CS 533 Project 3

Freddie Awuah-Gyasi `fawuahgyasi5@unm.edu`
Jaime Gould `egould@unm.edu`
Lasair Servilla `lservilla@unm.edu`
Qinghong Shao `qinghongshao@unm.edu`

due: April 21st 2025

1 Functions and Pseudocode

When we initially began to implement the Hopkins Statistic in ***Project 2***, we noticed that the algorithm could easily be inefficient when searching for the nearest neighbor in a dataset to a given point:

Algorithm 1 Hopkins Statistic

```
 1: function HopkinsStat($D, m$)
 2:     $D_m \leftarrow$ RandomSubset($D,\ m$)                                   ▷ $\mathcal{O}(m)$
 3:     $R_m \leftarrow$ RandomPoints($m,\ x_{range},\ y_{range}$)                  ▷ $\mathcal{O}(m)$
 4:     $r \leftarrow \{\ |R_i -$ NearestNeighbor$(R_i,\ D)|\ \}\ \forall\ R_i \in R_m$     ▷ $\mathcal{O}(m) \times \mathcal{O}($NearestNeighbor$)$
 5:     $p \leftarrow \{\ |D_i -$ NearestNeighbor$(D_i,\ D)|\ \}\ \forall\ D_i \in D_m$     ▷ $\mathcal{O}(m) \times \mathcal{O}($NearestNeighbor$)$
 6:     $r^2_{\text{sum}} \leftarrow \sum_{i=1}^{m} r_i^2\ \ \forall\ \ r_i \in r$                      ▷ $\mathcal{O}(m)$
 7:     $p^2_{\text{sum}} \leftarrow \sum_{i=1}^{m} p_i^2\ \ \forall\ \ p_i \in p$                      ▷ $\mathcal{O}(m)$
 8:     $H \leftarrow \frac{r^2_{\text{sum}}}{p^2_{sum} + r^2_{\text{sum}}}$                                   ▷ $\mathcal{O}(1)$
 9:     return $H$
10: end function
```

If we simply iterate over all points in a given dataset D (where the size of the dataset $|D| = n$) for each point $r \in R_m$ and $p \in D_m$ ($|R_m| = |D_m| = m$) in order to determine which point is closest, we end up with a complexity of $\mathcal{O}(m) \times \mathcal{O}(\text{NearestNeighbor}) = \mathcal{O}(m \times n)$ for a single pass of the algorithm alone. When we then compute the Hopkins Statistic for a bootstrap size of B, the complexity of this algorithm increases to $\mathcal{O}(B \times m \times n)$. This can be **very** slow depending on the size of B, n, and m.

In order to improve efficiency of our algorithm we created two classes, BoundingBox and Grid. BoundingBox merely contains the maximum and minimum of the x and y ranges over which the points can exist, while Grid organizes the points of a given dataset into a grid structure with bins of size *partition_size* (see Figure 1, left).

```
1: class BoundingBox
2:     $x_{min}$ : Float                          ▷ Our datasets have $x_{min} = -10000$
3:     $x_{max}$ : Float                                          ▷ $x_{max} = 10000$
4:     $y_{min}$ : Float                                          ▷ $y_{min} = -10000$
5:     $y_{max}$ : Float                                          ▷ $y_{max} = 10000$
6: end class
```

```
class GRID
    function INIT($points$ : LIST, $bbox$ : BOUNDINGBOX, $partition\_size$ : FLOAT)
        $x_{min} \leftarrow bbox.x_{min} - partition\_size$                                  ▷ $\mathcal{O}(1)$
        $y_{min} \leftarrow bbox.y_{min} - partition\_size$                                  ▷ $\mathcal{O}(1)$
        $i_{x_{max}} \leftarrow$ FLOOR$((bbox.x_{max} - x_{min})/partition\_size)$           ▷ $\mathcal{O}(1)$
        $i_{y_{max}} \leftarrow$ FLOOR$((bbox.y_{max} - y_{min})/partition\_size)$           ▷ $\mathcal{O}(1)$
        $grid \leftarrow$ array of empty arrays, where $||grid|| = i_{x_{max}} \times i_{y_{max}}$   ▷ $\mathcal{O}(i_{x_{max}} \times i_{y_{max}}) = \mathcal{O}(1)$
        for $p \in points$ do
            $x_{index} \leftarrow$ FLOOR$((p.x - x_{min})/partition\_size)$
            $y_{index} \leftarrow$ FLOOR$((p.y - y_{min})/partition\_size)$
            $grid[x_{index}][y_{index}] \leftarrow p$
        end for                                                                               ▷ $\mathcal{O}(n)$
    end function

    function GETNEIGHBORS($p, neighborhood$)
        $x_{idx} \leftarrow$ MAX(MIN(FLOOR$((p.x - x_{min})/partition\_size)$, $i_{x_{max}}$), 0)   ▷ $\mathcal{O}(1)$
        $y_{idx} \leftarrow$ MAX(MIN(FLOOR$((p.y - y_{min})/partition\_size)$, $i_{y_{max}}$), 0)   ▷ $\mathcal{O}(1)$
        $neighbors \leftarrow grid[x_{idx}][y_{idx}]$                                        ▷ $\mathcal{O}(1)$

        for MAX$(x_{idx} - neighborhood,\ 0) \leq i \leq$ MIN$(x_{idx} + neighborhood, i_{x_{max}})$ do
            for MAX$(y_{idx} - neighborhood,\ 0) \leq j \leq$ MIN$(y_{idx} + neighborhood, i_{y_{max}})$ do
                $neighbors \leftarrow neighbors + grid[i][j]$
            end for
        end for                                              ▷ $\leq \mathcal{O}(neighborhood \times neighborhood)$
        return $neighbors$
    end function

    function NEARESTNEIGHBOR($p$)
        $neighbors \leftarrow$ GETNEIGHBORS($p$, 0)                                           ▷ $\mathcal{O}(1)$
        if $|neighbors| < 2$ then $neighbors \leftarrow$ GETNEIGHBORS($p$, 1)                ▷ $\mathcal{O}(1 \times 1) = \mathcal{O}(1)$
        end if
        if $|neighbors| < 2$ then $neighbors \leftarrow$ GETNEIGHBORS($p$, 2)       ▷ $\mathcal{O}(2 \times 2) = \mathcal{O}(4) = \mathcal{O}(1)$
        end if
        if $|neighbors| < 2$ then $neighbors \leftarrow$ GETNEIGHBORS($p$, 3)       ▷ $\mathcal{O}(3 \times 3) = \mathcal{O}(9) = \mathcal{O}(1)$
        end if
        if $|neighbors| < 2$ then $neighbors \leftarrow$ GETNEIGHBORS($p$, 4)      ▷ $\mathcal{O}(4 \times 4) = \mathcal{O}(16) = \mathcal{O}(1)$
        end if
        if $|neighbors| < 2$ then $neighbors \leftarrow points$                              ▷ $\mathcal{O}(1)$
        end if

        $nearest \leftarrow \emptyset$
        $dist \leftarrow \infty$
        for $v \in neighbors$ do
            $new\_dist \leftarrow$ EUCLIDEANDISTANCE($v$, $p$)
            if $new\_dist < dist$ then
                $nearest \leftarrow v$
                $dist \leftarrow new\_dist$
            end if
        end for                                                                   ▷ $\mathcal{O}(|neighbors|)$
        return $nearest$
    end function
end class
```

2 Time Complexity Analysis

Grid is initialized once for a given dataset D ($|D| = n$) before the Hopkins Statistic is computed. Upon initialization, a grid data structure is created (Grid, line 7) and every point $p \in D$ is placed into it according to its x and y parameters (lines 8-12). Setting up Grid has initial time complexity $\mathcal{O}(n)$.

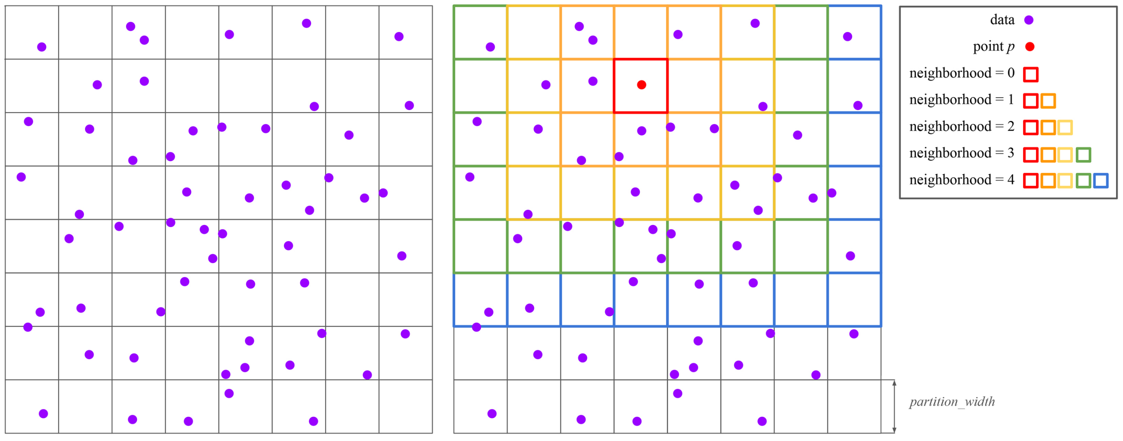

Figure 1: Left: an example of 2D data points in our Grid data structure. Right: the neighborhood of a given point p (in red) at different neighborhood sizes.

Once Grid has been initialized, we then use its NearestNeighbor function to find the neighbor nearest to a given point p in our HopkinsStatistic algorithm. In the best case scenario of this function, there are points in the closest neighborhood of p. However, if there are less than 2 points[1] in the list of neighbors we retrieve, we search for the next largest neighborhood = 1 (see Figure 1, right), and so on up to a neighborhood of 4. If the number of points is still $|neighbors| < 2$, we set our neighbors to be equal to the list of total points in the dataset D. We then compare the euclidean distance of our point p to each neighbor in $neighbors$, resulting in complexity of NearestNeighbor depending on the size of $neighbors$, $\mathcal{O}(|neighbors|)$.

In the best case scenario, a smaller neighborhood yields a number of points $|neighbors| = 2$, giving complexity $\mathcal{O}(2) = \mathcal{O}(1)$. If the smaller neighborhood fails to retrieve at least two points or happens to contain all points in the dataset due to the way the data is distributed in D, the euclidean distance is computed for all points in D, resulting in $\mathcal{O}(n)$. Therefore, the complexity of our algorithm for a single run of the the Hopkins Statistic is:

$$\mathcal{O}(\textsc{HopkinsStatistic}) = \mathcal{O}(m) \times \mathcal{O}(\textsc{NearestNeighbor}) = \mathcal{O}(m) \times \mathcal{O}(|neighbors|)$$

$$\mathcal{O}(m) \times \mathcal{O}(1) \leq \mathcal{O}(\textsc{HopkinsStatistic}) \leq \mathcal{O}(m) \times \mathcal{O}(n)$$
$$\mathcal{O}(m) \leq \mathcal{O}(\textsc{HopkinsStatistic}) \leq \mathcal{O}(m \times n)$$

The bootstrapping portion of our code, Bootstrap, which computes the HopkinsStatistic B times then has complexity $\mathcal{O}(\textsc{Bootstrap}) = \mathcal{O}(B) \times \mathcal{O}(\textsc{HopkinsStatistic})$:

$$\mathcal{O}(B) \times \mathcal{O}(m) \leq \mathcal{O}(\textsc{Bootstrap}) \leq \mathcal{O}(B) \times \mathcal{O}(m \times n)$$
$$\mathcal{O}(B \times m) \leq \mathcal{O}(\textsc{Bootstrap}) \leq \mathcal{O}(B \times m \times n)$$

[1] $D_m \subset D$, and the nearest neighbor of a point $p \in D_m$ in D should not be itself. Therefore we check for a set of neighbors of at least 2 so that *neighbors* does not include only p.

3 Experiments for Testing Complexity

In order to determine whether our time complexity analysis is correct, we have designed experiments in which we alter each of our variables that we have hypothesized mainly impact the complexity: B, m, and n (n is fixed at 5000 for each dataset in ***Project 2***. We will create random subsets of varying size n from our original datasets in order to simulate different dataset sizes).

We utilize linear regression in order to test whether our hypotheses are correct. We can format our Cost function C in the form of a linear equation:

$$r(x) = \beta_\circ + \beta_1 \cdot x$$

$$\beta_\circ + \beta_1(B \times m) \leq C(B, m, n) \leq \beta_\circ + \beta_1(B \times m \times n)$$

Where $\beta_\circ$ is the constant time the program takes when $B = m = n = 0$, β_1 is the rate at which the time increases as B, m, and n increase (at high values of B, m, and n when $\mathcal{O}(B \times m \times n)$ has the greatest impact), and y is the cost function, or the total time the program takes to run.

3.a Hypotheses

Null Hypothesis. *($H_\circ$) There is no relationship between B, m, n and time; therefore $\beta_1 = 0$.*

Alternative Hypothesis. *(H_1) There is a linear relationship between B, m, n and time; parameter β_1 is some constant value.*

We can use the F-Test to determine if our expected value $E(\beta_1) = 0$ under the null hypothesis falls within the 95% confidence interval of our computed β_1 value, where:

3.b Experiments

We will run our code while changing our parameter B at intervals of 1000 ($1000 \leq B \leq 11000$), our parameter m at intervals of 100 ($100 \leq m \leq 1100$), and our parameter n at intervals of 300 ($2000 \leq n \leq 5000$) for each of our 4 datasets (D_1, D_2, D_3, D_4) from ***Project 2***. This will result in $10 \times 10 \times 10 = 1000$ runs for each dataset.

We will record the time it takes for each individual run. We expect to see the time increasing as each of our parameters increases according to our time complexity analysis between lower bound $\mathcal{O}(B \times m)$ and upper bound $\mathcal{O}(B \times m \times n)$, so we will compute the linear regression fit and estimates for $\hat{\beta}_\circ$ and $\hat{\beta}_1$ and their 95% confidence intervals for each dataset and plot the time of our runs as a function of $x = B \times m \times n$.