Machine_Learning_Homework2

January 16, 2019

SUPPORT VECTOR MACHINE

In machine learning, Support Vector Machines are supervised learning models developed to deal with classification of multiple classes.

The elements in favor of this type of algorithm are: 1. it works really well with clear margin separator 2. it uses a subset of training points in the decision function (called support vector), so it is memory efficient

Instead, the unfavorable elements are: 1. It doesn't perform well, when we have large data set because the required training time is higher 2. It also doesn't perform very well, when the data set has more noise i.e. target classes are overlapping 3. SVM doesn't directly provide probability estimates, these are calculated using an expensive five-fold cross-validation. It is related SVC method of Python scikit-learn library.

In this homework, we will see how to manage in three different ways the classification of a multivariate dataset.

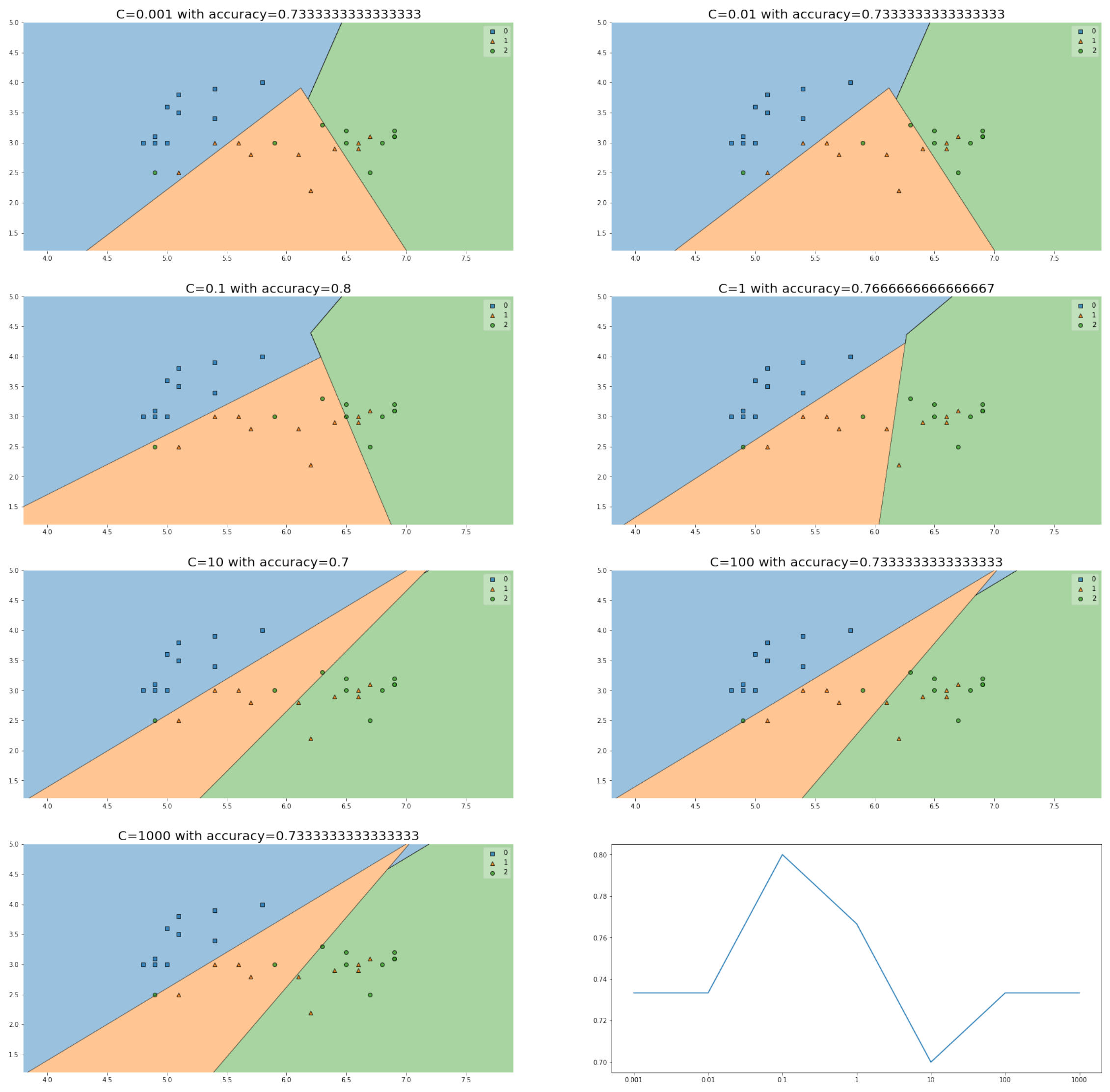
LINEAR SVM

In this case, we are using a linear kernel, whose equation for prediction for a new input is done using the dot product between the input (x) and each support vector (xi), as follows:

$f(x) = sum(a_i * (x, x_i)) + b$

This is an equation that involves calculating the inner products of a new input vector (x) with all support vectors in training data. The coefficients b and a_i (for each input) must be estimated from the training data by the learning algorithm

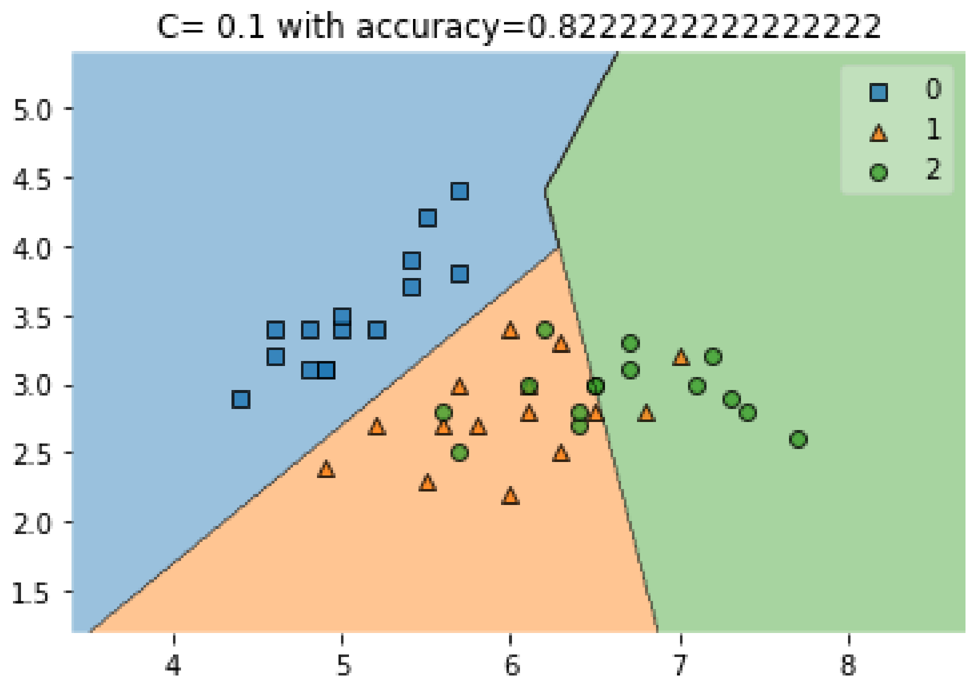
```python
#import dataset
iris = datasets.load_iris()
X = iris.data[:,:2]
y = iris.target

#splitting dataset 5:2:3
X_train, X_test_and_validation, y_train, y_test_and_validation = train_test_split(
    X, y, test_size=0.5, random_state=1, stratify=y)
X_test, X_validation, y_test, y_validation = train_test_split(
    X_test_and_validation, y_test_and_validation, test_size=0.4, random_state=1,
            stratify=y_test_and_validation)

C = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
Y = []

fig, ax = plt.subplots(4, 2, figsize=(30, 30))

for c, i, j in zip(C, [0, 0, 1, 1, 2, 2, 3], [0, 1, 0, 1, 0, 1, 0]):
    clf = svm.SVC(C=c, kernel='linear')
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_validation)
    accuracy = accuracy_score(y_validation, y_pred)
    Y.append(accuracy)
    ax[i, j].set_title("C="+str(c)+" with accuracy="+str(accuracy), fontsize=20)
    plot_decision_regions(np.asarray(X_validation), np.asarray(y_validation), clf=clf, ax=ax[i, j])

X2 = np.arange(7)
X = xticks(X2, C)
ax[3, 1].plot(Y)
plt.show()
```



The C parameter is a fundamental value for SVM. As a matter of fact, if we increase the value of it, the SVM optimization will choose a smaller-margin hyperplane only if the new hyperplane is able to better classify the training points.

On the other hand, choosing a small value of C will cause the optimizer to look for a larger-margin hyperplane, despite of the presence of a larger number of missclassifications. For example, if we take a very little value for C, we can see that the data will be wrongly classified even thought data are linearly separable.

```
clf = svm.SVC(C=0.1, kernel='linear')
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
plt.title("C= 0.1 with accuracy="+str(accuracy))
plot_decision_regions(np.asarray(X_test), np.asarray(y_test), clf=clf)
plt.show()
```



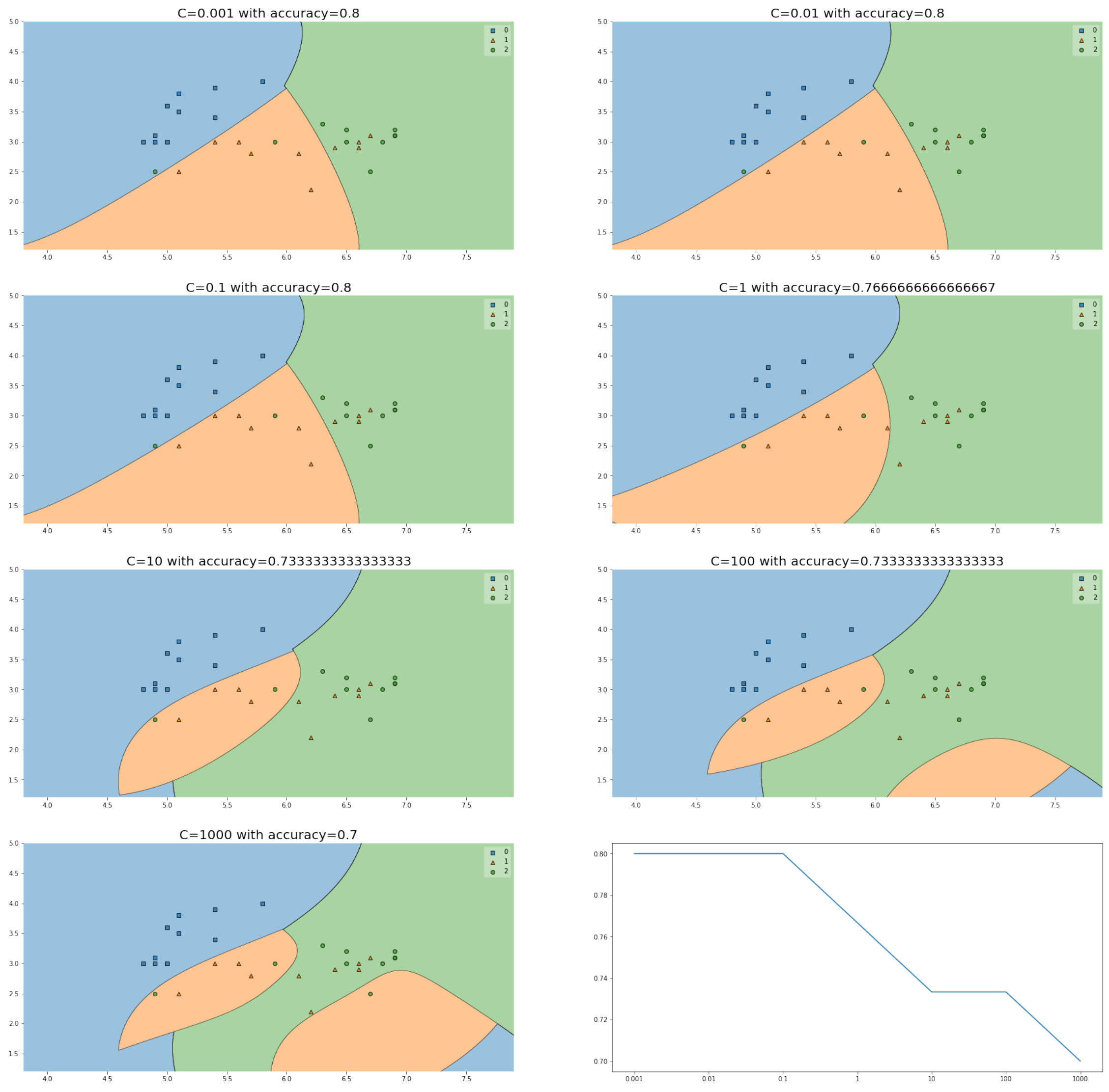
RADIAL BASIS FUNCTION SVM

When data are not linearly separable, we can not use Linear Kernel.

In this case, we have to use the Kernel Trick to map the features of our dataset to a higher dimension space where the data is linearly separable.

The RBF Kernel implements nonlinear functions to classify the data: in the following code we can see how the same data used previously are differently classified.

```
fig, ax = plt.subplots(4, 2, figsize=(30, 30))

Y = []

for c, i, j in zip(C, [0, 0, 1, 1, 2, 2, 3], [0, 1, 0, 1, 0, 1, 0]):
    clf = svm.SVC(C=c, kernel='rbf', gamma='auto')
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_validation)
    accuracy = accuracy_score(y_validation, y_pred)
    Y.append(accuracy)
    ax[i, j].set_title("C="+str(c)+" with accuracy="+str(accuracy), fontsize=20)
    plot_decision_regions(np.asarray(X_validation), np.asarray(y_validation), clf=clf, ax=ax[i, j])

X2 = np.arange(7)
X = xticks(X2, C)
ax[3, 1].plot(Y)
plt.show()
```

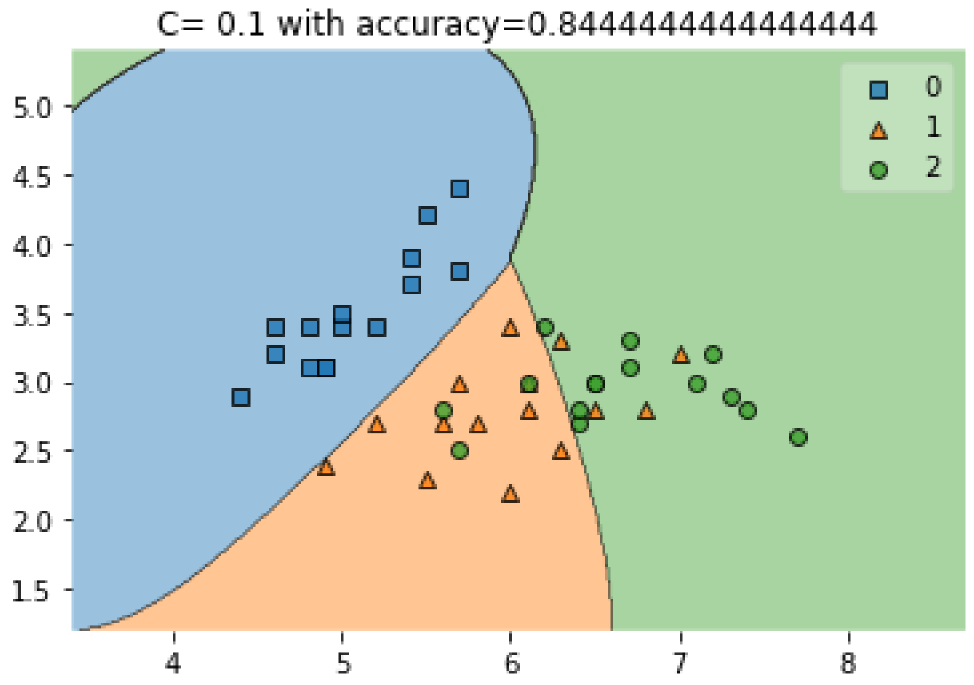


If we compare these graphs with the ones of Linear case, we can clearly see how non-linear functions help to classifiy the data. As we can see, the accuracy between the two methods are not so different, and this happens due to the particular dataset used.

The Linear Kernel implementation is much simpler and faster, which is why we prefer to use this method when there is the certainty that the data are linearly separable.

```
clf = svm.SVC(C=0.1, kernel='rbf', gamma='auto')
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

```
accuracy = accuracy_score(y_test, y_pred)
plt.title("C= 0.1 with accuracy="+str(accuracy))
plot_decision_regions(np.asarray(X_test), np.asarray(y_test), clf=clf)
plt.show()
```

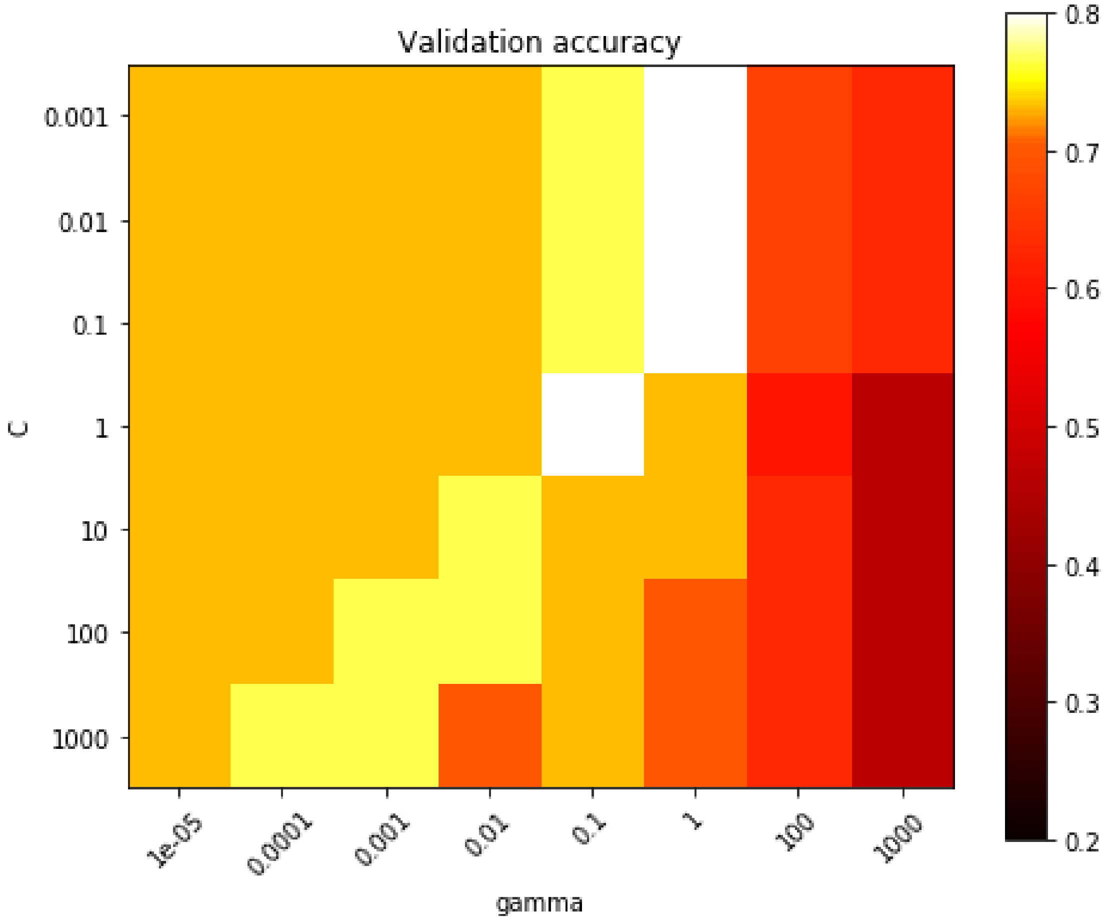


```
C = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
gamma = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 100, 1000]
max_score = 0.0
max_C = 0.0
max_gamma = 0.0
scores = []

for c in C:
    for igamma in gamma:
        clf = svm.SVC(kernel='rbf', gamma=igamma, C=c)
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_validation)
        accuracy = accuracy_score(y_validation, y_pred)
        scores.append(accuracy)
        #print("C: "+str(c)+" gamma: "+str(igamma)+" score: "+str(accuracy))
        if accuracy > max_score:
            max_score = accuracy
            max_C = c
            max_gamma = igamma

print()
print('Highest Accuracy Score: ', max_score)
print('Best C value: ', max_C)
print('Best gamma value: ', max_gamma)
```

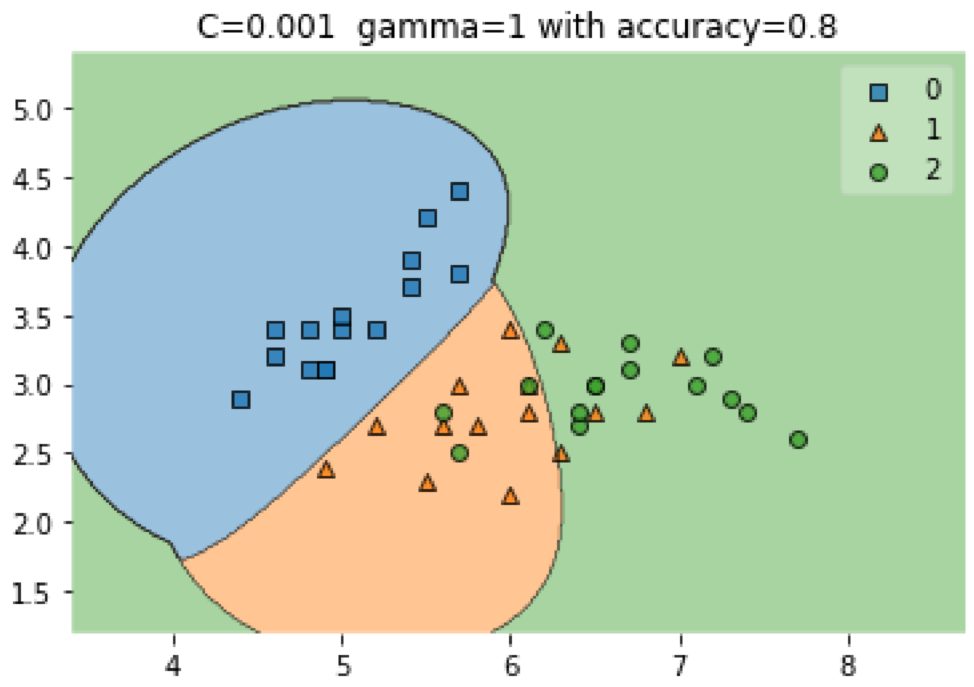
```
Highest Accuracy Score:  0.8
Best C value:  0.001
Best gamma value:  1
```

```
class MidpointNormalize(Normalize):

    def __init__(self, vmin=None, vmax=None, midpoint=None, clip=False):
        self.midpoint = midpoint
        Normalize.__init__(self, vmin, vmax, clip)

    def __call__(self, value, clip=None):
        x, y = [self.vmin, self.midpoint, self.vmax], [0, 0.5, 1]
        return np.ma.masked_array(np.interp(value, x, y))


C = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
gamma = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 100, 1000]
max_score = 0.0
max_C = 0.0
max_gamma = 0.0
scores = []

for c in C:
    for igamma in gamma:
        clf = svm.SVC(kernel='rbf', gamma=igamma, C=c)
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_validation)
        accuracy = accuracy_score(y_validation, y_pred)
        scores.append(accuracy)
        #print("C: "+str(c)+" gamma: "+str(igamma)+" score: "+str(accuracy))
        if accuracy > max_score:
            max_score = accuracy
            max_C = c
            max_gamma = igamma

scores = np.asarray(scores)
scores = scores.reshape(len(C), len(gamma))

print()
print('Highest Accuracy Score: ', max_score)
print('Best C value: ', max_C)
print('Best gamma value: ', max_gamma)

plt.figure(figsize=(8, 6))
plt.subplots_adjust(left=.2, right=0.95, bottom=0.15, top=0.95)
plt.imshow(scores, interpolation='nearest', cmap=plt.cm.hot,
           norm=MidpointNormalize(vmin=0.2, midpoint=scores.mean()))
plt.xlabel('gamma')
plt.ylabel('C')
plt.colorbar()
plt.xticks(np.arange(len(gamma)), gamma, rotation=45)
plt.yticks(np.arange(len(C)), C)
plt.title('Validation accuracy')
plt.show()
```

```
Highest Accuracy Score:  0.8
Best C value:  0.001
Best gamma value:  1
```



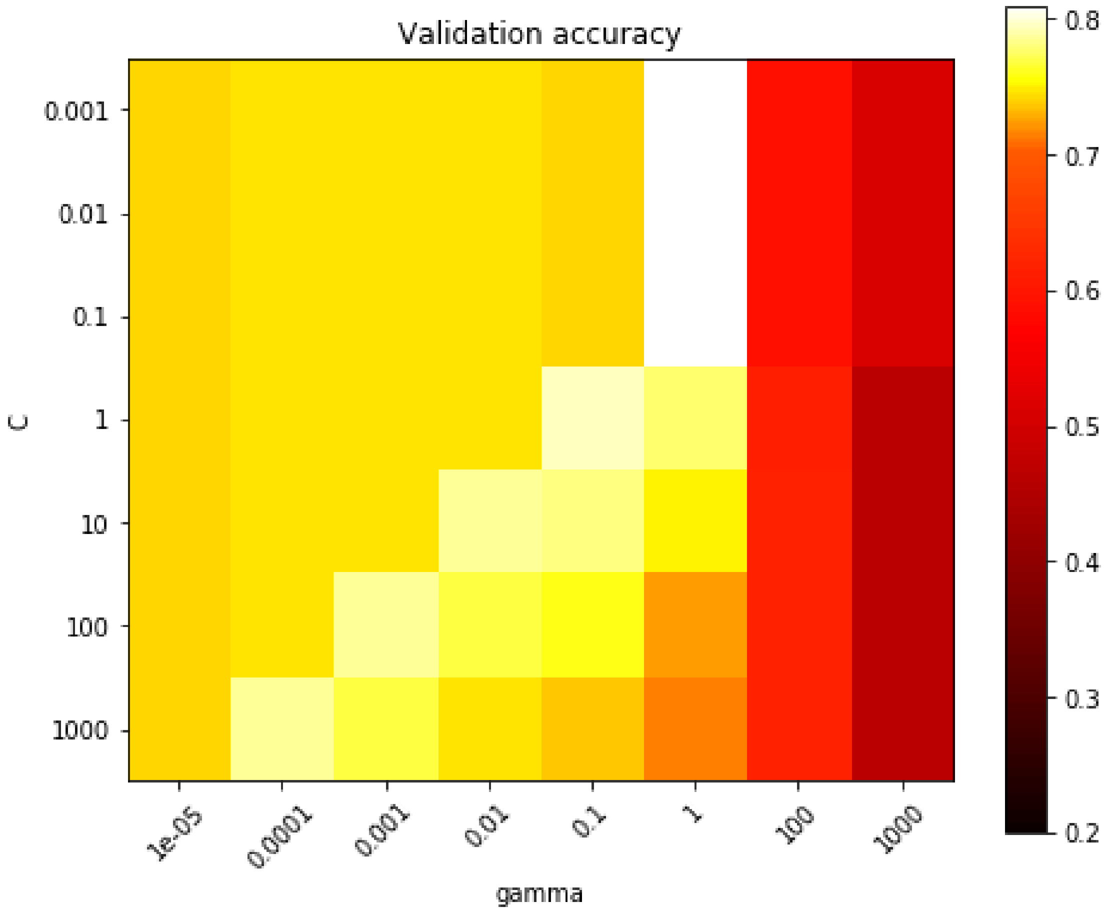
```
clf = svm.SVC(kernel='rbf', C=max_C, gamma=max_gamma)
clf.fit(X_train, y_train)
#y_pred = clf.predict(X_test)
accuracy = max_score
plt.title("C="+str(max_C)+ "  gamma="+str(max_gamma)+ " with accuracy="+str(accuracy))
plot_decision_regions(X_test, y_test, clf=clf)
plt.show()
```



```
class MidpointNormalize(Normalize):

    def __init__(self, vmin=None, vmax=None, midpoint=None, clip=False):
        self.midpoint = midpoint
        Normalize.__init__(self, vmin, vmax, clip)

    def __call__(self, value, clip=None):
        x, y = [self.vmin, self.midpoint, self.vmax], [0, 0.5, 1]
        return np.ma.masked_array(np.interp(value, x, y))

C = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
gamma = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 100, 1000]
cv = StratifiedShuffleSplit(n_splits=5, test_size=0.3, random_state=1)

hyper_parameters = [{'kernel': ['rbf'], 'gamma': gamma,
                     'C': C},]

clf = GridSearchCV(SVC(), hyper_parameters, cv = cv)
clf.fit(X, y)

print()
print("Highest Accuracy score: ", clf.best_score_)
print('Best C value: ', clf.best_params_['C'])
print('Best gamma value: ', clf.best_params_['gamma'])

means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
#table = pandas.DataFrame(columns=C, index=gamma)
```

```
scores = clf.cv_results_['mean_test_score'].reshape(len(C),len(gamma))

plt.figure(figsize=(8, 6))
plt.subplots_adjust(left=.2, right=0.95, bottom=0.15, top=0.95)
plt.imshow(scores, interpolation='nearest', cmap=plt.cm.hot,
           norm=MidpointNormalize(vmin=0.2, midpoint=scores.mean()))
plt.xlabel('gamma')
plt.ylabel('C')
plt.colorbar()
plt.xticks(np.arange(len(gamma)), gamma, rotation=45)
plt.yticks(np.arange(len(C)), C)
plt.title('Validation accuracy')
plt.show()
```

```
Highest Accuracy score:  0.8088888888888889
Best C value:  0.001
Best gamma value:  1
```



The k cross validation is used to divide the training set into k distinct subsets. Then every subset is used for training and others $k-1$ are used for validation in the entire trainging phase. This is done for the better training of the classification task.

The motivation to use cross validation techniques is that when we fit a model, we are fitting it to a training dataset. Without cross validation we only have information on how does our model perform to our in-sample data. Ideally we would like to see how does the model perform when we have a new data in terms of accuracy of its predictions.

We can see from the graphs above how the accuracy change.