main1_kNN

February 12, 2023

0.0.1 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.

0.0.2 Setup

[12]:
```
# 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,␣
 ↪getCVSplit, plotResultsCV, plotResultsDots, plotConfusionMatrixOCR
from evalFunctions import calcConfusionMatrix, calcAccuracy, calcAccuracyCM

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

```
'export' is not recognized as an internal or external command,
operable program or batch file.
```

0.0.3 *! IMPORTANT NOTE !*

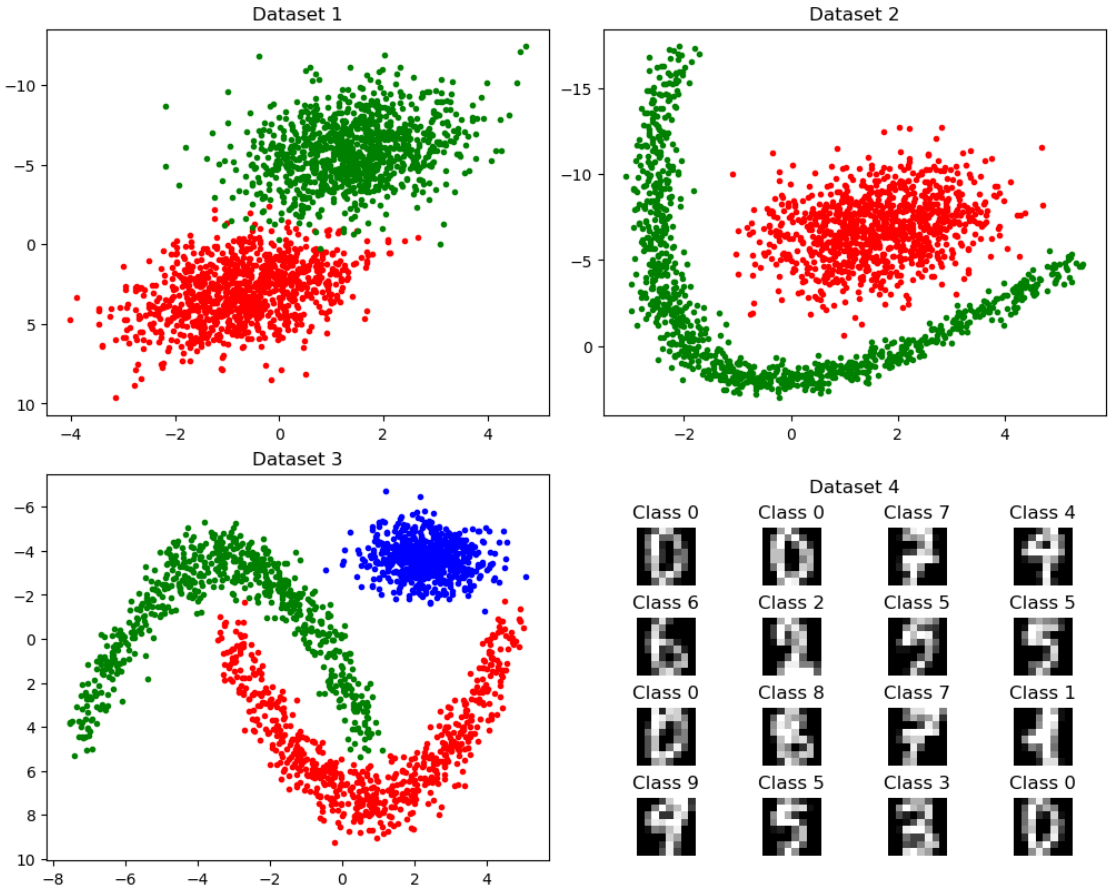
Your implementation should only use the `numpy` (`np`) module. The `numpy` module provides all the functionality you need for this assignment and makes it easier debuging your code. No other modules, e.g. `scikit-learn` or `scipy` among others, are allowed and solutions using modules other than `numpy`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

0.0.4 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.

[3]:
```
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.

[4]:
```
datasetNr = 1
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}")
```

```
X has shape (2000, 2)
D has shape (2000, 2)
L has shape (2000,)
```

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: [Your answers here] Dataset 1: Binary classification with two clusters 2000 samples 2 features: x and y coordinates 2 classes: Unknown Linear classifier; the two clusters can be separated by a linear classifier.

Dataset 2: Binary classification with two separable regions 2000 samples 2 features: x and y coordinates 2 classes: Unknown non-linear classifier; the two regions are not separable by a linear classifier. The green class wraps a bit around the red class when reading coordinates.

Dataset 3: Classification with 3 classes in three separable regions 2000 samples 2 features: x and y coordinates 3 classes: Unknown non-linear classifier; same reason as dataset 2

Dataset 4: Image recognition and classification of digits 5620 samples 64 features: every pixel of the 8x8 image 10 classes: 0-9 digits non-linear classifier; The patterns of digits do not follow a linear pattern. CNN are often used.

0.0.5 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.

2. From each row of the matrix, select the `k` points with the smallest distance.
3. Find the class that appears most often among the `k` closest points and assign it to the corresponding point in `X`.
4. 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.

[5]:
```
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]

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

    # Calculate all the distances between X and XTrain

    dists = np.array([np.linalg.norm(XTrain - p, axis=1) for p in X])

    # Sort distances and find k closest labels

    sortedMatrix = np.zeros((len(X), len(XTrain)), int)
    for i, row in enumerate(dists):
        rowIndices = np.zeros(len(row), int)

        rowIndices = np.array(sorted(range(len(row)), key=lambda x:row[x]))
        sortedMatrix[i] = rowIndices


    labelsMatrix = np.zeros((len(X), k), int)
    for i, point in enumerate(X):
        for j in range(k):
            labelsMatrix[i, j] = LTrain[sortedMatrix[i, j]]

    # Find the most common label, store in LPred
    LPred = np.zeros(len(X), int)

    for i, point in enumerate(X):
        LPred[i] = np.bincount(labelsMatrix[i]).argmax()

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

    return 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.

[6]:
```
# 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)
```

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

[7]:
```
# Set the number of neighbors
k = 10

# 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.

[8]:
```
# 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()
print("Test data confusion matrix:")
print(confMatrix)

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

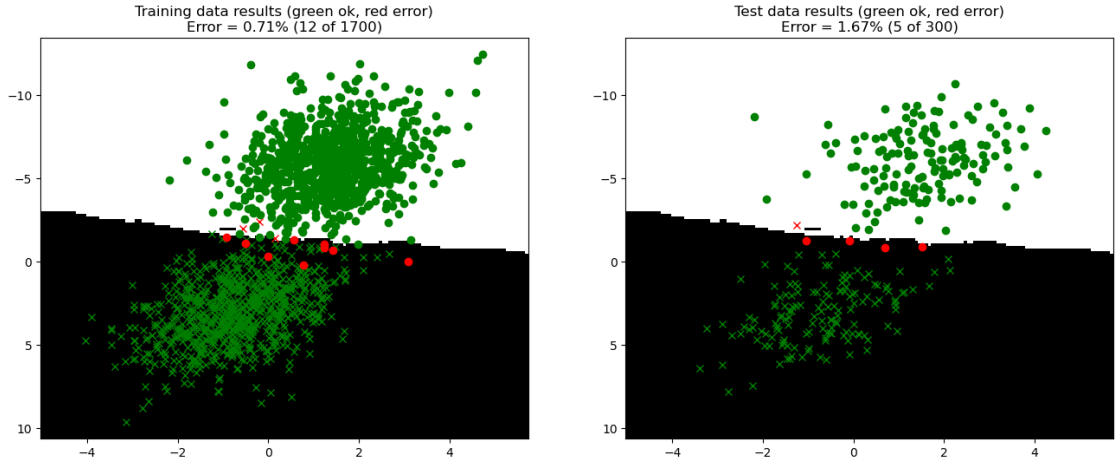
```
Train accuracy: 0.9929
Test accuracy: 0.9833

Test data confusion matrix:
[[146   4]
 [  1 149]]

Test accuracy from CM: 0.9833
```

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.

[9]:
```
if datasetNr < 4:
    plotResultsDots(XTrain, LTrain, LPredTrain, XTest, LTest, LPredTest, lambda␣
 ↪X: kNN(X, k, XTrain, LTrain))
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: [Your answers here] Our kNN implementation in the first step calculates the euclidean distance between the sample point x in X for all points in XTrain.

In the following step the rows of distances that correlate to a point x is used to sort a list of indices that represent the XTrain points for the distances. This list of indices is used to get the correct labels of the k first indices (representing the k lowest distances) for the XTrain points. The labels

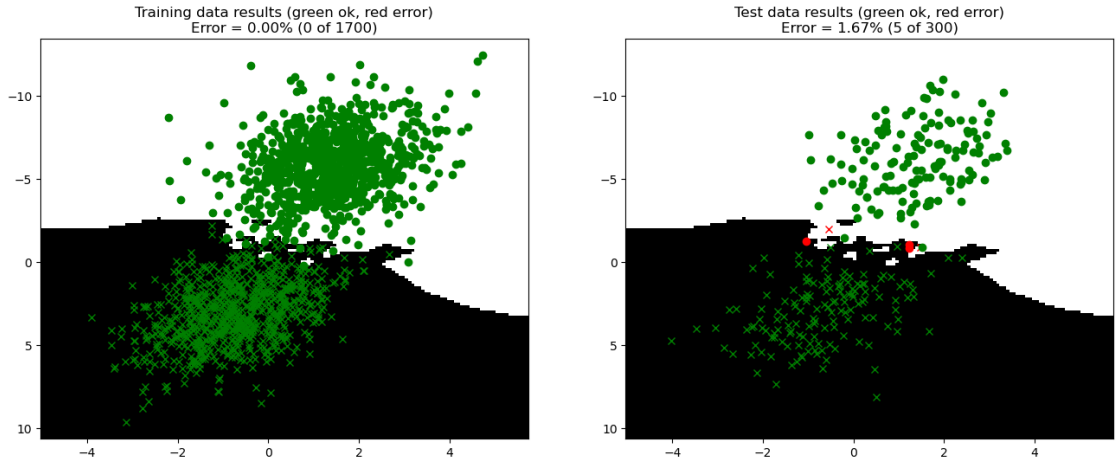
are extracted from the LTrain list and are put into an own list of labels that are closest to the sample X point. In the next step the amount of labels is counted and the most common label is chosen using bincount and argmax. These labels for the X points are stored in LPred and returned. A tie between labels is decided by taking the first occurring label which in this case is always the lowest valued class (0).

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.

[10]:
```
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, LPredTest,␣
↪lambda X: kNN(X, k, XTrain, LTrain))
    else:
        plotConfusionMatrixOCR(XTest, LTest, LPredTest)
```

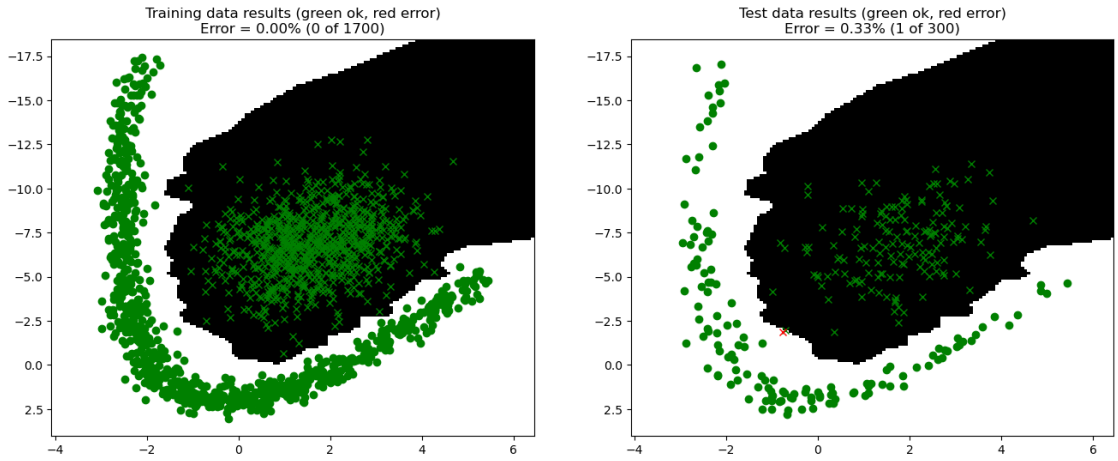
[200]:
```
runkNNOnDataset(1, testSplit=0.15, k=1)
```

```
Train accuracy: 1.0000
Test accuracy: 0.9833
Test data confusion matrix:
[[151   3]
 [  3 144]]
```



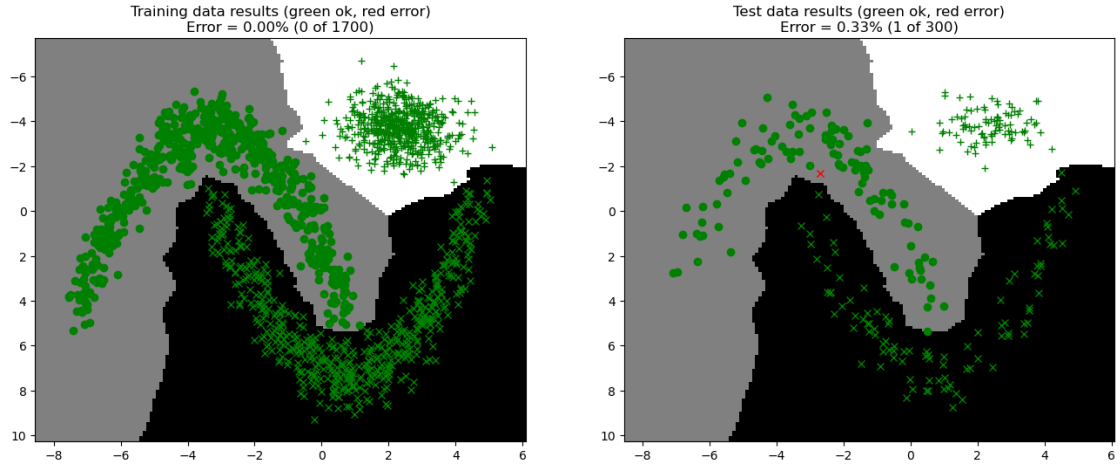
[11]:
```
runkNNOnDataset(2, testSplit=0.15, k=1)
```

```
Train accuracy: 1.0000
Test accuracy: 0.9967
Test data confusion matrix:
[[161   0]
 [  1 138]]
```



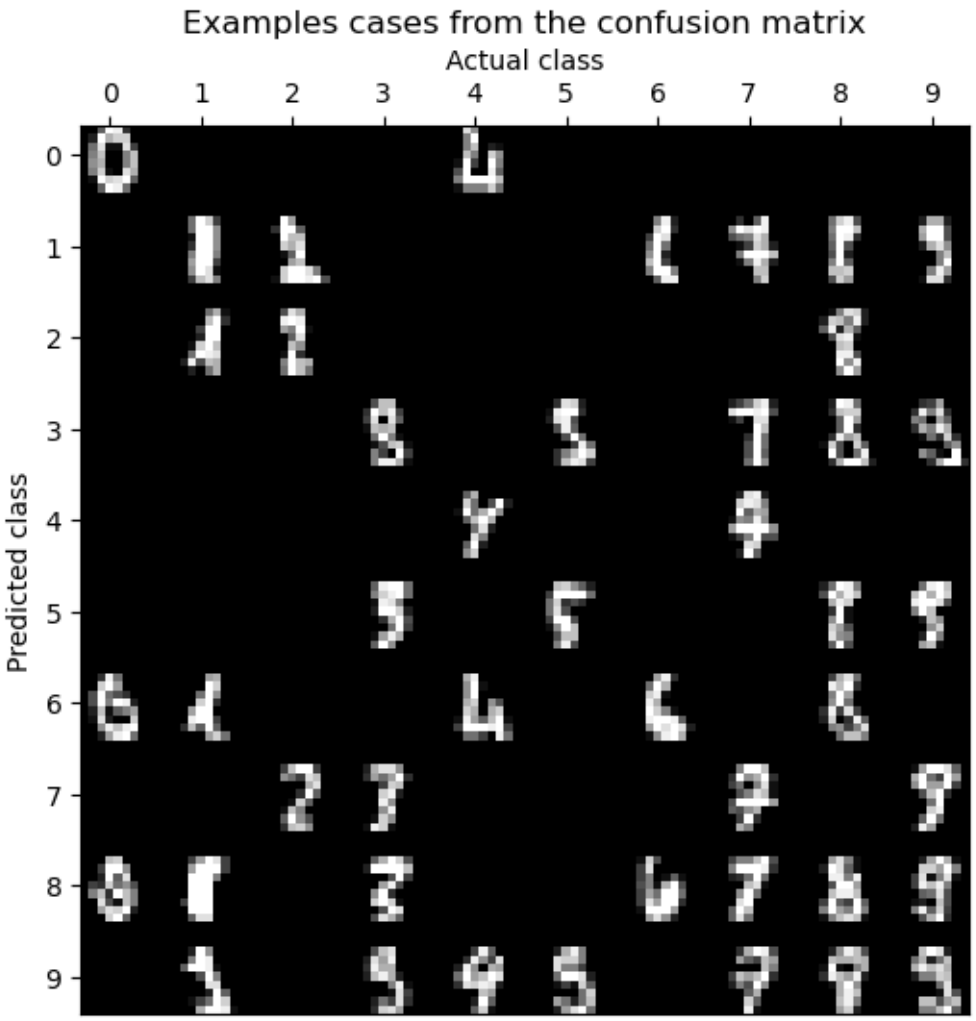
[118]:
```
runkNNOnDataset(3, testSplit=0.15, k=1)
```

```
Train accuracy: 1.0000
Test accuracy: 0.9967
Test data confusion matrix:
[[16843108 16843009 16843009]
 [16843010 16843116 16843009]
 [16843009 16843009 16843101]]
```



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

```
Train accuracy: 1.0000
Test accuracy: 0.9770
Test data confusion matrix:
[[ -359634692          279  -396150944          279          367            0
           -1           -1    27263092     27329404]
 [   75236477     23331869    92078107     60817524      8127604     74711427
     25363069     27329127  1115162256      8127866]
 [    4456835     39597663   142412044     91499640     25363068    542965828
    159187548      7604861   127927916    175900796]
 [   27328536    125568636    40241532   1181811160    879820887      7602264
     74711968     25362557    41682038     40239495]
 [   44107892   1316490109   108791156      4456835     39601166      8980349
     60817524      8914037   141820291    209519265]
 [  161480820    192154228     8914036     25362823     23462023     56886276
     73664644     27329165   209456254    125569919]
 [-1871642563      7602264    74711968     25362556     40239485     44107892
   1081610026      8127860     4456836     39597149]
 [  142411645    192162424    25362813    509411397     58460541    125569916
     40241532      1638840   125570428     40241537]
 [   26214461      5703282     5764209     60817529      8127604    141820291
    243073698     91499641    25362938    542965830]
 [  125633116      7604350   226233504      1640323    228589989    243007651
      3934076   1181811092   243007581    234881508]]
```

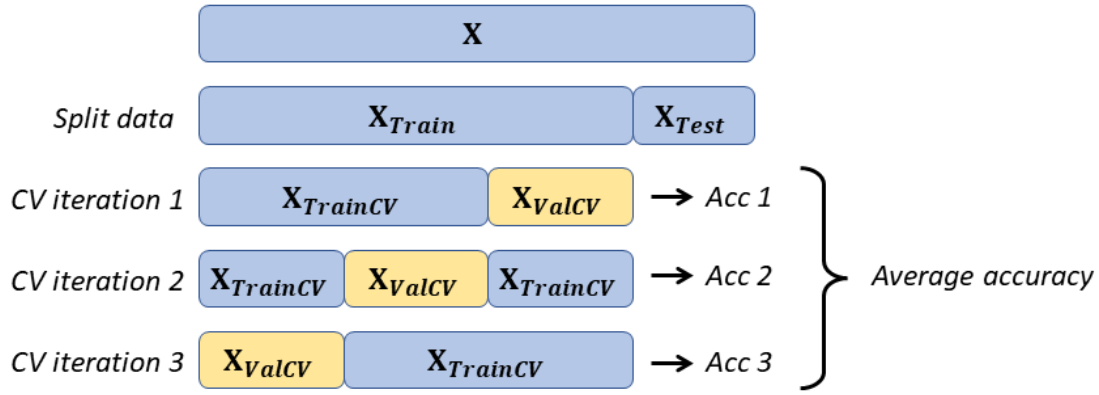


0.0.6 3. Cross-validation

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.

[122]:
```
# 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. Try␣
 ↪printing for example XBins[0].shape.
XBins, _, 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.

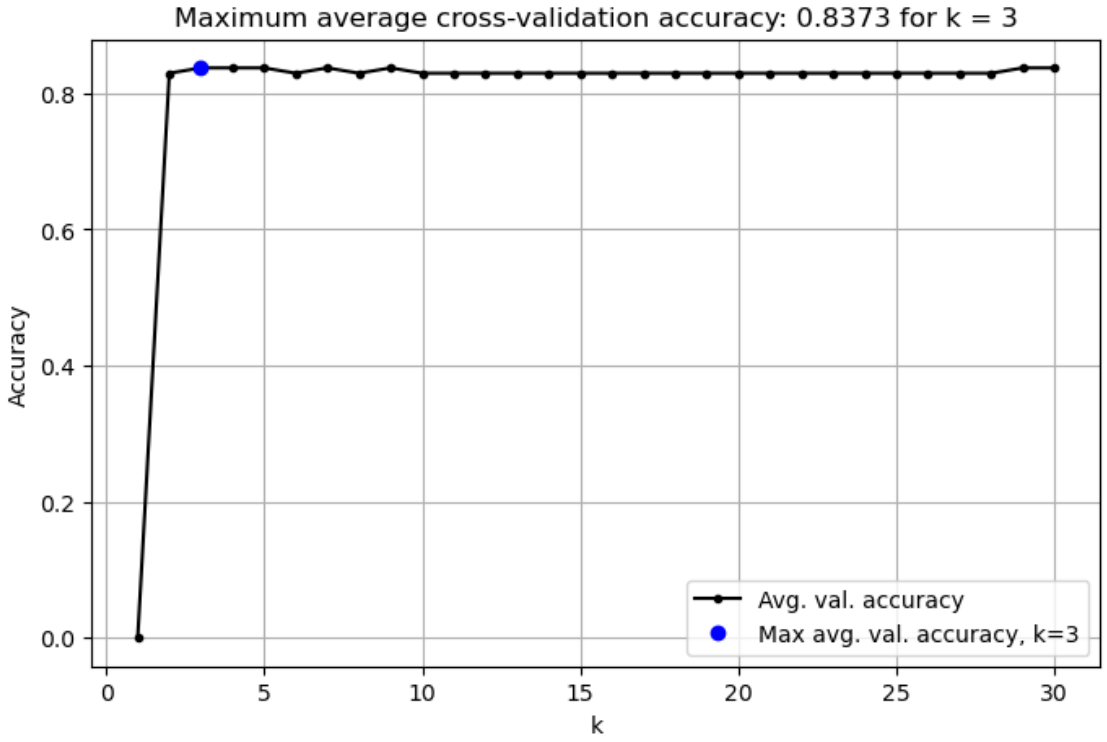
[173]:
```
def crossValiation(kMax, XBins, LBins):
    """Performs cross-validation using kNN

    Args:
        kMax (int): Maximum value of k to test. Values used will be [1-kMax].
        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 rows, and
            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

        XTrainCV, _, LTrainCV, XValCV, _, LValCV = getCVSplit(XBins, None,␣
↪LBins, nBins, i)
        for k in range(1,kMax):

            # Classify validation data using kNN

            LPredValCV = kNN(XValCV, k, XTrainCV, LTrainCV)

            # ... and store resulting accuracy in the accs matrix

            accs[i, k] = calcAccuracy(LPredValCV, LValCV)

            # Print progress and remaining time
            timeLeft = round((tic()-timeStart)*( nBins*kMax / (i*kMax + k + 1)␣
↪- 1))
            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 = [sum(accs[:,k])/kMax for k in range(kMax)]

    # And find the best k
    kBest    = meanAccs.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.

[180]:
```
meanAccs, kBest = crossValiation(30, XBins, LBins)
plotResultsCV(meanAccs, kBest)
```

```
b: 50, k: 30, ETA: 0:00:00
```



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.

[194]:
```
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: 0.9900
Test data confusion matrix:
[[145   2]
 [  4 152]]
```

Question 3:

- Describe how you implemented cross-validation.

Answer: [Your answers here] Firstly we loop over the amount of bins so that we can calculate a different training and validation data for each iteration according to the figure above. Then for every iteration we use getCVSplit to merge the correct bins together to create our training and validation data sets and their corresponding labels.

After that we loop through 1 to kMax k values to test our prediction on to later find the optimal k. For every k we calculate our predicted labels for the predicted validation dataset on the training dataset. We then calculate the accuracy between our predicted validation labels and the true validation labels.

The accuracy gets saved into a matrix with dimension nBins x kMax. So every element in the matrix represents an accuracy for a nested iteration.

The mean of the all the accuracys for each k is calculated by looping over every k and summing the column of all the accuracies from the different bin dataset setup and then dividing it by kMax (the amount of k's)

The best k is easily extracted by taking the max accuracy from this list and finding the index and adding 1 because the accuracies in the list are aligned with the k values, the k values are 1 indexed instead of 0 so we have to add 1.

0.0.7 4. Cross validation for all datasets

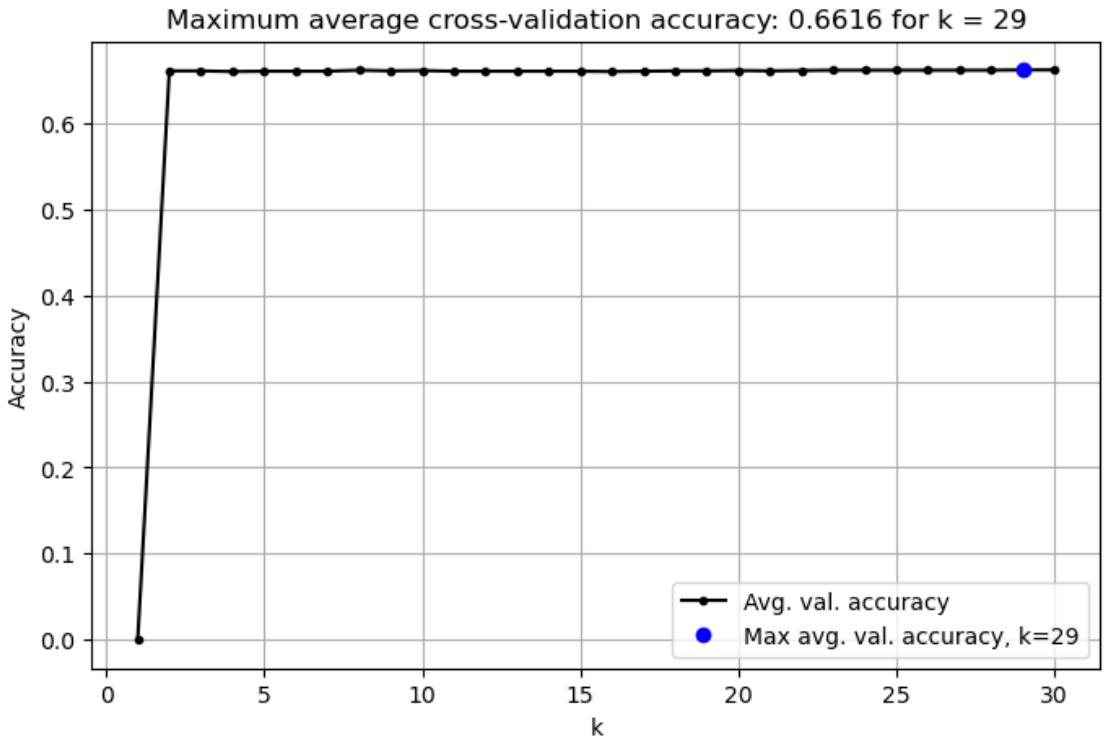
Once again we define a single function that performs all of the previous cross-validation for a given dataset and shows the results. Use it to perform cross-validation on all four datasets.

[195]:
```
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)
```

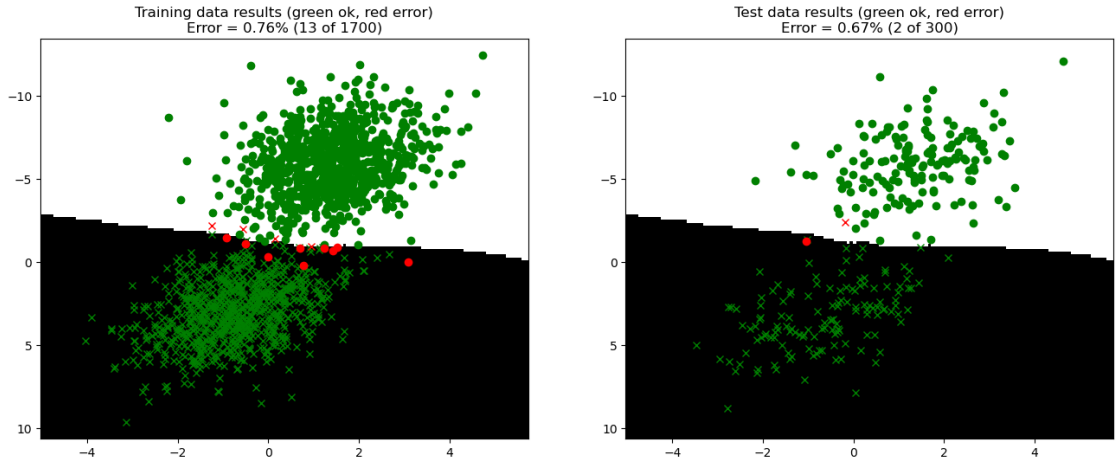
```
    confMatrix = calcConfusionMatrix(LPredTest, LTest)
    accTest = calcAccuracy(LPredTest, LTest)

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

    if datasetNr < 4:
        plotResultsDots(XTrain, LTrain, LPredTrain, XTest, LTest, LPredTest,␣
↪lambda X: kNN(X, kBest, XTrain, LTrain))
    else:
        plotConfusionMatrixOCR(XTest, LTest, LPredTest)
```

[196]:
```
runkNNCrossValidationOnDataset(1, testSplit=0.15, nBins=20, kMax=30)
```

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

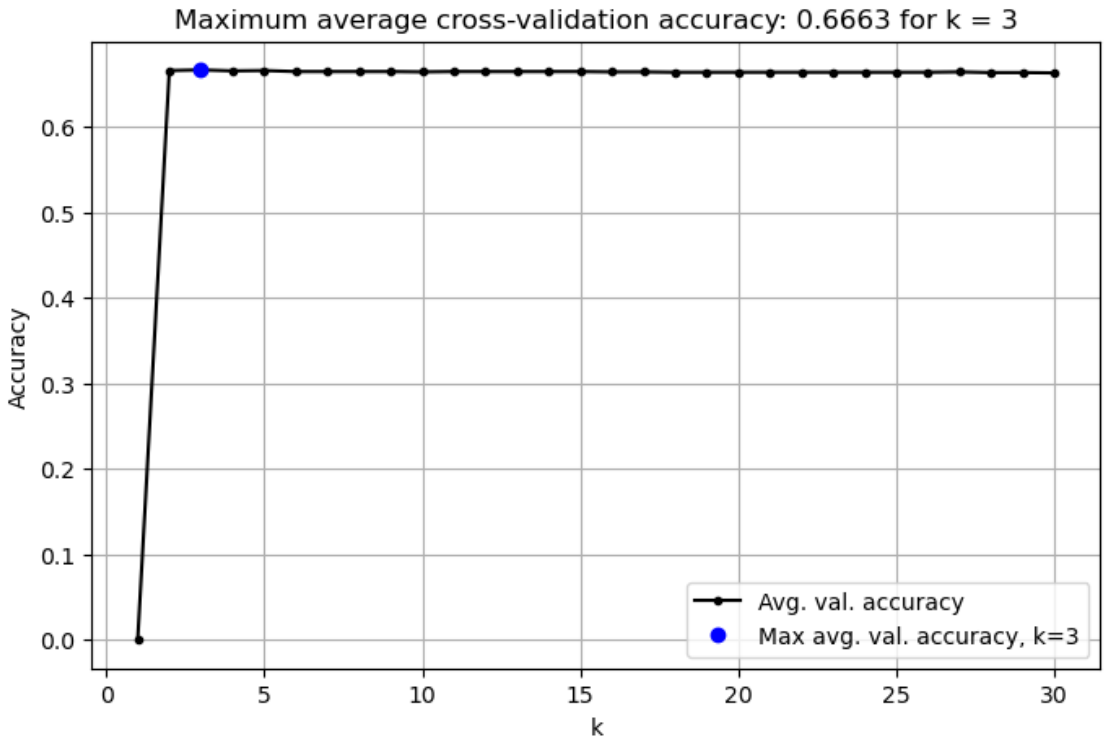



```
Test accuracy: 0.9933
Test data confusion matrix:
[[136   1]
 [ 30 162]]
```



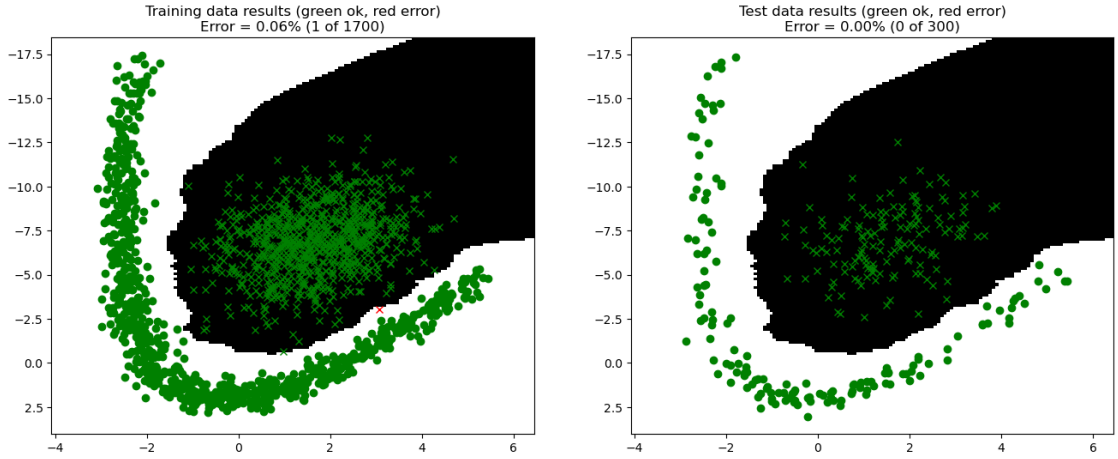
[197]:
```
runkNNCrossValidationOnDataset(2, testSplit=0.15, nBins=20, kMax=30)
```

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



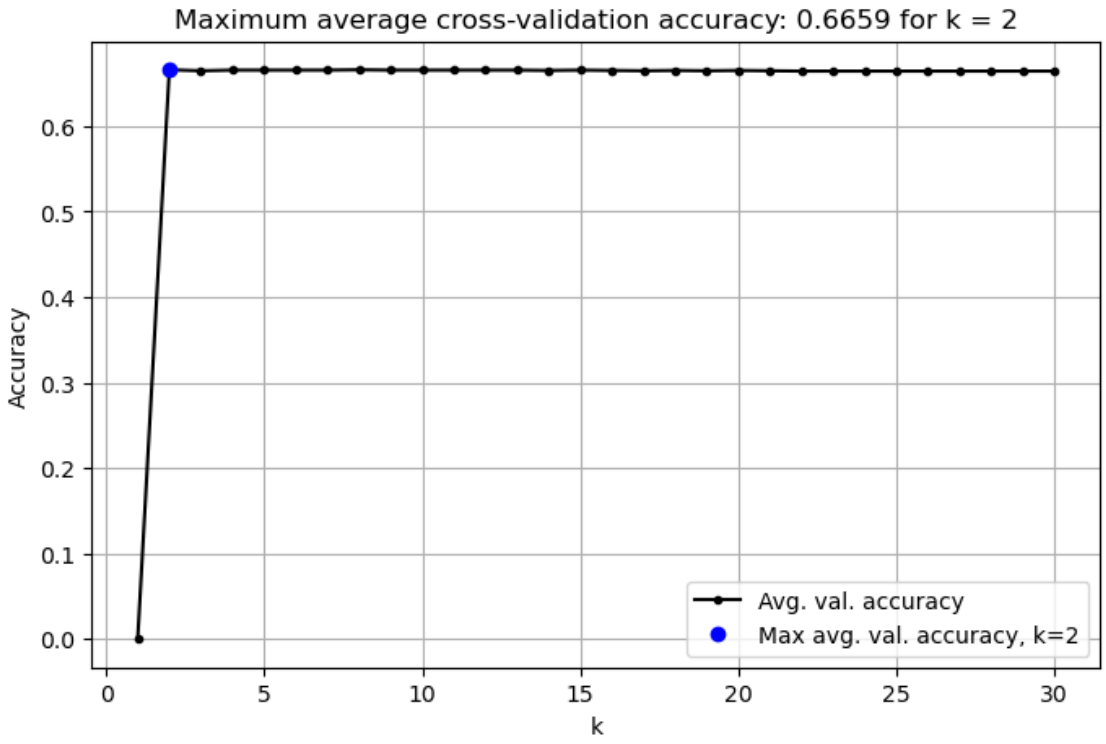

```
Test accuracy: 1.0000
Test data confusion matrix:
[[152   0]
```

```
[  3 148]]
```



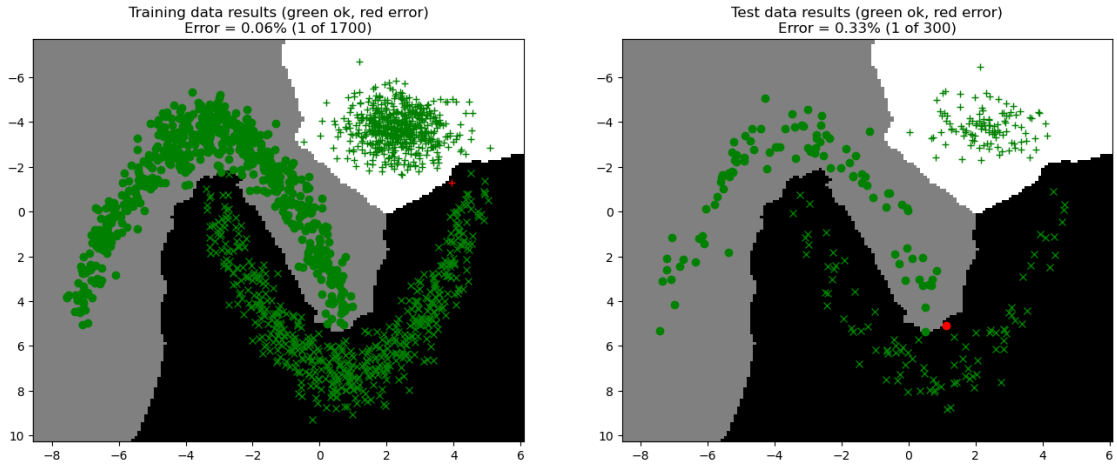
[202]:
```
runkNNCrossValidationOnDataset(3, testSplit=0.15, nBins=20, kMax=30)
```

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



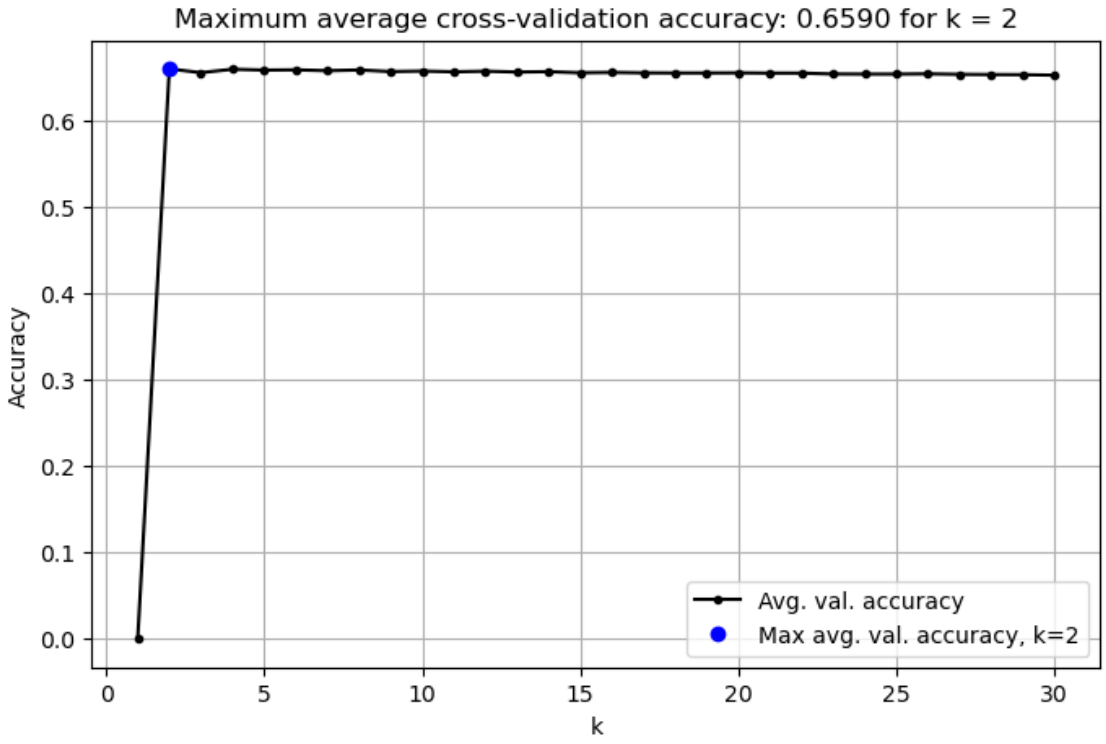

```
Test accuracy: 0.9967
Test data confusion matrix:
```

```
[[ 96   1   0]
 [  0  95   0]
 [  0   0 108]]
```

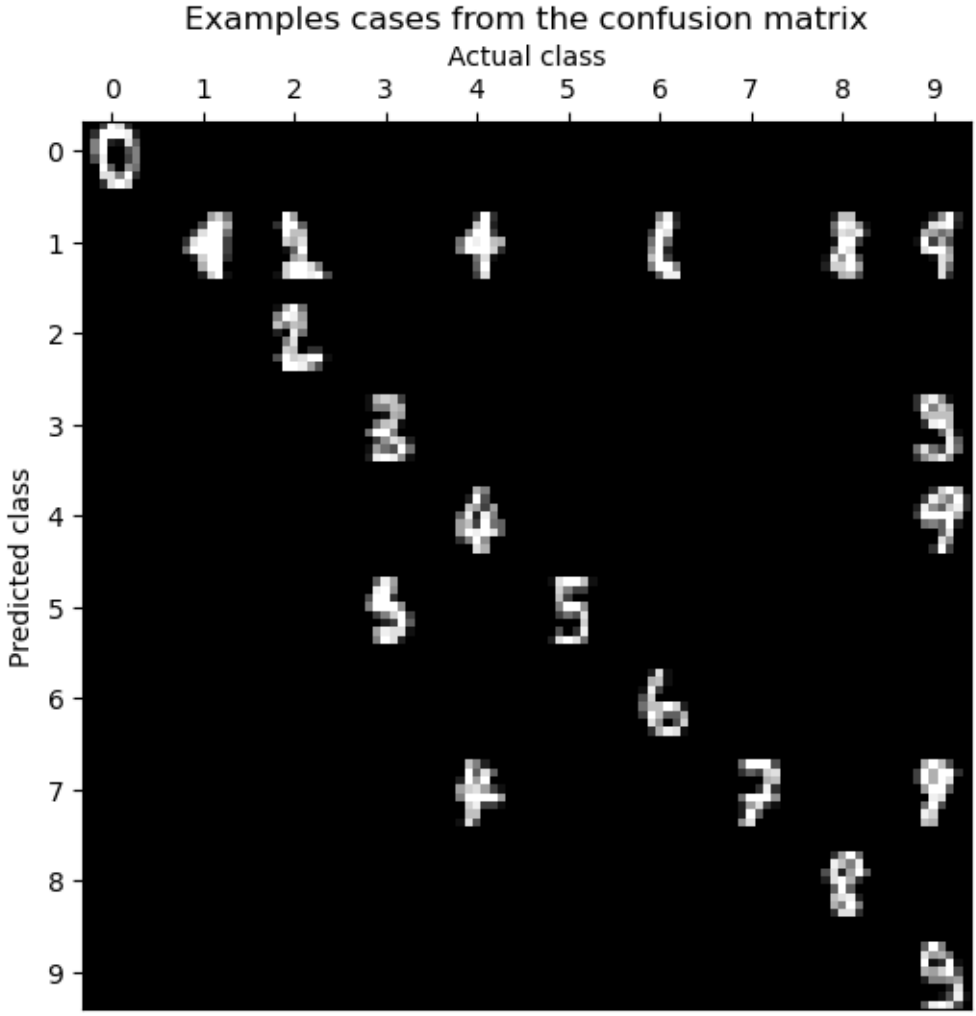


[203]:
```
runkNNCrossValidationOnDataset(4, testSplit=0.15, nBins=20, kMax=30)
```

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



```
Test accuracy: 0.9846
Test data confusion matrix:
[[87  0  0  0  0  0  0  0  0  0]
 [ 0 84  1  0  1  0  2  0  2  1]
 [ 0  0 88  0  0  0  0  0  0  0]
 [ 0  0  0 82  0  0  0  0  0  1]
 [ 0  0  0  0 82  0  0  0  0  1]
 [ 0  0  0  1  0 74  0  0  0  0]
 [ 0  0  0  0  0  0 92  0  0  0]
 [ 0  0  0  0  1  0  0 76  0  2]
 [ 0  0  0  0  0  0  0  0 86  0]
 [ 0  0  0  0  0  0  0  0  0 79]]
```



Question 4:

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

reasonable?

Answer: [Your answers here]

Dataset 1: k=29 This is reasonable as the data is two clusters that are linearly separable and the model can be very simple and still have good results. The hight k value is also because the clusters have a very similar density and volume of points which make the borderpoints be able to to consistently use it a lot of neighbours to classifiy itself.

Dataset 2: k=3 This k value seems resonable for the same motivation as dataset 1. The dataset is not linearly separable and the clusters have very different density and volumes. For example the borderpoints in the cirular cluster would have a lot more closer neighbours with the other cluster if we increase k only a bit, although it is obvious when looking at the data that it belongs to the circular cluster. It is essentially because the circular cluster has a very dense core of points but very sparse at the border.

Dataset 3: k=2 This k value is also reasonable in a very similar way to dataset 2. If you look at the dataset and look at the points at the end of both the boomerang shaped classes, you see that the points are mostly surrounded by the other boomerangs middlepoint whichhas a higher density of points than its own vicinity of samely classified neighours.

Dataset 4: k=2 This k value is reasonable because the resolution of the images of digits is very low which makes classifiying pixels to a digit based on its neighbours very limited. If the images would be a higher resolution, the k-value would most likely increase with it.

0.0.8 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]