Hi Kyriakos and Yi-Hung,

Good report! No need for resubmission

Quick introduction to jupyter notebooks

- Each cell in this notebook contains either code or text.
- You can run a cell by pressing Ctrl-Enter, or run and advance to the next cell with Shift-Enter.
- Code cells will print their output, including images, below the cell. Running it again deletes the previous output, so be careful if you want to save some results.
- You don't have to rerun all cells to test changes, just rerun the cell you have made changes to. Some exceptions might apply, for example if you overwrite variables from previous cells, but in general this will work.
- If all else fails, use the "Kernel" menu and select "Restart Kernel and Clear All Output". You can also use this menu to run all cells.
- A useful debug tool is the console. You can right-click anywhere in the notebook and select "New console for notebook". This opens a python console which shares the environment with the notebook, which let's you easily print variables or test commands.

Setup

In [1]:
```
# Automatically reload modules when changed
%reload_ext autoreload
%autoreload 2
# Plot figures "inline" with other output
%matplotlib inline

# Import modules, classes, functions
from datetime import timedelta
from time import perf_counter as tic

from matplotlib import pyplot as plt
import numpy as np

from utils import plotDatasets, loadDataset, splitData, splitDataBins, ge
from evalFunctions import calcConfusionMatrix, calcAccuracy, calcAccuracy

# Configure nice figures
plt.rcParams['figure.facecolor']='white'
plt.rcParams['figure.figsize']=(8,5)
```

! IMPORTANT NOTE !

Your implementation should only use the `numpy` (`np`) module. The `numpy` module provides all the functionality you need for this assignment and makes it easier debuging your code. No other modules, e.g. `scikit-learn` or `scipy` among others, are allowed and solutions using modules other than `numpy` will be sent for re-submission. You can find everything you need about `numpy` in the official documentation.

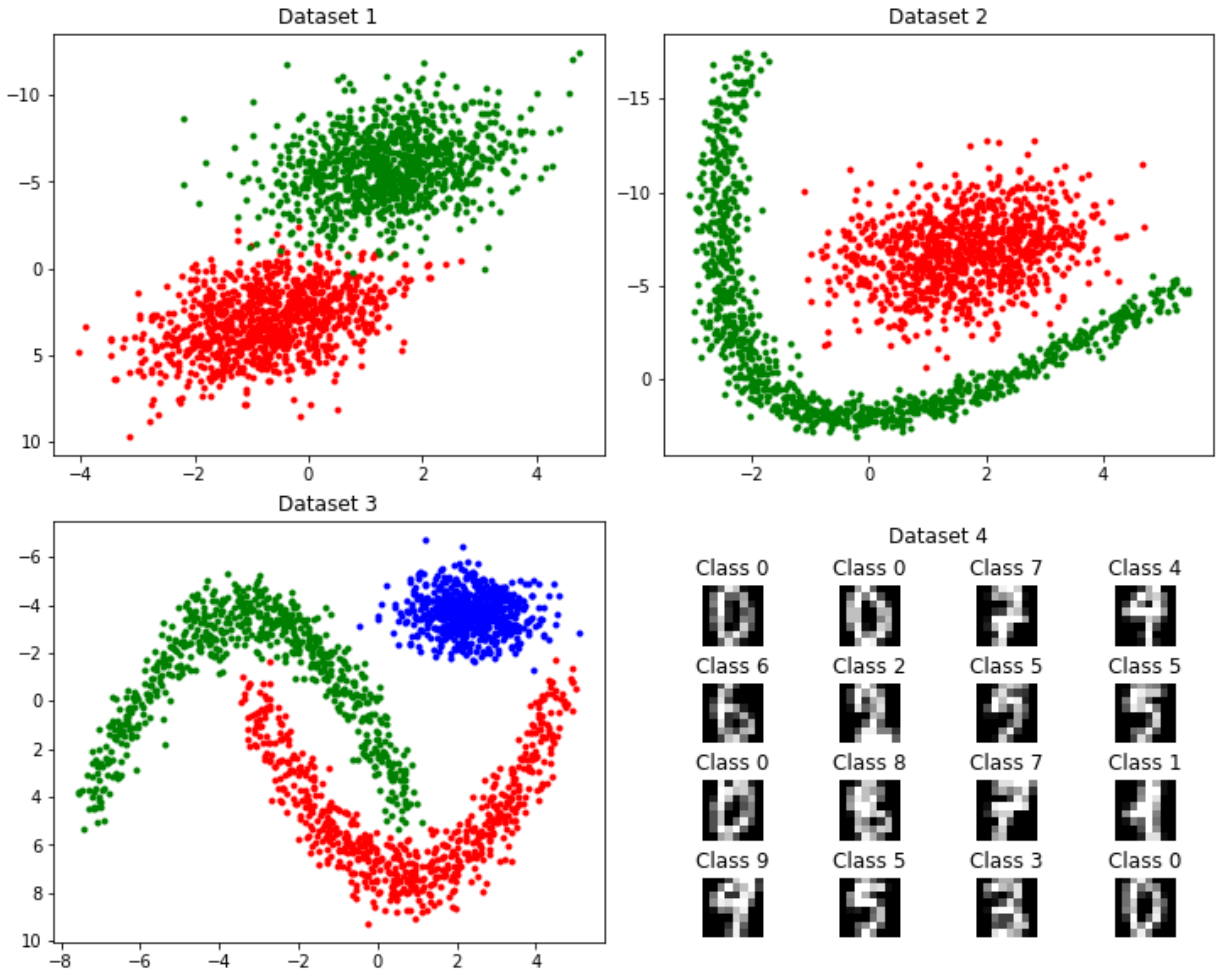
1. Introduction

The focus of this assignment is **supervised learning**. In particular, you will apply several machine learning algorithms to solve classification tasks. Throughout the three notebooks that consistute this assignment you will implement a kNN classifier, as well as single-layer and two-layer neural networks.

1.1 Data

Let's start by examining the datasets used in this assignments. Run the following cell to visualize the datasets.

In [2]:
```
plotDatasets()
```



As you can see, datasets 1, 2 and 3 are point clouds with various shapes and number of classes, while dataset 4 consists of 8x8 pixel images of handwritten digits (these are stored as 64-length vectors). Each dataset in this assignment consists of three variables:

- `X` contains the input features for the data samples.
- `D` contains neural network target output values for the data samples. These are not used with kNN, and will be explained in the other notebooks in this assignment.
- `L` contains the class labels for the data samples.

Use the code in the next cell to load and examine all four datasets. Note that this assignment follows the convention that data samples are in the rows of a matrix, while features are in the columns.

In [2]:
```
datasetNr = 4
X, D, L = loadDataset(datasetNr)

print(f"X has shape {X.shape}")
print(f"D has shape {D.shape}")
print(f"L has shape {L.shape}")
print(L)
```

```
X has shape (5620, 64)
D has shape (5620, 10)
L has shape (5620,)
[0 0 7 ... 8 9 8]
```

Question 1:

Describe all four datasets used in this assignment from a machine learning perspective:

- What does the dataset represent? What kind of data is it made of, and what can you tell about its arrangement?
- How many samples are in each dataset? How many features do they have?
- How many classes does each dataset have? What do they represent?
- Will the dataset require a linear or nonlinear classifier? Why?

Answer:

[1st dataset: This dataset represents the coordinates for 2 different classes, thus we are have a binary classification problem. The coordinates are represented by 2 points, so we can say that these 2 classes have 2 features. Both features are numerical. This dataset contains of 2000 saples and each samples has 2 features. So we have 2 *2000 = 4000 values in this dataset. Looking the plot for the first dataset we can say that we can use a linear classifier as the data look linearly seperetable. 2nd dataset: This dataset represents the coordinates for 2 different classes, thus we are have a binary classification problem. The coordinates are represented by 2 points, so we can say that these 2 classes have 2 features. Both features are numerical. This dataset contains of 2000 saples and each samples has 2 features. So we have 2* 2000 = 4000 values in this dataset. Looking the plot for the 2nd dataset we can say that we have to use a non- linear classifier as the data doesn't look linearly seperetable as the previous dataset. 3rd dataset: This dataset represents the coordinates for 3 different classes, thus we are have a classification problem with 3 different classes. The coordinates are represented by 2 points, so we can say that this 3 classes have 2 features. Both features are numerical. This dataset contains of 2000 saples and each samples has 2 features. So we have 2 * 2000 = 4000 values in this dataset. Looking the plot for the 3rd dataset we can say that we have to use a non- linear classifier as the classes dont look linearly seperable. 4th dataset: This dataset images of handwritten numbers. Each image is formed by 64 pixels(8x8). So each observation have 64 feature, each pixule is a feature. This dataset contains 5620 samples and we have 10 classes in total, the numbers from 0 to 9. Each class represent a handwritten digit from 0-9. the 4th dataset is not linear but because we are in 64d space and we have only 10 classes our data will be sparsely contributed so we can use a linear classifier.]

2. The kNN classifier

k-nearest neighbors (kNN) is a relatively simple classification algorithm, that nevertheless can be quite effective. It is a nonlinear classifier where each new sample is assigned the class that most commonly appears among its neighbors in the training data, i.e. those training samples with the shortest distance to it. Distances in kNN can actually be defined in many different ways based on the application, but here we will use the most common Euclidean distance. The number of neighboring samples to consider, called k, is the only parameter of the algorithm. Depending on the specific properties of the problem, different values of k might give optimal results.

Unlike other types of classifiers, such as support vector machines and neural networks, kNN does not have any trainable parameters, and thus requires no training. It does, however, require a training dataset, which is effectively "memorized", and used as reference to classify all future data. This has the advantage of no training time, but results in slow inference times, which are proportional to the amount of training data.

2.1 Implement the kNN algorithm

The `kNN` function takes as input arguments the set of samples to be classified `X`, the number of neighbors to consider `k`, an the training samples `XTrain` and labels `LTrain`. There are different ways to implement the kNN algorithm, but we recommend you to follow these steps:

1. Calculate the Euclidean distances between every point in `X` and every point in `XTrain` and save them in a large matrix. Recall that the Eucliden distance between two N-dimensional points $\mathbf{x}$ and $\mathbf{y}$ is given as

$$d = \sqrt{\sum_{i=1}^{N}(x_i - y_i)^2}.$$

Your implementation should not assume any specific number of features in the data, but should work for data of any number of features.

1. From each row of the matrix, select the `k` points with the smallest distance.
2. Find the class that appears most often among the `k` closest points and assign it to the corresponding point in `X`.
3. Sometimes there is a draw between two neighboring classes. Detect this and implement a strategy for choosing the class.

Keep in mind that, as was said previously, classifying data with kNN can be time-consuming, and an efficient implementation can really save you some time in the long run (especially once we implement cross-validation in section 3). Because of this, it is recommended that you try to avoid loops as much as possible, and instead take full advantage of `numpy`'s capacity for operating directly on arrays and broadcasting arrays. Some loops will likely be necessary, but you will see performance gains if you try to minimize their use.

In [2]:
```
import math

def most_common(lst):
    return max(set(lst), key=lst.count)

def kNN(X, k, XTrain, LTrain):
    """ KNN
    Your implementation of the kNN algorithm.

    Args:
        X (array): Samples to be classified.
        k (int): Number of neighbors.
        XTrain (array): Training samples.
        LTrain (array): Correct labels of each sample.

    Returns:
        LPred (array): Predicted labels for each sample.
    """

    classes = np.unique(LTrain)
    nClasses = classes.shape[0]
    LTrain = LTrain.tolist()

    # -------------------------------------------
    # === Your code here ========================
    # -------------------------------------------

    # Calculate all the distances between X and XTrain
    distances = np.zeros(shape=(X.shape[0], XTrain.shape[0]))
    for index, x in enumerate(X):
        for index2, x2 in enumerate(XTrain):
            distances[index, index2] = math.sqrt(sum((x - x2) ** 2))

    # Sort distances and find k closest labels
    closest_labels = np.zeros(shape=(X.shape[0], k))
    for index, distance in enumerate(distances):
        # The indeces of the closest observations
        idx = sorted(range(len(distance)), key = lambda sub: distance[sub
        closest_labels[index] = [int(i) for i in idx]

    # Find the most common label, store in LPred
    LPred = list()
    for closest_label in closest_labels:
        # bad naming here
        labels = list()
        for index in closest_label:
            labels.append(LTrain[int(index)])
        label = most_common(labels)
        LPred.append(label)

    # ===========================================

    return np.array(LPred)
```

2.2 Test it on some data

In order to test your implementation, you will first need to split the available data into training and test sets. You can then classify the test data using the training data as reference. Use the `splitData` function for this purpose.

In [3]:
```
# Select and load dataset
datasetNr = 1
X, D, L = loadDataset(datasetNr)

# Split data into training set (85%) and test set (15%)
XTrain, _, LTrain, XTest, _, LTest = splitData(X, D, L, 0.15)
print(XTrain.shape)
print(XTest[1])
print(type(XTrain))
```

```
(1700, 2)
[-1.26618282  4.65727523]
<class 'numpy.ndarray'>
```

Set a value for `k` and classify the training and test data.

In [4]:
```
# Set the number of neighbors
k = 45

# Classify training data
LPredTrain = kNN(XTrain, k, XTrain, LTrain)
# Classify test data
LPredTest  = kNN(XTest , k, XTrain, LTrain)
```

Calculate and print the training and test accuracies as well as the confusion matrix for the test data. For this to work, you first need to open the file `evalFunctions.py` and implement the functions `calcAccuracy`, `calcConfusionMatrix`, and `calcAccuracyCM`, based on the function descriptions.

In [5]:
```
# Calculate the training and test accuracy
accTrain = calcAccuracy(LPredTrain, LTrain)
accTest = calcAccuracy(LPredTest, LTest)
print(f"Train accuracy: {accTrain:.4f}")
print(f"Test accuracy: {accTest:.4f}")

# Calculate confunsion matrix of test data
confMatrix = calcConfusionMatrix(LPredTest, LTest)
print("Test data confusion matrix:")
print(confMatrix)

accTestCM = calcAccuracyCM(confMatrix)
print(accTestCM)
#print(f"Test accuracy from CM: {accTestCM:.4f}")
```

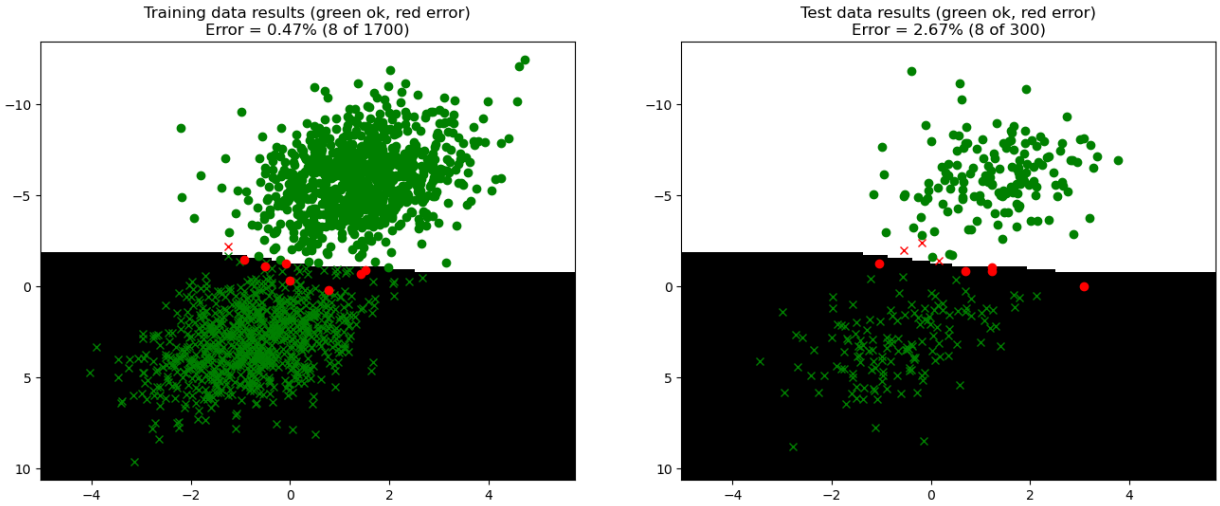
```
Train accuracy: 0.9935
Test accuracy: 0.9867
Test data confusion matrix:
[[140   3]
 [  1 156]]
0.9866666666666667
```

Now we can use some plotting functions to examine the classified training and test data, as well as the decision boundaries that separate the various classes. We will use these types of visualizations for all three classifier types.

For datasets 1-3 you will see classification results for the training and test data, where correctly classified samples appear in green and incorrectly classified samples appear in red. The backgrounds of these plots show in grayscale colors the different regions of the feature space which are assigned each class by the classifier. This is especially useful in order to examine the shape of the decision boundaries.

For the dataset 4 you will see a plot that shows examples of each type of correct and incorrect classification as given by the confusion matrix.

In [20]:
```
if datasetNr < 4:
    plotResultsDots(XTrain, LTrain, LPredTrain, XTest, LTest, LPredTest,
else:
    plotConfusionMatrixOCR(XTest, LTest, LPredTest)
```



Question 2:

- Describe how your kNN implementation works, step by step.
- Describe the way in which your implementation handles ties in the neighbor classes, that is, situations in which several classes are equally common among the neighbors of a point. For example, `k=4` and the classes of the neighbors are `[0,0,1,1]`, or `k=5` and the classes are `[1,1,2,3,3]`.

Answer:

[First we find the distance, using the euclidean function, from each new sample that we are given from every point from the training data. We save these distances in a matrix called distances(nxm) which n is the number of the samples that we want to predict and m is the number of traning points that we have. then in a matrix called closest_labels(nxk) we save the indeces of the smallest k values in the matrix distances for a spesific sample(row). Using the saved indices for each sample we find the labels of the k closest points to our sample and then we find the most common label that appeared using these indeces. When several classes are equally common among the neighbors of a point we select the first one that reaches the number of the most neighbours according to the indexes. for example, in the above example for k=4 would be 0 and for k=5 would be 1.]

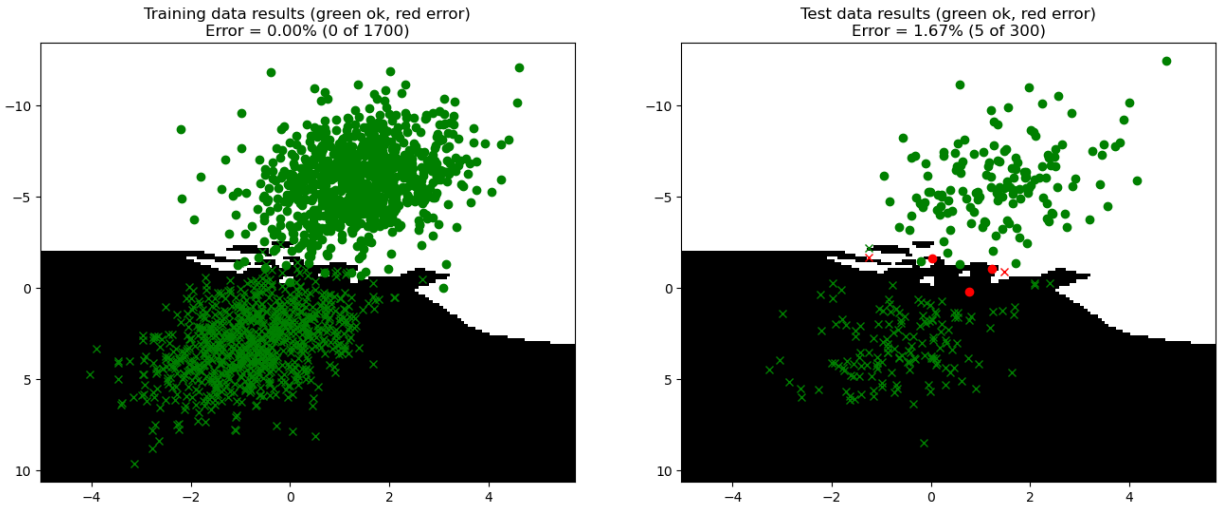
2.3 Try kNN on all datasets

Once you have made sure that your kNN implementation works correctly, we can define a function that performs all of the previous steps: it loads data, trains and evaluates a kNN on a specific dataset using your own kNN implementation, and prints the results. You can use it to experiment with applying your kNN implementation on all the datasets. Try experimenting with different values of `k` and note especially the effect that it has on the decision boundaries.

In [5]:
```
def runkNNOnDataset(datasetNr, testSplit, k):
    X, D, L = loadDataset(datasetNr)
    XTrain, _, LTrain, XTest, _, LTest = splitData(X, D, L, testSplit)

    LPredTrain = kNN(XTrain, k, XTrain, LTrain)
    LPredTest = kNN(XTest, k, XTrain, LTrain)

    accTrain = calcAccuracy(LPredTrain, LTrain)
    accTest = calcAccuracy(LPredTest, LTest)
    confMatrix = calcConfusionMatrix(LPredTest, LTest)

    print(f'Train accuracy: {accTrain:.4f}')
    print(f'Test accuracy: {accTest:.4f}')
    print("Test data confusion matrix:")
    print(confMatrix)

    if datasetNr < 4:
        plotResultsDots(XTrain, LTrain, LPredTrain, XTest, LTest, LPredTe
    else:
        plotConfusionMatrixOCR(XTest, LTest, LPredTest)
```

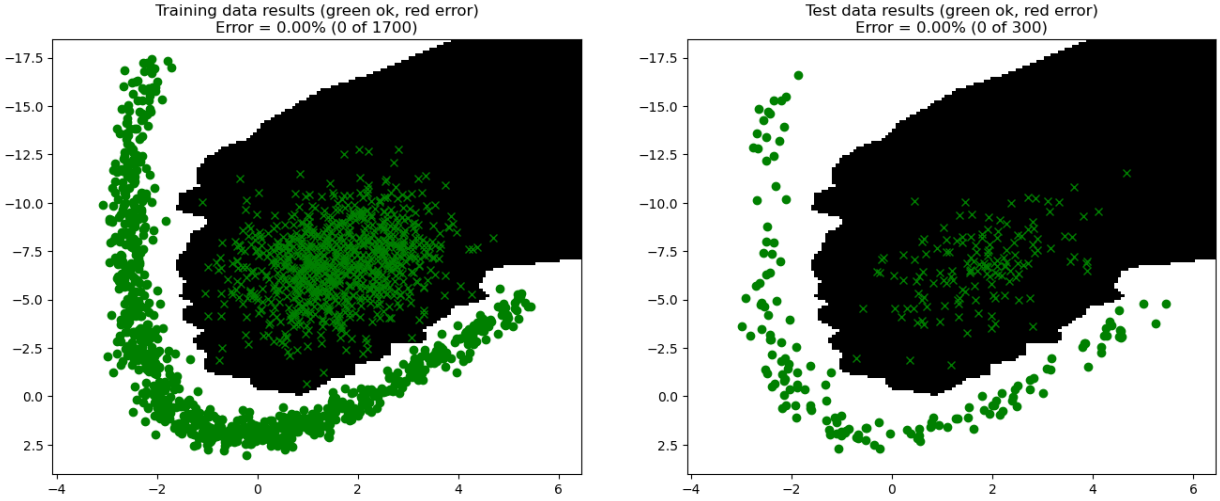
In [29]:
```
runkNNOnDataset(1, testSplit=0.15, k=1)
```

```
Train accuracy: 1.0000
Test accuracy: 0.9833
Test data confusion matrix:
[[148   3]
 [  2 147]]
```



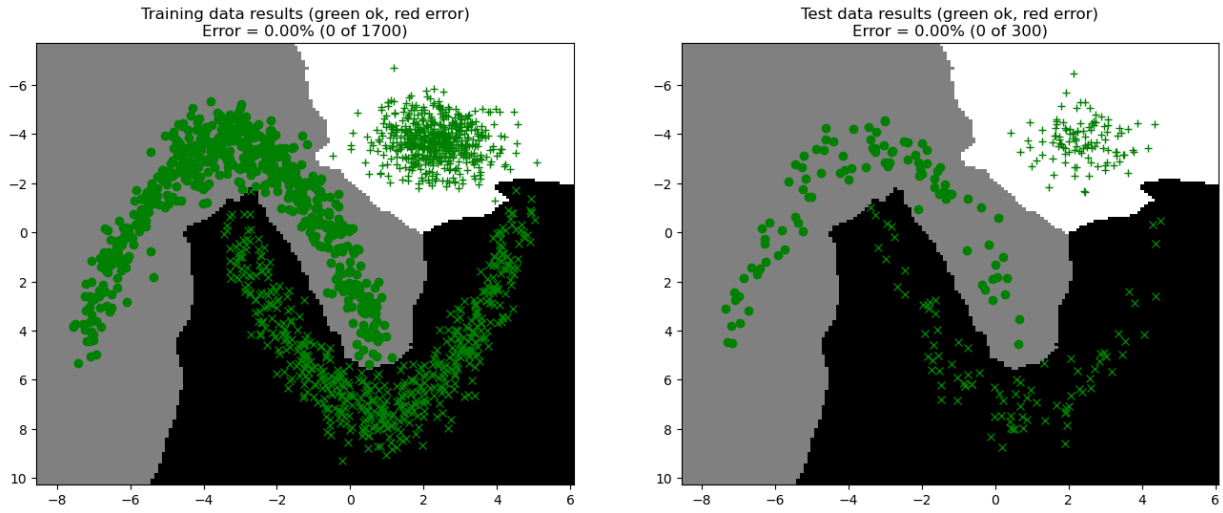
In [30]:
```
runkNNOnDataset(2, testSplit=0.15, k=1)
```

```
Train accuracy: 1.0000
Test accuracy: 1.0000
Test data confusion matrix:
[[146   0]
 [  0 154]]
```



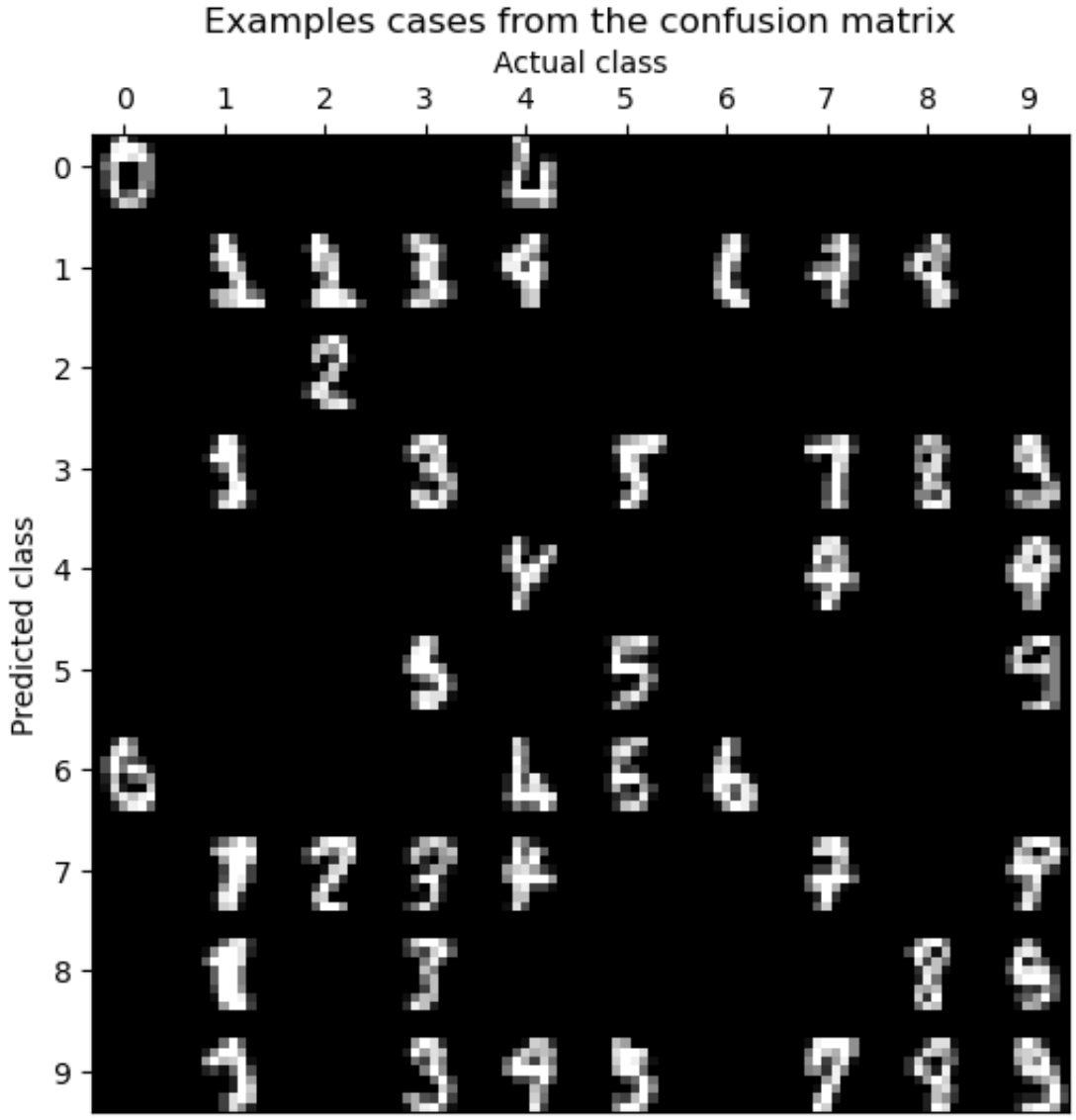
In [31]:
```
runkNNOnDataset(3, testSplit=0.15, k=1)
```

```
Train accuracy: 1.0000
Test accuracy: 1.0000
Test data confusion matrix:
[[ 91   0   0]
 [  0  99   0]
 [  0   0 110]]
```



In [32]:
```
runkNNOnDataset(4, testSplit=0.75, k=1)
```

```
Train accuracy: 1.0000
Test accuracy: 0.9784
Test data confusion matrix:
[[425   0   0   0   1   0   0   0   0   0]
 [  0 426   3   1   1   0   1   2  15   0]
 [  0   0 402   0   0   0   0   0   0   0]
 [  0   2   0 424   0   4   0   1   3   9]
 [  0   0   0   0 424   0   0   2   0   2]
 [  0   0   0   1   0 397   0   0   0   5]
 [  1   0   0   0   2   1 409   0   0   0]
 [  0   1   1   1   1   0   0 411   0   1]
 [  0   1   0   3   0   0   0   0 393   5]
 [  0   1   0   3   2   8   0   5   1 413]]
```


Examples cases from the confusion matrix
Actual class
0
1
2
3
4
5
6
7
8
9
Predicted class
0
1
2
3
4
5
6
7
8
9

3. Cross-validation

As mentioned previously, different values of `k` might work better or worse for each dataset. However, in order to establish which value is best for each dataset it is not enough to run kNN once for each `k` and select the one that gives the highest accuracy. This approach would not take into account the variations in performance that result from the random splitting of training and test data. Results obtained in this way will not reflect the performance that can be expected when the algorithm is applied on new data.

In order to thoroughly test which value of `k` is best we can resort to cross-validation methods, which rely on repeatedly testing the model on different splits of the data in order to assess its generalization performance. In particular, we will focus on n-fold cross-validation. In this method, we will first reserve a portion of the data for testing, `XTest`, which we will not touch until the very end, and use the remaining data `XTrain` for cross-validation. `XTrain` will again be split into `N` bins, which corresponds with the number of times that the kNN algorithm will be run for each value of `k`. For each iteration, one bin is used as validation data `XValCV`, and all the remaining bins are combined and used as training data `XTrainCV`. This will result in `N` accuracies for each value of `k`, which we will average to obtain the **average cross-validation accuracy**, which is the relevant metric for determining the optimal `k`. The higher the value of `N`, the more precise will be our determination of the accuracy of different values of `k`. This picture illustrates 3-fold cross validation for one value of `k`.

After determining the value of `k` that gives the highest accuracy, we will use it to classify `XTest` using all of `XTrain` as reference data. This will give as the **test accuracy** of our model, and is the definitive metric representing its performance.

Start by splitting the available data into training and test `splitData` function as before. Then, use the function `splitDataBins` to further split the training data into `N` bins. Finally, use the funciton `getCVSplit` to combine the data bins into `XTrainCV` and `XValCV`. This function takes in the degree of cross validation `N` and the current iteration of the cross validation `i`, indicating which bin will be used for the validation data (note that this is zero-indexed). Three-fold cross-validation should be a minimum, but do not be afraid to try using more bins, e.g. 50-100, as the resulting inference time increases less than linearly.

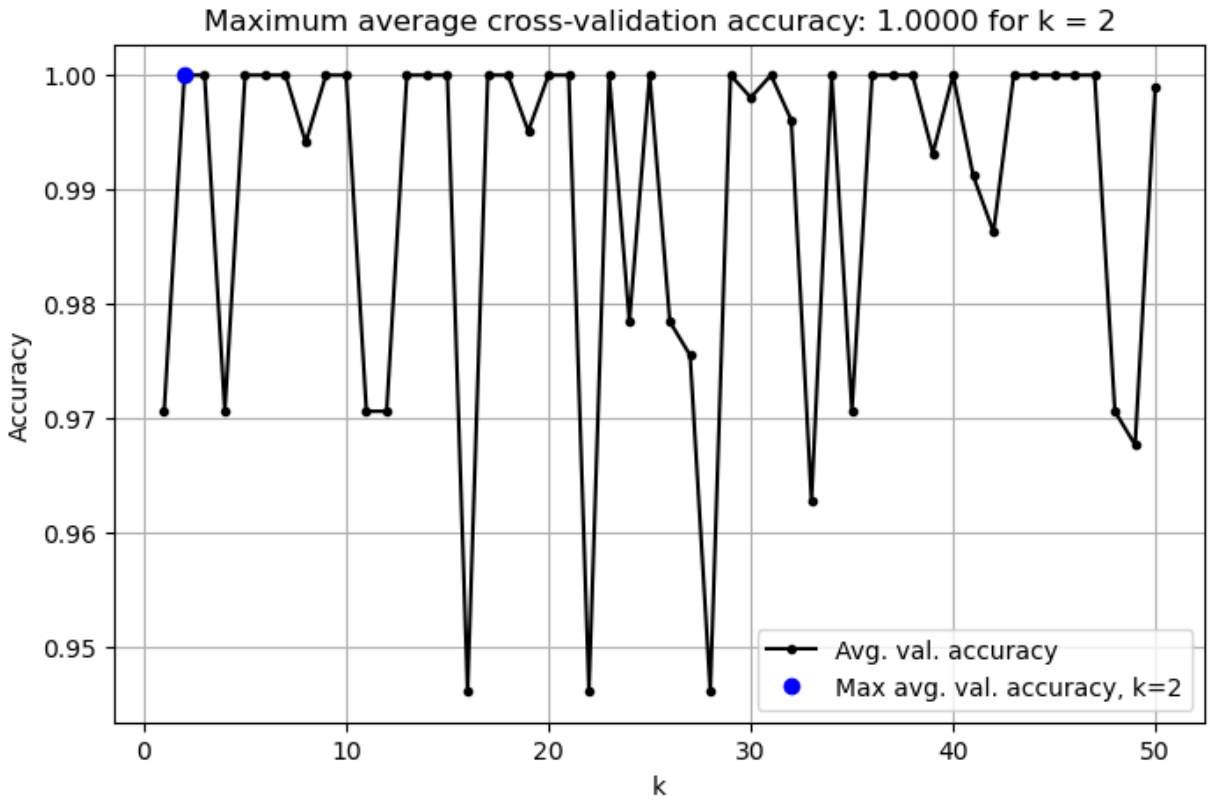
In [5]:
```
# Select and load dataset
datasetNr = 1
X, D, L = loadDataset(datasetNr)

# Split data into training and test sets
XTrain, _, LTrain, XTest, _, LTest = splitData(X, D, L, 0.15)

# Select the number of bins to split the data
nBins = 50

# Split data into bins based on the settings above
# The outputs are lists of length nBins, where each item is a data array.
XBins, DBins, LBins = splitDataBins(XTrain, None, LTrain, nBins)
```

Finish the implementation of the `crossValidation` function, which needs to take a maximum value of `k` and the cross validation bins and return a matrix containing the cross-validation accuracies obtained for different values of `k` and combinations of bins used for training.

In [6]:
```python
def crossValiation(kMax, XBins, LBins):
    """Performs cross-validation using kNN

    Args:
        kMax (int): Maximum value of k to test. Values used will be [1-kM
        XBins (list of arrays): Training+validation data samples.
        LBins (list of arrays): Training+validation data labels.

    Returns:
        meanAccs (array): Cross-validation accuracies. Bins are in the ro
            values of k in the columns.
        kBest (int): Optimal value of k based on cross validation results
    """

    nBins = len(XBins)
    accs = np.zeros((nBins, kMax))

    # This is used to show the progress
    timeStart = tic()

    # ------------------------------------------
    # === Your code here =======================
    # ------------------------------------------

    for i in range(nBins):

        # Use getCVSplit to combine bins for training and validation data
        XTrain, _, LTrain, XVal, _, LVal = getCVSplit(XBins, None, LBins,

        for k in range(kMax):

            # Classify validation data using kNN
            predictions = kNN(XVal, k+1, XTrain, LTrain)

            # ... and store resulting accuracy in the accs matrix
            accs[i, k] = calcAccuracy(predictions, LVal)

            # Print progress and remaining time
            timeLeft = round((tic()-timeStart)*( nBins*kMax / (i*kMax + k
            etaStr = str(timedelta(seconds=timeLeft))
            print(f"b: {i+1:2}, k: {k+1:2}, ETA: {etaStr}    ", end="\r")

    # Compute the mean cross validation accuracy for each k
    meanAccs = np.mean(accs, axis = 0)


    # And find the best k
    kBest = meanAccs.tolist().index(max(meanAccs)) + 1

    # ------------------------------------------

    return meanAccs, kBest
```

Test your cross-validation implementation and look at the resulting performance plot. This shows the average cross-validation accuracy for all the values of `k` tested.

In [7]:
```
meanAccs, kBest = crossValiation(30, XBins, LBins)
plotResultsCV(meanAccs, kBest)
```



After selecting the optimal value of `k` using cross-validation, use this value to classify `XTest` using the data in `XTrain` as reference to obtain the test accuracy.

In [8]:
```
LPredTest = kNN(XTest, kBest, XTrain, LTrain)

confMatrix = calcConfusionMatrix(LPredTest, LTest)
acc = calcAccuracy(LPredTest, LTest)

print(f"Test accuracy: {acc:.4f}")
print("Test data confusion matrix:")
print(confMatrix)
```

```
Test accuracy: 0.9867
Test data confusion matrix:
[[145   3]
 [  1 151]]
```

Question 3:

- Describe how you implemented cross-validation.

Answer:

[in first, we have splitted the training dataset in n bins. the n-1 bins will consist of the new training data set and 1 bin will be the validation bin. We have to go though every bin as validation bin and for the rest of the bins as the training dataset. for each validation bin we save the accuracy of every model that we made for each k (k = 1 until maximum k). Then we find the mean accuracy of validation bin for every k and choose the best k according to that.]

---

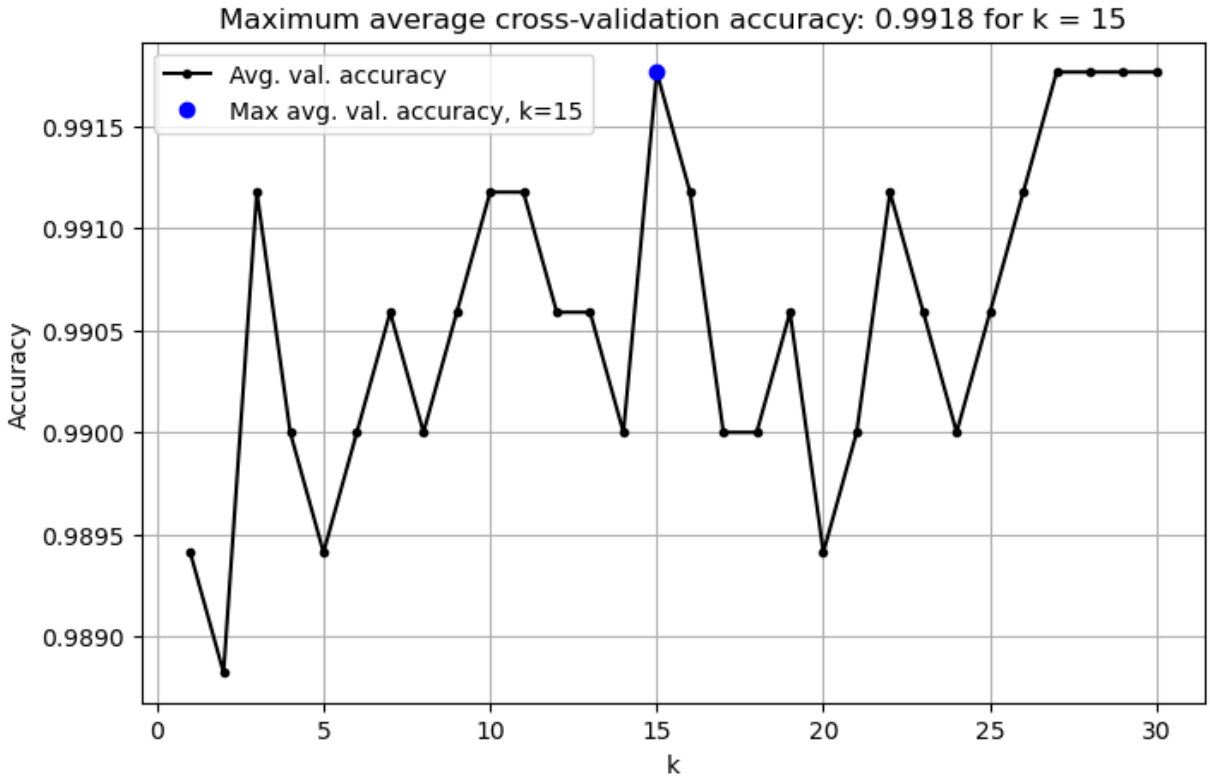
4. Cross validation for all datasets

Once again we define a single function that performs all of the previous cross-validation for a given dataset and shows the results. Use it to perform cross-validation on all four datasets.

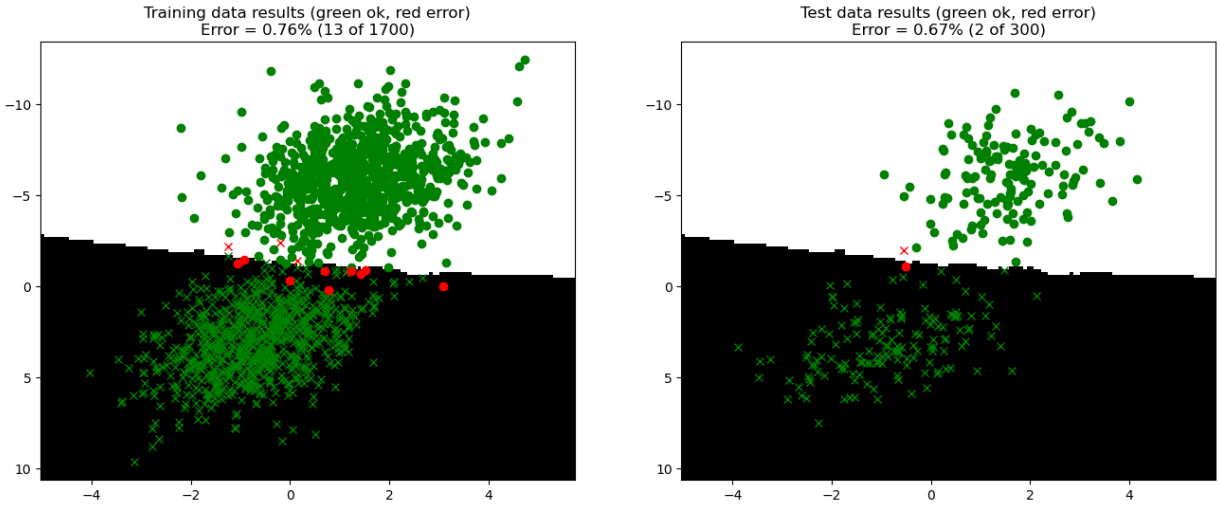
In [7]:
```
def runkNNCrossValidationOnDataset(datasetNr, testSplit, nBins, kMax):
    X, D, L = loadDataset(datasetNr)
    XTrain, _, LTrain, XTest, _, LTest = splitData(X, D, L, testSplit)
    XBins, _, LBins = splitDataBins(XTrain, None, LTrain, nBins)

    meanAccs, kBest = crossValiation(kMax, XBins, LBins)
    plotResultsCV(meanAccs, kBest)

    LPredTrain = kNN(XTrain, kBest, XTrain, LTrain)
    LPredTest = kNN(XTest, kBest, XTrain, LTrain)
    confMatrix = calcConfusionMatrix(LPredTest, LTest)
    accTest = calcAccuracy(LPredTest, LTest)

    print(f'Test accuracy: {accTest:.4f}')
    print("Test data confusion matrix:")
    print(confMatrix)

    if datasetNr < 4:
        plotResultsDots(XTrain, LTrain, LPredTrain, XTest, LTest, LPredTe
    else:
        plotConfusionMatrixOCR(XTest, LTest, LPredTest)
```

In [7]:
```
runkNNCrossValidationOnDataset(1, testSplit=0.15, nBins=20, kMax=30)
```

```
b: 20, k: 30, ETA: 0:00:00
```

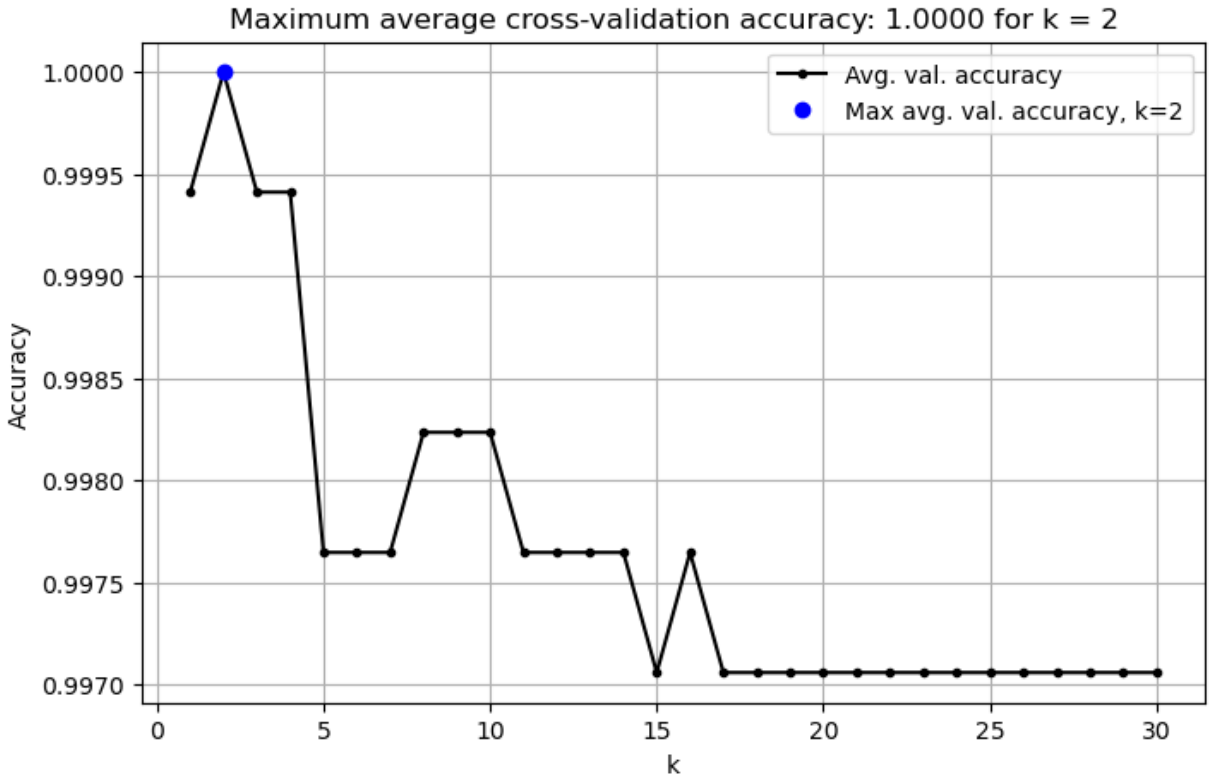



```
Test accuracy: 0.9933
Test data confusion matrix:
[[152   1]
 [  1 146]]
```

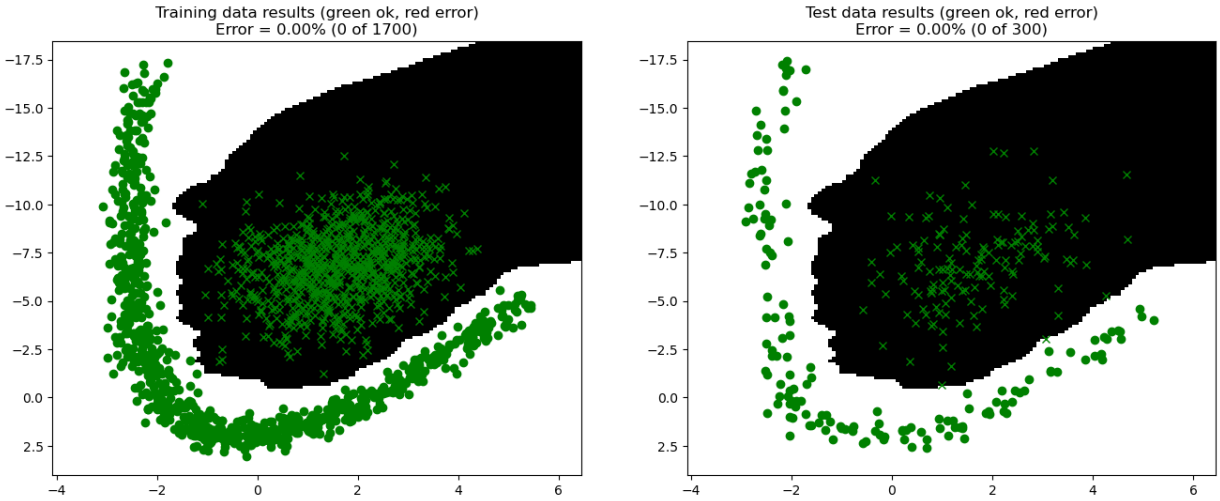


In [8]:
```
runkNNCrossValidationOnDataset(2, testSplit=0.15, nBins=20, kMax=30)
```

```
b: 20, k: 30, ETA: 0:00:00
```

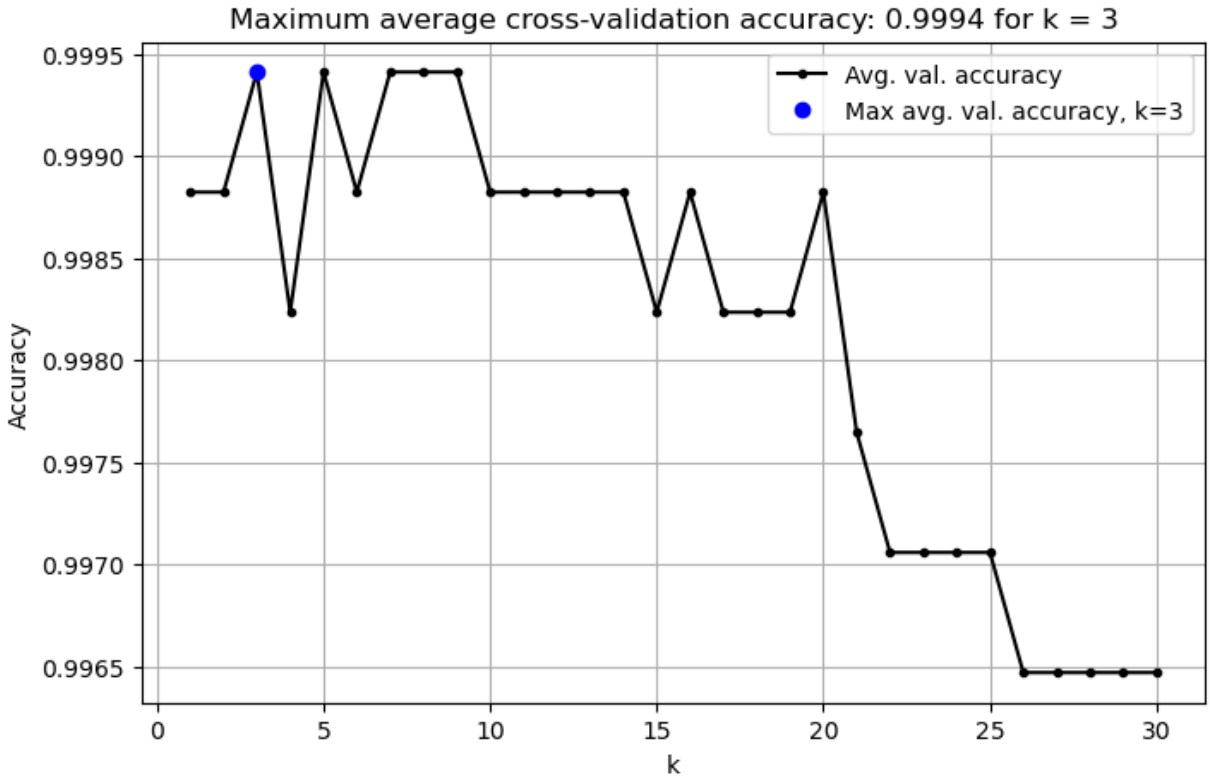



```
Test accuracy: 1.0000
Test data confusion matrix:
[[151   0]
 [  0 149]]
```

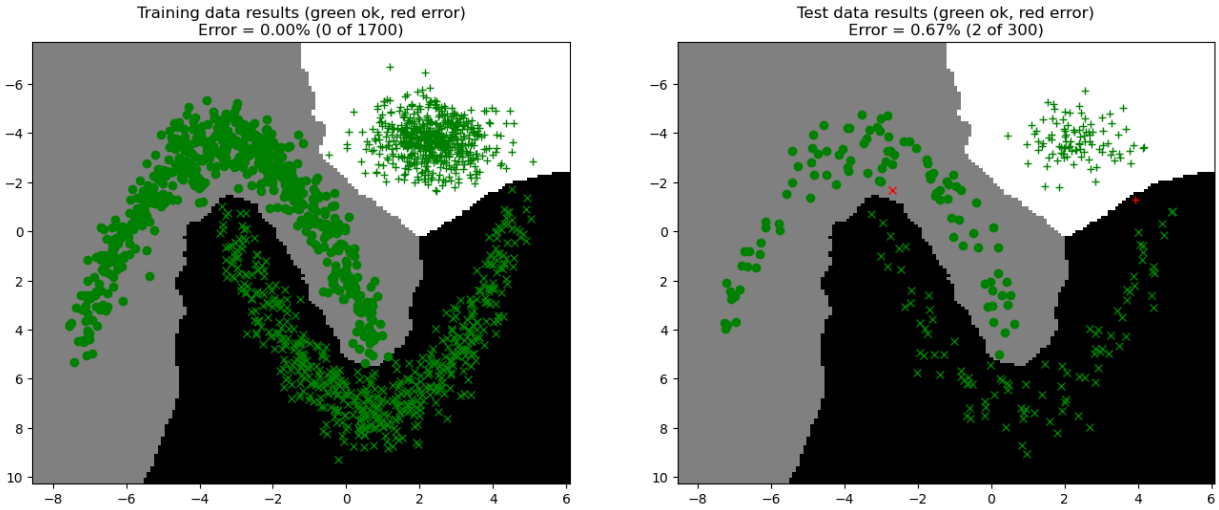


In [5]:
```
runkNNCrossValidationOnDataset(3, testSplit=0.15, nBins=20, kMax=30)
```

```
b: 20, k: 30, ETA: 0:00:00
```

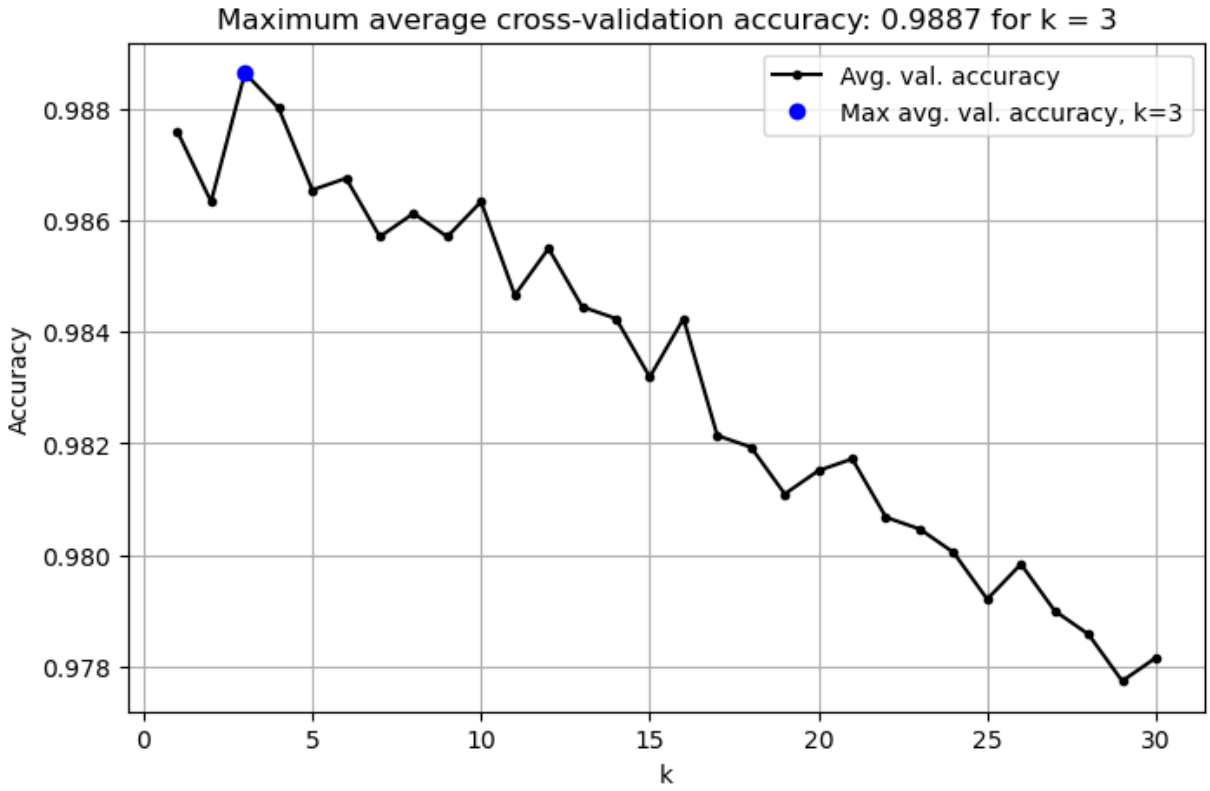



```
Test accuracy: 0.9933
Test data confusion matrix:
[[101   0   1]
 [  1 100   0]
 [  0   0  97]]
```

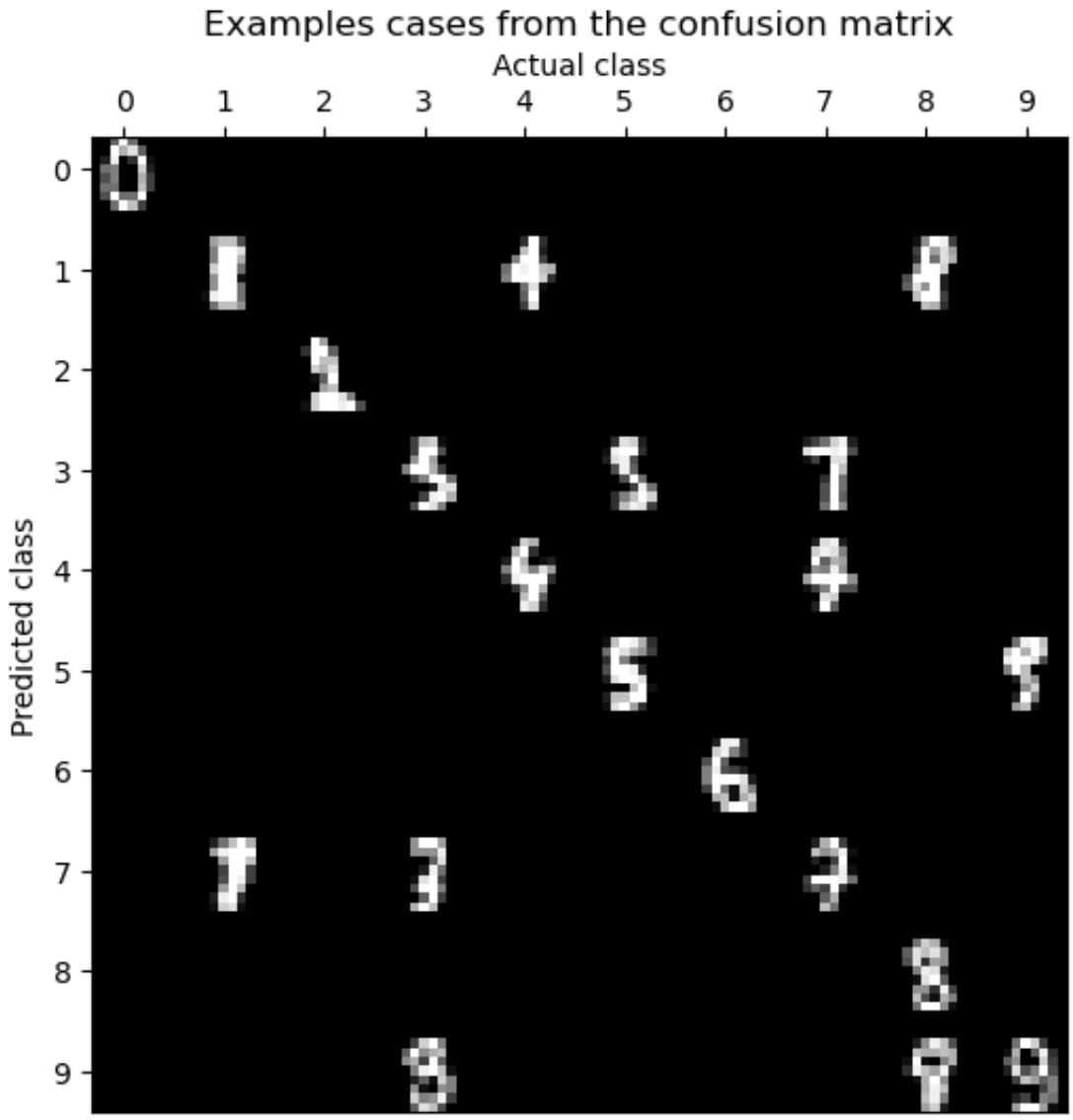


In [9]:
```
runkNNCrossValidationOnDataset(4, testSplit=0.15, nBins=20, kMax=30)
```

```
b: 20, k: 30, ETA: 0:00:00
```




```
Test accuracy: 0.9846
Test data confusion matrix:
[[79  0  0  0  0  0  0  0  0  0]
 [ 0 90  0  0  1  0  0  0  4  0]
 [ 0  0 89  0  0  0  0  0  0  0]
 [ 0  0  0 80  0  1  0  1  0  0]
 [ 0  0  0  0 74  0  0  1  0  0]
 [ 0  0  0  0  0 83  0  0  0  1]
 [ 0  0  0  0  0  0 87  0  0  0]
 [ 0  1  0  1  0  0  0 79  0  0]
 [ 0  0  0  0  0  0  0  0 89  0]
 [ 0  0  0  1  0  0  0  0  1 80]]
```



Question 4:

- Comment on the results for each dataset. What is the optimal k, and are those results reasonable?

Answer:

[1st dataset: The optimal k is 15 and its reasonable as they are only 2 classes (according the visualisation of the data) and they are very close to each other as we can see in the graph. So we need 15 nearest neighbors because in case we have a point between the 2 classes we will need many neighbors in order to ensure which class is the correct one. 2nd dataset: The optimal k is 2 and is reasonable as they are only 2 classes which are much more distinguishable compared to the first dataset. 3rd dataset: the optimal k is 3 for the same reason of the second dataset despite of having 3 classes. 4th dataset: the optimal k is 3 but its difficult to give aa certain answer as we can not represent a 64d space.]

---

5. Optional task

In section 2 in this notebook, where you first implemented the kNN algorithm, we said that "some loops will be necessary" in the implementation. *This is actually not true*. By rewriting the computation of the Euclidean distance in a clever way, and using the full capabilities of numpy broadcasting, it is possible to compute the distance matrix without a single loop. This solution is incredibly fast and therefore enables high degree cross validation over many values of k. Your optional task is to rewrite your implementation to have no loops, and to rerun the cross validation.

Question 5:

- How much faster is the new implementation? You can time the execution of a code cell by putting the magic command (yes, that is the official name) `%%timeit -n1 -r1` on the first row of the cell. Note that the double percentages are part of the command.

Answer:

[Your answers here]