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.

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

Relevant documentation

- introduction to probabilistic programming: https://nbviewer.jupyter.org/github/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/blob/master/Prologue/Prologue.ipynb
- `pymc3` : https://docs.pymc.io/en/stable/pymc-examples/examples/generalized_linear_models/GLM-robust-with-outlier-detection.html#5.-Linear-Model-with-Custom-Likelihood-to-Distinguish-Outliers:-Hogg-Method
- `matplotlib` : https://matplotlib.org/stable/api/pyplot_summary.html
- how to pick priors: https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations

In [1]:
```
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);
```

```
WARNING (theano.tensor.blas): Using NumPy C-API based implementation for BLAS functions.
```

Read in the data

First lets read in the data we will be fitting:

In [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()
```

What does centering the data do?

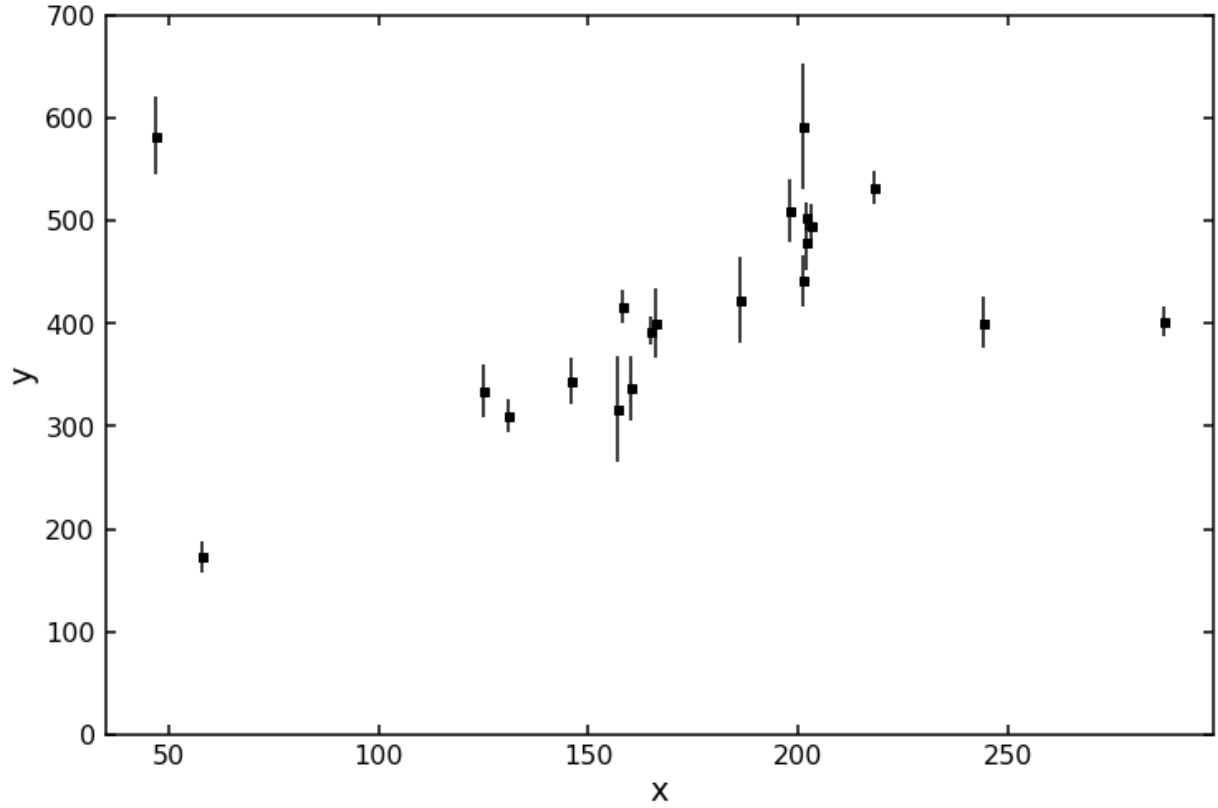
We will be fitting a line to our data with a slope and an intercept. In the original data space these two value are highly correlated, a small change in slope will result in a large change in the intercept and vice-versa. By centering and scaling the data we are ensuring the intercept is close to 0 and 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).

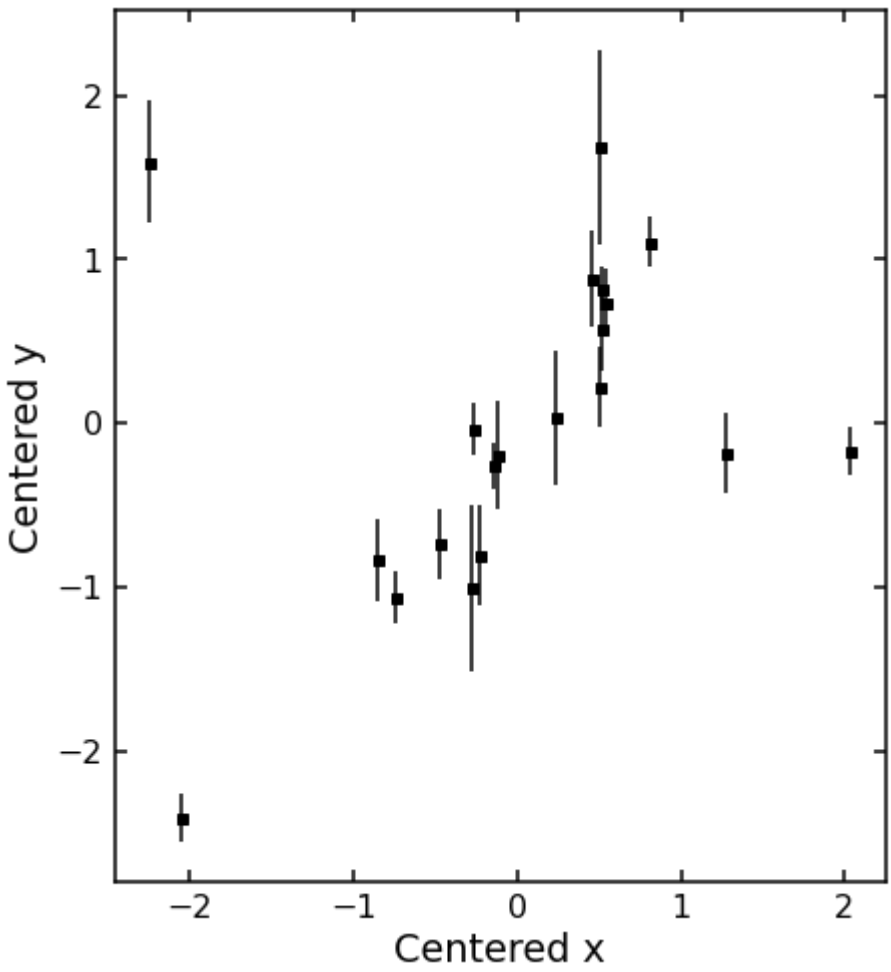
Plot the data

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

In [3]:

```
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);
```



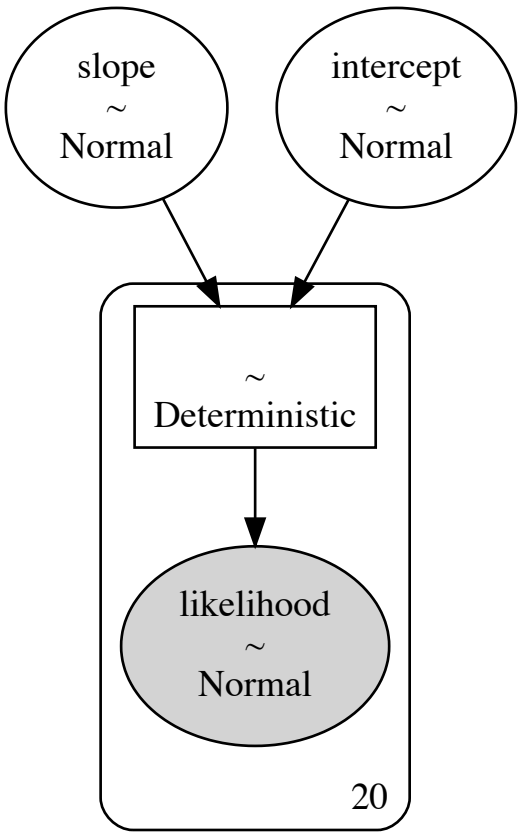


OLS Model

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

In [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_c

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

$$
\begin{aligned}
\text{intercept} &\sim \text{Normal} \\
\text{slope} &\sim \text{Normal} \\
y_{est} &\sim \text{Deterministic} \\
\text{likelihood} &\sim \text{Normal}
\end{aligned}
$$



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.

Run MCMC

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

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

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
```

```
Multiprocess sampling (3 chains in 3 jobs)
NUTS: [slope, intercept]
```

```
100.00% [15000/15000 00:02<00:00
Sampling 3 chains, 0 divergences]

Sampling 3 chains for 2_000 tune and 3_000 draw iterations (6_000 + 9_000 draw
s total) took 3 seconds.
```

