Aalto University
School of Electrical
Engineering

ELEC-E7130 Internet Traffic Measurements and Analysis

Assignment 7. Sampling

Name: Xingji Chen

Student ID: 101659554

E-mail: xingji.chen@aalto.fi

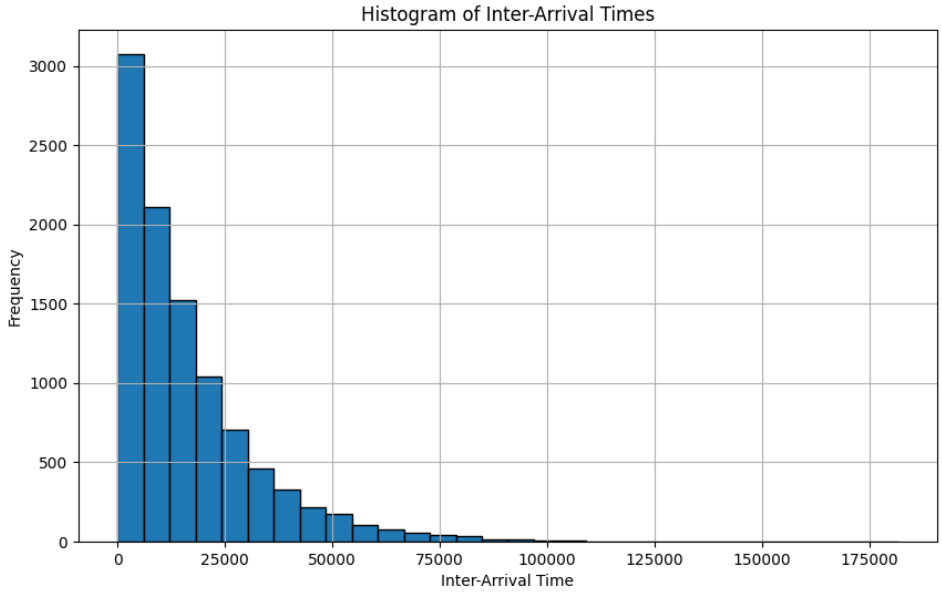
Aalto University
School of Electrical
Engineering

Task 1: Sampling and distributions (off-line sampling)

1. Plot the histogram of the original data and compute the mean.

```
file_path = r'file\sampling.txt'
with open(file_path, 'r') as file:
    inter_arrival_times = np.array([float(line.strip()) for line in
file])

mean_inter_arrival = np.mean(inter_arrival_times)
print(f"The mean inter-arrival time is: {mean_inter_arrival}")
```

Open the file in read mode, use a list comprehension to read the file line by line, strip the whitespace from the beginning and end of each line, convert it to a float, and store all these floating-point numbers in an array. Then calculate the mean of these inter-arrival times and print it out. Next, plot a histogram of the inter-arrival times.



The mean inter-arrival time is: 16432.250672819187.

2. Select 5000 random samples from original data. Plot its histogram and compute the mean.

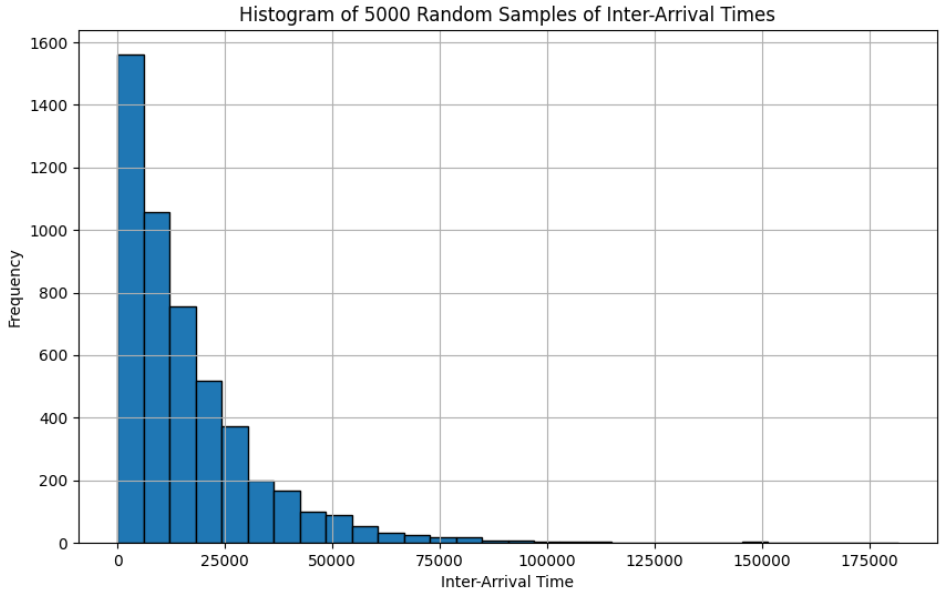
```
import numpy as np
import matplotlib.pyplot as plt

file_path = r'file\sampling.txt'
with open(file_path, 'r') as file:
    inter_arrival_times = np.array([float(line.strip()) for line in
file])

original_mean_inter_arrival = np.mean(inter_arrival_times)
print(f"The mean inter-arrival time of the original data is:
{original_mean_inter_arrival}")

random_samples = np.random.choice(inter_arrival_times, size=5000,
replace=True)

sample_mean_inter_arrival = np.mean(random_samples)
print(f"The mean inter-arrival time of the 5000 samples is:
{sample_mean_inter_arrival}")

plt.figure(figsize=(10, 6))
plt.hist(inter_arrival_times, bins=30, edgecolor='black')
plt.title('Histogram of Inter-Arrival Times')
plt.xlabel('Inter-Arrival Time')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

Firstly, the file is opened in read mode, and each line is read using a list comprehension, stripping any leading or trailing whitespace, converting it to a floating-point number, and then storing these numbers in an array.

Then, the mean inter-arrival time from the original data is calculated and printed. 5000 samples are randomly selected from the original data, and the mean value of these randomly selected 5000 samples is calculated and printed as the new mean value.

Finally, a histogram is plotted with a specified figure size. The chart title is "Histogram of 5000 Random Samples of Inter-Arrival Times," the x-axis is labeled "Inter-Arrival Time," and the y-axis is labeled "Frequency".

Aalto University
School of Electrical
Engineering



The mean inter-arrival time of the 5000 samples is: 16320.122172017602.

In the histogram of the original data, the majority of data points are concentrated on the left side (indicating shorter inter-arrival times), and as the inter-arrival time increases, the frequency gradually decreases. The skewness of the data suggests that shorter inter-arrival times are more common, while longer inter-arrival times occur less frequently.

In the histogram of the 5,000 random samples, the sample distribution is similar to the original distribution, but frequencies are lower due to the smaller dataset size. This indicates that the sample is relatively representative of the larger dataset.

According to the Law of Large Numbers, as sample size increases, the sample mean will tend to be closer to the population mean. The similarity in shape between the sample and the original data suggests that the sample mean is likely a good estimate of the population mean. A larger sample size typically reduces sampling error, that is, the difference between the sample statistic (such as the sample mean) and the population parameter (such as the population mean). A sample size of 5,000 is sufficient to provide a good estimation of the distribution of the original data. If the sample size were smaller, there would be more variability, potentially larger sampling errors, and consequently, reduced accuracy of the estimates.

3. Select 10000 times n random elements from the data to compute the mean of these n values.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats

file_path = r'file\sampling.txt'
with open(file_path, 'r') as file:
    inter_arrival_times = np.array([float(line.strip()) for line in
file])

def analyze_sample_means(n, data):
    sample_means = [np.mean(np.random.choice(data, n, replace=True))
for _ in range(10000)]

    plt.figure(figsize=(10, 6))
    plt.hist(sample_means, bins=50, edgecolor='black')
    plt.title(f'Histogram of Sample Means (n={n})')
    plt.xlabel('Sample Mean')
    plt.ylabel('Frequency')
    plt.grid(True)
    plt.show()

    plt.figure(figsize=(10, 6))
    (osm, osr), (slope, intercept, r) = stats.probplot(sample_means,
dist="norm", plot=plt)
    plt.title(f'Q-Q Plot of Sample Means (n={n})')
    plt.legend(['Data Quantiles', 'Normal Dist. Fit
($R^2={:.4f}$)'.format(r**2)])
    plt.show()

    mean_of_sample_means = np.mean(sample_means)
    std_dev_of_sample_means = np.std(sample_means)
    print(f"For n={n}:")
    print(f"Mean of sample means: {mean_of_sample_means}")
    print(f"Standard deviation of sample means:
{std_dev_of_sample_means}")

    sampling_error = mean_of_sample_means - np.mean(data)
    print(f"Sampling error: {sampling_error}\n")
```

Aalto University
School of Electrical
Engineering

```
for n in [10, 100, 1000]:
    analyze_sample_means(n, inter_arrival_times)
```

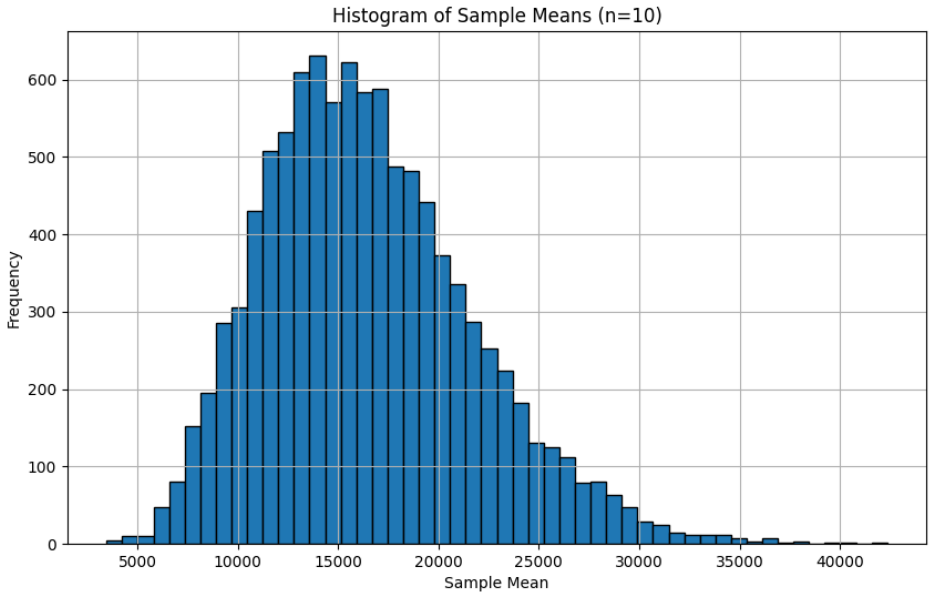
This Python script performs a statistical analysis on a set of inter-arrival time data read from a file. It starts by reading the inter-arrival times from the file, removing any whitespace, converting them into floating-point numbers, and storing them in a NumPy array.

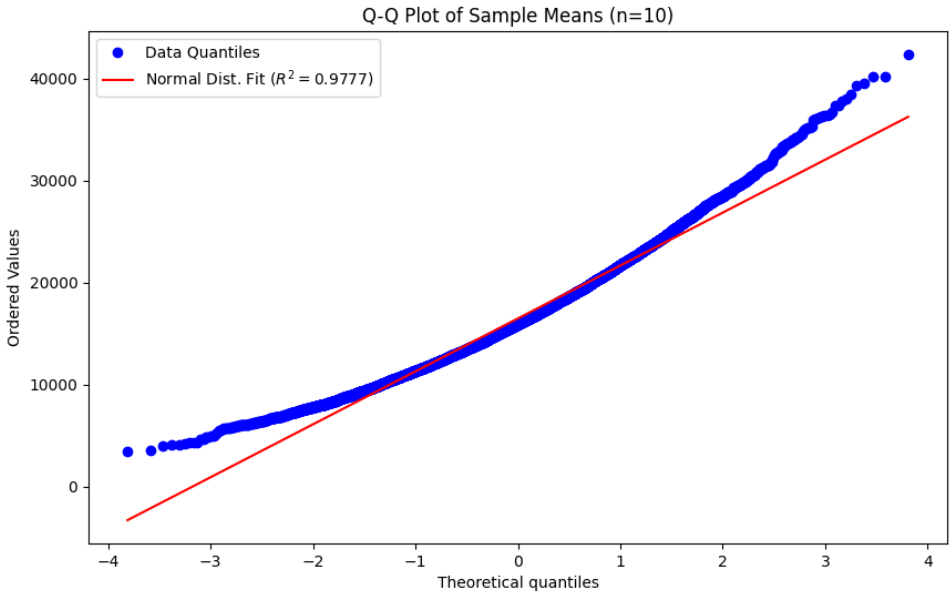
A function is then created to analyze sample means given a sample size n. It generates 10,000 sample means, each of which is obtained by randomly selecting n times from the data. A histogram is drawn for each sample mean.

Next, a quantile-quantile (Q-Q) plot is generated to compare the distribution of sample means with a normal distribution. A best fit line is drawn, and the coefficient of determination is calculated to assess the degree to which the sample means follow a normal distribution. The mean and standard deviation of the 10,000 sample means are then computed and printed to understand the expected value and variability of the sample means.

Finally, the function is called with different sizes of n=10, n=100, and n=1000. This demonstrates the impact of sample size on the distribution of sample means and sampling error.

- n = 10





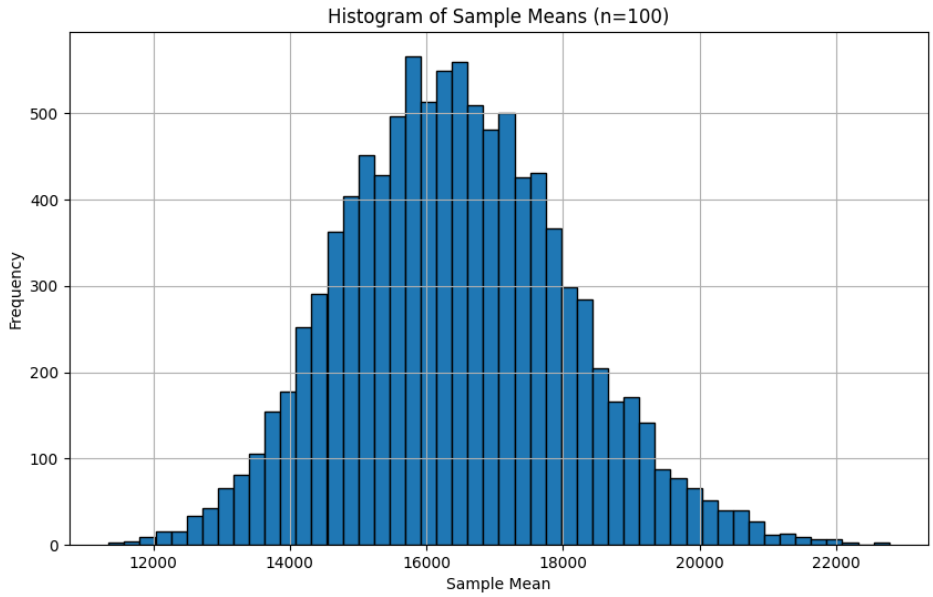
For n=10:

