Robust Bayesian inference of network structure from unreliable data

A tutorial on how to use our Stan model to infer the structure and properties of empirical complex systems from noisy, error-prone measurements. Based on the article "Robust Bayesian inference of network structure from unreliable data,", by J.-G. Young, G. T. Cantwell, and M.E.J. Newman.

Author: J.-G. Young jean-gabriel.young@uvm.edu
Date: August 2020

Important Note: In this notebook, I assume that you have installed `pystan` and will interact with the module in `python3` (check the README file otherwise). Check the project page for instruction on the R interface.

In [1]:
```
# external
import pystan
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
import networkx as nx
import pandas as pd

# core python modules
import pickle
import sys
import itertools as it

%matplotlib inline


print('python', sys.version)
print('pystan', pystan.__version__)
print('numpy', np.__version__)
print('networkx', nx.__version__)
print('pandas', pd.__version__)
```

```
python 3.8.6 (default, Sep 30 2020, 04:00:38)
[GCC 10.2.0]
pystan 2.19.1.1
numpy 1.19.2
networkx 2.5
pandas 1.1.3
```

The data

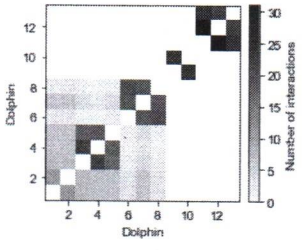
In this notebook, we reproduce the case-study of the dolphin network appearing in Section IIIA of our paper, "Robust Bayesian inference of network structure from unreliable data." Let's load this data first.

In [2]:
```
X = np.array(pd.read_csv('dolphin.csv', names=range(13)))
X[np.diag_indices(13)] = 0
```

To get a feeling for the data, we can start by reproducing Figure 1, which shows the raw data.

In [3]:
```
def set_axes():
    """Set origin at bottom left and set ticks"""
    plt.xticks(np.arange(1, 13, 2), np.arange(2, 14, 2));
    plt.yticks(np.arange(1, 13, 2), np.arange(2, 14, 2));
    plt.xlim(-0.5, 12.5)
    plt.ylim(-0.5, 12.5)

plt.figure(figsize=(3,3))
set_axes()
im=plt.imshow(X, cmap=plt.cm.Greys)
set_axes()
plt.colorbar(im,fraction=0.046, pad=0.04, label='Number of interactions')
plt.xlabel('Dolphin');
plt.ylabel('Dolphin');
```



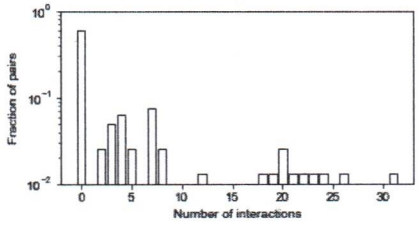
It is a symmetric 13x13 array, whose entry `X[i,j]` gives the number of observed interactions between dolphins `i` and `j`. To see how these entries are distributed, let's also plot the histogram of the entries --- that's Figure 1b.

In [4]:
```
# Construct the histogram of interaction
num_in_class = np.zeros(X.max() + 1)
obs_in_class = list(range(X.max() + 1))
for i, j in it.combinations(range(13), 2):
    num_in_class[X[i, j]] += 1
```

In [5]:
```
plt.figure(figsize=(5,2.5))

plt.bar(obs_in_class, num_in_class / np.sum(num_in_class), edgecolor='#333333', color='w')
plt.gca().set_yscale('log')
plt.xlabel('Number of interactions')
```

```
plt.ylabel('Fraction of pairs')
plt.ylim(1e-2,1);
```



Many dolphins don't interact at all, but some interact as many as 31 times.

The model

In the paper, we propose that as a first model, we can suppose that two dolphins interact a lot when they are connected, whereas an absence of ties translates into few recorded interactions. Hence, we propose to model `X[i,j]` as a Poisson random variable, with a large mean `lambda_1` if `i` and `j` are connected, and with a smaller mean `lambda_0<lambda_1` otherwise. In the notation used in the paper, we have the model :

$$\mu_{ij}(0,\lambda_0) = \frac{\lambda_0^{x_{ij}}}{x_{ij}!} e^{-\lambda_0}, \quad \mu_{ij}(1,\lambda_1) = \frac{\lambda_1^{x_{ij}}}{x_{ij}!} e^{-\lambda_1}$$

where `mu_ij(0)` stands for the distribution of `X[i,j]` when `i` and `j` are not connected, and similarly for `mu_ij(1)` .

Further, we argue for the following prior for the edges :

$$\nu_{ij}(0,\rho) = 1-\rho, \quad \nu_{ij}(1,\rho) = \rho$$

That is, we assume that the edges are all equally likely a priori, with unknown probability `rho` . The model is completed by using semi-normal distribution as priors for `lambda_0` and `lambda_1` , and a uniform prior on `rho` . (See the paper for more details on the notation and model.)

Estimation

As we show in the paper, one can fit this model by sampling from the distribution of parameters--- `lambda_0` , `lambda_1` and `rho` --- conditioned on `X` , and then from the distribution of networks conditioned on these parameters. (This is possible because the model is essentially a finite mixture whose components are indexed by pairs of nodes).

The first part, sampling the parameters, is done with `stan` .

Sampling parameters

Stan model

First, we have to load the `stan` model. It can be compiled together with all the other example models, using the provided compiler script. Alternatively, we can compile it inline:

In [6]:
```
# Uncomment and run if model was already compiled
with open('../examples/poisson_data_ER_prior.bin', 'rb') as f:
    model = pickle.load(f)
```

In [7]:
```
## Uncomment and run to compile model inline
# model = pystan.StanModel('../examples/poisson_data_ER_prior.stan')
```

The model code looks like this, and implements the parameter sampling step

In [8]:
```
print(model)
```

```
StanModel object 'poisson_data_ER_prior_c9cae126dadffc02c70fa844b7826077' coded as follows:
data {
  int<lower=1> n;
  int<lower=0> X[n, n];
  real<lower=0> rates_std_prior[2];
  real<lower=0> rho_prior[2];
}
parameters {
  positive_ordered[2] rates;
  real<lower=0, upper=1> rho;
}
model {
  rates[1] ~ normal(1, rates_std_prior[1]);
  rates[2] ~ normal(1, rates_std_prior[2]);
  rho ~ beta(rho_prior[1], rho_prior[2]);

  for (i in 1:n) {
    for (j in i + 1:n) {
      real log_mu_ij_0 = poisson_lpmf(X[i, j] | rates[1]);
      real log_mu_ij_1 = poisson_lpmf(X[i, j] | rates[2]);

      real log_nu_ij_0 = bernoulli_lpmf(0 | rho);
      real log_nu_ij_1 = bernoulli_lpmf(1 | rho);

      real z_ij_0 = log_mu_ij_0 + log_nu_ij_0;
      real z_ij_1 = log_mu_ij_1 + log_nu_ij_1;
      if (z_ij_0 > z_ij_1) {target += z_ij_0 + log1p_exp(z_ij_1 - z_ij_0);}
      else {target += z_ij_1 + log1p_exp(z_ij_0 - z_ij_1);}
    }
  }
}
generated quantities {
  real Q[n ,n];
  for (i in 1:n) {
    Q[i, i] = 0;
    for (j in i+1:n) {
```

```
      real log_mu_ij_0 = poisson_lpmf(X[i, j] | rates[1]);
      real log_mu_ij_1 = poisson_lpmf(X[i, j] | rates[2]);

      real log_nu_ij_0 = bernoulli_lpmf(0 | rho);
      real log_nu_ij_1 = bernoulli_lpmf(1 | rho);

      real z_ij_0 = log_mu_ij_0 + log_nu_ij_0;
      real z_ij_1 = log_mu_ij_1 + log_nu_ij_1;
      Q[i, j] = 1 / (1  + exp(z_ij_0 - z_ij_1));
      Q[j, i] = Q[i, j];
    }
  }
}
```

Some stan background (skip if you just want results)

Note the variables `log_mu_ij_0` , `log_mu_ij_1` , `log_nu_ij_0` and `log_nu_ij_1` . These variables are where the model is specified. In the present case, we have specified that `mu_ij` is a Poisson variable, and that `nu_ij` is a Bernoulli variable. (They appear twice, once in the model block and once in the generated quantities block, due to idiosyncratic details of how `Stan` works.)

Much of the remaining code is boilerplate, and provided already in our templates.

The fields that are specific to the present models are, in the `data` block:

- `rates_std_prior` are the standard deviations for the priors on the rates `lambda_0` and `lambda_1`
- `rho_prior` are parameters of the prior on rho, which is set to a Beta distribution here. We'll use 1 in both entries to encode a uniform distribution.

The `parameters` fields also contain model-specific information. In this case, the parameters that will be sampled are

- `rates` : The rates `lambda_0` and `lambda_1`
- `rho` : The density

Lastly, the priors that we chose appear at the top of the `model` block.

Sampling

So let's fit the model to the dolphin data. We'll use the default sampling parameters of `Stan` : 4 independent MCMC chains, of length 2000 each, with half of the iterations spent in burn-in.

In [9]:
```
# Sample
fit = model.sampling(data={"n": X.shape[0],
                           "X": X,
                           "rates_std_prior": [100, 100],
                           'rho_prior': [1,1]})
```

```
WARNING:pystan:n_eff / iter below 0.001 indicates that the effective sample size has likely been overestimated
WARNING:pystan:Rhat above 1.1 or below 0.9 indicates that the chains very likely have not mixed
```

Let's take a look at the fit.

In [10]:
```
fit
```

```
WARNING:pystan:Truncated summary with the 'fit.__repr__' method. For the full summary use 'print(fit)'
```

Out[10]:
```
Warning: Shown data is truncated to 100 parameters
For the full summary use 'print(fit)'

Inference for Stan model: poisson_data_ER_prior_c9cae126dadffc02c70fa844b7826077.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

           mean se_mean      sd   2.5%    25%    50%     75%  97.5%  n_eff   Rhat
rates[1]   0.62  8.5e-3    0.21   0.08   0.52   0.65    0.76   0.96    610   1.01
rates[2]  14.31    0.06    1.43  10.96  13.52  14.46   15.25  16.81    585   1.01
rho        0.28  2.1e-3    0.06   0.18   0.24   0.28    0.32   0.42    815    1.0
Q[1,1]      0.0     nan     0.0    0.0    0.0    0.0     0.0    0.0    nan    nan
Q[2,1]      1.0 2.1e-10  1.2e-8    1.0    1.0    1.0     1.0    1.0   3155    1.0
Q[3,1]      1.0  1.2e-5  6.2e-4    1.0    1.0    1.0     1.0    1.0   2771    1.0
Q[4,1]      1.0  1.8e-4  9.4e-3   0.97    1.0    1.0     1.0    1.0   2594    1.0
Q[5,1]      1.0  1.8e-4  9.4e-3   0.97    1.0    1.0     1.0    1.0   2594    1.0
Q[6,1]     0.22    0.01     0.3 6.1e-3   0.03   0.08    0.26    1.0    467   1.01
Q[7,1]      1.0  1.8e-4  9.4e-3   0.97    1.0    1.0     1.0    1.0   2594    1.0
Q[8,1]     0.22    0.01     0.3 6.1e-3   0.03   0.08    0.26    1.0    467   1.01
Q[9,1]   1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
Q[10,1]  1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
Q[11,1]  1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
Q[12,1]  1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
Q[13,1]  1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
Q[1,2]      1.0 2.1e-10  1.2e-8    1.0    1.0    1.0     1.0    1.0   3155    1.0
Q[2,2]      0.0     nan     0.0    0.0    0.0    0.0     0.0    0.0    nan    nan
Q[3,2]      1.0  1.2e-5  6.2e-4    1.0    1.0    1.0     1.0    1.0   2771    1.0
Q[4,2]      1.0  1.8e-4  9.4e-3   0.97    1.0    1.0     1.0    1.0   2594    1.0
Q[5,2]      1.0  1.8e-4  9.4e-3   0.97    1.0    1.0     1.0    1.0   2594    1.0
Q[6,2]     0.22    0.01     0.3 6.1e-3   0.03   0.08    0.26    1.0    467   1.01
Q[7,2]      1.0  1.8e-4  9.4e-3   0.97    1.0    1.0     1.0    1.0   2594    1.0
Q[8,2]     0.22    0.01     0.3 6.1e-3   0.03   0.08    0.26    1.0    467   1.01
Q[9,2]   1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
Q[10,2]  1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
Q[11,2]  1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
Q[12,2]  1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
Q[13,2]  1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
Q[1,3]      1.0  1.2e-5  6.2e-4    1.0    1.0    1.0     1.0    1.0   2771    1.0
Q[2,3]      1.0  1.2e-5  6.2e-4    1.0    1.0    1.0     1.0    1.0   2771    1.0
Q[3,3]      0.0     nan     0.0    0.0    0.0    0.0     0.0    0.0    nan    nan
Q[4,3]      1.0     nan     0.0    1.0    1.0    1.0     1.0    1.0    nan    nan
Q[5,3]      1.0 3.0e-17 1.7e-15    1.0    1.0    1.0     1.0    1.0   3241    1.0
Q[6,3]     0.07    0.01    0.21 3.2e-4 1.6e-3 3.9e-3    0.01   0.96    360   1.01
Q[7,3]     0.63  8.6e-3    0.28    0.1   0.39   0.65     0.9    1.0   1083    1.0
Q[8,3]     0.07    0.01    0.21 3.2e-4 1.6e-3 3.9e-3    0.01   0.96    360   1.01
Q[9,3]   1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
Q[10,3]  1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
Q[11,3]  1.5e-6  1.8e-7  4.1e-6 3.5e-8 1.7e-7 3.9e-7 10.0e-7 1.3e-5    511   1.01
```

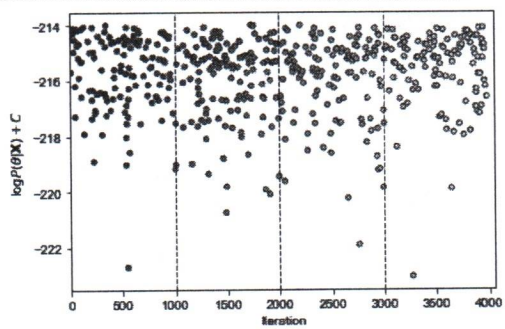
Q[12,3]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[13,3]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[1,4]	1.0	1.8e-4	9.4e-3	0.97	1.0	1.0	1.0	1.0	2594	1.0
Q[2,4]	1.0	1.8e-4	9.4e-3	0.97	1.0	1.0	1.0	1.0	2594	1.0
Q[3,4]	1.0	nan	0.0	1.0	1.0	1.0	1.0	1.0	nan	nan
Q[4,4]	0.0	nan	0.0	0.0	0.0	0.0	0.0	0.0	nan	nan
Q[5,4]	1.0	2.3e-18	1.3e-16	1.0	1.0	1.0	1.0	1.0	3295	1.0
Q[6,4]	0.02	3.9e-3	0.09	1.7e-5	7.7e-5	1.8e-4	5.6e-4	0.16	586	1.01
Q[7,4]	0.22	0.01	0.3	6.1e-3	0.03	0.08	0.26	1.0	467	1.01
Q[8,4]	0.02	3.9e-3	0.09	1.7e-5	7.7e-5	1.8e-4	5.6e-4	0.16	586	1.01
Q[9,4]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[10,4]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[11,4]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[12,4]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[13,4]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[1,5]	1.0	1.8e-4	9.4e-3	0.97	1.0	1.0	1.0	1.0	2594	1.0
Q[2,5]	1.0	1.8e-4	9.4e-3	0.97	1.0	1.0	1.0	1.0	2594	1.0
Q[3,5]	1.0	3.0e-17	1.7e-15	1.0	1.0	1.0	1.0	1.0	3241	1.0
Q[4,5]	1.0	2.3e-18	1.3e-16	1.0	1.0	1.0	1.0	1.0	3295	1.0
Q[5,5]	0.0	nan	0.0	0.0	0.0	0.0	0.0	0.0	nan	nan
Q[6,5]	0.07	0.01	0.21	3.2e-4	1.6e-3	3.9e-3	0.01	0.96	360	1.01
Q[7,5]	0.63	8.6e-3	0.28	0.1	0.39	0.65	0.9	1.0	1083	1.0
Q[8,5]	0.07	0.01	0.21	3.2e-4	1.6e-3	3.9e-3	0.01	0.96	360	1.01
Q[9,5]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[10,5]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[11,5]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[12,5]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[13,5]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[1,6]	0.22	0.01	0.3	6.1e-3	0.03	0.08	0.26	1.0	467	1.01
Q[2,6]	0.22	0.01	0.3	6.1e-3	0.03	0.08	0.26	1.0	467	1.01
Q[3,6]	0.07	0.01	0.21	3.2e-4	1.6e-3	3.9e-3	0.01	0.96	360	1.01
Q[4,6]	0.02	3.9e-3	0.09	1.7e-5	7.7e-5	1.8e-4	5.6e-4	0.16	586	1.01
Q[5,6]	0.07	0.01	0.21	3.2e-4	1.6e-3	3.9e-3	0.01	0.96	360	1.01
Q[6,6]	0.0	nan	0.0	0.0	0.0	0.0	0.0	0.0	nan	nan
Q[7,6]	1.0	nan	9.9e-18	1.0	1.0	1.0	1.0	1.0	nan	1.0
Q[8,6]	1.0	nan	0.0	1.0	1.0	1.0	1.0	1.0	nan	nan
Q[9,6]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[10,6]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[11,6]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[12,6]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[13,6]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[1,7]	1.0	1.8e-4	9.4e-3	0.97	1.0	1.0	1.0	1.0	2594	1.0
Q[2,7]	1.0	1.8e-4	9.4e-3	0.97	1.0	1.0	1.0	1.0	2594	1.0
Q[3,7]	0.63	8.6e-3	0.28	0.1	0.39	0.65	0.9	1.0	1083	1.0
Q[4,7]	0.22	0.01	0.3	6.1e-3	0.03	0.08	0.26	1.0	467	1.01
Q[5,7]	0.63	8.6e-3	0.28	0.1	0.39	0.65	0.9	1.0	1083	1.0
Q[6,7]	1.0	nan	9.9e-18	1.0	1.0	1.0	1.0	1.0	nan	1.0
Q[7,7]	0.0	nan	0.0	0.0	0.0	0.0	0.0	0.0	nan	nan
Q[8,7]	1.0	nan	9.9e-18	1.0	1.0	1.0	1.0	1.0	nan	1.0
Q[9,7]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[10,7]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[11,7]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[12,7]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[13,7]	1.5e-6	1.8e-7	4.1e-6	3.5e-8	1.7e-7	3.9e-7	10.0e-7	1.3e-5	511	1.01
Q[1,8]	0.22	0.01	0.3	6.1e-3	0.03	0.08	0.26	1.0	467	1.01
Q[2,8]	0.22	0.01	0.3	6.1e-3	0.03	0.08	0.26	1.0	467	1.01
Q[3,8]	0.07	0.01	0.21	3.2e-4	1.6e-3	3.9e-3	0.01	0.96	360	1.01
Q[4,8]	0.02	3.9e-3	0.09	1.7e-5	7.7e-5	1.8e-4	5.6e-4	0.16	586	1.01
Q[5,8]	0.07	0.01	0.21	3.2e-4	1.6e-3	3.9e-3	0.01	0.96	360	1.01
lp__	-215.7	0.06	1.45	-219.3	-216.5	-215.3	-214.6	-214.1	644	1.01

```
Samples were drawn using NUTS at Mon Oct 26 14:33:41 2020.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Everything appears normal save for a few entries of `Q[i,j]` ---the posterior probability that an edge exists---that do not have any variability. In factm this lack of variable causes a warning! One should look into **any** warning messages returned by stan. They are usually a sign that something's wrong with the model. In our cases, however, this is expected since the faulty entries are the diagonal of `Q`, which we set to 0 manually in the model since we don't consider self-loops. So there's no actual error---we can proceed.

We can do a few extra sanity check. For example, let's look at log posterior probability accross independent chains:

In [11]:
```
subset = sorted(np.random.choice(range(4000), size=500, replace=False))
plt.scatter(subset, fit['lp__'][subset], s=15, edgecolor='#333333', linewidth=1, c=subset, alpha=1, cmap=plt.cm.Blues_r)
plt.xlabel('Iteration')
plt.ylabel(r'$\log P(\theta|\mathbf{X}) + C$')
for bar in [1000,2000,3000]:
    plt.axvline(bar, c='k', ls='--', lw=1)
plt.xlim(0,4050);
```

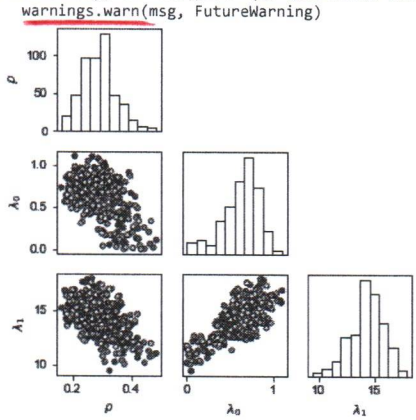


It is consistent across chains, and appears to have reached a steady-state. To see if the samples are also qualitatively similar, we can look at the pair-plot for the key parameters

In [12]:
```
pp = pd.DataFrame(np.array([fit['rho'], fit['rates'][:,0], fit['rates'][:,1]]).T, columns=[r'$\rho$', r'$\lambda_0$', r'
plt.figure(figsize=(5, 5))
plot_idx = {0: 1, 1:4, 2:7, 3:5, 4: 8, 5:9}
plot_data = [{'name': 'rho', 'label': r'$\rho$', 'samples': fit['rho'][subset]},
             {'name': 'l0', 'label': r'$\lambda_0$', 'samples': fit['rates'][subset,0]},
```

