Error Calculations

February 13, 2024

1 Error Calculations for Peruvian Bluberry Data: Price, Volume and Value

1.1 Introduction

We are interested in calculating standard errors for quantities such as price, value and volume. A parametric approach in our case would be asuming that the data for a specific week follows (let's say) a normal distribution. Normal distribution is the most widely used distribution in statistics. However, we must check our data's distribution before we make such a claim.

A widely used test in Statistics to check for normality is called the Shapiro Wilk test, We can carry out this test on some rows of our data to see if pricing for any given week is normally distributed. Below, we carry out the test on row 47 of our data.

[887]:
```
#Import the necessary libraries
import pandas as pd
from scipy.stats import shapiro
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

[888]:
```
file_path = 'Price.xlsx'

# Read the Excel file into a Pandas DataFrame
df = pd.read_excel(file_path)

first_row = df.iloc[47,:] #Select the 47th row

# Perform the Shapiro-Wilk test
statistic, p_value = shapiro(first_row)

# Display the test result
print("Shapiro-Wilk Test Statistic:", statistic)
print("P-value:", p_value)

# Interpret the result
alpha = 0.05
if p_value > alpha:
```

```
    print("The sample looks normally distributed (fail to reject H0)")
else:
    print("The sample does not look normally distributed (reject H0)")
```

```
Shapiro-Wilk Test Statistic: 0.7900190949440002
P-value: 0.0052005513571202755
The sample does not look normally distributed (reject H0)
```

According to our results, the data is not normally distributed. Therefore we follow a non-parametric approach for the calculation of the standard errors. A non parametric approach is one that does not assume an underlying distribution for the given data.

1.2 Non-Parametric Approach - Bootstrapping

Bootstrapping is a resampling technique that can be used to estimate the sampling distribution of a statistic, such as the standard error, even when the underlying data is not normally distributed. This makes bootstrapping a versatile and robust method for estimating parameters.

Bootstrapping is a non-parametric method, meaning it does not rely on assumptions about the underlying distribution of the data. Instead, it directly uses the observed data to estimate the sampling distribution of a statistic. This makes it more robust and flexible, especially when the underlying distribution is unknown or not easily characterized. Bootstrapping involves randomly sampling from the observed data with replacement to create multiple bootstrap samples. This process effectively captures the variability and structure of the original data, allowing for more accurate estimation of parameters and uncertainty measures.

Overall, bootstrapping is a powerful and widely used technique for statistical inference and estimation, providing valuable insights even in cases where the data is not normally distributed.

1.3 Moving Block Bootstrapping for Time Series Data

Moving block bootstrapping is particularly useful for time series data because it takes into account the temporal structure and dependencies present in the data. Time series data often exhibits temporal dependence, where the value of a data point is related to the values of previous data points. Moving block bootstrapping preserves this temporal dependence by sampling contiguous blocks of data, allowing the bootstrap samples to capture the autocorrelation structure of the original time series. It maintains the sequential ordering of observations in the time series. This is crucial for time series analysis, as the order of observations often carries important information about the underlying process.

1.4 Moving Block Bootstrapping With Overlap

In bootstrapping with time series data, using an overlap can be beneficial for several reasons:

Preserving Temporal Dependence: Overlapping blocks allow for the preservation of temporal dependence in the resampled data. By including overlapping segments from adjacent blocks, the resampled data maintains some level of continuity and autocorrelation structure, which is essential for capturing the characteristics of the original time series.

Reduced Variance: Overlapping blocks can help reduce the variance of estimates derived from bootstrapping. By incorporating information from neighboring blocks, the resampled data may exhibit less variability, leading to more stable estimates of parameters and statistics.

1.5 Error Calculations

I have used a modified formula for the calculation of the standard deviation. In place of mean, I have used the target value (which is the last value in the respective row and the value that we are comparing each value in the past to)

$$\text{Standard Deviation } (\sigma) = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \text{target value})^2}{n}}$$

Once we have the standard deviation, we can calculate the standard error for each week by dividing the standard deviation with the square root of the block size.

$$\text{Standard Error } = \left(\frac{\sigma}{\sqrt{n}}\right)$$

1.6 Code Summary

First, we Initialize variables **block_size** and **overlap** to specify the size of blocks and the overlap between consecutive blocks for moving block bootstrapping.

Then we set the number of bootstrap samples to generate (**num_bootstrap_samples**) to 1000. This helps us ensure that our results will be precise. This step could be thought of as replicating an experiment several times to get the best results.

We then define a function **calculate_standard_error** to calculate the standard error for each row using moving block bootstrapping with overlap.

This function iterates over each row of the DataFrame and performs the following steps: 1.Divides the row into blocks and generates bootstrap samples with overlap. 2.Calculates errors by subtracting the value of interest (from the "LastColumn" of the row) from the bootstrap samples. 3.Separates positive and negative errors. 4.Computes standard errors for positive and negative errors using the formula for standard deviation. 5.Returns the positive and negative standard errors as a Pandas Series. 6.Applies the calculate_standard_error function to each row of the DataFrame to compute the standard errors Both, the positive standard errors (positive_std_error) and negative standard errors (negative_std_error) are then printed and then the results are plotted. In the plot, positive standard errors are plotted above the original line while the negative standard errors are plotted below the original line.

1.7 Standard Error Calculations for Price (Using Moving Block Bootstrapping With Overlap)

[901]:

```
file_path = 'Price.xlsx'  # Read the Excel file containing the data

# Read the Excel file into a Pandas DataFrame
df = pd.read_excel(file_path)

# Set option to display all rows of a DataFrame if it is printed
```

```
pd.set_option('display.max_rows', None)

df['LastColumn'] = df.iloc[:, -1]  # Extract the last column of the DataFrame␣
↪as a new column named 'LastColumn'
block_size = 4  # Size of the blocks used in moving block bootstrapping
overlap = 3  # Size of the overlap between consecutive blocks

num_bootstrap_samples = 1000  # Number of bootstrap samples to generate
custom_labels = ['42', '43', '44', '45', '46', '47', '48', '49', '50', '51',␣
↪'52', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13',␣
↪'14', '15', '16', '17', '18', '19', '20', '21', '22', '23', '24', '25',␣
↪'26', '27', '28', '29', '30', '31', '32', '33', '34', '35', '36', '37',␣
↪'38', '39', '40', '41', '42', '43','44','45','46','47','48','49']
```

[902]:
```
# Function to calculate standard error for each row using moving block␣
↪bootstrapping with overlap
def calculate_standard_error(row):
    # Extract the value of interest from the row
    value_of_interest = row['LastColumn']

    # Calculate the number of blocks and initialize array to store bootstrap␣
↪samples
    num_blocks = (len(row[:-1]) - block_size) // overlap + 1
    bootstrap_samples = np.zeros((num_bootstrap_samples, block_size))

    last_index = 0  # Track the last used index
    # Generate bootstrap samples using moving block bootstrapping with overlap
    for j in range(num_bootstrap_samples):
        start_index = last_index
        last_index += overlap  # Increment by the overlap size for the next␣
↪block
        if last_index + block_size > len(row[:-1]):
            last_index = 0  # Wrap around if the next block goes beyond the␣
↪array
        bootstrap_samples[j] = row[start_index:start_index + block_size].values

    # Flatten the bootstrap samples array
    bootstrap_samples = bootstrap_samples.reshape((num_bootstrap_samples, -1))

    # Calculate errors (deviation from value of interest)
    errors = value_of_interest - bootstrap_samples

    # Separate positive and negative errors
    positive_errors = errors[errors >= 0]

    negative_errors = errors[errors < 0]
```

```python
    # Calculate standard error for positive and negative errors
    positive_std_error = np.sqrt((sum((positive_errors)**2) /
 ↪len(positive_errors))) / np.sqrt(block_size) if len(positive_errors) > 0
 ↪else 0
    negative_std_error = np.sqrt((sum((negative_errors)**2) /
 ↪len(negative_errors))) / np.sqrt(block_size) if len(negative_errors) > 0
 ↪else 0

    # Return standard errors as a Pandas Series
    return pd.Series({'Positive_Standard_Error': positive_std_error,
 ↪'Negative_Standard_Error': negative_std_error})

# Calculate standard error for each row using moving block bootstrapping with
 ↪overlap
standard_errors = df.apply(calculate_standard_error, axis=1)

# Create DataFrame for standard errors
df_se = pd.DataFrame(standard_errors)

# Create DataFrame for custom labels
df_cl = pd.DataFrame(custom_labels, columns=['Week'])

# Merge custom labels DataFrame with standard errors DataFrame
merged_df = pd.merge(df_cl, df_se, left_index=True, right_index=True)
merged_df.index.name = None  # Remove index name
(merged_df.head(62))
```

[902]:

	Week	Positive_Standard_Error	Negative_Standard_Error
0	42	0.000000	0.000000
1	43	0.000000	0.000000
2	44	0.000000	0.000000
3	45	0.000000	0.000000
4	46	0.000000	0.000000
5	47	0.000000	0.000000
6	48	0.000000	0.000000
7	49	0.000000	0.000000
8	50	0.000000	0.000000
9	51	0.000000	0.000000
10	52	0.039191	0.000000
11	1	0.000000	0.000000
12	2	0.000000	0.000000
13	3	0.000000	0.000000
14	4	0.000000	0.000000
15	5	0.000000	0.000000
16	6	0.000000	0.000000
17	7	0.000000	0.000000
18	8	0.000000	0.000000

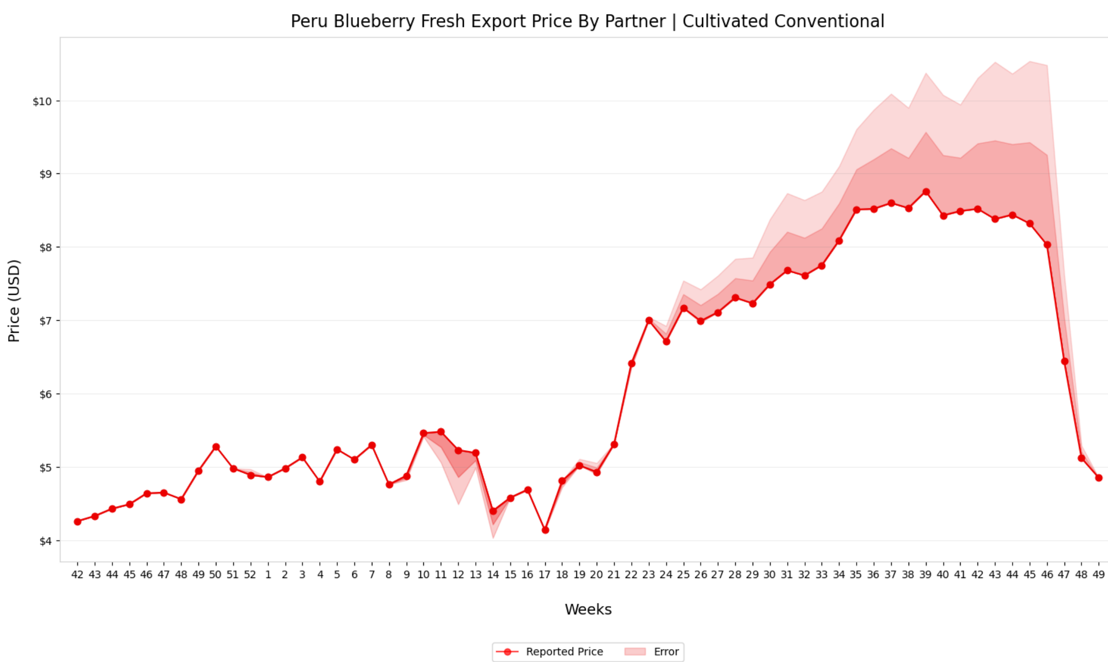
```
19    9              0.000000            0.030000
20   10              0.000000            0.023884
21   11              0.000000            0.210000
22   12              0.000000            0.370000
23   13              0.000000            0.100000
24   14              0.000000            0.185000
25   15              0.000000            0.000000
26   16              0.000000            0.005000
27   17              0.000000            0.005000
28   18              0.000000            0.045000
29   19              0.043714            0.000000
30   20              0.060947            0.015000
31   21              0.006250            0.000000
32   22              0.017912            0.030000
33   23              0.022256            0.000000
34   24              0.105933            0.000000
35   25              0.185018            0.005000
36   26              0.215433            0.010000
37   27              0.249321            0.000000
38   28              0.263391            0.005000
39   29              0.311839            0.000000
40   30              0.442761            0.000000
41   31              0.526259            0.000000
42   32              0.514027            0.000000
43   33              0.502078            0.000000
44   34              0.505547            0.000000
45   35              0.547613            0.000000
46   36              0.675783            0.000000
47   37              0.744653            0.000000
48   38              0.683418            0.000000
49   39              0.807181            0.000000
50   40              0.821675            0.000000
51   41              0.725644            0.000000
52   42              0.891458            0.000000
53   43              1.071726            0.000000
54   44              0.961428            0.000000
55   45              1.106961            0.000000
56   46              1.224763            0.000000
57   47              0.590099            0.000000
58   48              0.084853            0.000000
59   49              0.000000            0.000000
```

[903]:

```python
from matplotlib.lines import Line2D
# Scatterplot with area chart and markers
fig, ax = plt.subplots(figsize=(18, 9))

# Plot the original line
```

```python
ax.plot(range(len(df)), df['LastColumn'], color='#EA0000')

# Separate positive and negative standard errors
positive_errors = standard_errors['Positive_Standard_Error']
negative_errors = standard_errors['Negative_Standard_Error']

# Fill the area above the curve for positive errors
if not positive_errors.empty:
    fill=ax.fill_between(range(len(df)), df['LastColumn'], df['LastColumn'] +␣
↪positive_errors, color='#EA0000', alpha=0.2, label='Positive Errors')
    fill=ax.fill_between(range(len(df)), df['LastColumn'], df['LastColumn'] +␣
↪2*positive_errors, color='#EA0000', alpha=0.15, label='Positive Errors')
# Fill the area below the curve for negative errors
if not negative_errors.empty:
    fill=ax.fill_between(range(len(df)), df['LastColumn'] - negative_errors.
↪abs(), df['LastColumn'], color='#EA0000', alpha=0.3, label='Positive Errors')
    fill=ax.fill_between(range(len(df)), df['LastColumn'] - 2*negative_errors.
↪abs(), df['LastColumn'], color='#EA0000', alpha=0.2, label='Positive Errors')
# Scatterplot with opaque circular markers
ax.scatter(range(len(df)), df['LastColumn'], color='#EA0000', s=38)

# Customize the plot
ax.set_xlabel('Weeks', fontsize=14, labelpad=22)  # Set x-axis label and adjust␣
↪padding
ax.set_ylabel('Price (USD)', fontsize=14, labelpad=10)  # Set y-axis label and␣
↪adjust padding
ax.yaxis.set_major_formatter('${:,.0f}'.format)  # Add a dollar sign to y-axis␣
↪ticks

# Customize x-axis ticks
plt.xticks(range(len(custom_labels)), custom_labels)

# Set plot title
ax.set_title('Peru Blueberry Fresh Export Price By Partner | Cultivated␣
↪Conventional', fontsize=16, pad=10)

# Add gridlines
ax.grid(axis='y', color='grey', linestyle='-', linewidth=0.5, alpha=0.2)

# Set x-axis limit
ax.set_xlim(-1, len(df))

# Add legend
circle_line = Line2D([0], [0], color='red', marker='o',  markersize=6,␣
↪markerfacecolor='red', alpha=0.7)
legend_handles = [  circle_line,fill]
```

```
legend_labels = [ 'Reported Price','Error']

ax.legend(handles=legend_handles, labels=legend_labels, loc='lower center',
 ↪bbox_to_anchor=(0.5, -0.2), ncol=3)

# Customize spine color
for spine in plt.gca().spines.values():
    spine.set_edgecolor('#d3d3d3')

# Save the figure
fig.savefig('Price_errors.png')

# Show the plot
plt.show()
```



1.8 Standard Error Calculations for Value (Using Moving Block Bootstrapping With Overlap)

[910]:
```
file_path = 'Value.xlsx'   # Read the excel file


# Read the Excel file into a Pandas DataFrame
df = pd.read_excel(file_path)   # Read data into DataFrame
```

```
# Extract the last column as the column of interest
df['LastColumn'] = df.iloc[:, -1]

# Set parameters for moving block bootstrapping
block_size = 4
overlap = 3

# Set the number of bootstrap samples
num_bootstrap_samples = 4

# Define custom labels for the weeks
custom_labels = ['42', '43', '44', '45', '46', '47', '48', '49', '50', '51',␣
 ↪'52', '1', '2', '3', '4', '5', '6', '7',
                 '8', '9', '10', '11', '12', '13', '14', '15', '16', '17',␣
 ↪'18', '19', '20', '21', '22', '23', '24',
                 '25', '26', '27', '28', '29', '30', '31', '32', '33', '34',␣
 ↪'35', '36', '37', '38', '39', '40', '41',
                 '42', '43', '44', '45', '46', '47', '48', '49']
```

[911]:
```
# Function to calculate standard error for each row using moving block␣
 ↪bootstrapping with overlap
def calculate_standard_error(row):
    value_of_interest = row['LastColumn']

    # Perform moving block bootstrapping with overlap
    num_blocks = (len(row[:-1]) - block_size) // overlap + 1
    bootstrap_samples = np.zeros((num_bootstrap_samples, block_size))

    last_index = 0  # Track the last used index
    for j in range(num_bootstrap_samples):
        start_index = last_index
        last_index += overlap  # Increment by the overlap size for the next␣
 ↪block
        if last_index + block_size > len(row[:-1]):
            last_index = 0  # Wrap around if the next block goes beyond the␣
 ↪array
        bootstrap_samples[j] = row[start_index:start_index + block_size].values

    # Flatten the bootstrap samples array
    bootstrap_samples = bootstrap_samples.reshape((num_bootstrap_samples, -1))

    # Calculate standard error
    errors = value_of_interest-bootstrap_samples

    positive_errors = errors[errors >= 0]  # Filter positive errors
    print(positive_errors)
```

```
    negative_errors = errors[errors < 0]     # Filter negative errors

    positive_std_error = np.sqrt((sum((positive_errors)**2)/
 ↪len(positive_errors))) / np.sqrt(block_size) if len(positive_errors) > 0 ␣
 ↪else 0

    negative_std_error = np.sqrt((sum((negative_errors)**2)/
 ↪len(negative_errors))) / np.sqrt(block_size) if len(negative_errors) > 0␣
 ↪else 0

    return pd.Series({'Positive_Standard_Error': positive_std_error,␣
 ↪'Negative_Standard_Error': negative_std_error})

# Calculate standard error for each row using moving block bootstrapping with␣
 ↪overlap
standard_errors = df.apply(calculate_standard_error, axis=1)


# Create DataFrame for standard errors
df_se = pd.DataFrame(standard_errors)

# Create DataFrame for custom labels with column name 'Week'
df_cl = pd.DataFrame(custom_labels, columns=['Week'])

# Print a message indicating the purpose of the displayed results
print("Standard Error for each row using moving block bootstrapping with␣
 ↪overlap:\n")

# Merge the custom labels DataFrame and standard errors DataFrame on their␣
 ↪indices
merged_df = pd.merge(df_cl, df_se, left_index=True, right_index=True)

# Remove the index name
merged_df.index.name = None

# Display the merged DataFrame
(merged_df.head(62))
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[5497187.24 5497187.24 5497187.24 5497187.24 5497187.24 5497187.24
 5497187.24 5497187.24 5497187.24 5497187.24 5497187.24 4122890.43
 4122890.43 2748593.62 1374296.81       0.  ]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[2452.99 2452.99 2452.99 2452.99 2452.99 2452.99    0.      0.      0.
    0.      0.      0.      0.      0.      0.      0.  ]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0.]
[373.38   0.     0.     0.     0.     0.     0.     0.     0.     0.
   0.     0.     0.     0.  ]
[0. 0. 0. 0. 0.]
[58576.56 58576.56 58576.56 58576.56 58576.56 26598.88 26598.88     0.
     0.       0.       0.       0.       0.       0.       0.       0.  ]
[60436.47 64962.74 64962.74 64962.74 64962.74 15451.62 15451.62     0.
     0.       0.       0.       0.  ]
[15666.87     0.       0.       0.       0.       0.       0.       0.
     0.       0.       0.       0.       0.       0.  ]
[46391.03 14563.34 14563.34 14563.34 14563.34 14563.34 14563.34     0.
     0.       0.       0.       0.  ]
[26114.08 18024.83 18024.83 18024.83 18024.83 18024.83 11968.33 27953.88
 27953.88 22579.54     0.       0.  ]
[219360.11 224756.87 213561.77 213561.77 213561.77 213561.77 207558.85
  65226.93  65226.93  52434.35      0.        0.        0.        0.
      0.  ]
[426599.87 473376.01 469924.01 378519.28 378519.28 378519.28 193973.86
 217080.47 217080.47 200298.29      0.        0.        0.        0.
      0.  ]
[766146.19 689006.82 624798.24 402125.01 402125.01 402125.01 305798.82
 227996.22 227996.22  89814.63      0.        0.        0.  ]
[1373709.37 1289505.83 1096950.03  784130.39  784130.39  784130.39
  523791.56  336219.59  336219.59  326089.62       0.         0.
       0.         0.         0.         0.  ]
[1586212.46 1478694.11 1254114.03  817985.22  817985.22  817985.22
  367093.79  245459.83  245459.83  211852.35       0.         0.  ]
[2632202.55 2486193.84 1929341.01 1187741.67 1187741.67 1190908.37
```

```
  591965.32  244027.56  244027.56  192774.42   85177.     87429.51
   87429.51   27599.81       0.         0.  ]
 [4030457.07 3997503.02 3945873.85 2533620.35 2533620.35 2591394.24
 1310564.59  929059.82  929059.82  622840.63  117037.64   32456.38
   32456.38       0.         0.  ]
 [7082954.33 7053884.69 6827801.44 5319060.06 5319060.06 5065653.75
 2639363.74 1871539.62 1871539.62 1217803.46  224788.94  137883.48
  137883.48  143733.82       0.         0.  ]
 [7598351.32 7556358.35 7352144.75 6605770.17 6605770.17 6178594.43
 3454567.75 2354127.1  2354127.1  1339455.24  417010.4   292184.6
  292184.6   399568.12   72092.02       0.  ]
 [8106309.97       8056796.77       8056796.77       6916182.29
 6916182.29       6488801.68       4723055.27       3329025.18
 3329025.18       2166394.77       1114379.73999999  452760.13
  452760.13        246845.72         24197.66             0.        ]
 [8281261.44000001 8138180.44000001 8096948.54       7639435.65000001
 7639435.65000001 7601558.27       5798152.88       4661396.28
 4661396.28       3038444.29       1435920.54       1116527.13
 1116527.13        360987.72         56500.17             0.        ]
 [11186610.24  9026435.86  8269298.15  8256579.78  8256579.78  7940027.74
  7138836.93  6460720.55  6460720.55  5665002.19  3139010.23  2227878.01
  2227878.01  1131275.99    97699.68        0.  ]
 [24618072.62       13525203.75        9286553.25        8763806.97000001
  8763806.97000001  8608117.72000001  8249646.68        7847556.23
  7847556.23        6715231.12        4118270.42        3634993.46
  3634993.46        2191123.42         396543.07              0.        ]
 [27001926.93       24171004.88999999 10660220.79        7615169.98999999
  7615169.98999999  7401178.05999999  7338372.87        7232591.83
  7232591.83        7077326.08        4773665.2         3743284.88999999
  3743284.88999999  2298096.45         581246.38              0.        ]
 [33518404.25       25489967.33       12582662.         12582662.
  6527446.08        5789874.08        5982052.15000001  5982052.15000001
  5855774.28999999  5051887.86        4712073.81999999  4712073.81999999
  3794048.06        1021776.86              0.        ]
 [37449550.53       28786377.74       28786377.74        8738793.61
  5040047.91        4395674.17        4395674.17        4485559.91
  3979760.86        3693384.52        3693384.52        2795843.2
  1421619.65000001        0.        ]
 [33702610.49       33702610.49       27892974.46000001  9909926.42
  2179863.08        2179863.08        1681472.69000001  1633667.82000001
  1635093.84        1635093.84        1092170.16000001   838551.28
        0.        ]
 [39644498.30999999 31708516.68999999  5172073.83        5172073.83
  3628878.22999999   803541.31999999   278349.78999999   278349.78999999
   285627.72         272883.38999999        0.        ]
 [39679021.13       29397145.28999999 29397145.28999999 16630458.69
  2781951.52         759219.33         759219.33         268984.63
   223209.44              0.        ]
```

```
[41730237.74999999 41730237.74999999 38784496.41999999 13833902.25
  3238000.27999999  3238000.27999999   775720.97999999   101935.64
        0.        ]
[39094963.93       34001902.03       13812401.86       13812401.86
  3066980.81         674741.84999999        0.        ]
[32449827.16       28698163.42999999 28698163.42999999 12377000.88
  2916196.88              0.        ]
[28914909.36 28914909.36 24070520.62  8562518.4         0.  ]
[15517731.52 10858795.75        0.  ]
[2044647.90000001       0.        ]
[0.]
Standard Error for each row using moving block bootstrapping with overlap:
```

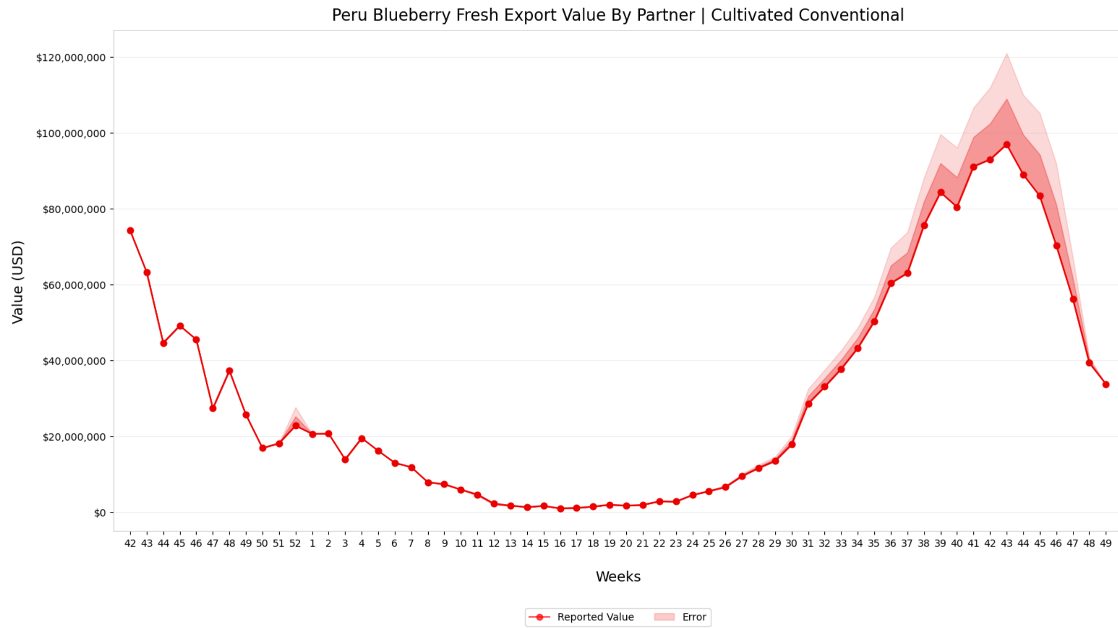
[911]:

	Week	Positive_Standard_Error	Negative_Standard_Error
0	42	0.000000e+00	0.000000
1	43	0.000000e+00	0.000000
2	44	0.000000e+00	0.000000
3	45	0.000000e+00	0.000000
4	46	0.000000e+00	0.000000
5	47	0.000000e+00	0.000000
6	48	0.000000e+00	0.000000
7	49	0.000000e+00	0.000000
8	50	0.000000e+00	0.000000
9	51	0.000000e+00	0.000000
10	52	2.423355e+06	0.000000
11	1	0.000000e+00	0.000000
12	2	0.000000e+00	0.000000
13	3	0.000000e+00	0.000000
14	4	7.510717e+02	0.000000
15	5	0.000000e+00	0.000000
16	6	0.000000e+00	0.000000
17	7	0.000000e+00	0.000000
18	8	0.000000e+00	0.000000
19	9	0.000000e+00	46990.475000
20	10	0.000000e+00	24831.012831
21	11	0.000000e+00	178421.605000
22	12	0.000000e+00	161095.430000
23	13	0.000000e+00	34555.325000
24	14	0.000000e+00	57149.505000
25	15	0.000000e+00	0.000000
26	16	0.000000e+00	1020.000000
27	17	4.989500e+01	813.540000
28	18	0.000000e+00	14095.805000
29	19	1.703446e+04	0.000000
30	20	2.092183e+04	5745.640000
31	21	2.093574e+03	828.480000

32	22	8.446736e+03	12729.450000
33	23	9.706455e+03	49.950000
34	24	7.476721e+04	38.350000
35	25	1.431040e+05	4068.430000
36	26	2.029940e+05	12906.624589
37	27	3.355165e+05	0.000000
38	28	4.228585e+05	4216.059066
39	29	5.809706e+05	0.000000
40	30	1.089311e+06	2330.155000
41	31	1.953978e+06	0.000000
42	32	2.234867e+06	0.000000
43	33	2.450085e+06	0.000000
44	34	2.698009e+06	0.000000
45	35	3.216148e+06	0.000000
46	36	4.656301e+06	0.000000
47	37	5.394924e+06	0.000000
48	38	6.267943e+06	0.000000
49	39	7.645343e+06	0.000000
50	40	7.808033e+06	0.000000
51	41	7.752921e+06	0.000000
52	42	9.471530e+06	0.000000
53	43	1.201853e+07	0.000000
54	44	1.048123e+07	0.000000
55	45	1.091989e+07	0.000000
56	46	1.078158e+07	0.000000
57	47	5.467430e+06	0.000000
58	48	7.228922e+05	0.000000
59	49	0.000000e+00	0.000000

[912]:
```
# Scatterplot with area chart and markers
fig, ax = plt.subplots(figsize=(18, 9))

# Plot the original line
ax.plot(range(len(df)), df['LastColumn'], color='#EA0000')

# Separate positive and negative standard errors
positive_errors = standard_errors['Positive_Standard_Error']
negative_errors = standard_errors['Negative_Standard_Error']

# Fill the area above the curve for positive errors
if not positive_errors.empty:
    fill=ax.fill_between(range(len(df)), df['LastColumn'], df['LastColumn'] +
 ↪positive_errors, color='#EA0000', alpha=0.3)
    fill=ax.fill_between(range(len(df)), df['LastColumn'], df['LastColumn'] +
 ↪2*positive_errors, color='#EA0000', alpha=0.15, label='Positive Errors')
# Fill the area below the curve for negative errors
if not negative_errors.empty:
```

```python
    fill=ax.fill_between(range(len(df)), df['LastColumn'] - negative_errors.
 ↪abs(), df['LastColumn'], color='red', alpha=0.3)
    fill=ax.fill_between(range(len(df)), df['LastColumn'] - 2*negative_errors.
 ↪abs(), df['LastColumn'], color='#EA0000', alpha=0.2, label='Positive Errors')
# Scatterplot with opaque circular markers
ax.scatter(range(len(df)), df['LastColumn'], color='#EA0000', s=38)

# Customize the plot
ax.set_xlabel('Weeks', fontsize=14, labelpad=22)
ax.set_ylabel('Value (USD)', fontsize=14, labelpad=10)
ax.yaxis.set_major_formatter('${:,.0f}'.format)  # Add a dollar sign to y-axis␣
 ↪ticks

# Starting from 47 and ending at 49
plt.xticks(range(len(custom_labels)), custom_labels)

ax.set_title('Peru Blueberry Fresh Export Value By Partner | Cultivated␣
 ↪Conventional', fontsize=16, pad=10)
ax.grid(axis='y', color='grey', linestyle='-', linewidth=0.5, alpha=0.2)
ax.set_xlim(-1, len(df))

# Add legend
circle_line = Line2D([0], [0], color='red', marker='o',  markersize=6,␣
 ↪markerfacecolor='red', alpha=0.7)
legend_handles = [  circle_line,fill]
legend_labels = [ 'Reported Value','Error']

ax.legend(handles=legend_handles, labels=legend_labels, loc='lower center',␣
 ↪bbox_to_anchor=(0.5, -0.2), ncol=3)

for spine in plt.gca().spines.values(): #Adjust the color for spines
    spine.set_edgecolor('#d3d3d3')

# Save the figure
fig.savefig('Value_errors1.png')

# Show the plot
plt.show()
```



1.9 Standard Error Calculations for Volume (Using Moving Block Bootstrapping With Overlap)

[895]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Replace 'your_file_path.xlsx' with the actual path to your Excel file
file_path = 'Volume.xlsx'


# Read the Excel file into a Pandas DataFrame
df = pd.read_excel(file_path)


df['LastColumn'] = df.iloc[:, -1]
```

[896]:
```
# Function to calculate standard error for each row using moving block␣
↪bootstrapping with overlap
def calculate_standard_error(row):
    value_of_interest = row['LastColumn']

    # Perform moving block bootstrapping with overlap
    num_blocks = (len(row[:-1]) - block_size) // overlap + 1
    bootstrap_samples = np.zeros((num_bootstrap_samples, block_size))
```

```
    last_index = 0  # Track the last used index
    for j in range(num_bootstrap_samples):
        start_index = last_index
        last_index += overlap  # Increment by the overlap size for the next␣
 ↪block
        if last_index + block_size > len(row[:-1]):
            last_index = 0  # Wrap around if the next block goes beyond the␣
 ↪array
        bootstrap_samples[j] = row[start_index:start_index + block_size].values

    # Flatten the bootstrap samples array
    bootstrap_samples = bootstrap_samples.reshape((num_bootstrap_samples, -1))

    # Calculate standard error
    errors = value_of_interest-bootstrap_samples


    positive_errors = errors[errors >= 0]  # Filter positive errors

    negative_errors = errors[errors < 0]    # Filter negative errors

    #Standard error formula for Positive errors
    positive_std_error = np.sqrt((sum((positive_errors)**2)/
 ↪len(positive_errors))) / np.sqrt(block_size) if len(positive_errors) > 0 ␣
 ↪else 0

    #Standard error formula for Negative errors
    negative_std_error = np.sqrt((sum((negative_errors)**2)/
 ↪len(negative_errors))) / np.sqrt(block_size) if len(negative_errors) > 0␣
 ↪else 0

    #Return positive and negative standard errors when the␣
 ↪calculate_standard_error function is called
    return pd.Series({'Positive_Standard_Error': positive_std_error,␣
 ↪'Negative_Standard_Error': negative_std_error})

# Call the calculate_standard_error function
standard_errors = df.apply(calculate_standard_error, axis=1)

df_se = pd.DataFrame(standard_errors)
df_cl = pd.DataFrame(custom_labels,columns=['Week'])

# Display the results
print("Standard Error for each row using moving block bootstrapping with␣
 ↪overlap:\n")
```

```
merged_df = pd.merge(df_cl, df_se, left_index=True, right_index=True)
merged_df.index.name = None

# Display the merged DataFrame

(merged_df.head(62))
```

Standard Error for each row using moving block bootstrapping with overlap:

[896]:

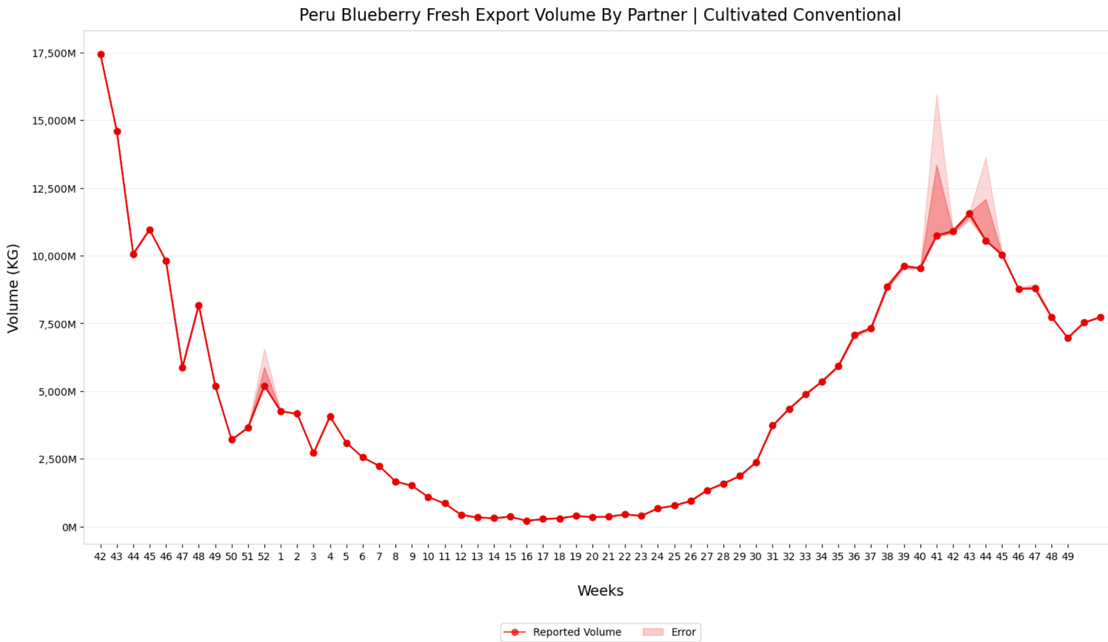
	Week	Positive_Standard_Error	Negative_Standard_Error
0	42	0.000000	0.000000
1	43	0.000000	0.000000
2	44	0.000000	0.000000
3	45	0.000000	0.000000
4	46	0.000000	0.000000
5	47	0.000000	0.000000
6	48	0.000000	0.000000
7	49	0.000000	0.000000
8	50	0.000000	0.000000
9	51	0.000000	0.000000
10	52	685.292315	0.000000
11	1	0.000000	0.000000
12	2	0.000000	0.000000
13	3	0.000000	0.000000
14	4	0.000000	0.000000
15	5	0.000000	0.000000
16	6	0.000000	0.000000
17	7	0.000000	0.000000
18	8	0.000000	0.000000
19	9	0.000000	0.000000
20	10	0.000000	0.000000
21	11	0.000000	0.000000
22	12	0.000000	0.000000
23	13	0.000000	0.000000
24	14	0.000000	0.000000
25	15	0.000000	0.000000
26	16	0.000000	0.000000
27	17	0.000000	0.000000
28	18	0.000000	0.000000
29	19	0.000000	0.000000
30	20	0.000000	0.000000
31	21	0.000000	0.000000
32	22	0.000000	0.000000
33	23	0.000000	0.000000
34	24	0.000000	0.000000

```
35    25             0.000000       0.000000
36    26             0.000000       0.000000
37    27             0.000000       0.000000
38    28             0.000000       0.000005
39    29             0.000000       0.000000
40    30             0.000000       0.000000
41    31             0.000000       0.000000
42    32             0.000000       0.000000
43    33             0.000001       0.000000
44    34             0.983544       2.889250
45    35             0.991947      14.480490
46    36             0.000000      43.624018
47    37             0.000000      24.329138
48    38            71.955293      40.584185
49    39             3.250944      52.200923
50    40             1.338108      14.166414
51    41          2616.676053       8.507557
52    42             0.727256      51.675162
53    43            10.863382     123.822454
54    44          1531.564448       0.000000
55    45            23.955829      38.939642
56    46            18.067917       0.000000
57    47            80.419596       0.000000
58    48            28.767713       0.000000
59    49             0.000000       5.664740
```

[897]:

```
# Scatterplot with area chart and markers
fig, ax = plt.subplots(figsize=(18, 9))

# Plot the original line
ax.plot(range(len(df)), df['LastColumn'], color='#EA0000')

# Separate positive and negative standard errors
positive_errors = standard_errors['Positive_Standard_Error']
negative_errors = standard_errors['Negative_Standard_Error']

# Fill the area above the curve for positive errors
if not positive_errors.empty:
    fill=ax.fill_between(range(len(df)), df['LastColumn'], df['LastColumn'] +␣
 ↪positive_errors, color='#EA0000', alpha=0.3)
    fill=ax.fill_between(range(len(df)), df['LastColumn'], df['LastColumn'] +␣
 ↪2*positive_errors, color='#EA0000', alpha=0.15, label='Positive Errors')
# Fill the area below the curve for negative errors
if not negative_errors.empty:
    fill=ax.fill_between(range(len(df)), df['LastColumn'] - negative_errors.
 ↪abs(), df['LastColumn'], color='#EA0000', alpha=0.3)
```

```python
    fill=ax.fill_between(range(len(df)), df['LastColumn'] - 2*negative_errors.
 ↪abs(), df['LastColumn'], color='#EA0000', alpha=0.2, label='Positive Errors')
# Scatterplot with opaque circular markers
ax.scatter(range(len(df)), df['LastColumn'], color='#EA0000', s=38)

# Customize the plot
ax.set_xlabel('Weeks', fontsize=14, labelpad=22)
ax.set_ylabel('Volume (KG)', fontsize=14, labelpad=10)
ax.yaxis.set_major_formatter('{:,.0f}M'.format)  # Add a dollar sign to y-axis␣
 ↪ticks

# Add x axis ticks, starting from 47 and ending at 49
plt.xticks(range(len(custom_labels)), custom_labels)

ax.set_title('Peru Blueberry Fresh Export Volume By Partner | Cultivated␣
 ↪Conventional', fontsize=16, pad=10)
ax.grid(axis='y', color='grey', linestyle='-', linewidth=0.5, alpha=0.2)
ax.set_xlim(-1, len(df))

# Add legend
circle_line = Line2D([0], [0], color='red', marker='o',  markersize=6,␣
 ↪markerfacecolor='red', alpha=0.7)
legend_handles = [  circle_line,fill]
legend_labels = [ 'Reported Volume','Error']

#Adjust the location of the legend
ax.legend(handles=legend_handles, labels=legend_labels, loc='lower center',␣
 ↪bbox_to_anchor=(0.5, -0.2), ncol=3)


for spine in plt.gca().spines.values(): #Set the color for spines
    spine.set_edgecolor('#d3d3d3')

# Save the figure
fig.savefig('Volume_errors.png')

# Show the plot
plt.show()
```


Peru Blueberry Fresh Export Volume By Partner | Cultivated Conventional
Volume (KG)
17,500M
15,000M
12,500M
10,000M
7,500M
5,000M
2,500M
0M
42 43 44 45 46 47 48 49 50 51 52 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
Weeks
Reported Volume
Error