Artificial intelligence

documentation for assignment 3

Contents

Assignment

Task 3a

The task is to develop a classifier for a 2D space with points belonging to four different classes (red, green, blue, and purple). The classifier uses the k-NN (k-Nearest Neighbors) algorithm to assign a class to new points based on their coordinates. The project involves creating a function classify that takes X and Y coordinates, classifies the point, adds it to the 2D space, and returns the assigned class.

My project was written in Python language and I used this libraries

```
import random as rand
import tkinter
```

The ‘random’ module provides functions to generate random numbers.

This line imports the ‘tkinter’ module, a standard Python interface to the Tk GUI toolkit. It is used for creating simple GUI applications.

Introduction

In classification problems, the k-NN (k-Nearest Neighbors) algorithm is a popular choice due to its simplicity and effectiveness. The k-NN algorithm classifies a data point based on the majority class among its k-nearest neighbors. In a 2D space, it considers the Euclidean distance between points to identify the nearest neighbors. The assignment involves implementing a classifier using the k-NN algorithm to categorize points into four classes (red, green, blue, and purple) based on their coordinates (X, Y).

K-NN Algorithm:

The k-NN algorithm operates on the principle that points with similar features are likely to belong to the same class. For a given point, it calculates the distance to all other points in the dataset and selects the k-nearest neighbors.

Reasons for Choosing k-NN:

Simplicity: K-NN is easy to understand and implement, making it a good choice for this assignment.

Non-Parametric: K-NN is a non-parametric algorithm, meaning it does not make assumptions about the underlying data distribution.

Adaptability: It works well for both linear and non-linear decision boundaries.

Versatility: K-NN can be applied to various types of data and is effective for multiclass classification.

Data Representation:

The 2D space is represented by X and Y coordinates ranging from -5000 to +5000. Each class has five initial points with specific coordinates. The task is to classify new points generated randomly into these classes using the k-NN algorithm with varying values of k (1, 3, 7, and 15).

Experimental Setup:

The experiment involves generating 40,000 new points, with 10,000 points for each class. The distribution of points is biased towards their respective quadrants to simulate real-world scenarios. The classify function assigns a class to each new point based on the k-NN algorithm.

Visualizations:

For each experiment (k = 1, 3, 7, 15), the resulting 2D surface is visualized by coloring the entire space. Empty spaces are colored based on the classifier's decision. This provides a clear representation of how well the classifier distinguishes between classes.

Functions and classes

Point Class

```
2 usages
class point:
    def __init__(self, x, y, color, section):
        self.x = x
        self.y = y
        self.color = color
        self.section = section
```

class point: Defines a new class named point.

def__init__(self, x, y, color, section): This is the constructor for the point class. It initializes a new instance of the class with the given parameters.

self.x = x; self.y = y: These lines set the x and y coordinates of the point.

self.color = color; self.section = section: These lines set the color and section attributes of the point.

KNNClassifier Class

```
class KNNClassifier:
    def __init__(self, k, x_range=(-5000, 5000), y_range=(-5000, 5000)):
        self.k = k
        self.x_range = x_range
        self.y_range = y_range
        self.points = []
```

class KNNClassifier: Defines a new class named KNNClassifier, which will implement a K-Nearest Neighbors classifier.

def__init__(self, k, x_range=(-5000, 5000), y_range=(-5000, 5000)): The constructor for the KNNClassifier class.

self.k = k: Sets the number of neighbors to consider in the classification.

self.x_range = x_range; self.y_range = y_range: Sets the range of x and y values for the points.

self.points = []: Initializes an empty list to store the points.

def generate_initial_points(self):: A method to generate initial points with predefined colors and locations.

```
def generate_initial_points(self):
    initial_points_data = {
        'R': [[-4500, -4400], [-4100, -3000], [-1800, -2400], [-2500, -3400], [-2000, -1400]],
        'G': [[4500, -4400], [4100, -3000], [1800, -2400], [2500, -3400], [2000, -1400]],
        'B': [[-4500, 4400], [-4100, 3000], [-1800, 2400], [-2500, 3400], [-2000, 1400]],
        'P': [[4500, 4400], [4100, 3000], [1800, 2400], [2500, 3400], [2000, 1400]],
    }

    initial_points = []

    for color, points in initial_points_data.items():
        for x, y in points:
            initial_points.append(point(x, y, color, color))
    self.points = initial_points.copy()
```

def classify(self, x, y, section):: A method that classifies a point based on the k nearest points in the classifier.

```
1 usage (1 dynamic)
def classify(self, x, y, section):
    sorted_points = sorted(self.points, key=lambda point: abs(point.x - x) + abs(point.y - y))
    color_counts = {'R': 0, 'G': 0, 'B': 0, 'P': 0}
    for i in range(self.k):
        color_counts[sorted_points[i].color] += 1
    col = max(color_counts, key=color_counts.get)
    self.points.append(point(x, y, col, section))
```

This function takes four arguments: self, which refers to the instance of the class that the method is called on, x and y, which are the coordinates of a point, and section, which is some section identifier.

In this line, the function starts by sorting a list of points (self.points) based on their distances from the input coordinates (x and y). The key parameter is used to define a custom sorting criteria. The lambda function calculates the distance between each point's x and y coordinates and the given x and y. Sorting in this way ensures that the closest points come first in the sorted_points list.

Here, a dictionary called color_counts is created to keep track of the count of different colors. It initializes counts for four colors: 'R', 'G', 'B', and 'P' with values all set to 0.

In this loop, the function iterates self.k times, which appears to be some predefined value. During each iteration, it looks at the color of the i-th closest point in the sorted_points list and increments the count for that color in the

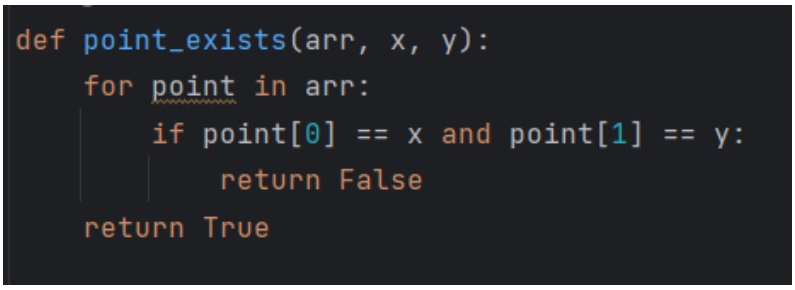
color_counts dictionary. This loop is used to count the colors of the k closest points.

Here, the function identifies the color with the highest count from the color_counts dictionary. The max() function is used with the key parameter set to color_counts.get, which means it finds the maximum value based on the counts of each color in the dictionary.

Finally, a new point is created with the input coordinates x and y, the color col determined earlier (the color with the highest count among the closest points), and the section identifier. This newly created point is then added to the self.points list, presumably to be used in further calculations or processing.

In summary, this classify function sorts a list of points based on their distances to a given point, counts the colors of the k closest points, and assigns the most frequent color to a new point with the same coordinates and a section identifier before adding it to the list of points.

def point_exists(arr, x, y):: A function to check if a point with coordinates (x, y) already exists in an array arr.



```
def point_exists(arr, x, y):
    for point in arr:
        if point[0] == x and point[1] == y:
            return False
    return True
```

Generate points

def generate_test_points():: This function generates a set of test points with specific distributions and conditions.

```python
def generate_test_points():
    test_points = []
    for i in range(10000):
        if 1 == rand.randint( a: 1,  b: 100):
            while True:
                x = rand.randint(-5000,  b: 5000)
                y = rand.randint(-5000,  b: 5000)
                if point_exists(test_points, x, y):
                    test_points.append([(x, y,), 'R'])
                    break
        else:
            while True:
                x = rand.randint(-5000,  b: 499)
                y = rand.randint(-5000,  b: 499)
                if point_exists(test_points, x, y):
                    test_points.append([(x, y,), 'R'])
                    break

        if 1 == rand.randint( a: 1,  b: 100):
            while True:
                x = rand.randint(-5000,  b: 5000)
                y = rand.randint(-5000,  b: 5000)
                if point_exists(test_points, x, y):
                    test_points.append([(x, y,), 'G'])
                    break
```

```
    else:
        while True:
            x = rand.randint(-499,  b: 5000)
            y = rand.randint(-5000,  b: 499)
            if point_exists(test_points, x, y):
                test_points.append([(x, y,), 'G'])
                break

    if 1 == rand.randint( a: 1,  b: 100):
        while True:
            x = rand.randint(-5000,  b: 5000)
            y = rand.randint(-5000,  b: 5000)
            if point_exists(test_points, x, y):
                test_points.append([(x, y,), 'B'])
                break
    else:
        while True:
            x = rand.randint(-5000,  b: 499)
            y = rand.randint(-499,  b: 5000)
            if point_exists(test_points, x, y):
                test_points.append([(x, y,), 'B'])
                break

    if 1 == rand.randint( a: 1,  b: 100):
        while True:
            x = rand.randint(-5000,  b: 5000)
            y = rand.randint(-5000,  b: 5000)
            if point_exists(test_points, x, y):
                test_points.append([(x, y,), 'P'])
```

```
    if 1 == rand.randint( a: 1,  b: 100):
        while True:
            x = rand.randint(-5000,  b: 5000)
            y = rand.randint(-5000,  b: 5000)
            if point_exists(test_points, x, y):
                test_points.append([(x, y,), 'P'])
                break
    else:
        while True:
            x = rand.randint(-499,  b: 5000)
            y = rand.randint(-499,  b: 5000)
            if point_exists(test_points, x, y):
                test_points.append([(x, y,), 'P'])
                break
return test_points
```

Visualization Function

def visualize_classifier(classifier, test_points, k, canvas):: A function to visualize the classification process on a tkinter canvas. It also calculates and prints the accuracy of the classifier

```
def visualize_classifier(classifier, test_points, k, canvas):
    label_colors = {'R': 'red', 'G': 'green', 'B': 'blue', 'P': 'purple'}
    accuracy = 0
    for point, color in test_points:
        x, y = point
        classifier.classify(x, y, color)
    for i in classifier.points:
        if k == 1:
            canvas.create_oval((i.x + 4950) * 0.025, (i.y + 4950) * 0.025, (i.x + 5050) * 0.025, (i.y + 5050) * 0.025
                               fill=label_colors[i.color], outline=label_colors[i.color])
        elif k == 3:
            canvas.create_oval((i.x + 4950) * 0.025 + 250, (i.y + 4950) * 0.025, (i.x + 5050) * 0.025 + 250,
                               (i.y + 5050) * 0.025,
                               fill=label_colors[i.color], outline=label_colors[i.color])
        elif k == 7:
            canvas.create_oval((i.x + 4950) * 0.025, (i.y + 4950) * 0.025 + 250, (i.x + 5050) * 0.025,
                               (i.y + 5050) * 0.025 + 250,
                               fill=label_colors[i.color], outline=label_colors[i.color])
        else:
            canvas.create_oval((i.x + 4950) * 0.025 + 250, (i.y + 4950) * 0.025 + 250, (i.x + 5050) * 0.025 + 250,
                               (i.y + 5050) * 0.025 + 250,
                               fill=label_colors[i.color], outline=label_colors[i.color])
        if i.color == i.section:
            accuracy += 1
    accuracy -= 20
    print(accuracy / 40000 * 100)
    canvas.update()
```

Main Function

def main():: The main function that creates a tkinter canvas, generates test points, and visualizes the classifier's performance for different values of k.

```
1 usage
def main():
    canvas = tkinter.Canvas(width=500, height=500)
    canvas.pack()
    test_points = generate_test_points()
    for k in [1, 3, 7, 15]:
        classifier = KNNClassifier(k)
        classifier.generate_initial_points()
        visualize_classifier(classifier, test_points, k, canvas)
    canvas.mainloop()


if __name__ == "__main__":
    main()
```

Testing

In my program, I using a form of empirical testing known as simulation-based testing. This form of testing involves creating a simulated environment to test the functionality and performance of your K-Nearest Neighbors (KNN) classifier.

Functionality Testing:

My code tests the basic functionality of the KNNClassifier class and its methods (generate_initial_points and classify). It ensures that the classifier can correctly classify points based on the nearest neighbors algorithm.

Data Generation Testing:

The generate_test_points function is used to generate a set of test points with specific distributions (in this case, randomly within certain ranges and with certain conditions). This tests the classifier's ability to handle a variety of input scenarios and distributions.

Visualization Testing:

The visualize_classifier function is crucial for visual testing. It allows to see how the classifier performs visually on a canvas. This is particularly important for understanding how well the classifier is working and can help in debugging and improving the model.

Parameter Variation Testing:

My main function varies the k parameter (number of neighbors) in the KNN algorithm. By testing with different values of k (1, 3, 7, 15), you are essentially testing how sensitive your classifier is to this parameter and how it impacts the overall accuracy and performance.

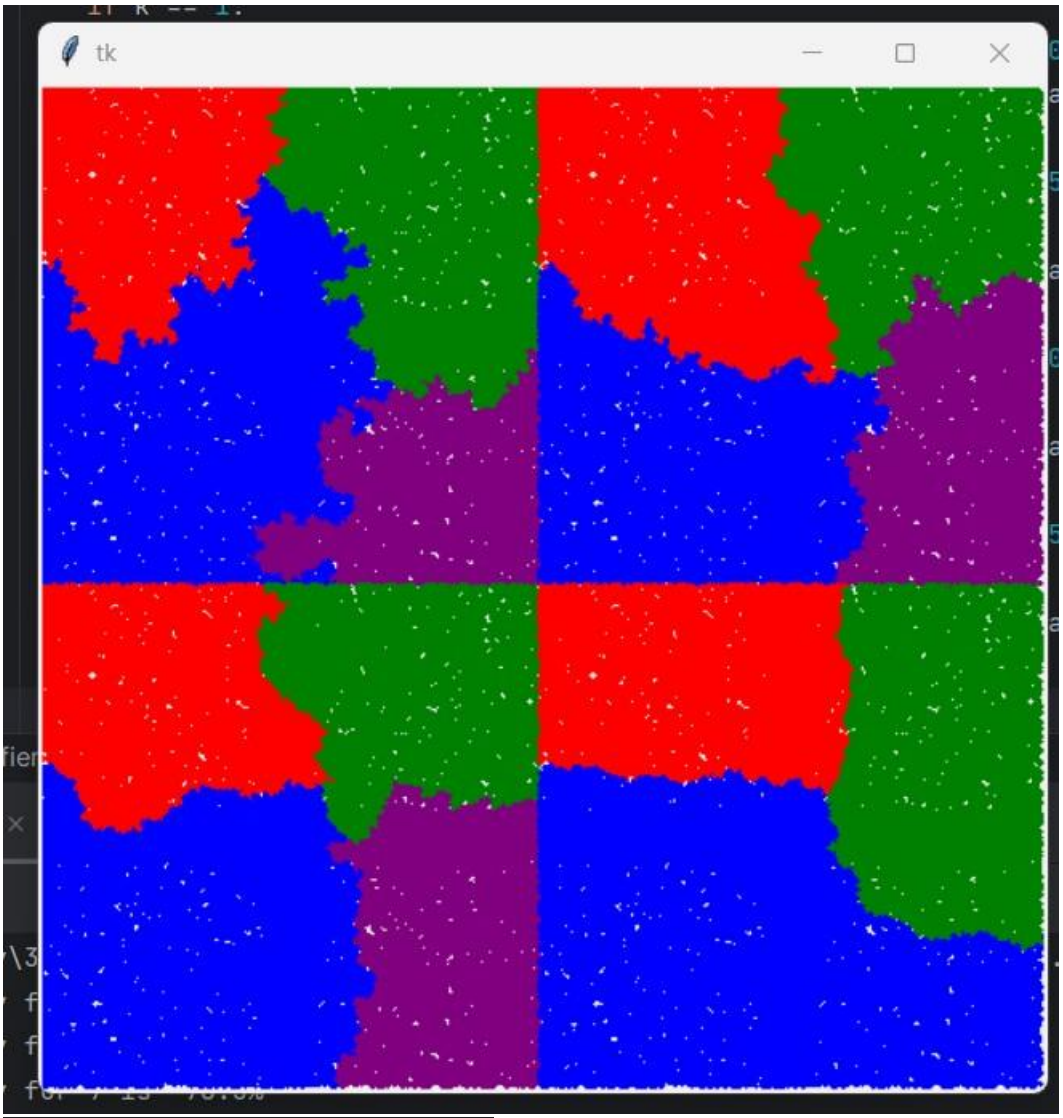
Accuracy Measurement:

My code also includes a rudimentary form of accuracy measurement, where it counts the number of correctly classified points and calculates the accuracy percentage. This is a form of performance testing, providing quantitative feedback on the model's effectiveness. In function visualize_classifier

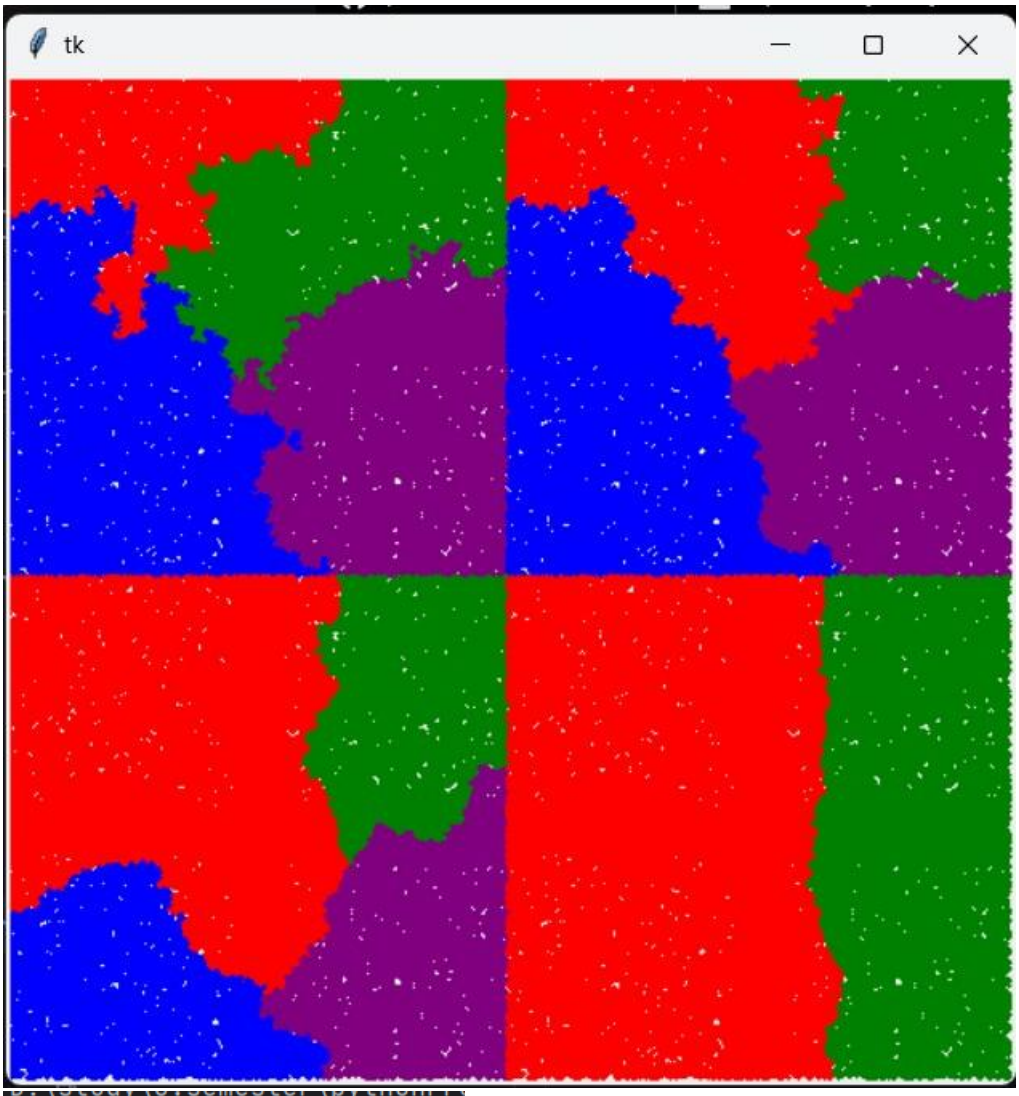
```
print(f"Accuracy for {k} is  {accuracy / 40000 * 100:.1f}%")
canvas.update()
```

Integration Testing with GUI:

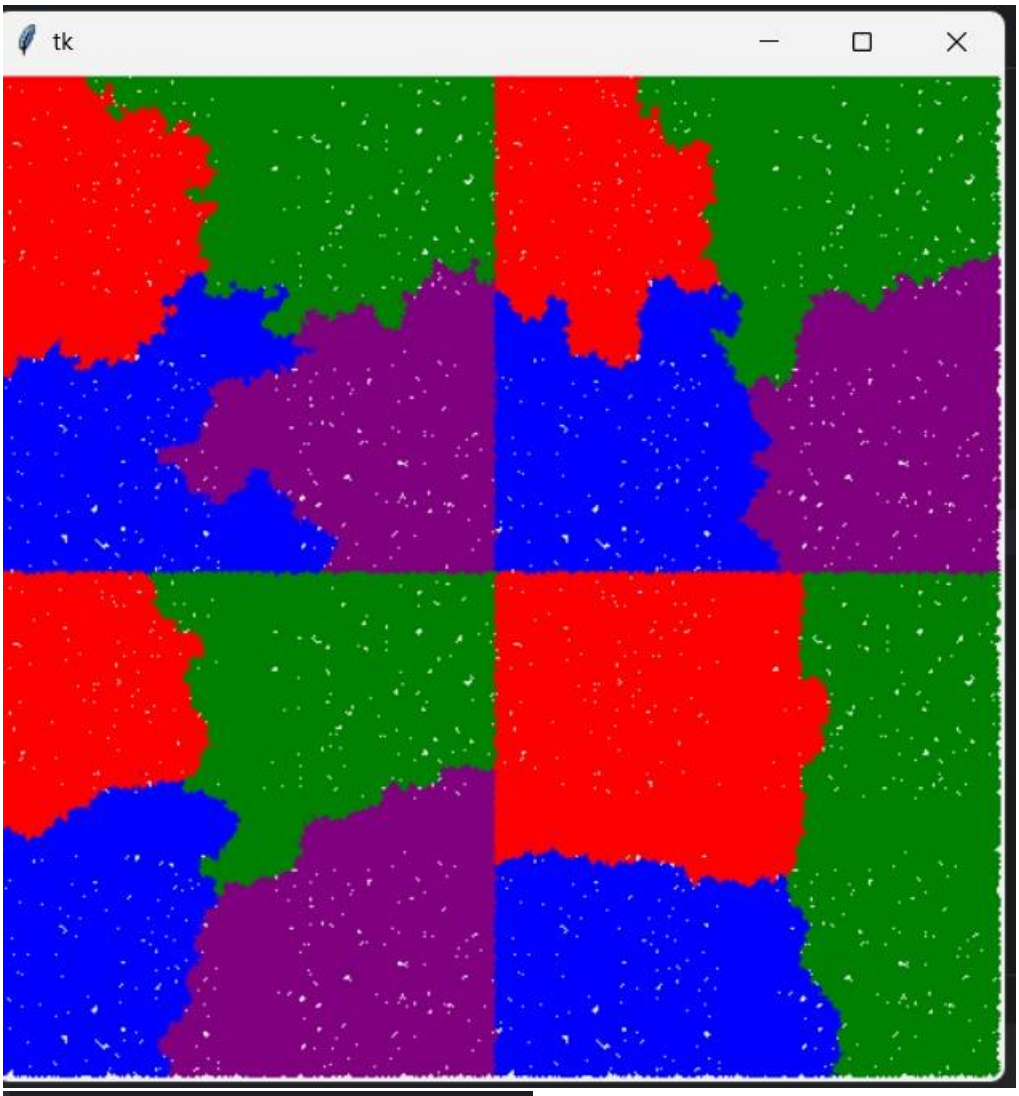
The integration of mine classifier with a tkinter canvas for visualization also serves as a form of integration testing. It ensures that mine classifier can work effectively in conjunction with a graphical user interface.




```
Accuracy for 1 is  73.3%
Accuracy for 3 is  76.6%
Accuracy for 7 is  76.3%
Accuracy for 15 is  59.7%
```




```
Accuracy for 1 is  72.0%
Accuracy for 3 is  72.0%
Accuracy for 7 is  69.8%
Accuracy for 15 is  41.9%
```




```
Accuracy for 1 is  75.2%
Accuracy for 3 is  77.9%
Accuracy for 7 is  75.5%
Accuracy for 15 is  60.9%
```

Conclusion

In summary, my program employs a combination of functionality testing, data generation testing, visualization testing, parameter variation testing, accuracy measurement, and integration testing with a GUI.

My project, encompassing the implementation and testing of a K-Nearest Neighbors (KNN) classifier using Python, showcases a practical application of machine learning algorithms in a controlled simulation environment. The core components of the project include the point and KNNClassifier classes, utility

functions for point generation and verification, and a visualization method utilizing the tkinter library.

This comprehensive approach helps in thoroughly evaluating the performance and reliability of your KNN classifier in various scenarios and parameter configurations.