```
            {'name': 'l1', 'label': r'$\lambda_1$', 'samples': fit['rates'][subset,1]}]
for i, pdata in enumerate(it.combinations_with_replacement(plot_data, 2)):
    plt.subplot(3, 3, plot_idx[i])
    if pdata[0]['name'] == pdata[1]['name']:
        # diagonal, histogram
        sns.distplot(pdata[0]['samples'], kde=False, color='w', hist_kws={"edgecolor" : '#333333', 'lw':1, 'alpha':1}, b
    else:
        # pairs
        plt.scatter(pdata[0]['samples'], pdata[1]['samples'], s=15, edgecolor='#333333', linewidth=1, c=subset, alpha=1,
    if plot_idx[i] < 7:
        plt.xticks([],[])
    else:
        plt.xlabel(pdata[0]['label'])
    if plot_idx[i] not in [1,4,7]:
        plt.yticks([],[])
    else:
        plt.ylabel(pdata[1]['label'])
```

```
/usr/lib/python3.8/site-packages/seaborn/distributions.py:2551: FutureWarning: `distplot` is a deprecated function and w
ill be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar
flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



and again we see no apparent issue.

As we see from this figure, the posterior distribution of the rates are centered close to 0 and 15. The estimated density is around 0.25.

So we have sampled the parameters of our model!

Network samples

Posterior probability of edges

The next step is to sample networks based on the samples of `lambda_0` , `lambda_1` and `rho` . In the paper, we show that one can use these parameters and the data matrix `X` to compute the posterior probability `Q_ij` that there is an edge between node `i` and `j` . As we have already seen, the `stan` model computes this probability automatically and returns it too in the `fit` object.

Let's focus on the first sample:

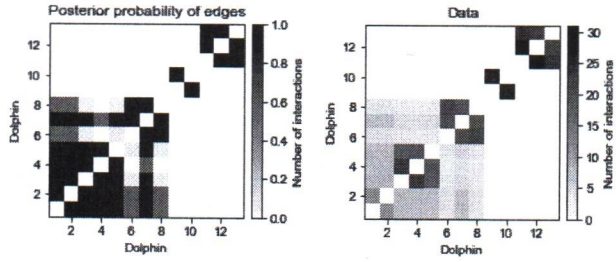
In [13]:
```
sample_idx = 0
print(fit['rho'][sample_idx], fit['rates'][sample_idx,0], fit['rates'][sample_idx,1])
```

```
0.34032325641231853 0.4817810576987878 12.583352885377606
```

The associated matrix of edge probabilities look like:

In [14]:
```
plt.figure(figsize=(7,3))
plt.subplot(121)
plt.title('Posterior probability of edges')
set_axes()
im=plt.imshow(fit['Q'][sample_idx,:], cmap=plt.cm.Greys)
set_axes()
plt.colorbar(im,fraction=0.046, pad=0.04, label='Number of interactions')
plt.xlabel('Dolphin');
plt.ylabel('Dolphin');


plt.subplot(122)
plt.title('Data')
im=plt.imshow(X, cmap=plt.cm.Greys)
set_axes()
plt.colorbar(im,fraction=0.046, pad=0.04, label='Number of interactions')
plt.xlabel('Dolphin');
plt.ylabel('Dolphin');
plt.tight_layout()
```



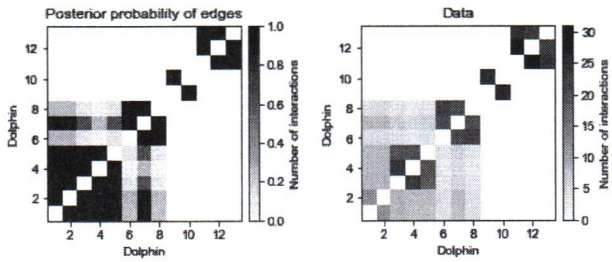
Comparing with the data, there's a clear correlation betweent he prediction and the number of observed interaction.

We can average these matrix to get a robust posterior estimate of the probability that every pair is connected:

In [15]:
```
Q = np.mean(fit['Q'], axis=0)
```

In [16]:
```
plt.figure(figsize=(7,3))
plt.subplot(121)
plt.title('Posterior probability of edges')
set_axes()
im=plt.imshow(Q, cmap=plt.cm.Greys)
set_axes()
plt.colorbar(im,fraction=0.046, pad=0.04, label='Number of interactions')
plt.xlabel('Dolphin');
plt.ylabel('Dolphin');


plt.subplot(122)
plt.title('Data')
im=plt.imshow(X, cmap=plt.cm.Greys)
set_axes()
plt.colorbar(im,fraction=0.046, pad=0.04, label='Number of interactions')
plt.xlabel('Dolphin');
plt.ylabel('Dolphin');
plt.tight_layout()
```



The predictions are similar, but now account for a broad range of parametrization, not the just the particular parameters of the first sample.

Network samples

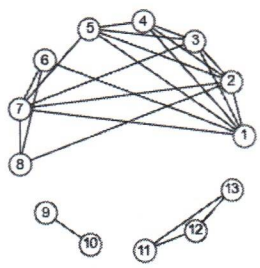
Neither the `Q` matrices nor their averages are network; they give the posterior probability of the edges.

If we want to compute network quantities, we have to generate network *samples* from these matrices. For example, using the first sampled matrix `Q`, we can generate a sample thus:

In [17]:
```
G = nx.Graph()
Q = fit['Q'][sample_idx,:]
for i,j in it.combinations(range(13), 2):
    if np.random.rand() < Q[i, j]:
        G.add_edge(i,j)
```

In [18]:
```
plt.figure(figsize=(3,3))
pos = nx.circular_layout(G)

nx.draw(G, pos=pos, linewidths=1, edgecolors='#333333', node_color='w')
nx.draw_networkx_labels(G, pos, labels={i : str(i + 1) for i in range(13)}, font_color='k');
plt.xlim(-1.2,1.2);
plt.ylim(-1.2,1.2);
```



Posterior averages over networks

To evaluate general averages over networks, we just have to loop over parameter samples, and generate a few networks for each of them. Then we'll be able to compute a histogram of the network quantity with these networks.

As an example, let's compute the posterior distribution of transitivity coefficients:

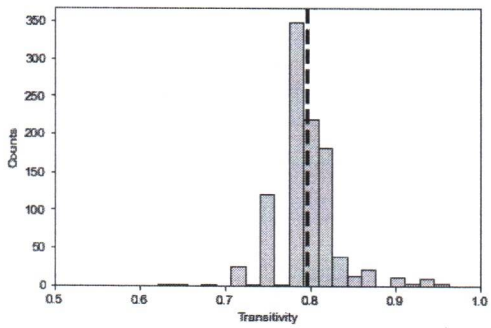
In [19]:
```
values = []

for param_id in range(100):  # loop over the first 100 network samples
    for net_id in range(10):  # generate 10 network samples for each parameter samples
        G = nx.Graph()
        Q = fit['Q'][param_id,:]
        for i,j in it.combinations(range(13), 2):
            if np.random.rand() < Q[i, j]:
                G.add_edge(i,j)
        values.append(nx.transitivity(G))
```

In [20]:
```
plt.hist(values, edgecolor='#333333', color='#DDDDDD', bins=20);
plt.xlabel('Transitivity')
plt.ylabel('Counts')
```

```
plt.axvline(np.mean(values), c='k', ls='--', lw=3);
plt.xlim(0.5, 1)
```

Out[20]: (0.5, 1.0)



So we now have a mean transitivity and error bars! Note that thresholding would have given a much different answer:

In [21]:
```
G = nx.Graph()
for i,j in it.combinations(range(13),2):
    if X[i,j] > 0:
        G.add_edge(i,j)
print("Tresholded transitivity:", nx.transitivity(G))
```

```
Tresholded transitivity: 1.0
```

Poster-predictive check

We have successfully obtained a fit of the model. The next step is to verify whether they actually fit the data well --- if they don't, then we can't trust the inference, and we'll have to revisit the model.

We'll do our model criticism using several standard Bayesian tools. First, let's compute the mean of the posterior predictive distribution, i.e., the mean data matrix `X_tilde` we get when we feed the samples back in the likelihood.

In [22]:
```
X_tilde = np.zeros((13, 13))

for param_id in range(100):  # loop over the first 100 network samples
    for net_id in range(5):  # simulate 5 network samples for each parameter samples
        G = nx.Graph()
        Q = fit['Q'][param_id,:]
        for i,j in it.combinations(range(13), 2):
            if np.random.rand() < Q[i, j]:
                X_tilde[i,j] += float(np.random.poisson(fit['rates'][param_id, 1]))
            else:
                X_tilde[i,j] += float(np.random.poisson(fit['rates'][param_id, 0]))

# Normalize and symmetrize
X_tilde /= 500
X_tilde += X_tilde.T
```

(There are faster ways to do this if we use a bit of maths, but this simulation technique will do for our purpose.)

Visual assessment

We can now compare this average posterior predictive matrix to the input `X`, to see where the model diverges from the true data and in what ways:

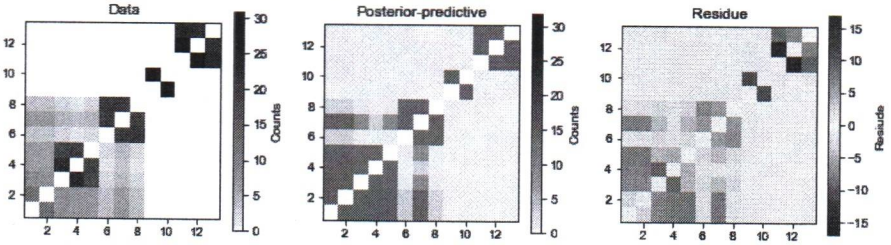
In [23]:
```
plt.figure(figsize=(10,2.8))

plt.subplot(131)
plt.title('Data')
plt.imshow(X, cmap=plt.cm.gray_r)
plt.colorbar(label='Counts')
set_axes()

plt.subplot(132)
plt.title('Posterior-predictive')
plt.imshow(X_tilde, vmin=0, vmax=32, cmap=plt.cm.gray_r)
plt.colorbar(label='Counts')
set_axes()


plt.subplot(133)
plt.title('Residue')
plt.imshow(X  - X_tilde, vmin=-17, vmax=17, cmap=plt.cm.RdBu)
plt.colorbar(label='Resiude')
set_axes()


plt.tight_layout(pad=1)
```



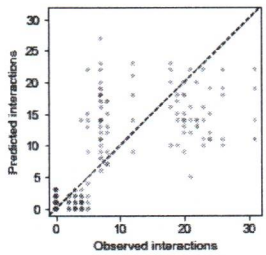
Visually the model is good at ***not*** placing any interactions where there's none in the real data. But it fails to account for both the mid-

density and high-density regions of the data matrix.

We can also see this by comparing the observed counts versus the predicted counts in a scatter plot.

In [24]:
```
plt.figure(figsize=(3,3))
for sample_id in range(5):
    # create artificial data sets
    A = np.random.rand(13, 13) < fit['Q'][sample_id,:]
    A[np.triu_indices(13, k=1)] = A.T[np.triu_indices(13, k=1)]
    X_tilde_0 = np.random.poisson(fit['rates'][sample_id, 0], size=Q.shape)
    X_tilde_1 = np.random.poisson(fit['rates'][sample_id, 1], size=Q.shape)
    X_tilde = (1 - A) * X_tilde_0  + A * X_tilde_1
    plt.scatter(X[np.triu_indices(13, k=1)], X_tilde[np.triu_indices(13,k=1)], c='blue', s=12, alpha=0.2, edgecolor='nor

plt.plot([-10, 50], [-10, 50], c='k', ls='--', lw=1)
plt.xlabel('Observed interactions')
plt.ylabel('Predicted interactions')
plt.xlim(-1, 32)
plt.ylim(-1, 32)
plt.gca().set_facecolor("none")
plt.gca().set_facecolor("none")
```



The points are far from the diagonal!

Formal assessment (skip if you just want results)

To get a more formal assessment of goodness-of-fit, we can also use the realized discrepancy technique proposed by Gelman, Meng, and Stern (1996). Basically, we must generate a bunch of parameter samples, and (i) compute how well they fit the actual data, as well as (ii) how well they fit an artificial data set generated with the samples themselves.

If the model is a good fit but does not overfit, we'll see no difference between (i) and (ii) on average.

We use the discrepancy to compute the quality of the fit provided by a particular parametrization:

$$D(\vec{X},\theta) = \Sigma_{ij}\, X_{ij} \log\left(\frac{X_{ij}}{\tilde{X}_{ij}(\theta)}\right)$$

where `X_tilde` is the average synthetic data when the parameters are fixed (`theta` stands in for `rho` , `lambda_0` and `lambda_1`).

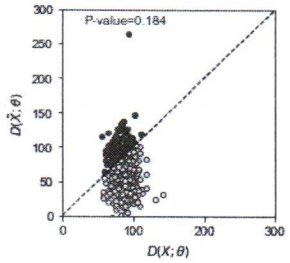
In [25]:
```
def discrepancy(X, Q, rates):
    """Compute the discrepancy"""
    # One can check that <X_tilde_ij> = Q_ij lambda_1 + (1 - Q_ij) lambda_0 in our model.
    X_tilde_mat = Q * rates[1] + (1 - Q) * rates[0]
    # Avoid double counting and numerically ill-defined 0*log(0) by selecting indices
    idx = np.triu_indices(13, k=1)
    idx2 = np.where(X[idx] > 0)[0]
    return np.sum(X[idx][idx2] * np.log(X[idx][idx2] / X_tilde_mat[idx][idx2]))


# Compute discrepancy distribution
d_data = np.zeros(500)          # data discrepancy
d_artificial = np.zeros(500)  # artificial data discrepancy

for sample_id in range(500):
    # Generate artificial data set
    A = np.random.rand(13, 13) < fit['Q'][sample_id,:]
    A[np.triu_indices(13, k=1)] = A.T[np.triu_indices(13, k=1)]
    X_tilde_0 = np.random.poisson(fit['rates'][sample_id, 0], size=Q.shape)
    X_tilde_1 = np.random.poisson(fit['rates'][sample_id, 1], size=Q.shape)
    X_tilde = (1 - A) * X_tilde_0  + A * X_tilde_1
    # calculate discrepancy
    d_data[sample_id] = discrepancy(X, fit['Q'][sample_id,:], fit['rates'][sample_id,:])
    d_artificial[sample_id] = discrepancy(X_tilde, fit['Q'][sample_id,:], fit['rates'][sample_id,:])
```

In [26]:
```
plt.figure(figsize=(3,3))
plt.scatter(d_data[d_data<d_artificial], d_artificial[d_data<d_artificial], s=15, edgecolor='#333333', linewidth=1, colo
plt.scatter(d_data[d_data>d_artificial], d_artificial[d_data>d_artificial], s=15, edgecolor='#333333', linewidth=1, colo
plt.plot([0, 300], [0, 300], c='k', ls='--', lw=1)
plt.xlim(0,300)
plt.ylim(0,300)
plt.xlabel(r'$D(X;\theta)$')
plt.ylabel(r'$D(\tilde{X};\theta)$')
plt.text(30, 280, "P-value=" + str(len(d_artificial[d_data<d_artificial]) / 500))
```

Out[26]:
```
Text(30, 280, 'P-value=0.184')
```



The data--model comparison yields lower discrepancy than the model--model comparison in only about 15% of cases. Hence, the model is not a great fit formally either.

Revisiting the model

Improved model

In the paper, we argue that we can fix the model by adding a new edge type to the network: strong ties, associated with a mean number of observed interactions `lambda_2 > lambda_1`.

Let's load this model..

In [27]:
```
## Uncomment and run if model was already compiled
with open('../examples/multitype_poisson_data_ER_prior.bin', 'rb') as f:
    model = pickle.load(f)
```

In [28]:
```
## Uncomment and run to compile model inline
# model = pystan.StanModel('../examples/multitype_poisson_data_ER_prior.stan')
```

In [29]:
```
print(model)
```

```
StanModel object 'multitype_poisson_data_ER_prior_6bced87e4f9c86d4c3c8d7ccdcddda6a' coded as follows:
data {
  int<lower=1> n;
  int<lower=0> X[n, n];
  int T; // number of edge types
  real<lower=0> rates_std_prior[T];
}
parameters {
  positive_ordered[T] rates;
  simplex[T] rho;
}
model {
  for (k in 1:T)
  {
    rates[k] ~ normal(1, rates_std_prior[k]);
  }

  for (i in 1:n) {
    for (j in i + 1:n) {
      vector[T] z_ij;
      vector[T] z_max_vector;
      real z_max;
      for (k in 1:T) {
        real log_mu_ij_k = poisson_lpmf(X[i, j] | rates[k]);
        real log_nu_ij_k = log(rho[k]);

        z_ij[k] = log_mu_ij_k + log_nu_ij_k;
      }
      z_max = max(z_ij);
      z_max_vector = rep_vector(z_max, T);
      target += z_max + log_sum_exp(z_ij - z_max_vector);
    }
  }
}
generated quantities {
  real Q[n ,n, T];
  for (i in 1:n) {
    for (k in 1:T) {
      Q[i, i, k] = 0;
    }
    for (j in i+1:n) {
      vector[T] z_ij;
      real accu;
      for (k in 1:T)
      {
        real log_mu_ij_k = poisson_lpmf(X[i, j] | rates[k]);
        real log_nu_ij_k = log(rho[k]);

        z_ij[k] =  log_mu_ij_k + log_nu_ij_k;
      }
      for (k in 1:T)
      {
        accu = 0;
        for (k_prime in 1:T)
        {
          accu += exp(z_ij[k_prime] - z_ij[k]);
        }
        Q[i, j, k] = 1 / accu;
        Q[j, i, k] = Q[i, j, k];
      }
    }
  }
}
```

This model is almost identical to before, but now has `T` edge types (we'll use 3 to encode: no edge, weak edges, strong edges). One notable difference in the code is that `rho` is now encoded as a simplex.

`rho[0]` denotes the prior probability that there's no edge, `rho[1]` the probability that there's a weak edge, and `rho[2]` the probability that there's a strong edge.

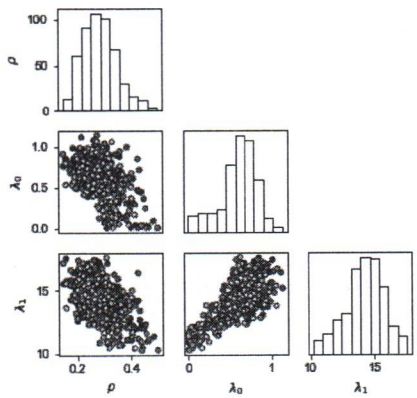
Sanity check

As a sanity check let's re-run the model on our data with T=2. We should get the same output as before.

In [30]:
```
# Sample
fit = model.sampling(data={"n": X.shape[0],
                           "X": X,
                           'T': 2,
                           "rates_std_prior": [100, 100]})
```

```
WARNING:pystan:n_eff / iter below 0.001 indicates that the effective sample size has likely been overestimated
WARNING:pystan:Rhat above 1.1 or below 0.9 indicates that the chains very likely have not mixed
```

In [31]:
```
pp = pd.DataFrame(np.array([fit['rho'][:,1], fit['rates'][:,0], fit['rates'][:,1]]).T, columns=[r'$\rho$', r'$\lambda_0$
plt.figure(figsize=(5, 5))
plot_idx = {0: 1, 1:4, 2:7, 3:5, 4: 8, 5:9}
plot_data = [{'name': 'rho', 'label': r'$\rho$', 'samples': fit['rho'][subset,1]},
             {'name': 'l0', 'label': r'$\lambda_0$', 'samples': fit['rates'][subset,0]},
             {'name': 'l1', 'label': r'$\lambda_1$', 'samples': fit['rates'][subset,1]}]
for i, pdata in enumerate(it.combinations_with_replacement(plot_data, 2)):
    plt.subplot(3, 3, plot_idx[i])
    if pdata[0]['name'] == pdata[1]['name']:
        # diagonal, histogram
        sns.distplot(pdata[0]['samples'], kde=False, color='w', hist_kws={"edgecolor" : '#333333', 'lw':1, 'alpha':1}, b
    else:
        # pairs
        plt.scatter(pdata[0]['samples'], pdata[1]['samples'], s=15, edgecolor='#333333', linewidth=1, c=subset, alpha=1,
    if plot_idx[i] < 7:
        plt.xticks([],[])
    else:
        plt.xlabel(pdata[0]['label'])
    if plot_idx[i] not in [1,4,7]:
        plt.yticks([],[])
    else:
        plt.ylabel(pdata[1]['label'])
```



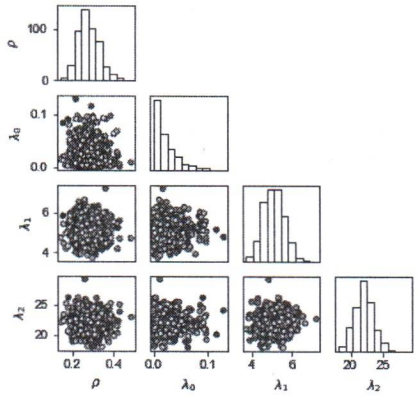
Estimation with the improved model

And indeed we find the same results! This just confirms that the model behaves as expected. Now for `T=3` ...

In [32]:
```
# Sample
fit = model.sampling(data={"n": X.shape[0],
                           "X": X,
                           'T': 3,
                           "rates_std_prior": [100, 100, 100]})
```

```
WARNING:pystan:n_eff / iter below 0.001 indicates that the effective sample size has likely been overestimated
WARNING:pystan:Rhat above 1.1 or below 0.9 indicates that the chains very likely have not mixed
```

In [33]:
```
pp = pd.DataFrame(np.array([fit['rho'][:,1], fit['rates'][:,0], fit['rates'][:,1], fit['rates'][:,2]]).T, columns=[r'$\r
plt.figure(figsize=(5, 5))
plot_idx = {0:1, 1:5, 2:9, 3:13, 4:6, 5:10, 6:14, 7:11, 8:15, 9:16}
plot_data = [{'name': 'rho', 'label': r'$\rho$', 'samples': fit['rho'][subset,1]},
             {'name': 'l0', 'label': r'$\lambda_0$', 'samples': fit['rates'][subset,0]},
             {'name': 'l1', 'label': r'$\lambda_1$', 'samples': fit['rates'][subset,1]},
             {'name': 'l2', 'label': r'$\lambda_2$', 'samples': fit['rates'][subset,2]}]
for i, pdata in enumerate(it.combinations_with_replacement(plot_data, 2)):
    plt.subplot(4, 4, plot_idx[i])
    if pdata[0]['name'] == pdata[1]['name']:
        # diagonal, histogram
        sns.distplot(pdata[0]['samples'], kde=False, color='w', hist_kws={"edgecolor" : '#333333', 'lw':1, 'alpha':1}, b
    else:
        # pairs
        plt.scatter(pdata[0]['samples'], pdata[1]['samples'], s=15, edgecolor='#333333', linewidth=1, c=subset, alpha=1,
    if plot_idx[i] < 13:
        plt.xticks([],[])
    else:
        plt.xlabel(pdata[0]['label'])
    if plot_idx[i] not in [1,5,9, 13]:
        plt.yticks([],[])
    else:
        plt.ylabel(pdata[1]['label'])
```



We now get a fit where `lambda_0` is still small, but where `lambda_1` is now much smaller than before (about 5, versus ~15). The new parameter `lamdba_2` has a high mean: On average, a strong tie translates to ~22 interactions.

What happened is that dolphin pairs we separate the class of connected dolphins in two: One with strong ties and one with weak ties. As a result, the mean number of interaction shifted from an in-between (15), to two extreme (5 and 22).

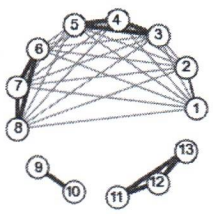
We can visualize the average predicted network with a graph, by (i) placing a **gray edge** between pairs of nodes that have a large posterior probability of being connected by a **weak tie**, and (ii) placing a **blue edge** between pairs of nodes that have a large posterior probability of being connected by a **strong tie**.

In [34]:
```
Q0 = np.mean(fit['Q'][:,:,:,0], axis=0)
Q1 = np.mean(fit['Q'][:,:,:,1], axis=0)
Q2 = np.ones_like(Q0) - Q1 - Q0
G1 = nx.Graph(Q1)
G2 = nx.Graph(Q2)
pos = nx.circular_layout(G1)
for u,v in it.combinations(G1.nodes(), 2):
    if Q1[u, v] < 0.1:
        G1.remove_edge(u, v)
for u,v in it.combinations(G1.nodes(), 2):
    if Q2[u, v] < 0.1:
        G2.remove_edge(u, v)


plt.figure(figsize=(3,3))
nx.draw_networkx_nodes(G1, pos=pos, node_color='w', edgecolors='#333333')
nx.draw_networkx_labels(G1, pos, labels={i : str(i + 1) for i in range(13)}, font_color='k');

w1 = nx.get_edge_attributes(G1, 'weight')
w2 = nx.get_edge_attributes(G2, 'weight')
nx.draw_networkx_edges(G1, pos=pos, width=[1.2*w1[e] for e in G1.edges()], edge_color="#AAAAAA")
nx.draw_networkx_edges(G2, pos=pos, width=[3*w2[e] for e in G2.edges()], edge_color='blue')
sns.despine(ax=plt.gca(), left=True, bottom=True)


plt.xlim(-1.17, 1.17)
plt.ylim(-1.17, 1.17);
```



The width of the edges is proportional to our certainty about their type. We see that edge 1--2 is classified as a strong tie, but with a small certainty:

In [35]:
```
np.mean(fit['Q'][:,0,1,2])  # we used a 1-indexing the figure and 0-indexing in the code.
```

Out[35]:
```
0.49156475756466983
```

The other edges are strongly classified, e.g., 9--10:

In [36]:
```
np.mean(fit['Q'][:,8,9,2])  # we used a 1-indexing the figure and 0-indexing in the code.
```

Out[36]:
```
0.9999999895007179
```

Likewise absent edges are classified with certainty, e.g., 1--13:

In [37]:
```
np.mean(fit['Q'][:,0,12,0])  # we used a 1-indexing the figure and 0-indexing in the code.
```

Out[37]:
```
0.9965734979268289
```

Posterior-predictive check of the improved model

Finally let's revisit the posterior predictive test, with the improved model:

In [38]:
```
X_tilde = np.zeros((13, 13))

for param_id in range(100):  # loop over the first 100 network samples
    for net_id in range(5):  # simulate 5 network samples for each parameter samples
        G = nx.Graph()
        Q = fit['Q'][param_id,:]
```

```
        for i,j in it.combinations(range(13), 2):
            rnd = np.random.rand()
            if rnd < Q[i, j,0]:
                X_tilde[i,j] += float(np.random.poisson(fit['rates'][param_id, 0]))
            elif Q[i, j, 0]< rnd and rnd < Q[i, j,1]:
                X_tilde[i,j] += float(np.random.poisson(fit['rates'][param_id, 1]))
            else:
                X_tilde[i,j] += float(np.random.poisson(fit['rates'][param_id, 2]))

# Normalize and symmetrize
X_tilde /= 500
X_tilde += X_tilde.T
```

First, visually

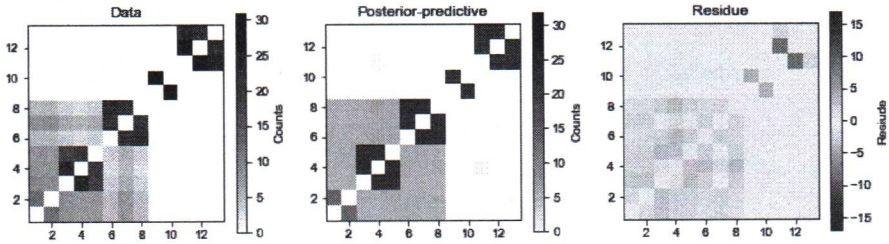
In [39]:
```
plt.figure(figsize=(10,2.8))

plt.subplot(131)
plt.title('Data')
plt.imshow(X, cmap=plt.cm.gray_r)
plt.colorbar(label='Counts')
set_axes()

plt.subplot(132)
plt.title('Posterior-predictive')
plt.imshow(X_tilde, vmin=0, vmax=32, cmap=plt.cm.gray_r)
plt.colorbar(label='Counts')
set_axes()


plt.subplot(133)
plt.title('Residue')
plt.imshow(X  - X_tilde, vmin=-17, vmax=17, cmap=plt.cm.RdBu)
plt.colorbar(label='Resiude')
set_axes()


plt.tight_layout(pad=1)
```

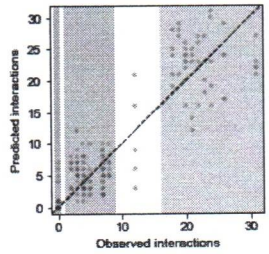


The residues are much smaller than before, and the posterior predictive matrix now resembles the data, minus random fluctuations.

