Regression, GLMs, and Bayesian vs Frequentist perspectives

Setup and imports

Running this command may make PyMC3 run faster.

In [1]:
```
!conda install mkl-service -y
```

```
Collecting package metadata (current_repodata.json): done
Solving environment: done

# All requested packages already installed.
```

In [2]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

import pymc3 as pm
from pymc3 import glm
import statsmodels.api as sm
import arviz
```

In this notebook, we'll be working with a dataset containing information on wind turbines.

In [3]:
```
turbines = pd.read_csv('turbines.csv')
# The "year" column contains how many years since the year 2000
turbines['year'] = turbines['p_year'] - 2000
turbines = turbines.drop('p_year', axis=1)
turbines.head()
```

Out[3]:

	t_state	t_built	t_cap	year
0	AK	6	390.0	-3.0
1	AK	6	475.0	-1.0
2	AK	2	100.0	0.0
3	AK	1	1500.0	1.0
4	AK	1	100.0	2.0

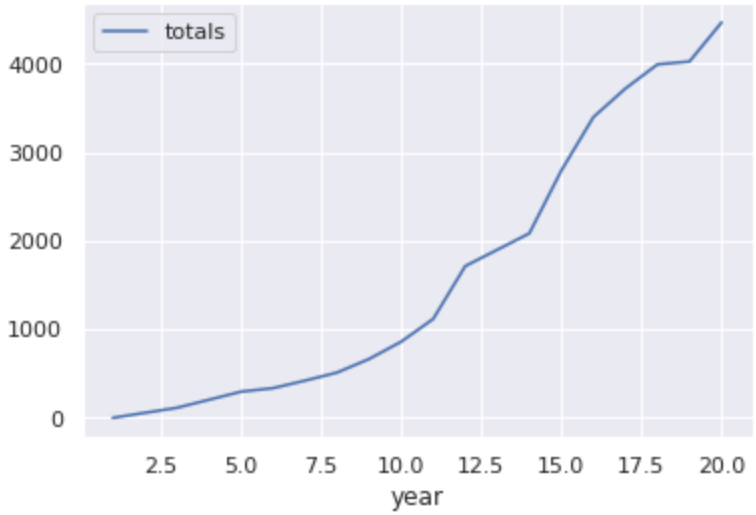
In [4]:
```
# Turbines in Oklahoma (where the wind comes sweepin' down the plain...)
ok_filter = (turbines.t_state == 'OK') & (turbines.year >= 0)
ok_turbines = turbines[ok_filter].sort_values('year')
ok_turbines["totals"] = np.cumsum(ok_turbines["t_built"])
# Log-transform the counts, too
ok_turbines["log_totals"] = np.log(ok_turbines["totals"])

plt.figure()
ok_turbines.plot('year', 'totals')
```

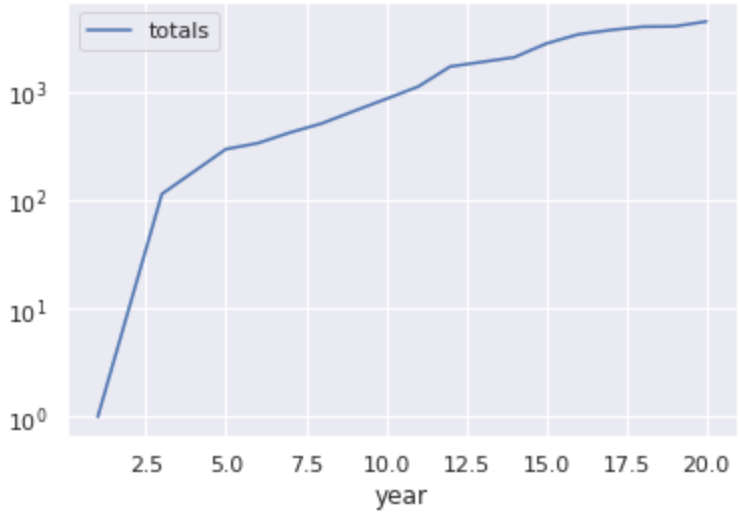
```
plt.figure()
ok_turbines.plot('year', 'totals')
plt.semilogy()
```

Out[4]: []

<Figure size 432x288 with 0 Axes>



<Figure size 432x288 with 0 Axes>



Recall our problem setup:

- X: an $n \times d$ matrix, where each row is a data point, and each column is a feature (fixed)
- β: a d-dimensional vector with one coefficient for each feature (random, unknown: this is what we want)
- y: an n-dimensional vector, with a number (response variable / dependent variable) that we want to predict for each data point (random, because it depends on β and on some error, often captured in a noise vector ϵ)

The likelihood $p(y|\beta)$ usually describes the error model: in standard least squares regression, it's

$$y|\beta \sim N(X\beta, \sigma^2 I_n)$$

Let's think about the implicit assumptions we're making by choosing this likelihood. Recall that for the normal distribution, we're very unlikely to see values more than 3σ away from the mean. That means that we're implicitly assuming that the vast majority of y-values we see will be within 3σ of the mean (i.e., the prediction $X\beta$).

