mseekr statistics

July 13, 2020

1 mSEEKR Score Probability

A central challenge in parsing lncRNA sequences with an HMM is the probability of getting a hit of length L and a corresponding score S. These two variables form a joint probability distribution

$$P(S, L)$$

The distribution of L is known from the definition of a hidden markov model, and follows a geometric distribution. In that, the probability of k successive same-state transitions, followed by the first transition to a different hidden state is:

$$P(L) = (1 - \theta)^k \theta$$

Where θ is the transition probability (e.g., query $\rightarrow$ query). The distribution of scores $P(S)$ is unknown, however. Scores are calculated by summing successive k-mers log-odds between the query and the null model.

$$S = \sum_{i}^{L} \log \left(\frac{q_i}{n_i} \right)$$

Where q_i is the frequency of k-mer i in the query state, and n_i is the frequency of i in the null state. A $S > 0$ indicates the sequence is more likely to belong to the query and $S \leq 0$ indicates that it is not. This scoring method is analgous to BLAST and other alignment techniques, wherein each nucleotide pairing has a score (e.g. +1 for a match, -1 for a mismatch), and these scores are summed along the length of a sequence. A requirement for these scoring matrices is that the expected score, i.e. $E[S]$, be negative, but that the probability of getting a positive score is > 0. This requirement ensures that random sequence will cause the cumulative score to trend negative, therefore preventing spurious hits.

While long hits are less likely than short hits, and therefore may be of interest, in reality we do not know what lengths are required for biological function. Furthermore, a long hit can have the same score as a short hit (hypothetically), if its k-mers are overall less similar to the query's than the short hit's k-mers.

Therefore, when assessing the significance of hits from hmmSEEKR – we are primarily interested in the probability of achieving a score S, regardless of the length of the hit.

While our method of ‘alignment’ between two sequences differs from actual pairwise local alignment, our method essentially meets all the assumptions of the statistics for ungapped local alignment for which there is well proven theory for how the scores are distributed.

The goal of this document is to demonstrate empircally that we can model the distribution of scores from mSEEKR in an analogous way, with some small alterations.

Assumptions that the statistics of local ungapped unalignment follow:

1. Scores can be based on frequencies of nucleotides in a query and background model i.e. $\log \frac{q_i}{n_i}$
2. The score is calculated by summing the log-odds for each letter/word
3. The expectation of all scores is negative (negative drift)
4. A positive score is possible, but rare
5. Sequences are drawn independently and identically distributed (an assumption always used, but usually only approximately true)
6. The sequences are long
7. Only high score matches (S > some threshold) are considered

The way we use mSEEKR to ‘align’ two lncRNAs based off of k-mer frequencies matches all these assumptions, the only real difference is that our alphabet is *k*-mers rather than individual nucleotides.

[1]:
```
import numpy as np
from scipy.stats import geom
import pickle
from itertools import product
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
```

[2]:
```
k=4
```

[3]:
```
kmers = [''.join(group) for group in product('ATCG',repeat=4)]
```

[4]:
```
hmm = pickle.load(open('../../mSEEKR/markovModels/mouseA_mm10Trscpts/4/hmm_MLE.
↪mkv','rb'))
```

[5]:
```
E = hmm['E']
```

[85]:
```
eD = pickle.load(open('../../mSEEKR/markovModels/hD_mm10Trscpts/2/hmm.
↪mkv','rb'))
kmers = [''.join(group) for group in product('ATCG',repeat=2)]
dLLR = [eD['E']['+'][i]-eD['E']['-'][i] for i in kmers]
dLLR = np.array(dLLR)
```

[86]:
```
kmers = [''.join(group) for group in product('ATCG',repeat=4)]
```

[6]:
```
llr = [E['+'][i]-E['-'][i] for i in kmers]
```

```
[7]: llr = np.array(llr)
```

```
[8]: haha = []
     for i in range(100):
         randL = geom.rvs(p=.01)
         score = np.zeros(randL)
         scores = np.random.choice(llr,randL,)
         scores = np.cumsum(scores)
         haha.append(scores)
```

1.1 Randomized Sequences

Below, I have generated 1000 random sequences of length distributed as described in the begining of this document (geometrically distributed). The plot below shows the cumulative sum of scores along the length of each transcript.

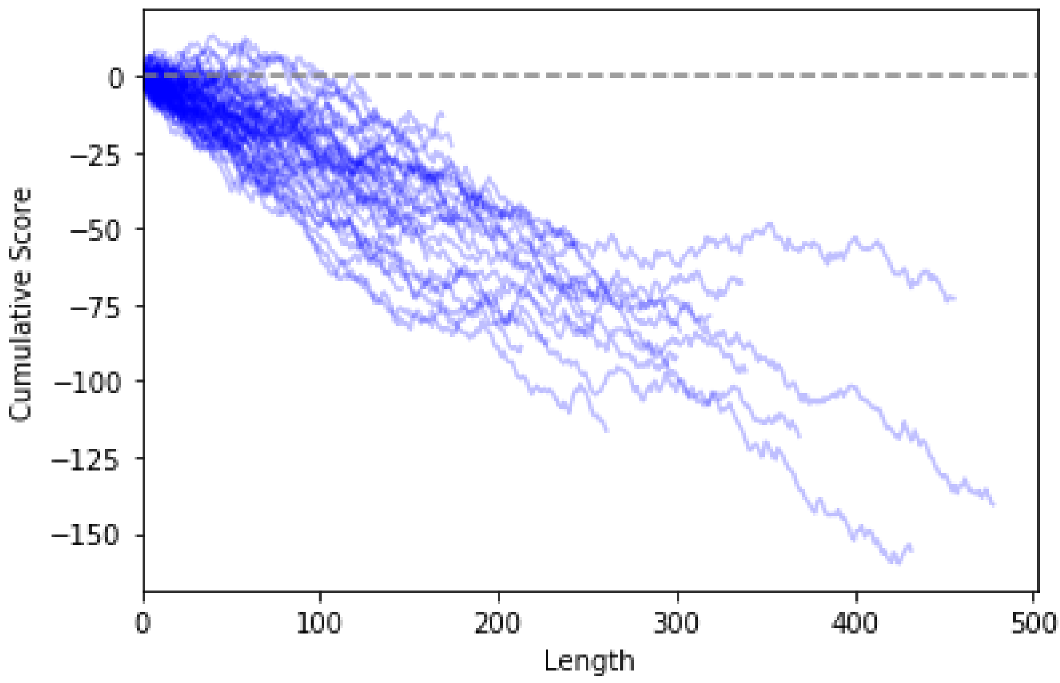
1. Draw a length L for the hit following distribution above P(L)
2. Draw L k-mers uniformly
3. Calculate the cumulative sum of k-mer log-odds (the score 'S')
4. Plot each series of cumalative scores as a plot below

The cumulative scores of these transcripts trend downwards because the average log-odds over all k-mers between the query and null is negative, so as the length of the sequence grows, $E[X_{t+1}|X_t] = X_t + \mu_{\texttt{log-ods}}$, where $\mu_{\texttt{log-odds}}$ is the average log-odds over all k-mers between the query and the null.

As is visible, there is a random distribution of lengths and scores

```
[9]: for i in haha:
         plt.plot(i,color='blue',alpha=.25)
     plt.ylabel('Cumulative Score')
     plt.xlabel('Length')
     plt.axhline(y=0,linestyle='--',color='grey')
     plt.xlim(0,)
```

```
[9]: (0, 501.9)
```



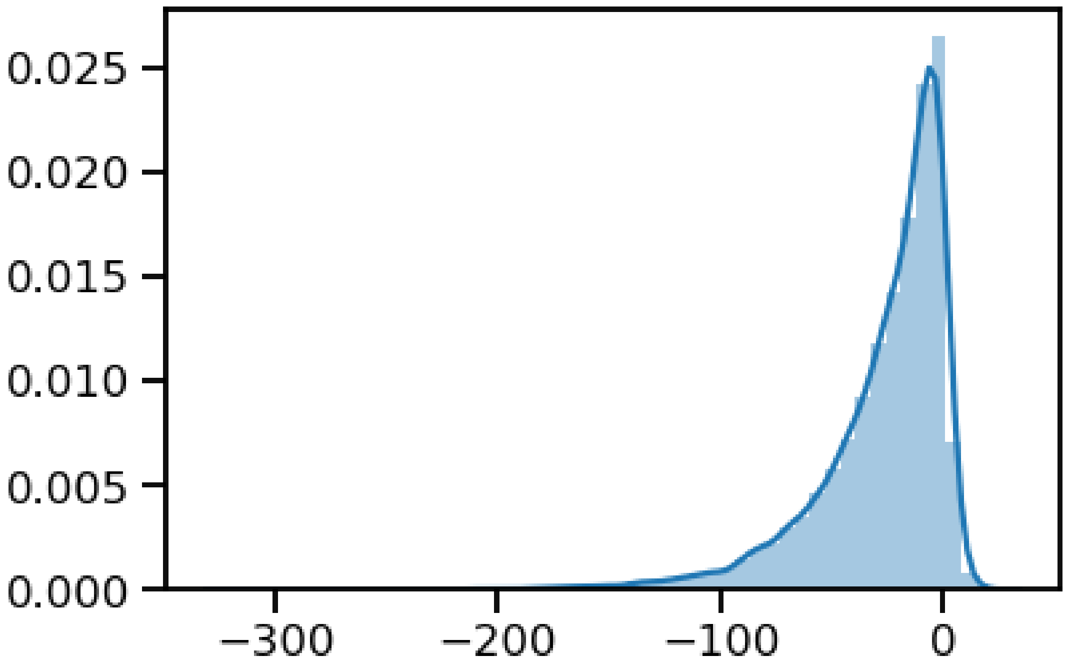
[10]:
```
haha = []
for i in range(10000):
    randL = geom.rvs(p=.01)
    score = np.zeros(randL)
    scores = np.random.choice(llr,randL,)
    scores = np.sum(scores)
    haha.append(scores)
```

2 Distribution of scores for random sequences in an HMM are *not* normally distributed

Below is the complete distribution of scores for the randomly generated sequences, only a very small number > 0. However, of these rare high scoring sequences, what is P(S>s), where s is the observed score.

