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).

Task 1: Visualization

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

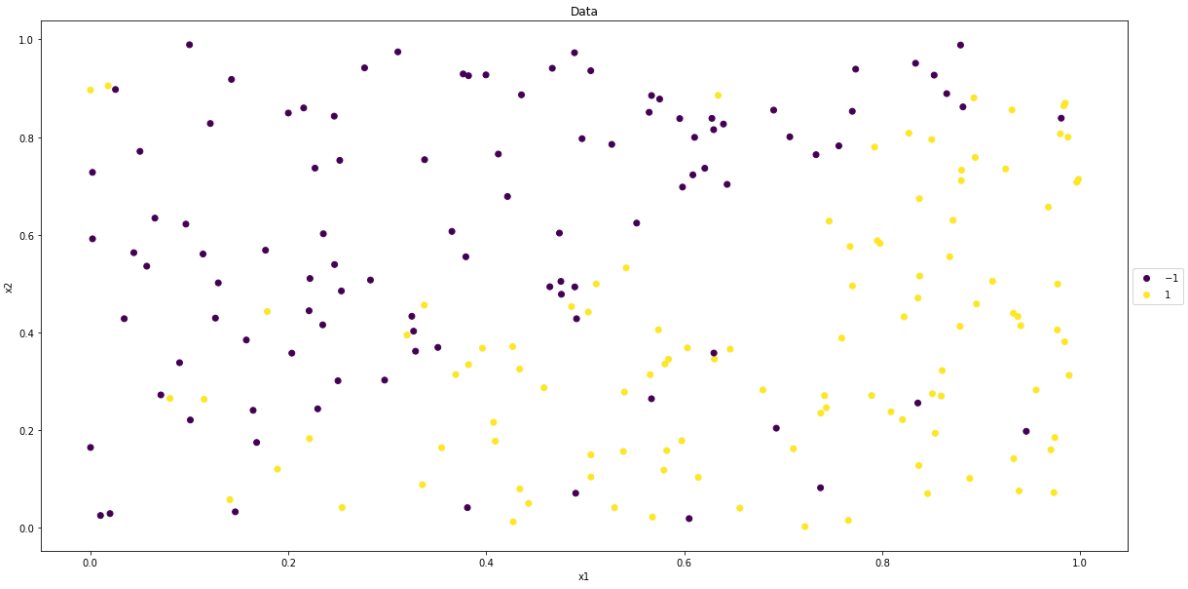
In [2]:
```
# read data, split into X(features) and y(labels)
Z = np.genfromtxt('DataSet1.csv', delimiter=';')
X, y = Z[:,:-1], Z[:,-1]
```

Task 1.1:

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 1.1 (10 points):

In [3]:
```
# your code goes here ↓↓↓

plt.rcParams['figure.figsize'] = (20, 10)
scatter = plt.scatter(X[:, 0],X[:, 1], c=y)
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Data")
plt.legend(handles=scatter.legend_elements()[0], labels =scatter.legend_elements()
plt.show()
```



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

Would outliers affect a kNN algorithm if k is small (<= 2)?

e) Yes
f) No

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 [4]:
```
#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_ = True
f_ = False
```

Question 1.2 (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
j) has many trainable model parameters

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

#your answers go here ↓↓↓
g_ = False
h_ = True
i_ = True
j_ = False
```

Task 2: 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 this functions together in `run_kNN` to fit a model to training data, make predictions on left our data and compute the loss for these predictions. To split the dataset into train 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 [6]:
```
from sklearn.model_selection import KFold
from sklearn import neighbors
```

Code 2.1 (5 points):

In [7]:
```
"""
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
"""
def train_kNN(X_train,y_train,k_train):
    #your code goes here ↓↓↓
    classifier = neighbors.KNeighborsClassifier(k_train)
    classifier.fit(X_train,y_train)

    return classifier
```

Code 2.2 (5 points):

In [8]:
```
"""
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
"""
def eval_kNN(classifier, X_eval):
    #your code goes here ↓↓↓
    predictions = classifier.predict(X_eval)

    return predictions
```

Code 2.3 (5 points):

In [9]:
```
"""
Function that calculates the mean zero-one loss for given true and predicted labels
@param y_true, np array, true labels
@param y_pred, np array, predicted labels

@output loss, float, mean zero-one loss
"""
def mean_zero_one_loss(y_true, y_pred):
    #your code goes here ↓↓↓
```

```
    loss =np.where(y_true==y_pred,0,1)
    loss = np.mean(loss)

    return loss
```

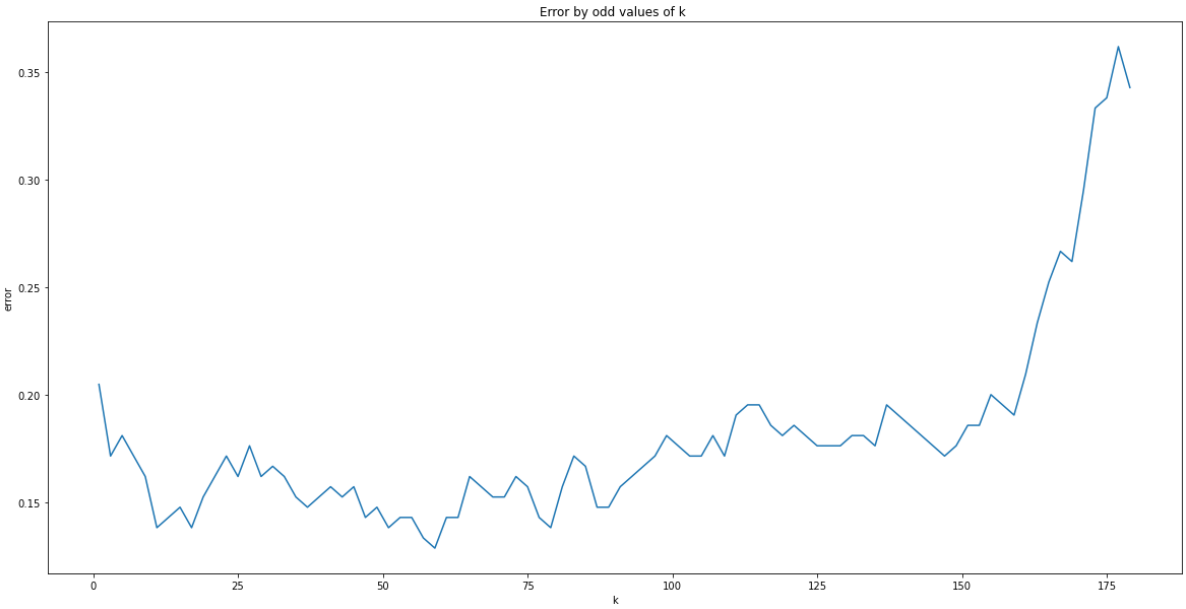
Code 2.4 (5 points):

In [10]:
```
"""
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
"""
def run_kNN(X,y,nf,k):
    #your code goes here ↓↓↓

    kf = KFold(n_splits=nf)
    mean_error = []
    for train_index, test_index in kf.split(X):

        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        classifier = train_kNN(X_train,y_train,k)
        y_pred = eval_kNN(classifier, X_test)

        mean_error.append(mean_zero_one_loss(y_test, y_pred))

    mean_error = np.array(mean_error)
    mean_error = np.mean(mean_error)

    return mean_error
```

In [11]:
```
# 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 2.5 (5 points):

In [12]:
```
#implement the plot as described in the task description
m = 179
k=range(1,m+1,2)

plt.plot(k, error_holder)
plt.xlabel("k")
plt.ylabel("error")
plt.title("Error by odd values of k ")

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?

k*) [0,5]*
l) [50,60]
m_) [150,175]

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

n*) 175*
o) 1

In [13]:
```
#examples for you
example_of_true_variable = True
example_of_false_variable = False

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

n_ = True
o_ = False
```

Question 2.6 (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)

p*) Model complexity increases with increasing k, as larger k means the model has more parameters.*
q) Model complexity increases with increasing k, as larger k means that more neighbors influence the decision.
r*) Model complexity increases with decreasing k, as smaller k means that fewer neighbors influence the decision.*

s) Model complexity increases with decreasing k, as smaller k means the model has fewer parameters.

t*) Very small values of k correspond lead to underfitting*
u) Very small values of k correspond lead to overfitting
v*) Very large values of k correspond lead to underfitting*
w) Very large values of k correspond lead to overfitting

In [14]:
```
#your answers go here ↓↓↓
p_ = False
q_ = False
r_ = True
s_ = False

t_ = True
u_ = False
v_ = False
w_ = True
```

Task 3: Adding noise to dataset

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

- Implement `generate_flip_vector` which should return an array of length n where exactly $\lfloor n/6 \rfloor$ entries as -1 and the rest are 1
- Now implement `flip_labels` that flips the labels according to the flip vector

Then perform the same steps as before, i.e. plot the data and plot the error (estimated via the empirical risk) vs. k for 10-fold cross validation.

Code 3.1 (10 points):

In [15]:
```
"""
Function that produces a flip vector consisting of -1's and 1's (1/6,5/6)
@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 flippe
"""

def generate_flip_vector(n):
    #your code goes here ↓↓↓

    flip_vector = np.random.choice([-1,1], p=[1/6,5/6], size=n)

    return flip_vector
```

Code 3.2 (5 points):

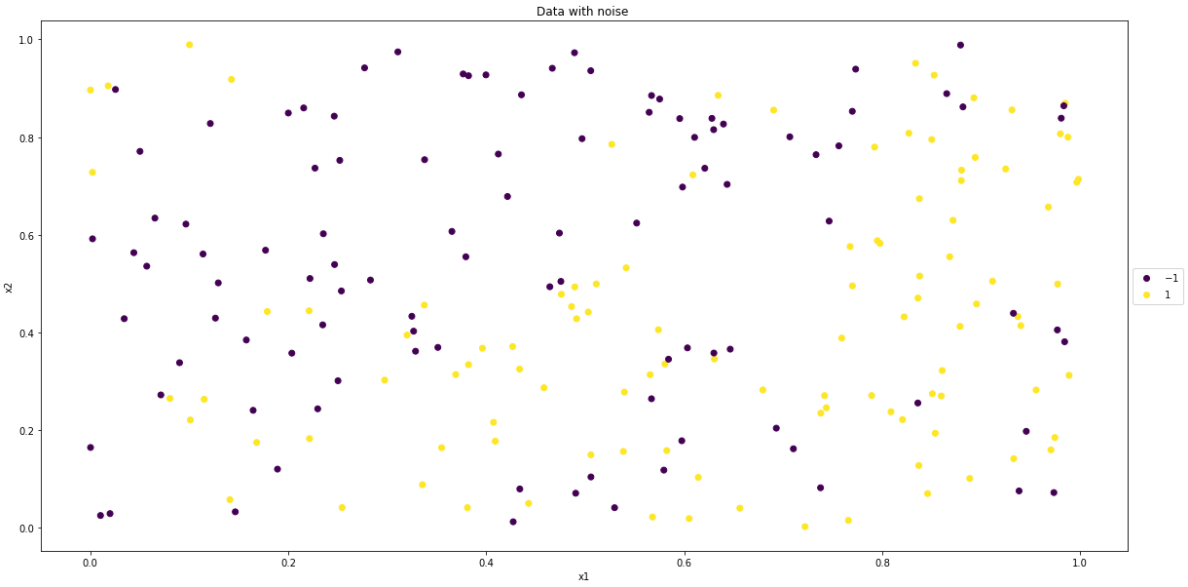
In [16]:
```
"""
Function that flips labels given a flip vector
@param y, np array, labels to flip (don't forget to copy the data in order not to a
@param flip_vector, np array, array that should be used to flip the labels

@output flipped_labels, np array, the labels where 1//6 labels are flipped
"""
```

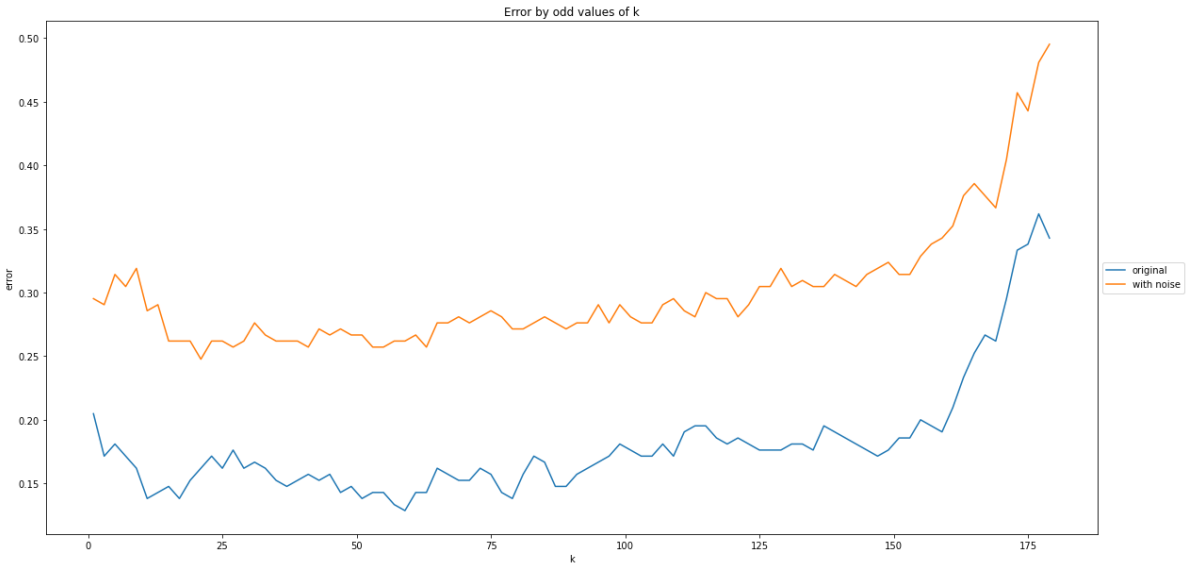
```
def flip_labels(y,flip_vector):
    #your code goes here ↓↓↓

    flipped_labels =np.where(flip_vector==-1,-y,y)


    return flipped_labels
```

In [17]:
```
#define new y vector by calling flip function
fl_vec = generate_flip_vector(len(y))
y_fl = flip_labels(y,fl_vec)

nf = 10
m = 179

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 3.3 (10 points)

In [18]:
```
#your plotting code goes here ↓↓↓

# plot the data
plt.rcParams['figure.figsize'] = (20, 10)
scatter = plt.scatter(X[:, 0],X[:, 1], c=y_fl)
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Data with noise ")
plt.legend(handles=scatter.legend_elements()[0], labels =scatter.legend_elements()
plt.show()

#plot the error

m = 179
nf = 10
k=range(1,m+1,2)

error_holder = []
for i in range(1,m+1,2): #range with 179 included and step of 2
    error_holder.append(run_kNN(X,y,nf,i))


plt.plot(k,error_holder, label = "original")
plt.plot(k, error_holder_flipped, label = "with noise")
plt.xlabel("k")
plt.ylabel("error")
plt.title("Error by odd values of k")
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()
```





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

x*) The two classes are well seperable and have nearly no overlap*
y) The two classes are now less seperable and have a larger overlap

z*) Random label flipping brings noise into the data*
a2) Random label flipping simply swaps data, but does not add noise

b2*) Overall, the error remains nearly unchanged compared to the original data set.*
c2) 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 [19]:
```
#your answers go here ↓↓↓
x_ = False
y_ = False

z_ = True
a2_ = False
```

```
b2_ = False
c2_ = True
```

Task 4: 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 $f = 2, 3, 4, 5, 6$ the number features. Thus, the first data point (where $f = 2$) shows the error for the original feature matrix X without extra dimensions.

Code 4.1 (10 points):

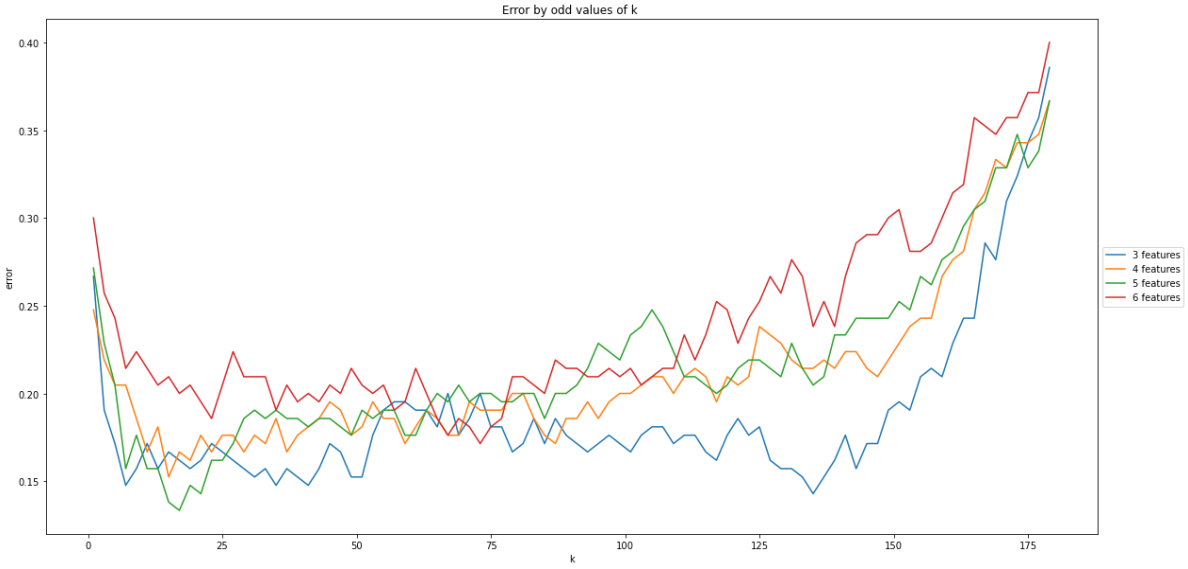
In [20]:
```
"""
Function that adds random features to dataset
@param X, np array, dataset

@output X_new, np array, dataset enhanced with 4 random features
"""
def add_features(X):
    np.random.seed(1234)
    #your code goes here ↓↓↓

    features = [np.random.uniform(size=len(y)) for i in range(4)]
    features = np.swapaxes(features,0,1)
    X_new =np.c_[ X,features]

    return X_new
```

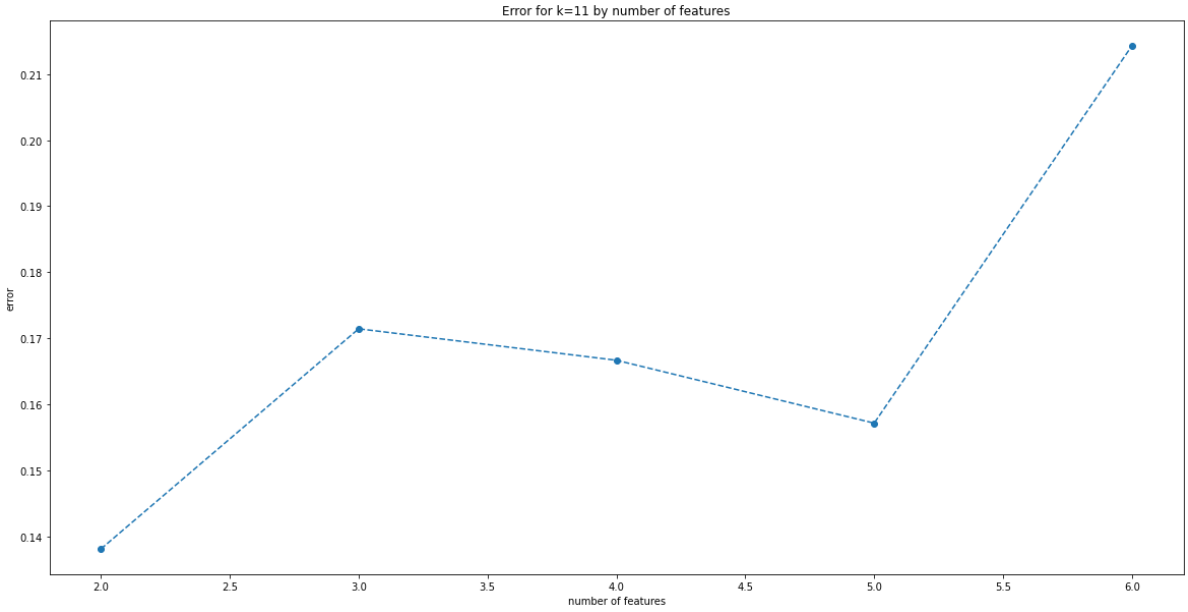
In [21]:
```
#define new feature matrix by calling add_features function
X_new = add_features(X)
```

Code & Question 4.2 (15 points):

In [22]:
```
#your code goes here ↓↓↓

nf = 10
m = 179
k=range(1,m+1,2)

fig = plt.figure()
ax = fig.add_subplot()

for i in range(3,7):
    error_holder_feat =[]
    for k_increased in range(1,m+1,2): #range with 179 included and step of 2
        error_holder_feat.append(run_kNN(X_new[:,:i],y,nf,k_increased))
    ax.plot(k, error_holder_feat, label = f"{i} features")
ax.set_xlabel("k")
ax.set_ylabel("error")
```

```
ax.set_title("Error by odd values of k ")
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()

fig2 = plt.figure()
ax2 = fig2.add_subplot()

error_holder_feat =[]
for i in range(2,7):
    error_holder_feat.append(run_kNN(X_new[:,:i],y,nf,11))

ax2.scatter(range(2,7), error_holder_feat)
ax2.plot(range(2,7), error_holder_feat, '--')
ax2.set_xlabel("number of features")
ax2.set_ylabel("error")
ax2.set_title("Error for k=11 by number of features")
plt.show()
```





Try to explain possible changes of the error.

d2) *k-nearest neighbors is robust against randomly added further features; noise is filtered out*
e2) k-nearest neighbors is not robust aginst randomly added further features; the error increases if extra dimension(s) with noise are added

f2*) Error seems to increase with growing number of features*

g2) Error seems to to decrease with growing number of features

h2_) Error seems to stay the same with any number of features

In [23]:
```
#your answers go here ↓↓↓
d2_ = False
e2_ = True

f2_ = True
g2_ = False
h2_ = False
```