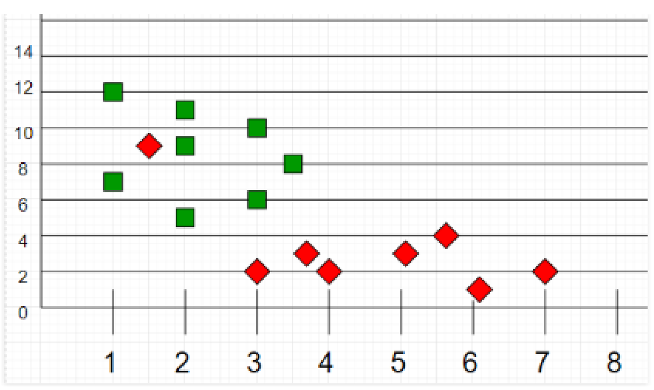
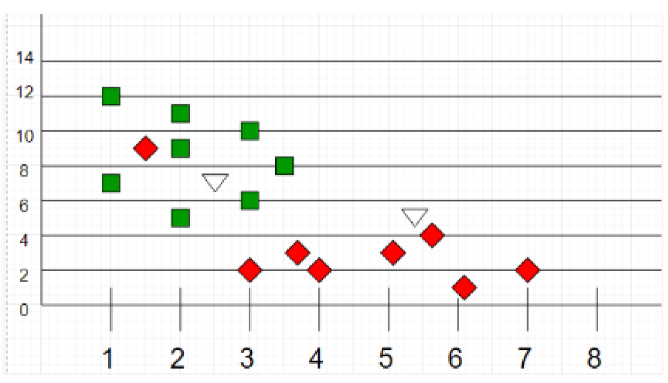
K-Nearest Neighbours

KNN is one of the essential supervised classification algorithms. KNN is a non-parametric algorithm which means we have no idea about the distribution of our data.

Consider the following scatter plot for a dataset:



Consider the last points as the training set. Now it is time to apply the test set to our model.

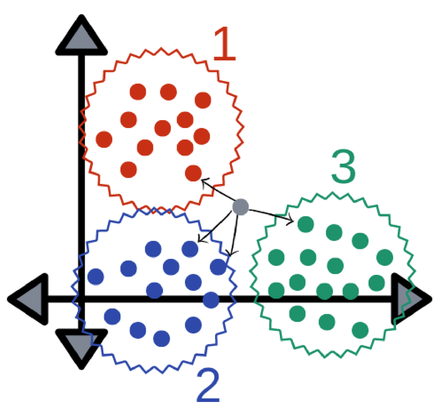


Our KNN model should classify the points into their suitable classes. As we can see on the visualization, separating our data from their class is so easy for human brains. The best method to do is finding the closest line between the data point and the selected neighbor point.

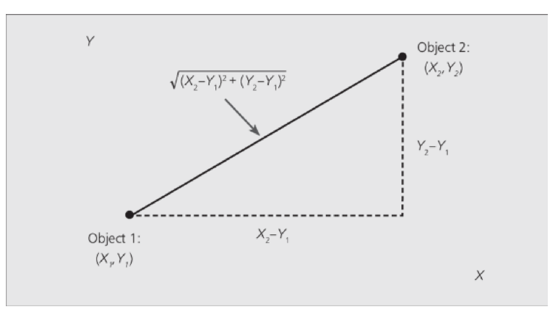
Steps

- Store the training set to an array.
- Select an odd number for the number of selected points around the test point (K).
- Calculate the distance for the test pint and each training points.

The whole process can be summarized with the image down below:



Euclidean Distance



This is how we calculate the distance between 2 points. In mathematics, the Euclidean distance between two points in Euclidean space is the length of a line segment between the two points.

All the steps with extra details are included in the code files with comments.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix

%matplotlib inline
```

Return the dataset as 2 set of array.

In [2]:
```
features, labels = load_digits(return_X_y = True)
```

In [3]:
```
print(features.shape)
print(labels.shape)
```

```
(1797, 64)
(1797,)
```

In [4]:
```
class KnnModel:
    def __init__(self, k, x_train, y_train):
        self.k = k
        self.x_train = x_train
        self.y_train = y_train
        self.m, self.n = x_train.shape

    def predict(self, x_test):
        self.x_test = x_test
        self.m_test, self.n = x_test.shape
        y_predict = np.zeros(self.m_test)

        for i in range(self.m_test):
            x = self.x_test[i]

            neighbors = np.zeros(self.k)
            neighbors = self.find_neighbors(x)
            y_predict[i] = np.min(neighbors)

        return y_predict

    def find_neighbors(self, x):
        eucl_distance = np.zeros(self.m)

        for i in range(self.m):
            d = self.calculate_distance(x, self.x_train[i])
            eucl_distance[i] = d

        index = eucl_distance.argsort()
        y_train_sorted = self.y_train[index]

        return y_train_sorted[:self.k]

    def calculate_distance(self, x, x_train):
        distance = np.sqrt(np.sum(np.square(x - x_train)))

        return distance
```

Validation set in KNN

In KNN algorithm, there is no training step because there is no model to build. Neither is there a validation step. Validation measures model accuracy against the training data as a function of iteration count (training progress). Therefore, there is no need to split the data to 3 set of data.

In [5]:
```
x_train, x_test, y_train, y_test = train_test_split(features, labels, random_state = 45, test_size = 0.2)
```

In [6]:
```
model = KnnModel(3, x_train, y_train)
```

In [7]:
```
predicted_values = model.predict(x_test)
```

In [8]:
```
accuracy_score(y_test, predicted_values)
```

Out[8]:
```
0.9805555555555555
```

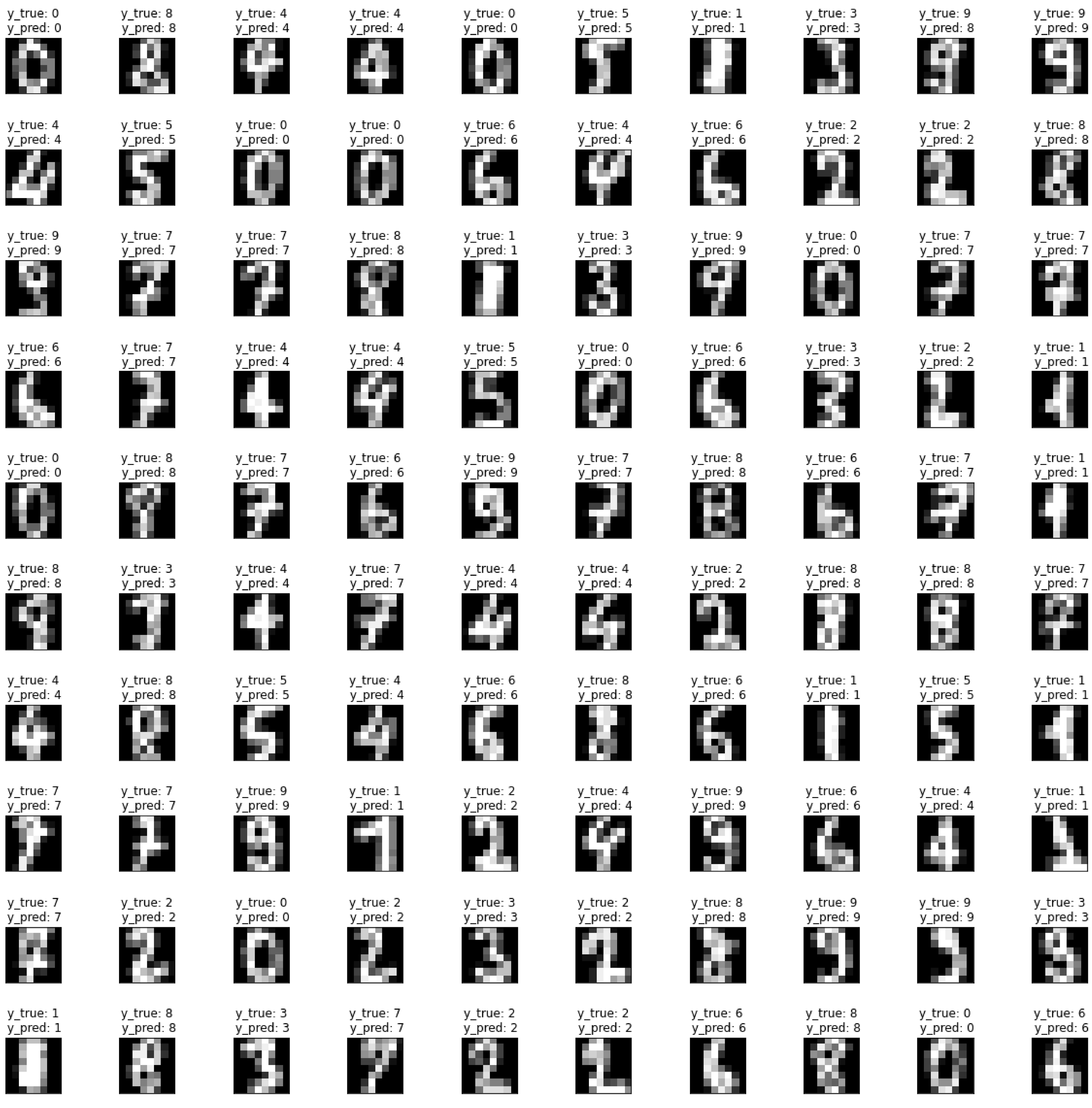
In [9]:
```
confusion_matrix(y_test, predicted_values)
```

Out[9]:
```
array([[27,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0, 33,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0, 35,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0, 41,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0, 45,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0, 29,  0,  0,  0,  1],
       [ 1,  0,  0,  0,  0,  0, 40,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0, 39,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0, 32,  0],
       [ 0,  0,  0,  1,  1,  0,  0,  1,  2, 32]], dtype=int64)
```

In [10]:
```
fig = plt.figure(figsize = (20, 20))
w = 10
h = 10

for i in range(w * h):
    ax = fig.add_subplot(w, h, i + 1)
    plt.gray()
    ax.matshow(x_test[i].reshape(8, 8))
    plt.xticks([])
    plt.yticks([])
    plt.subplots_adjust(hspace = 1, wspace = 1)
    plt.title(f'y_true: {y_test[i]}\n y_pred: {int(predicted_values[i])}')

plt.show()
```



In [ ]: