Intro

Bootstrapping is a resampling method which was developed by Bradley Efron in 1979, drawing multiple resamples from a dataset with replacement to create many simulated samples. These calculated statistics can then be used for statistical inference, such as estimating the mean and standard deviation in order to calculate confidence intervals and performing hypothesis testing. This can also be extended to evaluating the accuracy of machine learning models, such as by calculating confidence intervals using the accuracy rates of the model with bootstrapped resamples from the dataset.

Even though bootstrapping is a very useful tool for statistical inference, drawing samples and fitting them to machine learning models can be very time consuming as the number of resamples increases. One method for reducing the time taken to perform the bootstrapping is to apply parallel computing to the problem, splitting the problem into multiple smaller tasks and completed simultaneously. Throughout the report we shall try to implement parallel computing (with Python) in order to increase the speed of the method and apply it to a dataset to reduce the time taken to calculate summary statistics.

Parallel computing

Since we are implementing parallel computing using Python, we need to work around the limitations of the Global Interpreter Lock (GIL), which prevents multiple threads from executing at the same time. We can work around this by using the concurrent futures modules, which lets us run multiple processes at the same time by assigning each process its own GIL and memory space. Therefore, we will be able to take advantage of a multi core system and run a process in each core.

We know that it makes sense to implement parallel computing to the bootstrapping problem since it is a CPU bound problem, completing many computations in order to calculate the bootstrap statistics.

Introduction to dataset

Chenxing’s part

Bootstrap function

We first need to define the function which we will use for drawing bootstrap samples from the dataset. We will also create a dataframe for recording the

def h(n):

  X = df.sample(n=n,replace=True) # sample from training dataset

  return(np.mean(X))

We can see that it takes samples with replacement from the dataset and returns the mean of the sample. By taking multiple bootstrap samples we can calculate the mean and standard deviation and calculate confidence intervals. To do this we will create a function which performs each of theses tasks serially, which we will later compare to the function which implements parallel computing.

meanli = []

stdli = []

# n = sample size; m = no of samples

def serial(n,m):

  st = time.perf\_counter()

  means = []

  for a in range(m):

    means.append(h(n))

  meanli.append(np.mean(means,axis = 0))

  stdli.append(np.std(means, axis = 0))

  en = time.perf\_counter()

  return(en-st)

The function appends the calculated statistics to two lists and returns the time it takes to complete the calculations. We chose to return only the time as the performance of the code is what we are more focused on in this report. Now we will define the function which implements parallel computing using the concurrent futures module.

meanli2 = []

stdli2 = []

tim = []

def parallel(n,m,w):

  #n is how large each sample is, m is how many samples are taken

  inputs = [n]\*m

  if \_\_name\_\_ == "\_\_main\_\_":

    #uses current.futures module instead of multiprocessing

    with cf.ProcessPoolExecutor(max\_workers = w) as ex:

            #timing

      start = time.perf\_counter()

            #uses map to map h to the inputs and put into results list

      results = ex.map(h, inputs)

      #the results are a generator object, which we convert to a list

      means = [x for x in results]

      #append to lists

      meanli2.append(np.mean(means))

      stdli2.append(np.std(means))

      finish = time.perf\_counter()

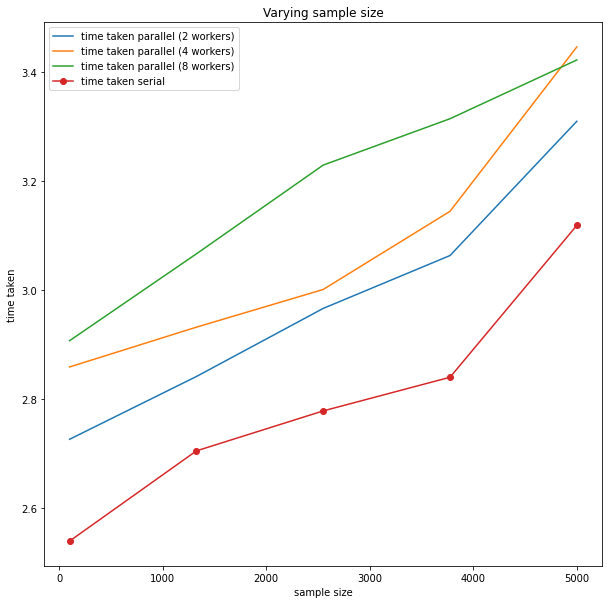
            #time taken

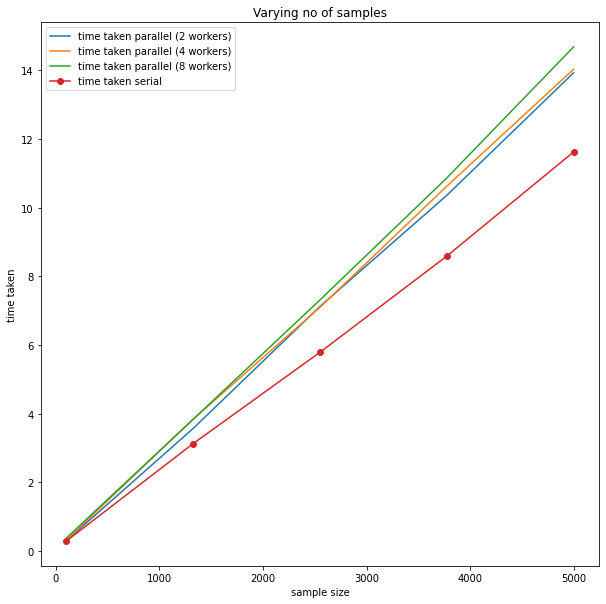
      t = finish - start

      return(t)

As we can see, the function uses the ProcessPoolExecutor subclass to create a process pool in order to run the (at most) w max\_workers at the same time. ProcessPoolExecutor uses the multiprocessing module in order to work around the GIL as we said previously. As with our serial function, the parallel function appends the calculated statistics and returns the time taken to execute the code when using 2,4 or 8 processes.

Testing

We wanted to test and compare the performance of the functions in terms of time taken to execute the code. We did this by varying the number of samples used and keeping the size of each sample constant in the first test and varying the sample size while fixing the number of samples in the second test. This represents varying the number of tasks needed to be completed when varying the number of samples taken and changing the work needed for each individual task when varying the size of each sample.

 From the graphs we can see that when varying the sample size for each bootstrapped sample the serial function outperforms the parallel computing function, with the difference in performance seemingly remaining fairly consistent as the sample sizes increase. On the other hand, when increasing the number of samples the difference in performance between the parallel and serial function increases for all values of the processes run. This shows that for a less computationally intensive task like bootstrapping samples the serial function performs much better in terms of time taken, since the processes in the parallel function need to spend a relatively significant amount of time sending data to the main process\*(wording should be checked).

[we can include graphs that show the average time taken for each number of processes (serial function would be one process)]

Model Evaluation

We can also create functions in a similar way to the more computationally intensive problem of evaluating a machine learning model’s accuracy. In this case we can apply a Decision Tree classifier (or bagging for more computationally intensive method) to the dataset in order to find the accuracy in predicting the response class. We will first define the function which each process will use in the parallel computing function will use, which draws bootstrapped samples as a training dataset, and uses the data points which were not chosen in the sample as the test dataset. The function then fits a Decision Tree Classifier and returns the accuracy score when comparing the predictions and test set response.

def g(n):

  big = list(x.index)

  X = x.sample(n=n,replace=True) # sample from training dataset

  ind = set(X.index)

  Y = y.loc[list(X.index)]

  rest = [x for x in big if x not in ind]

  xt = x.loc[rest]

  yt = y.loc[rest]

  clf = tree.DecisionTreeClassifier().fit(X,Y).predict(xt)

return(accuracy\_score(clf,yt))

We can now define the serial function which will use this function multiple times in order to calculate the statistics we need for calculating confidence intervals.

accuracy = []

