Imports

we import all requirements

In [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.base import BaseEstimator, ClassifierMixin
from scipy import stats
from sklearn import neighbors
from sklearn import datasets
from sklearn.metrics import pairwise_distances
from sklearn.metrics import confusion_matrix
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA


from knn_source import (rand_gauss, rand_bi_gauss, rand_tri_gauss,
                        rand_checkers, rand_clown, plot_2d, ErrorCurve,
                        frontiere_new, LOOCurve)


import seaborn as sns
from matplotlib import rc

plt.close('all')
rc('font', **{'family': 'sans-serif', 'sans-serif': ['Computer Modern Roman']})
params = {'axes.labelsize': 12,
          'font.size': 16,
          'legend.fontsize': 16,
          'text.usetex': False,
          'figure.figsize': (8, 6)}
plt.rcParams.update(params)

sns.set_context("poster")
sns.set_palette("colorblind")
sns.set_style("white")
_ = sns.axes_style()
```

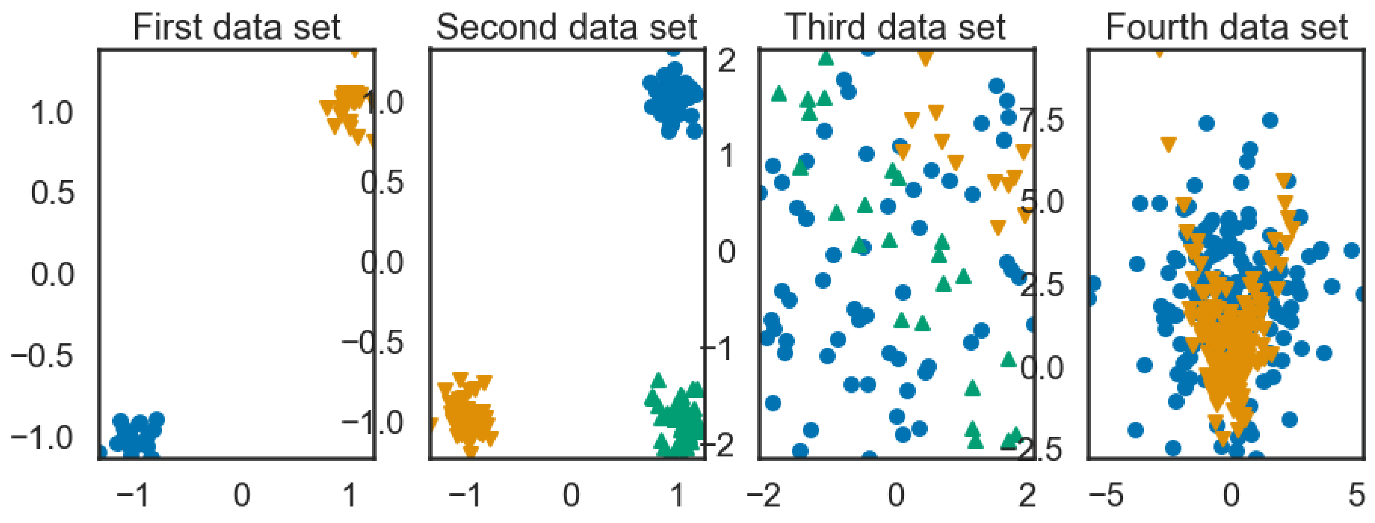
In the following we study the functions rand_bi_gauss, rand_tri_gauss, rand_clown and rand_checkers defined in knn_source.py. What do they yield ? Describing the distributions of the datasets. Which one are easy to classify ? Generate 4 datasets with these functions, and use plot_2d to plot them.

In [ ]:
```python
np.random.seed(42)  # fix seed globally

n = 100
# infer the parameters and choose their values
mu, sigma = [1, 1], [0.1, 0.1]
rand_gauss(n, mu, sigma)

n1_1 = 20
n2_1 = 20
#! for four functions
X1, y1 = rand_bi_gauss(n1_1, n2_1)

n1_2 = 50
n2_2 = 50
n3_2 = 50
X2, y2 = rand_tri_gauss(n1_2, n2_2, n3_2)

n1_3 = 50
n2_3 = 50

X3, y3 = rand_checkers(n1_3, n2_3, 0.1)
n1_4 = 150
```

```
n2_4 = 150
X4, y4 = rand_clown(n1_4, n2_4)
```

In [ ]:

```
############################################################################
#     Displaying labeled data
############################################################################

plt.show()
plt.close("all")
plt.ion()
plt.figure(1, figsize=(15, 5))
plt.subplot(141)
plt.title('First data set')
plot_2d(X1, y1)

plt.subplot(142)
plt.title('Second data set')
plot_2d(X2, y2)

plt.subplot(143)
plt.title('Third data set')
plot_2d(X3, y3)

plt.subplot(144)
plt.title('Fourth data set')
plot_2d(X4, y4)
```



The First data set gives two gaussian distributions with given number of samples for each distribution and given params as mu and sigma

---

The Second data set gives three gaussian distributions with given number of samples for each distribution and given params as mu and sigma

---

The Third data set gives random noisy sample with a shape of $(int(n1/8) + int(n2/8)) * 8$ with given n1 and n2 and mus and sigmas

---

The Fourth data set gives two distributions with given number of samples for each distribution and given params as mu and sigma,
but the first distribution coordinates are equal to
$(FirstNormalDistributionN1,$
$FirstNormalDistributionN1 * FirstNormalDistributionN1 + sigma1 * SecondNormalDistributio$
The second coordinates are equal to
$(sigma2 * FirstNormalDistributionN2, sigma2 * SecondNormalDistributionN2 + 2\sigma)$

As an adaptation of this algorithm to regression:

The predicted value could be the mean of nearest neighbors value (for example, if k =11 so the value will be $(valuesOfElevenNearestNeighbors)/11$)

The k-NN algorithm

reimplement the decision function of KNeighborsClassifier of scikit-learn with another approach

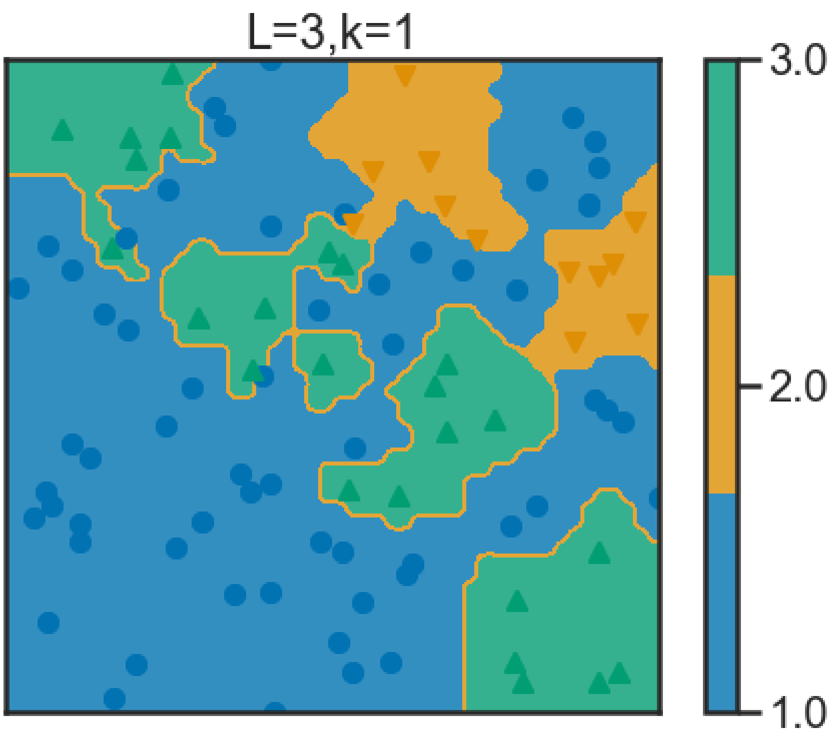
In [ ]:

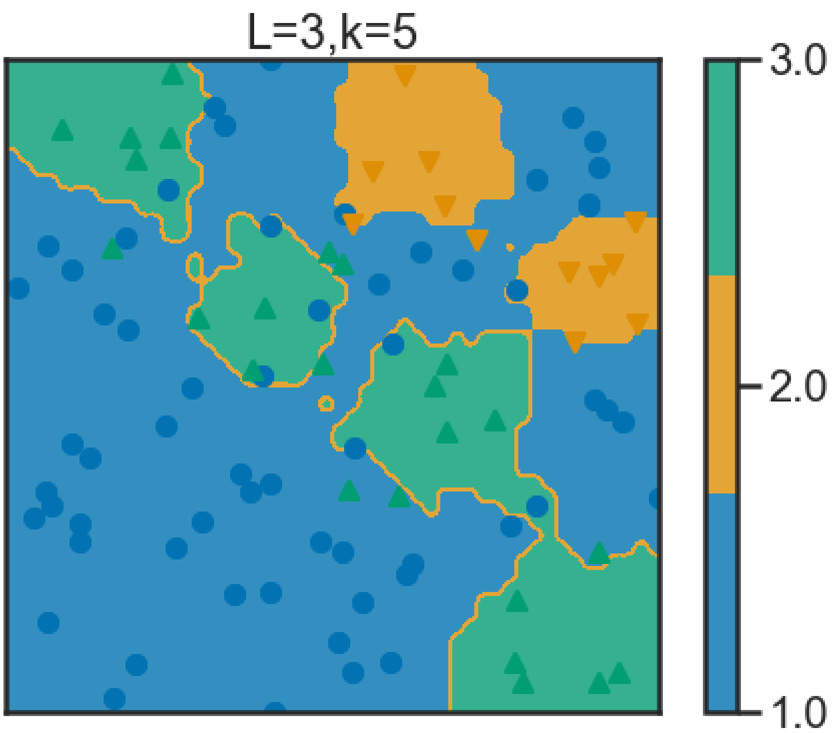
```
# Write your own implementation
class KNNClassifier(BaseEstimator, ClassifierMixin):
    """  This classifier should be competitive to scikitlearn classifier with KNN  """
    # ! we initiate our class with number of neighbors and the weights if needed
    def __init__(self, n_neighbors=1, weights=None):
        self.n_neighbors = n_neighbors
        self.weights = weights
    # ! calling .fit helps to store training data in our class self
    def fit(self, X, y):
        self.X_ = X
        self.y_ = y
        return self
    # ! calling .predict helps to compute the distances between independent variables of the tr
    def predict(self, X):
        n_samples, n_features = X.shape
        # ! : Compute all pairwise distances between X (the new data) and self.X_(independent v
        d = pairwise_distances(X, self.X_)

        # ! : Get indices to sort them
        sorted_indices = np.argsort(d)
        # ! : take the first n_neighbors and sort d
        d = np.take_along_axis(d, sorted_indices, axis=1)[:, :self.n_neighbors]
        # ! Get indices of neighbors
        neighbors_indices = sorted_indices[:, :self.n_neighbors]

        # !: Get labels of neighbors
        Y_neighbors = self.y_[neighbors_indices]

        # ! : Find the predicted labels y for each entry in X
        # we can use the scipy.stats.mode function but it's slower than the following method
        # y_pred,_ = stats.mode(Y_neighbors,axis=1) # this is the scipy function
        # ? we consider an empty array
        max_ = np.array([])
        for i in np.unique(Y_neighbors):
            N_repetitions = np.sum(
                (Y_neighbors == i) * (self.weights(d) if self.weights else 1), axis=1)
            max_ = np.append(max_, N_repetitions)
        y_pred = np.unique(Y_neighbors)[np.argmax(
            max_.reshape(-1, Y_neighbors.shape[0]).T, axis=1)] if max_.any() else np.array([])
        return y_pred.ravel()
```

k = 1 , k = n cases plots these cases on one dataset and their interpretation.

In [ ]:

```
n1 = n2 = 200
sigma = 0.1
X, y = X3, y3

def KnnWithFrontiere(X, y, n_neighbors=1):
 # the k in k-NN
    knn = neighbors.KNeighborsClassifier(n_neighbors=n_neighbors)
    knn.fit(X, y)
    plt.figure()
    plot_2d(X, y)
    n_labels = 3
    frontiere_new(knn, X, y, w=None, step=50, alpha_choice=1,
                            n_labels=n_labels, n_neighbors=n_neighbors)


KnnWithFrontiere(X, y, 1)
KnnWithFrontiere(X, y, 5)
```

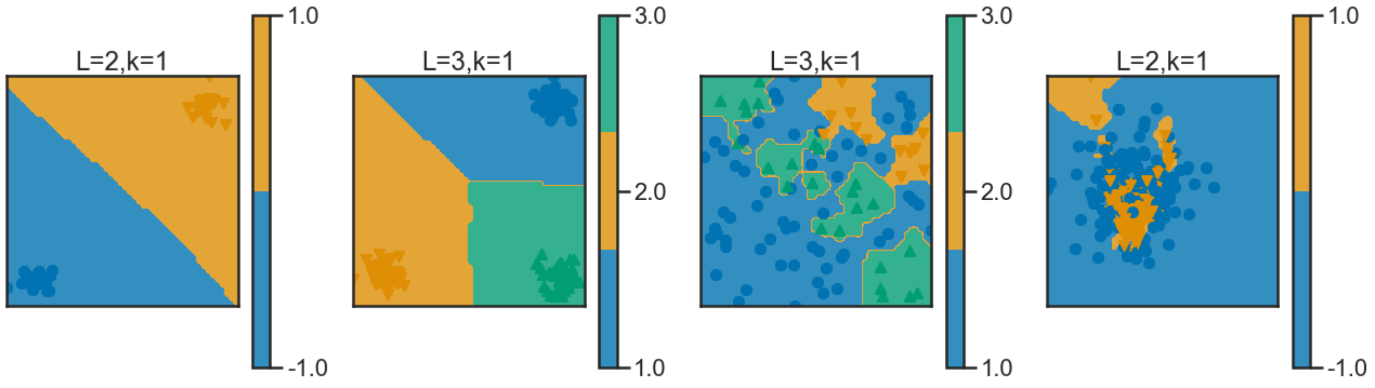




The frontier is very complex when k = 1 because the classifier is very sensitive to the noise present in the data, so absolutely is an overfitting. On the contrary, when k = n, just the most present label is predicted, so it's an underfitting. so the tunning gives the best parameter k

In [ ]:

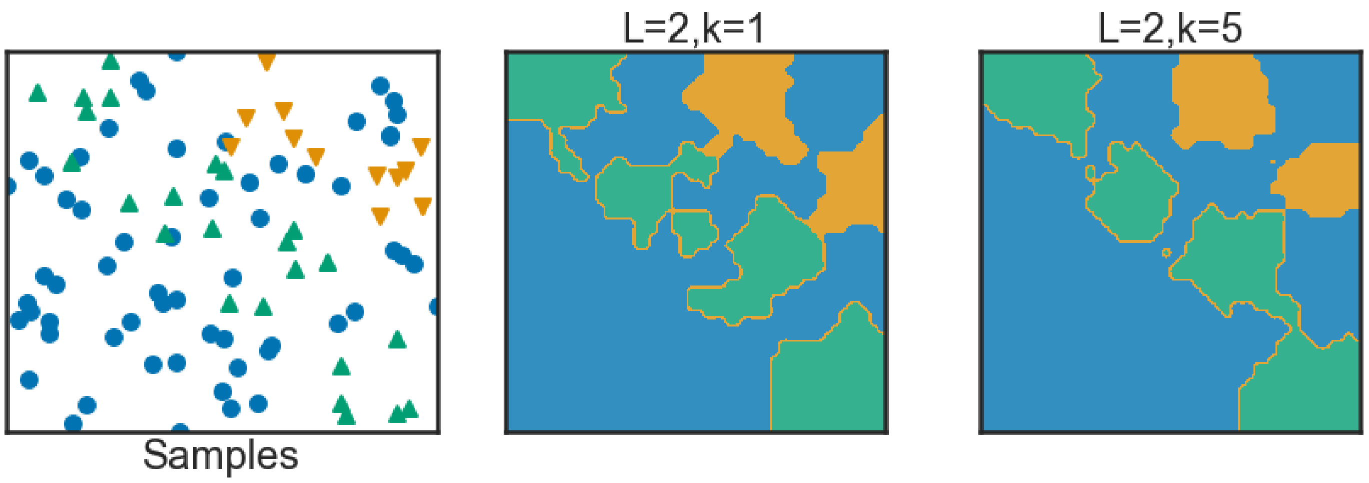
```
# test now for all datasets

n_neighbors = 1  # the k in k-NN
knn = neighbors.KNeighborsClassifier(n_neighbors=n_neighbors)
plt.figure(112, figsize=(22, 6))
i=1
# ! we run a loop to try k=1 for all datasets:
for (X, y, n) in [(X1, y1, 2), (X2, y2, 3), (X3, y3, 3), (X4, y4, 2)]:

    knn.fit(X, y)
    plt.subplot(1,4,i)
    i=i+1
    plot_2d(X, y)
    n_labels = n
    frontiere_new(knn, X, y, w=None, step=50, alpha_choice=1,
                  n_labels=n_labels, n_neighbors=n_neighbors)
```



we notice that when the groups on training are well separated, the predicted clusters are good enough but when it gets merged, our cluster doesn't perform well but it's overfitted

In [ ]:

```
# ! we take the noisy sample and we display the predictions when varying the value of k

X,y =X3,y3
plt.figure(3, figsize=(12, 8))
plt.subplot(2,3,1)
plot_2d(X,y)
plt.xlabel('Samples')
ax = plt.gca()
ax.get_yaxis().set_ticks([])
ax.get_xaxis().set_ticks([])

for n_neighbors,i in zip((1,5),(2,3)):
    # ! we plot the sample and then clusters predicted with k=1 then with k =5
    plt.subplot(2,3,i)
    knn = neighbors.KNeighborsClassifier(n_neighbors=n_neighbors)
    knn.fit(X,y)

    frontiere_new(knn, X, y, w=None, step=50, alpha_choice=1,n_labels=2,
                  colorbar=False, samples=False,n_neighbors=n_neighbors)
    plt.draw()  # update plot

plt.tight_layout()
```



In [ ]:

```
# ! we show here the score of the model with 1 neighbor on both training and testing data
n_neighbors = 1
n1 = n2 = 200
sigma = 0.1
X_train = X3[::2]
```

```
Y_train = y3[::2].astype(int)
X_test = X3[1::2]
Y_test = y3[1::2].astype(int)
knn = neighbors.KNeighborsClassifier(n_neighbors=n_neighbors)
knn.fit(X_train,Y_train)
print(f"the score on training is :{knn.score(X_train,Y_train):.2f}")
print(f"the score on testing is :{knn.score(X_test,Y_test):.2f}")
```

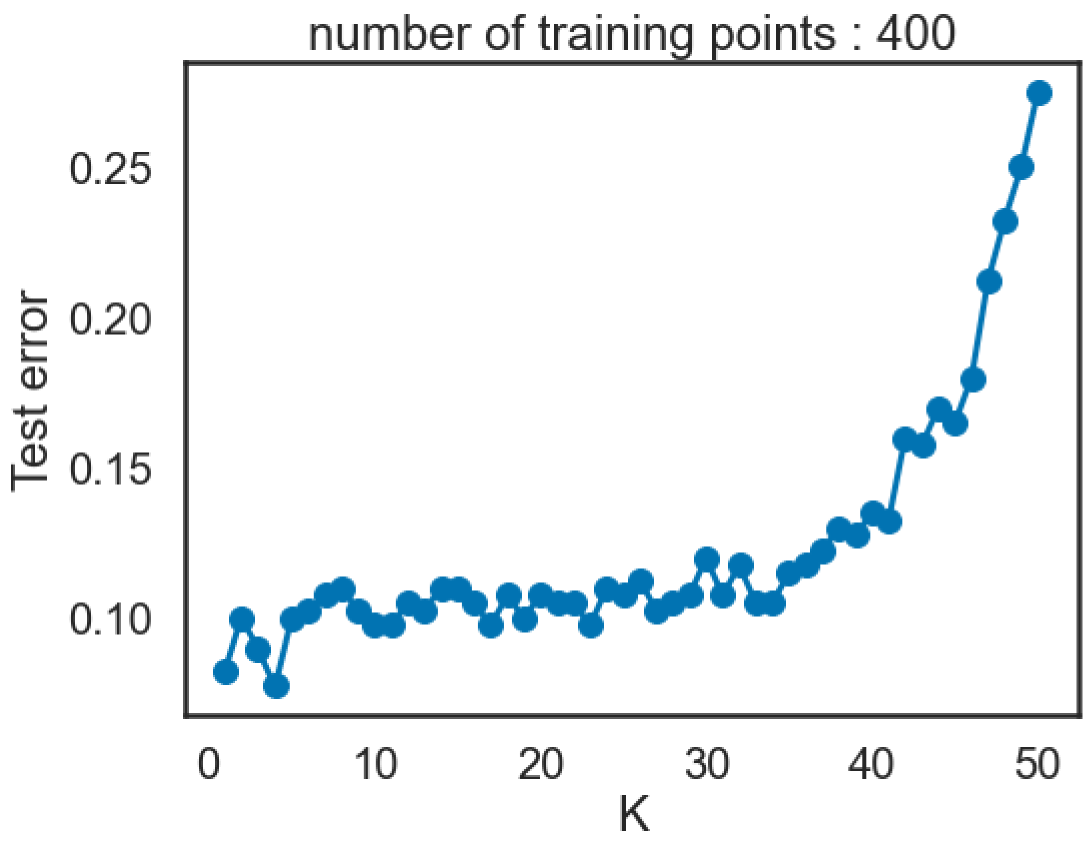
```
the score on training is :1.00
the score on testing is :0.79
```

it's obvious that the model overfit , since the score on training is 100% while on testing is much less

we evaluate the test error while changing the number of neighbors

In [ ]:

```
n1 = n2 = 200
sigma = 0.1
data4_X, data4_y = rand_checkers(2 * n1, 2 * n2, sigma) # generate data
#split data to test and train
X_train = data4_X[::2]
Y_train = data4_y[::2].astype(int)
X_test = data4_X[1::2]
Y_test = data4_y[1::2].astype(int)


# ! instantiate ErrorCurve with k_range=range(1, 51)
error_curve = ErrorCurve(k_range=range(1, 51))
# ! fit it, plot it
error_curve.fit_curve(X_train, Y_train, X_test, Y_test)
error_curve.plot()
```



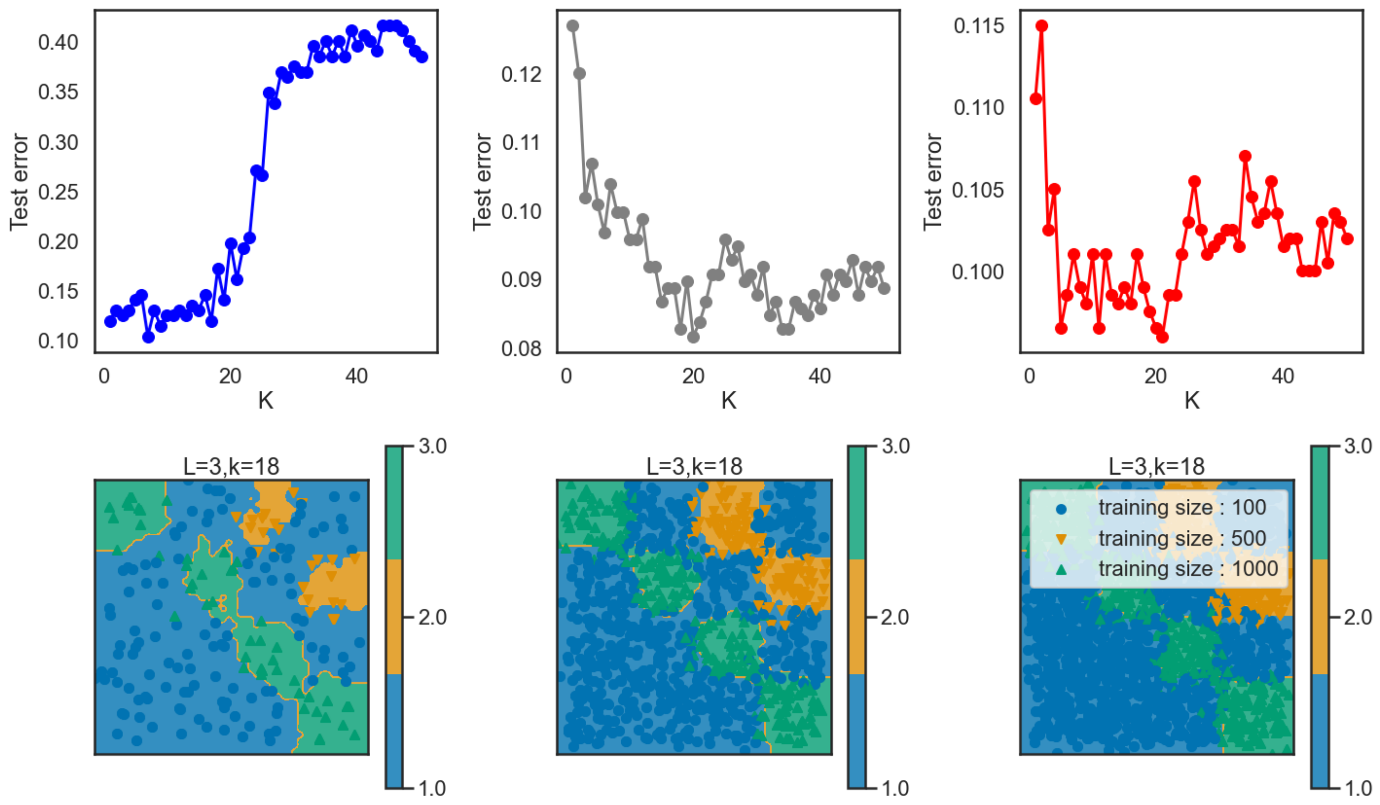
we evaluate the impact of the amount of samples on the error

In [ ]:

```
collist = ['blue', 'grey', 'red', 'purple', 'orange', 'salmon', 'black',
           'fuchsia']

sigma = 0.1

range_n_samples = [100, 500, 1000] # number of simples for each dataset
niter = len(range_n_samples)
```

```
plt.figure(5, figsize=(20, 12))
for n in range(niter):
    plt.subplot(2,3,n+1)
    n1 = n2 = range_n_samples[n]
    X_train, Y_train = rand_checkers(n1, n2, sigma) # training dataset
    X_test, Y_test = rand_checkers(n1, n2, sigma) #testing dataset , noisy with same norme and
    # ! fit and plot with color varying from collist
    error_curve.fit_curve(X_train, Y_train, X_test, Y_test)
    error_curve.plot(color=collist[n % len(collist)], maketitle=False,)

    plt.subplot(2,3,n+4)
    plot_2d(X_train, Y_train)
    # fit also the classifiers for 18 neighbors
    n_neighbors = 18
    knn = neighbors.KNeighborsClassifier(n_neighbors=n_neighbors)
    knn.fit(X_train, Y_train)

    frontiere_new(knn, X_train, Y_train, w=None,
                  step=50, alpha_choice=1, n_labels=3, n_neighbors=n_neighbors)
plt.tight_layout()
plt.legend(["training size : %d" % n for n in range_n_samples],
        loc='upper left')
```

```
Out[ ]: <matplotlib.legend.Legend at 0x20e79881f70>
```



it becomes obvious how the stable best k_neighbors depends on the dataset size

Pros :

- easy to interpret
- does not have many parameters

Cons :

- take too much time to compute, especially in high dimension or with large dataset
- choosing the best k param depends on the training set size and the nature of the distribution

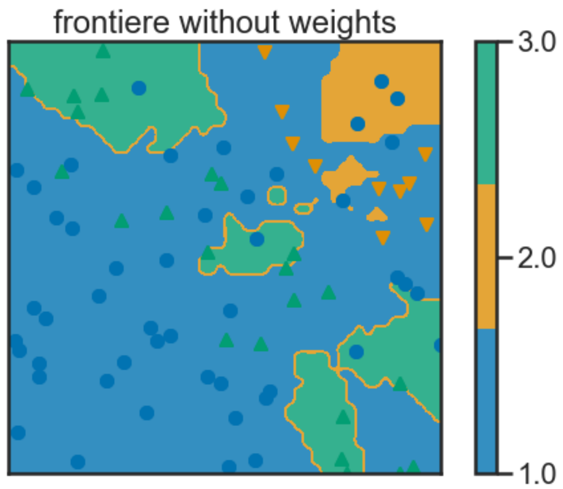
conclusion

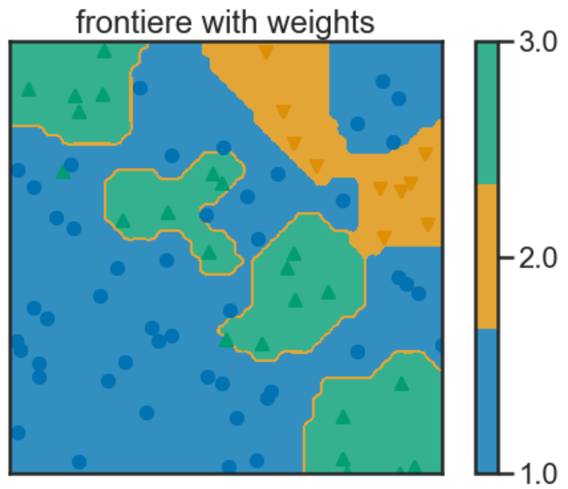
In breaf, Certainly, the k-nearest neighbor method is computationally expensive since it requires computing distances for all points in the training set for each label to be predicted. Moreover, in high dimension, the distances between points tend to grow and the notion of neighbor is not very reasonable anymore. This is called the 'curse of dimensionality'. However, this method has the advantage of being very intuitive and easy to understand to start with the idea of supervised classification. It is therefore very easy to interpret.

Implement weights for the kNN classifier

In [ ]:
```
def weights(dist):
    """Returns an array of weights, exponentially decreasing in the square
    of the distance.

    Parameters
    ----------
    dist : a one-dimensional array of distances.

    Returns
    -------
    weight : array of the same size as dist
    """
    # !: use weights equal to exp(- dist^2 / 0.1)
    return np.exp(- dist ** 2 / 0.1)
    # return dist


n_neighbors = 19

# Focus on dataset 2
index_train_test = int(4 * (n1_3 + n2_3) / 5)
X_train = X3[:index_train_test]
Y_train = y3[:index_train_test]
X_test = X3[index_train_test:]
Y_test = y3[index_train_test:]

# we train and predict with weights
wknn_w = neighbors.KNeighborsClassifier(n_neighbors=n_neighbors, weights=weights)
wknn_w.fit(X_train, Y_train)
Y_pred_w = wknn_w.predict(X_test)

# we train and predict without weights
wknn = neighbors.KNeighborsClassifier(n_neighbors=n_neighbors)
wknn.fit(X_train, Y_train)
Y_pred = wknn.predict(X_test)

# we plot frontiere without weights
plt.figure(4, figsize=(20, 6))
plt.subplot(1,2,1)
plot_2d(X_train, Y_train)
frontiere_new(wknn, X_train, Y_train, w=None, step=50,
              alpha_choice=1, n_neighbors=n_neighbors)
plt.title("frontiere without weights")

# we plot frontiere with weights
plt.subplot(1,2,2)
plot_2d(X_train, Y_train)
frontiere_new(wknn_w, X_train, Y_train, w=None, step=50,
              alpha_choice=1, n_neighbors=n_neighbors)
plt.title("frontiere with weights")

print(f"The accuracy with weights is {(Y_pred_w==Y_test).sum() /len(Y_test):.2f} , while withou
```

```
The accuracy with weights is 0.69 , while without weights is 0.50
```





we notice that the accuracy of the model is better with weight, and also the frontier are more stable

we compare our model predictions with scikitlearn's predictions

In [ ]:
```
knn = KNNClassifier(n_neighbors=n_neighbors, weights=weights)
knn.fit(X_train, Y_train)
Y_pred = knn.predict(X_test)

sknn = neighbors.KNeighborsClassifier(n_neighbors=n_neighbors, weights=weights)
sknn.fit(X_train, Y_train)
Y_pred_skl = sknn.predict(X_test)

print(f"The matching : {(Y_pred==Y_pred_skl).all()}\n")

print(
    f"The accuracy value our model is {(Y_pred==Y_test).sum()/len(Y_test):.2f}")
print(
    f"The accuracy value sklearn is {(Y_test==Y_pred_skl).sum()/len(Y_test):.2f}")
print(
    f"The accuracy value of matching our model with scikit learn is {(Y_pred==Y_pred_skl).sum()
```

```
The matching : True

The accuracy value our model is 0.69
The accuracy value sklearn is 0.69
The accuracy value of matching our model with scikit learn is 1.00
```

we compare our model performance with scikit learn performance and speed

our model speed

In [ ]:
```
%%timeit
knn = KNNClassifier(n_neighbors=n_neighbors, weights=weights)
knn.fit(X_train, Y_train)
Y_pred = knn.predict(X_test)
```

```
431 µs ± 34.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

scikit learn speed

In [ ]:
```
%%timeit
sknn = neighbors.KNeighborsClassifier(n_neighbors=n_neighbors, weights=weights)
sknn.fit(X_train, Y_train)
Y_pred_skl = sknn.predict(X_test)
```

```
1.08 ms ± 63.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
In [ ]: print(f"The matching : {(Y_pred==Y_pred_skl).all()}\n")

        print(
            f"The accuracy value our model is {(Y_pred==Y_test).sum()/len(Y_test):.2f}")
        print(
            f"The accuracy value sklearn is {(Y_test==Y_pred_skl).sum()/len(Y_test):.2f}")
        print(
            f"The accuracy value of matching our model with scikit learn is {(Y_pred==Y_pred_skl).sum()
```

```
The matching : True

The accuracy value our model is 0.69
The accuracy value sklearn is 0.69
The accuracy value of matching our model with scikit learn is 1.00
```

we notice that our model is simple and it outperform scikit learn , since it provides the same accuracy with faster computations