POLITECNICO DI TORINO

Artificial Intelligence and Machine Learning

2019-2020

HOMEWORK1

Antonino Musmeci

Homework1 Artificial Intelligence and Machine Learning

November 18, 2019

We want to analyze the Wine dataset that contains the results of a chemical analysis of wines grown in a specific area of Italy. The dataset contains 178 examples with 13 features

Alcohol
Malic acid
Ash
Alcalinity of ash
Magnesium
Total phenols
Flavanoids
Nonflavanoid phenols
Proanthocyanins
Color intensity
Hue
OD280/OD315 of diluted wines
Proline

Number of instances of each wine class

Class 1 - 59
Class 2 - 71
Class 3 - 48

The goal of the analysis is build a classification model using first K-Nearest Neighbors and then support vector machine to determine from which of the three cultivators the wine has come from

[1]:
```
# Import functions and library
from myfunc import *
wine_dataset = load_wine() #load the dataset
data = pd.DataFrame(data=wine_dataset.data,columns=wine_dataset.feature_names)
data.head()
```

[1]:
```
   alcohol  malic_acid   ash  alcalinity_of_ash  magnesium  total_phenols  \
0    14.23        1.71  2.43               15.6      127.0           2.80
1    13.20        1.78  2.14               11.2      100.0           2.65
2    13.16        2.36  2.67               18.6      101.0           2.80
3    14.37        1.95  2.50               16.8      113.0           3.85
4    13.24        2.59  2.87               21.0      118.0           2.80

   flavanoids  nonflavanoid_phenols  proanthocyanins  color_intensity   hue  \
```

```
0        3.06                  0.28             2.29               5.64  1.04
1        2.76                  0.26             1.28               4.38  1.05
2        3.24                  0.30             2.81               5.68  1.03
3        3.49                  0.24             2.18               7.80  0.86
4        2.69                  0.39             1.82               4.32  1.04

   od280/od315_of_diluted_wines  proline
0                          3.92   1065.0
1                          3.40   1050.0
2                          3.17   1185.0
3                          3.45   1480.0
4                          2.93    735.0
```

Firstly we need to split into training 50%, validation 20% and test 30% sets and standardize it.

Before starting the analysis, we must standardize the data. It deals with transforming the data that was acquired in different formats to a standard format. This in general is needed to obtain best results for ML algorithms. Standardize means to scale the features such that they have zero as mean and one as variance. The standard score of a sample x is calculated as:

$$x_{std} = (x - u)/s$$

where "x" is a feature value, "u" is the mean and "s" is the variance of the input features.

We can use sklearn.preprocessing.StandardScaler to perform standardization

Then we consider only the first two features for a 2D representation of the image and discard the others

[2]:
```
# wine_dataset = load_wine() #load the dataset

X = wine_dataset.data  #features
y = wine_dataset.target #labels

#we divide the dataset into training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, ␣
 ↪random_state=42)

scaler = preprocessing.StandardScaler()
scaler.fit(X_train)


X_test = scaler.transform(X_test)
X_train = scaler.transform(X_train)

# we divide the training set into training and validation set
X_train, X_val, y_train, y_val  = train_test_split(X_train, y_train,␣
 ↪test_size=2/7, random_state=42)
```

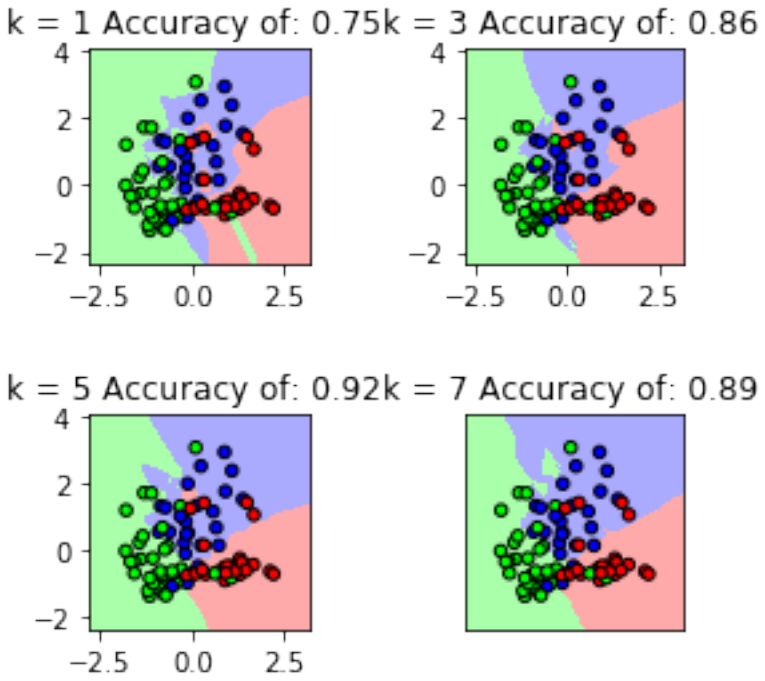
```
X_train_2D = X_train[:,:2]
X_test_2D = X_test[:,:2]
X_val_2D = X_val[:,:2]
```

K-Nearest Neighbors

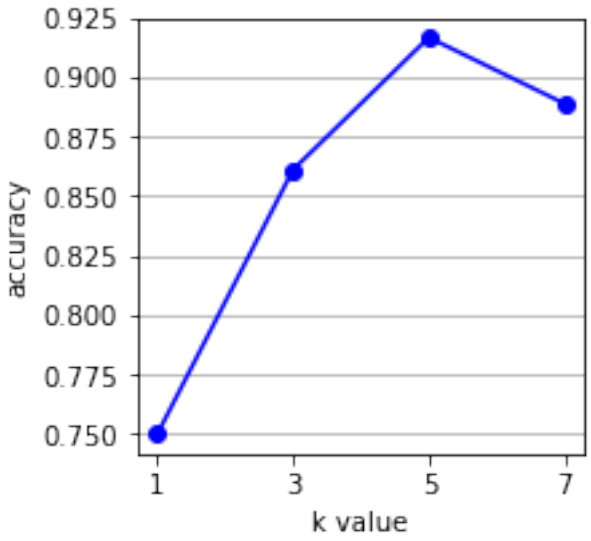
The K-Nearest Neighbors algorithm (KNN) is a non-parametric method, which considers the K closest training examples to the point of interest for predicting its class. This is done by a simple majority vote over the K closest points.

In the classification phase, k is a user-defined constant. The best choice of k depends upon the data; generally, larger values of k reduces effect of the noise on the classification, but make boundaries between classes less distinct.To choose a good k value we use the validation set to score the models fitted one the training set and we select the k value that give the best accuracy

[3]:
```
Ks=[1,3,5,7]
accuracies_knn = Apply_and_plot_Knn(Ks,X_train_2D,y_train,X_val_2D,y_val,2,2,0)
```



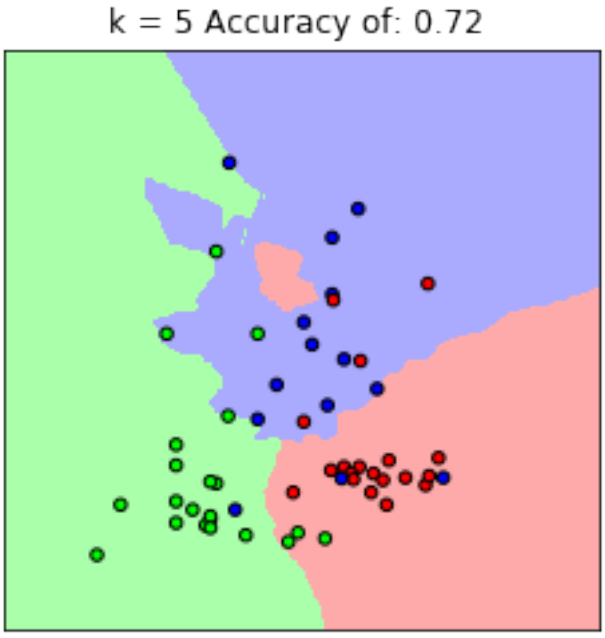
[4]:
```
plot_accuracy(Ks,accuracies_knn,'k value',0)
#select best k value
best_k = Ks[np.argmax(accuracies_knn)]
print("The value of k that gives the higher accuracy is k =", best_k)
```



The value of k that gives the higher accuracy is k = 5

[5]:
```
A = Apply_and_plot_Knn([best_k],X_train_2D,y_train,X_test_2D,y_test,1,1,1)
print("The accuracy of the model on the test set is %.2f" % A[0])
```

The accuracy of the model on the test set is 0.72



Support vector machine (SVM)

Linear SVM

A Support Vector Machine (SVM) is a discriminative classifier defined by a separating hyperplane. The aim of Support Vector classification is to find 'good' separating hyperplanes in a high dimensional feature space, where a good separation is achieved by the hyperplane that has the largest distance to the nearest training-data point of any class

In many real-world problems data are noisy and there will in general be no linear separation in the feature space

When the two classes are not linearly separable, the condition for the optimal hyper-plane can be relaxed by including an extra term:

$$y_i(\mathbf{x}_i^T\mathbf{w} + b) \geq 1 - \xi_i, \quad (i = 1, \cdots, m)$$

For minimum error, $\xi_i \geq 0$ should be minimized as well as $||\mathbf{w}||$, and the objective function becomes:

$$\begin{aligned} \text{minimize} \quad & \mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m}\xi_i^k \\ \text{subject to} \quad & y_i(\mathbf{x}_i^T\mathbf{w} + b) \geq 1 - \xi_i, \quad \text{and} \quad \xi_i \geq 0; \quad (i = 1, \cdots, m) \end{aligned}$$

Here C is a regularization parameter that controls the trade-off between maximizing the margin and minimizing the training error. The C parameter tells the SVM optimization how much you want to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. Conversely, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, even if that hyperplane misclassifies more points.

[6]:
```
C = [0.001, 0.01, 0.1, 1, 10, 100,1000]
```

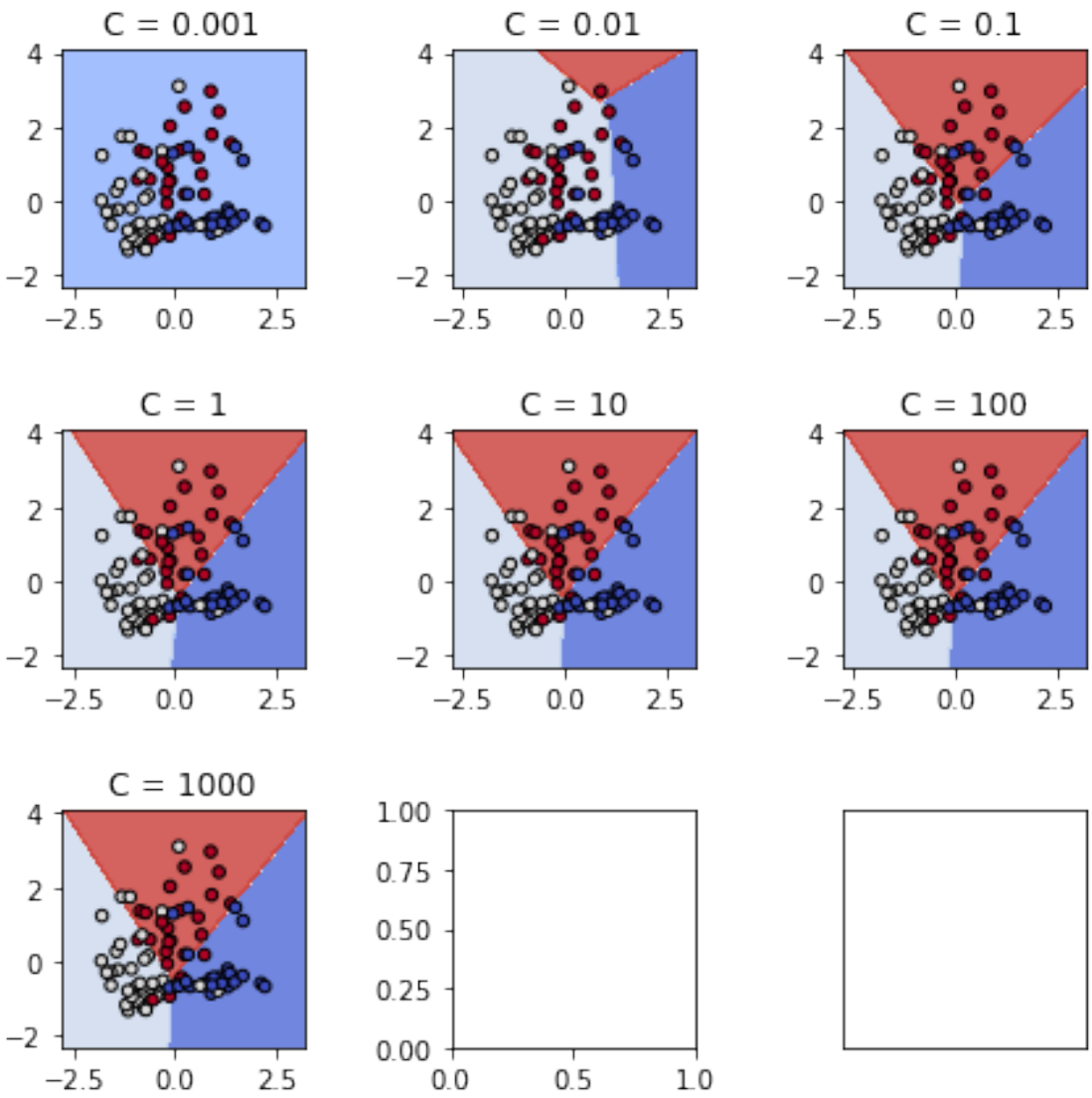
Now we'll train and evaluate a model for each value of C in the list. We want to choose the C that gives the highest accuracy on the validation set.

[7]:
```
models =[]
accuracies = []

for c in C:
    model, accuracy = SVM(X_train_2D,y_train,X_val_2D,y_val,'linear',c,'auto')
    models.append(model)
    accuracies.append(accuracy)
```

```
C = 0.001
---> Accuracy = 0.36
C = 0.01
---> Accuracy = 0.50
C = 0.1
```

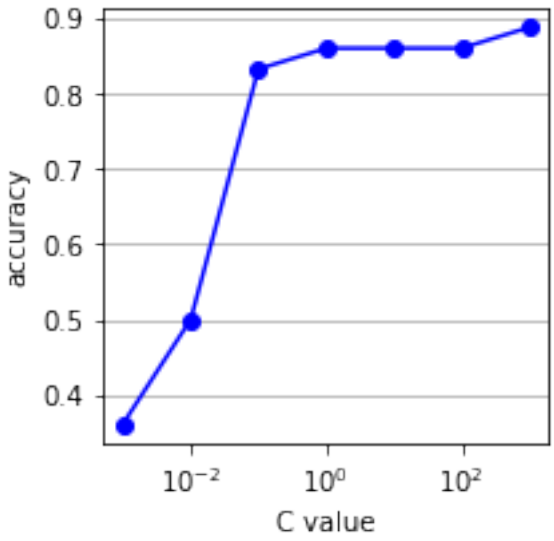
```
---> Accuracy = 0.83
C = 1
---> Accuracy = 0.86
C = 10
---> Accuracy = 0.86
C = 100
---> Accuracy = 0.86
C = 1000
---> Accuracy = 0.89
```

[8]:
```
fig, sub = plt.subplots(nrows = 3, ncols = 3,figsize=(7,7))
fig.subplots_adjust(hspace=0.6, wspace=0.6)
for i,(clf,ax) in enumerate(zip(models, sub.flatten())):
    plot_model(X_train_2D,y_train,X_train_2D,y_train,clf,
               'coolwarm',"C = " + str(C[i]), ax)
```



The plots above show the effect the parameter C. If we choose a small margin (large C), we don't trust that our data are well separated, so it will be difficult to classify them, and in this case, a small margin will help. But if the margin is too small, it could not be possible to separate the classes using too few samples as support vectors.

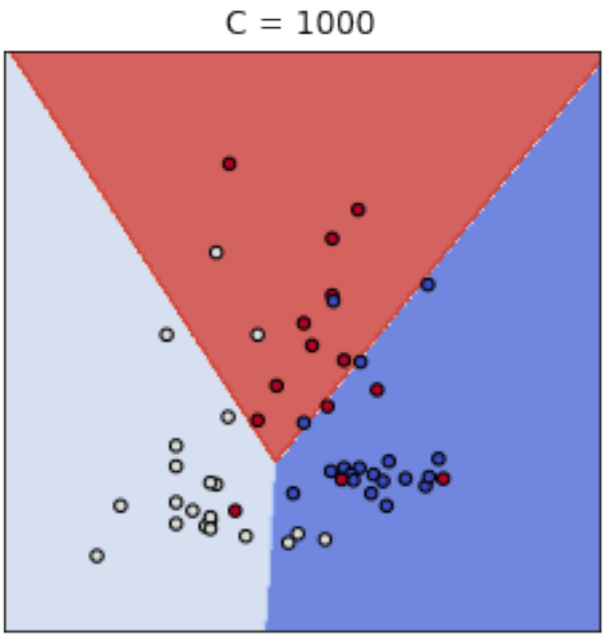
[9]:
```
plot_accuracy(C,accuracies,'C value',1)
```



As we can see from the above chart, the accuracy is increasing for higher values of C, the model with the highest accuracy is in fact a small margin SVM. If there are more values of C that give higher accuracy, we choose the smaller one which is the one that gives the larger margin

[10]:
```
ind = np.argmax(accuracies)
best_C = C[ind]
best_clf,a = SVM(X_train_2D,y_train,X_test_2D,y_test,'linear',best_C,'auto')
fig, sub = plt.subplots(nrows = 1, ncols = 1,figsize=(4,4))
plot_model(X_train_2D,y_train,X_test_2D,y_test,best_clf,'coolwarm',"C = " +␣
↪str(C[ind]),sub)
```

```
C = 1000
---> Accuracy = 0.78
```



RBF Kernel

The function of kernel is to take data as input and transform it into the required form. The kernel functions return the inner product between two points in a suitable feature space. Thus by defining a notion of similarity, with little computational cost even in very high-dimensional spaces.

Now we will use the RBF kernel, that is one of the most used,instead of the linear one

$$K(x, x\ ) = exp(\lambda||x - x\ ||^2)$$

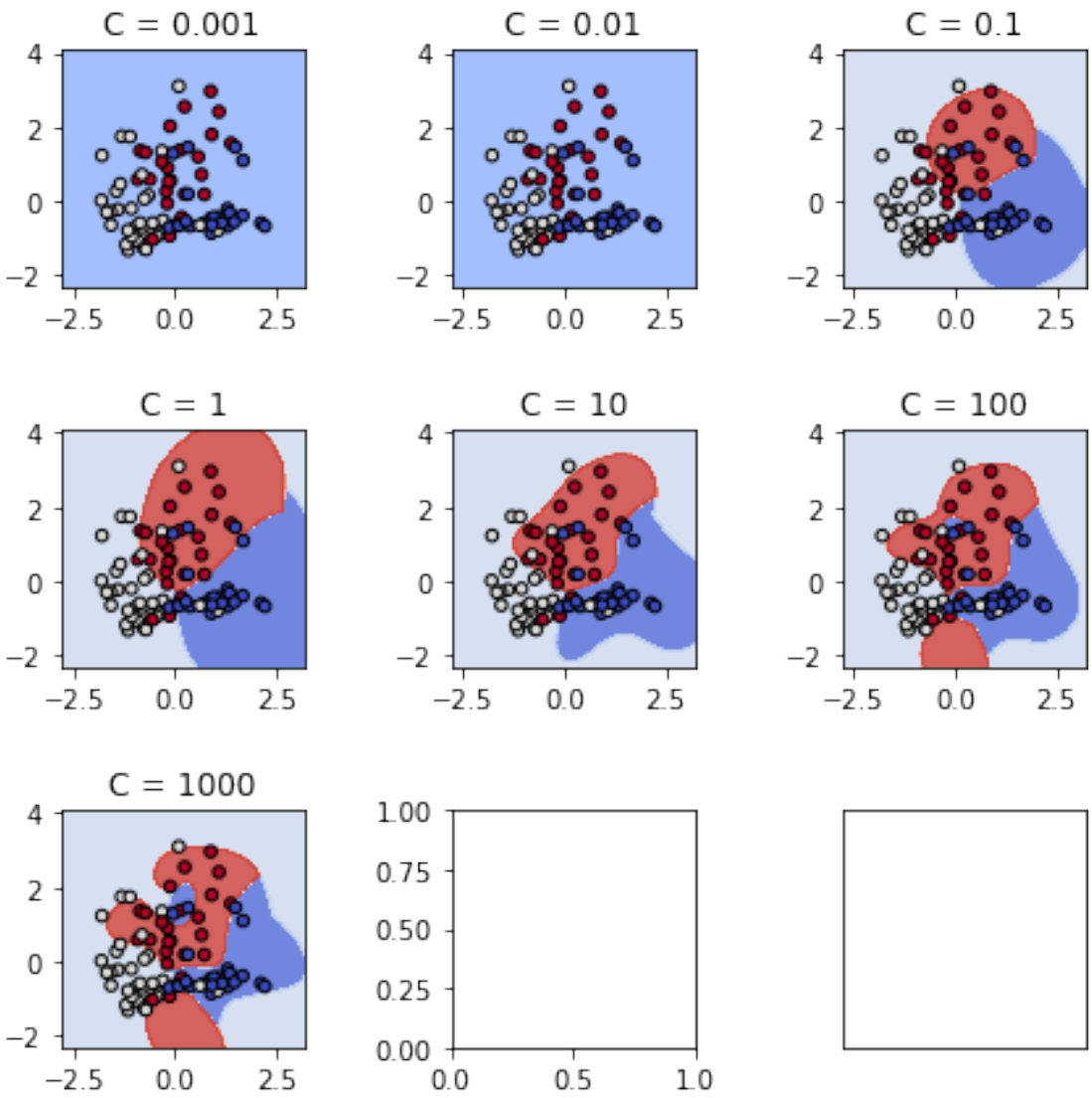
As for the linear, we will train and evaluate a model for each value of C and we will choose the C that gives the highest accuracy.

[11]:
```
models_rbf = []
accuracies_rbf = []

for c in C:
    model_rbf, accuracy_rbf =␣
 ↪SVM(X_train_2D,y_train,X_val_2D,y_val,'rbf',c,'auto')
    models_rbf.append(model_rbf)
    accuracies_rbf.append(accuracy_rbf)
```

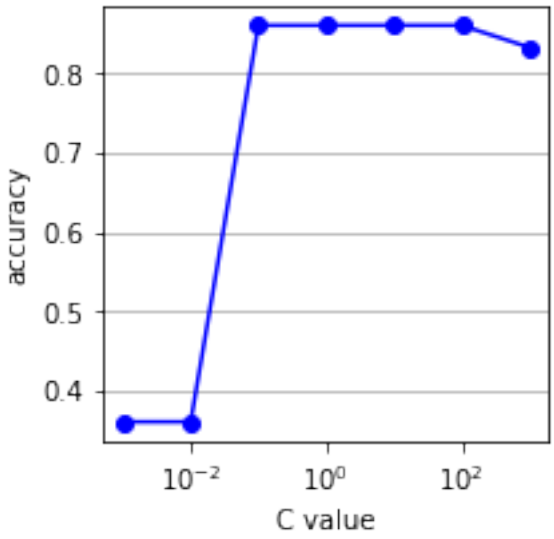
```
C = 0.001
---> Accuracy = 0.36
C = 0.01
---> Accuracy = 0.36
C = 0.1
```

```
---> Accuracy = 0.86
C = 1
---> Accuracy = 0.86
C = 10
---> Accuracy = 0.86
C = 100
---> Accuracy = 0.86
C = 1000
---> Accuracy = 0.83
```

[12]:
```
fig, sub = plt.subplots(nrows = 3, ncols = 3,figsize=(7,7))
fig.subplots_adjust(hspace=0.6, wspace=0.6)

for i,(clf,ax) in enumerate(zip(models_rbf, sub.flatten())):
    plot_model(X_train_2D,y_train,X_train_2D,y_train,clf,'coolwarm',"C = " +␣
↪str(C[i]),ax)
```



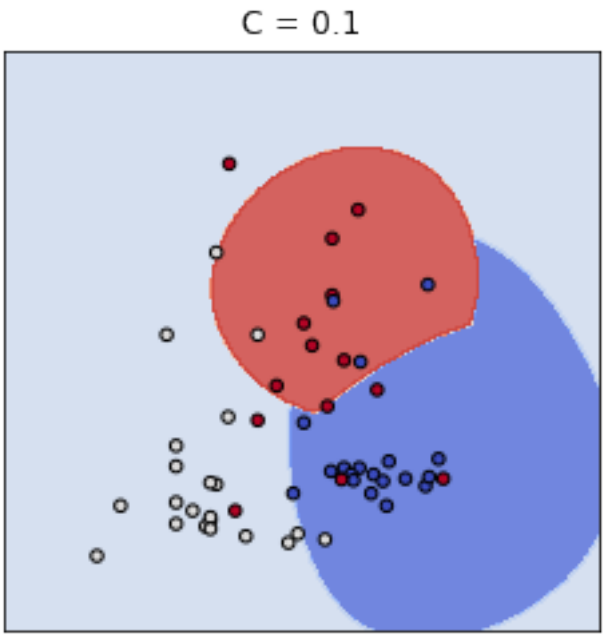
[13]:
```
plot_accuracy(C,accuracies_rbf,'C value',1)
```



As for the linear, if there are more values of C that give higher accuracy, we choose the smaller one

[14]:
```
ind = np.argmax(accuracies_rbf)
best_C_rbf = C[ind]
best_clf,a =SVM(X_train_2D,y_train,X_test_2D,y_test,'rbf',best_C_rbf,'auto')
fig, sub = plt.subplots(nrows = 1, ncols = 1,figsize=(4,4))
plot_model(X_train_2D,y_train,X_test_2D,y_test,best_clf,'coolwarm',"C = " +␣
↪str(best_C_rbf),sub)
```

```
C = 0.1
---> Accuracy = 0.78
```



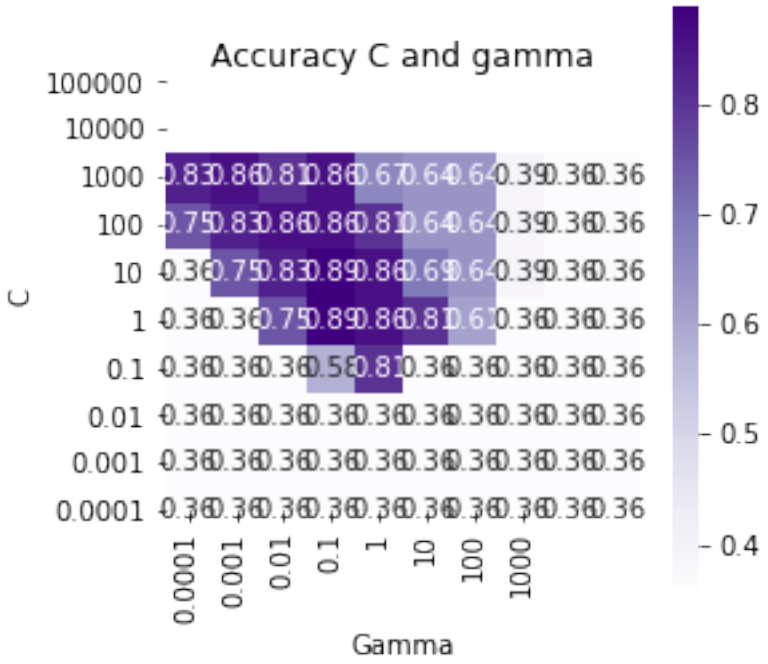
Grid search C and Gamma

This time we have to find the best pair C and gamma. The gamma parameter defines how far the influence of a single training example reaches. This means that high Gamma will consider only points close to the plausible hyperplane and low Gamma will consider points at greater distance. the gamma parameter can be said to adjust the curvature of the decision boundary.

[15]:
```
C_list = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]

Gamma_list = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]

models, hyperparameters, highest_accuracy = applySVM_C_Gamma('rbf', C_list,␣
 ↪Gamma_list,
            X_train_2D, y_train, X_val_2D, y_val,)
```

[16]:
```
plotHeat(hyperparameters,C_list, Gamma_list)

print("The pairs that give highest accuracy are: ")
C_best,Gamma_best =␣
 ↪Find_BestHyperparameters(hyperparameters,C_list,Gamma_list,highest_accuracy)
```

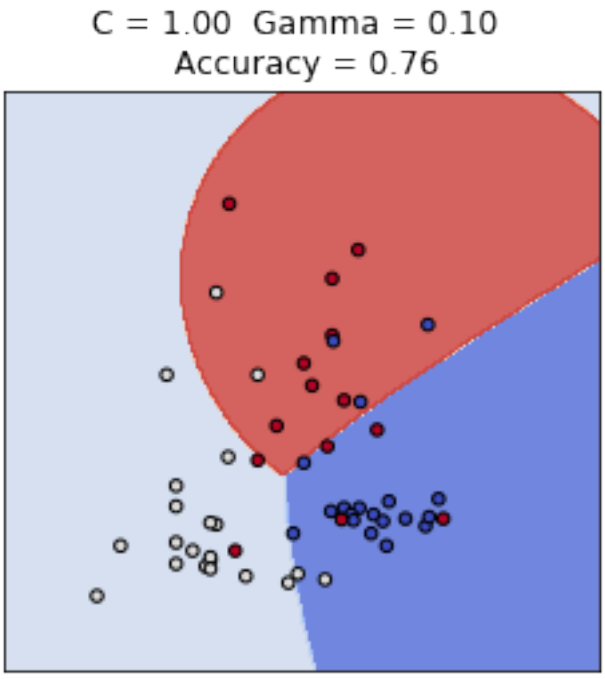
```
The pairs that give highest accuracy are:
C= 1 , Gamma= 0.1 accuracy  0.8888888888888888
C= 10 , Gamma= 0.1 accuracy  0.8888888888888888
```



The heatmap above shows the accuracy on the validation set for each pair C and Gamma. There is a region of the heatmap populated by high accuracy values. We can see that high value of Gamma doesn't working very well to classify our samples, even if we consider an high value for C.

[17]:
```
print("Selecting as best Hyperparameters: C=", C_best, ", Gamma=", Gamma_best)

best_clf,accuracy␣
 ↪=SVM(X_train_2D,y_train,X_test_2D,y_test,'rbf',C_best,Gamma_best)
fig, sub = plt.subplots(nrows = 1, ncols = 1,figsize=(4,4))
plot_model(X_train_2D,y_train,X_test_2D,y_test,best_clf,
           'coolwarm',"C = %.2f " % C_best + " Gamma = %.2f " % Gamma_best +␣
 ↪"\n Accuracy = %.2f" % accuracy,sub)
```

```
Selecting as best Hyperparameters: C= 1 , Gamma= 0.1
C = 1
---> Accuracy = 0.76
```



We obtained a lower accuracy than that we had obtained without grid search on gamma. We are evaluating the models on only 20% of the available samples, with a small dataset like the one we used, this means that we are using too less data to decide which values of the hyperparameters to use to tune our models.

K-Fold

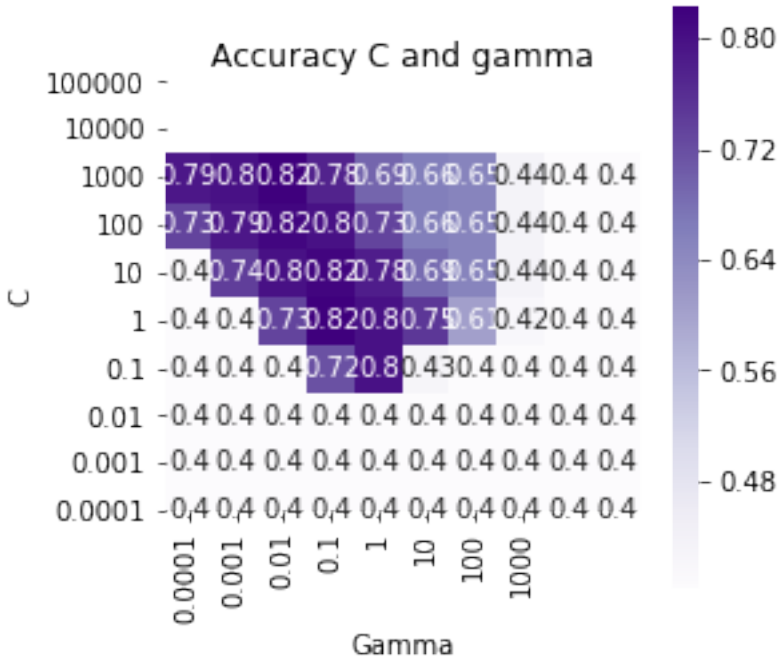
Now we'll use the K-fold crossvalidation technique to validate our models. As before, we want to find the values of C and Gamma that give us the highest accuracy. We merge the training and validation split. We now have 70% training and 30% test data. The training set will then be splitted into a number of "K" different sets, and we will perform K rounds of training and evaluation, each time training on K-1 folds, and testing on the remaining fold.

This will allow to perform the validation more in depth, by using more samples. We have a chance that by doing this we'll tune the model with the right parameters.

[18]:
```
from sklearn.model_selection import cross_val_score

X_train_Kfold = np.concatenate((X_train_2D, X_val_2D))
y_train_Kfold = np.concatenate((y_train,y_val))


models, hyperparameters, highest_accuracy = SVM_C_Gamma_Kfold('rbf', C_list,␣
↪Gamma_list,
                     X_train_Kfold, y_train_Kfold, 5)
```

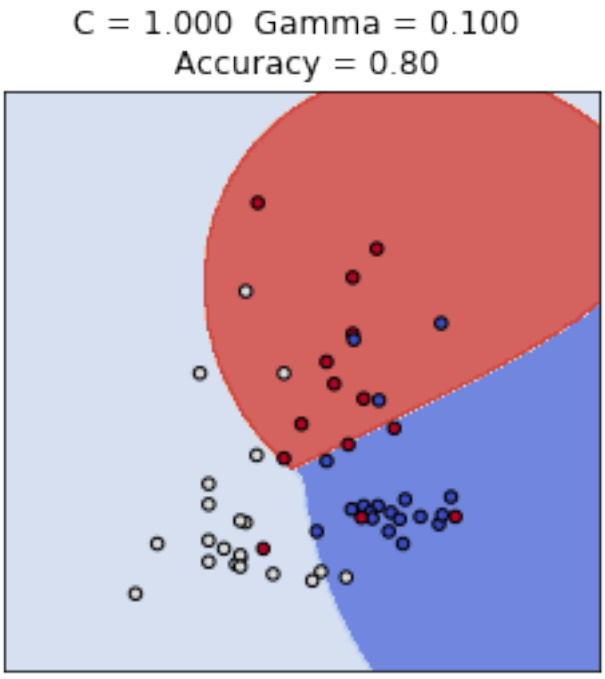
[19]:
```
plotHeat(hyperparameters,C_list, Gamma_list)
C_best,Gamma_best =␣
↪Find_BestHyperparameters(hyperparameters,C_list,Gamma_list,highest_accuracy)
```

```
C= 1 , Gamma= 0.1 accuracy  0.8233333333333335
C= 1000 , Gamma= 0.01 accuracy  0.8233333333333335
```



[20]:
```
print("Selecting as best Hyperparameters: C=", C_best, ", Gamma=", Gamma_best)

best_clf,accuracy␣
↪=SVM(X_train_Kfold,y_train_Kfold,X_test_2D,y_test,'rbf',C_best,Gamma_best)
fig, sub = plt.subplots(nrows = 1, ncols = 1,figsize=(4,4))
plot_model(X_train_Kfold,y_train_Kfold,X_test_2D,y_test,best_clf,
           'coolwarm',"C = %.3f " % C_best + " Gamma = %.3f " % Gamma_best +␣
↪"\n Accuracy = %.2f" % accuracy,sub)
```

```
Selecting as best Hyperparameters: C= 1 , Gamma= 0.1
C = 1
---> Accuracy = 0.80
```



In K-NN the sample is given a classification based only on the nearby instances, and anything farther away is ignored.
K-NN can generate a highly convoluted decision boundary as it is driven by the raw training data. SVM on the other hand, attempts to find a hyper-plane separating the different classes, with the maximum margin. This means that create a large margin between data points and the decision boundary. The most important training instances are the ones on the boundaries. The tradeoff between how many instances to allow on the wrong side and how complex the boundary is controls how complex the model is. The SVM is generally more complex becouse take more information into account when classifying the target instance and will generally have much better accuracy.

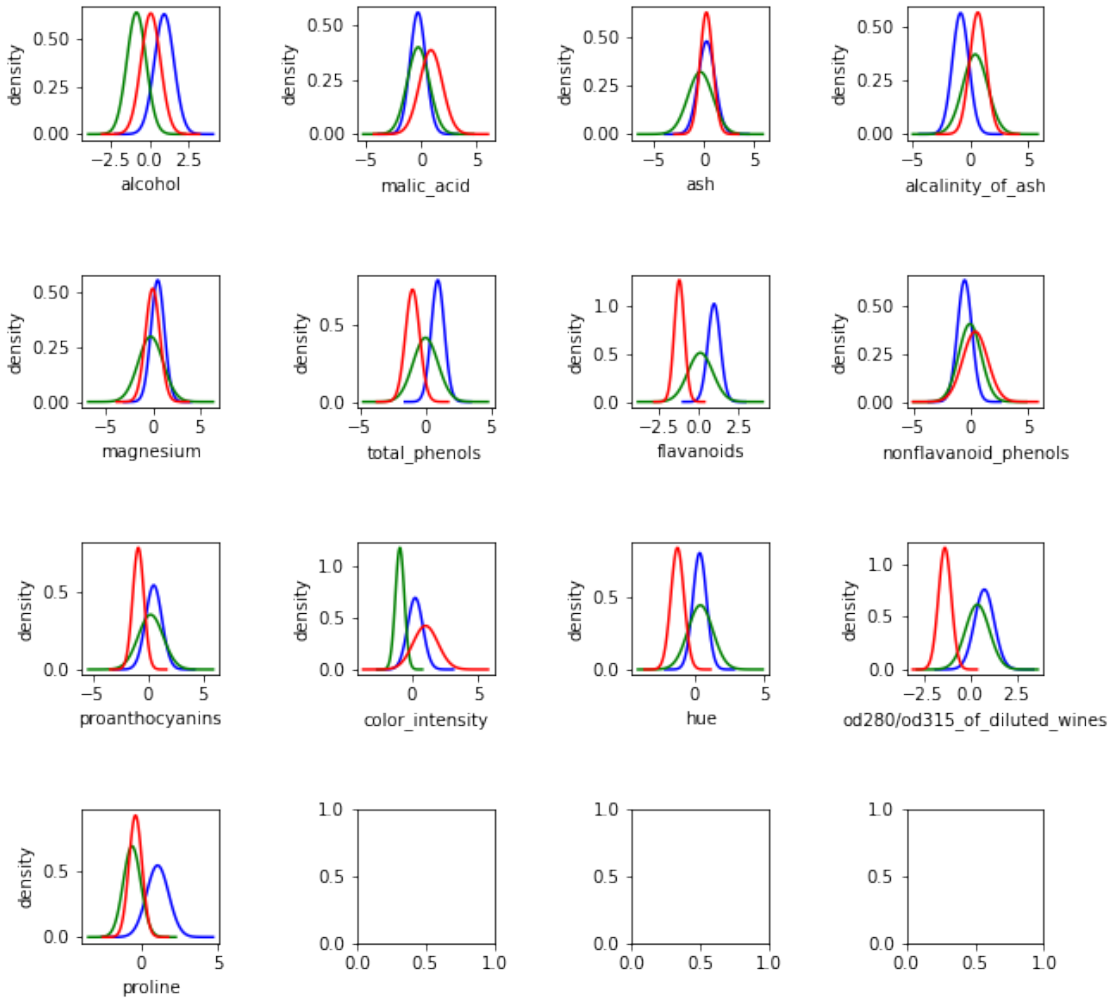
Analysis with other features pair

[21]:
```
data = pd.DataFrame(data=X_train,columns=wine_dataset.feature_names)

data['target']=y_train
# data['class']=data['target'].map(lambda ind: wine_dataset.target_names[ind])
means = data.groupby('target').mean() #mean for each features and class
std = data.groupby('target').std() #std for each features and class
```

the features that most helps discriminate among the three class is the one that has the most distant means and the smallest standard deviations. This corresponds to distributions of values that are well separated and all fall within a short distance on the mean value.

[22]:
```
from scipy.stats import norm
colors = ['b','g','r']
```

```
k = 0
fig, sub = plt.subplots(nrows = 4, ncols = 4,figsize=(10,10))
fig.subplots_adjust(hspace=1, wspace=1)
sub = sub.flatten()
for i, n in enumerate(wine_dataset.feature_names):

    for c, color in enumerate(colors):
        filter = data['target'] == c
#         values = data[n].where(filter)

#         plt.hist(values, density=True, alpha=0.2, color=color)
        u = means[n][c]
        s = std[n][c]
        x = np.linspace(u-5*s, u+5*s, 100)
        sub[k].plot(x, norm(u,s).pdf(x), color=color)
        sub[k].set_xlabel(f"{n}")
        sub[k].set_ylabel("density")
    k = k+1
```

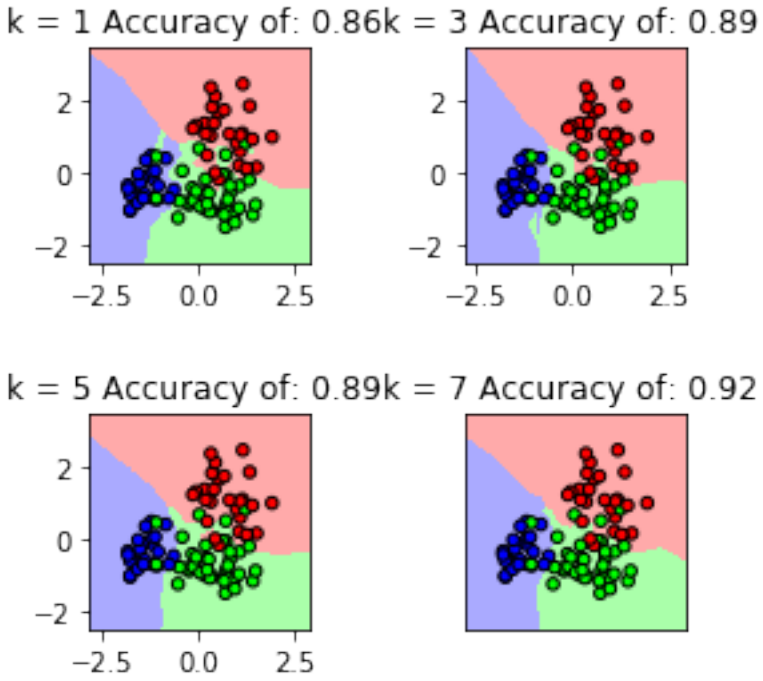


We use the last 2 features to try to achieve greater accuracy

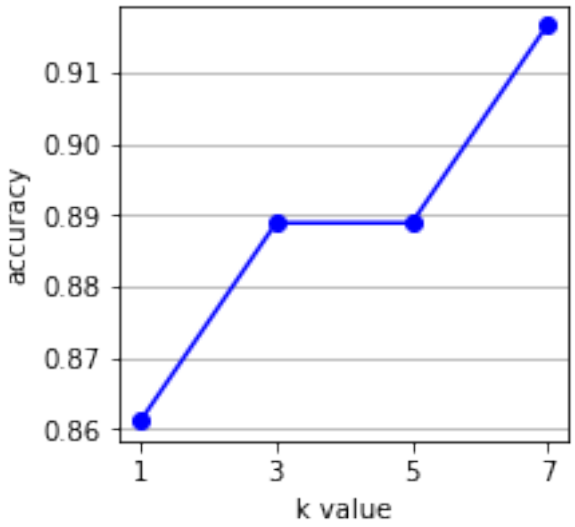
[23]:
```
X_train_2D = X_train[:,11:13]
X_test_2D = X_test[:,11:13]
X_val_2D = X_val[:,11:13]
```

KNN

[24]:
```
Ks=[1,3,5,7]
accuracies_knn = Apply_and_plot_Knn(Ks,X_train_2D,y_train,X_val_2D,y_val,2,2,0)
```



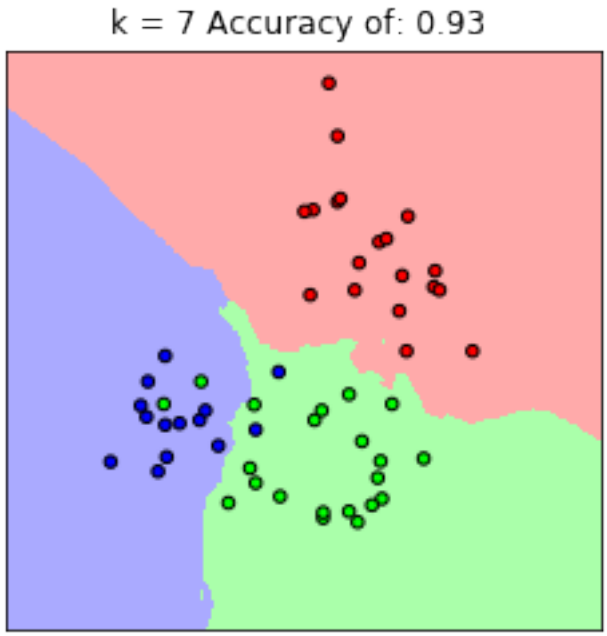
[25]:
```
plot_accuracy(Ks,accuracies_knn,'k value',0)
#select best k value
best_k = Ks[np.argmax(accuracies_knn)]
print("The value of k that gives the higher accuracy is k =", best_k)
```




```
The value of k that gives the higher accuracy is k = 7
```

[26]:
```
A = Apply_and_plot_Knn([best_k],X_train_2D,y_train,X_test_2D,y_test,1,1,1)
print("The accuracy of the model on the test set is %.2f" % A[0])
```

```
The accuracy of the model on the test set is 0.93
```



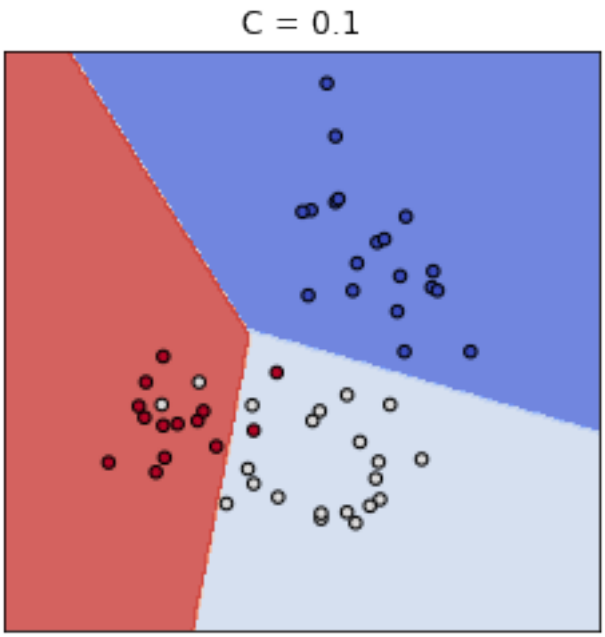
Linear SVM

[27]:
```
C = [0.001, 0.01, 0.1, 1, 10, 100,1000]
models =[]
accuracies = []

for c in C:
    model, accuracy = SVM(X_train_2D,y_train,X_val_2D,y_val,'linear',c,'auto')
    models.append(model)
    accuracies.append(accuracy)
```

```
C = 0.001
---> Accuracy = 0.36
C = 0.01
---> Accuracy = 0.56
C = 0.1
---> Accuracy = 0.92
C = 1
---> Accuracy = 0.89
C = 10
---> Accuracy = 0.92
C = 100
---> Accuracy = 0.92
C = 1000
---> Accuracy = 0.92
```

[28]:
```
ind = np.argmax(accuracies)
best_C = C[ind]
best_clf,a = SVM(X_train_2D,y_train,X_test_2D,y_test,'linear',best_C,'auto')
fig, sub = plt.subplots(nrows = 1, ncols = 1,figsize=(4,4))
plot_model(X_train_2D,y_train,X_test_2D,y_test,best_clf,'coolwarm',"C = " +␣
 ↪str(C[ind]),sub)
```

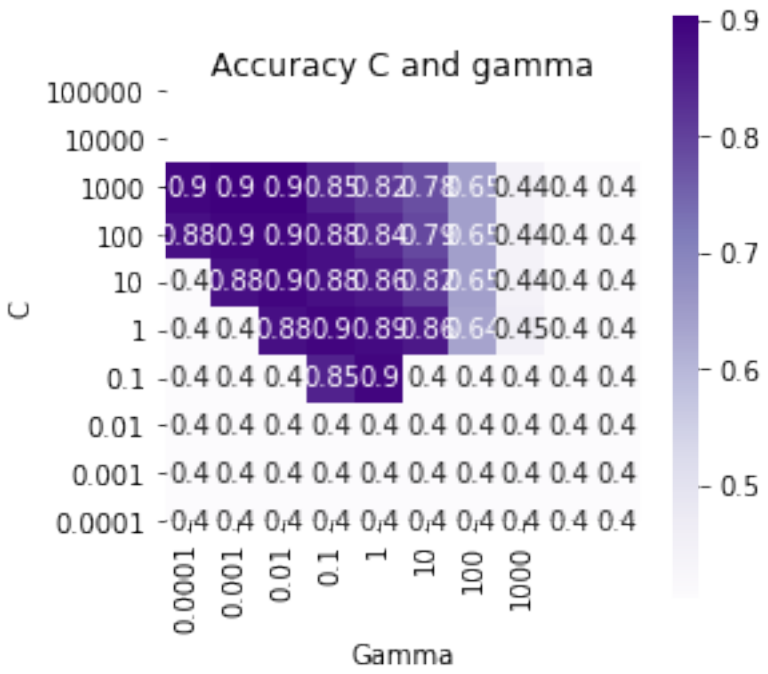
```
C = 0.1
---> Accuracy = 0.93
```



Grid search and K-fold

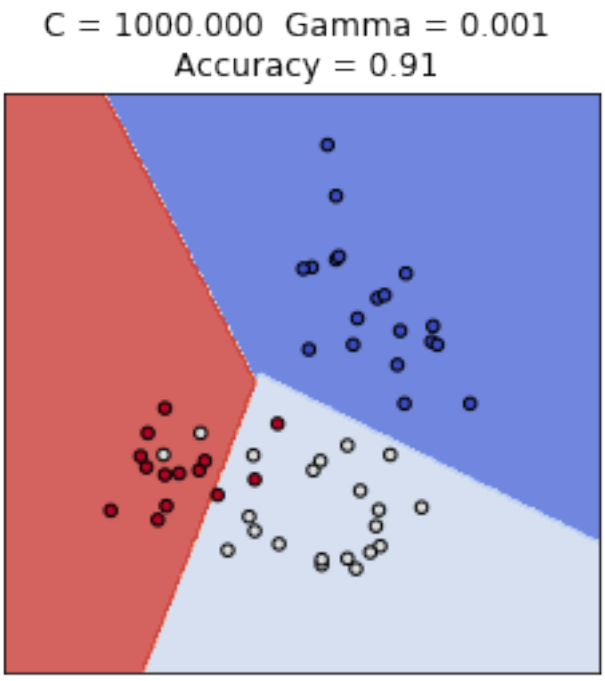
[29]:
```
C_list = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]

Gamma_list = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]
X_train_Kfold = np.concatenate((X_train_2D, X_val_2D))
y_train_Kfold = np.concatenate((y_train,y_val))


models, hyperparameters, highest_accuracy = SVM_C_Gamma_Kfold('rbf', C_list,␣
 ↪Gamma_list,
                     X_train_Kfold, y_train_Kfold, 5)
plotHeat(hyperparameters,C_list, Gamma_list)
C_best,Gamma_best =␣
 ↪Find_BestHyperparameters(hyperparameters,C_list,Gamma_list,highest_accuracy)
```

```
C= 1000 , Gamma= 0.001 accuracy  0.9036666666666665
C= 1000 , Gamma= 0.01 accuracy  0.9036666666666665
```



[30]:
```
print("Selecting as best Hyperparameters: C=", C_best, ", Gamma=", Gamma_best)

best_clf,accuracy␣
 ↪=SVM(X_train_Kfold,y_train_Kfold,X_test_2D,y_test,'rbf',C_best,Gamma_best)
fig, sub = plt.subplots(nrows = 1, ncols = 1,figsize=(4,4))
plot_model(X_train_Kfold,y_train_Kfold,X_test_2D,y_test,best_clf,
           'coolwarm',"C = %.3f " % C_best + " Gamma = %.3f " % Gamma_best +␣
 ↪"\n Accuracy = %.2f" % accuracy,sub)
```

```
Selecting as best Hyperparameters: C= 1000 , Gamma= 0.001
C = 1000
---> Accuracy = 0.91
```

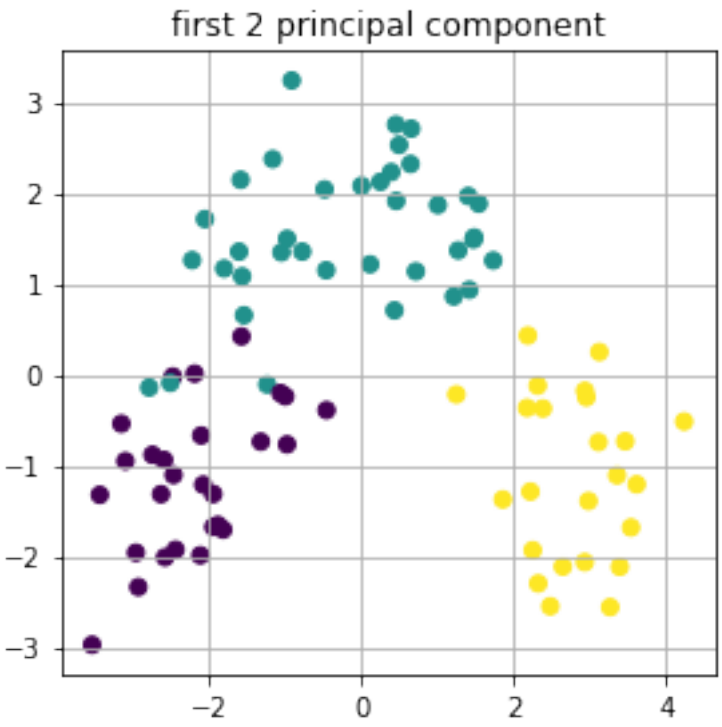


PCA

We use PCA to account for as much of the variability in the data as possible

[31]:
```
pca = PCA(2) # get first two principal components
pca.fit(X_train)
X_train_PCA = pca.transform(X_train)
X_val_PCA =pca.transform(X_val)
X_test_PCA = pca.transform(X_test)
```

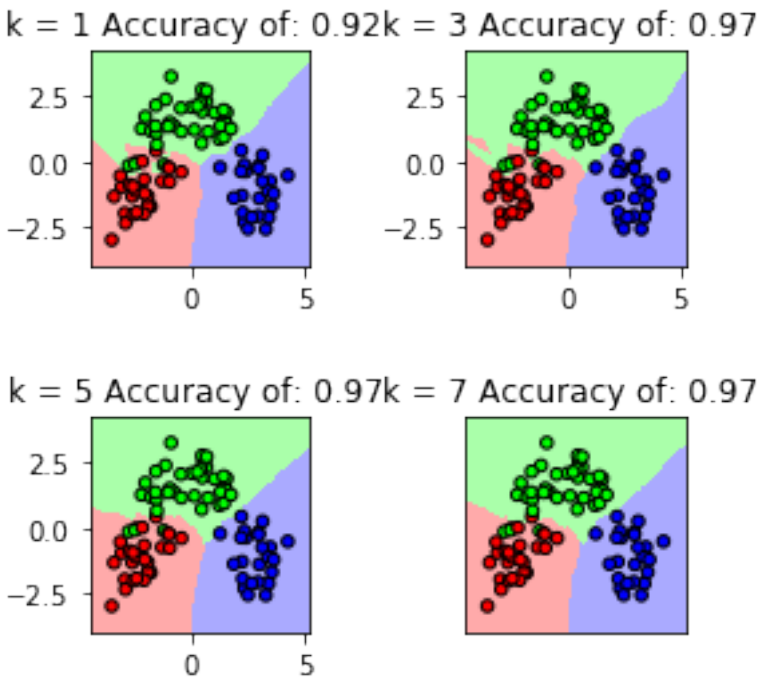
[32]:
```
fig, ax = plt.subplots(1, 1, figsize=(4, 4))

ax.scatter(X_train_PCA[:,0],X_train_PCA[:,1], c=y_train)
ax.set_title('first 2 principal component')
ax.grid()

plt.tight_layout()
plt.show()
```



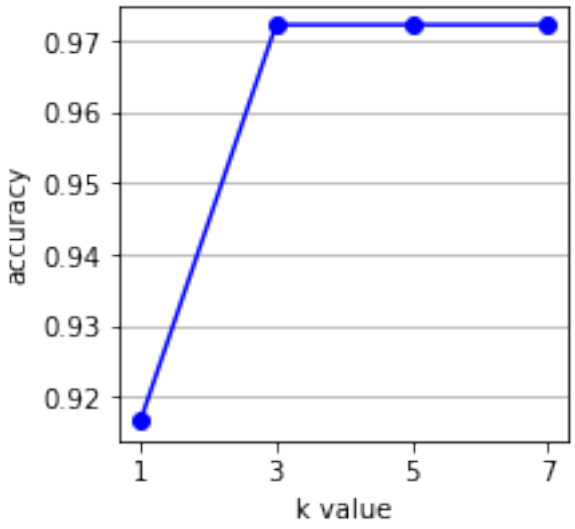
The chart above show that the first 2 principal components the data can be clearly separated into 3 classes so we should get a higher accuracy

KNN

[33]:
```
Ks=[1,3,5,7]
accuracies_knn =␣
↪Apply_and_plot_Knn(Ks,X_train_PCA,y_train,X_val_PCA,y_val,2,2,0)
```



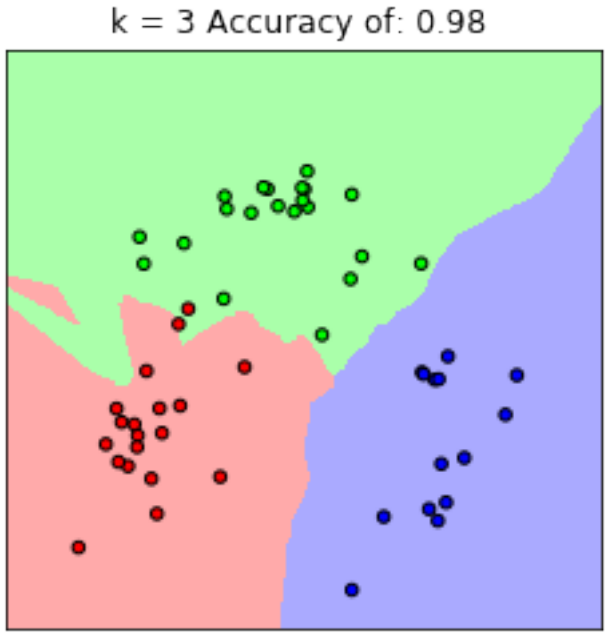
[34]:
```
plot_accuracy(Ks,accuracies_knn,'k value',0)
#select best k value
best_k = Ks[np.argmax(accuracies_knn)]
print("The value of k that gives the higher accuracy is k =", best_k)
```




```
The value of k that gives the higher accuracy is k = 3
```

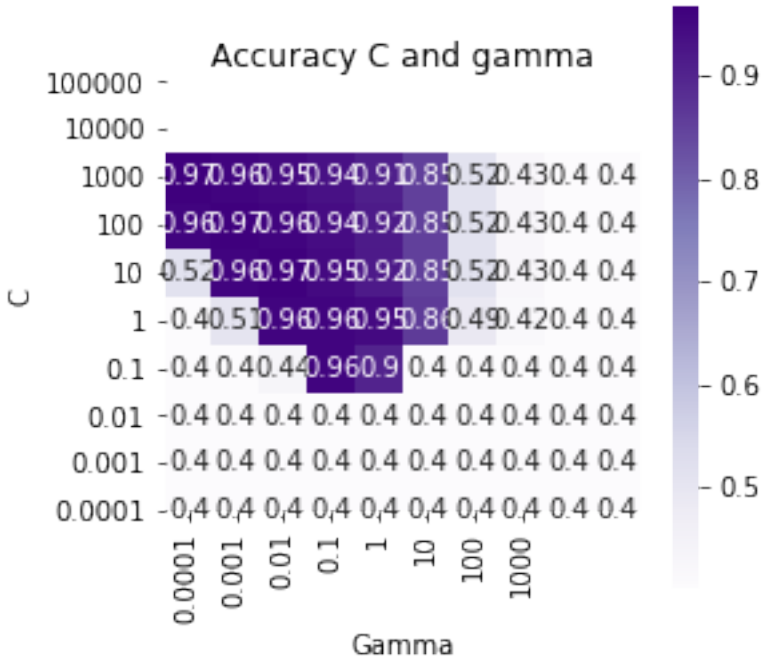
[35]:
```
A = Apply_and_plot_Knn([best_k],X_train_PCA,y_train,X_test_PCA,y_test,1,1,1)
print("The accuracy of the model on the test set is %.2f" % A[0])
```

```
The accuracy of the model on the test set is 0.98
```



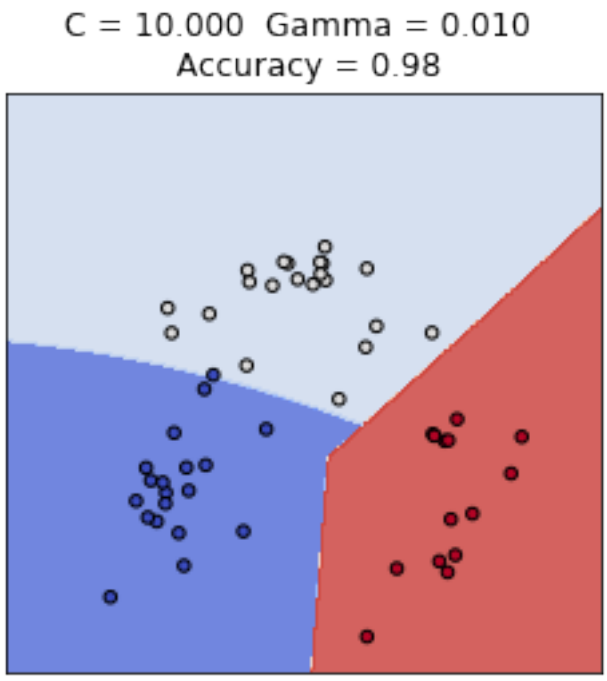
[36]:
```
C_list = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]

Gamma_list = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]
X_train_Kfold = np.concatenate((X_train_PCA, X_val_PCA))
y_train_Kfold = np.concatenate((y_train,y_val))


models, hyperparameters, highest_accuracy = SVM_C_Gamma_Kfold('rbf', C_list,␣
 ↪Gamma_list,
                     X_train_Kfold, y_train_Kfold, 5)
plotHeat(hyperparameters,C_list, Gamma_list)
C_best,Gamma_best =␣
 ↪Find_BestHyperparameters(hyperparameters,C_list,Gamma_list,highest_accuracy)
```

```
C= 10 , Gamma= 0.01 accuracy  0.968
C= 100 , Gamma= 0.001 accuracy  0.968
C= 1000 , Gamma= 0.0001 accuracy  0.968
```



[37]:
```
print("Selecting as best Hyperparameters: C=", C_best, ", Gamma=", Gamma_best)

best_clf,accuracy␣
 ↪=SVM(X_train_Kfold,y_train_Kfold,X_test_PCA,y_test,'rbf',C_best,Gamma_best)
fig, sub = plt.subplots(nrows = 1, ncols = 1,figsize=(4,4))
plot_model(X_train_Kfold,y_train_Kfold,X_test_PCA,y_test,best_clf,
           'coolwarm',"C = %.3f " % C_best + " Gamma = %.3f " % Gamma_best +␣
 ↪"\n Accuracy = %.2f" % accuracy,sub)
```

```
Selecting as best Hyperparameters: C= 10 , Gamma= 0.01
C = 10
---> Accuracy = 0.98
```


C = 10.000 Gamma = 0.010
Accuracy = 0.98