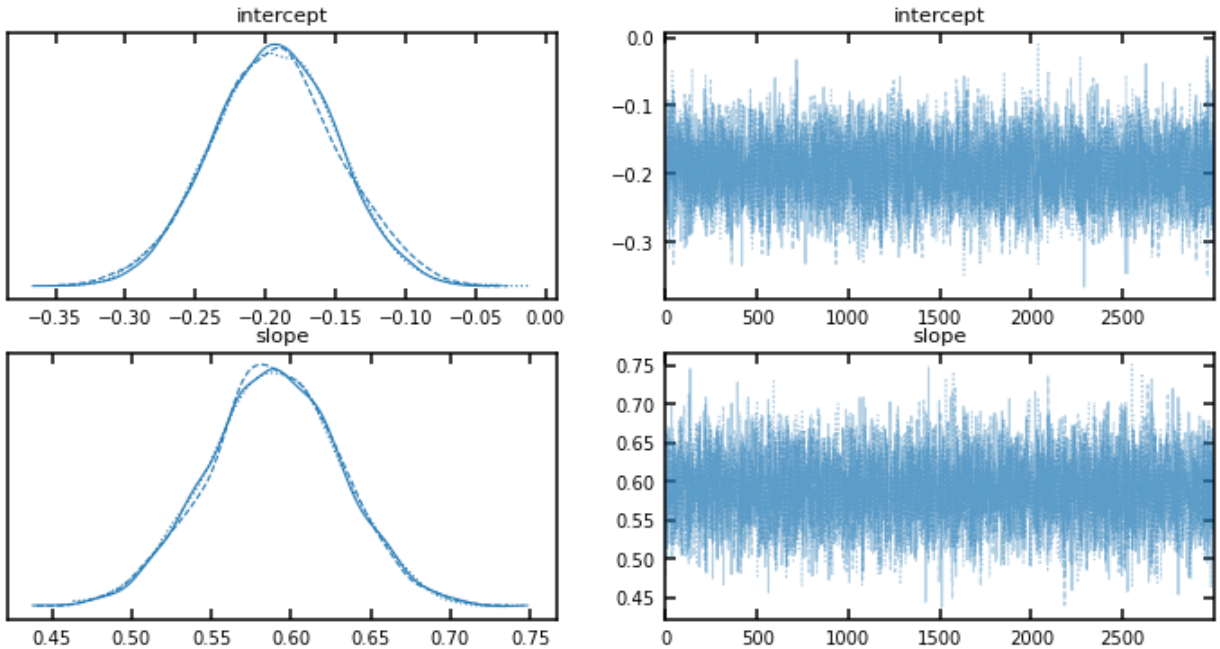
The default sampler is NUTS (No U-True Sampler), a powerful sampler that uses the likelihood's gradient information to propose new MC steps.

Check for convergence

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

In [6]:
```
with mdl_ols:
    display(pm.summary(
        traces_ols,
        var_names=['intercept', 'slope']
    ))
    pm.plot_trace(
        traces_ols,
        var_names=['intercept', 'slope'],
        figsize=(12, 6)
    );
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
intercept	-0.194	0.045	-0.279	-0.109	0.0	0.0	8159.0	6183.0	1.0
slope	0.589	0.042	0.511	0.669	0.0	0.0	8520.0	6642.0	1.0

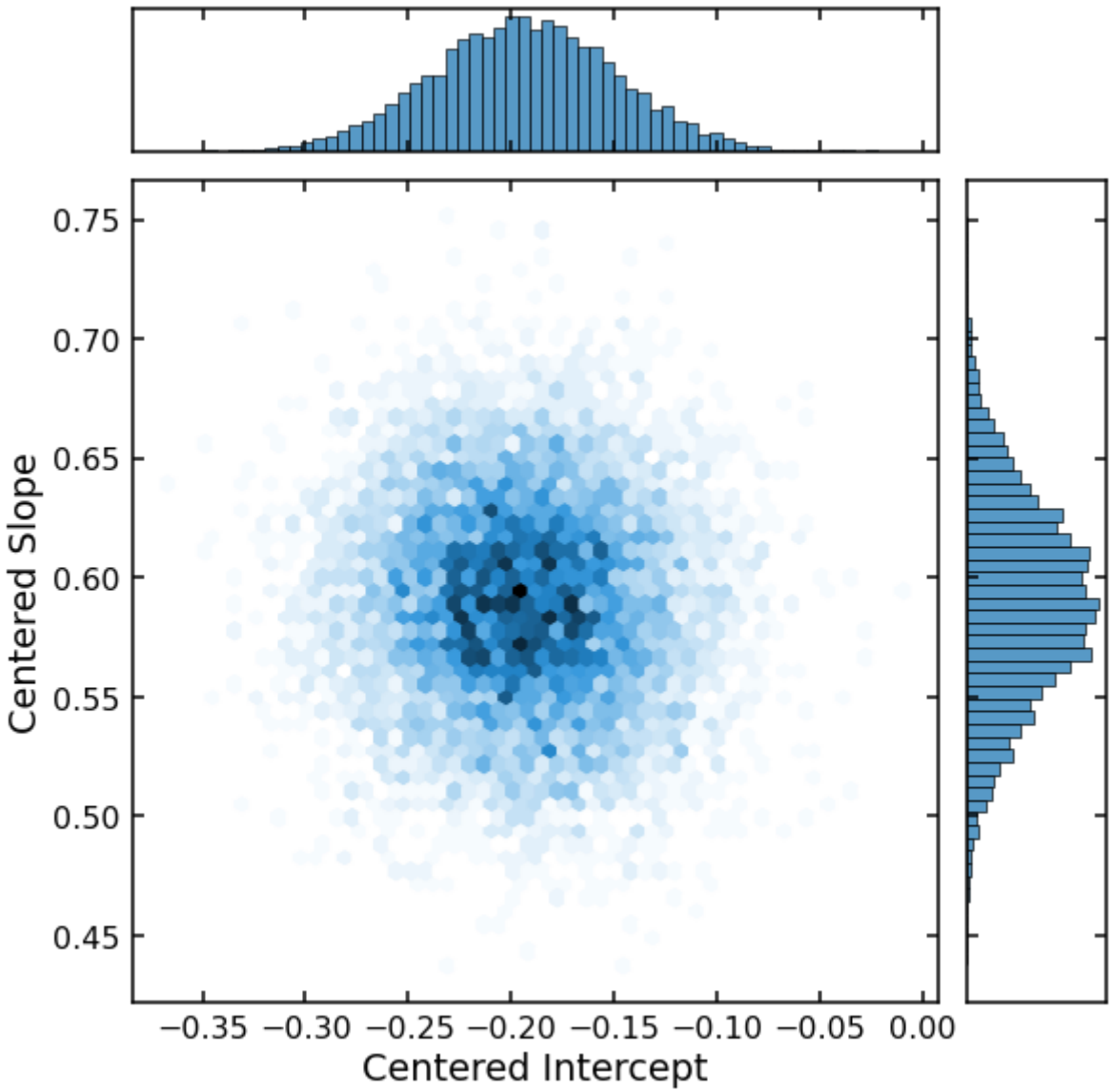


Everything looks good, the `Rhat` values are all close to 1, and the traces have all mixed well. Since we named the `y_est` variable it is also tracked in every sampling step. This makes plotting the results easier since we don't need to evaluate the model for every `b0` and `b1` value in the sample.

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

In [7]:

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

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

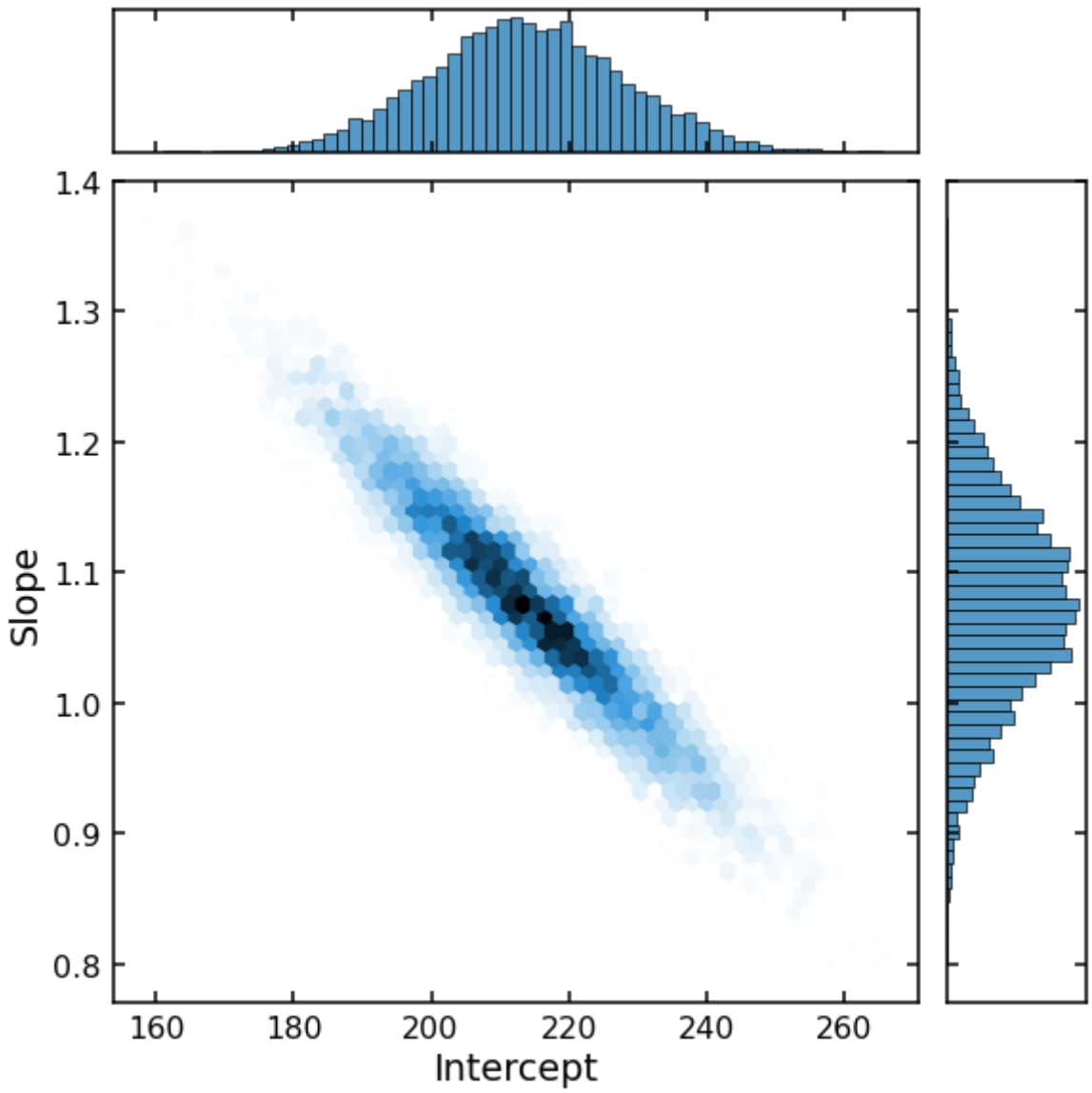


Notice how in the centered parameter space these two values are not correlated (as expected), let transform the fit back to the original data space.

In [8]:

```
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_st

fig = seaborn.jointplot(
    x=b0,
    y=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.

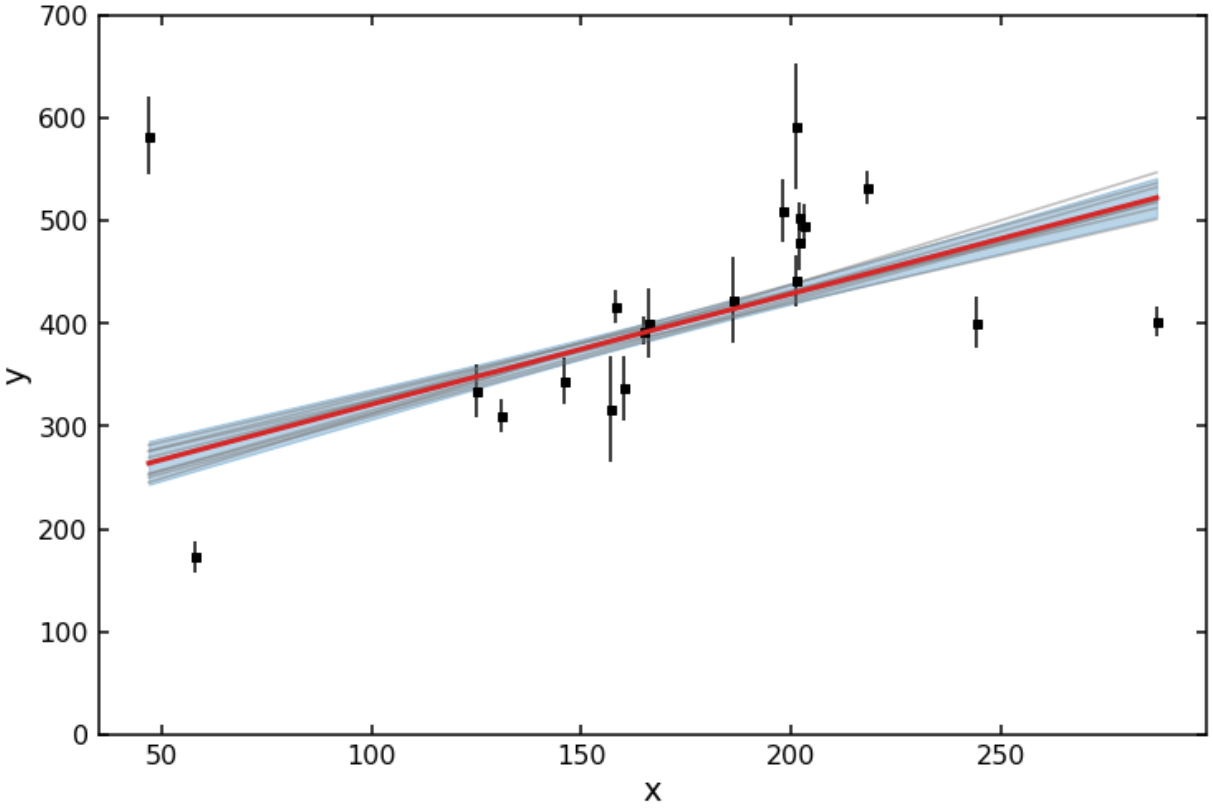
Plotting the best fit

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

In [9]:

```
# 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[::900].T[idx],
    alpha=0.5,
    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.

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:

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

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

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.

In [10]:

```
with pm.Model() as mdl_signoise:
    #  state input data as Theano shared vars
    tsv_x = pm.Data('tsv_x', x_center)
    tsv_y = pm.Data('tsv_y', y_center)
    tsv_sy = pm.Data('tsv_sy', sy_center)
    # 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 * tsv_x)
    # Define weakly informative priors for the mean and variance of the outli
    Yb = pm.Normal(r'$Y_b$', mu=0, sd=10)
    Vb = pm.InverseGamma(r'$V_b$', 2, 5)
    # crate in/outlier distributions to get logL evaluated on observed data
    logL_in = pm.Normal.dist(mu=y_est, sigma=tsv_sy).logp(tsv_y)
    sigma_out = pm.math.sqrt(Vb + tsv_sy**2)
    logL_out = pm.Normal.dist(mu=Yb, sigma=sigma_out).logp(tsv_y)
    # 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=tsv_x.eval().shape[0],
        testval=np.random.rand(tsv_x.eval().shape[0]) < 0.5
    )
    # User custom likelihood
    potential = pm.Potential(
        'likelihood',
        ((1 - qi) * logL_in).sum() + (qi * logL_out).sum()
    )

display(mdl_signoise)
```

$$
\begin{aligned}
\text{intercept} &\sim \text{Normal} \\
\text{slope} &\sim \text{Normal} \\
Y_b &\sim \text{Normal} \\
V_b\_\text{log\_\_} &\sim \text{TransformedDistribution} \\
P_b\_\text{interval\_\_} &\sim \text{TransformedDistribution} \\
q_i &\sim \text{Bernoulli} \\
y_{est} &\sim \text{Deterministic} \\
V_b &\sim \text{InverseGamma} \\
P_b &\sim \text{Uniform}
\end{aligned}
$$

In [15]:

```
with mdl_signoise:
    traces_signoise = pm.sample(
        3000,
        tune=2000,
        chains=3,
        cores=3,
        return_inferencedata=False
    )
```

```
Multiprocess sampling (3 chains in 3 jobs)
CompoundStep
>NUTS: [$P_b$, $V_b$, $Y_b$, slope, intercept]
>BinaryGibbsMetropolis: [$q_i$]
```

100.00% [15000/15000 00:11<00:00
Sampling 3 chains, 0 divergences]

```
Sampling 3 chains for 2_000 tune and 3_000 draw iterations (6_000 + 9_000 draw
s total) took 12 seconds.
/mnt/lustre/shared_python_environment/DataLanguages/lib/python3.8/site-package
s/arviz/stats/diagnostics.py:561: RuntimeWarning: invalid value encountered in
double_scalars
  (between_chain_variance / within_chain_variance + num_samples - 1) / (num_sa
mples)
```

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.

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).

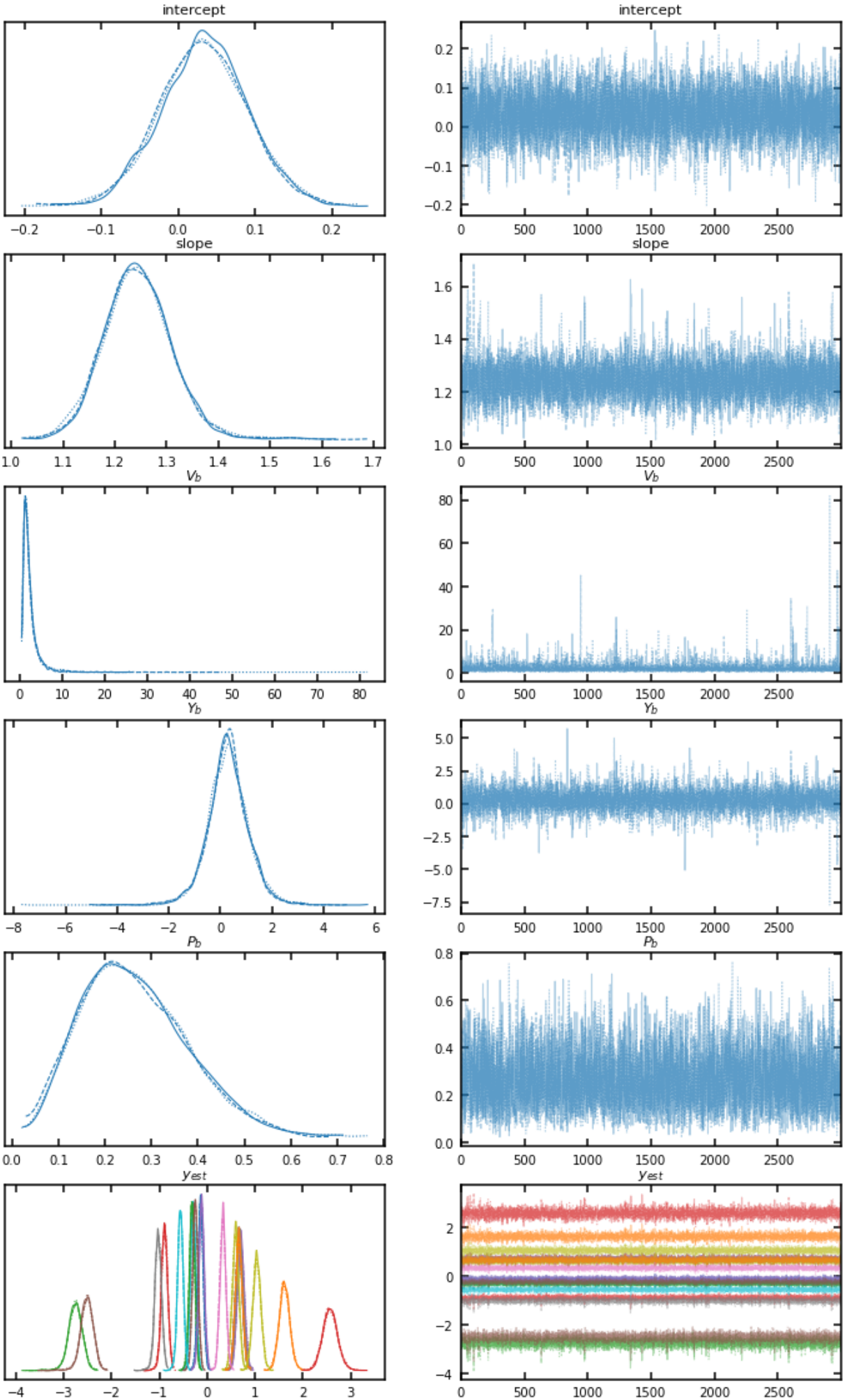
See https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations for other tips about picking priors.

Check for convergence

In [16]:
```
with mdl_signoise:
    display(pm.summary(
        traces_signoise,
        var_names=[
            'intercept',
            'slope',
            '$V_b$',
            '$Y_b$',
            '$P_b$'
        ]
    ))
    pm.plot_trace(
        traces_signoise,
        var_names=[
            'intercept',
            'slope',
            '$V_b$',
            '$Y_b$',
            '$P_b$',
            '$y_{est}$'
        ],
        figsize=(12, 20)
    );
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
intercept	0.033	0.058	-0.077	0.141	0.001	0.001	5174.0	5073.0	1.0
slope	1.244	0.067	1.122	1.369	0.001	0.001	4475.0	3319.0	1.0
V_b	2.426	2.287	0.493	5.305	0.036	0.026	4938.0	4659.0	1.0

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
Y_b	0.303	0.803	-1.133	1.882	0.012	0.009	4961.0	4093.0	1.0
P_b	0.266	0.115	0.069	0.481	0.002	0.001	3667.0	5387.0	1.0

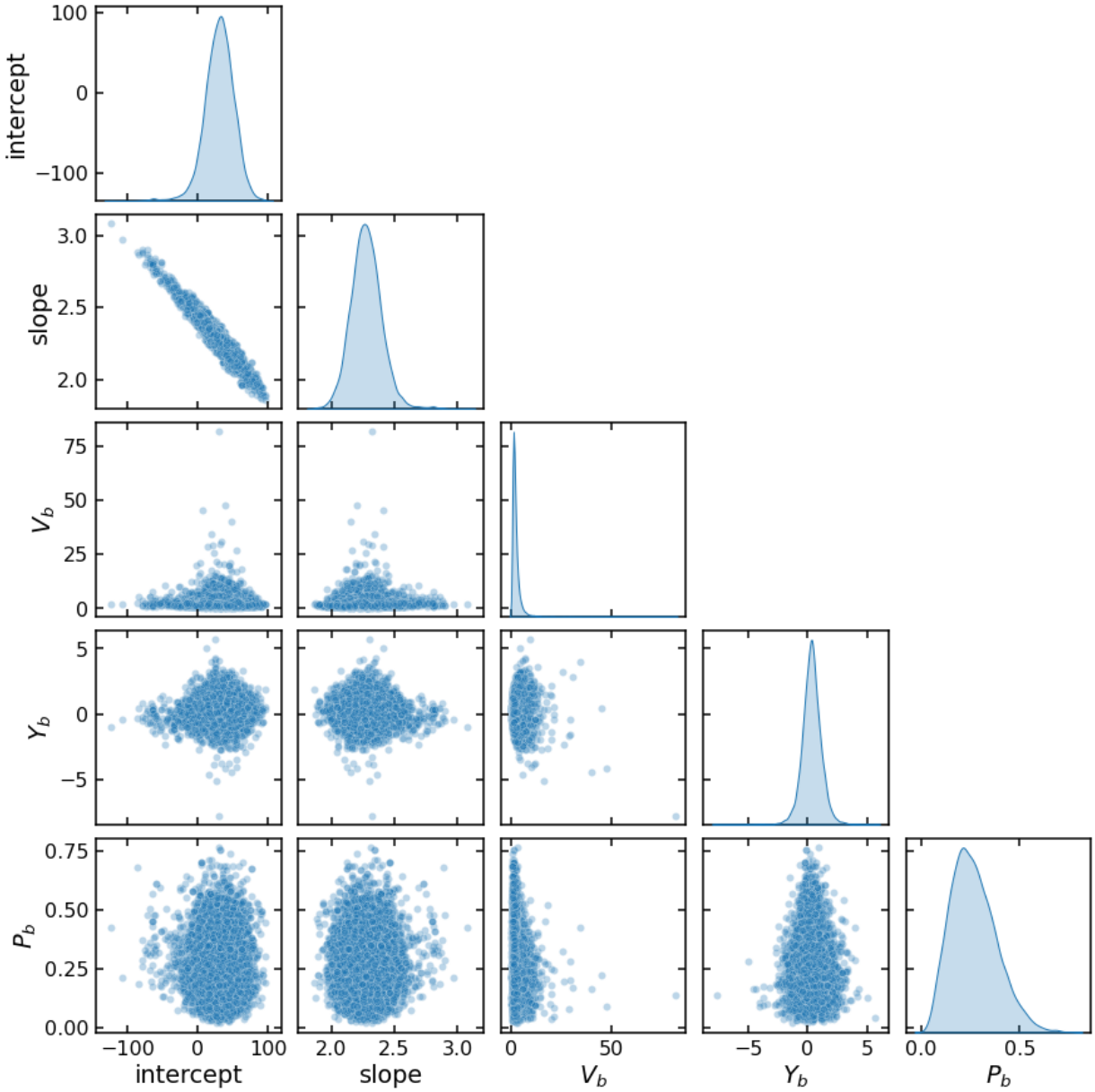


As before we can see the `Rhat` values are all close to 1 and all the chains are well mixed.

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

In [17]:

```
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)
```



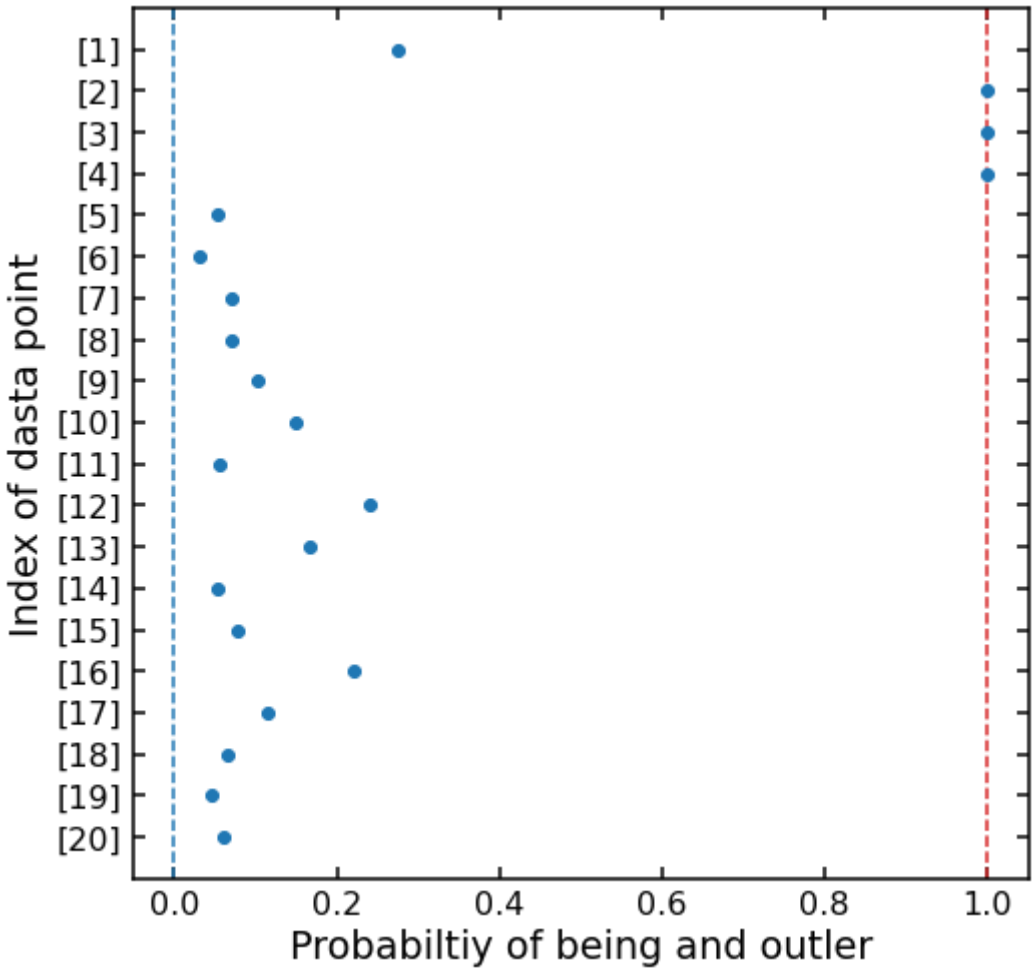
As before we see a strong correlation between the slope and intercept.

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.

In [18]:

```
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);
```

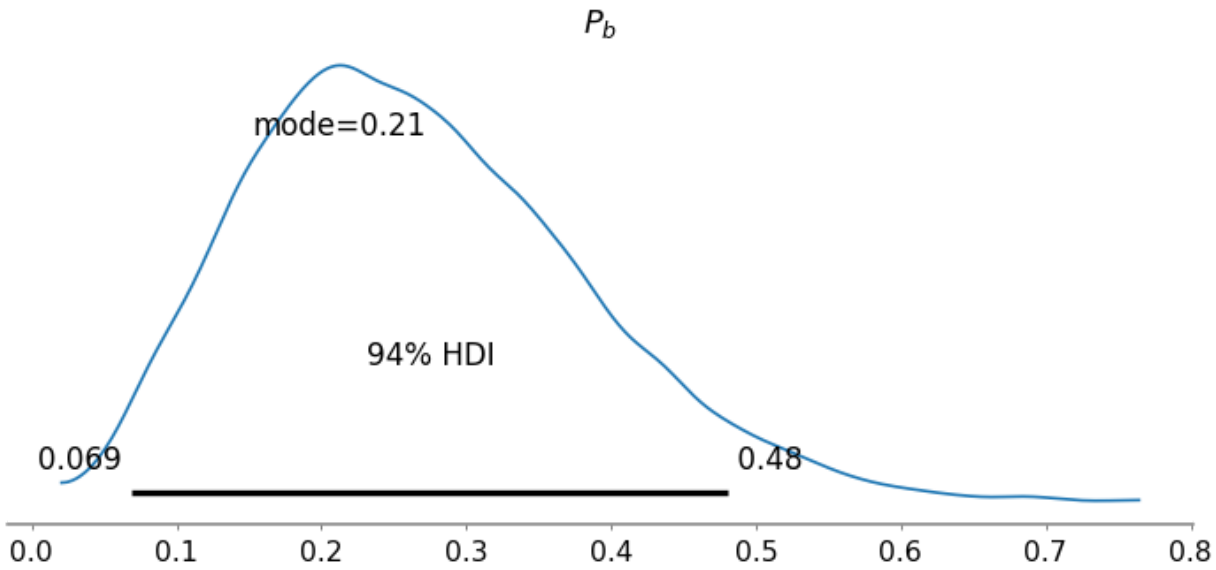


From this plot we can see that three points are clearly marked as being outliers 100% of the time. All of the other points are classed as outliers less that 33% of the time.

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 $3/20 = 0.15$.

In [19]:

```
with mdl_signoise:
    pm.plot_posterior(
        traces_signoise,
        var_names=[
            '$P_b$'
        ],
        figsize=(12, 5),
        point_estimate='mode'
    );
```



Interestingly it peaks at 0.21, so we would expect 4 to 5 outliers instead of 3, so where does this number come from?

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

Out[20]:
```
0.24250555555555559
```

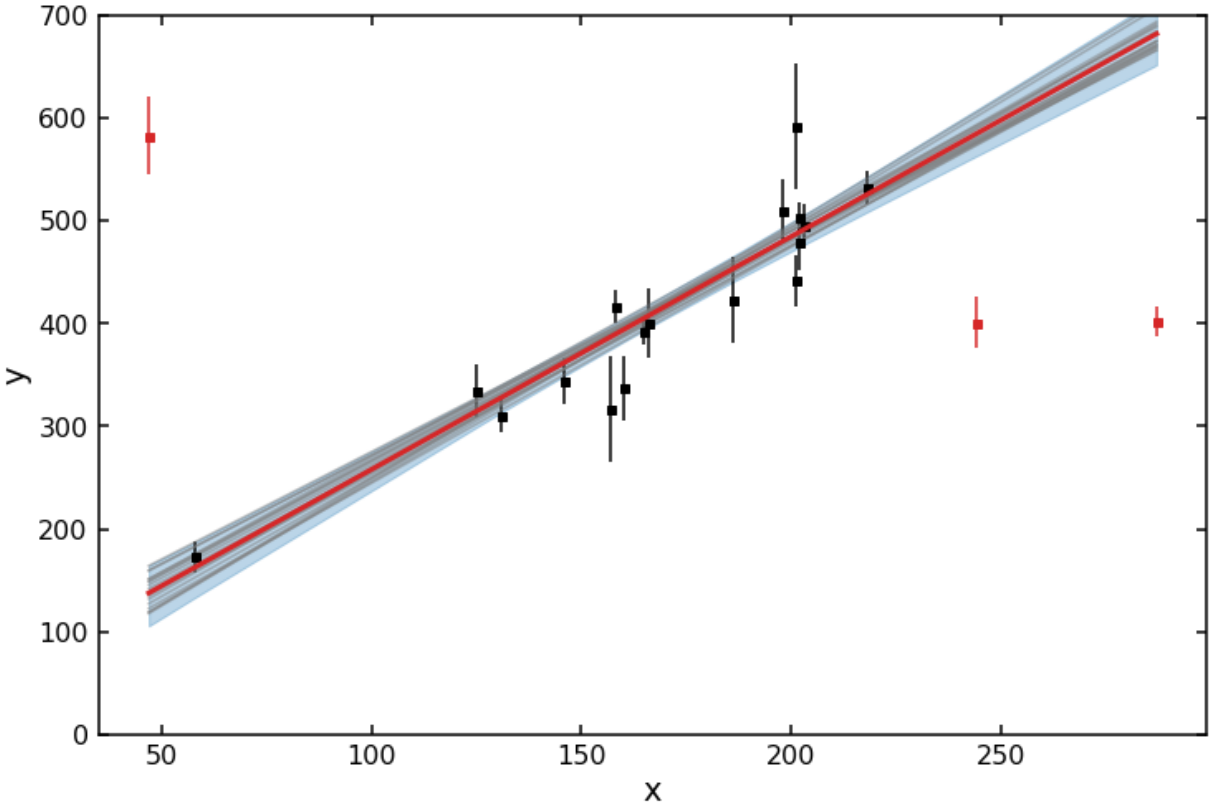
That is closer to the peak of the posterior. This is taking the sum of the outlier fraction for *all* points into account. So overall there are "5" outliers but 2 of those are split among 17 data points.

Plotting the fits

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

In [21]:
```
# 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=0.5,
    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.

What if you don't care about q_i?

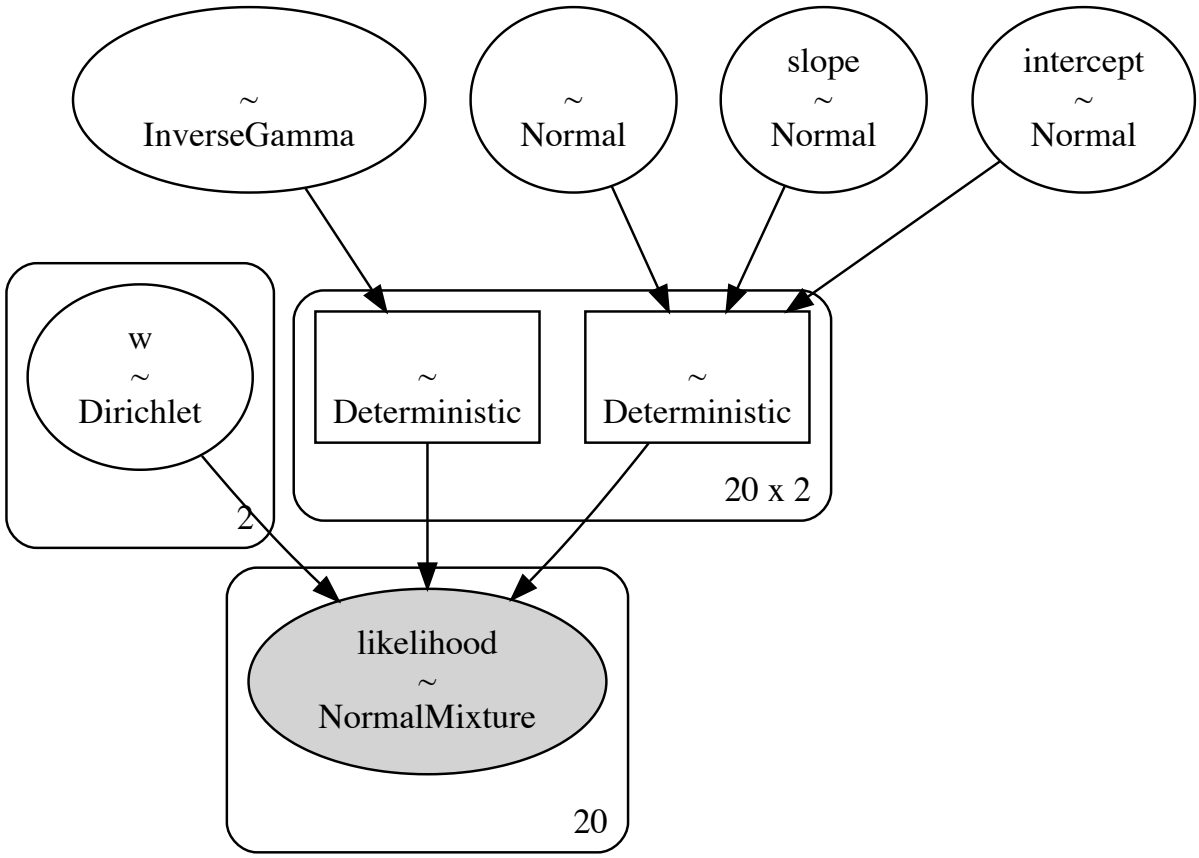
If you don't care about the q_i value for each data point you they can be marginalized over. Doing this results in a normal mixture model, and pymc3 has this built in. The plus side to using this kind of model is every variable is continuous (q_i was discrete) making it easier to sample from and faster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

In [22]:

```
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))
```

$$
\begin{aligned}
\text{w\_stickbreaking\_\_} &\sim \text{TransformedDistribution} \\
\text{intercept} &\sim \text{Normal} \\
\text{slope} &\sim \text{Normal} \\
Y_b &\sim \text{Normal} \\
V_b\text{\_log\_\_} &\sim \text{TransformedDistribution} \\
\text{w} &\sim \text{Dirichlet} \\
V_b &\sim \text{InverseGamma} \\
y_{est} &\sim \text{Deterministic} \\
\sigma &\sim \text{Deterministic} \\
\text{likelihood} &\sim \text{NormalMixture}
\end{aligned}
$$



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.

In [25]:

```
with mixture:
    traces_mixture = pm.sample(
        1000,
```

```
        tune=1000,
        target_accept=0.9,
        chains=3,
        cores=3,
        return_inferencedata=False
    )
```

```
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]
```

```
100.00% [6000/6000 00:03<00:00
Sampling 3 chains, 0 divergences]
```

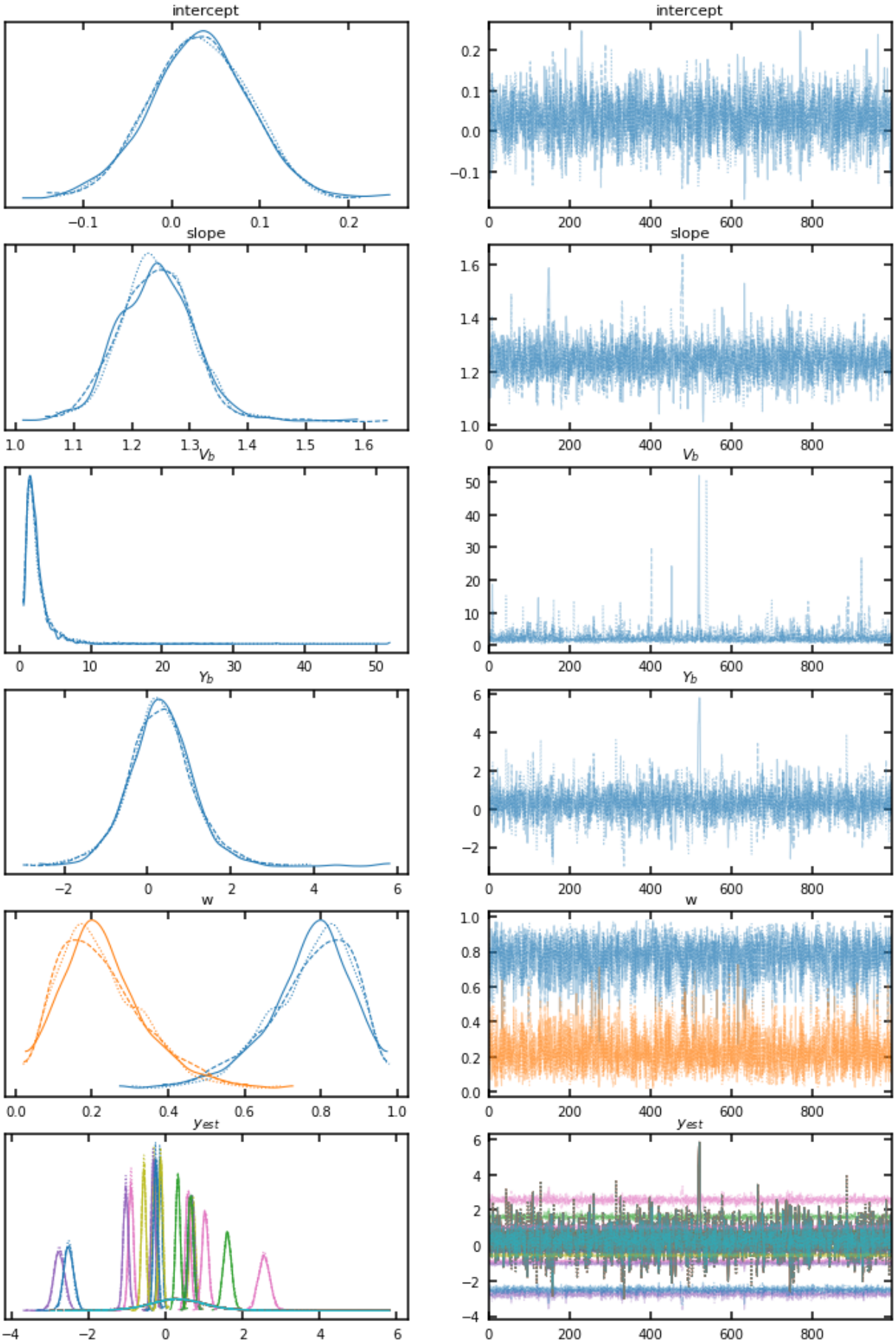
```
Sampling 3 chains for 1_000 tune and 1_000 draw iterations (3_000 + 3_000 draw
s total) took 3 seconds.
```

Check for convergence

In [26]:

```
with mixture:
    display(pm.summary(
        traces_mixture,
        var_names=[
            'intercept',
            'slope',
            '$V_b$',
            '$Y_b$',
            'w'
        ]
    ))
    pm.plot_trace(
        traces_mixture,
        var_names=[
            'intercept',
            'slope',
            '$V_b$',
            '$Y_b$',
            'w',
            '$y_{est}$'
        ],
        figsize=(12, 18)
    );
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
intercept	0.032	0.056	-0.067	0.143	0.001	0.001	2958.0	2133.0	1.0
slope	1.242	0.065	1.128	1.363	0.001	0.001	2686.0	1888.0	1.0
V_b	2.463	2.468	0.433	5.346	0.068	0.048	2188.0	1775.0	1.0
Y_b	0.317	0.810	-1.133	1.924	0.018	0.020	2418.0	1633.0	1.0
w[0]	0.774	0.110	0.570	0.958	0.002	0.002	2668.0	2068.0	1.0
w[1]	0.226	0.110	0.042	0.430	0.002	0.002	2668.0	2068.0	1.0



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.

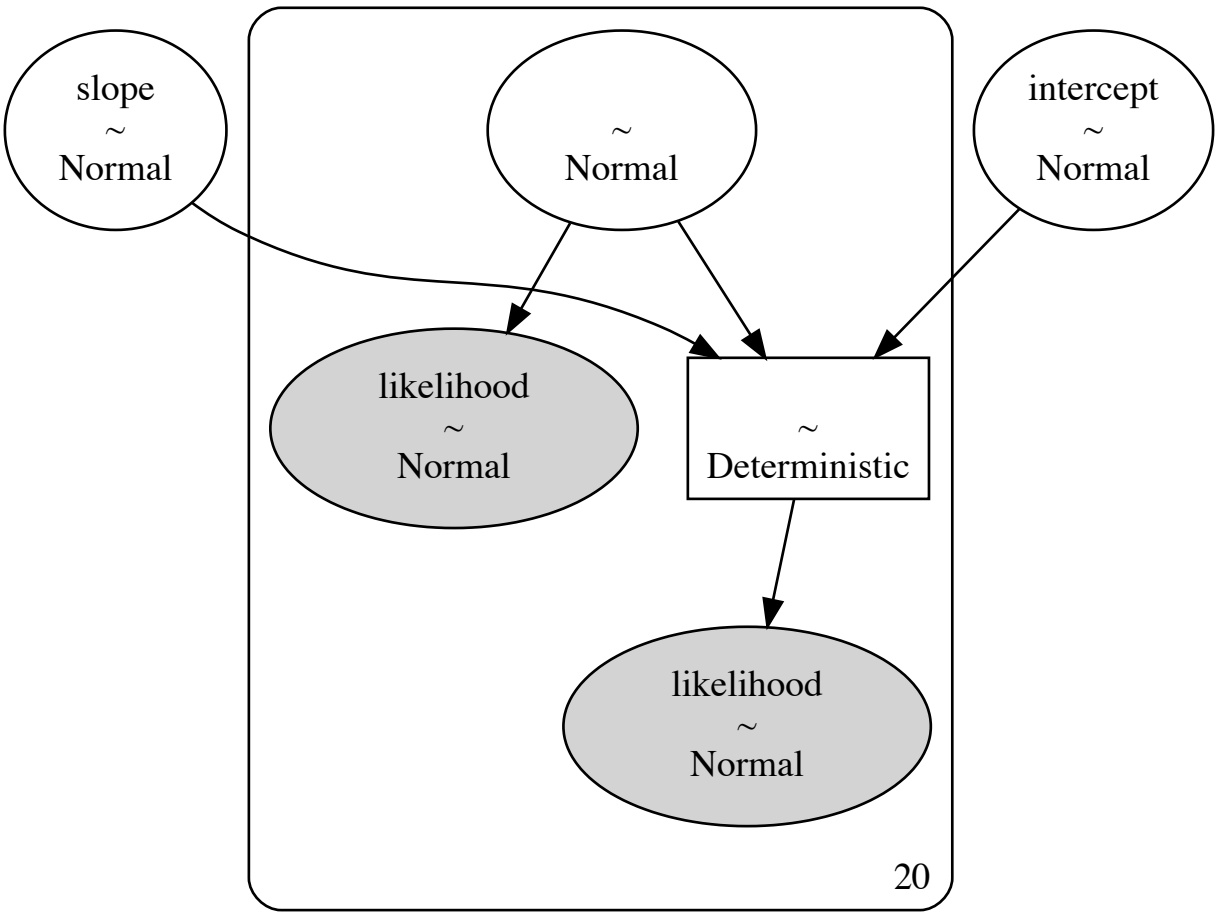
How to deal with errors in the x direction

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:

In [27]:

```
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, observe
    y_est = pm.Deterministic('$y_{est}$', b0 + b1 * x_est)
    likelihood_y = pm.Normal('likelihood$_y$', mu=y_est, sd=sy_center, observe

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

$$
\begin{aligned}
\text{intercept} &\sim \text{Normal} \\
\text{slope} &\sim \text{Normal} \\
x_{est} &\sim \text{Normal} \\
y_{est} &\sim \text{Deterministic} \\
\text{likelihood}_x &\sim \text{Normal} \\
\text{likelihood}_y &\sim \text{Normal}
\end{aligned}
$$



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.

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).

In [28]:
```
with mdl_ols_sx:
    traces_ols_2 = pm.sample(3000, tune=2000, chains=3, cores=3, return_infer
```

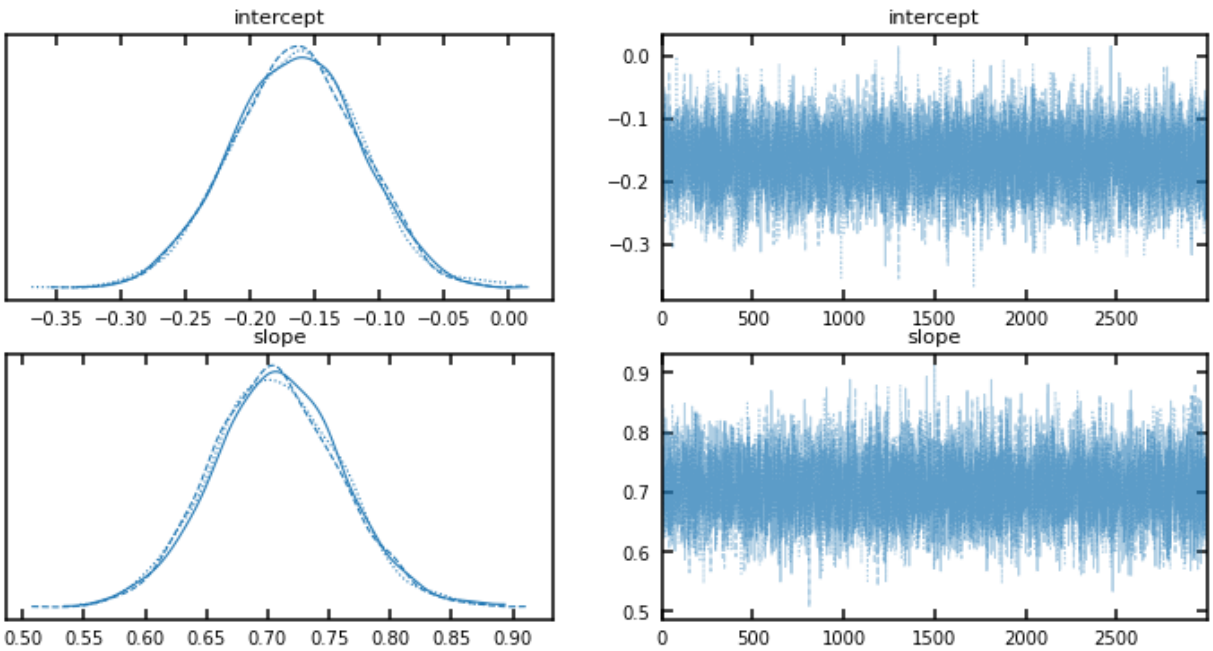
```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (3 chains in 3 jobs)
NUTS: [$x_{est}$, slope, intercept]
```

100.00% [15000/15000 00:04<00:00 Sampling 3 chains, 0 divergences]

```
Sampling 3 chains for 2_000 tune and 3_000 draw iterations (6_000 + 9_000 draw
s total) took 5 seconds.
```

In [29]:
```
with mdl_ols_sx:
    display(pm.summary(
        traces_ols_2,
        var_names=['intercept', 'slope']
    ))
    pm.plot_trace(
        traces_ols_2,
        var_names=['intercept', 'slope'],
        figsize=(12, 6)
    );
```

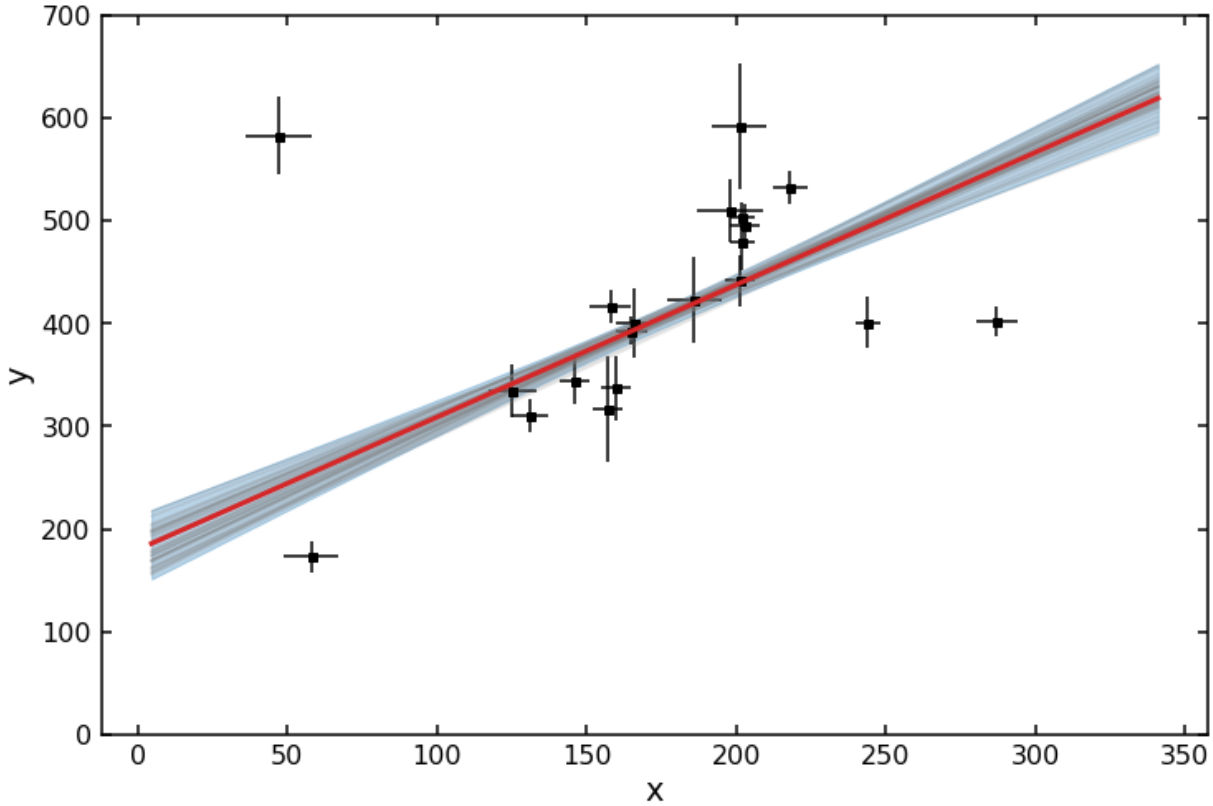
	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
intercept	-0.165	0.050	-0.258	-0.073	0.0	0.0	14364.0	7178.0	1.0
slope	0.707	0.053	0.610	0.807	0.0	0.0	12537.0	7273.0	1.0



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.

In [30]:
```
# 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['interc
```

```
# 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.

In [ ]: