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

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:

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).

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).

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

In [1]:
```
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.

T(n): 1 + n + 1 + 1 + 1 + 1 + 1 + n(2n + 7) = $2n^2$ + 8n + 6

$\therefore n^2$

In [83]:
```
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)):                                    # O(n)
        maxVal = max(confs)                                        # O(n)
        argMax = argmax(confs)                                     # O(n)
        #print(argMax)
        confs[argMax] = sentinelValue                              # O(1)
        #print(confs)
        if true_label[argMax]==1:                                   # O(1)
            truePositives += 1                                     # O(1)
        else:                                                     # O(1)
            falsePositives += 1                                   # O(1)

        truePositiveRate.append(truePositives/totalPositives)     # O(1)
        falsePositiveRate.append(falsePositives/totalNegatives)   # O(1)
    #print("Original TPR: ",truePositiveRate)
    #print("Original FPR: ",falsePositiveRate)
    '''
    # Plot FPR vs TPR for all possible thresholds
    plt.plot(falsePositiveRate,truePositiveRate, label ='class' + str(i) + ' to all'
    plt.legend()
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.show()
    '''
```

In [84]:

```
def testComputeAllTPRs(numSamples):
    confList = []                         # O(1)
    labels = []                           # O(1)
    maxVal = 10000                        # O(1)
    #numSamples = 10000
    for i in range(0,numSamples):          # O(n)
        n = random.randint(1,maxVal)
        confList.append(n/maxVal)         # O(1)
        if n/maxVal > .5:                  # O(1)
            lab = 1                       # O(1)
        else:                              # O(1)
            lab = 0                       # O(1)
        labels.append(lab)                # O(1)
    #print(labels)
    computeAllTPRs(labels, deepcopy(confList))  # Why a deepcopy here?

testComputeAllTPRs(15)
```

In [85]:

```
def merge_sort(A):
    if len(A) <= 1:
        return A
    mid = len(A) // 2
    left_list = A[:mid]
    right_list = A[mid:]
    left_list = merge_sort(left_list)
    right_list = merge_sort(right_list)

    return merge(left_list, right_list)

def merge(left, right):
    result = []
    while len(left) > 0 or len(right) > 0:
        if len(left) > 0 and len(right) > 0:
            if left[0] <= right[0]:
                result.append(left[0])
                left = left[1:]
            else:
                result.append(right[0])
                right = right[1:]
        elif len(left) > 0:
            result.append(left[0])
            left = left[1:]
        elif len(right) > 0:
            result.append(right[0])
            right = right[1:]
    return result
```

Below, provide your implementation for Question #2.

T(n): n + n + 1 + 1 + 1 + 1 + 1 + nlogn + n(6) + 2 = nlogn + 8n + 7

$\therefore nlogn$

In [86]:
```
def computeAllTPRs_improved(true_label, confs):
    totalPositives = sum(true_label)                               # O(N)
    totalNegatives = len(true_label) - totalPositives              # O(N)
    truePositives = 0                                              # O(1)
    falsePositives = 0                                             # O(1)
    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.

    a = list(zip(confs, true_label))                              # O(1)
    a = merge_sort(a)                                             # O(nlogn)
    #print(a)

    for i in range(len(a), 0, -1):                                 # O(n)
        maxVal = a[i-1][0]                                        # O(1)
        #print("maxVal:", maxVal)
        argMax = i-1                                              # O(1)
        #print(a[argMax][1])
        if a[argMax][1]==1:                                       # O(1)
            truePositives += 1                                    # O(1)
        else:                                                     # O(1)
            falsePositives += 1                                   # O(1)

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

    #print("Improved TPR: ",truePositiveRate)
    #print("Improved FPR: ",falsePositiveRate)

    # Plot FPR vs TPR for all possible thresholds
    '''
    plt.plot(falsePositiveRate,truePositiveRate, label ='class' + str(len(a)-1) + '
    plt.legend()
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.show()
    '''
```

In [87]:
```
def testComputeAllTPRs_improved(numSamples):
    confList = []                           # O(1)
    labels = []                             # O(1)
    maxVal = 10000                          # O(1)
    #numSamples = 10000
    for i in range(0,numSamples):            # O(n)
        n = random.randint(1,maxVal)
        confList.append(n/maxVal)           # O(1)
        if n/maxVal > .5:                    # O(1)
            lab = 1                         # O(1)
        else:                                # O(1)
            lab = 0                          # O(1)
        labels.append(lab)                   # O(1)
    #print(labels)
    #print(confList)
    computeAllTPRs_improved(labels, deepcopy(confList))  # Why a deepcopy here?

testComputeAllTPRs_improved(20)
```

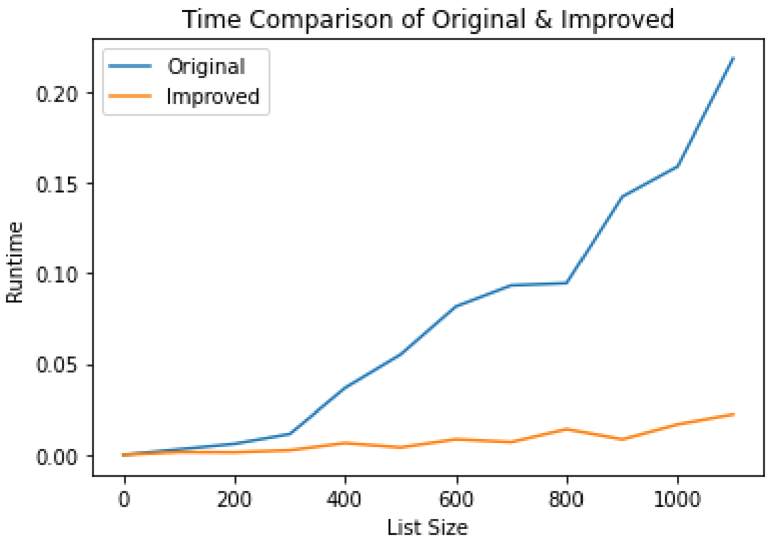
Question #3. Below, provide your code which records and plots the runtime for the original and improved methods.

In [94]:

```
'''
Run trials and record the runtimes of each sorting algorithm
'''
## record the time results
originalTime = []
improvedTime  = []

size = 1200
stepSize = 100
## calculate the time required to sort various size lists
for i in range(0, size, stepSize):
    ## generate the random list
    rList = random.sample(range(0, size), i)
    #print(rList)

    ## do the original
    start = time.perf_counter()
    testComputeAllTPRs(i)
    originalTime.append(time.perf_counter() - start)

    ## do the improved
    start = time.perf_counter()
    testComputeAllTPRs_improved(i)
    improvedTime.append(time.perf_counter() - start)

## plot the results
plt.plot(range(0, size, stepSize), originalTime, label = 'Original')
plt.plot(range(0, size, stepSize), improvedTime, label = 'Improved')
plt.legend(frameon = 'none')
plt.title('Time Comparison of Original & Improved')
plt.xlabel('List Size')
plt.ylabel('Runtime')
```

Out[94]:

```
Text(0, 0.5, 'Runtime')
```


Time Comparison of Original & Improved
Original
Improved
Runtime
0.00
0.05
0.10
0.15
0.20
0
200
400
600
800
1000
List Size

```
In [ ]:
```