Module 2 - HW - Sort First

June 7, 2025

1 5012 HW #2. Improve code Efficiency: Sort First!

1.1 Robert Clay Harris: jbm2rt

1.2 Scenario.

In a two class, classification problem, it is common to use a classifier that outputs confidences (rather than simply class labels). A good example of this is a Support Vector Machine. A pro for using such a classifier is that you gain more information – specifically the confidence in the classification result. A con is that in order to make a final classification decision, a threshold value must be determined. For example, if a threshold of 0.75 is chosen, the class label 1 would be assigned for confidences greater than 0.75 and for confidences less than 0.75 a class label of 0 would be assigned. However, this begs the question: how is the threshold chosen?

Many data scientists will choose a threshold based on the experimental results and/or operational constraints. In this code example, we assume that we have confidences and true labels for a large data set. To determine a good threshold we will compute the true positive rates (TPRs) and false positive rates (FPRs) at all relevant thresholds. The relevant thresholds are considered those that would change the TPRs and FPRs.

In the code below, a function is defined to compute the TPR and FPR at all relevant thresholds. However, the code is not very efficient and can be improved. (Note there are tips and hints found in the comments.)

Your task is the following:

1.3 Question 1

40 POINTS
Assess the time complexity of the method computeAllTPRs(…). Provide a line-by-line assessment in comments identifying the proportional number of steps (bounding notation is sufficient) per line: eg, O(1), O(log n), O(n), etc. Also, derive a time step function T(n) for the entire method (where n is the size of input true_label).

1.4 Question 2

30 POINTS
Implement a new function computeAllTPRs_improved(…) which performs the same task as computeAllTPRs but has a significantly reduced time complexity. Also provide a line-by-line assessment in comments identifying the proportional number of steps per line, and derive a time step function T(n) for the entire method (where n is the size of input true_label).

1.5 Question 3

30 POINTS

Compare the theoretical time complexities of both methods and predict which is more efficient. Next, test your prediction by timing both methods on sample inputs of varying sizes. Create a plot of inputSize vs runtime (as done in similar class examples).

NOTE: Do not include runtimes for graphing

TOTAL POINTS: 100

---

[11]:
```
import matplotlib.pyplot as plt
import random
from copy import deepcopy
from numpy import argmax
```

Answer Question #1 in the comments of the code chunk below.

[ ]:
```
    def computeAllTPRs(true_label, confs):
        '''

        inputs:
         - true_label: list of labels, assumed to be 0 or 1 (a two class␣
↪problem)
         - confs: list of confidences

        This method computes the True Positive Rate (TPRs) and FPRs for all␣
↪relevant
        thresholds given true_label and confs. Relevant thresholds are␣
↪considered
        all different values found in confs.
        '''

        # Define / initialize  variables
        sentinelValue = -1 # used to replace max value found thus far   # O(1)
        totalPositives = sum(true_label)                                # O(n)
        totalNegatives = len(true_label) - totalPositives               # O(1)
        #print(true_label)
        truePositives = 0                                               # O(1)
        falsePositives = 0                                              # O(1)
        # Hint: Consider Memory Management
        truePositiveRate = []                                           # O(1)
        falsePositiveRate = []                                          # O(1)

        #Hint: Although not explicitly clear, the loop structure below is an
            #embeded loop ie, O(n^2) ... do you see why??
        #Hint: If you sort the confidences first you can improve the iteration␣
↪scheme.
```

```
        # Iterate over all relevant thresholds. Compute TPR and FPR for each␣
 ↪and
        # append to truePositiveRate , falsePositiveRate lists.

        for i in range(len(confs)):                  # n iterations
          maxVal = max(confs)                        #   O(n)
          argMax = argmax(confs)                     #   O(n)
          confs[argMax] = sentinelValue              #   O(1)
          #print(argMax)
          if true_label[argMax]==1:                  #   O(1)
            truePositives += 1                       #     O(1)
          else:
            falsePositives += 1                      #     O(1)

          truePositiveRate.append(truePositives/totalPositives)   #   O(1)
          falsePositiveRate.append(falsePositives/totalNegatives) #   O(1)
          #print(truePositiveRate)

        # Plot FPR vs TPR for all possible thresholds
        # plt.plot(falsePositiveRate,truePositiveRate, label ='class' + str(i)␣
 ↪+ ' to all')
        # # O(n)
        # plt.legend()                                          # O(1)
        # plt.xlabel('False Positive Rate')                     # O(1)
        # plt.ylabel('True Positive Rate')                      # O(1)
        # plt.show()                                            # O(1)

# Total time T(n):
# = O(n)        for sum(true_label)
# + O(n) * (O(n) + O(n) + O(1))  for each loop: max + argmax -> O(n) per␣
 ↪iteration
# + O(n)        for plot
# = O(n) + O(n) * (O(n) + O(n) + O(1)) + O(n)
# = O(n^2) + O(n) + O(1)
# = O(n^2)
# Total time T(n) = O(n^2)
```

[20]:
```
def testComputeAllTPRs(numSamples):

  confList = []
  labels = []
  maxVal = 10000
  #numSamples = 10000
  for i in range(0,numSamples):
    n = random.randint(1,maxVal)
    confList.append(n/maxVal)
```

```
    if n/maxVal > .5:
      lab = 1
    else:
      lab = 0
    labels.append(lab)
  #print(labels)
  computeAllTPRs(labels, deepcopy(confList))  # Why a deepcopy here?
```

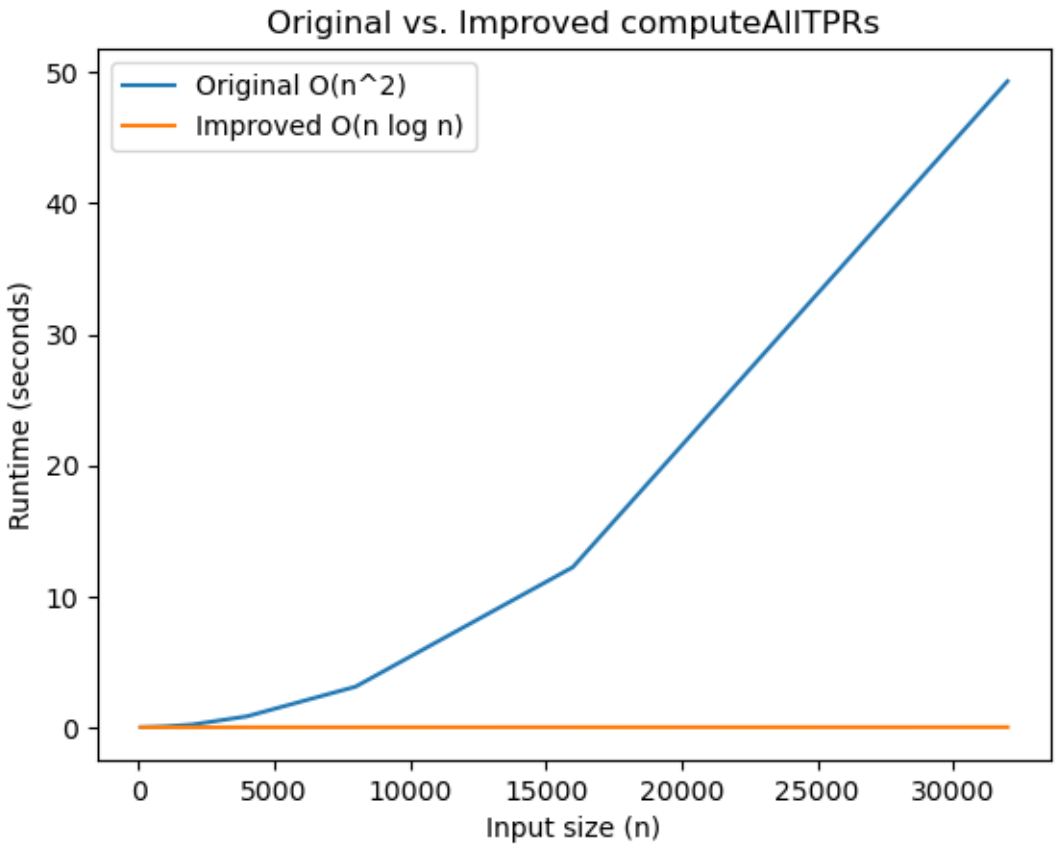
Below, provide your implementation for Question #2.

[ ]:
```
def computeAllTPRs_improved(true_label, confs):
    n = len(confs)                                      # O(1)
    totalPositives = sum(true_label)                    # O(n)
    totalNegatives = n - totalPositives                 # O(1)

    pairs = list(zip(confs, true_label))                # O(n)
    pairs.sort(key=lambda x: x[0], reverse=True)        # O(n log n)

    truePositives = 0                                   # O(1)
    falsePositives = 0                                  # O(1)
    truePositiveRate = []                               # O(1)
    falsePositiveRate = []                              # O(1)

    for conf, lbl in pairs:                             # n iterations
        if lbl == 1:                                    #     O(1)
            truePositives += 1                          #         O(1)
        else:
            falsePositives += 1                         #         O(1)
        truePositiveRate.append(                        #     O(1)
            truePositives/totalPositives
        )
        falsePositiveRate.append(                       #     O(1)
            falsePositives/totalNegatives
        )

    # plt.plot(falsePositiveRate, truePositiveRate, label ='class' + str(i) + '␣
 ↪to all')
    # # O(n)
    # plt.legend()                                                  # O(1)
    # plt.xlabel('False Positive Rate')                             # O(1)
    # plt.ylabel('True Positive Rate')                              # O(1)
    # plt.show()                                                    # O(1)


# Time complexity T(n):
# = O(n) + O(1)        for sum(true_label)
# + O(n)        for zip(confs, true_label)
```

```
# + O(n log n)  for sorting
# + O(n) + O(1)        for single pass + appends
# + O(n) + O(1)        for plot
# = O(n log n) + O(n) + O(1)
# = O(n log n)
# Total time T(n) = O(n log n)
```

Question #3. Below, provide your code which records and plots the runtime for the original and improved methods.

[26]:
```
import time


def generateTestData(numSamples):
    confList = []
    labels = []
    maxVal = 10000
    for _ in range(numSamples):
        n = random.randint(1, maxVal)
        c = n / maxVal
        confList.append(c)
        labels.append(1 if c > 0.5 else 0)
    return labels, confList

sizes = [100, 500, 1000, 2000, 4000, 8000, 16000, 32000]
orig_times = []
impr_times = []

for n in sizes:
    labels, confs = generateTestData(n)

    t0 = time.perf_counter()
    computeAllTPRs(labels, deepcopy(confs))
    orig_times.append(time.perf_counter() - t0)

    t0 = time.perf_counter()
    computeAllTPRs_improved(labels, confs)
    impr_times.append(time.perf_counter() - t0)

plt.plot(sizes, orig_times, label='Original O(n^2)')
plt.plot(sizes, impr_times, label='Improved O(n log n)')
plt.xlabel('Input size (n)')
plt.ylabel('Runtime (seconds)')
plt.title('Original vs. Improved computeAllTPRs')
plt.legend()
plt.show()
```


Original vs. Improved computeAllTPRs
Original O(n^2)
Improved O(n log n)
Runtime (seconds)
Input size (n)
0
10
20
30
40
50
0
5000
10000
15000
20000
25000
30000