GLM: Model Selection¶

In [1]:
```
%matplotlib inline
import pymc3 as pm
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from collections import OrderedDict
from ipywidgets import interactive, fixed

plt.style.use('seaborn-darkgrid')
print('Running on PyMC3 v{}'.format(pm.__version__))
rndst = np.random.RandomState(0)
```

```
Running on PyMC3 v3.6
```

A fairly minimal reproducable example of Model Selection using WAIC, and LOO as currently implemented in PyMC3.

This example creates two toy datasets under linear and quadratic models, and then tests the fit of a range of polynomial linear models upon those datasets by using Widely Applicable Information Criterion (WAIC), and leave-one-out (LOO) cross-validation using Pareto-smoothed importance sampling (PSIS).

The example was inspired by Jake Vanderplas' blogpost on model selection, although Cross-Validation and Bayes Factor comparison are not implemented. The datasets are tiny and generated within this Notebook. They contain errors in the measured value (y) only.

Local Functions¶

In [2]:
```
def generate_data(n=20, p=0, a=1, b=1, c=0, latent_sigma_y=20):
    '''
    Create a toy dataset based on a very simple model that we might
    imagine is a noisy physical process:
        1. random x values within a range
        2. latent error aka inherent noise in y
        3. optionally create labelled outliers with larger noise

    Model form: y ~ a + bx + cx^2 + e

    NOTE: latent_sigma_y is used to create a normally distributed,
    'latent error' aka 'inherent noise' in the 'physical' generating
    process, rather than experimental measurement error.
    Please don't use the returned `latent_error` values in inferential
    models, it's returned in the dataframe for interest only.
    '''

    df = pd.DataFrame({'x': rndst.choice(np.arange(100), n, replace=False)})

    # create linear or quadratic model
    df['y'] = a + b*(df['x']) + c*(df['x'])**2

    # create latent noise and marked outliers
    df['latent_error'] = rndst.normal(0, latent_sigma_y, n)
    df['outlier_error'] = rndst.normal(0, latent_sigma_y*10, n)
    df['outlier'] = rndst.binomial(1, p, n)

    # add noise, with extreme noise for marked outliers
    df['y'] += ((1-df['outlier']) * df['latent_error'])
    df['y'] += (df['outlier'] * df['outlier_error'])

    # round
    for col in ['y', 'latent_error', 'outlier_error', 'x']:
        df[col] = np.round(df[col], 3)

    # add label
    df['source'] = 'linear' if c == 0 else 'quadratic'

    # create simple linspace for plotting true model
    plotx = np.linspace(df['x'].min() - np.ptp(df['x'].values)*.1,
                        df['x'].max() + np.ptp(df['x'].values)*.1, 100)

    ploty = a + b * plotx + c * plotx ** 2
    dfp = pd.DataFrame({'x': plotx, 'y': ploty})

    return df, dfp


def interact_dataset(n=20, p=0, a=-30, b=5, c=0, latent_sigma_y=20):
    '''
    Convenience function:
    Interactively generate dataset and plot
    '''

    df, dfp = generate_data(n, p, a, b, c, latent_sigma_y)

    g = sns.FacetGrid(df, height=8, hue='outlier', hue_order=[True, False],
                      palette=sns.color_palette('Set1'), legend_out=False)

    g.map(plt.errorbar, 'x', 'y', 'latent_error', marker="o",
          ms=10, mec='w', mew=2, ls='', elinewidth=0.7).add_legend()

    plt.plot(dfp['x'], dfp['y'], '--', alpha=0.8)

    plt.subplots_adjust(top=0.92)
    g.fig.suptitle('Sketch of Data Generation ({})'.format(
        df['source'][0]), fontsize=16)


def plot_datasets(df_lin, df_quad, dfp_lin, dfp_quad):
    '''
    Convenience function:
    Plot the two generated datasets in facets with generative model
    '''

    df = pd.concat((df_lin, df_quad), axis=0)

    g = sns.FacetGrid(col='source', hue='source', data=df, height=6,
                      sharey=False, legend_out=False)

    g.map(plt.scatter, 'x', 'y', alpha=0.7, s=100, lw=2, edgecolor='w')
```

Generate toy datasets¶

Interactively draft data¶

Throughout the rest of the Notebook, we'll use two toy datasets created by a linear and a quadratic model respectively, so that we can better evaluate the fit of the model selection.

Right now, lets use an interactive session to play around with the data generation function in this Notebook, and get a feel for the possibilities of data we could generate.

$$y_i = a + bx_i + cx_i^2 + \epsilon_i$$

where:
$i \in n$ datapoints
$\epsilon \sim \mathcal{N}(0, latent\_sigma\_y)$

NOTE on outliers:

- We can use value p to set the (approximate) proportion of 'outliers' under a bernoulli distribution.
- These outliers have a 10x larger `latent_sigma_y`
- These outliers are labelled in the returned datasets and may be useful for other modelling, see another example Notebook `GLM-robust-with-outlier-detection.ipynb`

In [3]:
```
interactive(interact_dataset, n=[5,50,5], p=[0,.5,.05], a=[-50,50],
            b=[-10,10], c=[-3,3], latent_sigma_y=[0,1000,50])
```

Observe:

- I've shown the `latent_error` in errorbars, but this is for interest only, since this shows the *inherent noise* in whatever 'physical process' we imagine created the data.
- There is no *measurement error*.
- Datapoints created as outliers are shown in **red**, again for interest only.

Create datasets for modelling¶

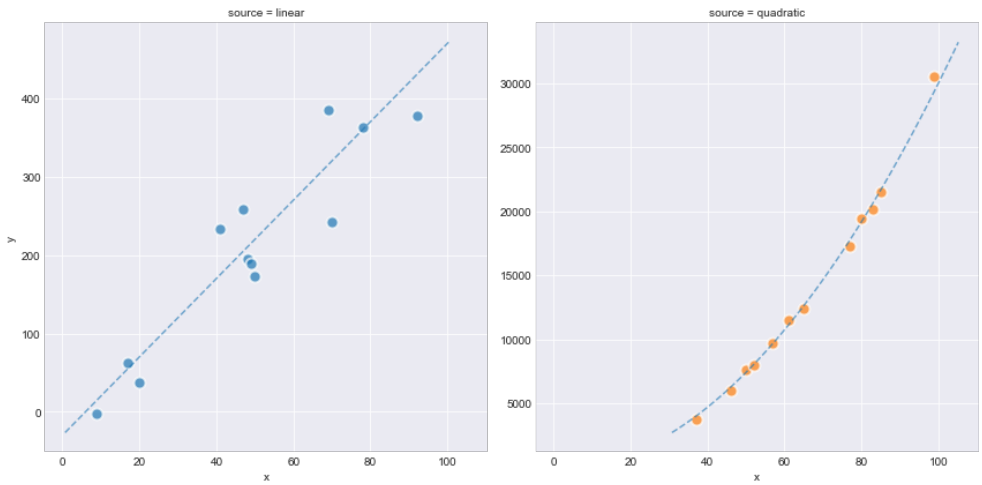
We can use the above interactive plot to get a feel for the effect of the params. Now we'll create 2 fixed datasets to use for the remainder of the Notebook.

1. For a start, we'll create a linear model with small noise. Keep it simple.
2. Secondly, a quadratic model with small noise

In [4]:
```
n = 12
df_lin, dfp_lin = generate_data(n=n, p=0, a=-30, b=5, c=0, latent_sigma_y=40)
df_quad, dfp_quad = generate_data(n=n, p=0, a=-200, b=2, c=3, latent_sigma_y=500)
```

Scatterplot against model line

In [5]:
```
plot_datasets(df_lin, df_quad, dfp_lin, dfp_quad)
```



Observe:

- We now have two datasets `df_lin` and `df_quad` created by a linear model and quadratic model respectively.
- You can see this raw data, the ideal model fit and the effect of the latent noise in the scatterplots above
- In the folowing plots in this Notebook, the linear-generated data will be shown in Blue and the quadratic in Green.

Standardize¶

In [6]:
```
dfs_lin = df_lin.copy()
dfs_lin['x'] = (df_lin['x'] - df_lin['x'].mean()) / df_lin['x'].std()

dfs_quad = df_quad.copy()
dfs_quad['x'] = (df_quad['x'] - df_quad['x'].mean()) / df_quad['x'].std()
```

Create ranges for later ylim xim

In [7]:
```
dfs_lin_xlims = (dfs_lin['x'].min() - np.ptp(dfs_lin['x'].values)/10,
                 dfs_lin['x'].max() + np.ptp(dfs_lin['x'].values)/10)

dfs_lin_ylims = (dfs_lin['y'].min() - np.ptp(dfs_lin['y'].values)/10,
                 dfs_lin['y'].max() + np.ptp(dfs_lin['y'].values)/10)

dfs_quad_ylims = (dfs_quad['y'].min() - np.ptp(dfs_quad['y'].values)/10,
                  dfs_quad['y'].max() + np.ptp(dfs_quad['y'].values)/10)
```

Demonstrate simple linear model¶

This *linear model* is really simple and conventional, an OLS with L2 constraints (Ridge Regression):

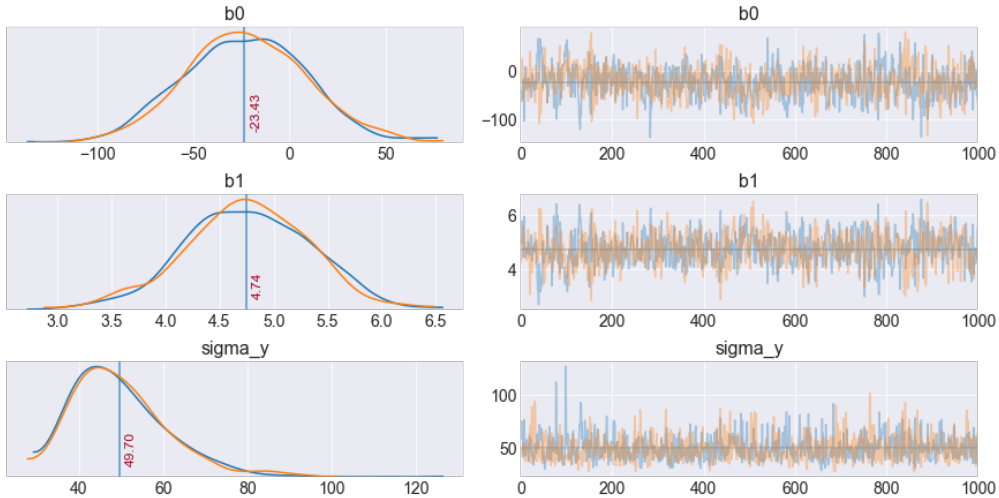
$$y = a + bx + \epsilon$$

Define model using explicit PyMC3 method¶

In [8]:
```
with pm.Model() as mdl_ols:
    ## define Normal priors to give Ridge regression
    b0 = pm.Normal('b0', mu=0, sigma=100)
    b1 = pm.Normal('b1', mu=0, sigma=100)

    ## define Linear model
    yest = b0 + b1 * df_lin['x']

    ## define Normal likelihood with HalfCauchy noise (fat tails, equiv to HalfT 1DoF)
    sigma_y = pm.HalfCauchy('sigma_y', beta=10)
    likelihood = pm.Normal('likelihood', mu=yest, sigma=sigma_y, observed=df_lin['y'])

    traces_ols = pm.sample(2000)
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sigma_y, b1, b0]
Sampling 2 chains: 100%|██████████| 5000/5000 [00:05<00:00, 915.99draws/s]
```

In [9]:
```
plot_traces(traces_ols, retain=1000)
```



Observe:

- This simple OLS manages to make fairly good guesses on the model parameters - the data has been generated fairly simply after all - but it does appear to have been fooled slightly by the inherent noise.

Define model using PyMC3 GLM method¶

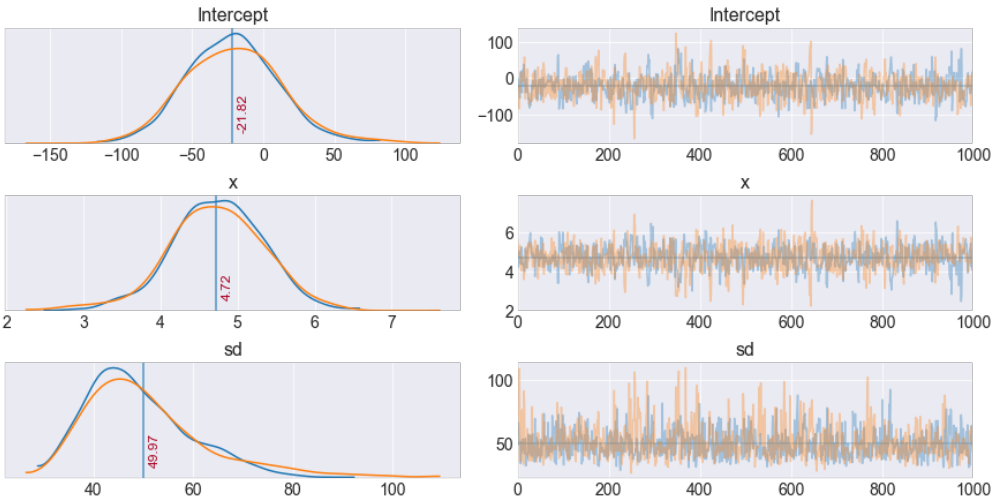
PyMC3 has a module - `glm` - for defining models using a `patsy`-style formula syntax. This seems really useful, especially for defining simple regression models in fewer lines of code.

Here's the same OLS model as above, defined using `glm`.

In [10]
```
with pm.Model() as mdl_ols_glm:
    # Define priors for intercept and regression coefficients.
    priors = {'Intercept': pm.Normal.dist(mu=0., sigma=100.),
              'x': pm.Normal.dist(mu=0., sigma=100.),
              }
    # setup model with Normal likelihood (which uses HalfCauchy for error prior)
    pm.glm.GLM.from_formula('y ~ 1 + x', df_lin, family=pm.glm.families.Normal())

    traces_ols_glm = pm.sample(2000)
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, x, Intercept]
Sampling 2 chains: 100%|██████████| 5000/5000 [00:04<00:00, 1132.82draws/s]
```

In [11]:
```
plot_traces(traces_ols_glm, retain=1000)
```



Observe:

- The output parameters are of course named differently to the custom naming before. Now we have:

  ```
  b0 == Intercept
  b1 == x
  sigma_y == sd
  ```
- However, naming aside, this `glm`-defined model appears to behave in a very similar way, and finds the same parameter values as the conventionally-defined model - any differences are due to the random nature of the sampling.
- We can quite happily use the `glm` syntax for further models below, since it allows us to create a small model factory very easily.

Create higher-order linear models¶

Back to the real purpose of this Notebook, to demonstrate model selection.

First, let's create and run a set of polynomial models on each of our toy datasets. By default this is for models of order 1 to 5.

Create and run polynomial models¶

Please see `run_models()` above for details. Generally, we're creating 5 polynomial models and fitting each to the chosen dataset

In [12]:
```
models_lin, traces_lin = run_models(dfs_lin, 5)
```

```
Running: k1

Auto-assigning NUTS sampler...
Initializing NUTS using advi+adapt_diag...
Average Loss = 95.185:  12%|▉         | 23589/200000 [00:11<01:25, 2069.12it/s]
Convergence achieved at 23700
Interrupted at 23,699 [11%]: Average Loss = 66,411
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, x, Intercept]
Sampling 2 chains: 100%|██████████| 6000/6000 [00:03<00:00, 1600.45draws/s]


Running: k2

Auto-assigning NUTS sampler...
Initializing NUTS using advi+adapt_diag...
Average Loss = 100.25:  12%|▉         | 23691/200000 [00:13<01:39, 1764.44it/s]
Convergence achieved at 23700
Interrupted at 23,699 [11%]: Average Loss = 66,768
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, np.power(x, 2), x, Intercept]
Sampling 2 chains: 100%|██████████| 6000/6000 [00:04<00:00, 1389.61draws/s]


Running: k3

Auto-assigning NUTS sampler...
Initializing NUTS using advi+adapt_diag...
Average Loss = 105.36:  12%|▉         | 23581/200000 [00:14<01:46, 1658.03it/s]
Convergence achieved at 23600
Interrupted at 23,599 [11%]: Average Loss = 68,508
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, np.power(x, 3), np.power(x, 2), x, Intercept]
Sampling 2 chains: 100%|██████████| 6000/6000 [00:06<00:00, 860.92draws/s]


Running: k4
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using advi+adapt_diag...
Average Loss = 109.99:  12%|▉         | 24092/200000 [00:15<01:50, 1597.36it/s]
Convergence achieved at 24200
Interrupted at 24,199 [12%]: Average Loss = 65,362
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, np.power(x, 4), np.power(x, 3), np.power(x, 2), x, Intercept]
Sampling 2 chains: 100%|██████████| 6000/6000 [00:11<00:00, 533.93draws/s]
There were 3 divergences after tuning. Increase `target_accept` or reparameterize.


Running: k5

Auto-assigning NUTS sampler...
Initializing NUTS using advi+adapt_diag...
Average Loss = 114.75:  12%|▉         | 24447/200000 [00:15<01:50, 1584.35it/s]
Convergence achieved at 24500
Interrupted at 24,499 [12%]: Average Loss = 64,276
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, np.power(x, 5), np.power(x, 4), np.power(x, 3), np.power(x, 2), x, Intercept]
Sampling 2 chains: 100%|██████████| 6000/6000 [00:28<00:00, 208.21draws/s]
There were 62 divergences after tuning. Increase `target_accept` or reparameterize.
There were 188 divergences after tuning. Increase `target_accept` or reparameterize.
The acceptance probability does not match the target. It is 0.6372358436158017, but should be close to 0.8. Try to increase th
The number of effective samples is smaller than 10% for some parameters.
```

In [13]:
```
models_quad, traces_quad = run_models(dfs_quad, 5)
```

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


Running: k1

Average Loss = 1.6448e+06:   7%|▊         | 13881/200000 [00:08<01:50, 1685.36it/s]
Convergence achieved at 14000
Interrupted at 13,999 [6%]: Average Loss = 5.4105e+08
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, x, Intercept]
Sampling 2 chains: 100%|██████████| 6000/6000 [00:03<00:00, 1700.82draws/s]


Running: k2

Auto-assigning NUTS sampler...
Initializing NUTS using advi+adapt_diag...
Average Loss = 193.06:  15%|▉         | 30693/200000 [00:15<01:26, 1959.84it/s]
Convergence achieved at 30700
Interrupted at 30,699 [15%]: Average Loss = 2.2511e+08
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, np.power(x, 2), x, Intercept]
Sampling 2 chains: 100%|██████████| 6000/6000 [00:03<00:00, 1679.86draws/s]


Running: k3

Auto-assigning NUTS sampler...
Initializing NUTS using advi+adapt_diag...
Average Loss = 197.5:  15%|▉         | 30539/200000 [00:19<01:45, 1605.67it/s]
Convergence achieved at 30700
Interrupted at 30,699 [15%]: Average Loss = 2.1858e+08
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, np.power(x, 3), np.power(x, 2), x, Intercept]
Sampling 2 chains: 100%|██████████| 6000/6000 [00:04<00:00, 1332.04draws/s]


Running: k4

Auto-assigning NUTS sampler...
Initializing NUTS using advi+adapt_diag...
Average Loss = 176.83:  16%|▉         | 31896/200000 [00:18<01:36, 1740.96it/s]
Convergence achieved at 32000
Interrupted at 31,999 [15%]: Average Loss = 2.0112e+08
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, np.power(x, 4), np.power(x, 3), np.power(x, 2), x, Intercept]
Sampling 2 chains: 100%|██████████| 6000/6000 [00:04<00:00, 1234.69draws/s]


Running: k5

Auto-assigning NUTS sampler...
Initializing NUTS using advi+adapt_diag...
Average Loss = 175.7:  16%|▉         | 32381/200000 [00:20<01:43, 1614.69it/s]
Convergence achieved at 32500
Interrupted at 32,499 [16%]: Average Loss = 2.1393e+08
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, np.power(x, 5), np.power(x, 4), np.power(x, 3), np.power(x, 2), x, Intercept]
Sampling 2 chains: 100%|██████████| 6000/6000 [00:05<00:00, 1145.86draws/s]
```

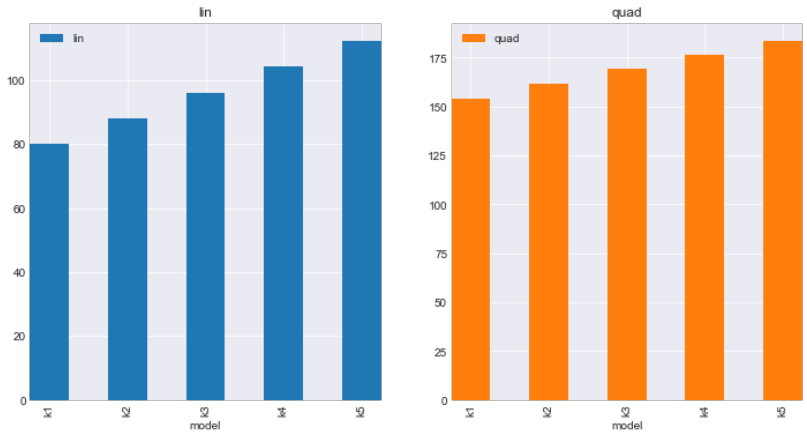
A really bad method for model selection: compare likelihoods¶

Evaluate log likelihoods straight from model.logp

In [14]:
```
dfll = pd.DataFrame(index=['k1','k2','k3','k4','k5'], columns=['lin','quad'])
dfll.index.name = 'model'

for nm in dfll.index:
    dfll.loc[nm,'lin'] = - models_lin[nm].logp(pm.summary(traces_lin[nm],
                                                          traces_lin[nm].varnames)['mean'].to_dict())

    dfll.loc[nm,'quad'] = - models_quad[nm].logp(pm.summary(traces_quad[nm],
                                                            traces_quad[nm].varnames)['mean'].to_dict())

dfll = pd.melt(dfll.reset_index(), id_vars=['model'],
               var_name='poly', value_name='log_likelihood')
dfll.index = pd.MultiIndex.from_frame(dfll[['model', 'poly']])
```

Plot log-likelihoods

In [15]:
```
ax = dfll['log_likelihood'].unstack().plot.bar(
    subplots=True, layout=(1, 2), figsize=(12, 6), sharex=True);

ax[0,0].set_xticks(range(5))
ax[0,0].set_xticklabels(['k1', 'k2', 'k3', 'k4', 'k5'])
ax[0,0].set_xlim(-.25, 4.25);
```



Observe:

- Again we're showing the linear-generated data at left (Blue) and the quadratic-generated data on the right (Green)
- For both datasets, as the models get more complex, the likelihood increases monotonically
- This is expected, since the models are more flexible and thus able to (over)fit more easily.
- This overfitting makes it a terrible idea to simply use the likelihood to evaluate the model fits.

View posterior predictive fit¶

Just for the linear, generated data, lets take an interactive look at the posterior predictive fit for the models k1 through k5.

As indicated by the likelihood plots above, the higher-order polynomial models exhibit some quite wild swings in the function in order to (over)fit the data

In [16]:
```
interactive(plot_posterior_cr, models=fixed(models_lin), traces=fixed(traces_lin),
            rawdata=fixed(dfs_lin), xlims=fixed(dfs_lin_xlims), datamodelnm=fixed('linear'),
            modelnm = ['k1','k2','k3','k4','k5'])
```

Compare models using WAIC¶

The Widely Applicable Information Criterion (WAIC) can be used to calculate the goodness-of-fit of a model using numerical techniques. See (Watanabe 2013) for details.

Observe:

- We get three different measurements:
  - waic: widely available information criterion
  - waic_se: standard error of waic
  - p_waic: effective number parameters

In this case we are interested in the WAIC score. We also plot error bars for the standard error of the estimated scores. This gives us a more accurate view of how much they might differ.

Now loop through all the models and calculate the WAIC

In [17]:
```
model_trace_dict = dict()
for nm in ['k1', 'k2', 'k3', 'k4', 'k5']:
    models_lin[nm].name = 'poly=lin, '+nm
    model_trace_dict.update({models_lin[nm]: traces_lin[nm]})

    models_quad[nm].name = 'poly=quad, '+nm
    model_trace_dict.update({models_quad[nm]: traces_quad[nm]})
```

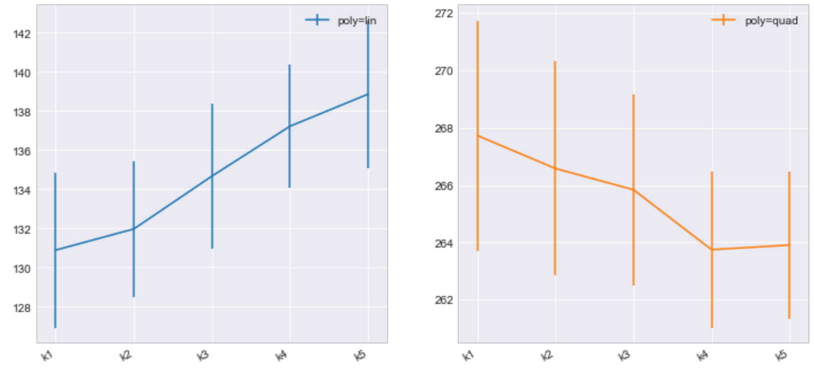
In [18]:
```
dfwaic = pm.compare(model_trace_dict, ic='WAIC')
dfwaic.index = pd.MultiIndex.from_tuples(
    [tuple(k.split(',')) for k,v in dfwaic.iterrows()])

dfwaic
```

```
/home/junpenglao/Documents/pymc3/pymc3/stats.py:218: UserWarning: For one or more samples the posterior variance of the
        log predictive densities exceeds 0.4. This could be indication of
        WAIC starting to fail see http://arxiv.org/abs/1507.04544 for details

  """)
/home/junpenglao/Documents/pymc3/pymc3/stats.py:218: UserWarning: For one or more samples the posterior variance of the
        log predictive densities exceeds 0.4. This could be indication of
        WAIC starting to fail see http://arxiv.org/abs/1507.04544 for details

  """)
/home/junpenglao/Documents/pymc3/pymc3/stats.py:218: UserWarning: For one or more samples the posterior variance of the
        log predictive densities exceeds 0.4. This could be indication of
        WAIC starting to fail see http://arxiv.org/abs/1507.04544 for details

  """)
/home/junpenglao/Documents/pymc3/pymc3/stats.py:218: UserWarning: For one or more samples the posterior variance of the
        log predictive densities exceeds 0.4. This could be indication of
        WAIC starting to fail see http://arxiv.org/abs/1507.04544 for details

  """)
/home/junpenglao/Documents/pymc3/pymc3/stats.py:218: UserWarning: For one or more samples the posterior variance of the
        log predictive densities exceeds 0.4. This could be indication of
        WAIC starting to fail see http://arxiv.org/abs/1507.04544 for details

  """)
/home/junpenglao/Documents/pymc3/pymc3/stats.py:218: UserWarning: For one or more samples the posterior variance of the
        log predictive densities exceeds 0.4. This could be indication of
        WAIC starting to fail see http://arxiv.org/abs/1507.04544 for details

  """)
/home/junpenglao/Documents/pymc3/pymc3/stats.py:218: UserWarning: For one or more samples the posterior variance of the
        log predictive densities exceeds 0.4. This could be indication of
        WAIC starting to fail see http://arxiv.org/abs/1507.04544 for details

  """)
/home/junpenglao/Documents/pymc3/pymc3/stats.py:218: UserWarning: For one or more samples the posterior variance of the
        log predictive densities exceeds 0.4. This could be indication of
        WAIC starting to fail see http://arxiv.org/abs/1507.04544 for details

  """)
```

Out[18]:

		WAIC	pWAIC	dWAIC	weight	SE	dSE	var_warn
poly=lin	k1	130.89	2.22	0	1	3.98	0	1
	k2	131.97	2.74	1.08	0	3.47	0.92	1
	k3	134.67	3.77	3.78	0	3.71	1.08	1
	k4	137.22	4.28	6.32	0	3.17	1.8	1
	k5	138.86	4.63	7.96	0	3.76	2.17	1
poly=quad	k4	263.74	0.92	132.85	0	2.73	5.21	1
	k5	263.9	1.04	133.01	0	2.57	5.27	1
	k3	265.83	0.85	134.93	0	3.33	5.25	1
	k2	266.57	0.64	135.67	0	3.73	5.4	0
	k1	267.72	0.63	136.82	0	4.01	5.53	0

In [19]:
```
ax = dfwaic['WAIC'].unstack().T.plot.line(
    yerr=dfwaic['SE'].unstack().T,
    subplots=True, layout=(1, 2), figsize=(12, 6), sharex=True);

ax[0,0].set_xticks(range(5))
ax[0,0].set_xticklabels(['k1', 'k2', 'k3', 'k4', 'k5'])
ax[0,0].set_xlim(-.25, 4.25);
```



Observe

- We should prefer the model(s) with lower WAIC
- Linear-generated data (lhs):
  - The WAIC seems quite flat across models
  - The WAIC seems best (lowest) for simpler models.
- Quadratic-generated data (rhs):
  - The WAIC is also quite flat across the models
  - The lowest WAIC is model **k4**, but **k3** - **k5** are more or less the same.

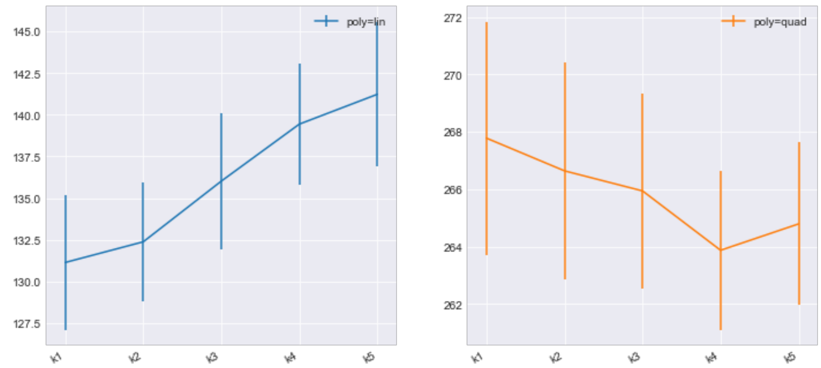
In [20]:
```
dfloo = pm.compare(model_trace_dict, ic='LOO')
dfloo.index = pd.MultiIndex.from_tuples(
    [tuple(k.split(',')) for k,v in dfloo.iterrows()])

dfloo
```

```
/home/junpenglao/Documents/pymc3/pymc3/stats.py:299: UserWarning: Estimated shape parameter of Pareto distribution is
        greater than 0.7 for one or more samples.
        You should consider using a more robust model, this is because
        importance sampling is less likely to work well if the marginal
        posterior and LOO posterior are very different. This is more likely to
        happen with a non-robust model and highly influential observations.
  happen with a non-robust model and highly influential observations.""")
/home/junpenglao/Documents/pymc3/pymc3/stats.py:299: UserWarning: Estimated shape parameter of Pareto distribution is
        greater than 0.7 for one or more samples.
        You should consider using a more robust model, this is because
        importance sampling is less likely to work well if the marginal
        posterior and LOO posterior are very different. This is more likely to
        happen with a non-robust model and highly influential observations.
  happen with a non-robust model and highly influential observations.""")
/home/junpenglao/Documents/pymc3/pymc3/stats.py:299: UserWarning: Estimated shape parameter of Pareto distribution is
        greater than 0.7 for one or more samples.
        You should consider using a more robust model, this is because
        importance sampling is less likely to work well if the marginal
        posterior and LOO posterior are very different. This is more likely to
        happen with a non-robust model and highly influential observations.
  happen with a non-robust model and highly influential observations.""")
/home/junpenglao/Documents/pymc3/pymc3/stats.py:299: UserWarning: Estimated shape parameter of Pareto distribution is
        greater than 0.7 for one or more samples.
        You should consider using a more robust model, this is because
        importance sampling is less likely to work well if the marginal
        posterior and LOO posterior are very different. This is more likely to
        happen with a non-robust model and highly influential observations.
  happen with a non-robust model and highly influential observations.""")
```

Out[20]:

		LOO	pLOO	dLOO	weight	SE	dSE	shape_warn
poly=lin	k1	131.14	2.34	0	1	4.05	0	0
	k2	132.38	2.94	1.24	0	3.54	0.97	0
	k3	136.02	4.44	4.88	0	4.08	1.38	1
	k4	139.44	5.39	8.3	0	3.62	2.52	1
	k5	141.23	5.82	10.09	0	4.35	2.67	1
poly=quad	k4	263.87	0.98	132.72	0	2.78	5.29	0
	k5	264.79	1.48	133.65	0	2.84	5.35	1
	k3	265.94	0.9	134.8	0	3.4	5.34	0
	k2	266.64	0.68	135.49	0	3.78	5.49	0
	k1	267.78	0.66	136.63	0	4.06	5.62	0

In [21]:
```
ax = dfloo['LOO'].unstack().T.plot.line(
    yerr=dfloo['SE'].unstack().T,
    subplots=True, layout=(1, 2), figsize=(12, 6), sharex=True);

ax[0,0].set_xticks(range(5))
ax[0,0].set_xticklabels(['k1', 'k2', 'k3', 'k4', 'k5'])
ax[0,0].set_xlim(-.25, 4.25);
```



Compare leave-one-out Cross-Validation [LOO]¶

Leave-One-Out Cross-Validation or K-fold Cross-Validation is another quite universal approach for model selection. However, to implement K-fold cross-validation we need to paritition the data repeatly and fit the model on every partition. It can be very time consumming (computation time increase roughly as a factor of K). Here we are applying the numerical approach using the posterier trace as suggested in Vehtari et al 2015.

Observe

- We should prefer the model(s) with lower LOO. You can see that LOO is nearly identical with WAIC. That's because WAIC is asymptotically equal to LOO. However, PSIS-LOO is supposedly more robust than WAIC in the finite case (under weak priors or influential observation).
- Linear-generated data (lhs):
  - The LOO is also quite flat across models
  - The LOO is also seems best (lowest) for simpler models.
- Quadratic-generated data (rhs):
  - The same pattern as the WAIC

Final remarks and tips¶

It is important to keep in mind that, with more data points, the real underlying model (one that we used to generate the data) should outperform other models.

There is some agreement that PSIS-LOO offers the best indication of a model's quality. To quote from avehtari's comment: "I also recommend using PSIS-LOO instead of WAIC, because it's more reliable and has better diagnostics as discussed in http://link.springer.com/article/10.1007/s11222-016-9696-4 (preprint https://arxiv.org/abs/1507.04544), but if you insist to have one information criterion then leave WAIC".

Alternatively, Watanabe says "WAIC is a better approximator of the generalization error than the pareto smoothing importance sampling cross validation. The Pareto smoothing cross validation may be the better approximator of the cross validation than WAIC, however, it is not of the generalization error".

Reference¶

For more information on Model Selection in PyMC3, and about Bayesian model selection, you could start with:

- Thomas Wiecki's detailed response to a question on Cross Validated
- The Deviance Information Criterion: 12 Years On (Speigelhalter et al 2014)
- Bayesian predictive information criterion for the evaluation of hierarchical Bayesian and empirical Bayes models (Ando 2007)
- A Widely Applicable Bayesian Information Criterion (Watanabe 2013)
- Efficient Implementation of Leave-One-Out Cross-Validation and WAIC for Evaluating Fitted Bayesian Models (Vehtari et al 2015)

Example originally contributed by Jonathan Sedar 2016-01-09 github.com/jonsedar. Edited by Junpeng Lao 2017-07-6 github.com/junpenglao