06 - Confidence Interval of Counted Data (Poisson Distribution)

April 24, 2024

1 Confidence Interval of Counted Data (Poisson Distribution)

1.1 Introduction

In this chapter, we will delve into the world of **counted data** and its statistical analysis using the **Poisson Distribution**. Counted data, often encountered in fields such as epidemiology, traffic engineering, and social sciences, refers to data that are counted as *discrete events in a fixed time or space*.

The Poisson Distribution is a powerful statistical tool used to model such data. Named after the French mathematician Siméon Denis Poisson, it provides a way to predict the probability of certain events from happening when you know how often the event has occurred. It is particularly useful when these events happen with a known **average rate** and *independently of the time since the last event*.

We will focus on the confidence interval of counted data. Confidence intervals are a range of values that are likely to contain the true population parameter. In the context of Poisson Distribution, it provides a range where the true average rate of occurrence (lambda, λ) lies with a certain level of confidence.

Following the theoretical foundation, we will transition into practical Python applications, in particular using the `scipy.stats` library. By the end of this chapter, we will have a solid understanding of how to apply Poisson Distribution to analyze counted data and calculate confidence intervals using Python.

1.2 The Poisson distribution

Let's start by generating two sets of random variables from the Poisson distribution using `numpy`. We will compare two distributions of some background radiation counts, one with a mean of 1.6 counts/min, and the second with 7.5 counts/min.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(421)

# Set the parameters for the Poisson distribution
lambda1 = 1.6 # e.g., mean radiation counts/min
lambda2 = 7.5
size = 10000
```

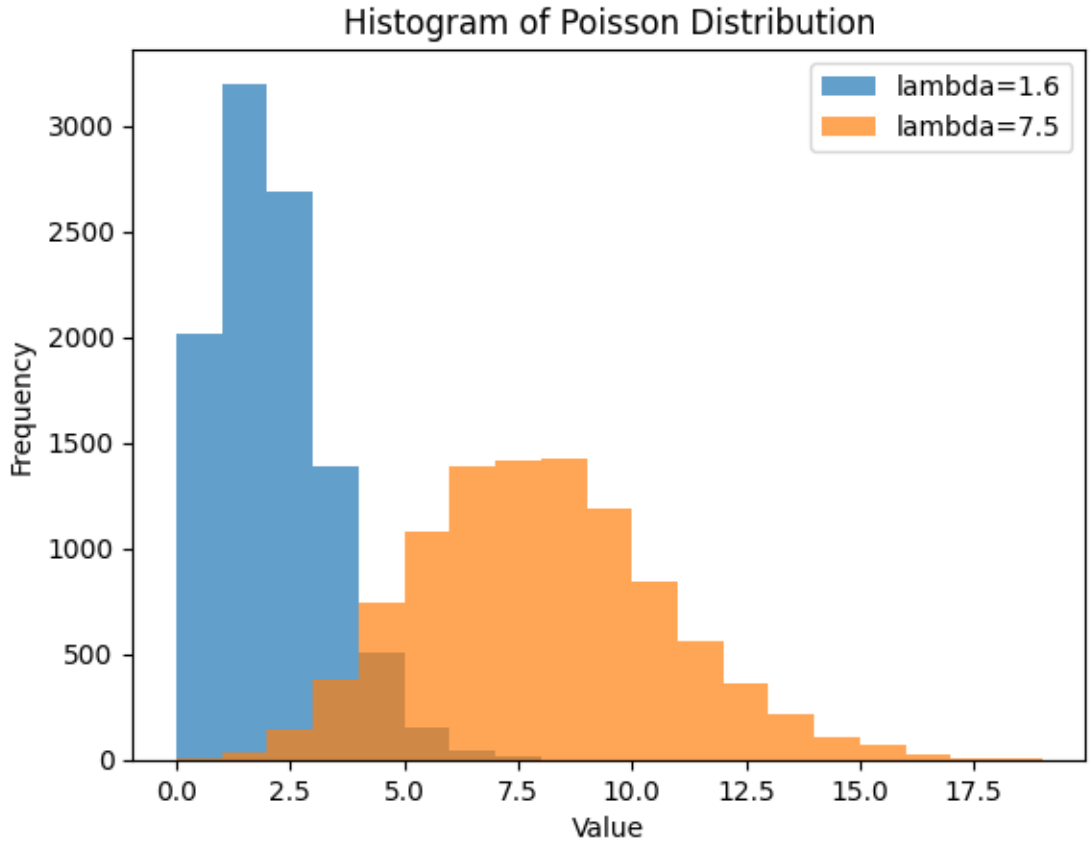
```
# Generate random variables
data1 = np.random.poisson(lambda1, size)
data2 = np.random.poisson(lambda2, size)
```

[ ]:
```
data1
```

[ ]:
```
array([735, 684, 666, …, 737, 654, 718], dtype=int64)
```

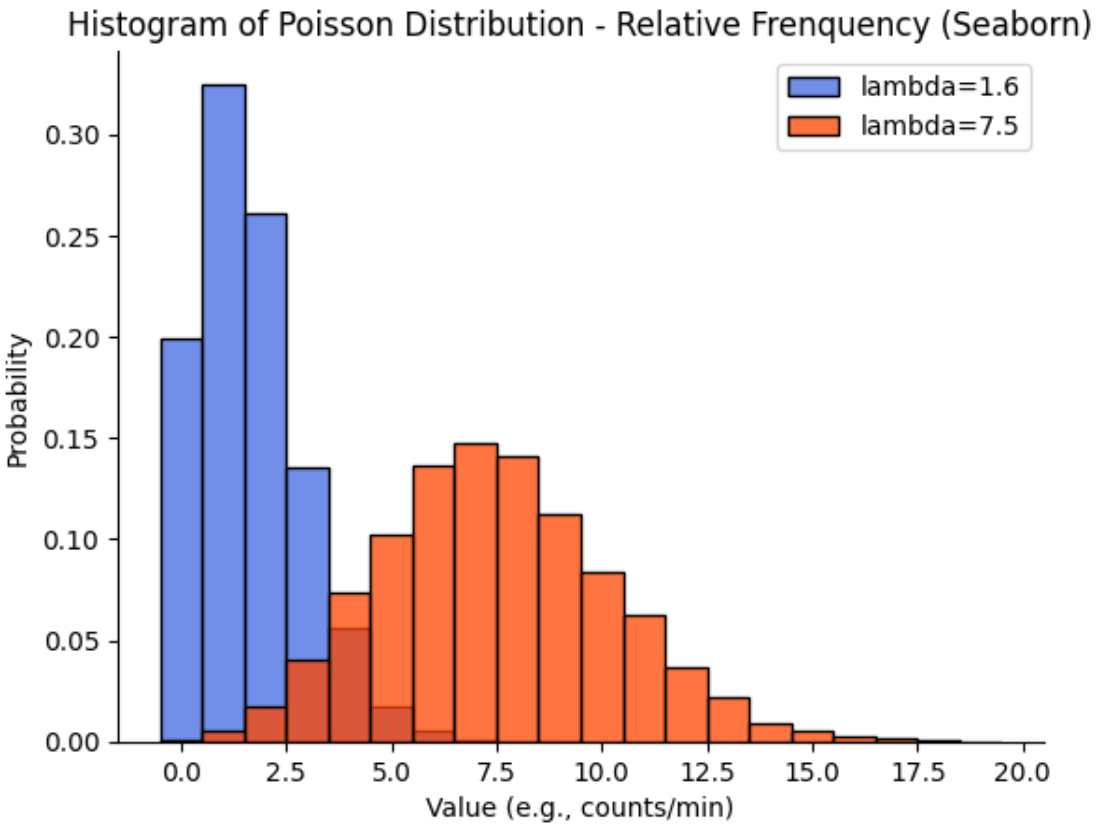
We then plot their histograms using `matplotlib`.

[ ]:
```
# Plot histograms using matplotlib
plt.hist(data1, bins=range(min(data1), max(data1) + 1), alpha=0.7,␣
 ↪label='lambda=1.6')
plt.hist(data2, bins=range(min(data2), max(data2) + 1), alpha=0.7,␣
 ↪label='lambda=7.5')
plt.title('Histogram of Poisson Distribution')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.legend(loc='upper right');
```



We can do similar things using the `scipy.stats` and `seaborn` packages.

[ ]:
```
import scipy.stats as ss
import seaborn as sns

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

sns.histplot(
    ss.poisson(lambda1).rvs(size=size),
    stat='probability',
    binwidth=1,
    discrete=True,
    label=f"lambda={lambda1}",
    color='royalblue',
)
sns.histplot(
    ss.poisson(lambda2).rvs(size=size),
    stat='probability',
    binwidth=1,
    discrete=True,
    label=f"lambda={lambda2}",
    color='orangered',
)

plt.title('Histogram of Poisson Distribution - Relative Frenquency (Seaborn)')
plt.xlabel('Value (e.g., counts/min)')
plt.ylabel('Probability')
plt.legend(loc='upper right')
sns.despine();
```



Remember, the Poisson distribution is discrete, so the values on the x-axis represent specific counts of events. The y-axis represents the frequency of these counts in our data set. The separation between the two histograms reflects the difference in the λ parameters of the two distributions. The distribution with $\lambda = 1.6$ is skewed to the right and has a peak at lower values, while the distribution with $\lambda = 7.5$ is more spread out and has a peak at higher values. This is consistent with the property of the Poisson distribution that its mean and variance are both equal to λ.

1.3 The example of raisins in bagels

1.3.1 Confidence interval

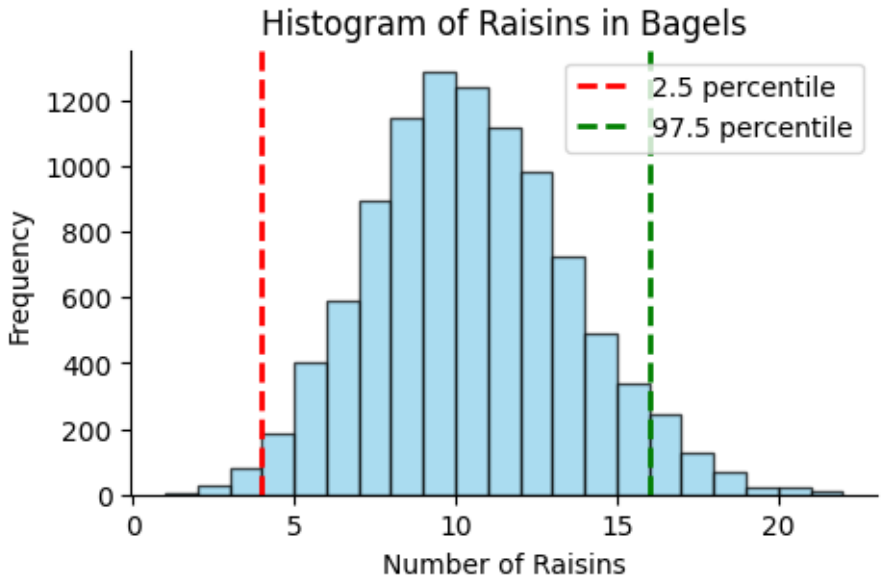
We'll use the `scipy.stats` Poisson distribution with an average (μ) of 10 raisins per bagel.

The Poisson distribution is a great fit for this scenario because it models the number of times an event (in this case, a raisin appearing in a bagel) occurs in a fixed interval of time or space. Here, our "fixed space" is the bagel, and the "event" is the appearance of a raisin.

We'll generate a sample data set, calculate the 2.5th and 97.5th percentiles of this data, which represent the lower and upper bounds of the 95% confidence interval, respectively, and plot the histogram.

[ ]:
```
# Set the parameters for the Poisson distribution
mu = 10 # average number of raisins in each bagel
size = 10000

# Generate random variables
data = ss.poisson.rvs(mu, size=size)

# Calculate the confidence interval
conf_int = np.percentile(data, [2.5, 97.5])

# Print the confidence interval
print(f"The 95% confidence interval is {conf_int}")

# Plot histogram
plt.figure(figsize=(5, 3))
plt.hist(data, bins=range(min(data), max(data) + 1), alpha=0.7,␣
 ↪color='skyblue', edgecolor='black')
plt.axvline(x=conf_int[0], color='red', linestyle='dashed', linewidth=2,␣
 ↪label='2.5 percentile')
plt.axvline(x=conf_int[1], color='green', linestyle='dashed', linewidth=2,␣
 ↪label='97.5 percentile')
plt.title('Histogram of Raisins in Bagels')
plt.xlabel('Number of Raisins')
plt.ylabel('Frequency')
plt.legend(loc='upper right')
sns.despine();
```

```
The 95% confidence interval is [ 4. 16.]
```



1.3.2 A shortcut approximation

The Central Limit Theorem states that the sum of a *large number* of independent and identically distributed random variables will approximately follow a *normal distribution*, regardless of the shape of their original distribution. In the case of the Poisson distribution, as the mean (λ) gets larger, the distribution becomes more symmetric and starts to look like a normal distribution.

When the count values are large (typically, $\lambda > 30$ is considered large enough), we can use the normal approximation to the Poisson distribution to calculate the confidence interval. The standard deviation of a Poisson distribution is the square root of λ (for Poisson distribution, the mean and the variance are equal to λ), so the standard error (which is the standard deviation of the sample mean) is the standard deviation divided by the square root of the sample size.

Therefore the **approximation for the 95% confidence interval** ranges from $\lambda - 1.96 \times \sqrt{\lambda}$ to $\lambda + 1.96 \times \sqrt{\lambda}$.

[ ]:
```
mu - 1.96 * np.sqrt(mu) , mu + 1.96 * np.sqrt(mu)
```

[ ]:
```
(3.8019357860699765, 16.198064213930024)
```

1.3.3 The Percent Point Function

The `ppf` stands for **Percent Point Function**, which is essentially the inverse of the Cumulative Distribution Function (CDF). The `ppf` takes a probability q and returns a value x such that the probability of the random variable being less than or equal to x is equal to q. In other words, it allows us to determine the value below which a given percentage of the data falls.

In the context of the Poisson distribution, the `ppf` can be used to calculate the value below which a certain percentage of the counts of events (e.g., raisins in bagels) fall. This can be particularly useful when you want to determine the confidence interval around the mean of the distribution.

[ ]:
```
# Set the parameters for the Poisson distribution
mu = 10
size = 10000

# Prepare the model of this distribution
model = ss.poisson(mu=mu)

# Calculate the confidence interval using ppf
conf_int = [model.ppf(0.025), model.ppf(0.975)]

# Print the confidence interval
print(f"The 95% confidence interval is {conf_int}")
```

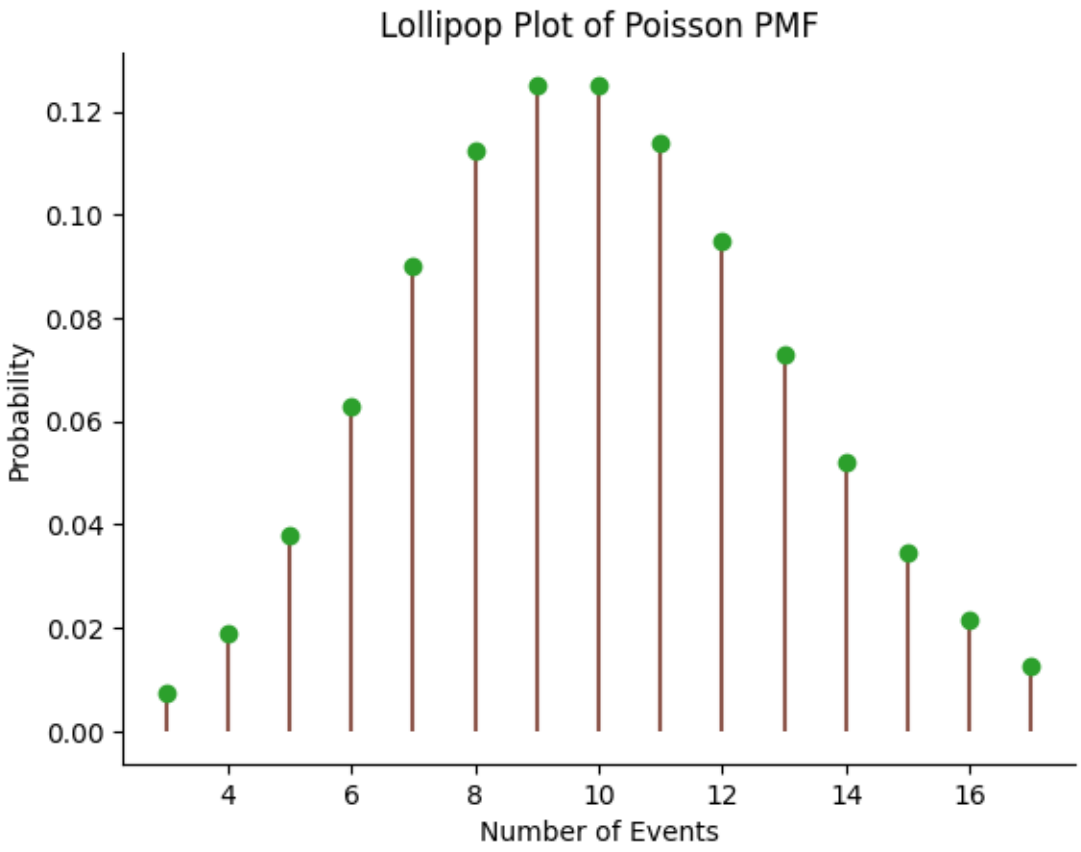
```
The 95% confidence interval is [4.0, 17.0]
```

1.3.4 Lollipop plot

The **Probability Mass Function** (PMF) of the Poisson distribution gives the probability of each possible outcome. In this case, it will give the probability of each possible count of events (e.g., raisins in a bagel).

We first generate the x values using the `ppf` function to get the range of values that contain 99% of the probability. We then calculate the PMF for these x values using the `pmf` function. Finally, we create the lollipop plot using the `stem` function in matplotlib.

[ ]:
```
# Set the parameters for the Poisson distribution
mu = 10

# Generate the x values
x = np.arange(
    model.ppf(0.01),
    model.ppf(0.99),
    # step=1,
)

# Calculate the PMF
pmf = ss.poisson.pmf(x, mu)

# Create the lollipop plot
plt.stem(x, pmf, markerfmt="C2o", linefmt="C5-", basefmt=" ")
# alernatively we can plot points + vertical lines
#plt.plot(x, model.pmf(x), 'ro', ms=8)
#plt.vlines(x, 0, model.pmf(x), colors='grey', lw=4, alpha=.5)
plt.title('Lollipop Plot of Poisson PMF')
plt.xlabel('Number of Events')
plt.ylabel('Probability')
sns.despine();
```



1.4 The example of radioactive count

let's create a Poisson model for radioactive counts with an average rate $\mu = 120$. We'll generate a sample data set, calculate the mean and variance, and estimate the confidence interval.

[ ]:
```
# Set the parameters for the Poisson distribution
mu = 120
size = 10000

# Generate random variables
desintegration_poisson_model = ss.poisson(mu=mu)
data = desintegration_poisson_model.rvs(size=size)

# Calculate the sample mean and variance
sample_mean = np.mean(data)
sample_variance = np.var(data)

# Calculate the confidence interval using 2.5th and 97.5th percentiles
conf_int = np.percentile(data, [2.5, 97.5])
# Calculate the confidence interval using ppf
```

```
# conf_int = [ss.poisson.ppf(0.025, mu), ss.poisson.ppf(0.975, mu)]

print(f"Sample Mean: {sample_mean}")
print(f"Sample Variance: {sample_variance}")
print(f"95% Confidence Interval: {conf_int}")
```

```
Sample Mean: 120.1426
Sample Variance: 121.87046524
95% Confidence Interval: [ 99. 142.]
```

There are several methods in the `scipy.stats.poisson` module that might be useful:

- `mean()`: This method returns the mean of the Poisson distribution. For a Poisson distribution, the mean is equal to mu (μ) or lambda (λ), which is the average rate of events.
- `stats()`: This method returns the mean ('m'), variance ('v'), skew ('s'), and/or kurtosis ('k') of the Poisson distribution. By default, it returns the mean and variance.
- `interval()`: This method returns the endpoints of the range that contains alpha percent of the distribution. For example, you can use it to calculate the confidence interval around the mean.

[ ]:
```
desintegration_poisson_model.mean()
```

[ ]:
```
120.0
```

[ ]:
```
mean, var, skew, kurt = desintegration_poisson_model.stats(moments='mvsk')
print(f"mean = {mean}")
```

```
mean = 120.0
```

[ ]:
```
desintegration_poisson_model.interval(.95)
```

[ ]:
```
(99.0, 142.0)
```

[ ]:
```
[desintegration_poisson_model.ppf(0.025), desintegration_poisson_model.ppf(0.
↪975)]
```

[ ]:
```
[99.0, 142.0]
```

[ ]:
```
# method used for large count values (i.e. >=25) as it approximates Gaussian␣
↪distribution
mu - 1.96 * mu**.5 , mu + 1.96 * mu**.5
```

[ ]:
```
(98.52927574579749, 141.4707242542025)
```

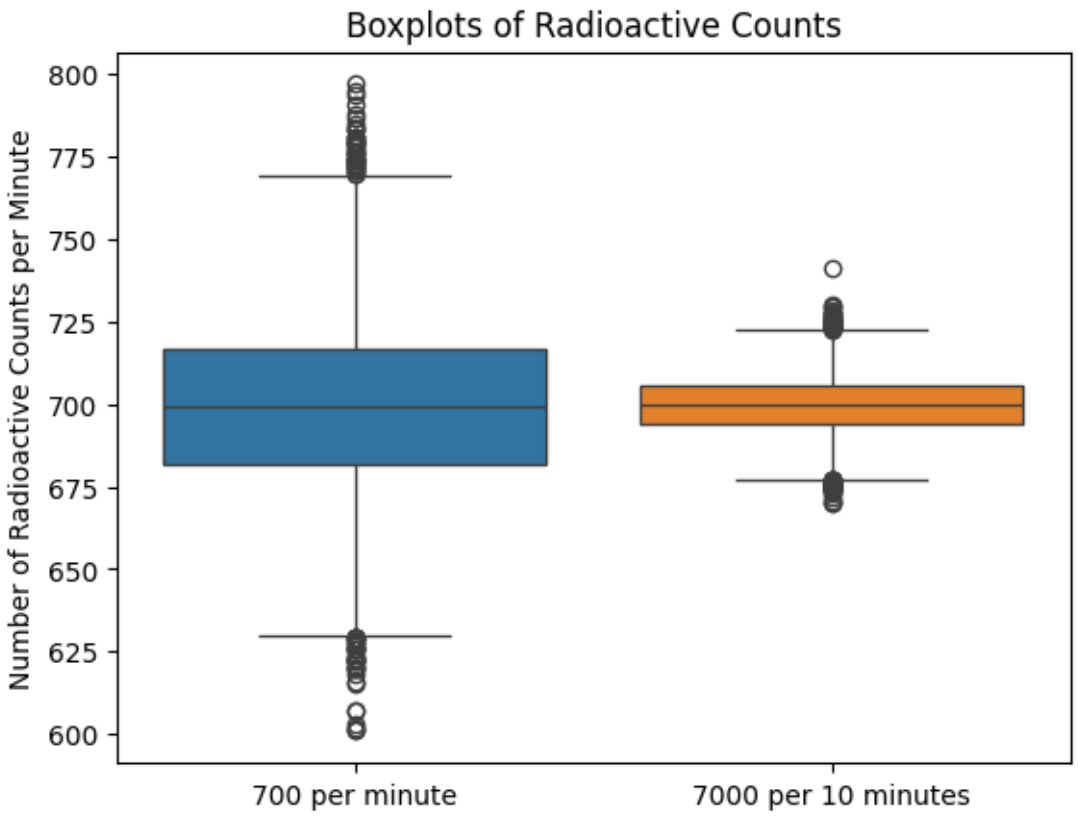
1.5 The advantage of counting for longer time intervals or in larger volumes

When we count events over longer time intervals or in larger volumes, the mean (λ) of the Poisson distribution increases. This results in a **narrower confidence interval**, which means our estimate of the average rate of events becomes more precise.

Let's illustrate this with Python code. We'll create two Poisson models: one with a mean of 700 events per minute, and another with a mean of 7000 events per 10 minutes.

[ ]:
```
# Set the parameters for the Poisson distribution
mu1 = 700  # 700 events per minute
mu2 = 7000  # 7000 events per 10 minutes
size = 10000

# Generate random variables
data1 = ss.poisson.rvs(mu1, size=size)
data2 = ss.poisson.rvs(mu2, size=size)

# Calculate the confidence intervals
conf_int1 = ss.poisson.interval(0.95, mu1)
conf_int2 = ss.poisson.interval(0.95, mu2)

print(f"The 95% confidence interval for mu=700 is {conf_int1}")
print(f"The 95% confidence interval for mu=7000 is {conf_int2}")

# Create boxplots
sns.boxplot(data=[data1, data2/10]) # normalize the second dataset
plt.xticks([0, 1], ['700 per minute', '7000 per 10 minutes'])
plt.title('Boxplots of Radioactive Counts')
plt.ylabel('Number of Radioactive Counts per Minute');
```

```
The 95% confidence interval for mu=700 is (649.0, 752.0)
The 95% confidence interval for mu=7000 is (6836.0, 7164.0)
```



When dealing with count data, especially in the context of a Poisson distribution, it's important to compute the confidence interval with the actual counts first and only then normalize by the time interval (or space interval) for several reasons:

1. Discreteness of Count Data: Count data are discrete, meaning they can only take on integer values (0, 1, 2, 3, …). The Poisson distribution, which is often used to model count data, is a discrete probability distribution. Therefore, it's appropriate to calculate the confidence interval based on the actual counts, which are also discrete.
2. Preservation of Variability: The actual counts capture the inherent variability in the data. When we divide by the time interval, we're essentially normalizing or standardizing the data, which can mask some of this variability. Calculating the confidence interval with the actual counts ensures that this variability is taken into account.
3. Accuracy of Confidence Interval: The confidence interval calculated with the actual counts gives the range of values that the true count is likely to fall within, with a certain level of confidence. If we were to divide by the time interval before calculating the confidence interval, the resulting interval would not accurately reflect the variability in the counts.

After calculating the confidence interval with the actual counts, we can then divide by the time interval to get the average rate of events per unit of time. This gives us a normalized measure, or the **average rate of events per unit of time** (or per unit of space), that is easier to interpret and compare across different time intervals or space intervals.

Remember, this normalization is valid under the assumption that the events are occurring at a

constant average rate. If the rate is not constant, the confidence interval for the average rate per unit of time might not be accurate.

[ ]:
```
print(f"The 95% confidence interval for the average rate mu=700/minute is␣
↪{[bound/10 for bound in conf_int2]}")
```

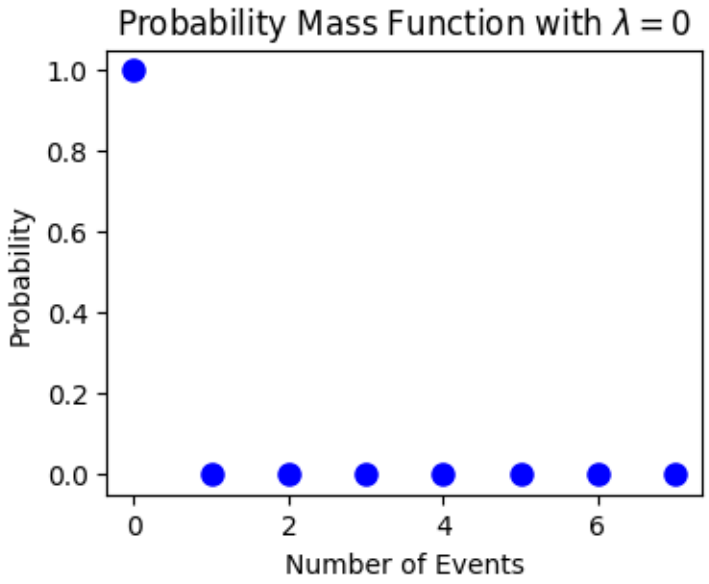
```
The 95% confidence interval for the average rate mu=700/minute is [683.6, 716.4]
```

1.6 What if the observed number of counts is 0?

In this section, we will explore the scenario where the observed number of events is 0. This corresponds to a Poisson distribution with a mean $\lambda = 0$.

The Poisson distribution is defined for all non-negative integers, so having an observed number of 0 is a valid scenario. However, if the mean of the distribution is 0, it means that the event is not expected to occur at all in the given interval of time or space.

[ ]:
```
# Set the parameters for the Poisson distribution
mu = 0

# Generate the x values
x = np.arange(0, 8)

# Calculate the PMF
pmf = ss.poisson.pmf(x, mu)

# Create the plot
plt.figure(figsize=(4,3))
plt.plot(x, pmf, 'bo', ms=8, label='poisson pmf')
plt.title(r'Probability Mass Function with $\lambda=0$')
plt.xlabel('Number of Events')
plt.ylabel('Probability');
```



As you we see from the plot, the probability of observing 0 events is 1, and the probability of observing any other number of events is 0. This is consistent with our expectation that if the average rate of events (λ) is 0, then we should not expect to observe any events.

In the case where the mean of the Poisson distribution is 0, the confidence interval would also be 0. This is because if the average rate of events (λ) is 0, then we should not expect to observe any events. Therefore, we can be 100% confident that the true average rate of occurrence (λ) lies within the interval [0, 0].

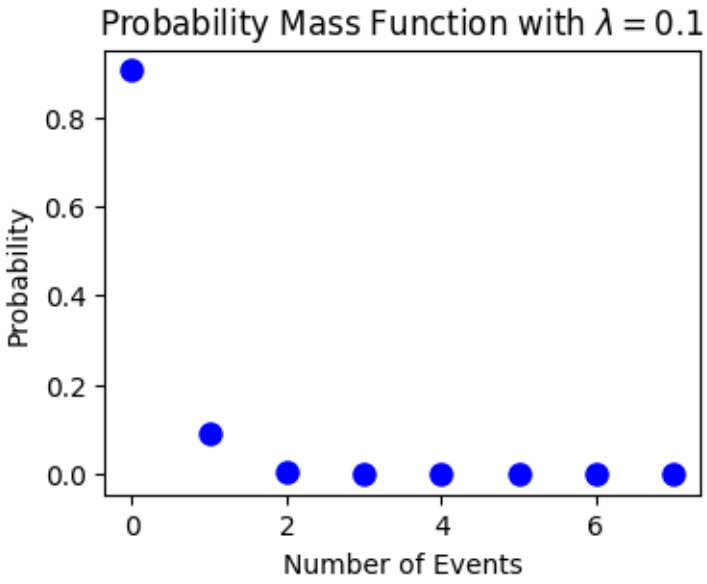
[ ]:
```
# Set the parameters for the Poisson distribution
mu = 0

# Calculate the confidence interval using ppf
conf_int = [ss.poisson.ppf(0.025, mu), ss.poisson.ppf(0.975, mu)]

print(f"The 95% confidence interval is {conf_int}")
```

```
The 95% confidence interval is [0.0, 0.0]
```

If the mean (λ) of the Poisson distribution is 0.1, it means that the event is expected to occur 0.1 times on average in the given interval of time or space. This would be a rare event, but it's still possible.

[ ]:
```
# Set the parameters for the Poisson distribution
mu = 0.1

# Generate the x values
x = np.arange(0, 8)
```

```
# Calculate the PMF
pmf = ss.poisson.pmf(x, mu)

# Create the plot
plt.figure(figsize=(4,3))
plt.plot(x, pmf, 'bo', ms=8, label='poisson pmf')
plt.title(r'Probability Mass Function with $\lambda=0.1$')
plt.xlabel('Number of Events')
plt.ylabel('Probability')
plt.show()

# Calculate the confidence interval using ppf
conf_int = [ss.poisson.ppf(0.025, mu), ss.poisson.ppf(0.975, mu)]

print(f"The 95% confidence interval is {conf_int}")
```




```
The 95% confidence interval is [0.0, 1.0]
```

2 Conclusion

In this chapter, we delved into the fascinating world of counted data and its statistical analysis using the Poisson Distribution. We explored how this powerful statistical tool can model discrete events in a fixed time or space, making it particularly useful in various fields such as epidemiology, traffic engineering, and social sciences.

We discussed several examples, including the number of raisins in bagels and radioactive counts, and showed how to generate data, calculate confidence intervals, and plot histograms for these

examples. We also explored different methods available in the `scipy.stats.poisson` module, such as `.pmf()`, `.cdf()`, `.ppf()`, and `.interval()`.

We highlighted the importance of computing the confidence interval with the actual counts before normalizing by the time interval. We also discussed the advantages of counting events over longer time intervals or in larger volumes, which leads to more precise estimates of the average rate of events.

Finally, we examined special cases where the mean (λ) of the Poisson distribution is 0 or close to 0, and discussed the implications of these scenarios.

3 Session Information

The output below details all packages and version necessary to reproduce the results in this report.

[ ]:
```
!python --version
print("-------------")
# List of packages we want to check the version
packages = ['numpy', 'scipy', 'matplotlib', 'seaborn']

# Initialize an empty list to store the versions
versions = []

# Loop over the packages
for package in packages:
    # Get the version of the package
    output = !pip show {package} | findstr "Version"
    # If the output is not empty, get the version
    if output:
        version = output[0].split()[-1]
    else:
        version = 'Not installed'
    # Append the version to the list
    versions.append(version)

# Print the versions
for package, version in zip(packages, versions):
    print(f'{package}: {version}')
```

```
Python 3.12.3
-------------
numpy: 1.26.4
scipy: 1.12.0
matplotlib: 3.8.3
seaborn: 0.13.2
```

[ ]:
```

```