Parameterization choices for hierarchical models: centered, non-centered

Overview

This note expands on the section in Bob Carpenter's most excellent case study Hierarchical Partial Pooling for Repeated Binary Trials which introduces the non-centered parameterization of a hierarchical model.

Hierarchical models provide partial pooling of information across parameters according to group membership. The hierarchical model provides group-level parameters which influence the fixed-effects parameters (individual distributions on the group members). For datasets where the groups have relatively few members, MCMC samplers cannot easily explore the resulting sampling density. The non-centered parameterization mitigates this problem by decoupling the group-level and fixed-effects parameters in the sampling distribution.

The case study dataset is taken from baseball, consisting of the number of hits and at-bats for a set of Major League Baseball players. While individual players have differing batting abilities, they are taken from the population of MLB baseball players. Therefore, it makes sense to build a hierarchical model of player ability.

The data consists of `N` observations `y`, where each observation y_n is the number of successes for $player_n$ in `K` trials. The dataset is small: 18 players ($N = 18$), 45 at-bats ($K = 45$). The model estimates θ_n, each player's chance of success for an at-bat. ($\theta * 1000$ is a player's "batting average".) It does so by recasting the problem in terms of parameter α, a player's log-odds of success. The hierarchical model puts a normal prior with group-level parameters μ and σ on the estimates for parameter α, which pulls the individual player estimates towards the group mean μ.

The log-odds parameterization makes it much easier to expand the model by adding more fixed effects and other multilevel effects. The change of success `theta` is computed in the model's generated quantities block as `inv_logit(alpha)`.

The centered parameterization for a hierarchical model corresponds directly to the data structure: the individual-level parameter `alpha` - a player's log-odds of success is given a prior distribution specified in terms of the group-level parameters: `alpha ~ normal(mu, sigma)`. The non-centered parameterization is recommended for hierarchical models where the groups have relatively few members. The trick is to decouple `alpha`, `mu`, and `sigma` in the sampling distribution by reparameterization. There are two ways to do this reparameterization:

- a non-centered parameterization with standard normal prior on parameter `alpha_std` and auxiliary variable `alpha`.
- a non-centered parameterization using an affine transform on parameter `alpha`.

In this note we show models for each and then plot the result.

Packages used in this notebook

We use CmdStanPy to do the model fitting and plot the results using plotnine, a ggplot2-like Python package. Pandas and NumPy are also used for data munging.

In [1]:
```
import os
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
from plotnine import *
%matplotlib inline

from cmdstanpy import CmdStanModel
```

In [2]:
```
theme_set(
  theme_grey() +
  theme(text=element_text(size=10),
        plot_title=element_text(size=14),
        axis_title_x=element_text(size=12),
        axis_title_y=element_text(size=12),
        axis_text_x=element_text(size=8),
        axis_text_y=element_text(size=8)
       )
)
```

Baseball Data: Number of hits in 45 at-bats for 18 MLB players in 1971

In [3]:
```
with open('efron-morris-75-data.tsv') as tsv_file:
    df = pd.read_csv("efron-morris-75-data.tsv", sep="\t")
df.style.hide_index().format(precision=3)
```

Out[3]:

FirstName	LastName	At-Bats	Hits	BattingAverage	RemainingAt-Bats	RemainingAverage	SeasonAt-Bats	SeasonHits	SeasonAverage
Roberto	Clemente	45	18	0.400	367	0.346	412	145	0.352
Frank	Robinson	45	17	0.378	426	0.298	471	144	0.306
Frank	Howard	45	16	0.356	521	0.276	566	160	0.283
Jay	Johnstone	45	15	0.333	275	0.222	320	76	0.238

FirstName	LastName	At-Bats	Hits	BattingAverage	RemainingAt-Bats	RemainingAverage	SeasonAt-Bats	SeasonHits	SeasonAverage
Ken	Berry	45	14	0.311	418	0.273	463	128	0.276
Jim	Spencer	45	14	0.311	466	0.270	511	140	0.274
Don	Kessinger	45	13	0.289	586	0.265	631	168	0.266
Luis	Alvarado	45	12	0.267	138	0.210	183	41	0.224
Ron	Santo	45	11	0.244	510	0.269	555	148	0.267
Ron	Swaboda	45	11	0.244	200	0.230	245	57	0.233
Rico	Petrocelli	45	10	0.222	538	0.264	583	152	0.261
Ellie	Rodriguez	45	10	0.222	186	0.226	231	52	0.225
George	Scott	45	10	0.222	435	0.303	480	142	0.296
Del	Unser	45	10	0.222	277	0.264	322	83	0.258
Billy	Williams	45	10	0.222	591	0.330	636	205	0.251
Bert	Campaneris	45	9	0.200	558	0.285	603	168	0.279
Thurman	Munson	45	8	0.178	408	0.316	453	137	0.302
Max	Alvis	45	7	0.156	70	0.200	115	21	0.183

In [4]:
```
baseball_data = {"N": df.shape[0],
                 "K": df['At-Bats'],
                 "y": df['Hits'],
                 "K_new": df['RemainingAt-Bats'],
                 "y_new": df['SeasonHits']-df['Hits']}

M = 10000  # desired number of draws from the posterior

# ggplot2 x_y plot with axis labels and optional title
def scatter_plot(df, x_lab, y_lab, title=''):
  return (ggplot(df, aes('x', 'y')) +
          geom_point(alpha=0.2) +
          xlab(x_lab) +
          ylab(y_lab) +
          ggtitle(title) +
          theme(figure_size=(8,6)))
```

The Model

The model we are interested in is a hierarchical model with a *normal prior* on the *log odds of success*. The mathematical model specification is

$$p(y_n \,|\, K_n, \alpha_n) \;=\; \text{Binomial}(y_n \,|\, K_n,\; \text{logit}^{-1}(\alpha_n))$$

with a simple normal hierarchical prior

$$p(\alpha_n \,|\, \mu, \sigma) = \text{Normal}(\alpha_n \,|\, \mu, \sigma).$$

a weakly informative hyperprior for μ

$$p(\mu) = \text{Normal}(\mu \,|\, -1, 1),$$

and a half normal prior on σ

$$p(\sigma) \;=\; 2\,\text{Normal}(\sigma \,|\, 0, 1) \;\propto\; \text{Normal}(\sigma \,|\, 0, 1).$$

Centered Parameterization

The Stan program `hier-logit-centered.stan` is a straightforward encoding of a hierarchical model with a normal prior on the log odds of success, but this is not the optimal way to code this model in Stan, as we will soon demonstrate.

```
parameters {
  real mu;                      // population mean of success log-odds
  real<lower=0> sigma;          // population sd of success log-odds
  vector[N] alpha;              // success log-odds
}
model {
  mu ~ normal(-1, 1);             // hyperprior
  sigma ~ normal(0, 1);           // hyperprior
  alpha ~ normal(mu, sigma);      // prior (hierarchical)
  y ~ binomial_logit(K, alpha);   // likelihood
}
```

The chance of success θ is computed as a generated quantity.

```
  generated quantities {
    vector[N] theta = inv_logit(alpha);
  }
```

In CmdStanPy, model fitting is done in two steps: first instantiate the model object from a Stan program file; then run the Stan inference algorithm, here the NUTS-HMC sampler, which returns the inferences.

