Baseball

A report created using bibat version 0.3.0

Teddy Groves

Table of contents

I used to do a lot of statistical analyses of sports data where there was a latent parameter for the player's ability. You can see an example here.

It was a natural choice to use a Bayesian hierarchical model where the abilities have a location/scale type distribution with support on the whole real line, and in fact this worked pretty well! You would see the kind of players at the top and bottom of the ability scale that you would expect.

Still, there were a few problems. In particular the data were typically unbalanced because better players tended to produce more data than worse players. The result of this was that my models would often inappropriately think the bad players were like the good players: they would not only tend to be too certain about the abilities of low-data players, but also be biased, thinking that these players are probably a bit better than they actually are. I never came up with a good way to solve this problem, despite trying a lot of things!

Even though I don't work with sports data very much any more, these issues still lingered in my mind, so when I read this great case study about geomagnetic storms it gave me an idea for yet another potential solution.

The idea was this: just as data about intense storms tell us about a tail of the bigger solar magnetism distribution, maybe data about professional sportspeople is best thought about as coming from a tail of the general sports ability distribution. If so, perhaps something like the generalised pareto distribution might be better than the bog-standard normal distribution for describing the pros' abilities.

I thought I'd test this out with some sports data, and luckily there is a really nice baseball example on the Stan case studies website, complete with data from the 2006 Major league season. After I posted some early results on the Stan discourse forum, other users suggested that it might be interesting to model similar data from different seasons. This data can now be found quite easily on the baseball databank.

With a few datasets and statistical models to consider, the full analysis looked like it would be too big to fit in a single file, so this seemed like a job for a batteries-included Bayesian analysis template like bibat.

The rest of this vignette describes how I used bibat to (relatively) painlessly see if my generalised Pareto distribution idea would work.

Check out the full analysis for all the details.

Setup

First I installed copier in my global Python 3.12 environment with this command:

```
> pipx install copier
```

Next I ran bibat's wizard like this:

```
  teddygroves  copier copy gh:teddygroves/bibat baseballl
 Name of your project
   Baseball
 Name of your project, with no spaces (used for venv and package names)
   baseball
 A short description of the project.
   Is the generalised Pareto distribution good for modelling latent hitting
ability?
 Your name (or your organization/company/team)
   Teddy Groves
 Author email (will be included in pyproject.toml)
   groves.teddy@gmail.com
 Code of conduct contact
   groves.teddy@gmail.com
 open_source_license
   MIT
 docs_format
   Quarto
 create_dotgithub_directory (bool)
   No
```

After I answered the wizard's questions bibat creted a new folder called `baseball` that looked like this:

```
  teddygroves  tree baseball
baseball
  CODE_OF_CONDUCT.md
  LICENSE
  Makefile
  README.md
  data
```

```
    raw
        raw_measurements.csv
        readme.md
docs
    bibliography.bib
    img
        example.png
        readme.md
    report.qmd
inferences
    fake_interaction
        config.toml
    interaction
        config.toml
    no_interaction
        config.toml
notebooks
    investigate.ipynb
plots
    readme.md
pyproject.toml
src
    __init__.py
    data_preparation.py
    fitting.py
    stan
        custom_functions.stan
        multilevel-linear-regression.stan
        readme.md
    stan_input_functions.py
tests
    test_integration
        test_data_preparation.py
```

This folder implements bibat's example analysis, which compares linear regression models with different design matrices. To check that everything was working correctly I ran the analysis like this:

```
$ cd baseball
$ make analysis
```

This ran without errors, running some data preparation functions and then analysing the data with Stan in a few configurations.

Getting raw data

To fetch raw data from the internet, I wrote a new script in the file `baseball/fetch_data.py`:

```
```

To get the files I ran the script:

```
(baseball)    baseball  python src/fetch_data.py --verbose
/Users/tedgro/repos/teddygroves/baseball/src/fetch_data.py:6: DeprecationWarning:
Pyarrow will become a required dependency of pandas in the next major release of pandas (pandas
(to allow more performant data types, such as the Arrow string type, and better interoperabili
but was not found to be installed on your system.
If this would cause problems for you,
please provide us feedback at https://github.com/pandas-dev/pandas/issues/54466

  import pandas as pd
INFO:root:Fetching 2006 data from https://raw.githubusercontent.com/stan-dev/example-models/mas
INFO:root:Writing 2006 data to data/raw/2006.csv
INFO:root:Fetching bdb-main data from https://raw.githubusercontent.com/cbwinslow/baseballdatal
INFO:root:Writing bdb-main data to data/raw/bdb-main.csv
INFO:root:Fetching bdb-post data from https://raw.githubusercontent.com/cbwinslow/baseballdatal
INFO:root:Writing bdb-post data to data/raw/bdb-post.csv
INFO:root:Fetching bdb-apps data from https://raw.githubusercontent.com/cbwinslow/baseballdatal
INFO:root:Writing bdb-apps data to data/raw/bdb-apps.csv
```

Finally, I removed the example analysis's raw data:

```
> rm data/raw/raw_measurements.csv
```

Data Preparation

The example project already has some data preparation code in the file `src/data_preparation.py`: I mostly just had to adapt what was already there.

The first data preparation step was to decide what prepared data should look like for my analysis and represent this definition using a subclass of bibat's `PreparedData` Pydantic `BaseModel`. Looking at the example analysis's `ExamplePreparedData` class, I could see that it already had the components that I needed, namely a name, a dictionary of coordinates and a table of measurements:

There are a few interesting things about this model. First, it uses some specialised types from bibat's `util` module, namely `CoordDict` and `DfInPydanticModel`. The latter is quite handy as it ensures that the table of measurements can be saved to json and then safely read back again. Second, the measurement table has a validator that refers to a pandera `DataFrameModel` called `ExampleMeasurementsDF` that defines what a measurement table should be like. It is defined just before `ExamplePreparedData`:

I only had to change a few things to make new definitions for my analysis:

The next step is to write functions that return `BaseballPreparedData` objects. In this case I wrote a couple of data preparation functions: `prepare_data_2006` and `prepare_data_bdb`:

To take into account the inconsistency between my two raw data sources, I first had to change the variable RAW_DATA_FILES:

Next I changed the `prepare_data` function to handle the two different data sources.

To finish off the data preparation step I deleted any unused code from the example analysis, and updated the function `load_prepared_data`'s signature:

To check that all this worked, I ran the data preparation script manually:[1]

```
> source .venv/bin/activate
(baseball) > python baseball/data_preparation.py
```

Now the folder `data/prepared/` contained json files `2006.json` and `bdb.json` that looked like this:

```
(baseball) > jq . data/prepared/bdb.json | head -n 10
{
  "name": "bdb",
  "coords": {
    "player_season": [
      "abreujo022017",
      "abreujo022018",
      "abreujo022019",
      "abreujo022020",
      "abreujo022021",
      "acunaro012018",
```

[1] I could also have just run `make analysis` again. This would have caused an error on the step after prepare_data.py, which is fine!

Specifying statistical models

I wanted to test two statistical models: one with the modelling the distribution of per-player logit-scale at-bat success rates as a normal distribution with unknown mean and standard deviation, and another where the same logit-scale rates have a generalised Pareto distribution.

So, given a table of N player profiles, with each player has y successes out of K at-bats and an unknown latent success rate α, I wanted to use this measurement model:

$$y \sim \text{binomial logit}(K, \alpha)$$

In the generalised Pareto model I would give the αs this prior model, with the hyperparameter min α assumed to be known exactly and k and σ given prior distributions that put the αs in the generally plausible range of between roughly 0.1 and 0.4.

$$\alpha \sim GPareto(\text{min } \alpha, k, \sigma)$$

In the normal model I would use a standard hierarchical regression model with an effect for the log-scale number of at-bats to attempt to explicitly model the tendency of players with more appearances to be better:

$$\alpha \sim Normal(\mu + b_K \cdot \ln K, \tau)$$

Again I would choose priors for the hyperparameters that put most of the alphas between 0.1 and 0.4.

To implement these models using Stan I first added the following function to the file `custom_functions.stan`. This was simply copied from [the relevant part of the geomagnetic storms case study]().

```
real gpareto_lpdf(vector y, real ymin, real k, real sigma) {
  // generalised Pareto log pdf
  int N = rows(y);
  real inv_k = inv(k);
  if (k<0 && max(y-ymin)/sigma > -inv_k)
```

```
    reject("k<0 and max(y-ymin)/sigma > -1/k; found k, sigma =", k, ", ", sigma);
  if (sigma<=0)
    reject("sigma<=0; found sigma =", sigma);
  if (fabs(k) > 1e-15)
    return -(1+inv_k)*sum(log1p((y-ymin) * (k/sigma))) -N*log(sigma);
  else
    return -sum(y-ymin)/sigma -N*log(sigma); // limit k->0
}
```

Next I wrote a file `gpareto.stan`:

Finally I wrote a file `normal.stan`:

Generating Stan inputs

Next I needed to tell the analysis how to turn some prepared data into a dictionary that can be used as input for Stan. The example analysis includes some functions that do this in the file `src/stan_input_functions.py`.

I followed the examples to create similar functions for my two Stan models:[2]

[2] Note that this code uses the scipy function `logit`, which it imported like this: `from scipy.special import logit`

The function has a handy decorator called `returns_stan_input` that uses the power of stanio to convert a dictionary to a json serialisable, Stan-friendly form. Thanks to this decorator I didn't have to spend time appending `.values.tolist()` to the inputs that came from pandas objects.

I now had functions that turned my prepared data into input for either of my Stan models. But why stop there? It can also be useful to generate fake input data, by running a model in simulation mode using hardcoded parameter values. Here are some functions that do this for both of my models: [3]

[3] These functions require some more imports: `numpy` and `scipy.special.expit`

Configuring fitting

To tell my analysis how to run inferences, I had to edit the file `src/ fitting.py`. The only changes I made from the example analysis were to remove the `kfold` fitting mode, which I didn't need to use in this analysis, and to add my Stan input functions to the `LOCAL_FUNCTIONS` constant:

Specifying inferences

Now all the building blocks for making statistical inferences – raw data, data preparation rules, statistical models, recipes for turning prepared data into model inputs and a procedure for running inferences – were in place. The last step before actually running my analysis was to write down how put the blocks together. Bibat has another concept for this, called 'inferences'.

An inference in bibat is a folder containing a special file called `config.toml`. This file sets out what inferences you want to make: which statistical model, which prepared data function, which Stan input function, which parameters have which dimensions, which sampling modes to use and how to configure the sampler. The folder will later be filled up with the results of performing the specified inferences.

I started by deleting the example inferences and creating two fresh folders, leaving me with an `inferences` folder looking like this:

```
inferences
  gpareto2006
      config.toml
  normal2006
      config.toml
```

Here is the file `inferences/gpareto2006/config.toml`:

Here is the file `inferences/normal2006/config.toml`:

Note that:

- The Stan file, prepared data folder and Stan input function are referred to by strings. The analysis should raise an error if you enter a non-existing value.
- Both inferences are set to run in `prior` and `posterior` modes
- You can enter arbitrary arguments to cmdstanpy's `CmdStanModel.sample` method in the `[sample_kwargs]` table.
- You can enter mode-specific overrides in `[mode_options.<MODE>]`. This can be handy if you want to run more or fewer iterations for a certain mode.

With all these changes made I ran `make analysis` again. There were no errors until after sampling, which was expected as I hadn't yet customised the investigation code, and I saw messages indicating that Stan had run. I also found that the `inferences` subfolders had been populated:

```
inferences
  gpareto2006
      config.toml
      idata
  normal2006
       config.toml
       idata
```

Investigating the inferences

Now that the inferences are ready it's time to check them out.
Bibat provides a Jupyter notebook at `notebooks/investigate.ipynb`

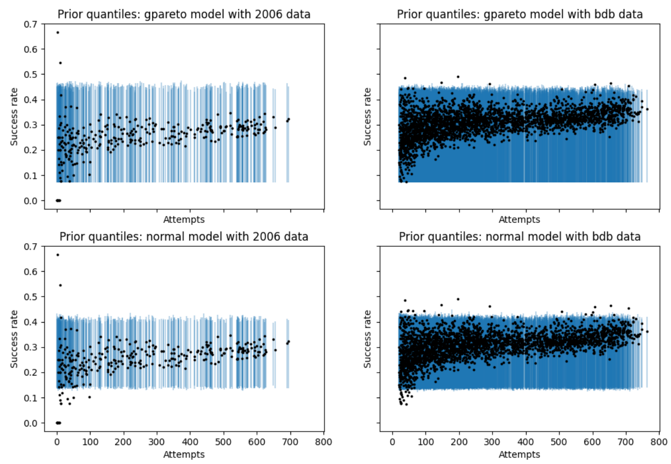
for exactly this purpose. The notebook's main job is to create plots and save them in the `plots` directory when it is executed with the command `jupyter execute notebooks/investigate.ipynb`, which is the final step in the chain of commands that is triggered by `make analysis`.

A notebook is arguably a nicer home for code that creates plots than a plain python script because it allows for literate documentation and an iterative workflow. A notebook makes it easy to, for example, add some code to change the scale of a plot, execute the code and see the new results, then update the relevant documentation all in the same place.

The code from the example analysis's notebook for loading `InferenceData` was reusable with a few tweaks to avoid missing file errors, so I kept it. On the other hand, I wanted to make some different plots from the ones in the example analysis, including some that required loading prepared data. To check out everything I did, see here.

Choosing priors using push-forward calibration

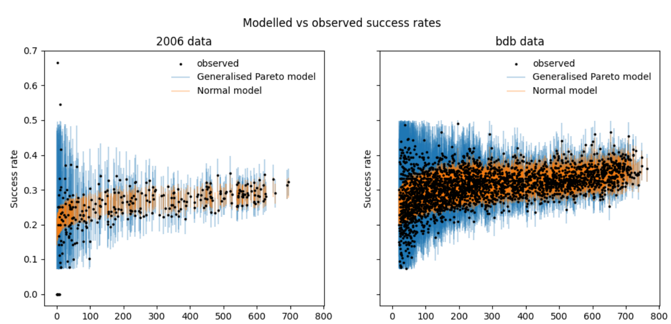
The trickiest thing about my analysis was setting prior distributions for the parameters k and σ in the generalised Pareto models. To choose some more or less plausible values I did a few prior-mode model runs and checked the distributions of the resulting `alpha` variables. I wanted to make sure that they all lay in the range corresponding to batting averages between about 0.1 and a little over 0.4. Here is a graph that shows the 1% to 99% prior quantiles for each player's latent success percentage in both datasets alongside their actually realised success rates.



Extending the analysis to the baseballdatabank data

To model the more recent data, all I had to do was create some new inference folders with appropriate `prepared_data` fields in their `config.toml` files. For example, here is the `config.toml` file for the `gparetobdb` inference:

After running `make analysis` one more time, I went back to the notebook `notebooks/investigate.ipynb` and made plots of both models' posterior 1%-99% success rate intervals for both datasets:



I think this is very interesting. Both models' prior distributions had similar regularisation levels, and they more or less agree about the abilities of the players with the most at-bats, both in terms of locations and the widths of the plausible intervals for the true success rate. However, the models ended up with dramatically different certainty levels about the abilities of players with few at-bats. This pattern was true both for the small 2006 dataset and the much larger baseballdatabank dataset.

Documenting the analysis

The final step was to document my analysis. To do this I edited the file `docs/report.qmd`, then ran `make docs`, which produced the very HTML document that you are probably reading now! You can find the complete `report.qmd` file here.