POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Corso di Laurea Magistrale in Ingegneria Informatica

Machine Learning and Artificial Intelligence

Homework 1: K-NN and SVM

Student:

Pietro Basci

266004

ACADEMIC YEAR 2019/2020

Introduction

The purpose of this homework is to build few classifiers, tune some hyper parameters and evaluate the performance changes. More specifically, it will be considered firstly the k-Nearest Neighbors classifier and then the Support Vector Machine with two kind of kernels: linear and rbf. The dataset considered is the wine dataset available in the scikit-learn library, of which we consider only the fist two features to more easy present graphically the data.

K-Nearest Neighbors

After loading the *wine dataset* from the *sckit-learn.datasets* module, and once selected the first two features for a 2D representation of data, the dataset has been randomly split into three sets: Train set (50% of data), Validation set (20% of data) and Test set (the last 30%). In this way, models are trained on the train set, evaluated on the validation set and then, the final evaluation can be done on the test set. This operation is important in order to avoid overfitting problems. In this report a random seed for the split has been set in order to make the experiment repeatable. To obtain different random sets at each iteration it is possible to set the *random_state* parameter to '*None*'. The code used to do this is reported below.

```
TRAIN_SIZE = 0.70
TEST_SIZE = 0.30
VALIDATION_SIZE = 0.20

# Load the Wine dataset - X: features, y: labels
X, y = load_wine(True)
# Select the first two features
X = X[:, 0:2]

# Split the whole dataset into train set (70%) and test set (30%)
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=TEST_SIZE,
                                                    random_state=42,
                                                    shuffle=True)
```

After dividing training set and test set, a feature scaling with Z-score normalization has been performed on the data using the *StandardScaler* provided by sckit-learn. In particular, it standardizes features by removing the mean and scaling to unit variance. This operation is important since it is possible that features are measured in different units.

The standard score of a sample x is calculated as:

z = (x - u) / s

where u and s are the mean and the standard deviation of the training samples respectively.

The fit method has been called on the train set in order to compute mean and the standard deviation on this set, and then use these values for later scaling.

```
# Standardize data using the StandardScaler()
scaler = StandardScaler().fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

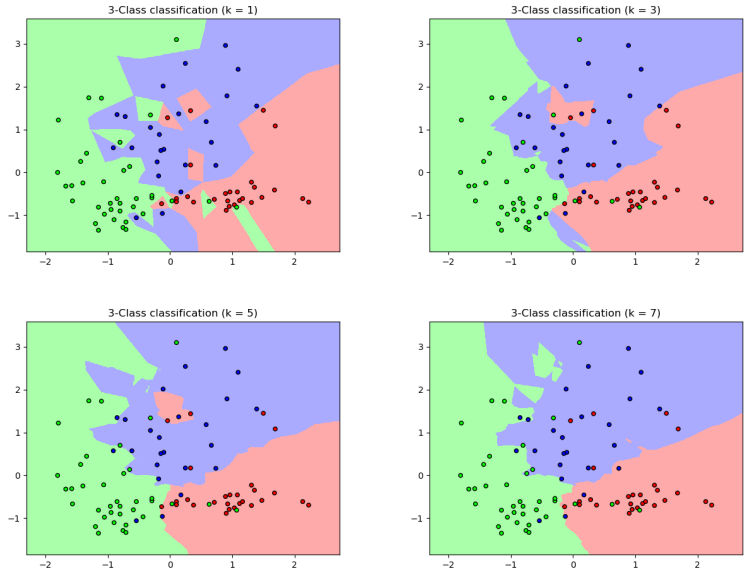
Finally, the train set has been further split into a new train set and validation set.

```
# Split the train set (70%) into new train set (50%) and validation set (20%)
X_train, X_validation, y_train, y_validation = train_test_split(X_train, y_train,
                                                                test_size=VALIDATION_SIZE/TRAIN_SIZE,
                                                                random_state=42,
                                                                shuffle=True)
```

Now, the K-NN algorithm has been fitted on the train set considering different values of K and evaluated on the validation set. In particular it has been tested for K = [1, 3, 5, 7] and for each of them, the resulting decision boundaries have been plotted.

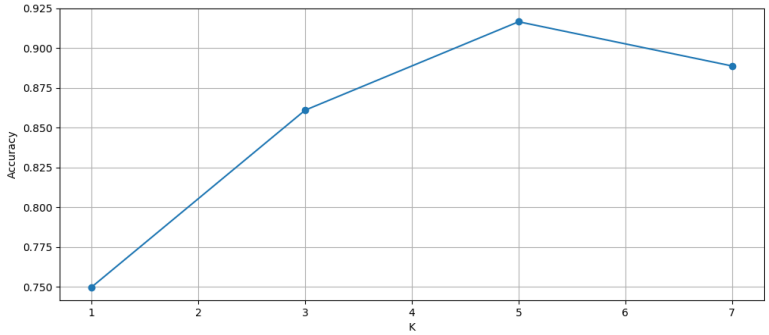
```
K = [1, 3, 5, 7]
accuracies = []
maxAccuracy = -1

for k_value in K:
    # Create K-Nearest Neighbors Classifier
    neigh = KNeighborsClassifier(n_neighbors=k_value)
    # Train the model on 50% of inizial dataset
    neigh.fit(X_train, y_train)

    # Plot data and decision boundaries
    myplt.knn_plot_data_boudaries(X_train, y_train, neigh, k_value,
                                  folder='K-NN_plots')
    # Evaluate the model on the validation set
    accuracy = neigh.score(X_validation, y_validation)
    accuracies.append(accuracy)

    # Update the max accuracy
    if accuracy > maxAccuracy:
        maxAccuracy = accuracy

    print("Accuracy = %.2f, obtained using K=%i" % (accuracy, k_value))


# Plot accuracy on K changes
myplt.knn_accuracy_plot(K, accuracies, folder='K-NN_plots')
```

The K parameter defines the number of nearest neighbors that must be considered in order to classify a new element. Specifically, the K-NN classifier will assign the most popular label among the k nearest neighbors to the new observation. The resulting decision boundaries with the respective scatter plot are shown in the following plots.



According to the figures, as the value of K grows the boundaries tend to become smoother. This is due to the fact that the nearest neighbor algorithm is particularly sensitive to outliers so that a single mislabeled example dramatically changes the boundary. Hence, using a value of K too low led the model to overfit and classification could be affected by outliers, while with a large value the model tend to underfit and everything tend to be classified as the most probable class.

In the graph below is reported the trend of accuracy in correspondence of the variation of K.
The accuracy reaches the peak with 92% using K = 5.



Once found the best K value, it has been used to train a new model using the 70% of initial data (i.e. train set + validation set) and then evaluated on the test set.

```
#Get the K with the best accuracy
best_K = K[accuracies.index(maxAccuracy)]
print("Best accuracy (%.2f) obtained using K=%i" %(maxAccuracy, best_K))

# Create K-Nearest Neighbors Classifier
neigh = KNeighborsClassifier(n_neighbors=best_K)
# Train the model on 70% of inizial dataset (train + validation)
neigh.fit(np.concatenate((X_train, X_validation)),
          np.concatenate((y_train, y_validation)))
# Evaluate the model on the test set
accuracy = neigh.score(X_test, y_test)
print("Accuracy on Test Set = %.2f, obtained using K=%i" %(accuracy, best_K))
```

Output:

```
Accuracy = 0.75, obtained using K=1
Accuracy = 0.86, obtained using K=3
Accuracy = 0.92, obtained using K=5
Accuracy = 0.89, obtained using K=7
Best accuracy (0.92) obtained using K=5
Accuracy on Test Set = 0.76, obtained using K=5
```

The accuracy obtained by the evaluation on the test set is significantly lower compared to the one achieved on the validation set. This can be explained by the method used to validate the model: partitioning the dataset into three sets, drastically reduce the number of samples which can be used for learning the model and moreover the results depend on a particular random choice for the pair (train, validation) set. Therefore, we can choose a value of K that seems quite good on the validation set, but it is not on other data. To solve this problem, another technique should be considered: the K-Fold Cross Validation that will be used in the last part of this homework. However, also in this case performing a test using the cross validation allows to get more realistic values of accuracy that are more close the the accuracy obtained on the test set. These are the values of accuracy returned using the 5-Fold Cross Validation.

Output:

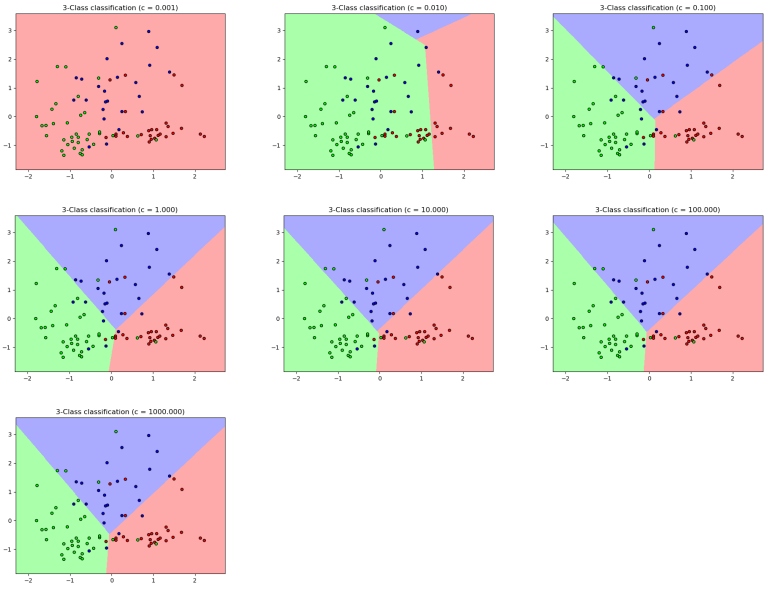
```
Accuracy = 0.70, obtained using K=1
Accuracy = 0.75, obtained using K=3
Accuracy = 0.73, obtained using K=5
Accuracy = 0.74, obtained using K=7
Best accuracy (0.75) obtained using K=3
Accuracy on Test Set = 0.80, obtained using K=3
```

Linear SVM

Considering the same splits of dataset, for each different value of C, a linear Support Vector Machine has been trained on the Train set (50% of initial data) and evaluated on the validation set. More specifically it has been tested for values of C in [0.001, 0.01, 0.1, 1, 10, 100, 1000] and for each of them, the resulting decision boundaries have been plotted.

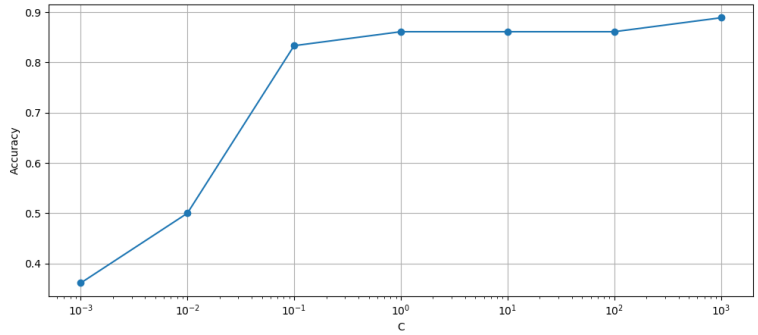
```
C = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
accuracies = []
maxAccuracy = -1

for c_value in C:
    # Create SVM Classifier with linear kernel
    linearSVM = svm.SVC(kernel='linear', C=c_value)
    # Train the model on 50% of inizial dataset
    linearSVM.fit(X_train, y_train)

    # Plot data and decision boundaries
    myplt.svm_plot_data_boudaries(X_train, y_train, linearSVM, c_value,
                                  folder='Linear_SVM_plots',
                                  fileName='LinearSVM classification')
    # Evaluate the model on the validation set
    accuracy = linearSVM.score(X_validation, y_validation)
    accuracies.append(accuracy)

    # Update the max accuracy
    if accuracy > maxAccuracy:
        maxAccuracy = accuracy

    print("Accuracy = %.2f, obtained using C=%f" % (accuracy, c_value))


# Plot accuracy on C changes
myplt.svm_accuracy_plot(C, accuracies, folder='Linear_SVM_plots')
```

In scikit-learn, the parameter C of the SVC learner is the penalty for misclassifying a data point. It allows to specify the degree of tolerance we want to give when finding the decision boundary for the SVM. The bigger the C, the more penalty SVM gets when it makes misclassification and therefore, we will obtain narrow margins that are rarely violated. On the other hand, when C is small, the margin will be wider and we allow more violations to it. The resulting decision boundaries with the relative scatter plot are shown in the following figures.



As shown above, when C is high we obtain a classifier that is highly fit to the data. As the value of C decrease, the margin become wider and this lead to a classifier that is less and less fit to the data.

In the chart below is reported the trend of accuracy in correspondence of the variation of C. As we can see the accuracy grows for higher value of C and hence with small margin SVM.



Once found the best C value, it has been used to train a new model using the 70% of initial data (i.e. train set + validation set) and then evaluated on the test set.

```
#Get the C with the best accuracy
best_C = C[accuracies.index(maxAccuracy)]
print("Best accuracy (%.2f) obtained using C=%f" % (maxAccuracy, best_C))

# Create SVM Classifier
linearSVM = svm.SVC(kernel='linear', C=c_value)
# Train the model on 70% of inizial dataset (train + validation)
linearSVM.fit(np.concatenate((X_train, X_validation)),
              np.concatenate((y_train, y_validation)))
# Evaluate the model on the test set
accuracy = linearSVM.score(X_test, y_test)
print("Accuracy on Test Set = %.2f, obtained using C=%f" % (accuracy, best_C))
```

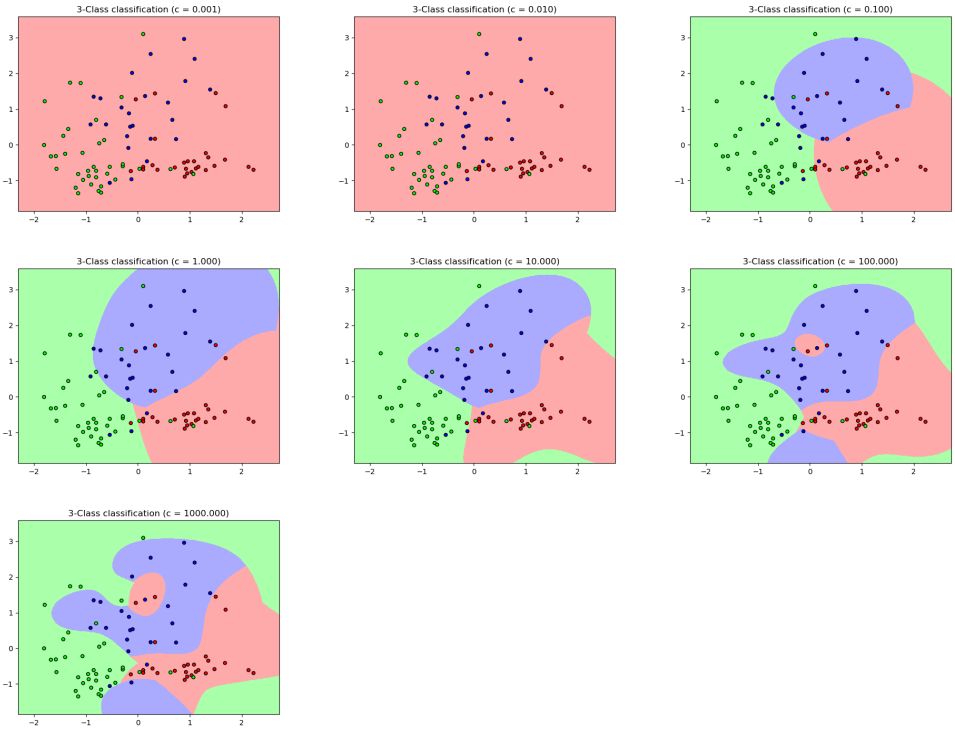
Output:

```
Accuracy = 0.36, obtained using C=0.001000
Accuracy = 0.50, obtained using C=0.010000
Accuracy = 0.83, obtained using C=0.100000
Accuracy = 0.86, obtained using C=1.000000
Accuracy = 0.86, obtained using C=10.000000
Accuracy = 0.86, obtained using C=100.000000
Accuracy = 0.89, obtained using C=1000.000000
Best accuracy (0.89) obtained using C=1000.000000
Accuracy on Test Set = 0.74, obtained using C=1000.000000
```

Also in this case, the accuracy obtained by the evaluation on the test set is significantly lower compared to the one achieved on the validation set. And the reasons are the same of the previous experiment: the technique used to validate the model.

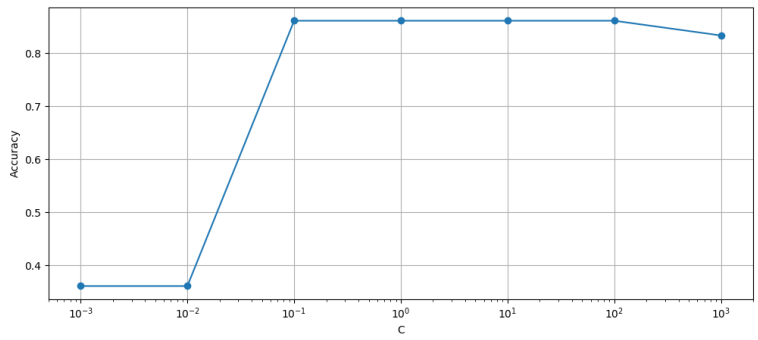
SVM with RBF Kernel

In this section, a similar experiment with SVM has been performed but this time using the Radial Basis Function kernel instead of the linear one. The resulting decision boundaries for different value of C are illustrated in the following plots.



The main difference with respect to the previous are the non-linear boundaries between the classes. Also in this case the parameter C allows to specify the degree of tolerance we want to give when finding the decision boundary: for larger value of C, a smaller margin will be accepted, a lower C will encourage a larger margin.

As in the previous case, a graph with the accuracy trend with respect to C is generated. In this case, the value of the accuracy is the same (0,86) for 4 different values of C. Under the same accuracy the lowest value of C has been preferred in order to ensure a larger margin.



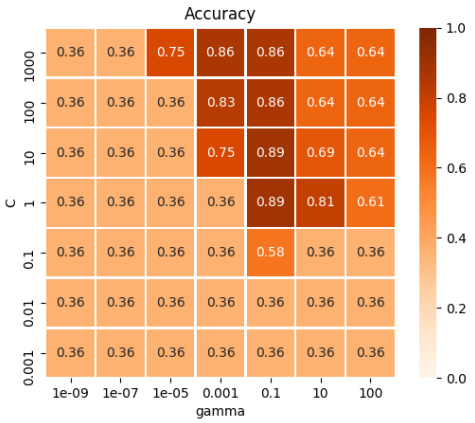
Evaluating the model trained on the 70% of initial data (i.e. train set + validation set) with the best value of C, the accuracy is only of 0.78 witch is quite far from the 0,86 achieved on the validation set.

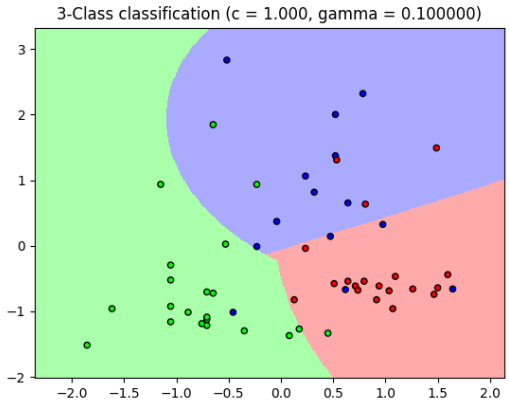
Grid Search

Now, a grid search has been performed in order to tune both gamma and C parameters. After selecting an appropriate range of both values, a new model has been trained and evaluated for each pair of values and the resulting accuracies have been reported in a heatmap as a function of C and gamma.

```
Gamma = [0.000000001, 0.0000001, 0.00001, 0.001, 0.1, 10, 100]
C = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
accuracies = []
maxAccuracy = -1
best_g = -1
best_C = -1

for g_value in Gamma:
    g_accuracy = []
    for c_value in C:
        # Create SVM Classifier with RBF kernel
        rbfSVM = svm.SVC(kernel='rbf', gamma=g_value, C=c_value)
        # Train the model on 50% of inizial dataset
        rbfSVM.fit(X_train, y_train)
        # Evaluate the model on the validation set
        accuracy = rbfSVM.score(X_validation, y_validation)
        g_accuracy.append(accuracy)

        # Update the max accuracy and best parameters
        if accuracy > maxAccuracy:
            maxAccuracy = accuracy
            best_g = g_value
            best_C = c_value

    accuracies.append(g_accuracy)


# Plot accuracy on Gamma and C changes
myplt.svm_accuracy_grid(Gamma, C, accuracies, folder='Rbf_SVM_plots')
print("Best accuracy (%.2f) obtained using gamma=%f, C=%f" % (maxAccuracy, best_g, best_C))
```

The gamma parameter controls the influence of a single training example on the decision boundary. The higher the gamma, the more influence it will have on the decision boundary, more wiggling the boundary will be. When gamma is low, the ‘curve’ of the decision boundary is very low and thus the decision region is very broad. When gamma is high, the ‘curve’ of the decision boundary is high, which creates islands of decision-boundaries around data points. Also in this case, under the same accuracy the lowest value of C has been preferred.





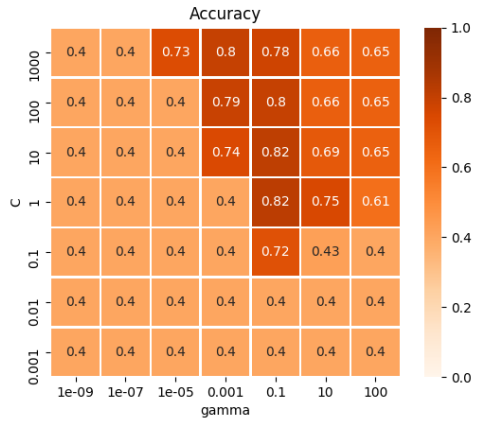
Once selected the best pair that gives the highest accuracy on the validation set, a new model has been trained on the 70% of data (i.e. train set + validation set) and than evaluated on the test set. Also in this case the value of accuracy obtained (0.80) is worse with respect to the one found on the validation test (0.89 using C=1, gamma=0.1). The decision boundaries for the best couple of parameters and the scatter plot of the test set are plotted on the right figure.

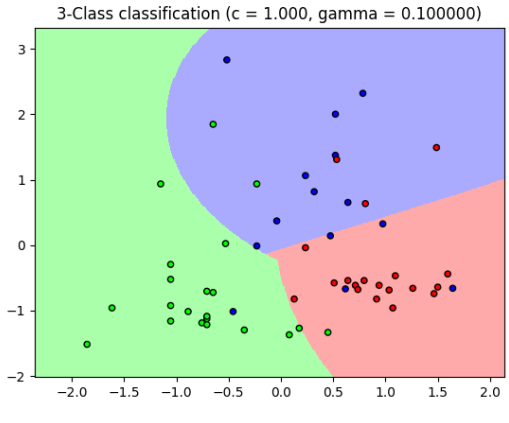
K-Fold Cross Validation

In this section, the K-Fold Cross Validation will be used as method to validate the model. After merging the training and validation split so that the training data is composed by 70% of the initial data, as in the previous case, a grid search for gamma and C is performed but this time using the 5-Fold Cross Validation and then evaluated on the test set.

```
Gamma = [0.000000001, 0.0000001, 0.00001, 0.001, 0.1, 10, 100]
C = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
accuracies = []
maxAccuracy = -1
best_g = -1
best_C = -1

for g_value in Gamma:
    g_accuracy = []
    for c_value in C:
        # Create SVM Classifier
        rbfSVM = svm.SVC(kernel='rbf', gamma=g_value, C=c_value)
        # Evaluate the model using 5-Fold cross validation
        scores = cross_val_score(rbfSVM, X_train, y_train, cv=5)
        accuracy = scores.mean()
        g_accuracy.append(accuracy)

        # Update the max accuracy and best parameters
        if accuracy > maxAccuracy:
            maxAccuracy = accuracy
            best_g = g_value
            best_C = c_value

    accuracies.append(g_accuracy)


# Plot accuracy on Gamma and C changes
myplt.svm_accuracy_grid(Gamma, C, accuracies, folder='Rbf_SVM_plots/Rbf_SVM_cv_plots')
print("Best accuracy (%.2f) obtained using gamma=%f, C=%f" % (maxAccuracy, best_g, best_C))
```

Using the 5-Fold Cross Validation, the training set is divided into 5 sets and for each of the 5 sets, a model is trained using 4 sets as training data and the accuracy of the resulting model is evaluated on the remaining set. The final value of the accuracy is the average of the values computed for each of the 5 splits. Also in this case, under the same accuracy the lowest value of C has been preferred.





The value of accuracy obtained on the test set using the best pair of C and gamma, is 0.80 which is only 2% less than the one achieved during the validation (0.82 using C=1, gamma=0.1). Repeating the experiment using different randomly generated splits for the training and test sets, the value of accuracy will remain almost the same. This is due to the fact that using the cross validation we are using much more data to evaluate the model and so we obtain a value that is closer to the performance on real data.

Differences between K-NN and SVM

The K-Nearest Neighbors is a very simple algorithm which is based on the idea that an element can be classified according to his neighbors. The principle behind this method indeed is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number K of samples is defined by the user while the distance can be any metric measure such as the standard Euclidean distance.

The idea behind an SVM is that it is possible to construct a hyperplane that separates the training data according to their class labels, and than classify a new element according to which side of the hyperplane it is located. The SVM is a generalization of the maximal margin classifier that choose the maximal margin hyperplane which is the separating hyperplane that is farthest from the training observations. However, in many cases data can belong to two classes that are not separable by a hyperplane, and so there is no maximal margin classifier. So, a soft margin approach must be considered: find an hyperplane that 'almost' separates the classes, allowing some observations to be on the incorrect side of the margin, or even the incorrect side of the hyperplane. This remove the effect that a single observation can have on a maximal margin hyperplane and that can lead to a tiny margin. When classes are not-linearly separable, it is possible to enlarge the feature space, in order to accommodate a non-linear boundary between the classes, using kernel functions such as the Radial Basis Function.

The K-NN classifiers are 'memory based': they do not attempt to construct a general internal model, but simply store instances of the training data. So, when a new observation has to be classified, the distances between this element and those in the train data have to be computed and then it is classified using a simple majority vote among the K nearest neighbors. Therefore, it can be inefficient when the train set is very huge and so inconvenient in real time applications where it is very important to have a prediction as fast as possible. Conversely, an SVM is characterized by a longer training time but it can ensure a faster prediction.

The nearest neighbor algorithm is particularly sensitive to outliers so that a single mislabeled example dramatically changes the boundary. Indeed, if the value of K is very low, the model tend to overfit and classification could be affected by outliers, while if is too large the model tend to underfit and everything tend to be classified as the most probable class.

Unlike the k-NN which is generally used as multi-class classifier, standard SVM are suitable for binary classification. This limit can be overcome using a One-Verus-One or a One-Versus-All approach. With the first, an SVM for each pair of classes is built and the final classification is performed by assigning the element to the class which was most frequently assigned by all classifiers. In the second case, an SVM for each class is built which compare this class to the remaining classes.

Despite its simplicity, nearest neighbors has been successful in a large number of classification: it is often successful in classification situations where the decision boundary is very irregular.

PCA

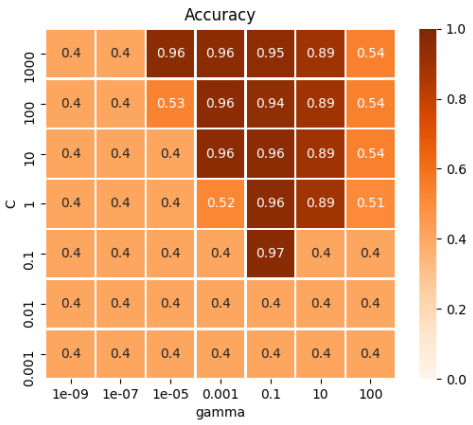
In this section, the experiment has been repeated considering another pair of attribute. In particular, the couple considered is composed by the first two principal components obtained using the PCA.

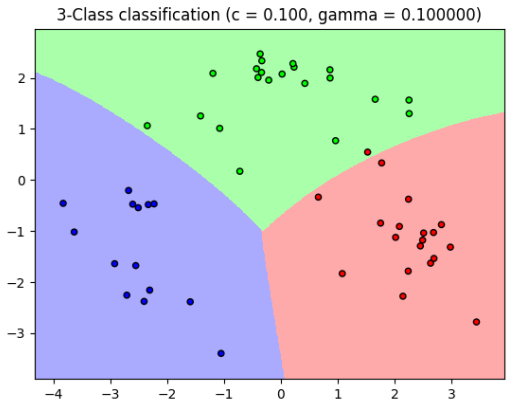
```
TRAIN_SIZE = 0.70
TEST_SIZE = 0.30

# Load the Wine dataset - X: features, y: labels
X, y = load_wine(True)

# Split the whole dataset into train set (70%) and test set (30%)
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=TEST_SIZE,
                                                    random_state=42,
                                                    shuffle=True)

# Standardize data using the StandardScaler()
scaler = StandardScaler().fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Select the first two principal components
pca = PCA(n_components=2)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

After that, as in the previous case, a grid search using the 5-Fold Cross Validation as validation method has been performed in order to find the best values of C and gamma. Once found the best pair, a new model has been trained on the 70% of data (i.e. train set + validation set) and than evaluated on the test set. The accuracy obtained are the following:

```
Best accuracy (0.97) obtained using gamma=0.100000, C=0.100000
Accuracy on Test Set = 0.98, obtained using gamma=0.100000, C=0.100000
```

With this approach the model achieve a very high accuracy with the cross validation which in this case is even worse with respect to the one reached the test set.





In the right figure are illustrated the decision boundaries and the scatter plot of the test set. As we can see, almost all the samples are in the correct region.