Modelling at-bats in baseball using the generalised Pareto distribution

A report created using bibat version 0.0.4

Teddy Groves

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 fproduce 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, the problem still haunted me, 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 that cause electricity problems 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, maybe 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])https://github.com/stan-dev/example-models/blob/master/knitr/pool-binary-trials/baseball-hits-2006.csv). After I posted some early results on the Stan discourse forum, user jd_c kindly gathered and posted even more data, covering 5 full major league baseball seasons.

With a few datasets and at least two different statistical models to consider, the full analysis looked like it would be a little too big to fit in a trivial file structure, so it seemed like a good opportunity to showcase my batteries-included Bayesian analysis template package 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 analysis's github repository for all the details.

Setup

First I installed bibat in my global Python 3.11 environment with this command:

```
$ pip install bibat
```

Next I started bibat's wizard like this:

```
bibat
```

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

```
baseball
   CODE_OF_CONDUCT.md
   LICENSE
   Makefile
   README.md
   bibat_version.txt
   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
   investigate.ipynb
   plots
       posterior_ll_comparison.png
       posterior_predictive_comparison.png
   prepare_data.py
   pyproject.toml
   requirements-tooling.txt
   requirements.txt
   sample.py
   src
       __init__.py
       data_preparation_functions.py
       inference_configuration.py
       prepared_data.py
       readme.md
       stan
           custom_functions.stan
```

```
            multilevel-linear-regression.stan
            readme.md
        stan_input_functions.py
        util.py
    tests
        test_integration
            test_data_preparation.py
        test_unit
            test_inference_configuration.py
            test_util.py
```

This folder implements bibat's example analysis - a comparison of linear regression with two different design matrices. To check that everything was working correctly I ran the analysis like this:

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

Some cogs turned, some new lines appeared in my terminal and some new files were created - nice, the setup worked!

Getting raw data

I added a line with the text `pyreadr` to the file `requirements.txt`, then added the following code to a new script called `fetch_data.py`:

```
import os
import requests
import pandas as pd
import pyreadr

URLS = {
    "2006": "https://raw.githubusercontent.com/stan-dev/example-models/master/knitr/pool-bin
    "recent": "https://discourse.mc-stan.org/uploads/short-url/cRJtia2jKxiX01ZEraHUVK7jbHI.R
}
OUT_FILES = {
    "2006": os.path.join("data", "raw", "baseball-hits-2006.csv"),
    "recent": os.path.join("data", "raw", "recent.RData")
}

if __name__ == "__main__":
    print(f"Fetching 2006 data from {URLS['2006']}")
    data_2006 = pd.read_csv(URLS['2006'], comment="#")
    print(f"Writing 2006 data to {OUT_FILES['2006']}")
    data_2006.to_csv(OUT_FILES['2006'])
    print(f"Fetching recent data from {URLS['recent']}")
    with requests.get(URLS["recent"]) as f:
        recent = f.text
    print(f"Writing to {OUT_FILES['recent']}")
    with open(OUT_FILES["recent"], "wb") as f:
        f.write(recent.encode("utf8", errors="replace"))
```

To get the files I ran the script:

```
$ source .venv/bin/activate
$ python fetch_data.py
```

Finally I removed the example analysis's raw data:

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

Preparing the data

The first step in preparing data is to decide what prepared data looks like for the purposes of our analysis. Bibat provides dataclasses called `PreparedData` and `MeasurementsDF` in the file `src/prepared_data.py` which can help get us started with this.

As it happens, prepared data looks very similar in our analysis and the example. All we need to do is change the `MeasurementsDF` definition a little[1]:

[1] note that this class uses pandera, a handy library for defining what a pandas dataframe should look like

```python
from typing import Optional
import pandera as pa

# ...

class MeasurementsDF(pa.SchemaModel):
    """A PreparedData should have a measurements dataframe like this.

    Other columns are also allowed!
    """

    player: pa.typing.Series[str]
    season: pa.typing.Series[str]
    n_attempt: pa.typing.Series[int] = pa.Field(ge = 1)
    n_success: pa.typing.Series[int] = pa.Field(ge = 0)
```

Next we can write some functions that create `PreparedData` objects. These should live in the file `data_preparation_functions.py`. In this case we write a couple of data preparation functions: `prepare_data_2006` and `prepare_data_recent`:

```python
"""Provides functions prepare_data_x.

These functions should take in a dataframe of measurements and return a
PreparedData object.

"""

import pandas as pd
from src.prepared_data import PreparedData


def prepare_data_2006(measurements_raw: pd.DataFrame) -> PreparedData:
    """Prepare the 2006 data."""
    measurements = (
        measurements_raw
        .rename(columns={"K": "n_attempt", "y": "n_success"})
        .assign(
            season="2006",
            player=lambda df: [f"2006-player-{i+1}" for i in range(len(df))]
        )
    )
    return PreparedData(
        name="2006",
        coords={
            "player": measurements["player"].tolist(),
            "season": measurements["season"].tolist()
        },
        measurements=measurements,
    )


def prepare_data_recent(measurements_raw: pd.DataFrame) -> PreparedData:
    """Prepare the recent data.

    There are a few substantive data choices here.

    First, the function excludes players who have a '1' in their position as
    these are likely pitchers, as well as players with fewer than 20 at bats.

    Second, the function defines a successes and attempts according to the
    'on-base percentage' metric, so a success is a time when a player got a hit,
    a base on ball/walk or a hit-by-pitch and an attempt is an at-bat or a
    base-on-ball/walk or a hit-by-pitch or a sacrifice fly. This could have
    alternatively been calculated as just hits divided by at-bats, but my
    understanding is that this method underrates players who are good at getting
    walks.

    """
    def filter_batters(df: pd.DataFrame):
        return (
            (df["AB"] >= 20)
            & ~df["Pos.Summary"].astype(str).str.contains("1")
        )
    measurements = (
        measurements_raw
        .rename(columns={"Name.additional": "player", "year": "season"})
        .assign(
            season=lambda df: df["season"].astype(str),
            n_attempt=lambda df: df[["AB", "BB", "HBP", "SF"]].fillna(0).sum(
                axis=1).astype(int),
            n_success=lambda df: (
                df[["H", "BB", "HBP"]].fillna(0).sum(axis=1).astype(int)
            )
        )
        .loc[filter_batters, ["player", "season", "n_attempt", "n_success"]]
        .copy()
    )
    return PreparedData(
        name="recent",
        coords={
            "player": measurements["player"].tolist(),
            "season": measurements["season"].tolist()
        },
        measurements=measurements,
    )
```

Finally we need to update `prepare_data.py`, the script that runs the data preparation functions. Again there isn't much to change from the example analysis.

```
"""Read the data in RAW_DIR and save prepared data to PREPARED_DIR."""

import os

import pandas as pd
from src import data_preparation_functions
from src.prepared_data import write_prepared_data

DATA_PREPARATION_FUNCTIONS_TO_RUN = {
    "2006": data_preparation_functions.prepare_data_2006,
    "recent": data_preparation_functions.prepare_data_recent,
}

DATA_DIR = os.path.join(os.path.dirname(__file__), "data")
RAW_DIR = os.path.join(DATA_DIR, "raw")
PREPARED_DIR = os.path.join(DATA_DIR, "prepared")
```

```
RAW_DATA_FILES = {
    "2006": os.path.join(RAW_DIR, "baseball-hits-2006.csv"),
    "recent": os.path.join(RAW_DIR, "recent.csv"),
}


def main():
    """Save prepared data in the PREPARED_DIR."""
    print("Reading raw data...")
    raw_data = {
        k: pd.read_csv(v, index_col=None) for k, v in RAW_DATA_FILES.items()
    }
    print("Preparing data...")
    for name, dpf in DATA_PREPARATION_FUNCTIONS_TO_RUN.items():
        print(f"Running data preparation function {dpf.__name__}...")
        prepared_data = dpf(raw_data[name])
        output_dir = os.path.join(PREPARED_DIR, prepared_data.name)
        print(f"\twriting files to {output_dir}")
        if not os.path.exists(PREPARED_DIR):
            os.mkdir(PREPARED_DIR)
        write_prepared_data(prepared_data, output_dir)


if __name__ == "__main__":
    main()
```

To check that all this works, we can run the script `prepare_data.py` manually or using `make analysis`. Now if we look in the file `data/prepared/recent/measurements.csv` we should see some lines that look like this:

```
,player,season,n_attempt,n_success
0,abreujo02,2018,553,180
1,acunaro01,2018,487,178
3,adamewi01,2018,322,112
6,adamsla01,2018,29,10
7,adamsma01,2018,337,104
8,adamsma01,2018,277,92
9,adamsma01,2018,60,12
```

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 distbution with unknown mean and standard deviation, and another where the same logit-scale rates have a generalised pareto distribution.

So, genve 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-reg.stan`:

```
functions {
#include custom_functions.stan
}
data {
  int<lower=0> N; // items
  array[N] int<lower=0> K; // trials
  array[N] int<lower=0> y; // successes
  real min_alpha; // noone worse than this would be in the dataset
  real max_alpha;
}
parameters {
  real<lower=0> sigma; // scale parameter of generalised pareto distribution
  real<lower=-sigma/(max_alpha-min_alpha)> k; // shape parameter of generalised pareto distr
  vector<lower=min_alpha>[N] alpha; // success log-odds
}
model {
  sigma ~ normal(0, 1); // hyperprior
  alpha ~ gpareto(min_alpha, k, sigma); // prior (hierarchical)
  alpha ~ normal(-1.38, 0.4);  // prob between 0.1 and 0.4 on prob scale
  y ~ binomial_logit(K, alpha); // likelihood
  // note no explicit prior for k
}
```

Finally I wrote a file `normal.stan`:

```
data {
  int<lower=0> N; // items
  array[N] int<lower=0> K; // trials
  array[N] int<lower=0> y; // successes
}
transformed data {
  vector[N] log_K = log(to_vector(K));
  vector[N] log_K_std = (log_K - mean(log_K)) / sd(log_K);
}
parameters {
  real mu; // population mean of success log-odds
  real<lower=0> tau; // population sd of success log-odds
  real b_K;
  vector[N] alpha_std; // success log-odds (standardized)
}
model {
  a_K ~ normal(0, 0.1);
  mu ~ normal(-1, 1); // hyperprior
  tau ~ normal(0, 1); // hyperprior
  alpha_std ~ normal(0, 1); // prior (hierarchical)
  y ~ binomial_logit(K, mu + b_K * log_K_std + tau * alpha_std); // likelihood
}
generated quantities {
  vector[N] alpha = mu + b_K * log_K_std + tau * alpha_std;
}
```

Generating Stan inputs

Next we need to tell our analysis how to turn some prepared data into a dictionary that can be used as input for Stan. Bibat assumes that this task is handled by functions that live in the file `src/stan_input_functions.py`. You can write as many Stan input functions as you like and choose which one to run for any given inference.

We can start by defining some Stan input functions that pass arbitary prepared data on to each of the models:

```
def get_stan_input_normal(ppd: PreparedData) -> Dict:
    """General function for creating a Stan input."""
    return {
        "N": len(ppd.measurements),
        "K": ppd.measurements["n_attempt"].values,
        "y": ppd.measurements["n_success"].values,
    }


def get_stan_input_gpareto(ppd: PreparedData) -> Dict:
    """General function for creating a Stan input."""
    return {
        "N": len(ppd.measurements),
        "K": ppd.measurements["n_attempt"].values,
        "y": ppd.measurements["n_success"].values,
        "min_alpha": 0.1,
    }
```

But why stop there? It can also be useful to generate Stan inputs consistently with a model, based on some hardcoded hyperparameter values. Here are some functions that do this for both of our models:

```
def get_stan_input_normal_fake(ppd: PreparedData) -> Dict:
    """Generate fake Stan input consistent with the normal model."""
    N = len(ppd.measurements)
    rng = np.random.default_rng(seed=1234)
    true_param_values = {
        "mu": -1.098,  # approx logit(0.25)
        "tau": 0.18,   # 2sds is 0.19 to 0.32 batting average
        "b_K": 0.04,    # slight effect of more attempts
        "alpha_std": rng.random.normal(loc=0, scale=1, size=N)
    }
    K = ppd.measurements["n_attempt"].values
    log_K_std = (np.log(K) - np.log(K).mean()) / np.log(K).std()
    alpha = (
        true_param_values["mu"]
        + true_param_values["b_K"] * log_K_std
        + true_param_values["tau"] * true_param_values["alpha_std"]
    )
    y = rng.random.binomial(K, expit(alpha))
    return {"N": N, "K": K, "y": y} | true_param_values


def get_stan_input_gpareto_fake(ppd: PreparedData) -> Dict:
    """Generate fake Stan input consistent with the gpareto model."""
    N = len(ppd.measurements)
    K = ppd.measurements["n_attempt"].values
    min_alpha = 0.1
    rng = np.random.default_rng(seed=1234)
    true_param_values = {
        "sigma": -1.098,
        "k": 0.18,
        "alpha":gpareto_rvs(rng, N, min_alpha, k, sigma)
    }
    y = rng.binomial(k, expit(alpha))
    return {"N": N, "K": k, "y": y, "min_alpha": min_alpha} | true_param_values


def gpareto_rvs(
        rng: np.random.Generator,
        size: int,
        mu: float,
        k: float,
        sigma: float
):
    """Generate random numbers from a generalised pareto distribution.

    See https://en.wikipedia.org/wiki/Generalized_Pareto_distribution for
    source.

    """
    U = rng.uniform(size)
    if k == 0:
        return mu - sigma * np.log(U)
    else:
        return mu + (sigma * (U ** -k) - 1) / sigma
```

Choosing priors using push-forward calibration

Generating inferences

Analysing the results