Yunting_quiz02

October 23, 2021

1 DATA 442/642 Quiz 02

2 Learning in Reproducing Kernel Hilbert Spaces

2.1 Author: Yunting Chiu

2.2 Exercise 1 (10 points)

Read Section 11.11, (COMPUTATIONAL CONSIDERATIONS FOR SVM) page 576 from your book. Explain the importance of the Big-O notation and provide some examples of common Big-O complexities by looking at a complexity chart. What is the most desirable among the different complexities? Then briefly report the different computational considerations for SVM.

To express the time complexity of algorithms, we typically utilize the Big-O notation. We describe the algorithm's efficiency depending on the increasing size of the input data when we use the Big-O notation (n). The ideal time complexity is $O(1)$. $O(1)$ means it takes a constant time no matter the size of data. Knowing the algorithm's time complexity with a input data is able to help us organize our resources, process, and deliver results more efficiently and effectively. As a result, when we write down the algorithms, we should try to reduce the operation time.

When solving a quadratic programming task, it requires decompose the task into smaller ones, such as SMO and SVM algorithm. In this process, Big-O tells you the complexity of an algorithm in terms of the size of its inputs. With Big-O Notation, we use the size of the input, which we call "N". So we can say things like the runtime grows "on the order of the size of the input" (O(N)) (linear) or "on the order of the square of the size of the input" (O(Nš)) (quadratic).

Using the Big-O notation, the resulting algorithm is very efficient. Especially for the case of linear SVM, the complexity becomes of order O(N). It's not that Big-O notation itself will help you. It's that if you understand Big-O notation, you understand the worst-case complexity of algorithms. Essentially, Big-O gives you a high-level sense of which algorithms are fast, which are slow, and what the tradeoffs are.

For instance, if the input is a string, the n will be the string's length. If it's a list, the n represents the number of items in the list:

[1]:
```
# Example
string = str("howareyoudude?")
n = len(string)
print(n)

ex_list = [1, 2, 3, 4, 5, 6]
n = len(ex_list)
```

```
print(n)
```

```
14
6
```

I have practiced several exercises in Leetcode algorithm, and the following examples will show the Big-O complexities.

2.3 Example: Leetcode problem - Two Sum

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

You may assume that each input would have **exactly one solution**, and you may not use the same element twice.

You can return the answer in any order.

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Output: Because nums[0] + nums[1] == 9, we return [0, 1].

Input: nums = [3,2,4], target = 6
Output: [1,2]

Input: nums = [3,3], target = 6
Output: [0,1]
```

2.3.1 Complexity Analysis: Brute Force

- Time complexity: $O(n^2)$ as we have nested for loops. For each element, we try to find its complement by looping through the rest of the array which takes $O(n)$ time. Therefore, the time complexity is $O(n^2)$.
- Space complexity: O(1). The space required does not depend on the size of the input array, so only constant space is used.

[2]:
```
def twoSum(nums, target):
  for i in range(len(nums)):
    for j in range(i + 1, len(nums)):
      if nums[j] == target - nums[i]:
        return [i, j]

nums, target = [2,7,11,15], 9
twoSum(nums, target)
```

[2]:
```
[0, 1]
```

2.3.2 Complexity Analysis: Using Hash Table

- Time complexity: $O(n)$. We traverse the list containing n elements exactly twice. Since the hash table (in Python we called dictionary) reduces the lookup time to $O(1)$, the overall time complexity is $O(n)$.
- Space complexity: $O(n)$. The extra space required depends on the number of items stored in the hash table, which stores exactly n elements.

[3]:
```
def twoSum(nums, target):
  """
  :type nums: List[int]
  :type target: int
  :rtype: List[int]
  """
  d = {}
  for idx, num in enumerate(nums):
    if target - num not in d:
      d[num] = idx
    else:
      return [d[target-num], idx]

nums, target = [2,7,11,15], 9
twoSum(nums, target)
```

[3]:
```
[0, 1]
```

2.4 Example: Constant Time — $O(1)$

We call time complexity is $O(1)$ if the algorithm does not dependent on the input data(n). No matter the size of the input data, the running time will always be the same. Therefore, the time complexity of the example is $O(1)$.

[4]:
```
def bigAndSmall(a, b):
  if a > b:
      return True
  else:
      return False

a, b = 100, 1000
bigAndSmall(a, b)
```

[4]:
```
False
```

3 References:

- https://leetcode.com/problems/two-sum/solution/
- https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7

4 Output

[ ]:
```
# Mount the Google Drive first
%%capture
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic
!jupyter nbconvert --to pdf "/content/drive/MyDrive/American_University/
↪2021_Fall/DATA-642-001_Advanced Machine Learning/GitHub/Quizes/02/submit/
↪Yunting_quiz02.ipynb"
```