In [40]:
```
plt.figure(figsize=(3,3))
for sample_id in range(5):
    # create artificial data sets
    Q0 = fit['Q'][sample_id,:,:,0]
    Q1 = fit['Q'][sample_id,:,:,1]
    U = np.random.rand(X.shape[0], X.shape[1])
    A = (U < Q1) * 1 + (U > (Q0 + Q1)) * 2
    A[np.triu_indices(13, k=1)] = A.T[np.triu_indices(13, k=1)]
    X_tilde_0 = np.random.poisson(fit['rates'][sample_id, 0], size=Q0.shape)
    X_tilde_1 = np.random.poisson(fit['rates'][sample_id, 1], size=Q1.shape)
    X_tilde_2 = np.random.poisson(fit['rates'][sample_id, 2], size=Q2.shape)
    X_tilde = (A == 0) * X_tilde_0  + (A == 1) * X_tilde_1 + (A == 2) * X_tilde_2
    plt.scatter(X[np.triu_indices(13, k=1)], X_tilde[np.triu_indices(13,k=1)], c='blue', s=12, alpha=0.2, edgecolor='nor

plt.plot([-10, 50], [-10, 50], c='k', ls='--', lw=1)
plt.xlabel('Observed interactions')
plt.ylabel('Predicted interactions')
plt.xlim(-1, 32)
plt.ylim(-1, 32)
plt.gca().set_facecolor("none")
plt.gca().set_facecolor("none")
plt.axvspan(-0.5, 0.5, -1, 32, zorder=-10, alpha=0.1, color='#333333', lw=0)
plt.axvspan(1, 9, -1, 32, zorder=-10, alpha=0.075, color='#333333', lw=0)
plt.axvspan(16, 32, -1, 32, zorder=-10, alpha=0.05, color='#333333', lw=0)
```

Out[40]:
```
<matplotlib.patches.Polygon at 0x7f14882a29d0>
```



We also get much closer agreement when using the scatter plot point of view.

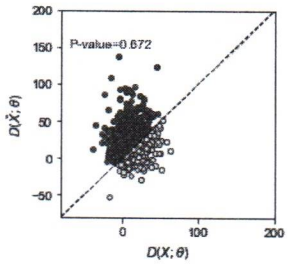
In [42]:
```
def discrepancy(X, Q0, Q1, rates):
    # One can check that <X_tilde_ij> = Q_ij(0) lambda_0 + (Q_ij(1)) lambda_1 + (Q_ij(2)) lambda_2  in our model.
    E_mat = Q0 * rates[0] + Q1 * rates[1] + (1 - Q0 - Q1) * rates[2]
    # Avoid double counting and numerically ill-defined 0*log(0) by selecting indices
    idx = np.triu_indices(13, k=1)
    idx2 = np.where(X[idx] > 0)[0]
```

```
    return np.sum(X[idx][idx2] * np.log(X[idx][idx2] / E_mat[idx][idx2]))


# Compute discrepancy distribution
d_data = np.zeros(500)        # data discrepancy
d_artificial = np.zeros(500)  # artificial data discrepancy

subset = sorted(np.random.choice(range(4000), size=500, replace=False))
for i, sample_id in enumerate(subset):
    # Generate artificial data set
    Q0 = fit['Q'][sample_id,:,:,0]
    Q1 = fit['Q'][sample_id,:,:,1]
    U = np.random.rand(X.shape[0], X.shape[1])
    A = (U < Q1) * 1 + (U > (Q0 + Q1)) * 2
    A[np.triu_indices(13, k=1)] = A.T[np.triu_indices(13, k=1)]
    X_tilde_0 = np.random.poisson(fit['rates'][sample_id, 0], size=Q0.shape)
    X_tilde_1 = np.random.poisson(fit['rates'][sample_id, 1], size=Q1.shape)
    X_tilde_2 = np.random.poisson(fit['rates'][sample_id, 2], size=Q2.shape)
    X_tilde = (A == 0) * X_tilde_0  + (A == 1) * X_tilde_1 + (A == 2) * X_tilde_2
    X_tilde[np.triu_indices(13, k=1)] = X_tilde.T[np.triu_indices(13, k=1)]
    # calculate discrepancy
    d_data[i] =  discrepancy(X, Q0, Q1, fit['rates'][sample_id,:])
    d_artificial[i] = discrepancy(X_tilde, Q0, Q1, fit['rates'][sample_id,:])
```

In [43]:
```
plt.figure(figsize=(3,3))
plt.scatter(d_data[d_data<d_artificial], d_artificial[d_data<d_artificial], s=15, edgecolor='#333333', linewidth=1, colo
plt.scatter(d_data[d_data>d_artificial], d_artificial[d_data>d_artificial], s=15, edgecolor='#333333', linewidth=1, colo
plt.plot([-80, 250], [-80, 250], c='k', ls='--', lw=1)
plt.xlim(-80,200)
plt.ylim(-80,200)
plt.xlabel(r'$D(X;\theta)$')
plt.ylabel(r'$D(\tilde{X};\theta)$')
plt.text(-70, 150, "P-value=" + str(len(d_artificial[d_data<d_artificial]) / 500));
```



The p-value shows that the data is also now typical among posterior-predictive samples. Hence our fit is good! We can trust the improved model.

Other models in closing

Rapid sampling model

The data we analyzed is small so sampling isn't costly. However, as can been seen from the `model` blocks, samples cost O(n^2) to generate, which can be costly for large `n`.

In [44]:
```
print(model)
```

```
StanModel object 'multitype_poisson_data_ER_prior_6bced87e4f9c86d4c3c8d7ccdcddda6a' coded as follows:
data {
  int<lower=1> n;
  int<lower=0> X[n, n];
  int T; // number of edge types
  real<lower=0> rates_std_prior[T];
}
parameters {
  positive_ordered[T] rates;
  simplex[T] rho;
}
model {
  for (k in 1:T)
  {
    rates[k] ~ normal(1, rates_std_prior[k]);
  }

  for (i in 1:n) {
    for (j in i + 1:n) {
      vector[T] z_ij;
      vector[T] z_max_vector;
      real z_max;
      for (k in 1:T) {
        real log_mu_ij_k = poisson_lpmf(X[i, j] | rates[k]);
        real log_nu_ij_k = log(rho[k]);

        z_ij[k] = log_mu_ij_k + log_nu_ij_k;
      }
      z_max = max(z_ij);
      z_max_vector = rep_vector(z_max, T);
      target += z_max + log_sum_exp(z_ij - z_max_vector);
    }
  }
}
generated quantities {
  real Q[n ,n, T];
  for (i in 1:n) {
    for (k in 1:T) {
      Q[i, i, k] = 0;
    }
    for (j in i+1:n) {
      vector[T] z_ij;
      real accu;
```

```
    for (k in 1:T)
    {
      real log_mu_ij_k = poisson_lpmf(X[i, j] | rates[k]);
      real log_nu_ij_k = log(rho[k]);

      z_ij[k] =  log_mu_ij_k + log_nu_ij_k;
    }
    for (k in 1:T)
    {
      accu = 0;
      for (k_prime in 1:T)
      {
        accu += exp(z_ij[k_prime] - z_ij[k]);
      }
      Q[i, j, k] = 1 / accu;
      Q[j, i, k] = Q[i, j, k];
    }
   }
  }
}
```

We can actually save quite a bit of computation, since in the case of our particular model, the distribution of `X[i,j]` depends only on the edge type and the value of `X[i,j]`, not the indexes. (See Section 6 of the paper's appendix). The following model implements these savings:

In [45]:
```
# Uncomment and run if model was already compiled
with open('../examples/fast_poisson.bin', 'rb') as f:
    model = pickle.load(f)
```

In [46]:
```
## Uncomment and run to compile model inline
# model = pystan.StanModel('../examples/fast_poissonstan')
```

In [47]:
```
print(model)
```

```
StanModel object 'fast_poisson_54ac6d69d5111cbde47da05cec9258a1' coded as follows:
data {
  int n;    // number of non-empty pairs class
  int X[n]; // number of pairs of nodes in each classes
  int Y[n]; // number of observations for each class
  // Priors
  real<lower=0> rates_std_prior[2];
  real<lower=0> rho_prior[2];
}
parameters {
  positive_ordered[2] rates;
  real<lower=0, upper=1> rho;
}
model {
  rates[1] ~ normal(1, rates_std_prior[1]);
  rates[2] ~ normal(1, rates_std_prior[2]);
  rho ~ beta(rho_prior[1], rho_prior[2]);

  for (i in 1:n)
  {

    real log_mu_i_0 = poisson_lpmf(Y[i] | rates[1]);
    real log_mu_i_1 = poisson_lpmf(Y[i] | rates[2]);
    real log_nu_i_0 = bernoulli_lpmf(0 | rho);
    real log_nu_i_1 = bernoulli_lpmf(1 | rho);

    real z = 0;
    real z_i_0 = log_mu_i_0 + log_nu_i_0;
    real z_i_1 = log_mu_i_1 + log_nu_i_1;
    if (z_i_0 > z_i_1) {z += z_i_0 + log1p_exp(z_i_1 - z_i_0);}
    else {z += z_i_1 + log1p_exp(z_i_0 - z_i_1);}
    target += X[i] * z;
  }
}
generated quantities {
  real Q[n];
  for (i in 1:n) {
    real log_mu_i_0 = poisson_lpmf(Y[i] | rates[1]);
    real log_mu_i_1 = poisson_lpmf(Y[i] | rates[2]);
    real log_nu_i_0 = bernoulli_lpmf(0 | rho);
    real log_nu_i_1 = bernoulli_lpmf(1 | rho);
    real z_i_0 = log_mu_i_0 + log_nu_i_0;
    real z_i_1 = log_mu_i_1 + log_nu_i_1;
    Q[i] = 1 / (1 + exp(z_i_1 - z_i_0));
  }
}
```

This model takes the *histogram* of `X` as input and genreates prediction in O(max(X)) time.

