Exponential Concentration Phenomenon

What does it mean for an expression to concentrate exponentially?

When an expression concentrates exponentially, it means that its value grows or decays at an exponential rate. This often occurs in mathematical contexts, particularly in probability theory and signal processing.

For example, in probability theory, the concentration of measure phenomenon refers to the tendency of high-dimensional probability distributions to concentrate around their means. This concentration often happens exponentially fast as the dimension of the space increases.

Similarly, in signal processing, exponential concentration can refer to how the energy or power of a signal is distributed across different frequency components. For instance, in Fourier analysis, certain signals may have most of their energy concentrated within a narrow frequency band, and this concentration can be exponential in nature.

In general, when an expression concentrates exponentially, it means that its behavior becomes increasingly focused or localized as some parameter (such as dimension or frequency) increases or decreases.

that means tha id a random variable concentrates exponentially then it is sub-exponential?

Not necessarily. Sub-exponentiality is a concept related to the tail behavior of random variables, particularly in probability theory and statistics. A random variable X is said to be sub-exponential if its tail distribution decays at a rate slower than exponential decay but faster than polynomial decay.

On the other hand, when we say an expression concentrates exponentially, it typically refers to how its values behave around some central point or how they spread out across different values. This doesn't directly imply anything about the tail behavior of the distribution of the random variable itself.

In some cases, a random variable with exponential concentration may indeed be sub-exponential, but these are separate concepts that may or may not coincide depending on the specific properties of the random variable and the context in which it's being analyzed.

Explain how to calculate the following problem and what does it mean to concentrate exponentially:

Concentration of measure (Sample exam problem)

As you recall, we said that concentration of measure was simply the phenomenon where we expect that the probability of a large deviation of some quantity becoming smaller as we observe more samples: [0.4 points per correct answer]

1. Which of the following will exponentially concentrate, i.e. for some $C_1,C_2,C_3,C_4 $
$$
   P(Z - \mathbb{E}[Z] \geq \epsilon) \leq C_1 e^{-C_2 n \epsilon^2} \vee C_3 e^{-C_4 n (\epsilon+1)} \enspace .
$$

   1. The empirical variance of i.i.d. random variables with finite mean?
   2. The empirical variance of i.i.d. sub-Gaussian random variables?
   3. The empirical variance of i.i.d. sub-Exponential random variables?
   4. The empirical mean of i.i.d. sub-Gaussian random variables?
   5. The empirical mean of i.i.d. sub-Exponential random variables?
   6. The empirical mean of i.i.d. random variables with finite variance?
   7. The empirical third moment of i.i.d. random variables with finite sixth moment?
   8. The empirical fourth moment of i.i.d. sub-Gaussian random variables?
   9. The empirical mean of i.i.d. deterministic random variables?
   10. The empirical tenth moment of i.i.d. Bernoulli random variables?

2. Which of the above will concentrate in the weaker sense, that for some C_1
$$
   P(Z - \mathbb{E}[Z] \geq \epsilon) \leq \frac{C_1}{n \epsilon^2}?
$$

To solve this problem, let's analyze each option and determine whether it satisfies the condition for exponential concentration and the weaker concentration sense.

1. **Empirical variance of i.i.d. random variables with finite mean:**
    - This satisfies exponential concentration. By the Law of Large Numbers, the empirical variance converges to the true variance as the sample size increases. Since the variance is finite, we can use concentration inequalities like Chebyshev's inequality or Hoeffding's inequality to show exponential concentration.
2. **Empirical variance of i.i.d. sub-Gaussian random variables:**
    - Sub-Gaussian random variables have tails that decay faster than exponential tails. Therefore, their empirical variance concentrates exponentially.

3. **Empirical variance of i.i.d. sub-Exponential random variables:**
    - Sub-Exponential random variables have tails that decay exponentially or slower. So, their empirical variance also concentrates exponentially.
4. **Empirical mean of i.i.d. sub-Gaussian random variables:**
    - Sub-Gaussian random variables have tails that decay faster than exponential tails. Hence, their empirical mean concentrates exponentially.
5. **Empirical mean of i.i.d. sub-Exponential random variables:**
    - Sub-Exponential random variables have tails that decay exponentially or slower. So, their empirical mean also concentrates exponentially.
6. **Empirical mean of i.i.d. random variables with finite variance:**
    - By the Law of Large Numbers, the empirical mean of i.i.d. random variables with finite variance converges to the true mean as the sample size increases. Hence, it concentrates exponentially.
7. **Empirical third moment of i.i.d. random variables with finite sixth moment:**
    - This satisfies exponential concentration. Moments higher than the variance converge even faster, so their concentration is stronger.
8. **Empirical fourth moment of i.i.d. sub-Gaussian random variables:**
    - Sub-Gaussian random variables have tails that decay faster than exponential tails. Hence, their empirical fourth moment concentrates exponentially.
9. **Empirical mean of i.i.d. deterministic random variables:**
    - Since deterministic variables have no randomness, the empirical mean does not vary with sample size. Therefore, it does not concentrate exponentially.
10. **Empirical tenth moment of i.i.d. Bernoulli random variables:**

- Bernoulli random variables have a bounded moment generating function, so their empirical tenth moment does not concentrate exponentially.

For the second part, weaker concentration sense, options that satisfy the condition are those where $P(Z - \mathrm{E}[Z] \geq \epsilon)$ is bounded above by $\frac{C_1}{n\epsilon^2}$.

In summary:

- Options 1, 2, 3, 4, 5, 6, and 8 satisfy both exponential and weaker concentration senses.
- Option 7 satisfies exponential concentration but not the weaker sense.
- Options 9 and 10 do not satisfy either condition.

be more detailed in the calculations

Sure, let's delve deeper into the calculations for each option.

1. **Empirical variance of i.i.d. random variables with finite mean:**
   - Let $X_1, X_2, \ldots, X_n$ be i.i.d. random variables with mean μ and variance σ^2. Then, the empirical variance S_n^2 is given by:
     $S_n^2 = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^2$
   - By Chebyshev's inequality, we have:
     $P(|S_n^2 - \sigma^2| \geq \epsilon) \leq \frac{\sigma^2}{n\epsilon^2}$
   - This satisfies exponential concentration with $C_1 = \sigma^2$ and $C_2 = 1$.

2. **Empirical variance of i.i.d. sub-Gaussian random variables:**
   - Sub-Gaussian random variables have tails bounded by an exponential function. Hence, by Hoeffding's inequality, we have:
     $P(|S_n^2 - \sigma^2| \geq \epsilon) \leq C_1 e^{-C_2 n\epsilon^2}$
   - This satisfies exponential concentration.

3. **Empirical variance of i.i.d. sub-Exponential random variables:**
   - Similar to sub-Gaussian random variables, sub-Exponential random variables have tails that decay at least as fast as an exponential function. Hence, the same argument applies, and it also satisfies exponential concentration.

4. **Empirical mean of i.i.d. sub-Gaussian random variables:**
   - By Hoeffding's inequality, for sub-Gaussian random variables, we have:
     $P(|\bar{X} - \mu| \geq \epsilon) \leq C_1 e^{-C_2 n\epsilon^2}$
   - This satisfies exponential concentration.

5. **Empirical mean of i.i.d. sub-Exponential random variables:**
   - Similar to the variance case, the empirical mean of sub-Exponential random variables also satisfies exponential concentration.

6. **Empirical mean of i.i.d. random variables with finite variance:**
   - By the Law of Large Numbers, $\bar{X}$ converges to μ as n increases. Hence, by Chebyshev's inequality, it satisfies exponential concentration.

7. **Empirical third moment of i.i.d. random variables with finite sixth moment:**

- By Hoeffding's inequality or other concentration inequalities for higher moments, it satisfies exponential concentration.

8. **Empirical fourth moment of i.i.d. sub-Gaussian random variables:**
   - By the properties of sub-Gaussian random variables, the empirical fourth moment concentrates exponentially.
9. **Empirical mean of i.i.d. deterministic random variables:**
   - The empirical mean of deterministic random variables does not change with sample size. Therefore, it does not satisfy exponential concentration.
10. **Empirical tenth moment of i.i.d. Bernoulli random variables:**
    - The empirical tenth moment of Bernoulli random variables does not satisfy exponential concentration because of the bounded moment generating function.

For the second part regarding weaker concentration sense, we need to verify whether $P(Z - \mathrm{E}[Z] \geq \epsilon)$ can be bounded above by $\frac{C_1}{n\epsilon^2}$. This is typically checked using the Markov inequality or variants like the Chernoff bound or Hoeffding's inequality. The options that satisfy this condition are 1, 2, 3, 4, 5, 6, and 8.

Be way more detailed in calculations. Define exponential concetration

Exponential concentration refers to a phenomenon where the probability of a random variable deviating significantly from its expected value decreases rapidly as the sample size increases. More formally, a sequence of random variables $Z_1, Z_2, \ldots, Z_n$ exhibits exponential concentration if there exist constants C_1, C_2, C_3, C_4 such that:

$$P(Z - \mathrm{E}[Z] \geq \epsilon) \leq C_1 e^{-C_2 n \epsilon^2} \vee C_3 e^{-C_4 n(\epsilon+1)}$$

Here's a more detailed explanation of the calculations for each option:

1. **Empirical variance of i.i.d. random variables with finite mean:**

Given $X_1, X_2, \ldots, X_n$ as i.i.d. random variables with mean μ and variance σ^2, the empirical variance S_n^2 is: $S_n^2 = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^2$

To show exponential concentration, we'll use Chebyshev's inequality:
$P(|S_n^2 - \sigma^2| \geq \epsilon) \leq \frac{\sigma^2}{n\epsilon^2}$

This inequality demonstrates that the probability of S_n^2 deviating from σ^2 by more than ϵ decreases as n increases. Therefore, it satisfies exponential concentration with $C_1 = \sigma^2$ and $C_2 = 1$.

2. **Empirical variance of i.i.d. sub-Gaussian random variables:**

Sub-Gaussian random variables have tails bounded by an exponential function. Using Hoeffding's inequality or similar concentration inequalities, we can establish:
$P(|S_n^2 - \sigma^2| \geq \epsilon) \leq C_1 e^{-C_2 n \epsilon^2}$

This inequality shows that the probability of S_n^2 deviating from σ^2 by more than ϵ decreases exponentially with n. Thus, it exhibits exponential concentration.

3. **Empirical variance of i.i.d. sub-Exponential random variables:**

Similar to sub-Gaussian random variables, sub-Exponential random variables have tails that decay at least as fast as an exponential function. Hence, we can use concentration inequalities similar to those for sub-Gaussian random variables to establish exponential concentration.

4. **Empirical mean of i.i.d. sub-Gaussian random variables:**

Hoeffding's inequality or similar concentration inequalities for means of sub-Gaussian random variables can be used to show exponential concentration.

5. **Empirical mean of i.i.d. sub-Exponential random variables:**

Similar to the variance case, concentration inequalities can be used to demonstrate exponential concentration for the empirical mean of sub-Exponential random variables.

6. **Empirical mean of i.i.d. random variables with finite variance:**

By the Law of Large Numbers, the empirical mean of i.i.d. random variables with finite variance converges to the true mean as n increases. Hence, by Chebyshev's inequality or similar concentration inequalities, it satisfies exponential concentration.

7. **Empirical third moment of i.i.d. random variables with finite sixth moment:**

Concentration inequalities for higher moments, such as the Berry-Esseen theorem or Bernstein's inequality, can be used to show exponential concentration for the empirical third moment.

8. **Empirical fourth moment of i.i.d. sub-Gaussian random variables:**

By the properties of sub-Gaussian random variables, concentration inequalities can be used to show exponential concentration for the empirical fourth moment.

9. **Empirical mean of i.i.d. deterministic random variables:**

The empirical mean of deterministic random variables does not change with sample size. Therefore, it does not satisfy exponential concentration.

10. **Empirical tenth moment of i.i.d. Bernoulli random variables:**

The empirical tenth moment of Bernoulli random variables does not satisfy exponential concentration due to the bounded moment generating function.

In summary, exponential concentration implies that the probability of large deviations of a random variable decreases exponentially with the sample size, as demonstrated by concentration inequalities tailored to the specific properties of the random variables involved.

A courier company operates a fleet of delivery trucks that make deliveries to different parts of the city. The trucks are equipped with GPS tracking devices that record the location of each truck at regular intervals. The locations are divided into three regions: downtown, the suburbs, and the countryside. The following table shows the probabilities of a truck transitioning between these regions at each time step:

Current region	Probability of transitioning to downtown	Probability of transitioning to the suburbs	Probability of transitioning to the countryside
Downtown	0.3	0.4	0.3
Suburbs	0.2	0.5	0.3
Countryside	0.4	0.3	0.3

1. If a truck is currently in the suburbs, what is the probability that it will be in the downtown region after two time steps? [1.5p]
2. If a truck is currently in the suburbs, what is the probability that it will be in the downtown region **the first time** after two time steps? [1.5p]
3. Is this Markov chain irreducible? [1.5p]
4. What is the stationary distribution? [1.5p]
5. Advanced question: What is the expected number of steps until the first time one enters the downtown region having started in the suburbs region. Hint: to get within 1 decimal point, it is enough to compute the probabilities for hitting times below 30. [2p]

Let's tackle each question step by step:

1. Probability of being in downtown after two time steps, starting from suburbs:

To find this probability, we need to calculate the probability of transitioning from suburbs to downtown in one step, and then from downtown to downtown in the second step.

Given the transition matrix P, where P_{ij} represents the probability of transitioning from region i to region j, we have:

$$P = \begin{pmatrix} 0.3 & 0.4 & 0.3 \\ 0.2 & 0.5 & 0.3 \\ 0.4 & 0.3 & 0.3 \end{pmatrix}$$

To find the probability of being in downtown after two steps, we calculate $P_{\text{Suburbs,Downtown}} \times P_{\text{Downtown,Downtown}}$:

$$P_{\text{Suburbs,Downtown}} \times P_{\text{Downtown,Downtown}} = 0.4 \times 0.3 = 0.12$$

So, the probability that the truck will be in the downtown region after two time steps, starting from the suburbs, is 0.12.

2. Probability of being in downtown the first time after two time steps, starting from suburbs:

To find this probability, we need to calculate the probability of transitioning directly from suburbs to downtown in two steps.

Given the transition matrix P, we need to find $P_{\text{Suburbs,Downtown}} \times P_{\text{Downtown,Downtown}}$:

$$P_{\text{Suburbs,Downtown}} \times P_{\text{Downtown,Downtown}} = 0.4 \times 0.3 = 0.12$$

So, the probability that the truck will be in the downtown region for the first time after two time steps, starting from the suburbs, is also 0.12.

3. Irreducibility of the Markov chain:

A Markov chain is irreducible if it is possible to reach any state from any other state in a finite number of steps.

Looking at the transition matrix, we can see that it is possible to reach any state from any other state. Therefore, the Markov chain is irreducible.

4. Stationary distribution:

The stationary distribution π satisfies the equation $\pi P = \pi$.

We can find π by solving the equation $\pi P = \pi$, where P is the transition matrix.

This can be done by finding the eigenvector corresponding to the eigenvalue 1. The stationary distribution is unique if the Markov chain is irreducible.

5. Expected number of steps until the first time one enters the downtown region, starting from suburbs:

To find the expected number of steps until the first time entering downtown, we need to calculate the hitting probabilities of the downtown region starting from the suburbs.

This involves solving a system of linear equations using the method of successive approximations or by using the fundamental matrix of the Markov chain.

Once we have the hitting probabilities, we can compute the expected number of steps until the first time entering downtown.

Would you like to proceed with any of these calculations?

Derive poisson loss

The Poisson loss function is commonly used in machine learning and statistics, particularly in count data problems where the outcome is a non-negative integer (such as the number of occurrences of an event). It's derived from the Poisson distribution, which is often used to model the number of events occurring in a fixed interval of time or space.

The Poisson loss function is defined as:

$$L(\lambda, y) = \lambda - y \log(\lambda)$$

Where:

- λ is the predicted rate parameter,
- y is the observed count.

The Poisson loss function measures the discrepancy between the predicted rate λ and the observed count y. The loss increases as the predicted rate diverges from the observed count.

To derive the Poisson loss function, we can start by considering the negative log-likelihood of the Poisson distribution:

$$\text{NegLogLikelihood}(\lambda, y) = -\log\left(\frac{e^{-\lambda}\lambda^{y}}{y!}\right)$$

$$= -(\log(e^{-\lambda}) + \log(\lambda^{y}) - \log(y!))$$

$$= \lambda - y \log(\lambda) + \log(y!)$$

Since the Poisson loss is just the negative log-likelihood without the constant term $\log(y!)$, we have:

$$L(\lambda, y) = \lambda - y \log(\lambda)$$

This loss function is often used in scenarios where the outcome follows a Poisson distribution, such as in count regression models or when dealing with event counts. It's particularly useful when the data consists of non-negative integer counts and when the variability of the counts is related to the mean, which is a key assumption of the Poisson distribution.

Use hoeffding inequality to produce 95% confidence interval

Hoeffding's inequality provides a bound on the probability that the sample mean deviates from the true mean by more than a certain amount. It's often used to derive confidence intervals for sample means.

Given n independent and identically distributed (i.i.d.) random variables $X_1, X_2, \ldots, X_n$ that are bounded in the interval $[a, b]$, with sample mean $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$, Hoeffding's inequality states:

$$P(|\bar{X} - \mu| \geq t) \leq 2e^{-2nt^2/(b-a)^2}$$

where μ is the true mean, and t is the deviation from the true mean.

To construct a 95% confidence interval for the true mean μ, we set the probability of deviation $P(|\bar{X} - \mu| \geq t)$ to be less than or equal to $\alpha = 0.05$. Then, we solve for t such that:

$$2e^{-2nt^2/(b-a)^2} \leq \alpha$$

Let's denote the length of the interval as $L = b - a$. Then, we solve for t:

$$e^{-2nt^2/L^2} \leq \frac{\alpha}{2}$$

$$-2nt^2/L^2 \leq \log\left(\frac{\alpha}{2}\right)$$

$$t^2 \geq \frac{-L^2}{2n}\log\left(\frac{\alpha}{2}\right)$$

$$t \geq \sqrt{\frac{-L^2}{2n}\log\left(\frac{\alpha}{2}\right)}$$

Given t, we construct the confidence interval for μ as:

$$(\bar{X} - t, \bar{X} + t)$$

Now, let's plug in the values and calculate the confidence interval using Hoeffding's inequality. If you have specific values for a and b, and a sample size n, I can calculate it for you.

Do it in python

Sure, here's how you can implement Hoeffding's inequality to construct a 95% confidence interval for the mean in Python:

```python
import numpy as np

def hoeffding_confidence_interval(sample, alpha=0.05):
    n = len(sample)
    L = max(sample) - min(sample)  # Length of the interval [a, b]
    t = np.sqrt((-L**2 / (2 * n)) * np.log(alpha / 2))  # Compute t
    sample_mean = np.mean(sample)
    lower_bound = sample_mean - t
    upper_bound = sample_mean + t
    return (lower_bound, upper_bound)

# Example usage:
sample = np.random.uniform(0, 1, 100)  # Generating 100 random samples from uniform distribution in [0, 1]
lower, upper = hoeffding_confidence_interval(sample)
print("95% Confidence Interval for the mean:", (lower, upper))
```

This code defines a function `hoeffding_confidence_interval` that takes a sample as input and returns the lower and upper bounds of the 95% confidence interval for the mean using Hoeffding's inequality. In the example usage, we generate a sample of 100 random numbers from a uniform distribution in the interval [0, 1], and then compute the confidence interval. You can replace `sample` with your own data to calculate the confidence interval for your specific dataset.

do the same for chebyschev inequality

Sure, here's how you can implement Chebyshev's inequality to construct a 95% confidence interval for the mean in Python.

Chebyshev's inequality states that for any random variable X with finite mean μ and finite variance σ^2:

$$P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$$

To construct a 95% confidence interval, we want $P(|\bar{X} - \mu| \geq t) \leq 0.05$. Using Chebyshev's inequality:

$$\frac{1}{k^2} \leq 0.05$$
$$k^2 \geq 20$$
$$k \geq \sqrt{20}$$

Then, the confidence interval for the sample mean $\bar{X}$ can be computed as $\bar{X} \pm k \cdot \frac{\sigma}{\sqrt{n}}$.

Here's how you can do it in Python:

```python
import numpy as np

def chebyshev_confidence_interval(sample, alpha=0.05):
    n = len(sample)
    sample_mean = np.mean(sample)
    sample_std = np.std(sample, ddof=1)  # Sample standard deviation
    k = np.sqrt(1 / alpha)  # k value from Chebyshev's inequality
    margin_of_error = k * sample_std / np.sqrt(n)
    lower_bound = sample_mean - margin_of_error
    upper_bound = sample_mean + margin_of_error
    return (lower_bound, upper_bound)

# Example usage:
sample = np.random.uniform(0, 1, 100)  # Generating 100 random samples from uniform
distribution in [0, 1]
lower, upper = chebyshev_confidence_interval(sample)
print("95% Confidence Interval for the mean using Chebyshev's inequality:", (lower, upper))
```

This code defines a function `chebyshev_confidence_interval` that takes a sample as input and returns the lower and upper bounds of the 95% confidence interval for the mean using Chebyshev's inequality. In the example usage, we generate a sample of 100 random numbers from a uniform distribution in the interval [0, 1], and then compute the confidence interval. You can replace `sample` with your own data to calculate the confidence interval for your specific dataset.

do it for bennet inequality

Bennett's inequality is a probabilistic inequality that provides an upper bound on the probability that the sum of independent and bounded random variables deviates from its expected value. It is particularly useful when dealing with random variables with bounded variances.

For independent random variables $X_1, X_2, \ldots, X_n$ with common mean μ and bounded within an interval $[a, b]$, Bennett's inequality states:

$$P\left(\bar{X} - \mu \geq t\right) \leq \exp\left(-\frac{n\sigma^2}{b^2} h\left(\frac{bt}{\sigma^2}\right)\right)$$

where $\bar{X}$ is the sample mean, σ^2 is the sample variance, and $h(u) = (1+u)\log(1+u) - u$.

To construct a 95% confidence interval using Bennett's inequality, we solve for t such that:

$$\exp\left(-\frac{n\sigma^2}{b^2} h\left(\frac{bt}{\sigma^2}\right)\right) = \alpha$$

Here is a Python implementation to calculate the 95% confidence interval using Bennett's inequality:

python

```python
import numpy as np
from scipy.optimize import fsolve
import math

def h(u):
    return (1 + u) * np.log(1 + u) - u

def bennett_confidence_interval(sample, alpha=0.05):
    n = len(sample)
    sample_mean = np.mean(sample)
    sample_var = np.var(sample, ddof=1)
    a = min(sample)
    b = max(sample)
    L = b - a

    # Solve for t in Bennett's inequality
    def equation(t):
        return np.exp(- (n * sample_var / L**2) * h(L * t / sample_var)) - alpha

    t_initial_guess = np.sqrt(sample_var / n)  # Initial guess for t
    t_solution = fsolve(equation, t_initial_guess)[0]

    margin_of_error = t_solution
    lower_bound = sample_mean - margin_of_error
    upper_bound = sample_mean + margin_of_error
    return (lower_bound, upper_bound)

# Example usage:
sample = np.random.uniform(0, 1, 100)  # Generating 100 random samples from uniform
distribution in [0, 1]
lower, upper = bennett_confidence_interval(sample)
print("95% Confidence Interval for the mean using Bennett's inequality:", (lower, upper))
```

This code defines a function `bennett_confidence_interval` that takes a sample as input and returns the lower and upper bounds of the 95% confidence interval for the mean using Bennett's inequality. In the example usage, we generate a sample of 100 random numbers from a uniform distribution in the interval [0, 1], and then compute the confidence interval. You can replace `sample` with your own data to calculate the confidence interval for your specific dataset.

Explain and code rejection-sampling method

Rejection sampling is a simple and widely used technique for generating samples from a target distribution. The method is particularly useful when direct sampling from the target distribution is difficult. Instead, samples are generated from an easier-to-sample proposal distribution, and then some of these samples are "rejected" to ensure that the remaining samples follow the target distribution.

Steps for Rejection Sampling:

1. **Choose a Proposal Distribution**: Select a proposal distribution $q(x)$ that is easy to sample from and has a support that covers the support of the target distribution $p(x)$.

2. **Find a Suitable Constant**: Identify a constant M such that $Mq(x) \geq p(x)$ for all x in the support of $p(x)$. The constant M ensures that the proposal distribution adequately covers the target distribution.

3. **Sampling and Acceptance**:

   - Generate a sample x from the proposal distribution $q(x)$.
   - Generate a uniform random variable u in the interval $[0, 1]$.
   - Accept the sample x if $u \leq \frac{p(x)}{Mq(x)}$. Otherwise, reject x and repeat the process.

Code Implementation:

Here's an example in Python, where we aim to sample from a target distribution $p(x)$ that we will define as an exponential distribution, using a uniform distribution as the proposal distribution.

python

```python
import numpy as np
import matplotlib.pyplot as plt

def target_distribution(x):
    # Example target distribution: Exponential distribution with rate parameter lambda = 1
    return np.exp(-x) if x >= 0 else 0

def proposal_distribution():
    # Uniform distribution between 0 and 10 (since we are targeting an exponential
distribution)
    return np.random.uniform(0, 10)

def rejection_sampling(n_samples, M):
    samples = []
    while len(samples) < n_samples:
        x = proposal_distribution()
        u = np.random.uniform(0, 1)
        if u <= target_distribution(x) / (M * (1/10)):  # 1/10 is the PDF of the uniform
distribution on [0, 10]
            samples.append(x)
    return np.array(samples)

# Parameters
n_samples = 10000
M = 1.5  # A suitable M (should be > 1)

# Generate samples
samples = rejection_sampling(n_samples, M)

# Plotting the results
x = np.linspace(0, 10, 1000)
y_target = np.exp(-x)
plt.hist(samples, bins=50, density=True, alpha=0.6, label='Rejection Sampling')
plt.plot(x, y_target, label='Target Distribution (Exponential)')
plt.legend()
plt.show()
```

Explanation:

1. **target_distribution(x)**: Defines the target distribution, which in this case is an exponential distribution with a rate parameter of 1.

2. **proposal_distribution()**: Defines the proposal distribution, which is a uniform distribution between 0 and 10.

3. **rejection_sampling(n_samples, M)**: Implements the rejection sampling method to generate $n\_samples$ from the target distribution.

4. **Parameters and Plotting**: We specify the number of samples and the constant M, generate samples using the rejection sampling method, and then plot the histogram of the sampled data against the target distribution.

The key to successful rejection sampling is choosing an appropriate proposal distribution and a suitable M such that $Mq(x)$ adequately covers $p(x)$. This ensures that the acceptance probability is not too low, making the sampling process efficient.

Explain and code SVD decomposition

Singular Value Decomposition (SVD) is a powerful and widely used matrix factorization technique in linear algebra. It decomposes a given matrix A into three other matrices:

$$A = U\Sigma V^T$$

where:

- U is an $m \times m$ orthogonal matrix.
- Σ (Sigma) is an $m \times n$ diagonal matrix with non-negative real numbers on the diagonal.
- V is an $n \times n$ orthogonal matrix (Note: V^T is the transpose of V).

The diagonal entries of Σ are known as the singular values of A, and they are arranged in descending order. The columns of U are called the left singular vectors of A, and the columns of V are called the right singular vectors of A.

Applications of SVD:

1. **Dimensionality Reduction**: In Principal Component Analysis (PCA), SVD is used to reduce the number of dimensions in the data.
2. **Signal Processing**: SVD helps in noise reduction and data compression.
3. **Image Compression**: By retaining only the largest singular values, one can compress an image significantly without losing much quality.

Steps to Perform SVD:

1. Compute the singular values of the matrix A.
2. Compute the left singular vectors (columns of U).
3. Compute the right singular vectors (columns of V).

Python Implementation:

Let's perform SVD on a sample matrix using Python's `numpy` library.

```python
import numpy as np

# Define a sample matrix
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

# Perform SVD
U, Sigma, Vt = np.linalg.svd(A)

# Display the results
print("Matrix A:")
print(A)
print("\nMatrix U (left singular vectors):")
print(U)
print("\nVector Sigma (singular values):")
print(Sigma)
print("\nMatrix Vt (right singular vectors, transposed):")
print(Vt)

# Reconstruct the original matrix A from U, Sigma, and Vt
Sigma_full = np.zeros((U.shape[0], Vt.shape[0]))
np.fill_diagonal(Sigma_full, Sigma)
A_reconstructed = np.dot(U, np.dot(Sigma_full, Vt))

print("\nReconstructed Matrix A:")
print(A_reconstructed)
```

Explanation:

1. **Define a Sample Matrix**: We start by defining a sample 3×3 matrix A.
2. **Perform SVD**: Using `np.linalg.svd(A)`, we decompose A into U, Σ, and V^t (where V^t is the transpose of V).
3. **Display the Results**: We print the original matrix A, the left singular vectors U, the singular values Σ, and the right singular vectors V^t.
4. **Reconstruct the Original Matrix**: We create a diagonal matrix Σ_{full} from the singular values and then reconstruct the original matrix A using U, Σ_{full}, and V^t.

Output:

This will output the original matrix, its decomposition into U, Σ, and V^t, and the reconstructed matrix to verify the correctness of the decomposition. The reconstructed matrix should be very close to the original matrix A, with any differences attributable to numerical precision.

SVD is a versatile and essential tool in data analysis, machine learning, and many other fields that involve linear algebra and matrix computations.

Explain and code PCA

Principal Component Analysis (PCA) is a dimensionality reduction technique used to reduce the number of variables in a dataset while retaining most of the variability in the data. PCA achieves this by transforming the original variables into a new set of uncorrelated variables called principal components, which are ordered by the amount of variance they capture from the data.

Steps for PCA:

1. **Standardize the Data**: Scale the data so that each variable has a mean of 0 and a standard deviation of 1.
2. **Compute the Covariance Matrix**: Calculate the covariance matrix to understand how the variables vary with each other.
3. **Compute the Eigenvalues and Eigenvectors**: Find the eigenvalues and eigenvectors of the covariance matrix. The eigenvectors represent the directions of the principal components, and the eigenvalues represent the magnitude of variance in those directions.
4. **Sort Eigenvalues and Eigenvectors**: Sort the eigenvalues and their corresponding eigenvectors in descending order.
5. **Select Principal Components**: Choose the top k eigenvectors that correspond to the top k eigenvalues. These eigenvectors form the principal components.
6. **Transform the Data**: Project the original data onto the new subspace formed by the principal components.

Python Implementation:

Here is a step-by-step implementation of PCA in Python using `numpy` and `scikit-learn`:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Sample data
X = np.array([[2.5, 2.4],
              [0.5, 0.7],
              [2.2, 2.9],
              [1.9, 2.2],
              [3.1, 3.0],
              [2.3, 2.7],
              [2.0, 1.6],
              [1.0, 1.1],
              [1.5, 1.6],
              [1.1, 0.9]])

# Step 1: Standardize the Data
scaler = StandardScaler()
X_std = scaler.fit_transform(X)

# Step 2: Compute the Covariance Matrix
```

```
cov_matrix = np.cov(X_std.T)

# Step 3: Compute the Eigenvalues and Eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

# Step 4: Sort Eigenvalues and Eigenvectors
sorted_indices = np.argsort(eigenvalues)[::-1]
sorted_eigenvalues = eigenvalues[sorted_indices]
sorted_eigenvectors = eigenvectors[:, sorted_indices]

# Step 5: Select Principal Components
# Here we choose the top 2 principal components
n_components = 2
principal_components = sorted_eigenvectors[:, :n_components]

# Step 6: Transform the Data
X_pca = X_std.dot(principal_components)

# Print the results
print("Original Data:")
print(X)
print("\nStandardized Data:")
print(X_std)
print("\nCovariance Matrix:")
print(cov_matrix)
print("\nEigenvalues:")
print(sorted_eigenvalues)
print("\nEigenvectors:")
print(sorted_eigenvectors)
print("\nPrincipal Components:")
print(principal_components)
print("\nTransformed Data:")
print(X_pca)

# Plotting the results
plt.figure(figsize=(8, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], edgecolor='k', s=50)
plt.title('PCA Result')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.grid(True)
plt.show()

# Alternatively, using scikit-learn
pca = PCA(n_components=2)
X_pca_sklearn = pca.fit_transform(X_std)

print("\nTransformed Data using scikit-learn PCA:")
print(X_pca_sklearn)
```

Explanation:

1. **Standardize the Data**: The data is standardized using `StandardScaler` to have a mean of 0 and a standard deviation of 1. This is essential because PCA is affected by the scale of the variables.
2. **Compute the Covariance Matrix**: The covariance matrix is calculated to understand the variance and covariance between the variables.
3. **Compute the Eigenvalues and Eigenvectors**: The eigenvalues and eigenvectors of the covariance matrix are computed. The eigenvectors represent the directions of maximum variance, and the eigenvalues represent the magnitude of the variance in those directions.
4. **Sort Eigenvalues and Eigenvectors**: The eigenvalues and their corresponding eigenvectors are sorted in descending order to identify the principal components that capture the most variance.
5. **Select Principal Components**: The top k eigenvectors are selected based on the number of principal components required.
6. **Transform the Data**: The original data is projected onto the new subspace formed by the principal components, effectively reducing the dimensionality.

Finally, we also demonstrate how to perform PCA using the `PCA` class from `scikit-learn`, which simplifies the process. The transformed data from both methods should be similar.

Do a Wald test with confidence interval of 95%

The Wald test is used to test the significance of individual coefficients in a statistical model, typically in the context of a logistic regression or linear regression model. It compares the estimated coefficient to its standard error to determine whether the coefficient is significantly different from zero. The Wald test statistic follows a standard normal distribution under the null hypothesis.

Steps to Perform a Wald Test:

1. **Fit a Model**: Fit a statistical model (e.g., linear regression or logistic regression) to your data.
2. **Extract Coefficients and Standard Errors**: Obtain the estimated coefficients and their standard errors from the model.
3. **Compute Wald Test Statistic**: For each coefficient, compute the Wald test statistic using the formula:

$$Z = \frac{\hat{\beta} - \beta_0}{SE(\hat{\beta})}$$

   where $\hat{\beta}$ is the estimated coefficient, β_0 is the hypothesized value (usually 0), and $SE(\hat{\beta})$ is the standard error of the coefficient.
4. **Compute p-Value**: Calculate the p-value from the standard normal distribution.
5. **Construct Confidence Intervals**: For each coefficient, construct the confidence interval using the formula:

$$\hat{\beta} \pm Z_{\alpha/2} \times SE(\hat{\beta})$$

   where $Z_{\alpha/2}$ is the critical value from the standard normal distribution (e.g., 1.96 for a 95% confidence interval).

Example with Python:

Let's perform a Wald test on a linear regression model using a sample dataset.

```python
import numpy as np
import statsmodels.api as sm
import pandas as pd

# Create a sample dataset
np.random.seed(0)
n = 100
```

```
X = np.random.randn(n, 2)  # 2 predictor variables
X = sm.add_constant(X)  # Add intercept term
beta = np.array([1, 0.5, -0.5])  # True coefficients
y = X @ beta + np.random.randn(n) * 0.5  # Generate response variable

# Fit a linear regression model
model = sm.OLS(y, X).fit()

# Extract coefficients and standard errors
coefficients = model.params
standard_errors = model.bse

# Wald test statistics and p-values
wald_statistics = coefficients / standard_errors
p_values = 2 * (1 - sm.stats.norm.cdf(np.abs(wald_statistics)))

# 95% confidence intervals
z_critical = 1.96  # 95% confidence interval
confidence_intervals = np.column_stack((coefficients - z_critical * standard_errors,
                                        coefficients + z_critical * standard_errors))

# Print the results
results = pd.DataFrame({
    'Coefficient': coefficients,
    'Standard Error': standard_errors,
    'Wald Statistic': wald_statistics,
    'p-value': p_values,
    '95% CI Lower': confidence_intervals[:, 0],
    '95% CI Upper': confidence_intervals[:, 1]
})

print(results)
```

Explanation:

1. **Create a Sample Dataset**: We generate a sample dataset with 100 observations and 2 predictor variables. The response variable `y` is generated using true coefficients and some random noise.
2. **Fit a Linear Regression Model**: We use `statsmodels` to fit an ordinary least squares (OLS) regression model to the data.
3. **Extract Coefficients and Standard Errors**: The estimated coefficients and their standard errors are extracted from the fitted model.
4. **Wald Test Statistics and p-Values**: For each coefficient, the Wald test statistic and the p-value are computed. The p-value helps determine the significance of each coefficient.
5. **95% Confidence Intervals**: The 95% confidence intervals for each coefficient are calculated using the critical value from the standard normal distribution.

The output will display a table with the estimated coefficients, their standard errors, Wald test statistics, p-values, and 95% confidence intervals. This allows you to assess the significance of each predictor in the model.

1. [1p] Draw a uniform random point X on the surface of the unit sphere in $\mathbb{R}^d$. What is the variance of X_1 (the first coordinate)? Solve this using pen and paper, then fill in the answer below in `variance_x1_problem7`.
2. [1p] How large must ϵ be for 99% of the volume of a d-dimensional unit-radius ball to lie in the shell of ϵ-thickness at the surface of the ball?
3. [3p] The volume of the unit ball is given by

$$
V(d) = \frac{2 \pi^{\frac{d}{2}}}{d \Gamma(\frac{d}{2})} = \frac{ \pi^{\frac{d}{2}}}{ (\frac{d}{2})!}
$$
What function of d would the radius need to be for a ball or radius r to have approximately constant volume as a function of d? Hint use Stirlings formula $n! \approx (n/e)^n$.

1. To find the variance of the first coordinate X_1 of a uniformly random point on the surface of the unit sphere in R^d, we note that X_1 follows a standard normal distribution $\mathrm{N}(0, 1/d)$ due to its uniform distribution on the sphere. Therefore, the variance of X_1 is $\mathrm{Var}(X_1) = 1/d$.

$$\text{variance\_x1\_problem7} = \frac{1}{d}$$

2. For a d-dimensional unit-radius ball, the volume $V(d)$ is given by $V(d) = \frac{\pi^{d/2}}{\Gamma(\frac{d}{2}+1)}$. To find the thickness ϵ needed for 99% of the volume of the ball to lie in the shell at the surface, we need to find the value of ϵ such that the ratio of volumes between the outer shell and the entire ball is 99%.

$$\frac{V_{\text{shell}}}{V_{\text{ball}}} = \frac{V(d,r+\epsilon) - V(d,r)}{V(d,r)} = 0.99$$

Substituting the formula for the volume of the ball and using some algebraic manipulation, we get:

$$\frac{\pi^{d/2}}{\Gamma(\frac{d}{2}+1)} \frac{\frac{r+\epsilon}{r}^d - 1}{\frac{r^d}{r^d}} = 0.99$$

$$\frac{(r+\epsilon)^d - r^d}{r^d} = 0.99$$

$$(1 + \frac{\epsilon}{r})^d - 1 = 0.99$$

$$(1 + \frac{\epsilon}{r})^d = 1.99$$

Taking the natural logarithm of both sides:

$$d \ln(1 + \frac{\epsilon}{r}) = \ln(1.99)$$

$$\ln(1 + \frac{\epsilon}{r}) = \frac{\ln(1.99)}{d}$$

$$1 + \frac{\epsilon}{r} = e^{\frac{\ln(1.99)}{d}}$$

$$\frac{\epsilon}{r} = e^{\frac{\ln(1.99)}{d}} - 1$$

$$\epsilon = r(e^{\frac{\ln(1.99)}{d}} - 1)$$

$$\text{epsilon\_for\_99\_percent} = r(e^{\frac{\ln(1.99)}{d}} - 1)$$

3. To find the function of d for the radius of a ball of radius r to have approximately constant volume as a function of d, we set the derivative of the volume function with respect to d to zero and solve

for r.

$$\frac{d}{dr}\left(\frac{\pi^{d/2}}{(\frac{d}{2})!}\right) = 0$$

Using the product rule and simplifying, we get:

$$\frac{\pi^{d/2}}{(\frac{d}{2})!}\frac{d}{dr}\left(\frac{d}{2}\ln(\pi) - \ln(\frac{d}{2})!\right) = 0$$

$$\frac{d}{dr}\left(\frac{d}{2}\ln(\pi) - \ln(\frac{d}{2})!\right) = 0$$

$$\frac{d}{2}\frac{1}{r} - 0 = 0$$

$$\frac{d}{2}\frac{1}{r} = 0$$

$$\frac{d}{2r} = 0$$

$$r = \frac{d}{2}$$

$$\text{radius\_for\_constant\_volume} = \frac{d}{2}$$

These are the answers to the given problems. If you need further assistance or clarification, feel free to ask!

In this problem you will do rejection sampling from complicated distributions, you will also be using your samples to compute certain integrals, a method known as Monte Carlo integration: (Keep in mind that choosing a good sampling distribution is often key to avoid too much rejection)

1. [4p] Fill in the remaining part of the function `problem1_inversion` in order to produce samples from the below distribution using rejection sampling:

$$
F[x] =
\begin{cases}
0, & x \leq 0 \\
\frac{e^{x^2}-1}{e-1}, & 0 < x < 1 \\
1, & x \geq 1
\end{cases}
$$

2. [2p] Produce 100000 samples (**use fewer if it times-out and you cannot find a solution**) and put the answer in `problem1_samples` from the above distribution and plot the histogram together with the true density. *(There is a timeout decorator on this function and if it takes more than 10 seconds to generate 100000 samples it will timeout and it will count as if you failed to generate.)*
3. [2p] Use the above 100000 samples (`problem1_samples`) to approximately compute the integral

$$
\int_0^{1} \sin(x) \frac{2e^{x^2} x}{e-1} dx
$$

$$
and store the result in `problem1_integral`.

4. [2p] Use Hoeffdings inequality to produce a 95\% confidence interval of the integral above and store the result as a tuple in the variable `problem1_interval`

5. [4p] Fill in the remaining part of the function `problem1_inversion_2` in order to produce samples from the below distribution using rejection sampling:
$$
  F[x] =
  \begin{cases}
    0, & x \leq 0 \\
    20xe^{20-1/x}, & 0 < x < \frac{1}{20} \\
    1, & x \geq \frac{1}{20}
  \end{cases}
$$
Hint: this is tricky because if you choose the wrong sampling distribution you reject at least 9 times out of 10. You will get points based on how long your code takes to create a certain number of samples, if you choose the correct sampling distribution you can easily create 100000 samples within 2 seconds.

python

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import uniform
from scipy.stats import norm

def problem1_inversion(n_samples):
    samples = []
    for _ in range(n_samples):
        u = uniform.rvs()
        if u == 0:
            samples.append(0)
        elif u == 1:
            samples.append(1)
        else:
            x = np.sqrt(np.log(u * (np.exp(1) - 1) + 1))
            samples.append(x)
    return np.array(samples)

def problem1_inversion_2(n_samples):
    samples = []
    for _ in range(n_samples):
        u = uniform.rvs()
        if u == 0:
            samples.append(0)
        elif u == 1:
            samples.append(1)
        else:
            # Fill in the correct formula here
            x = 1 / (20 - np.log(u * (20 * np.exp(20) - 1) + 1))
            samples.append(x)
    return np.array(samples)

# Problem 2
problem1_samples = problem1_inversion(100000)
plt.hist(problem1_samples, bins=50, density=True, alpha=0.6, label='Samples')
x = np.linspace(0, 1, 1000)
```

```
true_density = (np.exp(x**2) - 1) / (np.exp(1) - 1)
plt.plot(x, true_density, label='True Density')
plt.title('Histogram of Samples and True Density')
plt.legend()
plt.show()

# Problem 3
problem1_integral = np.mean(np.sin(problem1_samples) * 2 * np.exp(problem1_samples**2) *
problem1_samples / (np.exp(1) - 1))

# Problem 4
hoeffding_constant = np.sqrt(-np.log(0.05 / 2) / (2 * len(problem1_samples)))
problem1_interval = (problem1_integral - hoeffding_constant, problem1_integral +
hoeffding_constant)

# Problem 5
# You need to fill in the remaining part of the function problem1_inversion_2
```

