```
# start here without assistance
import seaborn as sns

def plot_graph_grid(X, y, legend_labels):
    """
    plots a grid of scatter plots and KDE plots for the given data X and labels y

    Parameters:
        X : the feature data
        y : the target labels
        legend_labels : the legend labels corresponding to the target labels

    Returns:
        nothing
    """
    fig, ax = plt.subplots(4, 4, figsize=(12, 12), sharex='col')

    for i, feature in enumerate(selected_features):
        for j, other_feature in enumerate(selected_features):
            # plot scatter plots for all combinations of features
            if feature != other_feature:
                for label in np.unique(y):
                    # learned about scatterplots from
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html
                    ax[i, j].scatter(X[y == label, selected_features.index(other_feature)], X[y ==
label, selected_features.index(feature)], label=legend_labels[label], edgecolor='white')
                    # remove tick labels
                    ax[i, j].tick_params(axis='both', which='both', labelbottom=False, labelleft=True)
            else:
                # plot kde plots for the same feature
                for label in np.unique(y):
                    # only show x label for the bottom row and y label for the left column
                    if i == 0:
                        # learned about seaborn and kde plots from
https://seaborn.pydata.org/generated/seaborn.kdeplot.html
                        sns.kdeplot(X[y == label, selected_features.index(feature)], ax=ax[i, j],
label=legend_labels[label], alpha=0.3, fill=True).set(xlabel='', ylabel=feature)
                    elif j == 3:
                        sns.kdeplot(X[y == label, selected_features.index(feature)], ax=ax[i, j],
label=legend_labels[label], alpha=0.3, fill=True).set(xlabel=feature, ylabel='')
                    else:
                        sns.kdeplot(X[y == label, selected_features.index(feature)], ax=ax[i, j],
label=legend_labels[label], alpha=0.3, fill=True).set(xlabel='', ylabel='')
                    ax[i, j].tick_params(axis='both', which='both', labelbottom=False, labelleft=False)

    # set x and y labels for the scatter plots
    for i in range(len(selected_features)):
        ax[3, i].set_xlabel(selected_features[i])
        ax[i, 0].set_ylabel(selected_features[i])
        ax[i, 0].tick_params(axis='y', labelleft=True, labelbottom=False)
        ax[3, i].tick_params(axis='x', labelbottom=True, labelleft=False)

    # show tick values on all axes
    for ax_row in ax:
        for ax_elem in ax_row:
            ax_elem.tick_params(axis='both', which='both', labelleft=True, labelbottom=True)

    # set the legend for all plots
    ax[0, 3].legend(loc='upper left', bbox_to_anchor=(1.05, 1))
    plt.tight_layout()
    plt.show()

    df_wine.head()

    # run the plotting function
plot_graph_grid(X, y, wine.target_names)

# noise code
mySeed = 12345
np.random.seed(mySeed)
XN=X+np.random.normal(0,0.6,X.shape)

plot_graph_grid(XN, y, wine.target_names)

# helper code

def split_data(X,y):
    """
    splits the given data into training and test sets

    Parameters:
        X : the feature data
        y : the target labels

    Returns:
        X_train : the training feature data
        X_test : the test feature data
        y_train : the training target labels
        y_test : the test target labels
    """
    # set the random seed
    np.random.seed(mySeed)

    samples = len(X)

    indices = np.random.permutation(samples)
    # get the number of training samples (70% of the data)
    train_samples = int(np.ceil(samples * 0.7))

    X_train = X[indices[:train_samples]]
    y_train = y[indices[:train_samples]]

    X_test = X[indices[train_samples:]]
    y_test = y[indices[train_samples:]]


    return X_train, X_test, y_train, y_test

def distance_between_vectors(x1, x2):
    """
    calculates the Euclidean distance between two vectors

    Parameters:
        x1 : the first vector
        x2 : the second vector

    Returns:
        the Euclidean distance between the two vectors
    """
    distance = np.sqrt(np.sum((x1 - x2)**2))
    return distance

def compare_distances(x, X_train, y_train, options):
    """
    calculates the Euclidean distance between a vector and all vectors in a dataset and
returns the shortest distances

    Parameters:
        x : the vector to compare distances to
        X_train : the feature data
        y_train : the target labels
        options : the number of shortest distances to return

    Returns:
        a list of the shortest distances and their labels
    """

    list_of_shortest_distances = []
    for i in range(len(X_train)):
        distance = distance_between_vectors(x, X_train[i])
        if len(list_of_shortest_distances) < options:
            list_of_shortest_distances.append((distance, y_train[i]))
        else:
            list_of_shortest_distances.sort(reverse=True)
            if distance < list_of_shortest_distances[0][0]:
                list_of_shortest_distances[0] = (distance, y_train[i])
    return list_of_shortest_distances

def mykNN(X,y,X_,options):
    """
    Classifies the given data using the k-Nearest Neighbors algorithm.

    Parameters:
        X : The training feature data.
        y : The training target labels.
        X_ : The test feature data.
        options : The number of neighbors to consider.

    Returns:
        y_ : The predicted target labels.

    """
    y_ = []
    for unknown in X_:
        closest = compare_distances(unknown, X, y, options)
        for i in range(len(closest)):
            closest[i] = closest[i][1]
        #find most common value and if there is a tie, choose the first one
        #had to freshen up on numpy using numpy docs, reference:
https://numpy.org/doc/stable/reference/routines.sort.html
        unique, counts = np.unique(closest, return_counts=True)
        most_occuring = unique[np.argmax(counts)] #
        y_.append(most_occuring)
    return y_

from sklearn.metrics import accuracy_score
X_train, X_test, y_train, y_test = split_data(XN, y)

y_pred = mykNN(X_train, y_train, X_test, 3)

print("Accuracy:", accuracy_score(y_test, y_pred))

# confusion matrix, accuracy, precision, recall, etc.

def confusion_matrix(y_true, y_pred, labels):
    """
    calculates the confusion matrix for the given data

    Parameters:
        y_true : the true target labels
        y_pred : the predicted target labels
        labels : the labels to use for the confusion matrix

    Returns:
        cm : the confusion matrix
    """
    # initialize the confusion matrix
    cm = np.zeros((len(labels), len(labels)))

    # populate the confusion matrix
    for i in range(len(y_true)):
        cm[y_true[i]][y_pred[i]] += 1

    return cm

def draw_confusion_matrix(cm, labels):
    """
    draws the confusion matrix via print statements

    Parameters:
        cm : the confusion matrix
        labels : the labels to use for the confusion matrix

    Returns:
        nothing
    """
    print("\tConfusion Matrix")
    print("T\P", end="\t")
    for label in labels:
        print(label, end="\t")
    print()
    for i in range(len(cm)):
        print(labels[i], end="\t")
        for j in range(len(cm[i])):
            print(cm[i][j], end="\t")
        print()



def calc_accuracy(y_true, y_pred):
    """
    calculates the accuracy of the given data

    Parameters:
        y_true : the true target labels
        y_pred : the predicted target labels

    Returns:
        the accuracy of the given data
    """
    # create the confusion matrix
    cm = confusion_matrix(y_true, y_pred, np.unique(y_true))

    # calculate the accuracy
    correct = cm[0][0] + cm[1][1] + cm[2][2]
    total = np.sum(cm)
    accuracy = correct / total

    return accuracy

def recall(y_true, y_pred, label):
    """
    calculates the recall of the given data for a given label

    Parameters:
        y_true : the true target labels
        y_pred : the predicted target labels
        label : the label to calculate the recall for

    Returns:
        the recall of the given data for the given label
    """
    # create the confusion matrix
    cm = confusion_matrix(y_true, y_pred, np.unique(y_true))

    # calculate the recall
    true_positives = cm[label][label]
    false_negatives = cm[label][0] + cm[label][1] + cm[label][2] - true_positives
    recall = true_positives / (true_positives + false_negatives)

    return recall

def precision(y_true, y_pred, label):
    """
    calculates the precision of the given data

    Parameters:
        y_true : the true target labels
        y_pred : the predicted target labels
        label : the label to calculate the precision for

    Returns:
        the precision of the given data
    """
    # create the confusion matrix
    cm = confusion_matrix(y_true, y_pred, np.unique(y_true))
    # calculate the precision
    true_positives = cm[label][label]
    false_positives = cm[0][label] + cm[1][label] + cm[2][label] - true_positives
    precision = true_positives / (true_positives + false_positives)

    return precision

# test evaluation code

print("y_test:", y_test)
print("y_pred:", y_pred)
print()
cm = confusion_matrix(y_test, y_pred, np.unique(y_test))
draw_confusion_matrix(cm, np.unique(y_test))
print()
print("Accuracy:", calc_accuracy(y_test, y_pred))
print("Recall:", recall(y_test, y_pred, 0))
print("Precision:", precision(y_test, y_pred, 0))
print()

#compare to sklearn
from sklearn.neighbors import KNeighborsClassifier
sk_classifier = KNeighborsClassifier(n_neighbors=3)
sk_classifier.fit(X_train, y_train)
y_pred_sk = sk_classifier.predict(X_test)
print("y_pred_sk:", y_pred_sk)
print()
cm = confusion_matrix(y_test, y_pred_sk, np.unique(y_test))
draw_confusion_matrix(cm, np.unique(y_test))
print()
print("SK_Accuracy:", calc_accuracy(y_test, y_pred_sk))
print("SK_Recall:", recall(y_test, y_pred_sk, 0))
print("SK_Precision:", precision(y_test, y_pred_sk, 0))
print()

#Make manhattan distance function
def manhattan_distance(x1, x2):
    """
    calculates the manhattan distance between two points

    Parameters:
        x1 : the first point
        x2 : the second point

    Returns:
        the manhattan distance between the two points
    """
    distance = np.sum(np.abs(x1 - x2))
    return distance


#redo compare distance function to use manhattan distance

def compare_distances_manhattan(x, X_train, y_train, options):
    """
    calculates the Manhattan distance between a vector and all vectors in a dataset and
returns the shortest distances

    Parameters:
        x : the vector to compare distances to
        X_train : the feature data
        y_train : the target labels
        options : the number of shortest distances to return

    Returns:
        a list of the shortest distances and their labels
    """

    list_of_shortest_distances = []
    for i in range(len(X_train)):
        distance = manhattan_distance(x, X_train[i])
        if len(list_of_shortest_distances) < options:
            list_of_shortest_distances.append((distance, y_train[i]))
        else:
            list_of_shortest_distances.sort(reverse=True)
            if distance < list_of_shortest_distances[0][0]:
                list_of_shortest_distances[0] = (distance, y_train[i])
    return list_of_shortest_distances

#redo mykNN to use manhattan distance

def mykNNManhattan(X, y, X_, options):
    """
    Classifies the given data using the k-Nearest Neighbors algorithm.

    Parameters:
        X : The training feature data.
        y : The training target labels.
        X_ : The test feature data.
        options : The number of neighbors to consider.

    Returns:
        y_ : The predicted target labels.

    """
    y_ = []
    for unknown in X_:
        closest = compare_distances_manhattan(unknown, X, y, options)
        for i in range(len(closest)):
            closest[i] = closest[i][1]
        #find most common value and if there is a tie, choose the first one
        unique, counts = np.unique(closest, return_counts=True)
        most_occuring = unique[np.argmax(counts)]
        y_.append(most_occuring)
    return y_

# myNestedCrossVal code

def myNestedCrossVal(X, y, k, neighbor_amount, methods, seed):
    """
    performs nested cross validation on the given data and prints the results

    Parameters:
        X : the feature data
        y : the target labels
        k : the number of folds
        neighbor_amount : the number of neighbors
        methods : the distance methods to use
        seed : the seed to use for the random number generator

    Returns:
        all_accuracies : a list of the accuracies for each fold

    """
    np.random.seed(seed)
    samples = len(X)
    indices = np.random.permutation(samples)

    # Shuffle the data
    X = X[indices]
    y = y[indices]

    # Split the data into k folds
    X_folds = np.array_split(X, k)
    y_folds = np.array_split(y, k)

    all_accuracies = []
    all_cm = []

    for i in range(k):
        fold_accuracies = []

        # Create the training and testing data
        X_train = np.concatenate(X_folds[:i] + X_folds[i+1:])
```

```
y_train = np.concatenate(y_folds[:i] + y_folds[i+1:])
X_test = X_folds[i]
y_test = y_folds[i]

best_method = ""
best_neighbors = 0
best_accuracy = 0

for method in methods:
    # Split the training data into k folds
    X_train_folds = np.array_split(X_train, k)
    y_train_folds = np.array_split(y_train, k)

    # Create the training and validation data
    X_train_inner = np.concatenate(X_train_folds[:0] + X_train_folds[1:])
    y_train_inner = np.concatenate(y_train_folds[:0] + y_train_folds[1:])
    X_val_inner = X_train_folds[0]
    y_val_inner = y_train_folds[0]

    if method == "euclidean":
        for neighbors in neighbor_amount:
            # Use my old code to create the model since it uses Euclidean distance
            y_pred_euc = mykNN(X_train_inner, y_train_inner, X_val_inner, neighbors)
            # Calculate the accuracy
            accuracy = calc_accuracy(y_val_inner, y_pred_euc)
            # Check if this is the best accuracy in the fold
            if accuracy > best_accuracy:
                best_accuracy = accuracy
                best_neighbors = neighbors
                best_method = method
            # Add the accuracy to the list
            fold_accuracies.append(accuracy)
            print(f"Fold {i+1} - {method} - Neighbors={neighbors}- Accuracy:", accuracy)
    elif method == "manhattan":
        for neighbors in neighbor_amount:
            # Use my Manhattan code to create the model
            y_pred_man = mykNNManhattan(X_train, y_train, X_test, neighbors)
            # Calculate the accuracy
            accuracy = calc_accuracy(y_test, y_pred_man)
            # Check if this is the best accuracy in the fold
            if accuracy > best_accuracy:
                best_accuracy = accuracy
                best_neighbors = neighbors
                best_method = method
            # Add the accuracy to the list
            fold_accuracies.append(accuracy)
            print(f"Fold {i+1} - {method} - Neighbors={neighbors} - Accuracy:", accuracy)
print()
```

```
        print(f"Best Accuracy in Fold {i+1}:", best_accuracy)
        print(f"Best Neighbors in Fold {i+1}:", best_neighbors)
        print(f"Best Method in Fold {i+1}:", best_method)
        print()

        if best_method == "euclidean":
            y_pred = mykNN(X_train, y_train, X_test, best_neighbors)
            accuracy = calc_accuracy(y_test, y_pred)
            all_accuracies.append(accuracy)
        elif best_method == "manhattan":
            y_pred = mykNNManhattan(X_train, y_train, X_test, best_neighbors)
            accuracy = calc_accuracy(y_test, y_pred)
            all_accuracies.append(accuracy)

        # Calculate the accuracy
        accuracy = accuracy_score(y_test, y_pred)
        cm = confusion_matrix(y_test, y_pred, np.unique(y))
        print(end="\t")
        print(f"Fold {i+1}")
        draw_confusion_matrix(cm, np.unique(y))
        all_cm.append(cm)
        print()
        print()

    print("OVERALL SUMMARY CONFUSION MATRIX")

    #combine all folds into one confusion matrix
    overall_cm = np.zeros((len(np.unique(y)), len(np.unique(y))))
    for i in range(len(all_accuracies)):
        overall_cm += all_cm[i]
    overall_cm = overall_cm.astype(int)
    draw_confusion_matrix(overall_cm, np.unique(y))
    print()

    return all_accuracies

# evaluate clean data code
myNestedCrossVal(X, y, 5, list(range(1,11)), ["euclidean", "manhattan"], mySeed)

# evaluate noisy  data code
myNestedCrossVal(XN, y, 5, list(range(1,11)), ["euclidean", "manhattan"], mySeed)

print('CLEAN')
# clean data summary results
myNestedCrossVal(X, y, 5, list(range(1,11)), ["euclidean", "manhattan"], mySeed)

print('NOISY')
# noisy data summary results
```

```
myNestedCrossVal(XN, y, 5, list(range(1,11)), ["euclidean", "manhattan"], mySeed)

#end here without assistance
```