In [48]:
```
fit = model.sampling(data={'n': len(obs_in_class),
                           'X': num_in_class.astype(int),
                           'Y': obs_in_class,
                           'rates_std_prior': [100, 100],
                           'rho_prior': [1,1]})
```

In [49]:
```
fit
```

Out[49]:
```
Inference for Stan model: fast_poisson_54ac6d69d5111cbde47da05cec9258a1.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
rates[1]	0.62	8.0e-3	0.21	0.09	0.51	0.64	0.76	0.97	723	1.01
rates[2]	14.3	0.05	1.46	11.0	13.46	14.44	15.32	16.81	786	1.01
rho	0.28	2.1e-3	0.06	0.17	0.24	0.28	0.32	0.42	906	1.01
Q[1]	1.0	1.8e-7	4.1e-6	1.0	1.0	1.0	1.0	1.0	526	1.01
Q[2]	1.0	2.0e-4	5.3e-3	1.0	1.0	1.0	1.0	1.0	682	1.0
Q[3]	0.98	5.1e-3	0.1	0.87	1.0	1.0	1.0	1.0	378	1.01

```
Q[4]        0.93     0.01    0.21    0.05    0.98     1.0     1.0     1.0    409   1.01
Q[5]        0.77     0.01    0.31  3.9e-4    0.72    0.92    0.97    0.99    625   1.01
Q[6]        0.37    7.9e-3    0.29  3.2e-6    0.09    0.35    0.62     0.9   1339    1.0
Q[7]        0.06    1.9e-3    0.09  2.4e-8  3.7e-3    0.02    0.08    0.33   2262    1.0
Q[8]      4.3e-3    2.0e-4    0.01 1.8e-10  1.4e-4  1.1e-3  4.1e-3    0.03   2769    1.0
Q[9]      2.5e-4    1.3e-5  7.2e-4  1.4e-12  5.3e-6  4.9e-5  2.1e-4  1.7e-3   3030    1.0
Q[10]     1.4e-5    8.6e-7  4.8e-5  1.1e-14  2.0e-7  2.2e-6  1.0e-5  9.9e-5   3108    1.0
Q[11]     8.4e-7    5.9e-8  3.3e-6  9.2e-17  7.5e-9  9.5e-8  5.2e-7  6.1e-6   3118    1.0
Q[12]     5.1e-8    4.2e-9  2.3e-7  7.5e-19  2.8e-10  4.2e-9  2.6e-8  3.7e-7   3062    1.0
Q[13]     3.2e-9    3.2e-10  1.7e-8  5.5e-21  1.0e-11  1.9e-10  1.3e-9  2.2e-8   2960    1.0
Q[14]    2.0e-10    2.5e-11  1.3e-9  4.0e-23  3.8e-13  8.2e-12  6.5e-11  1.4e-9   2842    1.0
Q[15]    1.3e-11    2.1e-12  1.1e-10  2.9e-25  1.4e-14  3.7e-13  3.4e-12  8.9e-11   2743    1.0
Q[16]    9.0e-13    1.8e-13  9.1e-12  2.3e-27  4.9e-16  1.6e-14  1.7e-13  5.6e-12   2680    1.0
Q[17]    6.2e-14    1.5e-14  7.9e-13  1.8e-29  1.8e-17  6.9e-16  8.5e-15  3.7e-13   2656    1.0
Q[18]    4.5e-15    1.4e-15  7.0e-14  1.4e-31  7.0e-19  3.0e-17  4.2e-16  2.3e-14   2665    1.0
Q[19]    3.3e-16    1.2e-16  6.3e-15  1.1e-33  2.6e-20  1.3e-18  2.1e-17  1.5e-15   2697    1.0
Q[20]    2.5e-17    1.1e-17  5.7e-16  8.8e-36  9.8e-22  5.8e-20  1.1e-18  1.0e-16   2745    1.0
Q[21]    2.0e-18   10.0e-19  5.3e-17  6.9e-38  3.6e-23  2.5e-21  5.7e-20  6.5e-18   2803    1.0
Q[22]    1.6e-19    9.1e-20  4.9e-18  5.5e-40  1.3e-24  1.1e-22  3.0e-21  4.2e-19   2865    1.0
Q[23]    1.3e-20    8.4e-21  4.5e-19  4.4e-42  4.6e-26  4.9e-24  1.5e-22  2.7e-20   2929    1.0
Q[24]    1.1e-21    7.7e-22  4.2e-20  3.5e-44  1.7e-27  2.1e-25  7.9e-24  1.7e-21   2993    1.0
Q[25]    9.5e-23    7.1e-23  3.9e-21  2.8e-46  6.0e-29  9.4e-27  4.0e-25  1.1e-22   3056    1.0
Q[26]    8.2e-24    6.5e-24  3.6e-22  2.2e-48  2.1e-30  4.2e-28  2.0e-26  7.3e-24   3116    1.0
Q[27]    7.3e-25    6.0e-25  3.4e-23  1.8e-50  8.0e-32  1.9e-29  1.0e-27  4.7e-25   3174    1.0
Q[28]    6.5e-26    5.6e-26  3.2e-24  1.4e-52  2.9e-33  8.0e-31  5.3e-29  3.0e-26   3228    1.0
Q[29]    5.8e-27    5.2e-27  3.0e-25  1.1e-54  1.1e-34  3.6e-32  2.7e-30  2.0e-27   3278    1.0
Q[30]    5.3e-28    4.8e-28  2.8e-26  9.2e-57  3.8e-36  1.6e-33  1.4e-31  1.3e-28   3325    1.0
Q[31]    4.8e-29    4.5e-29  2.6e-27  7.3e-59  1.3e-37  7.0e-35  7.0e-33  8.4e-30   3370    1.0
Q[32]    4.4e-30    4.1e-30  2.4e-28  5.9e-61  4.9e-39  3.1e-36  3.6e-34  5.3e-31   3412    1.0
lp__      -215.8     0.06    1.51  -219.6  -216.6  -215.4  -214.7  -214.1    635   1.01

Samples were drawn using NUTS at Mon Oct 26 15:09:54 2020.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

The sample come out must faster than before, and lead to the same inferences. Hence this model can scale.

(This is because the model maps directly to a "vanilla" finite-mixture, where only the value of an entry of the array matters, not its indexes.)

Richer network prior

We used a simple prior for the network: either and edge exists or it doesn't. We can introduce more degree variability by adding a parameter for the degree of each nodes.

In [50]:
```
# Uncomment and run if model was already compiled
with open('../examples/poisson_data_SCM_prior.bin', 'rb') as f:
    model = pickle.load(f)
```

In [51]:
```
## Uncomment and run to compile model inline
# model = pystan.StanModel('../examples/poisson_data_SCM_prior.stan')
```

Unfortunately, the model doesn't admit a condensed form like the fast implementation above, so we have to use O(n^2) loops:

In [52]:
```
print(model)
```

```
StanModel object 'poisson_data_SCM_prior_c1bfb5a3165dba32c9fb93758e5cdd71' coded as follows:
data {
  int<lower=1> n;
  int<lower=0> X[n, n];
  real<lower=0> rates_std_prior[2];
}
parameters {
  positive_ordered[2] rates;
  simplex[n] lambda;
  real<lower = 0> scale;
}
model {
  rates[1] ~ normal(1, rates_std_prior[1]);
  rates[2] ~ normal(1, rates_std_prior[2]);
  scale ~ exponential(100);

  for (i in 1:n) {
    for (j in i + 1:n) {
      real log_mu_ij_0 = poisson_lpmf(X[i, j] | rates[1]);
      real log_mu_ij_1 = poisson_lpmf(X[i, j] | rates[2]);

      real r = inv_logit(scale * lambda[i] * lambda[j]);
      real log_nu_ij_0 = bernoulli_lpmf(0 | r);
      real log_nu_ij_1 = bernoulli_lpmf(1 | r);

      real z_ij_0 = log_mu_ij_0 + log_nu_ij_0;
      real z_ij_1 = log_mu_ij_1 + log_nu_ij_1;
      if (z_ij_0 > z_ij_1) {target += z_ij_0 + log1p_exp(z_ij_1 - z_ij_0);}
      else {target += z_ij_1 + log1p_exp(z_ij_0 - z_ij_1);}
    }
  }
}
generated quantities {
  real Q[n ,n];
  for (i in 1:n) {
    Q[i, i] = 0;
    for (j in i+1:n) {
      real log_mu_ij_0 = poisson_lpmf(X[i, j] | rates[1]);
      real log_mu_ij_1 = poisson_lpmf(X[i, j] | rates[2]);

      real r = inv_logit(scale * lambda[i] * lambda[j]);
      real log_nu_ij_0 = bernoulli_lpmf(0 | r);
      real log_nu_ij_1 = bernoulli_lpmf(1 | r);

      real z_ij_0 = log_mu_ij_0 + log_nu_ij_0;
      real z_ij_1 = log_mu_ij_1 + log_nu_ij_1;
      Q[i, j] = 1 / (1  + exp(z_ij_0 - z_ij_1));
      Q[j, i] = Q[i, j];
    }
  }
```

}

Let's fit this model to the dolphin data...

In [53]:
```
# Sample
fit = model.sampling(data={"n": X.shape[0],
                           "X": X,
                           "rates_std_prior": [100, 100],
                           'rho_prior': [1,1]},
                     chains=4, iter=500)
```

```
WARNING:pystan:n_eff / iter below 0.001 indicates that the effective sample size has likely been overestimated
WARNING:pystan:Rhat above 1.1 or below 0.9 indicates that the chains very likely have not mixed
```

In [54]:
```
fit
```

```
WARNING:pystan:Truncated summary with the 'fit.__repr__' method. For the full summary use 'print(fit)'
```

Out[54]:
```
Warning: Shown data is truncated to 100 parameters
For the full summary use 'print(fit)'

Inference for Stan model: poisson_data_SCM_prior_c1bfb5a3165dba32c9fb93758e5cdd71.
4 chains, each with iter=500; warmup=250; thin=1;
post-warmup draws per chain=250, total post-warmup draws=1000.
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
rates[1]	0.11	4.1e-3	0.12	9.7e-4	0.03	0.07	0.14	0.47	839	1.0
rates[2]	11.16	0.03	0.81	9.79	10.62	11.07	11.63	12.99	726	1.0
lambda[1]	0.08	2.0e-3	0.07	2.5e-3	0.02	0.05	0.11	0.27	1381	1.0
lambda[2]	0.07	1.8e-3	0.07	1.3e-3	0.02	0.05	0.1	0.25	1355	1.0
lambda[3]	0.08	2.1e-3	0.07	2.1e-3	0.02	0.06	0.11	0.24	1104	1.0
lambda[4]	0.08	2.0e-3	0.07	2.5e-3	0.03	0.06	0.11	0.26	1184	1.0
lambda[5]	0.08	2.3e-3	0.07	2.1e-3	0.02	0.06	0.11	0.28	1045	1.0
lambda[6]	0.08	1.8e-3	0.07	1.3e-3	0.02	0.05	0.11	0.27	1570	1.0
lambda[7]	0.08	2.2e-3	0.07	2.3e-3	0.02	0.06	0.11	0.29	1103	1.0
lambda[8]	0.08	2.3e-3	0.07	1.5e-3	0.02	0.06	0.11	0.28	996	1.0
lambda[9]	0.08	2.0e-3	0.07	2.2e-3	0.02	0.06	0.12	0.26	1296	1.0
lambda[10]	0.08	2.0e-3	0.07	3.5e-3	0.03	0.06	0.11	0.27	1294	1.0
lambda[11]	0.08	2.2e-3	0.07	3.2e-3	0.02	0.05	0.11	0.25	943	1.0
lambda[12]	0.08	2.3e-3	0.07	3.0e-3	0.02	0.06	0.11	0.29	987	1.0
lambda[13]	0.08	2.1e-3	0.07	2.1e-3	0.02	0.05	0.11	0.27	1184	1.0
scale	0.01	2.7e-4	0.01	4.7e-4	3.2e-3	7.0e-3	0.01	0.04	1401	1.0
Q[1,1]	0.0	nan	0.0	0.0	0.0	0.0	0.0	0.0	nan	nan
Q[2,1]	1.0	6.3e-13	2.0e-11	1.0	1.0	1.0	1.0	1.0	992	1.0
Q[3,1]	1.0	6.6e-8	2.0e-6	1.0	1.0	1.0	1.0	1.0	929	1.0
Q[4,1]	1.0	1.2e-6	3.7e-5	1.0	1.0	1.0	1.0	1.0	888	1.0
Q[5,1]	1.0	1.2e-6	3.7e-5	1.0	1.0	1.0	1.0	1.0	888	1.0
Q[6,1]	0.98	3.3e-3	0.08	0.77	1.0	1.0	1.0	1.0	663	1.0
Q[7,1]	1.0	1.2e-6	3.7e-5	1.0	1.0	1.0	1.0	1.0	888	1.0
Q[8,1]	0.98	3.3e-3	0.08	0.77	1.0	1.0	1.0	1.0	663	1.0
Q[9,1]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[10,1]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[11,1]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[12,1]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[13,1]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[1,2]	1.0	6.3e-13	2.0e-11	1.0	1.0	1.0	1.0	1.0	992	1.0
Q[2,2]	0.0	nan	0.0	0.0	0.0	0.0	0.0	0.0	nan	nan
Q[3,2]	1.0	6.6e-8	2.0e-6	1.0	1.0	1.0	1.0	1.0	929	1.0
Q[4,2]	1.0	1.2e-6	3.7e-5	1.0	1.0	1.0	1.0	1.0	888	1.0
Q[5,2]	1.0	1.2e-6	3.7e-5	1.0	1.0	1.0	1.0	1.0	888	1.0
Q[6,2]	0.98	3.3e-3	0.08	0.77	1.0	1.0	1.0	1.0	663	1.0
Q[7,2]	1.0	1.2e-6	3.7e-5	1.0	1.0	1.0	1.0	1.0	888	1.0
Q[8,2]	0.98	3.3e-3	0.08	0.77	1.0	1.0	1.0	1.0	663	1.0
Q[9,2]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[10,2]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[11,2]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[12,2]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[13,2]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[1,3]	1.0	6.6e-8	2.0e-6	1.0	1.0	1.0	1.0	1.0	929	1.0
Q[2,3]	1.0	6.6e-8	2.0e-6	1.0	1.0	1.0	1.0	1.0	929	1.0
Q[3,3]	0.0	nan	0.0	0.0	0.0	0.0	0.0	0.0	nan	nan
Q[4,3]	1.0	nan	0.0	1.0	1.0	1.0	1.0	1.0	nan	nan
Q[5,3]	1.0	nan	0.0	1.0	1.0	1.0	1.0	1.0	nan	nan
Q[6,3]	0.87	8.8e-3	0.24	0.1	0.88	0.99	1.0	1.0	754	1.0
Q[7,3]	1.0	4.2e-4	0.01	0.99	1.0	1.0	1.0	1.0	767	1.0
Q[8,3]	0.87	8.8e-3	0.24	0.1	0.88	0.99	1.0	1.0	754	1.0
Q[9,3]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[10,3]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[11,3]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[12,3]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[13,3]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[1,4]	1.0	1.2e-6	3.7e-5	1.0	1.0	1.0	1.0	1.0	888	1.0
Q[2,4]	1.0	1.2e-6	3.7e-5	1.0	1.0	1.0	1.0	1.0	888	1.0
Q[3,4]	1.0	nan	0.0	1.0	1.0	1.0	1.0	1.0	nan	nan
Q[4,4]	0.0	nan	0.0	0.0	0.0	0.0	0.0	0.0	nan	nan
Q[5,4]	1.0	nan	0.0	1.0	1.0	1.0	1.0	1.0	nan	nan
Q[6,4]	0.42	0.02	0.36	3.7e-3	0.08	0.29	0.81	1.0	523	1.01
Q[7,4]	0.98	3.3e-3	0.08	0.77	1.0	1.0	1.0	1.0	663	1.0
Q[8,4]	0.42	0.02	0.36	3.7e-3	0.08	0.29	0.81	1.0	523	1.01
Q[9,4]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[10,4]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[11,4]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[12,4]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[13,4]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[1,5]	1.0	1.2e-6	3.7e-5	1.0	1.0	1.0	1.0	1.0	888	1.0
Q[2,5]	1.0	1.2e-6	3.7e-5	1.0	1.0	1.0	1.0	1.0	888	1.0
Q[3,5]	1.0	nan	0.0	1.0	1.0	1.0	1.0	1.0	nan	nan
Q[4,5]	1.0	nan	0.0	1.0	1.0	1.0	1.0	1.0	nan	nan
Q[5,5]	0.0	nan	0.0	0.0	0.0	0.0	0.0	0.0	nan	nan
Q[6,5]	0.87	8.8e-3	0.24	0.1	0.88	0.99	1.0	1.0	754	1.0
Q[7,5]	1.0	4.2e-4	0.01	0.99	1.0	1.0	1.0	1.0	767	1.0
Q[8,5]	0.87	8.8e-3	0.24	0.1	0.88	0.99	1.0	1.0	754	1.0
Q[9,5]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[10,5]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[11,5]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[12,5]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[13,5]	2.0e-5	4.6e-7	1.5e-5	2.8e-6	1.0e-5	1.7e-5	2.6e-5	5.7e-5	1056	1.0
Q[1,6]	0.98	3.3e-3	0.08	0.77	1.0	1.0	1.0	1.0	663	1.0
Q[2,6]	0.98	3.3e-3	0.08	0.77	1.0	1.0	1.0	1.0	663	1.0

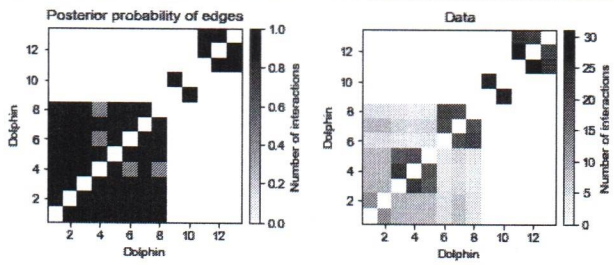
```
Q[3,6]       0.87  8.8e-3    0.24    0.1   0.88   0.99    1.0    1.0    754    1.0
Q[4,6]       0.42    0.02    0.36 3.7e-3   0.08   0.29   0.81    1.0    523   1.01
Q[5,6]       0.87  8.8e-3    0.24    0.1   0.88   0.99    1.0    1.0    754    1.0
Q[6,6]        0.0     nan     0.0    0.0    0.0    0.0    0.0    0.0    nan    nan
Q[7,6]        1.0     nan     0.0    1.0    1.0    1.0    1.0    1.0    nan    nan
Q[8,6]        1.0     nan     0.0    1.0    1.0    1.0    1.0    1.0    nan    nan
Q[9,6]     2.0e-5  4.6e-7  1.5e-5 2.8e-6 1.0e-5 1.7e-5 2.6e-5 5.7e-5   1056    1.0
Q[10,6]    2.0e-5  4.6e-7  1.5e-5 2.8e-6 1.0e-5 1.7e-5 2.6e-5 5.7e-5   1056    1.0
Q[11,6]    2.0e-5  4.6e-7  1.5e-5 2.8e-6 1.0e-5 1.7e-5 2.6e-5 5.7e-5   1056    1.0
Q[12,6]    2.0e-5  4.6e-7  1.5e-5 2.8e-6 1.0e-5 1.7e-5 2.6e-5 5.7e-5   1056    1.0
Q[13,6]    2.0e-5  4.6e-7  1.5e-5 2.8e-6 1.0e-5 1.7e-5 2.6e-5 5.7e-5   1056    1.0
Q[1,7]        1.0  1.2e-6  3.7e-5    1.0    1.0    1.0    1.0    1.0    888    1.0
Q[2,7]        1.0  1.2e-6  3.7e-5    1.0    1.0    1.0    1.0    1.0    888    1.0
Q[3,7]        1.0  4.2e-4    0.01   0.99    1.0    1.0    1.0    1.0    767    1.0
Q[4,7]       0.98  3.3e-3    0.08   0.77    1.0    1.0    1.0    1.0    663    1.0
Q[5,7]        1.0  4.2e-4    0.01   0.99    1.0    1.0    1.0    1.0    767    1.0
lp__       -265.7    0.18    3.11 -272.9 -267.7 -265.3 -263.5 -260.8    299    1.0

Samples were drawn using NUTS at Mon Oct 26 15:10:07 2020.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

In [55]:
```
Q = np.mean(fit['Q'], axis=0)
```

In [56]:
```
plt.figure(figsize=(7,3))
plt.subplot(121)
plt.title('Posterior probability of edges')
set_axes()
im=plt.imshow(Q, cmap=plt.cm.Greys)
set_axes()
plt.colorbar(im,fraction=0.046, pad=0.04, label='Number of interactions')
plt.xlabel('Dolphin');
plt.ylabel('Dolphin');


plt.subplot(122)
plt.title('Data')
im=plt.imshow(X, cmap=plt.cm.Greys)
set_axes()
plt.colorbar(im,fraction=0.046, pad=0.04, label='Number of interactions')
plt.xlabel('Dolphin');
plt.ylabel('Dolphin');
plt.tight_layout()
```



Comparing to previous results, we see that this model doesn't diffent much from the previous one. It predicts a slightly smaller average `lambda_1` and a slightly high quantity of edges.

In conclusion

Thanks for reading this far! We just reproduced every figure of the first case study of "Robust Bayesian inference of network structure from unreliable data,", by J.-G. Young, G. T. Cantwell, and M.E.J. Newman.

Get in touch at jean-gabriel.young@uvm.edu, or via the issues if you need further guindance.