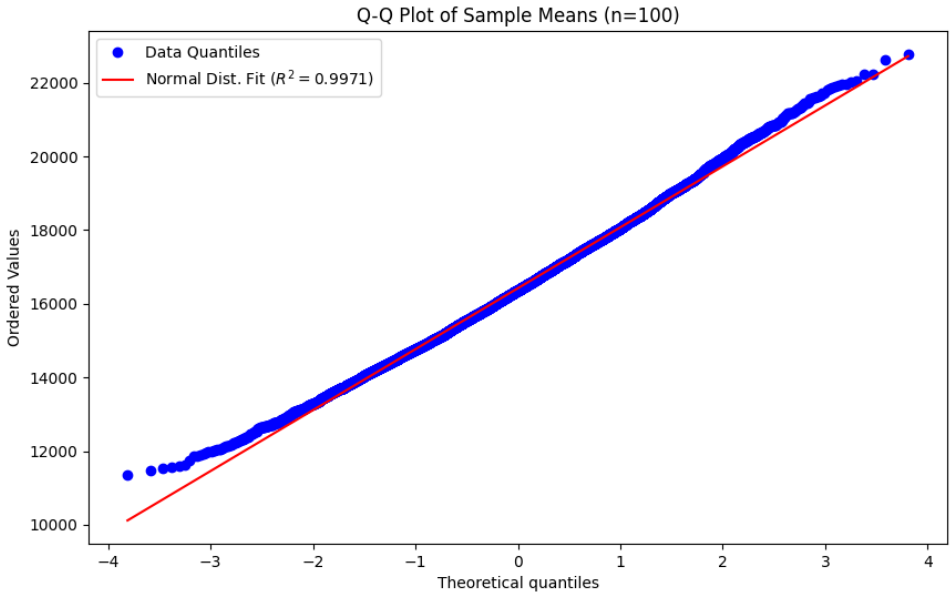
Mean of sample means: 16487.195322728265

Standard deviation of sample means: 5242.126912961678

Sampling error: 54.94464990907727

- n = 100





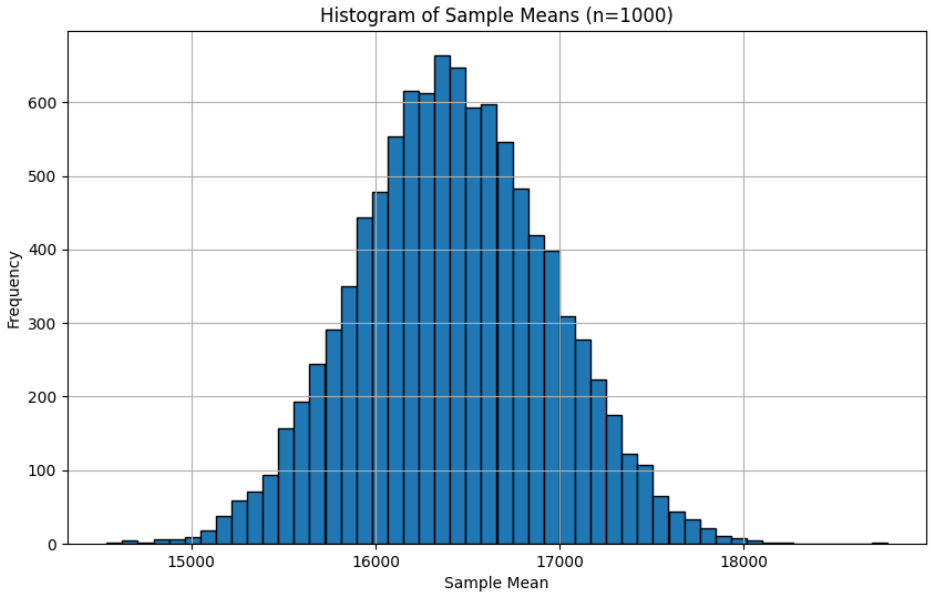
For n=100:

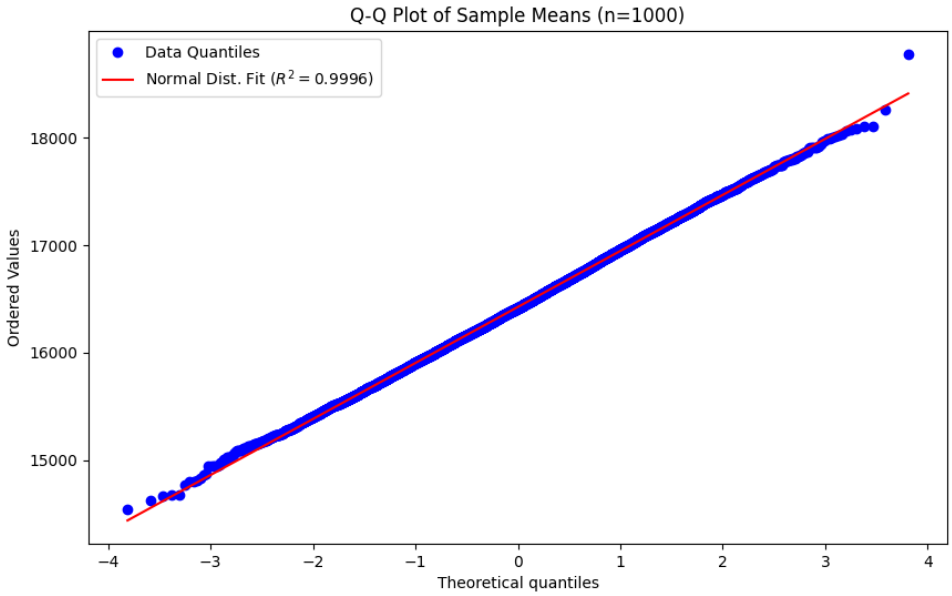
Mean of sample means: 16420.533266579536

Standard deviation of sample means: 1655.9228922710108

Sampling error: -11.71740623965161

- n = 1000





For n=1000:

Mean of sample means: 16426.684004789146

Standard deviation of sample means: 521.513739527369

Sampling error: -5.56666803004191

4. Discussions

These images effectively represent the concept of the Central Limit Theorem (CLT). The Central Limit Theorem states that the sampling distribution of the sample means will become approximately normally distributed as the sample size increases, regardless of the shape of the population distribution, assuming that all samples are of equal size and independent.

At n=10, the histogram displays the distribution of sample means for a sample size of 10. The shape of the histogram is somewhat bell-shaped, indicating that even with a small sample size, the distribution of the sample means is tending towards normality. The Q-Q plot is used to determine if a dataset comes from a certain theoretical distribution (a normal distribution in this case), with the red line representing how well the normal distribution fits the data points (blue dots). An R-squared value of 0.9777 suggests a good fit, although there are noticeable deviations at the tails.

At n=100, the histogram shows a more distinct bell curve, indicating that as the sample size increases, the sampling distribution becomes closer to normal. This is consistent with the CLT. The Q-Q plot for n=100 shows a better fit of the data points to the line of normal distribution, with an R-squared value of 0.9971. Compared to the Q-Q plot for n=10, the points lie closer to the red line, especially at the tails.

At n=1000, the histogram exhibits an even more pronounced bell curve, indicating a very strong tendency towards normality in the sample means as the sample size increases, which aligns with the Central Limit Theorem. At this point, a Q-Q plot would show an even tighter alignment of the data points to the normal distribution line, suggesting a very high R-squared value of 0.9996.

The progression from n=10 to n=100 in the histograms and the corresponding Q-Q plots, and extrapolating to n=1000, vividly demonstrates the power of the CLT. As the sample size increases, the distribution of the sample means becomes closer to a normal distribution, regardless of the underlying population distribution from which the samples are drawn.

Aalto University
School of Electrical
Engineering

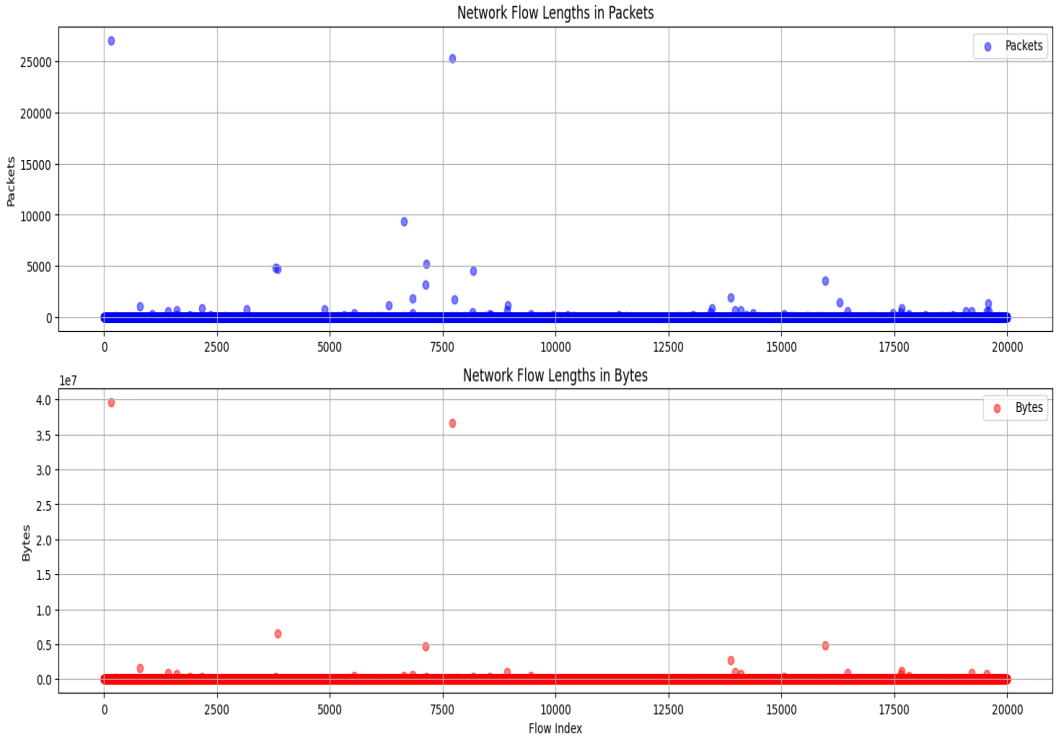
Task 2: High variability (on-line sampling)

1. Original data

```
import pandas as pd
import matplotlib.pyplot as plt

file_path = r'file\flows.txt'
df = pd.read_csv(file_path, sep='\t', header=None, names=['packets',
'bytes'])

mean_packets = df['packets'].mean()
median_packets = df['packets'].median()
mean_bytes = df['bytes'].mean()
median_bytes = df['bytes'].median()

print(f"Mean of packets: {mean_packets}")
print(f"Median of packets: {median_packets}")
print(f"Mean of bytes: {mean_bytes}")
print(f"Median of bytes: {median_bytes}")

plt.figure(figsize=(10, 8))
plt.subplot(2, 1, 1)
plt.scatter(df.index, df['packets'], alpha=0.5, label='Packets',
color='blue')
plt.title('Network Flow Lengths in Packets')
plt.ylabel('Packets')
plt.legend()
plt.grid(True)

plt.subplot(2, 1, 2)  # (rows, columns, panel number)
plt.scatter(df.index, df['bytes'], alpha=0.5, label='Bytes',
color='red')
plt.title('Network Flow Lengths in Bytes')
plt.xlabel('Flow Index')
plt.ylabel('Bytes')
plt.legend()
plt.grid(True)
```

```
plt.tight_layout()

plt.show()
```

First, import the required libraries and read the data. Use the `read_csv` function from pandas to read the file, specifying the delimiter as a tab and naming the two columns of data as 'packets' and 'bytes'. Then calculate the mean and median, and print out the statistical results. Next, scatter plots are drawn for both 'packets' and 'bytes' columns. For each subplot, the scatter plot shows the number of packets and bytes for each flow, with the flow index as the horizontal coordinate of the scatter plot.



Mean of packets: 7.7614

Median of packets: 1.0

Mean of bytes: 6015.6893

Median of bytes: 40.0

Aalto University
School of Electrical
Engineering

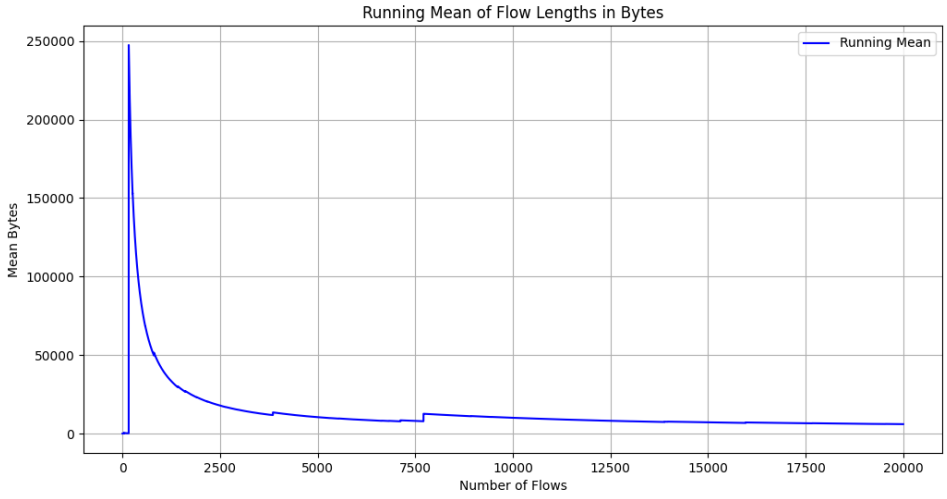
2. On-line measurement

- running_mean

```
# Function to calculate running mean
def running_mean(sequence):
    means = []
    current_sum = 0
    for i, value in enumerate(sequence):
        current_sum += value
        means.append(current_sum / (i + 1))
    return means
```

First, initialize an empty list named `means` to store the running averages. Initialize a variable `current_sum` to zero; this will track the sum of the elements as we iterate through the sequence. Use a `for` loop with `enumerate` to iterate over the `sequence` of numbers, where `enumerate` provides the index (`i`) and the value (`value`) at each iteration. In each iteration, add the current `value` to `current_sum`.

Then, calculate the running average by dividing `current_sum` by the current index plus 1 (since indices in Python start from 0, but the count of numbers starts from 1). Append the calculated running average to the `means` list.
After the loop ends, return the list of running averages.



This chart displays the output of the `running_mean` function, which plots the cumulative average of flow lengths in bytes as more flows are taken into consideration. The chart starts with a pronounced peak. This indicates that the byte size of the first few flows is very high, resulting in a high initial average. After this initial peak, the running mean drops sharply. This suggests that the byte sizes of subsequent flows are much smaller than those at the start, causing the average to decrease rapidly as more samples are included. As we move along the x-axis, which represents the number of flows, the running mean begins to stabilize. This suggests that as more flows are considered, their byte sizes become more consistent, and the mean value reaches a stable level.

The overall average of the original data is 16,432 bytes. Looking at the chart, the running mean seems to approach a value possibly within this range, which is expected if the data contains a large number of flows and the initial extreme values are outliers. Because the running mean starts very high and then stabilizes, it is clear that some very large flow sizes in the early data points disproportionately affected the mean value. As the number of flows increases, their impact on the overall mean diminishes. If we consider the sensitivity of the mean to the order of the data, the running mean tends to approach the mean of the original data as the number of flows increases, assuming that the data is not time-sequenced or otherwise meaningfully ordered.

