DSA Coursework 1

Robin Rai - 100242165

December 2019

Calculating Feature Vectors

This function finds how many words in a dictionary are featured in a document. For example, if there was a dictionary with words [A,B,C,D], and a document with words [A,B,D,D], the function would return [1,1,0,2].

Algorithm 1 calculateFeatureVector(**Dictionary**, s, **Document**, w) **return Output**, s

Require: An array of reference words **Dictionary** of length s and an array of words based from **Dictionary** called **Document** of length w.

Ensure: An array **Output** of length s.

```
1: for $i \leftarrow 1$ to $s$ do                                  ▷ for every item in dictionary
2:     $counter \leftarrow 0$                                      ▷ counter reset for each dictionary item
3:     for $j \leftarrow 1$ to $w$ do                              ▷ for every item in document
4:         if $Dictionary[i] == Document[j]$ then                  ▷ if items match
5:             $counter \leftarrow counter + 1$                    ▷ increment counter
6:     $Output[i] \leftarrow counter$                              ▷ putting result into output array
7: return $Output$
```

calculateFeatureVector Analysis

1. The fundamental operation is:
   if $Dictionary[i] == Document[j]$ **then**

2. For the case, every case is the worst case/same case. The algorithm goes through both arrays in a nested loop.

3. The base run time complexity calculation is:

$$f(n) = \sum_{i=1}^{s} \sum_{j=1}^{w} 1 \tag{1}$$

Where the right side is the inner loop and the left side the outer loop. The inner loop goes through each item in the Document array while the outer loop goes through each item in the Dictionary array.

4. This simplifies to become the run time complexity function:

$$f(n) = s * w \tag{2}$$

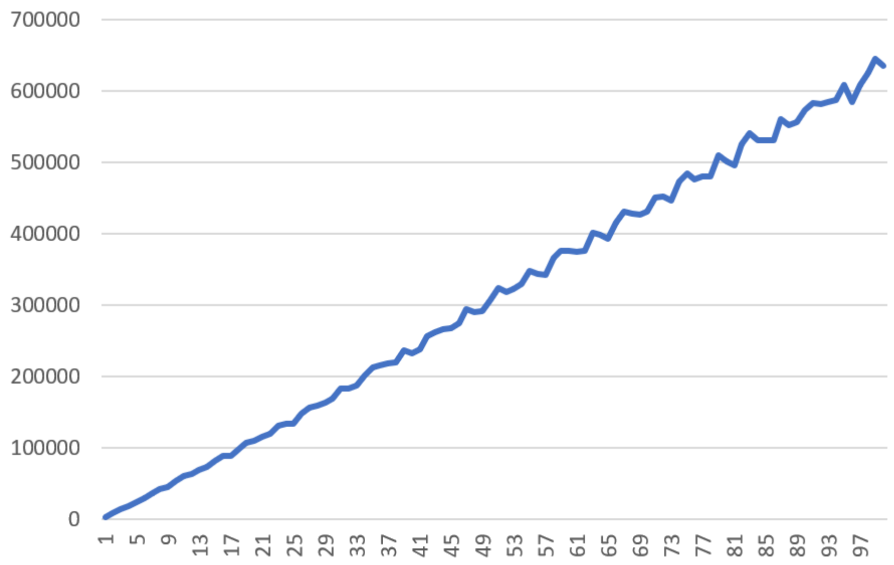
The function is polynomial. However, in the case that w is equal to s, the run time complexity function becomes:

$$f(n) = n^2 \tag{3}$$

In this case it can be described as both polynomial O(sw) and quadratic O(n^2).

calculateFeatureVector Timing experiments and code

Results when dictionary length, s is constant and document length, w increases. The values were as follows. s is 150, the length of words in the dictionary were 150. w starts at length 1 and increases by 10 until it reaches 1000. Each measured time for inputs w and s were averaged 10000 times.



Here we get a linear graph. This is because $s * w$ where s is constant is simply a linear graph of w that is steepened s times. Even though each time was averaged 10000 times, the result could still be smoother. This could be from the operating system changing priorities due to each time taking so long.

The code used for this experiment is as follows:

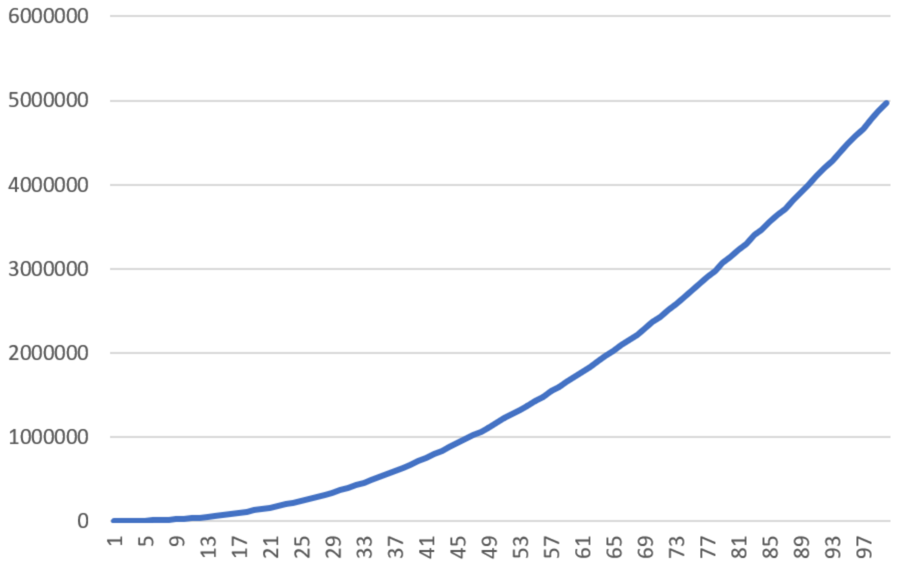
```
static void cFVConstantDic(int totalRuns, int runsPerValue,
    int dicLength, int wordLength, int startValue, int increment)
    {
    //increases doc length automatically and manually requires
        everything else
    for (int i = startValue; i <= totalRuns; i = i + increment) {
        //number of values to test
        long time = 0;
        for (int j = 1; j <= runsPerValue; j++) {
        //repeats testAmount of times for each value.
            time = time + calculateFeatureVectorTester(dicLength,
                i, wordLength);
                    //gets total time
        }
        time = time / runsPerValue;     //averages time
        System.out.println(time);
    }
}


static long calculateFeatureVectorTester(int dicLength,
    int docLength, int wordLength) {
    //test funtion that actually uses the function
    long startTime = 0;
        long endTime = 0;
    try {
        String[] dic = CourseworkUtilities.generateDictionary(
            dicLength, wordLength);
            String[] doc = CourseworkUtilities.generateDocument(
            dic, docLength);
            //generates dictionary and document
        startTime = System.nanoTime();
        calculateFeatureVector(dic, doc);
            //times only the function and nothing else
        endTime = System.nanoTime();
    } catch (Exception e) {
        System.out.println("ya done m ucked up" + e);
    }
    long timeTaken = endTime - startTime;
    //calculates time taken and returns
    return timeTaken;
}

static int[] calculateFeatureVector(String[] dictionary,
    String[] document) {
```

```
    int[] outputArray = new int[dictionary.length];
    //makes new output array, the same size of the dictionary
    for (int i = 0; i < dictionary.length; i++) {
        //for every item in dictionary
        int counter = 0;    //reset counter
        for (int j = 0; j < document.length; j++) {
            //nested loop, for every item in document
            if (document[j].equals(dictionary[i])) {
                //if current thing in doc is in current
                    item of dictionary
                counter++;
                //increment counter
                }
        }
        outputArray[i] = counter;
        //dictionary's word was in document counter number of times
    }
    return outputArray; //return array
}
```

Results when dictionary length, s as well as document length, w increases at the same time. The values were as follows: s and w start at 1 and increase by 10 until they reach 1000. Each word in the dictionary is 150 letters long. Each measured time for inputs w and s were averaged 1000 times.



As you can see the results fit the polynomial equation we were expecting.

The code for this experiment is as follows:

```
static void cFVSquare(int totalRuns, int runsPerValue, int
    wordLength, int startValue, int increment) {
    //increases doc and dic length automatically and equally.
    for (int i = startValue; i < totalRuns; i = i +
        increment) {
        //number of values to test.
        long time = 0;
        for (int j = 1; j <= runsPerValue; j++) {
        //repeats testAmount of times for each value.
            time = time + calculateFeatureVectorTester(i,
                i, wordLength);
            //gets total time
        }
        time = time / runsPerValue; //averages time
        System.out.println(time);
    }
}

static long calculateFeatureVectorTester(int dicLength,
    int docLength, int wordLength) {
        //test funtion that actually uses the function
        long startTime = 0;
        long endTime = 0;
        try {
            String[] dic =
                CourseworkUtilities.generateDictionary(dicLength,
                    wordLength);
            String[] doc =
                CourseworkUtilities.generateDocument(dic, docLength);
            //generates dictionary and document
            startTime = System.nanoTime();
            calculateFeatureVector(dic, doc);
            //times only the function and nothing else
            endTime = System.nanoTime();
        } catch (Exception e) {
            System.out.println("ya done m ucked up" + e);
        }
        long timeTaken = endTime - startTime;
        //calculates time taken and returns
        return timeTaken;
    }

static int[] calculateFeatureVector(String[] dictionary,
    String[] document) {
    int[] outputArray = new int[dictionary.length];
    //makes new output array, the same size of the dictionary
    for (int i = 0; i < dictionary.length; i++) {
        //for every item in dictionary
        int counter = 0;    //reset counter
        for (int j = 0; j < document.length; j++) {
            //nested loop, for every item in document
            if (document[j].equals(dictionary[i])) {
                //if current thing in doc is in current
                    item of dictionary
                    counter++;
                //increment counter
            }
        }
        outputArray[i] = counter;
        //dictionary's word was in document counter number of times
        }
    return outputArray; //return array
}
```

Calculating the Document Similarity Distance

This function tells us how similar a document is to another document. It takes in two feature vectors, and returns a number on how similar they are to each other. For example, if feature 1 was [1,2,0,1] and feature 2 was [1,1,1,1], the function would return 2. If it returns 0, the features are identical.

Algorithm 2 dsd(**featureVector1**, s, **featureVector2**, w) **return** *counter*

```
Require: Two feature vectors featureVector1 and featureVector2 both of length $n$.
Ensure: An integer $counter$.
1: for $i \leftarrow 1$ to $n$ do                    ▷ for every item in feature 1
2:     if $featureVector1[i] => featureVector2[i]$ then
   ▷ increase counter by difference, always positive increment
3:         $counter \leftarrow counter + (featureVector1[i] - featureVector2[i])$
4:     else
5:         $counter \leftarrow counter + (featurevector2[i] - featureVector1[i])$
6: return $counter$
```

Finding the closest matching documents

This function, findNearestDocuments, will go thorugh an array of documents, and will create an array telling us the index of the most similar document to each document in the array. For example, if we have four documents, [A,B,C,D], and B was most similar to A (making A most similar to B), and C was most similar to D (making C most similar to D), we'd get an output array of [2,1,4,3].

Algorithm 3 findNearestDocument(**D**, n, **Q**, s) **return outputArray**, n

```
Require: A list of documents D of length $n$ and a dictionary Q of length $s$.
Ensure: An array of indexes of the closes documents for each document $outputArray$.
 1: for $i \leftarrow 1$ to $n$ do                          ▷ make array of feature vectors
 2:     $featureArray[i] \leftarrow calculateFeatureVector(D[i], Q)$
 3: for $i \leftarrow 1$ to $n$ do
 4:     $minimum \leftarrow 2147483647$                     ▷ minimum is reset to largest value
 5:     for $j \leftarrow 1$ to $n$ do
 6:         if $i$ equals $j$ then continue                  ▷ skip comparing to itself
 7:         else
 8:             $temp \leftarrow dsd(featureArray[i], featureArray[j])$     ▷ check DSD
 9:             if $temp <= minimum$ then                    ▷ if it's the new smallest DSD
10:                 $minimum \leftarrow temp$                ▷ set minimum to it
11:                 $minimumIndex \leftarrow j$              ▷ record the location of it
12:     $OutputArray[i] \leftarrow minimumIndex$             ▷ after the loop the smallest
    distance is found, so add to output
13: return $OutputArray$
```

findNearestDocument analysis

1. The fundamental operation is:
   if $i == j$ **then**

   It is this as it is the comparison that is executed every time the loops execute.

2. For the case, every case is the worst case/same case. The algorithm goes through a nested loop.

3. The base run time complexity calculation is:

$$f(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 1 \tag{4}$$

   Where the right side is the inner loop and the left side the outer loop. The inner loop goes through the array of documents and finds the closest document to the document it's working on. The outside loop determines what document in said array is the one to be worked on.

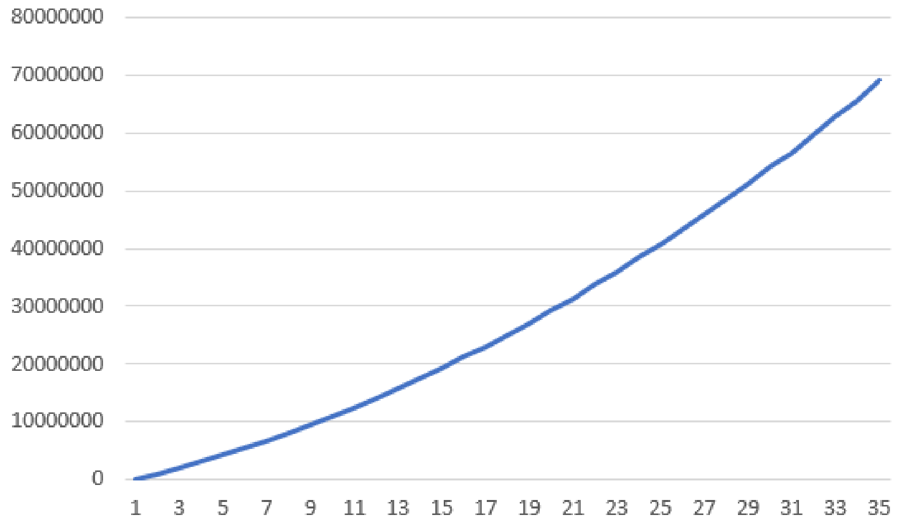
4. This simplifies to become the run time complexity function:

