# Learning Analytics

Dr. Bowen Hui
Computer Science
University of British Columbia Okanagan

# Prototype-Based Clustering

- Partitions data points into clusters
- Each cluster has a prototype which serves as the representative point
- Most popular methods:
  - K-means
    - Defines prototype by a centroid (based on a group of points)
    - Typically used on continuous n-dimensional data
  - K-medoid
    - Defines prototype by a medoid (an actual point)
    - Applicable to different types of data

# K-Means Clustering

- Partitional clustering method that finds k clusters
  - k is given
  - Each point is associated with one centroid
- General algorithm:
  - Select k points as initial centroids
  - Repeat
    - Form k clusters by assigning each point to its closest centroid
    - Update centroid of each cluster
  - Until centroids do not change
- Key operations:
  - Compute point-to-point distance
  - Update centroid

# K-means Demo

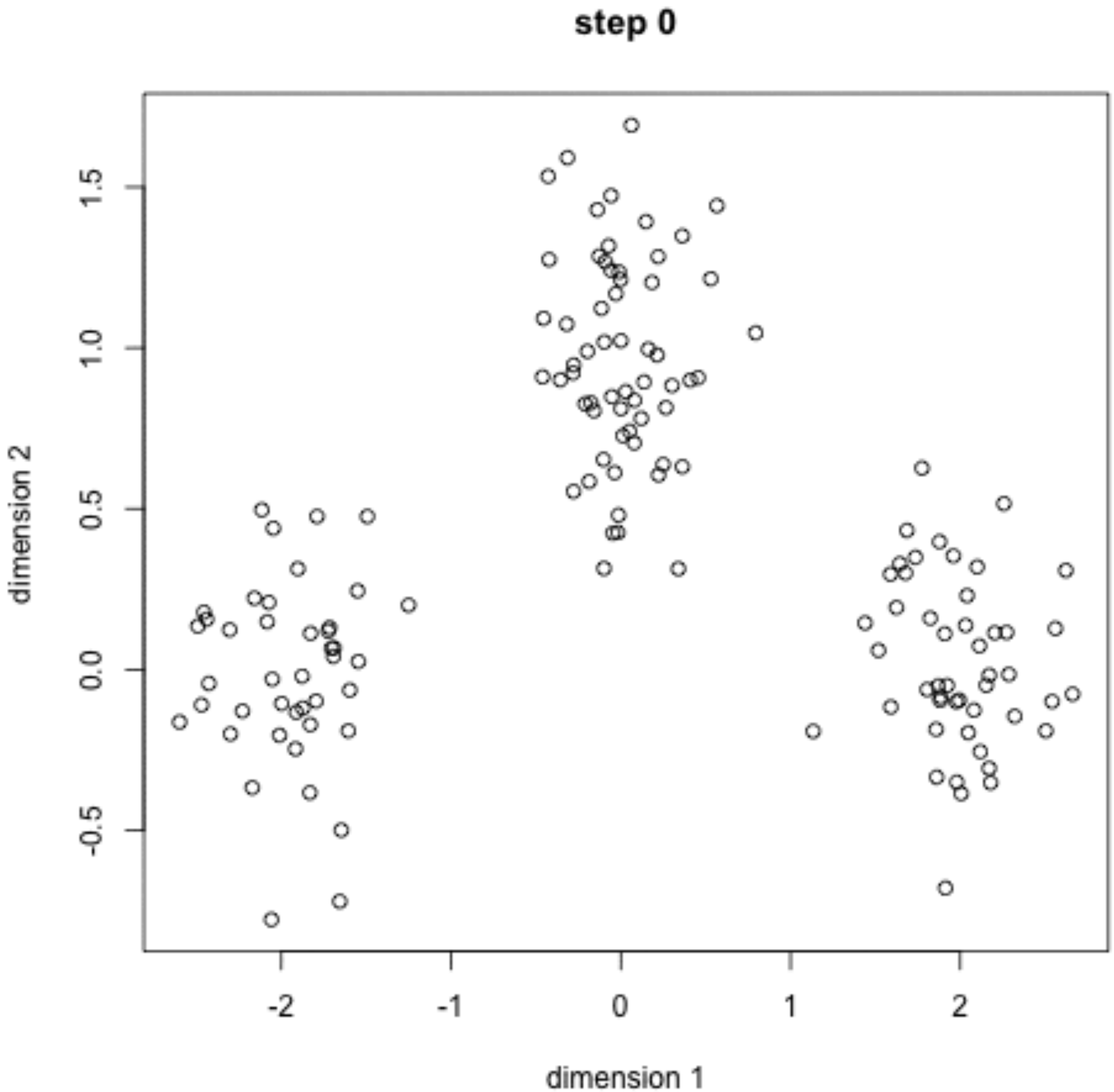



Image taken from towardsdatascience.com

# Calculating Distance Between Points

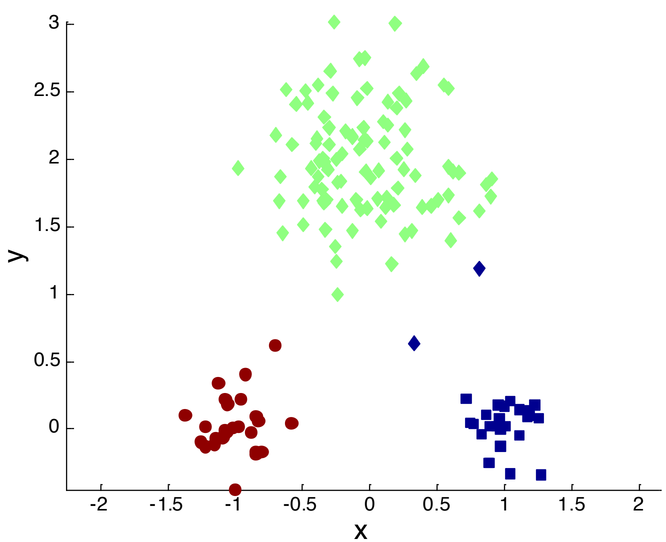
- 2D space:
  - Euclidean distance (L2 norm)
  - Also use Manhattan distance (L1 norm)
    - Sum of the magnitude of vector
    - $||x||_1 = \sum_{i=1}^{n} |xi|$
- For documents:
  - Cosine similarity (vector representation)
  - Jaccard measure (set theory)

# Updating a Cluster's Centroid

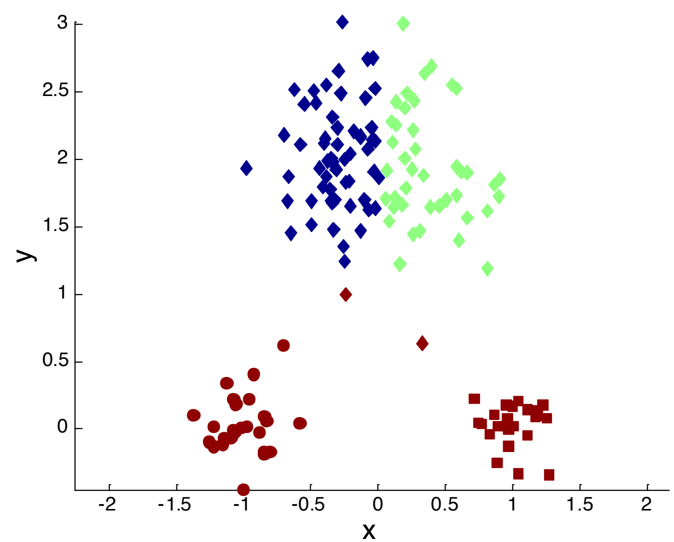
- Goal is typically expressed by an objective function that depends on proximities of points to one another or to cluster centroids
- Using the mean:
  - Compute mean of points in the cluster
  - Minimizes the sum of the squared error (SSE) in the clustering
- K-means will converge for common similarity measures
  - i.e., Centroids will not change

# SSE as the Objective Function

- A smaller SSE means the centroids of the clustering is a better representation of the points in the clusters obtained
- Given 2 clusterings, we prefer the one with a smaller SSE




**Optimal Clustering**




**Sub-optimal Clustering**

# SSE as the Objective Function

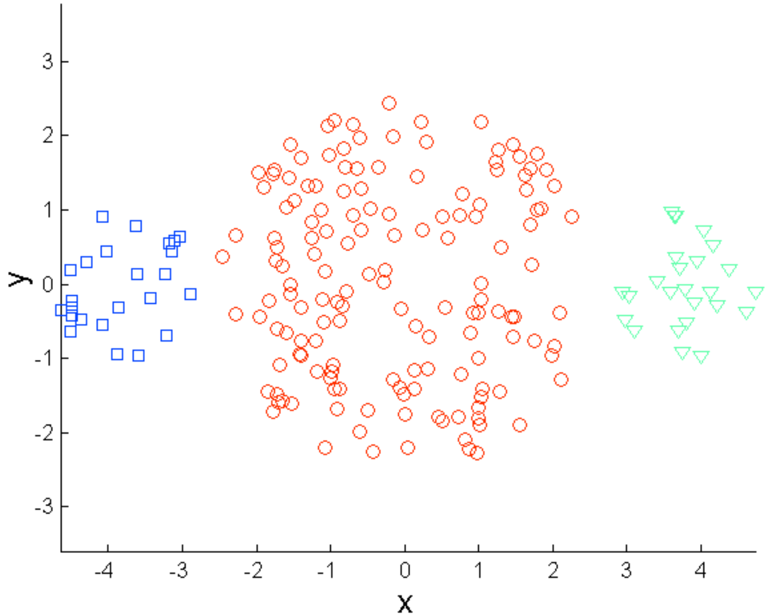
- A smaller SSE means the centroids of the clustering is a better representation of the points in the clusters obtained
- Given 2 clusterings, we prefer the one with a smaller SSE

- Definition: SSE = $\sum_{i=1}^{k} \sum_{x \in C_i} dist(m_i, x)^2$
  - Compute squared error between centroid (mean) and every point in cluster
  - Add up squared error of all the clusters

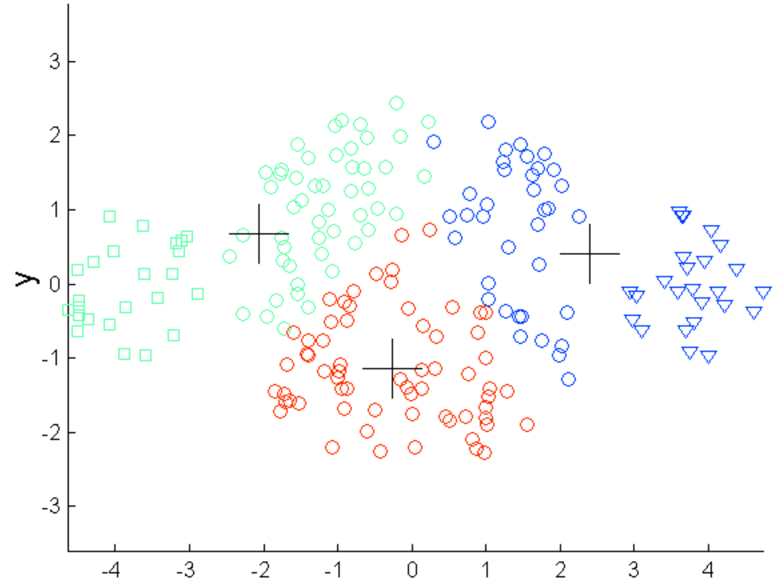
# Limitations

- Difficulty when clusters are of differing:
  - Sizes
  - Densities
  - Non-globular shapes
- Difficulty when data have outliers
- One solution:
  - Use many clusters
  - Find parts of clusters but need to put together

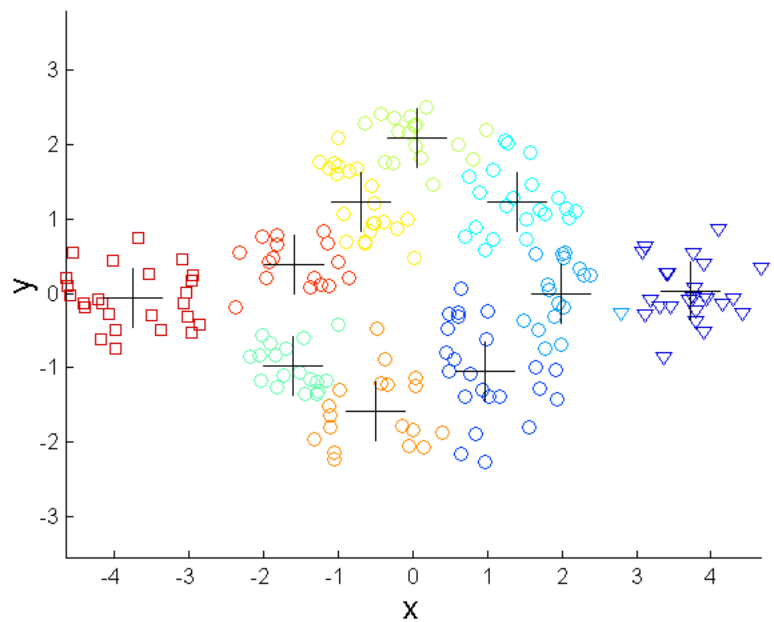
# Differing Sizes




**Original Points**

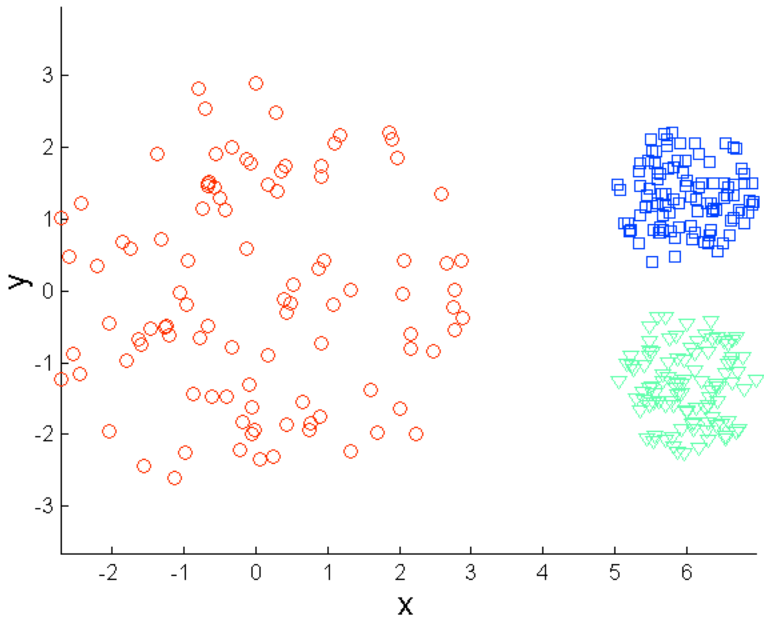



**K-means (3 Clusters)**

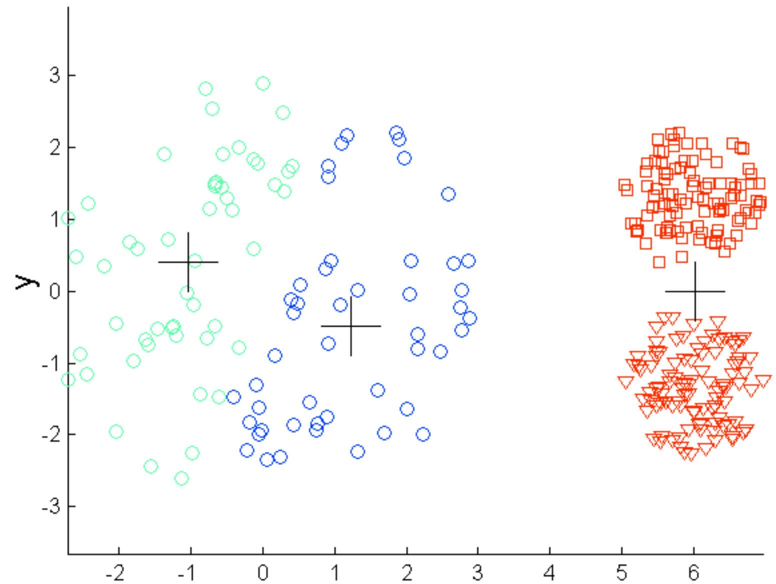



**K-means (10 Clusters)**

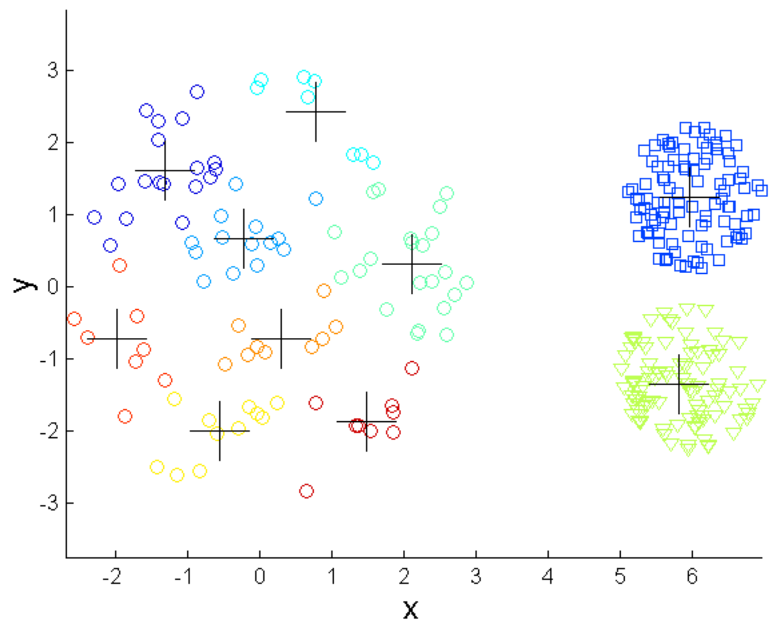