We instantiate the CmdStanModel object from the Stan program file 'hier-logit-centered.stan'. By default, CmdStanPy compiles the model on object instantiation, unless there is a corresponding exe file which has a more recent timestamp than the source file. The model's `code` method returns the Stan program.

In [5]:
```
hier_logit_centered_model = CmdStanModel(stan_file='hier-logit-centered.stan')
print(hier_logit_centered_model.code())
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
data {
  int<lower=0> N; // items
  array[N] int<lower=0> K; // initial trials
  array[N] int<lower=0> y; // initial successes
}
parameters {
  real mu; // population mean of success log-odds
  real<lower=0> sigma; // population sd of success log-odds
  vector[N] alpha; // success log-odds
}
model {
  mu ~ normal(-1, 1); // hyperprior
  sigma ~ normal(0, 1); // hyperprior
  alpha ~ normal(mu, sigma); // prior (hierarchical)
  y ~ binomial_logit(K, alpha); // likelihood
}
generated quantities {
  vector[N] theta = inv_logit(alpha);
}
```

Next run the NUTS-HMC sampler. By default the sampler runs 4 chains. The argument `iter_sampling` specifies the *per-chain* number of sampling iterations. The defaults are 1000 warmup and 1000 sampling iterations per chain, for a sample containing a total of 4000 draws. Since $M = 10000$, we override this default. We specify the random seed for reproducibility.

In [6]:
```
fit_centered = hier_logit_centered_model.sample(
    data=baseball_data,
    iter_sampling=int(M/4),
    seed=54321)
```

```
INFO:cmdstanpy:CmdStan start processing
```

```
INFO:cmdstanpy:CmdStan done processing.
```

The variable `theta` is the per-player chance of success, i.e., `theta * 1000` is a player's batting average. The estimates range from 0.24 to 0.3, batting averages between 240 and a respectable 300, which is in line with what we know about major league baseball players.

In [7]:
```
fit_centered.summary(sig_figs=3).round(decimals=3).filter(
    regex=r'mu|sigma|theta', axis="index")
```

Out[7]:

name	Mean	MCSE	StdDev	5%	50%	95%	N_Eff	N_Eff/s	R_hat
mu	-1.030	0.002	0.094	-1.180	-1.030	-0.873	2230.0	3480.0	1.00
sigma	0.189	0.004	0.103	0.054	0.173	0.382	696.0	1080.0	1.01
theta[1]	0.298	0.001	0.045	0.239	0.291	0.382	2188.0	3407.0	1.00
theta[2]	0.292	0.001	0.041	0.237	0.286	0.370	2826.0	4401.0	1.00
theta[3]	0.287	0.001	0.039	0.233	0.282	0.360	3337.0	5198.0	1.00
theta[4]	0.282	0.001	0.038	0.228	0.278	0.350	4080.0	6356.0	1.00
theta[5]	0.277	0.001	0.036	0.224	0.273	0.342	4967.0	7737.0	1.00
theta[6]	0.277	0.001	0.037	0.222	0.273	0.344	5192.0	8087.0	1.00
theta[7]	0.271	0.000	0.036	0.216	0.269	0.332	6397.0	9965.0	1.00
theta[8]	0.266	0.000	0.034	0.211	0.265	0.324	5932.0	9239.0	1.00
theta[9]	0.261	0.000	0.035	0.204	0.260	0.318	5166.0	8047.0	1.00
theta[10]	0.260	0.000	0.034	0.204	0.261	0.316	4744.0	7390.0	1.00
theta[11]	0.255	0.001	0.034	0.198	0.256	0.310	4289.0	6681.0	1.00
theta[12]	0.255	0.001	0.034	0.197	0.257	0.310	4539.0	7070.0	1.00
theta[13]	0.255	0.001	0.035	0.195	0.256	0.310	3886.0	6053.0	1.00

name	Mean	MCSE	StdDev	5%	50%	95%	N_Eff	N_Eff/s	R_hat
theta[14]	0.255	0.001	0.035	0.196	0.256	0.310	4018.0	6259.0	1.00
theta[15]	0.255	0.001	0.034	0.198	0.256	0.309	3790.0	5904.0	1.00
theta[16]	0.250	0.001	0.035	0.189	0.252	0.304	3067.0	4777.0	1.00
theta[17]	0.245	0.001	0.036	0.181	0.248	0.299	2293.0	3571.0	1.00
theta[18]	0.241	0.001	0.037	0.174	0.244	0.296	1877.0	2924.0	1.00

The reported Eff values for `sigma` are low and the R_hat value is above 1. CmdStan's `diagnose` method indicates that this model had problems fitting the data.

In [8]:
```
print(fit_centered.diagnose())
```

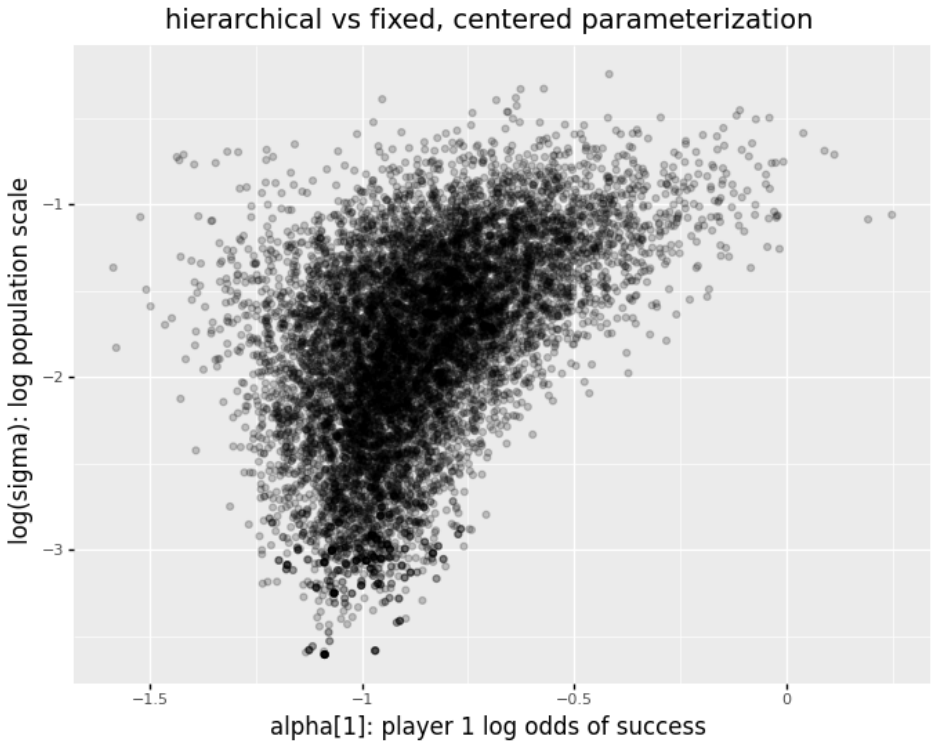
```
Processing csv files: /var/folders/db/4jnggnf549s42z50bd61jskm0000gq/T/tmp8rm98ht9/hier-logit-centered-20220128200245_1.cs
v, /var/folders/db/4jnggnf549s42z50bd61jskm0000gq/T/tmp8rm98ht9/hier-logit-centered-20220128200245_2.csv, /var/folders/db/4
jnggnf549s42z50bd61jskm0000gq/T/tmp8rm98ht9/hier-logit-centered-20220128200245_3.csv, /var/folders/db/4jnggnf549s42z50bd61j
skm0000gq/T/tmp8rm98ht9/hier-logit-centered-20220128200245_4.csv

Checking sampler transitions treedepth.
Treedepth satisfactory for all transitions.

Checking sampler transitions for divergences.
76 of 10000 (0.76%) transitions ended with a divergence.
These divergent transitions indicate that HMC is not fully able to explore the posterior distribution.
Try increasing adapt delta closer to 1.
If this doesn't remove all divergences, try to reparameterize the model.

Checking E-BFMI - sampler transitions HMC potential energy.
The E-BFMI, 0.22, is below the nominal threshold of 0.3 which suggests that HMC may have trouble exploring the target distr
ibution.
If possible, try to reparameterize the model.

Effective sample size satisfactory.

Split R-hat values satisfactory all parameters.

Processing complete.
```

The Funnel

These diagnostics indicate that the sampler failed to fit the data and that the resulting sample is not a sample from the true posterior. The reason for this failure is that given the small amount of data, the sampler cannot properly determine how much of the observed variance in the data is individual-level variance, parameter `alpha`, or group-level variance, parameter `sigma`. The diagnostics report low ESS and poor R-hat for `sigma`.

Plotting the estimate of `alpha[1]`, the log-odds success for player 1, against `log(sigma)`, the group-level variance, provide additional evidence of the problem. This plot shows a clear funnel shape with many draws at the bottom of the neck of the funnel. This is the reason for the low EFF numbers for `sigma`. The sampler "gets stuck" at the bottom of the funnel. The algorithm tries to jump to a new point, but large jumps fall outside of the posterior density, resulting in a divergence. Small jumps fail to exit the neck of the funnel.

In [9]:
```
df_x_y = pd.DataFrame(
    data={'x': fit_centered.stan_variable('alpha')[:,0],
          'y': np.log(fit_centered.stan_variable('sigma'))
         }
)

scatter_plot(df_x_y,
         x_lab = "alpha[1]: player 1 log odds of success",
         y_lab = "log(sigma): log population scale",
         title = "hierarchical vs fixed, centered parameterization")
```



Out[9]: <ggplot: (8793149014158)>

The Non-Centered Parameterization

Instead of a hierarchical prior, the non-centered parameterization takes a standard unit normal prior for a new variable,

$$\alpha_n^{\text{std}} = \frac{\alpha_n - \mu}{\sigma}.$$

Then we can parameterize in terms of α^{std}, which has a standard-normal distribution

$$p(\alpha_n^{\text{std}}) = \text{Normal}(\alpha_n^{\text{std}} \mid 0, 1).$$

We can then define our original α as a derived quantity.

$$\alpha_n = \mu + \sigma\, \alpha_n^{\text{std}}.$$

This decouples the sampling distribution for α^{std} from μ and σ, greatly reducing their correlation in the posterior. *The sampler only knows about the model parameters*. Since the prior on parameter α is not specified in terms of parameters μ and σ, the sampler can move more freely along their axes, and therefore explore the posterior more fully. Although we decouple the parameters, we still need to share information between the group-level and individual level parameters; this is done using auxiliary variables, either transformed parameters or directly in the model block.

Non-centered parameterization using a standard normal distribution

Prior to Stan 2.19, a Stan implementation directly encoded the above reparameterization. This requires 3 changes to the centered parameterization:

- In the parameters block, declaring a parameter `alpha_std` (instead of parameter `alpha`). This name implies that it will have a standard normal distribution.
- In the transformed parameters block define variable `alpha` as `mu + sigma * alpha_std`.
- In the model block we put a standard normal prior on `alpha_std`, which decouples the sampling distribution of `alpha_std` from `mu` and `sigma`.

The Stan program "hier-logit-nc-std-norm.stan" follows this pattern.

```
data {
  int<lower=0> N; // items
  array[N] int<lower=0> K; // initial trials
  array[N] int<lower=0> y; // initial successes
}
parameters {
  real mu; // population mean of success log-odds
  real<lower=0> sigma; // population sd of success log-odds
  vector[N] alpha_std; // success log-odds (standardized)
```

```
}
transformed parameters {
  vector[N] alpha = mu + sigma * alpha_std;
}
model {
  mu ~ normal(-1, 1); // hyperprior
  sigma ~ normal(0, 1); // hyperprior
  alpha_std ~ normal(0, 1); // prior (hierarchical)
  y ~ binomial_logit(K, alpha); // likelihood
}
generated quantities {
  vector[N] theta = inv_logit(alpha);
}
```

Non-centered parameterization using an affine transform

Since Stan version 2.19, the Stan language's affine transform construct provides a more efficient way to do this. For a real variable, the affine transform $x \mapsto \mu + \sigma * x$ with offset μ and (positive) multiplier σ is specified using a syntax like that used for upper/lower bounds, with keywords `offset` , `multiplier` . Specifying the affine transform in the parameter declaration for α^{std} eliminates the need for intermediate variables and makes it easier to see the hierarchical structure of the model.

When the parameters to the prior for σ are constants, the normalization for the half-prior (compared to the full prior) is constant and therefore does not need to be included in the notation. This only works if the parameters to the density are data or constants; if they are defined as parameters or as quantities depending on parameters, then explicit truncation is required.

The Stan program `hier-logit-nc-affine-xform.stan` uses the affine-transform syntax to specify the non-centered version of the hierarchical model with a normal prior on the log odds of success.

```
data {
  int<lower=0> N; // items
  array[N] int<lower=0> K; // initial trials
  array[N] int<lower=0> y; // initial successes
}
parameters {
  real mu; // population mean of success log-odds
  real<lower=0> sigma; // population sd of success log-odds
  vector<offset=mu, multiplier=sigma>[N] alpha; // success log-odds (standardized)
}
model {
  mu ~ normal(-1, 1); // hyperprior
  sigma ~ normal(0, 1); // hyperprior
  alpha ~ normal(mu, sigma); // prior (hierarchical)
  y ~ binomial_logit(K, alpha); // likelihood
}
generated quantities {
  vector[N] theta = inv_logit(alpha);
  vector[N] alpha_std = (alpha - mu)/sigma;
}
```

Fitting the standard normal reparameterization

The model `hier-logit-nc-std-norm.stan` fits the model using parameter `alpha_std` .

Full disclosure: the choice of random seed '54321' was far from random; this seed allows the sampler to fit the model without divergences. Other seeds may result in 1 or 2 divergences for a sample of 2500 draws.

In [10]:
```
nc_std_norm_model = CmdStanModel(stan_file='hier-logit-nc-std-norm.stan')
print(nc_std_norm_model.code())

fit_nc_std_norm = nc_std_norm_model.sample(
    data=baseball_data,
    iter_sampling=int(M/4),
    seed=54321)
```

```
INFO:cmdstanpy:compiling stan file /Users/mitzi/github/zmorris/stan_studies/offset_multiplier/hier-logit-nc-std-norm.stan t
o exe file /Users/mitzi/github/zmorris/stan_studies/offset_multiplier/hier-logit-nc-std-norm
INFO:cmdstanpy:compiled model executable: /Users/mitzi/github/zmorris/stan_studies/offset_multiplier/hier-logit-nc-std-norm
INFO:cmdstanpy:CmdStan start processing
data {
  int<lower=0> N; // items
  array[N] int<lower=0> K; // initial trials
  array[N] int<lower=0> y; // initial successes
}
parameters {
  real mu; // population mean of success log-odds
  real<lower=0> sigma; // population sd of success log-odds
  vector[N] alpha_std; // success log-odds (standardized)
}
transformed parameters {
```

```
  vector[N] alpha = mu + sigma * alpha_std;
}
model {
  mu ~ normal(-1, 1); // hyperprior
  sigma ~ normal(0, 1); // hyperprior
  alpha_std ~ normal(0, 1); // prior (hierarchical)
  y ~ binomial_logit(K, alpha); // likelihood
}
generated quantities {
  vector[N] theta = inv_logit(alpha);
}
```

```
INFO:cmdstanpy:CmdStan done processing.
```

Again, we check for problems by running CmdStan's `diagnose` method.

In [11]:
```
print(fit_nc_std_norm.diagnose())
```

```
Processing csv files: /var/folders/db/4jnggnf549s42z50bd61jskm0000gq/T/tmp8rm98ht9/hier-logit-nc-std-norm-20220128200255_1.
csv, /var/folders/db/4jnggnf549s42z50bd61jskm0000gq/T/tmp8rm98ht9/hier-logit-nc-std-norm-20220128200255_2.csv, /var/folder
s/db/4jnggnf549s42z50bd61jskm0000gq/T/tmp8rm98ht9/hier-logit-nc-std-norm-20220128200255_3.csv, /var/folders/db/4jnggnf549s4
2z50bd61jskm0000gq/T/tmp8rm98ht9/hier-logit-nc-std-norm-20220128200255_4.csv

Checking sampler transitions treedepth.
Treedepth satisfactory for all transitions.

Checking sampler transitions for divergences.
No divergent transitions found.

Checking E-BFMI - sampler transitions HMC potential energy.
E-BFMI satisfactory.

Effective sample size satisfactory.

Split R-hat values satisfactory all parameters.

Processing complete, no problems detected.
```

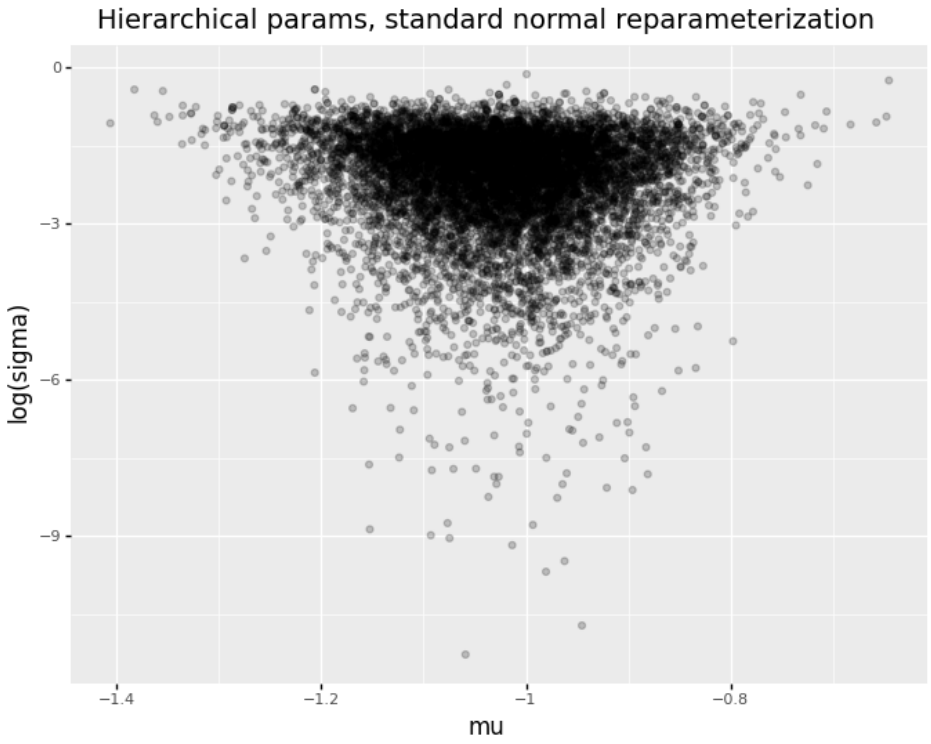
The estimates for `mu`, `sigma`, `theta` and `alpha` are roughly the same as for the centered parameterization. The non-centered parameterization results in a much larger effective sample size.

In [12]:
```
print("Centered parameterization")
print(fit_centered.summary(sig_figs=3).round(decimals=3).filter(
    ["mu", "sigma",
     "theta[1]", "theta[5]", "theta[10]", "theta[18]",
     "alpha[1]", "alpha[5]", "alpha[10]", "alpha[18]"],
    axis="index"))

print("\nNon-centered parameterization, std_normal reparameterization")
print(fit_nc_std_norm.summary(sig_figs=3).round(decimals=3).filter(
    ["mu", "sigma",
     "theta[1]", "theta[5]", "theta[10]", "theta[18]",
     "alpha[1]", "alpha[5]", "alpha[10]", "alpha[18]"],
    axis="index"))
```

```
Centered parameterization
            Mean   MCSE  StdDev     5%    50%    95%   N_Eff  N_Eff/s  R_hat
name
mu        -1.030  0.002   0.094 -1.180 -1.030 -0.873  2230.0   3480.0   1.00
sigma      0.189  0.004   0.103  0.054  0.173  0.382   696.0   1080.0   1.01
theta[1]   0.298  0.001   0.045  0.239  0.291  0.382  2188.0   3407.0   1.00
theta[5]   0.277  0.001   0.036  0.224  0.273  0.342  4967.0   7737.0   1.00
theta[10]  0.260  0.000   0.034  0.204  0.261  0.316  4744.0   7390.0   1.00
theta[18]  0.241  0.001   0.037  0.174  0.244  0.296  1877.0   2924.0   1.00
alpha[1]  -0.863  0.004   0.209 -1.160 -0.892 -0.480  2223.0   3462.0   1.00
alpha[5]  -0.968  0.003   0.180 -1.240 -0.978 -0.654  5031.0   7836.0   1.00
alpha[10] -1.050  0.003   0.181 -1.360 -1.040 -0.770  4591.0   7152.0   1.00
alpha[18] -1.160  0.005   0.211 -1.560 -1.130 -0.867  1911.0   2977.0   1.00

Non-centered parameterization, std_normal reparameterization
            Mean   MCSE  StdDev     5%    50%    95%    N_Eff  N_Eff/s  R_hat
name
mu        -1.030  0.001   0.090 -1.170 -1.030 -0.878   8040.0  10900.0    1.0
sigma      0.164  0.002   0.110  0.017  0.148  0.364   4120.0   5560.0    1.0
theta[1]   0.293  0.001   0.043  0.240  0.284  0.378   6968.0   9404.0    1.0
theta[5]   0.274  0.000   0.034  0.225  0.271  0.336   9617.0  12978.0    1.0
theta[10]  0.261  0.000   0.032  0.208  0.262  0.314  10816.0  14596.0    1.0
theta[18]  0.244  0.000   0.036  0.178  0.249  0.295   7175.0   9683.0    1.0
alpha[1]  -0.887  0.002   0.202 -1.150 -0.924 -0.496   6982.0   9422.0    1.0
alpha[5]  -0.978  0.002   0.170 -1.240 -0.989 -0.681   9709.0  13103.0    1.0
alpha[10] -1.050  0.002   0.170 -1.340 -1.040 -0.779  10581.0  14280.0    1.0
alpha[18] -1.140  0.002   0.207 -1.530 -1.100 -0.874   7012.0   9463.0    1.0
```

To consider how the reparameterization is working, we plot the posterior for the mean and log scale of the hyperprior. The prior location (μ) and scale (σ) are coupled in the posterior.

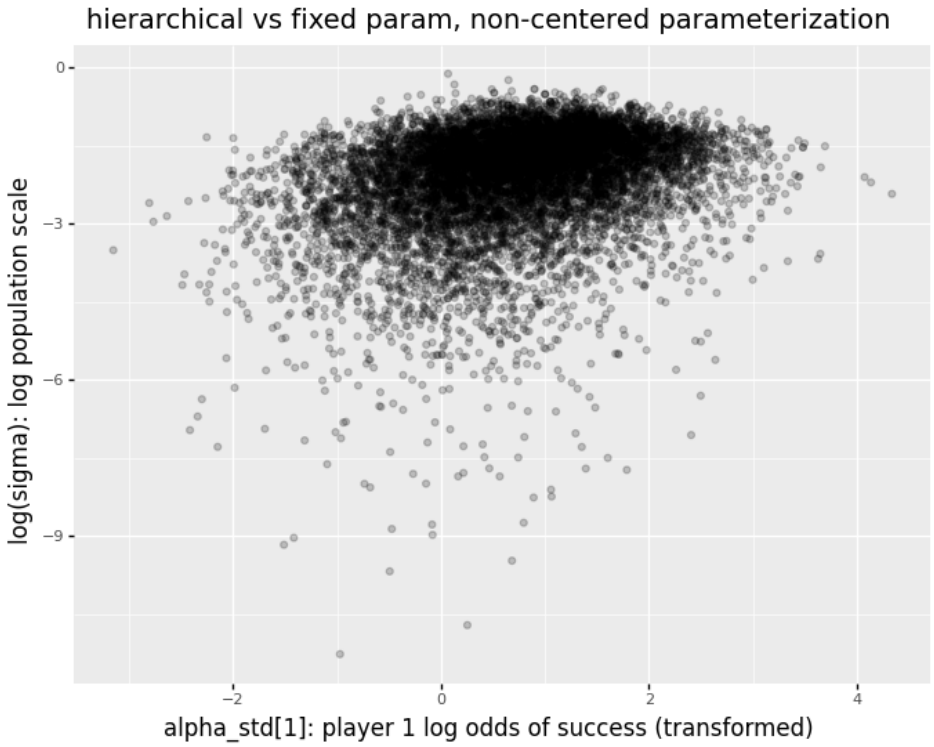
In [13]:
```
df_x_y = pd.DataFrame(data={'x': fit_nc_std_norm.stan_variable('mu'),
                            'y': np.log(fit_nc_std_norm.stan_variable('sigma'))})

scatter_plot(df_x_y,
         x_lab = "mu",
         y_lab = "log(sigma)",
         title = "Hierarchical params, standard normal reparameterization")
```



Out[13]: <ggplot: (8793216836964)>

Now when we plot the sample values for log scale and the first transformed parameter, `alpha_std[1]` , the range of both the X and Y axis are much wider. There is a diffuse set of points in the bottom half of the plot, not many points at the bottom of the Y axis. This indicates that the sampler has been able to properly explore the posterior density and therefore we have a valid sample from the posterior. As log `sigma` approaches zero the plot has a long right-hand tail.

