STAT 243 Problem Set 6

Treves Li

2024-11-05

Collaboration Statement

I did not collaborate with anyone.

Question 1

> In class we said that in the exponent of the 8 byte floating point representation, $e \in 0, ..., 2047$, since e is represented with 11 bytes ($2^{11} = 2048$). So that would suggest that the largest and smallest magnitude numbers (ignoring the influence of the 52 bit in the mantissa) are 2^{1024} and 2^{-1023}.

Question 1a

> However, 2^{1024} overflows. Create a numpy `float64` number larger than 10^{308} that overflows and is represented as `inf`. What is its bit-wise representation? Why does it now make sense that we can't work with 2^{1024} as a regular number?
>
> Note that there is still a bit of a mystery here in that it doesn't appear that the sequence 0 followed by 63 1s (or simply 64 1s) is used in any way, even though those would be a natural candidate to represent `inf` (and `-inf`). Extra credit for investigating and figuring out why that is (it might be difficult to determine and I don't know the answer myself).

I created a "big float" using `np.float64()` with a value larger than 10^{308}. To view its 64-bit representation in binary, I used `.view(np.int64)` to interpret it as an integer, then printed it in IEEE 754 standard format.

```
import numpy as np

# Create float64 greater than 10^308
big_float = np.float64(1.0e+309)
print(big_float)

# Get binary repr
big_float_bits = big_float.view(np.int64)
print(f"{big_float_bits:064b}")
```

```
inf
0111111111110000000000000000000000000000000000000000000000000000
```

The output shows that we have maxed out the allowable bits for the exponent field. IEEE 754 `float64` format uses an exponent range up to 2^{1023}, and values exceeding this limit trigger an overflow.

This bit pattern (where the exponent field is all `1`s and the mantissa field is all `0`s) is reserved specifically for representing "infinity". This design choice means we can't use 2^{1024} as a finite number, since the all-`1`s exponent configuration uniquely represents "infinity". In light of this, it makes sense then that we can't use 2^{1024} as a "regular" number (at least for `float64`).

Question 1b

> What is the bitwise repr of 2^{-1022} Given that, what would the bit-wise repr of be 2^{-1023} (work this out conceptually, not by trying to use 2^{-1023} in Python)? What number does that bitwise representation actually represent?
>
> I asked that you not try to use 2^{-1023} in Python. Doing that and exploring what is going on is part (c).

We know that floating points in IEEE 754 double-precision have the following structure:

$$(-1)^s \times 2^{e-1023} \times 1.f$$

This means that we can represent 2^{-1022} by matching the exponent of 2:

$$e - 1023 = -1022 \quad \Longrightarrow \quad e = 1$$

So, to represent 2^{-1022}, we want the exponent portion to simply be 1 in binary, which we can write out as:

0 00000000001 0000000000000000000000000000000000000000000000000000

Given the example above, we expect the bitwise representation of 2^{-1023} to be where the exponent is 0 (while also keeping the sign and the mantissa 0):

0 00000000000 0000000000000000000000000000000000000000000000000000

To see what 2^{-1023} actually represents in decimal, we can convert the binary representation to a 64-bit unsigned integer using `numpy`, and see that is 0.

```
# Prefix with 0b because we're giving numpy a binary literal
bin_repr = 0b0000000000000000000000000000000000000000000000000000000000000000

bin2int = np.uint64(bin_repr)

bin2int.view(np.float64)
```

```
np.float64(0.0)
```

Question 1c

> Extra credit: By trial and error, find the base 10 representation of the smallest positive number that can be represented in Python. Hint: it's rather smaller than 1×10^{-308}. Explain how it can be that we can store a number smaller than 1×2^{-1022}, which is the value of the smallest positive number that we saw above. Start by looking at the bit-wise representation of 1×2^{-1023} and see it is not the same as what you worked out in part (b). Given the actual bit-wise representation of 1×2^{-1023}, show the progression of numbers smaller than that that can be represented exactly and show the smallest number that can be represented in Python written in both base 2 and base 10.
>
> Hint: you'll be working with numbers that are not normalized (i.e., denormalized); numbers that do not have 1 as the fixed number before the radix point in the floating point representation we discussed in Unit 8.

We start by looking at the bit-wise representation of 2^{-1023}:

```
# Convert 2**(-1023) to float, then to binary repr
bin_repr = np.float64(2**(-1023)).view(np.int64)
np.binary_repr(bin_repr, width=64)
```

```
'0000000000001000000000000000000000000000000000000000000000000000'
```

Indeed, it appears that this representation is not the same as the all-0s that I would have assumed in part (b).

The progression of numbers smaller than 2^{-1023} in 64-bit representation is what I would assume to be:

0 00000000000 0111111111111111111111111111111111111111111111111111

0 00000000000 0111111111111111111111111111111111111111111111111110

0 00000000000 0111111111111111111111111111111111111111111111111101

etc.

We can test this out to see that the values are indeed decreasing:

```
print(np.uint64(
  0b0000000000000011111111111111111111111111111111111111111111111111
).view(np.float64))

print(np.uint64(
  0b0000000000000011111111111111111111111111111111111111111111111110
).view(np.float64))

print(np.uint64(
  0b0000000000000011111111111111111111111111111111111111111111111101
).view(np.float64))
```

```
1.1125369292536e-308
1.1125369292535997e-308
1.112536929253599e-308
```

Based on this pattern, I would also assume that the smallest positive number would be represented by a singular `1` in the mantissa, which in binary representation is:

$$0 \quad 00000000000 \quad 0000000000000000000000000000000000000000000000000001$$

In decimal, this value corresponds to $2^{0-1023-(52-1)} = 2^{-1074}$

```
print(np.uint64(
  0b0000000000000000000000000000000000000000000000000000000000000001
).view(np.float64))
```

```
5e-324
```

It seems that we can represent these "subnormal" numbers smaller than 1×2^{-1022} **when the exponent is 0**. The developers may have designed the special format below (where the number before the radix point is 0 instead of 1) to enable computations of very small numbers that are very close to 0 without losing precision, while also allowing for gradual underflow:

$$(-1)^s \times 2^{e-1023} \times 0.f$$

Question 2

Consider the following estimates of the variance of a set of numbers. The results depend on whether the magnitude of the numbers is large or small. You can assume that for a vector `w`, $\text{Var}(w)$ is calculated as $\sum_{i=1}^{n}(w_i - \bar{w})^2/(n-1)$.

```
import numpy as np
rng = np.random.default_rng(seed = 1)
def dg(x, form = '.20f'):
  print(format(x, form))

z = rng.normal(size = 100)
x = z + 1e12
## Calculate the empirical variances
dg(np.var(z))
dg(np.var(x))
```

```
0.72514887009499828796
0.72514631554484365594
```

Explain why these two estimates agree to only a small number of decimal places and which of the two is the more accurate answer, when mathematically the variance of `z` and the variance of `x` are exactly the same (since `x` is just the addition of a constant to `z`). How many digits of accuracy do you expect in the less accurate of the two?

Even though the variance of `x` and `z` are mathematically the same, the addition of the large constant `1e12` to `z` overwhelms its smaller values, leading to reduced precision and less accuracy in the variance calculation of `x`.

For `z`, we can expect to have 16 digits of accuracy in double-precision floating-point. However, adding `1e12` to each element in `z` diminishes the precision of `x` due to the much larger constant, causing most of the digits after the decimal point in `x` to lose meaningfulness. Below, I've denoted non-meaningful digits (rounding errors) as `X`s:

					.	5811	1810	4196	3531
+	1	0000	0000	0000	.	000X			
	1	0000	0000	0000	.	5811	XXXX	XXXX	XXXX

This would make `x` less accurate than `z`. For `z`, we can expect approximately 16 accurate digits, but in `x`, adding `1e12` leaves only about **4 to 5 meaningful digits** after the decimal point.

```
dg(z[7])
dg(x[7])
```

```
0.58111810419635312464
1000000000000.58117675781250000000
```

We can also consider **catastrophic cancellation**, but since the variance calculation subtracts the mean from each value $(x_i - \bar{x})^2$, we should be avoiding catastrophic cancellation (we are no long directly subtracting two very similar large numbers).

Question 3

Consider the following, in which we run into problems when trying to calculate on a computer. Suppose I want to calculate a predictive density for new data (e.g., in a model comparison in a Bayesian context):

$$f(y^*|y,x) = \int f(y^*|y,x,\theta)\pi(\theta|y,x)d\theta = E_{\theta|y,x}f(y^*|y,x,\theta)$$

Here $\pi(\theta|y,x)$ is the posterior distribution (the distribution of the parameter, θ, given the data, y, and predictors, x). All of θ, y, and x will generally be vectors.

If we have a set of samples θ for from the posterior distribution, $\theta_j \sim \pi(\theta|y,x)$, $j = 1,...,m$, we can estimate that quantity for a vector of conditionally IID observations using a Monte Carlo estimate of the expectation:

$$f(y^*|y,x) \approx \frac{1}{m}\sum_{j=1}^{m}\prod_{i=1}^{n} f(y_i^*|y,x,\theta_j)$$

Question 3a

Explain why I should calculate the product in the equation above on the log scale. What is likely to happen if I just try to calculate it directly?

Since we are taking the products of small numbers (since the probabilities will be between 0 and 1 in a Bayesian context), the resulting values will be even smaller. This can lead to **underflow**, where we exceed the accuracy of floating-point precision.

By calculating the product on a log scale, we can sum the log-transformed values instead of directly multiplying small probabilities. This approach keeps the values at a more reasonable magnitude, ensuring numerical stability. If needed, we can exponentiate at the end of our calculations to return to the original scale.

Question 3b

Here's a re-expression, using the log scale for the inner quantity,

$$\frac{1}{m}\sum_{i=1}^{m}\exp\left(\sum_{i=1}^{n}\log f(y_i^*|y,x,\theta_j)\right)$$

which can be re-expressed as

$$\frac{1}{m}\sum_{j=1}^{m}\exp(v_j)$$

where

$$v_j = \sum_{i=1}^{n}\log f(y_i^*|y,x,\theta_j)$$

What is likely to happen when I try to exponentiate v_j?

If, after summing all the log values, v_j becomes a very big number, then exponentiating v_j may result in **overflow**, where it exceeds the maximum representable value in floating-point and get some invalid value or simply infinity. This, of course, is also dependent on the number of observations n, since more observations can lead to larger sums in v_j, increasing the risk of overflow.

To avoid this situation, we can try to subtract a number similar in magnitude to v_j (like the mean or max of v_j, for example) so that we can normalise the values to within a manageable range.

Question 3c

Consider the log predictive density,

$$\log f(y^*|y,x) \approx \log\left(\frac{1}{m}\sum_{j=1}^{m}\exp(v_j)\right)$$

Figure out how you could calculate this log predictive density without running into the issues discussed in parts (a) and (b).

Hint: recall that with the logistic regression example in class, we scaled the problematic expression to remove the numerical problem. Here you can do something similar with the $\exp(v_j)$ terms, though at the end of the day you'll only be able to calculate the log of the predictive density and not the predictive density itself.

By calculating the product on a log scale, we can sum the log-transformed values instead of directly multiplying small probabilities. This approach keeps the values at a more reasonable magnitude, ensuring numerical stability and avoiding the underflow issues discussed in part (a).

Furthermore, as discussed in part (b), we can subtract a number similar in magnitude to vjvj (such as the maximum value $v_{\max}$ from the m samples) to normalize the values within a manageable range, thereby preventing the numerical issues due to overflow. Instead of calculating $\exp(v_j)$, we compute $\exp(v_j - v_{\max})$. At the end of our calculation, we can simply add $v_{\max}$ back in to return to the original scale:

$$\log f(y^*|y,x) \approx \log\left(\frac{1}{m}\sum_{j=1}^{m}\exp(v_j - v_{\max})\right) + v_{\max}$$

Question 4

Experimenting with importance sampling.

Question 4a

Use importance sampling to estimate the mean (i.e., $\phi = E_f X$) of a truncated t distribution with 3 degrees of freedom, truncated such that $X < -4$. Have your sampling density be a normal distribution centered at -4 and then truncated so you only sample values less than -4 (this is called a half-normal distribution). You should be able to do this without discarding any samples (how?). Use $m = 10000$ samples. Create histograms of the weights $f(x)/g(x)$ and the summand $h(x)f(x)/g(x)$ to get a sense for whether $\text{Var}(\hat{\phi})$ is large. Note if there are any extreme weights that would have a very strong influence on ϕ. Estimate $\text{Var}(\hat{\phi})$. Hint: remember that your $f(x)$ needs to be appropriately normalized or you need to adjust the weights per the class notes. For comparison, based on using numerical integration, which is feasible in this simple one-dimensional case but increasingly infeasible in higher dimensions, the mean is -6.216.

We know from the course notes that $\hat{\phi} = \frac{1}{m} \sum_i h(x_i) \frac{f(x_i)}{g(x_i)}$.

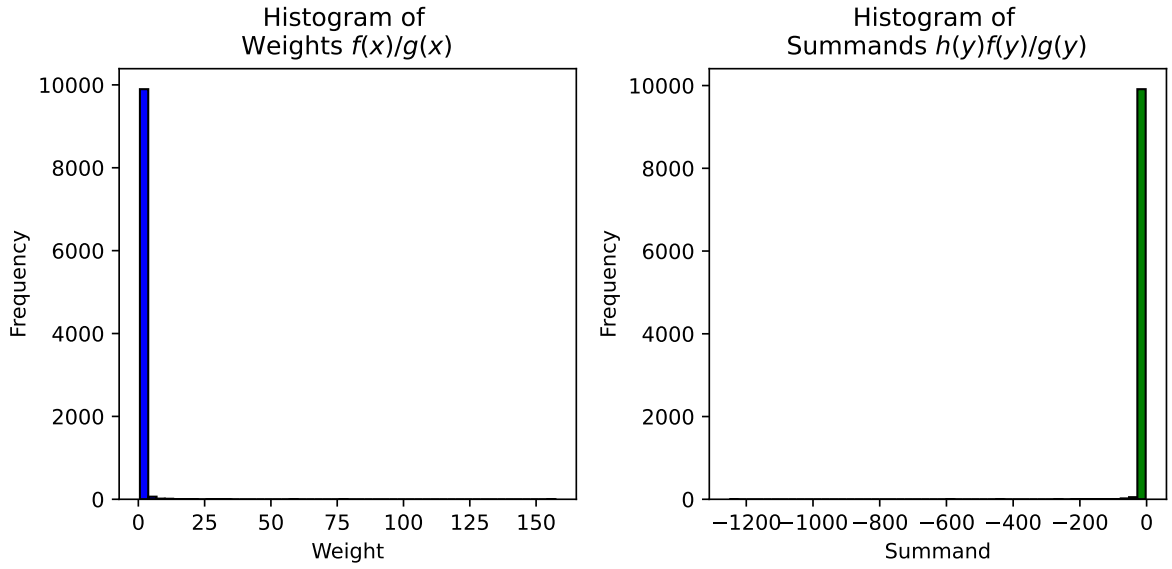
We can use that with the given information in the Python code below, using the `scipy.stats` package for the t and normal distributions. Note that I normalised the distributions by dividing them with their respective cumulative distribution functions.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import t, norm

np.random.seed(24)

# Parameters
m = 10000
trunc_limit = -4
dof = 3  # for the t-dist

# Target dist f(x) is truncated t-dist
def f(x):
    return t.pdf(x, df=dof) / t.cdf(trunc_limit, df=dof)
```

```
# Sampling dist g(x) is truncated normal dist
def g(x):
    return norm.pdf(x, loc=-4) / norm.cdf(trunc_limit, loc=-4)

# Generate samples from the truncated normal dist
# Use reflection trick
def sample_truncated_normal(size):
    samples = []
    while len(samples) < size:
      sample = np.random.normal(loc=-4, scale=1)

      if sample < trunc_limit:
        samples.append(sample)

    return np.array(samples)

# Generate samples
h_samples = sample_truncated_normal(m)

# Calculate weights
weights = f(h_samples) / g(h_samples)

# Get mean and variance estimates
mean_est = (1/m) * np.sum(h_samples * weights)
var_est = (1/m) * np.var(h_samples * weights)

# Print results
print(f"Estimated mean:     {mean_est}")
print(f"Estimated variance: {var_est}")

plt.figure(figsize=(8, 4))

# Plot weights
plt.subplot(1, 2, 1)
plt.hist(weights, bins=50, color='b', edgecolor='k')
plt.title('Histogram of \nWeights $f(x)/g(x)$')
plt.xlabel('Weight')
plt.ylabel('Frequency')

# Plot summands
plt.subplot(1, 2, 2)
weighted_summand = h_samples * weights
```

```
plt.hist(weighted_summand, bins=50, color='g', edgecolor='k')
plt.title('Histogram of \nSummands $h(y)f(y)/g(y)$')
plt.xlabel('Summand')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```

```
Estimated mean:     -4.424484684271216
Estimated variance: 0.025455553108134053
```



The IS method estimated a mean of approximately -4.424, differing from the numerical integration mean of -6.216. This discrepancy suggests that the current sampling distribution, a truncated normal distribution, does not fully capture the heavy tail of the truncated t-distribution, leading to extreme weights that skew the mean estimate.

The histogram of weights $\frac{f(x)}{g(x)}$ shows that most weights are stable, but a few large outliers disproportionately affect the mean estimate $\hat{\phi}$, indicating a mismatch between the normal sampling distribution and the heavy tail of the target t-distribution. Although the variance estimate appears low, the histograms suggest that the true variability is likely underestimated due to the influence of these outliers, which can lead to inaccurate mean estimates. To improve the result, a sampling distribution that better aligns with the target distribution's tail would be needed.

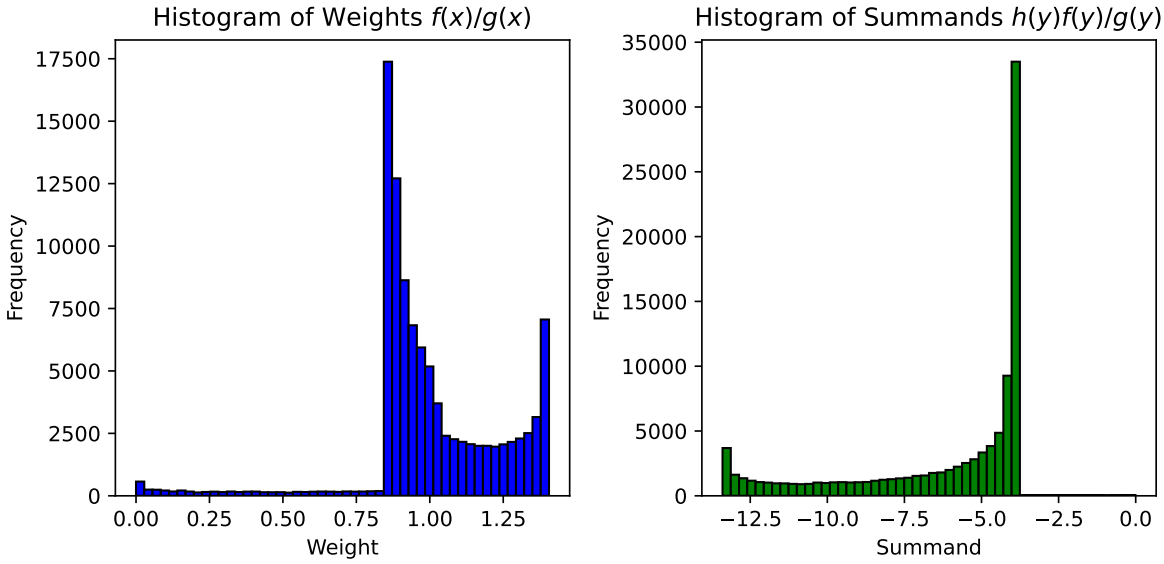
Question 4b

Now use importance sampling to estimate the mean of the same truncated t distribution with 3 degrees of freedom, truncated such that $X < -4$, but have your sampling density be a t distribution, with 1 degree of freedom (not 3), centered at -4 and truncated so you only sample values less than -4. Again you shouldn't have to discard any samples. Respond to the same questions as above in part (a). In addition, compute a 95% (simulation) uncertainty interval for your estimate, using the Monte Carlo simulation error, $\sqrt{\widehat{\text{Var}}(\hat{\phi})}$.

```
np.random.seed(24)

# Parameters
m = 100_000
trunc_limit = -4
dof_target = 3    # target dist
dof_sampling = 1  # sampling dist

# Target dist f(x): truncated t-dist
def f(x):
  return t.pdf(x, df=dof_target) / t.cdf(trunc_limit, df=dof_target)

# Sampling dist g(x): truncated t-dist with 1 df
# Update denominator to account for truncation
def g(x):
  return t.pdf(x + 4, df=dof_sampling) / t.cdf(0, df=dof_sampling)

# Generate samples from t-dist with 1 dof
def sample_truncated_t(size):
  samples = []
  while len(samples) < size:
      # Sample from a t-dist with 1 df
      sample = np.random.standard_t(dof_sampling)
      sample += trunc_limit  # centre sample at -4

      if sample < trunc_limit:
        samples.append(sample)

  return np.array(samples)
```

```
# Generate samples
h_samples = sample_truncated_t(m)

# Calculate weights
weights = f(h_samples) / g(h_samples)

# Get mean and variance estimates
mean_est = (1/m) * np.sum(h_samples * weights)
var_est = (1/m) * np.var(h_samples * weights)

# Calculate Monte Carlo simulation error
sim_uncertainty = np.sqrt(var_est)

# Calculate 95% CI
lower_bound = mean_est - 1.96 * sim_uncertainty
upper_bound = mean_est + 1.96 * sim_uncertainty

# Print results
print(f"Estimated mean:           {mean_est}")
print(f"Estimated variance:       {var_est}")
print(f"Simulation error:         {sim_uncertainty}")
print(f"95% Uncertainty interval: ({lower_bound:5f}, {upper_bound:5f})")

plt.figure(figsize=(8, 4))

# Plot weights
plt.subplot(1, 2, 1)
plt.hist(weights, bins=50, color='b', edgecolor='k')
plt.title('Histogram of Weights $f(x)/g(x)$')
plt.xlabel('Weight')
plt.ylabel('Frequency')

# Plot summands
plt.subplot(1, 2, 2)
weighted_summand = h_samples * weights
plt.hist(weighted_summand, bins=50, color='g', edgecolor='k')
plt.title('Histogram of Summands $h(y)f(y)/g(y)$')
plt.xlabel('Summand')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```

```
Estimated mean:            -6.212535929086711
Estimated variance:        9.254311531715418e-05
Simulation error:          0.009619933228310588
95% Uncertainty interval: (-6.231391, -6.193681)
```



With a t-distribution of 1 degree of freedom, we got an estimated mean of -6.213, which is much closer to the true mean of -6.216. Additionally, the variance estimate of `9.25e5` is much smaller than in part a, showing improved precision. The simulation error of 0.0096 further supports the greater accuracy and stability of this estimate. The 95% uncertainty interval of $[-6.231, -6.194]$ shows that the estimate is statistically consistent with the true mean, and within the bounds of the Monte Carlo simulation error.

The histogram of weights $\frac{f(x)}{g(x)}$ shows fewer extreme values than in part a, suggesting that the impact of outliers on the mean estimate has been reduced. This is further supported by the summand histogram, which shows a more stable spread of values without significant outliers. This stability likely contributes to the lower variance observed in this case.

By using the t-distribution with 1 degree of freedom as the sampling density, the sampling process is better aligned with the target distribution's left tail (since the truncated target distribution has its mass concentrated near the left tail). This "alignment" helps mitigate the influence of extreme values in the tail, resulting in a more stable and accurate estimation of the mean. The 1-dof t-distribution's heavier tails appear to match the truncated t-distribution more closely, leading to improved sampling efficiency and a reduction in the variance of the estimator.