CS 313E Term Project: Bootstrapping

Madeline Boss (mrb5727)
Tariq Ali (taa2536)

1. Our project goal will be to create a sampling distribution of bootstrapped samples from a defined sample given by a user. Bootstrapping refers to the statistical process of creating synthetic samples from an original sample with replacement, which is then used to conduct statistical tests. If needed, more information can be found about bootstrapping here. This project will create n amount of samples that the user requests, and then be able to pull information about descriptive statistics surrounding the sampling distribution (mean, median, range, standard error), along with a confidence interval (of the user's choosing), and the ability to test another sample against the confidence interval (testing statistical significance).

2. No datasets were used.

3. Description of the Project Modules

I. Function bootstrap
   - A. Takes a list of float values known as *base_sample*. Using those float values it creates a new synthetic sample through the process of bootstrapping, which creates a new list from values from *base_sample* with replacement to match the length of *base_sample*. It then calculates the mean of those samples, which is then rounded to two decimal values.

II. Class Node
   - A. Function __init__
     1. Initialize nodes. Each node has a data value known as *mean*, and two children known as *lchild* and *rchild.*

III. Class BootstrapBST
   - A. Function __init__
     1. Initialize Binary Search Tree (BST). Saves a user-given list of float values known as *self.base*. Also, it initializes a Node value known as *self.root.*
   - A. Function add_tree_value
     1. Adds a value to the tree. It makes a new node from the bootstrap function. Then it checks if the root is empty. If it's not then it loops until it reaches None. Then it checks whether the new node is less or greater than the parent and makes the node the left or right child.
     2. The logic follows the inset search algorithm
   - B. Function median
     1. Returns the median of the BST. It stores the sorted list in *sorted_list* and stores *self.length()* in *size*. Then it checks if the length is 0. If it is then it

returns None. Then it checks if the size is even or not and then finds the median.

C. Function minimum
   1. Finds the minimum of the BST. It loops until the current node is none while checking the left child of the current node.

D. Function maximum
   1. Finds the maximum of the BST. It loops until the current node is none while checking the right child of the current node.

E. Function range
   1. Returns the range of the BST. It stores the mean of the minimum node and maximum node in *min_bst* and *max_bst* respectively. Then it subracts them and returns the range.

F. Function mean
   1. Returns the mean of the BST. It returns None if *self.length()* is 0. Otherwise, it returns the sum of *self.sorted_tree_mean()* and divides it by *self.length()*.

G. Function sorted_tree_mean
   1. Returns the BST sorted. It makes and empty list, returns the empty list if the tree is empty, and calls *self.in_order_traversal()* and returns the sorted tree.

H. Function in_order_traversal
   1. Helper function for sorted_tree_mean. It has a node and a tree for parameters. It calls itself for the left and right children for the nodes and sorts them in numerical order.

I. Function length
   1. Returns the length of the sorted tree.

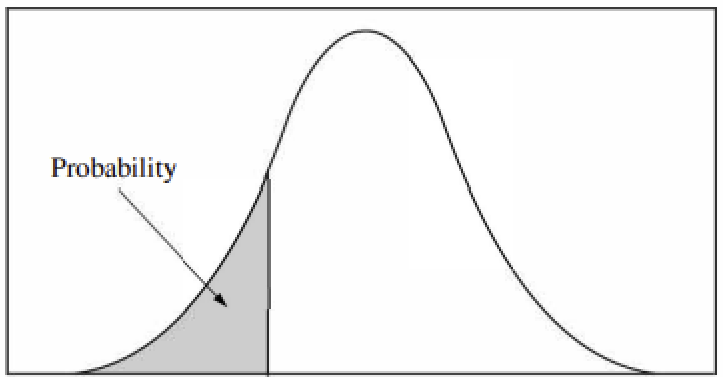
J. Function SE
   1. This function finds the standard error (SE) of the BST. To do this it calls function *self.sorted_tree_mean()*, function *self.length()*, and function *self.mean()* and stores them as a list, integer, and float respectively. It then loops through the *self.sorted_tree_mean()* and finds the square difference of each value minus *mean* and stores the total in *sum_of_sd*. After the loop, it divides the sum of squares by the *length* and then square root *sum_of_sd*. It then returns *sum_of_sd*. If for some reason *self.length()* is zero, the function returns none.
   2. The particular logic of finding standard error is from the Central Limit Theorem. It essentially states if a sampling distribution is normal and you have all the data points, a standard error can be found by using the standard deviation equation $\sigma = \sqrt{\frac{\sum(x_i - \mu)^2}{N}}$.

K. Function ci

1. This function finds the confidence interval (ci) of the BST. It needs one input known as *ci_percentage*, which defaults as a float value of 0.95. It also calls two functions *self.mean()* and *self.se()* and stores them as floats. The function will first check if *ci_percentage* divided by 100 is in the directory *z_star_dict*. If it is not, it will raise a *NotImplmentedError*. Otherwise, it will grab the *ci_percentage* divided by 100 in *z_star_dict* and save it as a float value known as *z_star*. Finally, it will calculate two values (*ci_min* and *ci_max*) and return them. This is done by taking *self.mean()* and subtracting/adding it by *z_star* times *se*.
2. The logic behind this function is based on De Moivre's equation. In this equation, it states if the sampling distribution is normal, the confidence interval can be taken from $CI = \bar{x} \pm z\frac{s}{\sqrt{n}}$. Note $\frac{s}{\sqrt{n}}$ is a variation of standard error based on another method of calculating confidence intervals, and is the same as the *se* output.

L. Function test_mean

1. This function finds the probability of getting a certain mean or rarer in the BST. It takes two inputs, a float known as *test_mean* and an integer known as *side*. It'll also grab values from *sorted_tree_mean.()* and store them as a list. The function will first create an integer value known as *chance.* It will then check what side the input is, with 1 designated as the left-hand side and 2 as the right-hand side. It then loops through *sorted_tree_mean.()* and checks if a given value is equal or rarer than the *test_mean* (rarity is based on what side they choose). Each time a given value is equal or rarer than the *test_mean*, it will add one to chance. It will then return the chance divided by the function *self.length()* of the BST.
2. The logic behind this function is based on finding the area of part of the sampling distribution curve. When this is done in statistics it finds the chance of a certain value (*test_mean*) being possible. Since the BST is the sampling distribution, counting the individual values in a given area is the same as finding the area under the curve. Attached is a picture that shows this concept visually. The gray portion is what this function is finding.

3. 

M. Class Main
   1. Initialization Inputs for BST
      a) Before the program does anything else, it asks for two inputs to create the BST. It first asks for a list of numbers (*sample_input*) to be the original sample to create the sampling distribution. *sample_input* is then made into a list. If any of the values in *sample_input* are nonnumeric, it will make the user re-enter the values.
      b) It then asks for the number of bootstrap samples (*num_samples_input*) that should be created (limited to 1000). Similar to *samples_input*, it will check if the input is a whole number. If it is not, the user will have to re-enter the input.
   2. Creating BST
      a) First, it creates a BST known as *my_tree* with input *sample_input.* Using the input *num_samples_input*, it then creates *num_samples_input* amount of Nodes to be placed in the tree through the function *add_tree_value()*.
   3. While Loop Interface
      a) The rest of the main function runs through a while loop that allows the user to get a variety of information about their BST. They will get a series of choices in a print statement and input a number to represent their choice known as *choice*. If *choice* is not a number between 1-7, it will make the user reinput *choice*.
      b) Mean
         (1) Prints *my_tree.mean()*
      c) Median
         (1) Prints *my_tree.median()*
      d) Range
         (1) Print *my_tree.range()*
      e) Standard Error
         (1) *my_tree.se()* is printed
      f) Confidence Interval

        (1) First, the program will ask for a float input known as *ci_choice* to represent the confidence interval they wish to see. They will be prompted with a list of acceptable choices. If they make an unacceptable choice, the program will make them re-input *ci_choice*. Otherwise, it outputs *my_tree.ci(ci_choice)* in a f-string.

    g) Test another Mean

        (1) First, the program will ask for a float input known as *test_mean* to represent the mean they want to compare to the BST. If it is not a number, they will have to re-input *test_mean*. It will then ask for a float input known as *ci_choice* to represent the confidence interval they wish to see. They will be prompted with a list of acceptable choices. If they make an unacceptable choice, the program will make them re-input *ci_choice*. It then saves the result of *self.ci()* with input *ci_choice* as *ci_min* and *ci_max*. Finally it will ask if they expect the mean to be less or greater than the expected mean and save the choice as *side*. It then runs *self.test_mean()* with inputs *test_mean*, *ci_choice*, and *side* and saves the result as *chance*. If *chance* is not between *ci_min* and *ci_max*, *test_mean* is an abnormal value and prints as such. Otherwise, it states *test_mean* is normal.

        (2) When it states a value is abnormal, it means that the value is outside a confidence interval. The full purpose of a confidence interval is that it shows the range of values that is the expected mean of a true population of a sample. If a number is outside that, it can be assumed something is going on with the sample to cause it to be outside the confidence interval and should be examined.

4. Library random and math were used. Library random was used in the function *bootstrap* to give random integers to pull values from a given list. It was also used to set a random seed to keep the results consistent. Library math was used for its function *sqrt* to calculate the square root of variance in function *se*. It was also used to give certain values a value of infinity.

5. Test Cases

I. SAT Scores in Texas Colleges
   - A. Let's say someone wanted to know what the average score needed to get into college in Texas. According to the US Department of Education, the following scores are the average SAT scores at a variety of Texas colleges:
     1. 1186 1061 1237 1288 1025 1081 1093 1271 1053 1051 1068 1155 1071 1235 918 1042 1012 992 1250 1107 1084 1147 1127 989 966 1534 1123 1041 1073 1448 1013 1267 1076 1088 1031 941 1050 1006 1275 1115 1371 1136 1291 1034 1119 919 1209 1012 1060 1392 1085 1000 1031 1120 958 1154
   - B. To get an accurate distribution of the true mean of acceptable SAT scores, let's calculate 10,000 bootstrap samples in the system to try to find the true mean.
   - C. Descriptive Statistics of Sampling Distribution:
     1. Mean: 1116.26
     2. Median: 1116.52
     3. Range: 137.66
     4. Standard Error: 17.55
   - D. Statistical Testing of Sampling Distribution:
     1. 95% Confidence Interval: 1081.25-1150.05
        - a) Note particularly with bootstrap Confidence Intervals, they can be checked against a confidence interval done by Central Limit Theorem and De Moirve's equation to see if they are accurate. You can use this link if you wish to test it yourself. It should be 5 points within our Confidence Interval (some variability should be expected, as is natural with bootstrapping)
     2. Test Mean of 1080, with 95% confidence interval, assuming mean is lower than expected: Abnormal Mean with a chance of 0.0195
   - E. Interpretation of Results
     1. The following can be inferred about SAT scores at Texas colleges. First, the mean of possible true Texas SAT score averages is 1116.26 points, while the median is 1116.52 points. The spread (standard deviation) of the synthetic data of Texas SAT means is 17.55 points and the range is 137.66 points. According to the 95% confidence interval, the true mean of the average SAT score among Texas colleges is between 1081.25 pt - 1150.05 pt. In other words, with 95% confidence, we can say the real mean of Texas college's SAT scores is between 1081.25 pt - 1150.05 pt. This means an average Texas SAT score of 1080 pt is abnormal and should be investigated for underlying causes of the low average Texas SAT score.

II. Random Numbers From 0-10
   - A. How about we use data where we already know the expected true mean: 5. Using a random number generator, let's input 50 random numbers from 0 to 10:
     1. 7 9 1 2 10 4 8 0 3 5 6 9 7 3 1 2 10 8 6 4 7 9 5 0 3 2 8 10 1 4 6 9 0 7 5 3 8 2 4 1 6 9 10 7 5 0 3 2 8 4

B. Let's once again create 10,000 samples to get a reflective sampling distribution of random numbers between 0 and 10

C. Descriptive Statistics of Sampling Distribution:
    1. Mean: 5.06
    2. Median: 5.06
    3. Range: 3.02
    4. Standard Error: 0.44

D. Statistical Testing of Sampling Distribution
    1. 95% Confidence Interval: 4.2-5.92
        a) Note particularly with Confidence Intervals, this can be checked against a confidence interval done by Central Limit Theorem and De Moirve's equation. You can use this link if you wish to test it yourself. It should be 0.1 points within our Confidence Interval (some variability should be expected, as is natural with bootstrapping)
    2. Test Mean of 5.9, with 95% confidence interval, assuming mean is higher than expected: Normal Mean with a chance of 0.0308

E. Interpretation of Results
    1. The following can be inferred about random numbers between 0-10. First, the mean of possible true means of numbers between 0-10 is 5.06, while the median is 5.06. The spread (standard deviation) of the synthetic data means is 0.44 and the range is 3.02. According to the 95% confidence interval, the true mean of random numbers between 0-10 is between 4.2-5.92. In other words, with 95% confidence, we can say the real mean for any generation of random numbers between 0-10 is between 4.2-5.92. This means an average of numbers between 0-10 being 5.9 is not abnormal, and should be expected as a possible result.

6. The current expectation of the software is that it can take a base sample of any length and calculate a sample distribution of 1000 values or less. The reason for this particular restriction is due to the *sorted_tree_mean* function using recursion and a worse-case binary tree would cause an overload in recursion. As long as the number of samples is below or equal to 1,000, the user will always be able to calculate the mean, median, range, standard error, and any given confidence interval, and to check a mean for rarity with an error. However it is still possible to calculate higher sampling numbers (up to 10,000), but a recursion error may occur. Another restriction is the amount of different confidence intervals a user can generate. Since z* values must be individually listed there's a finite amount of confidence intervals that can be calculated.