Checklist:

functionality f1: 20% implemented successfully

functionality f2: 20% implemented successfully

functionality f3: 40% implemented successfully

functionality f4: 10% implemented successfully

functionality f5: 10% implemented successfully

Chosen dataset: Optical recognition of handwritten digits dataset

Chosen learning model/algorithm: k nearest neighbour

Dataset information:

Classes	10
Samples per class	~180
Samples total	1797
Dimensionality	64
Features	integers Min:0 Max:16

- How to run my programme
  1. For the source code: Open the python file called 'sourcecode.py' in IDE and run it. It will cost about 10 seconds so please wait patiently. For implementing functionality f5: for convenience, another method called 'for_test()' is written for testers. It will ask you to input 2 Integer numbers after showing the results of 2 training model.

     • The first input is used for choosing learning model, only '0', '1' and '2' are allowed.
     '0' stands for quit.
     '1' stands for choosing the KNN learning model implementing by myself.
     '2' stands for choosing the KNN learning model imported from library.
     Once you enter the number '1' or '2', it will automatically load the saved model.

     • The second input is the index of test dataset. Since there are 1797 samples in the handwritten digit dataset, the allowed input is from '0' to '1796'.

     Once these two numbers are entered, the program will show the predict and target label of this test sample. Additionally, it will also display an 8*8 image of this handwritten digit for testers.

- Software dependencies

```
from sklearn import datasets
from collections import Counter
from sklearn.externals import joblib
from datetime import datetime
import numpy as np
```

  1. importing 'datasets' is used for loading the handwritten digits dataset

2. importing ‘Counter’ is used to find out the most frequent element in an array
3. importing ‘joblib’ for saving the train model
4. importing ‘datetime’ for analysing time consuming of the program
5. importing ‘numpy’ for array-processing
6. importing ‘matpltlib’ is used to convert the matrix to a grey picture of the handwritten digits

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier as KNN
```

7. importing method for splitting train and test data
8. calling the KNN learning algorithm from scikit-learn for functionality f2. Notice: It is not called when implementing my own KNN model.

- How the functionalities and additional requirements are implemented
  In this programme, k nearest neighbour algorithm is implemented for classifying handwritten digits dataset. Here are the steps:
  1. Pre-process the original data:
     The loaded handwritten digits dataset has 1797 samples in total. 80% of the samples are chosen randomly from the dataset as the training set and the other 20% samples are used as test cases.
  2. Measure distance between test data and training data:
     Squared Euclidean distance is chosen as the distance between two handwritten digit samples. Since every sample has 64 features, the expression of distance between two samples can be written as $d(x,y) = \sum_{i=1}^{64}(x_i - y_i)^2$, x and y stand for features in two different handwritten digit samples.
     Note: Euclidean distance is not chosen as the benchmark because it has to do the square root operation and it is not necessary and wasting time.
  3. Get labels of the nearest neighbours
     Assuming that we have got the distances between one test sample and all the training data, then the distances will be sorted in order to get the k minimum distances. After that, corresponding labels of these k minimum can be obtained for further process.
  4. Predict the label of a new test sample
     The last step is to acquire the most frequent element in the k labels. The most frequent one will be chosen as the prediction of the new test sample.

- Details of implementation
  1. initialization of ‘myKNN’ class

```
class myKNN():

    def __init__(self, k=5, train_X=[], train_y=[]):
        self.k=k
        self.train_X=train_X
        self.train_y=train_y
```

To initialize ‘myKNN’ class, 3 parameters are needed: ‘k’, ‘train_X’ and ‘train_y’. ‘k’ stands for how many nearest neighbours will vote for the final prediction. ‘train_X’ and ‘train_y’ stand for training dataset and corresponding labels of these samples.

2. Method 1: ‘distance()’

```
def distance(self, sample, train_X):
    sample=sample.reshape(1,-1)
    sample_mat=np.tile(sample, (train_X.shape[0],1))
    diff=sample_mat-train_X
    distances=np.power(diff,2).sum(axis=1)
    #distances=np.sqrt(np.power(diff,2).sum(axis=1))

    return distances
```

Method 'distance()' needs 2 parameters. 'sample' stands for one test sample. Firstly, 'sample' will be reshaped into one row. In this case, nomally, the shape of 'sample' after reshaping will be 1*64. Then, append this one-row 'sample' to itself until its shape fits the shape of 'train_X' and this new matrix is saved in 'sample_mat'. 'diff' records the difference between 'sample_mat' and 'train_X', using every feature value in 'sample_mat' minus the corresponding one in 'train_X'. Finally, get the sum row by row, recorded it in 'distances' and return it.

3. Method 2: 'get_klabels()'

```
def get_klabels(self, k, train_y,distances):
    labels=[]
    k_min_distances=np.sort(distances)[:k]

    for i in range(0,distances.shape[0]):
        for j in k_min_distances:
            if(distances[i] == j):
                label=train_y[i]
                labels.append(label)

    return labels
```

Method 'get_klabels()' needs 3 parameters. Firstly, initialize an empty list called 'labels'. Then sort the 'distances' list to get k minimum distances. Find these distances in the original list and get their indexes. These indexes are also the indexes of labels in 'train_y'. After obtaining k labels, append them in 'labels' and return it. These labels will act as voters to determine the final prediction.

4. Method 3: 'predict()'

```
def predict(self, test_X):
    res=[]

    for sample in test_X:
        distances=self.distance(sample,self.train_X)
        labels=self.get_klabels(self.k,self.train_y,distances)
        most_frequent_label=Counter(labels).most_common(1)[0][0]
        res.append(most_frequent_label)

    return res
```

Method 'predict()' needs only 1 parameter which is 'test_X'. Firstly, initialize an empty list called 'res' which will hold the final result. For each sample in 'test_X', call the method 'distance' to get its distances away from training samples. Then call the method 'labels' to get corresponding labels of those k nearest neighbours. Choose the most frequent one in 'labels' and regard it as the final prediction of this test sample. Finally, append all the prediction together in 'res' and return it.

- Evaluation/Comparation between my KNN model and Sklearn KNN model

The figure below shows result after running the program.

```
========train model analysis of "myKNN"=======
accuracy: 0.990953%
There are 13 errors in this set
========test model analysis of "myKNN"========
accuracy: 0.986111%
There are 5 errors in this set
Time elapsed (hh:mm:ss.ms) 0:00:08.335147


========train model analysis of "sklernKNN"=======
accuracy: 0.991649%
There are 12 errors in this set
========test model analysis of "sklernKNN"========
accuracy: 0.986111%
There are 5 errors in this set
Time elapsed (hh:mm:ss.ms) 0:00:00.232387
```

Using 80% as train datasets and 20% as test datasets (random seed=33)

Model	Training analysis		Test analysis		Time cost (sec)
My KNN model	Train errors	13	Test errors	5	8.335147
	Train accuracy	0.990953%	Test accuracy	0.986111%	
Sklearn KNN model	Train errors	12	Test errors	5	0.232387
	Train accuracy	0.991649%	Test accuracy	0.986111%	

Conclusion: According to the table above, although the accuracy of both models is similar, My KNN model costs much more time than Sklearn KNN model.

- How to choose the best k

  Tables below record my test results (accuracy is based on test samples)

Value of k	My KNN model accuracy	Sklearn KNN model accuracy
1	98.333333%	98.333333%
2	98.055556%	98.333333%
3	98.333333%	98.055556%
4	98.333333%	98.333333%
5	98.611111%	98.611111%
6	97.777778%	98.055556%
7	97.222222%	97.500000%
8	97.500000%	97.777778%
9	97.777778%	97.777778%

We can find that accuracy reach peak when k=5, thus, in this program 5 is chosen as the value of k.