# Differing Densities




**Original Points**

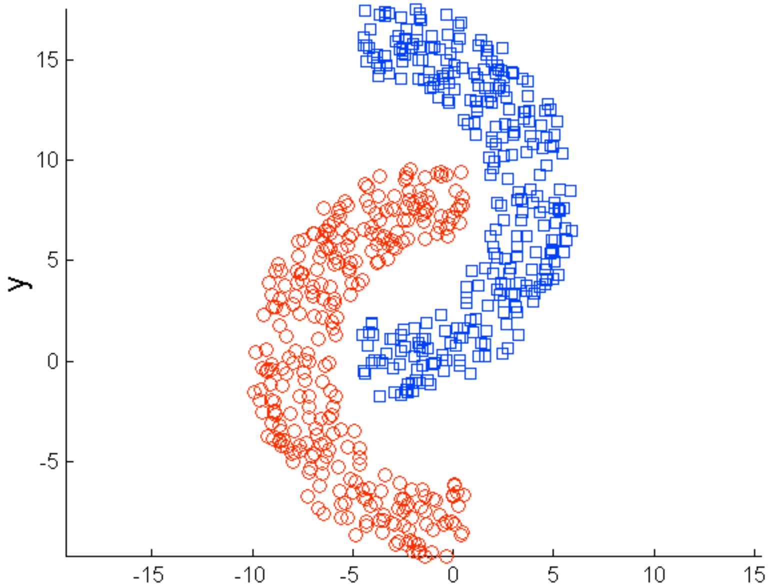



**K-means (3 Clusters)**

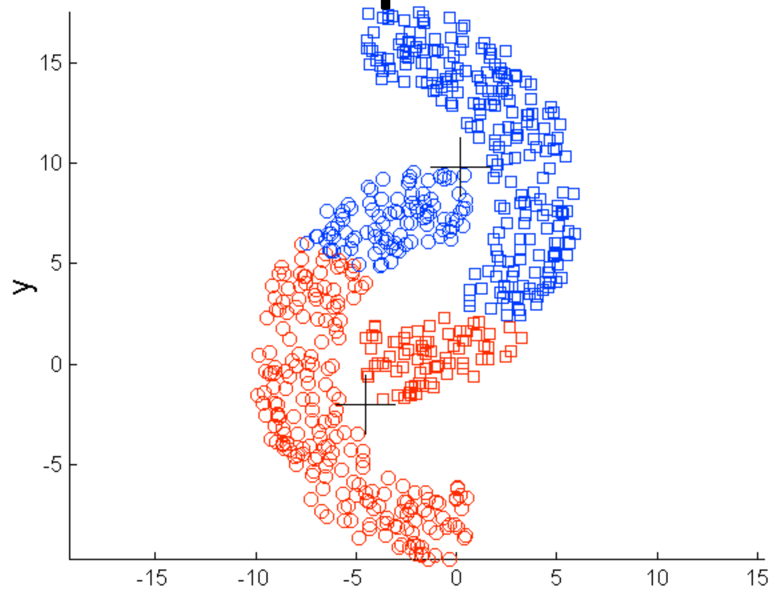



**K-means (10 Clusters)**

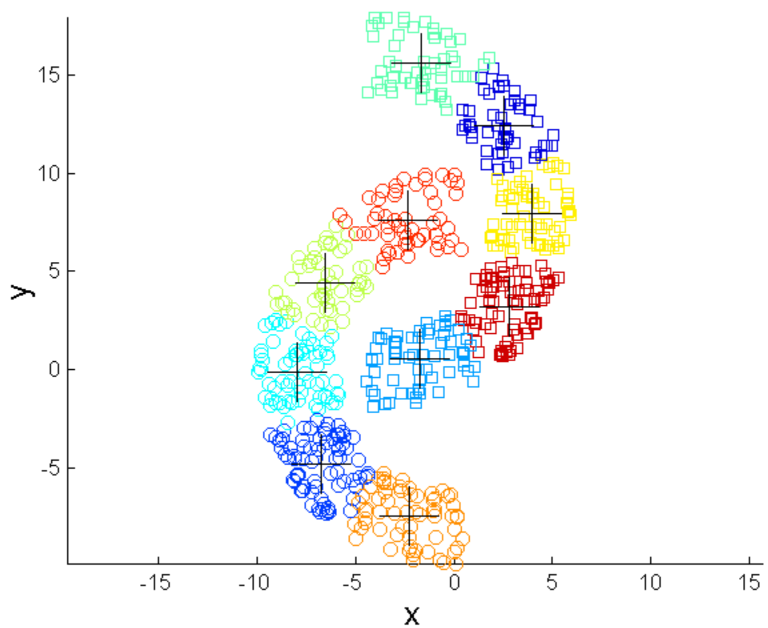
# Non-Globular Shapes




Original Points




**K-means (2 Clusters)**



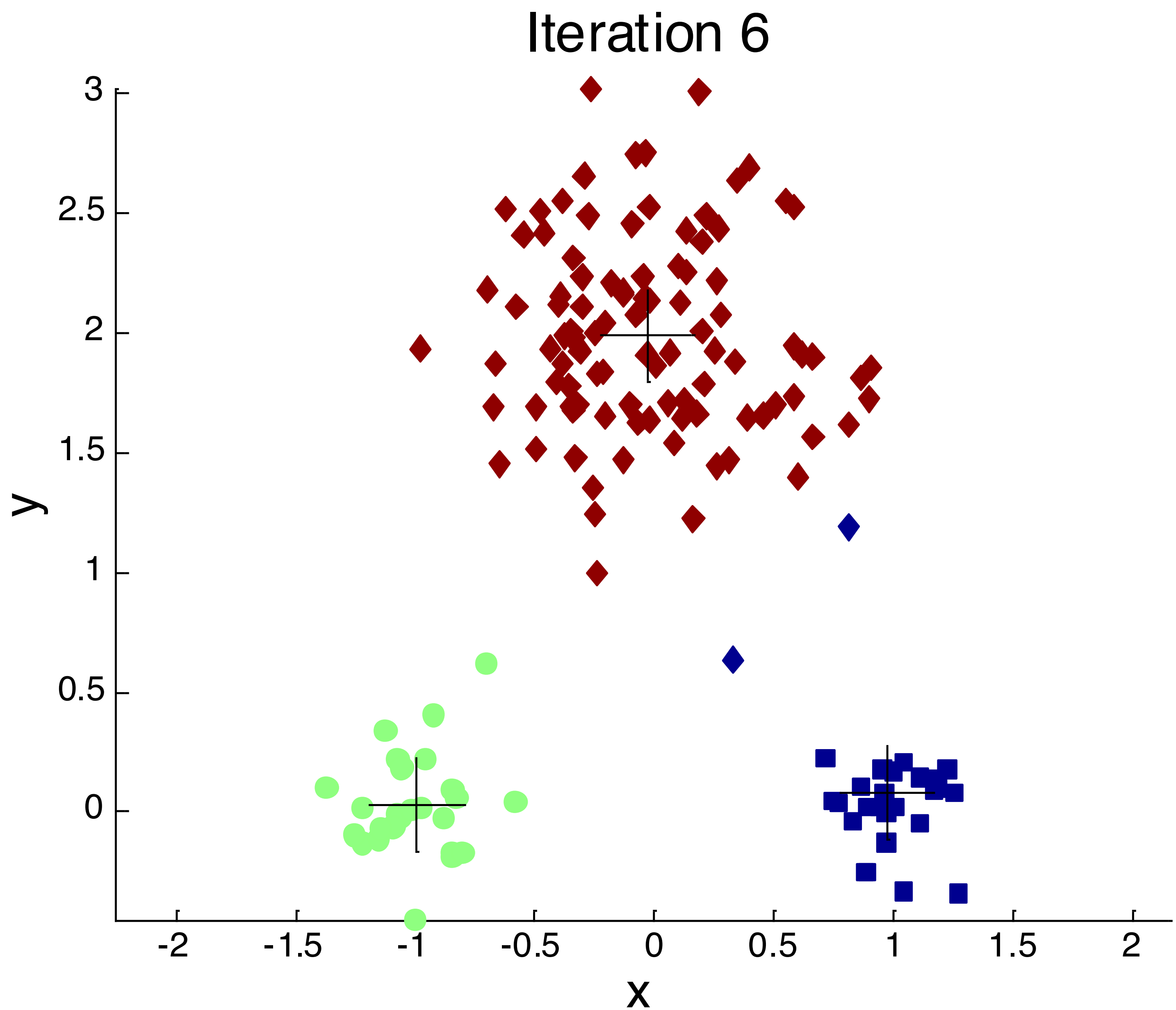

**K-means (10 Clusters)**

# Choosing Initial Centroids

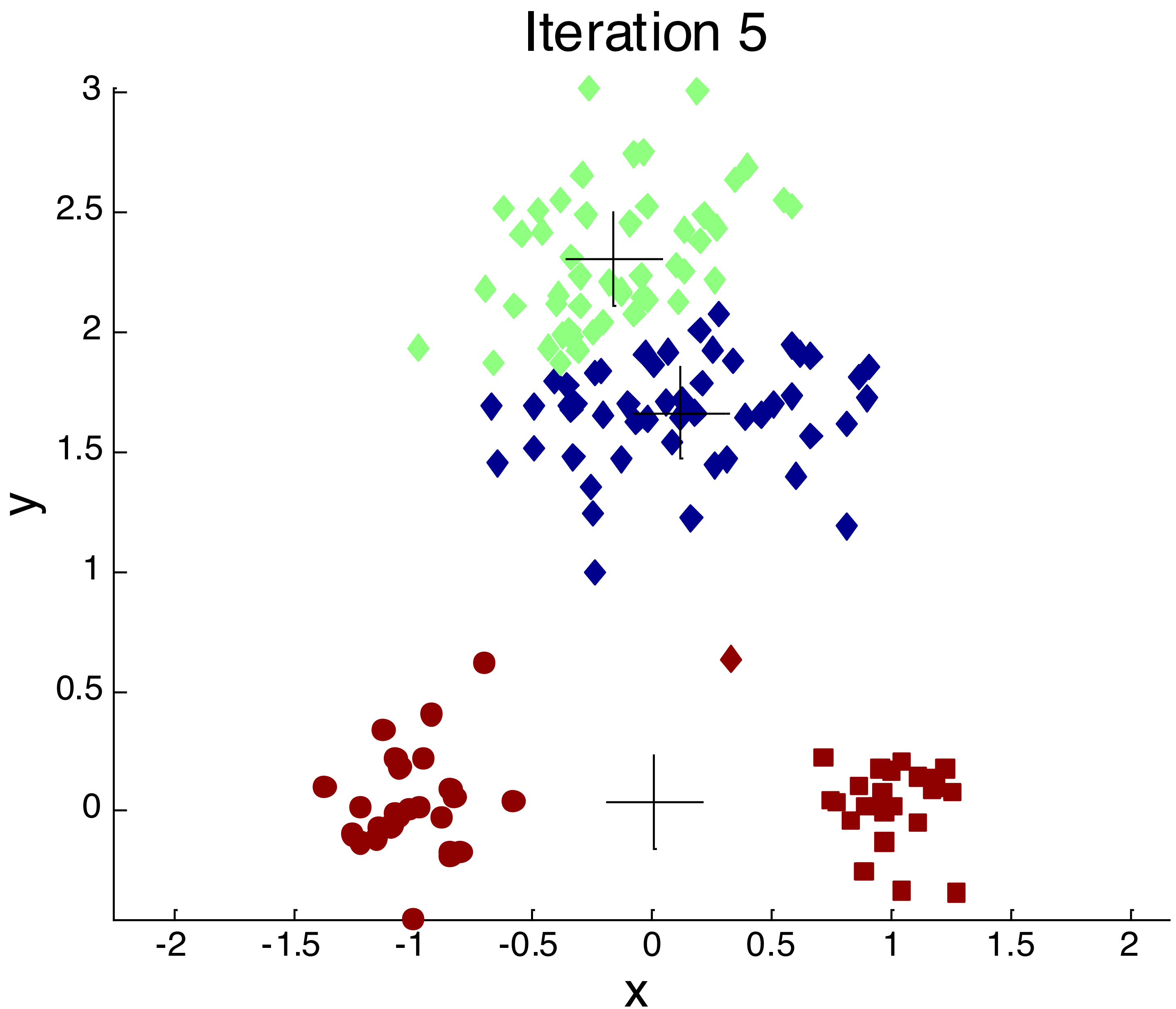
- Often done at random
  - Clusters produced vary from one run to another
  - Different optimal solutions exist
  - Often result in poor initial centroids

K-means typically converges to local minimum

# Importance of Choosing Initial Centroids

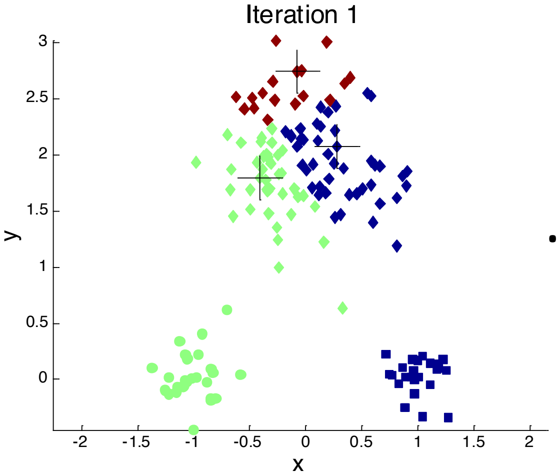


# Importance of Choosing Initial Centroids …

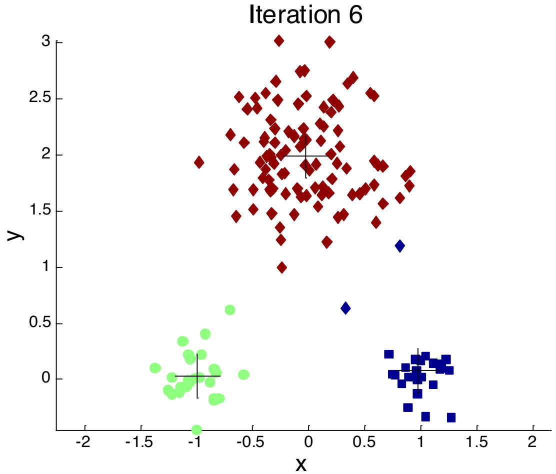


# Comparison between Two Initial Choices

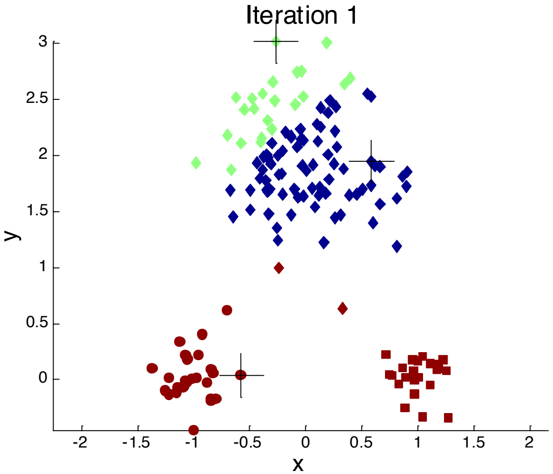
Option 1



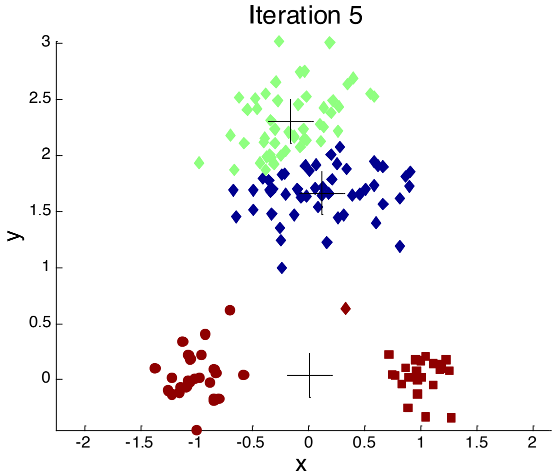

. . .




Option 2




. . .

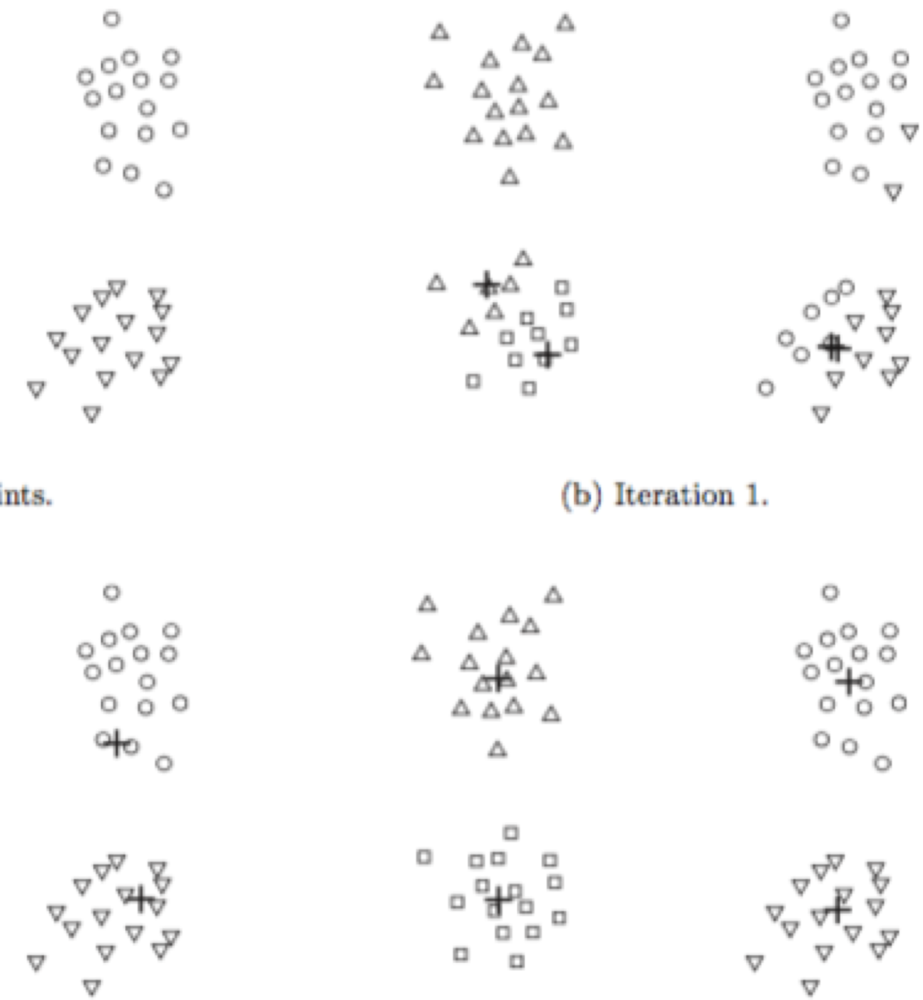


# Choosing Initial Centroids

- Often done at random
  - Clusters produced vary from one run to another
  - Different optimal solutions exist
  - Often result in poor initial centroids

- Solution 1: use multiple runs
  - Choose smallest SSE of the clusterings
  - Effectiveness depends on data set and k

# When Data Set and k Match



(a) Initial points.

(b) Iteration 1.

(c) Iteration 2.

(d) Iteration 3.

**Figure 7.6.** Two pairs of clusters with a pair of initial centroids within each pair of clusters.

# When Data Set and k Don't Match

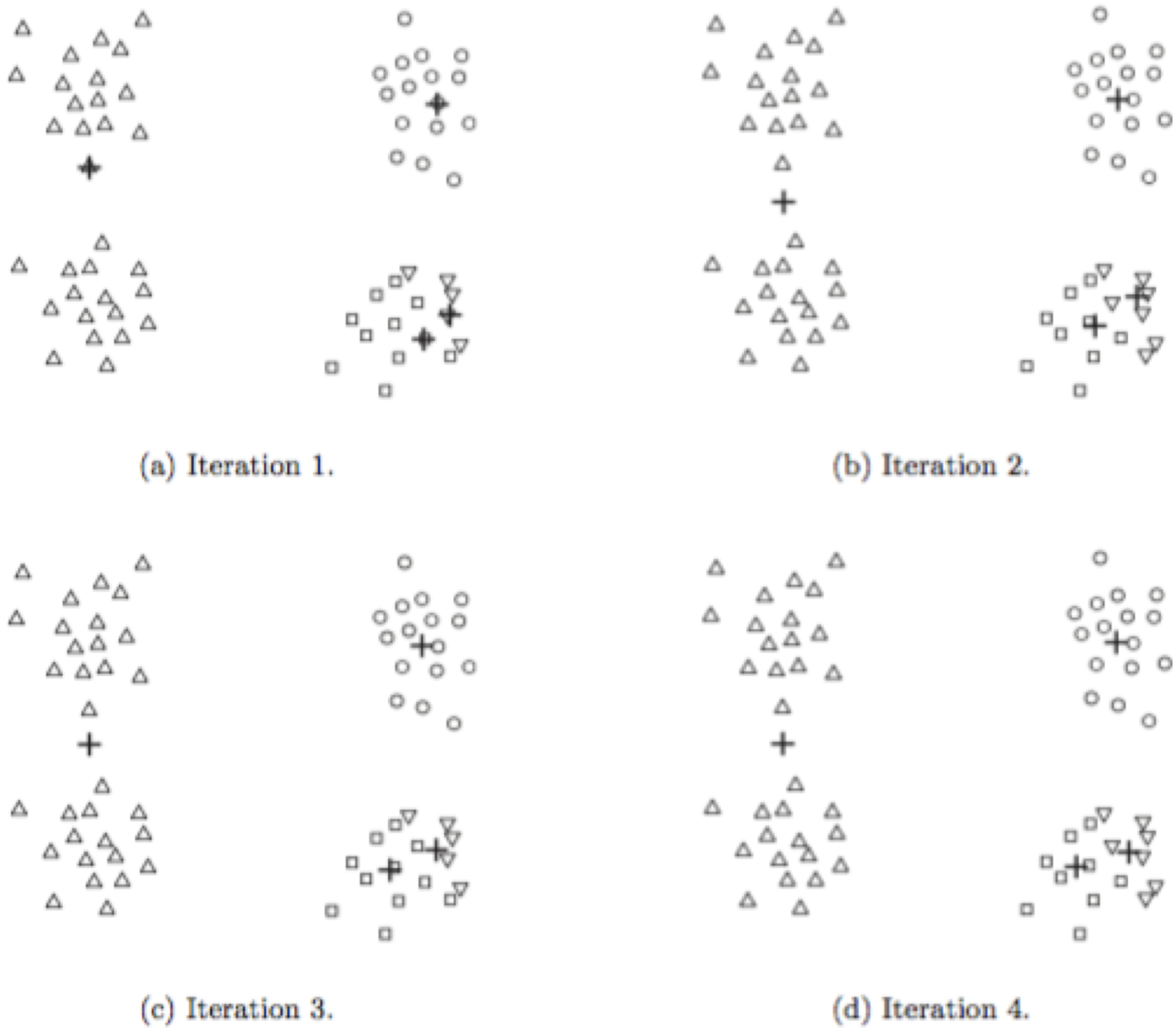



**Figure 7.7.** Two pairs of clusters with more or fewer than two initial centroids within a pair of clusters.

# Solutions to Initial Centroid Problem

- Multiple runs using random initials
- Sample and use hierarchical clustering to set initial centroids
  - Generally works well if sample and k are small
- Select more than k initial centroids then select a subset of most widely separated ones to use
  - Bad if selected an outlier
- Postprocessing
- Generate a larger number of clusters then perform hierarchical clustering
- Other variations: k-means++ and bisecting k-means

# Preliminary Case Study: COSC 111

- Understand different approaches to programming problem
- Data considered
  - Java programming assignments to implement (single player) Memory card game
    - Limited to 8 pairs of cards
    - Displayed on 4x4 board
  - Hands-on instructions with grading criteria
  - Sample output
  - Methods expected

# Sample Output

```
----------------START----------------
Remaining cards from the game:
   1 2 3 4
1  x x x x
2  x x x x
3  x x x x
4  x x x x
First card is (specify row,column):
1
1
   1 2 3 4
1  $ x x x
2  x x x x
3  x x x x
4  x x x x
Second card is (specify row,column):
1
2
   1 2 3 4
1  $ ? x x
2  x x x x
3  x x x x
4  x x x x
```

```
-------------------------------------
Remaining cards from the game:
   1 2 3 4
1  x x x x
2  x x x x
3  x x x x
4  x x x x
First card is (specify row,column):
1
3
   1 2 3 4
1  x x % x
2  x x x x
3  x x x x
4  x x x x
Second card is (specify row,column):
1
4
   1 2 3 4
1  x x % ?
2  x x x x
3  x x x x
4  x x x x
```

# Sample Output (cont.)

```
------------------------------------
Remaining cards from the game:
   1 2 3 4
1  x x x x
2  x x x x
3  x x x x
4  x x x x
First card is (specify row,column):
1
2
   1 2 3 4
1  x ? x x
2  x x x x
3  x x x x
4  x x x x
Second card is (specify row,column):
1
4
   1 2 3 4
1  x ? x ?
2  x x x x
3  x x x x
4  x x x x
You found a match!
```

```
------------------------------------
Remaining cards from the game:
   1 2 3 4
1  x   x
2  x x x x
3  x x x x
4  x x x x
First card is (specify row,column):
1
1
   1 2 3 4
1  $   x
2  x x x x
3  x x x x
4  x x x x
Second card is (specify row,column):
3
4
   1 2 3 4
1  $   x
2  x x x x
3  x x x $
4  x x x x
You found a match!
```

# Sample Output (cont.)

. . .

```
Remaining cards from the game:
   1 2 3 4
1
2
3      x
4  x
First card is (specify row,column):
4
1
   1 2 3 4
1
2
3      x
4  *
Second card is (specify row,column):
3
3
   1 2 3 4
1
2
3      *
4  *
You found a match!
-------------------------------------
```

# Program Structure

- Basic algorithm
  - Shuffle cards and lay out 4x4 board
  - While not all pairs have been matched
    - Call showBoard() with appropriate whitespace or card
    - Get two cards from user and open them on board with openCard()
    - Check if there’s a match and update variables as needed

# Solution's Code Structure

```
public class Memory
{
  public static void main( String[] args ) { }
  public static void showBoard( … ) { }
  public static void openCard( … ) { }
}
```

# main()

```
public static void main( String[] args )
{
  // initialize game vars

  // array to track cards that have been generated already
  for( int i=0; i<pairs.length; i++ ) { }

  // array to track what has been matched by user or not
  // initially nothing has been matched
  for( int i=0; i<MAX; i++ ) {
    for( int j=0; j<MAX; j++ ) {
  } }

  // randomly generate a 4 x 4 board for game
  for( int i=0; i<MAX; i++ )
  {
    for( int j=0; j<MAX; j++ )
    {
      while( pairs[ idx ] >= 2 ) { }
    }
  }

  System.out.println( "---------------START---------------" );
  while( numSolved < (MAX*MAX) )
  {
    showBoard( board, matched );

    openCard( board, matched, row1, col1, -1, -1 );
    openCard( board, matched, row1, col1, row2, col2 );

    // check if card1 matches card2
    if( board[ row1-1 ][ col1-1 ] == board[ row2-1 ][ col2-1 ] )
    {
    }
    System.out.println( "--------------------------------" );
  }
}
```

# showBoard() and openCard()

```
public static void showBoard( char[][] b, boolean[][] m )
{
  for( int j=0; j<MAX; j++ ) { }

  for( int i=0; i<MAX; i++ )
  {
    for( int j=0; j<MAX; j++ )
    {
      if( m[i][j] )
      else
      { }
    }
  }
}
```

```
public static void openCard( char[][] b, boolean[][] m,
  int r1, int c1, int r2, int c2 )
{
  // print header indices
  for( int j=0; j<MAX; j++ ) { }

  // open two cards (if available)
  for( int i=0; i<MAX; i++ )
  {
    // print cards so far
    for( int j=0; j<MAX; j++ ) {
      if( m[i][j] )
      else
      if( (r1-1) == i && (c1-1) == j )
      else
      if( (r2-1) == i && (c2-1) == j )
      else
      { }
    }
  }
}
```

# Clustering Student Solutions

- K-medoids clustering
  - Another partitional clustering method
  - Medoids are actual data points
- General algorithm:
  - Initialize k points as medoids (m)
  - Associate each data point (x) to a closest medoid
  - Compute cost = $\sum_{C_i} \sum_{x \in C_i} |x - mi|$
  - Repeat
    - For each m and x
      - Swap m and x
      - Reassign all data points to closest medoid, recompute cost
      - If total cost is more than previous step, undo swap
  - Until cost does not decrease

# Method

- Preprocess code to obtain sequence of tokens
- Created n-grams of token sequences, n=5
- Cluster using k-medoids and Jaccard similarity
  - Definition: $$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$
- Results:
  - Obtained total of 12 clusters
  - Select 4 medoids with varying program structures to look at

# Results

- Cluster 1 (num = 5/85; average score 37.7%):
  - Overly simplistic structure
  - Suggests incomplete solution

```
class{
    Method main{
        for{}
    }
    Method showbroad{}
    Method opencard{}
}
```

# Results

- Cluster 2 (num = 3/85; average score 85.1%):
  - Uses additional helper methods
  - Suggests careful planning

```
class{
    Method main{
        while{}
    }
    Method openCard{
        if{}
        else if{}
    }
    Method setupBoard{
        for{
            for{
                while{}
            }
        }
    }
    Method showBoard{
        for{}
        for{
            for{
                if{}
                else if{}
                else{}
            }
        }
    }
    Method countCardNum{
        for{
            for{
                if{}
            }
        }
    }
    Method isAllMatched{
        for{
            for{}
        }
    }
    Method reset{
        for{}
        for{}
    }
}
```

# Results

- Cluster 3 (num = 30/85; average score 80.1%):
  - Closest solution to instructor's solution
  - Additional conditional branching

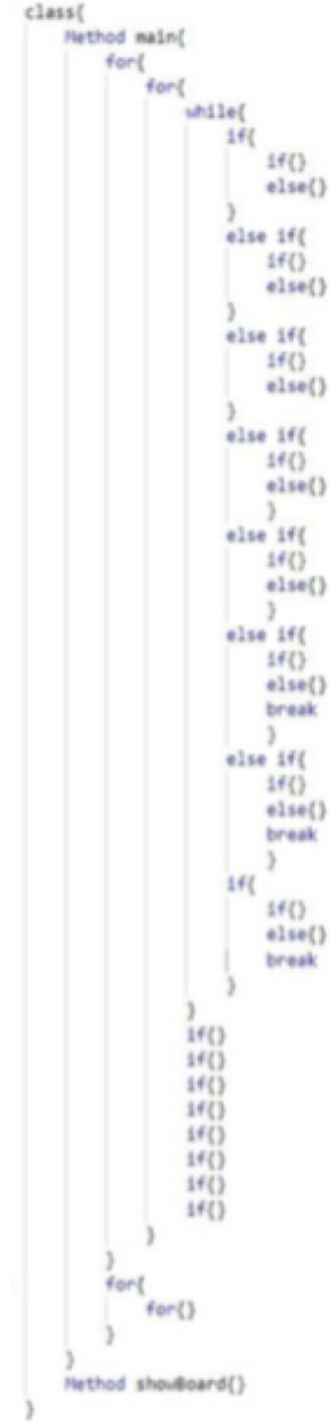
```
class{
    Method main{
        for{
            for{
                while{}
            }
        }
        for{
            for{
                while{
                    if{}
                    else{}
                }
            }
        }
    }
```

```
Method showBoard{
    for{
        for{
            for{
                if{}
                else if {}
            }
        }
    }
}
```

```
Method openCard{
    for{
        for{
            if{}
            else if{}
            else if{}
            else if{}
            else{}
        }
    }
}
}
```

# Results

- Cluster 4
(num = 3/85;
average score 44.8%):
  – Enumerate possible scenarios in main()
  – Missing openCard() and lack structure in showBoard()



# Key Ideas

- Prototype based clustering
  - Identifies a prototype within each cluster
  - K-means
  - K-medoids
- Key operations:
  - Initialization of default centroid
  - Define distance measure: L1 norm, L2 norm, cosine similarity, Jaccard similarity
  - Update new centroids
  - Overall objective function: sum of squared error
- Algorithm for k-means:
  - Select k points as initial centroids
  - Repeat
    - Form k clusters by assigning each point to its closest centroid
    - Update centroid of each cluster
  - Until centroids do not change