[11]:
```
sns.set_context('talk')
sns.distplot(haha,bins=50)
```

[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8e299a3a10>
```



2.0.1 Side-bar: Distribution of scores at a *fixed length* does converge to a normal

The bias in the distribution above is caused by varying lengths of hits… following P(L) outlined above. Small length hits are both frequent and more likely to randomly have a score > 0.

However for a fixed length, randomly sampled sequences should converge to a normal distribution. The plot below illustrates that for L = 1000 over 10,000 simulations.

Potential thoughts for the future: Can this be used to make an even faster/more accurate model of p-vals? i.e. $P(S, L)$ is not normal (above), but $P(S|L)$ *is normal*. Therefore, we could easily calculate $P(S, L) = P(S|L)P(L)$. There is a formula for calculating the parameters of a normal distribution for the sum of L iid variables… since the mean and variance should be related to the mean and the variance of the iid variables you are summing…

$$\mu_{S|L} = \mu_S * L$$

[342]:
```python
haha = []
for i in range(10000):
    randL = 100
    score = np.zeros(randL)
    scores = np.random.choice(llr,randL,)
    scores = np.sum(scores)
    haha.append(scores)
```

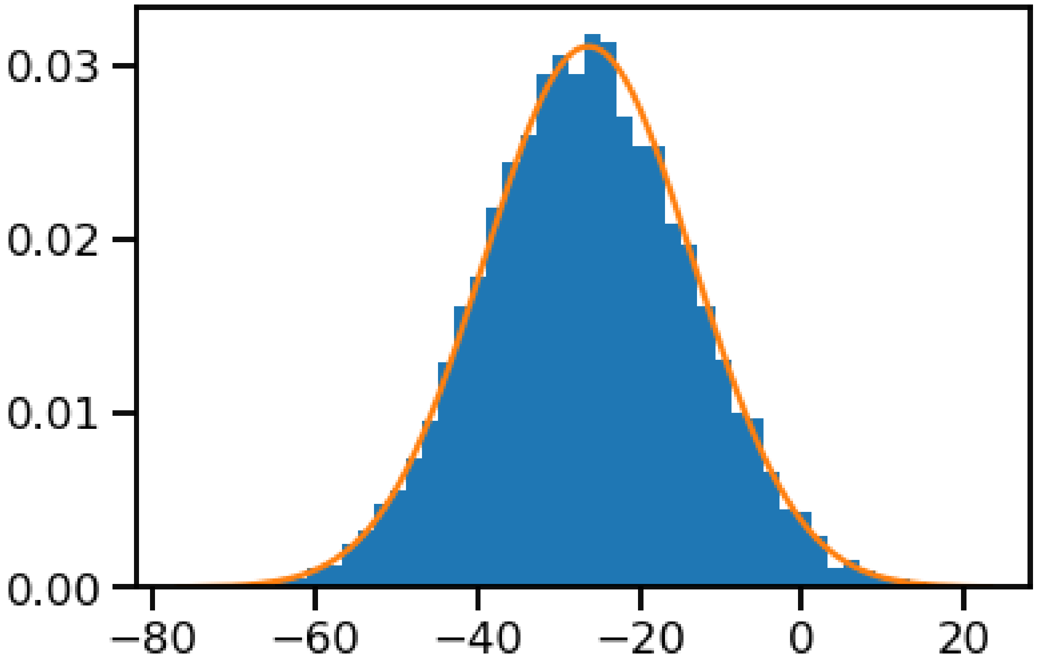
[343]:
```python
haha = sorted(haha)
```

[344]:
```
var = np.sqrt(np.var(llr))*np.sqrt(randL)
mean = randL * np.mean(llr)
```

The normal distribution fit (orange) was calculated only from the scoring matrix, no simulations. This would be an extremely fast calculation.

[349]:
```
sns.set_context('talk')
plt.hist(haha,bins=50,density=True)
plt.plot(haha,stats.norm.pdf(haha,*norm))
```

[349]:
```
[<matplotlib.lines.Line2D at 0x7f8e09af1410>]
```



Probability S = 20, L = 100

[351]:
```
print('P(S>s|L) = ',1-stats.norm.cdf(20,mean,var))
print('P(L=l) = ', stats.geom.pmf(randL,.01))

print('P(S>s,L=l) = ',(1-stats.norm.cdf(20,mean,var))*stats.geom.cdf(randL,.01))
```

```
P(S>s|L) =  0.00015500217379271763
P(L=l) =  0.0036972963764972644
P(S>s,L=l) =  9.826636521692919e-05
```

This may or may not work… it deviates significantly to how blast calculates p-vals. Again, we are not performing blast. However, what we are doing with the HMM and scoring hits essentially checks all the criteria for calculating a sequence similarity p-val. This above method assumes that we are summing L iid random variables (k-mer log odds), but in reality we are actually summing L maximizing values that are iid (i.e. only calculating scores in regions that are hits). Therefore, we would expect 'hits' of scores to follow an "extreme value distribution" rather than a normal distribution. Unfortunately, I don't know enough to say definitively if the above method works or not. My guess would be that it gives smaller p-values than what would actually be observed.

The work below shows that if you follow the general methodology of blast, we get a near perfect fit to a monte carlo simulation of scores > 0.

Indeed, if you convert the plot above to bit score (as shown further down this document), you find that this method yields p-values orders of magnitude smaller than the blast-like method below, meaning smaller scores may be considered significant when they are not. (Plot this below with bit scores?)

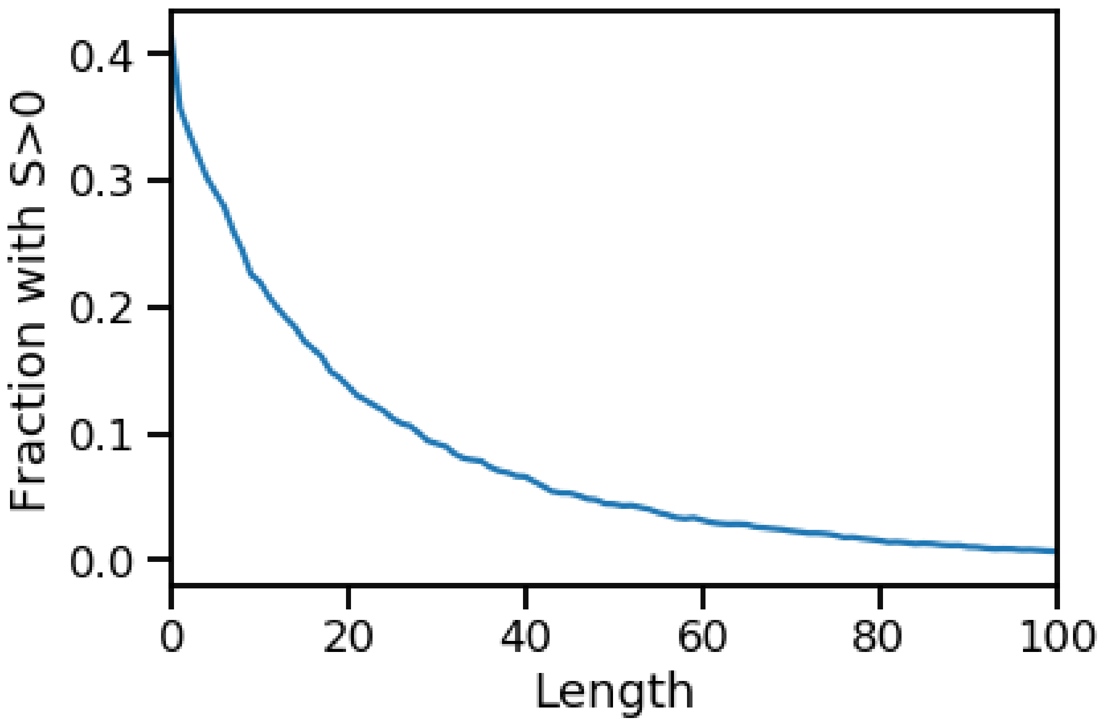
2.1 Small fraction of total sequences have > 0, and falls exponentially with length

As the length of a sequence grows – the chances of the random sequence having a positive score drops exponentially. Below, I have plotted the fraction of total randomly generated sequences (as above) that maintain a positive score at each position within its sequence.

[13]:
```python
haha = []
for i in range(10000):
    randL = geom.rvs(p=.01)
    score = np.zeros(randL)
    scores = np.random.choice(llr,randL,)
    scores = np.cumsum(scores)
    haha.append(scores)
data = pd.DataFrame(haha).values
np.sum(data>0,axis=0)
plt.plot(np.sum(data>0,axis=0)/10000)
plt.xlabel('Length')
plt.ylabel('Fraction with S>0')
plt.xlim(0,100)
```

```
/Users/dan/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:9:
RuntimeWarning: invalid value encountered in greater
  if __name__ == '__main__':
/Users/dan/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:10:
RuntimeWarning: invalid value encountered in greater
  # Remove the CWD from sys.path while we load stuff.
```

[13]:
```
(0, 100)
```



2.2 $P(S, L)$

For 10,0000 randomly generated sequences (as above), I have plotted (below) the joint distribution of lengths of each random sequence and the final cumulative score of that sequence for sequences that have a positive score (reflecting what mSEEKR is doing). There is essentially very little relationship between these two variables, and the distribution becomes harder and harder to estimate through monte carlo methodology as the sequences get longer. S and L are not completely independent of each other, however.

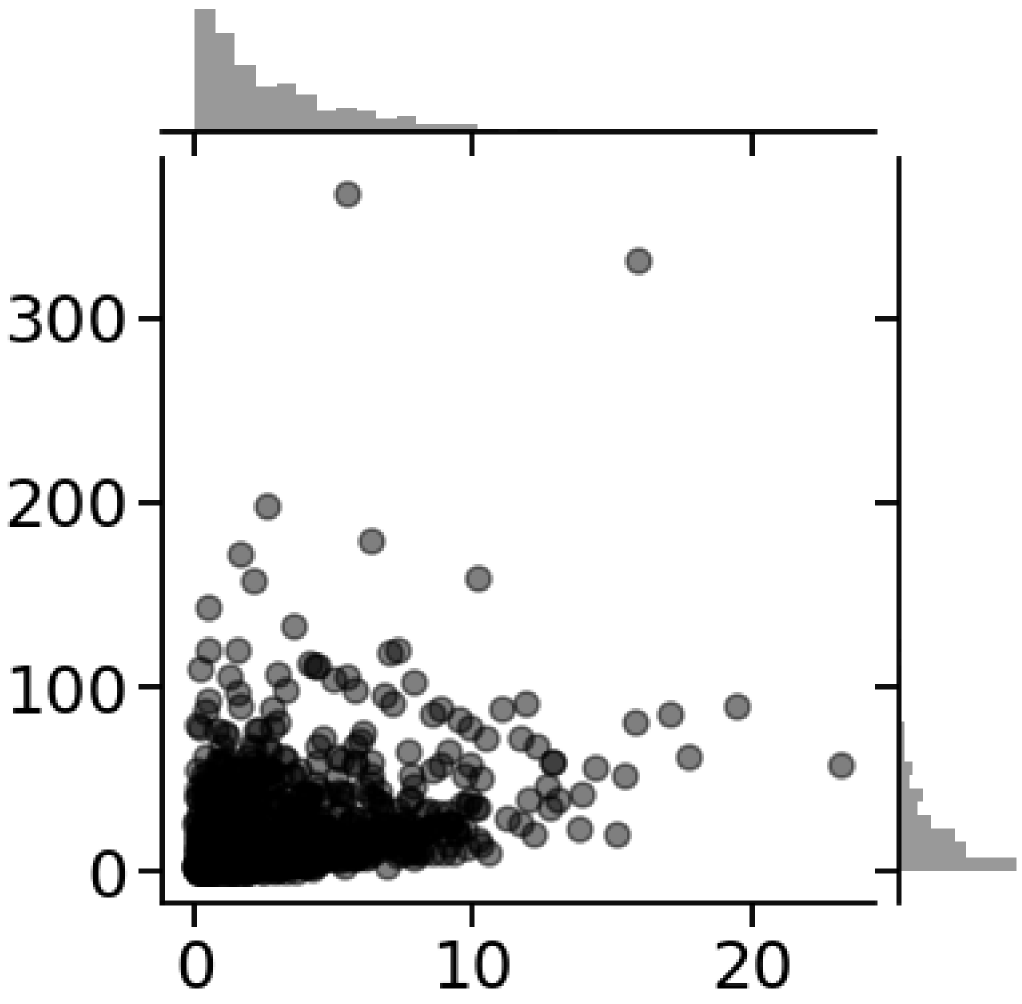
Therefore, when assessing the significance of hits in mSEEKR, our lack of understanding of the importance of length of these hits, I believe focusing exclusively on the distribution of scores, regardless of all lengths $P(S) = \sum_L P(S, L)$ to be most prudent.

2.3 Tail of $P(S)$

In mSEEKR we are attempting to model the tail behavior of S. In mSEEKR, we only select regions from the transcriptome where $S > 0$, despite the fact that the overwhelming majority of sequences have negative scores (line plot above). As an example, if you imagine a bell curve in your head, we are essentially trying to zoom in on the far far right end of the distribution (above some threshold, i.e. 0 in this case) and ask what the probability of a given score $S > 0$ is, relative *only* to the distribution of all scores greater than 0.

```
[53]: haha = []
      for i in range(10000):
```

```
    randL = geom.rvs(p=.01)
    score = np.zeros(randL)
    scores = np.random.choice(llr,randL,)
    scores = np.cumsum(scores)
    if scores[-1]>0:
        haha.append(scores)
scores = [i[-1] for i in haha]
lengths = [len(i) for i in haha]
sns.set_context('talk',font_scale=1.5)
g = sns.jointplot(scores,lengths,alpha=.5,color='k')
```



3 Expectation of Scores >0 for a given length Increases Logarithmically With Length Of Randomly Generated Sequence

A key result from the statistics of local ungapped alignment is that the random maximizing alignment score between two sequences grows logarithmically with the size of the search space (statistics of large scale sequence searching Spang & Vingron) (Dembo and Altschul 1992).

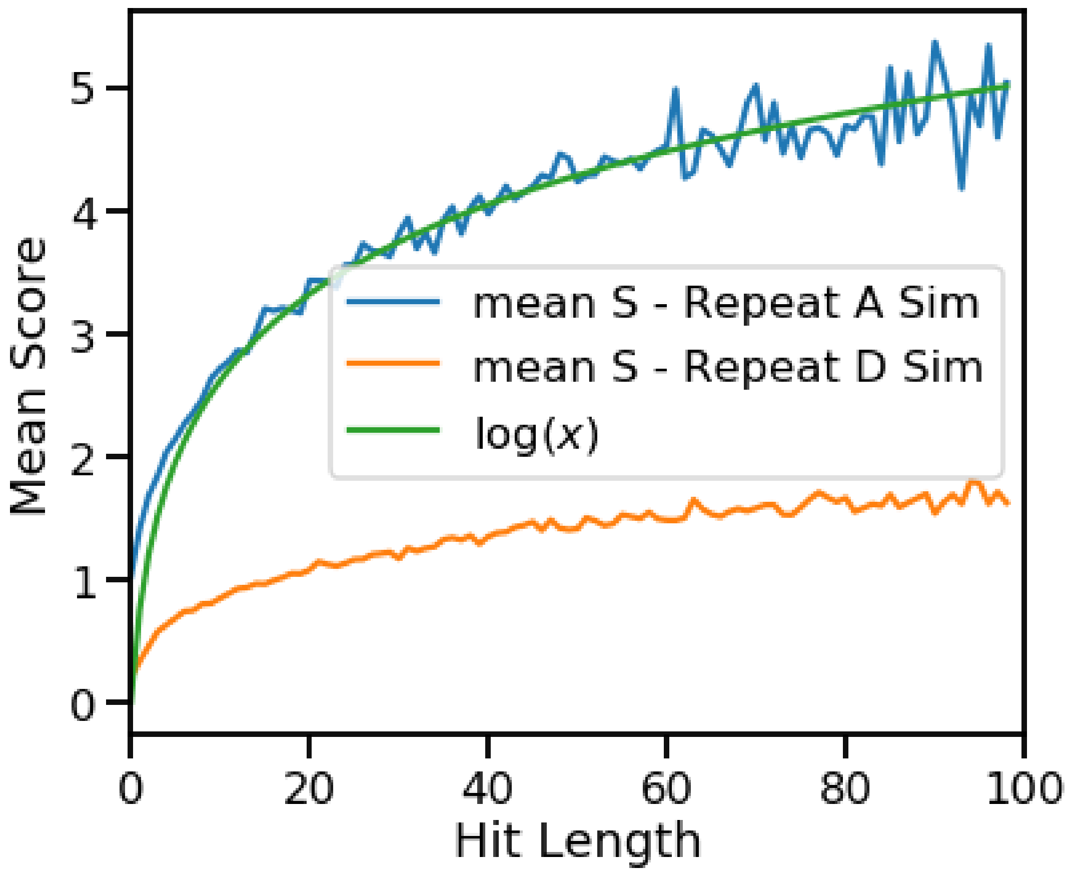
Similar behavior is observed with HMM hits (below).

Essentially, we're seeing the expected value of hits for each length (1,2,3,4,...,n) increases with log(n). I.e., the score is dependent on length of the hit, albeit the effect is relatively minor for most hits. The curve is not just log(n), however. There is a constant multiplying factor based on the scoring matrix (kmer frequencies).

In the plot below – see how the scores for repeat A grow faster than the scores for repeat D.

[149]:
```python
from collections import defaultdict

haha = defaultdict(list)
for i in range(1,100):
    randL = i
    for i in range(10000):
        score = np.zeros(randL)
        scores = np.random.choice(llr,randL,)
        scores = np.sum(scores)
        if scores>0:
            haha[randL].append(scores)

means = np.zeros(len(haha))
C = []
for j,i in enumerate(haha):
    data = np.array(haha[i])
    mean = np.mean(data)
    means[j] = mean

haha = defaultdict(list)
for i in range(1,100):
    randL = i
    for i in range(10000):
        score = np.zeros(randL)
        scores = np.random.choice(dLLR,randL,)
        scores = np.sum(scores)
        if scores>0:
            haha[randL].append(scores)

means2 = np.zeros(len(haha))
C = []
for j,i in enumerate(haha):
    data = np.array(haha[i])
    mean = np.mean(data)
    means2[j] = mean

plt.figure(figsize=(6,5))
plt.plot(means,label='mean S - Repeat A Sim')
plt.plot(means2,label='mean S - Repeat D Sim')
#plt.plot(1/l*np.log(range(1,100)),label="$c * \log(x)$")
plt.plot(1.09*np.log(range(1,100)),label='$\log(x)$')
plt.xlim(0,100)
plt.xlabel('Hit Length')
plt.ylabel('Mean Score')
plt.legend()
```