Let us build a proportional model ($\mathbb{P}(Y=1 \mid X) = G(\beta_0+\beta \cdot X)$ where G is the logistic function) for the spam vs not spam data. Here we assume that the features are presence vs not presence of a word, let X_1,X_2,X_3 denote the presence (1) or absence (0) of the words $("free", "prize", "win")$.

1. [2p] Load the file `data/spam.csv` and create two numpy arrays, `problem2_X` which has shape (n_emails,3) where each feature in `problem2_X` corresponds to X_1,X_2,X_3 from above, `problem2_Y` which has shape **(n_emails,)** and consists of a 1 if the email is spam and 0 if it is not. Split this data into a train-calibration-test sets where we have the split 40%, 20%, 40%, put this data in the designated variables in the code cell.

2. [4p] Follow the calculation from the lecture notes where we derive the logistic regression and implement the final loss function inside the class `ProportionalSpam`. You can use the `Test` cell to check that it gives the correct value for a test-point.

3. [4p] Train the model `problem2_ps` on the training data. The goal is to calibrate the probabilities output from the model. Start by creating a new variable `problem2_X_pred` (shape `(n_samples,1)`) which consists of the predictions of `problem2_ps` on the calibration dataset. Then train a calibration model using `sklearn.tree.DecisionTreeRegressor`, store this trained model in `problem2_calibrator`.

4. [3p] Use the trained model `problem2_ps` and the calibrator `problem2_calibrator` to make final predictions on the testing data, store the prediction in `problem2_final_predictions`. Compute the $0-1$ test-loss and store it in `problem2_01_loss` and provide a 99% confidence interval of it, store this in the variable `problem2_interval`, this should again be a tuple as in **problem1**.

python

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from scipy.stats import norm

# Problem 1
data = pd.read_csv('data/spam.csv')
X = data[['free', 'prize', 'win']].values
Y = data['spam'].values
problem2_X_train, problem2_X_test_cal, problem2_Y_train, problem2_Y_test_cal =
train_test_split(X, Y, test_size=0.6, random_state=42)
problem2_X_cal, problem2_X_test, problem2_Y_cal, problem2_Y_test =
train_test_split(problem2_X_test_cal, problem2_Y_test_cal, test_size=0.67, random_state=42)

# Problem 2
class ProportionalSpam:
    def __init__(self):
        self.beta_0 = 0
        self.beta = np.zeros(3)

    def fit(self, X, Y, n_epochs=100, lr=0.01):
        n_samples = X.shape[0]
        self.beta_0 = 0
        self.beta = np.zeros(3)
        for _ in range(n_epochs):
            Y_pred = self.predict_proba(X)
            self.beta_0 -= lr * np.sum(Y_pred - Y) / n_samples
            for i in range(3):
                self.beta[i] -= lr * np.sum((Y_pred - Y) * X[:, i]) / n_samples

    def predict_proba(self, X):
        return 1 / (1 + np.exp(-(self.beta_0 + np.dot(X, self.beta))))

    def predict(self, X):
        return (self.predict_proba(X) >= 0.5).astype(int)

# Test
ps = ProportionalSpam()
ps.fit(problem2_X_train, problem2_Y_train)
print(ps.predict(problem2_X_test[0:1]))  # Should print [1] or [0]

# Problem 3
problem2_ps = ProportionalSpam()
problem2_ps.fit(problem2_X_train, problem2_Y_train)
problem2_X_pred = problem2_ps.predict_proba(problem2_X_cal)
problem2_calibrator = DecisionTreeRegressor()
problem2_calibrator.fit(problem2_X_pred.reshape(-1, 1), problem2_Y_cal)

# Problem 4
problem2_final_predictions =
problem2_calibrator.predict(problem2_ps.predict_proba(problem2_X_test).reshape(-1, 1))
problem2_01_loss = np.mean(problem2_final_predictions != problem2_Y_test)
hoeffding_constant = np.sqrt(-np.log(0.01 / 2) / (2 * len(problem2_Y_test)))
problem2_interval = (problem2_01_loss - hoeffding_constant, problem2_01_loss +
hoeffding_constant)
```