worksheet_04

February 11, 2024

1 Worksheet 04

Name: Daniyal Ahmed UID: U11469883

1.0.1 Topics

- Distance & Similarity

1.0.2 Distance & Similarity

Part 1

a) In the minkowski distance, describe what the parameters p and d are.

p is a parameter that when set to one represents that manhattan distance when set to two represents the Euclidean distance

b) In your own words describe the difference between the Euclidean distance and the Manhattan distance.

The distance between two Points is the length of the line between the two points So for example if there are two points x and y, the best way to calculate the euclidean distance is by drawing a sort of right triangle between the two. Both points show have a diagonal line connecting the two representing the hypotenuse of our triangle. If you correctly calculate the hypotenuse that represents your Euclidean distance. Whereas Manhattan distance does not allow you to just draw a line across instead you can only travel up or across each block in the grid to get from x to y.

Consider A = (0, 0) and B = (1, 1). When:

- p = 1, d(A, B) = 2
- p = 2, d(A, B) = $\sqrt{2} = 1.41$
- p = 3, d(A, B) = $2^{1/3} = 1.26$
- p = 4, d(A, B) = $2^{1/4} = 1.19$

c) Describe what you think distance would look like when p is very large.

I feel that the distance of the two will get closer and closer to 1, this is because I am thinking of calculas and think of the distance as a function where p gets closer to infinity the distance function starts to converge at a specific point. It cannot be zero because that would obviously just break the distance function

d) Is the minkowski distance still a distance function when p < 1? Expain why / why not.

No, as it was proved by contradiction in lecture, as the 1/p in Minkowski will become over one while the ˆP inside the summation will become below one, henceforth breaking the distance function

e) when would you use cosine similarity over the euclidan distance?

- Say you have two documents talking about the same topic, but one uses more words and is longer overall Thus Euclidan distance would show them farther apart and imply they are very different since one technically has more words is therefore is ‘different’. However, the Cosine similarity angle will show that the documents are still quite similar even though one is longer and the other is shorter

f) what does the jaccard distance account for that the manhattan distance doesn’t?

If you have two documents but those two documents only differ in two words, all that the manhattan distance will tell you is 2, however now lets now you have two more documents that only have two words Manhatten distance will still tell you 2. The problem here is that the manhattan distance doesn’t give a lot of information when trying to compare two documents. However the Jaccard distance compares the intersection of the two documents over the union of all the words in either document. Giving you a more realistic comparison between the two.

Part 2 Consider the following two sentences:

[12]:
```
s1 = "hello my name is Alice"
s2 = "hello my name is Bob"
```

using the union of words from both sentences, we can represent each sentence as a vector. Each element of the vector represents the presence or absence of the word at that index.

In this example, the union of words is (“hello”, “my”, “name”, “is”, “Alice”, “Bob”) so we can represent the above sentences as such:

[13]:
```
v1 = [1,    1, 1,   1, 1,    0]
#     hello my name is Alice
v2 = [1,    1, 1,   1, 0, 1]
#     hello my name is    Bob
```

Programmatically, we can do the following:

[14]:
```
corpus = [s1, s2]
all_words = list(set([item for x in corpus for item in x.split()]))
print(all_words)
v1 = [1 if x in s1 else 0 for x in all_words]
print(v1)
```

```
['Bob', 'Alice', 'hello', 'is', 'name', 'my']
[0, 1, 1, 1, 1, 1]
```

Let’s add a new sentence to our corpus:

[15]:
```
s3 = "hi my name is Claude"
corpus.append(s3)
```

a) What is the new union of words used to represent s1, s2, and s3?

[16]:
```
all_words = list(set([item for x in corpus for item in x.split()]))
#Basically we have to recaculate the the union set to account for our new␣
 ↪addition into the
#corpus



print(all_words)
```

```
['Bob', 'Alice', 'Claude', 'hello', 'hi', 'is', 'name', 'my']
```

b) Represent s1, s2, and s3 as vectors as above, using this new set of words.

[17]:
```
v1 = [1 if x in s1 else 0 for x in all_words]
v2 = [1 if x in s2 else 0 for x in all_words]
v3 = [1 if x in s3 else 0 for x in all_words]
print(v1, "\n", v2, "\n", v3  )
```

```
[0, 1, 0, 1, 0, 1, 1, 1]
 [1, 0, 0, 1, 0, 1, 1, 1]
 [0, 0, 1, 0, 1, 1, 1, 1]
```

c) Write a function that computes the manhattan distance between two vectors. Which pair of vectors are the most similar under that distance function?

[18]:
```
'''I simplified the formula from Minkowski since this question is purely asking
about the manhatten distance'''



def manhatten_distance(x,y):
    if len(x)!= len(y):
        raise ValueError("x and y must be the same dimensions")

    res= 0
    for i in range(len(x)):
        res+= abs(x[i]-y[i])

    return res



arr = [manhatten_distance(v1,v2), manhatten_distance(v1,v3),␣
 ↪manhatten_distance(v2,v3)]

minimum_dis =  lambda x:( 'v1:',v1,'v2:', v2 )if min(x)==x[0] else (('v1:
 ↪',v1,'v3:',v3) if min(x)==x[1] else ('v2:' ,v2,'v3:', v3))
print(minimum_dis(arr))
```

```
('v1:', [0, 1, 0, 1, 0, 1, 1, 1], 'v2:', [1, 0, 0, 1, 0, 1, 1, 1])
```

d) Create a matrix of all these vectors (row major) and add the following sentences in vector form:

- “hi Alice”
- “hello Claude”
- “Bob my name is Claude”
- “hi Claude my name is Alice”
- “hello Bob”

[19]:

```
s4 = 'hi Alice'
s5= 'hello claude'
s6 = 'Bob my name is Claude'
s7 = 'hi Claude my name is Alice'
s8 = 'hello bob'

class Matrix:

    def __init__(self,corpus, union):
        #The Matrix itself is a list that will store lists
        self.Matrix = []
        #The Corpus was Just to account for the corresspondence
        self.Corpus = corpus
        #Union set of all the words in the corpus
        self.union = union
        #What vector corresponds to what sentence
        self.correspondence = []

    def add_row(self, s):
        self.Matrix.append([1 if x in s else 0 for x in self.union])
        #Every time we add a sentance vector to our matrix we append the␣
 ↪sentence in the corresponding indices
        self.correspondence.append(s)

    #Simple print method
    def __repr__(self):
        string = ""

        for i in range(len(self.Matrix)):
            string+= "[ "
            for j in range(len(self.Matrix[0])):
                string+=str(self.Matrix[i][j])+" "

            string+= "]\n"


        return string

    #Dummy function for Testing
    def append_to_corpus(self, arr):
        for x in arr:
            self.Corpus.append(x)
        self.union = list(set([item for x in self.Corpus for item in x.
↪split()]))

    #a sort of get function for part e
    def numCols(self):
        return(len(self.Matrix[0]))

    #a sort of get function for part e
    def numRows(self):
        return(len(self.Matrix))

    #get function dummy function for debugging
    def getMatrix(self):
        return self.Matrix

    #For Part F
    def most_similar(self):
        #Set the distance to infinity
        distance = float('inf')

        #the indices for the two vectors
        indices = (0,0)
        #for some vector in the matrix
        for i in range(len(self.Matrix)):
            #For some other vector in the matrix
            for j in range(len(self.Matrix)):
                # We cannot compare the same vectors in the matrix
                if(i==j):
                    pass

                else:
                    #If we find a distance between two vectors that is smaller␣
↪than any we have seen we can replace the distance and mark down the indices
                    if(distance> manhatten_distance(self.Matrix[i],self.
↪Matrix[j])):
                        distance = manhatten_distance(self.Matrix[i],self.
↪Matrix[j])
                        indices = (i,j)

        #We use our correspondence array to easily access the sentences
        return [self.correspondence[indices[0]],self.correspondence[indices[1]]]
```

```
Mat = Matrix(corpus=corpus,union=all_words)


Mat.add_row(s4)
Mat.add_row(s5)
Mat.add_row(s6)
Mat.add_row(s7)
Mat.add_row(s8)

print(Mat)
```

```
[ 0 1 0 0 1 0 0 0 ]
[ 0 0 0 1 0 0 0 0 ]
[ 1 0 1 0 0 1 1 1 ]
[ 0 1 1 0 1 1 1 1 ]
[ 0 0 0 1 0 0 0 0 ]
```

e) How many rows and columns does this matrix have?

[20]:

```
print(f"Columns {Mat.numCols()}")
print(f"Rows: {Mat.numRows()}")



#I did this more so in the matrix class for ease of use.
'''

    #a sort of get function for part e
    def numCols(self):
        return(len(self.Matrix[0]))

    #a sort of get function for part e
    def numRows(self):
        return(len(self.Matrix))




'''
```

```
Columns 8
Rows: 5
```

[20]:

```
'\n\n    #a sort of get function for part e\n    def numCols(self):\n
return(len(self.Matrix[0]))\n    \n    #a sort of get function for part e\n
def numRows(self):\n        return(len(self.Matrix))\n\n\n\n'
```

f) When using the Manhattan distance, which two sentences are the most similar?

[21]:

```
print(Mat.most_similar())

#I did this more so in the matrix class for ease of use.
'''
    #For Part F
    def most_similar(self):
        #Set the distance to infinity
        distance = float('inf')

        #the indices for the two vectors
        indices = (0,0)
        #for some vector in the matrix
        for i in range(len(self.Matrix)):
            #For some other vector in the matrix
            for j in range(len(self.Matrix)):
                # We cannot compare the same vectors in the matrix
                if(i==j):
                    pass

                else:
                    #If we find a distance between two vectors that is smaller␣
 ↪than any we have seen we can replace the distance and mark down the indices
                    if(distance> manhatten_distance(self.Matrix[i],self.
 ↪Matrix[j])):
                        distance = manhatten_distance(self.Matrix[i],self.
 ↪Matrix[j])
                        indices = (i,j)

        #We use our correspondence array to easily access the sentences
        return [self.correspondence[indices[0]],self.correspondence[indices[1]]]
'''
```

```
['hello claude', 'hello bob']
```

[21]:

```
"    \n    #For Part F\n    def most_similar(self):\n        #Set the distance
to infinity\n        distance = float('inf') \n\n        #the indices for the
two vectors\n        indices = (0,0)\n        #for some vector in the matrix\n
for i in range(len(self.Matrix)):\n            #For some other vector in the
matrix\n            for j in range(len(self.Matrix)):\n                # We
cannot compare the same vectors in the matrix\n                if(i==j):\n
pass\n\n                else:\n                    #If we find a distance
between two vectors that is smaller than any we have seen we can replace the
distance and mark down the indices\n                    if(distance>
manhatten_distance(self.Matrix[i],self.Matrix[j])):\n
distance = manhatten_distance(self.Matrix[i],self.Matrix[j])\n
```

```
indices = (i,j)\n\n        #We use our correspondence array to easily access the
sentences \n        return
[self.correspondence[indices[0]],self.correspondence[indices[1]]]\n"
```

Part 3 Challenge Given a set of graphs $\mathcal{G}$, each graph $G \in \mathcal{G}$ is defined over the same set of nodes V. The graphs are represented by their adjacency matrices, which are 2D arrays where each element indicates whether a pair of nodes is connected by an edge.

Your task is to compute the pairwise distances between these graphs based on a specific distance metric. The distance $d(G, G')$ between two graphs $G = (V, E)$ and $G' = (V, E')$ is defined as the sum of the number of edges in G but not in G', and the number of edges in G' but not in G. Mathematically, this can be expressed as:

$$d(G, G') = |E \quad E'| + |E' \quad E|.$$

Requirements:

1. **Input**: Should take a list of 2D numpy arrays as input. Each array represents the adjacency matrix of a graph.
2. **Output**: Should output a pairwise distance matrix. If there are n graphs in the input list, the output should be an $n \times n$ matrix where the entry at position (i, j) represents the distance between the i^{th} and j^{th} graph.

[22]:
```python
import numpy as np


def distance_between_graphs(numpy_arrays):

    #Make the pairwise distance matrix
    result = [[0 for x in numpy_arrays] for y in numpy_arrays]

    for  i in range(len(numpy_arrays)):
        for j in range(len(numpy_arrays)):
            #E being our first array
            E = numpy_arrays[i]
            #E prime being our second array coressponding to the formula
            E_prime= numpy_arrays[j]


            if(i==j):
                pass

            else:

                '''
                The equation of | E \\ E' | + |E' \\ E| is equal to the total␣
 ↪number of edges they both have
```

```
                in difference, this is because the | E \\ E' | represents the␣
↪number of edges that G has that
                G' does not, or hence the set of Unique edges. The same can be␣
↪said for |E' \\ E| without loss of
                Generality, thus the Set of unique edges of the two is equal␣
↪the total number of edges that only one of them
                as but not both. This is why we can use np.not_equal since the␣
↪adjaceny matrices are represented with
                either a 1 or a 0. Thus if the i and j are not equal that means␣
↪they are unique to that specifc graph.
                '''

                result[i][j]= sum(sum(np.not_equal(E,E_prime)))


    return np.array(result)
```