```
[149]: <matplotlib.legend.Legend at 0x7f8e10e583d0>
```



Scoring matrix scaling parameter λ

mSEEKR allows for training on different queries and null models, and different values of k. Each of these choices will result in different log-odds between the k-mers. Stronger differences between the query and null will cause scores to increase at a faster rate, or vice versa. To compare significance regardless of the model chosen, a scaling factor is required. This is a parameter estimated in Altschul BLAST statistics – normalizes the probability distributions across different scoring methods (i.e. different queries). This means that the scores for difference queries can be **directly compared** if we multiply the scoring matrix by λ.

Solve the equation for λ

$$\sum_i p_i \exp[\lambda s_i] = 1$$

Where "exp" is e. This is a transcendental equation that must be solved using a numerical method. Below is code to use the bisection root finding method to solve for lambda.

HOWEVER – this was derived for the maximal scoring pair from an alignment… not just scores above a threshold (S>0)… so I'm not enough of a mathematician to know if the logic still applies. Empirical data (below) suggest it does.

Calculation Below is code to calculate λ using bisection root solving algorithm, as well as the same plot above, except with each queries log-odds scores multiplied by their respective value of λ. You can see the curves for the simulated hits for A and D queries are approximately equal, despite having different values of k and wildly different log-odds for those k-mers.

[124]:
```
def func(l,E,k):
    kmers = [''.join(group) for group in product('ATCG',repeat=k)]
    val = 1/(4**k)
    summation = 0
    for i in kmers:
        summation+=np.e**((E['+'][i]-E['-'][i])*l)
    return (val *summation)-1
```

Here I have defined a and b so that f(a) < 0 and f(b) > 0 for bisection method of root finding

[125]:
```
a = .00001
b = 1
```

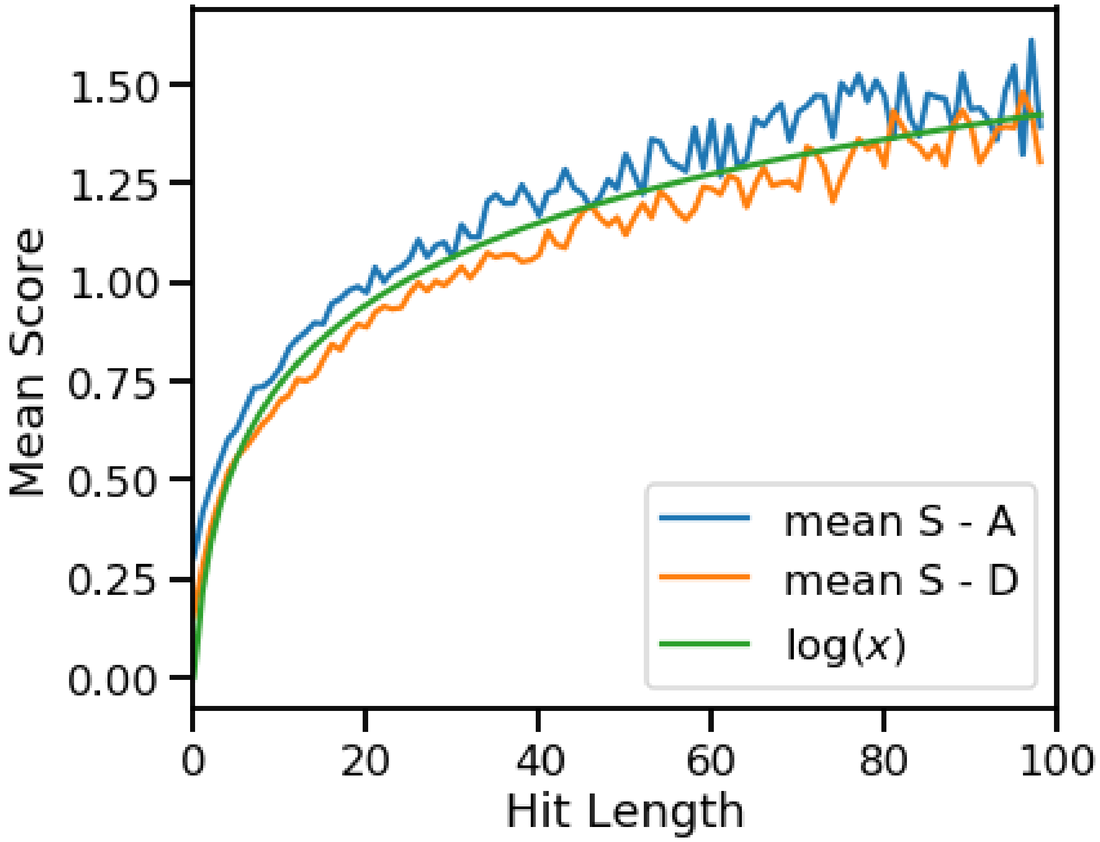
[126]:
```
def bisection(f,a,b,N,E,k):
    if f(a,E,k)*f(b,E,k) >= 0:
        return None
    a_n = a
    b_n = b
    for n in range(1,N+1):
        m_n = (a_n + b_n)/2
        f_m_n = f(m_n,E,k)
        if f(a_n,E,k)*f_m_n < 0:
            a_n = a_n
            b_n = m_n
        elif f(b_n,E,k)*f_m_n < 0:
            a_n = m_n
            b_n = b_n
        elif f_m_n == 0:
            print("Found exact solution.")
            return m_n
        else:
            print("Bisection method fails.")
            return None
    return (a_n + b_n)/2
```

[181]:
```
from collections import defaultdict
adjaLLR = llr*bisection(func,a,b,1000,E,4)
adjdLLR = dLLR*bisection(func,a,b,1000,eD['E'],2)


haha = defaultdict(list)
for i in range(1,100):
    randL = i
    for i in range(10000):
        score = np.zeros(randL)
        scores = np.random.choice(adjaLLR,randL,)
        scores = np.sum(scores)
        if scores>0:
            haha[randL].append(scores)

means = np.zeros(len(haha))
C = []
for j,i in enumerate(haha):
    data = np.array(haha[i])
    mean = np.mean(data)
    means[j] = mean

haha = defaultdict(list)
for i in range(1,100):
    randL = i
    for i in range(10000):
        score = np.zeros(randL)
        scores = np.random.choice(adjdLLR,randL,)
        scores = np.sum(scores)
        if scores>0:
            haha[randL].append(scores)

means2 = np.zeros(len(haha))
C = []
for j,i in enumerate(haha):
    data = np.array(haha[i])
    mean = np.mean(data)
    means2[j] = mean
```

```
Found exact solution.
Found exact solution.
```

```
[188]: plt.figure(figsize=(6,5))
       plt.plot(means,label='mean S - A')
       plt.plot(means2,label='mean S - D')
       #plt.plot(1/l*np.log(range(1,100)),label="$c * \log(x)$")
       plt.plot(.31*np.log(np.array(range(1,100))),label='$\log(x)$')
       plt.xlim(0,100)
       plt.xlabel('Hit Length')
       plt.ylabel('Mean Score')
       plt.legend()
```

```
[188]: <matplotlib.legend.Legend at 0x7f8e0426d850>
```



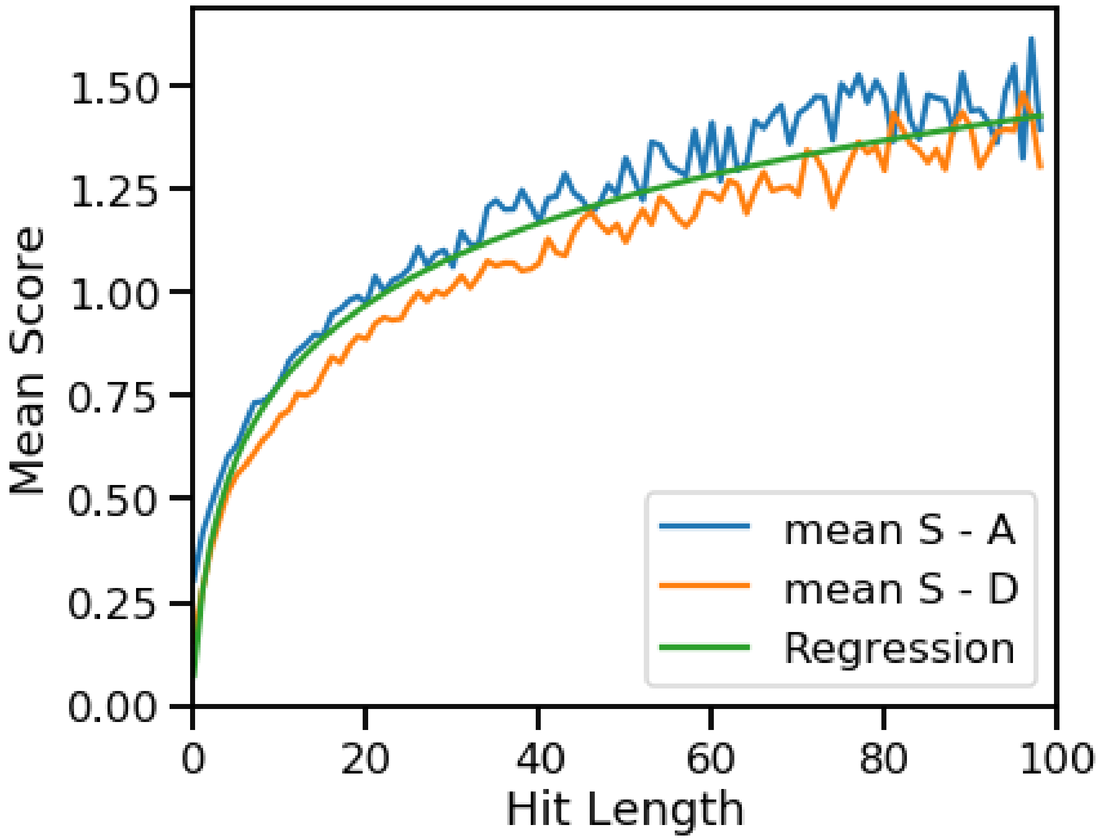
3.1 Regress to calculate the constant in front of log

In the above plot, I had to multiply log(x) by .31 to get the curve to scale correctly to the random score means. This is more-or-less another constant in the BLAST algorithm – this one is far less clear cut to calculate. Here, I will simply perform a regression against log(x) to find this constant.

[217]:
```
logfit = np.polyfit(np.log(np.linspace(1,99,99)), (means+means2)/2,1)
```

[219]:
```
plt.figure(figsize=(6,5))
plt.plot(means,label='mean S - A')
plt.plot(means2,label='mean S - D')
#plt.plot(1/l*np.log(range(1,100)),label="$c * \log(x)$")
plt.plot(logfit[0]*np.log(range(1,100))+logfit[1],label='Regression')
plt.xlim(0,100)
plt.xlabel('Hit Length')
plt.ylabel('Mean Score')
plt.legend()
```

[219]:
```
<matplotlib.legend.Legend at 0x7f8e040b5790>
```



[220]:
```
print(f'Coefficient = {logfit[0]}')
```

```
Coefficient = 0.2945597271026243
```

4 Fit distribution to 10^6 random hits

[354]:
```
haha = []
for i in range(1000000):
    randL = geom.rvs(p=.01)
    score = np.zeros(randL)
    scores = np.random.choice(llr,randL,)
    scores = np.sum(scores)
    if scores>0:
        haha.append((scores,randL))

# D = []
# for i in range(1000000):
#     randL = geom.rvs(p=.01)
#     score = np.zeros(randL)
#     scores = np.random.choice(dLLR,randL,)
#     scores = np.sum(scores)
#     if scores>0:
#         D.append((scores,randL))
```

4.1 Calculate bit score

This calculation converts the score to a bit score. The score is multiplied by λ (discussed above for interpretation), $c\log(x)$ is subtracted from this to account for the increase in expected score w.r.t. length as shown above, and then divided by ln 2 to convert to 'bits'.

[355]:
```
def bitscore(score,length,l):
    return ((l*score) - (logfit[0]*np.log(length))/np.log(2))
```

[356]:
```
test = []
Aadj = bisection(func,a,b,1000,E,4)
Dadj = bisection(func,a,b,1000,eD['E'],2)


for i in range(len(haha)):
    test.append(bitscore(haha[i][0],haha[i][1],Aadj))
test = np.array(test)

# dTest = []

# for i in range(len(D)):
#     dTest.append(bitscore(D[i][0],D[i][1],Dadj))
# dTest = np.array(dTest)
```

```
Found exact solution.
Found exact solution.
```

5 Fit a generalized pareto distribution

Finally, we can use the simulations of random mSEEKR hits as well as the parameters we have calculated above to fit a GP (generalized pareto distribution) to the distribution of bit scores calculated above. The GP distribution is generally used to model "exceedences", i.e. the probability of a value exceeding a threshold. This differs from blast and other alignment statistics, because they are modeling extreme values, i.e. chosing the maximum of a series of samples, and asking what the probability is of a maximal score being higher than some value (this follows an "extreme value distribution").

"The GP distribution can be defined constructively in terms of exceedances. Starting with a probability distribution whose right tail drops off to zero, such as the normal, we can sample random values independently from that distribution. If we fix a threshold value, throw out all the values that are below the threshold, and subtract the threshold off of the values that are not thrown out, the result is known as exceedances. The distribution of the exceedances is approximately a GP. Similarly, we can set a threshold in the left tail of a distribution, and ignore all values above that threshold. The threshold must be far enough out in the tail of the original distribution for the approximation to be reasonable." - MathWorks

Below, I fit a GP distribution to the simulated data above and then calculate some p-values for different bit-score thresholds. You can see the fit is near-perfect.

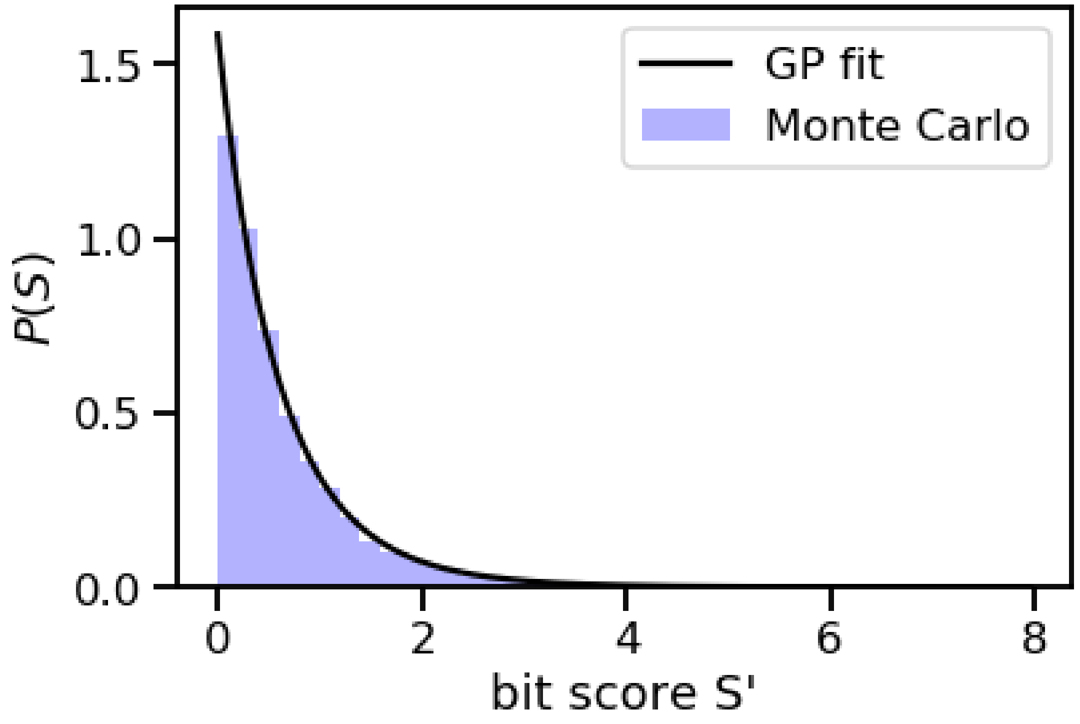
Crucially, this all can be done relatively quickly in train.py – meaning that the learned parameters for this GP distribution can simply be supplied to mSEEKR.py and p-vals quickly calculated

[357]:
```python
from scipy import stats
scores = test[test>0]
fit = stats.genpareto.fit(scores)
print(fit)
scores = sorted(scores)
genpareto=stats.genpareto.pdf(scores,*fit)
sns.set_context('talk')
plt.plot(scores,genpareto,label='GP fit',color='black')
plt.hist(scores,density=True,bins=40,color='b',alpha=.3,label='Monte Carlo')
plt.xlabel('bit score S\'')
plt.ylabel('$P(S)$')
plt.legend()
```

```
/Users/dan/opt/anaconda3/lib/python3.7/site-
packages/scipy/stats/_continuous_distns.py:2432: RuntimeWarning: divide by zero
encountered in true_divide
  val = val + cnk * (-1) ** ki / (1.0 - c * ki)
/Users/dan/opt/anaconda3/lib/python3.7/site-
packages/scipy/stats/_distn_infrastructure.py:1063: RuntimeWarning: invalid
value encountered in subtract
  mu2 = mu2p - mu * mu
/Users/dan/opt/anaconda3/lib/python3.7/site-
packages/scipy/stats/_distn_infrastructure.py:2407: RuntimeWarning: invalid
value encountered in double_scalars
  Lhat = muhat - Shat*mu
```

```
(0.07158379636823939, 2.656012500567077e-05, 0.6304214939970808)
```

```
[357]: <matplotlib.legend.Legend at 0x7f8e0a9b8110>
```



6 P(S'>s')

Probability of length normalized score is greater than some common values...

```
[329]: GP = stats.genpareto(*fit)
```

```
[340]: print(f'P(S>10) = {1-GP.cdf(10)}')
       print(f'P(S>20) = {1-GP.cdf(20)}')
       print(f'P(S>30) = {1-GP.cdf(30)}')
       print(f'P(S>40) = {1-GP.cdf(40)}')
```

```
P(S>10) = 2.0032027499450322e-05
P(S>20) = 3.9346412461505054e-08
P(S>30) = 4.84887241469778e-10
P(S>40) = 1.6194712237904696e-11
```

7 Challenges

Imagine the following scaled down version of an mSEEKR parse of "lncRNA X":

$$+ + - - + + - - + + - - - - - - - + + + + + + + + + + + + + + + + + - - - - - - - - -$$

Where + represents the query state and - represents the null state. Clearly, the long stretch of +'s at the right side of this hypothetical mSEEKR parse is a 'real hit' and lets suppose it has an extremely significant p-val (<E-100). But what about the $+ + - - + + - - + + - - \ldots$? If these ++'s have scores that hover right around the range of looking like statistical noise (score 20-30), could it still be a functional domain?

I.e., is the actual 'functional map'

$$[+ + - - + + - - ++] - - - - - - - + + + + + + + + + + + + + + + + + - - - - - - - -$$

Where the idea here being that the ++'s are close enough to each other to form a functional domain and recruit sufficient protein/have a functional purpose?

Are small fluctuations that could in principal be found in a randomly generated sequence still be functional?

8 Conclusions

I imagine that these results represent a good approximation of reality, with the caveat of being almost entirely empircal. mSEEKR essentially meets all the assumptions of BLAST – EXCEPT, BLAST samples the scores from the MAXIMUM of all possible alignments between sequence pairs, whereas mSEEKR merely samples above a (strong) threshold of 0.

```
[ ]:
```