Bayesian Analysis via MCMC

Contents

4.1. Linear Regression Problem

This section follows the notes from Jake VanderPlas and webpage of emcee

Here I will focus on demonstrating how to practically use Bayesian analysis in Python. In the realm of modern statistics, Markov Chain Monte Carlo (MCMC) is an approach to facilitate implementing Bayesian analysis. MCMC is a class of algorithms used to efficiently sample posterior distributions.

In this section we will learn how to use the emcee package to perform Bayesian analysis via MCMC.

So here I am not going to teach these packages, instead my goal is to teach you how to use these packages to solve a real-word problem. For more detailed information about these packages, I refer you to the above hyperlinked pages.

Here I am going to define a test problem, in which we will define a "three-parameter model" which fits a straight line to data. The parameters will be the slope, the intercept, and the scatter about the line; the scatter in this case will be treated as a nuisance parameter.

Data: let's assume the following x and y data that we will work here with.

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

np.random.seed(42)
theta_true = (25, 0.5)
xdata = 100 * np.random.random(20)
ydata = theta_true[0] + theta_true[1] * xdata

# add scatter to points
xdata = np.random.normal(xdata, 10)
ydata = np.random.normal(ydata, 10)
```

```
plt.plot(xdata, ydata, 'ok')
plt.xlabel('x')
plt.ylabel('y')

# Show the figure
plt.tight_layout()
plt.savefig('/content/drive/MyDrive/PSU/ColabNotebooks/Example4_1.png',
dpi=300)
plt.show()
```

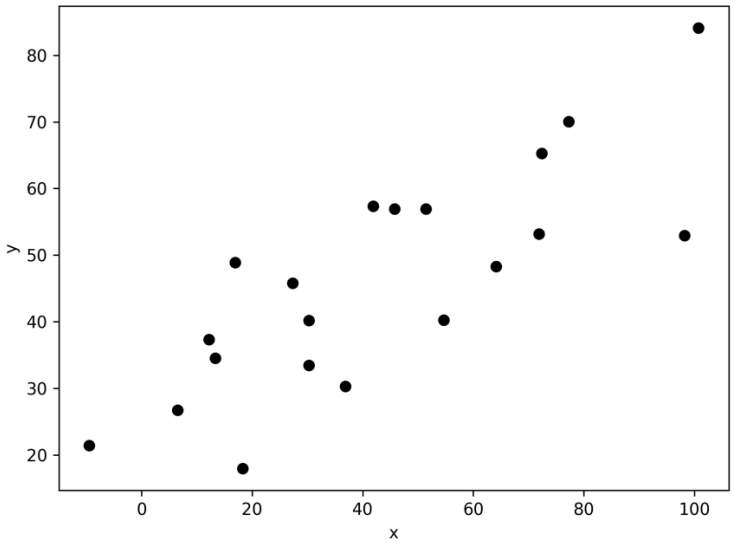


Figure 4.1

As we see here, the x and y are well correlated, let's assume that we do not know the errors. Let's build a linear regression model to fit the data.

The Model: Slope, Intercept, & Unknown Scatter: let's use the Bayes' Theorem

$$P(\theta \mid D) \propto P(D \mid \theta)P(\theta)$$

So here as we know the θ represents the model and D represents the observed data.

If we assume the following is the linear regression model that fits to data:

$$y_i = \alpha + \beta x_i$$

Then we can re-write this in the following format:

$$y(x_i \mid \alpha, \beta) = \alpha + \beta x_i$$

If we assume there is Gaussian error associated with the observed y values, the probability for any data point under this model is expressed as:

$$P(x_i y_i \mid \alpha, \beta, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[\frac{-[y_i - y(x_i \mid \alpha, \beta)]^2}{2\sigma^2}\right]$$

Here σ represents an unknown measurement error, which we'll treat as a nuisance parameter.

Multiplying these for all data points i gives the likelihood:

$$P(\{x_i\}\{y_i\} \mid \alpha, \beta, \sigma) \propto (2\pi\sigma^2)^{-N/2} \exp\left[-\frac{1}{2\sigma^2}\sum_{i=1}^{N}[y_i - y(x_i \mid \alpha, \beta)]^2\right]$$

The Prior: so here the main equation is how to choose prior. We can simply use flat prior, as we don't have any information about the model parameters, but we must keep in mind that flat priors are not always uninformative priors! The better choice is to follow Jeffreys and use symmetry and/or maximum entropy to choose maximally noninformative priors. This is something that always frequentist statisticians complain about – but this approach is well-founded and very well-supported from an information theoretic standpoint.

So, let's see why relying on flat prior is not a good choice. For example, in the case of slope β, using a flat prior we can end up with the following ensemble of lines:

```
fig, ax = plt.subplots(subplot_kw=dict(aspect='equal'))
x = np.linspace(-1, 1)

for slope in np.arange(0, 10, 0.1):
    plt.plot(x, slope * x, '-k')