Log-transforming data

Log-transforming the y variable is an important preprocessing step in many analyses: above, it turned an exponential-looking relationship into a linear one. We're saying that $\log(y) = X\beta$, or equivalently that $y = \exp(X\beta)$.

Bayesian model

We're fitting what's called a Generalized Linear Model (GLM) using PyMC3: we'll learn more about GLMs a little later. The code used is adapted from this tutorial.

In [5]:
```
# Bayesian regression model using Gaussian likelihood (equivalent to OLS)

with pm.Model() as gaussian_model:
    # Specify glm and pass in data. This is similar to the code from
    # Lab 3 that created `theta = pm.Beta(...)`, etc., but using PyMC3's
    # GLM module sets everything up automaticaly.
    # The resulting linear model, its likelihood and
    # and all its parameters are automatically added to our model.
    glm.GLM.from_formula('log_totals ~ year', ok_turbines)
    # draw posterior samples using NUTS sampling
    gaussian_trace = pm.sample(1000, cores=2, target_accept=0.95, return_inferencedata=T
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, year, Intercept]
```

100.00% [4000/4000 00:05<00:00 Sampling 2 chains, 0 divergences]

```
/opt/conda/lib/python3.8/site-packages/pymc3/math.py:246: RuntimeWarning: divide by zero
encountered in log1p
  return np.where(x < 0.6931471805599453, np.log(-np.expm1(-x)), np.log1p(-np.exp(-x)))
Sampling 2 chains for 1_000 tune and 1_000 draw iterations (2_000 + 2_000 draws total) t
ook 5 seconds.
```

Try clicking around and inspecting the output:

In [6]:
```
gaussian_trace
```

Out[6]: arviz.InferenceData

▶ posterior

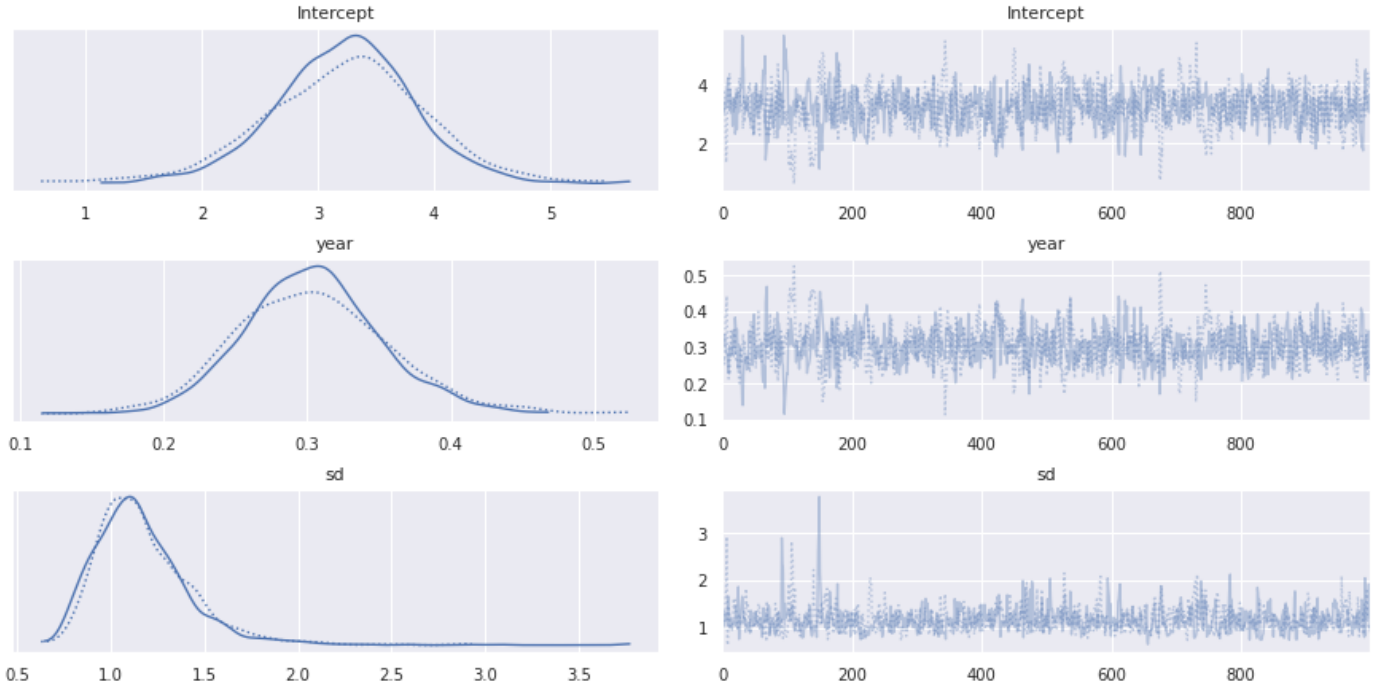
▶ log_likelihood

▶ sample_stats

▶ observed_data

In [7]:
```
arviz.plot_trace(gaussian_trace)
```

Out[7]:
```
array([[<AxesSubplot:title={'center':'Intercept'}>,
        <AxesSubplot:title={'center':'Intercept'}>],
       [<AxesSubplot:title={'center':'year'}>,
        <AxesSubplot:title={'center':'year'}>],
       [<AxesSubplot:title={'center':'sd'}>,
        <AxesSubplot:title={'center':'sd'}>]], dtype=object)
```



The plots on the left show histograms of the samples for each hidden variable (in this case, the hidden variables are the regression intercept, the regression slope (coefficient of x), and the standard deviation of the errors. The plots on the right show how those variables changed from sample to sample.

The mean of the coefficient for `year` is around 0.4. What does this mean? It means that for every unit increase in `year` (i.e., every year), we see a linear increase of about 0.4 in `log_totals`. But we're not really interested in `log_totals`!

We're really interested in how `year` affects the turbine count (rather than the log). Let N_t be the number of turbines in year t, and let $y_t = \log(N_t)$ be what we're predicting in the regression above. Then we have:

$$y_{t+1} = y_t + 0.4$$
$$\log(N_{t+1}) = \log(N_t) + 0.4$$
$$N_{t+1} = N_t e^{0.4}$$

In other words, every year, this regression tells us that the prediction is $e^{0.4}$ times the value from the previous year. That's the effect that log-transforming the data has on the output: instead of predicting an additive increase, we're predicting a multiplicative increase.

In [8]:
```
np.exp(0.4)
```

Out[8]:
```
1.4918246976412703
```

That's an increase of about 49% every year.

In [9]:
```
# From the graph above, it looks like the coefficient could
# reasonably be between 0.25 and 0.55. How does that affect
```

```
# the year-over-year change?
(np.exp(0.25), np.exp(0.55))
```

Out[9]: (1.2840254166877414, 1.7332530178673953)

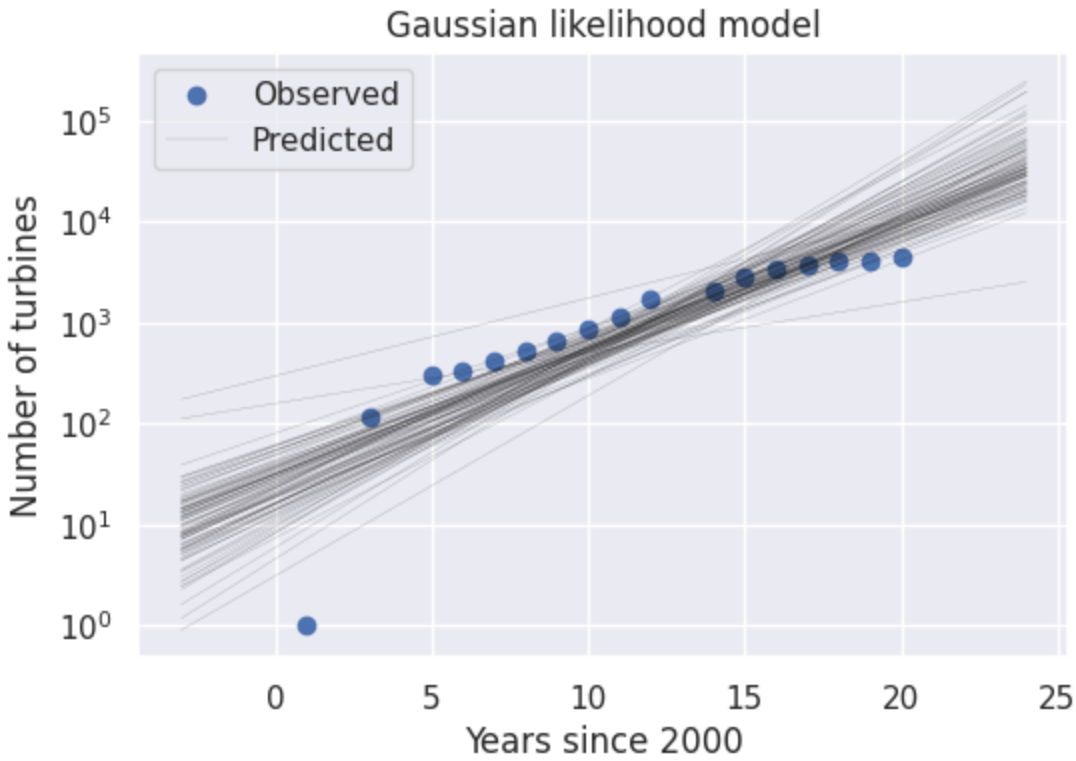
How can we connect this uncertainty ("the coefficient is probably between 0.25ish and 0.55ish") with the visual representation of the data from the plot we made earlier?

This utility will bridge that gap: it samples several lines and plots them.

In [10]:
```
def show_posterior_predictive(
    trace, x=ok_turbines['year'], y=ok_turbines['totals'], num_lines=100,
    title=""
):
    """
    Makes a posterior predictive plot for turbine data,
    showing possible lines in gray
    """

    converter = lambda x, sample: np.exp(sample['Intercept'] + sample['year'] *  x)
    plt.figure(figsize=(6, 4), dpi=100)
    plt.semilogy(x, y, marker='o', linestyle='', label='Observed')
    plt.xlabel(f'Years since 2000')
    plt.ylabel('Number of turbines')
    pm.plot_posterior_predictive_glm(
        trace, samples=num_lines, eval=np.arange(-3, 25),
        lm=converter, label='Predicted', alpha=0.4
    )
    plt.title(title)
    plt.legend()
```

In [11]:
```
show_posterior_predictive(gaussian_trace, title="Gaussian likelihood model")
```



We can see that the candidate lines are all reasonable. Note that for all of them, the slope is somewhat on the lower side: this is due to the outlier value of 1 turbine in 2001.

Count regression

The regression model above works reasonably well, but it doesn't account for the fact that the variable we're predicting is a whole number (meaning that it takes values $0, 1, 2, 3, \ldots$). When we say that $y|\beta \sim N(X\beta, \sigma^2 I)$, and using log-transformed data for y, we're implicitly saying that y can never be 0. Can we use a different likelihood that is designed specifically for this kind of data?

We've already seen one different model for the case where y is binary: logistic regression. In this section, we'll explore two models for predicting count data: Poisson regression and negative binomial regression.

Poisson regression

Recall that the Poisson distribution is a distribution over counts and count-like values. It has one *positive* parameter λ that represents its mean (and variance, too).

In Poisson regression, we're going to assume a Poisson likelihood for each y_i. The parameter of the distribution must be positive, but $x_i^T \beta$ could be negative. There are several ways to make the possibly-negative value into a positive one, but we'll use $\exp(x_i^T \beta)$ as the mean. This way, we don't have to log-transform the data. We can write out our likelihood:

$$y_i|\beta \sim \text{Poisson}(\exp(x_i^T \beta))$$

Let's try it out in PyMC3! Even though the different likelihood means that we're optimizing a completely different loss function, we only need to make a few tiny changes to our code from earlier:

In [12]:
```
# Not counting variable name changes, there are two differences
# between this cell and the version we did before: can you find them?
with pm.Model() as poisson_model:
    glm.GLM.from_formula('totals ~ year', ok_turbines, family=glm.families.Poisson())
    # draw posterior samples using NUTS sampling
    poisson_trace = pm.sample(1000, cores=2, target_accept=0.95, return_inferencedata=Tr
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [mu, year, Intercept]
```

100.00% [4000/4000 00:31<00:00 Sampling 2 chains, 0 divergences]

```
/opt/conda/lib/python3.8/site-packages/pymc3/math.py:246: RuntimeWarning: divide by zero
encountered in log1p
  return np.where(x < 0.6931471805599453, np.log(-np.expm1(-x)), np.log1p(-np.exp(-x)))
/opt/conda/lib/python3.8/site-packages/pymc3/math.py:246: RuntimeWarning: divide by zero
encountered in log1p
  return np.where(x < 0.6931471805599453, np.log(-np.expm1(-x)), np.log1p(-np.exp(-x)))
Sampling 2 chains for 1_000 tune and 1_000 draw iterations (2_000 + 2_000 draws total) t
ook 31 seconds.
```

One consequence of the Poisson likelihood is that $E[y|\beta] = exp(x_i^T \beta)$, which means that our interpretation of the coefficient(s) is the same as it was in the earlier case.

In [13]:
```
poisson_trace
```

Out[13]: arviz.InferenceData

▶ posterior

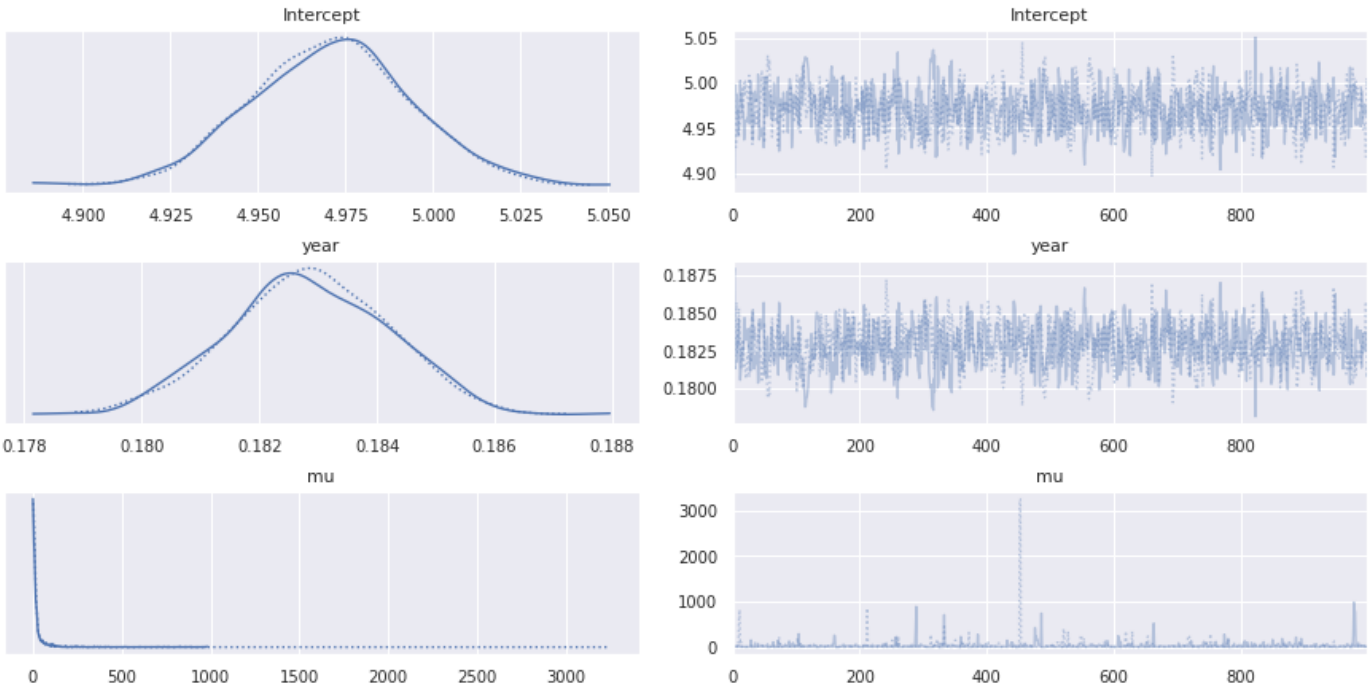
▶ log_likelihood

▶ sample_stats

▶ observed_data

In [14]:
```
arviz.plot_trace(poisson_trace)
```

Out[14]:
```
array([[<AxesSubplot:title={'center':'Intercept'}>,
        <AxesSubplot:title={'center':'Intercept'}>],
       [<AxesSubplot:title={'center':'year'}>,
        <AxesSubplot:title={'center':'year'}>],
       [<AxesSubplot:title={'center':'mu'}>,
        <AxesSubplot:title={'center':'mu'}>]], dtype=object)
```



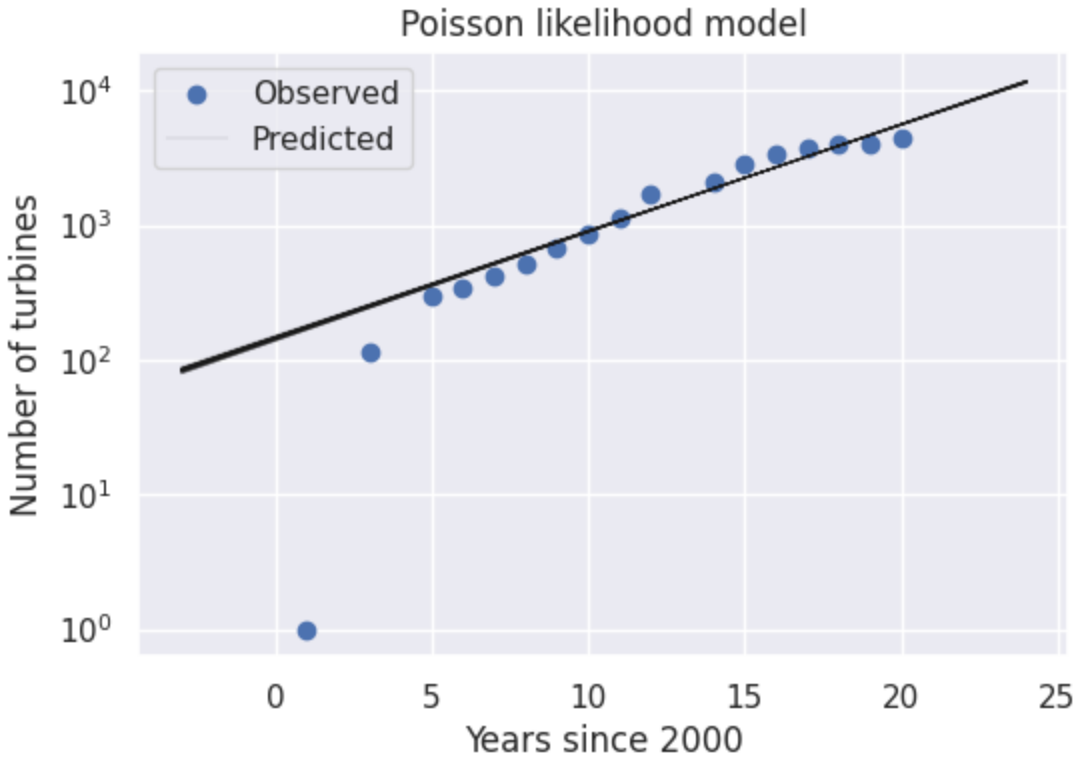
Comparing this to the results from the Gaussian model, we can see that the posteriors are **much** narrower. On top of that, the values of the coefficient for `year` seem much smaller: whereas before we saw values around 0.4, now it looks like the values are around 0.203.

In [15]:
```
np.exp(0.203)
```

Out[15]:
```
1.2250724682474992
```

This corresponds to only a 22.5% average growth rate, compared to our 49% average growth rate from the earlier model. Why are the results so different? Let's look at some sample lines and see if we can understand what's going on.

In [16]:
```
show_posterior_predictive(poisson_trace, title="Poisson likelihood model")
```



From this plot, we can see that all of the gray lines look almost exactly the same! Why is this happening? Let's think about the implicit assumptions we're making when choosing a Poisson likelihood. The Poisson distribution's mean is equal to its variance. So, when we use a Poisson distribution for y, we're implicitly assuming that $E[y]$ and $\text{var}(y)$ are reasonably close together. Does this assumption hold up?

In [17]:
```
print("Mean:    ", np.mean(ok_turbines.totals))
print("Variance:", np.var(ok_turbines.totals))
```

```
Mean:     1793.7058823529412
Variance: 2399097.9723183387
```

Clearly, the Poisson is a poor choice for fitting this data! When the model assumes a lower variance than is actually present in the data, we say that the data are **overdispersed** (i.e., that they're too spread out relative to the model's assumptions). We should choose a different distribution that gives us the ability to control the variance as well as the mean.

(Note that this wasn't a problem with the normal likelihood earlier: because the normal distribution has two separate parameters for mean and variance, we can choose them separately to reflect the fact that the variance may be higher than the mean.)

Negative binomial regression

The negative binomial distribution is also a distribution over counts, but it's more sophisticated than the Poisson distribution. We can think of it one of two ways:

- It's the sum of r [Geometric random variables], each with parameter p (success probability).
- It's like a Poisson distribution if the mean parameter (λ above) were also random.

There are several different ways to parametrize the negative binomial. How do we choose which one to use? There are two answers:

1. We want a parametrization that lets us choose the mean, since we want the mean value for y_i to be $\exp(x_i^T \beta)$.
2. Since we're using PyMC3, we're limited to whichever parametrization(s) it supports.

Even though the form of the distribution is significantly more complex and manipulating it involves more work, using it in our regression model requires only a tiny change to what we were doing before:

In [18]:
```
# Not counting variable name changes, there's only one difference
# between this cell and the Poisson regression code: can you find it?

with pm.Model() as negbin_model:
    glm.GLM.from_formula('totals ~ year', ok_turbines, family=glm.families.NegativeBinom
    # draw posterior samples using NUTS sampling
    negbin_trace = pm.sample(1000, cores=2, target_accept=0.95, return_inferencedata=Tru
```

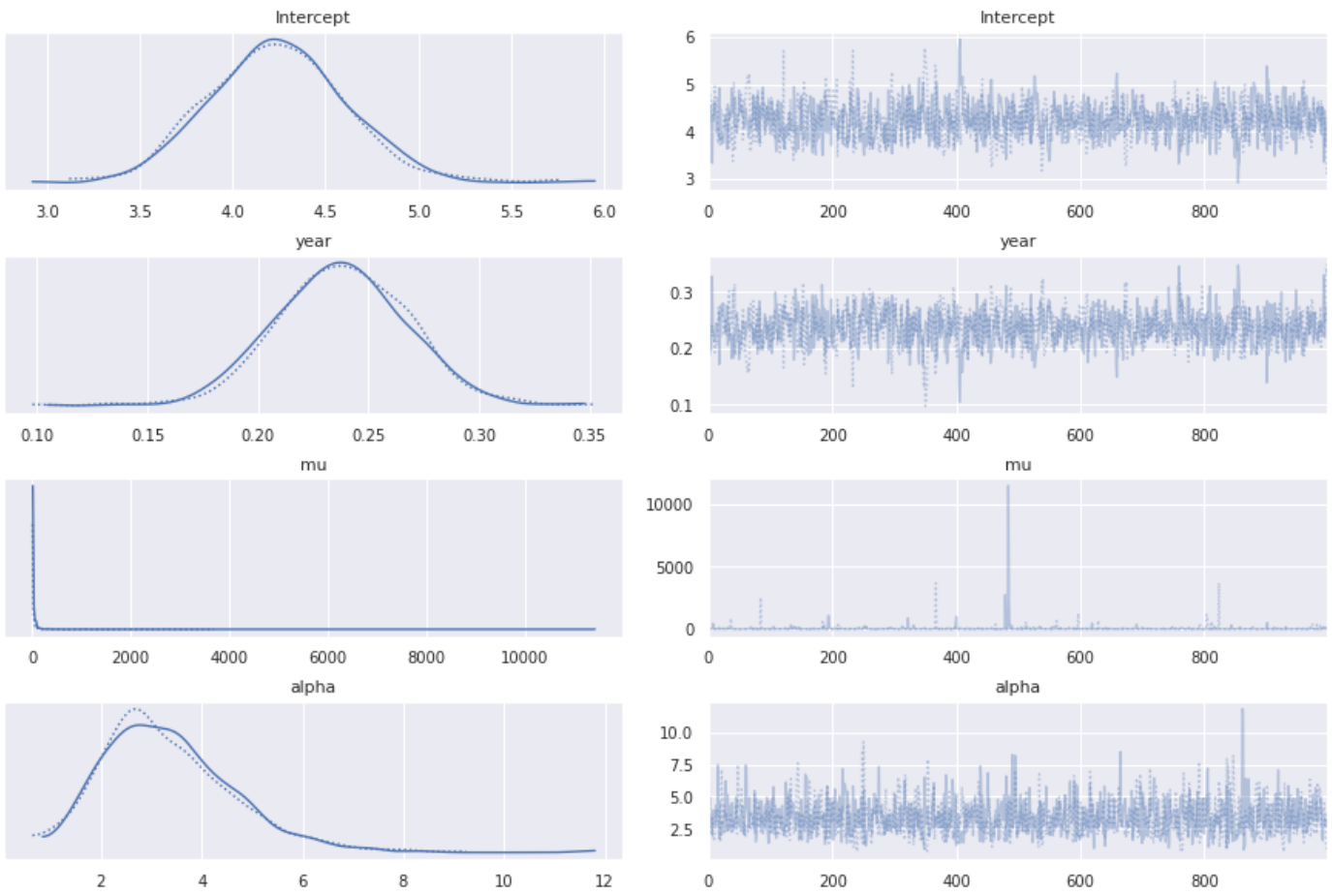
```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [alpha, mu, year, Intercept]
```

100.00% [4000/4000 00:09<00:00 Sampling 2 chains, 0 divergences]

```
Sampling 2 chains for 1_000 tune and 1_000 draw iterations (2_000 + 2_000 draws total) t
ook 10 seconds.
```

In [19]:
```
arviz.plot_trace(negbin_trace)
```

Out[19]:
```
array([[<AxesSubplot:title={'center':'Intercept'}>,
        <AxesSubplot:title={'center':'Intercept'}>],
       [<AxesSubplot:title={'center':'year'}>,
        <AxesSubplot:title={'center':'year'}>],
       [<AxesSubplot:title={'center':'mu'}>,
        <AxesSubplot:title={'center':'mu'}>],
       [<AxesSubplot:title={'center':'alpha'}>,
        <AxesSubplot:title={'center':'alpha'}>]], dtype=object)
```



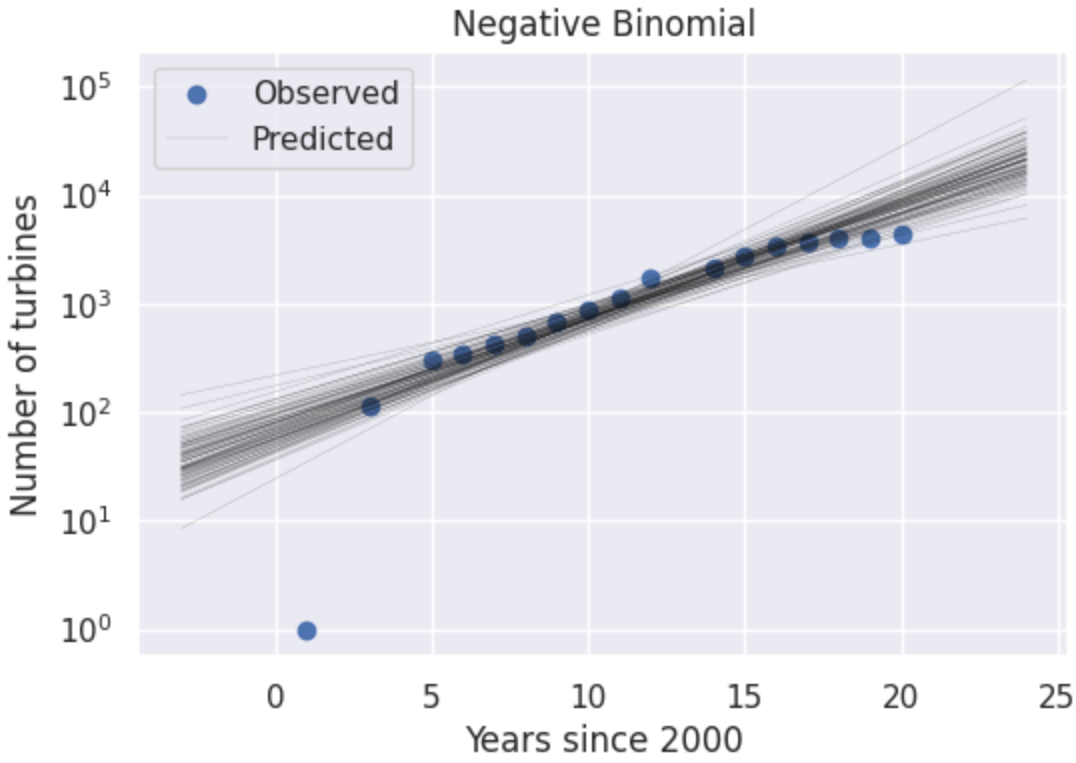
Here, the posterior distribution for the x coefficient has a wider spread again. It looks like the mean is around 0.26:

In [20]:
```
np.exp(0.26)
```

Out[20]:
```
1.2969300866657718
```

This corresponds to a growth rate around 30%. Let's see how the fit lines look:

In [21]:
```
show_posterior_predictive(negbin_trace, title="Negative Binomial")
```



These look much better than either of the previous two: we're more robust to the outlier value in 2001, but the posterior properly captures uncertainty in the prediction.

What happened to the priors?

You may have noticed that I claimed we're doing "Bayesian inference", but there were no priors used above! By default, PyMC3 uses a "flat" prior, which is uniform over all real numbers. This is what's called an "improper" prior, because it doesn't and can't integrate to 1 (or to any finite number).

You can specify priors for β: can you find out how using the documentation?

Generalized Linear Models

By now, you've seen four different versions of regression in the Bayesian setting:

- Linear regression, for predicting real-valued outputs
- Logistic regression, for predicting binary outputs (classification)
- Poisson regression, for predicting counts
- Negative binomial regression, for predicting counts

Let's review what they had in common and what was different between them:

1. For all four, computing our prediction for y_i begins with computing $x_i^T \beta$. This part is a *linear* function of x_i, even if we do something nonlinear with it later.

2. Each one had a different function that we used to compute the average value of y_i from $x_i^T \beta$. Since this function links the linearly transformed input x to the output y, you might expect us to call it the **link function**: this would make a lot of sense. However, the convention is to do the opposite, and call it the **inverse link function**. As you might expect from this name, the **link function** is the inverse of the inverse link function.

3. For each one, we used a different distribution for the likelihood. In all cases, the output of the function above was always the mean of this distribution.

The following table summarizes the different choices of likelihood and link function for the four versions that we've seen:

Regression	Inverse link function	Link function	Likelihood
Linear	identity	identity	Gaussian
Logistic	sigmoid	logit	Bernoulli
Poisson	exponential	log	Poisson
Negative binomial	exponential	log	Negative binomial

These ideas form the basis for what are known as Generalized Linear Models, or GLMs. Once we choose a link function and a likelihood distribution, our model is fully specified, and we can approximate the posterior distribution over the coefficients in β.

Frequentist GLMs

We'll look at the same models we implemented before, but this time we'll use the `statsmodels` package to look at things through a frequentist lens.

In [22]:
```
gaussian_model = sm.GLM(
    np.log(ok_turbines.totals), ok_turbines.year,
    family=sm.families.Gaussian()
)
gaussian_results = gaussian_model.fit()
print(gaussian_results.summary())
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:                 totals   No. Observations:                   17
Model:                            GLM   Df Residuals:                       16
Model Family:                Gaussian   Df Model:                            0
Link Function:               identity   Scale:                          3.3610
Method:                          IRLS   Log-Likelihood:                -33.911
Date:                Wed, 17 Feb 2021   Deviance:                       53.776
Time:                        12:51:50   Pearson chi2:                     53.8
No. Iterations:                     3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
year           0.5346      0.035     15.098      0.000       0.465       0.604
==============================================================================
```

This model's results don't look like the others, because it's missing an intercept. We need to use the `add_constant` function in `statsmodels`:

In [23]:
```
gaussian_model_intercept = sm.GLM(
    np.log(ok_turbines.totals), sm.add_constant(ok_turbines.year),
    family=sm.families.Gaussian()
)
gaussian_results = gaussian_model_intercept.fit()
print(gaussian_results.summary())
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:                 totals   No. Observations:                   17
Model:                            GLM   Df Residuals:                       15
Model Family:                Gaussian   Df Model:                            1
Link Function:               identity   Scale:                          1.1810
Method:                          IRLS   Log-Likelihood:                -24.472
Date:                Wed, 17 Feb 2021   Deviance:                       17.716
Time:                        12:51:50   Pearson chi2:                     17.7
No. Iterations:                     3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          3.2602      0.590      5.526      0.000       2.104       4.417
year           0.3023      0.047      6.435      0.000       0.210       0.394
==============================================================================
```

Take a minute to look over the results. What does it mean that the standard error for the estimated `year` coefficient is 0.047?

In [24]:
```
poisson_model = sm.GLM(
    ok_turbines.totals, sm.add_constant(ok_turbines.year),
    family=sm.families.Poisson()
)
poisson_results = poisson_model.fit()
print(poisson_results.summary())
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:                 totals   No. Observations:                   17
Model:                            GLM   Df Residuals:                       15
Model Family:                 Poisson   Df Model:                            1
Link Function:                    log   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:                -755.42
Date:                Wed, 17 Feb 2021   Deviance:                       1366.3
Time:                        12:51:51   Pearson chi2:                 1.20e+03
No. Iterations:                     5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          4.9697      0.023    219.386      0.000       4.925       5.014
year           0.1829      0.001    132.547      0.000       0.180       0.186
==============================================================================
```

The coefficients on `p_year` looks similar to what we got with the Bayesian approach. Notice that the standard error is extremely small: we have the same problem with overconfidence!

Note that in addition to the coefficients at the bottom, we also get goodness of fit measures such as log-likelihood, deviance, and chi-squared: we'll talk a little more about these and what they mean later.

For now, let's try the negative binomial model:

In [25]:
```
negbin_model = sm.GLM(
    ok_turbines.totals, sm.add_constant(ok_turbines.year),
    family=sm.families.NegativeBinomial()
)
negbin_results = negbin_model.fit()
print(negbin_results.summary())
```

Generalized Linear Model Regression Results

Dep. Variable:	totals	No. Observations:	17
Model:	GLM	Df Residuals:	15
Model Family:	NegativeBinomial	Df Model:	1
Link Function:	log	Scale:	1.0000
Method:	IRLS	Log-Likelihood:	-134.14
Date:	Wed, 17 Feb 2021	Deviance:	7.1483
Time:	12:51:51	Pearson chi2:	1.90
No. Iterations:	11		
Covariance Type:	nonrobust		

	coef	std err	z	P>|z|	[0.025	0.975]
const	4.2059	0.544	7.725	0.000	3.139	5.273
year	0.2389	0.043	5.514	0.000	0.154	0.324