
Within this report we will be creating a implementation of the KNN algorithm on the wine dataset.

This report will consist of the following sections:

- Dataset Exploration
- Implementing kNN
- Classifier evaluation
- Nested Cross-validation

```
# Lets just load all the libraries we going to use in one shot
from sklearn import datasets
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import math
```

Dataset Exploration

In report we are going to be working with the **Wine** dataset. This is a 178 sample dataset that categorises 3 different types of Italian wine using 13 different features.

```
# set matplotlib backend to inline
%matplotlib inline

# load data
wine=datasets.load_wine()

# this dataset has 13 features, we will only choose a subset of these
df_wine = pd.DataFrame(wine.data, columns = wine.feature_names )
selected_features = ['alcohol','flavanoids','color_intensity','ash']

# extract the data as numpy arrays of features, X, and target, y
X = df_wine[selected_features].values
y = wine.target
```

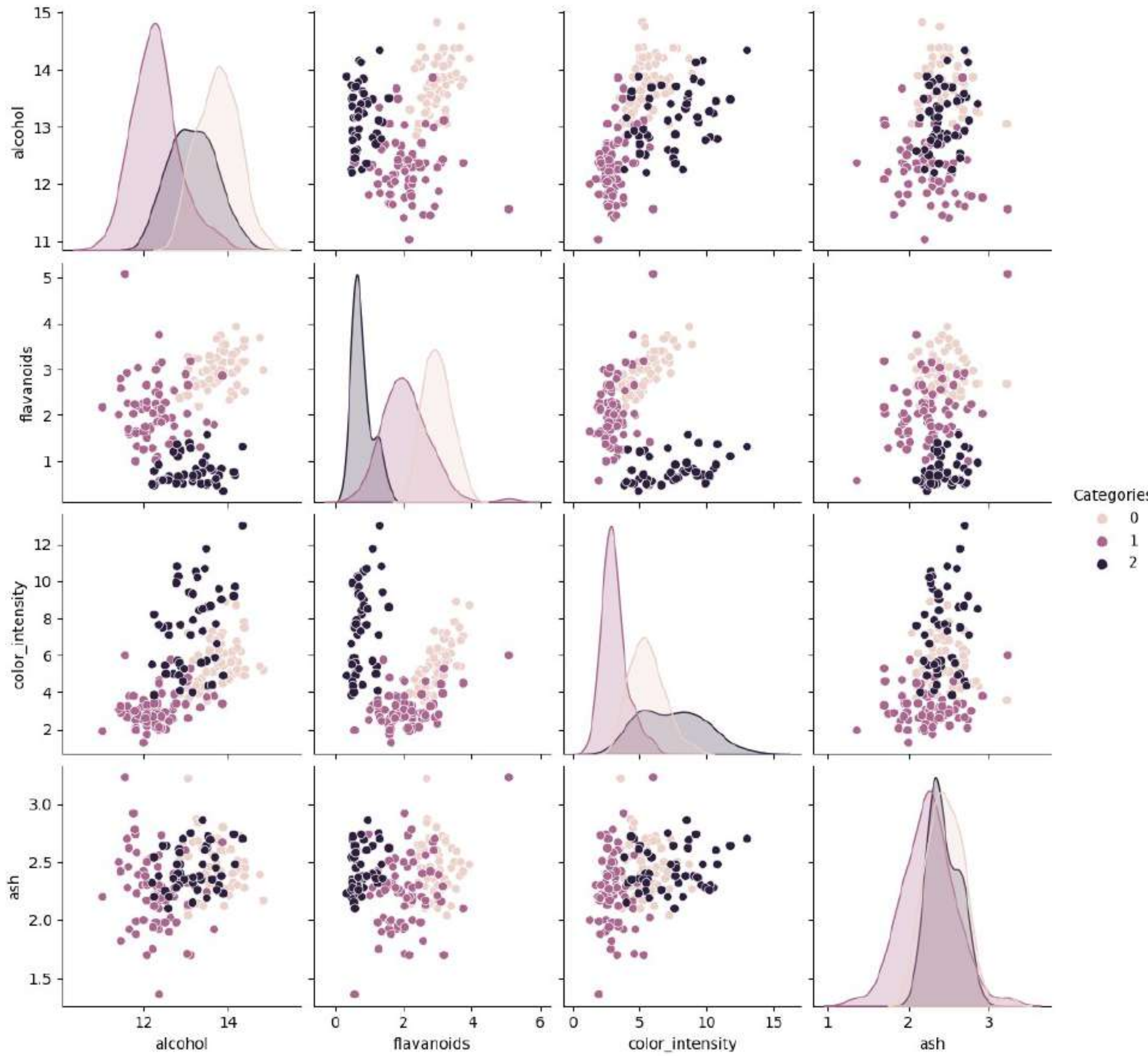
Visually exploring the data

Lets plot the data as is, and see if we can identify any patterns and relations.

```
# define plotting function
def myplotGrid(X,y,features):
    """
    Plots a matrix pair scatter plot.
```

```
    Args:
        X (array-like): Array of vectors.
        y (array-like): Array of dependent variables.
        features (array-like): Array of feature labels

    Returns:
        Nothing
    """
    # Creates a dataframe
    df = pd.DataFrame(X, columns = features)
    # Concatenate dependent variables with independent variables
    df['Categories'] = y
    return sns.pairplot(df, hue="Categories")

# run the plotting function
myplotGrid(X,y,selected_features)
```

```
<seaborn.axisgrid.PairGrid at 0x7f938e1cda80>
```

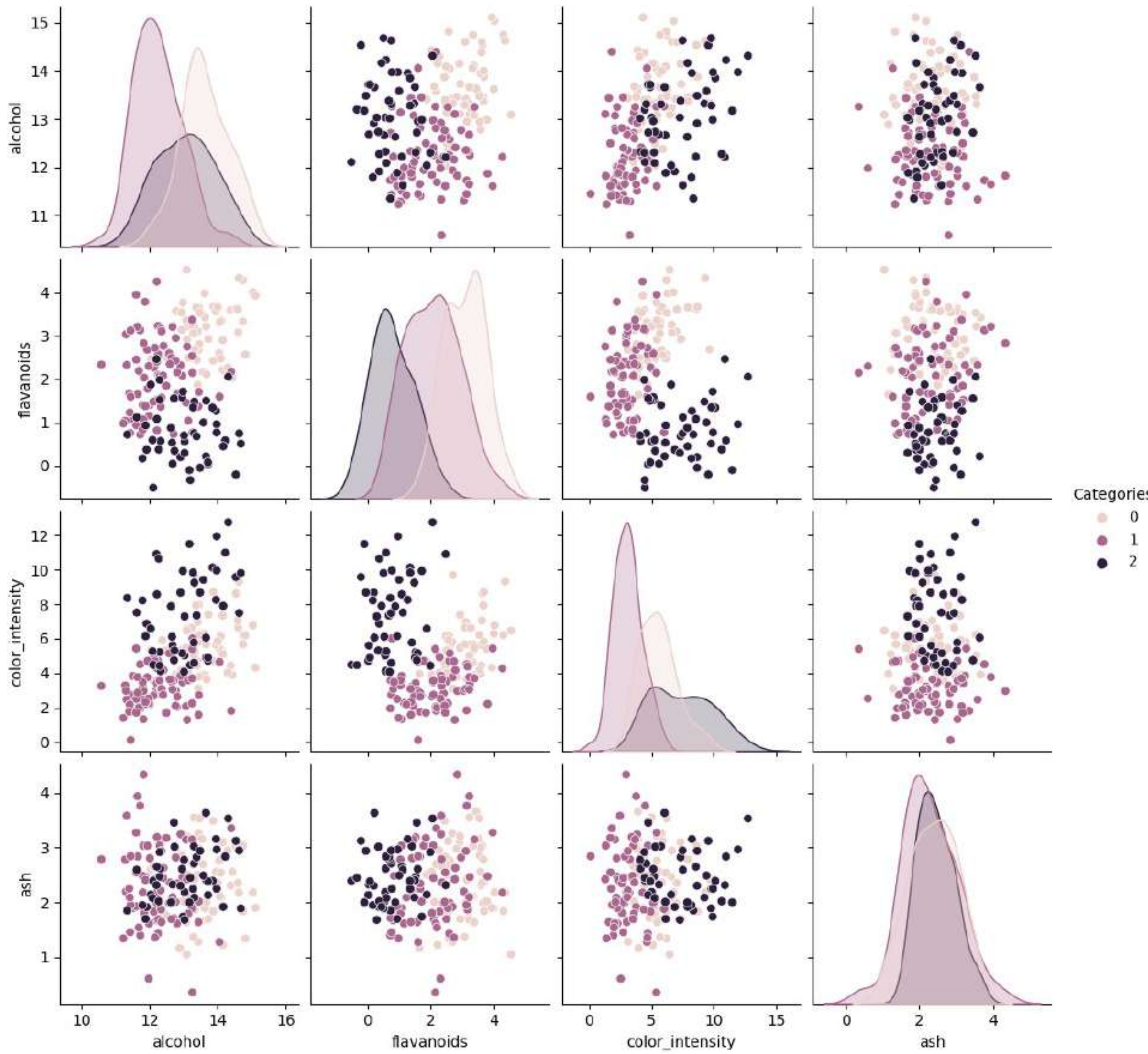


Exploratory Data Analysis under noise

When data is collected under real-world settings they usually contain some amount of noise that makes classification more challenging. Here we introduce some noise to the dataset.

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

# Plotting noisy data
myplotGrid(XN,y,selected_features)
```

```
<seaborn.axisgrid.PairGrid at 0x7f93355f9420>
```

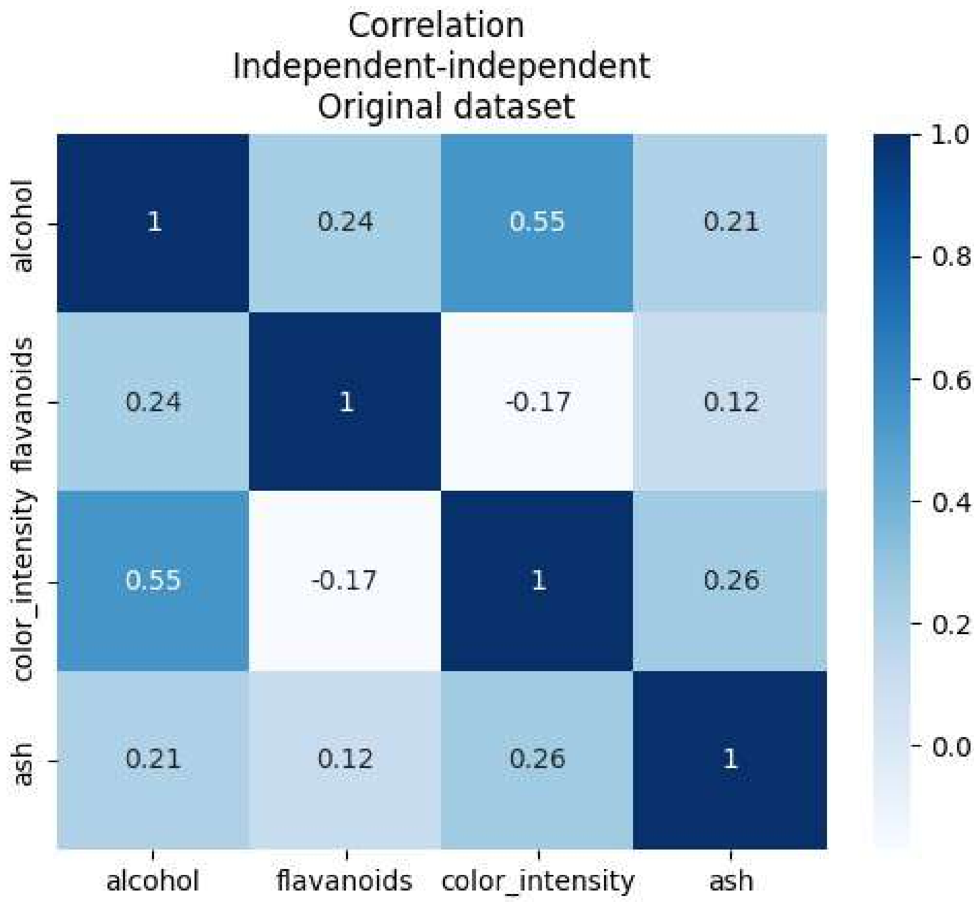


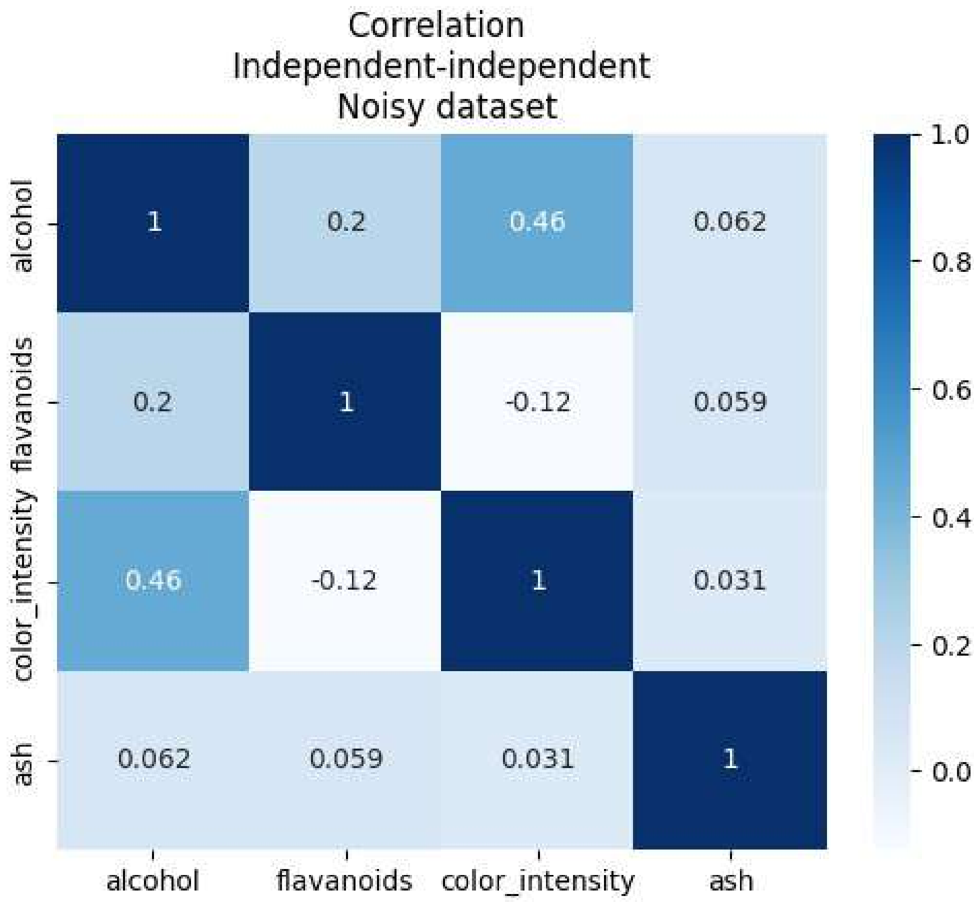
Exploratory data analysis

Here we will visualise correlations amoungst the variables.

