Coursework 2 PwD - Marco Catania

December 9, 2018

1 Phase 1: MyHealthcare device: Vital signs simulator

Phase 1 is about simulate data generated by a medical device (vital sign records). The simulation will be realized by generating random numbers for each vital sign category.

The data is generated by the function myHealthcare(n) using random module with seed(404) for number of observations (n) = 1000. Results of the simulation are stored in a dictionary with record type - observations as key:values. The timestamp (ts) used is [0,1,2,...,999].

```
In [1]: def myHealthcare(n):
            '''Generate data simulating a wearable device that collect vital sign data.
            Seed(404) to ensure consistency of results.
            n= size of the data to generate (int)'''

            import random
            random.seed(404)
            data = {'ts':[_ for _ in range(n)],
            'temp':[random.randint(36,39) for _ in range(n)],
            'hr':[random.randint(55,100) for _ in range(n)],
            'pulse':[random.randint(55,100) for _ in range(n)],
            'bloodpr':[random.randint(120,121) for _ in range(n)],
            'resrate':[random.randint(11,17) for _ in range(n)],
            'oxsat':[random.randint(93,100) for _ in range(n)],
            'ph':[round(random.uniform(7.1,7.6), 1) for _ in range(n)]}
            return data

        data = myHealthcare(1000)
```

2 Phase 2: Run analytics

This phase aims to find abnormal values for pulse and blood pressure and to present a frequency histogram of pulse rates (respectively by two separate functions), following by a plot of the results. The function are run on a small sample from the data generated by the function myHealthcare(n) (for n = 1000 on seed(404)).

No seed has been used.

2.1 Abnormal values for pulse or blood pressure

The data is generated by the function abnormalSignAnalytics() using random module. The function returns an array containing abnormal values from a sample, on the form [vital sign type, number of observations, [timestamp, value]...]. The example generated below is for pulse data with sample size = 50.

```
In [2]: def abnormalSignAnalytics(data, record, size):
            """Get a sample of records from the Vital sign simulator observations
            and return the abnormal values for pulse or blood pressure.
            data= data to get the sample from (dict) - generated by myHealthcare()
            record= type of the record - either 'pulse' or 'blood pressure' (str)
            size= size of the sample (int)
            """
            import random
            record = record.lower()
            sample = random.sample(data['ts'],size)
            result = list()
            count = 0
            for _ in sample:
                if record == 'pulse':
                    if data[record][_] < 60 or data[record][_] > 99:
                        count += 1
                        anomaly = list()
                        anomaly.append(_)
                        anomaly.append(data[record][_])
                        result.append(anomaly)
                else:
                    if data[record][_] == 121:
                        count += 1
                        anomaly = list()
                        anomaly.append(_)
                        anomaly.append(data[record][_])
                        result.append(anomaly)

            return record, count, result

        abnormal = abnormalSignAnalytics(data=data, record='pulse', size=50)
        abnormal
```

```
Out[2]: ('pulse', 4, [[638, 56], [570, 57], [138, 57], [0, 57]])
```

2.2 Frequency histogram of pulse rates

The data is generated by the function frequencyAnalytics() using random module. The function returns an array containing frequency histogram of pulse rate from a sample, on the form [[pulse value, frequency],...]. The example generated below is for pulse data with sample size = 50.

```
In [3]: def frequencyAnalytics(data,size):
            """Get a sample of records fron from the Vital signs simulator observations
```

```
    and return the frequency for pulse rate values.
    data = data to get the sample from (dict)
    size = size of the sample
    """

    import random
    sample = random.sample(data['pulse'],size)
    freq = {}
    for _ in {*sample}:
        count = 0
        for element in sample:
            if _ == element:
                count += 1
        freq[_] = count
    return freq

freq = frequencyAnalytics(data,50)
freq
```

```
Out[3]: {57: 2,
 59: 2,
 61: 2,
 63: 1,
 65: 4,
 66: 1,
 68: 2,
 69: 1,
 71: 2,
 72: 1,
 73: 3,
 74: 2,
 75: 2,
 78: 3,
 83: 1,
 85: 2,
 86: 2,
 87: 3,
 88: 1,
 91: 3,
 92: 1,
 94: 2,
 96: 3,
 97: 1,
 98: 1,
 100: 2}
```