ax.axis([-1, 1, -1, 1]);
# Show the figure
plt.tight_layout()
plt.savefig('/content/drive/MyDrive/PSU/ColabNotebooks/Example4_2.png',
dpi=300)
plt.show()
```

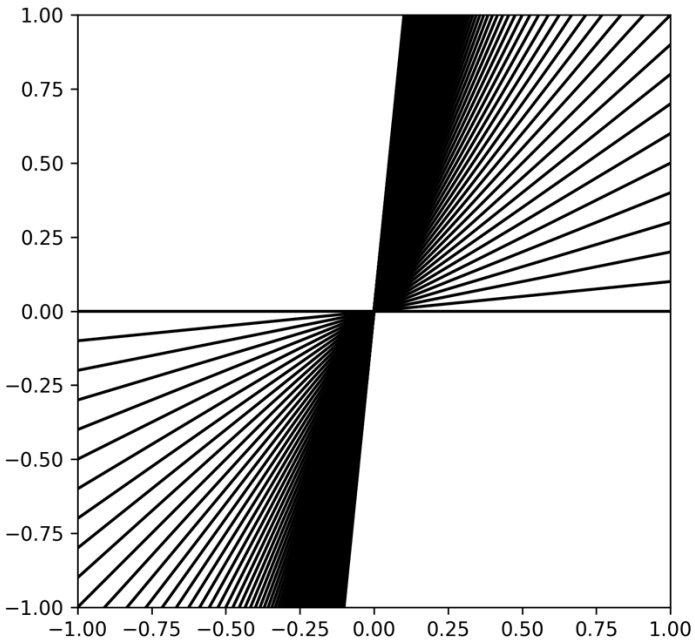


Figure 4.2

With a flat prior, you're essentially saying that any one of these slopes is just as likely as another.

Prior on Slope and Intercept: this literature suggests the following is the best choice of prior for the regression line coefficients.

$$P(\alpha, \beta) \propto (1 + \beta^2)^{-3/2}$$

Let's assume that σ follows the Jeffreys prior. The Jeffreys prior is defined as follows:

$$P(\theta) \propto \sqrt{I(\theta)}$$

$P(\theta)$ is the prior distribution for the parameter θ.
$I(\theta)$ represents the Fisher information, which quantifies the amount of information provided by the data about the parameter θ.

In the case of the parameter σ, which is the standard deviation, the Fisher information $I(\sigma)$ can be calculated as follows:

$$I(\sigma) = -\mathbb{E}[\partial^2 \log(\mathrm{P}(\mathrm{data} \mid \sigma))/\partial(\sigma)^2]$$

Here, $P(\text{data} \mid \sigma)$ represents the likelihood of the data given σ, and the expectation $\mathbb{E}[\dots]$ is taken over the data distribution. The Fisher information measures how the likelihood function changes with respect to σ.
In the specific case of a Gaussian likelihood (normal distribution) for the data, where the likelihood is:

$$P(\text{data} \mid \sigma) = (1/\sigma\sqrt{(2\pi)}) * \exp(-0.5 * (\text{data}/\sigma)^2)$$

The Fisher information for σ simplifies to:

$$I(\sigma) = 1/\sigma^2$$
$$P(\sigma) \propto \sqrt{(I(\sigma))} = \sqrt{(1/\sigma^2)} = 1/\sigma$$

Therefore, the Jeffreys prior for the standard deviation σ in a Gaussian model is indeed $P(\sigma) \propto 1/\sigma$, which is what you have indicated. This prior is chosen because it is invariant under reparameterization and is often considered non-informative, meaning it doesn't introduce any strong prior beliefs about the value of σ and allows the data to speak for itself in the Bayesian analysis.

Putting priors together, we will have the following:

$$P(\alpha, \beta, \sigma) \propto \frac{1}{\sigma}(1+\beta^2)^{-3/2}$$

To able to use the above-mentioned packages to sample posterior distributions, we must ensure the posterior is represented in a logarithmic fashion. Therefore, we must have prior in a logarithmic fashion.

$$\text{Log}[P(\alpha, \beta, \sigma)] \propto \text{Log}\left[\frac{1}{\sigma}(1+\beta^2)^{-\frac{3}{2}}\right] \propto -1.5\log(1+\beta^2) - \log(\sigma)$$

Now that we have the data, the likelihood, and the prior, let's show how to solve this problem in Python using emcee, PyMC, and PyStan. First, though, let's quickly define some convenience routines which will help us visualize the results:

```
# Create some convenience routines for plotting

def compute_sigma_level(trace1, trace2, nbins=20):
    """From a set of traces, bin by number of standard deviations"""
    L, xbins, ybins = np.histogram2d(trace1, trace2, nbins)
```

```
    L[L == 0] = 1E-16
    logL = np.log(L)

    shape = L.shape
    L = L.ravel()

    # obtain the indices to sort and unsort the flattened array
    i_sort = np.argsort(L)[::-1]
    i_unsort = np.argsort(i_sort)

    L_cumsum = L[i_sort].cumsum()
    L_cumsum /= L_cumsum[-1]

    xbins = 0.5 * (xbins[1:] + xbins[:-1])
    ybins = 0.5 * (ybins[1:] + ybins[:-1])

    return xbins, ybins, L_cumsum[i_unsort].reshape(shape)


def plot_MCMC_trace(ax, xdata, ydata, trace, scatter=False, **kwargs):
    """Plot traces and contours"""
    xbins, ybins, sigma = compute_sigma_level(trace[0], trace[1])
    ax.contour(xbins, ybins, sigma.T, levels=[0.683, 0.955], **kwargs)
    if scatter:
        ax.plot(trace[0], trace[1], ',k', alpha=0.1)
    ax.set_xlabel(r'$\alpha$')
    ax.set_ylabel(r'$\beta$')


def plot_MCMC_model(ax, xdata, ydata, trace):
    """Plot the linear model and 2sigma contours"""
    ax.plot(xdata, ydata, 'ok')

    alpha, beta = trace[:2]
    xfit = np.linspace(-20, 120, 10)
    yfit = alpha[:, None] + beta[:, None] * xfit
    mu = yfit.mean(0)
    sig = 2 * yfit.std(0)

    ax.plot(xfit, mu, '-k')
    ax.fill_between(xfit, mu - sig, mu + sig, color='lightgray')

    ax.set_xlabel('x')
    ax.set_ylabel('y')
```

```
def plot_MCMC_results(xdata, ydata, trace, colors='k'):
    """Plot both the trace and the model together"""
    fig, ax = plt.subplots(1, 2, figsize=(10, 4))
    plot_MCMC_trace(ax[0], xdata, ydata, trace, True, colors=colors)
    plot_MCMC_model(ax[1], xdata, ydata, trace)
```

4.2. emcee

The emcee package (also known as MCMC Hammer, which is in the running for best Python package name in history) is a Pure Python package written by Astronomer Dan Foreman-Mackey. It is a lightweight package which implements a sophisticated Affine-invariant Hamiltonian MCMC. Because the package is pure Python (i.e. it contains no compiled extensions) it is extremely easy to install; with pip, simply type at the command-line.

```
import emcee
print(emcee.__version__)
```

Make sure to pip install emcee if you have not done so.

```
# Define our posterior using Python functions
# for clarity, I've separated-out the prior and likelihood
# but this is not necessary. Note that emcee requires log-posterior

def log_prior(theta):
    alpha, beta, sigma = theta
    if sigma < 0:
        return -np.inf  # log(0)
    else:
        return -1.5 * np.log(1 + beta ** 2) - np.log(sigma)

def log_likelihood(theta, x, y):
    alpha, beta, sigma = theta
    y_model = alpha + beta * x
    return -0.5 * np.sum(np.log(2 * np.pi * sigma ** 2) + (y - y_model) **
2 / sigma ** 2)

def log_posterior(theta, x, y):
    return log_prior(theta) + log_likelihood(theta, x, y)
```

```
# Here we'll set up the computation. emcee combines multiple "walkers",
# each of which is its own MCMC chain. The number of trace results will
# be nwalkers * nsteps
```

```
ndim = 3  # number of parameters in the model
nwalkers = 50  # number of MCMC walkers
nburn = 1000  # "burn-in" period to let chains stabilize
nsteps = 2000  # number of MCMC steps to take

# set theta near the maximum likelihood, with
np.random.seed(0)
starting_guesses = np.random.random((nwalkers, ndim))
```

```
# Here's the function call where all the work happens:
# we'll time it using IPython's %time magic

sampler = emcee.EnsembleSampler(nwalkers, ndim, log_posterior,
args=[xdata, ydata])
%time sampler.run_mcmc(starting_guesses, nsteps)
print("done")
```

```
CPU times: user 3.86 s, sys: 3.91 ms, total: 3.86 s
Wall time: 4.4 s
done
```

```
# sampler.chain is of shape (nwalkers, nsteps, ndim)
# we'll throw-out the burn-in points and reshape:
emcee_trace = sampler.chain[:, nburn:, :].reshape(-1, ndim).T
plot_MCMC_results(xdata, ydata, emcee_trace)
# Show the figure
plt.tight_layout()
plt.savefig('/content/drive/MyDrive/PSU/ColabNotebooks/Example4_3.png',
dpi=300)
plt.show()
```

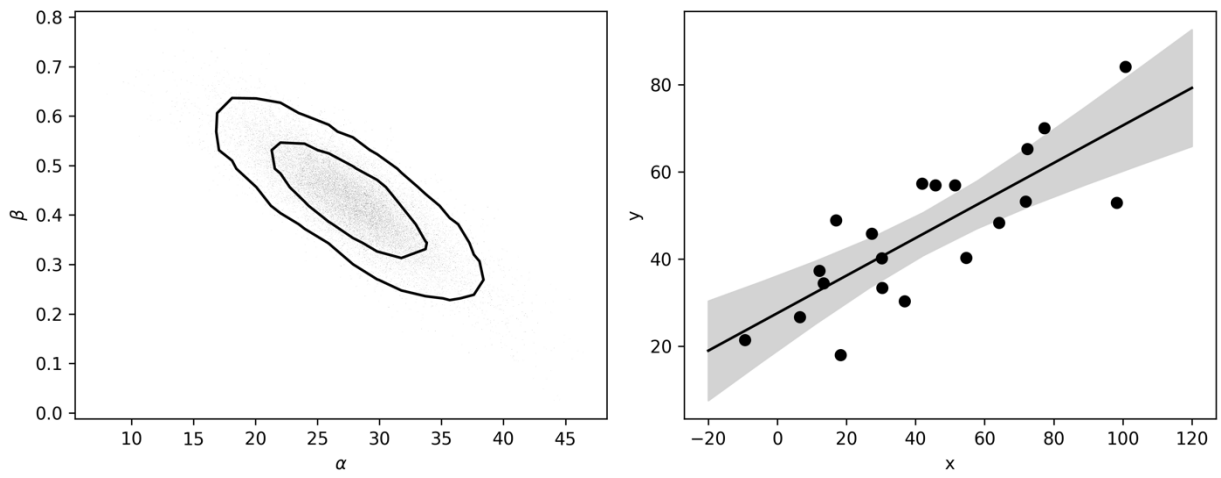


Figure 4.3

On the left we show the resulting traces marginalized over the nuisance parameter σ. On the right, we show the line of best-fit along with the 2-σ uncertainty region. This is exactly the type of result we expect from MCMC: marginalized uncertainty contours around a model which provides a good by-eye fit to the data.

4.3. Generative Probabilistic Model

When you approach a new problem, the first step is generally to write down the *likelihood function* (the probability of a dataset given the model parameters). This is equivalent to describing the generative procedure for the data. In this case, we're going to consider a linear model where the quoted uncertainties are underestimated by a constant fractional amount. You can generate a synthetic dataset from this model:

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

np.random.seed(123)

# Choose the "true" parameters.
m_true = -0.9594
b_true = 4.294
f_true = 0.534

# Generate some synthetic data from the model.
N = 50
x = np.sort(10 * np.random.rand(N))
yerr = 0.1 + 0.5 * np.random.rand(N)
```

```
y = m_true * x + b_true
y += np.abs(f_true * y) * np.random.randn(N)
y += yerr * np.random.randn(N)

plt.errorbar(x, y, yerr=yerr, fmt=".k", capsize=0)
x0 = np.linspace(0, 10, 500)
plt.plot(x0, m_true * x0 + b_true, "k", alpha=0.3, lw=3)
plt.xlim(0, 10)
plt.xlabel("x")
plt.ylabel("y");

# Show the figure
plt.tight_layout()
plt.savefig('/content/drive/MyDrive/PSU/ColabNotebooks/Example4_4.png',
dpi=300)
plt.show()
```

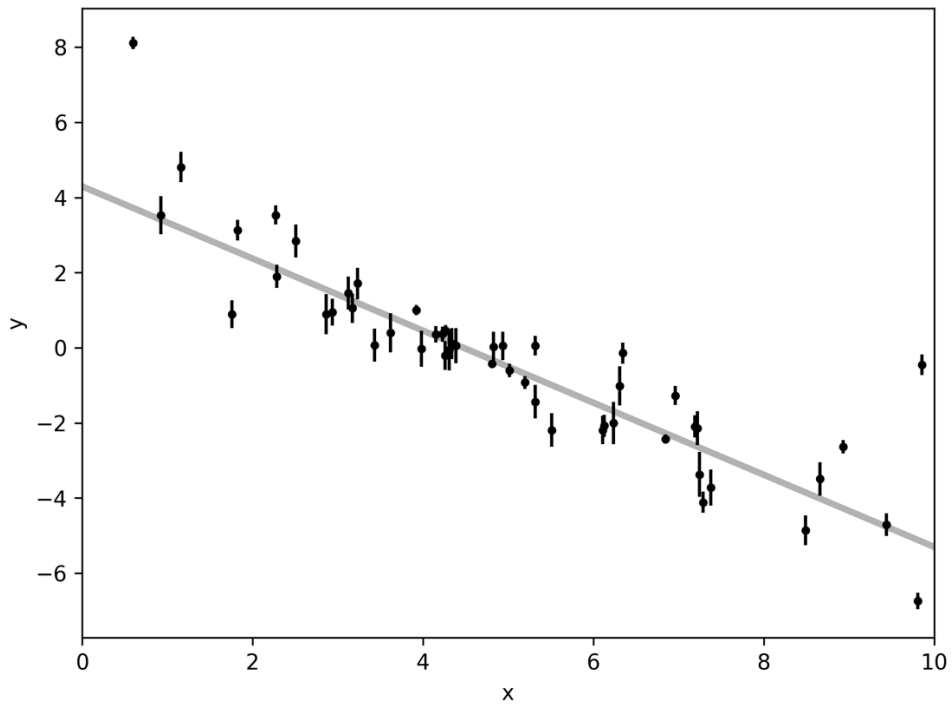


Figure 4.4

The true model is shown as the thick grey line and the effect of the underestimated uncertainties is obvious when you look at this figure. The standard way to fit a line to these data (assuming independent Gaussian error bars) is linear least squares. Linear least squares is appealing because solving for the parameters—and their associated uncertainties—is simply a linear algebraic operation.

The ***linear least squares*** solution to these data is:

```
A = np.vander(x, 2)
C = np.diag(yerr * yerr)
ATA = np.dot(A.T, A / (yerr**2)[:, None])
cov = np.linalg.inv(ATA)
w = np.linalg.solve(ATA, np.dot(A.T, y / yerr**2))
print("Least-squares estimates:")
print("m = {0:.3f} ± {1:.3f}".format(w[0], np.sqrt(cov[0, 0])))
print("b = {0:.3f} ± {1:.3f}".format(w[1], np.sqrt(cov[1, 1])))

plt.errorbar(x, y, yerr=yerr, fmt=".k", capsize=0)
plt.plot(x0, m_true * x0 + b_true, "k", alpha=0.3, lw=3, label="truth")
plt.plot(x0, np.dot(np.vander(x0, 2), w), "--k", label="LS")
plt.legend(fontsize=14)
plt.xlim(0, 10)
plt.xlabel("x")
plt.ylabel("y");

# Show the figure
plt.tight_layout()
plt.savefig('/content/drive/MyDrive/PSU/ColabNotebooks/Example4_5.png',
dpi=300)
plt.show()
```

```
Least-squares estimates:
m = -1.104 ± 0.016
b = 5.441 ± 0.091
```

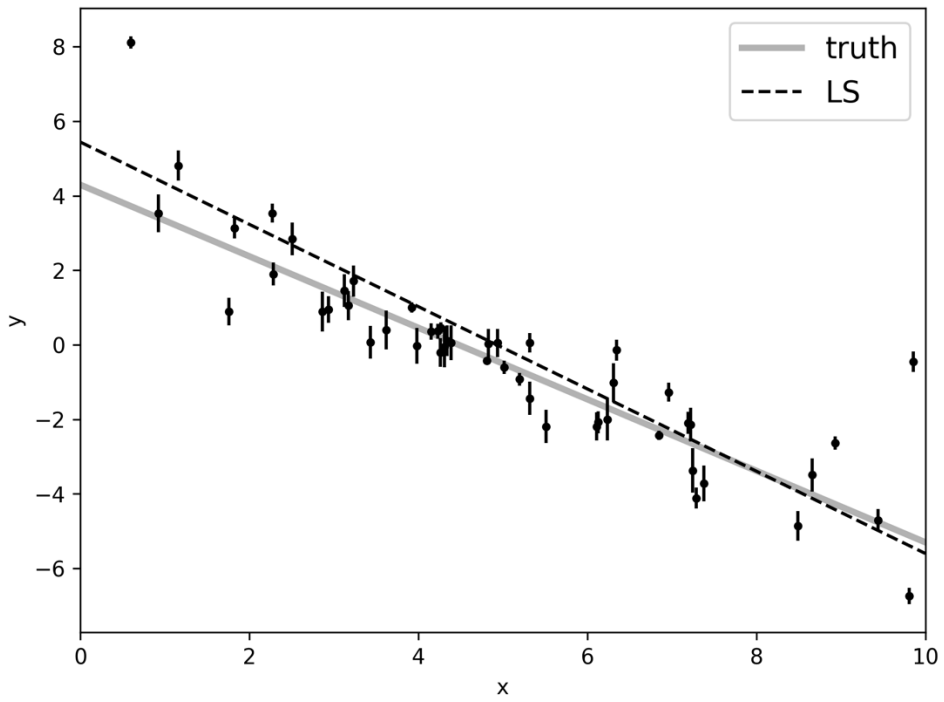


Figure 4.5

In a weighted linear regression, you have the following model:

$$y = Aw + \epsilon$$

y is the vector of observed target values.
A is the design matrix, which contains the independent variables (features).
w is the vector of model parameters to be estimated.
ϵ is the error term.

You also have the weight matrix C, where $\mathrm{C} = \mathrm{diag}(1/\sigma^2)$ and represents the standard deviation of the measurement error for each observation.

The goal in weighted least squares is to minimize the weighted sum of squared residuals:

$$\min_{\mathrm{w}}(\mathrm{y} - \mathrm{Aw})^{\mathrm{T}}\mathrm{C}(\mathrm{y} - \mathrm{Aw})$$

To find the solution, you can take the derivative of this expression with respect to w:

$$\mathrm{w} = (\mathrm{A}^{\mathrm{T}}\mathrm{CA})^{-1}\mathrm{A}^{\mathrm{T}}\mathrm{Cy}$$

In your code, the final line `w = np.linalg.solve(ATA, np.dot(A.T, y / yerr**2))`

is equivalent to this equation, where ATA is equivalent to $\mathrm{A^T CA}$. AT is equivalent to $\mathrm{A^T}$, and `y / yerr**2` is equivalent to $C^{-1}y$. This code calculates w by solving the linear system, efficiently obtaining the estimated model parameters.

C = np.diag(yerr * yerr): This line creates a diagonal matrix C where each diagonal element is the square of the corresponding element in the yerr array. This diagonal matrix is typically used to represent the weights or uncertainties associated with the observations in a least squares problem.

ATA = np.dot(A.T, A / (yerr**2)[:, None]): Here, A is presumably a matrix representing your independent variables, and A.T is its transpose. This line calculates ATA, which is the result of multiplying the transpose of A by A element-wise divided by the square of yerr. The [:, None] part is used to ensure that the division is broadcast correctly.

cov = np.linalg.inv(ATA): This line calculates the inverse of the ATA matrix, which is often used to estimate the covariance matrix of the parameters in a linear regression model. In this context, cov will represent the covariance matrix of your model parameters.

w = np.linalg.solve(ATA, np.dot(A.T, y / yerr**2)): Here, w is the vector of estimated model parameters. It is obtained by solving a linear system of equations, where the right-hand side is given by the product of the transpose of A, the target variable y (presumably, the dependent variable), and the inverse of yerr squared. This is a common step in solving linear regression problems.

In summary, these commands are typically used to perform weighted linear regression, where yerr represents the uncertainties or weights associated with each data point, and the goal is to estimate the model parameters (w) while taking these uncertainties into account. The cov matrix can be used to estimate the uncertainties in the model parameters.

This figure shows the least-squares estimate of the line parameters as a dashed line. This isn't an unreasonable result but the uncertainties on the slope and intercept seem a little small (because of the small error bars on most of the data points).

Maximum likelihood estimation: The least squares solution found in the previous section is the maximum likelihood result for a model where the error bars are assumed correct, Gaussian and independent. We know, of course, that this isn't the right model. Unfortunately, there isn't a generalization of least squares that supports a model like the one that we know to be true. Instead, we need to write down the likelihood function and numerically optimize it. In mathematical notation, the correct likelihood function is:

$$\mathrm{Log}\, P(y \mid x, \sigma, m, b, f) = -\frac{1}{2}\sum_n \left[\frac{(y_n - (mx_n + b))^2}{s_n^2} + \log\left(2\pi s_n^2\right)\right]$$

Where:

$$s_n^2 = \sigma_n^2 + f^2(mx_n + b)^2$$

This likelihood function is simply a Gaussian where the variance is underestimated by some fractional amount: f. In Python, you would code this up as:

```
def log_likelihood(theta, x, y, yerr):
    m, b, log_f = theta
    model = m * x + b
    sigma2 = yerr**2 + model**2 * np.exp(2 * log_f)
    return -0.5 * np.sum((y - model) ** 2 / sigma2 + np.log(sigma2))
```

In this code snippet, you'll notice that we're using the logarithm of f, instead of f itself. For now, it should at least be clear that this isn't a bad idea because it will force f to be always positive. A good way of finding this numerical optimum of this likelihood function is to use the scipy.optimize module:

```
from scipy.optimize import minimize

np.random.seed(42)
nll = lambda *args: -log_likelihood(*args)
initial = np.array([m_true, b_true, np.log(f_true)]) + 0.1 *
np.random.randn(3)
soln = minimize(nll, initial, args=(x, y, yerr))
m_ml, b_ml, log_f_ml = soln.x

print("Maximum likelihood estimates:")
print("m = {0:.3f}".format(m_ml))
print("b = {0:.3f}".format(b_ml))
print("f = {0:.3f}".format(np.exp(log_f_ml)))

plt.errorbar(x, y, yerr=yerr, fmt=".k", capsize=0)
plt.plot(x0, m_true * x0 + b_true, "k", alpha=0.3, lw=3, label="truth")
plt.plot(x0, np.dot(np.vander(x0, 2), w), "--k", label="LS")
plt.plot(x0, np.dot(np.vander(x0, 2), [m_ml, b_ml]), ":k", label="ML")
plt.legend(fontsize=14)
plt.xlim(0, 10)
plt.xlabel("x")
plt.ylabel("y");
```

```
# Show the figure
plt.tight_layout()
plt.savefig('/content/drive/MyDrive/PSU/ColabNotebooks/Example4_6.png',
dpi=300)
plt.show()
```

```
Maximum likelihood estimates:
m = -1.003
b = 4.528
f = 0.454
```

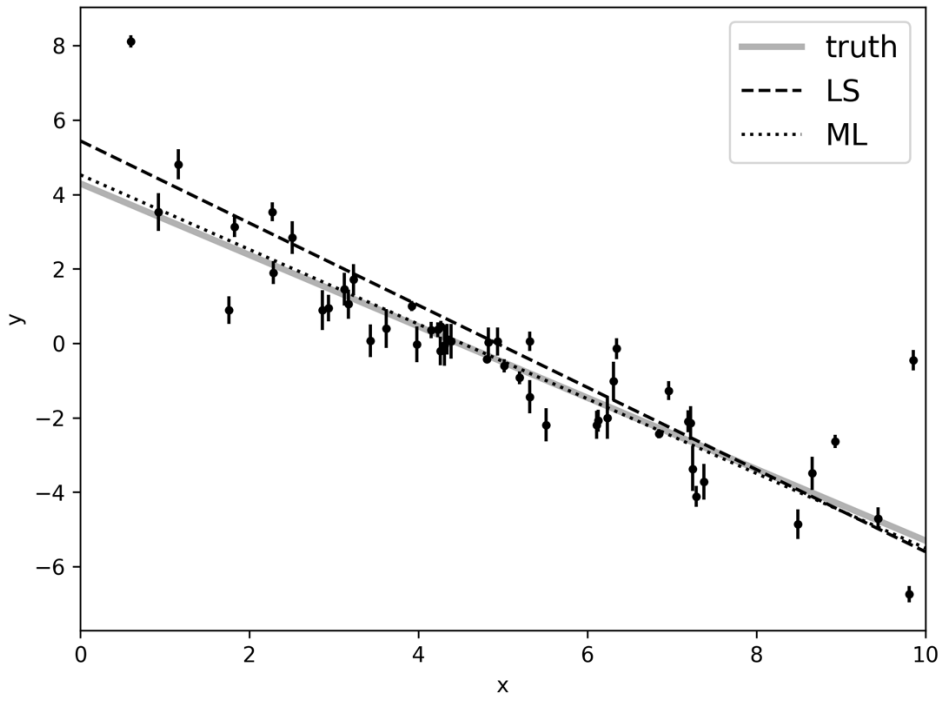


Figure 4.6

likelihood. This goal is equivalent to minimizing the negative likelihood (or in this case, the negative log likelihood). In this figure, the maximum likelihood (ML) result is plotted as a dotted black line—compared to the true model (grey line) and linear least-squares (LS; dashed line). That looks better!

The problem now: how do we estimate the uncertainties on m and b? What's more, we probably don't really care too much about the value of f but it seems worthwhile to propagate any uncertainties about its value to our final estimates of m and b. This is where MCMC comes in.

Marginalization and uncertainty estimation: This isn't the place to get into the details of why you might want to use MCMC in your research, but it is worth commenting that a common reason is that you would like to marginalize over some "nuisance parameters" and find an estimate of the posterior probability function (the distribution of parameters that is consistent with your dataset)

for others. MCMC lets you do both things in one fell swoop! You need to start by writing down the posterior probability function (up to a constant):

$$P(m, b, f \mid x, y, \sigma) \propto P(m, b, f) P(y \mid x, \sigma, m, b, f)$$

We have already, in the previous section, written down the likelihood function:

$$P(y \mid x, \sigma, m, b, f)$$

So, the missing component is the "prior" function:

$$P(m, b, f)$$

This function encodes any previous knowledge that we have about the parameters: results from other experiments, physically acceptable ranges, etc. It is necessary that you write down priors if you're going to use MCMC because all that MCMC does is draw samples from a probability distribution and you want that to be a probability distribution for your parameters. This is important: *you cannot draw parameter samples from your likelihood function*. This is because a likelihood function is a probability distribution over datasets so, conditioned on model parameters, you can draw representative datasets, but you cannot draw parameter samples.

In this example, we'll use uniform (so-called "uninformative") priors on m, b, and the logarithm of f.

```
def log_prior(theta):
    m, b, log_f = theta
    if -5.0 < m < 0.5 and 0.0 < b < 10.0 and -10.0 < log_f < 1.0:
        return 0.0
    return -np.inf
```

Then, combining this with the definition of log_likelihood from above, the full log-probability function is:

```
def log_probability(theta, x, y, yerr):
    lp = log_prior(theta)
    if not np.isfinite(lp):
        return -np.inf
    return lp + log_likelihood(theta, x, y, yerr)
```

After all this setup, it's easy to sample this distribution using emcee. We'll start by initializing the walkers in a tiny Gaussian ball around the maximum likelihood result and then run 5,000 steps of MCMC.

```
import emcee

pos = soln.x + 1e-4 * np.random.randn(32, 3)
nwalkers, ndim = pos.shape

sampler = emcee.EnsembleSampler(
    nwalkers, ndim, log_probability, args=(x, y, yerr)
)
sampler.run_mcmc(pos, 5000, progress=True);
```

```
100%|██████████| 5000/5000 [00:14<00:00, 349.47it/s]
```

Let's take a look at what the sampler has done. A good first step is to look at the time series of the parameters in the chain. The samples can be accessed using the EnsembleSampler.get_chain() method. This will return an array with the shape (5000, 32, 3) giving the parameter values for each walker at each step in the chain. The figure below shows the positions of each walker as a function of the number of steps in the chain:

```
fig, axes = plt.subplots(3, figsize=(10, 7), sharex=True)
samples = sampler.get_chain()
labels = ["m", "b", "log(f)"]
for i in range(ndim):
    ax = axes[i]
    ax.plot(samples[:, :, i], "k", alpha=0.3)
    ax.set_xlim(0, len(samples))
    ax.set_ylabel(labels[i])
    ax.yaxis.set_label_coords(-0.1, 0.5)

axes[-1].set_xlabel("step number");

# Show the figure
plt.tight_layout()
plt.savefig('/content/drive/MyDrive/PSU/ColabNotebooks/Example4_7.png',
dpi=300)
plt.show()
```

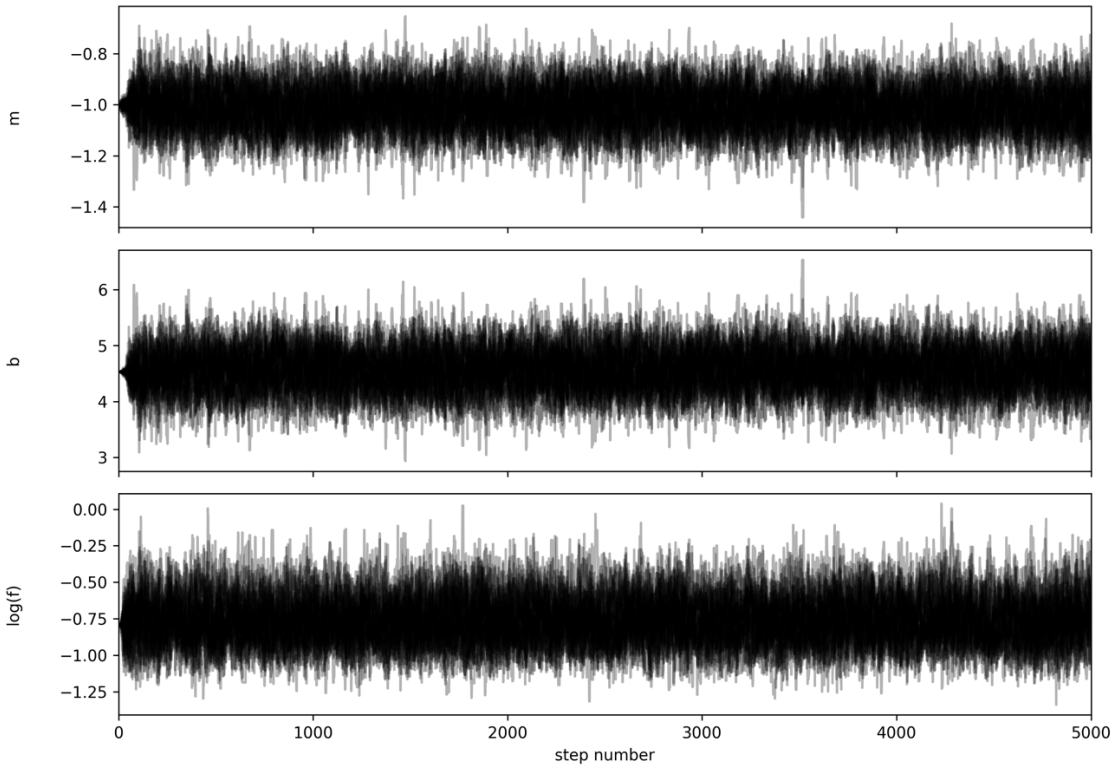


Figure 4.7

As mentioned above, the walkers start in small distributions around the maximum likelihood values and then they quickly wander and start exploring the full posterior distribution. In fact, after fewer than 50 steps, the samples seem well "burnt-in". That is a hard statement to make quantitatively, but we can look at an estimate of the integrated autocorrelation time:

```
tau = sampler.get_autocorr_time()
print(tau)
```

```
[36.03979862 35.45827512 37.02933472]
```

This suggests that only about 40 steps are needed for the chain to "forget" where it started. It's not unreasonable to throw away a few times this number of steps as "burn-in". Let's discard the initial 100 steps, thin by about half the autocorrelation time (15 steps), and flatten the chain so that we have a flat list of samples:

```
flat_samples = sampler.get_chain(discard=100, thin=15, flat=True)
print(flat_samples.shape)
```

```
(10432, 3)
```

Now that we have this list of samples, let's make one of the most useful plots you can make with your MCMC results: a corner plot. You'll need the [corner.py module] but once you have it, generating a corner plot is as simple as:

Make sure to pip install corner, and then use the following code:

```
import corner

fig = corner.corner(
    flat_samples, labels=labels, truths=[m_true, b_true, np.log(f_true)]
);

# Show the figure
plt.tight_layout()
plt.savefig('/content/drive/MyDrive/PSU/ColabNotebooks/Example4_8.png',
dpi=300)
plt.show()
```

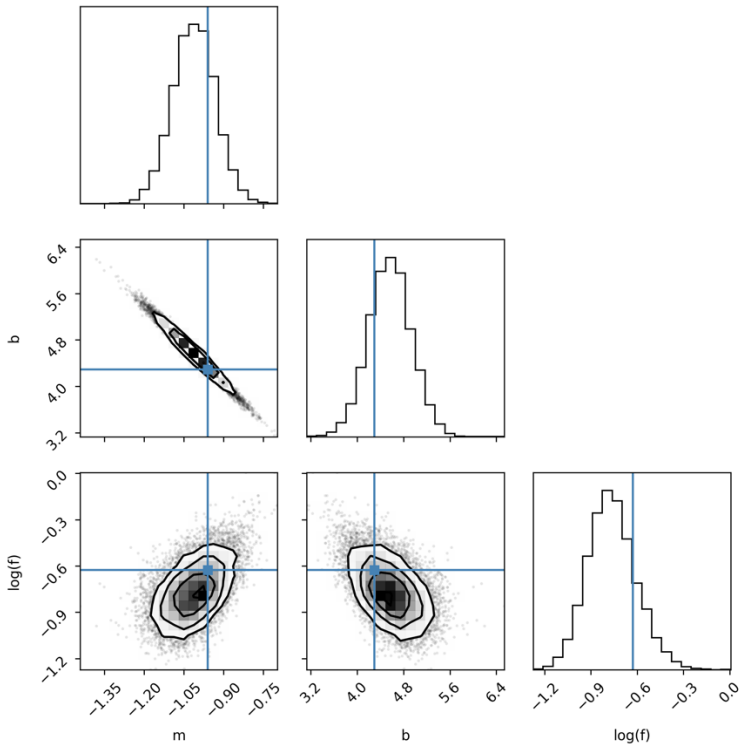


Figure 4.8

The corner plot shows all the one and two dimensional projections of the posterior probability distributions of your parameters. This is useful because it quickly demonstrates all of the covariances between parameters. Also, the way that you find the marginalized distribution for a

parameter or set of parameters using the results of the MCMC chain is to project the samples into that plane and then make an N-dimensional histogram. That means that the corner plot shows the marginalized distribution for each parameter independently in the histograms along the diagonal and then the marginalized two dimensional distributions in the other panels.

Another diagnostic plot is the projection of your results into the space of the observed data. To do this, you can choose a few (say 100 in this case) samples from the chain and plot them on top of the data points:

```
inds = np.random.randint(len(flat_samples), size=100)
for ind in inds:
    sample = flat_samples[ind]
    plt.plot(x0, np.dot(np.vander(x0, 2), sample[:2]), "C1", alpha=0.1)
plt.errorbar(x, y, yerr=yerr, fmt=".k", capsize=0)
plt.plot(x0, m_true * x0 + b_true, "k", label="truth")
plt.legend(fontsize=14)
plt.xlim(0, 10)
plt.xlabel("x")
plt.ylabel("y");

# Show the figure
plt.tight_layout()
plt.savefig('/content/drive/MyDrive/PSU/ColabNotebooks/Example4_9.png',
dpi=300)
plt.show()
```

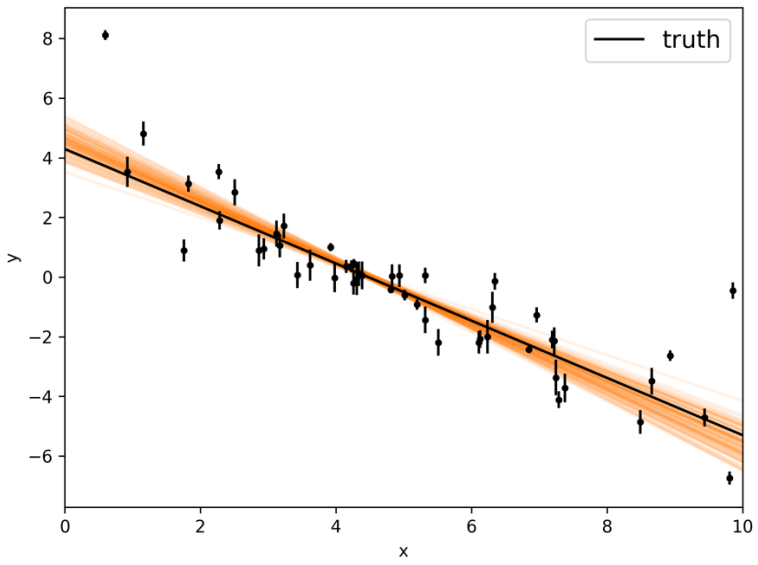


Figure 4.9

This leaves us with one question: which numbers should go in the abstract? There are a few different options for this but my favorite is to quote the uncertainties based on the 16th, 50th, and 84th percentiles of the samples in the marginalized distributions. To compute these numbers for this example, you would run:

```
from IPython.display import display, Math

for i in range(ndim):
    mcmc = np.percentile(flat_samples[:, i], [16, 50, 84])
    q = np.diff(mcmc)
    txt = "\mathrm{{{3}}} = {0:.3f}_{{-{1:.3f}}}^{{{2:.3f}}}"
    txt = txt.format(mcmc[1], q[0], q[1], labels[i])
    display(Math(txt))
```

$$\mathrm{m} = -1.011^{0.078}_{-0.079}$$
$$\mathrm{b} = 4.565^{0.362}_{-0.359}$$
$$\log(\mathrm{f}) = -0.770^{0.162}_{-0.146}$$

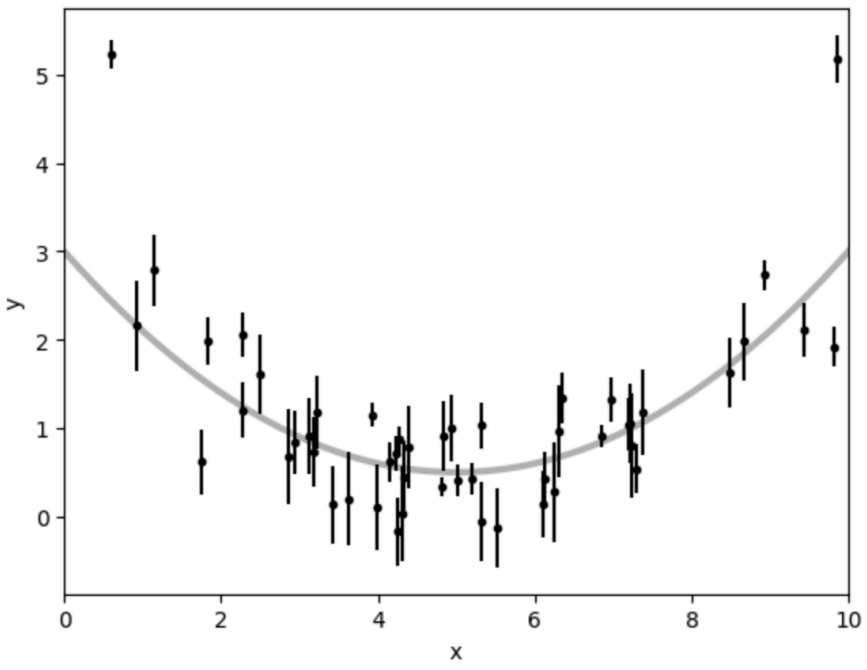
4.4 Homework #4 (10 points)

1- In section "Generative Probabilistic Model", we used "`w = np.linalg.solve(ATA, np.dot(A.T, y / yerr**2)`" to find the model parameters and fit a line to data. Use what we learned in chapter 2, least square fitting (page 7), to find the model parameters and fit a line to data. Provide your Python script and visualize the results like what we did in Chapter 2, page 8. ***(2 points)***

2- For the following synthetic data and quadratic model:

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

np.random.seed(123)

# Choose the "true" parameters for a quadratic model.
a_true = 0.1
b_true = -1.0
c_true = 3.0
f_true = 0.5

# Generate some synthetic data from the quadratic model.
N = 50
```

```
x = np.sort(10 * np.random.rand(N))
yerr = 0.1 + 0.5 * np.random.rand(N)
y = a_true * x**2 + b_true * x + c_true
y += np.abs(f_true * y) * np.random.randn(N)
y += yerr * np.random.randn(N)

# Create a scatter plot with error bars
plt.errorbar(x, y, yerr=yerr, fmt=".k", capsize=0)

# Generate the quadratic curve based on the "true" parameters
x0 = np.linspace(0, 10, 500)
y0 = a_true * x0**2 + b_true * x0 + c_true

# Plot the quadratic curve
plt.plot(x0, y0, "k", alpha=0.3, lw=3)

plt.xlim(0, 10)
plt.xlabel("x")
plt.ylabel("y")

# Show the plot
plt.show()
```



Redo all the steps in the Generative Probabilistic Model section. Use these uniform priors for model parameters (0<a<3, -6<b<6, 2<c<5, and -2<f<2): ***(8 points)***