$$f(n) = n^2 \tag{5}$$

The function is quadratic - O(n^2).

findNearestDocument Timing experiments and code

Results when the number of documents in the document list increases. The number of them start at 1 and increase to 350, in increments 10. The number of words in dictionary, as well as the word length in both dictionary and all documents were 150. Each measured time was averaged 10000 times.



Here we get the polynomial (n^2) graph we are looking for. It is noted that the use of other operations and other functions in findNearestDocument could change timings to be of a different polynomial, as it appears to be less curved than desired.

The code used for this experiment is as follows:

```
static void fNDTDocListLength(int maxDocListLength, int increment,
    int startValue, int dicLength, int docLength, int wordLength,
        int runsPerValue) {
        //increases the length of docList automatically,
        //requiring manual values for everything else
        for (int i = startValue; i <= maxDocListLength; i +=
            increment) {
            // for every value of docList
            long time = 0;
            for (int j = 0; j <= runsPerValue; j++) {
                //do each value multiple times
                time = time + findNearestDocumentsTester(dicLength,
                    docLength, wordLength, i);
                //increment total time
            }
            time = time / runsPerValue;
            //average time
            System.out.println(time);
        }
    }

static long findNearestDocumentsTester(int dicLength, int docLength,
    int wordLength, int docListLength) {
    //generates a dictionary and a document array full of
    //documents and runs findNearestDocuments
    long startTime = 0;
    long endTime = 0;
    try {
        String[] dic = CourseworkUtilities.generateDictionary(
            dicLength, wordLength);
        String[][] docArray = new String[docListLength][];
        //creates docArray for the function test
        for (int i = 0; i < docListLength; i++) {
        //fills docArray with documents generated from dictionary.
            docArray[i] = CourseworkUtilities.generateDocument(dic,
                docLength);
            }
        startTime = System.nanoTime();
        //timing only the function call, and nothing else.
        findNearestDocuments(docArray, dic);
        //runs function
        endTime = System.nanoTime();
        //timing only the function call
    } catch (Exception e) {
        System.out.println("bad things have happened");
    }
    long timeTaken = endTime - startTime;   //calculates time taken
    return timeTaken;   // returns time taken
}

static int dsd(int[] feature1, int[] feature2) {
    int counter = 0;
    int n = feature1.length;
    for (int i = 0; i < n; i++) {
    //for every item in the first feature vector
       if (feature1[i] >= feature2[i]) {
            counter = counter + (feature1[i] - feature2[i]);
            //increments counter by distance of elements
        } else {
            counter = counter + (feature2[i] - feature1[i]);
            //increments counter by distance of elements
        }
    }
    return counter;     //returns total distance for the two features.
}

static int[] findNearestDocuments(String[][] documentArray,
    String[] dictionary) {
    int n = documentArray.length;
    //for loop is for every element in the docArray
    int minimumIndex = 0;   //index of the smallest value
    int[] outputArray = new int[n];
    int[][] featureArray = new int[n][];
    //where the feature vectors of the documents go

    for (int i = 0; i < n; i++) {
    //loop calculates all the feature vectors for the documents in
    //documentArray
        featureArray[i] = calculateFeatureVector(documentArray[i],
            dictionary);
    }
    for (int i = 0; i < n; i++) {
    //go through every item
        int minimum = 2147483647;
        //resets minimum to largest value
        for (int j = 0; j < n; j++) {
        //nested loop - for every item again
            if (i == j) {
            //if comparing document with itself, ignore
                continue;
            } else {
                int temp = dsd(featureArray[i], featureArray[j]);
                //finds distance between documents
                if (temp < minimum) {
                //if it's the newest smallest distance
                    minimum = temp;
                    //set minimum to this distance
                    minimumIndex = j;
                    //record the index/document of the new smallest
                    //distance
                }
            }
        }
        outputArray[i] = minimumIndex;
        //the closest document to i is minimumIndex/j
    }
    return outputArray;
}
```