```
# Helper functions

# plotHeatMap
def plotHeatMap(matrix, x_labels, y_labels, title, xlabel = None, 
ylabel = None):
    """
    Plots a heat map of a matrix.

    Args:
        matrix (array-like) 2D: 2D form of matrix.
        x_labels (array-like): Lables for the x axis
```

```python
        y_labels (array-like): Lables for the y axis
        title (str): Title of the plot
        xlabel (str): Title for the x labels, default = None
        ylabel (str): Title for the y labels, default = None

    Returns:
        Nothing
    """
    sns.heatmap(matrix, annot=True, cmap='Blues',
xticklabels=x_labels, yticklabels=y_labels)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.show()

# First we will create dataframes of the original and noisy data
X_df = pd.DataFrame(X, columns=selected_features)
XN_df = pd.DataFrame(XN, columns=selected_features)

# 1. Independent-independent correlation

# Corrolation between features of original dataset
correlations_original = X_df.corr()
plotHeatMap(correlations_original, correlations_original.columns,
correlations_original.columns, "Correlation \n Independent-independent
\n Original dataset")

# Corrolation between features of noisy dataset
correlations_noisy = XN_df.corr()
plotHeatMap(correlations_noisy, correlations_noisy.columns,
correlations_noisy.columns, "Correlation \n Independent-independent \n
Noisy dataset")
```


Correlation
Independent-independent
Original dataset
alcohol
flavanoids
color_intensity
ash
1
0.24
0.55
0.21
0.24
1
-0.17
0.12
0.55
-0.17
1
0.26
0.21
0.12
0.26
1
1.0
0.8
0.6
0.4
0.2
0.0



From the plots we can see the difference between the original and noisy data isn't really effecting the correlation analysis. So we could use either one in determining which variables to use.

E.G. We can notice that (color_intensity,alcohol) are the higher correlations, meaning they more dependent to eachother than the rest, so if this is one of the options, we would prefer the latter.

```
# 2. Independent-dependent correlation

# Corrolation between independent and dependent variables in the noisy dataset
XN_df_full = XN_df.copy()
XN_df_full['Categories'] = y
print("===============\nCorrelation Independent-dependent Noisy dataset")
pd.DataFrame(XN_df_full.corr()['Categories'])

===============
Correlation Independent-dependent Noisy dataset

                  Categories
alcohol            -0.268099
```

```
flavanoids        -0.764258
color_intensity    0.252048
ash                0.036660
Categories         1.000000
```

From this analysis we can see that 'flavanoids' will defiantly be one of our variables as it shows a high correlation. And there is a very close second between 'alcohol' and 'color_intensity'.

Due to our results we have two possible feature sets. Where 'flavanoids' is our constant

- (flavanoids, alcohol)
- (flavanoids, color_intensity)

The difference between 'alcohol' and 'color_intensity' in the independent-dependent correlation is 0.016 (at my run: seed 12345). The difference between 'alcohol' and 'color_intensity' in the independent-independent correlation against 'flavanoids' is 0.094 (at my run: seed 12345).

As the differences are more prominent in the independent-independent correlations, this is the values i will let decide.

Conclusion: We will opt for: (flavnoids, color_intensity)

Data with noise

Comparing data with and without noise.

```
# Here we plotting the variance for each variable in the Original and noisy dataset
data = {
    "Original" : X_df.var(),
    "Noisy" : XN_df.var()
}
pd.concat(data, axis = 1)
```

	Original	Noisy
alcohol	0.659062	0.901877
flavanoids	0.997719	1.327416
color_intensity	5.374449	5.976439
ash	0.075265	0.426641

- There wasnt much of a difference in the correlation comparison.
- There is visually a big difference, where visualising 'groups' of the categories is much more difficult from the plotted graph
- There is a larger spread, being indicated on the Gaussian distributions, this can be seen in the diagonal histograms, as the peaks arent as sharp, as well as comparing the variance we notice all variables get increased.

Implementing kNN

Here we implement our one instance of a KNN algorithm.

```
# helper code

# Euclidean distance
def euclidean_distances(vector_1, vector_2):
    """
    Calculates the distance of two vectors using euclidean method.

    Args:
        vector_1 (array-like): Array of vector.
        vector_2 (array-like): Array of vector.

    Returns:
        A float
    """
    return np.sqrt(sum(np.square(vector_1 - vector_2)))

# Manhattan distance
def manhattan_distances(vector_1, vector_2):
    """
    Calculates the distance of two vectors using manhattan method.

    Args:
        vector_1 (array-like): Array of vector.
        vector_2 (array-like): Array of vector.

    Returns:
        A float
    """
    return sum(np.abs(v_1 - v_2) for v_1, v_2 in
zip(vector_1,vector_2))

# highest_count
def highest_count(neighbors_list):
    """
    Counts the highest occurence of a dependent variable.

    Args:
        neighbors_list (array-like): Array of objects of the form: {
            'dependent_variable': ...,
            'distance': 0.00
        }

    Returns:
        highest dependent variables (array-like)
    """
```

```
    occurrences = {}
    # Totalling up the dependent variables distances
    for neighbor in neighbors_list:
        if not neighbor['dependent_variable'] in occurrences:
            occurrences[neighbor['dependent_variable']] = 1
        else:
            occurrences[neighbor['dependent_variable']] += 1
    # Finding what dependent variable occured the most
    max_occured_dependent_variable = max(occurrences,
key=occurrences.get)
    # Finding all the max occured dependent variables
    max_occured_dependent_variables = []
    # Populating the max occurences
    for key in occurrences.keys():
        if occurrences[key] ==
occurrences[max_occured_dependent_variable]:
            max_occured_dependent_variables.append(key)
    return max_occured_dependent_variables

# lowest_distance
def lowest_distance(neighbors_list):
    """
    Totals the distances and returns the lowest dependent variable.

    Args:
        neighbors_list (array-like): Array of object of the form: {
            'dependent_variable': ...,
            'distance': 0.00
        }.

    Returns:
        A dependent variable with the lowest distance
    """
    totals = {}
    # Totalling up the dependent variables distances
    for neighbor in neighbors_list:
        if not neighbor['dependent_variable'] in totals:
            totals[neighbor['dependent_variable']] =
neighbor['distance']
        else:
            totals[neighbor['dependent_variable']] +=
neighbor['distance']
    # Return the smallest distance variable
    return min(totals, key=totals.get)

# find_who_majority
def find_who_majority(train_y, neighbors_to_consider):
    """
    Finds the majority variable from an array, though only considering
```

```
a set of neighbors.

    Args:
        train_y (array-like): Array of the dependent variables
        neighbors_to_consider (array-like): Array of dependent 
variables to consider

    Returns:
        A dependent variable with the highest occurence
    """
    value_counts = {}
    # Totalling the occurences of all variables
    for value in train_y:
        if value in value_counts:
            value_counts[value] += 1
        else:
            value_counts[value] = 1
    # Sorting the results decending
    sorted_counts = dict(sorted(value_counts.items(), key=lambda x: 
x[1], reverse=True))
    # Iterating through results to find the highest
    for key in sorted_counts.keys():
        if key in neighbors_to_consider:
            return key

# mykNN code
def mykNN(X,y,X_,K=4, distance='euclidian', even_decider='distant'):
    """
    Uses KNN brute force, to predict dependent variables

    Args:
        X (array-like): Train_Xs Array of vector.
        y (array-like): Train_ys Array of vector.
        X_ (array-like): Test_Xs Array of vector.
        K (int): Neighbors to consider, default = 4
        distance (str): Distance metric to use, options = 
['euclidean', 'manhattan'], default = 'euclidean'
        even_decider (string): This is the method in deciding which 
variable in the case there is an even count of neighbors, options = 
['distant', 'majority'], default = 'distant'

    Returns:
        Predictions (array-like)
    """
    # Initialising the predicted result
    predictions = []
    for train_value in X_:
        # Initialising the neighbors result
        neighbors = []
        for train_i, test_value in enumerate(X):
```

```
            # Calculating the distance between vectors
            if(distance == 'manhattan'):
                neighbor_distance = manhattan_distances(train_value, 
test_value)
            else:
                neighbor_distance = euclidean_distances(train_value, 
test_value)
            # Recording the neighbor data point and calculated 
distance
            neighbor_object = {
                'dependent_variable': y[train_i],
                'distance': neighbor_distance
            }
            # Handling while neighbors havent reached the max 
neighbors
            if len(neighbors) < K:
                neighbors.append(neighbor_object)
            else:
                # Checking if there exists a neighbor that has a 
greater distance
                for neighbor_i, neighbor in enumerate(neighbors):
                    if neighbor['distance'] > neighbor_distance:
                        # Replace neighbor
                        neighbors[neighbor_i] = neighbor_object
                        break
        # Calculate the most prominent neighbor
        highest_count_neighbor = highest_count(neighbors)
        # Handle if there isnt a single priminent neighbor
        if len(highest_count_neighbor) > 1:
            if even_decider == 'majority':
                predictions.append(find_who_majority(y, 
highest_count_neighbor))
            else:
                predictions.append(lowest_distance(neighbors))
        else:
            predictions.append(highest_count_neighbor[0])
    # Checking the length of the predictions is valid
    if len(X_) != len(predictions):
        raise Exception("something went wrong")
    return predictions
```

Classifier evaluation

Here we will create the tools for evaluating our model

```
# Confusion matrix functions
```

```python
# Confusion matrix
def confusionMatrix(predicted_values, true_values,
dependent_variables):
    """
    Creates a confusion matrix.

    Args:
        predicted_values (array-like): Array of predicted values.
        true_values (array-like): Array of true values.
        dependent_variables (array-like): Array of dependent
variables.

    Returns:
        confustion matrix (array-like) 2D
    """
    # Initialise the confustion matrix with 0's
    confusion_matrix = np.zeros((len(dependent_variables),
len(dependent_variables)))
    # Iterate over each value and increment the matrix
    for true_value, predicted_value in zip(true_values,
predicted_values):
        true_index = dependent_variables.index(true_value)
        pred_index = dependent_variables.index(predicted_value)
        confusion_matrix[true_index][pred_index] += 1
    return confusion_matrix

# Normalise confusion matrix
def normalise_confusion_matrix(confusion_matrix):
    """
    Normalises a confusion matrix to be values between 0 - 1.

    Args:
        confustion_matrix (array-like) 2D: 2D form of confusion
matrix.

    Returns:
        normalised confustion matrix (array-like) 2D
    """
    # Calculating the sum of individual rows
    row_sums = np.sum(confusion_matrix, axis=1)
    # Dono if this is a correct hack???
    row_sums[row_sums == 0] = 1
    # print("row_sums",row_sums)
    # Reshaping result from 1d to 2d
    row_sums_reshaped = row_sums.reshape(-1, 1)
    # print("row_sums_reshaped",row_sums_reshaped)
    # print("confusion_matrix",confusion_matrix)
    return confusion_matrix / row_sums_reshaped
```

```
# Evaluating functions

# Calculate Precision
def calculate_precision(confusion_matrix, dependent_variable_index):
    """
    Calculate precision of a dependent variable in a confusion matrix

    Args:
        confusion_matrix (array-like) 2D: 2D form of confusion matrix.
        dependent_variable_index (int): The index of the dependent
variable to be calculated.

    Returns:
        float: Precision value.
    """
    # Getting the TP being the diagonal
    true_positives = confusion_matrix[dependent_variable_index]
[dependent_variable_index]
    # Getting the false positives being the rows
    false_positives = np.sum(confusion_matrix[:,
dependent_variable_index]) - true_positives
    # Calculating precision
    if((true_positives + false_positives) == 0):
        return 0
    return true_positives / (true_positives + false_positives)

# Calculate Recall
def calculate_recall(confusion_matrix, dependent_variable_index):
    """
    Calculate recall of a dependent variable in a confusion matrix

    Args:
        confusion_matrix (array-like) 2D: 2D form of confusion matrix.
        dependent_variable_index (int): The index of the dependent
variable to be calculated.

    Returns:
        float: Recall value.
    """
    # Getting the TP being the diagonal
    true_positives = confusion_matrix[dependent_variable_index]
[dependent_variable_index]
    # Getting the false negatives being the columns
    false_negatives =
np.sum(confusion_matrix[dependent_variable_index, :]) - true_positives
    # Calculating recall
    if((true_positives + false_negatives) == 0):
        return 0
    return true_positives / (true_positives + false_negatives)
```

```
# Calculate F1
def calculate_f1(confusion_matrix, dependent_variable_index):
    """
    Calculate F1 of a dependent variable in a confusion matrix

    Args:
        confusion_matrix (array-like) 2D: 2D form of confusion matrix.
        dependent_variable_index (int): The index of the dependent
variable to be calculated.

    Returns:
        float: F1 value.
    """
    # Getting precision score
    precision = calculate_precision(confusion_matrix,
dependent_variable_index)
    # Getting recall score
    recall = calculate_recall(confusion_matrix,
dependent_variable_index)
    # Calculating recall
    if((precision + recall) == 0):
        return 0
    return 2 * (precision * recall) / (precision + recall)

# Calculate Accuracy
def calculate_accuracy(confusion_matrix):
    """
    Calculate Accuracy of a confusion matrix

    Args:
        confusion_matrix (array-like) 2D: 2D form of confusion matrix.

    Returns:
        float: Accuracy value.
    """
    # Getting the total of the values along the diagonal
    true_prodictions = np.trace(confusion_matrix)
    # Getting the total of the values in the matrix
    total_predictions = np.sum(confusion_matrix)
    # Calculating recall
    if(total_predictions == 0):
        return 0
    return true_prodictions / total_predictions

# Split a dataset
def split_dataset(independent_variables, dependent_variables,
test_size = 0.2, seed = 0):
    """
    Splits a dataset into a train and test set
```

```
    Args:
        independent_variables (array-like): X.
        dependent_variables (array-like): Y.
        test_size (double): The percentage of the set to be test set
        seed (int): The seed value for the random permutation

    Returns:
        tuple: X_train, X_test, y_train, y_test
    """
    # Generating a random permutation of indicies per the seed
    np.random.seed(seed)
    indexes = np.random.permutation(len(independent_variables))
    # Calculate the splitting position amoungst the indexes
    splitting_index = int(len(independent_variables) * (1 -
test_size))
    # Shuffel the arrays
    independent_variables_shuffled = independent_variables[indexes]
    dependent_variables_shuffled = dependent_variables[indexes]
    # Create the return sets
    X_train = independent_variables_shuffled[:splitting_index]
    X_test = independent_variables_shuffled[splitting_index:]
    y_train = dependent_variables_shuffled[:splitting_index]
    y_test = dependent_variables_shuffled[splitting_index:]

    return (X_train, X_test, y_train, y_test)

# Scale value
def scale_value(value, min, max):
    """
    Scales a value between 0 - 1.

    Args:
        value (double): The value to be scaled.
        min (double): The min value of the class
        max (double): The max value of the class

    Returns:
        Scaled value (double)
    """
    range = max - min
    scaled_value = (value - min) / range
    return scaled_value

# Scale independent variables
def scale_independent_variables(independent_variables):
    """
    Scales values in a 2d array, of independent variables to values
between 0 - 1.

    Args:
```

```
        independent_variables (array-like) 2D: The independent
variables to scale.

    Returns:
        Scaled independent variables (array-like)
    """
    # Initialise the dimensions
    dimensions = len(independent_variables[0])
    # Find the min and max for each variable
    min_max_s = []
    for i in range(dimensions):
        elements = [data_point[i] for data_point in
independent_variables]
        min_value = min(elements)
        max_value = max(elements)
        min_max_s.append((min_value, max_value))
    # Scale each data point
    for index_1 in range(len(independent_variables)):
        for index_2 in range(len(independent_variables[index_1])):
            independent_variables[index_1][index_2] =
scale_value(independent_variables[index_1][index_2],
min_max_s[index_2][0], min_max_s[index_2][1])
    return independent_variables
```

Evaluate our model

Lets use our model to predict

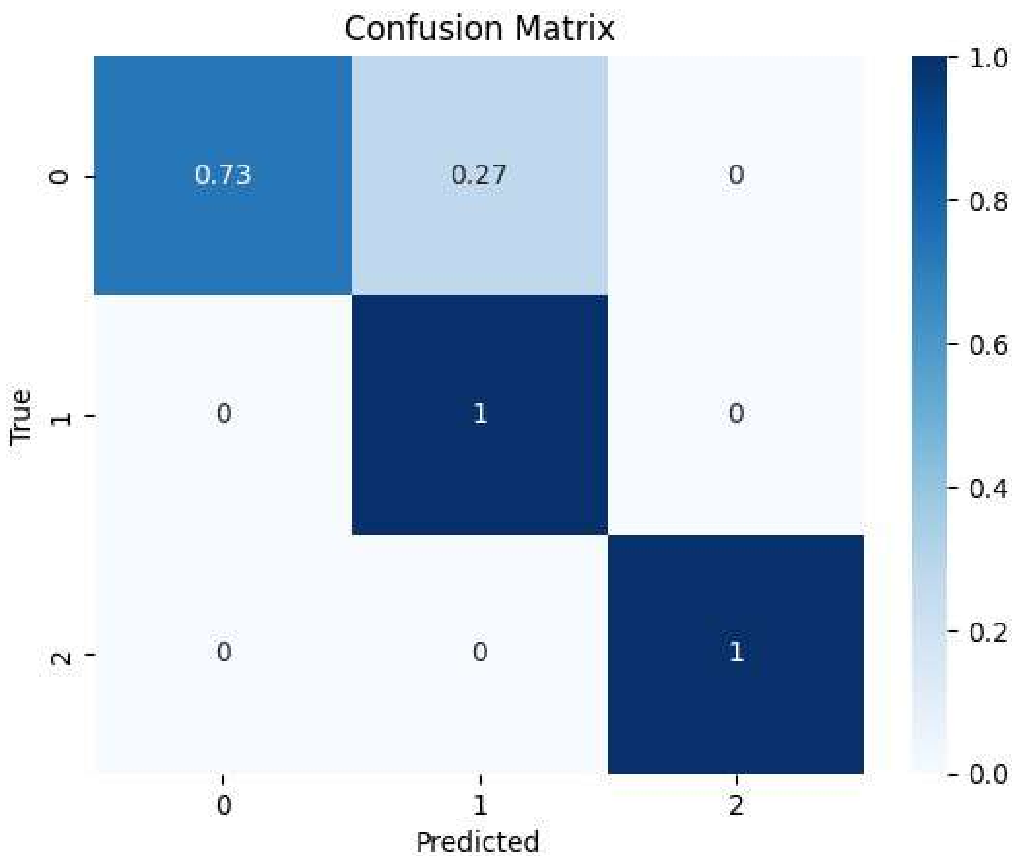
```
# First we should normalise our data to prevent bias amoungst
independent variables
X_scaled = scale_independent_variables(X)
XN_scaled = scale_independent_variables(XN)

# First we need to split our dataset
X_train, X_test, y_train, y_test = split_dataset(XN_scaled, y,
test_size = 0.2, seed = 5)

# We can now use our model to predict
predicted_ys = mykNN(X_train, y_train, X_test, K=6,
distance='euclidian', even_decider='majority')
```

Lets evaluate our model

```
# First we can create a confusion matrix with the results
confusion_matrix = confusionMatrix(predicted_ys, y_test, [0,1,2])
# We will normalise these results
normalised_confusion_matrix =
normalise_confusion_matrix(confusion_matrix)
# Lets see our accuracy
```

```
print("myKNN accuracy: ",
calculate_accuracy(normalised_confusion_matrix))
# Lets plot the confusion matrix in a heat map
plotHeatMap(normalised_confusion_matrix, [0,1,2], [0,1,2], 'Confusion
Matrix', xlabel = 'Predicted', ylabel = 'True')
```

```
myKNN accuracy:  0.9090909090909092
```

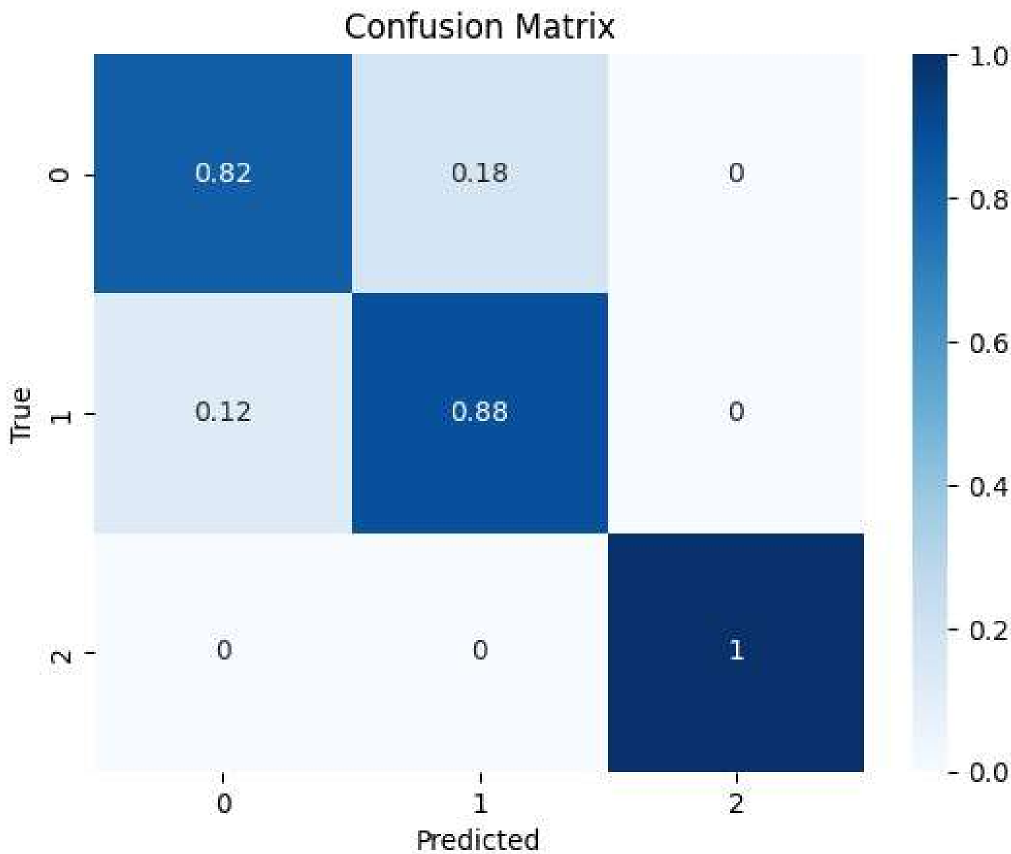


Compare our model

Lets compare our model with sklearns model

```
# Importing modules
from sklearn.neighbors import KNeighborsClassifier

# Fit and predict with sklearns model
neigh = KNeighborsClassifier(n_neighbors=6, metric='euclidean')
neigh.fit(X_train, y_train)
sk_predicted_ys = neigh.predict(X_test)

# First we can create a confusion matrix with the results
sk_confusion_matrix = confusionMatrix(sk_predicted_ys, y_test,
```

```
[0,1,2])
# We will normalise these results
sk_normalised_confusion_matrix =
normalise_confusion_matrix(sk_confusion_matrix)
# Lets see sk knn accuracy
print("SK_knn accuracy: ",
calculate_accuracy(sk_normalised_confusion_matrix))
# Lets plot the confusion matrix in a heat map
plotHeatMap(sk_normalised_confusion_matrix, [0,1,2], [0,1,2],
'Confusion Matrix', xlabel = 'Predicted', ylabel = 'True')

SK_knn accuracy:  0.8977272727272728
```



Conclusion

To conclude what we have noticed comparing our knn model to sklearns model

Our model seems simular but different in terms of an accuracy reading, this is likely to be due to how we have handled the colliding count of neighbors. We have chosen to have our decision based on counting the closest neighbors, and what we call the 'even_decider' being a method used when there is more than one neighbor who wins the count.

- even_decider = 'majority' This method calculates which dependent_variable of the winners have the majority in the entire dataset

- even_decider = 'distant' This method calculates which dependent_variables of the winners are closest to the new data-point

Nested Cross-validation

Here we will perform cross-validation to further evaluate and train our model.

```
# myNestedCrossVal code
def myNestedCrossVal(X, y, outer_folds, inner_folds, k_neighbors,
distances, even_deciders, seed):
    """
    Perform nested cross validation on dataset, using myKNN algorithm

    Args:
        X (array-like): All independent variables.
        y (array-like): All dependent variables.
        outer_folds (int): Number of outer folds
        inner_folds (int): Number of parameter inner folds
        k_neighbors (array-like): Different neighbor counts to
evaluate against
        distances (array-like): Distance metric to evaluate, options =
['euclidean', 'manhattan']
        even_deciders (array-like): This is the methods in deciding
which variable in the case there is an even count of neighbors,
options = ['distant', 'majority']
        seed (int): The seed value for the random permutation

    Returns:
        Tuple as below:

        (array-like) objects containing the results of each outer
fold, in the form: {
            fold: (int),
            accuracy: (double),
            'confusion_matrix': (2D array)
            k: (int),
            distance: (str),
            even_decider: (str)
        }
        (double) mean of accuracies
        (double) std of accuracies
    """
    # Create instance to hold the results from outer folds
    results_outer_fold = [] # array of -> {fold: 1, accuracy: 30.3, k:
3, 'confusion_matrix': (2d array), distance: 'manhattan',
even_decider: 'majority'} -> for each fold
    # Randomise the indicies according to seed
    np.random.seed(seed)
```

```
    entire_fold_outer_indicies_X = np.random.permutation(len(X))
    entire_fold_outer_indicies_Y = entire_fold_outer_indicies_X
    # Create outer fold indices
    outer_fold_start_end_indicies = []
    spaces_outer = int((len(entire_fold_outer_indicies_X) /
outer_folds) // 1)
    for i in range(outer_folds):
        start_index = i * spaces_outer
        end_index = (i * spaces_outer) + (spaces_outer - 1)
        outer_fold_start_end_indicies.append((start_index,end_index))
    # For each outer fold
    for outer_fold_start_end_index_i, outer_fold_start_end_index in
enumerate(outer_fold_start_end_indicies):
        # Create instance to hold the results from inner fold
        results_inner_fold_average = [] # array of ->
{accuracy_average: 30.3, k: 3, distance: 'manhattan', even_decider:
'majority'} -> average for each combination
        # Construct fold indicies
        indicies_to_exclude = np.arange(outer_fold_start_end_index[0],
outer_fold_start_end_index[1] + 1)
        entire_fold_inner_indicies_X =
entire_fold_outer_indicies_X[~np.isin(np.arange(entire_fold_outer_indi
cies_X.shape[0]), indicies_to_exclude)]
        entire_fold_inner_indicies_Y =
entire_fold_outer_indicies_Y[~np.isin(np.arange(entire_fold_outer_indi
cies_Y.shape[0]), indicies_to_exclude)]
        # Create inner fold indices
        inner_fold_start_end_indicies = []
        spaces_inner = int((len(entire_fold_inner_indicies_X) /
inner_folds) // 1)
        for i in range(inner_folds):
            start_index = i * spaces_inner
            end_index = (i * spaces_inner) + (spaces_inner - 1)

inner_fold_start_end_indicies.append((start_index,end_index))
        # For each property in k_neighbors
        for k in k_neighbors:
            # For each property in distances
            for distance in distances:
                # For each property in even_deciders
                for even_decider in even_deciders:
                    # Create instance to record average from
inner_folds
                    results_inner_fold = [] # array of -> values
(doubles)
                    # For each inner fold
                    for inner_fold_start_end_index in
inner_fold_start_end_indicies:
                        # Construct fold indicies for train and
```

```
evaluate sets
                        indicies_to_exclude =
np.arange(inner_fold_start_end_index[0], inner_fold_start_end_index[1]
+ 1)
                        train_set_X =
entire_fold_inner_indicies_X[~np.isin(np.arange(entire_fold_inner_indi
cies_X.shape[0]), indicies_to_exclude)]
                        train_set_Y =
entire_fold_inner_indicies_Y[~np.isin(np.arange(entire_fold_inner_indi
cies_Y.shape[0]), indicies_to_exclude)]
                        evaluate_set_X =
entire_fold_inner_indicies_X[inner_fold_start_end_index[0] :
inner_fold_start_end_index[1] + 1]
                        evaluate_set_Y =
entire_fold_inner_indicies_Y[inner_fold_start_end_index[0] :
inner_fold_start_end_index[1] + 1]
                        # Train model
                        predicted_ys = mykNN(X[train_set_X],
y[train_set_Y], X[evaluate_set_X],K=k, distance=distance,
even_decider=even_decider)
                        # Generate confusion matrix
                        confusion_matrix =
confusionMatrix(predicted_ys, y[evaluate_set_Y], [0,1,2])
                        normalised_confusion_matrix =
normalise_confusion_matrix(confusion_matrix)
                        # Calculate accuracy
                        accuracy =
calculate_accuracy(normalised_confusion_matrix)
                        # Push result to results_inner_fold
                        results_inner_fold.append(accuracy)
                    # Push average in results_inner_fold_average from
results_inner_fold
                    average_accuracy = sum(results_inner_fold) /
len(results_inner_fold)
                    results_inner_fold_average.append({
                        'accuracy_average': average_accuracy,
                        'k': k,
                        'distance': distance,
                        'even_decider': even_decider
                    })
        # Find the best parameters from results_inner_fold_average
        best_parameters = max(results_inner_fold_average, key=lambda
obj: obj["accuracy_average"])
        # Construct fold indicies for train and test sets
        indicies_to_exclude = np.arange(outer_fold_start_end_index[0],
outer_fold_start_end_index[1] + 1)
        train_set_X =
entire_fold_outer_indicies_X[~np.isin(np.arange(entire_fold_outer_indi
cies_X.shape[0]), indicies_to_exclude)]
```

```
        train_set_Y =
entire_fold_outer_indicies_Y[~np.isin(np.arange(entire_fold_outer_indi
cies_Y.shape[0]), indicies_to_exclude)]
        evaluate_set_X =
entire_fold_outer_indicies_X[outer_fold_start_end_index[0] :
outer_fold_start_end_index[1] + 1]
        evaluate_set_Y =
entire_fold_outer_indicies_Y[outer_fold_start_end_index[0] :
outer_fold_start_end_index[1] + 1]
        # Train model with full dataset and best conbination parameter
results from results_inner_fold_average
        predicted_ys = mykNN(X[train_set_X], y[train_set_Y],
X[evaluate_set_X],K=best_parameters['k'],
distance=best_parameters['distance'],
even_decider=best_parameters['even_decider'])
        # Generate confusion matrix
        confusion_matrix = confusionMatrix(predicted_ys,
y[evaluate_set_Y], [0,1,2])
        normalised_confusion_matrix =
normalise_confusion_matrix(confusion_matrix)
        # Calculate accuracy
        accuracy = calculate_accuracy(normalised_confusion_matrix)
        # Push to results_outer_fold
        results_outer_fold.append({
            'fold': outer_fold_start_end_index_i + 1,
            'accuracy': accuracy,
            'confusion_matrix': normalised_confusion_matrix,
            'k': best_parameters['k'],
            'distance': best_parameters['distance'],
            'even_decider': best_parameters['even_decider']
        })
        # Calculating the mean and standard deviation of folds
        accuracies = [fold['accuracy'] for fold in results_outer_fold]
        accuracy_mean = sum([fold['accuracy'] for fold in
results_outer_fold]) / len(results_outer_fold)
        squared_diff_sum = sum((x - accuracy_mean) ** 2 for x in
accuracies)
        variance = squared_diff_sum / len(results_outer_fold)
        accuracy_std = math.sqrt(variance)
    return (results_outer_fold, accuracy_mean, accuracy_std)

# evaluate clean data code
clean_results, clean_results_mean, clean_results_std =
myNestedCrossVal(X_scaled, y, 5, 5, list(range(1,10)),
['euclidean','manhattan'], ['majority', 'distance'], 4)

# evaluate noisy  data code
noisy_results, noisy_results_mean, noisy_results_std =
myNestedCrossVal(XN_scaled, y, 5, 5, list(range(1,10)),
['euclidean','manhattan'], ['majority', 'distance'], 4)
```

```python
# Print the summaries Clean Data
print("==============================================")
print("Clean data summary")
print("Accuracy Mean:       ", clean_results_mean)
print("Accuracy STD:        ", clean_results_std)
print("----- Results")
for fold in clean_results:
    print("Fold =", fold['fold'], "| K =", fold['k'], "| Distance =", 
fold['distance'], "| Even Decider =", fold['even_decider'], "| 
Accuracy =", fold['accuracy'])
print("==============================================")
```

```
==============================================
Clean data summary
Accuracy Mean:        0.9629222629222628
Accuracy STD:         0.03656766724781932
----- Results
Fold = 1 | K = 6 | Distance = manhattan | Even Decider = majority |
Accuracy = 1.0
Fold = 2 | K = 4 | Distance = manhattan | Even Decider = distance |
Accuracy = 0.9285714285714285
Fold = 3 | K = 8 | Distance = euclidean | Even Decider = distance |
Accuracy = 1.0
Fold = 4 | K = 7 | Distance = manhattan | Even Decider = majority |
Accuracy = 0.9116809116809118
Fold = 5 | K = 4 | Distance = euclidean | Even Decider = distance |
Accuracy = 0.9743589743589745
==============================================
```

```python
# Print the summaries Noisy Data
print("==============================================")
print("Noisy data summary")
print("Accuracy Mean:       ", noisy_results_mean)
print("Accuracy STD:        ", noisy_results_std)
print("----- Results")
for fold in noisy_results:
    print("Fold =", fold['fold'], "| K =", fold['k'], "| Distance =", 
fold['distance'], "| Even Decider =", fold['even_decider'], "| 
Accuracy =", fold['accuracy'])
print("==============================================")
```

```
==============================================
Noisy data summary
Accuracy Mean:        0.88773199023199
Accuracy STD:         0.049730865523894205
----- Results
Fold = 1 | K = 8 | Distance = manhattan | Even Decider = distance |
Accuracy = 0.9209401709401711
Fold = 2 | K = 8 | Distance = euclidean | Even Decider = majority |
Accuracy = 0.798941798941799
```

```
Fold = 3 | K = 6 | Distance = euclidean | Even Decider = distance |
Accuracy = 0.9458333333333333
Fold = 4 | K = 4 | Distance = euclidean | Even Decider = distance |
Accuracy = 0.8860398860398858
Fold = 5 | K = 8 | Distance = euclidean | Even Decider = distance |
Accuracy = 0.8869047619047619
==============================================
```

Summary of results (seed 4)

The results from above:

Fold	accuracy	k	distance	even decider
1	1.0	6	manhattan	majority
2	.93	4	manhattan	distance
3	1.0	8	euclidean	distance
4	.91	7	manhattan	majority
5	.97	4	euclidean	distance
total	.96 ± 0.04			

The results from above (noisy data):

Fold	accuracy	k	distance	even decider
1	.92	8	manhattan	distance
2	.80	8	euclidean	majority
3	.95	6	euclidean	distance
4	.89	4	euclidean	distance
5	.89	8	euclidean	distance
total	.88 ± 0.05			

Confusion matrix summary

Summarise the overall results of your nested cross validation evaluation of your K-NN algorithm using two summary confusion matrices (one for the noisy data, one for the clean data). You might want to adapt your `myNestedCrossVal` code above to also return a list of confusion matrices.

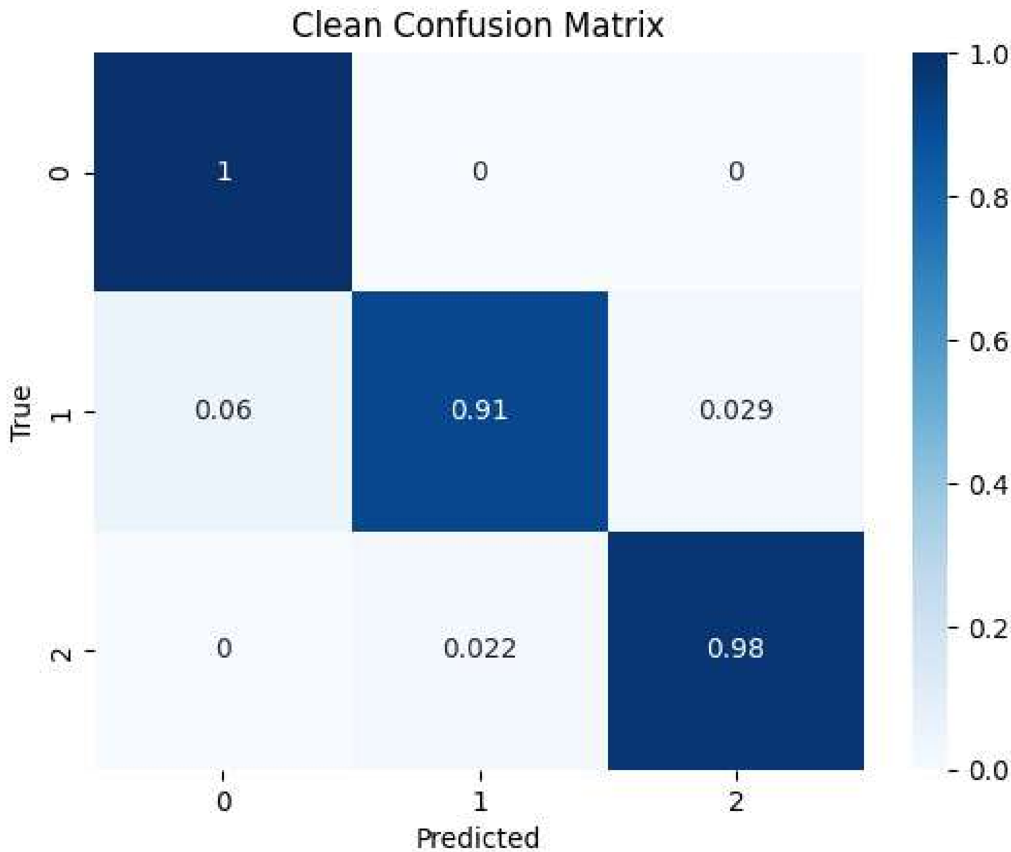
Use or adapt your evaluation code above to print the two confusion matrices below. Make sure you label the matrix rows and columns. You might also want ot show class-relative precision and recall.

```
# Average a set of confusion_matricies
def average_confusion_matricies(confusion_matricies):
    """
    Averages multiple confusion matricies into one matrix
```

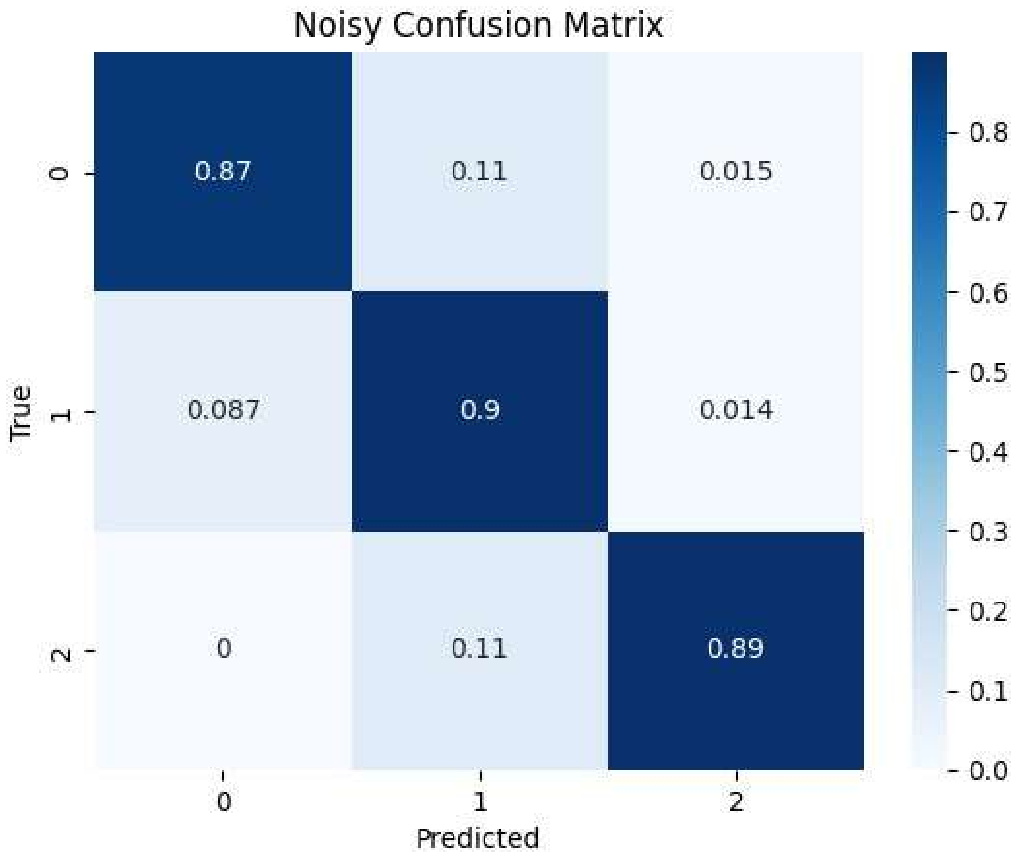
```
    Args:
        confusion_matricies (array-like): An array of multiple
confusion matricies

    Returns:
        confusion_matrix (array-like): Single confusion matrix
    """
    # Initialise a base confusion matrix
    base_matrix = np.zeros(np.shape(confusion_matricies[0]))
    # Totalling up all confusion_matricies
    for confusion_matrix in confusion_matricies:
        base_matrix = [[x + y for x, y in zip(row1, row2)] for row1,
row2 in zip(confusion_matrix, base_matrix)]
    # Averaging the base matrix
    base_matrix = [[element / len(confusion_matricies) for element in
row] for row in base_matrix]
    return base_matrix


print("===============================================")
print('CLEAN')
print("----- Scoring metrics")
# Issolate confusion matricies
clean_confusion_matricies = [fold['confusion_matrix'] for fold in
clean_results]
# Retrieve a average matrix
clean_confusion_matricies_average =
average_confusion_matricies(clean_confusion_matricies)
# Printing precision, recall and f1 score
data = [
    [calculate_precision(np.array(clean_confusion_matricies_average),
0),calculate_recall(np.array(clean_confusion_matricies_average),
0),calculate_f1(np.array(clean_confusion_matricies_average), 0)],
    [calculate_precision(np.array(clean_confusion_matricies_average),
1),calculate_recall(np.array(clean_confusion_matricies_average),
1),calculate_f1(np.array(clean_confusion_matricies_average), 1)],
    [calculate_precision(np.array(clean_confusion_matricies_average),
2),calculate_recall(np.array(clean_confusion_matricies_average),
2),calculate_f1(np.array(clean_confusion_matricies_average), 2)]
]
scoring_metrics = pd.DataFrame(data,
columns=['precision','recall','F1'], index=['Label 0', 'Label 1',
'Label 2'])
print(scoring_metrics)
print("----- Confusion Matrix")
# Plot the average matrix
plotHeatMap(clean_confusion_matricies_average, [0,1,2], [0,1,2],
'Clean Confusion Matrix', xlabel = 'Predicted', ylabel = 'True')
```

```python
print("==============================================")
print('NOISY')
print("----- Scoring metrics")
# Issolate confusion matricies
noisy_confusion_matricies = [fold['confusion_matrix'] for fold in
noisy_results]
# Retrieve a average matrix
noisy_confusion_matricies_average =
average_confusion_matricies(noisy_confusion_matricies)
# Printing precision, recall and f1 score
data = [
    [calculate_precision(np.array(noisy_confusion_matricies_average),
0),calculate_recall(np.array(noisy_confusion_matricies_average),
0),calculate_f1(np.array(noisy_confusion_matricies_average), 0)],
    [calculate_precision(np.array(noisy_confusion_matricies_average),
1),calculate_recall(np.array(noisy_confusion_matricies_average),
1),calculate_f1(np.array(noisy_confusion_matricies_average), 1)],
    [calculate_precision(np.array(noisy_confusion_matricies_average),
2),calculate_recall(np.array(noisy_confusion_matricies_average),
2),calculate_f1(np.array(noisy_confusion_matricies_average), 2)]
]
scoring_metrics = pd.DataFrame(data,
columns=['precision','recall','F1'], index=['Label 0', 'Label 1',
'Label 2'])
print(scoring_metrics)
print("----- Confusion Matrix")
# Plot the average matrix
plotHeatMap(noisy_confusion_matricies_average, [0,1,2], [0,1,2],
'Noisy Confusion Matrix', xlabel = 'Predicted', ylabel = 'True')
```

```
==============================================
CLEAN
----- Scoring metrics
         precision    recall        F1
Label 0   0.943005  1.000000  0.970667
Label 1   0.976187  0.910989  0.942462
Label 2   0.971609  0.977778  0.974684
----- Confusion Matrix
```




```
==============================================
NOISY
----- Scoring metrics
         precision    recall        F1
Label 0   0.909165  0.873040  0.890737
Label 1   0.803372  0.898489  0.848272
Label 2   0.967796  0.891667  0.928173
----- Confusion Matrix
```


Noisy Confusion Matrix
0.87
0.11
0.015
0.087
0.9
0.014
0
0.11
0.89
True
Predicted
0
1
2
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8