# n = sample size; m = no of samples

def serial2(n,m):

  st = time.perf\_counter()

  meanacc = []

  for a in range(m):

    meanacc.append(g(n))

  accuracy.append(np.mean(meanacc))

  en = time.perf\_counter()

return(en-st)

As we can see, the serial function returns the time taken to execute the code and append the calculated statistics to a list.

acc = []

def parallel2(n,m,w):

  #n is how large each sample is, m is how many samples are taken

  inputs = [n]\*m

  start = time.perf\_counter()

  if \_\_name\_\_ == "\_\_main\_\_":

    #uses current.futures module instead of multiprocessing

    with cf.ProcessPoolExecutor(max\_workers = w) as ex:

            #timing

            #uses map to map the function to the inputs and put into results list

      results = ex.map(g, inputs)

      #returns a generator object which we need to convert to a list

      pred = np.mean([x for x in results])

      acc.append(pred)

    finish = time.perf\_counter()

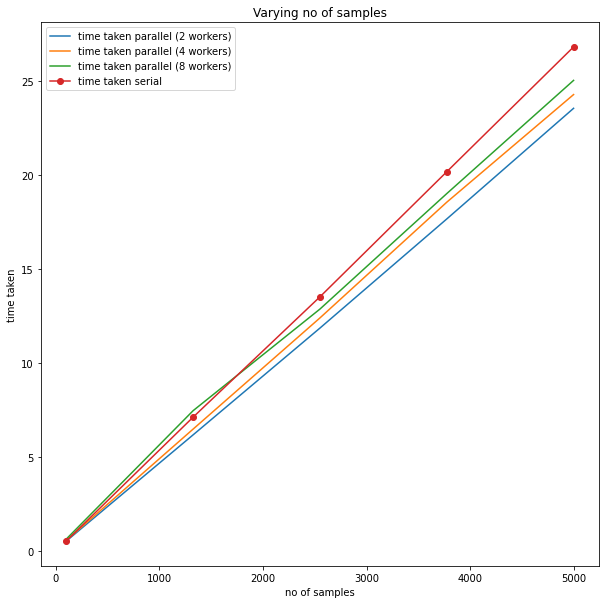
            #time taken

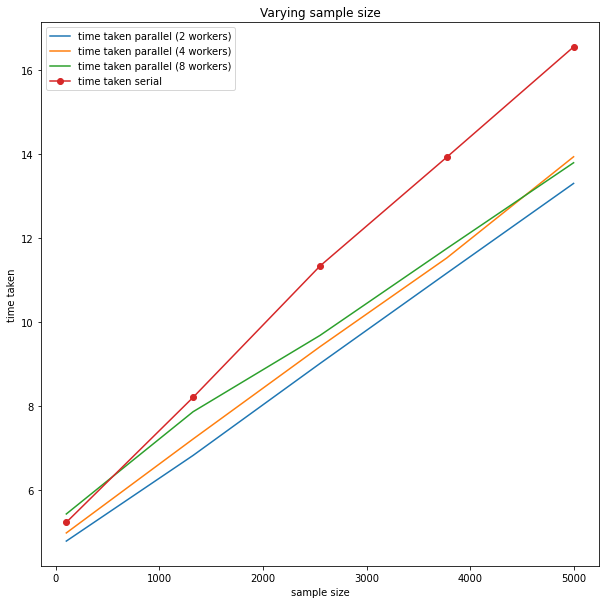
    t = finish - start

    return(t)

As we can see, the functions work very similarly to those which we tested in the previous section, and we can once again compare their performance by running them and varying the no of samples and sample size.

Results of testing

****



As we can see from the above graphs, by varying the number and size of each bootstrapped samples, the parallel computing function outperforms the serial function, with the difference in performance increasing as the sample size and (and less so the number of samples) increases. On the other hand, the process pool with only two processes performs the best in both tests compared to the cases with more processes (although the performance is very similar across all the parallel computing functions).

This shows us that for a more computationally intensive problem the extra time which the processes spend sending data is negligible relative to the time needed to run each bootstrap resampling and model fitting.

Advantages/ Disadvantages of parallel computing

Leena’s part

Conclusion