Course Topic:

K-nearest Neighbors (KNN) Activity with Iris Dataset

Data Used:

Iris.csv downloaded from Kaggle

Dataset pt.1: Iris Data Set

Source of Data:

https://www.kaggle.com/datasets/uciml/iris

Data Information:

The Iris dataset was used in R.A. Fisher's classic 1936 paper, The Use of Multiple Measurements in Taxonomic Problems, and can also be found on the UCI Machine Learning Repository.

It includes three iris species with 50 samples each as well as some properties about each flower. One flower species is linearly separable from the other two, but the other two are not linearly separable from each other.

The columns in this dataset are:

- Id
- SepalLengthCm
- SepalWidthCm
- PetalLengthCm
- PetalWidthCm
- Species

Problem Statement:

The task is to read and load the provided dataset file into the program. This can be done using appropriate functions or libraries depending on the programming language being used. After loading the dataset, the next step involves handling any missing or null values. It is crucial to ensure the dataset is free from such inconsistencies, as they can introduce errors and affect the accuracy of subsequent analyses. One common approach is to drop the rows or instances that contain null or empty values. This ensures that only complete and reliable data is used for further processing. The next one is to create a distance measurement to assess the similarity of two points in a multidimensional space. This procedure uses Euclidean, Manhattan, and Minkowski.

In this activity, we will employ the following procedures:

1. Create our own distance measurements.

    1.1. Manhattan Distance

    1.2. Minkowski Distance
2. Documentation
    2.1. Document maximum of 10 bugs.

Method:

Algorithm:

We will implement the K-Nearest Neighbors (KNN) classifier algorithm, which is a supervised machine-learning technique suitable for classification and regression tasks. In this case, our focus will be on its application for classification. The KNN algorithm operates by using a set of labeled training data that represents the known class labels of points in the feature space. When a new, unlabeled data point is provided, the algorithm calculates the distance between this point and all the points in the training data. By selecting the K-nearest points based on the chosen distance metric, the algorithm determines the closest neighbors to the given point. Once the K-nearest neighbors are identified, the algorithm assigns the most frequent class label among those K points as the predicted class label for the given point. In simpler terms, it makes a decision by taking a majority vote among its nearest neighbors to determine the class label.

Solution:

To implement the KNN algorithm on the Iris dataset, several preprocessing steps are required to ensure that the data is in a suitable format for training and testing. Firstly, the Iris dataset is loaded using the appropriate function, providing access to the feature matrix (X) and the corresponding class labels (y).

Next, the dataset is split into training and testing sets using the train_test_split function from scikit-learn. This allows for independent evaluation of the trained model's performance on unseen data. It is important to specify the desired proportion of the test set and use a random state for reproducibility.

Depending on the nature of the data and the chosen distance metric, feature scaling may be necessary. This step ensures that all features are on a similar scale and prevents certain features from dominating the distance calculations. Common techniques for feature scaling include standardization, which involves subtracting the mean and dividing by the standard deviation, and normalization, which scales the values to a specific range.

In the case of missing values in the dataset, appropriate handling techniques should be applied. This can involve imputation, where missing values are replaced with estimated values based on other data points, or deletion of instances or features with missing values, depending on the extent and impact of the missing data.

Categorical variables need to be encoded into numerical values. This can be achieved through techniques such as one-hot encoding or label encoding, depending on the nature of the categorical variables and the requirements of the algorithm.

Additional data preprocessing steps may be required based on the specific characteristics of the dataset. This can include feature selection to identify the most relevant features, dimensionality reduction techniques to reduce the number of features, or outlier detection and handling to address any outliers that may exist in the data.

By performing these preprocessing steps, the Iris dataset is appropriately prepared for training the KNN algorithm. This ensures that the algorithm can effectively learn from the data and make accurate predictions on new, unseen instances. The specific preprocessing steps may vary depending on the dataset characteristics and the requirements of the algorithm being applied.

Evaluation:

In evaluating the performance of the K-Nearest Neighbors (KNN) algorithm we use various evaluation metrics. These metrics provide insights into the algorithm's accuracy and confusion matrix. Accuracy is a widely used metric that measures the proportion of correctly classified instances out of the total number of instances in the test set. It provides an overall assessment of how well the algorithm correctly predicts the class labels. A confusion matrix provides a detailed analysis of the classifier's performance by displaying the counts of true positives, true negatives, false positives, and false negatives.

By combining these evaluation metrics, it becomes feasible to conduct a thorough analysis of the model's performance, allowing for a meticulous assessment of its strengths and weaknesses. These metrics will play a crucial role in evaluating the accuracy of the KNN model, especially when considering different distance metrics. The insights gained from this evaluation will serve as valuable information for future optimization and refinement of the model, facilitating its further development.

Data Preparation:

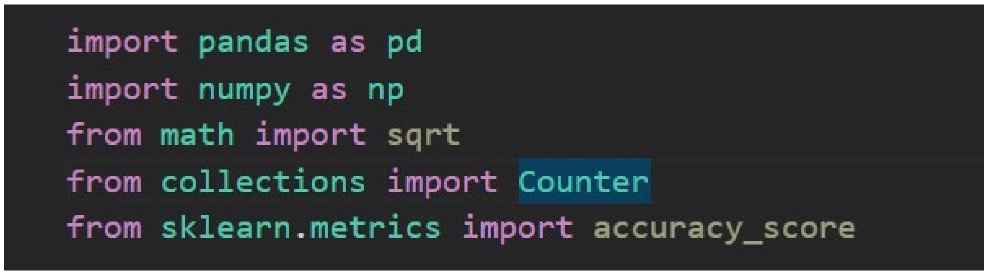
In this activity, these are the following processes we have done in data preparation.

1. Gathering/ loading dataset using the read_csv function
2. Checking for null or inconsistent values in the data.
3. Removing the null values.
4. Changing the needed values to new values

Results & Discussion

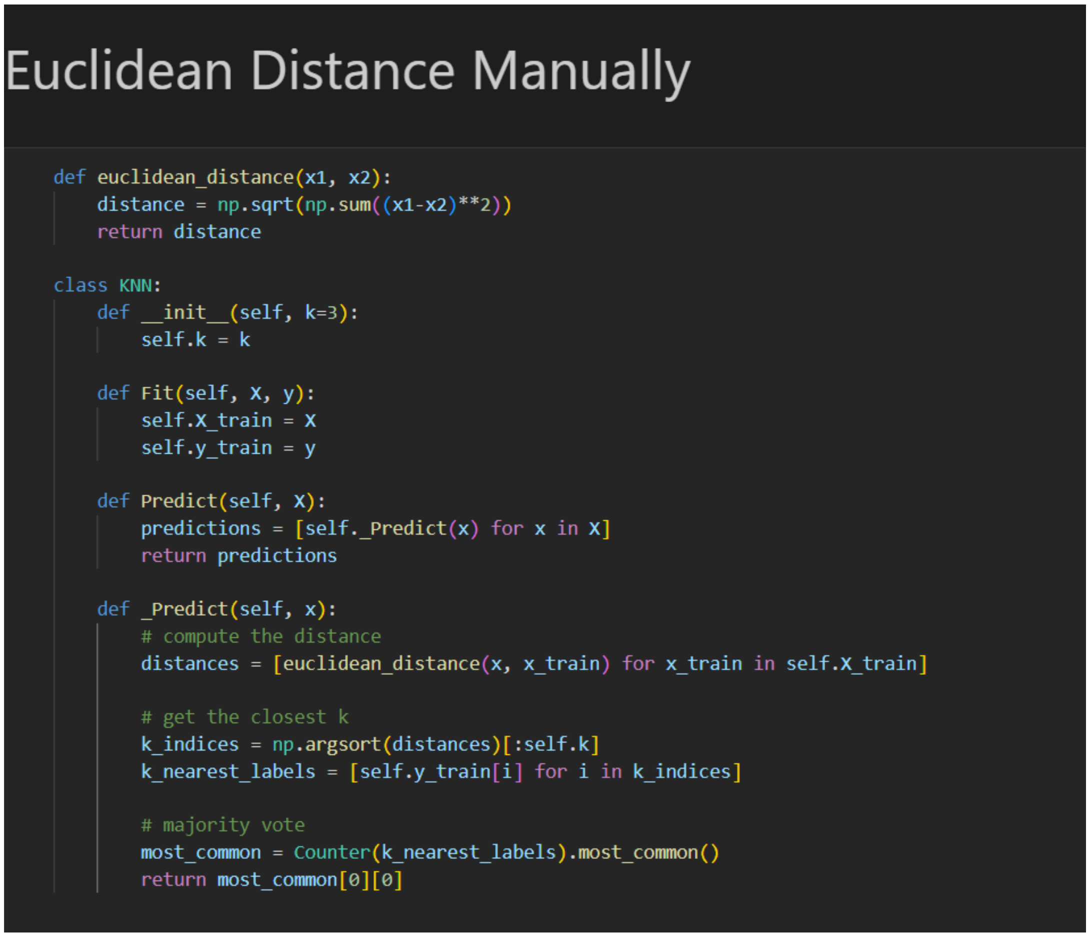
Data cleaning plays a fundamental role in the data analysis process as it ensures the accuracy, reliability, and quality of the data used for analysis. By identifying and addressing errors, inconsistencies, and missing values, data cleaning significantly enhances the integrity and usefulness of the dataset specifically the iris dataset that is used in this activity. First step is to load and clean and split the dataset, after that Euclidean, Manhattan, and Minkowski were used to calculate the test data and the training data.

Screenshots:



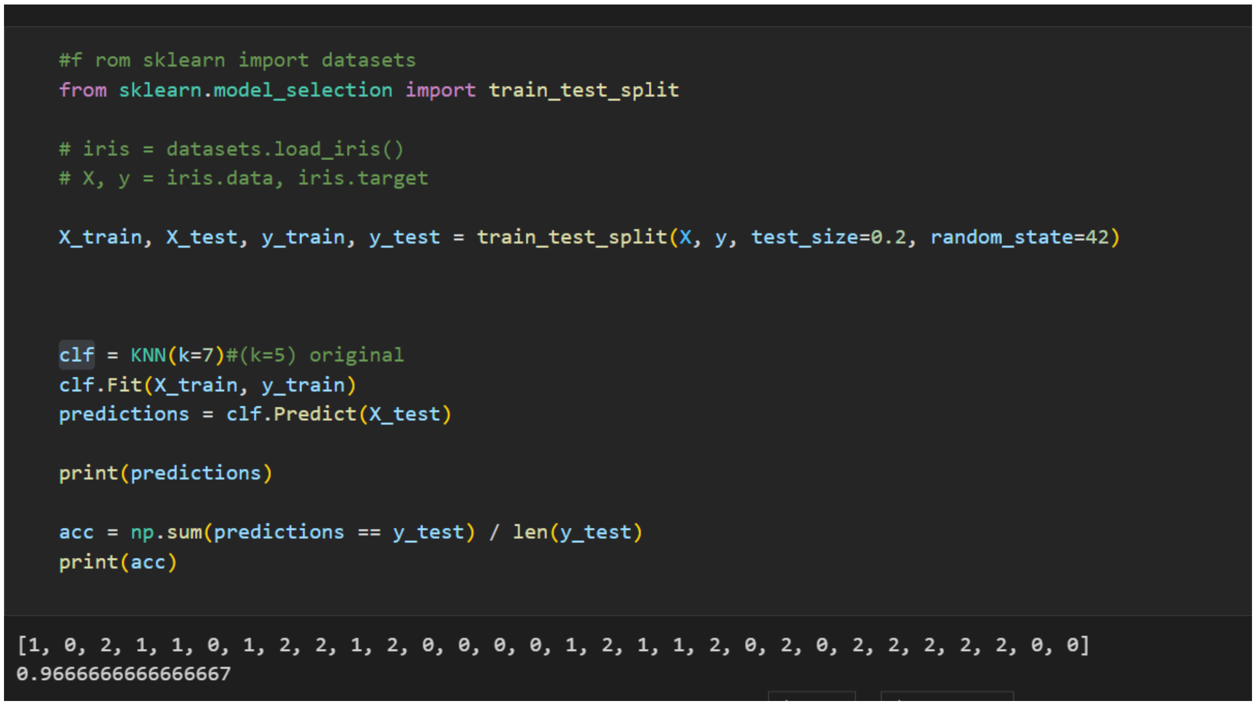
```
import pandas as pd
import numpy as np
from math import sqrt
from collections import Counter
from sklearn.metrics import accuracy_score
```

The code imports the Pandas and NumPy libraries for data manipulation and numerical computations. It also imports the sqrt function from the math module for square root calculations and the Counter class from the collections module for counting frequencies. Additionally, it imports the accuracy_score function from the Scikit-learn library for evaluating classification model accuracy.



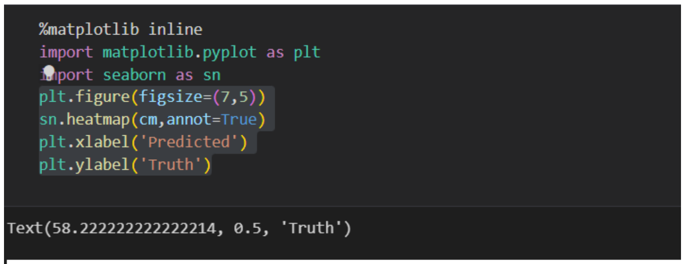
Euclidean Distance Manually

```python
def euclidean_distance(x1, x2):
    distance = np.sqrt(np.sum((x1-x2)**2))
    return distance

class KNN:
    def __init__(self, k=3):
        self.k = k

    def Fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def Predict(self, X):
        predictions = [self._Predict(x) for x in X]
        return predictions

    def _Predict(self, x):
        # compute the distance
        distances = [euclidean_distance(x, x_train) for x_train in self.X_train]

        # get the closest k
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]

        # majority vote
        most_common = Counter(k_nearest_labels).most_common()
        return most_common[0][0]
```



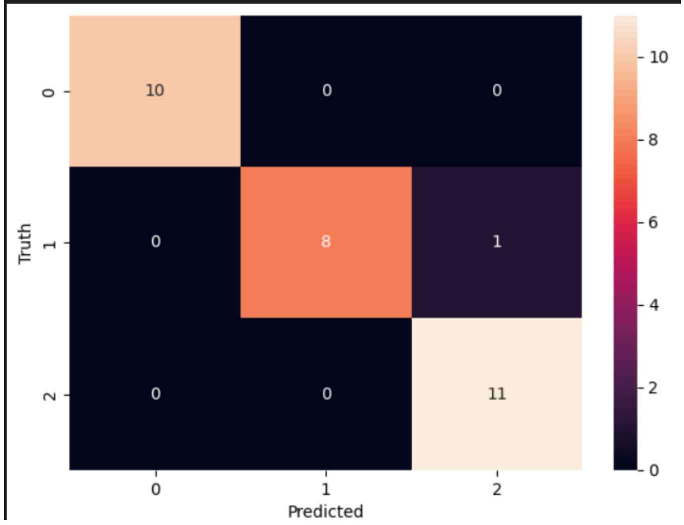
```python
#f rom sklearn import datasets
from sklearn.model_selection import train_test_split

# iris = datasets.load_iris()
# X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)



clf = KNN(k=7)#(k=5) original
clf.Fit(X_train, y_train)
predictions = clf.Predict(X_test)

print(predictions)

acc = np.sum(predictions == y_test) / len(y_test)
print(acc)
```

```
[1, 0, 2, 1, 1, 0, 1, 2, 2, 1, 2, 0, 0, 0, 0, 1, 2, 1, 1, 2, 0, 2, 0, 2, 2, 2, 2, 2, 0, 0]
0.9666666666666667
```

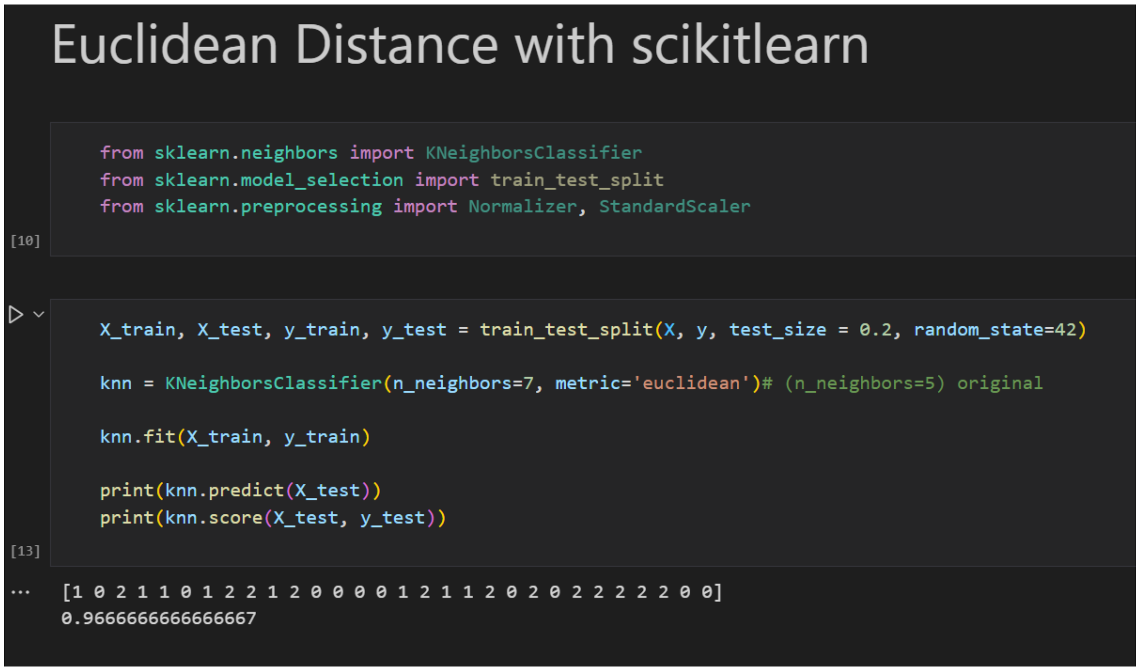


```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sn
plt.figure(figsize=(7,5))
sn.heatmap(cm,annot=True)
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

```
Text(58.222222222222214, 0.5, 'Truth')
```



This code provides a basic implementation of the KNN algorithm for classification, allowing training and prediction on new data with customizable k values.

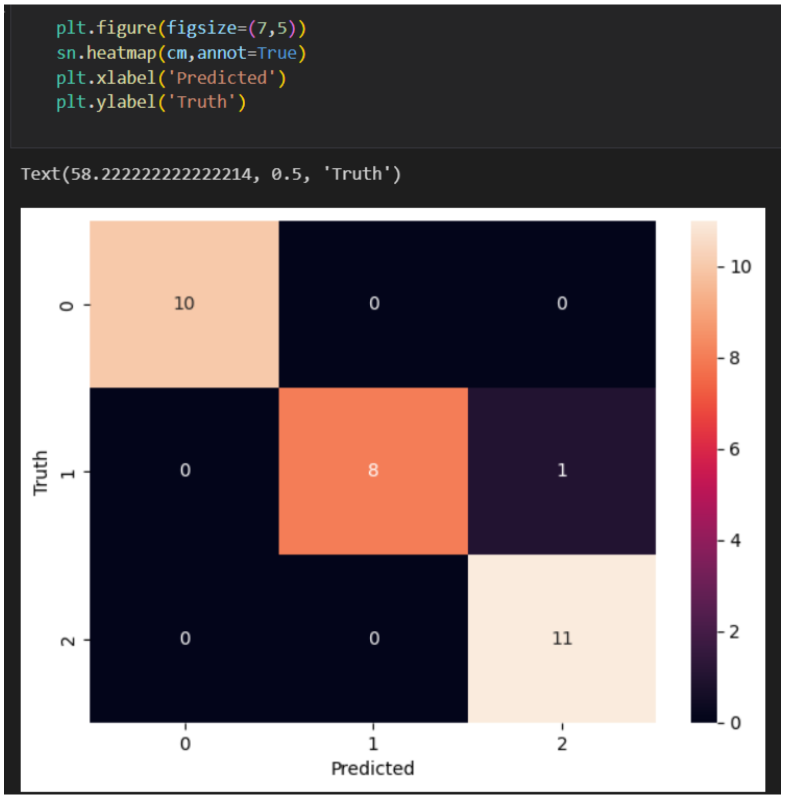


Euclidean Distance with scikitlearn

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import Normalizer, StandardScaler
```
[10]

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42)

knn = KNeighborsClassifier(n_neighbors=7, metric='euclidean')# (n_neighbors=5) original

knn.fit(X_train, y_train)

print(knn.predict(X_test))
print(knn.score(X_test, y_test))
```
[13]

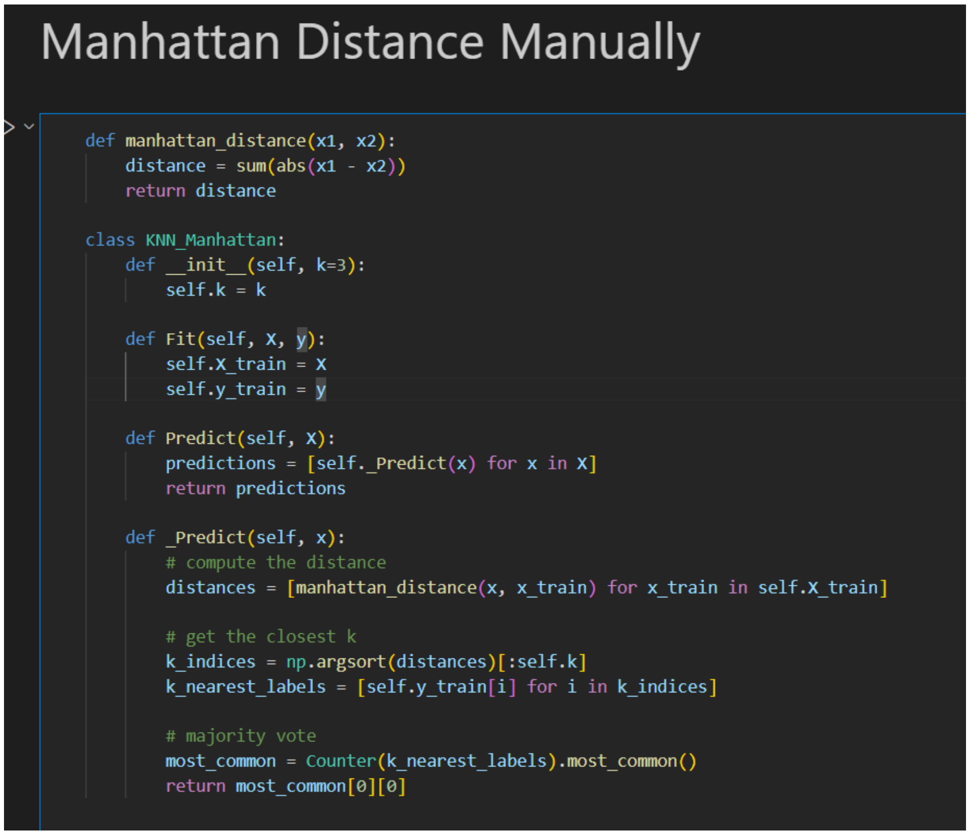
```
[1 0 2 1 1 0 1 2 2 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0]
0.9666666666666667
```



```
plt.figure(figsize=(7,5))
sn.heatmap(cm,annot=True)
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

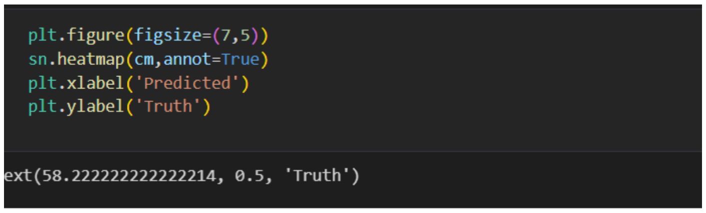
```
Text(58.222222222222214, 0.5, 'Truth')
```

The provided code snippet showcases the usage of scikit-learn for K-nearest neighbors (KNN) classification



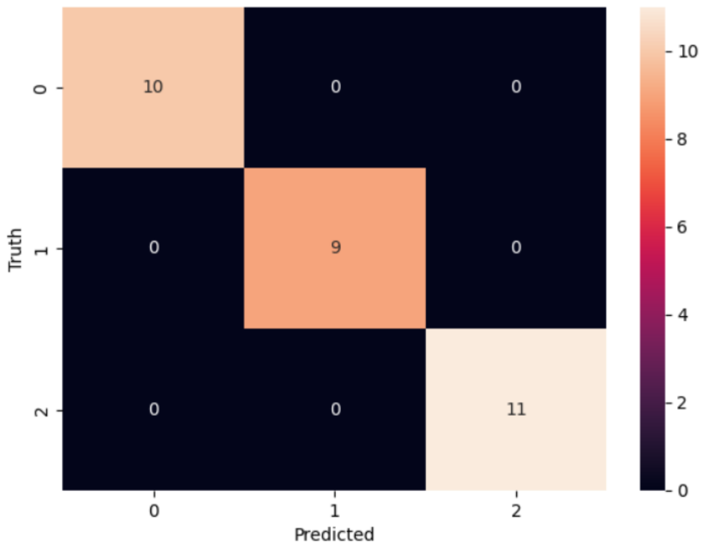
Manhattan Distance Manually

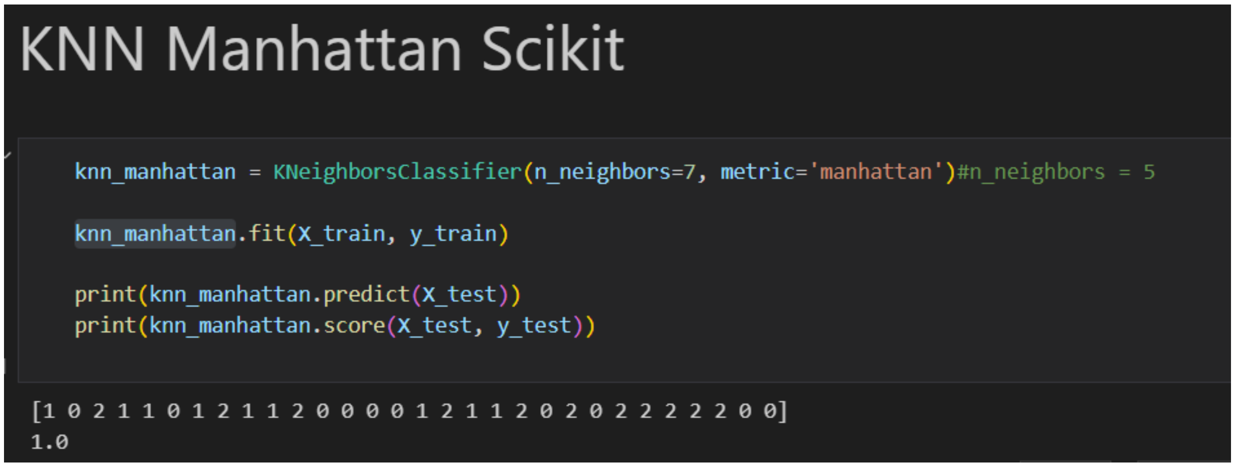
```
def manhattan_distance(x1, x2):
    distance = sum(abs(x1 - x2))
    return distance

class KNN_Manhattan:
    def __init__(self, k=3):
        self.k = k

    def Fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def Predict(self, X):
        predictions = [self._Predict(x) for x in X]
        return predictions

    def _Predict(self, x):
        # compute the distance
        distances = [manhattan_distance(x, x_train) for x_train in self.X_train]

        # get the closest k
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]

        # majority vote
        most_common = Counter(k_nearest_labels).most_common()
        return most_common[0][0]
```



```
plt.figure(figsize=(7,5))
sn.heatmap(cm,annot=True)
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

```
ext(58.222222222222214, 0.5, 'Truth')
```

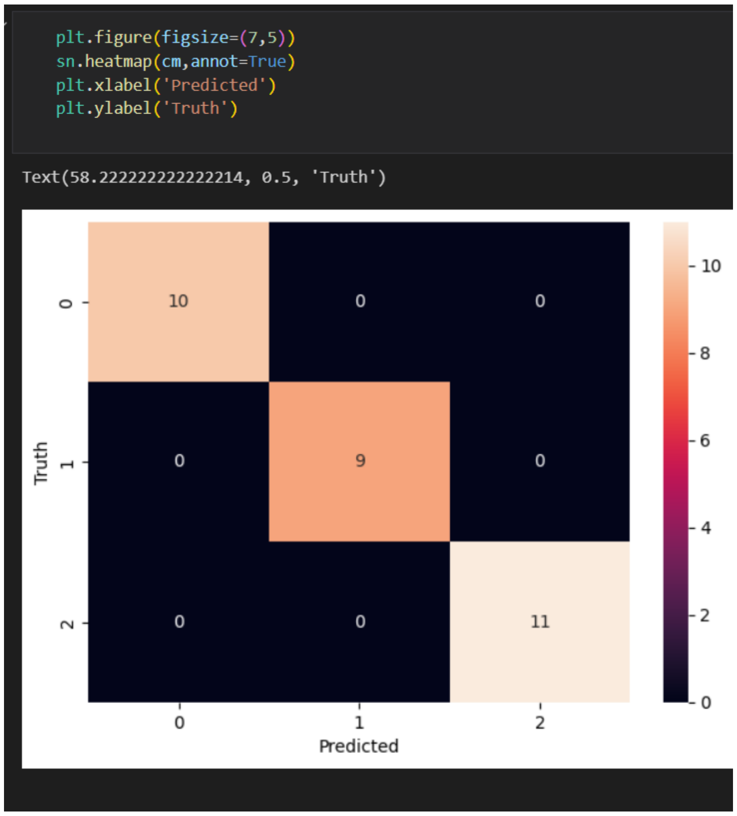




KNN Manhattan Scikit

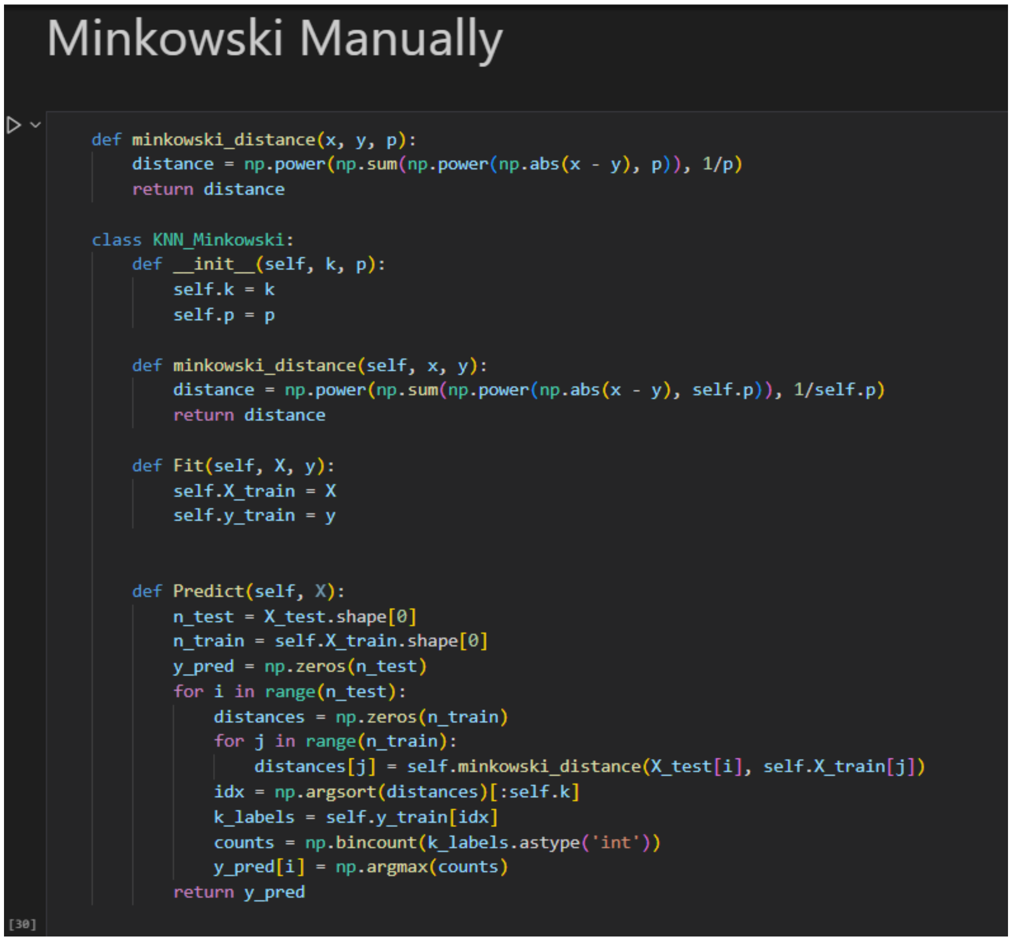
```
knn_manhattan = KNeighborsClassifier(n_neighbors=7, metric='manhattan')#n_neighbors = 5

knn_manhattan.fit(X_train, y_train)

print(knn_manhattan.predict(X_test))
print(knn_manhattan.score(X_test, y_test))
```

```
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0]
1.0
```



```
plt.figure(figsize=(7,5))
sn.heatmap(cm,annot=True)
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

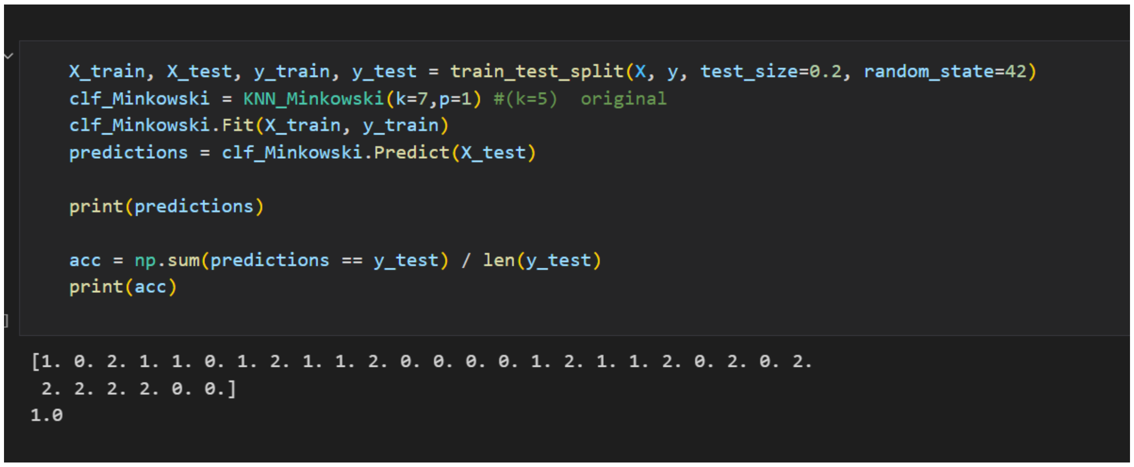
```
Text(58.222222222222214, 0.5, 'Truth')
```



Minkowski Manually

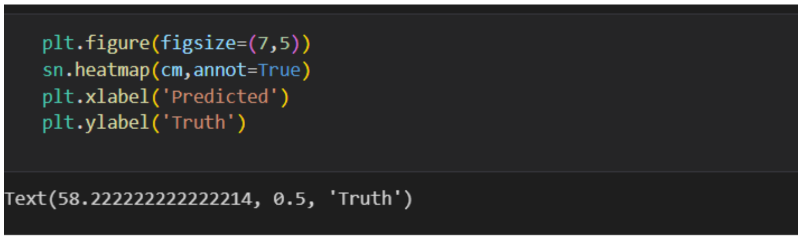
```
def minkowski_distance(x, y, p):
    distance = np.power(np.sum(np.power(np.abs(x - y), p)), 1/p)
    return distance

class KNN_Minkowski:
    def __init__(self, k, p):
        self.k = k
        self.p = p

    def minkowski_distance(self, x, y):
        distance = np.power(np.sum(np.power(np.abs(x - y), self.p)), 1/self.p)
        return distance

    def Fit(self, X, y):
        self.X_train = X
        self.y_train = y


    def Predict(self, X):
        n_test = X_test.shape[0]
        n_train = self.X_train.shape[0]
        y_pred = np.zeros(n_test)
        for i in range(n_test):
            distances = np.zeros(n_train)
            for j in range(n_train):
                distances[j] = self.minkowski_distance(X_test[i], self.X_train[j])
            idx = np.argsort(distances)[:self.k]
            k_labels = self.y_train[idx]
            counts = np.bincount(k_labels.astype('int'))
            y_pred[i] = np.argmax(counts)
        return y_pred
```

[30]



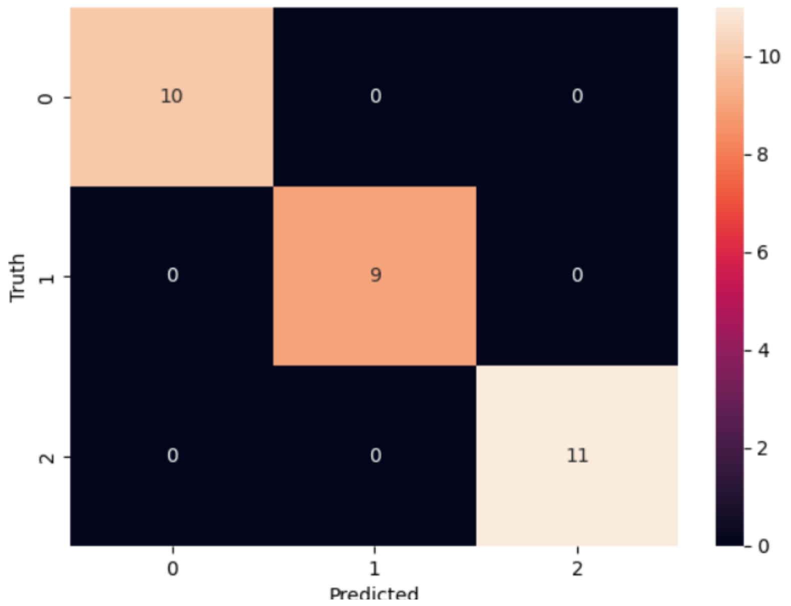
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
clf_Minkowski = KNN_Minkowski(k=7,p=1) #(k=5)  original
clf_Minkowski.Fit(X_train, y_train)
predictions = clf_Minkowski.Predict(X_test)

print(predictions)

acc = np.sum(predictions == y_test) / len(y_test)
print(acc)
```

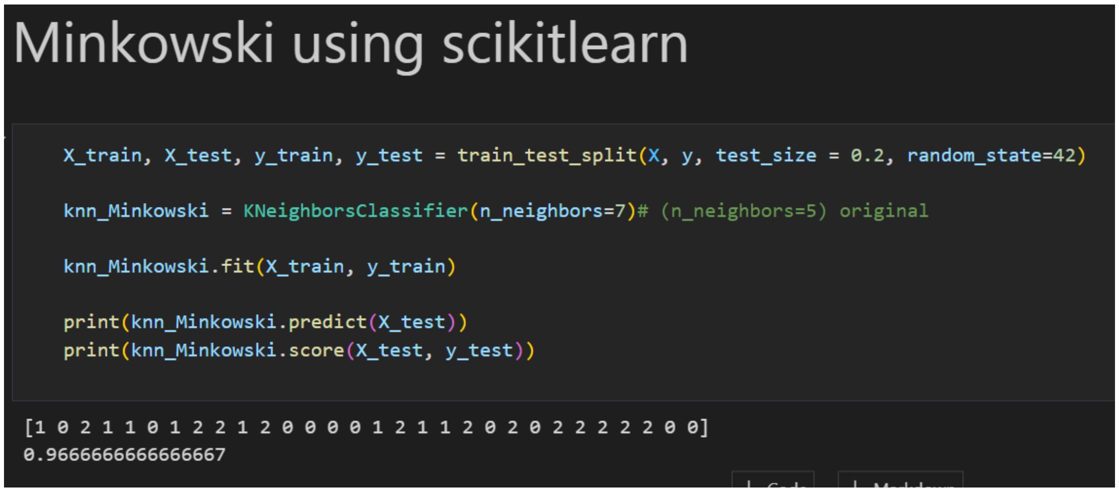
```
[1. 0. 2. 1. 1. 0. 1. 2. 1. 1. 2. 0. 0. 0. 0. 1. 2. 1. 1. 2. 0. 2. 0. 2.
 2. 2. 2. 2. 0. 0.]
1.0
```



```
plt.figure(figsize=(7,5))
sn.heatmap(cm,annot=True)
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

```
Text(58.222222222222214, 0.5, 'Truth')
```



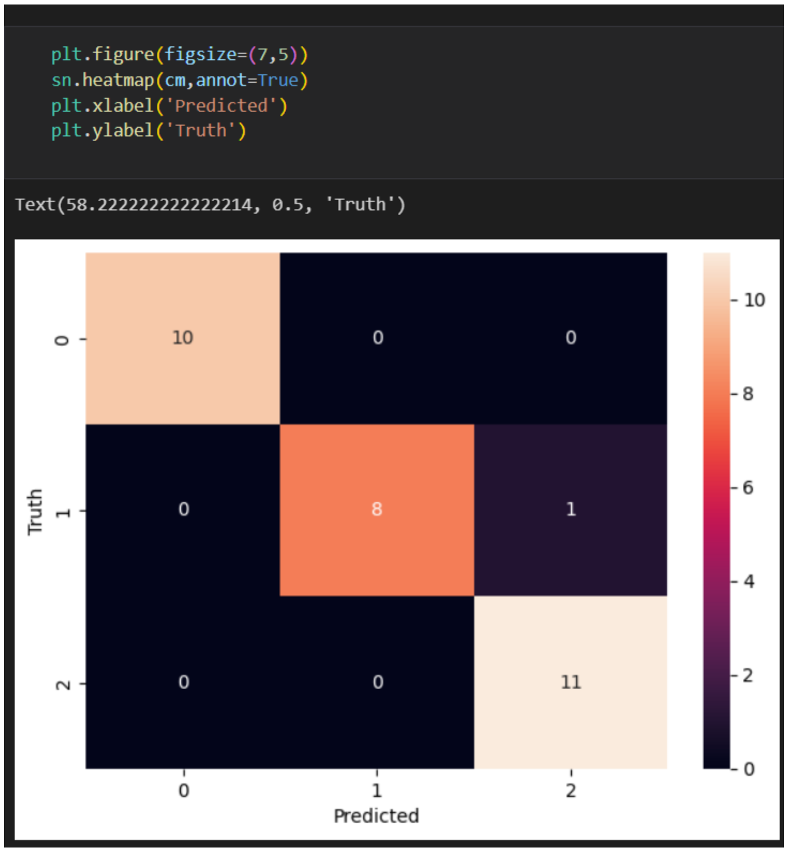


Minkowski using scikitlearn

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42)

knn_Minkowski = KNeighborsClassifier(n_neighbors=7)# (n_neighbors=5) original

knn_Minkowski.fit(X_train, y_train)

print(knn_Minkowski.predict(X_test))
print(knn_Minkowski.score(X_test, y_test))
```

```
[1 0 2 1 1 0 1 2 2 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0]
0.9666666666666667
```

The bug is that the Minkowski distance using the SciKit Learn function has different accuracy than the manual Minkowski distance



```
plt.figure(figsize=(7,5))
sn.heatmap(cm,annot=True)
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

```
Text(58.222222222222214, 0.5, 'Truth')
```

```
X = data[['SepalLengthCm','SepalWidthCm','PetalLengthCm','PetalWidthCm']].values
y = data['Species'].values
```

X is the independent values or the features of the model.

y is the dependent values or the labels/output of the model

```
data['Species'] = data['Species'].apply(lambda x: 0 if x == 'Iris-setosa' else 1 if x == 'Iris-versicolor'
else 2)
data['Species'].unique()
```

this code snippet is used to convert 'Iris-setosa' to 0, 'Iris-versicolor' to 1 and else or 'Iris-virginica' to 2

MANUAL CODES

```
clf = KNN(k=7)#(k=5) original
clf.Fit(X_train, y_train)
predictions = clf.Predict(X_test)
print(predictions)
```

The code snippet above is used by the manual codes for Euclidean, Manhattan and Minkowski methods of KNN Model to train the datasets, and then get the predictions that classifies if Species are Iris-Setosa, Iris-Versicolor or Iris Virginica

```
acc = np.sum(predictions == y_test) / len(y_test)
print(acc)
```

This code snippet is used to get the accuracy of the model using the manual codes for Euclidean, Manhattan and Minkowski methods of KNN Model.

SCIKIT CODES:

```
knn_ = KNeighborsClassifier(n_neighbors=7, metric='')
```

n_neighbors is the parameter Number of neighbors to use for **kneighbors** queries.

metric is the parameter to use for distance computation. Default is "minkowski" but it can also be changed to "manhattan" or "euclidean" to get the results from those methods

```
knn_.fit(X_train, y_train)
```

This code snippet is used to train the dataset for the model.

```
print(knn_.predict(X_test))
print(knn_.score(X_test, y_test))
```

.predict(X_test) is the method in Python is used to make predictions using a trained machine learning model.

.score(X_test, y_test) method returns a score that indicates how well the model was able to predict the values in the test data.

```
y_pred = clf_Manhattan.Predict(X_test)
cm = confusion_matrix(y_test,y_pred)
cm
```

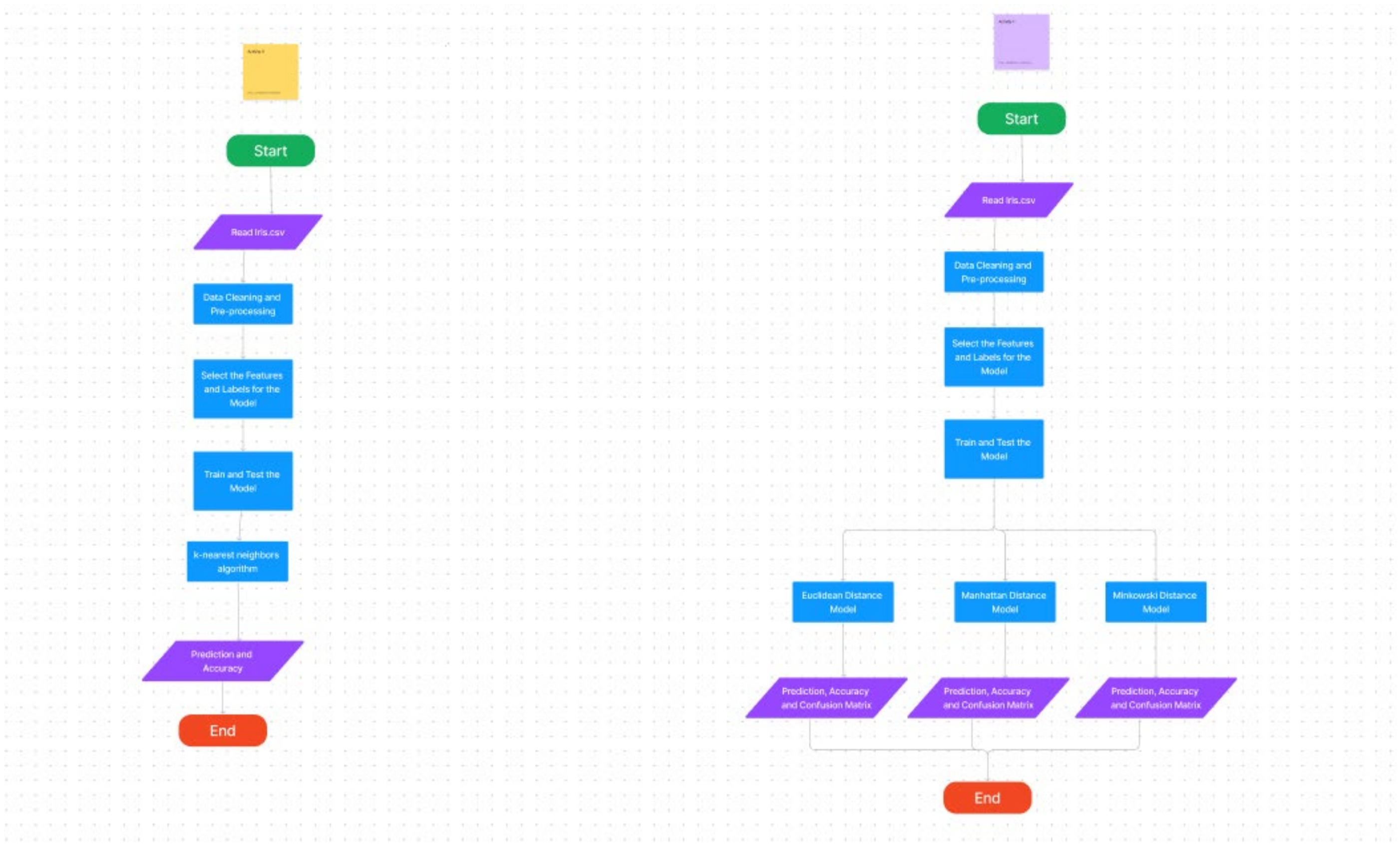
and

```
plt.figure(figsize=(7,5))
sn.heatmap(cm,annot=True)
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

Both of the code snippets are used to show the confusion matrix. The second code snippet shows an actual graph while the first only prints a matrix without a graph. The confusion matrix can be used to evaluate the performance of a machine learning model by providing information about the number of correct and incorrect predictions made by the model.

The output in the diagonal line of the confusion matrix represents the number of correct predictions made by the model. The off-diagonal elements of the confusion matrix represent the number of incorrect predictions made by the model. The diagonal line of the confusion matrix can be used to calculate the overall accuracy of the model, while the off-diagonal elements can be used to calculate the false positive and false negative rates.

Flowchart:


Start
Read Iris.csv
Data Cleaning and Pre-processing
Select the Features and Labels for the Model
Train and Test the Model
k-nearest neighbors algorithm
Prediction and Accuracy
End
Start
Read Iris.csv
Data Cleaning and Pre-processing
Select the Features and Labels for the Model
Train and Test the Model
Euclidean Distance Model
Manhattan Distance Model
Minkowski Distance Model
Prediction, Accuracy and Confusion Matrix
Prediction, Accuracy and Confusion Matrix
Prediction, Accuracy and Confusion Matrix
End

Activity0 and Activity1 KNN flowchart – Figma

RESULTS:
As seen from the prediction, accuracy and confusion matrix each model shows that the both the manual and Sci-kit codes models are performing well. With the exception of the Minkowski Distance manual and Sci-kit codes producing different accuracies and the Sci-kit code producing one incorrect prediction in its confusion matrix, all the models are performing well.