2.3 Plot of the results and algorithms complexity analysis

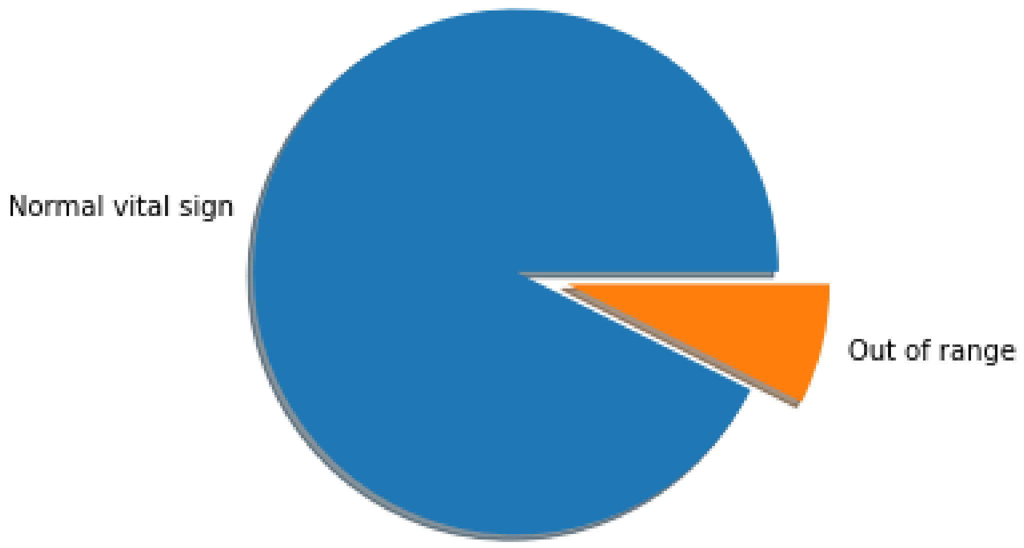
- abnormalSignAnalytics()

```
In [11]: import matplotlib.pyplot as plt

         labels = 'Normal vital sign', 'Out of range',
         sizes = [50,abnormal[1]]
         explode = (0, 0.2)

         plt.pie(sizes, explode=explode, labels=labels,
                 shadow=True)
         plt.axis('equal')
```

```
Out[11]: (-1.117404405040972,
          1.3047048447061314,
          -1.1021123194248155,
          1.107798343713656)
```



The pie chart shows a small ratio of out of range values in the sample.

The algorithm has a time and space complexity O(n) for n=sample size, in the worst case (positive linear complexity). Since the size of the sample to be choose is relatively small, there is no need to optimize the algorithm to improve performance.

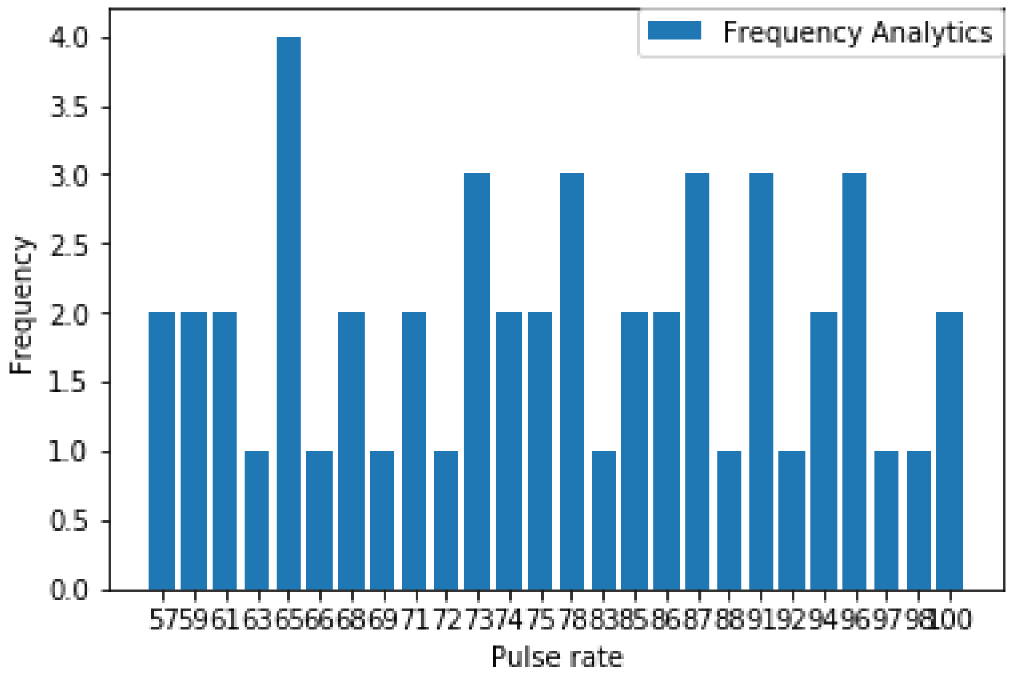
- frequencyAnalytics()

```
In [5]: import matplotlib.pyplot as plt
        import collections
```

```
freq = collections.OrderedDict(sorted(freq.items()))
keys = [str(_) for _ in freq.keys()]
vals = [_ for _ in freq.values()]
y_pos = [_ for _ in keys]
plt.bar(y_pos, vals, label="Frequency Analytics")
plt.xticks(y_pos, keys)

plt.ylabel ('Frequency')
plt.xlabel ('Pulse rate')
plt.xticks(list(keys))
plt.legend (bbox_to_anchor=(1, 1), loc="upper right", borderaxespad=0.)

plt.show()
```



The algorithm has a time and space complexity O(n) for n=sample size, in the worst case (positive linear complexity). Since the size of the sample to be choose is relatively small, there is no need to optimize the algorithm to improve performance.

3 Phase 3: Search for heart rates using the HealthAnalyzer

Phase 3 aims to develop a function healthAnalyzer() that allows to search for a particular pulse value rate. The function returns a multidimensional list with all the records associated with this value.

3.1 Algorithm

The algorithm chose to perform the search is a sequential search algorithm. Since the list on which the search is performed is unordered, we cannot use a more sophisticated algorithm with lower complexity (O(log n)). In fact we will have first to sort the list and then perform for example binary search. In that case the complexity of the algorithm with be at fast O(n*log n), far slower than the sequential search algorithm

```
In [6]: def healthAnalyzer(data=dict(), element=int):
            '''Realize a sequential search on data generated by myHealthcare() to find
            a particular value for pulse rate. The function returns a multidimensional
            list with all the records associated with this value.
            data = data used to perform the sarch on (dict)
            element = item to find in the data(int).'''

            count = 0
            ind = list()
            result = list()
            while count < len(data):
                for _ in data['pulse']:
                    if _ == element:
                        ind.append(count)
                        result.append([(col[count]) for col in list(data.values())])
                    count += 1
            return result

        healthAnalyzer(data, 56)
```

```
Out[6]: [[52, 37, 60, 56, 121, 13, 97, 7.3],
         [182, 37, 67, 56, 120, 15, 99, 7.3],
         [204, 36, 87, 56, 121, 13, 95, 7.2],
         [208, 37, 100, 56, 121, 12, 93, 7.3],
         [273, 36, 78, 56, 121, 14, 100, 7.1],
         [356, 38, 99, 56, 120, 13, 99, 7.4],
         [431, 37, 78, 56, 120, 12, 94, 7.3],
         [638, 38, 87, 56, 120, 13, 96, 7.1],
         [710, 39, 82, 56, 120, 12, 97, 7.6],
         [712, 36, 86, 56, 120, 14, 99, 7.5],
         [716, 39, 66, 56, 120, 16, 97, 7.5],
         [735, 38, 65, 56, 121, 14, 100, 7.1],
         [739, 36, 71, 56, 120, 11, 96, 7.2],
         [741, 37, 69, 56, 121, 14, 98, 7.3],
         [773, 38, 81, 56, 120, 17, 95, 7.6],
         [813, 39, 59, 56, 121, 16, 94, 7.3],
         [923, 39, 57, 56, 121, 14, 100, 7.2]]
```

3.2 Complexity

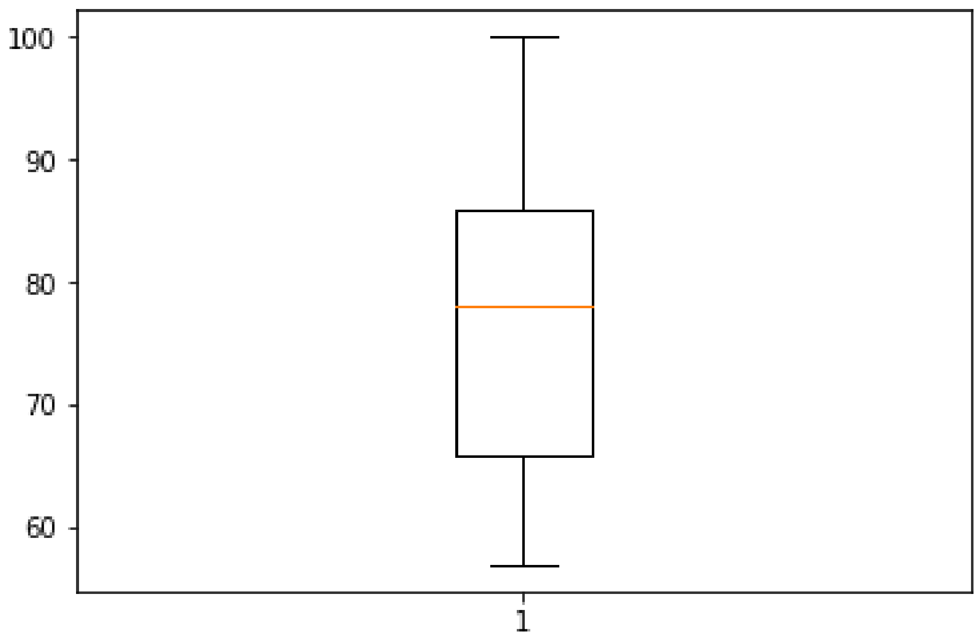
The algorithm has a time complexity O(n) in the worst case. Since the list on which the search is performed is unordered, we cannot use a more sophisticated algorithm with lower complexity (O(log n)). In fact we will have first to sort the list and then perform for example binary search. In that case the complexity of the algorithm would in the best case O(n*log n), far slower than the sequential search algorithm.

3.3 Plot of heart rate values for records having pulse rate 56

```
In [7]: import matplotlib.pyplot as plt
        hr = list()
        d = healthAnalyzer(data, 56)
        for x in range(len(d)):
          hr.append(d[x][2])

        plt.boxplot(hr)
```

```
Out[7]: {'whiskers': [<matplotlib.lines.Line2D at 0x11907c668>,
          <matplotlib.lines.Line2D at 0x11907ca90>],
         'caps': [<matplotlib.lines.Line2D at 0x11907ceb8>,
          <matplotlib.lines.Line2D at 0x11907cf98>],
         'boxes': [<matplotlib.lines.Line2D at 0x11907c128>],
         'medians': [<matplotlib.lines.Line2D at 0x119088748>],
         'fliers': [<matplotlib.lines.Line2D at 0x119088b70>],
         'means': []}
```



The boxplot shows that with pulse 56 (normal rate), the mean of heart rate in our sample is approximately 78 (also normal rate). There is a correlation between having a normal pulse rate and a normal heart rate.

4 Phase 4: Testing scalability of the algorithm

4.1 Meaure the running time and plot of the results

The function benchmarking() below is used to perform the benchmarking. The result is expressed in millisecond.

```
In [8]: def benchmarking(function, n=int):
            '''Benchmark any function similar to myHealthcare(), as requiring n as size of rec
            Return running time in milliseconds.
            -function= function to run the algorithm on.
            - n= size of the records simulated by myHealthcare (int)'''

            import time
            start = int(round(time.time() * 1000))
            test = function(n)
            end = int(round(time.time() * 1000))
            return end-start
```

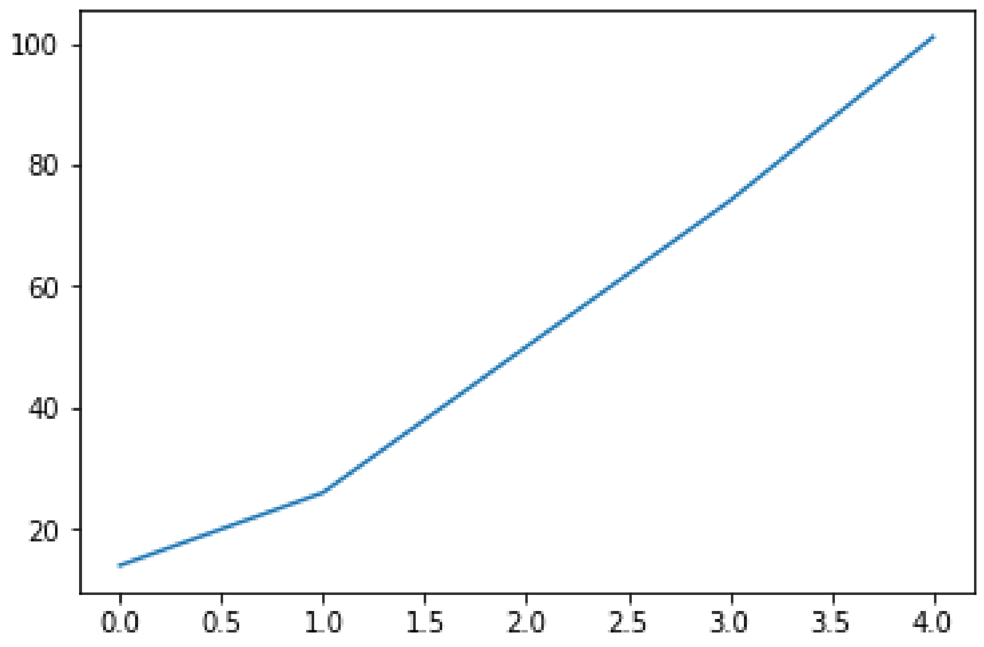
- Benchmarking on n = 1000, 2500, 5000, 7500 and 10000

```
In [9]: b1 = benchmarking(myHealthcare, 1000)
        print(b1)
        b2 = benchmarking(myHealthcare, 2500)
        print(b2)
        b3 = benchmarking(myHealthcare, 5000)
        print(b3)
        b4 = benchmarking(myHealthcare, 7500)
        print(b4)
        b5 = benchmarking(myHealthcare, 10000)
        print(b5)
```

```
14
26
50
74
101
```

```
In [10]: import matplotlib.pyplot as plt


         plt.plot([b1,b2,b3,b4,b5])
         plt.show()
         plt.clf()
```



```
<Figure size 432x288 with 0 Axes>
```

4.2 Discussion

The algorithm executes n+1 loops for each vital sign category as the number of observations generated increases. The algorithm has a time and space complexity O(n) in the worst case (linear complexity). O(n) complexity will imply slower execution as n increases.