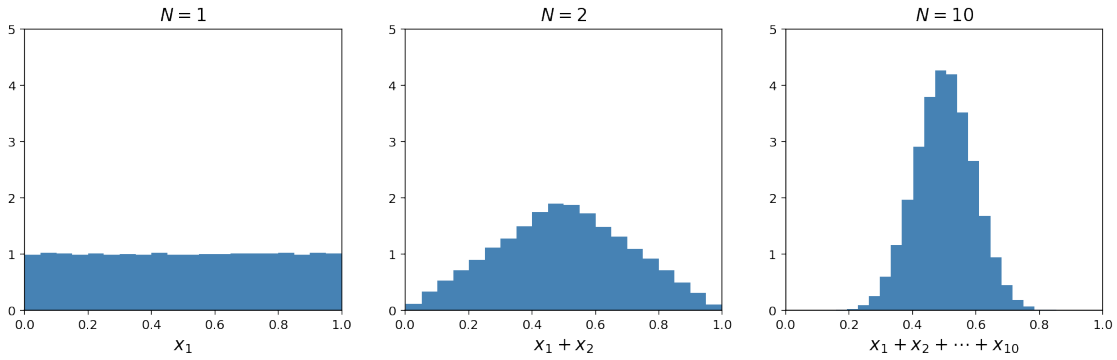
For a D-dimensional vector $\mathbf{x}$, the multivariate Gaussian distribution takes the form

$$\mathcal{N}(\mathbf{x}|\mu, \ ) = \frac{1}{(2\pi)^{D/2}| \ |^{1/2}} \exp\Big\{ -\frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}} \ ^{-1}(\mathbf{x}-\mu)\Big\}$$

where μ is a D-dimensional mean vector, is a $D \times D$ covariance matrix, while $|\ |$ denotes the determinant of .

The Gaussian distribution can be motivated from a variety of perspectives. For instance, we have seen that for continuous variables, the distribution **maximizing the entropy** is a Gaussian distribution. Moreover, the *central limit theorem* tell us that the sum of a set of random variables, which is of course itself a random variable, has a distribution that becomes increasingly Gaussian as the number of terms increases.

[16]:
```
plt.figure(figsize=(15, 4))

plt.subplot(1, 3, 1)
sample = np.mean(np.random.uniform(0, 1, size=(1, 100000)), axis=0)
plt.hist(sample, bins=20, density=True, color="steelblue")
plt.xlabel("$x_1$", fontsize=14); plt.xlim(0, 1); plt.ylim(0, 5)
plt.title("$N=1$", fontsize=14)

plt.subplot(1, 3, 2)
sample = np.mean(np.random.uniform(0, 1, size=(2, 100000)), axis=0)
plt.hist(sample, bins=20, density=True, color="steelblue")
plt.xlabel("$x_1+x_2$", fontsize=14); plt.xlim(0, 1); plt.ylim(0, 5)
plt.title("$N=2$", fontsize=14)

plt.subplot(1, 3, 3)
sample = np.mean(np.random.uniform(0, 1, size=(10, 100000)), axis=0)
plt.hist(sample, bins=20, density=True, color="steelblue")
plt.xlabel("$x_1+x_2+\cdots+x_{10}$", fontsize=14); plt.xlim(0, 1); plt.ylim(0,␣
↪5)
plt.title("$N=10$", fontsize=14)

plt.show()
```

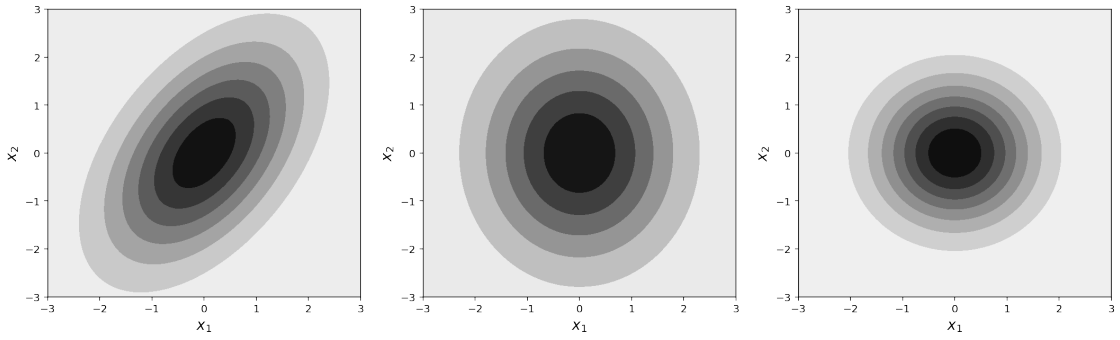


Although the Gaussian distribution is widely used as a density model, it suffers from significant limitations:

1. Consider the number of free parameters in the distribution. A general symmetric covariance matrix has $D(D+1)/2$ independent parameters, and there are another D independent parameters in μ, resulting in $D(D+3)/2$ parameters in total. Since the total number of parameters grows quadratically with D, manipulation and inversion of large matrices becomes prohibitive.
2. One way of addressing these problems is by considering restricted forms of the covariance matrix, such as *diagonal*, $= diag(\sigma_i^2)$, or even *isotropic*, $= \sigma^2\mathbf{I}$, covariance matrices. Unfortunately, whereas such approaches limit the number of degrees of freedom in the distribution, making the inversion of the covariance matrix faster, they also greatly restrict the form of the probability density and its ability to capture interesting correlations in the data.
3. The Gaussian distribution is intrinsically unimodal (i.e., has a single maximum), and so is unable to approximate multimodal distributions.

[17]:
```
# Generate 100 points in the interval [-5, 5]
N = 100
X1, X2 = np.meshgrid(np.linspace(-5, 5, N), np.linspace(-5, 5, N))
X = np.array([np.ravel(X1), np.ravel(X2)])

plt.figure(figsize=(18, 5))

# General covariance matrix
generic_sigma = np.array(
    [[1.5, 0.92],
     [0.92, 2.2]]
)
N_distribution = MultivariateGaussian(np.zeros((2, 1)), generic_sigma)
p = np.diag(N_distribution.pdf(X)).reshape(X1.shape)

plt.subplot(1, 3, 1)
plt.contourf(X1, X2, p, cmap='binary')
plt.xlabel('$x_1$', fontsize=14); plt.ylabel('$x_2$', fontsize=14)
plt.axis([-3, 3, -3, 3])

# Diagonal covariance matrix
diagonal_sigma = np.array(
    [[1.5, 0],
     [0, 2.2]]
)
N_distribution = MultivariateGaussian(np.zeros((2, 1)), diagonal_sigma)
p = np.diag(N_distribution.pdf(X)).reshape(X1.shape)

plt.subplot(1, 3, 2)
plt.contourf(X1, X2, p, cmap='binary')
plt.xlabel('$x_1$', fontsize=14); plt.ylabel('$x_2$', fontsize=14)
```

```
plt.axis([-3, 3, -3, 3])

# Isotropic covariance matrix
isotropic_sigma = np.array(
    [[1.0, 0],
     [0, 1.0]]
)
N_distribution = MultivariateGaussian(np.zeros((2, 1)), isotropic_sigma)
p = np.diag(N_distribution.pdf(X)).reshape(X1.shape)

plt.subplot(1, 3, 3)
plt.contourf(X1, X2, p, cmap='binary')
plt.xlabel('$x_1$', fontsize=14); plt.ylabel('$x_2$', fontsize=14)
plt.axis([-3, 3, -3, 3])

plt.show()
```



2.3.4 Maximum likelihood for the Gaussian

Given a data set $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_N)^{\mathrm{T}}$, in which the observations $\mathbf{x}_n$ are assumed to be drawn indepdendently from a multivariate Gaussian distribution, we can estimate the parameters of the distribution by maximum likelihood. The logarithm of the likelihood function is given by

$$
\begin{aligned}
\ln p(\mathbf{X}|\mu,\ ) &= \ln\Big(\prod_{n=1}^{N}\mathcal{N}(\mathbf{x}_n|\mu,\ )\Big)\\
&= \ln\Big(\prod_{n=1}^{N}\frac{1}{(2\pi)^{D/2}|\ |^{1/2}}\exp\Big\{-\frac{1}{2}(\mathbf{x}_n-\mu)^{\mathrm{T}}\ ^{-1}(\mathbf{x}_n-\mu)\Big\}\Big)\\
&= \sum_{n=1}^{N}\ln\Big(\frac{1}{(2\pi)^{D/2}|\ |^{1/2}}\exp\Big\{-\frac{1}{2}(\mathbf{x}_n-\mu)^{\mathrm{T}}\ ^{-1}(\mathbf{x}_n-\mu)\Big\}\Big)\\
&= \sum_{n=1}^{N}\ln\Big(\frac{1}{(2\pi)^{D/2}|\ |^{1/2}}\Big)+\sum_{n=1}^{N}\ln\exp\Big\{-\frac{1}{2}(\mathbf{x}_n-\mu)^{\mathrm{T}}\ ^{-1}(\mathbf{x}_n-\mu)\Big\}\\
&= N\ln\Big(\frac{1}{(2\pi)^{D/2}|\ |^{1/2}}\Big)-\frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n-\mu)^{\mathrm{T}}\ ^{-1}(\mathbf{x}_n-\mu)\\
&= -N\ln\left((2\pi)^{D/2}|\ |^{1/2}\right)-\frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n-\mu)^{\mathrm{T}}\ ^{-1}(\mathbf{x}_n-\mu)\\
&= -\frac{ND}{2}\ln(2\pi)-\frac{N}{2}\ln|\ |-\frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n-\mu)^{\mathrm{T}}\ ^{-1}(\mathbf{x}_n-\mu)
\end{aligned}
$$

Taking the derivative of the log-likelihood over μ and set it equal to 0, we obtain the solution for the maximum likelihood estimate of the mean, given by

$$
\begin{aligned}
\frac{\partial}{\partial\mu}\ln p(\mathbf{X}|\mu,\ ) = 0 &\Leftrightarrow \frac{\partial}{\partial\mu}\Big(-\frac{ND}{2}\ln(2\pi)-\frac{N}{2}\ln|\ |-\frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n-\mu)^{\mathrm{T}}\ ^{-1}(\mathbf{x}_n-\mu)\Big)=0\\
&\Leftrightarrow \frac{\partial}{\partial\mu}\Big(-\frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n-\mu)^{\mathrm{T}}\ ^{-1}(\mathbf{x}_n-\mu)\Big)=0\\
&\Leftrightarrow -\frac{1}{2}\sum_{n=1}^{N}\ ^{-1}(\mathbf{x}_n-\mu)=0\\
&\Leftrightarrow \sum_{n=1}^{N}\mathbf{x}_n-N\mu=0\\
&\Leftrightarrow \mu_{ML}=\frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n
\end{aligned}
$$

which is often called the *sample mean vector*. Calculating the derivative of the log-likelihood over and set it equal to 0 is given by

$$
\begin{aligned}
\frac{\partial}{\partial}\ln p(\mathbf{X}|\mu,\ ) = 0 &\Leftrightarrow \frac{\partial}{\partial}\Big(-\frac{ND}{2}\ln(2\pi)-\frac{N}{2}\ln|\ |-\frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n-\mu)^{\mathrm{T}}\ ^{-1}(\mathbf{x}_n-\mu)\Big)=0\\
&\Leftrightarrow \frac{\partial}{\partial}\Big(-\frac{N}{2}\ln|\ |\Big)-\frac{\partial}{\partial}\Big(\frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n-\mu)^{\mathrm{T}}\ ^{-1}(\mathbf{x}_n-\mu)\Big)=0
\end{aligned}
$$

By using (C.28) the first term can be reduced to

$$\frac{\partial}{\partial}\left(-\frac{N}{2}\ln|\,|\right) = -\frac{N}{2}(\ ^{-1})^{\mathrm{T}} = -\frac{N}{2}\ ^{-1}$$

where the last equality holds because is symmetric. For the second term, we can derive that,

$$\frac{\partial}{\partial}\left(\frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n-\mu)^{\mathrm{T}}\ ^{-1}(\mathbf{x}_n-\mu)\right) = \frac{N}{2}\ ^{-1}\mathbf{S}\ ^{-1}$$

where

$$\mathbf{S} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n-\mu)(\mathbf{x}_n-\mu)^{\mathrm{T}}$$

Thus, we obtain

$$\begin{aligned}
-\frac{N}{2}\ ^{-1} + \frac{N}{2}\ ^{-1}\mathbf{S}\ ^{-1} = 0 \Leftrightarrow \\
\frac{N}{2}\ ^{-1}\mathbf{S}\ ^{-1} = \frac{N}{2}\ ^{-1} \Leftrightarrow \\
= \mathbf{S} \Leftrightarrow \\
_{ML} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n-\mu_{ML})(\mathbf{x}_n-\mu_{ML})^{\mathrm{T}}
\end{aligned}$$

However, if we evaluate the expectations of the maximum likelihood solutions under the true distribution, we obtain the following results

$$\mathbb{E}[\mu_{ML}] = \frac{1}{N}\mathbb{E}\left[\sum_{n=1}^{N}\mathbf{x}_n\right] = \frac{1}{N}\sum_{n=1}^{N}\mathbb{E}[\mathbf{x}_n] = \frac{1}{N}N\mu = \mu$$

and

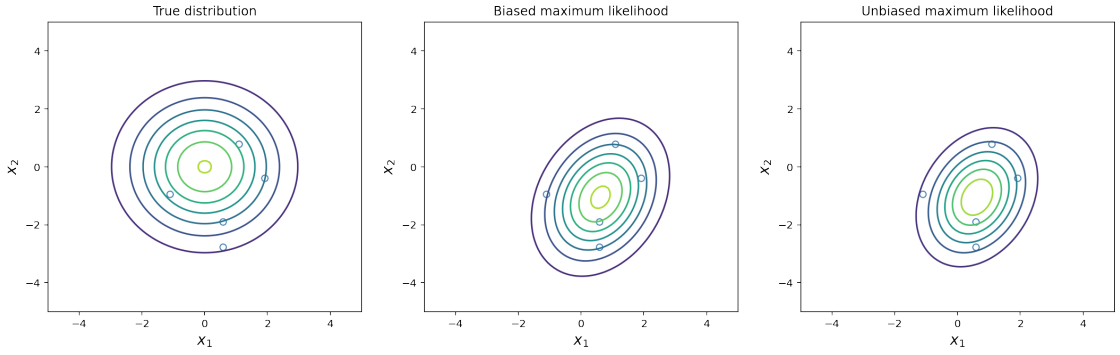
$$\mathbb{E}[\ _{ML}] = \frac{N-1}{N}$$

Thus, similar to the univariate Gaussian distribution, the expectation of the maximum likelihood estimate for the mean is equal to the true mean, while for the covariance has an expectation that is less than the true value, and hence it is biased. We can correct the bias by defining a different estimator given by

$$\tilde{\ } = \frac{1}{N-1}\sum_{n=1}^{N}(\mathbf{x}_n-\mu_{ML})(\mathbf{x}_n-\mu_{ML})^{\mathrm{T}}$$

[18]:
```
X = np.random.multivariate_normal(mean=np.array([0, 0]), cov=1.5 ** 2 * np.
↪eye(2), size=5)
```

```
x, y = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))

plt.figure(figsize=(18, 5))

gaussian = MultivariateGaussian(mu=np.zeros((2, 1)), cov=1.5 ** 2 * np.eye(2))
p = np.diag(gaussian.pdf(np.array([x, y]).reshape(2, -1))).reshape(100, 100)

plt.subplot(1, 3, 1)
plt.contour(x, y, p)
plt.scatter(X[:, 0], X[:, 1], facecolor="none", edgecolor="steelblue")
plt.xlabel("$x_1$", fontsize=14); plt.ylabel("$x_2$", fontsize=14)
plt.title('True distribution')

gaussian = MultivariateGaussian(dim=2)
gaussian.ml(X, unbiased=False)
p = np.diag(gaussian.pdf(np.array([x, y]).reshape(2, -1))).reshape(100, 100)

plt.subplot(1, 3, 2)
plt.contour(x, y, p)
plt.scatter(X[:, 0], X[:, 1], facecolor="none", edgecolor="steelblue")
plt.xlabel("$x_1$", fontsize=14); plt.ylabel("$x_2$", fontsize=14)
plt.title('Biased maximum likelihood')

gaussian1 = MultivariateGaussian(dim=2)
gaussian1.ml(X, unbiased=True)
p1 = np.diag(gaussian1.pdf(np.array([x, y]).reshape(2, -1))).reshape(100, 100)

plt.subplot(1, 3, 3)
plt.contour(x, y, p1)
plt.scatter(X[:, 0], X[:, 1], facecolor="none", edgecolor="steelblue")
plt.xlabel("$x_1$", fontsize=14); plt.ylabel("$x_2$", fontsize=14)
plt.title('Unbiased maximum likelihood')

plt.show()
```



2.3.5 Sequential estimation

Sequential or online methods allow data points to be processed one at a time and then discarded. They are important when large data sets are involved so that batch processing of all data at once is infeasible. Consider for instance, the maximum likelihood solution for the multivariate Gaussian distribution. Lets denote the solution based on N observations as $\mu_{ML}^{(N)}$. The, by dissecting the contribution from the final data point $\mathbf{x}_N$ we obtain

$$\begin{aligned}\mu_{ML}^{(N)} &= \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n \\ &= \frac{1}{N}\mathbf{x}_{\mathrm{N}} + \frac{1}{N}\sum_{n=1}^{N-1}\mathbf{x}_n \\ &= \frac{1}{N}\mathbf{x}_{\mathrm{N}} + \frac{N-1}{N}\mu_{ML}^{(N-1)} \\ &= \mu_{ML}^{(N-1)} + \frac{1}{N}(\mathbf{x}_N - \mu_{ML}^{(N-1)})\end{aligned}$$

Therefore, observing the data point $\mathbf{x}_N$ revises the estimate by moving the old estimate $\mu_{ML}^{(N-1)}$ a small amount proportional to $1/N$ in the direction of the *error signal* $(\mathbf{x}_N - \mu_{ML}^{(N-1)})$. Note that, as N increases, the constribution of successive data points gets smaller.

Unfortunately, there is not always possible to derive a sequential algorithm by this route. Therefore, we seek a more general formulation of sequential learning, which leads us to the **Robbins-Monro** algorithm. Consider a pair of random variables θ and z governed by a joint distribution $p(\theta, z)$. The conditional expectation of z given θ is given by

$$\mathbb{E}[z|\theta] = \int zp(z|\theta)dz$$

The Robbins-Monro procedure defines a sequence of succesive estimates of the root $\theta^\star$ given by

$$\theta^{(N)} = \theta^{(N-1)} - a_{N-1}z(\theta^{(N-1)})$$

where $z(\theta^{(N-1)})$ is the observed value of z when θ takes the value $\theta^{(N-1)}$.

In the case of the maximum likelihood solution we know that

$$\frac{\partial}{\partial\theta}\Big\{-\frac{1}{N}\sum_{n=1}^{N}\ln p(x_n|\theta)\Big\} = 0 \Leftrightarrow -\frac{1}{N}\sum_{n=1}^{N}\frac{\partial}{\partial\theta}\ln p(x_n|\theta) = 0$$

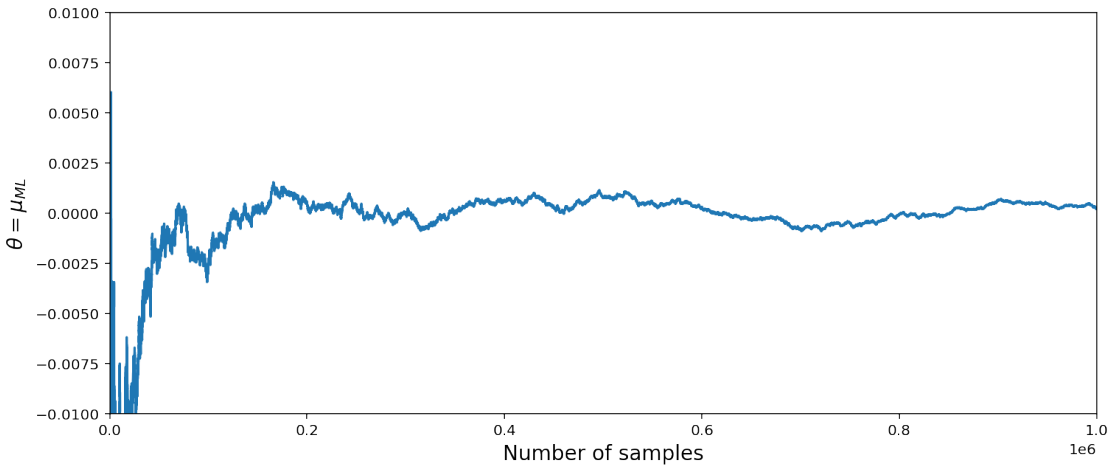
Taking the limit $N \to \infty$ we have

$$-\lim_{N\to\infty}\frac{1}{N}\sum_{n=1}^{N}\frac{\partial}{\partial\theta}\ln p(x_n|\theta) = \mathbb{E}\Big[-\frac{\partial}{\partial\theta}\ln p(x_n|\theta)\Big]$$

and so finding the maximum likelihood solution corresponds to finding the root of the regression function. Thus, Robbins-Monro procedure takes the form

$$\theta^{(N)} = \theta^{(N-1)} - a_{N-1} \frac{\partial}{\partial \theta^{(N-1)}} [-\ln p(x_n|\theta^{(N-1)})]$$

[19]:
```
sample = np.random.randn(1000000)

sum_s = sample[0]
theta_n = sum_s / 1
history = [theta_n]
for N, s in enumerate(sample[1:], 2):
    theta_n = sum_s / N + 1 / N * (s - sum_s / N)
    sum_s += s
    history.append(theta_n)

plt.figure(figsize=(12, 5))
plt.plot(history)
plt.xlim(0, 1000000); plt.ylim(-0.01, 0.01)
plt.xlabel("Number of samples", fontsize=14); plt.ylabel("$\\theta=\mu_{ML}$",␣
↪fontsize=14)
plt.show()
```



2.3.6 Bayesian inference for the Gaussian

Now we develop a Bayesian treatment by introducing prior distributions over the parameters for the mean and variance. Consider a single Gaussian random variable x, for which we are given a set of N observations $\mathsf{x} = \{x_1, \ldots, x_N\}$.

1. Suppose that the variance σ^2 is known and we consider the task of inferring the mean μ:

The likelihood function is given by

$$p(\mathbf{x}|\mu) = \prod_{n=1}^{N} p(x_n|\mu) = \prod_{n=1}^{N} \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma^2}(x_n-\mu)^2\right\} = \frac{1}{(2\pi\sigma^2)^{N/2}} \exp\left\{-\frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2\right\}$$

NOTE: The likelihood function is **not** a probability distribution over μ and is not normalized.

A Gaussian prior $p(\mu)$ is a conjugate for the likelihood function, because the corresponding posterior is then given by the product of exponentials of quadratic functions of μ that results in a Gaussian. Thererfore, the prior has the form

$$p(\mu) = \mathcal{N}(\mu|\mu_0, \sigma_0^2)$$

and the posterior distribution can be proved to be given by

$$p(\mu|\mathbf{x}) = \mathcal{N}(\mu|\mu_N, \sigma_N^2)$$

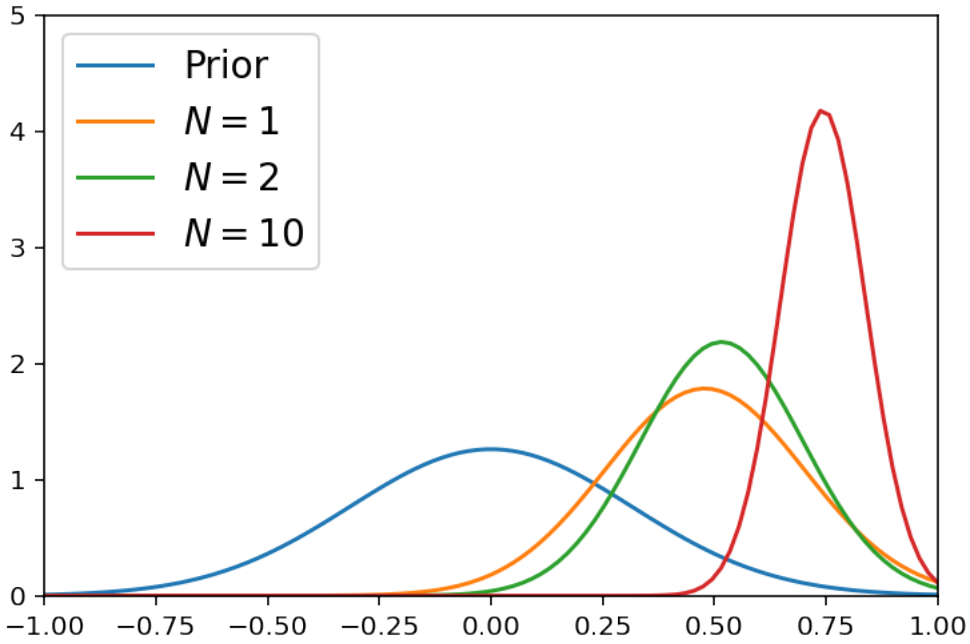
where

$$\mu_N = \frac{\sigma^2}{N\sigma_0^2 + \sigma^2}\mu_0 + \frac{N\sigma_0^2}{N\sigma_0^2 + \sigma^2}\mu_{ML}^2$$
$$\sigma_N^2 = \sigma_0^2 + \frac{1}{N}\sigma^2$$

Note that the mean of the posterior is a compromise between the prior mean μ_0 and the maximum likelihood solution μ_{ML}. If the number of observed data points $N = 0$, then μ_N and σ_N^2 reduce to the prior mean μ_0 and variance σ_0^2 as expected. For $N \to \infty$, the posterior mean is given by the maximum likelihood solution μ_{ML}, while the variance goes to zero and the posterior distribution becomes infinitely peaked around the maximum likelihood solution.

[20]:
```
x = np.linspace(-1, 1, 100)

prior = Gaussian(0, 0.1)
plt.plot(x, prior.pdf(x), label="Prior")

N=1
sample = Gaussian(mu=0.8, var=0.1**2).draw(N)
mN = (N * 0.1 * np.mean(sample)) / (N * 0.1 + 0.1)
varN= 1 / ((1 / 0.1) + (N / 0.1))
plt.plot(x, Gaussian(mN, varN).pdf(x), label="$N=1$")

N=2
sample = Gaussian(mu=0.8, var=0.1**2).draw(N)
mN = (N*(0.1)*np.mean(sample)) / (N * 0.1 + 0.1)
varN= 1 / ((1 / 0.1) + (N / 0.1))
plt.plot(x, Gaussian(mN, varN).pdf(x), label="$N=2$")
```

```
N=10
sample = Gaussian(mu=0.8, var=0.1**2).draw(N)
mN = (N * 0.1 * np.mean(sample)) / (N * 0.1 + 0.1)
varN= 1 / ((1 / 0.1) + (N / 0.1))
plt.plot(x, Gaussian(mN, varN).pdf(x), label="$N=10$")

plt.xlim(-1, 1); plt.ylim(0, 5)
plt.legend(fontsize=14)
plt.show()
```



2. Supporse that the mean μ is known and we wish to infer the variance σ^2:

The likelihood function is given by

$$p(\mathrm{x}|\lambda^{-1}) = \prod_{n=1}^{N} p(x_n|\lambda^{-1}) = \frac{\lambda^{N/2}}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{\lambda}{2}\sum_{n=1}^{N}(x-\mu)^2\right\}$$

where we choose to use the precision $\lambda = 1/\sigma^2$ for convenience.

The corresponding conjugate prior should therefore be proportional to the product of a power of λ and the exponential of a linear function of λ, which corresponds to a *gamma* distribution

$$\mathrm{Gam}(\lambda|a,b) = \frac{1}{\Gamma(a)} b^a \lambda^{a-1} \exp(-b\lambda)$$

In the case of variance, the conjugate prior is called the *inverse gamma* distribution. The mean of the gamma distribution is given by

$$\begin{aligned}
\mathbb{E}[\lambda] &= \int_0^\infty \lambda \frac{1}{\Gamma(a)} b^a \lambda^{a-1} \exp(-b\lambda) d\lambda \\
&= \frac{b^a}{\Gamma(a)} \int_0^\infty \lambda^a \exp(-b\lambda) d\lambda \\
&\overset{u=b\lambda}{=} \frac{b^a}{\Gamma(a)} \int_0^\infty \frac{u^a}{b^a} \exp(-u) \frac{1}{b} du \\
&= \frac{1}{b\Gamma(a)} \int_0^\infty u^a \exp(-u) du \\
&= \frac{1}{b\Gamma(a)} \int_0^\infty u^a \exp(-u) du \\
&= \frac{1}{b\Gamma(a)} \Gamma(a+1) = \frac{1}{b\Gamma(a)} a\Gamma(a) = \frac{a}{b}
\end{aligned}$$

where we have used the gamma function definition

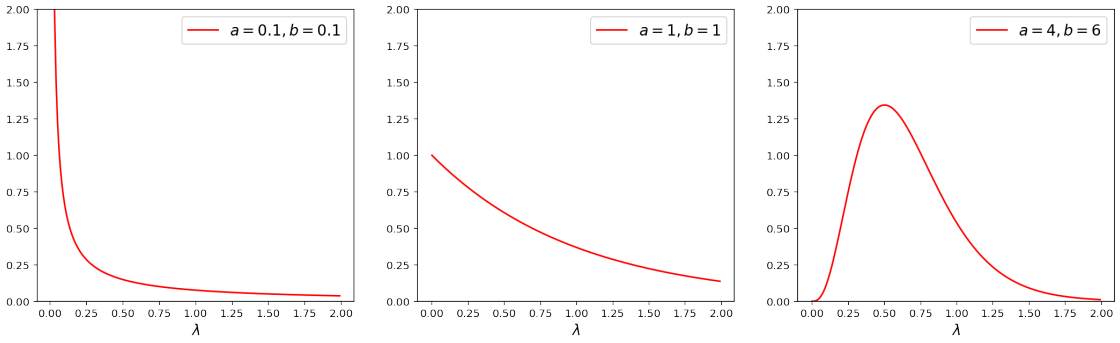
$$\Gamma(x) = \int_0^\infty u^{x-1} e^{-u} du$$

The variance is given by

$$\begin{aligned}
\text{var}[\lambda] &= \mathbb{E}[\lambda^2] - \mathbb{E}[\lambda]^2 \\
&= \mathbb{E}[\lambda^2] - \frac{a^2}{b^2} \\
&= \int_0^\infty \lambda^2 \text{Gam}(\lambda | a, b) d\lambda - \frac{a^2}{b^2} \\
&= \int_0^\infty \lambda^2 \frac{1}{\Gamma(a)} b^a \lambda^{a-1} \exp(-b\lambda) d\lambda - \frac{a^2}{b^2} \\
&= \frac{b^a}{\Gamma(a)} \int_0^\infty \lambda^{a+1} \exp(-b\lambda) d\lambda - \frac{a^2}{b^2} \\
&\overset{u=b\lambda}{=} \frac{b^a}{\Gamma(a)} \int_0^\infty \frac{u^{a+1}}{b^{a+1}} \exp(-u) \frac{1}{b} du - \frac{a^2}{b^2} \\
&= \frac{1}{b^2\Gamma(a)} \int_0^\infty u^{a+1} \exp(-u) du - \frac{a^2}{b^2} \\
&= \frac{1}{b^2\Gamma(a)} \Gamma(a+2) - \frac{a^2}{b^2} \\
&= \frac{1}{b^2\Gamma(a)} a(a+1)\Gamma(a) - \frac{a^2}{b^2} = \frac{a(a+1)}{b^2} - \frac{a^2}{b^2} = \frac{a}{b^2}
\end{aligned}$$

[21]:
```
lambda_space = np.arange(0, 2, 0.01)
```

```
plt.figure(figsize=(18, 5))

plt.subplot(1, 3, 1)
plt.plot(lambda_space, Gamma(a=0.1, b=0.1).pdf(lambda_space), color="red",␣
 ↪label="$a=0.1, b=0.1$")
plt.ylim(0, 2)
plt.xlabel("$\lambda$", fontsize=14);
plt.legend(fontsize=14)

plt.subplot(1, 3, 2)
plt.plot(lambda_space, Gamma(a=1, b=1).pdf(lambda_space), color="red",␣
 ↪label="$a=1, b=1$")
plt.ylim(0, 2)
plt.xlabel("$\lambda$", fontsize=14)
plt.legend(fontsize=14)

plt.subplot(1, 3, 3)
plt.plot(lambda_space, Gamma(a=4, b=6).pdf(lambda_space), color="red",␣
 ↪label="$a=4, b=6$")
plt.ylim(0, 2)
plt.xlabel("$\lambda$", fontsize=14)
plt.legend(fontsize=14)

plt.show()
```



Considering a prior $\text{Gam}(\lambda|a_0, b_0)$ and multiplying by the likelihood function, we obtain an **unnormalized** posterior distribution of the form

$$p(\lambda|\mathrm{x}) \propto \lambda^{a_0-1}\lambda^{N/2}\exp\{-b_0\lambda - \frac{\lambda}{2}\sum_{n=1}^{N}(x_n-\mu)^2\}$$

which is a gamma distribution $\text{Gam}(\lambda|a_N, b_N)$

$$a_N = a_0 + \frac{N}{2}$$
$$b_N = b_0 + \frac{1}{2}\sum_{n=1}^{N}(x_n - \mu)^2 = b_0 + \frac{N}{2}\sigma^2_{ML}$$

Thus, the effect of observing N data points is to increase the value of the coefficient a by $\frac{N}{2}$ and the value of coefficient b by $\frac{N}{2}\sigma^2_{ML}$.

3. Suppose that both the mean μ and the variance σ^2 are unknown:

TODO

2.3.7 Student's t-distribution

Consider a univariate Gaussian distribution $\mathcal{N}(x|\mu, \tau^{-1})$ and a gamma prior $\text{Gam}(\tau|a, b)$. If we integrate out the precision τ, we obtain the marginal distribution of x in the form

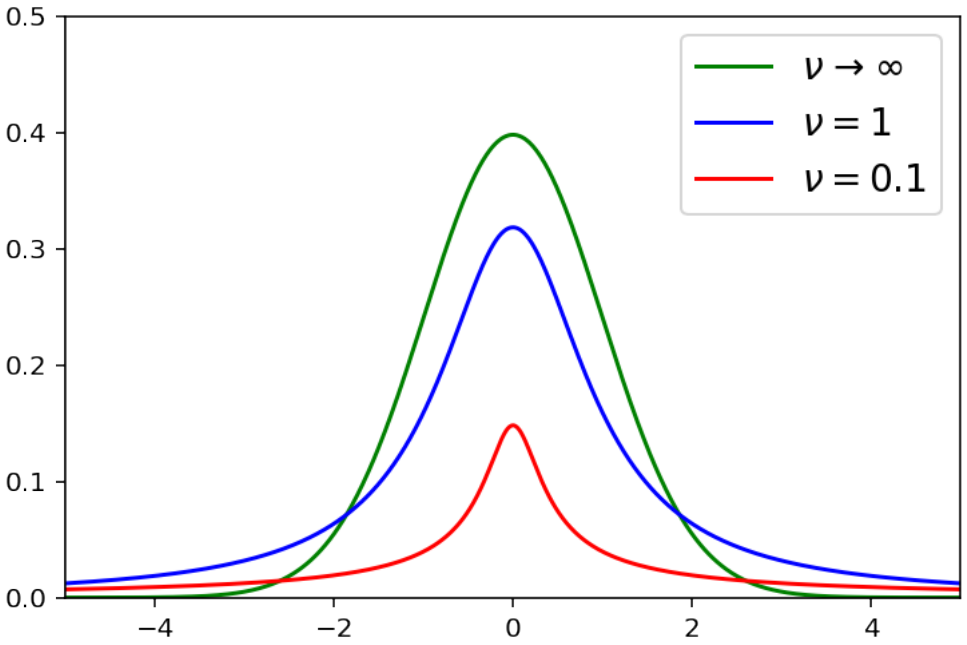
$$p(x|\mu, a, b) = \int_0^\infty \mathcal{N}(x|\mu, \tau^{-1})\text{Gam}(\tau|a, b)d\tau = \frac{b^a}{\Gamma(a)}\left(\frac{1}{2\pi}\right)^{1/2}\left[b + \frac{(x-\mu)^2}{2}\right]^{-a-1/2}\Gamma(a + 1/2)$$

where we have made the change of variable $z = \tau[b + (x - \mu)^2/2]$. By defining new parameters given by $\nu = 2a$ and $\lambda = a/b$, the distribution $p(x|\mu, a, b)$ takes the form

$$\text{St}(x|\mu, \lambda, \nu) = \frac{\Gamma(\nu/2 + 1/2)}{\Gamma(\nu/2)}\left(\frac{\lambda}{\pi\nu}\right)^{1/2}\left[1 + \frac{\lambda(x-\mu)^2}{\nu}\right]^{-\nu/2-1/2}$$

which is known as *Student's t-distribution*. The parameter λ is called the *precision* of the t-distribution, **even though it is not equal to inverse of the variance**, while the parameter ν is called the degrees of freedom.

[22]:
```
x_space = np.arange(-5, 5, 0.01)

plt.plot(x_space, StudentT(nu=100).pdf(x_space), color="green", label="$\\nu\\;
 ↪\\to\\;\\infty$")
plt.plot(x_space, StudentT(nu=1).pdf(x_space), color="blue", label="$\\nu=1$")
plt.plot(x_space, StudentT(nu=0.1).pdf(x_space), color="red", label="$\\nu=0.
 ↪1$")
plt.xlim(-5, 5); plt.ylim(0, 0.5)
plt.legend(fontsize=14)

plt.show()
```



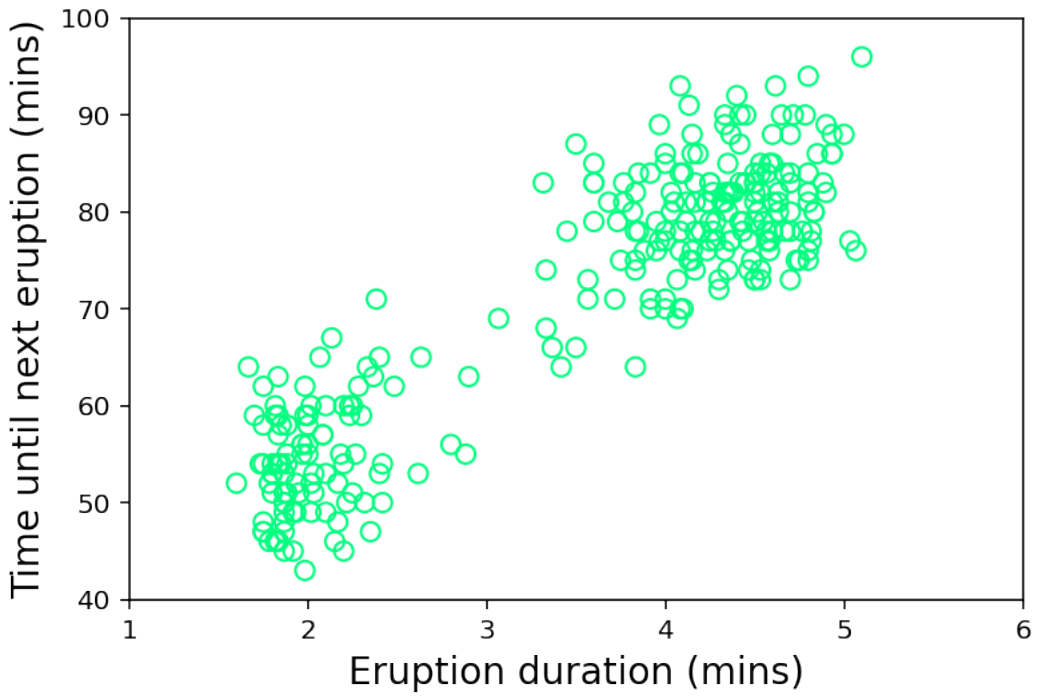
For $\nu = 1$ the t-distribution reduces to the *Cauchy* distribution, while for $\nu \to \infty$ the t-distribution becomes a Gaussian.

Note that the Student's t-distribution is obtained by adding up an infinite number of Gaussian distributions having the same mean, but different precisions. Thus, the t-distribution can be interpreted as a *mixture* of Gaussians. The result of this mixture is that the distribution has longer *tails*, which gives the t-distribution an important property called *robustness*. Essentially, is much less sensitive, than the Gaussian, to the presence of few data points which are *outliers*.

2.3.9 Mixtures of Gaussians

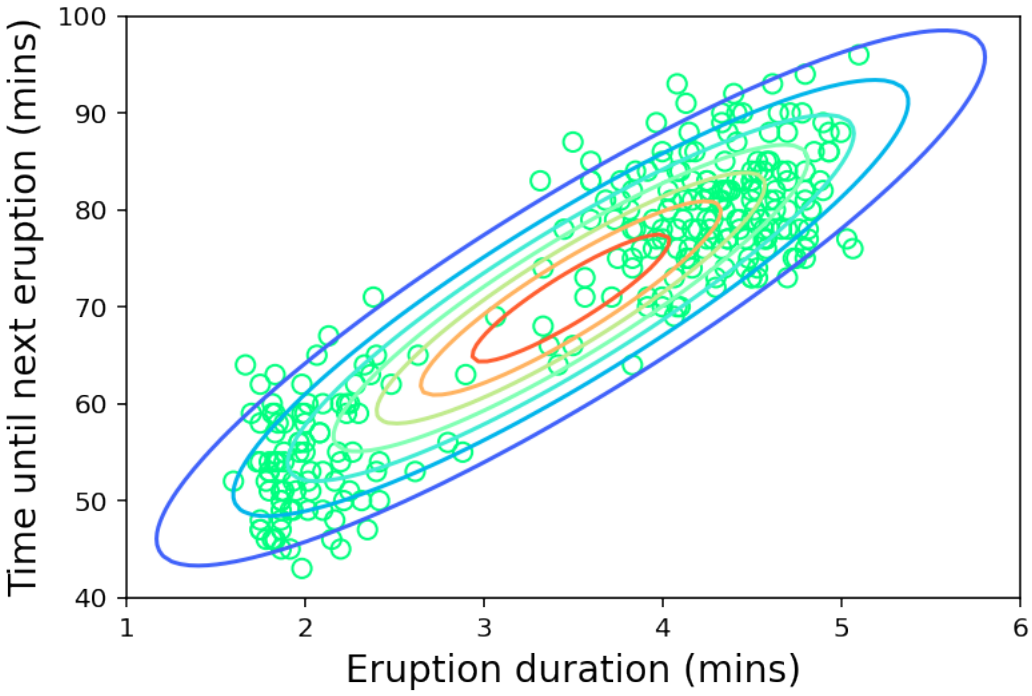
Although the Gaussian distribution has important analytical properties, it also has significant limitations regarding to modelling real data sets. Consider for instance the *Old Faithful* dataset comprise 272 measurements of the eruption of the Old Faithful geyser at Yellowstone National Park in the USA.

[23]:
```
old_faithful = load_old_faithful()
plt.scatter(old_faithful[:,:1], old_faithful[:,1:2], color='springgreen',␣
 ↪facecolors='none', s=50)
plt.xlim(1, 6); plt.ylim(40, 100)
plt.xlabel('Eruption duration (mins)', fontsize=14); plt.ylabel('Time until␣
 ↪next eruption (mins)', fontsize=14)
plt.show()
```



Each measurement comprises the duration of the eruption in minutes (x-axis) and the time in minutes to the next eruption (y-axis). Note that the data forms two dominant modes. Therefore, a simple Gaussian distribution **should be unable to capture this structure**.

[24]:
```
x, y = np.meshgrid(np.linspace(1, 6, 100), np.linspace(40, 100, 100))

# Perform maximum likelihood over the dataset
model = MultivariateGaussian(dim=2)
model.ml(old_faithful)
p = np.diag(model.pdf(np.array([x, y]).reshape(2, -1))).reshape(100, 100)

plt.scatter(old_faithful[:,:1], old_faithful[:,1:2], color='springgreen',␣
 ↪facecolors='none', s=50)
plt.contour(x, y, p, cmap="rainbow")
plt.xlim(1, 6); plt.ylim(40, 100)
plt.xlabel('Eruption duration (mins)', fontsize=14); plt.ylabel('Time until␣
 ↪next eruption (mins)', fontsize=14)
plt.show()
```



On the other hand, a linear combination (superposition) of two Gaussians should give a better characterization of the data since it should be able to place one Gaussian on each mode of the data. Such superpositions, formed by taking linear combinations of more basic distributions such as Gaussians, can be formulated as probabilistic models known as *mixture distributions*. We therefore define a superposition of K Gaussian densities of the form

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \ _k)$$

which is called a *mixture of Gaussians*. Each Gaussian density is called a *component of the mixture* and the parameters π_k are called the *mixing coefficients*. By using a sufficient number of Gaussians, and by optimizing their means and covariances, as well as the coefficients π_k in the linear combination, almost any continuous density can be approximated to arbitrary accuracy.

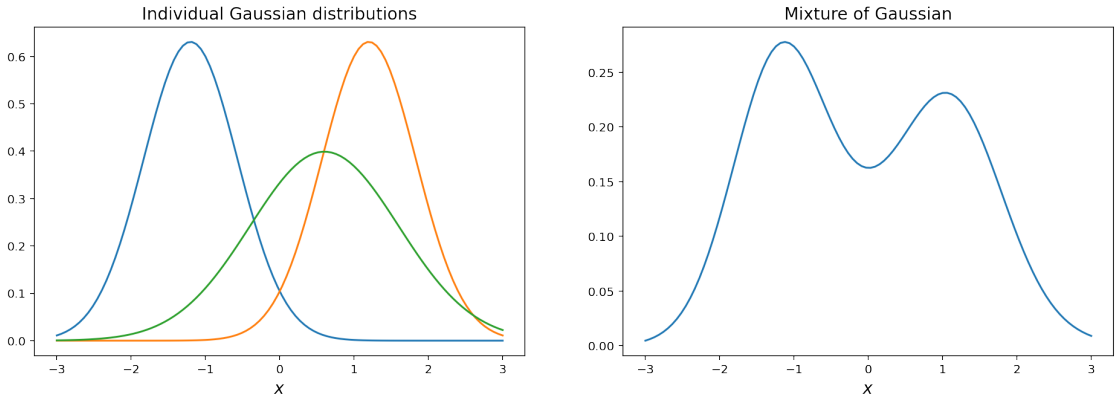
By integrating both sides of (2.188) over $\mathbf{x}$, and noting that both $p(\mathbf{x})$ and the individual Gaussian components are normalized, we obtain

$$\int p(\mathbf{x})d\mathbf{x} = \int \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \ _k)d\mathbf{x} \Leftrightarrow$$

$$\sum_{k=1}^{K} \pi_k \int \mathcal{N}(\mathbf{x}|\mu_k, \ _k)d\mathbf{x} = 1 \Leftrightarrow$$

$$\sum_{k=1}^{K} \pi_k = 1$$

[25]:
```
x_space = np.linspace(-3, 3, 100)

# Mixture component's parameters and coefficients
mu1, mu2, mu3 = -1.2, 1.2, 0.6
var1, var2, var3 = 0.4, 0.4, 1
pi1, pi2, pi3 = 0.4, 0.2, 0.3

component1 = Gaussian(mu=mu1, var=var1).pdf(x_space)
component2 = Gaussian(mu=mu2, var=var2).pdf(x_space)
component3 = Gaussian(mu=mu3, var=var3).pdf(x_space)

plt.figure(figsize=(16, 5))

plt.subplot(1, 2, 1)
plt.plot(x_space, component1, label="component 1")
plt.plot(x_space, component2, label="component 1")
plt.plot(x_space, component3, label="component 1")
plt.xlabel("$x$", fontsize=14)
plt.title("Individual Gaussian distributions", fontsize=14)

plt.subplot(1, 2, 2)
plt.plot(x_space, pi1 * component1 + pi2 * component2 + pi3 * component3)
plt.xlabel("$x$", fontsize=14)
plt.title("Mixture of Gaussian", fontsize=14)

plt.show()
```



Given a dataset $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, one way of finding the values of the parameters is to use maximum likelihood

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

Due to the presence of the summation over k inside the logarithm, the derivation is much more complex than the case of a single Gaussian. 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 or employ a powerful framework called *Expectation-Maximization*.

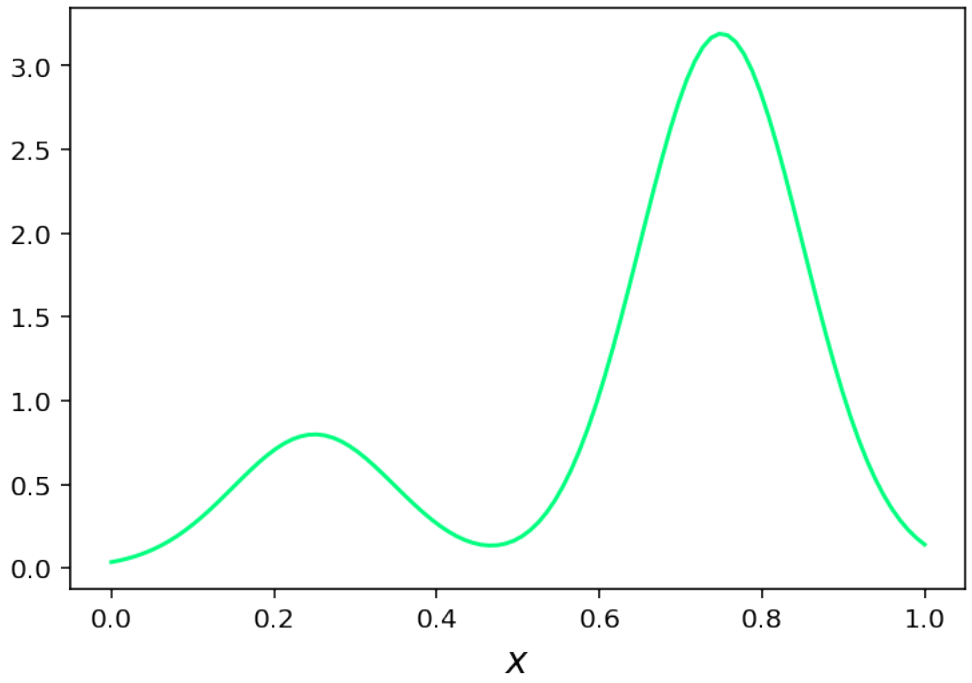
2.5 Nonparametric Methods

The use of probability distributions of specific functional forms governed by a number of parameters whose values are to be determined from the data are called **parametric** approaches to density modelling. The main limitation of such approches is that the assumed density might be a poor model of the true underlying distribution that generated the data in the first place. This leads to poor predictive performance, such as in the case of the *Old faithful* dataset and the simple unimodal Gaussian distribution.

Now we consider some **nonparametric** approaches to density estimation that make fewer assumptions about the form of the distribution. We focus mainly on simple frequentist methods, however, there are also nonparametric Bayesian methods.

For presentation purposes, consider a single continuous variable x that follows a mixture of two Gaussian distributions.

[26]:
```
means = [0.25, 0.75]
variances = [.01, .01]
proportions = [.2, .8]

component1 = Gaussian(mu=means[0], var=variances[0])
component2 = Gaussian(mu=means[1], var=variances[1])

x_space = np.linspace(0, 1, 100)
y_space = proportions[0] * component1.pdf(x_space) + proportions[1] *␣
 ↪component2.pdf(x_space)

plt.plot(x_space, y_space, color="springgreen")
plt.xlabel("$x$", fontsize=14)
plt.show()
```



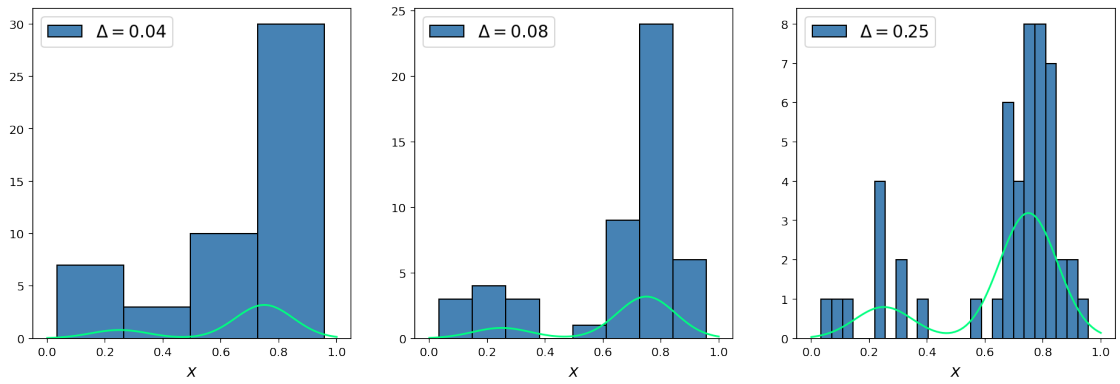
First, lets discuss the histogram methods for density estimation. Standard histograms simply partition x into distinct bins of width Δ_i and then count the number n_i of observations of x falling in bin i. In order to turn the counts into a normalized probability density, we divide by the total number N of observations and by the width Δ_i of the bins to obtain probability values for each bin given by

$$p_i = \frac{n_i}{N\Delta_i}$$

Given a sample dataset of $N = 50$ points, we plot three histogram density estimates corresponding to three choices for the bin width Δ. For very small Δ, the resulted density model is very spiky, while for very large Δ the result is too smooth and fails to capture the bimodal property of the curve. The best results are obtained by the intermediate value.

[27]:
```
# Sample N points from the mixture
N = 50
N1 = int(N * proportions[0])
N2 = N - N1
sampled_points = component1.draw(N1)
sampled_points = np.append(sampled_points, component2.draw(N2))

plt.figure(figsize=(16, 5))

plt.subplot(1, 3, 1)
plt.plot(x_space, y_space, color="springgreen")
```

```
plt.hist(sampled_points, bins=4, edgecolor='black', color="steelblue",␣
 ↪label="$\\Delta=0.04$")
plt.xlabel("$x$", fontsize=14)
plt.legend(fontsize=14)

plt.subplot(1, 3, 2)
plt.plot(x_space, y_space, color="springgreen")
plt.hist(sampled_points, bins=8, edgecolor='black', color="steelblue",␣
 ↪label="$\\Delta=0.08$")
plt.xlabel("$x$", fontsize=14)
plt.legend(fontsize=14)

plt.subplot(1, 3, 3)
plt.plot(x_space, y_space, color="springgreen")
plt.hist(sampled_points, bins=25, edgecolor='black', color="steelblue",␣
 ↪label="$\\Delta=0.25$")
plt.xlabel("$x$", fontsize=14)
plt.legend(fontsize=14)

plt.show()
```



The histogram method has the desired property that, once the histogram has been computed, the data itself can be discarded. Moreover, the approach is easily applied to data points are arriving sequentially. However, a major limitation is its scaling on high dimensional data. If we divide each variable in a D-dimensional space into M bins, then the total number of bins is M^D. Therefore, in a space of high dimensionality, the quantity of data needed to provide meaningful estimates of local probability density would be prohibitive.

The histogram approach to density estimation does, however, teach us two important lessons.

1. In order to estimate the probability density at a particular location, we should consider the data points that lie within some local neighbourhood of that location. For histograms, the neighbourhood property was defined by the bins.

2. The value of the smoothing parameter should be neither too large nor too small in order to

obtain good results.

2.5.1 Kernel density estimators

Now, suppose that observations are being drawn from an unknown probability density $p(\mathbf{x})$ in a D-dimensional Euclidean space. From our earlier insights about locality, let us consider a small region $\mathcal{R}$ containing $\mathbf{x}$. Then, the probability mass of this region is given by

$$P_{\mathcal{R}} = \int_{\mathcal{R}} p(\mathbf{x}) d\mathbf{x}$$

Then, given a data set comprising N observations drawn from $p(\mathbf{x})$, each data point has a probability $P_{\mathcal{R}}$ of falling into the region $\mathcal{R}$. Thus, in general, the total number K of points out of N, that lie inside $\mathcal{R}$ are distributed according to a binomial distribution

$$\text{Bin}(K|N, P_{\mathcal{R}}) = \frac{N!}{K!(N-K)!} P_{\mathcal{R}}^K (1 - P_{\mathcal{R}})^{1-K}$$

We can prove that the mean fraction of points falling inside the region $\mathcal{R}$ is $\mathbb{E}[K/N] = \frac{1}{N} N P_{\mathcal{R}} = P_{\mathcal{R}}$, and the variance around the mean $\text{var}[K/N] = P_{\mathcal{R}}(1 - P_{\mathcal{R}})/N$. Therefore, assuming a large N, variance is going to shrink and the distribution is going to be sharply peaked around the mean, leading to a $P_{\mathcal{R}}$ proportion of the N points (K) to be located inside $\mathcal{R}$

$$K \approx N P_{\mathcal{R}}$$

Furthermore, assuming that the region $\mathcal{R}$ is sufficiently small that the probability density $p(\mathbf{x})$ is roughly constant over the region, then we have

$$P_{\mathcal{R}} \approx p(\mathbf{x}) V$$

where V is the volume of $\mathcal{R}$. Combining these two assumptions, we obtain a density estimate in the form

$$p(\mathbf{x}) = \frac{K}{NV}$$

Note that the validity of the density depends on two **contradictory assumptions**, namely that the region $\mathcal{R}$ is sufficiently small that the density is approximately constant over the region and yet sufficiently large (in relation to the value of that density) that the number K of points falling inside the region is sufficient for the binomial distribution to be sharply peaked.

We can exploit the resulted density in two different ways:

1. Assume a K and determine the value of V from the data, giving rise to the k-nearest neighbour technique.
2. Assume a V and determine K from the data, giving rise to the kernel approach

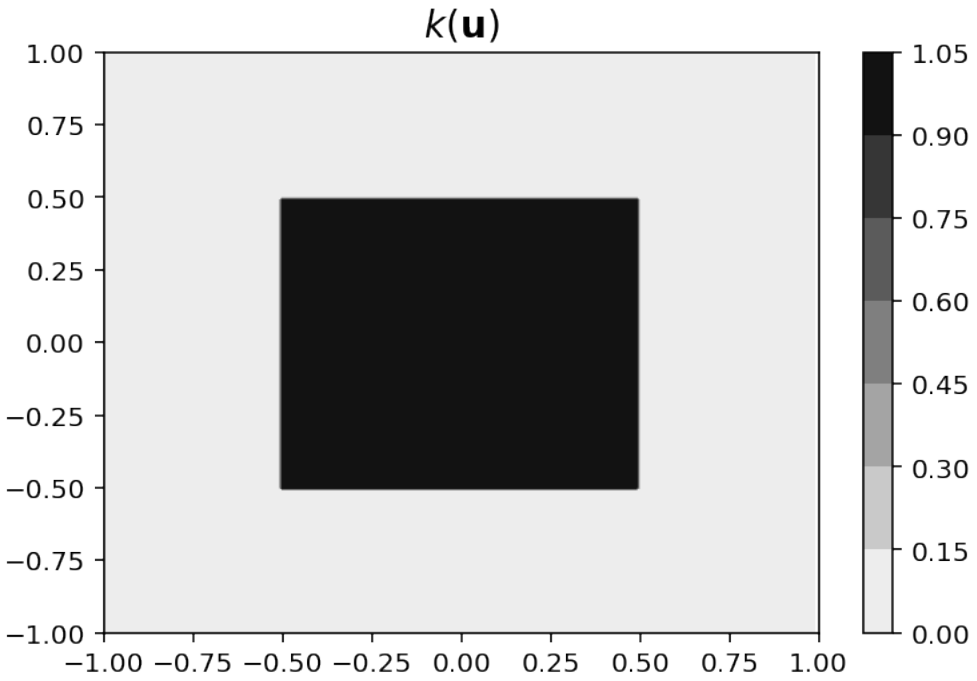
It can be proved that both the k-nearest neighbour density estimator and the kernel density estimator converge to the true probability density in the limit $N \to \infty$ provided V shrinks suitably with N, and K grows with N.

Consider the region $\mathcal{R}$ to be a small hypercube centred on the point $\mathbf{x}$ at which we wish to determine the probability density. In order to count the number K of points falling inside $\mathcal{R}$, it is convenient to define the function

$$k(\mathbf{u}) = \begin{cases} 1, & |u_i| \leq \frac{1}{2}, \quad \forall i \in \{1, \ldots, D\} \\ 0, & \text{otherwise} \end{cases}$$

which represents a unit cube centred on the origin.

[28]:
```
def k(u): return np.all(np.abs(u) <= 0.5)

U1, U2 = np.meshgrid(np.arange(-1, 1, 0.01), np.arange(-1, 1, 0.01))
plt.contourf(U1, U2, np.apply_along_axis(k, 0, np.array([U1, U2])),␣
 ↪cmap="binary")
plt.title("$k(\mathbf{u})$", fontsize=14)
plt.xlim(-1, 1); plt.ylim(-1, 1)
plt.colorbar(); plt.show()
```



The function $k(\mathbf{u})$ is an example of a *kernel function*, also called a **Parzen window**. The quantity $k(\frac{\mathbf{x}-\mathbf{x}_n}{h})$ is 1 if the data point $\mathbf{x}_n$ lies inside a cube of side h centred on $\mathbf{x}$, and zero otherwise. The total number of data points lying inside this cube is therefore given by

$$K = \sum_{n=1}^{N} k\left(\frac{\mathbf{x} - \mathbf{x}_n}{h}\right)$$

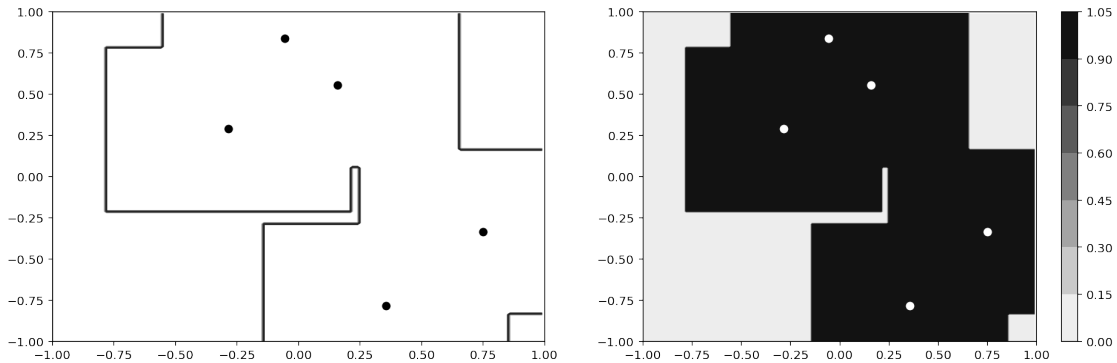
By substituting the expression for K into the formula of $p(\mathbf{x})$ derived above, we obtain the estimated density at $\mathbf{x}$ using a Parzen window

$$p(\mathbf{x}) = \frac{1}{N} \sum_{n=1}^{N} \frac{1}{h^D} k\left(\frac{\mathbf{x} - \mathbf{x}_n}{h}\right)$$

where $h^D = V$ for the volume of a hypercube of side h in D dimensions.

We can also interpret the above equation, not as a single cube centred on $\mathbf{x}$ but instead as the sum over N cubes centred on the N data points $\mathbf{x}_n$.

[29]:
```
N = 5
h = 1
data = np.random.uniform(-1, 1, size=(N, 2))
U1, U2 = np.meshgrid(np.arange(-1, 1, 0.01), np.arange(-1, 1, 0.01))

K = np.zeros(U1.shape)
for x in data:
    k = lambda u: np.all(np.abs(x - u) <= h / 2)
    K += np.apply_along_axis(k, 0, np.array([U1, U2]))

plt.figure(figsize=(16, 5))

plt.subplot(1, 2, 1)
plt.contour(U1, U2, K > 0, cmap="binary")
plt.scatter(data[:,0], data[:,1], color="black")
plt.xlim(-1, 1); plt.ylim(-1, 1)

plt.subplot(1, 2, 2)
c = plt.contourf(U1, U2, K > 0, cmap="binary")
plt.scatter(data[:,0], data[:,1], color="w")
plt.xlim(-1, 1); plt.ylim(-1, 1)
plt.colorbar(c); plt.show()
```

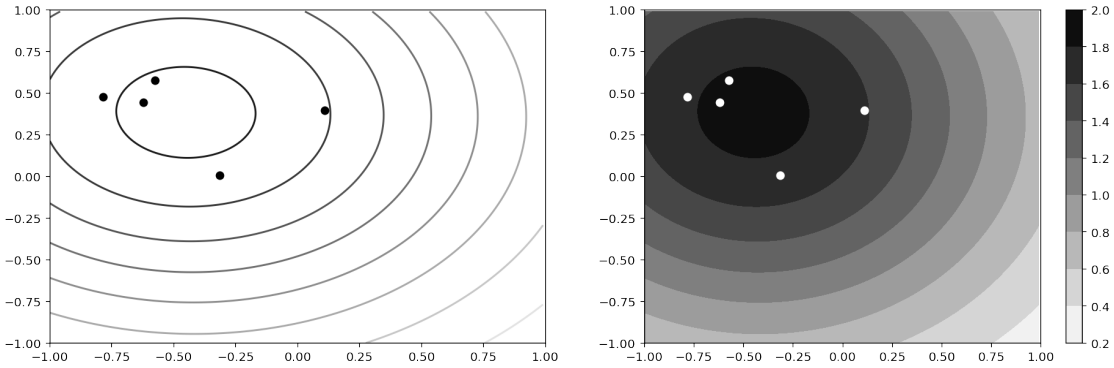


The kernel density estimator suffers from one of the same problems that the histogram method suffered from, namely the presence of artificial discontinuities, in this case at the boundaries of the cubes. We can obtain a smoother density model if we choose a smoother kernel function, and a common choice is the Gaussian, that gives rise to the following kernel density model

$$p(\mathbf{x}) = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{(2\pi h^2)^{1/2}}\exp\left\{-\frac{\|\mathbf{x}-\mathbf{x}_n\|^2}{2h^2}\right\}$$

where h represents the standard deviation of the Gaussian components. Thus, the density model is obtained by placing a Gaussian over each data point, adding up the contributions over the whole data set, and then dividing by N so that the density is correctly normalized.

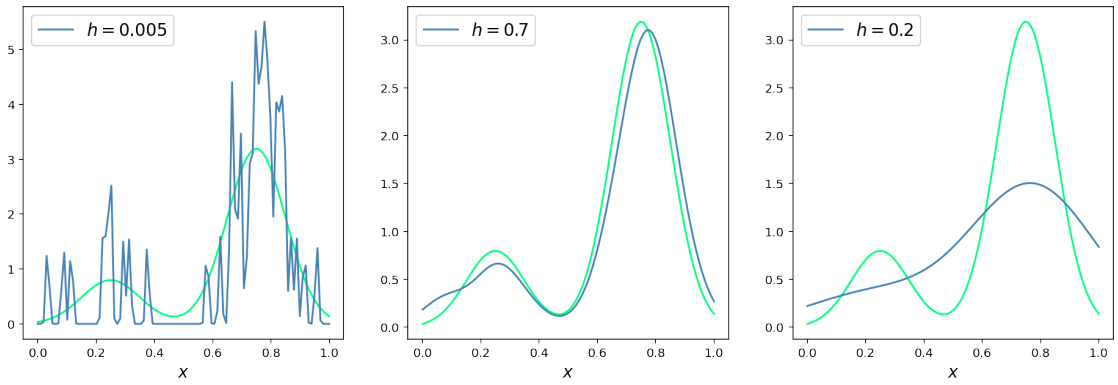
[30]:
```
N = 5
h = 1
data = np.random.uniform(-1, 1, size=(N, 2))
U1, U2 = np.meshgrid(np.arange(-1, 1, 0.01), np.arange(-1, 1, 0.01))

K = np.zeros(U1.shape)
for x in data:
    k = lambda u: np.exp(-np.linalg.norm(x - u)**2 / (2*h**2)) / np.sqrt(2*np.
 ↪pi*h**2)
    K += np.apply_along_axis(k, 0, np.array([U1, U2]))

plt.figure(figsize=(16, 5))

plt.subplot(1, 2, 1)
plt.contour(U1, U2, K, cmap="binary")
plt.scatter(data[:,0], data[:,1], color="black")
plt.xlim(-1, 1); plt.ylim(-1, 1)

plt.subplot(1, 2, 2)
c = plt.contourf(U1, U2, K, cmap="binary")
plt.scatter(data[:,0], data[:,1], color="w")
plt.xlim(-1, 1); plt.ylim(-1, 1)
plt.colorbar(c); plt.show()
```



Applying the Gaussian kernel to the data set used earlier to demonstrate the histogram technique, we see that, as expected, the parameter h plays the role of a smoothing parameter, and there is a trade-off between sensitivity to noise at small h and over-smoothing at large h. Again, the optimization of h is a problem in model complexity, analogous to the choice of bin width in histogram density estimation, or the degree of the polynomial used in curve fitting.

[31]:
```
def gaussian_kernel_density(h, data, x):
    K = 0
    for xn in data:
        k = lambda x: np.exp(-np.linalg.norm(x - xn,)**2 / (2*h**2)) / np.
 ↪sqrt(2*np.pi*h**2)
        K += k(x)
    return K / len(data)

def kdeg(x, X, h):
    """
    KDE under a gaussian kernel
    """
    N, D = X.shape
    nden, _ = x.shape # number of density points
    Xhat = X.reshape(D, 1, N)
    xhat = x.reshape(D, nden, 1)

    px = np.exp(-np.linalg.norm(xhat - Xhat, axis=0) ** 2 / (2*h**2)).
 ↪sum(axis=1) / (N*np.sqrt(2*np.pi*h**2))
    return px

plt.figure(figsize=(16, 5))

plt.subplot(1, 3, 1)
plt.plot(x_space, y_space, color="springgreen")
plt.plot(x_space, kdeg(x_space.reshape(-1, 1), sampled_points.reshape(-1, 1), 0.
 ↪005),
         color="steelblue", label="$h=0.005$")
```

```
plt.xlabel("$x$", fontsize=14)
plt.legend(fontsize=14)

plt.subplot(1, 3, 2)
plt.plot(x_space, y_space, color="springgreen")
plt.plot(x_space, [gaussian_kernel_density(0.07, sampled_points, x) for x in␣
 ↪x_space],
        color="steelblue", label="$h=0.7$")
plt.xlabel("$x$", fontsize=14)
plt.legend(fontsize=14)

plt.subplot(1, 3, 3)
plt.plot(x_space, y_space, color="springgreen")
plt.plot(x_space, [gaussian_kernel_density(0.2, sampled_points, x) for x in␣
 ↪x_space],
         color="steelblue", label="$h=0.2$")
plt.xlabel("$x$", fontsize=14)
plt.legend(fontsize=14)

plt.show()
```



The class of density models given by the Parzen estimators have the advantage that there is no computation involved during the *training* phase. They simply store the training set. However, this is also their great weakness, since the cost of evaluating the density grows linearly with the size of the data.

2.5.2 Nearest-neighbour methods

This issue is addressed by nearest-neighbour methods for density estimation.

We therefore return to our general result (2.246) for local density estimation, and assume a value for K and use the data to find an appropriate value for V. To that end, we consider a small sphere centred on the point $\mathbf{x}$ at which we wish to estimate the ensity $p(\mathbf{x})$. Then, we allow the radius of the sphere to grow until it contains precisely K data points. The estimate of the density is then

given by

$$p(\mathbf{x}) = \frac{K}{V_{\text{shpere}} N}$$

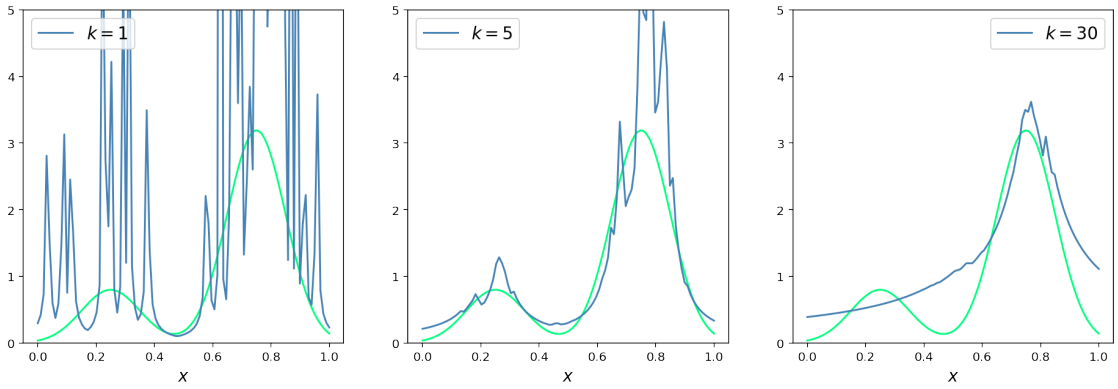
where V_{shpere} is volume of the resulting sphere. This technique is known as K nearest neighbours.

The n-dimensional volume of a Euclidean ball of radius R in n-dimensional Euclidean space is

$$V_{\text{shpere}} = \frac{\pi^{D/2} r^D}{\Gamma(\frac{D}{2} + 1)}$$

and is illustrated in Figure 2.26, for various choices of the parameter K, using the same data set as used in Figure 2.24 and Figure 2.25. We see that the value of K now governs the degree of smoothing and that again there is an optimum choice for K that is neither too large nor too small. Note that the model produced by K nearest neighbours is not a true density model because the integral over all space diverges.

[32]:
```
plt.figure(figsize=(16, 5))

plt.subplot(1, 3, 1)
plt.plot(x_space, y_space, color="springgreen")
model = NearestNeighborsDensity(k=1, data=sampled_points)
plt.plot(x_space, model.predict(x_space), color="steelblue", label="$k=1$")
plt.xlabel("$x$", fontsize=14); plt.ylim(0, 5)
plt.legend(fontsize=14)

plt.subplot(1, 3, 2)
plt.plot(x_space, y_space, color="springgreen")
model = NearestNeighborsDensity(k=5, data=sampled_points)
plt.plot(x_space, model.predict(x_space), color="steelblue", label="$k=5$")
plt.xlabel("$x$", fontsize=14); plt.ylim(0, 5)
plt.legend(fontsize=14)

plt.subplot(1, 3, 3)
plt.plot(x_space, y_space, color="springgreen")
model = NearestNeighborsDensity(k=30, data=sampled_points)
plt.plot(x_space, model.predict(x_space), color="steelblue", label="$k=30$")
plt.xlabel("$x$", fontsize=14); plt.ylim(0, 5)
plt.legend(fontsize=14)

plt.show()
```



The K-nearest-neighbour technique for density estimation can also be used for classification. To do so, we apply the K-nearest-neighbour density estimation technique to each class separately and then use the Bayes' theorem. Consider a data set comprising N_k points in class $\mathcal{C}_k$ and N points in total, so that $\sum_k N_k = N$. A sphere of volume V containing K_k points from class $\mathcal{C}_k$ defines the density of each class, given by

$$p(\mathbf{x}|\mathcal{C}_k) = \frac{K_k}{N_k V}$$

while, the unconditional density is given by

$$p(\mathbf{x}) = \frac{K}{NV}$$

and the class prior is given by

$$p(\mathcal{C}_k) = \frac{N_k}{N}$$

Using Bayes' theorem, we obtain the posterior probability of class membership

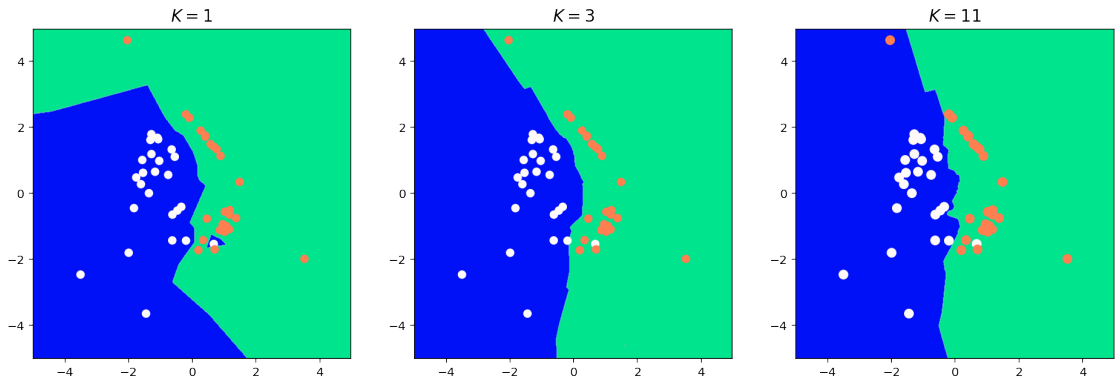
$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})} = \frac{K_k}{K}$$

Thus, to classify a new point, we identify the K-nearest points from the training data set and then assign the new point to the class having the largest number of representatives amongst this set. Ties can be broken at random. The particular case of $K = 1$ is called the *nearest-neighbour rule*, because a test point is simply assigned to the same class as the nearest point from the training set.

An interesting property of the nearest-neighbour rule classifier is that, in the limit $N \to \infty$, the error rate is never more than double the minimum achievable error rate of an optimal classifier, i.e., one that uses the true class distributions.

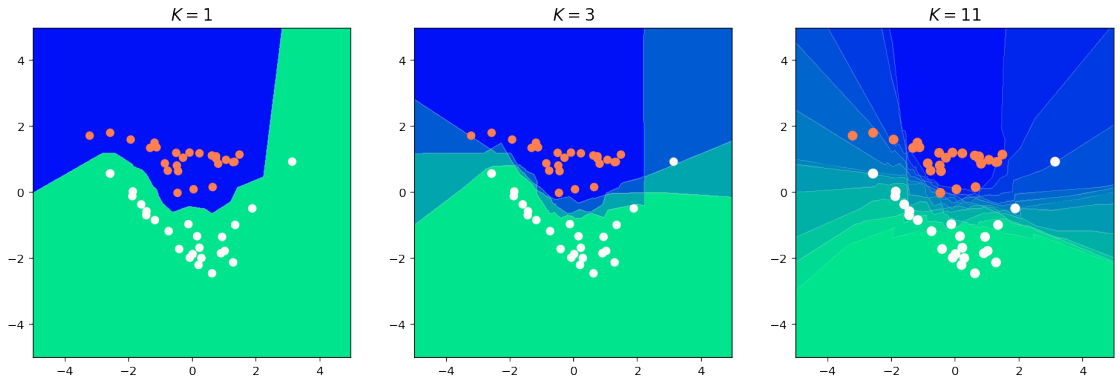
[33]:
```
N = 50
D = 2
```

```
x, t = make_classification(n_features=D,
                           n_informative=D,
                           n_redundant=0,
                           n_classes=2,
                           n_samples=N)

x_space = np.arange(-5, 5, 0.02)
x1, x2 = np.meshgrid(x_space, x_space)
x_hat = np.vstack((x1.ravel(), x2.ravel())).T

class_colors = ['coral' if t[i] == 1 else 'snow' for i in range(t.shape[0])]

plt.figure(figsize=(16, 5))

plt.subplot(1, 3, 1)
model = KNearestNeighborsClassifier(1, x, t)
Z = model.predict(x_hat).reshape(-1, 1)
plt.contourf(x_space, x_space, Z[:,0].reshape(x1.shape),
             cmap ='winter',levels=np.arange(-0.1, 1.1, 0.05), antialiased=True)
plt.scatter(x[:,0], x[:,1], color=class_colors)
plt.title("$K=1$", fontsize=14)

plt.subplot(1, 3, 2)
model = KNearestNeighborsClassifier(3, x, t)
Z = model.predict(x_hat).reshape(-1, 1)
plt.contourf(x_space, x_space, Z[:,0].reshape(x1.shape),
             cmap ='winter',levels=np.arange(-0.1, 1.1, 0.05), antialiased=True)
plt.scatter(x[:,0], x[:,1], color=class_colors)
plt.title("$K=3$", fontsize=14)

plt.subplot(1, 3, 3)
model = KNearestNeighborsClassifier(11, x, t)
Z = model.predict(x_hat).reshape(-1, 1)
plt.contourf(x_space, x_space, Z[:,0].reshape(x1.shape),
             cmap ='winter',levels=np.arange(-0.1, 1.1, 0.05), antialiased=True)
plt.scatter(x[:,0], x[:,1], color=class_colors, s=50)
plt.title("$K=11$", fontsize=14)

plt.show()
```



[34]:
```
N = 50
D = 2

x, t = make_classification(n_features=D,
                           n_informative=D,
                           n_redundant=0,
                           n_classes=2,
                           n_samples=N)

x_space = np.arange(-5, 5, 0.02)
x1, x2 = np.meshgrid(x_space, x_space)
x_hat = np.vstack((x1.ravel(), x2.ravel())).T

class_colors = ['coral' if t[i] == 1 else 'snow' for i in range(t.shape[0])]

plt.figure(figsize=(16, 5))

plt.subplot(1, 3, 1)
model = KNearestNeighborsClassifier(1, x, t)
Z = model.predict_proba(x_hat)
plt.contourf(x_space, x_space, Z[:,0].reshape(x1.shape),
             cmap ='winter',levels=np.arange(-0.1, 1.1, 0.05), antialiased=True)
plt.scatter(x[:,0], x[:,1], color=class_colors)
plt.title("$K=1$", fontsize=14)

plt.subplot(1, 3, 2)
model = KNearestNeighborsClassifier(3, x, t)
Z = model.predict_proba(x_hat)
plt.contourf(x_space, x_space, Z[:,0].reshape(x1.shape),
             cmap ='winter',levels=np.arange(-0.1, 1.1, 0.05), antialiased=True)
plt.scatter(x[:,0], x[:,1], color=class_colors)
plt.title("$K=3$", fontsize=14)
```

```
plt.subplot(1, 3, 3)
model = KNearestNeighborsClassifier(11, x, t)
Z = model.predict_proba(x_hat)
plt.contourf(x_space, x_space, Z[:,0].reshape(x1.shape),
             cmap ='winter',levels=np.arange(-0.1, 1.1, 0.05), antialiased=True)
plt.scatter(x[:,0], x[:,1], color=class_colors, s=50)
plt.title("$K=11$", fontsize=14)

plt.show()
```



3. Linear Models for Regression

Table of Contents

[1]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
from prml.datasets import generate_toy_data
from prml.distribution import (
    Gaussian,
```

```
    MultivariateGaussian
)
from prml.preprocessing import (
    PolynomialFeature,
    GaussianFeature,
    SigmoidFeature
)
from prml.linear import (
    LinearRegression,
    RidgeRegression,
    BayesianRegression,
    EvidenceApproximation
)

# Set random seed to make deterministic
np.random.seed(0)

# Ignore zero divisions and computation involving NaN values.
np.seterr(divide = 'ignore', invalid='ignore')

# Enable higher resolution plots
%config InlineBackend.figure_format = 'retina'
```

The goal of regression is to predict the value of one or more **continuous** target variables t given the value of a D-dimensional vector $\mathbf{x}$ of input variables. The polynomial curve that we used in Chapter 1 belongs to a broader class of functions called linear regression models, that are linear functions of the adjustable parameters. The simplest form of linear regression models are also linear functions of the input variables $\mathbf{x}$. However, a much more useful class of functions is the linear combinations of a fixed set of nonlinear functions of the input variables, known as *basis functions*.

3.1 Linear Basis Function Models

The simplest linear model for regression is one that involves a linear combination of the input variables

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + \cdots + w_D x_D$$

which is simply known as *linear regression*. This model is a linear function of the parameters and a linear function of the input variables, and this imposes significant limitations on the model. We therefore extend the class of models by considering linear combinations of fixed nonlinear functions of the input variables, of the form

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x}) = \sum_{j=0}^{M-1} w_j \phi_j(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x})$$

where $\phi_j(\mathbf{x})$ are known as *basis functions*.

The polynomial regression considered in Chapter 1 is an example of this model in which there is a single input variable x, and the basis functions take the form of powers of x so that $\phi_j(x) = x^j$. There is a plethora of possible choices for the basis functions, for instance

$$\phi_j(x) = \exp\left\{-\frac{(x-\mu_j)^2}{2s^2}\right\}$$

are referred to as *Gaussian* basis functions, where μ_j govern the locations of the functions in input space, and s governs their spatial scale. Another possibility is the sigmoidal basis function of the form

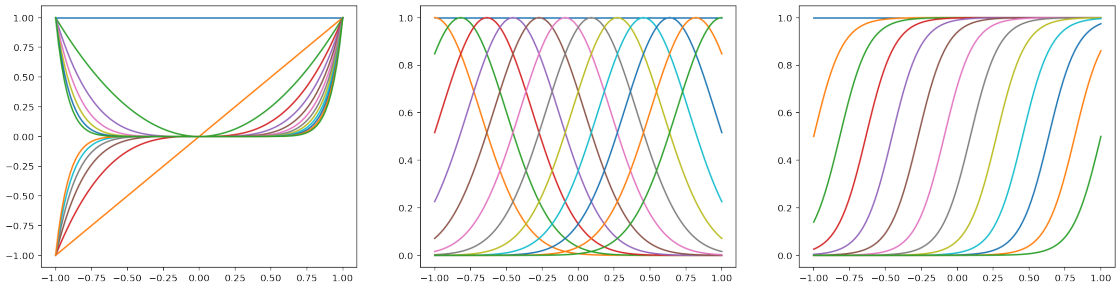
$$\phi_j(x) = \sigma\left(\frac{x-\mu_j}{s}\right)$$

where $\sigma(a)$ is the logistic sigmoid function defined by

$$\sigma(a) = \frac{1}{1+\exp(-a)}$$

We present examples of these three families of basis functions using different parameters.

[2]:
```
x_space = np.linspace(-1, 1, 100)

# Create 10 degree polynomial basis functions
x_polynomial = PolynomialFeature(degree=12).transform(x_space)

# Create 10 Gaussian basis functions
x_gaussian = GaussianFeature(mean=np.linspace(-1, 1, 12), sigma=0.1).
 ↪transform(x_space)

# Create 10 sigmoid basis functions
x_sigmoid = SigmoidFeature(mean=np.linspace(-1, 1, 12), sigma=0.1).
 ↪transform(x_space)

plt.figure(figsize=(20, 5))
for i, x in enumerate([x_polynomial, x_gaussian, x_sigmoid]):
    plt.subplot(1, 3, i + 1)
    for j in range(x.shape[1]):
        plt.plot(x_space, x[:, j])
```



3.1.1 Maximum likelihood and least squares

In Chapter 1, we fitted polynomial functions to data sets by minimizing a sum-of-squares error function. We also showed that this error function could be motivated as the maximum likelihood solution under an assumed Gaussian noise model. Let us return to the discussion of Chapter 1 and consider the least squares approach, and its relation to maximum likelihood, in more detail.

As before, we assume that the target variable t is given by a deterministic function $y(\mathbf{x}, \mathbf{w})$ having additive Gaussian noise so that

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon$$

where ϵ is a zero mean Gaussian random variable with precision (inverse variance) β. Thus we can write

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1})$$

Note that the Gaussian noise assumption implies that the conditional distribution of t given $\mathbf{x}$ is unimodal, which may be inappropriate for some applications.

Now consider a data set of inputs $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$ along corresponding target alues $\mathtt{t} = (t_1, \dots, t_N)^{\mathrm{T}}$. Assuming that the data points are i.i.d, we obtain the likelihood function (function of the adjustable parameters $\mathbf{w}$ and β), in the form

$$p(\mathtt{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(t_n|y(\mathbf{x}_n, \mathbf{w}), \beta^{-1}) = \prod_{n=1}^{N} \mathcal{N}(t_n|\mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n), \beta^{-1})$$

NOTE: In many textbooks, the input variables $\mathbf{x}$ are dropped from the set of conditioning variables, since, we do not seek to model the distribution of $\mathbf{x}$.

Taking the logarithm of the likelihood function, we have,

$$\begin{aligned}
\ln p(\mathtt{t}|\mathbf{X}, \mathbf{w}, \beta) &= \sum_{n=1}^{N} \ln \mathcal{N}(t_n|\mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n), \beta^{-1}) \\
&= \frac{N}{2} \ln \beta - \frac{N}{2} \ln 2\pi - \frac{\beta}{2} \sum_{n=1}^{N} \left(t_n - \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n)\right)^2 \\
&= \frac{N}{2} \ln \beta - \frac{N}{2} \ln 2\pi - \beta E_D(\mathbf{w})
\end{aligned}$$

By maximizing likelihood we can determine the parameters $\mathbf{w}$ and β. As already observed in Chapter 1 the maximization under a conditional Gaussian noise distribution is equivalent to minimizing the sum-of-squares error function given by $E_D(\mathbf{w})$. The gradient of the log likelihood function takes the form

$$\nabla p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \sum_{n=1}^{N} (t_n - \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n))\phi(\mathbf{x}_n)^{\mathrm{T}}$$

Setting this gradient to zero and solving for $\mathbf{w}$ gives

$$\mathbf{w}_{ML} = (\ ^{\mathrm{T}}\ )^{-1}\ ^{\mathrm{T}}\mathbf{t}$$

which are known as the *normal equations* for the least squares problem. Here is an $N \times M$ matrix, called the *design matrix*, whose elements are given by $\Phi_{nj} = \phi_j(\mathbf{x}_n)$, so that

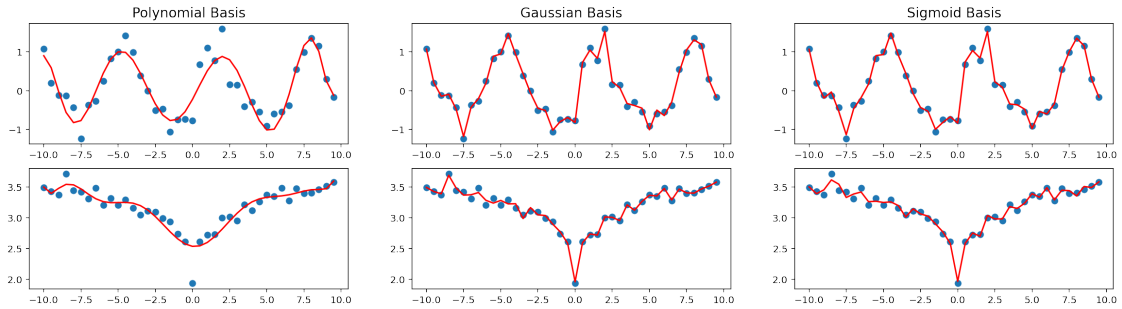
$$= \begin{pmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \cdots & \phi_{M-1}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \cdots & \phi_{M-1}(\mathbf{x}_N) \end{pmatrix}$$

By maximizing the log likelihood function over the noise precision parameter β, we obtain

$$\beta_{ML} = \frac{1}{N} \sum_{n=1}^{N} (t_n - \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n))^2$$

[3]:
```
# Create polynomial, gaussian and sigmoid basis functions
polynomial_basis = PolynomialFeature(degree=10)
gaussian_basis = GaussianFeature(mean=np.arange(-9, 9, 0.5), sigma=1)
sigmoid_basis = SigmoidFeature(mean=np.arange(-9, 9, 0.5), sigma=1)

# Lets train a linear regression model on a couple of datasets
model = LinearRegression()

plt.figure(figsize=(20, 5))

X = np.arange(-10, 10, 0.5).reshape((-1, 1))
y = np.sin(X) + np.random.randn(X.shape[0], X.shape[1]) * 0.3

for i, basis in enumerate([polynomial_basis, gaussian_basis, sigmoid_basis]):
    plt.subplot(2, 3, i + 1)
    plt.scatter(X, y)
    x_train_features = basis.transform(X)
    model.fit(x_train_features, y)
    plt.plot(X, model.predict(x_train_features)[0], color='red')
    plt.title(['Polynomial Basis', 'Gaussian Basis', 'Sigmoid Basis'][i],␣
 ↪fontsize=14)

X = np.arange(-10, 10, 0.5)
y = 0.8 * abs(X)**0.3 + 2 + np.random.randn(X.shape[0]) * 0.1
```

```
for i, basis in enumerate([polynomial_basis, gaussian_basis, sigmoid_basis]):
    plt.subplot(2, 3, i + 4)
    plt.scatter(X, y)
    x_train_features = basis.transform(X)
    model.fit(x_train_features, y)
    plt.plot(X, model.predict(x_train_features)[0], color='red')

plt.show()
```



3.1.3 Sequential learning

Batch learning, such as the maximum likelihood solution, requires processing of the entire training set at once. However, this can be computationally costly for large datasets. Sequential learning or *online* learning algorithms consider data points one at a time and update the model parameters after processing each point. We can obtain a sequential learning algorithm by applying a technique called *stochastic gradient descent* also known as *sequential gradient descent*.

If an error function comprises a sum over data points, then for a data point n, the stochastic gradient descent updates the parameter vector $\mathbf{w}$ as follows,

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n$$

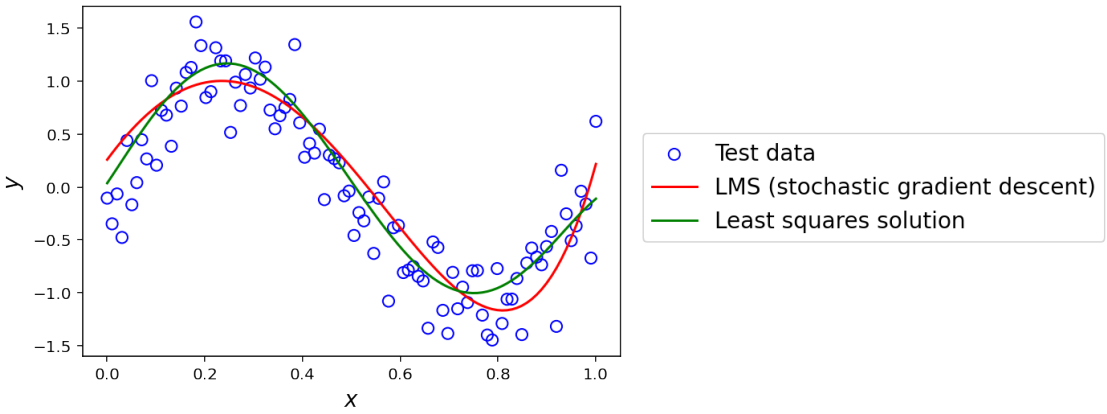
where τ is the iteration number and η is a learning rate parameter. The value of $\mathbf{w}^{(0)}$ can be initialized to some random vector. For the case of the sum-of-squares error function, this gives

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta(\mathbf{w}^{(\tau)T}\phi_n - t_n)\phi_n$$

This is also known as the *least-mean-squares* (LMS) algorithm or more commonly as gradient descent.

[4]:
```
# Generate an example dateset
sin = lambda x: np.sin(2 * np.pi * x)
x_train, y_train = generate_toy_data(sin, sample_size=100, std=0.3)
x_test, y_test = generate_toy_data(sin, sample_size=100, std=0.3)

# Create polynomial features
```

```
polynomial = PolynomialFeature(degree=5)
x_train_features = polynomial.transform(x_train)
x_test_features = polynomial.transform(x_test)

# Fit linear regression using both gradient descent and least squares
model = LinearRegression()
model.fit_lms(x_train_features, y_train, eta=0.1)
y_pred_lms, _ = model.predict(x_test_features)

model.fit(x_train_features, y_train)
y_pred_ls, _ = model.predict(x_test_features)

plt.scatter(x_test, y_test, facecolor="none", edgecolor="b", s=50, label="Test␣
 ↪data")
plt.plot(x_test, y_pred_lms, color="r", label="LMS (stochastic gradient␣
 ↪descent)")
plt.plot(x_test, y_pred_ls, color="g", label="Least squares solution")
plt.xlabel('$x$', fontsize=14); plt.ylabel('$y$', fontsize=14)
plt.legend(bbox_to_anchor=(1, 0.7), loc=2, borderaxespad=1, fontsize=14);
plt.show()
```



3.1.4 Regularized least squares

In Chapter 1, we introduced the idea of adding a regularization term to the error function in order to control over-fitting. To that end, the total error function to be minimized takes the form,

$$E_D(\mathbf{w}) + \lambda E_W(\mathbf{w})$$

where λ is the regularization coefficient that controls the balance between the *data-dependent error* $E_D(\mathbf{w})$ and the regularization term over the parameters $E_W(\mathbf{w})$. One of the simplest regularizers we can employ is given by the sum-of-squares of the weight vector

$$E_W(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{w}$$

Combining the sum-of-squares error function for the data and the quadratic regularizer leads to the following total error function

$$\frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^T\phi(\mathbf{x}_n)\}^2 + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

This particular regularizer is also known as *weight decay* since it encourages weight values to decay towards zero, unless supported by data. In statistics, it provides an example of a *parameter shrinkage* method. Moreover, it has the advantage that the error function remains quadratic over $\mathbf{w}$, thus, it can be minimized in closed form using calculus.

A more general formulation for the regularized error is given by

$$E_W(\mathbf{w}) = \frac{\lambda}{2}\sum_{j=1}^{M}|w_j|^q$$

where $q = 2$ recovers the quadratic regularizer. The case of $q = 1$ is known as the *lasso* in the literature. It has the property that if λ is sufficiently large, some of the parameters w_j are driven to zero, leading to sparse models in which the corresponding basis function plays no role.

Regularization allows complex models (having a large number of parameters) to be trained on data sets of limited size, avoiding over-fitting. Unfortunately, the problem of determining the optimal model is then shifted from finding the appropriate number of basis functions to determining a suitable value for the regularization coefficient λ.

3.2 Bias-Variance Decomposition

The introduction of regularization terms can control over-fitting for models having many parameters. However, the question of how to determine a suitable value for the regularization coefficient λ remains. Seeking the solution that minimizes the regularized error function with respect to both the weight vector and the regularization coefficient λ is not the right approach since it leads to the unregularized solution $\lambda = 0$.

The optimal prediction for the squared loss function is given by the conditional expectation

$$h(\mathbf{x}) = \mathbb{E}[t|\mathbf{x}] = \int tp(t|\mathbf{x})\mathrm{d}t$$

Moreover, the expected squared loss is given by

$$\mathbb{E}[L] = \int \big\{y(\mathbf{x}) - h(\mathbf{x})\big\}^2 p(\mathbf{x})\mathrm{d}\mathbf{x} + \int\int \big\{h(\mathbf{x}) - t\big\}^2 p(\mathbf{x},t)\mathrm{d}\mathbf{x}\mathrm{d}t$$

The second term is independent of $y(\mathbf{x})$ and arises from the intrinsic noise on the data. Assuming we can find a function $y(\mathbf{x}) = h(\mathbf{x})$, the second term represents the minimum achievable value of

the expected loss. Thus, our goal is indeed to find a function $y(\mathbf{x})$ that makes the first term a minimum, ideally zero, since the loss function is always non-negative.

A frequentist treatment of the problem involves making an estimate of $\mathbf{w}$ based on a data set $\mathcal{D}$. In order to interpret the uncertainty of this estimate consider the following thought experiment. Suppose we had a large number of data sets each of size N, drawn independently from $p(t, \mathbf{x})$. Then, considering each dataset in turn, we can obtain a prediction function $y(\mathbf{x}; \mathcal{D})$. As expected, each dataset should give a different functions and consequently different values for the squared loss. The performance of a particular learning algorithm can then be assesed by taking the average over this ensemble of data sets.

Now consider the integrand of the first them, given a particular dataset,

$$\{y(\mathbf{x}; \mathcal{D}) - h(\mathbf{x})\}^2$$

If we add and subtract the average of $y(\mathbf{x}; \mathcal{D})$ over the ensemble of data sets, expressed as $\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]$, we obtain

$$\begin{aligned}
&\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] + \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2 \\
&= \{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}^2 + \{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2 - 2\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}
\end{aligned}$$

By taking the expectation over all data sets $\mathcal{D}$ on both sides of the equation, we obtain

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D}}\Big[\{y(\mathbf{x}; \mathcal{D}) - h(\mathbf{x})\}^2\Big] \\
&= \mathbb{E}_{\mathcal{D}}\Big[\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}^2 + \{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2 - 2\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}\Big] \\
&= \mathbb{E}_{\mathcal{D}}\Big[\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}^2\Big] + \mathbb{E}_{\mathcal{D}}\Big[\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2\Big] - \mathbb{E}_{\mathcal{D}}\Big[2\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h\Big] \\
&= \mathbb{E}_{\mathcal{D}}\Big[\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}^2\Big] + \mathbb{E}_{\mathcal{D}}\Big[\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2\Big] \\
&- \mathbb{E}_{\mathcal{D}}\Big[2\{y(\mathbf{x}; \mathcal{D})\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - y(\mathbf{x}; \mathcal{D})h(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]^2 + \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]h(\mathbf{x})\}\Big] \\
&= \mathbb{E}_{\mathcal{D}}\Big[\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}^2\Big] + \mathbb{E}_{\mathcal{D}}\Big[\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2\Big] \\
&- 2\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]^2 + 2\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]h(\mathbf{x}) + 2\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]^2 - 2\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]h(\mathbf{x}) \\
&= \mathbb{E}_{\mathcal{D}}\Big[\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2\Big] + \mathbb{E}_{\mathcal{D}}\Big[\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}^2\Big] \\
&= \{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2 + \mathbb{E}_{\mathcal{D}}\Big[\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}^2\Big]
\end{aligned}$$

Note that the last term vanished giving a sum of two terms for the expected squarred loss. The first term, called the **squared bias**, represents the extend to which the average prediction over all data sets differs from the optimal loss function $h(\mathbf{x})$. The second term, called **variance**, measures the extend to which the solutions for each data set vary around their average, in other words, it measures how sensitive is function $y(\mathbf{x}; \mathcal{D})$ to the particular choice of data set.

By substituting this decomposision of the squared loss back into the expected squared loss, we note that

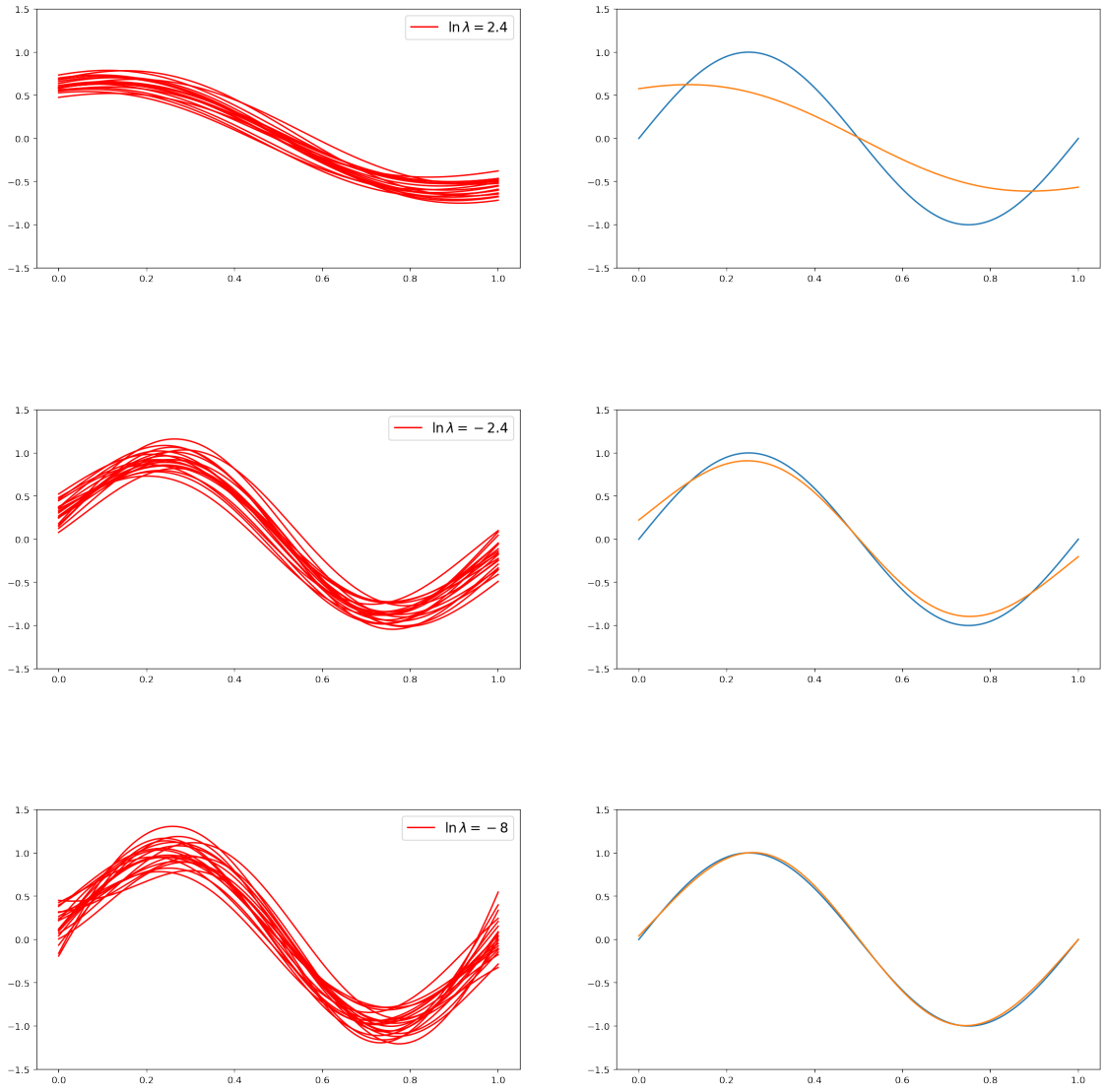
$$\text{expected loss} = (\text{bias})^2 + \text{variance} + \text{noise}$$

Our goal is to minimize the expected loss, thus, minimizing both the bias and the variance. However, there is a trade-off between bias and variance. Very flexible models have low bias and high variance, while relatively rigid models have high bias and low variance. The best model is the one that balances these two quantities.

Consider $L = 100$ data sets, each containing $N = 25$ data points, independently from the sinusoidal curve $h(x) = \sin(2\pi x)$. For each data set $\mathcal{D}^{(l)}$, a model using 24 Gaussian basis functions is trained, by minimizing the regularized error function to give a prediction function $y^{(l)}$.

[5]:
```
L = 100 # number of datasets
N = 25 # number of points per dataset

# the optimal regression function is the sinusoidal
def h(x):
    return np.sin(2 * np.pi * x)

gaussian_basis = GaussianFeature(np.linspace(0, 1, 24), 0.1)

# create a test set
x_test = np.linspace(0, 1, 1000)
x_test_features = gaussian_basis.transform(x_test)
y_test = h(x_test)

# create L datasets
datasets = []
for i in range(L):
    x_train, y_train = generate_toy_data(h, N, 0.3)
    x_train_features = gaussian_basis.transform(x_train)
    datasets.append((x_train_features, y_train))

# apply ridge regression on the L datasets
for ln_lambda in [2.4, -2.4, -8]:
    predictions = []
    plt.figure(figsize=(20, 5))
    plt.subplot(1, 2, 1)
    for i, (x_train, y_train) in enumerate(datasets):
        model = RidgeRegression(alpha=math.exp(ln_lambda))
        model.fit(x_train, y_train)
        y, _ = model.predict(x_test_features)
        predictions.append(y)
        if i == 0:
            plt.plot(x_test, y, color="red", label=f"$\ln\lambda={ln_lambda}$")
            plt.ylim(-1.5, 1.5)
```

```
        elif i < 20:
            plt.plot(x_test, y, color="red")
        plt.legend(fontsize=14)

    plt.subplot(1, 2, 2)
    plt.plot(x_test, y_test)
    plt.plot(x_test, np.asarray(predictions).mean(axis=0))
    plt.ylim(-1.5, 1.5)
    plt.show()
```



The top row corresponds to a larger value for the regularization coefficient λ and results in low variance but high bias. The bottom row for which λ is small, there is a large variance but low bias.

Note that the result of averaging many solutions for a complex model ($M = 25$) is very good,

suggesting that averaging may be a beneficial procedure. Indeed, the weighted average of multiple solutions lies at the core of a Bayesian approach, although the averaging is done respect to the posterior distribution of the parameters, not to multiple datasets.

3.3 Bayesian Linear Regression

In order to tackle the over-fitting of maximum likelihood, we turn to the Bayesian treatment of linear regression that leads to automatic methods for determining model complexity using the training data alone.

3.3.1 Parameter distribution

We begin our discussion of the Bayesian treatment of linear regression by introducing a prior probability distribution over the model parameters $\mathbf{w}$. Assume for the moment, that the noise precision parameter β is a known constant. Note that the likelihood function $p(\mathtt{t}|\mathbf{X}, \mathbf{w})$ is the exponential of a quadratic function of $\mathbf{w}$. Thus, the corresponding conjugate prior is given by a Gaussian distribution of the form

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|, \mathbf{m}_0, \mathbf{S}_0)$$

The posterior distribution is proportional to the product of the likelihood and the prior. Due to the choice of a conjugate Gaussian prior distribution, the posterior is also Gaussian. Thus, to derive the form of the posterior, we focus on the exponential term

$$\begin{aligned}
\text{exponential term} &= -\frac{\beta}{2}\sum_{n=1}^{N}\left\{t_n - \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n)\right\}^2 - \frac{1}{2}(\mathbf{w}-\mathbf{m}_0)^{\mathrm{T}}\mathbf{S}_0^{-1}(\mathbf{w}-\mathbf{m}_0) \\
&= -\frac{\beta}{2}\sum_{n=1}^{N}\left\{t_n^2 - 2t_n\mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n) - \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n)\phi(\mathbf{x}_n)^{\mathrm{T}}\mathbf{w}\right\} - \frac{1}{2}(\mathbf{w}^{\mathrm{T}}\mathbf{S}_0^{-1}\mathbf{w} - 2\mathbf{m}_0^{\mathrm{T}}\mathbf{S}_0^{-1}\mathbf{w} + \mathbf{m}_0^{\mathrm{T}}\mathbf{S}_0^{-1}\mathbf{m}_0) \\
&= -\frac{1}{2}\mathbf{w}^{\mathrm{T}}\left(\sum_{n=1}^{N}\beta\phi(\mathbf{x}_n)\phi(\mathbf{x}_n)^{\mathrm{T}} + \mathbf{S}_0^{-1}\right)\mathbf{w} - \frac{1}{2}\mathbf{w}^{\mathrm{T}}\left(-2\mathbf{S}_0^{-1}\mathbf{m}_0 - \sum_{n=1}^{N}2\beta t_n\phi(\mathbf{x}_n)\right) - \frac{1}{2}\left(\sum_{n=1}^{N}\beta t_n^2 + \mathbf{n}\right. \\
&= -\frac{1}{2}\mathbf{w}^{\mathrm{T}}\left(\sum_{n=1}^{N}\beta\phi(\mathbf{x}_n)\phi(\mathbf{x}_n)^{\mathrm{T}} + \mathbf{S}_0^{-1}\right)\mathbf{w} + \mathbf{w}^{\mathrm{T}}\left(\mathbf{S}_0^{-1}\mathbf{m}_0 + \sum_{n=1}^{N}\beta t_n\phi(\mathbf{x}_n)\right) - \frac{1}{2}\left(\sum_{n=1}^{N}\beta t_n^2 + \mathbf{m}_0^{\mathrm{T}}\mathbf{S}_0^{-1}\mathbf{r}\right.
\end{aligned}$$

Completing the square is a common operation for Gaussian distributions, where given a quadratic form defining the exponent terms in a Gaussian distribution, and we seek to determine the corresponding mean and covariance. Such problems can be solved easily by noting that the exponent in a general Gaussian distribution $\mathcal{N}(\mathbf{x}|\mu, \Sigma)$ can be formulated as

$$-\frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}}\Sigma^{-1}(\mathbf{x}-\mu) = -\frac{1}{2}\mathbf{x}^{\mathrm{T}}\Sigma^{-1}\mathbf{x} + \mathbf{x}^{\mathrm{T}}\Sigma^{-1}\mu - \frac{1}{2}\mu^{\mathrm{T}}\Sigma^{-1}\mu$$

then, we can equate the matrix of coefficients in the second order term to the inverse covariance matrix Σ^{-1} and the coefficient of the linear term to $\Sigma^{-1}\mu$, in order to obtain μ.

Hence, by comparing the quadratic term to the standard Gaussian Distribution we obtain

$$\mathbf{S}_N^{-1} = \mathbf{S}_0^{-1} + \sum_{n=1}^{N} \beta\phi(\mathbf{x}_n)\phi(\mathbf{x}_n)^{\mathrm{T}} = \mathbf{S}_0^{-1} + \beta\Phi^{\mathrm{T}}\Phi$$

Then, by comparing the linear term we obtain

$$\mathbf{S}_N^{-1}\mathbf{m}_N = \mathbf{S}_0^{-1}\mathbf{m}_0 + \sum_{n=1}^{N} \beta t_n \phi(\mathbf{x}_n) = \mathbf{S}_0^{-1}\mathbf{m}_0 + \beta\Phi^{\mathrm{T}}\mathtt{t} \Leftrightarrow$$
$$\mathbf{m}_N = \mathbf{S}_N(\mathbf{S}_0^{-1}\mathbf{m}_0 + \beta\Phi^{\mathrm{T}}\mathtt{t})$$

Therefore, the posterior distribution is given by

$$p(\mathbf{w}|\mathbf{X}, \mathtt{t}) = \mathcal{N}(\mathbf{w}|\mathbf{m}_N, \mathbf{S}_N)$$

where

$$\mathbf{m}_N = \mathbf{S}_N(\mathbf{S}_0^{-1}\mathbf{m}_0 + \beta\Phi^{\mathrm{T}}\mathtt{t})$$
$$\mathbf{S}_N^{-1} = \mathbf{S}_0^{-1} + \beta\Phi^{\mathrm{T}}\Phi$$

Consider now, for the sake of simplicity, a zero-mean isotropic Gaussian prior governed by a single parameter α so that,

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I})$$

Then, the corresponding posterior distribution over $\mathbf{w}$ becomes

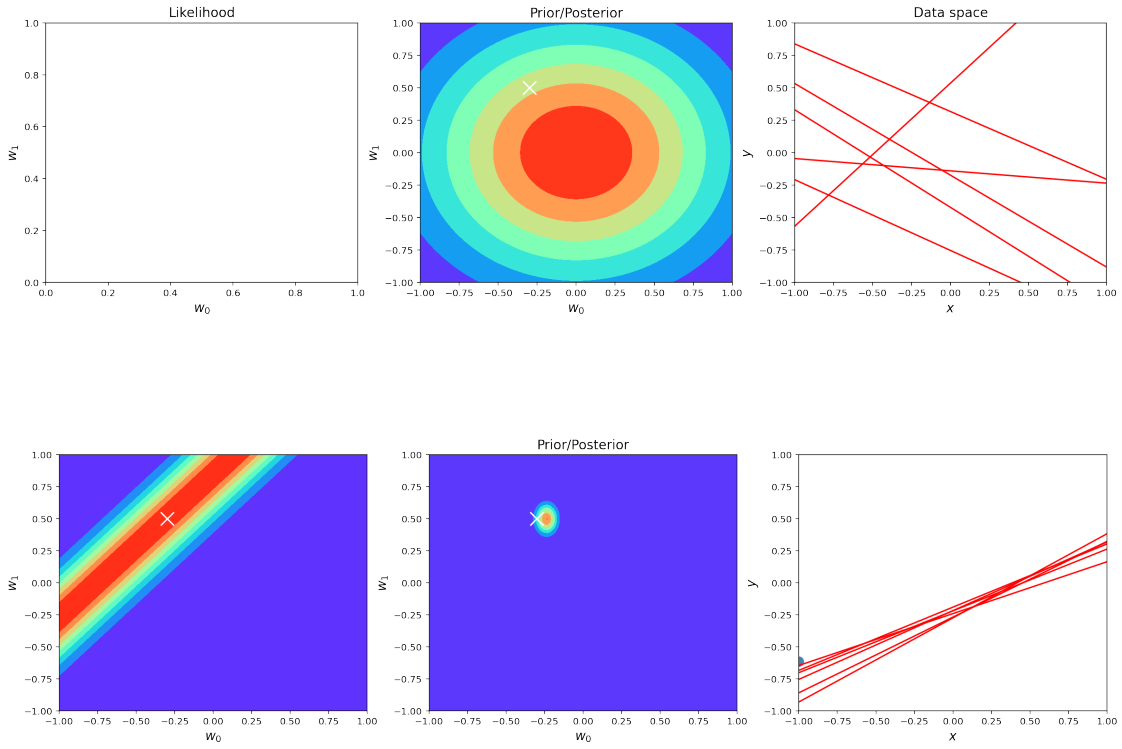
$$\mathbf{m}_N = \beta\mathbf{S}_N\Phi^{\mathrm{T}}\mathtt{t}$$
$$\mathbf{S}_N^{-1} = \alpha\mathbf{I} + \beta\Phi^{\mathrm{T}}\Phi$$

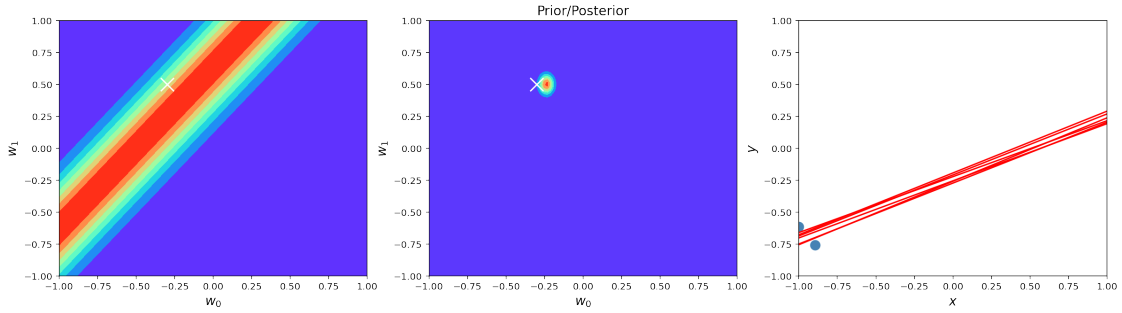
For this particular choice of prior, the maximization of the log of the posterior is equivalent to the minimization of the regularized sum-of-squares error function.

[6]:
```
alpha = 2.
beta = 25.
N = 20
f = lambda x: 0.5 * x - 0.3

x_train, y_train = generate_toy_data(f, N, 0.2, (-1, 1))
x_train_linear = PolynomialFeature(degree=1).transform(x_train)

w0, w1 = np.meshgrid(np.linspace(-1, 1, 100), np.linspace(-1, 1, 100))
w = np.array([np.ravel(w0), np.ravel(w1)])

# Prior for the parameters (assuming zero-mean isotropic Gaussian)
```

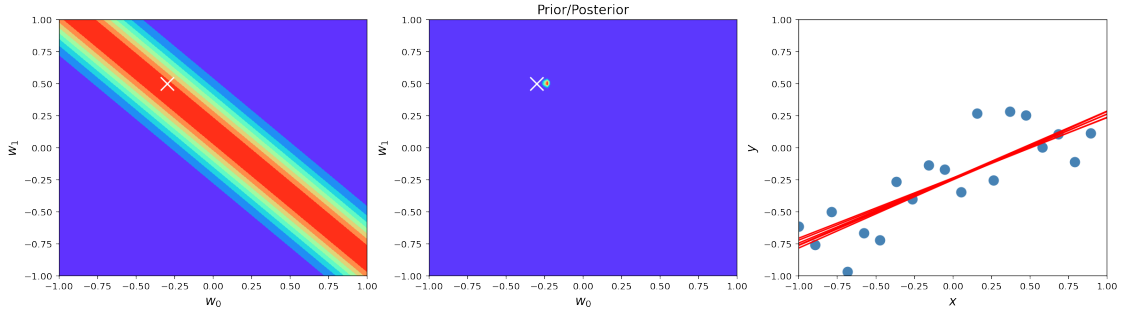
```
prior_posterior = MultivariateGaussian(mu=np.zeros((2, 1)), cov=(1 / alpha) *␣
 ↪np.eye(2))

for i, x in enumerate(x_train_linear):

    # compute the posterior according to (3.50) and (3.51)
    if i > 0:
        prev_precision = np.linalg.inv(prior_posterior.cov)
        cur_precision = prev_precision + beta * x_train_linear.T @␣
 ↪x_train_linear
        cur_cov = np.linalg.inv(cur_precision)
        cur_mean = cur_cov @ ((prev_precision @ prior_posterior.mu).T + beta *␣
 ↪x_train_linear.T @ y_train).reshape(-1, 1)

        prior_posterior = MultivariateGaussian(mu=cur_mean, cov=cur_cov)

    # create figures only for the three starting iterations and the last one
    if i < 3 or i == N - 1:

        plt.figure(figsize=(20, 5))

        # likelihood
        plt.subplot(1, 3, 1)
        if i == 0:
            plt.title('Likelihood', fontsize=14)
        else:
            likelihood = Gaussian(var=1 / beta).pdf(y_train[i])
            z = likelihood.pdf(mu=w.T @ x).reshape(w0.shape)
            plt.contourf(w0, w1, z, cmap='rainbow')
            plt.scatter(-0.3, 0.5, s=200, marker="x", color='white') # optimal␣
 ↪parameter vector

        plt.xlabel('$w_0$', fontsize=14)
        plt.ylabel('$w_1$', fontsize=14)

        # prior/posterior
        plt.subplot(1, 3, 2)
        plt.title('Prior/Posterior', fontsize=14)
        z = np.diag(prior_posterior.pdf(w)).reshape(w0.shape)
        plt.contourf(w0, w1, z, cmap='rainbow')
        plt.scatter(-0.3, 0.5, s=200, marker="x", color='white') # optimal␣
 ↪parameter vector

        plt.xlabel('$w_0$', fontsize=14)
        plt.ylabel('$w_1$', fontsize=14)

        # data space
```

```
        plt.subplot(1, 3, 3)
        if i == 0:
            plt.title('Data space', fontsize=14)
        else:
            plt.scatter(x_train[:i], y_train[:i], s=100, color='steelblue')

        w_sample = prior_posterior.draw(6)
        y_sample = x_train_linear @ w_sample.T
        plt.plot(x_train, y_sample, color='red')

        plt.xlim(-1, 1); plt.ylim(-1, 1)
        plt.xlabel('$x$', fontsize=14)
        plt.ylabel('$y$', fontsize=14)

        plt.show()
```






```
/Users/vagmcs/Work/dev/personal/prml/prml/distribution/multivariate_gaussian.py:
98: RuntimeWarning: overflow encountered in exp
  * np.exp(-0.5 * (np.linalg.solve(self.cov, d).T.dot(d)))
/Users/vagmcs/Work/dev/personal/prml/prml/distribution/multivariate_gaussian.py:
96: RuntimeWarning: overflow encountered in multiply
  1
```



3.3.2 Predictive distribution

In practice however, our goal is make predictions of t for unseen values of $\mathbf{x}_{\text{unseen}}$, and thus, we are not actually interested in the value of $\mathbf{w}$ itself. To that end, we evaluate the *predictive distribution* given by

$$p(t|\mathbf{x}_{\text{unseen}}, \mathsf{t}, \mathbf{X}, \alpha, \beta) = \int p(t|\mathbf{x}_{\text{unseen}}, \mathbf{w}, \beta) p(\mathbf{w}|\mathsf{t}, \mathbf{X}, \alpha, \beta) \mathrm{d}\mathbf{w}$$

Note that the predictive distribution involves the convolution of the conditional Gaussian distribution of the target variable and the posterior weight Gaussian distribution.

$$\begin{aligned} p(t|\mathbf{x}, \mathbf{w}, \beta) &= \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}) = \mathcal{N}(t|\phi(\mathbf{x})^{\mathrm{T}}\mathbf{w}, \beta^{-1}) \\ p(\mathbf{w}|\mathsf{t}, \mathbf{X}, \alpha, \beta) &= \mathcal{N}(\mathbf{w}|\mathbf{m}_N, \mathbf{S}_N) \end{aligned}$$

Taking advantage of **(2.113)**, **(2.114)** and **(2.115)**, we can obtain

$$p(t|\mathbf{x}_{\text{unseen}}, \mathbf{t}, \mathbf{X}, \alpha, \beta) = \mathcal{N}(t|\mathbf{m}_N^{\mathrm{T}}\phi(\mathbf{x}_{\text{unseen}}), \sigma_N^2(\mathbf{x}_{\text{unseen}}))$$

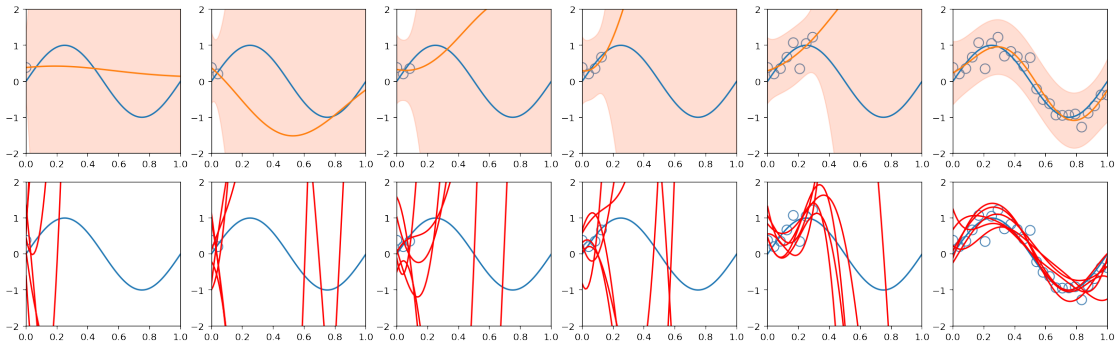
where the variance $\sigma_N^2(\mathbf{x})$ of the predictive distribution is given by

$$\sigma_N^2(\mathbf{x}) = \frac{1}{\beta} + \phi(\mathbf{x})^{\mathrm{T}}\mathbf{S}_N\phi(\mathbf{x})$$

The first term represents the noise on the data whereas the second term reflects the uncertainty associated with the parameters $\mathbf{w}$.

[7]:
```
N = 25
sinusoidal = lambda x: np.sin(2 * np.pi * x)

x_train, y_train = generate_toy_data(sinusoidal, N, 0.25)
x_test = np.linspace(0, 1, 100)
y_test = sinusoidal(x_test)

feature = GaussianFeature(np.linspace(0, 1, 9), 0.1)
X_train = feature.transform(x_train)
X_test = feature.transform(x_test)

fig_idx = 1
plt.figure(figsize=(20, 6))
for i, x in enumerate(X_train):
    if i < 4 or i == 7 or i == N - 1:
        model = BayesianRegression(alpha=1e-3, beta=2.)
        model.fit(X_train[:i+1], y_train[:i+1])
        y, y_std = model.predict(X_test)

        plt.subplot(2, 6, fig_idx); fig_idx += 1
        plt.scatter(x_train[:i+1], y_train[:i+1], s=100, facecolor="none",␣
 ↪edgecolor="steelblue")
        plt.plot(x_test, y_test)
        plt.plot(x_test, y)
        plt.fill_between(x_test, y - y_std, y + y_std, color="coral", alpha=0.
 ↪25)
        plt.xlim(0, 1); plt.ylim(-2, 2)


for i, x in enumerate(X_train):
    if i < 4 or i == 7 or i == N - 1:
        model = BayesianRegression(alpha=1e-3, beta=2.)
        model.fit(X_train[:i+1], y_train[:i+1])
        w_samples = model.draw(6)

        plt.subplot(2, 6, fig_idx); fig_idx += 1
```

```
        plt.scatter(x_train[:i+1], y_train[:i+1], s=100, facecolor="none",␣
↪edgecolor="steelblue")
        plt.plot(x_test, y_test)
        plt.plot(x_test, X_test @ w_samples.T, color='red')
        plt.xlim(0, 1); plt.ylim(-2, 2)

plt.show()
```



3.3.3 Equivalent kernel

The posterior mean solution for the linear basis function model has an interesting interpretation. If we substitute the posterior mean solution $\mathbf{m}_N$, given by **(3.53)**, into the expression **(3.3)**, we note that the predictive mean can be expressed in the form

$$y(\mathbf{x}, \mathbf{m}_N) = \mathbf{m}_N^{\mathrm{T}} \phi(\mathbf{x}) = \beta \phi(\mathbf{x})^{\mathrm{T}} \mathbf{S}_N \mathbf{\Phi}^{\mathrm{T}} \mathbf{t} = \sum_{n=1}^{N} \beta \phi(\mathbf{x})^{\mathrm{T}} \mathbf{S}_N \phi(\mathbf{x}_n) t_n$$

Thus the mean of the predictive distribution at point $\mathbf{x}$ is given by a linear combination of the training data set target variables t_n.

$$y(\mathbf{x}, \mathbf{m}_N) = \sum_{n=1}^{N} k(\mathbf{x}, \mathbf{x}_n) t_n$$

where the function

$$k(\mathbf{x}, \mathbf{x}') = \beta \phi(\mathbf{x})^{\mathrm{T}} \mathbf{S}_N \phi(\mathbf{x}')$$

is known as the *smoother matrix* or the *equivalent kernel*. Regression functions that make predictions by taking linear combinations of the training set target values are known as *linear smoothers*.

Further insight into the equivalent kernel can be obtained by considering the covariance between $y(\mathbf{x})$ and $y(\mathbf{x}')$,

$$\text{cov}[y(\mathbf{x}), y(\mathbf{x}')] = \text{cov}[\phi(\mathbf{x})^{\mathrm{T}}\mathbf{w}, \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x})] = \phi(\mathbf{x})^{\mathrm{T}}\mathbf{S}_N\phi(\mathbf{x}') = \beta^{-1}k(\mathbf{x}, \mathbf{x}')$$

Therefore, the predictive mean at nearby points is highly correlated, whereas for more distant pairs of points the correlation is smaller.

!!! This formulation of linear regression suggests an alternative approach to regression as follows. Instead of introducing a set of basis functions ϕ_j to derive an equivalent kernel, we may instead define the kernel $k(\mathbf{x}, \mathbf{x}')$ directly and use the it to make predictions, given an observed training set. This leads to a practical framework called *Gaussian processes*.

3.5 The Evidence Approximation

In a fully Bayesian treatment of the linear basis function model one can introduce prior distributions over the hyperparameters α and β and make predictions by marginalizing over these hyperparameters in addition to the model parameters $\mathbf{w}$. However, the complete marginalization over these variables is analytically intractable. A useful approximation is to set the hyperparameters to specific values by maximizing the marginal likelihood obtained by integrating over the model parameters $\mathbf{w}$. This framework is known in statistics as *empirical bayes* or *type 2 maximum likelihood* or *generalized maximum likelihood*. In machine learning is also called *evidence approximation*.

The predictive distribution is obtained by marginalizing over $\mathbf{w}$, α and β so that

$$p(t|\tilde{\mathbf{x}}, \mathtt{t}, \mathbf{X}) = \int\int\int p(t|\mathbf{w}, \tilde{\mathbf{x}}, \beta)p(\mathbf{w}|\mathtt{t}, \mathbf{X}, \alpha, \beta)p(\alpha, \beta|\mathtt{t}, \mathbf{X})\mathrm{d}\mathbf{w}\mathrm{d}\alpha\mathrm{d}\beta$$

Assuming that the posterior distribution $p(\alpha, \beta|\mathtt{t}, \mathbf{X})$ is sharply peaked around some values $\hat{\alpha}$ and $\hat{\beta}$, then the predictive distribution is obtained by fixing α and β to these values and marginalizing over $\mathbf{w}$

$$p(t|\tilde{\mathbf{x}}, \mathtt{t}, \mathbf{X}) \approx p(t|\tilde{\mathbf{x}}, \mathtt{t}, \mathbf{X}, \hat{\alpha}, \hat{\beta}) = \int p(t|\tilde{\mathbf{x}}, \mathbf{w}, \hat{\beta})p(\mathbf{w}|\mathtt{t}, \mathbf{X}, \hat{\alpha}, \hat{\beta})\mathrm{d}\mathbf{w}$$

in which case we arrive at the simpler predictive distribution defined by (3.57).

The posterior distribution for α and β is given by

$$p(\alpha, \beta|\mathtt{t}, \mathbf{X}) \propto p(\mathtt{t}|\mathbf{X}, \alpha, \beta)p(\alpha, \beta)$$

Further assuming that the prior is relatively flat, that is, our prior belief is that different values of α and β are somewhat equiprobable. Then, the sharply peaked area assumed for the posterior ($\hat{\alpha}$ and $\hat{\beta}$) should be found by maximizing the marginal likelihood function $p(\mathtt{t}|\mathbf{X}, \alpha, \beta)$.

3.5.1 Evaluation of the evidence function

The marginal likelihood function $p(\mathtt{t}|\mathbf{X}, \alpha, \beta)$ is obtained by integrating over the model parameters $\mathbf{w}$, so that,

$$p(\mathtt{t}|\mathbf{X}, \alpha, \beta) = \int p(\mathtt{t}|\mathbf{X}, \mathbf{w}, \beta)p(\mathbf{w}|\alpha)\mathrm{d}\mathbf{w}$$

From **(3.10)** we have that

$$\begin{aligned} p(\mathsf{t}|\mathbf{X}, \mathbf{w}, \beta) &= \prod_{n=1}^{N} \mathcal{N}(t_n|\mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n), \beta^{-1}) \\ &= \prod_{n=1}^{N} \frac{1}{(2\pi\beta^{-1})^{1/2}} \exp\{-\frac{1}{2\beta^{-1}}(t_n - \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n))^2\} \\ &= N\left(\frac{\beta}{2\pi}\right)^{1/2} \exp\{\sum_{n=1}^{N} -\frac{1}{2\beta^{-1}}(t_n - \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n))^2\} \\ &= \left(\frac{\beta}{2\pi}\right)^{N/2} \exp\{-\frac{\beta}{2}\|\mathsf{t} - \Phi\mathbf{w}\|^2\} \end{aligned}$$

From **(3.52)** we have that

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I}) = \frac{\alpha^{M/2}}{(2\pi)^{M/2}} \exp\{-\frac{\alpha}{2}\|\mathbf{w}\|^2\}$$

Substituting both these quantities back into the integral of the marginal likelihood, we have

$$p(\mathsf{t}|\mathbf{X}, \alpha, \beta) = \left(\frac{\beta}{2\pi}\right)^{N/2} \left(\frac{\alpha}{2\pi}\right)^{M/2} \int \exp\{-\frac{\beta}{2}\|\mathsf{t} - \Phi\mathbf{w}\|^2 - \frac{\alpha}{2}\|\mathbf{w}\|^2\}\mathrm{d}\mathbf{w}$$

Using **(3.25)** and **(3.26)** we obtain

$$p(\mathsf{t}|\mathbf{X}, \alpha, \beta) = \left(\frac{\beta}{2\pi}\right)^{N/2} \left(\frac{\alpha}{2\pi}\right)^{M/2} \int \exp\{-E(\mathbf{w})\}\mathrm{d}\mathbf{w}$$

where

$$E(\mathbf{w}) = \beta E_D(\mathbf{w}) + \alpha E_W(\mathbf{w})$$

Completing the square over $\mathbf{w}$, we obtain

$$\begin{aligned} E(\mathbf{w}) &= \frac{\beta}{2}\|\mathsf{t} - \Phi\mathbf{w}\|^2 + \frac{\alpha}{2}\|\mathbf{w}\|^2 \\ &= \frac{\beta}{2}(\mathsf{t}^{\mathrm{T}}\mathsf{t} - 2\mathsf{t}^{\mathrm{T}}\Phi\mathbf{w} + \mathbf{w}^{\mathrm{T}}\Phi^{\mathrm{T}}\Phi\mathbf{w}) + \frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \\ &= \frac{1}{2}(\mathbf{w}^{\mathrm{T}}(\alpha\mathbf{I} + \beta\Phi^{\mathrm{T}}\Phi)\mathbf{w} - 2\beta\mathsf{t}^{\mathrm{T}}\Phi\mathbf{w} + \beta\mathsf{t}^{\mathrm{T}}\mathsf{t}) \end{aligned}$$

where

$$
\begin{aligned}
\mathbf{A} = \mathbf{S}_N^{-1} &= \alpha\mathbf{I} + \beta\Phi^{\mathrm{T}}\Phi \\
\mathbf{m}_N^{\mathrm{T}}\mathbf{S}_N^{-1} &= \beta\mathsf{t}^{\mathrm{T}}\Phi \Leftrightarrow \\
\mathbf{m}_N^{\mathrm{T}} &= \beta\mathbf{S}_N\mathsf{t}^{\mathrm{T}}\Phi \Leftrightarrow \\
\mathbf{m}_N &= \beta\mathbf{S}_N\Phi^{\mathrm{T}}\mathsf{t} = \beta\mathbf{A}^{-1}\Phi^{\mathrm{T}}\mathsf{t}
\end{aligned}
$$

By exploiting the fact $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ and adding $\mathbf{0} = \mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N - \mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N$, we can further derive that

$$
\begin{aligned}
E(\mathbf{w}) &= \frac{1}{2}(\mathbf{w}^{\mathrm{T}}(\alpha\mathbf{I} + \beta\Phi^{\mathrm{T}}\Phi)\mathbf{w} - 2\beta\mathsf{t}^{\mathrm{T}}\Phi\mathbf{w} + \beta\mathsf{t}^{\mathrm{T}}\mathsf{t}) \\
&= \frac{1}{2}(\mathbf{w}^{\mathrm{T}}\mathbf{A}\mathbf{w} - 2\beta\mathsf{t}^{\mathrm{T}}\Phi\mathbf{A}^{-1}\mathbf{A}\mathbf{w} + \beta\mathsf{t}^{\mathrm{T}}\mathsf{t}) \\
&= \frac{1}{2}(\mathbf{w}^{\mathrm{T}}\mathbf{A}\mathbf{w} - 2\mathbf{m}_N\mathbf{A}\mathbf{w} + \beta\mathsf{t}^{\mathrm{T}}\mathsf{t} + \mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N - \mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N) \\
&= \frac{1}{2}(\beta\mathsf{t}^{\mathrm{T}}\mathsf{t} - \mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N) + \frac{1}{2}(\mathbf{w} - \mathbf{m}_N)^{\mathrm{T}}\mathbf{A}(\mathbf{w} - \mathbf{m}_N)
\end{aligned}
$$

At this point note that the first term is independent of the model parameters $\mathbf{w}$ and the second term is an exponent of a Gaussian distribution over the model parameters. Therefore the integral over $\mathbf{w}$ of $p(\mathsf{t}|\mathbf{X}, \alpha, \beta)$ is given by

$$
p(\mathsf{t}|\mathbf{X}, \alpha, \beta) = \left(\frac{\beta}{2\pi}\right)^{N/2}\left(\frac{\alpha}{2\pi}\right)^{M/2}\exp\{-\frac{1}{2}(\beta\mathsf{t}^{\mathrm{T}}\mathsf{t} - \mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N)\}\int\exp\{-\frac{1}{2}(\mathbf{w} - \mathbf{m}_N)^{\mathrm{T}}\mathbf{A}(\mathbf{w} - \mathbf{m}_N)\}\mathrm{d}\mathbf{w}
$$

However, based on the standard form of a multivariate normal distribution, we know that

$$
\begin{aligned}
&\int\frac{1}{(2\pi)^{M/2}}\frac{1}{|\mathbf{A}|^{1/2}}\exp\{-\frac{1}{2}(\mathbf{w} - \mathbf{m}_N)^{\mathrm{T}}\mathbf{A}(\mathbf{w} - \mathbf{m}_N)\}\mathrm{d}\mathbf{w} = 1 \Leftrightarrow \\
&\int\exp\{-\frac{1}{2}(\mathbf{w} - \mathbf{m}_N)^{\mathrm{T}}\mathbf{A}(\mathbf{w} - \mathbf{m}_N)\}\mathrm{d}\mathbf{w} = (2\pi)^{M/2}|\mathbf{A}|^{-1/2}
\end{aligned}
$$

Thus,

$$
p(\mathsf{t}|\mathbf{X}, \alpha, \beta) = \left(\frac{\beta}{2\pi}\right)^{N/2}\left(\frac{\alpha}{2\pi}\right)^{M/2}\exp\{-\frac{1}{2}(\beta\mathsf{t}^{\mathrm{T}}\mathsf{t} - \mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N)\}(2\pi)^{M/2}|\mathbf{A}|^{-1/2}
$$

Thus, the log of the marginal likelihood is given by

$$
\begin{aligned}
\log p(\mathsf{t}|\mathbf{X}, \alpha, \beta) &= \frac{N}{2}\log\left(\frac{\beta}{2\pi}\right) + \frac{M}{2}\log\left(\frac{\alpha}{2\pi}\right) - \frac{1}{2}(\beta\mathsf{t}^{\mathrm{T}}\mathsf{t} - \mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N) + \frac{M}{2}\log(2\pi) - \frac{1}{2}|\mathbf{A}| \\
&= \frac{M}{2}\log\alpha + \frac{N}{2}\log\beta - \frac{1}{2}(\beta\mathsf{t}^{\mathrm{T}}\mathsf{t} - \mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N) - \frac{1}{2}|\mathbf{A}| - \frac{N}{2}\log(2\pi)
\end{aligned}
$$

As a final step we are about to show that $\frac{1}{2}(\beta\mathsf{t}^{\mathrm{T}}\mathsf{t} - \mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N) = E(\mathbf{w})$

$$\begin{aligned}
\frac{1}{2}(\beta \mathsf{t}^{\mathrm{T}}\mathsf{t} - \mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N) &= \frac{1}{2}(\beta \mathsf{t}^{\mathrm{T}}\mathsf{t} - 2\mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N + \mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N) \\
&\overset{(3.81)}{=} \frac{1}{2}(\beta \mathsf{t}^{\mathrm{T}}\mathsf{t} - 2\mathbf{m}_N^{\mathrm{T}}\mathbf{A}\mathbf{m}_N - \mathbf{m}_N^{\mathrm{T}}(\alpha\mathbf{I} + \beta^{\mathrm{T}})\mathbf{m}_N) \\
&\overset{(3.84)}{=} \frac{1}{2}(\beta \mathsf{t}^{\mathrm{T}}\mathsf{t} - 2\beta \mathsf{t}^{\mathrm{T}}\mathbf{A}^{-1}\mathbf{A}\mathbf{m}_N - \mathbf{m}_N^{\mathrm{T}}(\alpha\mathbf{I} + \beta^{\mathrm{T}})\mathbf{m}_N) \\
&= \frac{1}{2}(\beta \mathsf{t}^{\mathrm{T}}\mathsf{t} - 2\beta \mathsf{t}^{\mathrm{T}}\mathbf{m}_N - \mathbf{m}_N^{\mathrm{T}}(\beta^{\mathrm{T}})\mathbf{m}_N) + \frac{\alpha}{2}\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N \\
&= \frac{1}{2}\|\mathsf{t} - \mathbf{m}_N\|^2 + \frac{\alpha}{2}\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N = E(\mathbf{w})
\end{aligned}$$

3.5.2 Maximizing the evidence function

Lets consider the maximization of $p(\mathsf{t}|\mathbf{X}, \alpha, \beta)$ over α,

$$\begin{aligned}
\frac{d}{d\alpha}p(\mathsf{t}|\mathbf{X}, \alpha, \beta) &= \frac{d}{d\alpha}\frac{M}{2}\log\alpha - \frac{d}{d\alpha}E(\mathbf{w}) - \frac{d}{d\alpha}\frac{1}{2}\log|\mathbf{A}| \\
&= \frac{M}{2\alpha} - \frac{1}{2}\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N - \frac{1}{2}\frac{d}{d\alpha}\ln|\mathbf{A}|
\end{aligned}$$

In order to find the derivative of $\ln|\mathbf{A}|$, we use the eigenvector equation of **(C.29)**

$$\begin{aligned}
(\beta^{\mathrm{T}})\mathbf{u}_i = \lambda_i\mathbf{u}_i &\overset{(C.30)}{\Leftrightarrow} |\beta^{\mathrm{T}} - \lambda_i\mathbf{I}| = \mathbf{0} \\
&\overset{(3.81)}{\Leftrightarrow} |\mathbf{A} - \alpha\mathbf{I} - \lambda_i\mathbf{I}| = \mathbf{0} \\
&\Leftrightarrow |\mathbf{A} - (\lambda_i + \alpha)\mathbf{I}| = \mathbf{0}
\end{aligned}$$

Therefore, we can derive that $\mathbf{A}$ has eigenvalues $\lambda_i + \alpha$. In conclusion, the derivative of the term involving $\ln|\mathbf{A}|$ is given by

$$\frac{d}{d\alpha}\ln|\mathbf{A}| \overset{(C.47)}{=} \frac{d}{d\alpha}\ln\prod_{i=1}^{M}(\lambda_i + \alpha) = \frac{d}{d\alpha}\sum_{i=1}^{M}\ln(\lambda_i + \alpha) = \sum_{i=1}^{M}\frac{1}{\lambda_i + \alpha}$$

Thus, the resulting solution for α, as presented in eq. **(3.92)**, is given by

$$
\begin{aligned}
0 &= \frac{M}{2\alpha} - \frac{1}{2}\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N - \frac{1}{2}\sum_{i=1}^{M}\frac{1}{\lambda_i+\alpha} \Leftrightarrow \\
\frac{1}{2}\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N &= \frac{M}{2\alpha} - \frac{1}{2}\sum_{i=1}^{M}\frac{1}{\lambda_i+\alpha} \overset{\times 2\alpha}{\Leftrightarrow} \\
\alpha\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N &= M - \alpha\sum_{i=1}^{M}\frac{1}{\lambda_i+\alpha} \Leftrightarrow \\
\alpha\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N &= M\frac{\lambda_i+\alpha}{\lambda_i+\alpha} - \sum_{i=1}^{M}\frac{\alpha}{\lambda_i+\alpha} \Leftrightarrow \\
\alpha\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N &= \sum_{i=1}^{M}\frac{\lambda_i}{\lambda_i+\alpha} = \gamma \Leftrightarrow \\
\alpha &= \frac{\gamma}{\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N}
\end{aligned}
$$

Note that this is an implicit solution for α, since both γ and $\mathbf{m}_N$ depend on α. Thus, we have to use an iterative procedure to estimate α by making an starting choice for the value of α, computing $\mathbf{m}_N$, evaluating γ **(3.91)** and re-estimating α **(3.92)**, until convergence. It should be *emphasized* that the value of α can be determined purely by looking at the training data. No independent dataset is required in order to optimize model complexity.

The maximization of $p(\mathtt{t}|\mathbf{X},\alpha,\beta)$ over β is given by

$$
\begin{aligned}
\frac{d}{d\beta}p(\mathtt{t}|\mathbf{X},\alpha,\beta) &= \frac{d}{d\beta}\frac{N}{2}\log\beta - \frac{d}{d\beta}E(\mathbf{w}) - \frac{d}{d\beta}\frac{1}{2}\log|\mathbf{A}| \\
&= \frac{N}{2\beta} - \frac{d}{d\beta}E(\mathbf{w}) - \frac{1}{2}\frac{d}{d\beta}\ln|\mathbf{A}|
\end{aligned}
$$

Lets take a closer look to the second term,

$$
\begin{aligned}
\frac{d}{d\beta}E(\mathbf{w}) &= \frac{d}{d\beta}\frac{\beta}{2}\|\mathrm{t} - \mathbf{m}_N\|^2 + \frac{d}{d\beta}\frac{\alpha}{2}\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N \\
&\overset{\text{product rule}}{=} \frac{1}{2}\|\mathrm{t} - \mathbf{m}_N\|^2 + \frac{\beta}{2}\frac{d}{d\beta}\|\mathrm{t} - \mathbf{m}_N\|^2 + \frac{d}{d\beta}\frac{\alpha}{2}\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N \\
&\overset{\times\, d\mathbf{m}_N/d\mathbf{m}_N}{=} \frac{1}{2}\|\mathrm{t} - \mathbf{m}_N\|^2 + \left(\frac{\beta}{2}\frac{d}{d\mathbf{m}_N}\|\mathrm{t} - \mathbf{m}_N\|^2 + \frac{d}{d\mathbf{m}_N}\frac{\alpha}{2}\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N\right)\frac{d\mathbf{m}_N}{d\beta} \\
&= \frac{1}{2}\|\mathrm{t} - \mathbf{m}_N\|^2 + \left(\frac{\beta}{2}(-2\,^{\mathrm{T}}(\mathrm{t} - \mathbf{m}_N)) + \frac{\alpha}{2}2\mathbf{m}_N\right)\frac{d\mathbf{m}_N}{d\beta} \\
&= \frac{1}{2}\|\mathrm{t} - \mathbf{m}_N\|^2 + \left(-\beta\,^{\mathrm{T}}(\mathrm{t} - \mathbf{m}_N) + \alpha\mathbf{m}_N\right)\frac{d\mathbf{m}_N}{d\beta} \\
&= \frac{1}{2}\|\mathrm{t} - \mathbf{m}_N\|^2 + \left(-\beta\,^{\mathrm{T}}\mathrm{t} - (\alpha\mathbf{I} + {}^{\mathrm{T}})\mathbf{m}_N\right)\frac{d\mathbf{m}_N}{d\beta} \\
&\overset{(3.81)}{=} \frac{1}{2}\|\mathrm{t} - \mathbf{m}_N\|^2 + (-\beta\,^{\mathrm{T}}\mathrm{t} - \mathbf{A}\mathbf{m}_N)\frac{d\mathbf{m}_N}{d\beta} \\
&\overset{(3.84)}{=} \frac{1}{2}\|\mathrm{t} - \mathbf{m}_N\|^2 = \frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{m}_N^{\mathrm{T}}(\mathbf{x}_n)\}^2
\end{aligned}
$$

The last term involving the derivative of $\ln|\mathbf{A}|$ becomes

$$
\frac{d}{d\beta}\ln|\mathbf{A}| = \frac{d}{d\beta}\sum_{i=1}^{M}\ln(\lambda_i + \alpha) = \sum_{i=1}^{M}\frac{1}{\lambda_i + \alpha}\frac{d}{d\beta}\lambda_i = \frac{1}{\beta}\sum_{i=1}^{M}\frac{\lambda_i}{\lambda_i + \alpha} = \frac{\gamma}{\beta}
$$

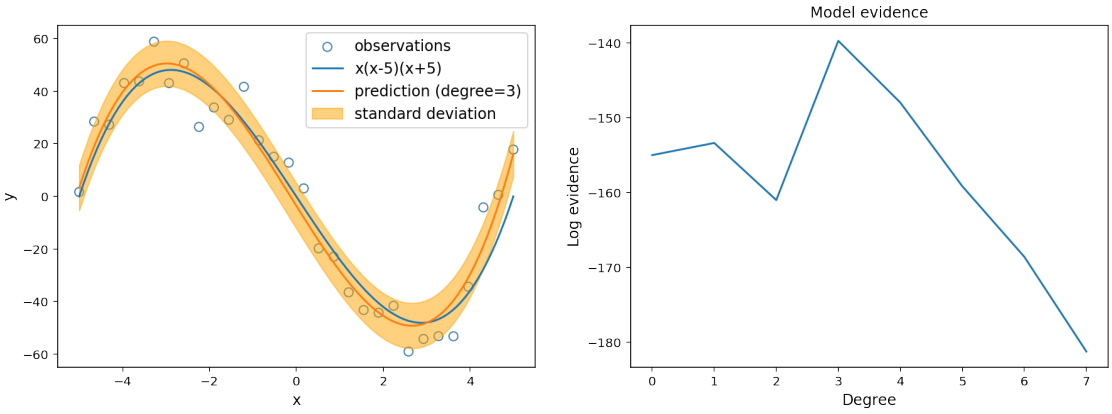
Finally, if we combine all these expressions together, we obtain

$$
\begin{aligned}
0 &= \frac{\mathrm{N}}{2\beta} - \frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{m}_N^{\mathrm{T}}\phi(\mathbf{x}_n)\}^2 - \frac{\gamma}{2\beta} \Leftrightarrow \\
\frac{1}{\beta} &= \frac{1}{N - \gamma}\sum_{n=1}^{N}\{t_n - \mathbf{m}_N^{\mathrm{T}}\phi(\mathbf{x}_n)\}^2
\end{aligned}
$$

As expected, the solution for β is also implicit. If both α and β are to be determined, then their values can be re-estimated together after each update of γ.

[8]:
```
cubic = lambda x: x * (x - 5) * (x + 5)
x_train, y_train = generate_toy_data(cubic, 30, 10, [-5, 5])
x_test = np.linspace(-5, 5, 100)

models = []
evidences = []
for i in range(8):
    feature = PolynomialFeature(degree=i)
    X_train = feature.transform(x_train)
    model = EvidenceApproximation(alpha=100., beta=100.)
    model.fit(X_train, y_train, n_iter=100)
    evidences.append(model.log_evidence(X_train, y_train))
```

```
    models.append(model)

# select the best performing degree (the one having the highest evidence)
degree = np.nanargmax(evidences)
regression = models[degree]

X_test = PolynomialFeature(degree=int(degree)).transform(x_test)
y, y_std = regression.predict(X_test)

plt.figure(figsize=(15, 5))

plt.subplot(1, 2, 1)
plt.scatter(x_train, y_train, s=50, facecolor="none", edgecolor="steelblue",␣
 ↪label="observations")
plt.plot(x_test, cubic(x_test), label="x(x-5)(x+5)")
plt.plot(x_test, y, label=f"prediction (degree={degree})")
plt.fill_between(x_test, y - y_std, y + y_std, alpha=0.5, label="standard␣
 ↪deviation", color="orange")
plt.xlabel("x", fontsize=12)
plt.ylabel("y", fontsize=12)
plt.legend(fontsize=12)

plt.subplot(1, 2, 2)
plt.plot(evidences)
plt.title("Model evidence", fontsize=12)
plt.xlabel("Degree", fontsize=12)
plt.ylabel("Log evidence", fontsize=12)
plt.show()
```



The left figure presents the fitted function estimated by evidence approximation for degree 3. The figure in the right depicts the log evidence of each polynomial feature degree. Note that the highest evidence is achieved for polynomial feature of degree 3.

4. Linear Models for Classification

Table of Contents

[1]:
```
import math
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from prml.preprocessing import LinearFeature
from prml.linear import LeastSquaresClassifier
from prml.linear import FisherLinearDiscriminant
from prml.linear import Perceptron
from prml.linear import GenerativeClassifier
from prml.linear import LogisticRegression
from prml.linear import SoftmaxRegression
from prml.distribution import Gaussian

# Set random seed to make deterministic
np.random.seed(0)

# Ignore zero divisions and computation involving NaN values.
np.seterr(divide = 'ignore', invalid='ignore')

# Enable higher resolution plots
%config InlineBackend.figure_format = 'retina'

# Enable autoreload all modules before executing code
%load_ext autoreload
%autoreload 2
```

The goal in classification is to take an input vector $\mathbf{x}$ and assign it to one of K discrete classes $\mathcal{C}_k$, where $k = 1 \ldots, K$. The input space is thereby divided into *decision regions* whose boundaries are called *decision boundaries* or *decision surfaces*. Linear models define decision surfaces as linear functions of the input vector $\mathbf{x}$ and hence are defined by $(D-1)$-dimensional hyperplanes inside the D-dimensional space. Datasets whose classes can be separated exactly by linear decision surfaces are called *linearly separable*.

There are three distinct approaches to the classification problem:

1. Discriminant functions that directly assign each input vector $\mathbf{x}$ to a class.

2. Models that directly learn the conditional probability $p(\mathcal{C}_k|\mathbf{x})$ using parametric modelling.
3. Generative approaches that model the class conditional density $p(\mathbf{x}|\mathcal{C}_k)$, and the prior probabilities $p(\mathcal{C}_k)$ for the classes. Then they derive the posterior using the Bayes theorem.

In the linear regression models, the model prediction $y(\mathbf{x}, \mathbf{w})$ was given by a linear function of the parameters $\mathbf{w}$. For classification problems, however, we wish to predict discrete class labels. To that end, we consider a generalization of the above model in which we transform the linear function using a nonlinear function $f(\cdot)$ so that

$$y(\mathbf{x}) = f(\mathbf{w}^T\mathbf{x} + w_0)$$

In machine learning, the function f is known as an *activation function*.

4.1 Discriminant Functions

A discriminant is a function that assigns one of K classes to an input vector $\mathbf{x}$. *Linear discriminants* define decision surfaces that are hyperplanes.

4.1.1 Two Classes

The simplest linear discriminant function is obtained by taking a linear function of the input vector so that,

$$y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$$

where $\mathbf{w}$ is a *weight vector* and w_0 is a *bias* (the negative of the bias is also called *threshold*). Then, an input $\mathbf{x}$ is assigned to a class $\mathcal{C}_1$ if $y(\mathbf{x}) \geq 0$ and to class $\mathcal{C}_2$ otherwise. Thus, the decision boundary is defined by $y(\mathbf{x}) = 0$.

Consider two points $\mathbf{x}_A$ and $\mathbf{x}_B$ onto the decision surface. Then, $y(\mathbf{x}_A) = y(\mathbf{x}_B) = 0 \Leftrightarrow \mathbf{w}^T(\mathbf{x}_A - \mathbf{x}_B) = 0$, which implies that the vector $\mathbf{w}$ is orthogonal to every vector lying in the decision surface as depicted below:

Note that for more than two classes ($K > 2$), a *one-vs-the-rest* classifier can be used in order to avoid regions of input space that are ambiguously classified. The linear function of each class takes the form $y_k(\mathbf{x}) = \mathbf{w}_k^T\mathbf{x} + w_{k0}$, and assigns a point $\mathbf{x}$ to class $\mathcal{C}_k$ if $y_k(\mathbf{x}) > y_j(\mathbf{x})\ \forall j \neq k$.

In the following section*s we explore three approaches to learning the parameters of linear discriminant functions:

1. Least squares
2. Fisher's linear discriminant
3. Perceptron algorithm

4.1.3 Least squares for classification

In Chapter 3, we minimized the sum-of-squared error function led to a closed-form solution for the parameter values. Can we apply the same principle to classification problems?

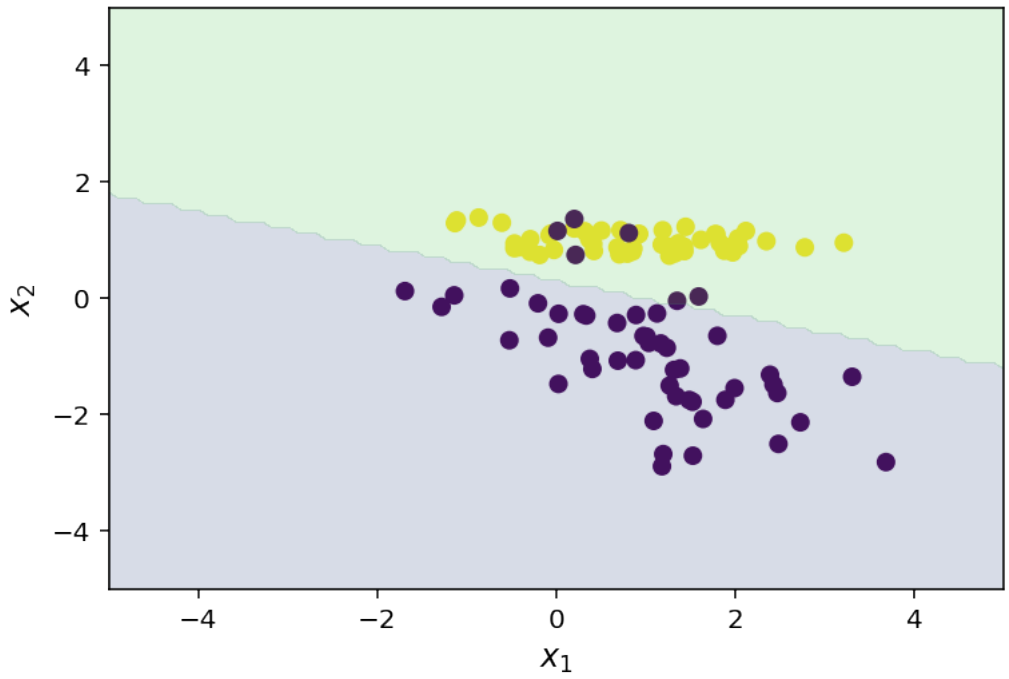
Consider a general classification problem having K classes, using a 1-of-K binary coding scheme or *one-hot* encoding for the target vector. Each class $\mathcal{C}_k$ is described by its own linear model y_k. We can group these models together using vector notation so that

$$\mathbf{y}(\mathbf{x}) = \tilde{\mathbf{W}}^T\tilde{\mathbf{x}}$$

where $\tilde{\mathbf{W}}$ is a matrix whose k^{th} column comprises the $D+1$-dimensional vector $\tilde{\mathbf{w}}_k = (w_{k0}, \mathbf{w}_k^T)^T$ and $\tilde{\mathbf{x}}$ is the augmented vector $(1, \tilde{\mathbf{x}}^T)^T$. The parameter matrix $\tilde{\mathbf{W}}$ is determined by minimizing the sum-of-squares error function, as presented in Chapter 3. Thus, the solution for $\tilde{\mathbf{W}}$ is obtained from

$$\tilde{\mathbf{W}} = (\tilde{\mathbf{X}}^T\tilde{\mathbf{X}})^{-1}\tilde{\mathbf{X}}^T\mathbf{T}$$

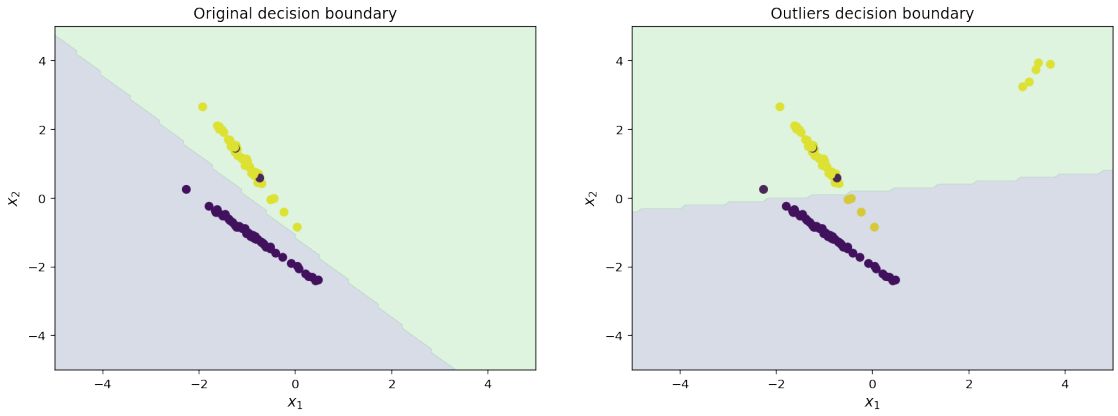
[2]:
```
# number of training points
N = 100

x_train, t = make_classification(
    n_features=2,
    n_informative=2,
    n_redundant=0,
    n_classes=2,
    n_clusters_per_class=1,
    n_samples=N
)

x1, x2 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
x_test = np.array([x1, x2]).reshape(2, -1).T

feature = LinearFeature()
x_train_linear = feature.transform(x_train)
x_test_linear = feature.transform(x_test)

model = LeastSquaresClassifier()
model.fit(x_train_linear, t)
predicted = model.predict(x_test_linear)

plt.scatter(x_train[:, 0], x_train[:, 1], c=t)
plt.contourf(x1, x2, predicted.reshape(100, 100), alpha=0.2, levels=np.
 ↪linspace(0, 1, 3))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.xlabel("$x_1$", fontsize=12); plt.ylabel("$x_2$", fontsize=12)
plt.show()
```



The least-squares approach gives an exact closed-form solution for the discriminant function parameters. However, even as a discriminant function (making decisions directly) it suffers from some problems. We already know that least-squares solutions lack robustness to outliers, and this applies equally to classification, as depicted in the following figure. Note that the additional outlier data points produce a change in the location of the decision boundary, even though these point would be correctly classified by the original decision boundary. The sum-of-squares error function penalizes predictions that are *too correct* in that they lie a long way on the correct side of the decision boundary.

[3]:
```
# number of training points
N = 100

# number of outlier points
n_outliers = 5

x_train, t = make_classification(
    n_features=2,
    n_informative=2,
    n_redundant=0,
    n_classes=2,
    n_clusters_per_class=1,
    n_samples=N,
    random_state=12
)
```

```
x1, x2 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
x_test = np.array([x1, x2]).reshape(2, -1).T

outliers = np.random.random_sample((n_outliers, 2)) + 3
x_train_outliers = np.vstack((x_train, outliers))
t_outliers = np.hstack((t, np.ones(n_outliers, dtype=int)))

feature = LinearFeature()
x_train_linear = feature.transform(x_train)
x_train_linear_outliers = feature.transform(x_train_outliers)
x_test_linear = feature.transform(x_test)

model = LeastSquaresClassifier()
model.fit(x_train_linear, t)
predicted = model.predict(x_test_linear)

plt.figure(figsize=(15, 5))

plt.subplot(1, 2, 1)
plt.scatter(x_train[:, 0], x_train[:, 1], c=t)
plt.contourf(x1, x2, predicted.reshape(100, 100), alpha=0.2, levels=np.
 ↪linspace(0, 1, 3))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.xlabel("$x_1$", fontsize=12); plt.ylabel("$x_2$", fontsize=12)
plt.title("Original decision boundary")

model.fit(x_train_linear_outliers, t_outliers)
predicted_outliers = model.predict(x_test_linear)

plt.subplot(1, 2, 2)
plt.scatter(x_train_outliers[:, 0], x_train_outliers[:, 1], c=t_outliers)
plt.contourf(x1, x2, predicted_outliers.reshape(100, 100), alpha=0.2, levels=np.
 ↪linspace(0, 1, 3))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.xlabel("$x_1$", fontsize=12); plt.ylabel("$x_2$", fontsize=12)
plt.title("Outliers decision boundary")

plt.show()
```



The failure of least squares should not surprise us since it corresponds to maximum likelihood under the assumption of a Gaussian conditional distribution, whereas binary target vectors clearly do not have a Gaussian distribution.

4.1.4 Fisher's linear discriminant

Consider a two-class problem in which there are N_1 points of class $\mathcal{C}_1$ and N_2 points from class $\mathcal{C}_2$, so that the mean vectors of the two classes are given by

$$\mathbf{m}_1 = \frac{1}{N_1} \sum_{n \in \mathcal{C}_1} \mathbf{x}_n, \qquad \mathbf{m}_2 = \frac{1}{N_2} \sum_{n \in \mathcal{C}_2} \mathbf{x}_n$$

Then, the simplest measure of separation of the classes, when projected onto $\mathbf{w}$, is the separation of the projected class means. This suggests that we might choose $\mathbf{w}$ so as to maximize

$$m_2 - m_1 = \mathbf{w}^T(\mathbf{m}_2 - \mathbf{m}_1)$$

where

$$m_k = \mathbf{w}^T \mathbf{m}_k$$

is the mean of the projected data from class $\mathcal{C}_k$.

The idea proposed by Fisher is to maximize the function that gives a large separation between the projected class means while also giving a small variance within each class, thereby minimizing the class overlap. The within-class variance of the projected data from class $\mathcal{C}_k$ is given by,

$$s_k^2 = \sum_{n \in \mathcal{C}_k} (y_n - m_k)^2$$

where $y_n = \mathbf{w}^T \mathbf{x}_n$ is the projected data point in the one-dimentional space. We can further define the total within-class variance for the whole data set to be simply $s_1^2 + s_2^2$. Then, the Fisher criterion is defined as the ratio of the *between-class* variance to the *within-class* variance as follows,

$$J(\mathbf{w}) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2}$$

In order to explicitly show the dependence on $\mathbf{w}$, we may rewrite $J(\mathbf{w})$, using (4.20), (4.23), and (4.24), as follows,

$$\begin{aligned}
J(\mathbf{w}) &= \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2} \\
&= \frac{(\mathbf{w}^T\mathbf{m}_2 - \mathbf{w}^T\mathbf{m}_1)^2}{\sum_{n\in\mathcal{C}_k}(y_n - m_1)^2 + \sum_{n\in\mathcal{C}_k}(y_n - m_2)^2} \\
&= \frac{(\mathbf{w}^T(\mathbf{m}_2 - \mathbf{m}_1))^2}{\sum_{n\in\mathcal{C}_k}(\mathbf{w}^T(\mathbf{x}_n - \mathbf{m}_1))^2 + \sum_{n\in\mathcal{C}_k}(\mathbf{w}^T(\mathbf{x}_n - \mathbf{m}_2))^2} \\
&= \frac{\mathbf{w}^T(\mathbf{m}_2 - \mathbf{m}_1)^2\mathbf{w}}{\mathbf{w}^T(\sum_{n\in\mathcal{C}_k}(\mathbf{x}_n - \mathbf{m}_1)^2 + \sum_{n\in\mathcal{C}_k}(\mathbf{x}_n - \mathbf{m}_2)^2)\mathbf{w}} \\
&= \frac{\mathbf{w}^T\mathbf{S}_B\mathbf{w}}{\mathbf{w}^T\mathbf{S}_W\mathbf{w}}
\end{aligned}$$

Then, by computing the derivative with respect to $\mathbf{w}$, we find that $J(\mathbf{w})$ is maximized when,

$$(\mathbf{w}^T\mathbf{S}_B\mathbf{w})\mathbf{S}_W\mathbf{w} = (\mathbf{w}^T\mathbf{S}_W\mathbf{w})\mathbf{S}_B\mathbf{w} \Leftrightarrow \mathbf{w} \propto \mathbf{S}_W^{-1}(\mathbf{m}_2 - \mathbf{m}_1)$$

This result is known as the *Fisher's linear discriminant*. To that end, the projected data are compared against a threshold y_0 and classified as belonging to class $\mathcal{C}_1$ if $y(\mathbf{x}) \geq y_0$, and to class $\mathcal{C}_2$ otherwise.

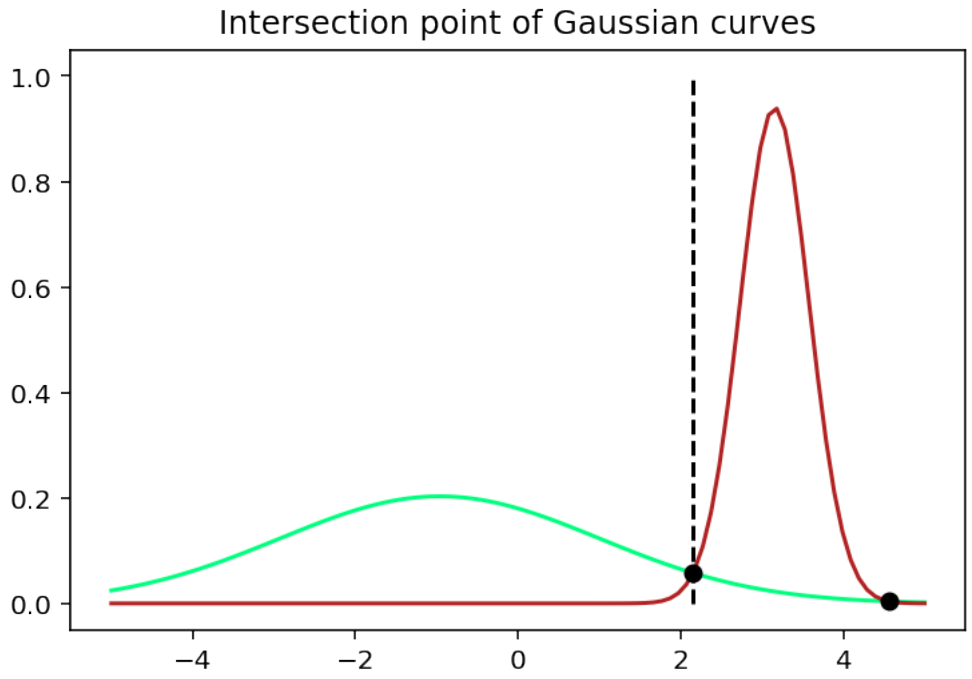
[4]:
```
# number of training points
N = 100

x_train, t = make_classification(
    n_features=2,
    n_informative=2,
    n_redundant=0,
    n_classes=2,
    n_clusters_per_class=1,
    n_samples=N,
    random_state=15
)

x1, x2 = np.meshgrid(np.linspace(-5, 5, N), np.linspace(-5, 5, N))
x_test = np.array([x1, x2]).reshape(2, -1).T

plt.scatter(x_train[:, 0], x_train[:, 1], c=t)
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.title("Training data (2 classes)")
```

```
plt.show()
```

One way of choosing the threshold y_0 is to model the class-conditional densities (one per class) $p(y|\mathcal{C}_k)$ as Gaussian distributions, then estimate their parameters using maximum likelihood, and finally, estimate the optimal threshold using decision theory. In the case of binary classification, we can equate the Gaussian functions and solve for $\mathbf{x}$. The result is a quadratic equation having coefficients relating to the gaussian means and variances.

[5]:
```
# split data points according to the classes
x_0 = x_train[t == 0]
x_1 = x_train[t == 1]

model = FisherLinearDiscriminant()
model.fit(x_train, t)

# create a Gaussian distribution per class
g0 = Gaussian()
g0.ml(x_0 @ model._w)
g1 = Gaussian()
g1.ml(x_1 @ model._w)

root = np.roots([
    g1.var - g0.var,
    2 * (g0.var * g1.mu - g1.var * g0.mu),
```

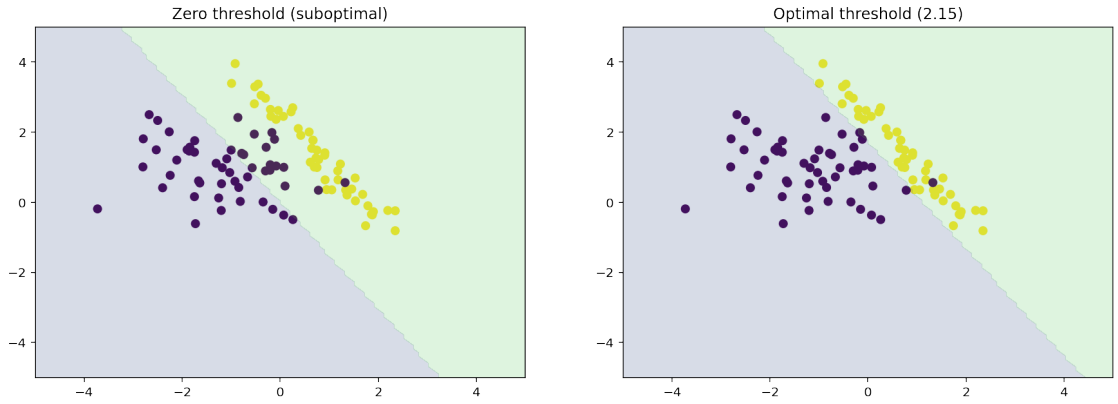
```
    g1.var * g0.mu ** 2 - g0.var * g1.mu ** 2 - g1.var * g0.var * np.log(g1.var␣
 ↪/ g0.var),
])

x = np.linspace(-5, 5, N)
plt.plot(x, g0.pdf(x), 'springgreen')
plt.plot(x, g1.pdf(x), 'firebrick')
plt.plot(root[0], g0.pdf(root[0]), 'ko')
plt.plot(root[1], g0.pdf(root[1]), 'ko')
plt.plot(np.zeros(x.size) + root[1], np.linspace(0, 1, N), 'k--')
plt.title("Intersection* point of Gaussian curves")

plt.show()
```



The figures below compares the optimal threshold against the naive zero threshold. Note that the selection of the decision threshold is crucial for an effective model.

[6]:
```
model = FisherLinearDiscriminant()
model.fit(x_train, t)
optimal_threshold = model._threshold

plt.figure(figsize=(15, 5))

# predict classes using the optimal threshold
predicted = model.predict(x_test)
```

```
plt.subplot(1, 2, 2)
plt.scatter(x_train[:, 0], x_train[:, 1], c=t)
plt.contourf(x1, x2, predicted.reshape(N, N), alpha=0.2, levels=np.linspace(0,␣
 ↪1, 3))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.title(f"Optimal threshold ({round(optimal_threshold, 2)})")

# set threshold to zero and make predictions
model._threshold = 0
predicted = model.predict(x_test)

plt.subplot(1, 2, 1)
plt.scatter(x_train[:, 0], x_train[:, 1], c=t)
plt.contourf(x1, x2, predicted.reshape(N, N), alpha=0.2, levels=np.linspace(0,␣
 ↪1, 3))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.title("Zero threshold (suboptimal)")

plt.show()
```



4.1.7 The perceptron algorithm

Another linear discriminant model is the perceptron. It corresponds to a two-class model in which the input vector $\mathbf{x}$ is first transformed using a nonlinear transformation to give a feature vector $\phi(\mathbf{x})$, and then use it to construct a generalized linear model of the form

$$y(\mathbf{x}) = f(\mathbf{w}^T \phi(\mathbf{x}))$$

We assume that the vector $\phi(\mathbf{x})$ typically includes the bias component ϕ_0. The nonlinear activation function $f(\cdot)$ is given by a step function of the form

$$f(\alpha) = \begin{cases} +1, & \alpha \geq 0 \\ -1, & \alpha < 0 \end{cases}$$

That is because for the perceptron it is more convenient to use target values $t = +1$ for class $\mathcal{C}_1$ and $t = -1$ for class $\mathcal{C}_2$, instead of $t \in \{0, 1\}$. We consider an error function called the *perceptron criterion*. Note that we are seeking a weight vector $\mathbf{w}$, such that the inputs $\mathbf{x}_n$, belonging in class $\mathcal{C}_1$, have $\mathbf{w}^T\phi(\mathbf{x}_n) > 0$, whereas the ones belonging in class $\mathcal{C}_2$, have $\mathbf{w}^T\phi(\mathbf{x}_n) < 0$. Given the coding scheme $t \in \{-1, +1\}$, it follows that all inputs must satisfy $\mathbf{w}^T\phi(\mathbf{x}_n)t_n > 0$. Thus, the perceptron criterion tries to minimize the quantity $-\mathbf{w}^T\phi(\mathbf{x}_n)t_n$, for all misclassified inputs. More formally,

$$E_P(\mathbf{w}) = -\sum_{n \in \mathcal{M}} \mathbf{w}^T\phi(\mathbf{x}_n)t_n$$

where $\mathcal{M}$ denotes the set of misclassified patterns.

We apply the stochastic gradient descent algorithm to the error function. Thus, the change in the weight vector, according to gradient descent, is given by,

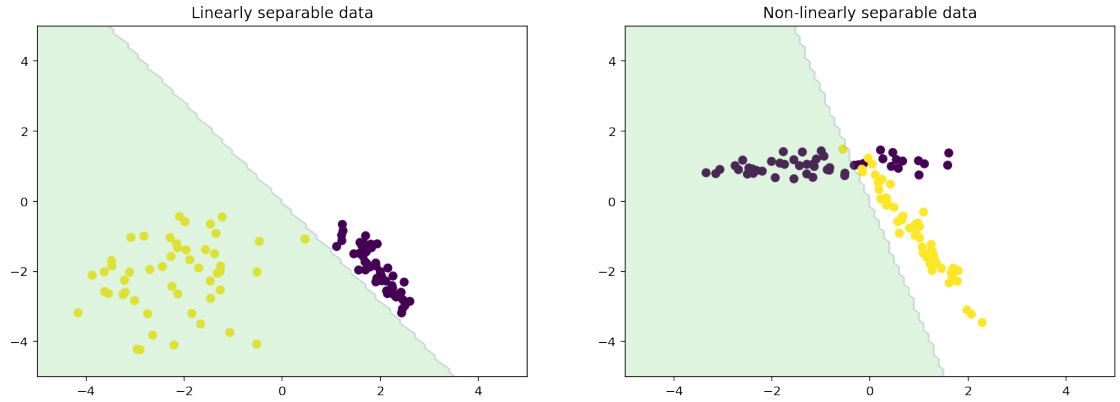
$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\nabla E_P(\mathbf{w}) = \mathbf{w}^{(\tau)} + \eta\phi_n t_n$$

The *perceptron convergence theorem* states that if there exists an exact solution (the data are linearly separable), the the perceptron algorithm is guaranteed to find the solution in a finite number of steps. On the other hand, if the data are not linearly separable the perceptron never converges. Moreover, note that the perceptron **does not provide probabilistic outputs, nor does it generalize to $K > 2$ classes**.

Another important limitation arises from the fact that it is based on linear combinations of fixed basis functions.

[11]:
```
# number of training points
N = 100

x1, x2 = np.meshgrid(np.linspace(-5, 5, N), np.linspace(-5, 5, N))
x_test = np.array([x1, x2]).reshape(2, -1).T

model = Perceptron()

plt.figure(figsize=(15, 5))

x_train, t = make_classification(
    n_features=2,
    n_informative=2,
    n_redundant=0,
    n_classes=2,
    n_clusters_per_class=1,
    n_samples=N,
    random_state=10,
```

```
    class_sep=2,
)

model.fit(x_train, np.where(t == 0, -1, 1))
predicted = model.predict(x_test)

plt.subplot(1, 2, 1)
plt.scatter(x_train[:, 0], x_train[:, 1], c=t)
plt.contourf(x1, x2, predicted.reshape(N, N), alpha=0.2, levels=np.linspace(0,␣
 ↪1, 3))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.title("Linearly separable data")

x_train, t = make_classification(
    n_features=2,
    n_informative=2,
    n_redundant=0,
    n_classes=2,
    n_clusters_per_class=1,
    n_samples=N,
    random_state=14,
)

model.fit(x_train, np.where(t == 0, -1, 1))
predicted = model.predict(x_test)

plt.subplot(1, 2, 2)
plt.scatter(x_train[:, 0], x_train[:, 1], c=t)
plt.contourf(x1, x2, predicted.reshape(N, N), alpha=0.2, levels=np.linspace(0,␣
 ↪1, 3))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.title("Non-linearly separable data")

plt.show()
```



4.2 Probabilistic Generative Models

Models having linear decision boundaries arise from simple assumptions about the distribution of the data. A generative approach models the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$, as well as the class priors $p(\mathcal{C}_k)$, and use them to compute the posterior probability $p(\mathcal{C}_k|\mathbf{x})$ throught *Bayes theorem*. To that end, the posterior probability for class $\mathcal{C}_1$, in a binary classification problem, is as follows,

$$\begin{aligned} p(\mathcal{C}_1|\mathbf{x}) &= \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x})} \\ &= \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1) + p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} \\ &= \frac{1}{1+\exp(-\alpha)} = \sigma(\alpha) \end{aligned}$$

where,

$$\alpha = \ln \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)}$$

Proof

$$\begin{aligned} \sigma(\alpha) &= \frac{1}{1+\exp(-\alpha)} \\ &= \frac{1}{1+\frac{1}{\exp(\alpha)}} = \frac{\exp(\alpha)}{1+\exp(\alpha)} \\ &= \frac{\exp\left(\ln \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)}\right)}{1+\exp\left(\ln \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)}\right)} \\ &= \frac{\frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)}}{1+\frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)}} = \frac{\frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)}}{\frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)+p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)}} \\ &= \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)(p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1) + p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2))} \\ &= \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1) + p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} \end{aligned}$$

The function $\sigma(\alpha)$ is the **logistic sigmoid** briefly presented in Chapter 3.

For $K > 2$ classes, the posterior for class $\mathcal{C}_k$ is as follows,

$$
\begin{aligned}
p(\mathcal{C}_k|\mathbf{x}) &= \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})} \\
&= \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{\sum_i p(\mathbf{x}|\mathcal{C}_i)} \\
&= \frac{\exp(\alpha_k)}{\sum_i \exp(\alpha_i)}
\end{aligned}
$$

which is known as the *normalized exponential* and can be regarded as a multiclass generalization of the logistic sigmoid function. The quantities α_k are defined as follows,

$$
\alpha_k = \ln(p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k))
$$

The normalized exponential is also known as the *softmax function*, since it represents a smoothed version of the max function, because if $\alpha_k \gg \alpha_i \; \forall i \neq k$, then $p(\mathcal{C}_k|\mathbf{x}) \approx 1$ and $p(\mathcal{C}_i|\mathbf{x}) \approx 0$.

4.2.1 Continuous inputs

Given the formulation above, the next step is to assume the form of the class-conditional densities. The Gaussian distributions may be used for modelling continuous variables. Assuming that all classes share the same covariance matrix, the density for class $\mathcal{C}_k$ is given by

$$
p(\mathbf{x}|\mathcal{C}_k) = \mathcal{N}(\mathbf{x}|\mu_k, \Sigma)
$$

Thus, from (4.58), we have,

$$
\begin{aligned}
\alpha &= \ln \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} = \ln \frac{\mathcal{N}(\mathbf{x}|\mu_1, \Sigma)p(\mathcal{C}_1)}{\mathcal{N}(\mathbf{x}|\mu_2, \Sigma)p(\mathcal{C}_2)} \\
&= \ln \frac{\mathcal{N}(\mathbf{x}|\mu_1, \Sigma)}{\mathcal{N}(\mathbf{x}|\mu_2, \Sigma)} + \ln \frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)} \\
&= \ln \frac{\exp\{-\frac{1}{2}(\mathbf{x}-\mu_1)^T\Sigma^{-1}(\mathbf{x}-\mu_1)\}}{\exp\{-\frac{1}{2}(\mathbf{x}-\mu_2)^T\Sigma^{-1}(\mathbf{x}-\mu_2)\}} + \ln \frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)} \\
&= -\frac{1}{2}(\mathbf{x}-\mu_1)^T\Sigma^{-1}(\mathbf{x}-\mu_1) + \frac{1}{2}(\mathbf{x}-\mu_2)^T\Sigma^{-1}(\mathbf{x}-\mu_2) + \ln \frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)} \\
&= -\frac{1}{2}\mathbf{x}^T\Sigma^{-1}\mathbf{x} + \mu_1^T\Sigma^{-1}\mathbf{x} - \frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mathbf{x}^T\Sigma^{-1}\mathbf{x} - \mu_2^T\Sigma^{-1}\mathbf{x} + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + \ln \frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)} \\
&= \mu_1^T\Sigma^{-1}\mathbf{x} - \mu_2^T\Sigma^{-1}\mathbf{x} - \frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + \ln \frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)} \\
&= (\mu_1 - \mu_2)^T\Sigma^{-1}\mathbf{x} - \frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + \ln \frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)}
\end{aligned}
$$

To that end, using (4.57), we derive that,

$$
p(\mathcal{C}_1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + w_0)
$$

where,

$$\mathbf{w} = \Sigma^{-1}(\mu_1 - \mu_2)$$
$$w_0 = -\frac{1}{2}\mu_1^T\Sigma^{-1}\mu_1 + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + \ln\frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)}$$

Note that the prior probabilities $p(\mathcal{C}_k)$ enter through the bias parameter w_0, thus making parallel shifts of the decision boundary.

For the general case of K classes, from (4.63), we have,

$$\begin{aligned}\alpha_k &= \ln\left(\frac{1}{(2\pi)^{D/2}}\right) + \ln\left(\frac{1}{|\Sigma|^{1/2}}\right) - \frac{1}{2}(\mathbf{x}-\mu_k)^T\Sigma^{-1}(\mathbf{x}-\mu_k) + \ln p(\mathcal{C}_k)\\ &= \ln\left(\frac{1}{(2\pi)^{D/2}}\right) + \ln\left(\frac{1}{|\Sigma|^{1/2}}\right) - \frac{1}{2}\mathbf{x}^T\Sigma^{-1}\mathbf{x} + \mu_k^T\Sigma^{-1}\mathbf{x} - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \ln p(\mathcal{C}_k)\\ &= \ln A + \ln B + Q + \mathbf{w}_k^T\mathbf{x} + w_{k0}\end{aligned}$$

where,

$$\begin{aligned}A &= \ln\left(\frac{1}{(2\pi)^{D/2}}\right)\\ B &= \ln\left(\frac{1}{|\Sigma|^{1/2}}\right)\\ Q &= -\frac{1}{2}\mathbf{x}^T\Sigma^{-1}\mathbf{x}\\ \mathbf{w}_k &= \Sigma^{-1}\mu_k\\ \mathbf{w}_{k0} &= -\frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \ln p(\mathcal{C}_k)\end{aligned}$$

Then using (4.62), we derive,

$$\begin{aligned}p(\mathcal{C}_k|\mathbf{x}) &= \frac{\exp(\alpha_k)}{\sum_j \exp(\alpha_j)}\\ &= \frac{\exp(A+B+Q)\exp(\mathbf{w}_k^T\mathbf{x} + w_{k0})}{\exp(A+B+Q)\sum_j \exp(\mathbf{w}_j^T\mathbf{x} + w_{j0})}\end{aligned}$$

and re-define α_k as follows,

$$a_k(\mathbf{x}) = \mathbf{w}_k^T\mathbf{x} + w_{k0}$$

Therefore, we see that for $K > 2$ classes, α_k are linear functions of $\mathbf{x}$ since quadratic terms cancel each other due to the shared covariances. By relaxing the assumption of the shared covariance matrix among the classes, allowing each class to have each won covariance matrix Σ_k, then we obtain quadratic functions of $\mathbf{x}$, giving rise to *quadratic discriminant*.

4.2.2 Maximum likelihood solution

Given a set of data, comprising observations $\mathbf{x}$ and corresponding class labels, we can determine the parameters of the class-conditional densities and class prior probabilities, using maximum likelihood. Suppose that we are given a dataset $\{\mathbf{x}, t_n\}$, where $t_n = 1$ denotes class $\mathcal{C}_1$ and $t_n = 0$ denotes $\mathcal{C}_2$. Then, for a data point $\mathbf{x}_n$ belonging to class $\mathcal{C}_1$ ($t_n = 1$), we have,

$$p(\mathbf{x}_n, \mathcal{C}_1) = p(\mathcal{C}_1)p(\mathbf{x}_n|\mathcal{C}_1) = \pi\mathcal{N}(\mathbf{x}_n|\mu_1, \Sigma)$$

Similarly, for class $\mathcal{C}_2$ ($t_n = 0$),

$$p(\mathbf{x}_n, \mathcal{C}_2) = p(\mathcal{C}_2)p(\mathbf{x}_n|\mathcal{C}_2) = (1-\pi)\mathcal{N}(\mathbf{x}_n|\mu_2, \Sigma)$$

where $p(\mathcal{C}_1) = \pi$ and complementary $p(\mathcal{C}_2) = 1 - \pi$.

Thus, the likelihood function is given by,

$$p(\mathbf{t}, \mathbf{X}|\pi, \mu_1, \mu_2, \Sigma) = \prod_{n=1}^{N} \left[\pi\mathcal{N}(\mathbf{x}_n|\mu_1, \Sigma)\right]^{t_n} \left[(1-\pi)\mathcal{N}(\mathbf{x}_n|\mu_2, \Sigma)\right]^{1-t_n}$$

and the log-likelihood is as follows,

$$\ln p(\mathbf{t}, \mathbf{X}|\pi, \mu_1, \mu_2, \Sigma) = \sum_{n=1}^{N} t_n(\ln\pi + \ln\mathcal{N}(\mathbf{x}_n|\mu_1, \Sigma)) + (1-t_n)(\ln(1-\pi) + \ln\mathcal{N}(\mathbf{x}_n|\mu_2, \Sigma))$$

1. Setting the derivative for π equal to zero, we obtain,

$$\begin{aligned}
&\frac{d}{d\pi}\ln p(\mathbf{t}, \mathbf{X}|\pi, \mu_1, \mu_2, \Sigma) = 0 \Leftrightarrow \\
&\frac{d}{d\pi}\sum_{n=1}^{N}\{t_n\ln\pi + (1-t_n)\ln(1-\pi)\} = 0 \Leftrightarrow \\
&\frac{1}{\pi}\sum_{n=1}^{N} t_n - \frac{1}{1-\pi}\sum_{n=1}^{N}(1-t_n) = 0 \Leftrightarrow \\
&\frac{1}{\pi}\sum_{n=1}^{N} t_n - \frac{1}{1-\pi}(N - \sum_{n=1}^{N} t_n) = 0 \Leftrightarrow \\
&\frac{1}{\pi}\sum_{n=1}^{N} t_n = \frac{1}{1-\pi}(N - \sum_{n=1}^{N} t_n) \Leftrightarrow \\
&\frac{1-\pi}{\pi}\sum_{n=1}^{N} t_n = N - \sum_{n=1}^{N} t_n \Leftrightarrow \\
&\frac{1}{\pi}\sum_{n=1}^{N} t_n - \sum_{n=1}^{N} t_n = N - \sum_{n=1}^{N} t_n \Leftrightarrow \\
&\pi = \frac{1}{N}\sum_{n=1}^{N} t_n
\end{aligned}$$

As expected, the maximum likelihood estimate for π, is simply the fraction of points in class $\mathcal{C}_1$.

2. Setting the derivative for μ_1 equal to zero, we obtain,

$$\begin{aligned}
&\frac{d}{d\mu_1} \ln p(\mathbf{t}, \mathbf{X}|\pi, \mu_1, \mu_2, \Sigma) = 0 \Leftrightarrow \\
&\frac{d}{d\mu_1} \sum_{n=1}^{N} t_n \ln \mathcal{N}(\mathbf{x}_n|\mu_1, \Sigma) = 0 \Leftrightarrow \\
&\frac{d}{d\mu_1} \left[-\frac{1}{2} \sum_{n=1}^{N} t_n (\mathbf{x}_n - \mu_1)^T \Sigma^{-1} (\mathbf{x}_n - \mu_1) \right] = 0 \Leftrightarrow \\
&-\frac{1}{2} \sum_{n=1}^{N} -2 t_n \Sigma^{-1} (\mathbf{x}_n - \mu_1) = 0 \Leftrightarrow \\
&\sum_{n=1}^{N} t_n (\mathbf{x}_n - \mu_1) = 0 \overset{\sum_{n=1}^{N} t_n = N_1}{\Leftrightarrow} \\
&\sum_{n=1}^{N} t_n \mathbf{x}_n = N_1 \mu_1 \Leftrightarrow \\
&\mu_1 = \frac{1}{N_1} \sum_{n=1}^{N} t_n \mathbf{x}_n
\end{aligned}$$

3. Similarly, the corresponding result for μ_2 is given by,

$$\mu_2 = \frac{1}{N_2} \sum_{n=1}^{N} (t_n - 1) \mathbf{x}_n$$

4. Finally, the solution for the shared covariance matrix Σ is similar to the one derived for the multivariate Gaussian distribution is Chapter 2, where the matrix Σ is defined in (4.78), (4.79), and (4.80).

Note: Fitting Gaussian distributions to the classes is not robust to outliers, because the maximum likelihood estimation of a Gaussian is not robust itself.

4.2.3 Discrete features

Consider the case of discrete binary feature values $x_i \in \{0, 1\}$. When there are D inputs, then a general distribution would correspond to $2^D - 1$ independent variables. Assuming a *naive Bayes* approach, we have the following class-conditional mass functions,

$$p(\mathbf{x}|\mathcal{C}_k) = \prod_{i=1}^{D} \mu_{ki}^{x_i} (1 - \mu_{ki})^{1 - x_i}$$

For K classes, substituting into (4.63), gives,

$$\alpha_k(\mathbf{x}) = \sum_{i=1}^{D} \left(x_i \ln \mu_{ki} + (1 - x_i) \ln(1 - \mu_{ki}) \right) + \ln p(\mathcal{C}_k)$$

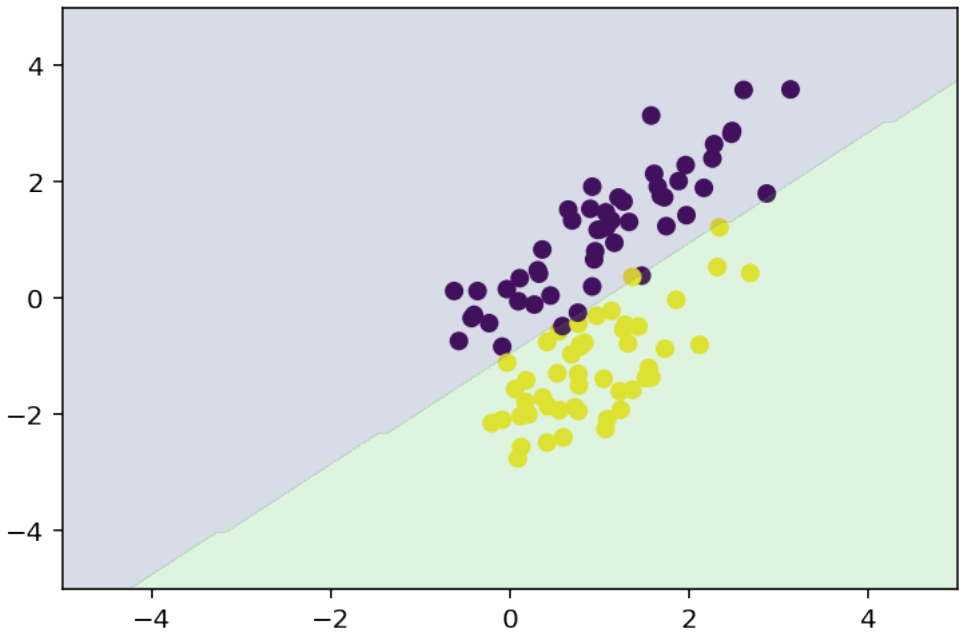
In the more general case, where discrete variables can take $M > 2$ states, the class-conditional mass functions are defined as follows,

$$p(\mathbf{x}|\mathcal{C}_k) = \prod_{i=1}^{D} \prod_{m=1}^{M} \mu_{kim}^{\phi(x_i)_m}$$

where $\phi(x_i)$ produces a 1-of-M binary coding scheme, where only one of the value among $\phi(x_i)_1, \ldots, \phi(x_i)_M$ is 1, and the others are all 0. Thus, by substituting the expression above into (4.63), gives,

$$\alpha_k(\mathbf{x}) = \sum_{i=1}^{D} \sum_{m=1}^{M} (\phi(x_i)_m \ln \mu_{kim}) + \ln p(\mathcal{C}_k)$$

[3]:
```
# number of training points
N = 100

