CMP 5014Y Data Structures and Algorithms - Coursework Assignment 1

Martin Siddons

Contents

1 Feature Vector - Formal Description

Algorithm 1 CalculateFeatureVector(**D**,o **Q**,l) **return F**

Require: Array **D** of length o containing Strings. Array **D** of length l containing Strings.
Ensure: **F**, the Feature Vector of $(\mathbf{D}, \mathbf{Q})$

```
1: F ← 0                                   ▷ initialise array F to store Feature Vector
2: for i ← 1 to l do
3:     c ← o                               ▷ initialise counter
4:     for j ← 1 to o do
5:         if Q_i := D_i then
6:             c ← c + 1
7:         F_i ← c                         ▷ insert the count into position i of array F
   return F
```

2 Feature Vector - Run-time Complexity Analysis

The fundamental operation is: $c \leftarrow c + 1$. The case we are looking for is worst case - Order O. The Runtime Complexity function is as follows. The fundamental operation happens once on every loop, within another loop, therefore for any given value of o and l, we get

$$t_{calculateFeatureVector} = \sum_{i=1}^{l} \sum_{j=1}^{o} 1 \tag{1}$$

For solving the summations, we first remove the rightmost summation and apply the summation rule

$$\sum_{i=1}^{p} 1 = p \tag{2}$$

on the innermost loop to give o, which leaves

$$t_{calculateFeatureVector} = \sum_{i=1}^{l} o \tag{3}$$

Then we deal with the remaining summation using the rule

$$\sum_{i=1}^{p} a = a \sum_{i=1}^{p} 1 = ap \tag{4}$$

to get

$$t_{calculateFeatureVector} = lo \tag{5}$$

where $l \leq o$.

By characterising the above Runtime Function we find that the worst case for this algorithm is Order $n \times m$, or O(nm). In addition, I've found that the best case and average cases can be improved via sorting or dividing the Document into bins of words, but this would not improve the worst case and so I've settled with the naïve approach as shown above.

3 Feature Vector - Implementation

```
public static int[] calculateFeatureVector (String[] document,
                                            String[] dictionary){
    int size = dictionary.length;
    int[] featureVector = new int[size];
    int counter;

    for (int i = 0; i < dictionary.length; i++) {
        counter = 0;
        for (int j = 0; j < document.length; j++) {
            if (dictionary[i] == document[j]){
                counter++;
            }
        }
        featureVector[i] = counter;
    }
    return featureVector;
}
```

Code used for testing:

```
public class Main {

    public static int[] calculateFeatureVector (String[] document,
                                                String[] dictionary){
        int dictSize = dictionary.length;
        int[] featureVector = new int[dictSize];
        // initialise featureVector as the same size as the dictionary.

        int counter;
        long[] timeTaken = new long[10]; // array size 10 as we're performing 10
        long startTime, endTime;

        for (int k = 0; k < 10; k++) { // perform 10 tests at once
            startTime = System.nanoTime(); // start the timer

            for (int i = 0; i < dictionary.length; i++) {
                counter = 0;
                for (int j = 0; j < document.length; j++) {

                    if (dictionary[i] == document[j]) {
                        counter++;
                    }
                }
                featureVector[i] = counter;
            }

            endTime = System.nanoTime(); // end timer, as feature vector has bee
            timeTaken[k] = endTime - startTime; // add the time to the timer arr
        }

        // print the timing results in a row
```

```
        System.out.println("Timing Experiment results:");
        for (int i = 0; i < 10; i++) {
                System.out.print(timeTaken[i] + " ");
        }
        System.out.println();

        return featureVector;
    }

    public static void main(String[] args) throws Exception {
        // generate the dictionary
        String[] dict = CourseworkUtilities.generateDictionary(10000, 5);
        String[] doc = CourseworkUtilities.generateDocument(dict, 10000);
        int[] featureVector = calculateFeatureVector(doc, dict);
    }
}
```

4 Feature Vector - Timing Experiments

As we previously found, the Order of this algorithm is O(nm). By using the runtime complexity function

$$t_{calculateFeatureVector} = \sum_{i=1}^{l} \sum_{j=1}^{o} f \tag{6}$$

where f is the time taken to perform the fundamental operation, we could work out how long each iteration will take to compute. I first attempted using System.nanoTime() to find the average time to compute f directly, across ten sets of 10,000 fundamental operations (where $l = 10$ and $o = 1,000$) and found it was 77.70 nanoseconds. However, this varies between 55 and 96 nanosecond averages across the ten sets, and each fundamental operation is given rounded to the nearest 100 nanoseconds, so this approach does not work in practice. I understand that this is the case since, although nanoTime() provides nanosecond precision, it doesn't provide nanosecond accuracy.

To get a better idea of how the algorithm performs, I next calculated f indirectly using five different values for o where the amount of fundamental operations increases by a factor of ten on every set, while keeping l fixed to 10, as l must always be equal or less than o. I performed timing tests on each combination of values 100 times each, totalling 500 separate experiments in order to ensure I had the closest approximation. I then used four different values for o, increasing again by a factor of ten each time while keeping l fixed to 100. I completed 400 separate experiments on this data set. My third value for l was fixed to 1,000 with three variations for o, for a total of 300 tests. Finally, I performed 100 tests where both l and o were equal to 10,000. I have compiled a graph below tracking the value of f across the different values of l and o, labelled as Figure 1. Figure 2 shows how the average time to compute one set of operations differs depending on how many total operations are being performed.

Figure 1 gives the result I would expect, as the number of fundamental operations increases, so too does the time taken to compute the Feature Vector. This is not surprising based on the analysis we've already performed. This small variation could be down to characteristics of the Operating System and Processor used, including the amount of data held in cache and how much time the operating system gives our process to run. I attempted to minimise the latter by running the algorithm in 'Realtime' priority, closing as many unnecessary processes as possible and performing as many separate tests to get the closest average as possible. We can tell that in general, there is little to no variation within a group of a certain number of operations despite

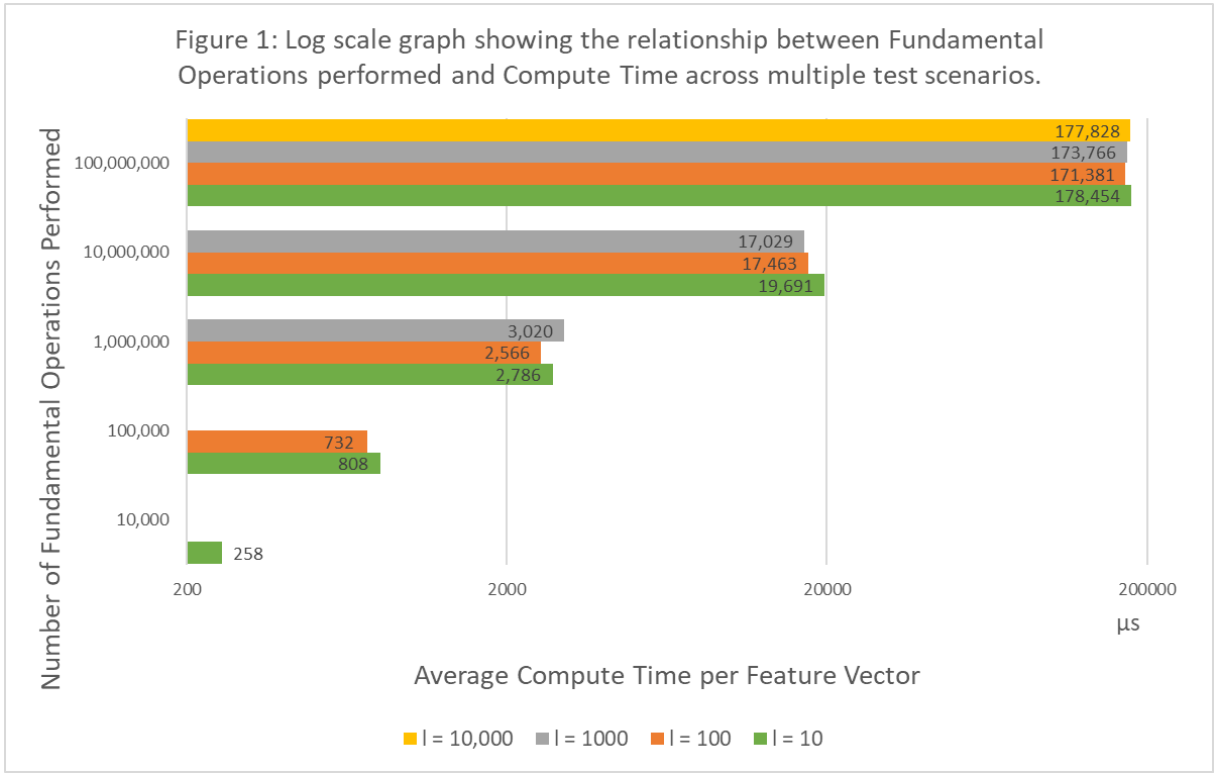

Figure 1: Comparisons performed plotted against time taken to compute a feature vector.

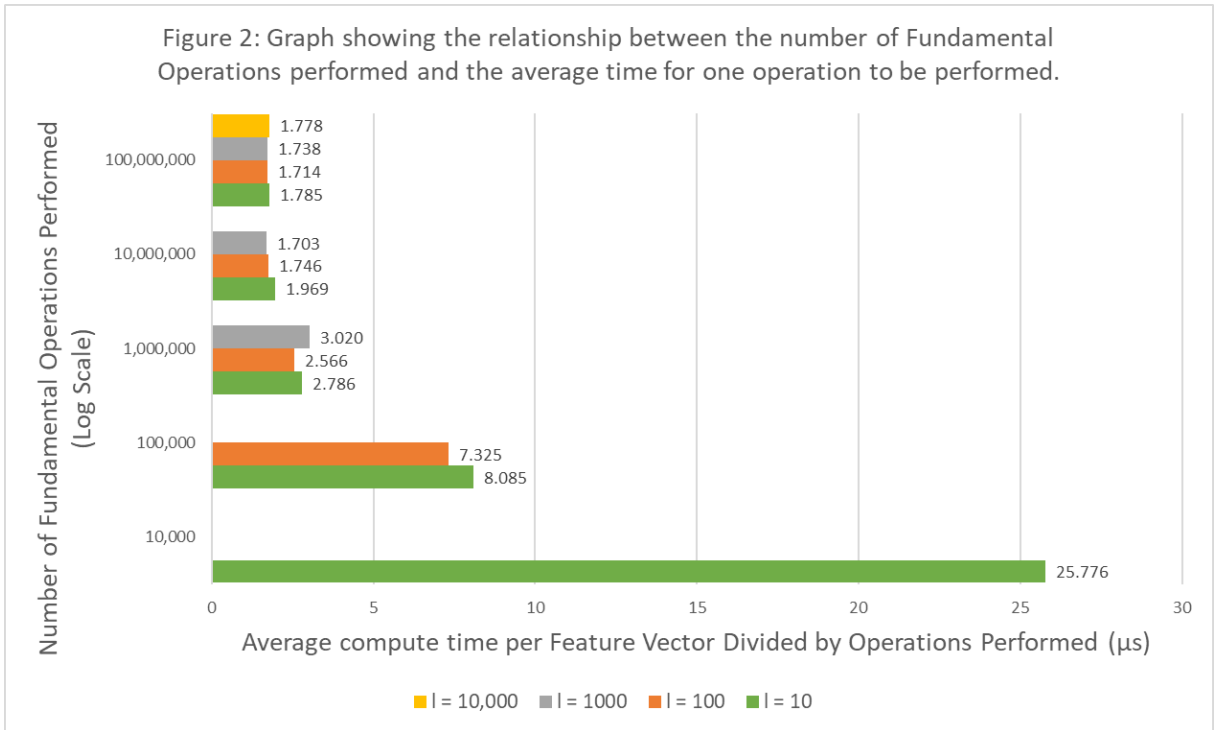

Figure 2: Comparisons performed plotted against time taken per comparison.

changing the sizes of the Dictionary and Document. i.e. the compute time is mostly the same where the Dictionary size is ten words and the Document size is 100,000 words compared to when the Dictionary and Document size are 1,000 words each. This is also as expected as the algorithm's Runtime Complexity Function is lo.

However, when examining Figure 2, we notice something more unusual. It seems the time taken to perform one fundamental operation loop of our algorithm does not scale linearly with the number of fundamental operations performed, as would be expected. In fact, it appears that the time taken to perform one average fundamental loop is in the form of a limit function, such that as x (the total number of operations performed) approaches infinity, the time taken for one fundamental operation loop approaches 1.74μs, with my test setup. I imagine this isn't due to a variance in the time it takes the fundamental operation to compute, but rather that as the number of operations increases, the less likely the worst case is followed, so the algorithm tends more towards the average case, which we have not covered here. This makes it difficult to give an estimate on how long the algorithm might take to compute in the real world without finding the limit function, but we could say where $lo > 10,000,000$

$$t = k \times l \times o \tag{7}$$

where k is the time it takes for one average operation to be performed on your chosen hardware, 1.7μs here. Finally, I wish to note that it's possible to test with longer length words however I chose not to perform testing on this as the data generated would begin to become unmanageable.

5 Document Similarity Distance - Formal Description

Algorithm 2 findDocumentSimilarity(**A1**,n **B**,n) **return** s

Require: Integer arrays **A** and **B** of length n representing feature vectors.
Ensure: The total distance between the two given feature vectors as an integer, s.

1: $s \leftarrow 0$ ▷ *initialise document similarity distance variable*
2: **for** $i \leftarrow 1$ to n **do**
3: $s \leftarrow s + |A_i - B_i|$ ▷ *Add the absolute difference between the input integers to s*
return s

6 Find Nearest Documents - Formal Description

ADT :

Algorithm 3 ListInitialise() **return L**

Ensure: An ADT, **L**.

1: *head* ← *null* ▷ *initialise head and tail to null*
2: *tail* ← *null*
3: **return L**

Algorithm 4 NodeInitialise($d1$, $d2$, *dist*)

Require: $d1$, an integer representing the first comparison document. $d2$, an integer representing the second comparison document. *dist*, the document similarity distance between the two passed documents.

1: $doc1 \leftarrow d1$
2: $doc2 \leftarrow d2$
3: *similarity* ← *dist*
4: *next* ← *null*

Algorithm 5 AddNode(**L**, $d1$, $d2$, *dist*)

Require: **L**, a previously initialised ADT. $d1$, an integer representing the first comparison document. $d2$, an integer representing the second document. *dist*, the document similarity distance between the two passed documents.

1: $node \leftarrow$ **NodeInitialise**$(d1, d2, dist)$
2: **if** $L.head := null$ **then**
3: $L.head \leftarrow n$
4: **if** $L.tail$ NOT $:= null$ **then**
5: $L.tail.next \leftarrow node$
6: $L.tail \leftarrow node$

Algorithm :

Algorithm 6 findNearestDocuments($\mathbf{D}$,f $\mathbf{Q}$,l) **return** $\mathbf{E}$

Require: $\mathbf{D}$, a list of length f of Documents. $\mathbf{Q}$, a Dictionary.

Ensure: An integer array $\mathbf{E}$ of length n representing the most-similar document to each entry in $\mathbf{D}$.

1: $\mathbf{E} \leftarrow \mathbf{0}$
2: $\mathbf{B} \leftarrow \mathbf{0}$ ▷ *initialise list of arrays B to store feature vectors*
3: $\mathbf{L} \leftarrow$ **ListInitialise**() ▷ *initialise the ADT*
4: **for** $i \leftarrow 1$ to f **do**
5: $B_i \leftarrow$ **calculateFeatureVector**($\mathbf{D_i}, \mathbf{Q}$) ▷ *call the algorithm from question 1 for each document in the list*
6: **for** $i \leftarrow 1$ to f **do**
7: **for** $j \leftarrow (i-1)$ to f **do**
8: $s \leftarrow$ **findDocumentSimilarity**($\mathbf{B_i}, \mathbf{B_j}$) ▷ *call the algorithm from question 5*
9: **AddNode**($\mathbf{L}, \mathbf{i}, \mathbf{j}, \mathbf{s}$) ▷ *add the document number and s to the list*
10: **for** $node \leftarrow L.head$ to $null$ **do**
11: **if** $node.similarity := C_{node.doc1}$ **then** ▷ *check the node's similarity on doc1*
12: $E_{node.doc1} \leftarrow node.doc2$
13: $C_{node.doc1} \leftarrow node.similarity$
14: **if** $node.similarity := C_{node.doc2}$ **then** ▷ *check similarities for doc2*
15: $E_{node.doc2} \leftarrow node.doc1$
16: $C_{node.doc2} \leftarrow node.similarity$
17: $node \leftarrow node.next$
return E

7 Find Nearest Documents - Run-time Complexity Analysis

With this algorithm we have many different components so we should be careful to select the correct component as the fundamental operation. Starting with the ADT algorithms, they all complete their work without iteration, therefore in Constant time, so it's unlikely they are what we are looking for.

Looking at the algorithm it's difficult to see which operation could be considered the Fundamental so I will do analysis on two operations, the computation using **calculateFeatureVector** which we will call $findFeatureVectors$ - and the one using **findDocumentSimilarity** which we will call $findSimilarities$ - in order to ensure we analyse the correct one.

7.1 Analysis of Fundamental Operation findFeatureVectors

The case we are analysing is the worst case - Order O. As we've already previously calculated the innermost part of the runtime complexity function, we can start with

$$t = lo + 1 \tag{8}$$

where l is the length of a Dictionary and o is the length of a Document. To this we can add the outer loop, which iterates through the entire list of documents of count f to make

$$t_{findFeatureVectors} = \sum_{i=1}^{f} lo + 1 \tag{9}$$

We can remove the constant and resolve the summation using the rule

$$t_{findFeatureVectors} = \sum_{i=1}^{p} ab = ab \sum_{i=1}^{p} 1 = abp \tag{10}$$

to leave

$$t_{findFeatureVectors} = flo \tag{11}$$

where $l \leq o$ and f can be any value greater than two independent of lo.

By characterising this runtime function we find that the worst case for this algorithm is Order $n \times m \times f$, or O(nmf). Again, this could partially be improved via sorting but that would not change the worst case.

7.2 Analysis of Fundamental Operation findSimilarities

We will again be analysing the worst case - Order O. First, we need to find the runtime complexity for findDocumentSimilarity, as designed in question 5, plus our AddNode operation. As AddNode is constant this works out to be

$$1 + \sum_{k=1}^{n} 1 \tag{12}$$

where n is the number of elements of the passed feature vectors, which will always be identical to the variable l, above.

Working back from here, we perform those operations on $i+1$ entries, so adding both remaining loops gives us the complexity:

$$t_{findSimilarities} = \sum_{i=1}^{f} \sum_{j=i+1}^{f} 1 + \sum_{k=1}^{n} 1 \tag{13}$$

To simplify this, first we remove the constants and apply the rule

$$\sum_{i=1}^{p} 1 = p \tag{14}$$

to give

$$t_{findSimilarities} = \sum_{i=1}^{f} \sum_{j=i+1}^{f} n \tag{15}$$

Next, we can use the rule

$$\sum_{i=q}^{p} a = ((p-q)+1) \times a \tag{16}$$

to give

$$t_{findSimilarities} = \sum_{i=1}^{f} ((f-(i+1))+1)n \tag{17}$$

To simplify the final sigma, we first need to reduce it using the combined rule

$$\sum_{i=1}^{p} a(p-i) = 1/2a(p-1)p \tag{18}$$

and remove the constants to give

$$t_{findSimilarities} = \frac{n(f-1)f}{2} \tag{19}$$

and finally, remove the constants once again gives

$$t_{findSimilarities} = nf^2 \tag{20}$$

Characterising this runtime function gives worst case Order $n \times m^2$ or O(nm^2).

As $n = l$, the defining attribute between the two functions is the size of o. In cases where $o > f$, findFeatureVectors will have longer runtime, and in cases where $o < f$, findSimilarities will have the longer runtime. Taking a naïve approach, I feel you can expect a document to have many hundreds or thousands of words on average but the documents to be compared could be on the order of tens to maybe hundreds, making findFeatureVectors have the longer runtime in the majority of situations, in my opinion.

8 Find Nearest Documents - Implementation

```
public static int findDocumentSimilarity(int[] doc1, int[] doc2){
    int similarity = 0;
    for (int i = 0; i < doc1.length; i++) {
        similarity = similarity + Math.abs(doc1[i] - doc2[i]);
    }
    return similarity;
}

public static int[] findNearestDocuments (String[][] documentList,
                                          String[] dictionary) {
    int dictLength = dictionary.length;
    int docListLength = documentList.length; // 'f' in formal description.
    int[] dsdList = new int[docListLength]; // 'E' in formal description.
    int[][] featureVectors = new int[docListLength][dictLength]; // 'B' in
            //formal description.
    LinkedList documentSimilarities = new LinkedList(); // 'L' in formal
            //description.

    // find the feature vectors for each document in the list.
    for (int i = 0; i < docListLength; i++) {
        featureVectors[i] = calculateFeatureVector(documentList[i], dictionary);
    }

    // fill the linkedList with the results from comparing feature vectors.
    // as the result for comparing doc1 against doc2 is the same for comparing
    // doc2 against doc1, we only have to compare each pair once, meaning we
    // only need to do (n(n+1))/2 operations, rather than n^2.
    for (int i = 0; i < docListLength - 1; i++) {
        for (int j = i+1; j < docListLength; j++) {
            int distance = findDocumentSimilarity(featureVectors[i],
                                                  featureVectors[j]);
            documentSimilarities.AddNode(i, j, distance);
        }
    }
```

```
    // For each document, find the document with the closest match to it.
    int closest[] = new int[docListLength]; // 'C' in formal description.
    Arrays.fill(closest, 999999999);

    for (ListNode node = documentSimilarities.getHead();
         node != null; node = node.getNext()) {
        int doc1 = node.getDoc1();
        int similarity = node.getSimilarity();
        int doc2 = node.getDoc2();

        if (similarity < closest[doc1] ){
            dsdList[doc1] = doc2;
            closest[doc1] = similarity;
        }
        if (similarity < closest[doc2] ){
            dsdList[doc2] = node.getDoc1();
            closest[doc2] = similarity;
        }
    }
    return dsdList;
}
```

Complete code used for testing:

```
package com.company;
import java.lang.Math;
import java.util.Arrays;

public class Main {
    public static int[] calculateFeatureVector (String[] document,
                                                String[] dictionary){
        int dictSize = dictionary.length;
        int[] featureVector = new int[dictSize];
        // initialise featureVector as the same size as the dictionary.
        int counter;

            for (int i = 0; i < dictionary.length; i++) {
                counter = 0;
                for (int j = 0; j < document.length; j++) {

                    if (dictionary[i] == document[j]) {
                        counter++;
                    }
                }
                featureVector[i] = counter;
            }
            return featureVector;
    }

    public static int findDocumentSimilarity(int[] doc1, int[] doc2){
        int similarity = 0;
```

```
    for (int i = 0; i < doc1.length; i++) {
        similarity = similarity + Math.abs(doc1[i] - doc2[i]);
    }
    return similarity;
}
public static int[] findNearestDocuments (String[][] documentList,
                                          String[] dictionary) {
    int dictLength = dictionary.length;
    int docListLength = documentList.length; // 'f' in formal description.
    int[] dsdList = new int[docListLength]; // 'E' in formal description.
    int[][] featureVectors = new int[docListLength][dictLength]; // 'B'
            // in formal.
    LinkedList documentSimilarities = new LinkedList(); // 'L' in formal
            //description.

    // Timing test variables
    long[] timeTaken = new long[100];
    long startTime, endTime;

    for (int j = 0; j < 100; j++) {
        startTime = System.nanoTime(); // start the timer

        // find the feature vectors for each document in the list.
        for (int i = 0; i < docListLength; i++) {
            featureVectors[i] = calculateFeatureVector(documentList[i], dict
        }

        endTime = System.nanoTime(); // end timer
        timeTaken[j] = endTime - startTime; // add the time to the array
    }

    // print the timing results in a row
    System.out.println("Timing Experiment results:");
    for (int i = 0; i < 100; i++) {
        System.out.print(timeTaken[i] + " ");
    }
    System.out.println();

    // fill the linkedList with the results from comparing feature
    // vectors. As the result for comparing doc1 against doc2 is the
    // same for comparing doc2 against doc1, we only have to compare
    // each pair once, meaning we only need to do (n(n+1))/2 operations,
    // rather than n^2.
    for (int i = 0; i < docListLength - 1; i++) {
        for (int j = i+1; j < docListLength; j++) {
            int distance = findDocumentSimilarity(featureVectors[i],
                                                  featureVectors[j]);
            documentSimilarities.AddNode(i, j, distance);
        }
    }

    // For each document, find the document with the closest match to it.
```

```
        int closest[] = new int[docListLength]; // 'C' in formal description.
        Arrays.fill(closest, 999999999);

        for (ListNode node = documentSimilarities.getHead();
             node != null; node = node.getNext()) {
            int doc1 = node.getDoc1();
            int similarity = node.getSimilarity();
            int doc2 = node.getDoc2();

            if (similarity < closest[doc1] ){
                dsdList[doc1] = doc2;
                closest[doc1] = similarity;
            }
            if (similarity < closest[doc2] ){
                dsdList[doc2] = node.getDoc1();
                closest[doc2] = similarity;
            }
        }
        return dsdList;
    }

    public static void main(String[] args) throws Exception {
        // generate the dictionary
        String[] dict = CourseworkUtilities.generateDictionary(100, 5);
        int numberOfDocs = 10000; // number of documents in the doc list.
        int lengthOfDocs = 1000; // number of words in a document.
        String[][] docList = new String[numberOfDocs][lengthOfDocs];

        // generate the list of documents.
        for (int i = 0; i < numberOfDocs; i++) {
            docList[i] = CourseworkUtilities.generateDocument
                                                (dict, lengthOfDocs);
        }

        int[] dsdList = findNearestDocuments(docList, dict);
    }
}
```

9 Find Nearest Documents - Timing Experiments

As we found in section 7, the part of our algorithm we called findFeatureVectors was determined to have the longer runtime on the majority of sets, so I will ensure that for all tests the amount of documents to be compared does not exceed the length of each document and only that section of the algorithm is timed. For this series of testing, there are four variables we can change: w, the length of each word in Dictionary Q and Document D, n (or l), the length of Dictionary Q, o, the length of Document D, and f, the count of Document D in List E. As o, l and f are the elements of the algorithm that directly affect timing, I would focus on varying these, however using more than two variables would make testing difficult therefore I choose to set o to a constant value for these experiments. I will set w to 5, as before, and o will be kept at 1,000 to keep processing times low. I will perform measurements on combinations of 10, 100 and 1000 for the value of l, and the same for f, with the lower values for l being tested at values 10,000 and 100,000 for f. For each combination, I will perform 100 tests in a for-loop to ensure

we have a good estimate of the time taken. Figure 3 tracks the relationship between the set of operations being performed and their compute time, with Figure 4 showing the relationship between a set of operations being performed, at the time it takes for roughly one operation in that set to execute.

As expected for Figure 3, this time around the graph increases almost directly linearly with the increasing numbers of operations. Towards the lower end of the graph the operations do seem to take a fraction longer to compute across the entire set, but in general, increasing the number of operations by log base ten increases the time taken to compute tenfold. Most notably, however, is that there is a clear trend between the compute time taken and the length of the Dictionary, with larger dictionaries taking less time to compute overall. This is counter to the previous tests where changing the size of the Dictionary had no overall effect on the time taken to compute the algorithm, but obvious when thought about. Since having a shorter Dictionary would mean there are more documents needed in the list to equal the number of fundamental operations performed here (since one million operations could be a Dictionary of length 10 on a Document of length 1000 on a list of Documents of length 100, or a Dictionary of length 100 on a Document of length 1000 on a list of Documents of length 10). These results clearly show that performing operations on a larger number of documents with a shorter dictionary is more computationally expensive than the inverse, despite what I would assume from my analysis of the algorithm, which stated that $10 \times 1000 \times 100 = 100 \times 1000 \times 10$. I can only assume that the extra time in this case is due to writing a greater number of values to the array featureVectors, which would be an element that would be lost as a constant in algorithmic analysis.

When plotting the operations performed against the average time one operation takes, we can clearly see two trends in Figure 4 that, after seeing Figures 2 and 3, should now be expected. Firstly, like in Figure 2, the time for the average operation to be performed continues to decrease as the number of operations performed increases. The curve isn't quite as pronounced on Figure 4 compared to Figure 2, however Figure 4 is shifted a factor of ten up the scale, which would appear to account for the difference. The time per operation is also clearly larger this time around, however that is likely due to more processing having to occur this time, as each fundamental operation computes the feature vector, adds it to an array and iterates through a list of Documents, requiring much more data to be loaded from RAM and processed in one million operations compared to the operations in Figure 2. Secondly, it's clear that as we found with Figure 3, the larger the List of Documents (and the smaller the Dictionary), the longer those operations take compared to their inverse in the same set. This is not surprising as we previously discovered this from looking at Figure 3, and the intent of Figure 4 was not to bring any further attention to this.

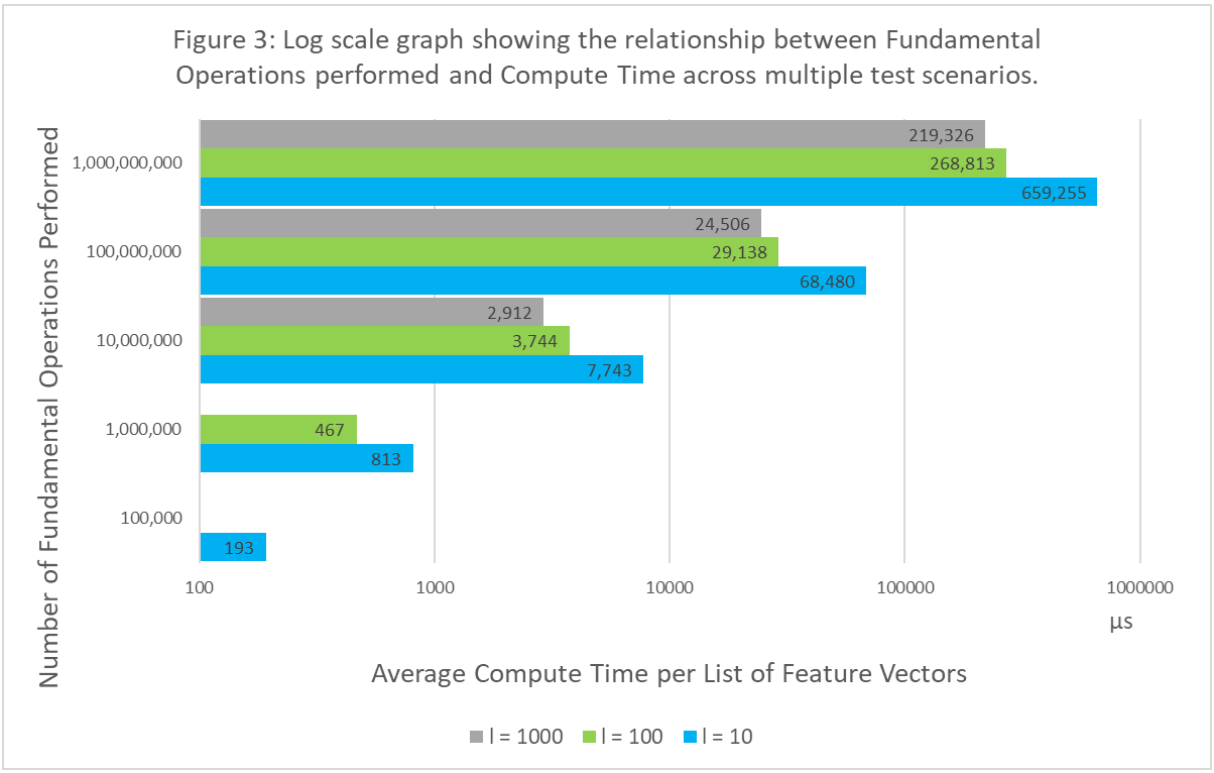

Figure 3: Comparisons performed plotted against time taken to compute a set of feature vectors.

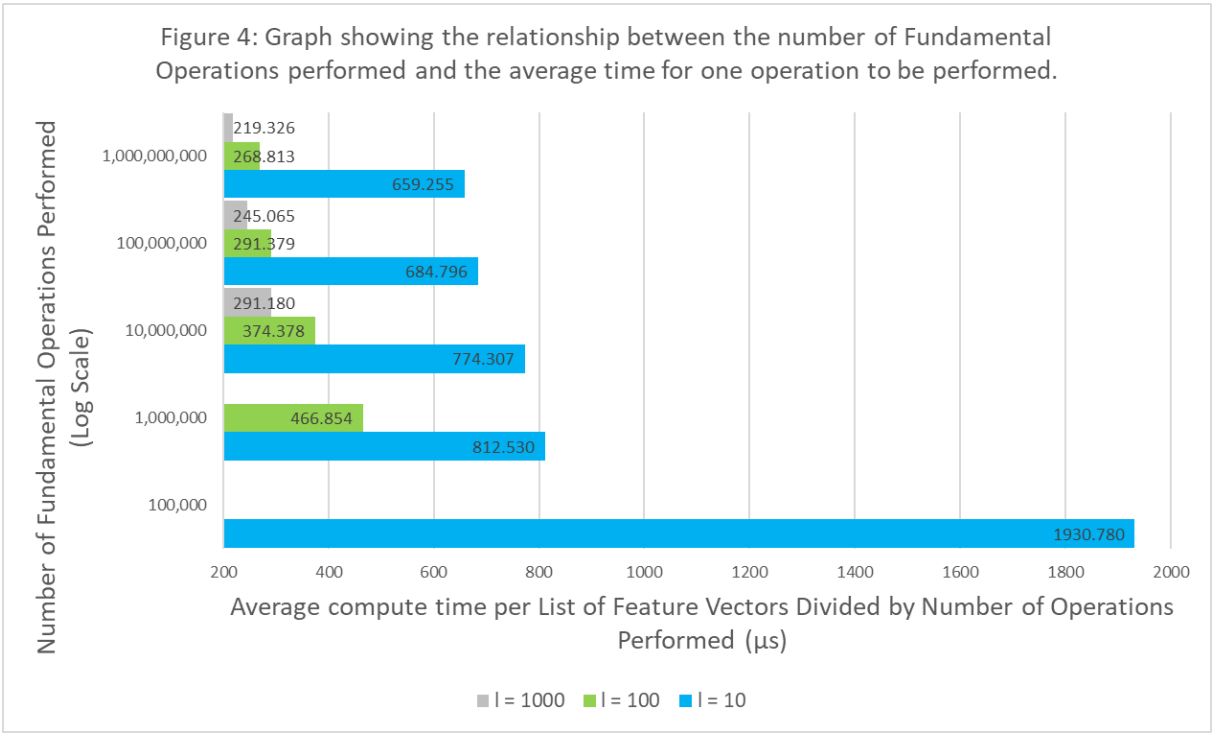

Figure 4: Comparisons performed on all documents in the list plotted against time taken per comparison.