Mieux comprendre le fonctionnement d'un classifieur

Comme nous l'avons vu dans les premières classifications, le comportement des classifieurs peut être très différent. Nous présentons à présent les régions de décisions dans lesquelles le classifieur recherche les valeurs pour pouvoir prédire. L'objectif est donc ici de mieux comprendre le fonctionnement d'un classifieur, les raisons d'une bonne ou mauvaise classification et avoir une idée de l'impact des hyperparamètres.

Dans un problème de classification à deux classes, une région de décision ou surface de décision est une hypersurface qui partitionne l'espace vectoriel sous-jacent en deux ensembles : un pour chaque classe. Le classificateur classera tous les points d'un côté de la limite de décision comme appartenant à une classe et tous ceux de l'autre côté comme appartenant à l'autre classe.

Les illustrations sont faites à partir du jeu de données IRIS et nous retenons celui disponible dans scikit learn et qui a été introduit à la fin du notebook Ingénierie des données.

In [1]:

```
#Sickit learn met régulièrement à jour des versions et
#indique des futurs warnings.
#ces deux lignes permettent de ne pas les afficher.
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
# la ligne ici est ajouté principalement pour SVC dont des mises à jour
# sont annoncées mais jamais mise à jour :)
```

In [2]:

```
from sklearn import datasets
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.svm import LinearSVC

import matplotlib.pyplot as plt
from mlxtend.plotting import plot_decision_regions
import matplotlib.gridspec as gridspec
import itertools

```

Pour pouvoir afficher l'espace de décision, le plus simple est de se mettre dans un espace en 2 dimensions. Par la suite nous ne considérerons que les 2 attributs *sepal length* et *sepal width* dans notre jeu de données.

In [3]:

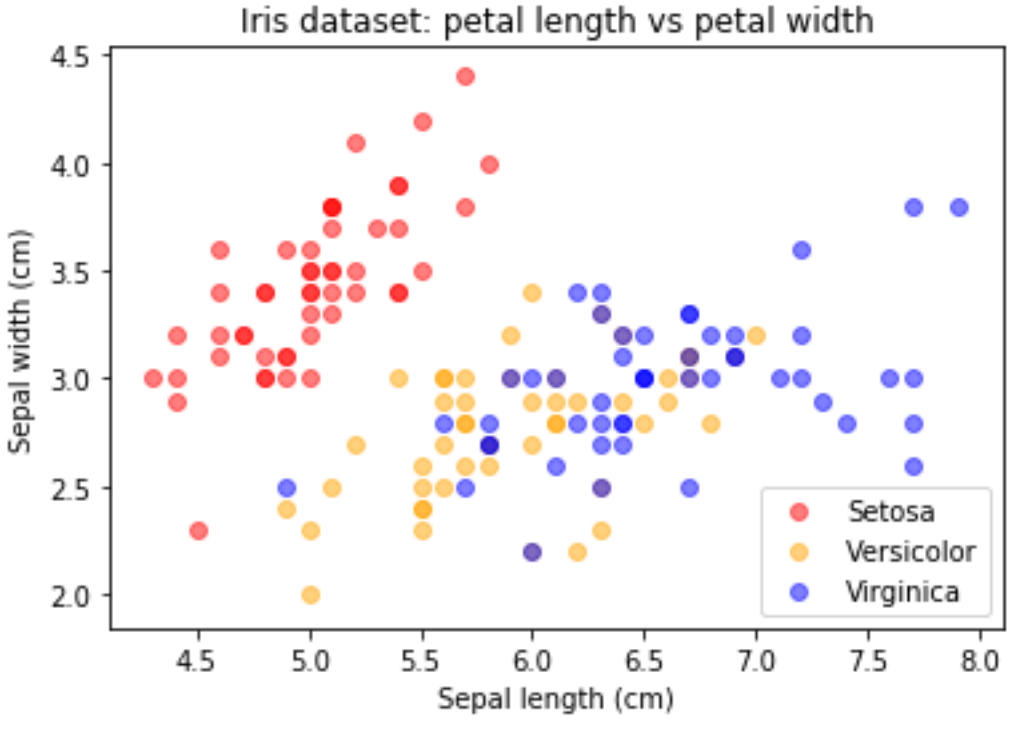
```
print ('Lecture du fichier iris\n')
iris = datasets.load_iris()
#sélection des deux attributs sepals
X = iris.data[:, :2]
y = iris.target
```

```
Lecture du fichier iris
```

Affichage des valeurs des attributs afin de voir comment elles se répartissent.

In [4]:

```
#Passage par un dataframe par soucis de simplification
iris_df = pd.DataFrame(iris['data'], columns=iris['feature_names'])
iris_df['species'] = iris['target']

colours = ['red', 'orange', 'blue']
species = ['Setosa', 'Versicolor', 'Virginica']

for i in range(0, 3):
    species_df = iris_df[iris_df['species'] == i]
    plt.scatter(
        species_df['sepal length (cm)'],
        species_df['sepal width (cm)'],
        cmap=plt.cm.coolwarm,
        color=colours[i],
        alpha=0.5,
        label=species[i]
    )

plt.xlabel('Sepal length (cm)')
plt.ylabel('Sepal width (cm)')
plt.title('Iris dataset: petal length vs petal width')
plt.legend(loc='lower right')

plt.show()
```



Comme nous pouvons le constater Setosa est très séparée des autres classes et doit pouvoir facilement être séparé. Il est évident que la séparation entre Versicolor et Virginica va être plus difficile pour un classifieur.

Création d'un jeu de données d'apprentissage et de test.

In [5]:

```
validation_size=0.3 #30% du jeu de données pour le test

testsize= 1-validation_size
seed=30
X_train,X_test,y_train,y_test=train_test_split(X, y,
                                               train_size=validation_size,
                                               random_state=seed,
                                               test_size=testsize)
```

Test de 4 classifieurs différents pour voir comment ils se comportent.

In [6]:

```
lr=LogisticRegression(random_state=1,
                      solver='newton-cg',
                      multi_class='multinomial')

gnb = GaussianNB()
deci= DecisionTreeClassifier(random_state=1)
svm = SVC(gamma='auto')
```

Pour afficher les régions de décisions, nous utilisons la librairie mlxtend (cf notebook règles d'association) qui offre de nombreuses facilités pour afficher la zone.

Le principe consiste à parcourir la liste des classifieurs, de faire le fit, de faire la prédiction et d'afficher la zone de décision. Cette zone correspond aux différentes valeurs dans lesquelles pour une classe, le clasifieur va chercher ses valeurs. La zone est définie par rapport à l'ensemble des données. La fonction plot_decision_regions va récupérer les valeurs min et max de tous les attributs et ensuite en fonction de la classe va plutôt étendre ou modifier la zone.

Ci-dessous une fonction qui appelle plot_decision_regions.

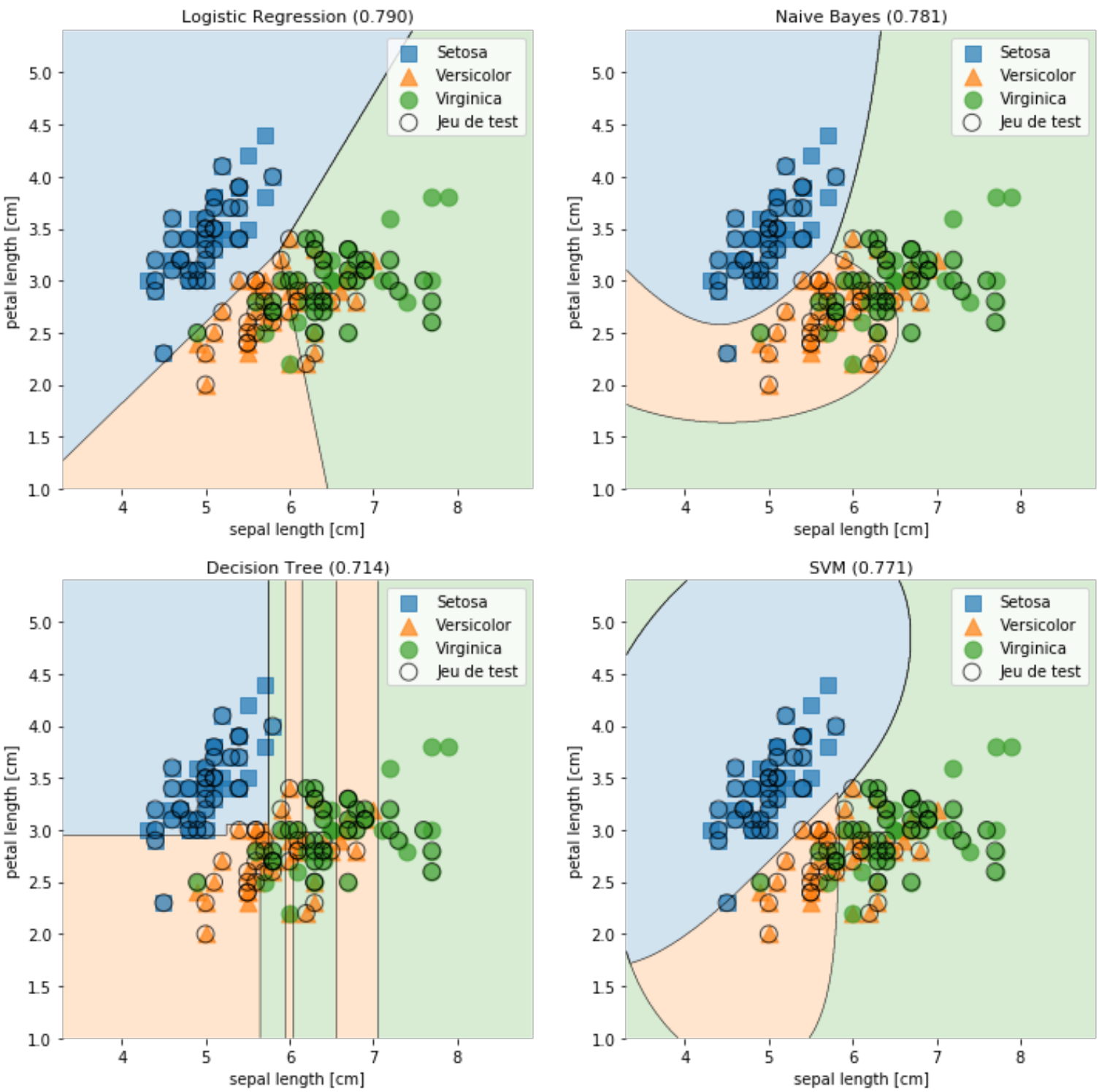
In [7]:

```
def call_decision_regions_iris (labels,list_clf,X_train,y_train,X_test,y_te
    # pour afficher les points du jeu de test plus clair
    scatter_kwargs = {'s': 120, 'edgecolor': None, 'alpha': 0.7}
    contourf_kwargs = {'alpha': 0.2}
    scatter_highlight_kwargs = {'s': 120, 'label': 'Jeu de test', 'alpha':

    #pour afficher les 4 valeurs sur 2 colonnes et 2 lignes
    gs = gridspec.GridSpec(2, 2)

    fig = plt.figure(figsize=(12,12))

    for clf, label, grd in zip(list_clf,
                               labels,
                               itertools.product([0, 1], repeat=2)):
        #fit du modele
        clf.fit(X_train, y_train)

        #prediction sur le jeu de test
        result=clf.predict(X_test)
        acc=accuracy_score(result, y_test)

        #affichage de la zone de décision
        ax = plt.subplot(gs[grd[0], grd[1]])
        fig=plot_decision_regions(X, y, clf=clf, legend=2,
                              X_highlight=X_test,
                              scatter_kwargs=scatter_kwargs,
                              contourf_kwargs=contourf_kwargs,
                              scatter_highlight_kwargs=scatter_highlight_kwargs)



        L = plt.legend()
        L.get_texts()[0].set_text('Setosa')
        L.get_texts()[1].set_text('Versicolor')
        L.get_texts()[2].set_text('Virginica')
        accu='%0.3f'%acc
        plt.xlabel('sepal length [cm]')
        plt.ylabel('petal length [cm]')
        label=label+ " ("+accu+')'
        plt.title(label, size=11)
    plt.show()

```

Vous pouvez constater qu'en fonction de la stratégie de l'algorithme les zones sont très différentes.

In [8]:

```
labels = ['Logistic Regression',
          'Naive Bayes',
          'Decision Tree',
          'SVM']
list_clf=[lr,gnb,deci,svm]

call_decision_regions_iris (labels,list_clf,X_train,y_train,X_test,y_test)
```



Nous allons, à présent, tester le même classifieur mais des hyperparamètres différents. L'objectif ici est de montrer l'importance de ces choix dans un classifieur.

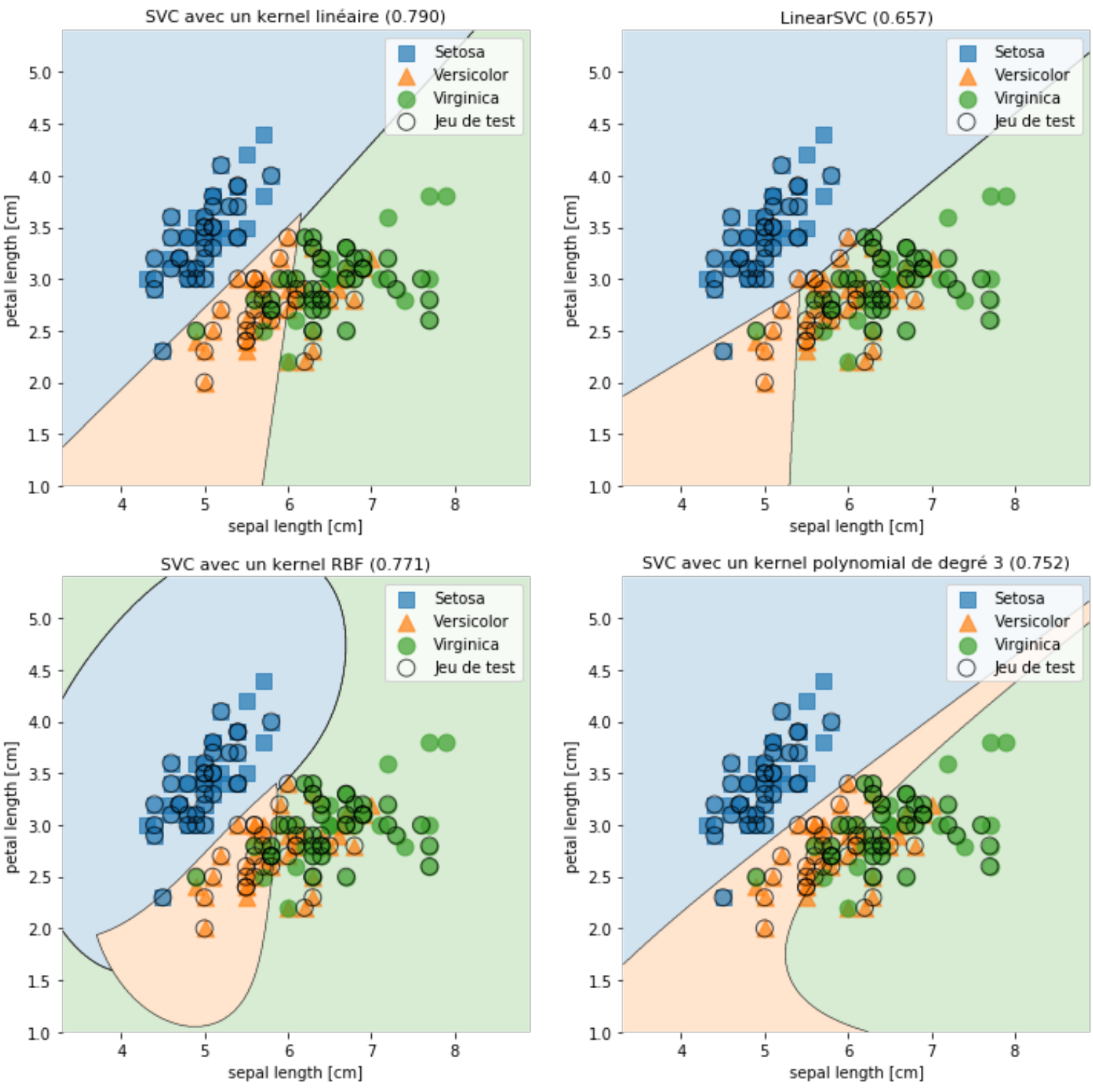
Dans l'exemple nous prenons le classifieur SVM. Tout d'abord avec un kernel (noyau) linéaire. Dans ce cas, la séparation entre les différentes classes se fait à l'aide de droites. Le second est LinearSVC. Il est un peu similaire au précédent mais l'implémentation est différente notamment sur la prise en compte du choix des pénalités et des fonctions de pertes. Il se comporte mieux que le précédent dans le cas de plus gros volumes de données. Le troisième est SVM avec un noyeau de type 'rbf'. Il s'agit d'une Radial Basis Function qui prend comme paramètre gamma (le paramètre qui permet de spécifier la région de décision) et C (la pénalité pour les données mal classées. Si C est petit le classifieur est ok pour les points mal classés). Enfin le dernier considére un polynome de degré trois.

In [9]:

```
C = 1.0  # valeur de pénalité
svc = SVC(kernel='linear', C=C)
# LinearSVC (linear kernel)
lin_svc = LinearSVC(max_iter=2000,C=C)
# SVC avec noyau RBF
rbf_svc = SVC(kernel='rbf', gamma=0.7, C=C)
# SVC avec noyeau polynomial de degre 3
poly_svc = SVC(kernel='poly', degree=3, C=C)
```

In [10]:

```
labels = ['SVC avec un kernel linéaire',
          'LinearSVC',
          'SVC avec un kernel RBF',
          'SVC avec un kernel polynomial de degré 3']
list_clf=[svc,lin_svc,rbf_svc,poly_svc]
call_decision_regions_iris (labels,list_clf,X_train,y_train,X_test,y_test)
```



Dans l'exemple suivant nous étudions différentes valeurs de gamma pour voir l'impact de SVM avec un kernel rbf.

In [11]:

```
C = 1.0

# SVC avec des valeurs différentes de gamma
rbf_svc1 = SVC(kernel='rbf', gamma=1, C=C)
rbf_svc2 = SVC(kernel='rbf', gamma=10, C=C)
rbf_svc3 = SVC(kernel='rbf', gamma=50, C=C)
rbf_svc4 = SVC(kernel='rbf', gamma=100, C=C)
```

In [12]:

```
labels = ['SVC RBF gamma=1',
          'SVC RBF gamma=10',
          'SVC RBF gamma=50',
          'SVC RBF gamma=100']
list_clf=[rbf_svc1,rbf_svc2,rbf_svc3,rbf_svc4]
call_decision_regions_iris (labels,list_clf,X_train,y_train,X_test,y_test)
```