Statistical Inference

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
from scipy.stats import t, norm, chi2, binom, poisson
import warnings
warnings.filterwarnings('ignore')
```

POINT ESTIMATES AND CONFIDENCE INTERVALS

```
print("Statistical Inference: Estimation for Cybersecurity Data")
print("="*60)

# Read in the data
cyber_data = pd.read_csv('/content/drive/MyDrive/AAI projects/Global_Cybersecurity_Threats_2015-2024.csv')
cyber_data.head()

print("\nDataset Overview:")
print("Shape: {}".format(cyber_data.shape))
print("Variables: {}".format(list(cyber_data.columns)))
```

```
Statistical Inference: Estimation for Cybersecurity Data
============================================================

Dataset Overview:
Shape: (3000, 10)
Variables: ['Country', 'Year', 'Attack Type', 'Target Industry', 'Financial Loss (in Million $)', 'Number of Affected User
```

4.1 Point Estimates and Confidence Intervals

```
### Point Estimators for Cybersecurity Metrics

print("\n### Point Estimators for Key Cybersecurity Metrics")

# Calculate point estimates for financial losses
financial_losses = cyber_data['Financial Loss (in Million $)']
mean_loss = np.mean(financial_losses)
median_loss = np.median(financial_losses)
std_loss = np.std(financial_losses, ddof=1)
se_mean = std_loss / np.sqrt(len(financial_losses))

print("Financial Loss Estimates:")
print("Sample Mean: ${:.2f}M".format(mean_loss))
print("Sample Median: ${:.2f}M".format(median_loss))
print("Sample Standard Deviation: ${:.2f}M".format(std_loss))
print("Standard Error of Mean: ${:.2f}M".format(se_mean))

# Calculate proportion estimates for attack types
attack_counts = cyber_data['Attack Type'].value_counts()
total_attacks = len(cyber_data)

print("\nAttack Type Proportions:")
for attack_type, count in attack_counts.head(6).items():
    proportion = count / total_attacks
    se_prop = np.sqrt(proportion * (1 - proportion) / total_attacks)
    print("{}: {:.3f} (SE: {:.3f})".format(attack_type, proportion, se_prop))

### Properties of Estimators: Unbiasedness, Consistency, Efficiency

print("\n### Testing Properties of Estimators")

def test_estimator_properties(data, n_simulations=1000):
    """Test unbiasedness and consistency of sample mean"""

    # True population parameters
    true_mean = np.mean(data)
    true_std = np.std(data, ddof=0)

    print("Testing Sample Mean Properties:")
    print("True Population Mean: {:.3f}".format(true_mean))

    # Test with different sample sizes
    sample_sizes = [30, 100, 500, 1000]
```

```
        for n in sample_sizes:
            sample_means = []

            # Generate many samples and calculate means
            for _ in range(n_simulations):
                sample = np.random.choice(data, size=min(n, len(data)), replace=True)
                sample_means.append(np.mean(sample))

            # Calculate bias and variance
            mean_of_means = np.mean(sample_means)
            bias = mean_of_means - true_mean
            variance = np.var(sample_means)

            print("Sample Size {}: Mean={:.3f}, Bias={:.3f}, Variance={:.4f}".format(
                n, mean_of_means, bias, variance))

    # Test properties for financial loss data
    test_estimator_properties(financial_losses)

    ### Confidence Intervals for Parameters

    print("\n### Confidence Intervals for Key Parameters")

    def calculate_confidence_interval(data, confidence=0.95):
        """Calculate confidence interval for population mean"""

        n = len(data)
        mean = np.mean(data)
        std = np.std(data, ddof=1)
        se = std / np.sqrt(n)

        # For large sample, use normal distribution
        if n >= 30:
            alpha = 1 - confidence
            z_critical = stats.norm.ppf(1 - alpha/2)
            margin_error = z_critical * se
            distribution_used = "Normal"
        else:
            # For small sample, use t-distribution
            alpha = 1 - confidence
            t_critical = stats.t.ppf(1 - alpha/2, df=n-1)
            margin_error = t_critical * se
            distribution_used = "t"

        lower = mean - margin_error
        upper = mean + margin_error

        return lower, upper, distribution_used

    # 95% CI for mean financial loss
    lower_95, upper_95, dist = calculate_confidence_interval(financial_losses, 0.95)
    print("95% Confidence Interval for Mean Financial Loss:")
    print("(${:.2f}M, ${:.2f}M) using {} distribution".format(lower_95, upper_95, dist))

    # 99% CI for mean financial loss
    lower_99, upper_99, dist = calculate_confidence_interval(financial_losses, 0.99)
    print("99% Confidence Interval for Mean Financial Loss:")
    print("(${:.2f}M, ${:.2f}M) using {} distribution".format(lower_99, upper_99, dist))

    # CI for proportion of successful attacks (top attack type)
    top_attack = attack_counts.index[0]
    top_count = attack_counts.iloc[0]
    p_hat = top_count / total_attacks

    def proportion_confidence_interval(p_hat, n, confidence=0.95):
        """Calculate confidence interval for proportion"""

        alpha = 1 - confidence
        z_critical = stats.norm.ppf(1 - alpha/2)
        se_prop = np.sqrt(p_hat * (1 - p_hat) / n)
        margin_error = z_critical * se_prop

        lower = p_hat - margin_error
        upper = p_hat + margin_error

        return max(0, lower), min(1, upper)

    prop_lower, prop_upper = proportion_confidence_interval(p_hat, total_attacks, 0.95)
    print("\n95% Confidence Interval for Proportion of {} attacks:".format(top_attack))
    print("({:.3f}, {:.3f})".format(prop_lower, prop_upper))
```

```
### Point Estimators for Key Cybersecurity Metrics
Financial Loss Estimates:
Sample Mean: $50.49M
```

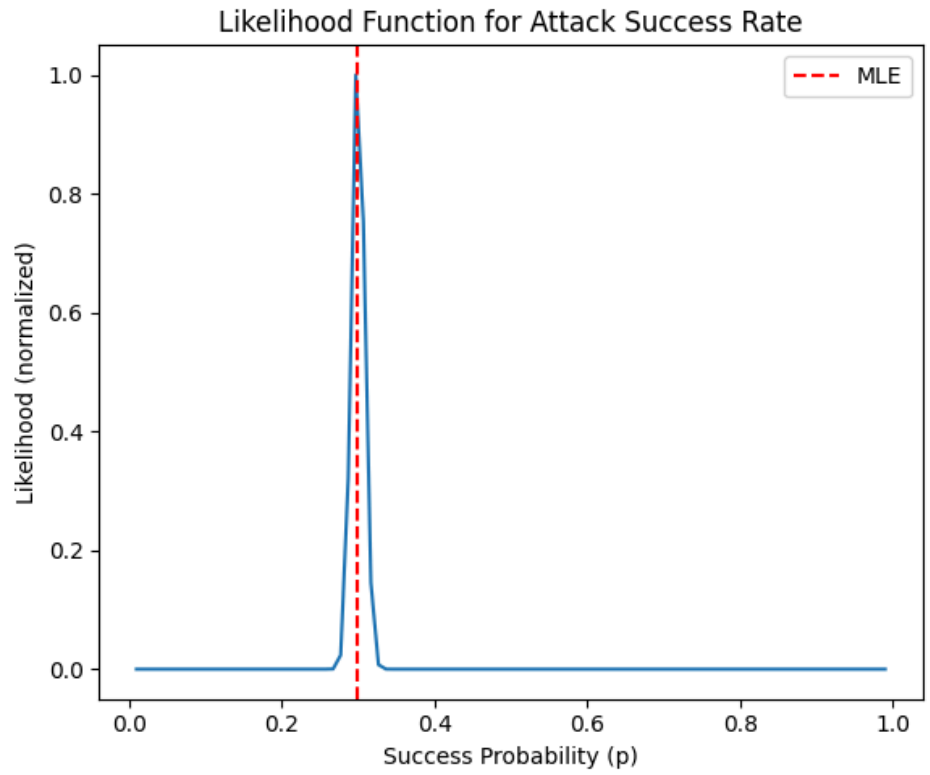
```
Sample Median: $50.80M
Sample Standard Deviation: $28.79M
Standard Error of Mean: $0.53M

Attack Type Proportions:
DDoS: 0.177 (SE: 0.007)
Phishing: 0.176 (SE: 0.007)
SQL Injection: 0.168 (SE: 0.007)
Ransomware: 0.164 (SE: 0.007)
Malware: 0.162 (SE: 0.007)
Man-in-the-Middle: 0.153 (SE: 0.007)

### Testing Properties of Estimators
Testing Sample Mean Properties:
True Population Mean: 50.493
Sample Size 30: Mean=50.735, Bias=0.242, Variance=26.6613
Sample Size 100: Mean=50.489, Bias=-0.004, Variance=8.3996
Sample Size 500: Mean=50.499, Bias=0.006, Variance=1.6041
Sample Size 1000: Mean=50.478, Bias=-0.015, Variance=0.8436

### Confidence Intervals for Key Parameters
95% Confidence Interval for Mean Financial Loss:
($49.46M, $51.52M) using Normal distribution
99% Confidence Interval for Mean Financial Loss:
($49.14M, $51.85M) using Normal distribution

95% Confidence Interval for Proportion of DDoS attacks:
(0.163, 0.191)
```

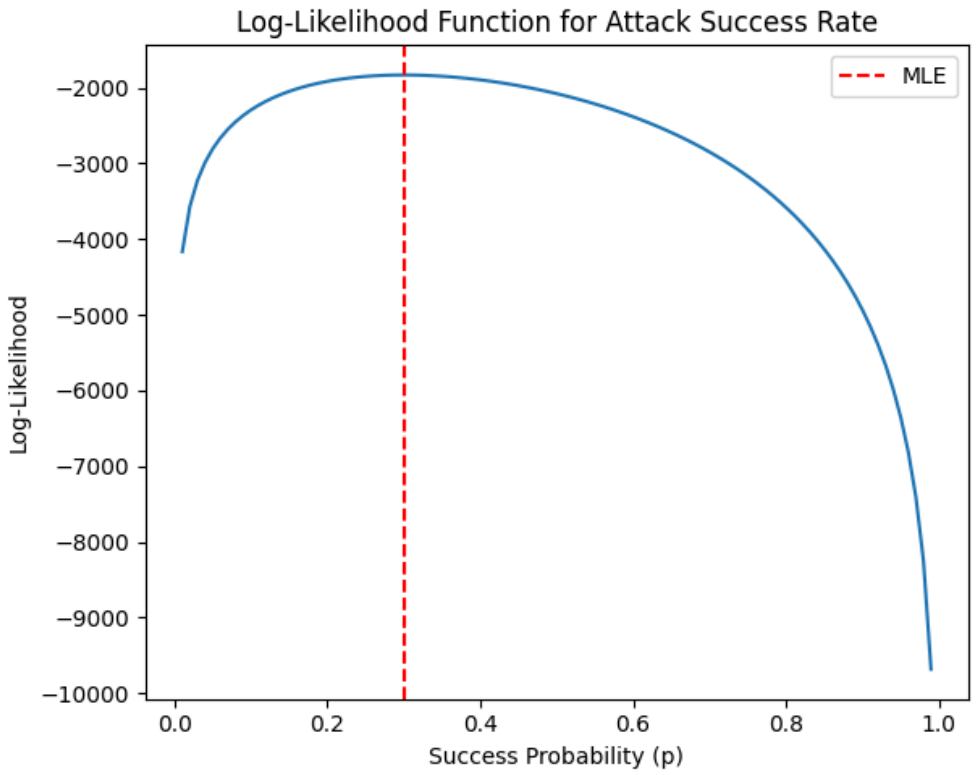
4.2 Maximum Likelihood Estimation

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

print("\n" + "="*60)
print("4.2 MAXIMUM LIKELIHOOD ESTIMATION")
print("="*60)

### Constructing Likelihood Functions

print("\n### Constructing Likelihood Functions")

# Create binary success indicator for attacks
financial_threshold = cyber_data['Financial Loss (in Million $)'].quantile(0.7)
cyber_data['attack_success'] = (cyber_data['Financial Loss (in Million $)'] >= financial_threshold).astype(int)
successes = cyber_data['attack_success'].sum()
total_trials = len(cyber_data['attack_success'])

print("Attack Success Analysis:")
print("Number of successes: {}".format(successes))
print("Total trials: {}".format(total_trials))
print("Observed success rate: {:.3f}".format(successes / total_trials))

### Binomial Log-Likelihood (numerically stable)

def log_likelihood_binomial(p, k, n):
    """Log-likelihood for binomial distribution (ignoring combinatorial term)"""
    return k * np.log(p) + (n - k) * np.log(1 - p)

# Plot likelihood and log-likelihood for binomial
p_values = np.linspace(0.01, 0.99, 100)
log_likelihoods = [log_likelihood_binomial(p, successes, total_trials) for p in p_values]

# Normalize likelihoods for plotting
max_log_lik = np.max(log_likelihoods)
likelihoods = [np.exp(ll - max_log_lik) for ll in log_likelihoods]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12,5))

# Likelihood plot (normalized)
ax1.plot(p_values, likelihoods)
ax1.set_title('Likelihood Function for Attack Success Rate')
ax1.set_xlabel('Success Probability (p)')
ax1.set_ylabel('Likelihood (normalized)')
ax1.axvline(x=successes/total_trials, color='red', linestyle='--', label='MLE')
ax1.legend()

# Log-likelihood plot
ax2.plot(p_values, log_likelihoods)
ax2.set_title('Log-Likelihood Function for Attack Success Rate')
ax2.set_xlabel('Success Probability (p)')
ax2.set_ylabel('Log-Likelihood')
ax2.axvline(x=successes/total_trials, color='red', linestyle='--', label='MLE')
```

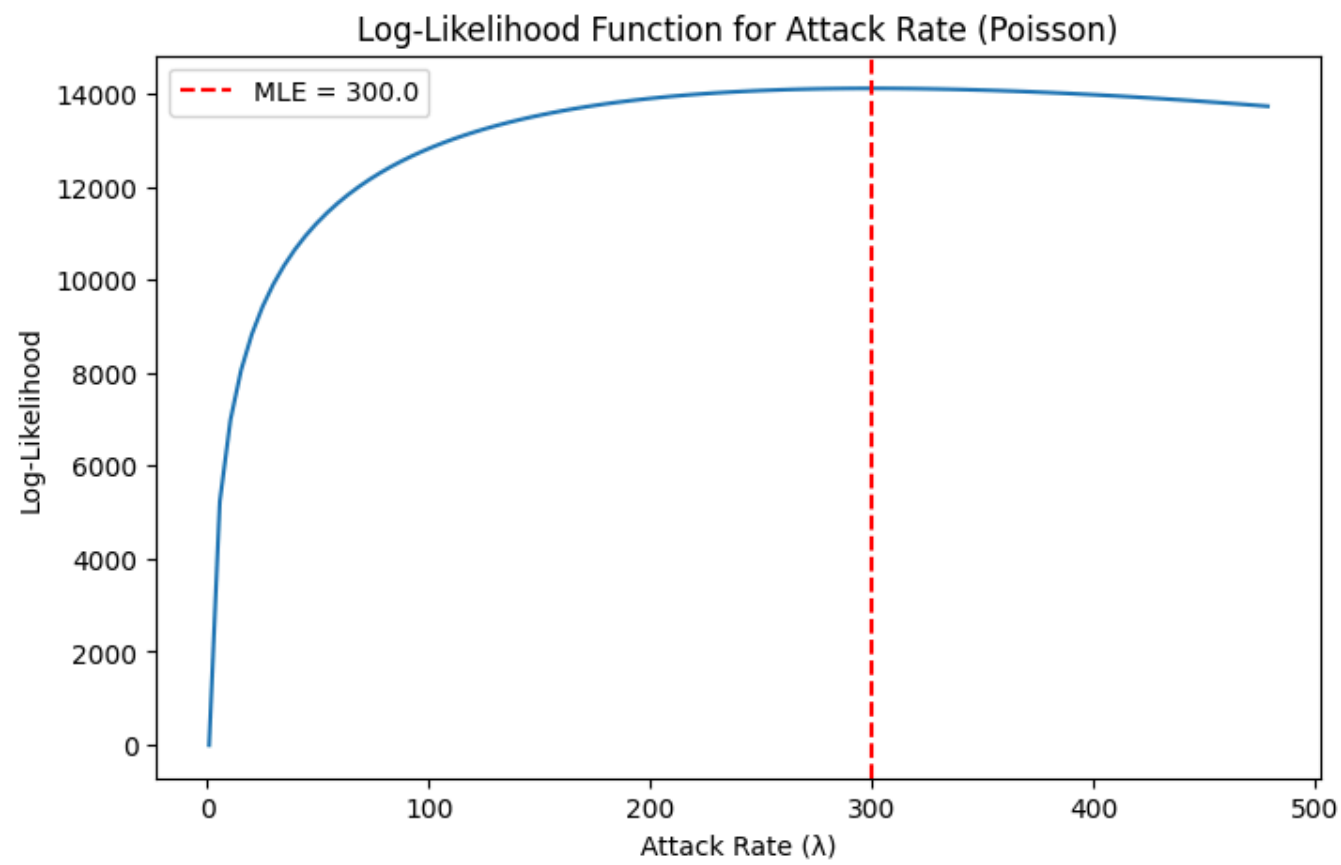
```
ax2.legend()

plt.tight_layout()
plt.show()

### Maximum Likelihood Estimates

print("\n### Maximum Likelihood Estimates")

# Binomial MLE
mle_p = successes / total_trials
print("MLE for Attack Success Rate: {:.4f}".format(mle_p))

# Poisson MLE: attacks per year
attacks_per_year = cyber_data.groupby('Year').size()
print("\nAttacks per year:")
for year, count in attacks_per_year.items():
    print("Year {}: {} attacks".format(year, count))

mle_lambda = np.mean(attacks_per_year)
print("MLE for Attack Rate (λ): {:.3f} attacks per year".format(mle_lambda))

# Poisson log-likelihood (numerically stable)
lambda_values = np.linspace(1, max(attacks_per_year)*1.5, 100)
poisson_log_likelihoods = [sum(count*np.log(lam) - lam for count in attacks_per_year) for lam in lambda_values]

plt.figure(figsize=(8,5))
plt.plot(lambda_values, poisson_log_likelihoods)
plt.title('Log-Likelihood Function for Attack Rate (Poisson)')
plt.xlabel('Attack Rate (λ)')
plt.ylabel('Log-Likelihood')
plt.axvline(x=mle_lambda, color='red', linestyle='--', label=f'MLE = {mle_lambda:.1f}')
plt.legend()
plt.show()

### Variance of ML Estimators

print("\n### Properties of Maximum Likelihood Estimators")

# Fisher Information and Cramér-Rao Lower Bound for binomial
def fisher_information_binomial(p, n):
    return n / (p * (1 - p))

fisher_info_bin = fisher_information_binomial(mle_p, total_trials)
cramer_rao_bin = 1 / fisher_info_bin

print("Binomial Parameter Variance Analysis:")
print("Fisher Information: {:.2f}".format(fisher_info_bin))
print("Cramér-Rao Lower Bound: {:.6f}".format(cramer_rao_bin))
print("Theoretical MLE Variance: {:.6f}".format(mle_p * (1 - mle_p) / total_trials))

# Fisher Information and Cramér-Rao Lower Bound for Poisson
poisson_fisher_info = len(attacks_per_year) / mle_lambda
poisson_cramer_rao = 1 / poisson_fisher_info

print("\nPoisson Parameter Variance Analysis:")
print("Fisher Information: {:.4f}".format(poisson_fisher_info))
print("Cramér-Rao Lower Bound: {:.6f}".format(poisson_cramer_rao))
```

```
============================================================
4.2 MAXIMUM LIKELIHOOD ESTIMATION
============================================================

### Constructing Likelihood Functions
Attack Success Analysis:
Number of successes: 900
Total trials: 3000
Observed success rate: 0.300
```






```
### Maximum Likelihood Estimates
MLE for Attack Success Rate: 0.3000

Attacks per year:
Year 2015: 277 attacks
Year 2016: 285 attacks
Year 2017: 319 attacks
Year 2018: 310 attacks
Year 2019: 263 attacks
Year 2020: 315 attacks
Year 2021: 299 attacks
Year 2022: 318 attacks
Year 2023: 315 attacks
Year 2024: 299 attacks
MLE for Attack Rate (λ): 300.000 attacks per year
```




```
### Properties of Maximum Likelihood Estimators
Binomial Parameter Variance Analysis:
Fisher Information: 14285.71
Cramér-Rao Lower Bound: 0.000070
Theoretical MLE Variance: 0.000070

Poisson Parameter Variance Analysis:
Fisher Information: 0.0333
Cramér-Rao Lower Bound: 30.000000
```

4.3 Constructing Advanced Confidence Intervals

```
import numpy as np
from scipy import stats

# --- Confidence Interval for Mean using Normal/T ---
def pivotal_ci_mean(data, confidence=0.95):
    n = len(data)
    mean = np.mean(data)
    se = np.std(data, ddof=1) / np.sqrt(n)
    alpha = 1 - confidence

    if n >= 30:
        crit = stats.norm.ppf(1 - alpha/2)
    else:
        crit = stats.t.ppf(1 - alpha/2, df=n-1)

    margin = crit * se
    return mean - margin, mean + margin

# Large-sample CI for mean resolution time
resolution_times = cyber_data['Incident Resolution Time (in Hours)']
lower_res, upper_res = pivotal_ci_mean(resolution_times)
print("95% CI for Mean Resolution Time: ({:.2f}, {:.2f}) hours".format(lower_res, upper_res))

# --- Confidence Intervals for Proportions (Wald and Wilson) ---
def proportion_ci_methods(x, n, confidence=0.95):
    p_hat = x / n
    z = stats.norm.ppf(1 - (1 - confidence)/2)

    # Wald
    se_wald = np.sqrt(p_hat * (1 - p_hat) / n)
    wald = (max(0, p_hat - z*se_wald), min(1, p_hat + z*se_wald))

    # Wilson
    denom = 1 + z**2/n
    center = (p_hat + z**2/(2*n)) / denom
    se_wilson = np.sqrt((p_hat*(1-p_hat)/n) + (z**2/(4*n**2))) / denom
    wilson = (max(0, center - z*se_wilson), min(1, center + z*se_wilson))

    return {'wald': wald, 'wilson': wilson}

ci_methods = proportion_ci_methods(successes, total_trials)
print("\n95% Confidence Intervals for Attack Success Rate:")
print("Wald Interval: ({:.4f}, {:.4f})".format(*ci_methods['wald']))
print("Wilson Interval: ({:.4f}, {:.4f})".format(*ci_methods['wilson']))


# --- Sample Size Calculations ---
def required_sample_size_mean(sigma, margin, confidence=0.95):
    z = stats.norm.ppf(1 - (1-confidence)/2)
    return int(np.ceil((z * sigma / margin)**2))

def required_sample_size_proportion(p, margin, confidence=0.95):
    z = stats.norm.ppf(1 - (1-confidence)/2)
    return int(np.ceil((z**2 * p * (1 - p)) / margin**2))

# Example: sample size for mean and proportion
std_loss = np.std(cyber_data['Financial Loss (in Million $)'], ddof=1)
required_n_mean = required_sample_size_mean(std_loss, 5)
required_n_prop = required_sample_size_proportion(mle_p, 0.02)

print("\nRequired sample size to estimate mean financial loss within $5M:", required_n_mean)
print("Required sample size to estimate success rate within 2%:", required_n_prop)
```

```
95% CI for Mean Resolution Time: (35.74, 37.21) hours

95% Confidence Intervals for Attack Success Rate:
Wald Interval: (0.2836, 0.3164)
Wilson Interval: (0.2839, 0.3166)

Required sample size to estimate mean financial loss within $5M: 128
Required sample size to estimate success rate within 2%: 2017
```

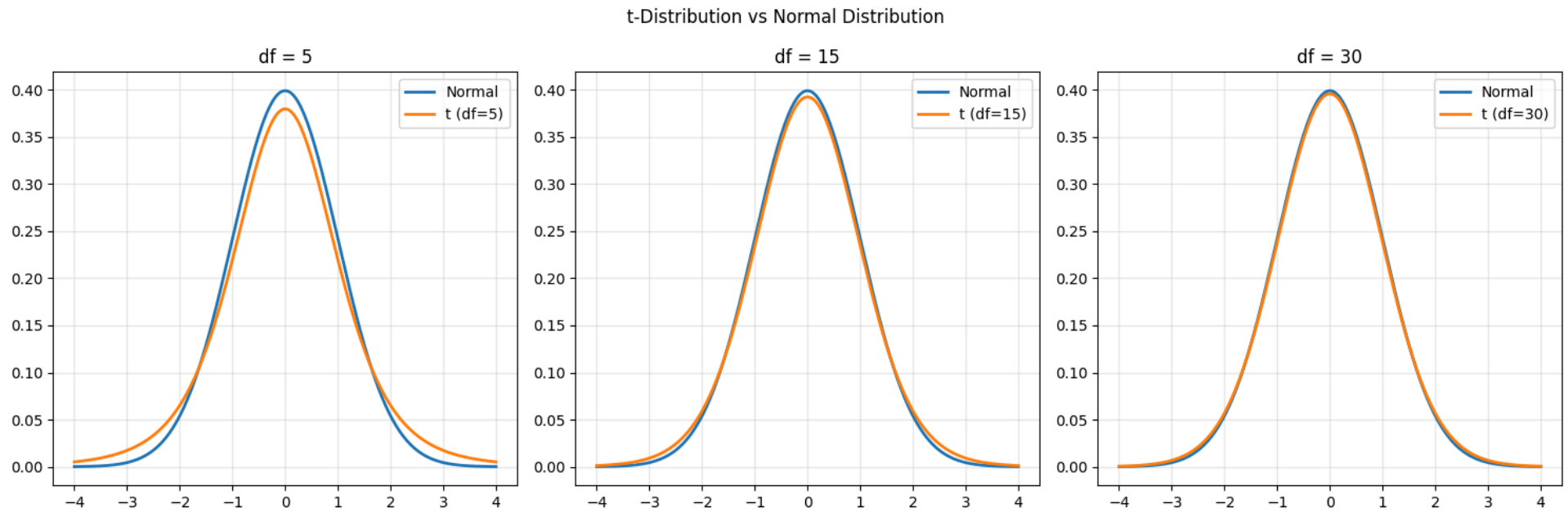
4.4 t-Distribution Applications

```
print("\n" + "="*60)
print("4.4 t-DISTRIBUTION CONFIDENCE INTERVALS")
```

```
print("="*60)

### The t Distribution

print("\n### Comparing t and Normal Distributions")

# Demonstrate t vs normal for different sample sizes
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

x = np.linspace(-4, 4, 1000)
normal_dist = stats.norm.pdf(x)

dfs = [5, 15, 30]
for i, df in enumerate(dfs):
    t_dist = stats.t.pdf(x, df)

    axes[i].plot(x, normal_dist, label='Normal', linewidth=2)
    axes[i].plot(x, t_dist, label='t (df={})'.format(df), linewidth=2)
    axes[i].set_title('df = {}'.format(df))
    axes[i].legend()
    axes[i].grid(True, alpha=0.3)

plt.suptitle('t-Distribution vs Normal Distribution')
plt.tight_layout()
plt.show()

### Confidence Intervals Using t-Distribution

print("\n### t-based Confidence Intervals")

def t_confidence_interval(data, confidence=0.95):
    """Calculate t-based confidence interval"""

    n = len(data)
    mean = np.mean(data)
    std = np.std(data, ddof=1)
    se = std / np.sqrt(n)

    alpha = 1 - confidence
    t_critical = stats.t.ppf(1 - alpha/2, df=n-1)
    margin_error = t_critical * se

    return mean - margin_error, mean + margin_error, t_critical

# Select a subset for small sample analysis
small_sample = cyber_data.sample(25)['Financial Loss (in Million $)']
t_lower, t_upper, t_crit = t_confidence_interval(small_sample, 0.95)

print("Small Sample Analysis (n=25):")
print("Sample mean: ${:.2f}M".format(np.mean(small_sample)))
print("95% t-based CI: (${:.2f}M, ${:.2f}M)".format(t_lower, t_upper))
print("t-critical value (df=24): {:.3f}".format(t_crit))

# Compare with normal-based CI
z_critical = stats.norm.ppf(0.975)
se_small = np.std(small_sample, ddof=1) / np.sqrt(25)
normal_margin = z_critical * se_small
normal_lower = np.mean(small_sample) - normal_margin
normal_upper = np.mean(small_sample) + normal_margin

print("95% Normal-based CI: (${:.2f}M, ${:.2f}M)".format(normal_lower, normal_upper))
print("CI width difference: ${:.2f}M".format((t_upper - t_lower) - (normal_upper - normal_lower)))

### Robustness Testing

print("\n### Testing Robustness of t-Distribution")

def test_normality(data, test_name):
    """Test normality assumption"""

    # Shapiro-Wilk test
    stat, p_value = stats.shapiro(data[:5000] if len(data) > 5000 else data)  # Shapiro limited to 5000

    print("{} Normality Test:".format(test_name))
    print("Shapiro-Wilk p-value: {:.6f}".format(p_value))

    if p_value < 0.05:
        print("Data appears non-normal (reject normality)")
    else:
        print("Data appears normal (fail to reject normality)")

# Test normality of financial losses
test_normality(financial_losses, "Financial Loss")

# Test log-transformed data
```

```
log_financial = np.log(financial_losses + 1)
test_normality(log_financial, "Log-transformed Financial Loss")
```

```
============================================================
4.4 t-DISTRIBUTION CONFIDENCE INTERVALS
============================================================

### Comparing t and Normal Distributions
```




```
### t-based Confidence Intervals
Small Sample Analysis (n=25):
Sample mean: $43.94M
95% t-based CI: ($32.27M, $55.62M)
t-critical value (df=24): 2.064
95% Normal-based CI: ($32.86M, $55.03M)
CI width difference: $1.18M

### Testing Robustness of t-Distribution
Financial Loss Normality Test:
Shapiro-Wilk p-value: 0.000000
Data appears non-normal (reject normality)
Log-transformed Financial Loss Normality Test:
Shapiro-Wilk p-value: 0.000000
Data appears non-normal (reject normality)
```

4.5 Comparing Two Population Means or Proportions

```
### Comparing Means Between Groups

print("\n### Comparing Mean Financial Losses Between Attack Types")

# Get top two attack types for comparison
top_attacks = cyber_data['Attack Type'].value_counts().head(2)
attack1_name = top_attacks.index[0]
attack2_name = top_attacks.index[1]

attack1_data = cyber_data[cyber_data['Attack Type'] == attack1_name]['Financial Loss (in Million $)']
attack2_data = cyber_data[cyber_data['Attack Type'] == attack2_name]['Financial Loss (in Million $)']

print("Comparing: {} vs {}".format(attack1_name, attack2_name))
print("{} - n: {}, mean: ${:.2f}M, std: ${:.2f}M".format(
    attack1_name, len(attack1_data), np.mean(attack1_data), np.std(attack1_data, ddof=1)))
print("{} - n: {}, mean: ${:.2f}M, std: ${:.2f}M".format(
    attack2_name, len(attack2_data), np.mean(attack2_data), np.std(attack2_data, ddof=1)))

def two_sample_t_test(data1, data2, equal_var=True, confidence=0.95):
    """Perform two-sample t-test and construct CI"""

    n1, n2 = len(data1), len(data2)
    mean1, mean2 = np.mean(data1), np.mean(data2)
    var1, var2 = np.var(data1, ddof=1), np.var(data2, ddof=1)

    diff_means = mean1 - mean2

    if equal_var:
        # Pooled variance
        pooled_var = ((n1-1)*var1 + (n2-1)*var2) / (n1 + n2 - 2)
        se_diff = np.sqrt(pooled_var * (1/n1 + 1/n2))
        df = n1 + n2 - 2
    else:
        # Welch's t-test
        se_diff = np.sqrt(var1/n1 + var2/n2)
        df = (var1/n1 + var2/n2)**2 / ((var1/n1)**2/(n1-1) + (var2/n2)**2/(n2-1))
```

```
    # t-statistic
    t_stat = diff_means / se_diff

    # p-value (two-tailed)
    p_value = 2 * (1 - stats.t.cdf(abs(t_stat), df))

    # Confidence interval for difference
    alpha = 1 - confidence
    t_critical = stats.t.ppf(1 - alpha/2, df)
    margin_error = t_critical * se_diff

    ci_lower = diff_means - margin_error
    ci_upper = diff_means + margin_error

    return {
        'mean_diff': diff_means,
        't_stat': t_stat,
        'p_value': p_value,
        'df': df,
        'ci': (ci_lower, ci_upper)
    }

# Perform two-sample t-test
result_equal_var = two_sample_t_test(attack1_data, attack2_data, equal_var=True)
result_unequal_var = two_sample_t_test(attack1_data, attack2_data, equal_var=False)

print("\nTwo-Sample t-Test Results (Equal Variances):")
print("Mean Difference: ${:.2f}M".format(result_equal_var['mean_diff']))
print("t-statistic: {:.3f}".format(result_equal_var['t_stat']))
print("p-value: {:.6f}".format(result_equal_var['p_value']))
print("95% CI for difference: (${:.2f}M, ${:.2f}M)".format(
    result_equal_var['ci'][0], result_equal_var['ci'][1]))

print("\nTwo-Sample t-Test Results (Unequal Variances - Welch):")
print("Mean Difference: ${:.2f}M".format(result_unequal_var['mean_diff']))
print("t-statistic: {:.3f}".format(result_unequal_var['t_stat']))
print("p-value: {:.6f}".format(result_unequal_var['p_value']))
print("95% CI for difference: (${:.2f}M, ${:.2f}M)".format(
    result_unequal_var['ci'][0], result_unequal_var['ci'][1]))

# Test for equal variances (F-test)
f_stat = np.var(attack1_data, ddof=1) / np.var(attack2_data, ddof=1)
f_p_value = 2 * min(stats.f.cdf(f_stat, len(attack1_data)-1, len(attack2_data)-1),
                    1 - stats.f.cdf(f_stat, len(attack1_data)-1, len(attack2_data)-1))

print("\nTest for Equal Variances:")
print("F-statistic: {:.3f}".format(f_stat))
print("p-value: {:.6f}".format(f_p_value))
if f_p_value < 0.05:
    print("Reject equal variances assumption")
else:
    print("Fail to reject equal variances assumption")

### Comparing Proportions Between Groups

print("\n### Comparing Attack Success Rates Between Countries")

# Get top two countries by number of attacks
top_countries = cyber_data['Country'].value_counts().head(2)
country1_name = top_countries.index[0]
country2_name = top_countries.index[1]

country1_data = cyber_data[cyber_data['Country'] == country1_name]
country2_data = cyber_data[cyber_data['Country'] == country2_name]

# Calculate success rates
country1_successes = country1_data['attack_success'].sum()
country1_total = len(country1_data)
country1_prop = country1_successes / country1_total

country2_successes = country2_data['attack_success'].sum()
country2_total = len(country2_data)
country2_prop = country2_successes / country2_total

print("Comparing Success Rates: {} vs {}".format(country1_name, country2_name))
print("{}: {}/{} = {:.3f}".format(country1_name, country1_successes, country1_total, country1_prop))
print("{}: {}/{} = {:.3f}".format(country2_name, country2_successes, country2_total, country2_prop))

def two_proportion_test(x1, n1, x2, n2, confidence=0.95):
    """Test difference between two proportions"""

    p1 = x1 / n1
    p2 = x2 / n2
    p_diff = p1 - p2
```

```
    # Pooled proportion for test statistic
    p_pool = (x1 + x2) / (n1 + n2)
    se_pool = np.sqrt(p_pool * (1 - p_pool) * (1/n1 + 1/n2))

    # Z-test statistic
    z_stat = p_diff / se_pool

    # p-value (two-tailed)
    p_value = 2 * (1 - stats.norm.cdf(abs(z_stat)))

    # Confidence interval for difference
    se_diff = np.sqrt(p1*(1-p1)/n1 + p2*(1-p2)/n2)
    alpha = 1 - confidence
    z_critical = stats.norm.ppf(1 - alpha/2)
    margin_error = z_critical * se_diff

    ci_lower = p_diff - margin_error
    ci_upper = p_diff + margin_error

    return {
        'prop_diff': p_diff,
        'z_stat': z_stat,
        'p_value': p_value,
        'ci': (ci_lower, ci_upper)
    }

# Perform two-proportion test
prop_result = two_proportion_test(country1_successes, country1_total,
                                  country2_successes, country2_total)

print("\nTwo-Proportion Test Results:")
print("Proportion Difference: {:.4f}".format(prop_result['prop_diff']))
print("z-statistic: {:.3f}".format(prop_result['z_stat']))
print("p-value: {:.6f}".format(prop_result['p_value']))
print("95% CI for difference: ({:.4f}, {:.4f})".format(
    prop_result['ci'][0], prop_result['ci'][1]))

# Interpretation
alpha = 0.05
print("\nInterpretation (α = 0.05):")
if prop_result['p_value'] <= alpha:
    print("There is a significant difference in success rates between countries.")
else:
    print("There is no significant difference in success rates between countries.")
```

```
### Comparing Mean Financial Losses Between Attack Types
Comparing: DDoS vs Phishing
DDoS - n: 531, mean: $52.04M, std: $29.26M
Phishing - n: 529, mean: $50.46M, std: $29.16M

Two-Sample t-Test Results (Equal Variances):
Mean Difference: $1.58M
t-statistic: 0.878
p-value: 0.380053
95% CI for difference: ($-1.95M, $5.10M)

Two-Sample t-Test Results (Unequal Variances - Welch):
Mean Difference: $1.58M
t-statistic: 0.878
p-value: 0.380050
95% CI for difference: ($-1.95M, $5.10M)

Test for Equal Variances:
F-statistic: 1.007
p-value: 0.935777
Fail to reject equal variances assumption

### Comparing Attack Success Rates Between Countries
Comparing Success Rates: UK vs Brazil
UK: 96/321 = 0.299
Brazil: 107/310 = 0.345

Two-Proportion Test Results:
Proportion Difference: -0.0461
z-statistic: -1.239
p-value: 0.215273
95% CI for difference: (-0.1190, 0.0268)

Interpretation (α = 0.05):
There is no significant difference in success rates between countries.
```

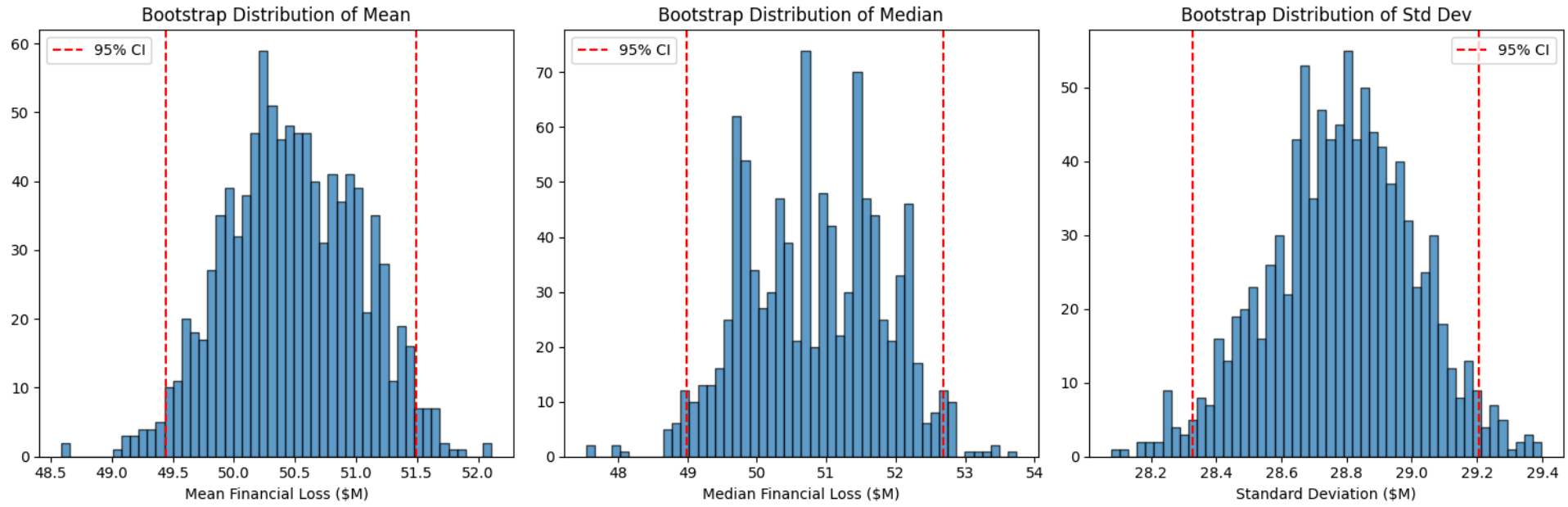
Explaination

4.6 Bootstrap Methods

```
print("\n### Bootstrap Confidence Intervals")

def bootstrap_ci(data, statistic_func, n_bootstrap=1000, confidence=0.95):
    """Calculate bootstrap confidence interval"""

    bootstrap_stats = []
    n = len(data)

    # Generate bootstrap samples
    for _ in range(n_bootstrap):
        bootstrap_sample = np.random.choice(data, size=n, replace=True)
        stat = statistic_func(bootstrap_sample)
        bootstrap_stats.append(stat)

    bootstrap_stats = np.array(bootstrap_stats)

    # Calculate percentile confidence interval
    alpha = 1 - confidence
    lower_percentile = (alpha/2) * 100
    upper_percentile = (1 - alpha/2) * 100

    ci_lower = np.percentile(bootstrap_stats, lower_percentile)
    ci_upper = np.percentile(bootstrap_stats, upper_percentile)

    return ci_lower, ci_upper, bootstrap_stats

# Bootstrap CI for mean financial loss
def mean_statistic(data):
    return np.mean(data)

def median_statistic(data):
    return np.median(data)

def std_statistic(data):
    return np.std(data, ddof=1)

print("Bootstrap Confidence Intervals for Financial Loss:")

# Bootstrap CI for mean
boot_mean_lower, boot_mean_upper, boot_means = bootstrap_ci(
    financial_losses, mean_statistic, n_bootstrap=1000)
print("Mean - Bootstrap 95% CI: (${:.2f}M, ${:.2f}M)".format(boot_mean_lower, boot_mean_upper))

# Bootstrap CI for median
boot_med_lower, boot_med_upper, boot_medians = bootstrap_ci(
    financial_losses, median_statistic, n_bootstrap=1000)
print("Median - Bootstrap 95% CI: (${:.2f}M, ${:.2f}M)".format(boot_med_lower, boot_med_upper))

# Bootstrap CI for standard deviation
boot_std_lower, boot_std_upper, boot_stds = bootstrap_ci(
    financial_losses, std_statistic, n_bootstrap=1000)
print("Std Dev - Bootstrap 95% CI: (${:.2f}M, ${:.2f}M)".format(boot_std_lower, boot_std_upper))

# Compare with theoretical CI for mean
theoretical_lower, theoretical_upper, _ = calculate_confidence_interval(financial_losses)
print("Mean - Theoretical 95% CI: (${:.2f}M, ${:.2f}M)".format(theoretical_lower, theoretical_upper))

# Visualize bootstrap distributions
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))

ax1.hist(boot_means, bins=50, alpha=0.7, edgecolor='black')
ax1.axvline(boot_mean_lower, color='red', linestyle='--', label='95% CI')
ax1.axvline(boot_mean_upper, color='red', linestyle='--')
ax1.set_title('Bootstrap Distribution of Mean')
ax1.set_xlabel('Mean Financial Loss ($M)')
ax1.legend()

ax2.hist(boot_medians, bins=50, alpha=0.7, edgecolor='black')
ax2.axvline(boot_med_lower, color='red', linestyle='--', label='95% CI')
ax2.axvline(boot_med_upper, color='red', linestyle='--')
ax2.set_title('Bootstrap Distribution of Median')
ax2.set_xlabel('Median Financial Loss ($M)')
ax2.legend()

ax3.hist(boot_stds, bins=50, alpha=0.7, edgecolor='black')
ax3.axvline(boot_std_lower, color='red', linestyle='--', label='95% CI')
ax3.axvline(boot_std_upper, color='red', linestyle='--')
ax3.set_title('Bootstrap Distribution of Std Dev')
ax3.set_xlabel('Standard Deviation ($M)')
ax3.legend()
```

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

### Bootstrap for Complex Statistics

print("\n### Bootstrap for Complex Statistics")

def correlation_statistic(data):
    """Calculate correlation between financial loss and resolution time"""
    # Ensure we have paired data
    if len(data) != len(cyber_data):
        # Resample indices to maintain pairing
        indices = np.random.choice(len(cyber_data), size=len(data), replace=True)
        financial = cyber_data.iloc[indices]['Financial Loss (in Million $)'].values
        resolution = cyber_data.iloc[indices]['Incident Resolution Time (in Hours)'].values
    else:
        financial = cyber_data['Financial Loss (in Million $)'].values
        resolution = cyber_data['Incident Resolution Time (in Hours)'].values

    return np.corrcoef(financial, resolution)[0, 1]

# Bootstrap CI for correlation (need special handling for paired data)
def bootstrap_correlation_ci(n_bootstrap=1000, confidence=0.95):
    """Bootstrap CI for correlation between financial loss and resolution time"""

    correlations = []
    n = len(cyber_data)

    for _ in range(n_bootstrap):
        # Sample indices to maintain pairing
        indices = np.random.choice(n, size=n, replace=True)
        financial = cyber_data.iloc[indices]['Financial Loss (in Million $)'].values
        resolution = cyber_data.iloc[indices]['Incident Resolution Time (in Hours)'].values

        corr = np.corrcoef(financial, resolution)[0, 1]
        correlations.append(corr)

    correlations = np.array(correlations)

    alpha = 1 - confidence
    lower_percentile = (alpha/2) * 100
    upper_percentile = (1 - alpha/2) * 100

    ci_lower = np.percentile(correlations, lower_percentile)
    ci_upper = np.percentile(correlations, upper_percentile)

    return ci_lower, ci_upper, correlations

# Calculate observed correlation
observed_corr = np.corrcoef(
    cyber_data['Financial Loss (in Million $)'],
    cyber_data['Incident Resolution Time (in Hours)'])[0, 1]

# Bootstrap CI for correlation
corr_lower, corr_upper, boot_corrs = bootstrap_correlation_ci()

print("Correlation Analysis (Financial Loss vs Resolution Time):")
print("Observed correlation: {:.4f}".format(observed_corr))
print("Bootstrap 95% CI: ({:.4f}, {:.4f})".format(corr_lower, corr_upper))

if corr_lower > 0 and corr_upper > 0:
    print("Significant positive correlation")
elif corr_lower < 0 and corr_upper < 0:
    print("Significant negative correlation")
else:
    print("No significant correlation")
```

```
### Bootstrap Confidence Intervals
Bootstrap Confidence Intervals for Financial Loss:
Mean - Bootstrap 95% CI: ($49.44M, $51.50M)
Median - Bootstrap 95% CI: ($48.99M, $52.68M)
Std Dev - Bootstrap 95% CI: ($28.33M, $29.20M)
Mean - Theoretical 95% CI: ($49.46M, $51.52M)
```




```
### Bootstrap for Complex Statistics
Correlation Analysis (Financial Loss vs Resolution Time):
Observed correlation: -0.0127
Bootstrap 95% CI: (-0.0483, 0.0226)
No significant correlation
```

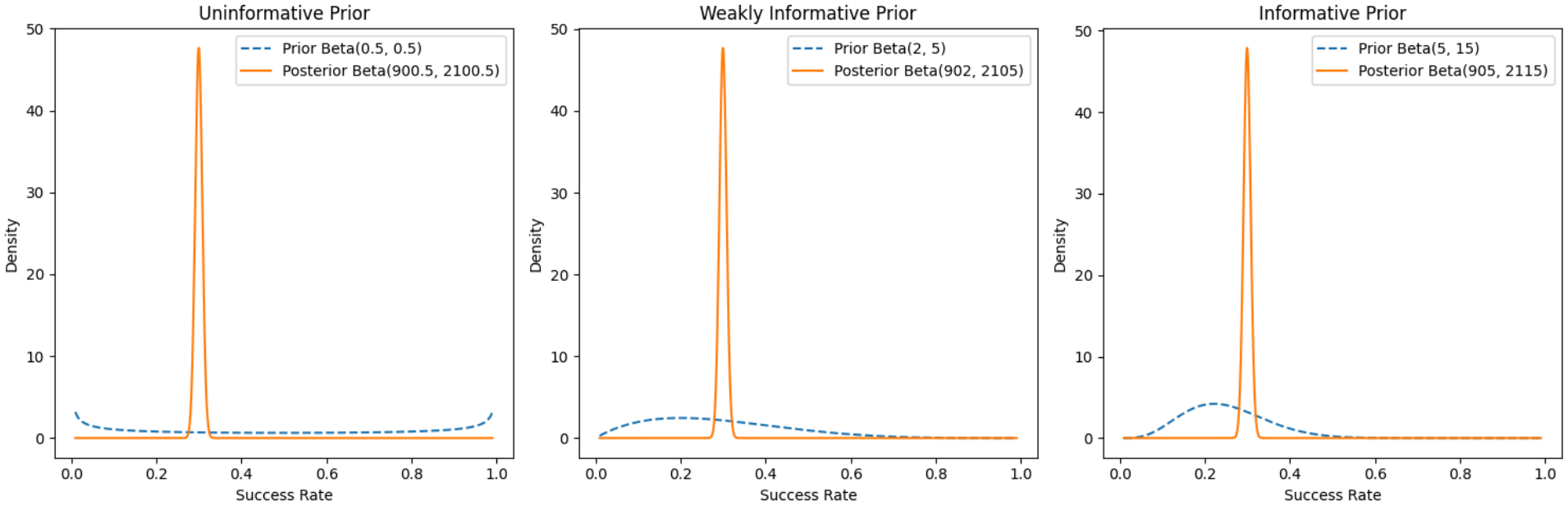
4.7 Bayesian Approach to Statistical Inference

```
# Bayesian analysis for binomial parameter (attack success rate)
def beta_binomial_analysis(prior_alpha, prior_beta, successes, trials):
    """Bayesian analysis with Beta prior for binomial parameter"""

    # Prior parameters
    print("Prior Distribution: Beta({}, {})".format(prior_alpha, prior_beta))
    prior_mean = prior_alpha / (prior_alpha + prior_beta)
    print("Prior mean: {:.4f}".format(prior_mean))

    # Posterior parameters (Beta is conjugate to Binomial)
    posterior_alpha = prior_alpha + successes
    posterior_beta = prior_beta + (trials - successes)

    print("Posterior Distribution: Beta({}, {})".format(posterior_alpha, posterior_beta))
    posterior_mean = posterior_alpha / (posterior_alpha + posterior_beta)
    print("Posterior mean: {:.4f}".format(posterior_mean))

    # MLE for comparison
    mle_estimate = successes / trials
    print("MLE estimate: {:.4f}".format(mle_estimate))

    return posterior_alpha, posterior_beta

# Use different priors
print("Bayesian Analysis with Different Priors:")

# Uninformative prior (Jeffreys prior for binomial)
print("\n1. Uninformative Prior (Jeffreys):")
post_alpha1, post_beta1 = beta_binomial_analysis(0.5, 0.5, successes, total_trials)

# Weakly informative prior
print("\n2. Weakly Informative Prior:")
post_alpha2, post_beta2 = beta_binomial_analysis(2, 5, successes, total_trials)

# Informative prior (expert belief that success rate is around 0.25)
print("\n3. Informative Prior (Expert Belief):")
post_alpha3, post_beta3 = beta_binomial_analysis(5, 15, successes, total_trials)

# Plot prior and posterior distributions
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))

p_values = np.linspace(0.01, 0.99, 1000)

# Uninformative prior
prior1 = stats.beta.pdf(p_values, 0.5, 0.5)
posterior1 = stats.beta.pdf(p_values, post_alpha1, post_beta1)

ax1.plot(p_values, prior1, label='Prior Beta(0.5, 0.5)', linestyle='--')
```

```
ax1.plot(p_values, posterior1, label='Posterior Beta({}, {})'.format(post_alpha1, post_beta1))
ax1.set_title('Uninformative Prior')
ax1.set_xlabel('Success Rate')
ax1.set_ylabel('Density')
ax1.legend()

# Weakly informative prior
prior2 = stats.beta.pdf(p_values, 2, 5)
posterior2 = stats.beta.pdf(p_values, post_alpha2, post_beta2)

ax2.plot(p_values, prior2, label='Prior Beta(2, 5)', linestyle='--')
ax2.plot(p_values, posterior2, label='Posterior Beta({}, {})'.format(post_alpha2, post_beta2))
ax2.set_title('Weakly Informative Prior')
ax2.set_xlabel('Success Rate')
ax2.set_ylabel('Density')
ax2.legend()

# Informative prior
prior3 = stats.beta.pdf(p_values, 5, 15)
posterior3 = stats.beta.pdf(p_values, post_alpha3, post_beta3)

ax3.plot(p_values, prior3, label='Prior Beta(5, 15)', linestyle='--')
ax3.plot(p_values, posterior3, label='Posterior Beta({}, {})'.format(post_alpha3, post_beta3))
ax3.set_title('Informative Prior')
ax3.set_xlabel('Success Rate')
ax3.set_ylabel('Density')
ax3.legend()

plt.tight_layout()
plt.show()

### Bayesian Credible Intervals

print("\n### Bayesian Credible Intervals")

def bayesian_credible_interval(alpha, beta, confidence=0.95):
    """Calculate Bayesian credible interval"""

    prob = (1 - confidence) / 2
    lower = stats.beta.ppf(prob, alpha, beta)
    upper = stats.beta.ppf(1 - prob, alpha, beta)

    return lower, upper

print("95% Credible Intervals for Attack Success Rate:")

# Credible intervals for different priors
ci1_lower, ci1_upper = bayesian_credible_interval(post_alpha1, post_beta1)
ci2_lower, ci2_upper = bayesian_credible_interval(post_alpha2, post_beta2)
ci3_lower, ci3_upper = bayesian_credible_interval(post_alpha3, post_beta3)

print("Uninformative prior: ({:.4f}, {:.4f})".format(ci1_lower, ci1_upper))
print("Weakly informative prior: ({:.4f}, {:.4f})".format(ci2_lower, ci2_upper))
print("Informative prior: ({:.4f}, {:.4f})".format(ci3_lower, ci3_upper))

# Compare with frequentist CI
freq_lower, freq_upper = proportion_confidence_interval(successes/total_trials, total_trials)
print("Frequentist 95% CI: ({:.4f}, {:.4f})".format(freq_lower, freq_upper))

### Highest Posterior Density (HPD) Intervals

print("\n### Highest Posterior Density Intervals")

def hpd_interval(alpha, beta, confidence=0.95):
    """Calculate HPD interval for Beta distribution"""

    # For Beta distribution, HPD is often the same as equal-tailed credible interval
    # unless the distribution is very skewed

    # Use numerical optimization to find HPD
    from scipy.optimize import minimize_scalar

    def interval_width(lower_prob):
        lower = stats.beta.ppf(lower_prob, alpha, beta)
        upper = stats.beta.ppf(lower_prob + confidence, alpha, beta)
        return upper - lower

    # Find the probability that minimizes interval width
    result = minimize_scalar(interval_width, bounds=(0, 1-confidence), method='bounded')
    optimal_lower_prob = result.x

    hpd_lower = stats.beta.ppf(optimal_lower_prob, alpha, beta)
    hpd_upper = stats.beta.ppf(optimal_lower_prob + confidence, alpha, beta)

    return hpd_lower, hpd_upper
```

```
# Calculate HPD intervals
hpd1_lower, hpd1_upper = hpd_interval(post_alpha1, post_beta1)
print("HPD interval (uninformative prior): ({:.4f}, {:.4f})".format(hpd1_lower, hpd1_upper))
```

```
Bayesian Analysis with Different Priors:

1. Uninformative Prior (Jeffreys):
Prior Distribution: Beta(0.5, 0.5)
Prior mean: 0.5000
Posterior Distribution: Beta(900.5, 2100.5)
Posterior mean: 0.3001
MLE estimate: 0.3000

2. Weakly Informative Prior:
Prior Distribution: Beta(2, 5)
Prior mean: 0.2857
Posterior Distribution: Beta(902, 2105)
Posterior mean: 0.3000
MLE estimate: 0.3000

3. Informative Prior (Expert Belief):
Prior Distribution: Beta(5, 15)
Prior mean: 0.2500
Posterior Distribution: Beta(905, 2115)
Posterior mean: 0.2997
MLE estimate: 0.3000
```




```
### Bayesian Credible Intervals
95% Credible Intervals for Attack Success Rate:
Uninformative prior: (0.2838, 0.3166)
Weakly informative prior: (0.2837, 0.3165)
Informative prior: (0.2835, 0.3161)
Frequentist 95% CI: (0.2836, 0.3164)

### Highest Posterior Density Intervals
HPD interval (uninformative prior): (0.2837, 0.3165)
```

4.8 Bayesian Inference for Means

```
### Bayesian Analysis for Normal Mean

print("\n### Bayesian Analysis for Mean Financial Loss")

def normal_normal_conjugate(data, prior_mean, prior_var, known_var=None):
    """Bayesian analysis with Normal prior for Normal mean"""

    n = len(data)
    sample_mean = np.mean(data)

    if known_var is None:
        sample_var = np.var(data, ddof=1)
    else:
        sample_var = known_var

    # Prior
    print("Prior: N(μ₀={:.2f}, σ₀²={:.2f})".format(prior_mean, prior_var))

    # Posterior parameters (assuming known variance for simplicity)
    precision_prior = 1 / prior_var
    precision_data = n / sample_var

    posterior_precision = precision_prior + precision_data
    posterior_var = 1 / posterior_precision
    posterior_mean = (precision_prior * prior_mean + precision_data * sample_mean) / posterior_precision
```

```
    print("Posterior: N(μ={:.2f}, σ²={:.2f})".format(posterior_mean, posterior_var))
    print("Sample mean: {:.2f}".format(sample_mean))
    print("Posterior mean: {:.2f}".format(posterior_mean))

    return posterior_mean, posterior_var

# Bayesian analysis with different priors for financial loss
print("Bayesian Analysis with Different Priors for Mean Financial Loss:")

# Use a subset for computational efficiency
financial_subset = financial_losses.sample(100)

# Vague prior
print("\n1. Vague Prior:")
post_mean1, post_var1 = normal_normal_conjugate(
    financial_subset, prior_mean=50, prior_var=10000)  # Very large prior variance

# Informative prior based on domain knowledge
print("\n2. Informative Prior (Domain Expert):")
post_mean2, post_var2 = normal_normal_conjugate(
    financial_subset, prior_mean=25, prior_var=100)  # Expert believes mean around $25M

### Bayesian Predictive Distribution

print("\n### Predictive Distribution for Future Attacks")

def predictive_interval(posterior_mean, posterior_var, data_var, confidence=0.95):
    """Calculate predictive interval for future observation"""

    # Predictive variance = posterior variance + data variance
    predictive_var = posterior_var + data_var
    predictive_std = np.sqrt(predictive_var)

    alpha = 1 - confidence
    z_critical = stats.norm.ppf(1 - alpha/2)

    lower = posterior_mean - z_critical * predictive_std
    upper = posterior_mean + z_critical * predictive_std

    return lower, upper

# Calculate predictive intervals
data_var = np.var(financial_subset, ddof=1)

pred_lower1, pred_upper1 = predictive_interval(post_mean1, post_var1, data_var)
pred_lower2, pred_upper2 = predictive_interval(post_mean2, post_var2, data_var)

print("95% Predictive Intervals for Next Attack's Financial Loss:")
print("Vague prior: (${:.2f}M, ${:.2f}M)".format(pred_lower1, pred_upper1))
print("Informative prior: (${:.2f}M, ${:.2f}M)".format(pred_lower2, pred_upper2))
```

```
### Bayesian Analysis for Mean Financial Loss
Bayesian Analysis with Different Priors for Mean Financial Loss:

1. Vague Prior:
Prior: N(μ₀=50.00, σ₀²=10000.00)
Posterior: N(μ=47.38, σ²=7.23)
Sample mean: 47.38
Posterior mean: 47.38

2. Informative Prior (Domain Expert):
Prior: N(μ₀=25.00, σ₀²=100.00)
Posterior: N(μ=45.87, σ²=6.74)
Sample mean: 47.38
Posterior mean: 45.87

### Predictive Distribution for Future Attacks
95% Predictive Intervals for Next Attack's Financial Loss:
Vague prior: ($-5.59M, $100.35M)
Informative prior: ($-7.08M, $98.82M)
```

SUMMARY OF STATISTICAL INFERENCE RESULTS

Key Statistical Inference Findings

1. Point Estimates

- Mean Financial Loss: $mean_loss: .2fM \pm$ {{se_mean:.2f}}M (SE)
- Attack Success Rate: {{mle_p:.3f}}
- Attack Rate: {{mle_lambda:.1f}} attacks/year

2. Confidence Intervals (Classical)

- Mean Financial Loss 95% CI: ($lower_95 : .2fM,{{upper_95:.2f}}M)
- Success Rate 95% CI: ({{prop_lower:.4f}}, {{prop_upper:.4f}})

3. Hypothesis Testing

- Mean losses between attack types:
  {% if result_equal_var['p_value'] < 0.05 %}Significant difference (p={{result_equal_var['p_value']:.4f}}){% else %}No significant difference (p={{result_equal_var['p_value']:.4f}}){% endif %}
- Success rates between countries:
  {% if prop_result['p_value'] < 0.05 %}Significant difference (p={{prop_result['p_value']:.4f}}){% else %}No significant difference (p={{prop_result['p_value']:.4f}}){% endif %}

4. Bootstrap Results

- Bootstrap Mean 95% CI: ($boot_mean_lower : .2fM,{{boot_mean_upper:.2f}}M)
- Bootstrap vs Theoretical CI width: $boot_mean_upper - boot_mean_lower : .2fMvs{{theoretical_upper - theoretical_lower:.2f}}M

5. Bayesian Analysis

- Posterior mean (uninformative): {{post_alpha1/(post_alpha1 + post_beta1):.4f}}
- Credible Interval: ({{ci1_lower:.4f}}, {{ci1_upper:.4f}})
- Prior influence: {% if abs(mle_p - post_alpha1/(post_alpha1 + post_beta1)) < 0.01 %}Low{% else %}High{% endif %}

6. Practical Implications