Assignment 1: k-nearest neighbors

Automatic Testing Guidelines

Automatic unittesting requires you to submit a notebook which contains strictly defined objects. Strictness of definition consists of unified shapes, dtypes, variable names and more.

Within the notebook, we provide detailed instruction which you should follow in order to maximise your final grade.

Name your notebook properly, follow the pattern in the template name:

Assignment_N_NameSurname_matrnumber

1. N - number of assignment
2. NameSurname - your full name where every part of the name starts with a capital letter, no spaces
3. matrnumber - you student number on ID card (without k, potenitially with a leading zero)

Don't add any cells but use the ones provided by us. You may notice that all cells are tagged such that the unittest routine can recognise them. Before you sumbit your solution, make sure every cell has its (correct) tag!

You can implement helper functions where needed unless you put them in the same cell they are actually called. Always make sure that implemented functions have the correct output and given variables contain the correct data type. In the descriptions for every function you can find information on what datatype an output should have and you should stick to that in order to minimize conflicts with the unittest. Don't import any other packages than listed in the cell with the "imports" tag.

Questions are usually multiple choice (except the task description says otherwise) and can be answered by changing the given variables to either "True" or "False". "None" is counted as a wrong answer in any case!

Note: Never use variables you defined in another cell in your functions directly; always pass them to the function as a parameter. In the unitest, they won't be available either. If you want to make sure that everything is executable for the unittest, try executing cells/functions individually (instead of running the whole notebook).

Plagiarism

Your submissions will be scanned for plagiarism!
If plagiarism is detected, all involved students will automatically receive 0 points for the whole submission, until further explanation is brought forward to us.

Task 1: Risk Calulation (10 Points)

Assume the simple situation of a binary classification task, i.e. $y = \pm 1$, and a 1-dimensional feature x, i.e. $d = 1$. Moreover, the feature is discrete (categorical) and can only have 3 possible values $x = 1, 2, 3$.

Data is drawn from the following distribution:

$$p(x = 1, y = +1) = 0.1 \tag{1}$$
$$p(x = 1, y = -1) = 0.2 \tag{2}$$
$$p(x = 2, y = +1) = 0.2 \tag{3}$$
$$p(x = 2, y = -1) = 0.1 \tag{4}$$
$$p(x = 3, y = +1) = 0.3 \tag{5}$$
$$p(x = 3, y = -1) = 0.1 \tag{6}$$

Compute the risk for the 0-1-loss for these two classifiers:

(1) The Bayes-optimal classifier g_{opt}.
(2) A classifier g_{dom} which always outputs $+1$, i.e. the dominant label, independent of x.

$$R(g(.\,; w))) = \sum_x \sum_y p(x, y \neq g(x, w))$$

$$g_{opt} = sign(p(y = +1|x) - p(y = -1|x)) = [-1, 1, 1]$$

$$R_{opt}(g(.\,; w)) = p(x = 1, y = 1) + p(x = 2, y = -1) + p(x = 3, y = -1) = 0.1 + 0.1 + 0.1$$

$$R_{opt}(g(.\,; w)) = 0.1 + 0.1 + 0.1 = 0.3$$

$$g_{dom} = [1, 1, 1]$$

$$R_{dom}(g(.\,; w)) = p(x = 1, y = -1) + p(x = 2, y = -1) + p(x = 3, y = -1)$$

$$R_{dom}(g(.\,; w)) = 0.2 + 0.1 + 0.1 = 0.4$$

Task 2: Visualization

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

```
from matplotlib.figure import Figure
from matplotlib.collections import PathCollection
```

In [2]:
```
# read data, split into X(features) and y(labels)
Z = np.genfromtxt('DataSet1.csv', delimiter=';')
X, y = Z[:,:-1], Z[:,-1]
```

Task 2.1:

Now visualize the data stored in `DataSet1.csv` with a scatter plot.
The first two columns are the features which hold the x_1 and x_2 coordinates of the data.
The last column provides the labels y (± 1) of the data. Use different colors for different labels.
Always label the axes of all your plots.

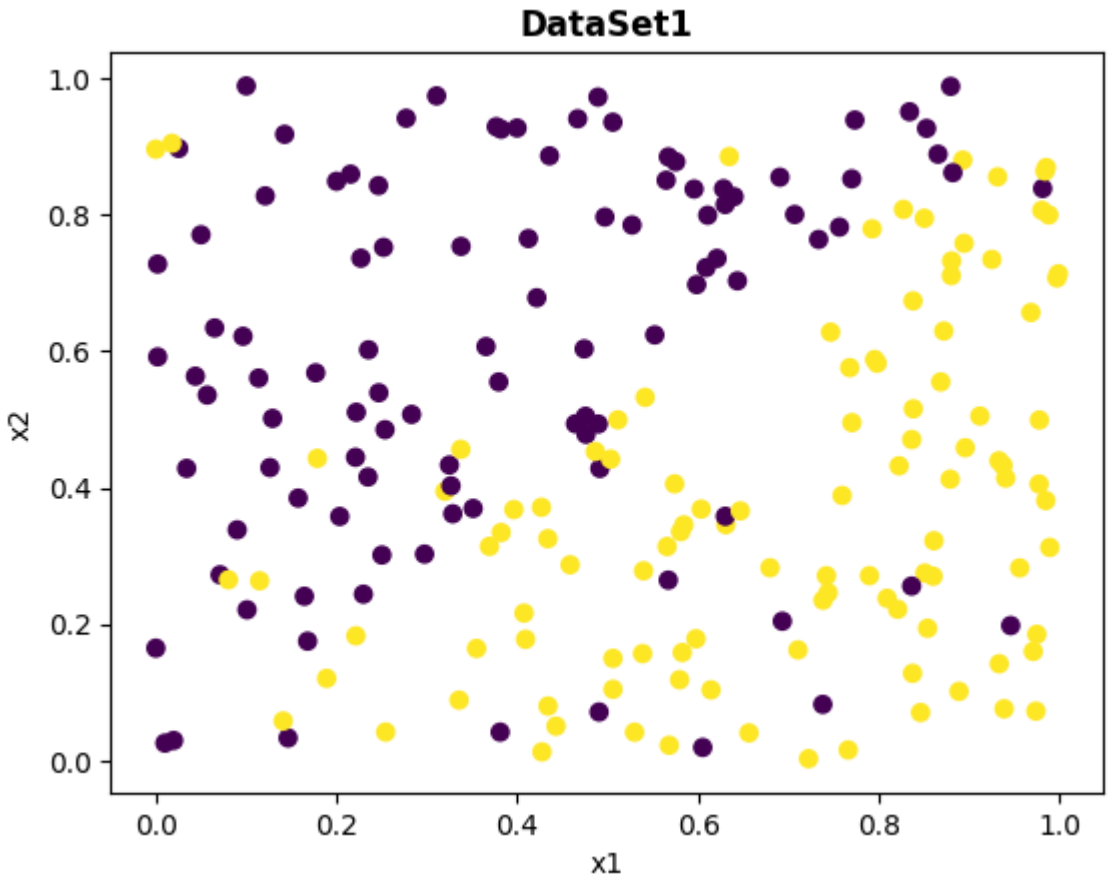
Code & Question 2.1 (5 points):

Plotting

Visualize the data stored in `DataSet1.csv` with a scatter plot by filling out the function below. Make sure to only use function-parameters in your plot.

In [3]:
```
def scatter_plot(X, y):
    """creates a scatter-plot for the dataset X with labels y

    Parameters
    ----------
    X : np.ndarray
        data
    y : np.ndarray
        labels

    Returns
    -------
    Figure
        a matplotlib figure object
    """
    fig = plt.figure()
    # your code goes here ↓↓↓
    plt.scatter(X[:, 0], X[:, 1], c=y)
    plt.title('DataSet1', fontweight ='bold')
    plt.xlabel("x1")
    plt.ylabel("x2")
    return fig
```

In [4]:
```
fig = scatter_plot(X, y)
assert isinstance(fig, Figure)
```



__Answer the following yes/no questions concerning the distribution of the data:__

Are the two classes linearly seperable?

a_) Yes
b_) No

Are there any outliers?

c_) Yes
d_) No

Outliers would affect a kNN algorithm when k is ...?

e_) large (>2)
f_) small (<=2)

To answer the question assign to variables in the next cell "True" or "False" boolean values. To earn points, assign values to all variables. Note: Do not reuse these variable names. They are used for testing.

In [5]:
```
#examples for you
example_of_true_variable = True
example_of_false_variable = False
```

```
#your answers go here ↓↓↓
a_ = False
b_ = True

c_ = True
d_ = False

e_ = False
f_ = True
```

Question 2.2 (5 points):

Which of the following statements about k-nearest neighbors are correct?
(Multiple answers might be correct)

g_) requires very long training time already for small data sets
h_) not suited for large datasets
i_) sensitive to the rescaling of individual features (individual dimensions of feature vectors)
j_) has many trainable model parameters
k_) k-NN only allows for classification, not for regression

In [6]:
```
#examples for you
example_of_true_variable = True
example_of_false_variable = False

#your answers go here ↓↓↓
g_ = False # technically no training time at all as classification is done during c
h_ = True # algorithm does not scale well (more data points means also more possibl
i_ = True # features with larger magnitudes can lead to biased results, therefore r
j_ = False # k-NN has only one model parameter which is the number of neighbors "k"
k_ = False # allows for both (mentioned in slides)
```

Task 3: Training the model

Now we want to put the kNN into action. To this end, work through the following points

- Implement `train_kNN` which fits newly created instance of `KNeighborsClassifier` (`sklearn`) to some training data
- Complete `eval_kNN` such that it outputs the prediction for some input data using a passed classifier
- Program the function `mean_zero_one_loss` that calculates the mean zero-one loss (see lecture slides) of predicted values and samples from the test set
- Put all of these functions together in `run_kNN` to fit a model to training data, make predictions on left-out (test) data and compute the loss for these predictions. To split the dataset into training and test sets, use 10-fold cross validation (CV), loop over all the splits and collect the mean error for each split.

At the end of this task, visualize the mean error for $k \in \{1, 3, 5, \ldots, 177, 179\}$ in a plot and answer the following questions.

In [7]:
```
from sklearn.model_selection import KFold
from sklearn import neighbors
```

Code 3.1 (5 points):

In [8]:
```
def train_kNN(X_train,y_train,k_train):
    """
    Function that fits a kNN to given data
    @param X_train, np array, training data
    @param y_train, np array, training labels
    @param k_train, integer, k for the kNN

    @output classifier, kNN instance, classifier that was fitted to training data
    """
    #your code goes here ↓↓↓
    classifier = neighbors.KNeighborsClassifier(n_neighbors=k_train)
    classifier.fit(X_train, y_train)

    return classifier
```

Code 3.2 (5 points):

In [9]:
```
def eval_kNN(classifier, X_eval):
    """
    Function that returns predictions for some input data
    @param classifier, kNN instance, trained kNN classifier
    @param X_eval, np array, data that you want to predict the labels for

    @output predicitons, np array, predicted labels
    """
    #your code goes here ↓↓↓
    predictions = classifier.predict(X_eval)

    return predictions
```

Code 3.3 (5 points):

In [10]:
```
def mean_zero_one_loss(y_true, y_pred):
    """
    Function that calculates the mean zero-one loss for given true and predicted la
    @param y_true, np array, true labels
    @param y_pred, np array, predicted labels

    @output loss, float, mean zero-one loss
    """
    #your code goes here ↓↓↓
    loss = sklearn.metrics.zero_one_loss(y_true, y_pred)
```

```
    return loss
```

Code 3.4 (5 points):

In [11]:
```
def run_kNN(X,y,nf,k):
    """
    Function that combines all functions using CV
    @param X, np array, training data
    @param y, np array, training labels
    @param nf, integer, number of folds for CV
    @param k, integer, k for kNN

    @output mean_error, float, mean error over all folds
    """
    #your code goes here ↓↓↓
    error = 0
    # split lists into nf distinct sets
    sub_sets_X = np.array_split(X, nf)
    sub_sets_y = np.array_split(y, nf)

    for i in range(nf):
        # use set at index i as evaluation set
        classifier = train_kNN(
            np.concatenate(sub_sets_X[:i] + sub_sets_X[i+1:]), # leave out test sub
            np.concatenate(sub_sets_y[:i] + sub_sets_y[i+1:]),
            k)
        predictions = eval_kNN(classifier, sub_sets_X[i])
        error += mean_zero_one_loss(sub_sets_y[i], predictions)

    return error/nf
```

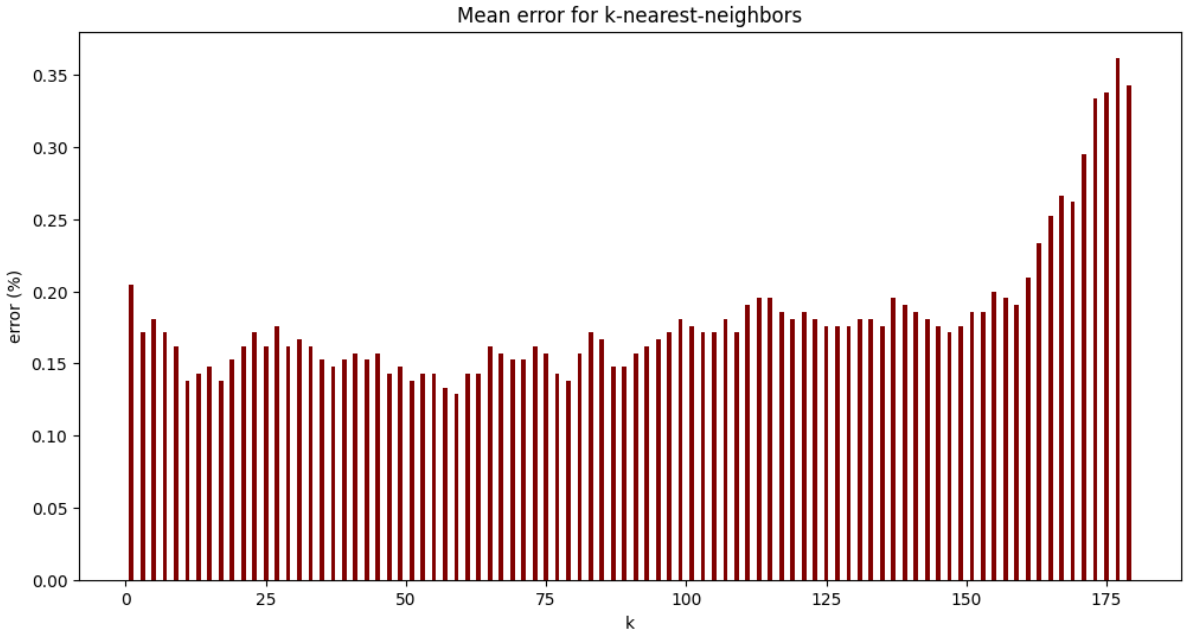
In [12]:
```
# Nothing to do here - just run this cell
m = 179
nf = 10
error_holder = []
for k in range(1,m+1,2): #range with 179 included and step of 2
    error_holder.append(run_kNN(X,y,nf,k))
```

Code & Question 3.5 (5 points):

In [13]:
```
#implement the plot as described in the task description within the function below

def plot_error_vs_k(error_holder, m):
    """create a plot

    Parameters
    ----------
    error_holder : list[float]
        the error list generated bove
    m : int
        range in the error list (see above)
```

```
    Returns
    -------
    Figure
        matplotlib figure object
    """
    fig = plt.figure(figsize=(12, 6))
    # your code goes here ↓↓↓
    plt.bar(range(1, m+1, 2), error_holder, color='maroon')

    plt.xlabel("k")
    plt.ylabel("error (%)")
    plt.title("Mean error for k-nearest-neighbors")

    return fig
```

In [14]:
```
fig = plot_error_vs_k(error_holder, m)
assert isinstance(fig, Figure)
plt.show()
```



__Answer the following questions about the plot you just created:__

What range for k holds the lowest errors (on average) - by visual inspection?

l_) [0,5]
m_) [50,60]
n_) [150,175]

Is the error larger for $k = 175$ or for $k = 1$?

o_) 175
p_) 1

In [15]:
```
#examples for you
example_of_true_variable = True
```

```
example_of_false_variable = False

#your answers go here ↓↓↓
l_ = False
m_ = True
n_ = False

o_ = True
p_ = False
```

Question 3.6 (10 points):

Thinking of model complexity as the ability of the model to fit to noise, what choice of k leads to complex models? Why?
(Multiple answers might be correct)

q_) Model complexity increases with increasing k, as larger k means the model has more parameters.
r_) Model complexity increases with increasing k, as larger k means that more neighbors influence the decision.
s_) Model complexity increases with decreasing k, as smaller k means that fewer neighbors influence the decision.
t_) Model complexity increases with decreasing k, as smaller k means the model has fewer parameters.

u_) Very small values of k lead to underfitting
v_) Very small values of k lead to overfitting
w_) Very large values of k lead to underfitting
x_) Very large values of k lead to overfitting

In [16]:
```
#your answers go here ↓↓↓
q_ = False
r_ = True
s_ = False
t_ = False

u_ = True
v_ = False
w_ = False
x_ = True
```

Task 4: Adding noise to dataset

To make things more interesting, mix up the dataset a bit:

- Implement `generate_flip_vector` which should return a 1-dimensional array of length n where exactly $\lfloor n/5 \rfloor$ entries are -1 and the rest are $+1$. The entries in the array should appear in a random order.

- Now implement `flip_labels` that flips the labels according to the flip vector.

Then perform the same steps as before. Generate the following 4 subplots in one big plot:
(1) top left: visualize the original dataset (with data points colored differently according to the binary labels)
(2) top right: visualize the flipped dataset (with data points colored differently according to the binary labels)
(3) bottom left: visualize the mean error for $k \in \{1, 3, 5, \ldots, 177, 179\}$ (same as in Task 2) for the original dataset
(4) bottom right: visualize the mean error for $k \in \{1, 3, 5, \ldots, 177, 179\}$ (same as in Task 2) for the flipped dataset

plot the data and plot the error (estimated via the empirical risk) vs. k for 10-fold cross validation.

Code 4.1 (5 points):

In [17]:
```
def generate_flip_vector(n):
    """
    Function that produces a flip vector consisting of -1's and 1's (1/5,4/5)
    @param n, integer, the length of the vector that should be returned

    @output flip_vector, np array, the vector that indicates what labels will be fl
    """
    #your code goes here ↓↓↓
    i = int(np.floor(n / 5))
    flip_vector = np.ones(n)
    flip_vector[:i] = [-1] * i
    np.random.shuffle(flip_vector)

    return flip_vector
```

Code 4.2 (5 points):

In [18]:
```
def flip_labels(y,flip_vector):
    """
    Function that flips labels given a flip vector
    @param y, np array, labels to flip (don't forget to copy the data in order not
    @param flip_vector, np array, array that should be used to flip the labels

    @output flipped_labels, np array, the labels where 1/5 labels are flipped
    """
    #your code goes here ↓↓↓
    return np.multiply(y, flip_vector)
```

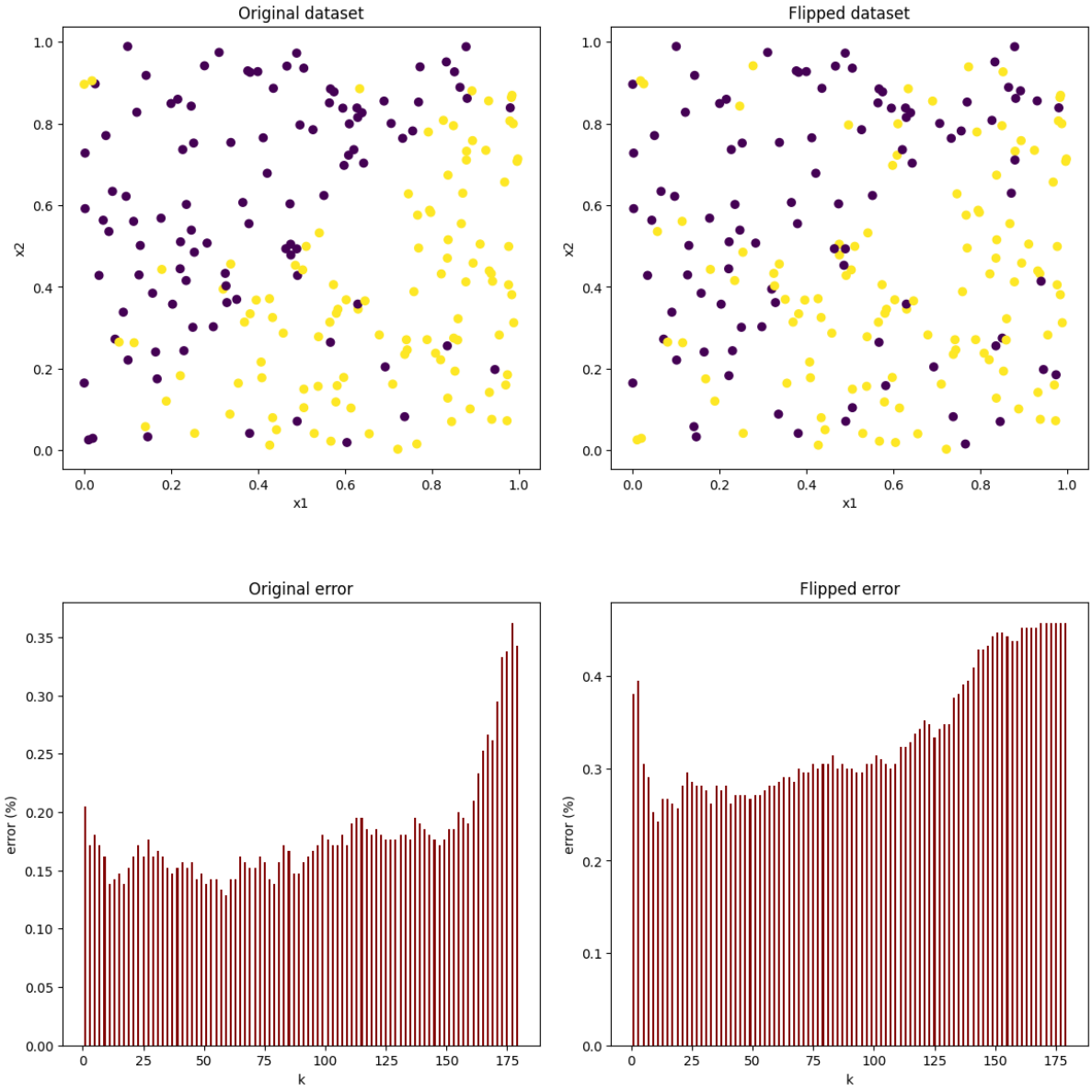
In [19]:
```
# Nothing to do here - just run this cell
# define new y vector by calling flip function
fl_vec = generate_flip_vector(len(y))
y_fl = flip_labels(y,fl_vec)
```

```
error_holder_flipped = []
for k in range(1,m+1,2): # range with 179 included and step of 2
    error_holder_flipped.append(run_kNN(X,y_fl,nf,k))
```

Plot & Question 4.3 (10 points)

In [20]:
```
def plot_subplots(X, y, y_fl, m, error_holder, error_holder_flipped):
    """The function creating the plots as described above.

    The Parameters are called the same as in the notebook.
    Do not use anything besides them.
    """
    fig, axs = plt.subplots(2,2,figsize=(14, 14), gridspec_kw = {'wspace':0.15,'hsp
    #your plotting code goes here ↓↓↓
    # original dataset
    axs[0, 0].scatter(X[:, 0], X[:, 1], c=y)
    axs[0, 0].set_title('Original dataset')
    axs[0, 0].set(xlabel='x1', ylabel='x2')

    # flipped dataset
    axs[0, 1].scatter(X[:, 0], X[:, 1], c=y_fl)
    axs[0, 1].set_title('Flipped dataset')
    axs[0, 1].set(xlabel='x1', ylabel='x2')

    # original error
    axs[1, 0].bar(range(1, m + 1, 2), error_holder, color='maroon')
    axs[1, 0].set(xlabel='k', ylabel='error (%)')
    axs[1, 0].set_title('Original error')

    # flipped error
    axs[1, 1].bar(range(1, m + 1, 2), error_holder_flipped, color='maroon')
    axs[1, 1].set(xlabel='k', ylabel='error (%)')
    axs[1, 1].set_title('Flipped error')

    return fig
```

In [21]:
```
fig = plot_subplots(X, y, y_fl, m, error_holder, error_holder_flipped)
assert isinstance(fig, Figure)
plt.show()
```



*__Which differences do you observe?__
__Which conclusions do you draw from that?__*
(Multiple answers might be correct)

y_) After flipping, the two classes are well separable and have nearly no overlap anymore
z_) After flipping, the two classes are less separable (than before) and have a larger overlap (than before)

a2_) Random label flipping brings noise into the data
b2_) Random label flipping simply swaps data, but does not add noise

c2_) Overall, the error remains nearly unchanged compared to the original data set.
d2_) Overall, the error increases compared to the original data set.

To answer the question assign to variables in the next cell "True" or "False" boolean values. To earn points, assign values to all variables. Note: Do not reuse these variable names. They are used for testing.

In [22]:
```
#your answers go here ↓↓↓
y_ = False
z_ = True

a2_ = True
b2_ = False

c2_ = False
d2_ = True
```

Task 5: k-NN in higher dimensions

Going back to unflipped labels: Write a function "add_features(X)" which will add 4 additional features x_3, x_4, x_5, x_6 to the matrix X, calling the resulting matrix X_new. Each of the new features should be uniformly distributed between 0 and 1.

As before, plot the error versus k for 10-folds CV for with 1, 2, 3, 4 incrementally added features. (4 plots)

Additionaly, for the particular choice $k = 11$, plot the mean error versus f with $f = 2, 3, 4, 5, 6$ being the number of features. Thus, the first data point in this plot (where $f = 2$) shows the error for the original feature matrix X without extra dimensions. The second data point in this plot (where $f = 3$) shows the error for the feature matrix X with one additional dimension, and so on.

Code 5.1 (10 points):

In [23]:
```
def add_features(X):
    """
    Function that adds random features to dataset
    @param X, np array, dataset

    @output X_new, np array, dataset enhanced with 4 random features
    """
    np.random.seed(1234)
    #your code goes here ↓↓↓
    # generate matrix of random numbers with 4 columns
    C = np.random.rand(X.shape[0], 4)
    # append C to X along column axis
    X_new = np.append(X, C, axis=1)

    return X_new
```

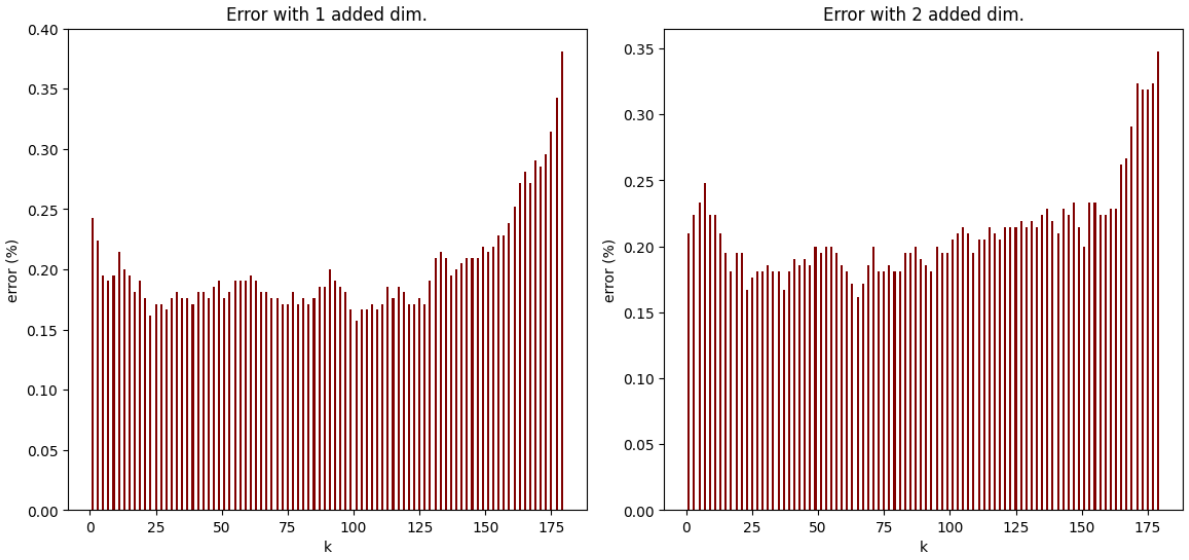
In [24]:
```
# Nothing to do here - just run this cell
# define new feature matrix by calling add_features function
X_new = add_features(X)
```

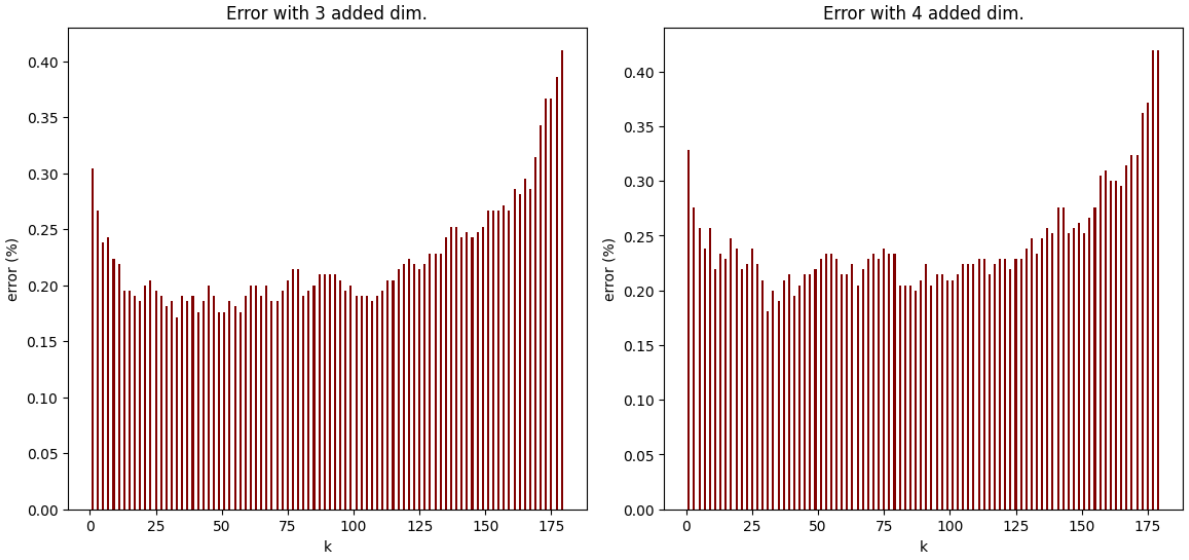
Code & Question 5.2 (15 points):

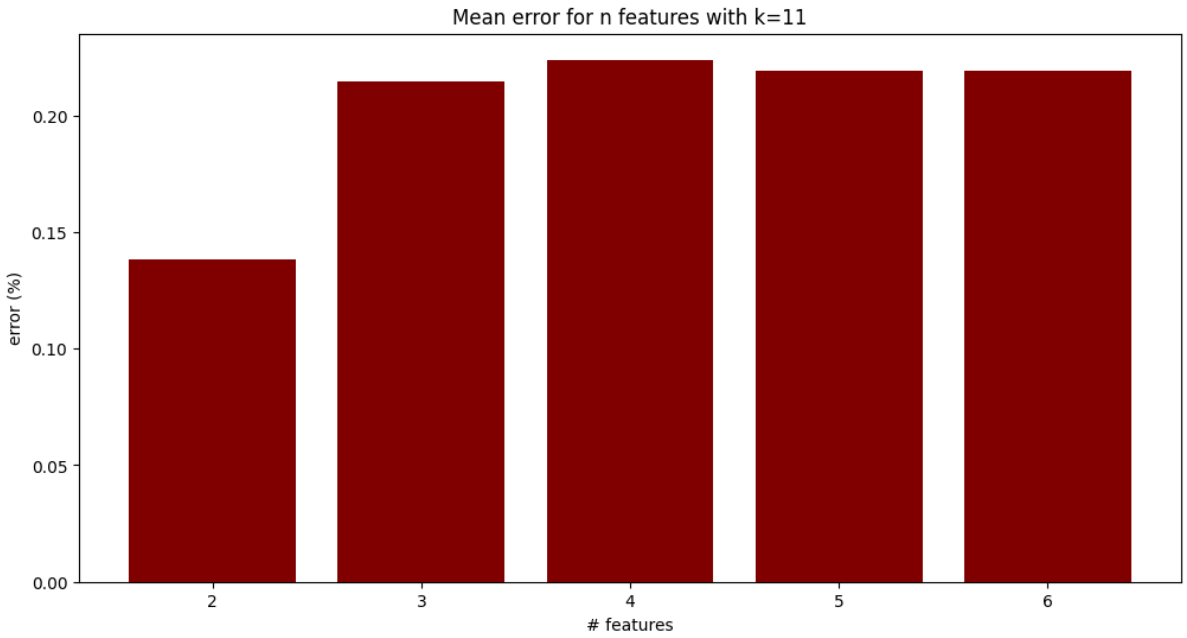
In [25]:
```
m = 179
def plot_error_vs_k_extra_dims(X_new, y, m):
    """function that implements the plot from Task5.

    This function should create 2! plots.
    The first plot to visualize error vs k with 10 folds and <X> extra dimensions.
    The second plot to visualize the error versus dimension with k=11

    Returns
    -------
    tuple[Figure, Figure]
        A tuple with 2! matplotlib figures
    """
    #your code goes here ↓↓↓
    fig1, axs = plt.subplots(2,2,figsize=(14, 14),gridspec_kw = {'wspace':0.15,'hsp
    # implement first plot
    # 1 extra dim
    error_holder = []
    for k in range(1, m + 1, 2):  # range with 179 included and step of 2
        error_holder.append(run_kNN(X_new[:, 0:3], y, 10, k))
    axs[0, 0].bar(range(1, m + 1, 2), error_holder, color='maroon')
    axs[0, 0].set(xlabel='k', ylabel='error (%)')
    axs[0, 0].set_title('Error with 1 added dim.')

    # 2 extra dim
    error_holder = []
    for k in range(1, m + 1, 2):  # range with 179 included and step of 2
        error_holder.append(run_kNN(X_new[:, 0:4], y, 10, k))
    axs[0, 1].bar(range(1, m + 1, 2), error_holder, color='maroon')
    axs[0, 1].set(xlabel='k', ylabel='error (%)')
    axs[0, 1].set_title('Error with 2 added dim.')

    # 3 extra dim
    error_holder = []
    for k in range(1, m + 1, 2):  # range with 179 included and step of 2
        error_holder.append(run_kNN(X_new[:, 0:5], y, 10, k))
    axs[1, 0].bar(range(1, m + 1, 2), error_holder, color='maroon')
    axs[1, 0].set(xlabel='k', ylabel='error (%)')
    axs[1, 0].set_title('Error with 3 added dim.')

    # 4 extra dim
    error_holder = []
    for k in range(1, m + 1, 2):  # range with 179 included and step of 2
        error_holder.append(run_kNN(X_new, y, 10, k))
    axs[1, 1].bar(range(1, m + 1, 2), error_holder, color='maroon')
    axs[1, 1].set(xlabel='k', ylabel='error (%)')
    axs[1, 1].set_title('Error with 4 added dim.')

    fig2 = plt.figure(figsize=(12, 6))
    # implement second plot
    # no extra dim
    feature_error = []
    for f in range(2,7):
        feature_error.append(run_kNN(X_new[:, 0:f], y, 10, 11))

    plt.bar(range(2,7), feature_error, color='maroon')
```

```
    plt.xlabel("# features")
    plt.ylabel("error (%)")
    plt.title("Mean error for n features with k=11")

    return fig1, fig2
```

In [26]:
```
fig1, fig2 = plot_error_vs_k_extra_dims(X_new, y, m)
assert isinstance(fig, Figure)
assert isinstance(fig2, Figure)
```







__Try to explain possible changes of the error.__

e2_) k-nearest neighbors is robust against randomly added further features; noise is filtered out
f2_) k-nearest neighbors is not robust aginst randomly added further features; the error increases if extra dimension(s) with noise are added

g2_) The error with additional features ($f \geq 3$) is overall larger than for $f = 2$
h2_) The error with additional features ($f \geq 3$) is overall smaller than for $f = 2$
i2_) The error stays overall the same with additional features

In [27]:
```
#your answers go here ↓↓↓
e2_ = False
f2_ = True

g2_ = True
h2_ = False
i2_ = False
```