- running_median

```
# Function to calculate running median using min and max heaps
def running_median(sequence):
    min_heap, max_heap = [], []
    medians = []

    for number in sequence:
        # Ensure max_heap always contains the smaller half of numbers
        if not max_heap or number < -max_heap[0]:
            heapq.heappush(max_heap, -number)
        else:
            heapq.heappush(min_heap, number)

        # Rebalance heaps if necessary
        if len(max_heap) > len(min_heap) + 1:
            heapq.heappush(min_heap, -heapq.heappop(max_heap))
```

```
    if len(min_heap) > len(max_heap):
        heapq.heappush(max_heap, -heapq.heappop(min_heap))

    # Compute the median
    if len(max_heap) == len(min_heap):
        medians.append(float(-max_heap[0] + min_heap[0]) / 2)
    else:
        medians.append(float(-max_heap[0]))

return medians
```

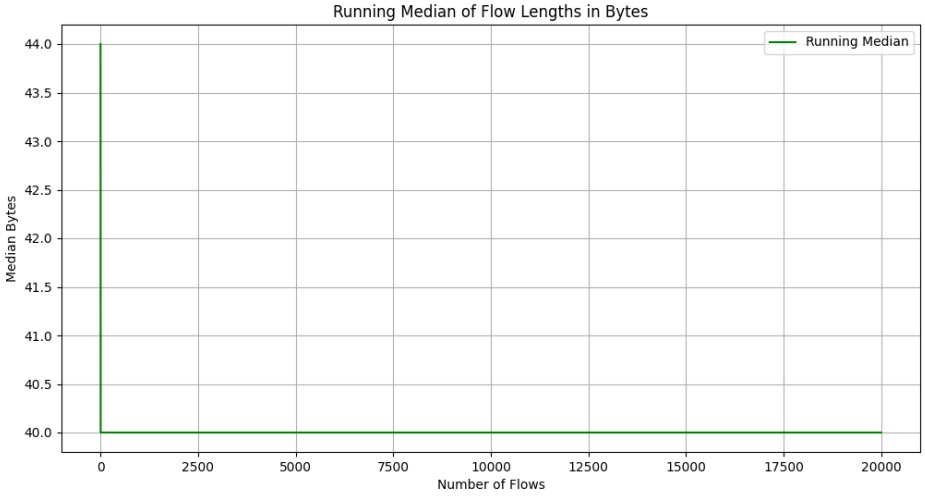
This segment of code defines a function called `running_median`, which calculates the median of numbers as it iterates through each number in a sequence. It employs two heaps (a max heap and a min heap) to maintain the smallest and largest halves of the input sequence, thereby allowing for efficient computation of the median.

The function begins by initializing two heaps: `min_heap` to store the larger half of the numbers, and `max_heap` for the smaller half. The `medians` list will store the calculated median values. For each value provided, the function decides which heap to add it to. If `max_heap` is empty or the `number` is less than the root of `max_heap` (the largest number in `max_heap`), then that number is added to `max_heap`. Since Python's heap is a min heap by default, the number is negated when added to simulate a max heap. If the number is greater than the root of `max_heap`, it is added to `min_heap`.

Next, the lengths of the two heaps are compared to ensure they remain roughly balanced (the max heap's length can only be at most one more than the min heap's length). If one heap becomes too large, the root of that heap is popped and pushed into the other heap to maintain the balance.

The median is then calculated; if the two heaps are of equal length, the median is the average of the roots of both heaps. If the lengths of the heaps are unequal (which can only occur when `max_heap` has one more element than `min_heap` due to the rebalancing condition), then the root of `max_heap` represents the median. After processing all the numbers in the sequence, the function returns a list of medians calculated at each iteration step.

Aalto University
School of Electrical
Engineering



The chart displays the output of the `running_median` function, which calculates the median of the first `n` flow length samples, where `n` increases with each processed flow.

The running median starts at just over 43 bytes. This indicates that the median for the earliest flows in the dataset was above 43 bytes. After the initial values, the running median quickly falls and stabilizes around 40 bytes. This rapid decline suggests that the first set of flow lengths included some larger values, which initially skewed the median higher, but as more flows were included, the median shifted towards a more typical flow length.

Once stabilized, the running median shows very little change as the number of flows increases. This indicates that flow lengths are consistent across the majority of the sampled flows. There are no significant peaks or troughs in the chart, suggesting that there are no significant outliers affecting the median after the initial set of flows.

The median calculated from the original data is 40 bytes. Given that the running median stabilizes at this value, it suggests that the running median has converged to the overall median. This type of convergence would be expected if the function processed enough flows to mitigate any initial anomalies. This may imply that the majority of packet sizes in the network traffic are similar, indicating that the communication patterns in the network are regular and consistent.

Aalto University
School of Electrical
Engineering

Task 3: Data pre-processing for ML purposes

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, StandardScaler,
MinMaxScaler

def prepare_dataset(file_path):
    # Step 1: Load the data and remove instances with missing values
    df = pd.read_csv(file_path)
    df.dropna(inplace=True)

    # Step 2: Perform stratified random sampling
    less_than_2000 = df[df['duration'] < 2000].sample(n=100,
random_state=1)
    more_than_2000 = df[df['duration'] >= 2000].sample(n=100,
random_state=1)
    df = pd.concat([less_than_2000, more_than_2000])

    # Step 3: Encode non-numeric data
    label_encoder = LabelEncoder()
    df['srcip'] = label_encoder.fit_transform(df['srcip'])
    df['dstip'] = label_encoder.fit_transform(df['dstip'])

    # Step 4: Standardize values
    standard_scaler = StandardScaler()
    df[['srcport', 'dstport', 'proto', 'duration']] =
standard_scaler.fit_transform(
        df[['srcport', 'dstport', 'proto', 'duration']])

    # Step 5: Normalize values between 0 and 1
    minmax_scaler = MinMaxScaler()
    df[['srcip', 'srcport', 'dstip', 'dstport', 'proto', 'duration']]
= minmax_scaler.fit_transform(
        df[['srcip', 'srcport', 'dstip', 'dstport', 'proto',
'duration']])

    # Return the preprocessed dataset
    return df.reset_index(drop=True)
```

Aalto University
School of Electrical
Engineering

```
# Assuming the CSV file is in the 'file' directory as specified:
file_path = 'file/simple_flow_data.csv'
preprocessed_dataset = prepare_dataset(file_path)
print(preprocessed_dataset)
```

First, the dataset is loaded from a CSV file, and instances containing missing values are removed. Then, two subsets are created based on the condition of the 'duration' feature, with each subset containing 100 instances, and then they are merged. The categorical data 'srcip' and 'dstip' are encoded into numeric labels. After that, the values in each column are normalized, transformed to a range between 0 and 1. Finally, the processed data is printed and stored.

	srcip	srcport	dstip	dstport	proto	duration
0	0.551724	0.684129	0.158730	0.153839	0.3125	0.000003
1	0.551724	0.932757	0.158730	0.153839	0.3125	0.000002
2	0.551724	0.738982	0.158730	0.153839	0.3125	0.000002
3	0.551724	0.931475	0.158730	0.153839	0.3125	0.000002
4	0.551724	0.930489	0.174603	0.000892	0.3125	0.000002
..	...	...	...	...	...	...
195	0.551724	0.653663	0.619048	0.006001	0.3125	0.288773
196	0.551724	0.864462	0.857143	0.006001	0.3125	0.055536
197	0.068966	0.002251	0.142857	0.001293	1.0000	0.017837
198	0.551724	0.824761	0.111111	0.006001	0.3125	0.197099
199	0.896552	0.002251	0.142857	0.001293	1.0000	0.002591

1. Mention three types of probability sampling applied in ML apart from the one already mentioned.

   1) Simple Random Sampling: This is the most fundamental form of probability sampling. Each member of the population has an equal chance of being selected. This can be carried out either with replacement or without replacement.
   2) Systematic Sampling: In an ordered dataset, elements are selected at regular intervals determined by a starting point and a sampling interval. For example, one might select every 10th row in the data after randomly choosing a starting point between 1 and 10.
   3) Cluster Sampling: The population is divided into groups, or clusters, and a random sample of these clusters is then chosen. All individuals within the selected clusters are included in the sample.

Aalto University
School of Electrical
Engineering

2. What is the purpose of encoding the values in ML?

- In machine learning, encoding values typically refer to the process of converting categorical data into numerical format. The primary purpose of encoding values is to make the data interpretable and usable by machine learning algorithms, which generally require numerical input. Most machine learning algorithms can only handle numerical data and are unable to process categorical data in its raw form. Encoding transforms the data into a format that the algorithms can work with. Values that have been encoded are often more efficiently processed by machine learning models. When categorical variables contain a large number of categories, some encoding methods can condense these categories into fewer dimensions. By properly encoding categorical variables, machine learning models can make better use of the information in the data, which often results in improved performance in terms of accuracy, precision, and recall.

3. What are the differences between standardization and normalization in terms of feature scaling in ML?

1) Standardization
   - Objective: Standardization rescales data to have a mean (μ) of 0 and a standard deviation (σ) of 1. The result of standardization is that feature values will be centered around 0 with a standard deviation of 1.
   - When to Use: It is used when we want features to have the properties of a standard normal distribution. It is particularly useful for algorithms that assume the input variables follow a normal distribution and are more compatible with standardized data, such as Support Vector Machines, Linear Regression, and Logistic Regression.
   - Impact on Outliers: Standardization does not have a bounded range, meaning that outliers may significantly affect the rescaling. It preserves the outliers.

2) Normalization
   - Objective: Normalization, also known as Min-Max scaling, rescales the features to a fixed range, typically 0 to 1, or in some cases, -1 to 1.
   - When to Use: It is often used when the algorithm assumes data within a bounded interval, as is the case with Neural Networks, or it can also be beneficial for algorithms that use distance calculations, such as k-Nearest Neighbors (k-NN).
   - Impact on Outliers: Normalization is sensitive to outliers, meaning that if there are

outliers in the data, they will affect the rescale such that the majority of the data will be transformed into a smaller range of values.