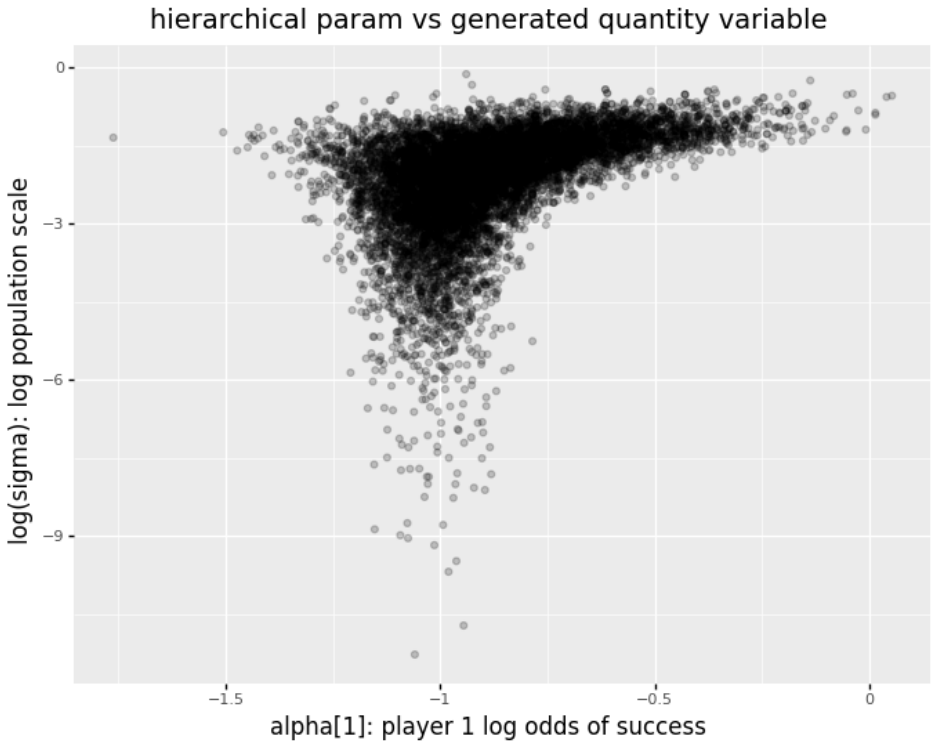
In [14]:
```
df_x_y = pd.DataFrame(
    data={'x': fit_nc_std_norm.stan_variable('alpha_std')[: , 0],
          'y': np.log(fit_nc_std_norm.stan_variable('sigma'))}
)

scatter_plot(df_x_y,
         x_lab = "alpha_std[1]: player 1 log odds of success (transformed)",
         y_lab = "log(sigma): log population scale",
         title = "hierarchical vs fixed param, non-centered parameterization")
```



Out[14]: <ggplot: (8793216885119)>

We can also plot the value for the generated quantities variable `alpha[1]` against `log(sigma)` and compare it to the first plot from the centered parameterization. We recover the funnel shape, but now the Y axis ranges from (-12, 0) instead of (-4, 0). As above, as log `sigma` approaches zero the plot has a long right-hand tail.

In [15]:
```
df_x_y = pd.DataFrame(
    data={'x': fit_nc_std_norm.stan_variable('alpha')[: , 0],
          'y': np.log(fit_nc_std_norm.stan_variable('sigma'))}
)

scatter_plot(df_x_y,
         x_lab = "alpha[1]: player 1 log odds of success",
         y_lab = "log(sigma): log population scale",
         title = "hierarchical param vs generated quantity variable")
```



Out[15]: <ggplot: (8793201155319)>

We still don't have enough data to determine whether or not the observed variance is hierarchical or individual-level variance. The model still provides us with an estimate for `alpha`, a player's log-odds of success at bat. Critically, because `alpha` is no longer a parameter variable, replaced by `alpha_std` in the sampling distribution, the sampler can fully explore the posterior.

Fitting the affine transform parameterization

The model `hier-logit-nc-affine-xform.stan` looks just like the centered parameterization, with the exception that parameter `alpha` is defined with `<offset = mu, multiplier = sigma>`.

To show that the affine transform reparameterization and the standard normal reparameterization are equivalent we fit the model to the data and plot the results.

In [16]:
```
nc_affine_xform_model = CmdStanModel(stan_file='hier-logit-nc-affine-xform.stan')
print(nc_affine_xform_model.code())
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
data {
  int<lower=0> N; // items
  array[N] int<lower=0> K; // initial trials
  array[N] int<lower=0> y; // initial successes
}
parameters {
  real mu; // population mean of success log-odds
  real<lower=0> sigma; // population sd of success log-odds
  vector<offset=mu, multiplier=sigma>[N] alpha; // success log-odds (standardized)
}
model {
  mu ~ normal(-1, 1); // hyperprior
  sigma ~ normal(0, 1); // hyperprior
  alpha ~ normal(mu, sigma); // prior (hierarchical)
  y ~ binomial_logit(K, alpha); // likelihood
}
generated quantities {
  vector[N] theta = inv_logit(alpha);
  vector[N] alpha_std = (alpha - mu)/sigma;
}
```

In [17]:
```
fit_nc_affine = nc_affine_xform_model.sample(
    data=baseball_data,
    iter_sampling=int(M/4),
    seed=54321)
```

```
INFO:cmdstanpy:CmdStan start processing



INFO:cmdstanpy:CmdStan done processing.
```

As usual, we check for problems by running CmdStan's diagnose method.

In [18]:
```
print(fit_nc_affine.diagnose())
```

```
Processing csv files: /var/folders/db/4jnggnf549s42z50bd61jskm0000gq/T/tmp8rm98ht9/hier-logit-nc-affine-xform-2022012820025
9_1.csv, /var/folders/db/4jnggnf549s42z50bd61jskm0000gq/T/tmp8rm98ht9/hier-logit-nc-affine-xform-20220128200259_2.csv, /va
r/folders/db/4jnggnf549s42z50bd61jskm0000gq/T/tmp8rm98ht9/hier-logit-nc-affine-xform-20220128200259_3.csv, /var/folders/db/
4jnggnf549s42z50bd61jskm0000gq/T/tmp8rm98ht9/hier-logit-nc-affine-xform-20220128200259_4.csv

Checking sampler transitions treedepth.
Treedepth satisfactory for all transitions.

Checking sampler transitions for divergences.
No divergent transitions found.

Checking E-BFMI - sampler transitions HMC potential energy.
E-BFMI satisfactory.

Effective sample size satisfactory.

Split R-hat values satisfactory all parameters.

Processing complete, no problems detected.
```

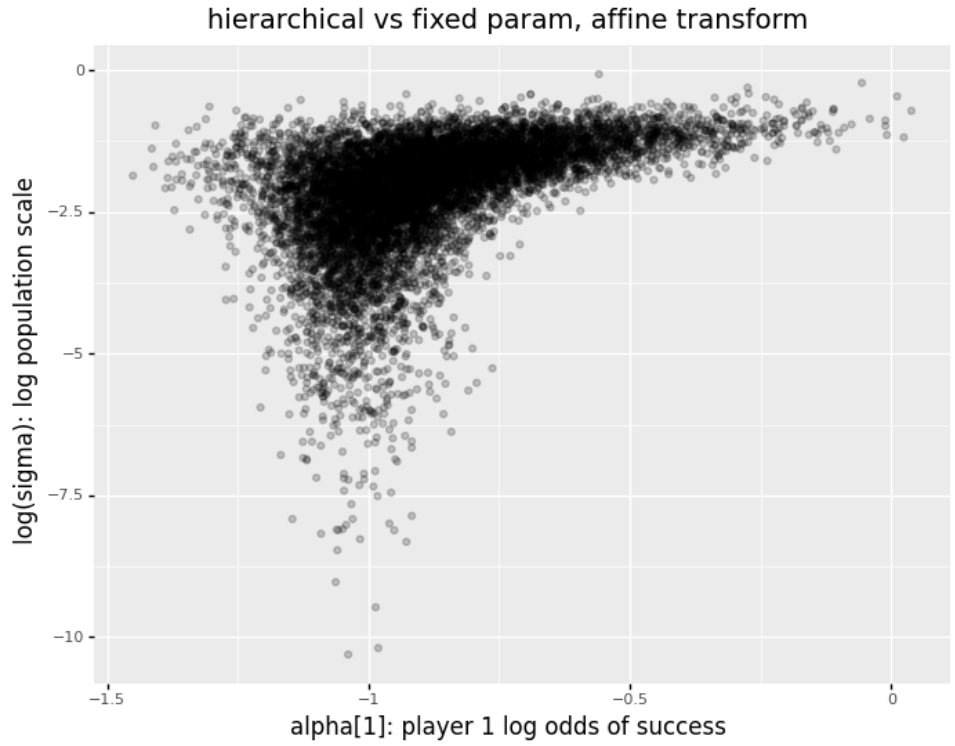
In [19]:
```
print("Centered parameterization")
print(fit_centered.summary(sig_figs=3).round(decimals=3).filter(
    ["mu", "sigma",
     "theta[1]", "theta[5]", "theta[10]", "theta[18]",
     "alpha[1]", "alpha[5]", "alpha[10]", "alpha[18]"],
    axis="index"))

print("\nNon-centered parameterization, std_normal reparameterization")
print(fit_nc_std_norm.summary(sig_figs=3).round(decimals=3).filter(
    ["mu", "sigma",
     "theta[1]", "theta[5]", "theta[10]", "theta[18]",
     "alpha[1]", "alpha[5]", "alpha[10]", "alpha[18]"],
```

```
    axis="index"))

print("\nNon-centered parameterization, affine transform reparameterization")
print(fit_nc_affine.summary(sig_figs=3).round(decimals=3).filter(
    ["mu", "sigma",
     "theta[1]", "theta[5]", "theta[10]", "theta[18]",
     "alpha[1]", "alpha[5]", "alpha[10]", "alpha[18]"],
    axis="index"))
```

```
Centered parameterization
            Mean   MCSE  StdDev     5%    50%    95%   N_Eff  N_Eff/s  R_hat
name
mu        -1.030  0.002   0.094 -1.180 -1.030 -0.873  2230.0   3480.0   1.00
sigma      0.189  0.004   0.103  0.054  0.173  0.382   696.0   1080.0   1.01
theta[1]   0.298  0.001   0.045  0.239  0.291  0.382  2188.0   3407.0   1.00
theta[5]   0.277  0.001   0.036  0.224  0.273  0.342  4967.0   7737.0   1.00
theta[10]  0.260  0.000   0.034  0.204  0.261  0.316  4744.0   7390.0   1.00
theta[18]  0.241  0.001   0.037  0.174  0.244  0.296  1877.0   2924.0   1.00
alpha[1]  -0.863  0.004   0.209 -1.160 -0.892 -0.480  2223.0   3462.0   1.00
alpha[5]  -0.968  0.003   0.180 -1.240 -0.978 -0.654  5031.0   7836.0   1.00
alpha[10] -1.050  0.003   0.181 -1.360 -1.040 -0.770  4591.0   7152.0   1.00
alpha[18] -1.160  0.005   0.211 -1.560 -1.130 -0.867  1911.0   2977.0   1.00

Non-centered parameterization, std_normal reparameterization
            Mean   MCSE  StdDev     5%    50%    95%    N_Eff  N_Eff/s  R_hat
name
mu        -1.030  0.001   0.090 -1.170 -1.030 -0.878   8040.0  10900.0    1.0
sigma      0.164  0.002   0.110  0.017  0.148  0.364   4120.0   5560.0    1.0
theta[1]   0.293  0.001   0.043  0.240  0.284  0.378   6968.0   9404.0    1.0
theta[5]   0.274  0.000   0.034  0.225  0.271  0.336   9617.0  12978.0    1.0
theta[10]  0.261  0.000   0.032  0.208  0.262  0.314  10816.0  14596.0    1.0
theta[18]  0.244  0.000   0.036  0.178  0.249  0.295   7175.0   9683.0    1.0
alpha[1]  -0.887  0.002   0.202 -1.150 -0.924 -0.496   6982.0   9422.0    1.0
alpha[5]  -0.978  0.002   0.170 -1.240 -0.989 -0.681   9709.0  13103.0    1.0
alpha[10] -1.050  0.002   0.170 -1.340 -1.040 -0.779  10581.0  14280.0    1.0
alpha[18] -1.140  0.002   0.207 -1.530 -1.100 -0.874   7012.0   9463.0    1.0

Non-centered parameterization, affine transform reparameterization
            Mean   MCSE  StdDev     5%    50%    95%    N_Eff  N_Eff/s  R_hat
name
mu        -1.030  0.001   0.092 -1.180 -1.030 -0.882  10600.0  13900.0    1.0
sigma      0.165  0.002   0.112  0.016  0.149  0.369   3870.0   5090.0    1.0
theta[1]   0.293  0.000   0.042  0.241  0.284  0.376   7682.0  10095.0    1.0
theta[5]   0.274  0.000   0.034  0.225  0.271  0.336  13773.0  18098.0    1.0
theta[10]  0.261  0.000   0.033  0.207  0.261  0.313  13102.0  17217.0    1.0
theta[18]  0.243  0.000   0.036  0.178  0.249  0.293   7881.0  10356.0    1.0
alpha[1]  -0.887  0.002   0.198 -1.150 -0.923 -0.508   7695.0  10112.0    1.0
alpha[5]  -0.979  0.001   0.171 -1.240 -0.990 -0.681  13781.0  18109.0    1.0
alpha[10] -1.050  0.002   0.172 -1.340 -1.040 -0.785  12765.0  16773.0    1.0
alpha[18] -1.140  0.002   0.205 -1.530 -1.100 -0.880   7619.0  10012.0    1.0
```

We plot the sample values for log scale and the first player ability parameter, `alpha[1]`. This plot is almost identical to the above plot, "hierarchical vs generated quantity variable". Critically, this plot differs from the first plot from the centered parameterization.

In [20]:
```
df_x_y = pd.DataFrame(
    data={'x': fit_nc_affine.stan_variable('alpha')[: , 0],
          'y': np.log(fit_nc_affine.stan_variable('sigma'))}
)

scatter_plot(df_x_y,
         x_lab = "alpha[1]: player 1 log odds of success",
         y_lab = "log(sigma): log population scale",
         title = "hierarchical vs fixed param, affine transform")
```



Out[20]: <ggplot: (8793216999400)>

Don't Panic!

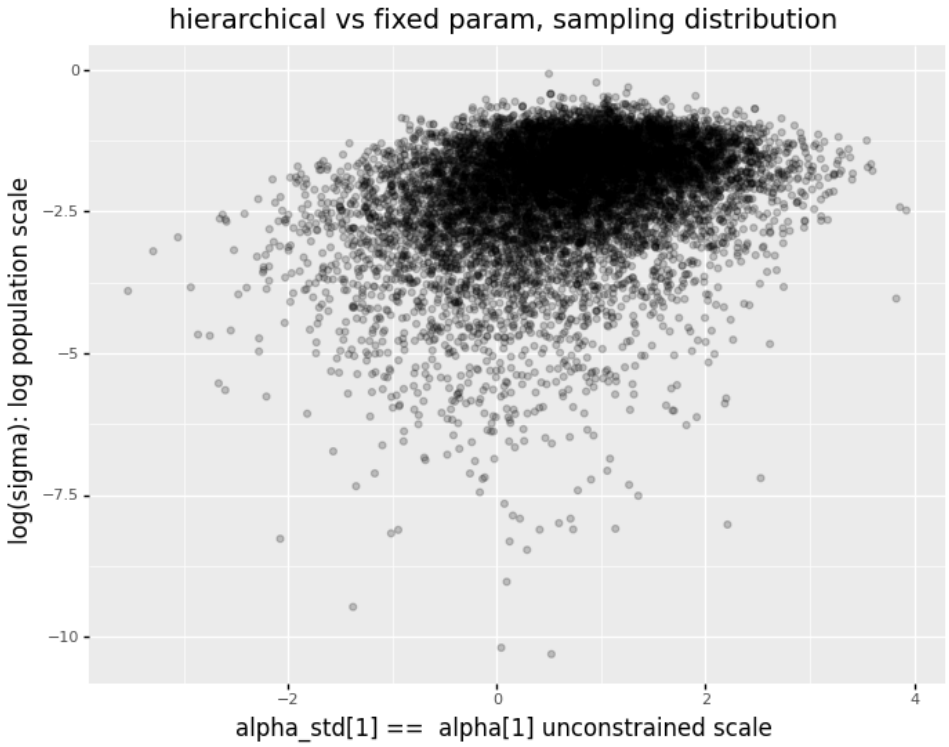
You may be asking: *"but this is a funnel plot, isn't this bad?"*

The answer is: *"no!*

Stan reports the parameter estimates on the *constrained* scale, but it computes on the *unconstrained* scale. The corresponding unconstrained value for `alpha` is $\frac{\alpha_n - \mu}{\sigma}$. In the generated quantities block we recover this as variable `alpha_std` .

Plotting `log(sigma)` against `alpha_std[1]` we see the same sampling distribution as in the standard normal parameterization.

In [21]:
```
df_x_y = pd.DataFrame(
    data={'x': fit_nc_affine.stan_variable('alpha_std')[: , 0],
          'y': np.log(fit_nc_affine.stan_variable('sigma'))}
)

scatter_plot(df_x_y,
         x_lab = "alpha_std[1] ==  alpha[1] unconstrained scale",
         y_lab = "log(sigma): log population scale",
         title = "hierarchical vs fixed param, sampling distribution")
```



Out[21]: <ggplot: (8793166089451)>

Both `hier-logit-nc-std-norm.stan` and `hier-logit-nc-affine-xform.stan` produce essentially the same results; this is because both models are essentially the same model: they encode the non-centered parameterization.

In program `hier-logit-nc-std-norm.stan` we define `alpha` as a transformed parameter and recover `theta` in the generated quantities block.

```
transformed parameters {
  vector[N] alpha = mu + sigma * alpha_std;
}
...
generated quantities {
  vector[N] theta = inv_logit(mu + sigma * alpha_std);
}
```

In program `hier-logit-nc-affine-xform.stan` variable `alpha` is a parameter with hierarchical prior `normal(mu, sigma)` . In the generated quantities block we recover `theta` , our estimate of a player's chance of success. Were there a need for it, we would be able to generate variable `alpha_std` as well.

```
generated quantities {
  vector[N] theta = inv_logit(alpha);
  vector[N] alpha_std = (alpha - mu)/sigma;
}
```

Discussion

Hierarchical models where the of hierarchical prior is specified in terms of a location and scale can be parameterized in one of two ways: centered or non-centered. When there are enough per-group observations, the sampler can determine the amount of group-level variance from the amount of individual-level variance and the centered parameterization is recommended. For smaller amounts of per-group observations, the non-centered parameterization is preferred.

For the non-centered parameterization, using the affine transform makes it easier to see the hierarchical structure of the model. When using the affine transform, the sampler computes on the unconstrained scale reports the parameter value on the constrained scale. For this reason, using standard normal parameterization may be more computationally efficient, as it eleminates extra transforms, but for simple models this difference may not be noticeable.

In this note we only consider models with a normal hierarchical prior, which can be coded either by use of the Stan language's `offset, multiplier` syntax, or by explicitly introducing a standardized parameter. Non-normal hierarchical priors are more challenging to reparameterize and are beyond the scope of this discussion.

In [ ]: