mcmc_fit_with_outliers_pymc

October 3, 2019

1 Fitting a line to data using MCMC

In this example we will be going over Exercise 6 from Hogg 2010. We will be fitting a line to data using a model that rejects outliers using an MCMC sampler.

1.1 Packages being used

- `numpy`: doing math on arrays
- `pymc3`: this does the heavy lifting for the MCMC code
- `matplotlib`: plot our results
- `seaborn`: useful plotting functions
- `python-graphviz`: plotting pymc models as graphs
- `pandas`: read in data table

1.2 Relevant documentation

- introduction to probabilistic programming: https://nbviewer.jupyter.org/github/CamDavidsonPilon/Pro Programming-and-Bayesian-Methods-for-Hackers/blob/master/Prologue/Prologue.ipynb
- `pymc3`: https://pymc3.readthedocs.io/en/latest/notebooks/GLM-robust-with-outlier-detection.html#Create-Robust-Model-with-Outliers:-Hogg-Method
- `matplotlib`: http://matplotlib.org/2.0.2/api/pyplot_summary.html
- how to pick priors: https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations

[9]:
```
import numpy as np
from matplotlib import pyplot as plt
import pymc3 as pm
import theano.tensor as tt
import pandas
import seaborn
import mpl_style
%matplotlib inline
plt.style.use(mpl_style.style1)
seaborn.axes_style(mpl_style.style1);
```

1.2.1 Read in the data

First lets read in the data we will be fitting:

[2]:
```
data = pandas.read_csv('data.csv')

x_mean = data.x.mean()
x_std = data.x.std()
y_mean = data.y.mean()
y_std = data.y.std()

# center and scale data
x_center = (data.x - x_mean) / x_std
y_center = (data.y - y_mean) / y_std
sy_center = data.sy / y_std
sx_center = data.sx / x_std

# data order
idx = data.x.argsort()
```

1.2.2 What does centering the data do?

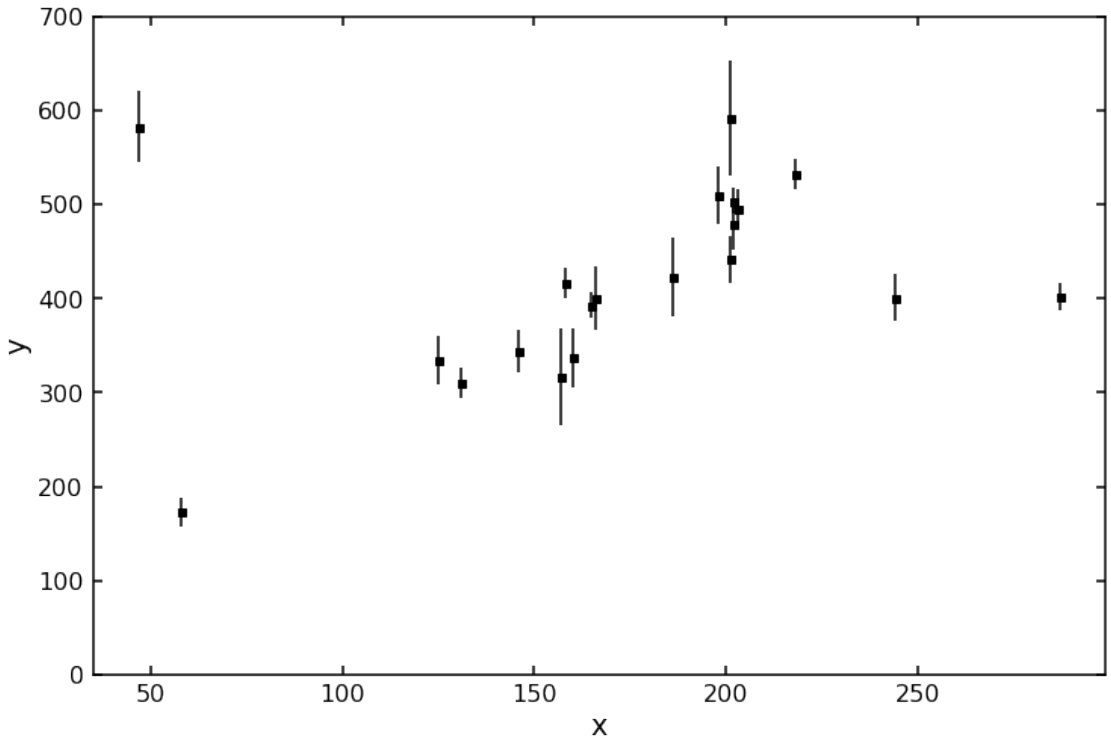
We will be fitting a line to our data with a slope and an intercept. In the original data space these two value are highly correlated, a small change in slope will result in a large change in the intercept and vice-versa. By centering and scaling the data we are ensuring the intercept is close to 0 and the x and y values are about the same size. This reduces the correlation between the two fit parameters (e.g. you can adjust the slope and keep the intercept the same).

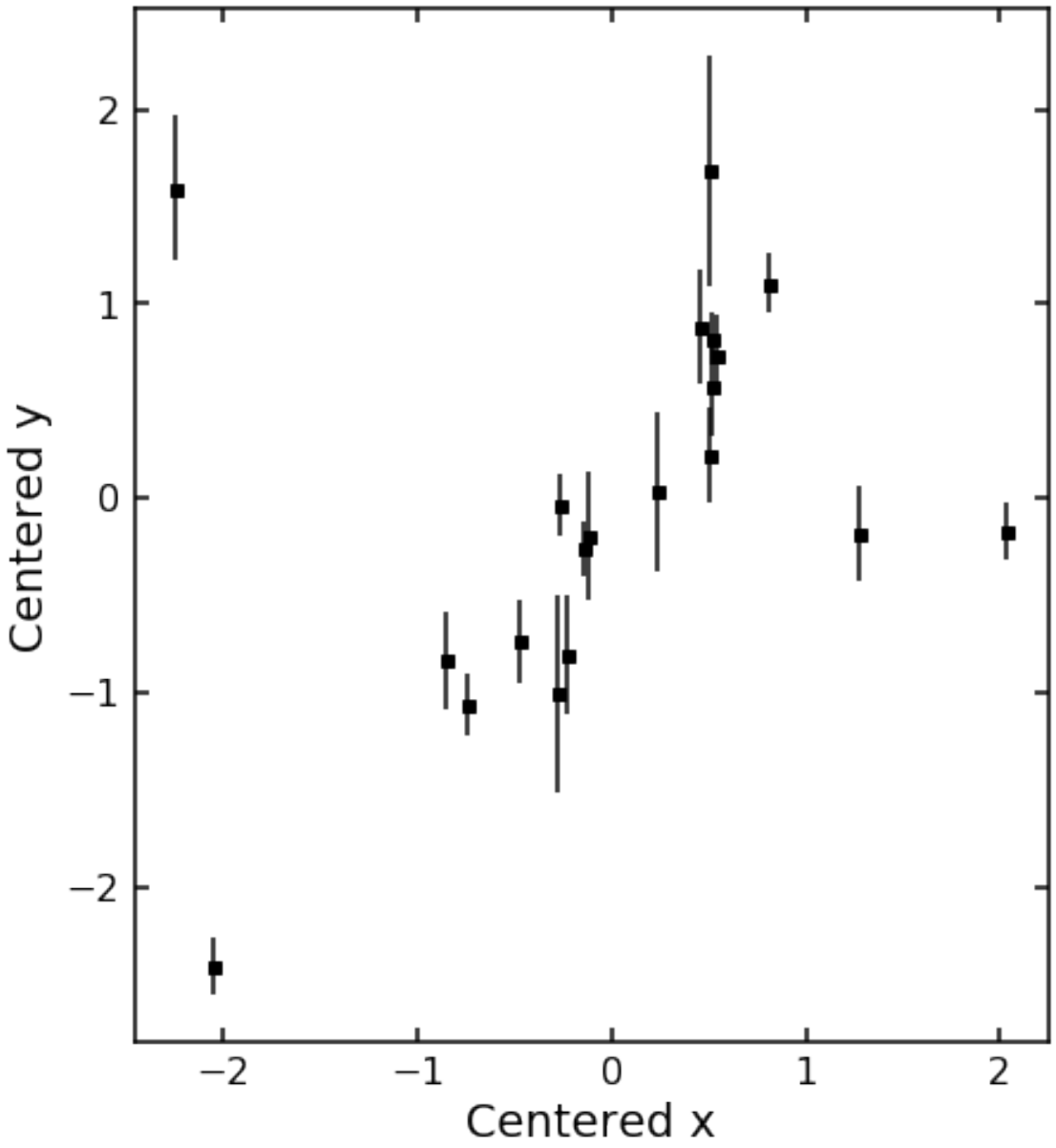
1.2.3 Plot the data

Lets take a look at our data to see what we are fitting:

[10]:
```
plt.figure(1, figsize=(12, 8))
plt.errorbar(
    data.x,
    data.y,
    data.sy,
    ls='None',
    mfc='k',
    mec='k',
    ms=5,
    marker='s',
    ecolor='k'
)
plt.xlabel('x')
plt.ylabel('y')
plt.ylim(0, 700)

plt.figure(12, figsize=(12, 8))
```

```
plt.errorbar(
    x_center,
    y_center,
    sy_center,
    ls='None',
    mfc='k',
    mec='k',
    ms=5,
    marker='s',
    ecolor='k'
)
plt.xlabel('Centered x')
plt.ylabel('Centered y')
plt.gca().set_aspect(1);
```





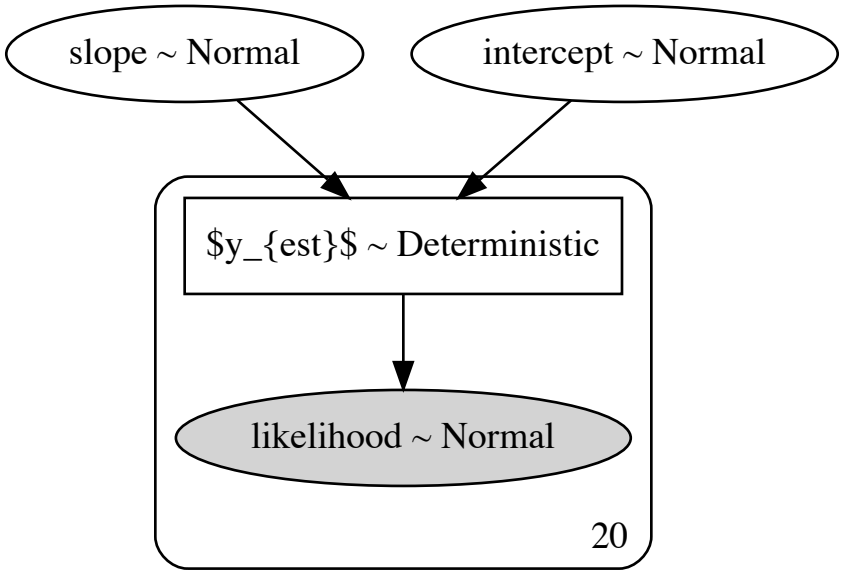
1.3 OLS Model

As a starting point we will first build a least squares linear regression model and see how that does.

[4]:
```
with pm.Model() as mdl_ols:
    # Define weakly informative Normal priros (Ridge regression)
    b0 = pm.Normal('intercept', mu=0, sd=100)
    b1 = pm.Normal('slope', mu=0, sd=100)
    # Define y_est as a Deterministic variable so we can plot it later
    y_est = pm.Deterministic(r'$y_{est}$', b0 + b1 * x_center)
```

```
    likelihood = pm.Normal('likelihood', mu=y_est, sd=sy_center,␣
 ↪observed=y_center)

display(mdl_ols)
display(pm.model_to_graphviz(mdl_ols))
```

$$
\begin{aligned}
\text{intercept} &\sim \text{Normal}(mu = 0,\ sd = 100.0) \\
\text{slope} &\sim \text{Normal}(mu = 0,\ sd = 100.0) \\
y_{est} &\sim \text{Deterministic}(\text{intercept},\ \text{slope},\ \text{Constant}) \\
\text{likelihood} &\sim \text{Normal}(mu = y_{est},\ sd = array)
\end{aligned}
$$



We can see the likelihood of the fit given the data is defined as a Normal distribution with scatter defined by observed y-errors. The priors chosen for the slope and intercept are weakly informative Normal priors. The use of these particular priors is called Ridge regression.

1.3.1 Run MCMC

Now we can run the MCMC sampler to find the best fit.

[5]:
```
with mdl_ols:
    traces_ols = pm.sample(3000, tune=2000, chains=3, cores=3)
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (3 chains in 3 jobs)
NUTS: [slope, intercept]
Sampling 3 chains: 100%|| 15000/15000 [00:08<00:00, 1715.27draws/s]
```

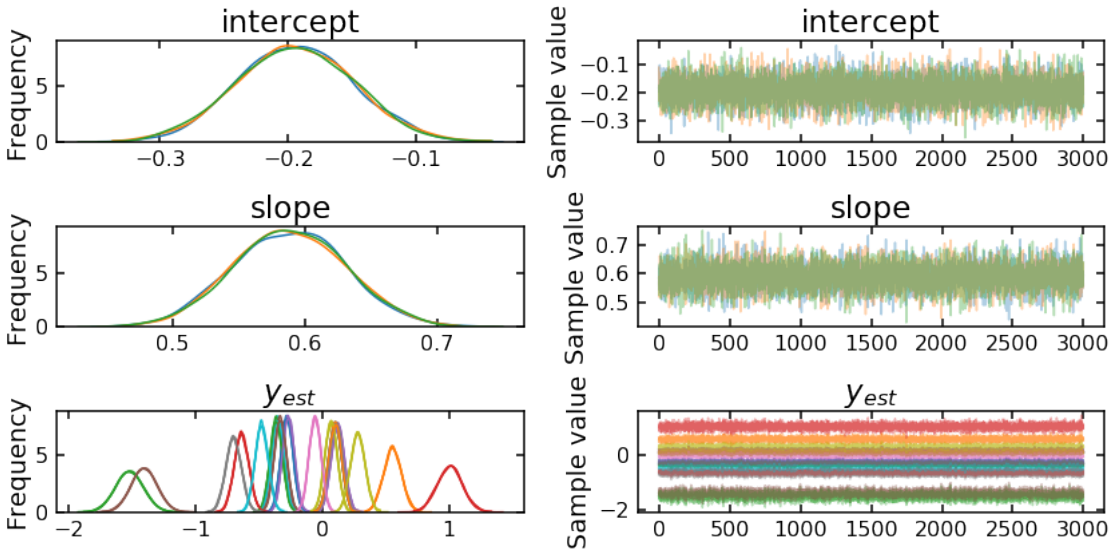
The default sampler is NUTS (No U-True Sampler), a powerful sampler that uses the likelihood's gradient information to propose new MC steps.

1.3.2 Check for convergence

Lets look at the results and see if the MCMC converged:

[11]:
```
display(pm.summary(traces_ols, varnames=['intercept', 'slope']).round(4))
pm.traceplot(traces_ols);
```

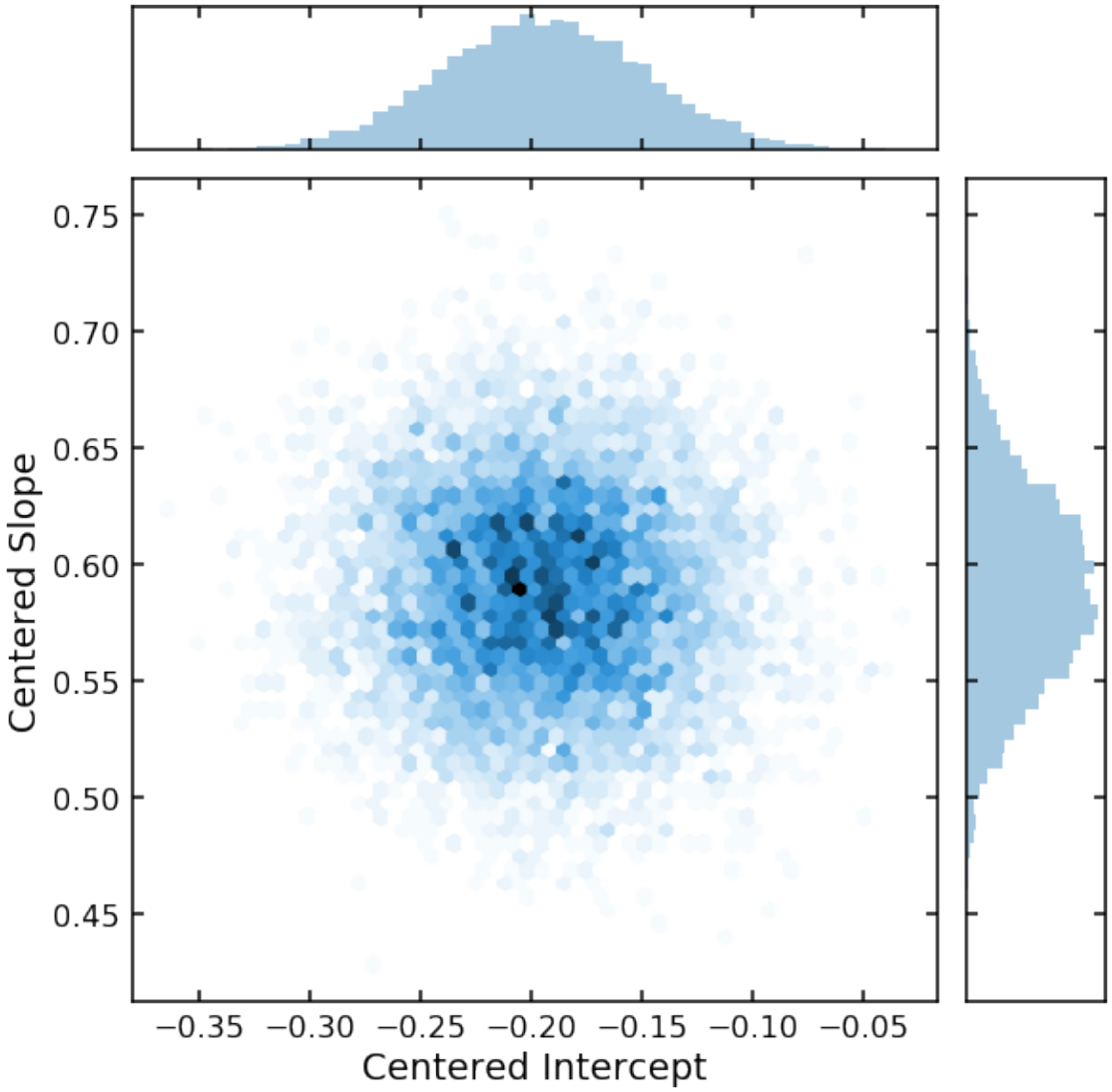
	mean	sd	mc_error	hpd_2.5	hpd_97.5	n_eff	Rhat
intercept	-0.1940	0.0456	0.0004	-0.2816	-0.1032	9048.5884	0.9999
slope	0.5888	0.0425	0.0004	0.5089	0.6727	9084.8458	0.9999



Everything looks good, the `Rhat` values are all close to 1, and the traces have all mixed well. Since we named the `y_est` variable it is also tracked in every sampling step. This makes plotting the results easier since we don't need to evaluate the model for every `b0` and `b1` value in the sample.

Lets take a look at the covariance between the fitted intercept and slope.

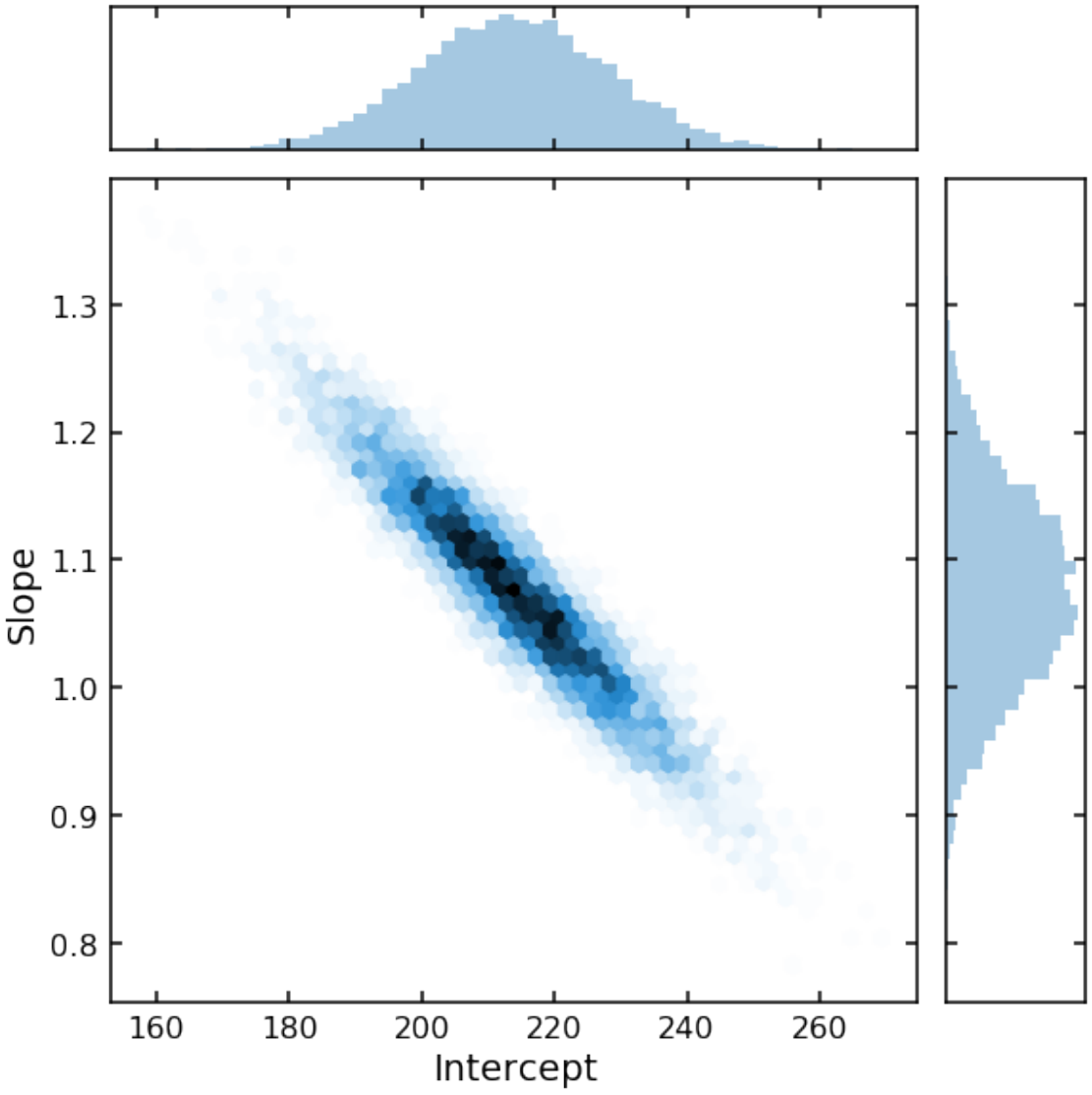
[12]:
```
fig = seaborn.jointplot(
    traces_ols['intercept'],
    traces_ols['slope'],
    kind='hex',
    height=8,

)
fig.set_axis_labels('Centered Intercept', 'Centered Slope')
seaborn.despine(top=False, right=False)
```



Notice how in the centered parameter space these two values are not correlated (as expected), let transform the fit back to the original data space.

[13]:
```
def un_center(b0_prime, b1_prime, x_mean, x_std, y_mean, y_std):
    b0 = (b0_prime * y_std) - (b1_prime * x_mean * y_std/x_std) + y_mean
    x = x_mean - (x_std / b1_prime) * ((y_mean / y_std) + b0_prime)
    b1 = -b0 / x
    return b0, b1

b0, b1 = un_center(traces_ols['intercept'], traces_ols['slope'], x_mean, x_std,␣
 ↪y_mean, y_std)

fig = seaborn.jointplot(
    b0,
    b1,
```

```
    kind='hex',
    height=8,

)
fig.set_axis_labels('Intercept', 'Slope')
seaborn.despine(top=False, right=False)
```



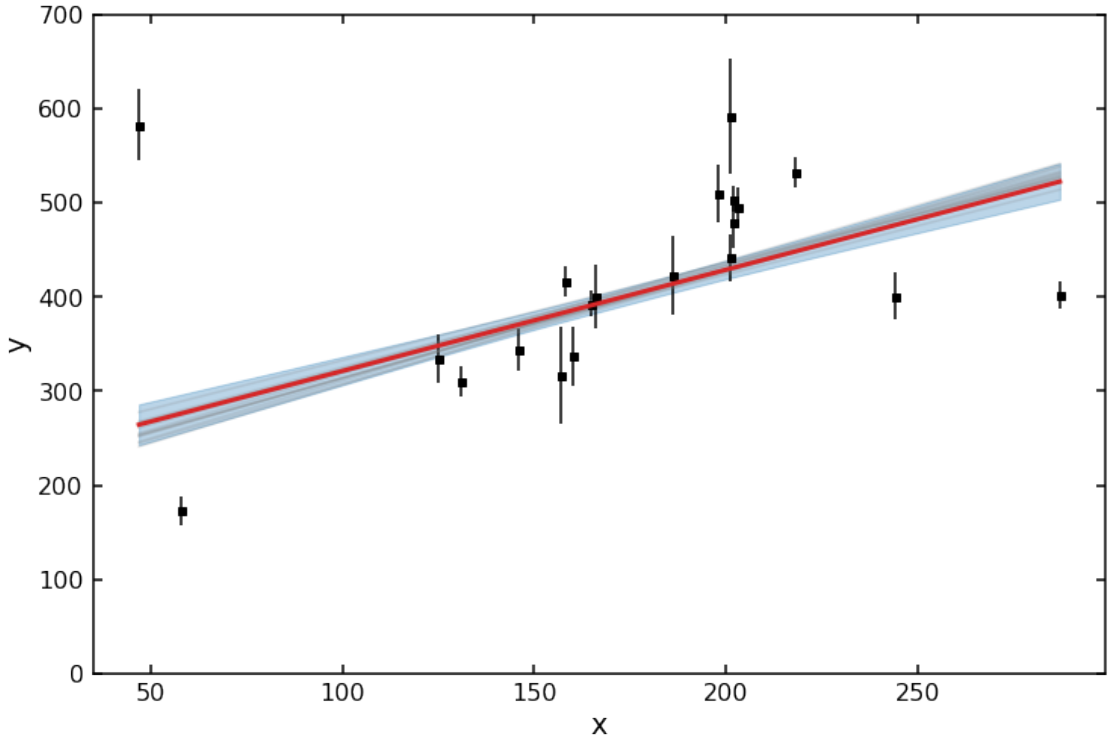
Now we can see a large correlation in the fit parameters. If we tried to run the MCMC sampler in this data space it would have issues.

1.3.3 Plotting the best fit

Finally we can plot the best fit on the original data.

[14]:
```
# uncenter and scale
y_est = (traces_ols['$y_{est}$'] * y_std) + y_mean

# plot original data
plt.figure(2, figsize=(12, 8))
plt.errorbar(
    data.x,
    data.y,
    data.sy,
    ls='None',
    mfc='k',
    mec='k',
    ms=5,
    marker='s',
    ecolor='k'
)

# find 2-sigma and meadian of best fit lines
y_est_minus_2_sigma, y_est_median, y_est_plus_2_sigma = np.percentile(
    y_est[:, idx],
    [2.5, 50, 97.5],
    axis=0
)

# plot the mean of all best fit lines
plt.plot(
    data.x[idx],
    y_est_median,
    color='C3',
    lw=3,
    zorder=3
)

# plot 2-sigma best fit region
plt.fill_between(
    data.x[idx],
    y_est_minus_2_sigma,
    y_est_plus_2_sigma,
    color='C0',
    alpha=0.3,
    zorder=1
)

# plot a selection of best fit lines
plt.plot(
    data.x[idx],
    y_est[::600].T[idx],
```

```
    alpha=0.2,
    color='C7'
)

plt.xlabel('x')
plt.ylabel('y')
plt.ylim(0, 700);
```



From this plot we can see that the fitted line is being effected by several outliers in the data. Lets build a model that will filter out these outliers.

1.4 Mixture model

To use `pymc3` we need to define the likelihood function we are trying to maximize. The likelihood we need for this fit is shown in equation 13 of Hogg 2010:

$$\mathscr{L} \propto \prod_{i=1}^{N} \left[\frac{1}{\sqrt{2\pi\sigma_{yi}^2}} \exp\left(-\frac{[y_i - mx_i - b]^2}{2\sigma_{yi}^2} \right) \right]^{[1-q_i]} \times \left[\frac{1}{\sqrt{2\pi[V_b + \sigma_{yi}^2]}} \exp\left(-\frac{[y_i - Y_b]^2}{2[V_b + \sigma_{yi}^2]} \right) \right]^{q_i}$$

$$\{q_i\} \sim \text{Bernoulli}(P_b)$$

where x_i, y_i, σ_{yi} are the data from the `.csv` file, m, b are the slope and intercept of the line we are fitting, q_i is 1 if a point is an outlier and 0 otherwise. The set of these flags follow a Bernoulli

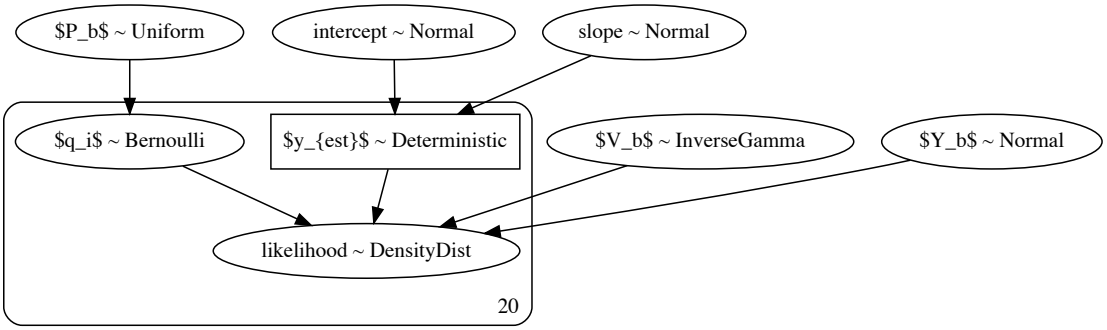
distribution with a probability of being an outlier of P_b. Y_b, V_b are the parameters of the distribution the outliers are draw from. See section 3 of Hogg's paper for a full derivation.

Since we will be using a Bayesian approach to this fitting, we need to define priors for our fit parameters $\theta = [m, b, P_b, \{q_i\}, Y_b, V_b]$. We will use weakly informative priors in all the parameters.

[15]:
```
from types import SimpleNamespace
# User the theano.tensor versions of all the function calls
# This ensures automatic derivatives can be taken at function evaluation
def logp_signoise(y_obs, y_est, sigma_y, qi, Yb, Vb):
    # likelihood for inliers
    pdfs_in = tt.exp(-(y_obs - y_est + 1e-4)**2 / (2 * sigma_y**2))
    pdfs_in /= tt.sqrt(2 * np.pi * sigma_y**2)
    logL_in = tt.sum(tt.log(pdfs_in) * (1 - qi))
    # likelihood for outliers
    pdfs_out = tt.exp(-(y_obs - Yb + 1e-4)**2 / (2 * (sigma_y**2 + Vb)))
    pdfs_out /= tt.sqrt(2 * np.pi * (Vb + sigma_y**2))
    logL_out = tt.sum(tt.log(pdfs_out) * qi)
    return logL_in + logL_out

with pm.Model() as mdl_signoise:
    # Define weakly informative Normal priros (Ridge regression)
    b0 = pm.Normal('intercept', mu=0, sd=100)
    b1 = pm.Normal('slope', mu=0, sd=100)
    # Define y_est as a Deterministic variable so we can plot it later
    y_est = pm.Deterministic(r'$y_{est}$', b0 + b1 * x_center)
    # Define weakly informative priors for the mean and variance of the outliers
    Yb = pm.Normal(r'$Y_b$', mu=0, sd=10)
    Vb = pm.InverseGamma(r'$V_b$', 2, 5)
    # Define Bernoulli inlier / outlier probability (Pb) with uniform prior
    Pb = pm.Uniform(r'$P_b$', lower=0.0, upper=1.0, testval=0.5)
    qi = pm.Bernoulli(
        r'$q_i$',
        p=Pb,
        shape=len(data.x),
        testval=np.random.rand(len(data.x)) < 0.5
    )
    # User custom likelihood
    likelihood = pm.DensityDist(
        'likelihood',
        logp_signoise,
        observed={
            'y_obs': y_center,
            'y_est': y_est,
            'sigma_y': sy_center,
            'qi': qi,
            'Yb': Yb,
            'Vb': Vb
        }
```

```
    )
    # defind the latex representation of the custom liklihood
    likelihood.__latex__ = lambda : r'$\text{likelihood} \sim␣
↪\prod{\left[(1-P_{b}) * \text{Normal}(mu=y_{est}, sd=\sigma_{y}) + P_{b} *␣
↪\text{Normal}(mu=Y_{b}, sd=\sqrt{V_{b} + \sigma_{y}^2})\right]}$'
    likelihood.tag = SimpleNamespace(test_value = data.x)

display(mdl_signoise)
display(pm.model_to_graphviz(mdl_signoise))
```

$$
\begin{aligned}
\text{intercept} &\sim \text{Normal}(mu = 0,\ sd = 100.0) \\
\text{slope} &\sim \text{Normal}(mu = 0,\ sd = 100.0) \\
Y_b &\sim \text{Normal}(mu = 0,\ sd = 10.0) \\
q_i &\sim \text{Bernoulli}(p = P_b) \\
y_{est} &\sim \text{Deterministic}(\text{intercept},\ \text{slope},\ \text{Constant}) \\
V_b &\sim \text{InverseGamma}(alpha = 2,\ beta = 5) \\
P_b &\sim \text{Uniform}(lower = 0.0,\ upper = 1.0) \\
\text{likelihood} &\sim \prod \left[(1 - P_b) * \text{Normal}(mu = y_{est}, sd = \sigma_y) + P_b * \text{Normal}(mu = Y_b, sd = \sqrt{V_b + \sigma_y^2})\right]
\end{aligned}
$$



[17]:
```
with mdl_signoise:
    traces_signoise = pm.sample(
        3000,
        tune=2000,
        target_accept=0.999,
        chains=3,
        cores=3
    )
```

```
Multiprocess sampling (3 chains in 3 jobs)
CompoundStep
>NUTS: [$P_b$, $V_b$, $Y_b$, slope, intercept]
>BinaryGibbsMetropolis: [$q_i$]
Sampling 3 chains: 100%|| 15000/15000 [00:37<00:00, 402.50draws/s]
```

If you see any warning messages about divergent chains it typically means one or more of your priors are not set to reasonable distributions, or some of your parameters have high covariance and should be rescaled. In the above example if the prior on V_b is set to be `Uniform` on the the $\log(V_b)$ (as suggested by Hogg 2010) it will lead to divergent chains, but changing this to be an `InverseGamma` function clears up the issues. Even with this change we also need to increase `target_accept` from its default of 0.8 to 0.999 to avoid divergent chains.

1.4.1 Note about priors

Often times the priors you use will effect how fast the sampler will run. If you are getting a small number of draws/s try changing all your priors to `Normal` distributions to see if it runs any faster (`Uniform` priors can be very slow if they cover a large range).

See https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations for other tips about picking priors.

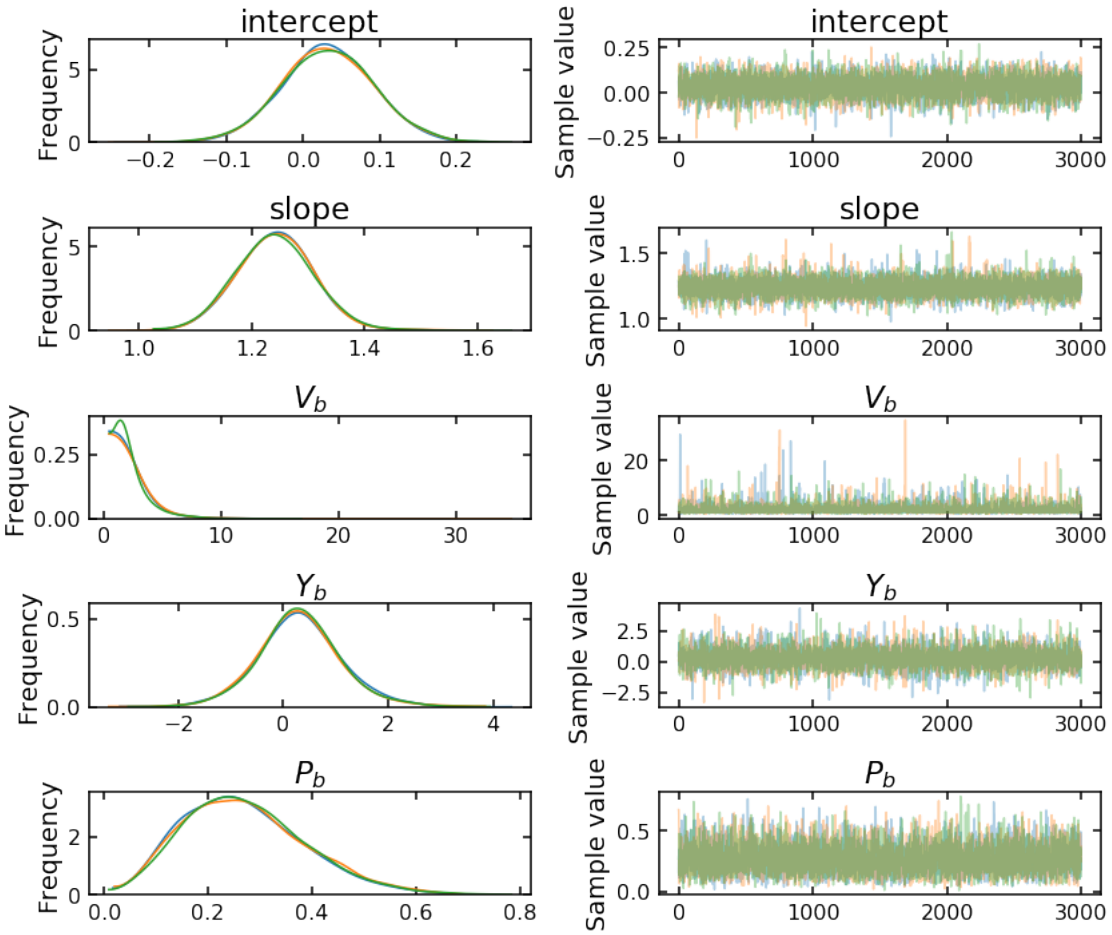
1.4.2 Check for convergence

[18]:
```
display(pm.summary(
    traces_signoise,
    varnames=[
        'intercept',
        'slope',
        '$V_b$',
        '$Y_b$',
        '$P_b$'
    ]
).round(4))
pm.traceplot(
    traces_signoise,
    varnames=[
        'intercept',
        'slope',
        '$V_b$',
        '$Y_b$',
        '$P_b$'
    ]
);
```

```
/Users/coleman/anaconda3/lib/python3.7/site-packages/pymc3/stats.py:974:
FutureWarning: The join_axes-keyword is deprecated. Use .reindex or
.reindex_like on the result to achieve the same functionality.
  axis=1, join_axes=[dforg.index])
```

	mean	sd	mc_error	hpd_2.5	hpd_97.5	n_eff	Rhat
intercept	0.0321	0.0599	0.0008	-0.0837	0.1521	4510.6458	1.0000
slope	1.2433	0.0684	0.0010	1.1113	1.3697	5614.7896	0.9999
V_b	2.3639	1.9070	0.0296	0.5333	5.5260	3565.2595	1.0001
Y_b	0.3175	0.7855	0.0104	-1.2575	1.9378	5206.3833	1.0001

```
$P_b$       0.2720   0.1186     0.0019     0.0589     0.5016   3640.7975   1.0003
```

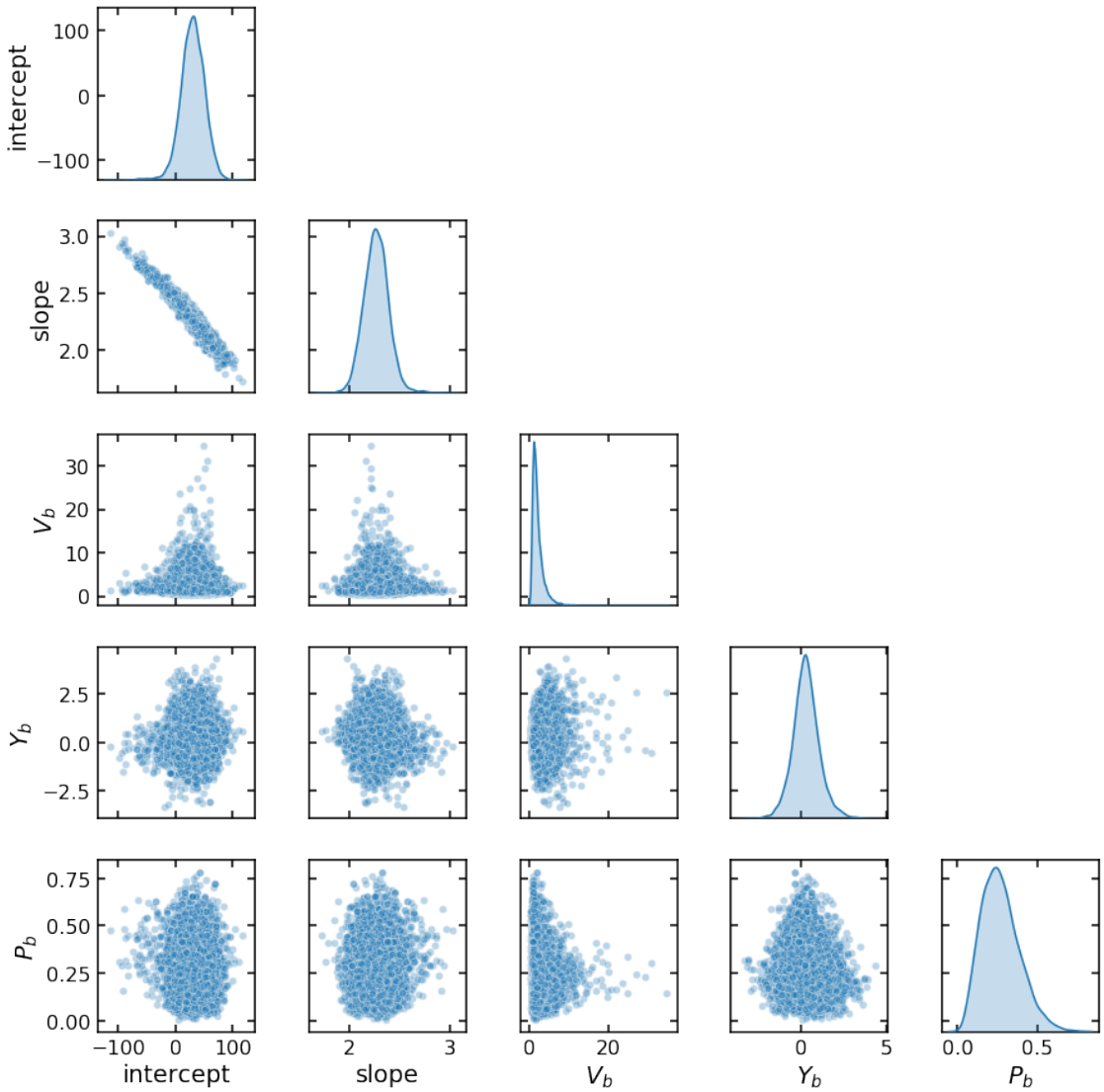


As before we can see the `Rhat` values are all close to 1 and all the chains are well mixed.

Lets take a look at a corner plot of these fit variables after converting the slope and intercept back into the original data space.

[19]:
```
df = pm.trace_to_dataframe(traces_signoise)[
    [
        'intercept',
        'slope',
        '$V_b$',
        '$Y_b$',
        '$P_b$'
    ]
]

b0, b1 = un_center(df['intercept'], df['slope'], x_mean, x_std, y_mean, y_std)

df['intercept'] = b0
```

```
df['slope'] = b1

def hide_current_axis(*args, **kwargs):
    plt.gca().set_visible(False)


g = seaborn.pairplot(
    df,
    diag_kind='kde',
    plot_kws={'alpha': 0.3}
)
g.map_upper(hide_current_axis)
seaborn.despine(top=False, right=False)
```

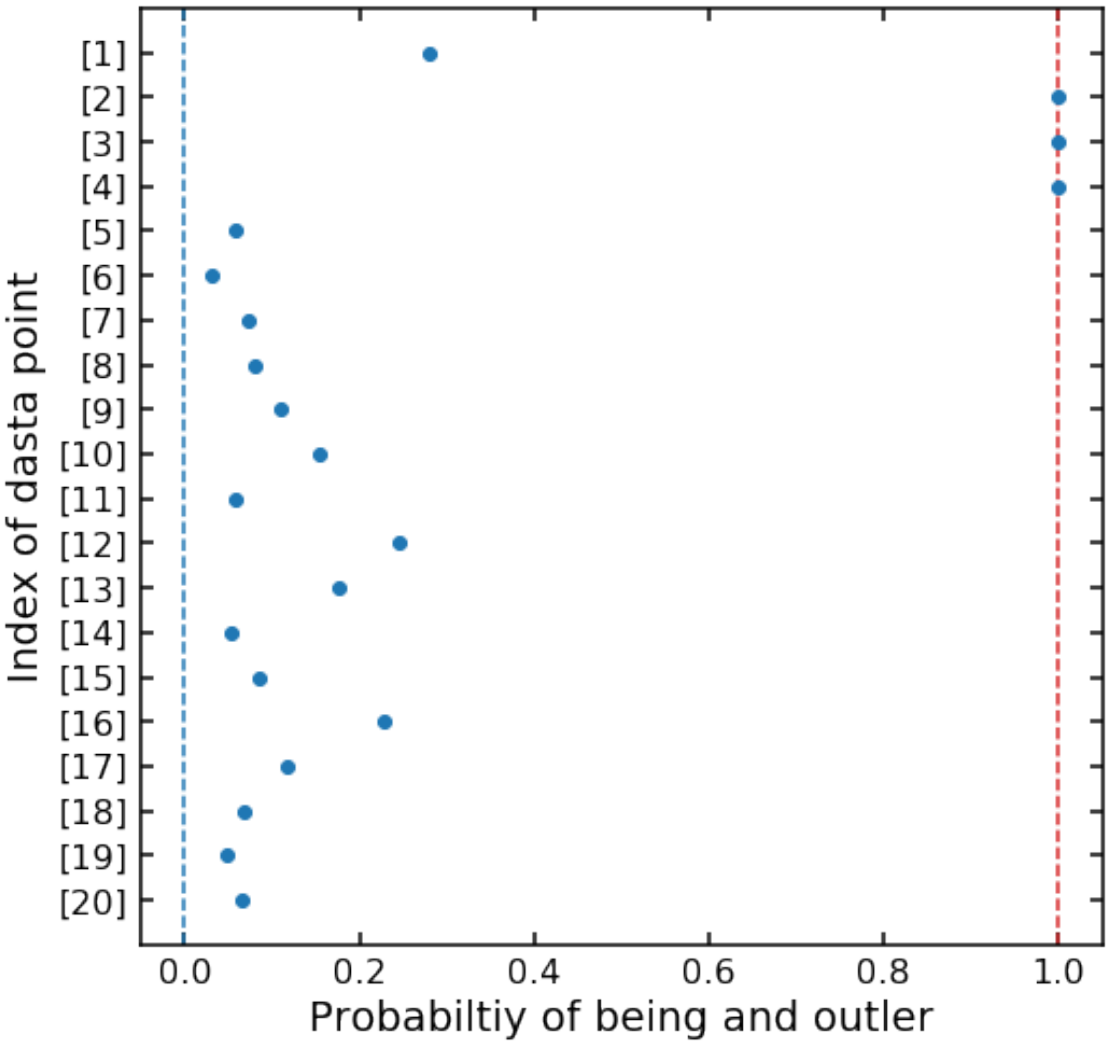


As before we see a strong correlation between the slope and intercept.

1.4.3 Finding the outliers

Unlike the OLS fit, this fit also has information about the probability of each point being an outlier (q_i). For each step in the MC chain a `True` or `False` array was recored indicating what points belonged to the outlier distribution. Averaging these across each step in the chin gives us the probability of each point being an outlier.

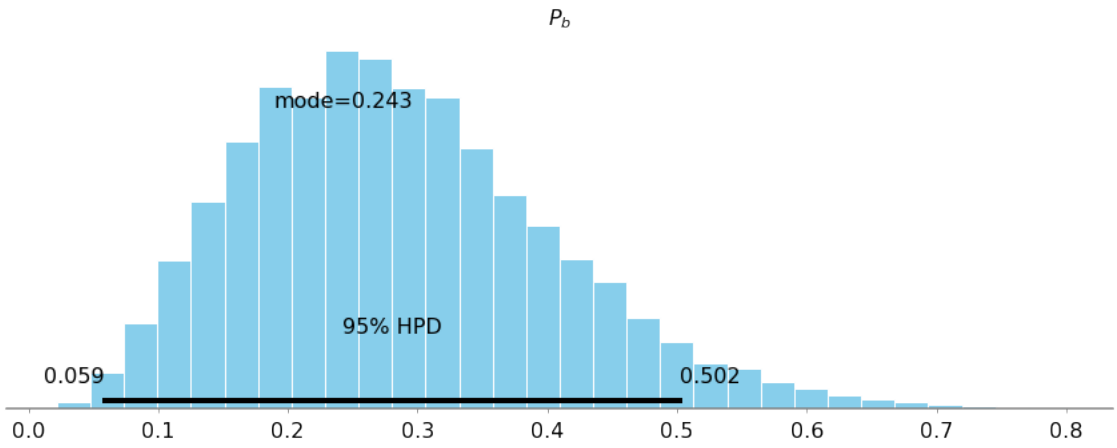
[20]:
```
plt.figure(3, figsize=(8, 8))
plt.plot(
    traces_signoise['$q_i$'].mean(axis=0),
    range(20),
    'o'
)
plt.vlines([0, 1], -1, 20, ['C0', 'C3'], '--')
ax=plt.gca()
ax.set_yticks(range(20))
ax.set_yticklabels(['[{0}]'.format(i) for i in data.ID])
plt.xlabel('Probabiltiy of being and outler')
plt.ylabel('Index of dasta point')
plt.ylim(20, -1);
```



From this plot we can see that three points are clearly marked as being outliers 100% of the time. All of the other points are classed as outliers less that 33% of the time.

Lets take a closer look at the posterior distribution for the fraction of data points belonging to the outlier distribution (P_b). Looking at the above plot we might expect this to peak at $3/20 = 0.15$.

[21]:
```
pm.plot_posterior(
    traces_signoise,
    varnames=[
        '$P_b$'
    ],
    figsize=(12, 5),
    text_size=16,
    point_estimate='mode'
);
```



Interestingly it peaks at 0.217, so we would expect 4 to 5 outliers instead of 3, so where does this number come from?

[22]:
```
traces_signoise['$q_i$'].mean(axis=0).sum() / 20
```

[22]:
```
0.2464555555555555
```

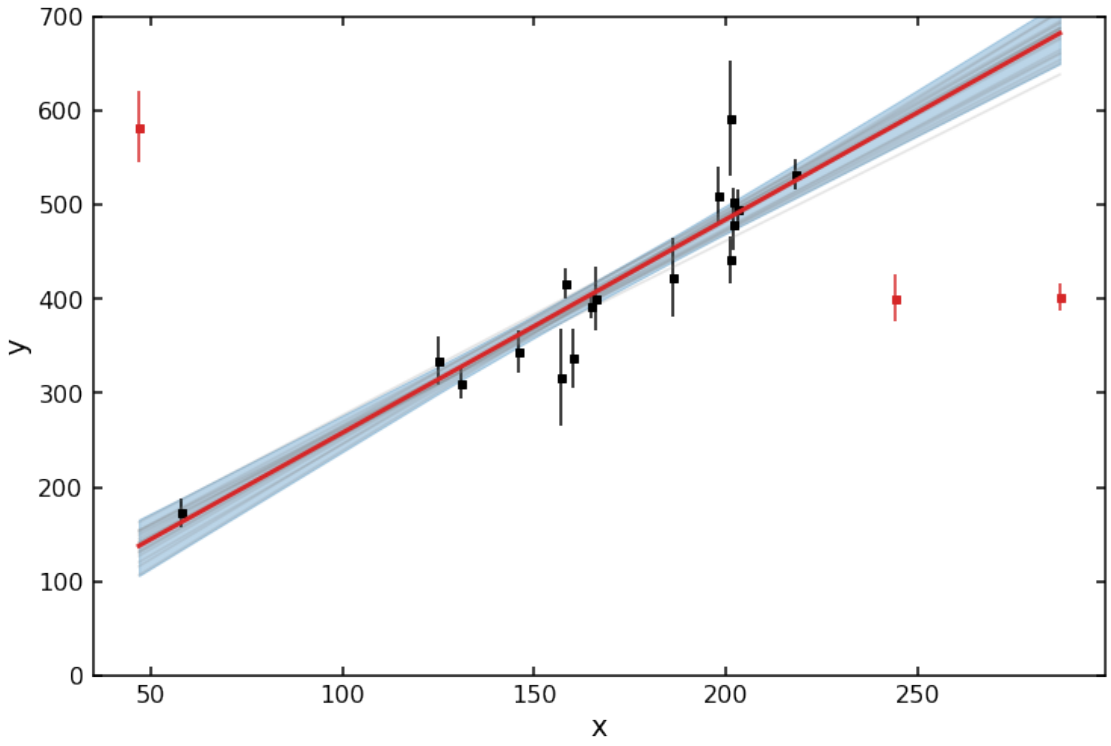
That is closer to the peak of the posterior. This is taking the sum of the outlier fraction for *all* points into account. So overall there are "5" outliers but 2 of those are split among 17 data points.

1.4.4 Plotting the fits

As before lets plot these fits on the original data points but this time we will highlight the outliers.

[23]:
```
# uncenter and scale
y_est = (traces_signoise['$y_{est}$'] * y_std) + y_mean

# find the outliers
prob_outlier = traces_signoise['$q_i$'].mean(axis=0)
outliers = prob_outlier > 0.8

plt.figure(4, figsize=(12, 8))
# plot non-outliers
plt.errorbar(
    data.x[~outliers],
    data.y[~outliers],
    data.sy[~outliers],
    ls='None',
    mfc='k',
    mec='k',
    ms=5,
    marker='s',
    ecolor='k'
)
# plot outliers
```

```
plt.errorbar(
    data.x[outliers],
    data.y[outliers],
    data.sy[outliers],
    ls='None',
    mfc='C3',
    mec='C3',
    ms=5,
    marker='s',
    ecolor='C3'
)

y_est_minus_2_sigma, y_est_median, y_est_plus_2_sigma = np.percentile(
    y_est[:, idx],
    [2.5, 50, 97.5],
    axis=0
)

# plot the mean of all best fit lines
plt.plot(
    data.x[idx],
    y_est_median,
    color='C3',
    lw=3,
    zorder=3
)
# plot 2-sigma best fit region
plt.fill_between(
    data.x[idx],
    y_est_minus_2_sigma,
    y_est_plus_2_sigma,
    color='C0',
    alpha=0.3,
    zorder=1
)
# plot a selection of best fit lines
plt.plot(
    data.x[idx],
    y_est[::500].T[idx],
    alpha=0.2,
    color='C7'
)

plt.xlabel('x')
plt.ylabel('y')
plt.ylim(0, 700);
```



Looking much better! Our best fit line goes through all the data points without being confused by the outliers.

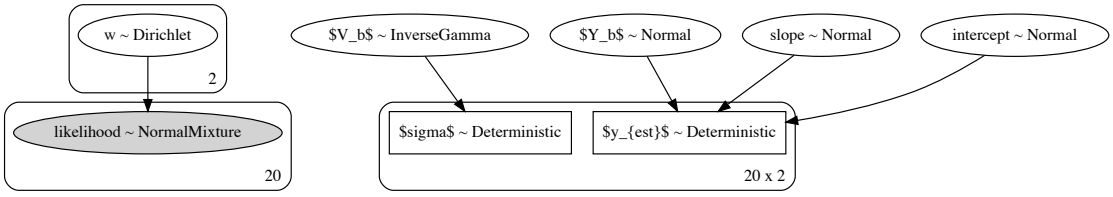
1.4.5 What if you don't care about q_i?

If you don't care about the q_i value for each data point you they can be marginalized over. Doing this results in a normal mixture model, and pymc3 has this built in. The plus side to using this kind of model is every variable is continuous (q_i was discrete) making it easier to sample from and faster to converge.

[24]:
```
with pm.Model() as mixture:
    w = pm.Dirichlet('w', np.array([0.8, 0.2]))
    b0 = pm.Normal('intercept', mu=0, sd=100, testval=0)
    b1 = pm.Normal('slope', mu=0, sd=100, testval=1.2)
    Yb = pm.Normal(r'$Y_b$', mu=0, sd=10, testval=0)
    Vb = pm.InverseGamma(r'$V_b$', 2, 5, testval=2.5)
    y_est = pm.Deterministic(
        r'$y_{est}$',
        tt.stack(
            [
                b0 + b1 * x_center,
                np.ones_like(x_center) * Yb
            ],
            axis=1
        )
```

```
    )
    sigma = pm.Deterministic(
        r'$\sigma$',
        tt.stack(
            [
                sy_center,
                np.sqrt(np.array(sy_center)**2 + Vb)
            ],
            axis=1
        )
    )
    likelihood = pm.NormalMixture(
        'likelihood',
        w,
        y_est,
        sd=sigma,
        observed=y_center
    )

display(mixture)
display(pm.model_to_graphviz(mixture))
```

$$
\begin{aligned}
\text{intercept} &\sim \text{Normal}(mu = 0,\ sd = 100.0) \\
\text{slope} &\sim \text{Normal}(mu = 0,\ sd = 100.0) \\
Y_b &\sim \text{Normal}(mu = 0,\ sd = 10.0) \\
w &\sim \text{Dirichlet}(a = array) \\
V_b &\sim \text{InverseGamma}(alpha = 2,\ beta = 5) \\
y_{est} &\sim \text{Deterministic}(\text{Constant, intercept, slope, Constant, Constant, } Y_b) \\
\sigma &\sim \text{Deterministic}(\text{Constant, Constant, Constant, } V_b\text{\_log\_\_}) \\
\text{likelihood} &\sim \text{NormalMixture}(w = \text{w},\ mu = y_{est},\ sigma = f(f(),\ \sigma))
\end{aligned}
$$



w is a two element array that is equivalent to $[1 - P_b, P_b]$. These are the coefficients used to say how much of the likelihood comes from the inlier and outlier distributions.

[27]:
```
with mixture:
    traces_mixture = pm.sample(
        3000,
        tune=2000,
        target_accept=0.9,
```

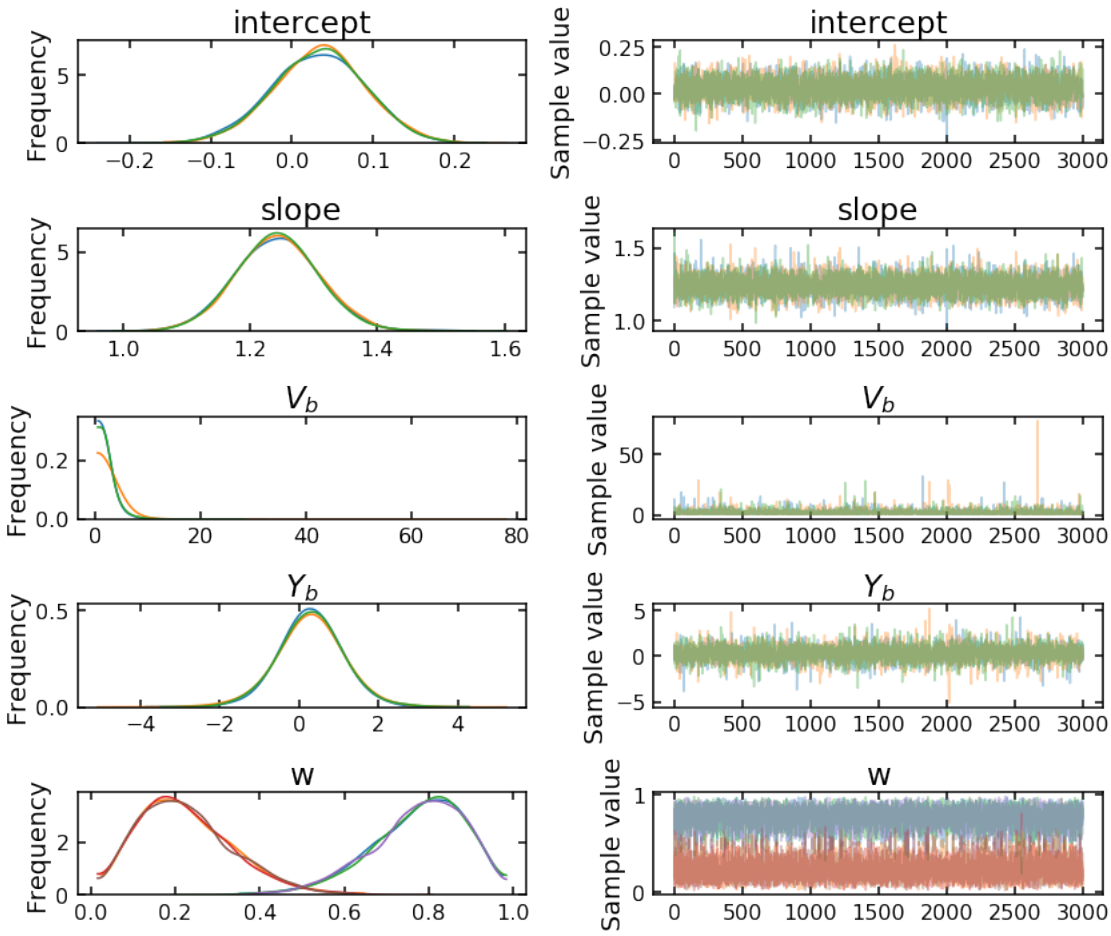
```
        chains=3,
        cores=3
    )
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (3 chains in 3 jobs)
NUTS: [$V_b$, $Y_b$, slope, intercept, w]
Sampling 3 chains: 100%|| 15000/15000 [00:27<00:00, 548.79draws/s]
```

1.4.6 Check for convergence

[28]:
```
display(pm.summary(
    traces_mixture,
    varnames=[
        'intercept',
        'slope',
        '$V_b$',
        '$Y_b$',
        'w'
    ]
).round(4))
pm.traceplot(
    traces_mixture,
    varnames=[
        'intercept',
        'slope',
        '$V_b$',
        '$Y_b$',
        'w'
    ]
);
```

```
              mean      sd  mc_error  hpd_2.5  hpd_97.5      n_eff    Rhat
intercept   0.0336  0.0568    0.0007  -0.0791    0.1447  6206.9633  1.0002
slope       1.2441  0.0652    0.0007   1.1204    1.3737  8491.5558  0.9999
$V_b$       2.4752  2.1466    0.0294   0.5018    5.7840  4558.2457  1.0001
$Y_b$       0.2902  0.8338    0.0101  -1.4526    1.8566  5741.7307  0.9999
w__0        0.7740  0.1114    0.0012   0.5586    0.9636  7282.6991  0.9999
w__1        0.2260  0.1114    0.0012   0.0364    0.4414  7282.6991  0.9999
```



Everything looks good. If you compare the values in the summary table to the values found in the previous fit they are in agreement for all values.

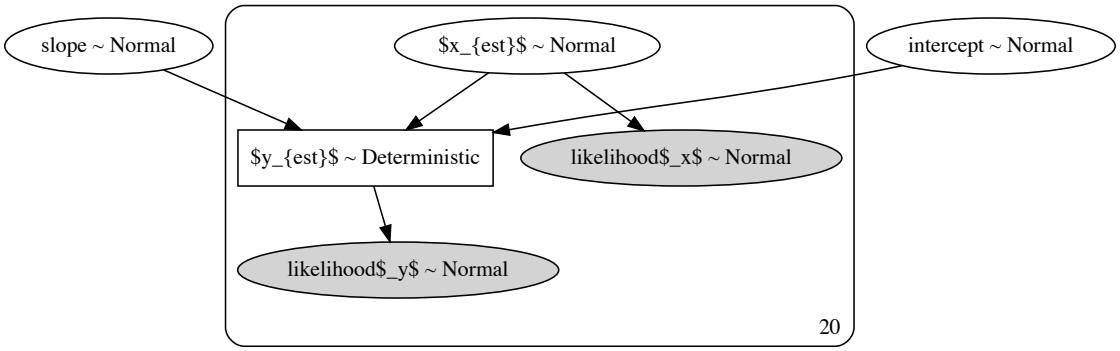
1.5 How to deal with errors in the x direction

Up until now we have only taken into account the errorbars in the y direction. If you also wanted to account for the errors in the x direction (assuming there are no covariances) the OLS model would look like this:

[29]:
```
with pm.Model() as mdl_ols_sx:
    b0 = pm.Normal('intercept', mu=0, sd=100)
    b1 = pm.Normal('slope', mu=0, sd=100)
    x_est = pm.Normal('$x_{est}$', mu=0, sd=50, shape=len(x_center))
    likelihood_x = pm.Normal('likelihood$_x$', mu=x_est, sd=sx_center,␣
↪observed=x_center)
    y_est = pm.Deterministic('$y_{est}$', b0 + b1 * x_est)
    likelihood_y = pm.Normal('likelihood$_y$', mu=y_est, sd=sy_center,␣
↪observed=y_center)
```

```
display(mdl_ols_sx)
display(pm.model_to_graphviz(mdl_ols_sx))
```

$$
\begin{aligned}
\text{intercept} &\sim \text{Normal}(mu = 0,\ sd = 100.0) \\
\text{slope} &\sim \text{Normal}(mu = 0,\ sd = 100.0) \\
x_{est} &\sim \text{Normal}(mu = 0,\ sd = 50.0) \\
y_{est} &\sim \text{Deterministic}(\text{intercept},\ \text{slope},\ x_{est}) \\
\text{likelihood}_x &\sim \text{Normal}(mu = x_{est},\ sd = array) \\
\text{likelihood}_y &\sim \text{Normal}(mu = y_{est},\ sd = array)
\end{aligned}
$$



This looks much like the model before, except this time we assume a `Normal` prior on the x positions and add in a second likelihood using the observed x data.

1.5.1 Note

This is a case where placing a `Uniform` prior on x_{est} will cause it to take much longer to run (over 20 mins as opposed to 20 secs).

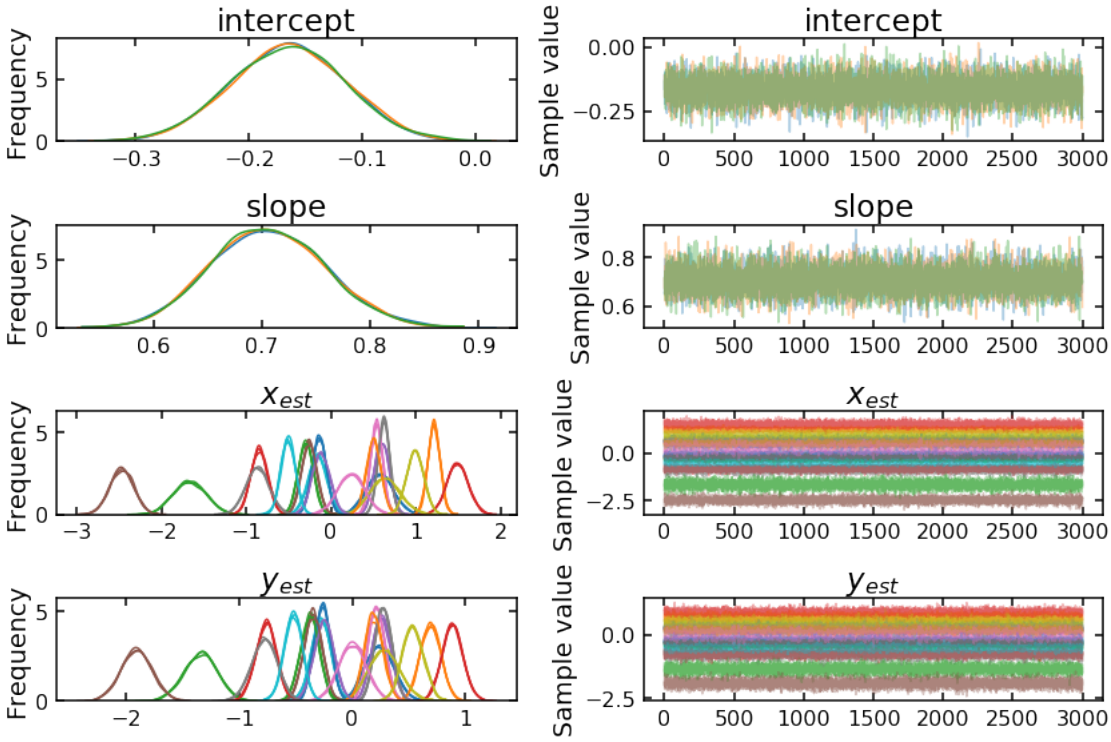
[30]:
```
with mdl_ols_sx:
    traces_ols_2 = pm.sample(3000, tune=2000, chains=3, cores=3)
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (3 chains in 3 jobs)
NUTS: [$x_{est}$, slope, intercept]
Sampling 3 chains: 100%|| 15000/15000 [00:17<00:00, 868.52draws/s]
```

[31]:
```
display(pm.summary(traces_ols_2, varnames=['intercept', 'slope']).round(4))
pm.traceplot(traces_ols_2);
```

```
/Users/coleman/anaconda3/lib/python3.7/site-packages/pymc3/stats.py:974:
FutureWarning: The join_axes-keyword is deprecated. Use .reindex or
.reindex_like on the result to achieve the same functionality.
  axis=1, join_axes=[dforg.index])
```

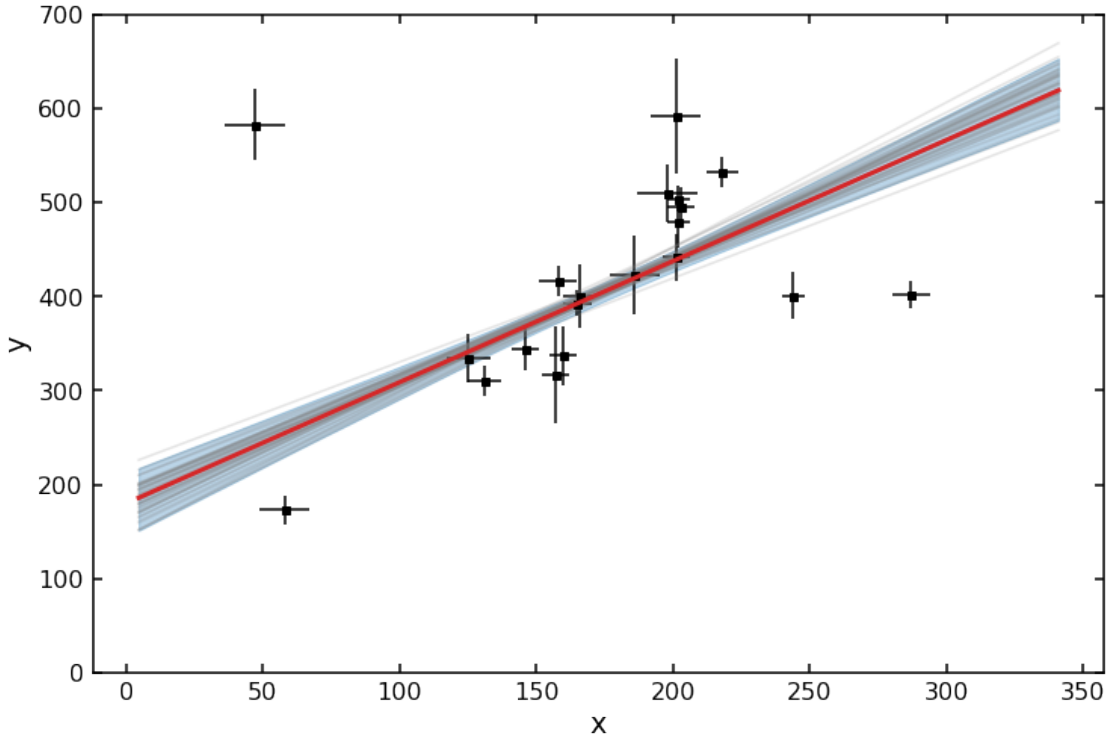
```
               mean      sd  mc_error  hpd_2.5  hpd_97.5       n_eff    Rhat
intercept -0.1650  0.0501    0.0004  -0.2657   -0.0709  16593.6350  1.0000
slope      0.7065  0.0524    0.0005   0.6067    0.8105  12735.6519  0.9999
```



Care should be taken when plotting these results as each y_{est} has been calculated using slightly different x_{est} values, so they can't be averaged as nicely as before.

[33]:
```
# get y_est evaluated at all the same x positinos
x_eval = np.linspace(-3, 3, 200)
y_eval = traces_ols_2['slope'].reshape(-1, 1) * x_eval +␣
 ↪traces_ols_2['intercept'].reshape(-1, 1)

# uncenter data
x_eval = (x_eval * x_std) + x_mean
y_eval = (y_eval * y_std) + y_mean

# get 2-sigma region and median
y_est_minus_2_sigma, y_est_median, y_est_plus_2_sigma = np.percentile(
    y_eval,
    [2.5, 50, 97.5],
    axis=0
)

plt.figure(5, figsize=(12, 8))
```

```
plt.errorbar(
    data.x,
    data.y,
    data.sy,
    data.sx,
    ls='None',
    mfc='k',
    mec='k',
    ms=5,
    marker='s',
    ecolor='k'
)

# plot the mean of all best fit lines
plt.plot(
    x_eval,
    y_est_median,
    color='C3',
    lw=3,
    zorder=3
)
# plot 2-sigma best fit region
plt.fill_between(
    x_eval,
    y_est_minus_2_sigma,
    y_est_plus_2_sigma,
    color='C0',
    alpha=0.3,
    zorder=1
)
# plot a selection of best fit lines
plt.plot(
    x_eval,
    y_eval[::200].T,
    alpha=0.2,
    color='C7'
)

plt.xlabel('x')
plt.ylabel('y')
plt.ylim(0, 700);
```



A similar setup can be used in either of the mixture models used above.

[ ]: