# Major assignment 1: Your submission

This is your assignment template for [BigDataX Major assignment 1](https://courses.edx.org/courses/course-v1:AdelaideX+BigDataX+3T2018/courseware/1536116e98fc4b679c36d26ce63f188c/deb84cbeb0e04ac5bfff48a0952ebc45/1?activate_block_id=block-v1%3AAdelaideX%2BBigDataX%2B3T2018%2Btype%40vertical%2Bblock%409418efc14310446c98133a5caa1e39b3). Save this document on your local machine and include all of your work within the relevant sections. Once you’ve completed all two parts of the assignment, upload the document via the submission area on the “[Submit your assignment](https://courses.edx.org/courses/course-v1:AdelaideX+BigDataX+3T2018/courseware/1536116e98fc4b679c36d26ce63f188c/deb84cbeb0e04ac5bfff48a0952ebc45/6?activate_block_id=block-v1%3AAdelaideX%2BBigDataX%2B3T2018%2Btype%40vertical%2Bblock%406ee27edd182f4683859794f5991991ec)” page at the end of Major assignment 1.

# You will need to include:

# the answer to the questions

* **all of your calculations/working that sufficiently justify your answers**

# Your answers and calculations/working will assist the University of Adelaide academic staff member to assess your submission.

**Quick links:**

[Major assignment 1: Part 2 Clustering](#_Major_assignment_1:)

[Major assignment 1: Part 3 PageRank](#_Major_assignment_1:_1)

# Major assignment 1: Part 2 Clustering

1. Your answer and all associated working/calculations to step 1 [4 points]

Your answer goes here:

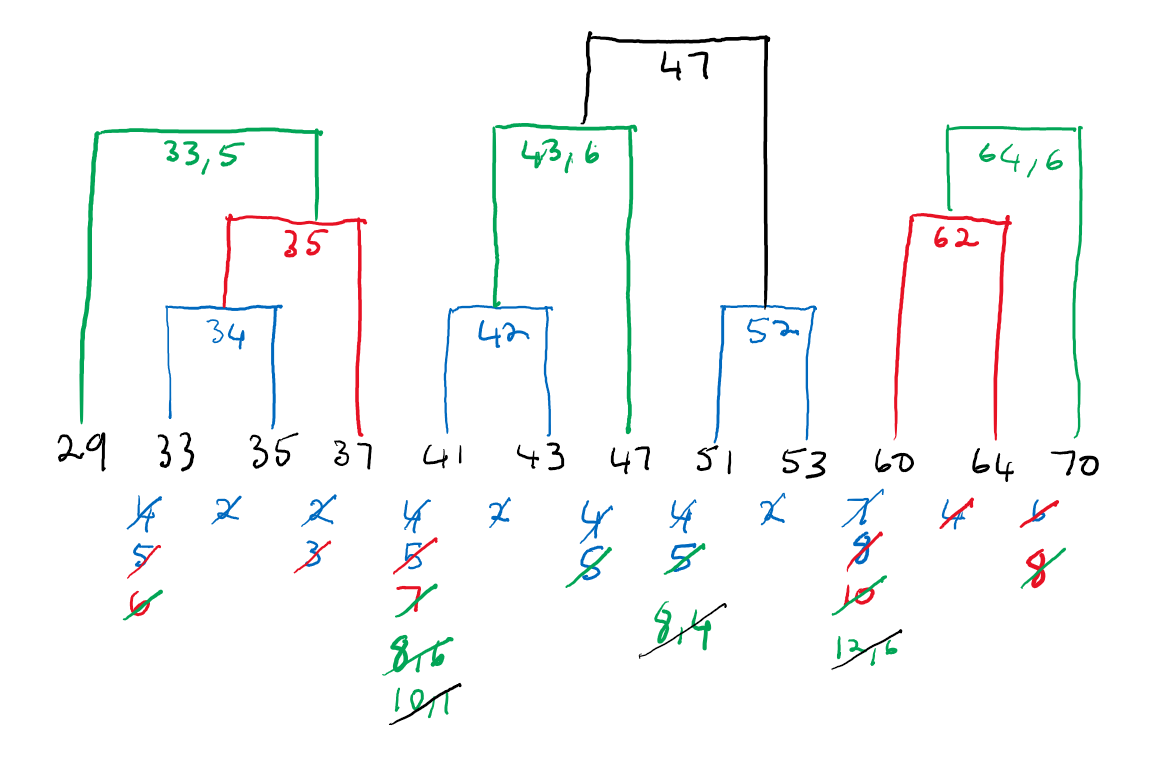


Figure . Clustering by closes centroid.

Your associated working/calculations:

* Clusters were manually calculated by clustering from left to right by the closes distance between cluster centroids.
* Centroids were calculated as the average of the points contained in a cluster *(mental arithmetic and pocket calculator where I doubted myself)*.
* Clustering was stopped after only 3 clusters remained.
* The manual process is error prone, thus as a bare minimum check a dendrogram was plotted with SciPy to visually confirm the clusters match.

data = [[i] for i in [29, 33, 35, 37, 41, 43, 47, 51, 53, 60, 64, 70]]

data\_linkage = scipy.cluster.hierarchy.linkage(data, method="centroid")

dn = scipy.cluster.hierarchy.dendrogram(data\_linkage, count\_sort='descending', labels=data)

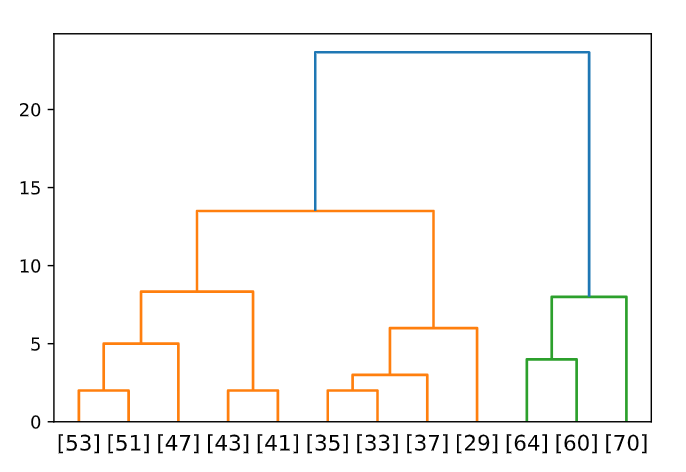


Figure . Closes centroid dendrogram.

1. Your answer and all associated working/calculations to step 2 [4 points]

Your answer goes here:

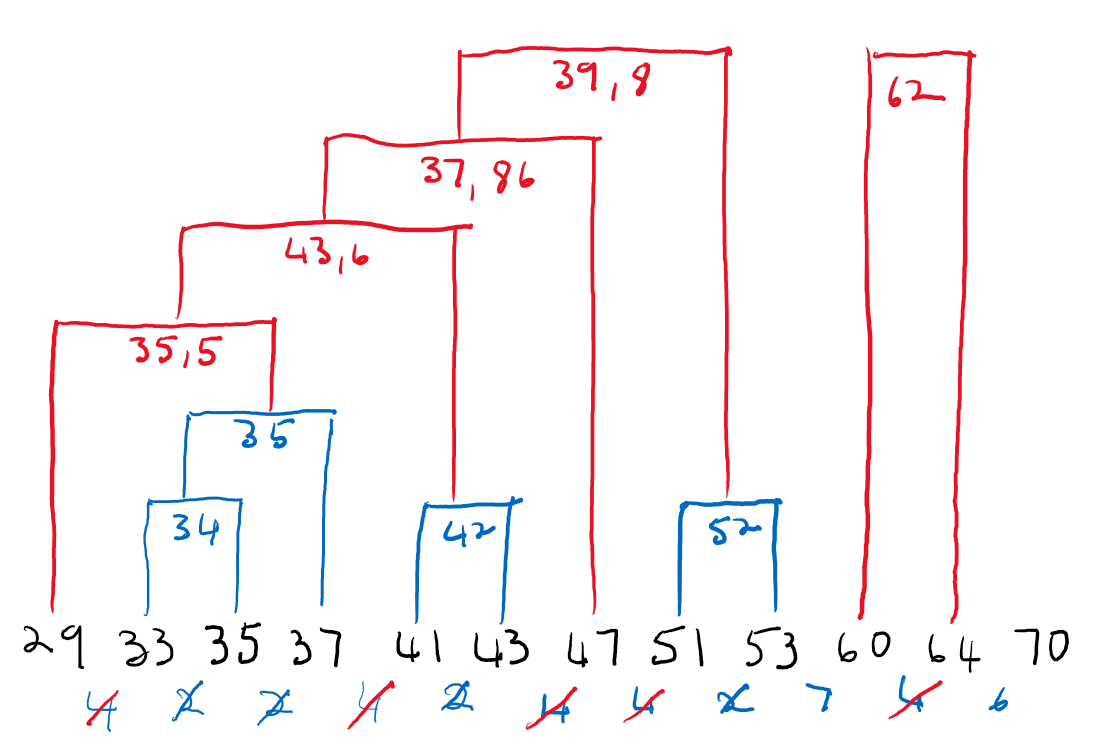


Figure . Clustering by minimum distance between any two points.

Your associated working/calculations:

* Clusters were manually calculated by clustering from left to right by the closes distance between two points in a cluster.
* Centroids were calculated as the average of the points contained in a cluster *(mental arithmetic and pocket calculator where I doubted myself)*.
* Clustering was stopped after only 3 clusters remained.
* The manual process is error prone, thus as a bare minimum check a dendrogram was plotted with SciPy to visually confirm the clusters match.

data = [[i] for i in [29, 33, 35, 37, 41, 43, 47, 51, 53, 60, 64, 70]]

data\_linkage = scipy.cluster.hierarchy.linkage(data, method="single")

d = scipy.cluster.hierarchy.dendrogram(data\_linkage, count\_sort='descending', labels=data)

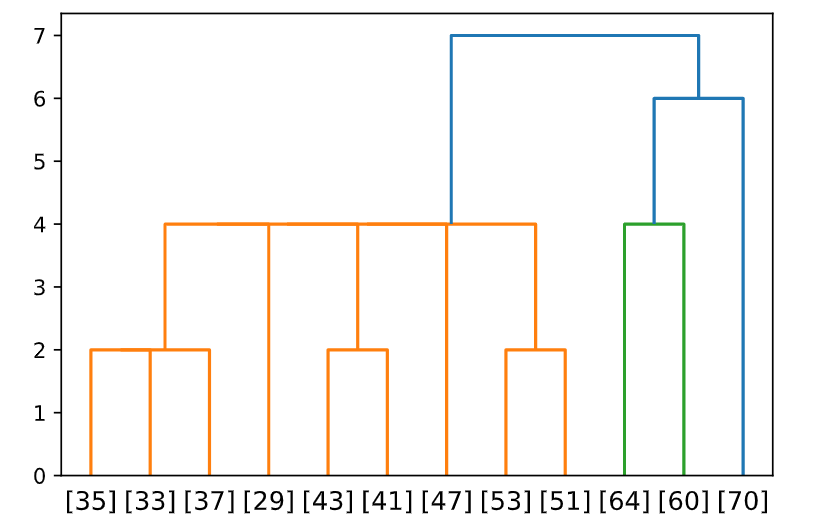


Figure . Closes two points dendrogram.

# Major assignment 1: Part 3 PageRank

In the previous sections I had to continually re-work the manual calculations since there are so many opportunities for mistakes to creep in. By necessity the following class was created to aid in the calculations related to the PageRank algorithm:

class PageRank:

    """

    Implement the PageRank algorithm using matrix manipulation.

    """

    def \_\_init\_\_(self, edges):

        """

        Create a dafault instance of the class.

        Args:

            edges: The edges to create the page graph from.

        """

        self.graph = self.createGraph(edges)

    def createGraph(self, edges):

        """

        Construct a graph from the page edges.

        Args:

            edges: The edges to create the page graph from.

        Returns:

            The graph created from the input edges.

        """

        return gp.Graph.TupleList(edges, directed=True)

    def plotGraph(self, width=300, height=300, margin=15, layout='auto', edge\_curved=True):

        """

        Plot the page graph.

        """

        return gp.plot(self.graph,

            vertex\_label=self.graph.vs["name"],

            layout=layout,

            margin=margin,

            bbox=(0, 0, width, height),

            edge\_curved=edge\_curved)

    def get\_transitionmatrix(self):

        """

        Get the transation matrix of the graph.

        Returns:

            A numpy array containing the transition matrix.

        """

        # initialize the transition matrix with zeros

        node\_count = len(self.graph.vs)

        transition\_matrix = np.zeros([node\_count, node\_count])

        # get the in and out adjacency list

        out\_adj = self.graph.get\_adjlist(mode='out')

        in\_adj = self.graph.get\_adjlist(mode='in')

        # calculate the transition matrix

        for i in range(node\_count):

            for j in range(node\_count):

                # if there is no link from j to i, then mij equals zero

                if i not in out\_adj[j]:

                    transition\_matrix[i][j] = 0

                else: # mij equals 1/k, if there's a link from page j to i. k is the total number of outgoing links from j

                    transition\_matrix[i][j] = 1.0 /  len(out\_adj[j])

        return transition\_matrix

    def get\_initial\_vector(self):

        """

        Get the initial pagerank vector v0.

        """

        node\_count = len(self.graph.vs)

        vector = np.empty(node\_count)

        vector.fill(1/node\_count)

        return vector

    def get\_distribution(self, n, beta=None):

        """

        Get the distribution after n steps.

        Args:

            n: The number of steps to perform.

            beta: The beta value to use to deal with spider traps.

        """

        M = self.get\_transitionmatrix()

        v = self.get\_initial\_vector()

        for i in range(n):

            if beta == None:

                v = np.matmul(M, v)

            else:

                v = beta \* np.matmul(M, v) + ((1-beta) / len(v)) \* np.ones(len(v))

        return v

    def get\_pagerank(self):

        """

        Calculates the PageRank.

        Returns:

            A list with the PageRank values calculated from the input graph.

        """

        step = 1

        converged = False

        rankings = self.get\_initial\_vector()

        while not(converged):

            # get the new rankings

            new\_rankings = self.get\_distribution(step)

            # determine if the rankings have stabilized

            #print(new\_rankings)

            converged = np.sum(np.absolute(rankings -  new\_rankings)) < 1e-5

            rankings = new\_rankings

            step = step + 1

            if step == 500:

                print('\*\*\* COULD NOT CONVERGE! \*\*\*')

                converged = True

        return rankings

1. Your answers to Part 3a (Graph 1.) and all associated working/calculations [4 points]

graph\_edges = [

    ['A', 'B'],

    ['A', 'C'],

    ['A', 'D'],

    ['B', 'A'],

    ['B', 'C'],

    ['C', 'A'],

    ['C', 'B'],

    ['C', 'D'],

    ['D', 'A'],

    ['D', 'C'],

]

pageRank = PageRank(graph\_edges)

pageRank.plotGraph(width=200, height=200, layout='circle', edge\_curved=False)

Transition matrix goes here:

pageRank.get\_transitionmatrix()

|  |
| --- |
| array([[0. , 0.5 , 0.3333, 0.5 ],  [0.3333, 0. , 0.3333, 0. ],  [0.3333, 0.5 , 0. , 0.5 ],  [0.3333, 0. , 0.3333, 0. ]]) |

Initial PageRank vector goes here:

pageRank.get\_initial\_vector()

|  |
| --- |
| array([0.25, 0.25, 0.25, 0.25]) |

First matrix-vector multiplication goes here:

pageRank.get\_distribution(1)

|  |
| --- |
| array([0.3333, 0.1667, 0.3333, 0.1667]) |

Second matrix-vector multiplication goes here:

pageRank.get\_distribution(2)

|  |
| --- |
| array([0.2778, 0.2222, 0.2778, 0.2222]) |

Third matrix-vector multiplication goes here:

pageRank.get\_distribution(3)

|  |
| --- |
| array([0.3148, 0.1852, 0.3148, 0.1852]) |

2. Your answer to Part 3b (Graph 2.) and explanation to justify your answer [2 points]

# create the graph

graph\_edges = [

    ['A', 'B'],

    ['A', 'C'],

    ['A', 'D'],

    ['B', 'A'],

    ['B', 'C'],

    ['B', 'D'],

    ['C', 'A'],

    ['C', 'B'],

    ['C', 'D'],

    ['D', 'A'],

    ['D', 'B'],

    ['D', 'C']

]

pageRank = PageRank(graph\_edges)

pageRank.plotGraph(width=200, height=200, layout='circle', edge\_curved=False)

Your answer goes here:

pageRank.get\_pagerank()

|  |
| --- |
| array([0.25, 0.25, 0.25, 0.25]) |

Your explanation to justify your answer goes here:

Even without performing any calculations one can intuitively see that the final PageRank will be equal to V0 since the graph is unidirectional with every node connecting to all other nodes. Thus every page has an equal probability to be landed on by the hypothetical random surfer.

1. Your answers to Part 3c (Graph 3.) and all associated working/calculations [4 points]

# create the graph

graph\_edges = [

    ['A', 'B'],

    ['A', 'C'],

    ['A', 'D'],

    ['B', 'A'],

    ['B', 'C'],

    ['B', 'F'],

    ['C', 'A'],

    ['C', 'B'],

    ['C', 'D'],

    ['C', 'E'],

    ['D', 'A'],

    ['D', 'C'],

    ['D', 'E']

]

pageRank = PageRank(graph\_edges)

pageRank.plotGraph(width=200, height=200, layout='circle', edge\_curved=False)

Transition matrix goes here:

pageRank.get\_transitionmatrix()

|  |
| --- |
| array([[0. , 0.3333, 0.25 , 0.3333, 0. , 0. ],  [0.3333, 0. , 0.25 , 0. , 0. , 0. ],  [0.3333, 0.3333, 0. , 0.3333, 0. , 0. ],  [0.3333, 0. , 0.25 , 0. , 0. , 0. ],  [0. , 0.3333, 0. , 0. , 0. , 0. ],  [0. , 0. , 0.25 , 0.3333, 0. , 0. ]]) |

Initial PageRank vector goes here:

pageRank.get\_initial\_vector()

|  |
| --- |
| array([0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667]) |

First matrix-vector multiplication goes here:

pageRank.get\_distribution(n=1, beta=0.8)

|  |
| --- |
| array([0.1556, 0.1111, 0.1667, 0.1111, 0.0778, 0.1111]) |

Second matrix-vector multiplication goes here:

pageRank.get\_distribution(n=2, beta=0.8)

|  |
| --- |
| array([0.1259, 0.1081, 0.1341, 0.1081, 0.063 , 0.0963]) |

Third matrix-vector multiplication goes here:

pageRank.get\_distribution(n=3, beta=0.8)

|  |
| --- |
| array([0.1178, 0.0937, 0.1246, 0.0937, 0.0622, 0.089 ]) |