Age_Structure

December 18, 2023

0.1 The Principle

When dealing with age structures the recipe is pretty straightforward:

1. Group fish into bins by length
2. Age a sample of fish in each bin
3. Extrapolate that age key across all the fish in each bin
4. Get an age structure

However usually those bins are equally sized. Let's see if we can do better using what we know about the Fisher Information.

First we need to start with a probability function. Specifically let's look at $P(t|b)$ - the probability of getting a specific age, given a specific bin. Note that both t and b are discrete in this situation do we're looking at a probability function as opposed to a probability distribution function.

Now we don't actually know $P(t|b)$ (if we did we wouldn't be asking this question in the first place). But we do have a model for the other way around:

$$P(b|t) = \int_b \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{L-L_\infty(1-e^{-k(t-t_0)})}{\sigma}\right)^2} dL$$

which we'll go ahead and assume is readily computable (i.e. we've already fit L_∞, k, and σ). Furthermore given we're assuming we've taking loads of length samples we also know $P(b)$. So by Bayes' Theorem:

$$P(t|b) = \frac{P(b|t)P(t)}{P(b)}$$

However this seems to present a problem - aren't we trying to figure out what $P(t)$ is? Certainly, but for now let's assume it is a parameter of our model - $P(t) = \theta_t$. Therefore our model is:

$$P(t|b) = \frac{P(b|t)\theta_t}{P(b)}$$

Alright so our log likelihood is:

$$l = \ln\left(P(b|t)\right) - \ln\left(P(b)\right) + \ln\theta_t$$

and:

$$\partial_t l = \frac{1}{\theta_t}$$

and:

$$\partial_t^2 l = -\frac{1}{\theta_t^2}$$

Given none of the other derivatives exist (technically the θ_t are related in the fact that they must sum to 1, but that doesn't give them meaningful derivatives with respect to each other as we're more or less free to choose all of them but one).

Now note that we know for every $\tau \neq t$ that

$$\partial_t^2 l = 0$$

. Therefore:

$$I_{t,t} = -E[\partial_t^2 l] = -\sum_\tau \partial_t^2 l \bullet P(\tau|b) = \frac{1}{\theta_t^2} P(t|b) = \frac{1}{\theta_t^2} \frac{P(b|t)\theta_t}{P(b)} = \frac{1}{\theta_t} \frac{P(b|t)}{P(b)}$$

Alright so we now know we have a diagonal matrix made up of these components. Furthermore given we know that a multiplier on a row of our matrix just results in a multiplier on our determinant, we can just remove the θ_t^{-1} components as they won't actually contribute to the maximization of our information (they'll just represent a collective constant multiplier). What we are interested in then is the matrix:

$$\begin{pmatrix} \sum_b n\frac{P(b|0)}{P(b)} & 0 & ... & 0 \\ 0 & \sum_b n\frac{P(b|1)}{P(b)} & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & \sum_b n\frac{P(b|A)}{P(b)} \end{pmatrix}$$

where A is the maximum practical age of the species and n is the number of samples to be taken per bin b.

Let's go ahead and use this now!

0.2 A Working Example in Simulation

The question now is, what are the parameters of our optimization? Well we can imagine that we want to sample the full range of lengths. So we're really choosing how we want to divide things up. Furthermore we could specify the number of bins we want in advance. Then it's just a matter of how much of the full range each one takes. So if we're going to have N buckets we can imagine N numbers that all sum to 1. Then this can be extrapolated across our full bin size.

Before we build our optimization however, let's build a population of fish.

[ ]:
```
import numpy as np
import pandas as pd
```

```python
import plotly.express as px

L_inf = 1000
K = 0.3
sigma = 50

def length(age):
    return L_inf * (1 - np.exp(-K * age))

Z = 0.2
age_dist = np.array([np.exp(-Z * a) for a in range(100)])
age_dist = age_dist[age_dist > 0.01]
age_dist = age_dist[1:]
max_A = len(age_dist)
print(max_A)

age_dist = age_dist / np.sum(age_dist)

num_samples = 1000000
population = []
for age, portion in zip(range(1, max_A + 1), age_dist):
    num_samples_for_age = int(num_samples * portion)
    for _ in range(num_samples_for_age):
        population.append({
            'age': age,
            'length': max(0, length(age) + np.random.normal(0, sigma))
        })
population = pd.DataFrame(population)
print(population.shape)
population.head()
```

```
23
(999988, 2)
```

[ ]:

	age	length
0	1	319.332069
1	1	345.770808
2	1	218.881771
3	1	227.066793
4	1	287.335444

[ ]:

```python
px.scatter(population.sample(10000), x='age', y='length', marginal_x='histogram', marginal_y='histogram')
```

Alright with that out of the way we can build our optimization.

```
[ ]: from deap import base, creator, tools
     import numpy as np

     creator.create("FitnessMax", base.Fitness, weights=(1.0,))
     creator.create("Individual", list, fitness=creator.FitnessMax)

     N = max_A

     def init_individual():
         individual = np.random.random(N)
         individual = individual / np.sum(individual)
         for el in individual:
             yield el

     toolbox = base.Toolbox()
     toolbox.register("individual", tools.initIterate, creator.Individual,␣
      ↪init_individual)
     toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

For mutation in order to avoid negative numbers, we're simply going to resize one or more of the binsby increasing them and then resize all of the other bins in response.

```
[ ]: def mutate(individual, prob, step=0.25):
         for i in range(len(individual)):
             if np.random.random() < prob:
                 individual[i] = individual[i] + step * individual[i]
         divisor = np.sum(individual)
         for i in range(len(individual)):
             individual[i] = individual[i] / divisor

     toolbox.register("mutate", mutate, prob=0.5)
```

Selection is straightforward.

```
[ ]: toolbox.register("select", tools.selTournament, tournsize=3)
```

```
[ ]: def mate(ind1, ind2):
         cx1 = np.random.randint(0, len(ind1)-1)
         cx2 = np.random.randint(cx1+1, len(ind1))
         for i in range(cx1, cx2):
             ind1[i], ind2[i] = ind2[i], ind1[i]
         divisor1 = np.sum(ind1)
         divisor2 = np.sum(ind2)
         for i in range(len(ind1)):
             ind1[i] = ind1[i] / divisor1
             ind2[i] = ind2[i] / divisor2
```

```
toolbox.register("mate", mate)
```

[ ]:
```
import scipy.stats as stats

def evaluate(individual, cutoff=0.000):
    end = 0
    trace = np.zeros(max_A)
    for i, bin_prop in enumerate(individual):
        start = end
        end = start + bin_prop * L_inf
        if i == len(individual) - 1:
            end = 1000000
        prob_length = population[(population['length'] >= start) &␣
 ↪(population['length'] <= end)].shape[0] / population.shape[0]
        if prob_length < cutoff:
            return 0,
        for i, age in enumerate(range(1, max_A + 1)):
            prob_length_given_t = (
                stats.norm.cdf(end, loc=length(age), scale=sigma)
                - stats.norm.cdf(start, loc=length(age), scale=sigma)
            )
            trace[i] += prob_length_given_t / prob_length
    score = np.sum(np.log(trace))
    if score == np.inf:
        score = 0
    return score,


toolbox.register("evaluate", evaluate)
```

[ ]:
```
import random
from tqdm import tqdm

pop = toolbox.population(n=50)
CXPB, MUTPB, NGEN = 0.5, 0.2, 50
best_fitnesses = []

# Evaluate the entire population
fitnesses = map(toolbox.evaluate, pop)
for ind, fit in zip(pop, fitnesses):
    ind.fitness.values = fit

for g in tqdm(range(NGEN)):
    # Select the next generation individuals
    offspring = toolbox.select(pop, len(pop))
    # Clone the selected individuals
    offspring = list(map(toolbox.clone, offspring))
```

```python
    # Apply crossover and mutation on the offspring
    for child1, child2 in zip(offspring[::2], offspring[1::2]):
        if random.random() < CXPB:
            toolbox.mate(child1, child2)
            del child1.fitness.values
            del child2.fitness.values

    for mutant in offspring:
        if random.random() < MUTPB:
            toolbox.mutate(mutant)
            del mutant.fitness.values

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
    fitnesses = map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    # The population is entirely replaced by the offspring
    pop[:] = offspring

    best_fitnesses.append(tools.selBest(pop, k=1)[0].fitness.values[0])

best_pop = tools.selBest(pop, k=1)[0]
```

```
/tmp/ipykernel_983/659842223.py:19: RuntimeWarning:

divide by zero encountered in scalar divide

100%|      | 50/50 [03:45<00:00,  4.51s/it]
```

[ ]:
```python
px.line(x=range(len(best_fitnesses)), y=best_fitnesses)
```

[ ]:
```python
best_pop
```

[ ]:
```
[0.11239807244279633,
 0.06440028540226556,
 0.18103634541905725,
 0.1309684177279674,
 0.011322295070648468,
 0.13241196736855845,
 0.023820364807438762,
 0.08476187223538392,
 0.03833347614070624,
 0.024146035085229108,
 0.08544396840036989,
```

```
 0.015342564915180266,
 0.01353509681610802,
 0.016029600169582933,
 0.023903410628168024,
 0.01868907433227862,
 0.007636752345753128,
 0.002831008606793264,
 0.002267447202997696,
 0.0033181767552007,
 0.0013971038219842645,
 0.0022209305842156505,
 0.003785733721316894]
```

[ ]:
```
def bin_it(bins):
    bin_starts = [0]
    for bin_prop in bins[:-1]:
        bin_starts.append(bin_starts[-1] + bin_prop * L_inf)
    population['bin'] = 0
    for i, bin_start in enumerate(bin_starts):
        population.loc[(population['length'] >= bin_start), 'bin'] = i
    return population


pop = bin_it(best_pop)
px.scatter(pop.sample(10000), y='bin', x='length', marginal_x='histogram',␣
↪marginal_y='histogram')
```

[ ]:
```
px.scatter(pop.sample(10000), y='bin', x='age', marginal_x='histogram',␣
↪marginal_y='histogram')
```

[ ]:
```
def produce_age_key(pop, n):
    age_key = []
    for bin in pop['bin'].unique():
        df = pop[pop['bin'] == bin].sample(n, replace=True)
        df = df.groupby(['bin', 'age']).count().rename({'length': 'prop'},␣
↪axis=1).reset_index()
        df['prop'] = df['prop'] / n
        age_key.append(df)
    age_key = pd.concat(age_key)
    return age_key

def estimate_from_age_key(pop, age_key):
    counts = pop.groupby(['bin']).count().rename({'length': 'count'}, axis=1).
↪reset_index()
    counts = counts[['bin', 'count']]
    counts = counts.merge(age_key, on='bin')
    counts['age_count'] = counts['count'] * counts['prop']
```

```
    counts = counts.groupby('age').sum().reset_index()[['age', 'age_count']]
    counts['age_prop'] = counts['age_count'] / counts['age_count'].sum()
    return counts[['age', 'age_prop']]

def gather_stats(pop, n, trials):
    dfs = []
    for _ in tqdm(range(trials)):
        age_key = produce_age_key(pop, n)
        df = estimate_from_age_key(pop, age_key)
        rows = []
        for age in range(1, max_A + 1):
            if age not in df['age'].values:
                rows.append({
                    'age': age,
                    'age_prop': 0
                })
        df = pd.concat([df, pd.DataFrame(rows)])
        dfs.append(df)
    return pd.concat(dfs).groupby('age').describe().reset_index()
```

[ ]:
```
age_dist
```

[ ]:
```
array([0.18310984, 0.14991765, 0.12274219, 0.10049281, 0.08227655,
       0.06736234, 0.05515162, 0.04515433, 0.03696924, 0.03026785,
       0.02478122, 0.02028915, 0.01661135, 0.01360022, 0.01113492,
       0.0091165 , 0.00746396, 0.00611097, 0.00500324, 0.00409631,
       0.00335377, 0.00274584, 0.0022481 ])
```

[ ]:
```
pop = bin_it(best_pop)
print(evaluate(best_pop, 0))
gather_stats(pop, 15, 100)
```

```
(81.94794278967966,)

100%|          | 100/100 [00:11<00:00,  8.97it/s]
```

[ ]:

	age	age_prop						\
		count	mean	std	min	25%	50%	75%
0	1	100.0	0.181859	0.010240	0.148460	0.174820	0.183345	0.191228
1	2	100.0	0.150976	0.016004	0.099864	0.140608	0.151399	0.161730
2	3	100.0	0.124307	0.017238	0.060571	0.114919	0.123767	0.134331
3	4	100.0	0.100160	0.016487	0.063567	0.088466	0.101234	0.110442
4	5	100.0	0.078703	0.015670	0.029875	0.069440	0.077162	0.089387
5	6	100.0	0.067705	0.017549	0.027267	0.055862	0.067880	0.078321
6	7	100.0	0.057005	0.014810	0.019416	0.047119	0.056083	0.066026
7	8	100.0	0.045197	0.011836	0.023134	0.037730	0.044599	0.051058
8	9	100.0	0.034687	0.010393	0.015268	0.027386	0.033184	0.041278
9	10	100.0	0.031569	0.010113	0.013481	0.024326	0.029665	0.037695

```
10  11    100.0  0.025276  0.007820  0.009817  0.020591  0.025106  0.028795
11  12    100.0  0.019813  0.007673  0.007363  0.013935  0.019148  0.024108
12  13    100.0  0.015704  0.006614  0.002141  0.011186  0.015023  0.019455
13  14    100.0  0.013347  0.006135  0.002276  0.008432  0.013293  0.018242
14  15    100.0  0.011832  0.006086  0.000921  0.006947  0.011074  0.015088
15  16    100.0  0.009059  0.004695  0.000613  0.005448  0.008614  0.012789
16  17    100.0  0.007757  0.005270  0.000200  0.003809  0.006487  0.011380
17  18    100.0  0.007064  0.004609  0.000200  0.003599  0.006799  0.009629
18  19    100.0  0.004991  0.003704  0.000000  0.001642  0.004276  0.008308
19  20    100.0  0.004301  0.003775  0.000000  0.000797  0.003972  0.006503
20  21    100.0  0.003116  0.002669  0.000000  0.000594  0.002958  0.004849
21  22    100.0  0.003104  0.003037  0.000000  0.000267  0.002764  0.004616
22  23    100.0  0.002468  0.002692  0.000000  0.000267  0.001305  0.004084


         max
0   0.203249
1   0.195513
2   0.164855
3   0.139977
4   0.113506
5   0.123897
6   0.096920
7   0.080109
8   0.062227
9   0.064541
10  0.045490
11  0.039727
12  0.036853
13  0.028005
14  0.030220
15  0.021438
16  0.023339
17  0.020668
18  0.016897
19  0.015519
20  0.009608
21  0.012491
22  0.010776
```

[ ]:
```python
even_pop = np.ones(max_A) / max_A
print(evaluate(even_pop, 0))
pop = bin_it(even_pop)
gather_stats(pop, 15, 100)
```

```
/tmp/ipykernel_983/659842223.py:19: RuntimeWarning:
```

```
divide by zero encountered in scalar divide
```

```
(0,)
```

```
100%|      | 100/100 [00:11<00:00,  8.99it/s]
```

[ ]:

	age	age_prop						
		count	mean	std	min	25%	50%	75%
0	1	100.0	0.183193	0.003935	0.173552	0.179842	0.183937	0.186210
1	2	100.0	0.150648	0.007578	0.128423	0.145390	0.151188	0.155128
2	3	100.0	0.122537	0.009391	0.097113	0.116837	0.121801	0.128572
3	4	100.0	0.100436	0.013459	0.070537	0.091676	0.103233	0.108894
4	5	100.0	0.083274	0.015480	0.043810	0.072881	0.083134	0.093177
5	6	100.0	0.066761	0.014752	0.036002	0.056813	0.066396	0.077498
6	7	100.0	0.056869	0.015288	0.021142	0.046402	0.056194	0.064409
7	8	100.0	0.044929	0.014121	0.017244	0.036566	0.043547	0.052571
8	9	100.0	0.034823	0.014126	0.008984	0.023965	0.032934	0.045098
9	10	100.0	0.029597	0.012116	0.004718	0.021196	0.030593	0.037496
10	11	100.0	0.025499	0.012205	0.000000	0.016969	0.023770	0.032009
11	12	100.0	0.018796	0.011608	0.000000	0.008239	0.018593	0.024995
12	13	100.0	0.015816	0.011021	0.000000	0.008239	0.013416	0.021655
13	14	100.0	0.015569	0.008934	0.000000	0.008239	0.014244	0.021655
14	15	100.0	0.010665	0.008728	0.000000	0.005062	0.008239	0.016478
15	16	100.0	0.007559	0.006828	0.000000	0.000000	0.008239	0.013416
16	17	100.0	0.007494	0.007482	0.000000	0.000000	0.008239	0.013416
17	18	100.0	0.006109	0.006990	0.000000	0.000000	0.005177	0.008239
18	19	100.0	0.005947	0.006949	0.000000	0.000000	0.005177	0.008239
19	20	100.0	0.003875	0.005193	0.000000	0.000000	0.000000	0.008239
20	21	100.0	0.003435	0.004543	0.000000	0.000000	0.000000	0.008239
21	22	100.0	0.003714	0.005381	0.000000	0.000000	0.000000	0.008239
22	23	100.0	0.002455	0.004127	0.000000	0.000000	0.000000	0.005177

	max
0	0.190056
1	0.172952
2	0.144696
3	0.126727
4	0.119480
5	0.121199
6	0.097725
7	0.094513
8	0.087523
9	0.067275
10	0.064213
11	0.051548
12	0.043964

```
13   0.043309
14   0.039788
15   0.032956
16   0.037131
17   0.037674
18   0.034612
19   0.016478
20   0.016478
21   0.029435
22   0.016478
```

[ ]:
```
even_pop = np.ones(max_A) / max_A
print(evaluate(even_pop, 0))
pop = bin_it(even_pop)
gather_stats(pop, 30, 100)
```

```
/tmp/ipykernel_983/659842223.py:19: RuntimeWarning:

divide by zero encountered in scalar divide


(0,)

100%|      | 100/100 [00:11<00:00,  9.00it/s]
```

[ ]:
```
    age age_prop                                                        \
          count      mean       std       min       25%       50%       75%
0     1   100.0  0.182939  0.002533  0.177259  0.181171  0.182595  0.184573
1     2   100.0  0.149868  0.005861  0.137599  0.146302  0.149246  0.153562
2     3   100.0  0.123084  0.008395  0.104086  0.117901  0.123534  0.129225
3     4   100.0  0.100801  0.009826  0.077160  0.093309  0.100130  0.108022
4     5   100.0  0.082815  0.009783  0.056817  0.077366  0.082622  0.089436
5     6   100.0  0.066386  0.010224  0.046480  0.059341  0.066119  0.073621
6     7   100.0  0.054986  0.009030  0.035571  0.048044  0.055396  0.060670
7     8   100.0  0.045662  0.010076  0.019679  0.038961  0.045087  0.053879
8     9   100.0  0.036408  0.008683  0.013514  0.030902  0.035151  0.043371
9    10   100.0  0.030526  0.010082  0.010827  0.023483  0.030182  0.037454
10   11   100.0  0.025086  0.009034  0.006708  0.019654  0.024477  0.030283
11   12   100.0  0.021175  0.009354  0.002588  0.013416  0.020124  0.027288
12   13   100.0  0.014825  0.006844  0.002588  0.009296  0.014947  0.019066
13   14   100.0  0.013443  0.006309  0.002359  0.009067  0.012772  0.017535
14   15   100.0  0.011319  0.006127  0.000000  0.007995  0.010827  0.014947
15   16   100.0  0.008982  0.004685  0.000000  0.005889  0.009182  0.012003
16   17   100.0  0.008035  0.005736  0.000000  0.004119  0.008239  0.012358
17   18   100.0  0.006100  0.004788  0.000000  0.002588  0.004119  0.008503
18   19   100.0  0.005000  0.004653  0.000000  0.002588  0.004119  0.006972
19   20   100.0  0.003925  0.003803  0.000000  0.000000  0.004119  0.005560
20   21   100.0  0.003312  0.003330  0.000000  0.000000  0.004119  0.004119
```

21	22	100.0	0.002817	0.003077	0.000000	0.000000	0.002588	0.004119
22	23	100.0	0.002505	0.002773	0.000000	0.000000	0.002588	0.004119

	max
0	0.189992
1	0.166415
2	0.145592
3	0.122074
4	0.110504
5	0.100551
6	0.078356
7	0.066924
8	0.057189
9	0.058381
10	0.055337
11	0.044137
12	0.035071
13	0.032482
14	0.027305
15	0.023186
16	0.022956
17	0.021425
18	0.020597
19	0.016478
20	0.016004
21	0.014947
22	0.009296

So there we have it! A much more relatively stable result with far less data using an optimized query design approach. Pretty cool!