Cheat Sheet: Stan, PyStan and ArviZ

Preliminaries

Start by importing these Python modules

```
import pystan
import arviz as az
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Version check

```
import sys
print('Python:', sys.version)
print('PySTAN:', pystan.__version__)
print('ArviZ:', az.__version__)
```

Python version 3.7.1, PyStan 2.18.0.0, and ArviZ 0.3.1

Code samples can be found online

```
https://drive.google.com/drive/folders/1Yjqav
hopT5RAjvXqPH5RtWjFPHf-Ugzh?usp=sharing
```

Caching compiled Stan models: stan_cache()

Because models can take some time to compile, I use a Python caching function as follows.

- I keep the Python caching function in a file called "./bin/stan-cache.py" relative to the current directory.
- The function expects to find a "./stan-cache" subdirectory from the current directory for storing the cached stan model compilations.

```
import sys
import re
import gzip
import pystan
import pickle
from hashlib import md5

def stan_cache(model_code, m_name=None):
    code_hash = md5(model_code.encode(
        'ascii')).hexdigest()

    pre = './stan-cache/' # Cache home dir
    fname = pre + m_name if m_name else pre
    fname += ('-' + sys.version + '-' +
        pystan.__version__ + '-' + code_hash
        + '.pkl.gz')
    rex = re.compile('[^a-zA-Z0-9_ ,/\.\-]')
    fname = rex.sub('', fname)

    try:
        sm = pickle.load(
            gzip.open(fname, 'rb'))
    except:
        print("About to compile model")
        sm = pystan.StanModel(
            model_code=model_code)
        with gzip.open(fname, 'wb') as fh:
            pickle.dump(sm, fh)
    else:
        print("Using cached model")

    return sm
```

Putting it all together

In this cheat sheet we will follow an iterative build-fit-analyse process for Bayesian analysis:

1. ***Build*** – using **Stan** (the language) we will build a probability model that describes the problem we are trying to solve.

2. ***Fit*** – once we have built a probability model in the Stan language, we use the **PyStan** module in **Python** to call **Stan** (the software application) to
    - first compile our model; and then to
    - condition our model on the observed data (also known as fitting the model).

3. ***Analyse*** – finally evaluate the fit of our model using **ArviZ** (another Python package), which has:
    - functions for the visualisation of and reporting on our model parameters; and
    - a raft of diagnostic tools to confirm that Stan has sufficiently and appropriately explored the posterior distribution (such that we can be confident in the results).

4. ***Repeat*** – after considering our analysis, we may need to refine the probability model, reparameterise the model, rescale our data, or build a different model entirely.

A Simple Linear Regression

Model

$y_i = \alpha + \beta x_i + \varepsilon_i$ where: $\varepsilon_i \sim N(0, \sigma)$

Which is equivalent to: $y_i \sim N(\alpha + \beta x_i, \sigma)$

Stan model – a simple linear regression

```
data {
    // input data passed from Python
    int<lower=1> N;
    vector[N] x;
    vector[N] y;
}
parameters {
    real alpha;
    real beta;
    real<lower=0> sigma;      // must be +ve
}
model {
    // priors set with an eye to data
    // but only to be weakly informative
    alpha ~ normal(0, 200);
    beta ~ normal(0, 10);
    sigma ~ cauchy(0, 10);     // half Cauchy
    y ~ normal(alpha + beta * x, sigma);
}
```

The data

We will use this regression formula to model weight (in pounds) as a function of height (in inches) with data for 717 NHL players from 2013-14. The data is found here:

```
http://users.stat.ufl.edu/~winner/data/nhl_ht
_wt.csv
```

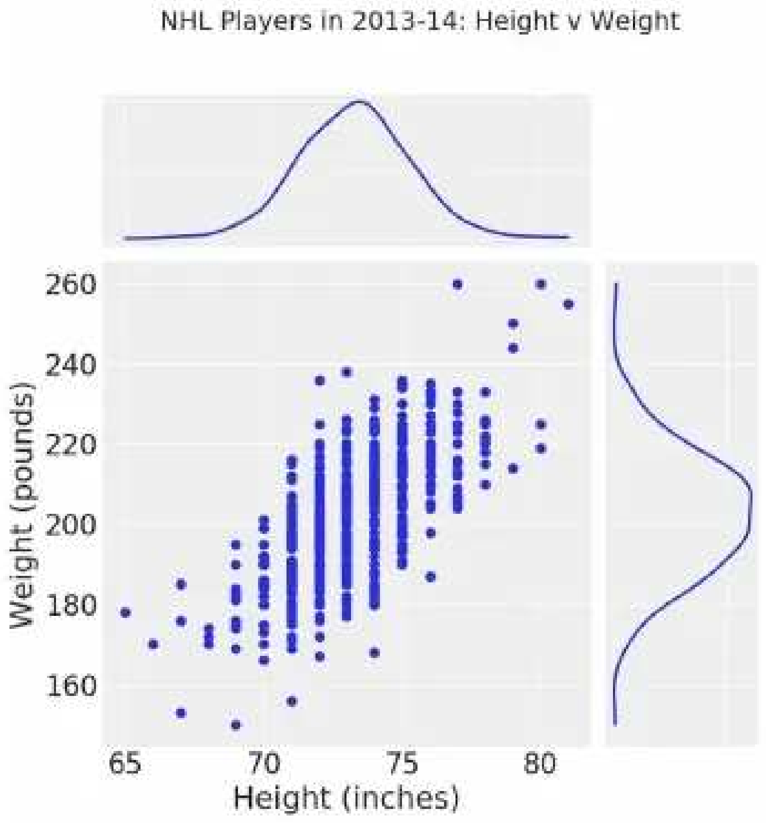
Python – load the data

```
# --- preliminaries
import pystan
import pystan.diagnostics as psd
import arviz as az
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import sys
sys.path.append( './bin' )
from stan_cache import stan_cache
az.style.use('arviz-darkgrid')
graph_dir = './graphs/simple-regression-'

# --- get and package-up the data
df = pd.read_csv('http://users.stat.ufl.edu'+
    '/~winner/data/nhl_ht_wt.csv',
    encoding='windows-1252', header=0)
x = df['Height'].astype(float).values
y = df['Weight'].astype(float).values
data = {'N': len(x), 'x': x, 'y': y}
```

Python – let's look at the data before doing anything

```
# --- look at the data
azd={'Height (inches)':x,'Weight (pounds)':y}
az.plot_joint(azd, figsize=(8,8))
fig = plt.gcf()
fig.suptitle('NHL Players in 2013-14: '+
    'Height v Weight', size='xx-large')
fig.tight_layout(pad=1)
fig.savefig(graph_dir+'joint.png', dpi=125)
plt.close()
```

The data appears to have a linear relationship that is suited to a simple regression:



Hint: az.plot_joint(data_dict) plots integers with histograms and floating point numbers with densities. We type-cast the data from integers to floats to get density plots.

Python – get and run the Stan model

```
# --- get the Stan model code
with open ("./models/simple-linear-"+
    "regression.stan", "r") as f:
    model_code = f.read()
    f.close()

# --- compile and run the Stan model
model = stan_cache(model_code=model_code)
fit = model.sampling(data=data, iter=2000,
    chains=4)
samples = fit.extract()
```

Note: You should run Stan with four or more chains. Never use just one chain.

Python – check the diagnostics

```
# --- always check the diagnostics
import pystan.diagnostics as psd
d = pd.Series(psd.check_hmc_diagnostics(fit))
print(d) # --- False indicates a problem
if not d.all():
    print(fit.stansummary())
```

Python – look at the results using ArviZ

```
# --- get the inference data for ArviZ
id = az.convert_to_inference_data(obj=fit)

# --- print key results
params = ['alpha', 'beta', 'sigma']
print(az.summary(id, params))
```

ArviZ output – summary statistics / diagnostics

```
          mean    sd mcerror  hpd 3%  hpd 97%  eff_n r_hat
alpha  -162.44 14.30     0.2 -192.51  -138.19   1032   1.0
beta      4.99  0.19    0.00    4.64     5.38   1032   1.0
sigma    11.09  0.29    0.01   10.57    11.66   1451   1.0
```

Both the `fit.stansummary()` PyStan method and the az.summary() ArviZ function summarise the Stan model parameters and provide diagnostics:

- **eff_n** – also reported as **n_eff** – because Markov chains are serially auto-correlated, we use a scale reduction factor on split chains as a crude indicator of the number of effective samples drawn in Stan's processes. Ensure eff_n is large enough for your use case.

- **Rhat** – We use the Gelman-Rubin convergence diagnostic to see if the different chains converged to the same part of the parameter space. Rhat should be greater than 0.9 and less than 1.1. If this test fails, your parameter estimates could well be invalid. Either draw more samples (the `iter` argument) or re-parameterise the model.

- **mc-error** –the Monte Carlo simulation error arises from only having a finite number of draws from the posterior. As a rule of thumb: aim for an mc-error of less than 5% of the standard deviation. To reduce the mc-error you should draw more samples.

ArviZ summary function will also tell you if it finds other problems with Stan's computations:

- **Divergences** – are bad – really bad. They occur when the model and data results in a region of extremely high curvature, which is difficult to explore. Divergences result in biased (invalid) parameter estimates. Sometimes it can be fixed by changing the step size parameter in the sampling method from its default of 0.8, for example: `control={'adapt_delta':0.90}`, but stay in the range >0.8 and <1.0. More likely, you will need to re-parameterise the model to fix things.
- The **treedepth** warnings are less serious than divergences. They go to the efficiency with which Stan is exploring the parameter space. With complex models, these warnings can often be fixed by increasing the treedepth from the default of 10. Try: `control={'max_treedepth': 12}` or even 15. If this fails, re-parameterise the model.
- **Energy** (also known as the Bayesian Fraction of Missing Information – BFMI) diagnostics may suggest the warmup phase was incomplete or not long enough. Their occurrence suggests that one or more chains is not exploring the posterior distribution efficiently. Try increasing `iter`. If that fails, re-parameterise the model.

Hint: pay extra attention to Rhat and divergence.
See also: http://mc-stan.org/misc/warnings.html

Python – more diagnostic plots (if needed)

```
# --- Further explore diagnostics if needed
d = id.sample_stats.diverging.values.any()
print('\nAny divergences: {}'.format(d))
if d :
    az.plot_pair(id, params,
        divergences= True, figsize=(8,8))
    fig = plt.gcf()
    fig.savefig(graph_dir+
        'divergent.png', dpi=125)
    plt.close()

print('\nCheck energy greater than 0.3 ...')
e = az.bfmi(id.sample_stats.energy).min()
print(e)
if e < 0.4:
    az.plot_energy(id)
    fig = plt.gcf()
    fig.set_size_inches(8, 4)
    fig.tight_layout(pad=1)
    fig.savefig(graph_dir+
        'enery.png', dpi=125)
    plt.close()
```

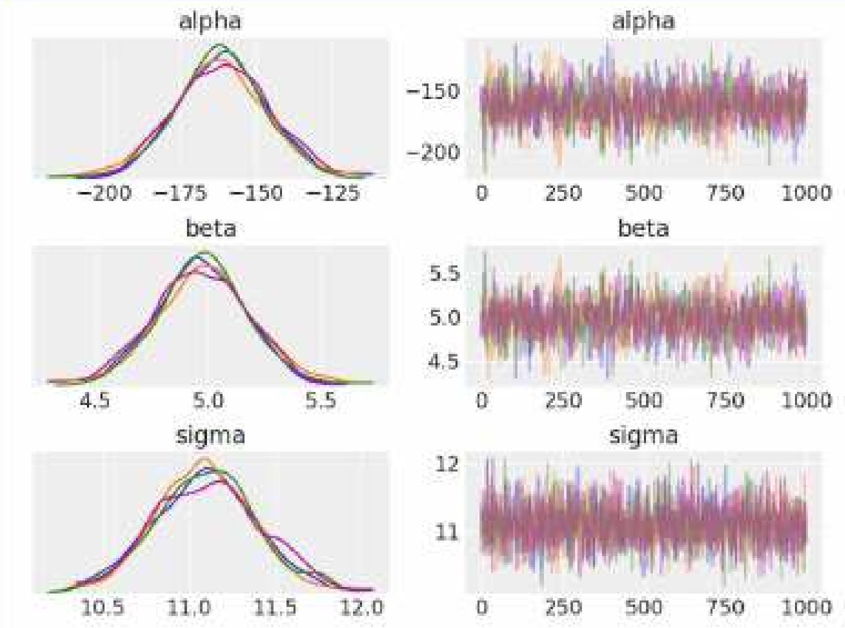
Further diagnostic output – all okay

```
Any divergences: False

Check energy greater than 0.3 ...
1.0859169253490843
```

Python – diagnostic parameter plots

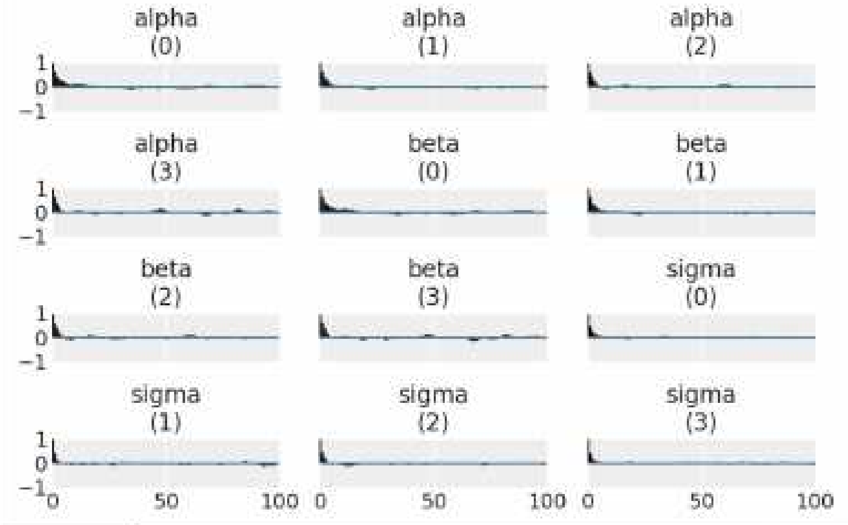
```
# --- density and trace plot the results
az.plot_trace(id, var_names=params)
fig = plt.gcf()
fig.set_size_inches(8, 6)
fig.tight_layout(pad=1)
fig.savefig(graph_dir+'trace.png', dpi=125)
plt.close()
```

On the left-hand side of this plot you can check that the kernel density estimate for each chain is roughly the same. On the right-hand side of this plot you can check that all the traces look like "hairy caterpillars". Lumpy, stepped, troughed or uneven traces suggest Stan is not exploring all the parameter space evenly.



We can also check auto-correlation in the chains

```
# --- auto-correlation plot
az.plot_autocorr(fit, var_names=['alpha',
    'beta', 'sigma'])
fig = plt.gcf()
fig.set_size_inches(8, 5)
fig.tight_layout(pad=1)
fig.savefig(graph_dir+'acf.png', dpi=125)
plt.close()
```



You expect some auto-correlation with Markov chain processes. The above plot is not bad at all. It is consistent with the eff_n diagnostic above.

So, what did we find?
From the diagnostics, we did not see any problems in Stan's computational processes.

For the regression, we need to check that zero does not lie in the credible interval for either alpha or beta. If zero is within the credible interval, then that term is not adding to the regression equation. In our case, zero is not within either credible interval, so our result is meaningful.

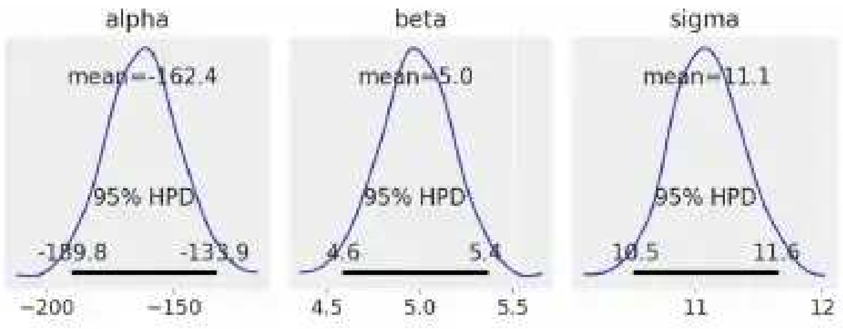
If we use the mean results, we found the following relationship between weight and height for NHL players in 2013-14:

$$Weight(pounds) = -162.4 + 5.0 * Height(inches)$$

(for NHL players in 2013-14)

Python – plot key results and the credible interval

```
# --- posterior plots
az.plot_posterior(id, var_names=params,
    credible_interval=0.95)
fig = plt.gcf()
fig.set_size_inches(8, 3)
fig.savefig(graph_dir+'post.png', dpi=125)
plt.close()
```

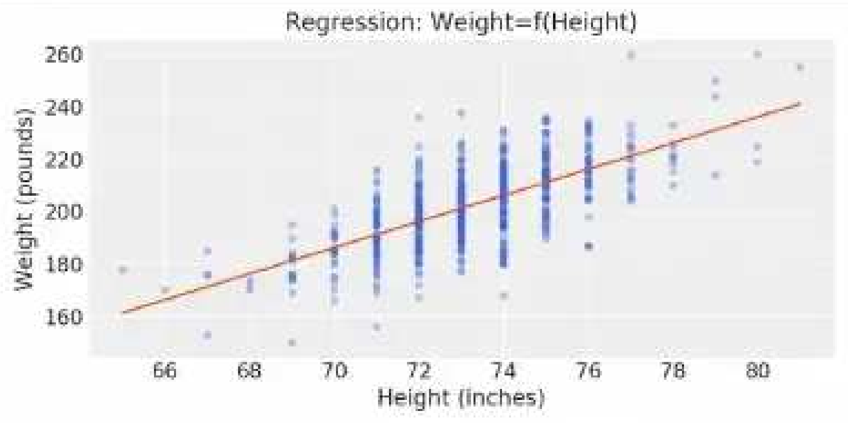


Note: HPD = highest posterior density = credible interval

Python – Scatter plot of the solved regression

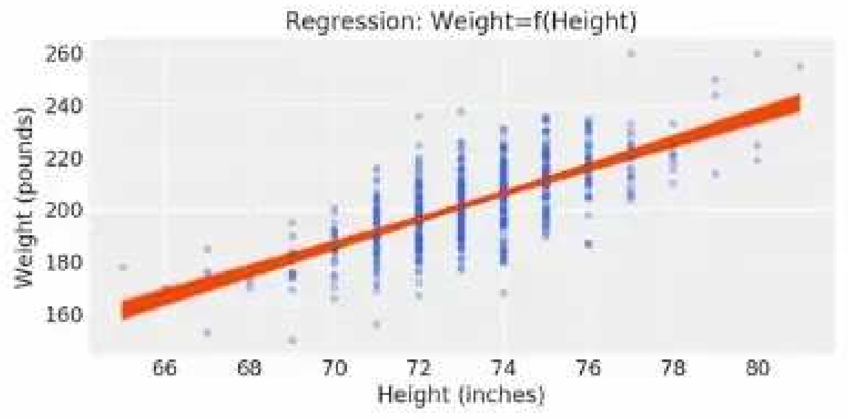
```
# --- scatter plot with regression line(s)
dF = pd.DataFrame({'Height':x, 'Weight':y})
ax = dF.plot.scatter(x='Height', y='Weight',
    color='royalblue', alpha=0.3)
a = np.median(results['alpha'])
b = np.median(results['beta'])
x1 = np.min(x)
x2 = np.max(x)
line = pd.Series([a+b*x1, a+b*x2],
    index=[x1, x2])
ax = line.plot(ax=ax, color='orangered')
ax.set_title('Regression: Weight=f(Height)')
ax.set_ylabel('Weight (pounds)')
ax.set_xlabel('Height (inches)')
fig = ax.figure
fig.set_size_inches(8, 4)
fig.tight_layout(pad=1)
fig.savefig(graph_dir+'scatter.png', dpi=125)
```

The next plot has the median regression line.



Plot all sampled regression lines

```
for i in range(len(samples['alpha'])):
    a = samples['alpha'][i]
    b = samples['beta'][i]
    line = pd.Series([a+b*x1, a+b*x2],
        index=[x1, x2])
    ax = line.plot(ax=ax, color='orangered',
        alpha=0.02, linewidth=0.5)
fig.savefig(graph_dir+'scatter+.png',dpi=125)
plt.close()
```



Python – final check in with classical statistics

```
# --- compare with classic regression
import statsmodels.api as sm
xc = sm.add_constant(x)
model = sm.OLS(y, xc).fit()
print(model.summary())
```

Output from classical statistcs

```
                 coef    std err      P>|t|
-------------------------------------------
const       -224.4988      3.411      0.000
x1             5.9618      0.049      0.000
```

Multiple Regression

Model

$$y = X\beta + \varepsilon \quad \text{where each } \varepsilon_i \sim N(0,\sigma)$$

Which is equivalent to: $y \sim N(X\beta,\sigma)$

Note: This model is written in matrix form: **y** is a column vector of response variables. **X** is the design matrix of regressors (or covariates), with variables in columns, and the cases are in rows. ε is the error column vector. An intercept term would be recorded as a column of ones (1s) in the design matrix.

Stan model

```
data {
    // from Python
    int<lower=1> N; // design matrix row num
    int<lower=1> K; // design matrix col num
    matrix[N, K] x; // the design matrix
    vector[N] y; // response vector - quality
}
parameters {
    // will be returned to Python
    vector[K] beta;
    real<lower=0> sigma;      // must be +ve
}
model {
    // Note: priors set with reference to
    // the data to be weakly informative.
    beta ~ normal(0, 500);
    sigma ~ cauchy(0, 200);   // half Cauchy
    yy ~ normal(x * beta, sigma);
}
```

Hint: it is often necessary to re-parameterise a model or rescale the data before a Stan HMC model will run smoothly. There are two options in the Stan manual for re-parameterising a multiple regression: the QR representation, and the non-centred representation.

The data

```
addr = 'http://archive.ics.uci.'+
    'edu/ml/machine-learning-databases/'+
    'wine-quality/winequality-red.csv'
```

This file contains a quality score (between 0 and 10), as well as eleven measured physical attributes for 1599 red wines. All of the values for the physical attributes are greater than or equal to zero.

In Python, I have rescaled the quality score from 0 to 10 to -500 to +500. This data rescaling was necessary for the model to work smoothly.

```
y = (y - 5) * 100 # data rescaling
```

ArviZ summary of the initial run

The initial run had the 11 covariates and an intercept term. There are warning diagnostics. There are also regression parameters that include zero in the credible interval (highlighted below).

```
              mean     sd mcerrr   hpd 3% hpd 97%  eff_n r_hat
beta[0]       0.86   1.66   0.05    -2.34    3.90 2082.0   1.0
beta[1]    -109.55  12.08   0.33  -131.81  -87.49 2576.0   1.0
beta[2]     -18.38  14.88   0.40   -47.29    8.60 2093.0   1.0
beta[3]       0.92   1.19   0.03    -1.25    3.17 3782.0   1.0
beta[4]    -190.27  41.60   1.22  -268.63 -114.73 3004.0   1.0
beta[5]       0.45   0.22   0.00     0.03    0.85 2289.0   1.0
beta[6]      -0.33   0.07   0.00    -0.48   -0.20 2205.0   1.0
beta[7]     -67.50 350.91   6.28  -673.40  644.84 1690.0   1.0
beta[8]     -50.39  15.84   0.27   -80.66  -20.55 2840.0   1.0
beta[9]      89.38  11.27   0.24    68.43  110.15 3586.0   1.0
beta[10]     29.24   1.76   0.02    25.86   32.45 3287.0   1.0
beta[11]     12.30 345.30   5.19  -663.75  639.36 1759.0   1.0

WARNING:pystan:3607 of 4000 iterations saturated the
maximum tree depth of 10 (90.175%)
WARNING:pystan:Run again with max_treedepth larger than
10 to avoid saturation
```

Note: ArviZ indexes beta from 0 to 11. In Stan, they would have been indexed from 1 to 12.

What should I do?

Because we cannot be certain that the beta parameters 0, 2, 3 and 7 are significantly different to zero, we cannot be certain that they should be contributing to the regression model. So let's run the model again without the betas that include zero within their 94 per cent credible intervals (ie. we will drop betas 0, 2, 3 and 7). We will also drop the intercept term (beta[11]) as it includes zero within its 95 per cent credible interval.

We will check in again on the treedepth warnings after we have removed these parameters.

ArviZ summary with the revised covariate set

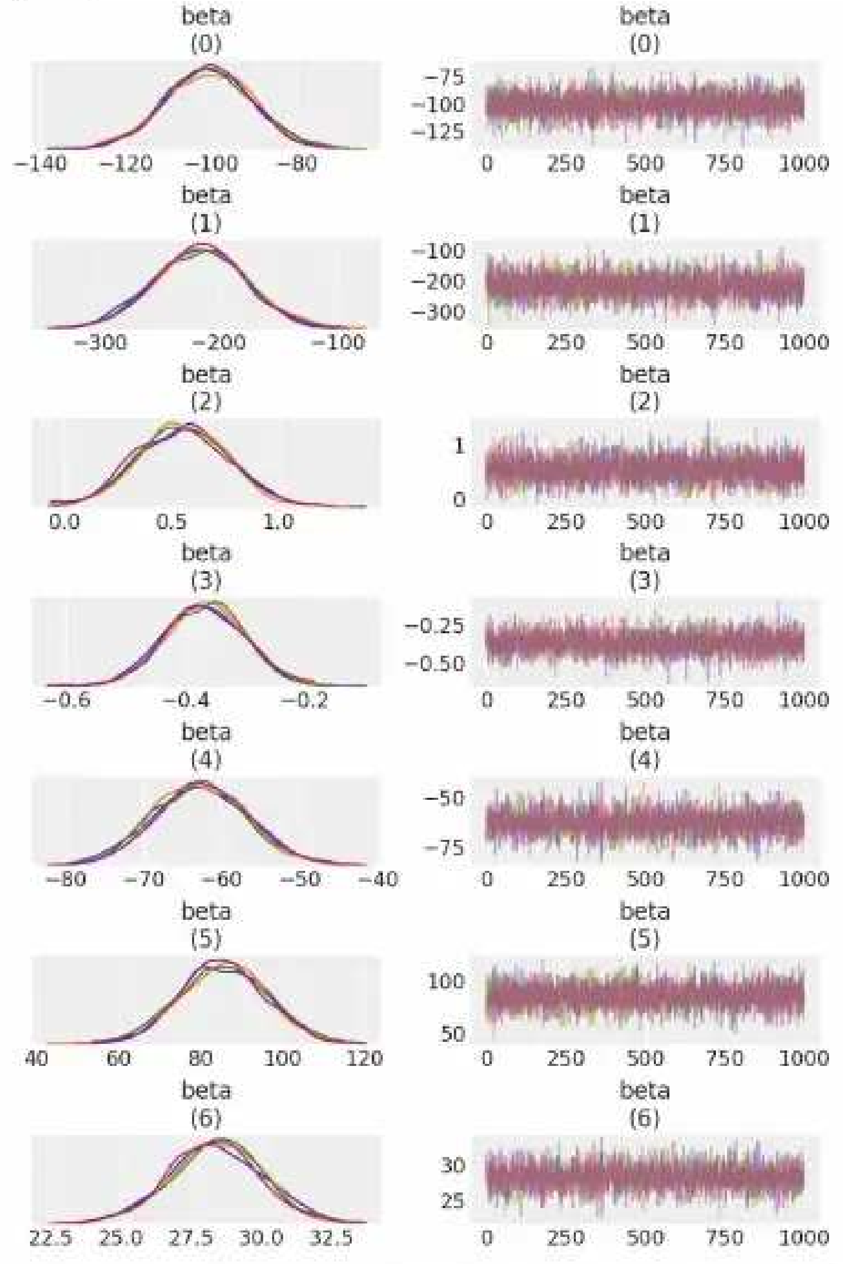
```
             mean    sd mcerrr   hpd 3% hpd 97%   eff_n r_hat
beta[0]   -100.92  9.96   0.07  -121.01  -83.78  3005.0   1.0
beta[1]   -215.00 37.48   0.46  -284.37 -147.97  3147.0   1.0
beta[2]      0.55  0.21   0.00     0.14    0.94  2937.0   1.0
beta[3]     -0.37  0.07   0.00    -0.49   -0.24  3002.0   1.0
beta[4]    -62.57  5.92   0.06   -73.36  -51.32  2074.0   1.0
beta[5]     85.41 10.60   0.09    65.44  104.66  2843.0   1.0
beta[6]     28.31  1.61   0.02    25.18   31.23  2195.0   1.0
```

Well that worked a treat. No Stan diagnostics, and zero is not found within the 94% credible interval for the remaining parameters for this regression.

Note: the regression now runs very fast, which is usually a good sign.

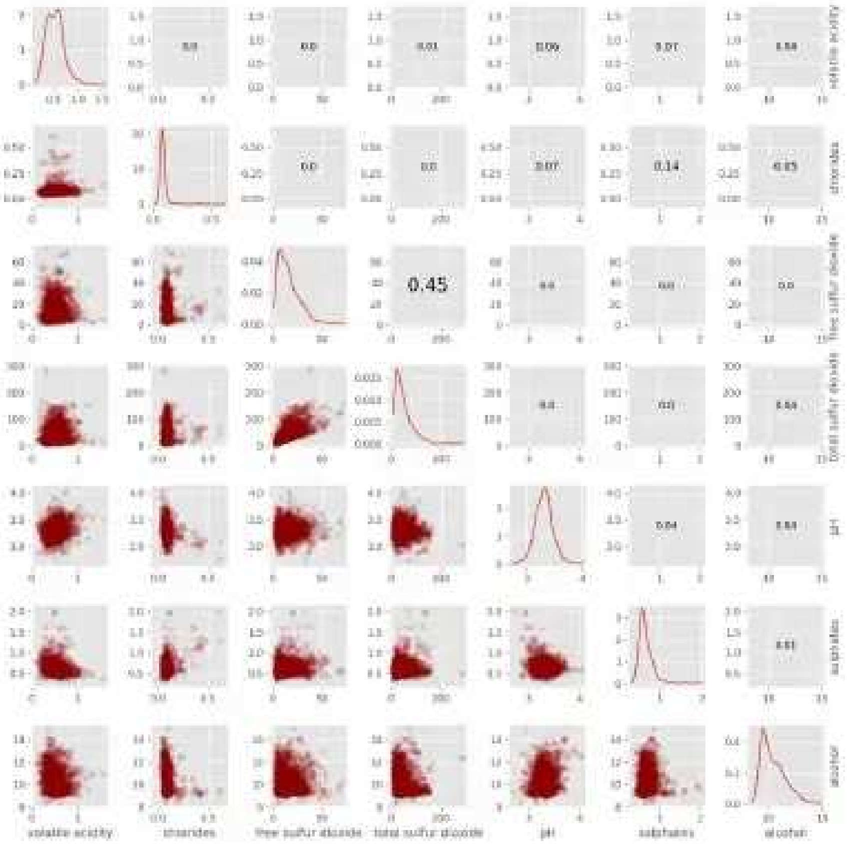
ArviZ – trace plots from the second run

The remaining betas (now renumbered) can be seen in the next parameter kernel density and trace plots. These all look good.



Check in on a scatter plot matrix (SPLOM)

Check whether the covariates are too correlated. This looks fine (all pair-wise R^2s are less than 0.9). If you find pairs that are highly correlated, you might consider removing one from the pair from the regression.



Note: The Python code for this splom plot can be found in the bin directory. Splom plots work well with continuous variables, less so with discrete variables.

Conclusion:

$$RedWineQuality = -101 * volatileAcidity - 215 * chlorides + 0.55 * freeSulfurDisoxide - 0.37 * totalSulfurDioxide - 62.57 * pH + 85.4 * sulphates + 28.3 * alcohol$$

Where wine quality is scored from -500 to +500.

Finally, compare the results with classical statistics

```
==============================================
           coef    std err     [0.025    0.975]
----------------------------------------------
x1  -100.6917     10.079   -120.461   -80.922
x2  -217.0610     38.270   -292.127  -141.995
x3     0.5497      0.211      0.137     0.963
x4    -0.3683      0.067     -0.500    -0.236
x5   -62.6377      5.915    -74.239   -51.036
x6    85.7080     10.844     64.437   106.979
x7    28.3225      1.624     25.137    31.508
==============================================
```

Note: the p-values for these coefficients were all close to zero.

A Bayesian t-test – comparing means

In classical statistics, the t-test is used to compare two means (or averages) and say whether the difference in those means is significant. It is often the test behind the statement that the difference between two groups is statistically significant (which means the difference is unlikely to have happened by chance).

There are three common types of t-test:

- 2-sample t-test – to see whether the means from two intependent sample groups are the same or different (eg. compare men versus women or a control group against a treatment group).
- Paired-sample t-test – compares the same group on two occasions (eg. before and after a treatment).
- 1-sample t-test: compares the mean for a sample group with a known population mean.

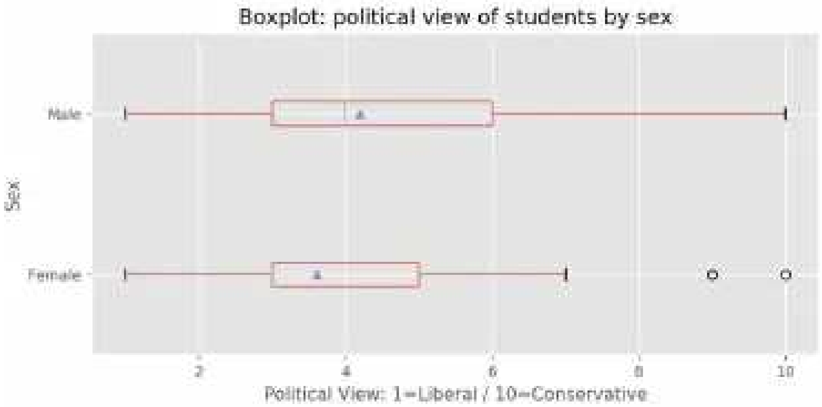
The Data

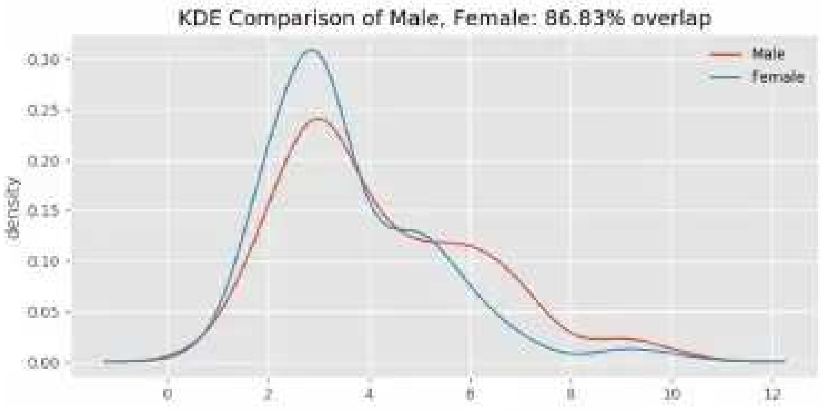
```
ds = ('https://dasl.datadescription.com/'+
    'download/data/3465')
```

This data is a survey of 299 students in a class, which rates student political views (among other questions) between 1 (Liberal) and 10 (Conservative). The question we will explore is whether there is a significant difference between the mean view of male and female students. For more information see: https://dasl.datadescription.com/datafile/student-survey/

A little data cleaning was necessary as one of the records did not include a numeric political view score.

The data is visualised in the next two plots. The mean view is indicated on the boxplot by a blue triangle. We will use a two-sample t-test to answer this question.





The Stan code for a Bayesian 2-sample t-test

```
// STAN: Bayesian t-Test
data {
    int<lower=1> Nx;
    int<lower=1> Ny;
    vector[Nx] x;
    vector[Ny] y;
}
transformed data{
    real<lower=0> sd_x = (max(x)-min(x))/4.0;
    real<lower=0> sd_y = (max(y)-min(y))/4.0;
    real mean_input_x = mean(x);
    real mean_input_y = mean(y);
}
parameters {
    real<lower=0> sigma_x;
    real<lower=0> sigma_y;
    real mu_x;
    real mu_y;
    real<lower=1> nu_x;
    real<lower=1> nu_y;
}
model {
    // a reparameterisation option follows
    //nu_x ~ exponential(1.0 / 29.0);
    //nu_y ~ exponential(1.0 / 29.0);

    nu_x ~ gamma(2, 0.1);
    nu_y ~ gamma(2, 0.1);
    sigma_x ~ cauchy(0, sd_x);
    sigma_y ~ cauchy(0, sd_y);
    mu_x ~ normal(mean_input_x, sd_x);
    mu_y ~ normal(mean_input_y, sd_y);

    x ~ student_t(nu_x, mu_x, sigma_x);
    y ~ student_t(nu_y, mu_y, sigma_y);
}
generated quantities {
    // check is zero in the credible interval
    // if so, accept the null hypothesis.
    real diff_mu = mu_x - mu_y;
    real diff_sigma = sigma_x - sigma_y;

    // Effect size:
    real Cohens_d = diff_mu / sqrt(
        (sigma_x^2 + sigma_y^2)/ 2 );
    real Hedges_g = diff_mu / sqrt(
        ((Nx-1)*sigma_x^2 + (Ny-1)*sigma_y^2)
        / (Nx + Ny - 2));
}
```

Note: This Stan code implements John Kruschke's 2013 paper, *Bayesian Estimation Supersedes the t-Test*. Kruschke's paper is easily found on the internet and is well worth reading.

Results

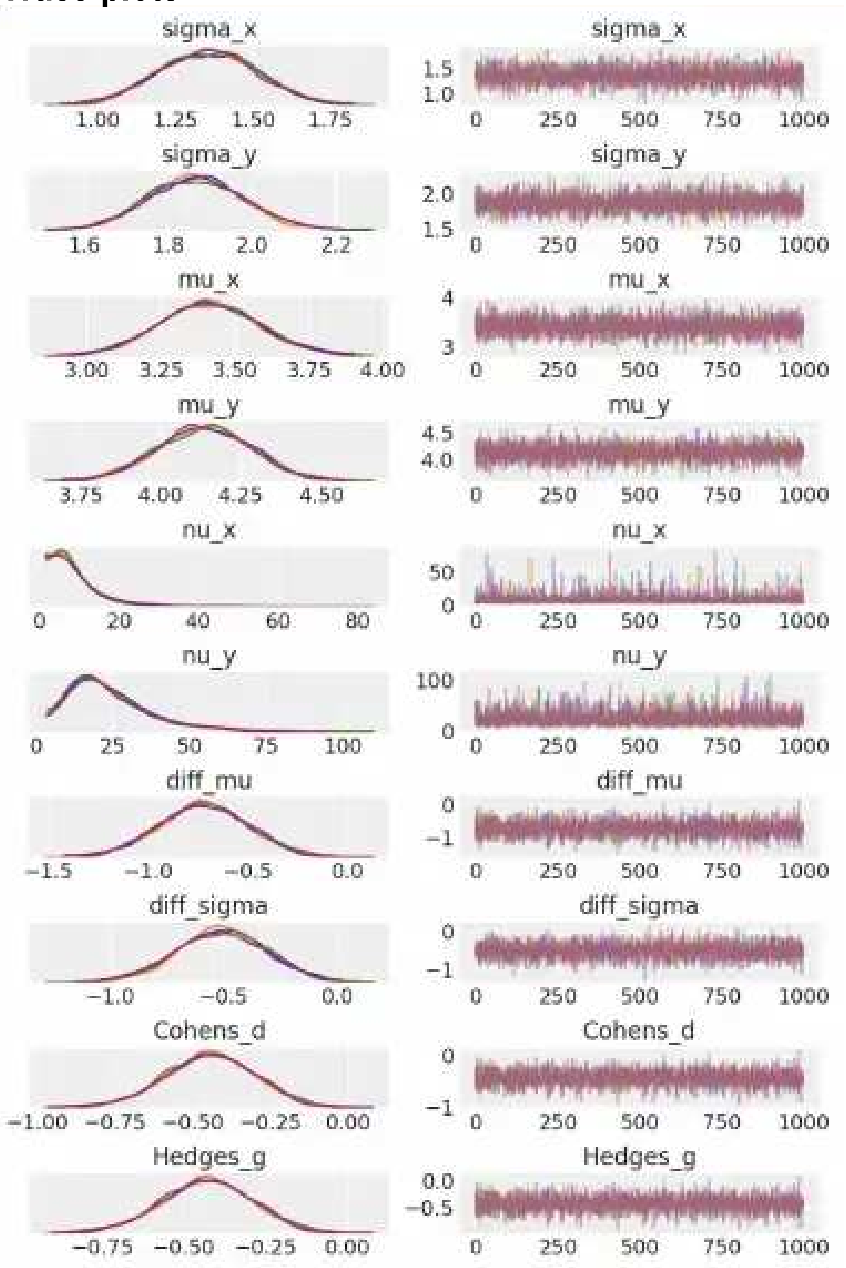
```
              mean    sd  mcerrr  hpd 3%  hpd 97%   eff_n  rhat
sigma_x       1.35  0.16    0.00    1.07     1.65  2172.0   1.0
sigma_y       1.87  0.12    0.00    1.65     2.09  3493.0   1.0
mu_x          3.41  0.17    0.00    3.09     3.72  2648.0   1.0
mu_y          4.13  0.14    0.00    3.85     4.40  3650.0   1.0
nu_x          9.45  7.65    0.21    1.86    21.89  2498.0   1.0
nu_y         24.80 13.95    0.15    5.43    51.11  3908.0   1.0
diff_mu      -0.72  0.22    0.00   -1.14    -0.32  2807.0   1.0
diff_sigma   -0.51  0.20    0.00   -0.87    -0.13  2196.0   1.0
Cohens_d     -0.45  0.14    0.00   -0.73    -0.19  2660.0   1.0
Hedges_g     -0.43  0.14    0.00   -0.69    -0.18  2750.0   1.0
```

No Stan warnings were reported.

The critical question is whether the credible interval for the difference between the two means (diff_mu) includes or excludes zero.

In this case, the credible interval for the difference excludes zero. This means we can reject the null hypothesis of no difference in the political views of men and women in this class. From our samples, the mean female political view score was 3.41. The mean male view was 4.13 (which is more conservative).

Trace plots

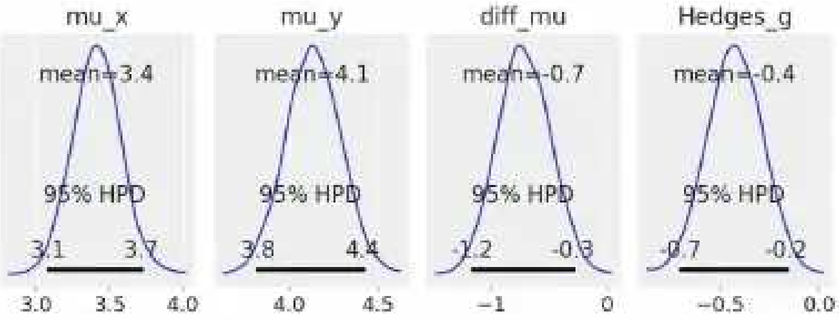


Effect size

We can also use Cohen's d or Hedges' g to look at effect size, which we encoded as a generated quantity in Stan.

In this case we have a mean effect size of -0.4 to -0.5. Using Cohen's rules of thumb, the difference between the political views for male and female students in this class was of a small effect size.

Plot key findings



Final check against classical statistics

```
tstat: test statisic
pvalue: p-value of the t-test
df: degrees of freedom used in the t-test
Welch's t-test:  (-2.801, 0.00544, 276.1519)
Student's t-test:  (-2.700, 0.00733, 296.0)
```

In Welch's and Student's t-Test, the critical value is the p-value (the middle value in the set above). Here the p-value is below 0.05: our rejection threshold. Therefore, we can reject the null hypothesis (of no difference in the mean political view of male and female students), and we can accept the alternative hypothesis that there is a difference in the mean views of students by gender.

Other bits and pieces

Parameter initialisation
The default initialisation of parameters in Stan, where Stan starts exploring, are uniform random numbers between -2 and +2. This usually, but not always, works well. Complex models, models being conditioned on a lot of data, and parameter values far from the default initialisation may converge faster (or generate fewer diagnostics) with bespoke initialisation.

For example, if I want to initialise a matix parameter centre_track to (say) 100, I could do the following:

```
ct_init = np.full([n_rows, n_cols], 100.0)
def initfun():
    return dict(centre_track=ct_init)
fit = sm.sampling(data=data, iter=iterations,
    chains=chains, init=initfun)
```

Two things to keep in mind

First, the results from Stan do not come to us as single numbers or an algebraic equation. Someone (and I am not sure who) once said: "Bayesian inference is hard because integration is hard". Most probability models translate to complex multi-dimensional integral equations that cannot be solved algebraically. Stan uses a numeric computational technique known as Hamiltonian Monte Carlo integration to solve these equations. It is this integration method that takes the time when using Stan.

The results data from Stan comprise (typically) hundreds or thousands, (or perhaps even tens of thousands) of random samples from a probability distribution for each parameter in our model. Probability distributions have the feature that the area under the curve sums to one. From the many samples for each parameter, we can use (for example) a kernel density estimate to visualise it.

Second, Hamiltonian Monte Carlo can be a little temperamental. Lots of things need to be checked before you can be confident in the result from a model run. These days, Stan is reasonably communicative when something may not be right. But you need to look. Don't believe Stan without checking the diagnostics.

So why use Stan? Because Hamiltonian Monte Carlo is a very powerful tool. It usually works. And, it can find the answer to problems where earlier Monte Carlo algorithms (Metropolis-Hastings and Gibbs Sampling) failed or took an unreasonable amount of time. Nonetheless, with the good comes the occasional pain of re-parameterising a model or re-scale he data to avoid diagnostics.

More information

For more on Hamiltonian Monte Carlo:

- https://arxiv.org/pdf/1701.02434.pdf

For more on Stan:

- http://mc-stan.org/users/documentation/
- https://www.jstatsoft.org/article/view/v076i01

For more on PyStan:

- https://pystan.readthedocs.io/en/latest/

For more on ArviZ:

- https://arviz-devs.github.io/arviz/api.html

For more on taming divergences in Stan models:

- https://dev.to/martinmodrak/taming-divergences-in-stan-models-5762