x_train, t = make_classification(
    n_features=2,
    n_informative=2,
    n_redundant=0,
    n_classes=2,
    n_clusters_per_class=1,
    n_samples=N,
    random_state=21
)

x1, x2 = np.meshgrid(np.linspace(-5, 5, N), np.linspace(-5, 5, N))
x_test = np.array([x1, x2]).reshape(2, -1).T

model = GenerativeClassifier()
model.fit(x_train, t)
predicted = model.predict(np.array([np.ravel(x1), np.ravel(x2)]))

plt.scatter(x_train[:, 0], x_train[:, 1], c=t)
plt.contourf(x1, x2, predicted.reshape(N, N), alpha=0.2, levels=np.linspace(0,␣
 ↪1, 3))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.show()
```



4.3 Probabilistic Discriminative Models

An alternative approach, called *discriminative training*, is to directly maximize the likelihood function defined through the conditional distribution $p(\mathcal{C}_k|\mathbf{x})$.

4.3.2 Logistic Regression

Consider the binary classification problem. In the analysis of generative approaches we saw that under rather general assumptions, the posterior probability of class $\mathcal{C}_1$ can be expressed as a logistic sigmoid acting on a linear function of the input vectors $\mathbf{x}$ or the feature vector ϕ (see 4.65) so that,

$$p(\mathcal{C}_1|\phi) = y(\phi) = \sigma(\mathbf{w}^T\phi)$$

In the terminology of statistics, this model is known as *logistic regression*, although its a classification model.

> One advantage of the discriminative approach is that there are typically fewer adaptive parameters to be determined. For an M-dimensional feature space, this model has M adjustable parameters. By contrast, the generative model using Gaussian class conditional densities, would have used $2M$ parameters for the means and $M(M+1)/2$ parameters for the (shared) covariance matrix.

We can use maximum likelihood to determine the parameters of the logistic regression model. Given a data set $\{\phi_n, t_n\}$, where $t_n \in \{0, 1\}$, the likelihood function is given by,

$$p(\mathbf{t}|\Phi, \mathbf{w}) = \prod_{n=1}^{N} p(\mathcal{C}_1|\phi_n)^{t_n}(1 - p(\mathcal{C}_1|\phi_n))^{1-t_n} = \prod_{n=1}^{N} y_n^{t_n}\{1 - y_n\}^{1-t_n}$$

The maximum likelihood is equivalent to the minimum of the negative of the logarithm of the likelihood, which gives the *cross-entropy error function*,

$$E(\mathbf{w}) = -\ln p(\mathbf{t}|\Phi, \mathbf{w}) = -\sum_{n=1}^{N} t_n \ln y_n + (1 - t_n) \ln(1 - y_n)$$

Why is the error function called cross-entropy?

The cross-entropy for discrete probability distributions p and q is defined as $H(p, q) = \sum_x p(x) \log q(x)$. Since we assume that the target variables t_n are probabilities taking only extreme values 0 or 1, and y_n is a probability distribution, then $E(\mathbf{w})$ can be interpreted as the cross entropy of the target variables and the posterior probability distribution.

Then, taking the gradient of the error function over $\mathbf{w}$, we obtain,

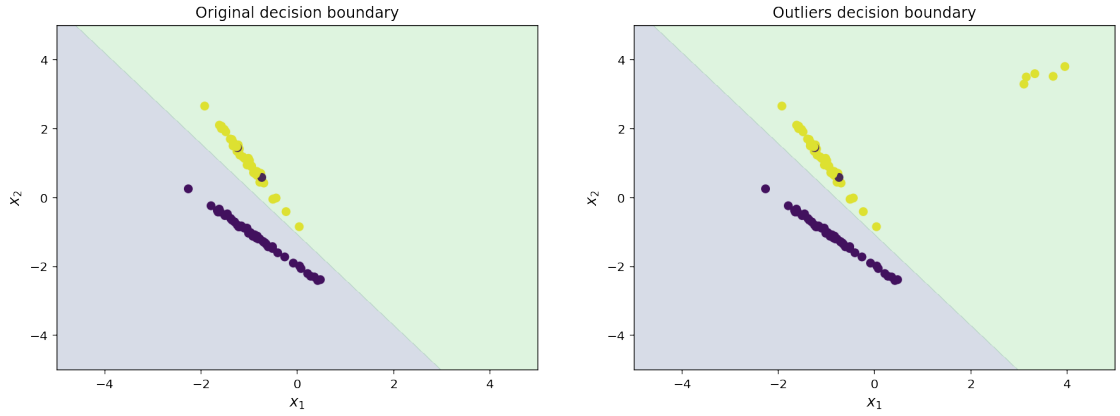
$$\begin{aligned}
\nabla E(\mathbf{w}) &= -\nabla \ln p(\mathbf{t}|\Phi, \mathbf{w}) \\
&= -\nabla \sum_{n=1}^{N} t_n \ln y_n + (1 - t_n) \ln(1 - y_n) \\
&= -\sum_{n=1}^{N} \frac{d}{dy_n} t_n \ln y_n + \frac{d}{dy_n}(1 - t_n) \ln(1 - y_n) \\
&\overset{\frac{d}{dx} \ln f(x) = \frac{f'(x)}{f(x)}}{=} -\sum_{n=1}^{N} \frac{d}{dy_n} t_n \ln y_n + \frac{d}{dy_n}(1 - t_n) \ln(1 - y_n) \\
&= -\sum_{n=1}^{N} \frac{t_n}{y_n} \frac{d}{da_n} y_n \frac{d}{d\mathbf{w}} a_n - \frac{1 - t_n}{1 - y_n} \frac{d}{da_n} y_n \frac{d}{d\mathbf{w}} a_n \\
&= -\sum_{n=1}^{N} (\frac{t_n}{y_n} - \frac{1 - t_n}{1 - y_n}) \frac{d}{da_n} y_n \frac{d}{d\mathbf{w}} a_n \\
&= -\sum_{n=1}^{N} (\frac{t_n}{y_n} - \frac{1 - t_n}{1 - y_n}) y_n (1 - y_n) \phi_n \\
&\overset{(4.88)}{=} -\sum_{n=1}^{N} \frac{t_n - y_n}{y_n(1 - t_n)} y_n (1 - y_n) \phi_n \\
&= \sum_{n=1}^{N} (y_n - t_n) \phi_n
\end{aligned}$$

Note that the gradient takes the same form as the gradient of the sum-of-squares error function, however, y_n involves a non-linear function. At this point we can make use of (4.91) and (3.22) to obtain a sequential algorithm (gradient descent) for optimizing the parameters.

[6]:
```
# number of training points
N = 100

# number of outlier points
n_outliers = 5
```

```
x_train, t = make_classification(
    n_features=2,
    n_informative=2,
    n_redundant=0,
    n_classes=2,
    n_clusters_per_class=1,
    n_samples=N,
    random_state=12
)

x1, x2 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
x_test = np.array([x1, x2]).reshape(2, -1).T

outliers = np.random.random_sample((n_outliers, 2)) + 3
x_train_outliers = np.vstack((x_train, outliers))
t_outliers = np.hstack((t, np.ones(n_outliers, dtype=int)))

feature = LinearFeature()
x_train_linear = feature.transform(x_train)
x_train_linear_outliers = feature.transform(x_train_outliers)
x_test_linear = feature.transform(x_test)

model = LogisticRegression()
model.fit_lms(x_train_linear, t, 0.01)
predicted = model.predict(x_test_linear)

plt.figure(figsize=(15, 5))

plt.subplot(1, 2, 1)
plt.scatter(x_train[:, 0], x_train[:, 1], c=t)
plt.contourf(x1, x2, predicted.reshape(100, 100), alpha=0.2, levels=np.
 ↪linspace(0, 1, 3))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.xlabel("$x_1$", fontsize=12); plt.ylabel("$x_2$", fontsize=12)
plt.title("Original decision boundary")

model.fit_lms(x_train_linear_outliers, t_outliers, 0.01)
predicted_outliers = model.predict(x_test_linear)

plt.subplot(1, 2, 2)
plt.scatter(x_train_outliers[:, 0], x_train_outliers[:, 1], c=t_outliers)
plt.contourf(x1, x2, predicted_outliers.reshape(100, 100), alpha=0.2, levels=np.
 ↪linspace(0, 1, 3))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.xlabel("$x_1$", fontsize=12); plt.ylabel("$x_2$", fontsize=12)
plt.title("Outliers decision boundary")
```

```
plt.show()
```



Note that logistic regression is robust to outliers in contrast to linear discriminants presented in section* 4.1.

4.3.3 Iterative reweighted least squares

In linear regression models, the maximum likelihood solution, on the assumption of Gaussian noise model, leads to a closed-form solution. For logistic regression, there is no longer a closed-form solution, due to the nonlinearity of the logistic sigmoid function. However, the error function is still convex and can be minimized by an efficient iterative technique based on *Newton-Raphson* iterative optimization scheme. This algorithm uses a local quadratic approximation to the log-likelihood, and takes the form

$$\mathbf{w}^{new} = \mathbf{w}^{old} - \mathbf{H}^{-1}\nabla E(\mathbf{w})$$

where $\mathbf{H}$ is the Hessian matrix whose elements comprise the second derivatives of $E(\mathbf{w})$ over $\mathbf{w}$.

Note that, if we apply the *Newton-Raphson* algorithm to the linear regression model, we derive the standard least squares solution (see 4.94 and 4.95).

Applying the *Newton-Raphson* update to the cross-entropy error function for the logistic regression model, we obtain,

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N}(y_n - t_n)\phi_n = \Phi^T(\mathsf{y} - \mathsf{t})$$

and

$$
\begin{aligned}
\mathbf{H} &= \nabla\nabla E(\mathbf{w}) \\
&= \nabla \sum_{n=1}^{N} (y_n - t_n)\phi_n \\
&= \nabla \sum_{n=1}^{N} y_n \phi_n \\
&\overset{(4.88)}{=} \sum_{n=1}^{N} y_n(1-y_n)\phi_n \frac{d}{d\mathbf{w}} \mathbf{w}^T \phi_n \\
&= \sum_{n=1}^{N} y_n(1-y_n)\phi_n \phi_n^T \\
&= {}^T\mathbf{R}
\end{aligned}
$$

where $\mathbf{R}$ is a diagonal matrix whose elements are $R_{nn} = y_n(1-y_n)$. Then, the update formula becomes,

$$
\mathbf{w}^{new} = \mathbf{w}^{old} - (\ {}^T\mathbf{R}\ )^{-1}\Phi^T(\mathsf{y} - \mathsf{t})
$$

Note that Hessian depends on $\mathbf{w}$ through the weighting matrix $\mathbf{R}$, corresponding to the fact that the error function is no longer quadratic. Thus, we must apply the update formula iteratively, each time using the new weight vector $\mathbf{w}$ to compute the revised weighting matrix $\mathbf{R}$. To that end, the algorithm is known as *iterative reweighted least squares* or *IRLS*.

The elements of $\mathbf{R}$ can be interpreted as variances, given by,

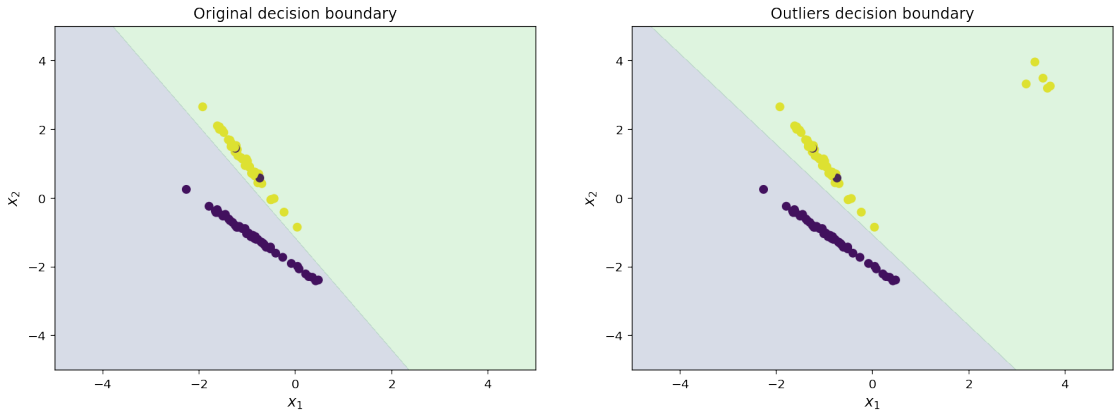
$$
\mathbb{E}[t] = \sum_{t \in \{0,1\}} t p(t|\mathbf{x}) = \sigma(x)
$$

and

$$
\mathrm{var}[t] = \mathbb{E}[t^2] - \mathbb{E}[t]^2 \overset{t^2=t}{=} \mathbb{E}[t] - \mathbb{E}[t]^2 = \sigma(x) - \sigma(x)^2 = y(1-y)
$$

[11]:
```
# number of training points
N = 100

# number of outlier points
n_outliers = 5

x_train, t = make_classification(
    n_features=2,
    n_informative=2,
    n_redundant=0,
    n_classes=2,
    n_clusters_per_class=1,
    n_samples=N,
```

```
    random_state=12
)

x1, x2 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
x_test = np.array([x1, x2]).reshape(2, -1).T

outliers = np.random.random_sample((n_outliers, 2)) + 3
x_train_outliers = np.vstack((x_train, outliers))
t_outliers = np.hstack((t, np.ones(n_outliers, dtype=int)))

feature = LinearFeature()
x_train_linear = feature.transform(x_train)
x_train_linear_outliers = feature.transform(x_train_outliers)
x_test_linear = feature.transform(x_test)

model = LogisticRegression()
model.fit(x_train_linear, t)
predicted = model.predict(x_test_linear)

plt.figure(figsize=(15, 5))

plt.subplot(1, 2, 1)
plt.scatter(x_train[:, 0], x_train[:, 1], c=t)
plt.contourf(x1, x2, predicted.reshape(100, 100), alpha=0.2, levels=np.
 ↪linspace(0, 1, 3))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.xlabel("$x_1$", fontsize=12); plt.ylabel("$x_2$", fontsize=12)
plt.title("Original decision boundary")

model.fit_lms(x_train_linear_outliers, t_outliers, 0.01)
predicted_outliers = model.predict(x_test_linear)

plt.subplot(1, 2, 2)
plt.scatter(x_train_outliers[:, 0], x_train_outliers[:, 1], c=t_outliers)
plt.contourf(x1, x2, predicted_outliers.reshape(100, 100), alpha=0.2, levels=np.
 ↪linspace(0, 1, 3))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.xlabel("$x_1$", fontsize=12); plt.ylabel("$x_2$", fontsize=12)
plt.title("Outliers decision boundary")

plt.show()
```



4.3.4 Multiclass logistic regression

We have seen that for $K > 2$ classes, the posterior probabilities are given by a softmax transformation of linear functions of feature variables. Here, we consider the maximum likelihood to determine the parameters $\mathbf{w}_k$ of the model directly. To that end, we need to calculate the derivatives of y_k (see 4.104) over the activation functions α_j (see 4.68 and 4.105).

In order to find the derivatives, we need to consider $k \neq j$ and $k = j$.

1. $k \neq j$

$$\begin{aligned}
\frac{\partial y_k}{\partial \alpha_k} &= \frac{\partial}{\partial \alpha_k} \frac{\exp(\alpha_k)}{\sum_j \exp(\alpha_j)} \\
&= \frac{-\exp(\alpha_k)\exp(\alpha_j)}{\left(\sum_j \exp(\alpha_j)\right)^2} \\
&= -\frac{\exp(\alpha_k)}{\sum_j \exp(\alpha_j)} \frac{\exp(\alpha_j)}{\sum_j \exp(\alpha_j)} \\
&\overset{4.104}{=} -y_k y_j
\end{aligned}$$

2. $k = j$

$$
\begin{aligned}
\frac{\partial y_k}{\partial \alpha_k} &= \frac{\partial}{\partial \alpha_k} \frac{\exp(\alpha_k)}{\sum_j \exp(\alpha_j)} \\
&= \frac{\exp(\alpha_k) \sum_j \exp(\alpha_j) - \exp(\alpha_k)^2}{\left(\sum_j \exp(\alpha_j)\right)^2} \\
&= \frac{\exp(\alpha_k) \sum_j \exp(\alpha_j)}{\left(\sum_j \exp(\alpha_j)\right)^2} - \frac{\exp(\alpha_k)^2}{\left(\sum_j \exp(\alpha_j)\right)^2} \\
&= \frac{\exp(\alpha_k)}{\sum_j \exp(\alpha_j)} - \left(\frac{\exp(\alpha_k)}{\sum_j \exp(\alpha_j)}\right)^2 \\
&= y_k - y_k^2 \\
&= y_k(1 - y_k)
\end{aligned}
$$

where we have used the quotient rule $\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$.

Combining (1) and (2), we obtain,

$$
\frac{\partial y_k}{\partial \alpha_k} = y_k(I_{kj} - y_j)
$$

where $I_k j$ are the elements of the identity matrix.

Assuming a 1-of-K coding scheme in which the target vector $\mathbf{t}_k$ is a binary vector having all elements zero except for element k, which equals to one, then ,the likelihood function is then given by,

$$
p(\mathbf{T}|\mathbf{w}_1, \ldots, \mathbf{w}_k) = \prod_{n=1}^{N} \prod_{k=1}^{K} p(\mathbf{C}_k|\phi_n)^{t_{nk}} = \prod_{n=1}^{N} \prod_{k=1}^{K} y_k(\phi_n)^{t_{nk}}
$$

where $\mathbf{T}$ is a $N \times K$ matrix of target variables with elements t_{nk}. Taking the negative logarithm the gives,

$$
E(\mathbf{w}_1, \ldots, \mathbf{w}_k) = -\ln p(\mathbf{T}|\mathbf{w}_1, \ldots, \mathbf{w}_k) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \ln y_k(\phi_n)
$$

which is the *cross-entropy* error function for the multiclass problem.

Taking the gradient of the error function over the parameter vector $\mathbf{w}_j$, we obtain

$$
\begin{aligned}
\nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \dots, \mathbf{w}_K) &= -\nabla_{\mathbf{w}_j} \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \ln y_k(\phi_n) \\
&= -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \frac{1}{y_{nk}} y_{nk}(I_{kj} - y_{nj})\phi_n \\
&= -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk}(I_{kj} - y_{nj})\phi_n \\
&= \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} y_{nj} \phi_n - \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} I_{kj} \phi_n \\
&= \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} y_{nj} \phi_n - \sum_{n=1}^{N} t_{nj} \phi_n \\
&\overset{\sum_k t_{nk}=1}{=} \sum_{n=1}^{N} y_{nj} \phi_n - \sum_{n=1}^{N} t_{nj} \phi_n \\
&= \sum_{n=1}^{N} (y_{nj} - t_{nj}) \phi_n
\end{aligned}
$$

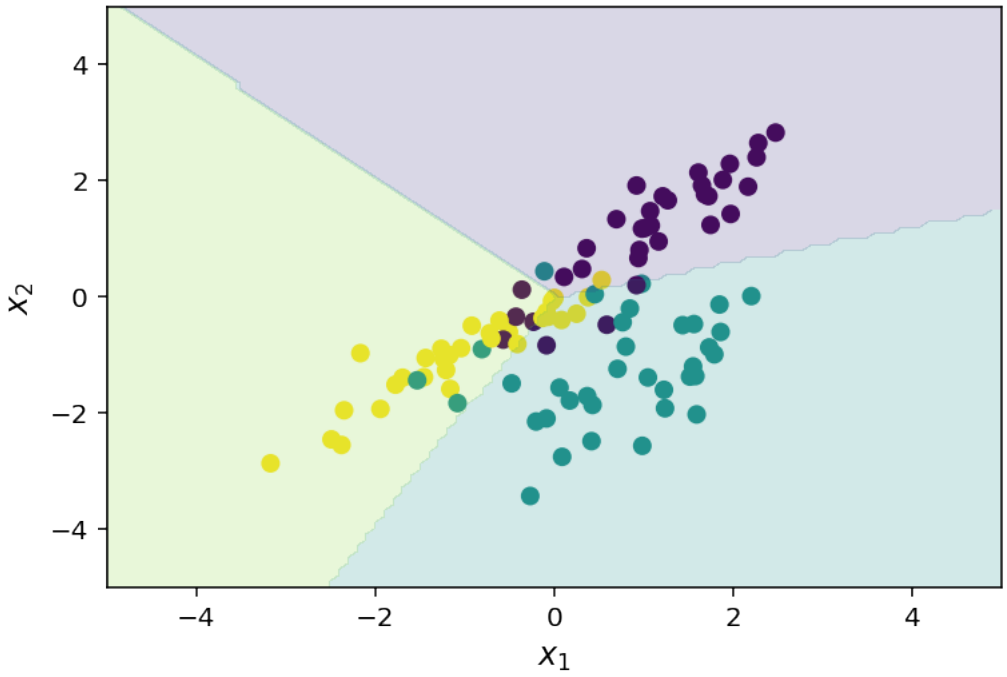
The Newton-Raphson update formula, requires the evaluation of the Hessian matrix, given by

$$
\begin{aligned}
\nabla_{\mathbf{w}_k} \nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \dots, \mathbf{w}_K) &= -\nabla_{\mathbf{w}_k} \nabla_{\mathbf{w}_j} \ln p(\mathbf{T} | \mathbf{w}_1, \dots, \mathbf{w}_k) \\
&= -\nabla_{\mathbf{w}_k} \sum_{n=1}^{N} (y_{nj} - t_{nj}) \phi_n \\
&= -\nabla_{\mathbf{w}_k} \sum_{n=1}^{N} y_{nj} \phi_n \\
&= \sum_{n=1}^{N} \frac{\partial}{\partial \mathbf{w}_k} y_{nj} \phi_n \\
&= \sum_{n=1}^{N} y_{nk}(I_{kj} - y_{nj}) \phi_n \phi_n^T
\end{aligned}
$$

Below we present a softmax regression example trained using gradient descent.

[13]:
```
# number of training points
N = 100

x_train, t = make_classification(
    n_features=2,
    n_informative=2,
    n_redundant=0,
    n_classes=3,
    n_clusters_per_class=1,
    n_samples=N,
    random_state=21
```

```
)

x1, x2 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
x_test = np.array([x1, x2]).reshape(2, -1).T

model = SoftmaxRegression()
model.fit(x_train, t)
predicted = model.predict(x_test)

plt.scatter(x_train[:, 0], x_train[:, 1], c=t)
plt.contourf(x1, x2, predicted.reshape(100, 100), alpha=0.2, levels=np.
 ↪linspace(0, 2, 4))
plt.xlim(-5, 5); plt.ylim(-5, 5)
plt.xlabel("$x_1$", fontsize=12); plt.ylabel("$x_2$", fontsize=12)

plt.show()
```



4.4 Laplace Approximation

In contrast to the Bayesian treatment of linear regression, in the Bayesian treatment of logistic regression, we cannot integrate exactly over the parameter vector $\mathbf{w}$ since the posterior distribution is no longer Gaussian. To that end, we may use a widely used framework called the Laplace approximation, that aims to find a Gaussian approximation to a probability density defined over a set of continuous variables. Consider a single continuous variable z, having a distribution $p(z)$ defined by

$$p(z) = \frac{1}{Z} f(z)$$

where $Z = \int f(z)\mathrm{d}z$ is the normalization coefficient. The goal is to find a Gaussian approximation $q(z)$ centered on a mode of the distribution $p(z)$. The first step is to find a mode of $p(z)$, that is, a point z_0 such that $p'(z_0) = 0$ or equivalently

$$\frac{df(z)}{dz}_{z=z_0} = 0$$

Then, we use a second-order Taylor expansion to approximate $g(z) = \ln f(z)$ (because the logarithm of any Gaussian distribution is a quadratic function of the variables), centered on the mode z_0 so that,

$$g(z) = \ln f(z) \approx \sum_{n=0}^{2} \frac{g^{(n)}(z_0)}{n!}(z - z_0)^n = g(z_0) - \frac{1}{2} g''(z_0)(z - z_0)^2$$

Note that the first-order term is omitted since z_0 is a local maximum of the distribution and thus the derivative is zero. Then, taking the exponential on both sides of the expansion, we obtain

$$f(z) \approx f(z_0) \exp\Big\{ -\frac{A}{2}(z - z_0)^2 \Big\}$$

Therefore, using the standard result for the normalization of a Gaussian, the final normalized distribution $q(z)$ has the form,

$$q(z) = \Big(\frac{A}{2\pi}\Big)^{1/2} \exp\Big\{ -\frac{A}{2}(z - z_0)^2 \Big\} = \mathcal{N}(z|z_0, A)$$

Note that the Gaussian approximation is well defined only when its precision $A > 0$, which implies that z_0 must be a local maximum, not a minimum! In practice a mode may be found by running some form of numerical optimization. The Laplace approximation is depicted in the next Figure,

The same approximation can be applied to an M-dimentional space $\mathbf{z}$.

[ ]: