Daniele Genta, s276922
Machine Learning and Deep Learning
Homework 1 – Report
Academic Year: 2019-2020

0 - Introduction

The goal of the homework is to exploit the Python implementation of several machine learning algorithms, studied during the lectures, to perform a classification task. So, we are operating in a supervised learning environment and in particular we're dealing with a multi-class classification problem.
In order to perform this task, we use the **Scikit** library and in particular the built-in dataset Wine.

1 - Data exploration

The first step of my pipeline was to import the required libraries and load the Wine dataset, then I used the following basic data exploration commands in order to become familiar with it.

```
# data exploration
print(data.head())
print(data.describe())
```

So, we're working on a dataset that details some specifics about wines, to be more precise 13 attributes and a target column are present, 178 rows are available (small dataset).
We're dealing with a multi-class classification problem since the rows describe three different classes.
Inspecting the dataset using the **Pandas** library, we can see that there're no missing values and no further preprocessing is needed, yet the three classes (represented by the "target column") are not perfectly balanced, as we can see from the following histogram (**fig. 1**).

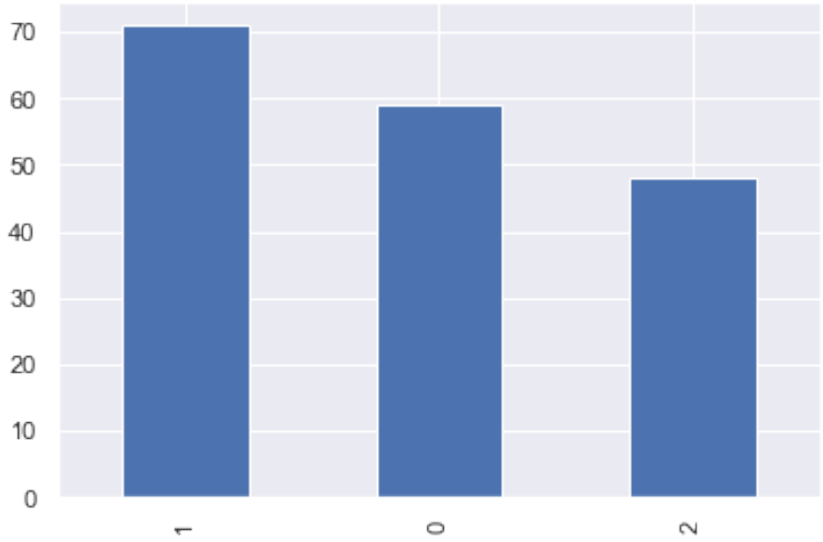


***Fig. 1**: Column "target" distribution*

The following step is to select just the first two attributes of the dataset, those are "*alcohol*" and "*malic_acid*" and plot a 2d representation of them.
This reduced version of the dataset will be the one used throughout the entire homework.

In order to provide a two-dimensional representation of the reduced dataset, we exploit the **Seaborn** library, in particular the following piece of code:

```
represented_data = data[['alcohol', 'malic_acid', 'target']]
ax = sns.scatterplot(x="alcohol", y="malic_acid",
data=represented_data, hue = 'target')
```

Which yields the following result (**fig. 2**).

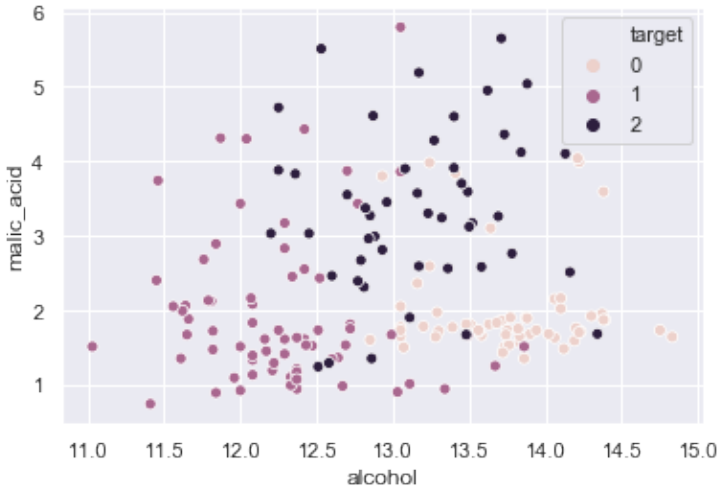


***Fig.2**: 2d representation of the dataset*

The above plot shows that the three classes are quite separated, yet some outliers are present.
The data is split according to the following proportion: 50/20/30 into respectively: training, validation and test set, operatively this is done by invoking two times the *train_test_split* method.
So, to be more specific: half the dataset will be used to build the model, 20% of it to test its performances and tune optimally its hyper-parameters while the remaining 30% will be used to effectively assess the accuracy of the model.
It is important to point out that, with respect to the split of the data chosen, the results could change drastically. As for this homework, I've set a random seed equal to 20 and I have carried out all the analysis maintaining the same coherent split.
Finally, it's important to consider that the data used are not normalized (i.e the different features might have a very different scale of values), in order to perform normalization **Sklearn** an easy to implement class, called *StandardScaler*. However, in this homework, I have not implemented such methodology.

2 – K nearest Neighbors

After having randomly split the dataset into train, test and validation (proportion 50:30:20), we apply the KNN classifier.
Using the *NearestNeighbor* class, provided by **Scikit** library, we have to set the parameter K, which is the number of neighbors that the classifier considers in order to decide the class for each given unlabeled object, the label will be assigned as the most recurrent class among the neighbors.
The procedure has been repeated for multiple values of K: 1, 3, 5 and 7.
The resulting data and decision boundaries are the following.

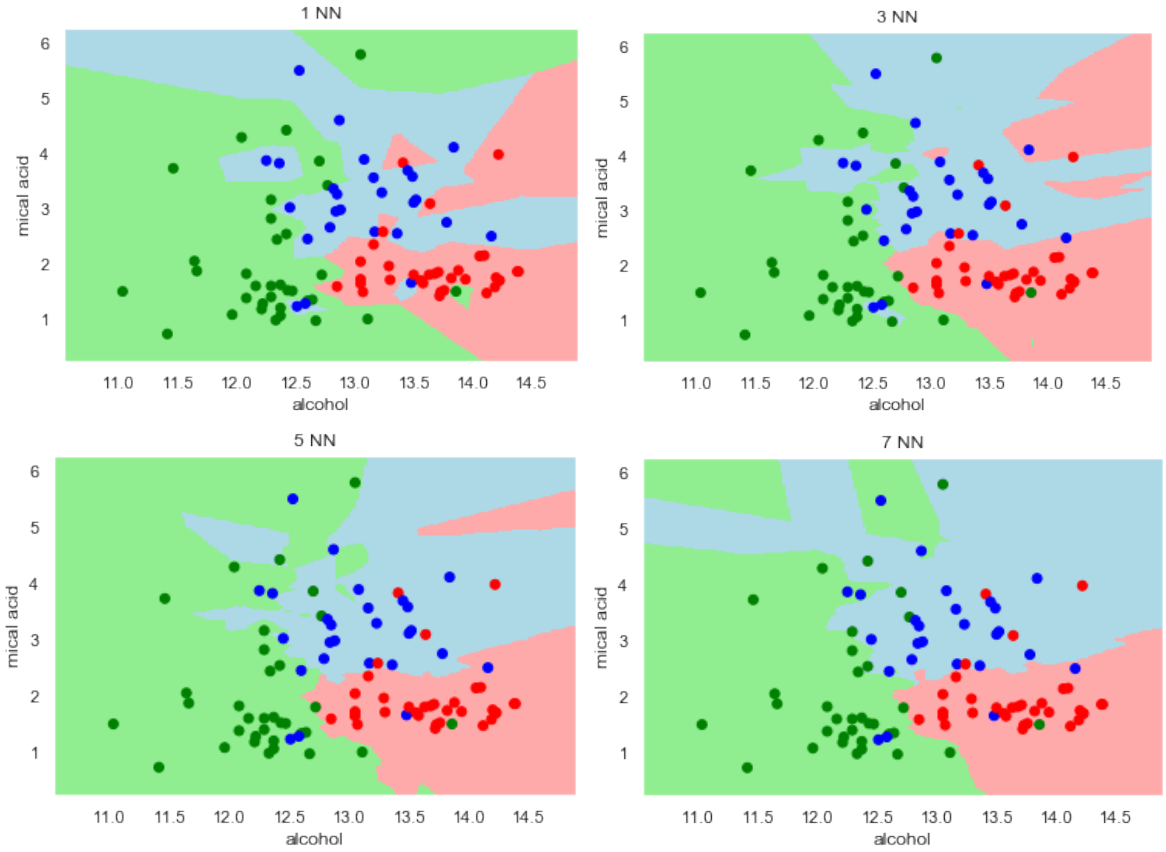


Fig. 3: *KNN decision boundaries using different values for K*

At each step the model is evaluated on the validation set by means of the **accuracy score**, which represents the proportion of correctly classified points with respect to the total number of points.
As a side note: this metric will be used also in the other sections of the homework.
Changing the number of nearest neighbors (K parameter) implies a change in the decision boundaries. What we can see from the previous plots is that: a low number of K results in a classifier which is very sensible to outliers, while if the number grows too much there could be a mis-classification problem (we risk including point of other classes).
After having collected all the accuracies corresponding to the different settings of the parameter K, I have plotted the accuracy against the number of neighbors (K), as it is shown by the following figure.

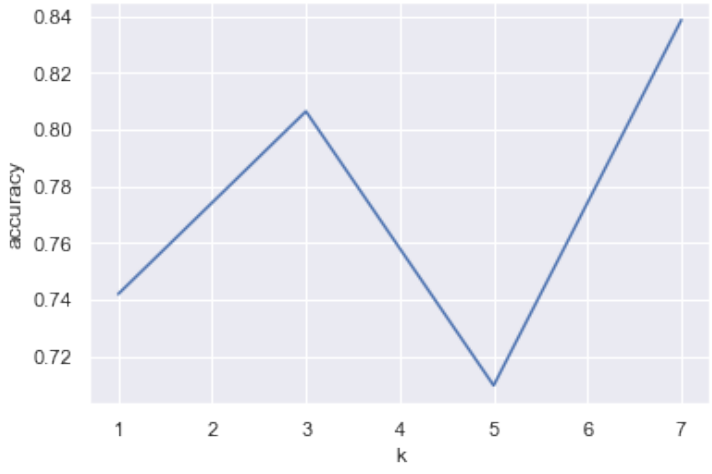


Fig. 4*: Accuracy against K (number of nearest neighbors)*

Analyzing the previous plot, we can see that the best score is reached when K = 7 and, using the following piece of code, we can evaluate the model on the test set by selecting the hyper-parameter K corresponding to the best score.

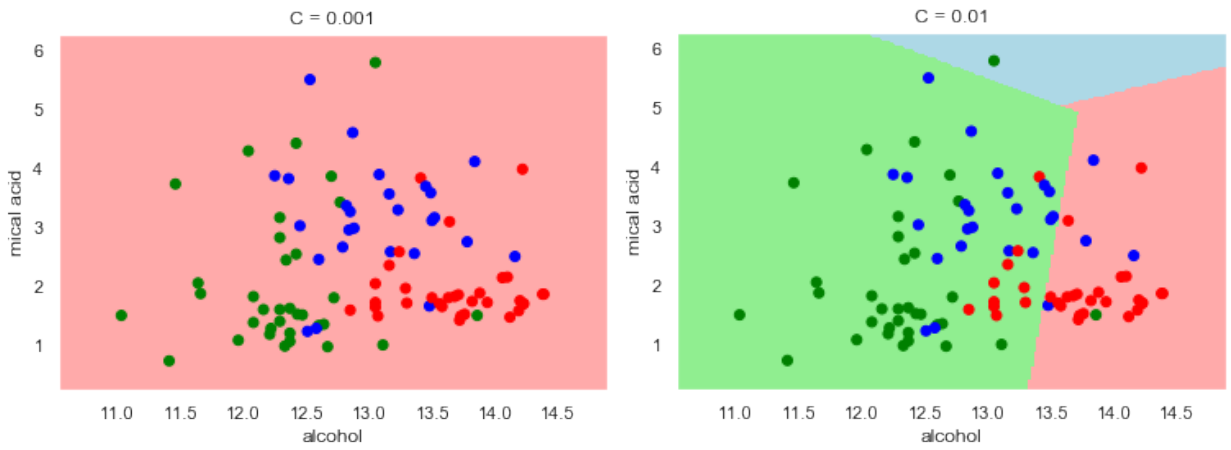
```
best_k = acc_df.loc[acc_df['accuracy'] ==
acc_df['accuracy'].max(), 'k'].iloc[0]
neigh = KNeighborsClassifier(n_neighbors = best_k)
neigh.fit(X_train, y_train)
y_pred = neigh.predict(X_test)
test_acc= metrics.accuracy_score(y_test, y_pred)
print(test_acc)
```

```
> Result: 0.8333333333333334
```

We can say that we're overall satisfied with this result since, as shown in the first plot (**fig. 1**), some outliers are present among the different classes and so an accuracy above 80% may be considered as a success.
Since the peak accuracy on the validation set was somehow similar to the one achieved on the test set, we could say that our model is not over-fitting, yet it is able to generalize.
Finally, it is important to keep in mind that KNN could become very onerous if the dataset is large or the number of features is high, whereas in this specific problem KNN was a good choice given the small dataset.
The next step will be to consider the same classification problem by using different methodologies, in particular Support Vector Machines (SVM).

3 – Linear SVM

Now we want to solve the same classification problem by using SVM, in particular a linear kernel.
The methodology being used is similar to before, but this time we had to set the hyper-parameter C properly. C represents in SVM the regularization parameter. The strength of the regularization is inversely proportional to C, which must be positive.
Again, we trained our model by using multiple C values (0.0001, 0.01, 0.1, 1, 10, 100 and 1000) and validated it in order to tune the hyper-parameter C optimally.
The resulting plots and decision boundaries are reported below (**fig. 5**).



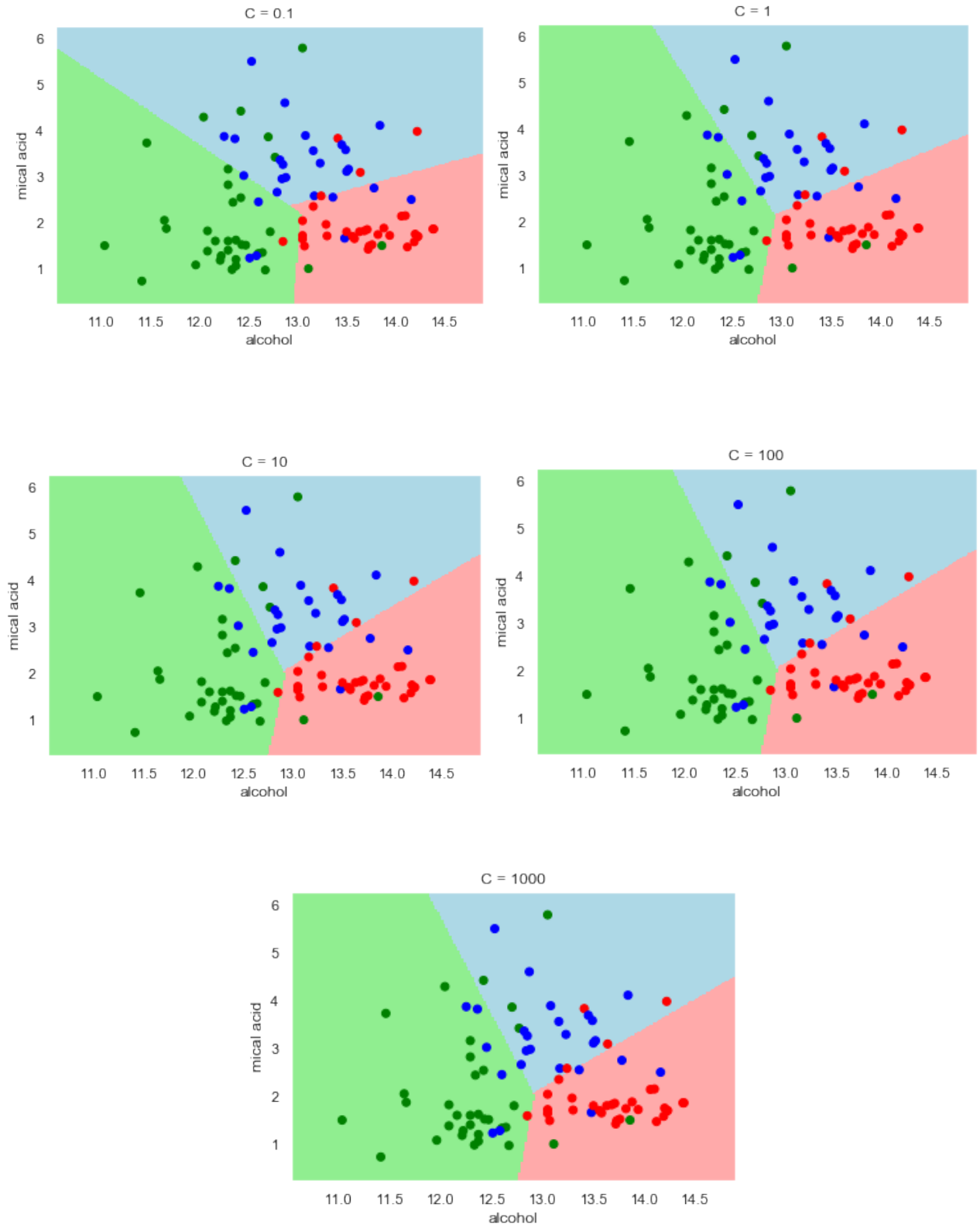


Fig. 5: *Linear SVM decision boundaries using different values of C*

A first insight from the plots may be that large C's train a better model with respect to very small values of C. Let's dig into it to understand what happens.

Firstly, SVM is a tool to achieve the following goals: find the largest minimum margin separating hyperplane and correctly separate as many instances as possible.

The meaning of the parameter C is to choose, based on each problem specification, the optimal tradeoff between the two.
Qualitatively, it may be asserted that the C parameter represents how strongly we want to avoid misclassification in each training sample. That means that a large C will result in softer margin hyperplanes (the classifier is strongly penalized for misclassified data).
On the other hand, a very small value of C will cause the optimizer to try to find a larger margin separating hyperplane, even if the hyperplane misclassifies some points. That explains why using a value of C which is very tiny, we end up with all the point classified into one class.
Using small values of C's may result in many misclassified examples, even though the training set is linearly separable. However, if C is too large, we may result in having a model which is not able to generalize, since it tries very hard to classify each training sample correctly (overfitting).
That explains why the decision boundaries change in the previous plots and how.

In order to select the best C to train the model in this problem, we plot the accuracy score achieved on the validation set, against the different C values and the result is the following (**fig. 6**).
The result reports the value of C that somehow achieves the better tradeoff between cost and misclassification.

Note: a log scale on the x axis is used due to graphical and interpretation result.

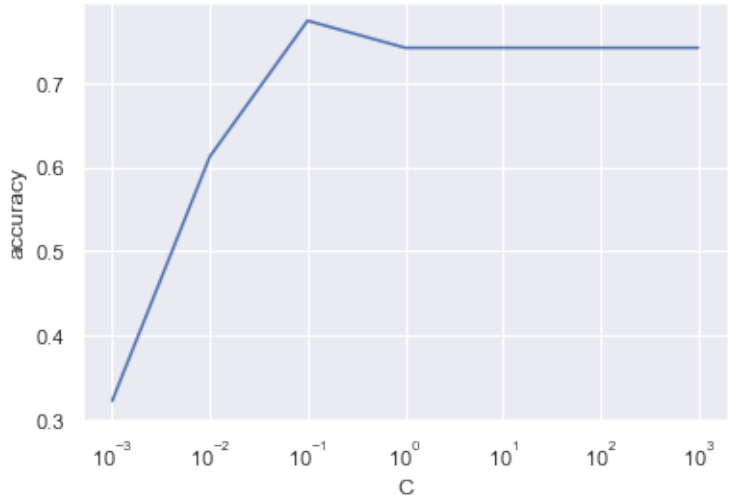


***Fig. 6**: Accuracy against C (Linear SVM)*

As in the previous section, the next step is to extract the best value of C, which corresponds to 0.1 in this case, and evaluate the corresponding model on the test set.
The chosen score metric is again the accuracy.

```
best_c = acc_df.loc[acc_df['accuracy'] == acc_df['accuracy'].max(),
'C'].iloc[0]
clf = SVC(C = best_c, random_state=0, kernel = "linear")
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

local_acc= metrics.accuracy_score(y_test, y_pred)
local_acc
```

> **Result: 0.8148148148148148**

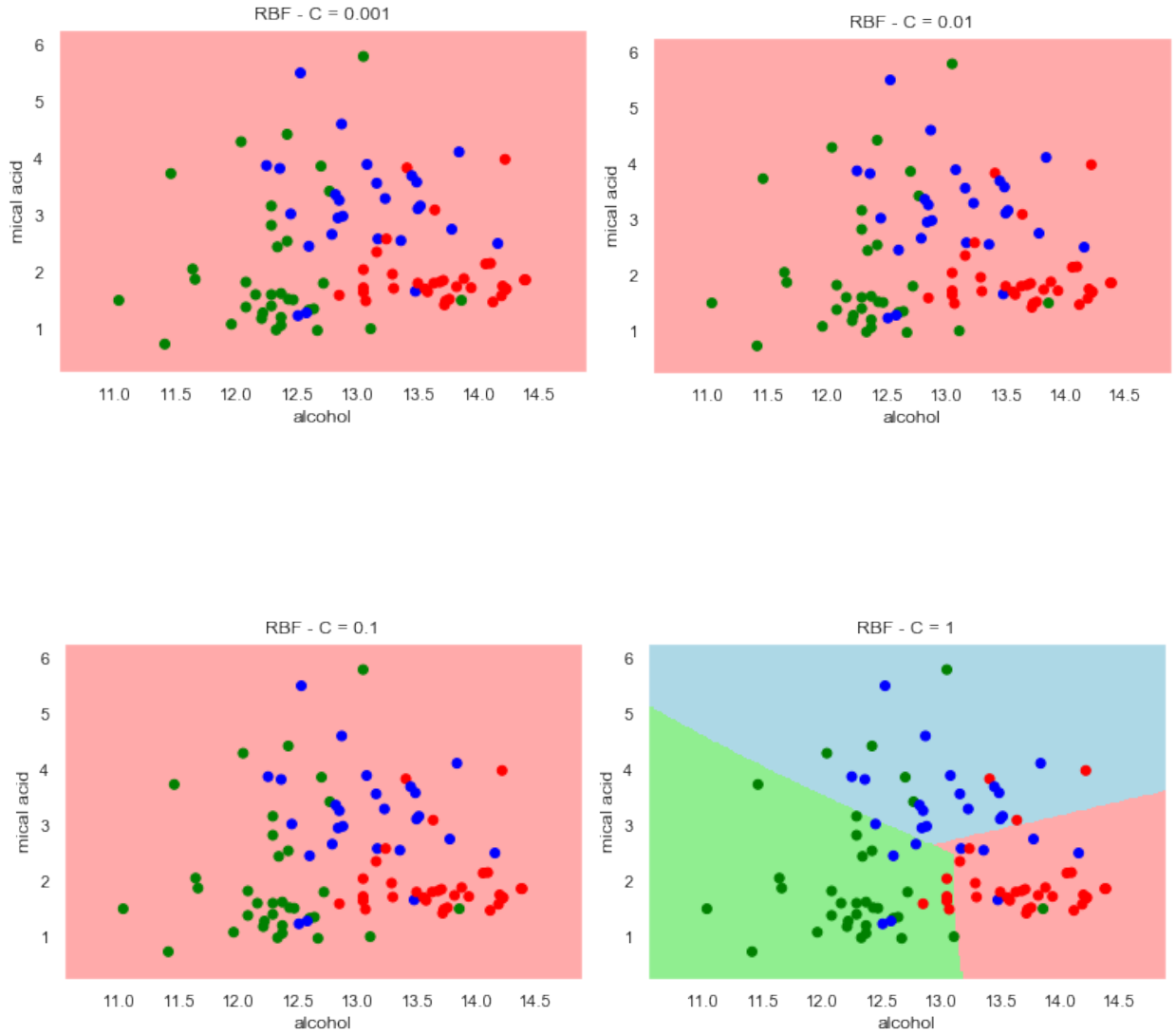
Again, the accuracy results in a value which is above the 80%, so we may conclude that the model works quite well, since the dataset is not entirely linearly separable, and we expected some misclass -ified points.

4 – RBF kernel

We repeat the previous section by using a Gaussian RBF kernel, which means that our classifier will not use linear functions to separate the data, yet it will be based on the following equation:

$$K_{\text{RBF}}(x,x) = \exp -\gamma\|x-x\|$$

Repeating the pipeline, that is training the model by using the same C values yields the following re -sults in terms of decision boundaries (**fig. 7**).



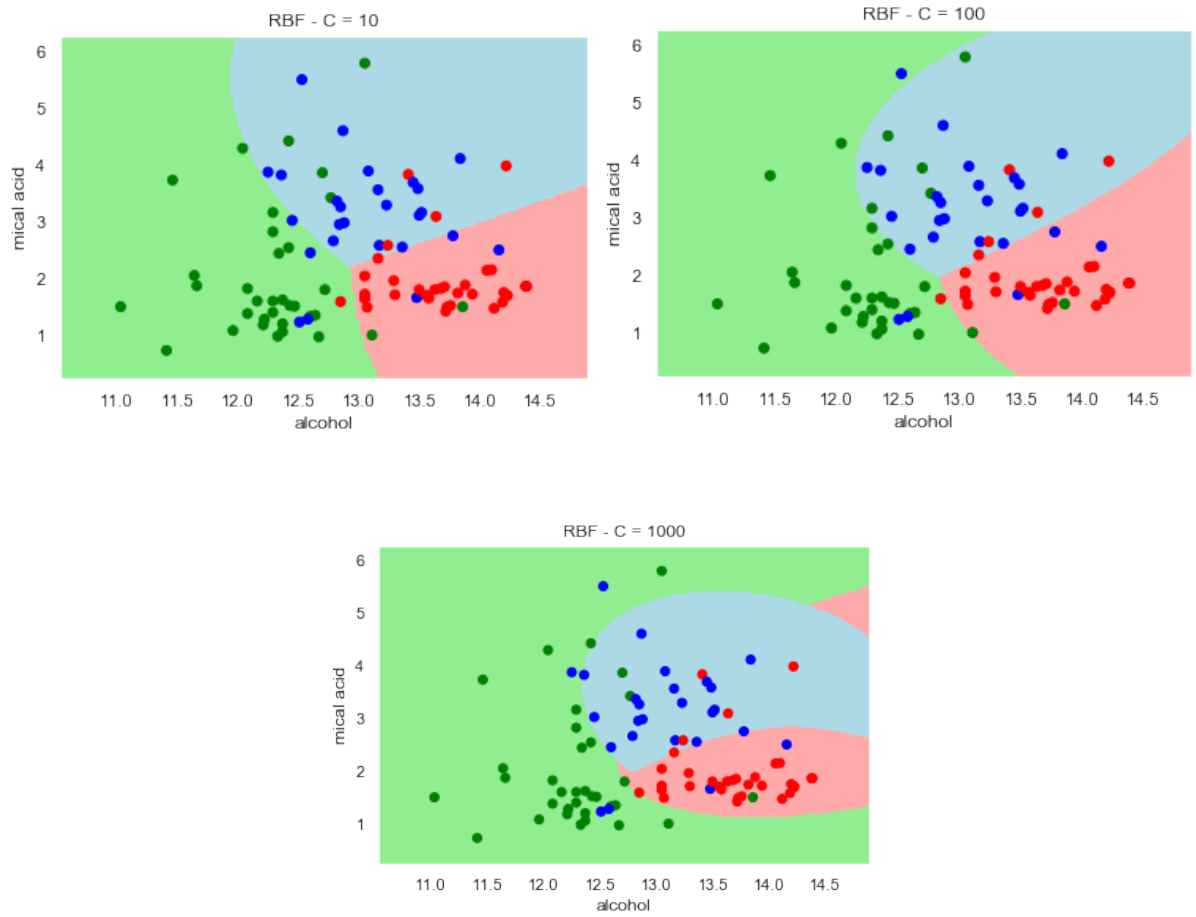


Fig. 7: *SVM with RBF kernel decision boundaries using different values of C*

We can immediately spot the differences with respect to the linear kernel: in fact, the boundaries are not linear anymore. Intuitively, in this kind of problem, this allows us a higher degree of flexibility. If we plot again the accuracy scores achieved by the models against the value of the C parameter selected on the validation set, we obtain the following plot.

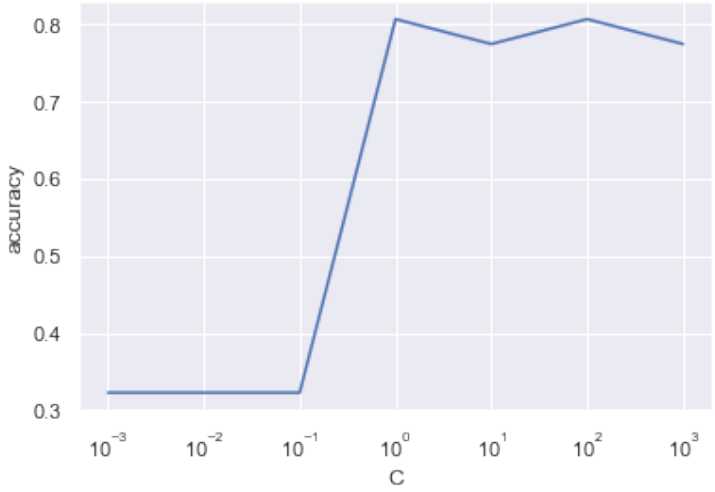


Fig. 8*: Accuracy against C (SVM with RBF kernel)*

Finally, by choosing the C value that performs better on the validation set, using it to train a model and evaluating it on the test set we result in an accuracy slightly smaller than the one achieved in the linear case (≃0.80).

As the last step, we want to perform a grid search on two parameters: C and gamma, that need to be tuned together to train the model.
Gamma, qualitatively, represents the spread of the kernel, therefore of the decision region. By choosing a very small gamma, we result in a very low curve of the decision boundary and in a very broad decision region. On the other hand, when gamma is large the curve of the decision boundary is high and decision regions will be isolated around data points.
The range that has been chosen is: 0.0001, 0.001, 0.01,1 and 10.

Performing a grid search means to train a model on each possible combination of the C and gamma parameters and evaluate it by using a chosen score metric, accuracy in this case.
The best resulting model is then applied on the test set.
The resulting decision boundaries and the accuracy on the test set are reported below.

>>> Output:

```
Fitting 5 folds for each of 35 candidates, totalling 175 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 175 out of 175 | elapsed:    0.6s finished
{'C': 0.1, 'gamma': 1}
SVC(C=0.1, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=1, kernel='rbf', max_iter=-1,
    probability=False, random_state=None, shrinking=True, tol=0.001,
    verbose=False)
```

>>> Test accuracy: 0.7962962962962963

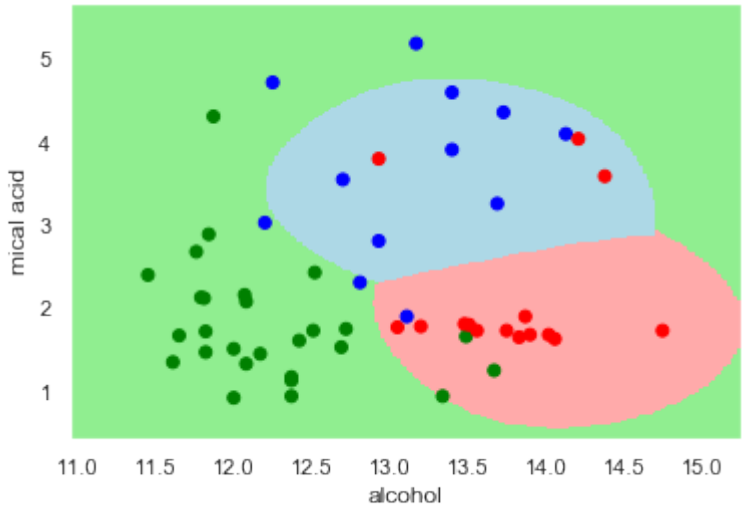


***Fig. 9**: SVM with RBF kernel decision boundaries using grid-search on C, Gamma*

Again, the accuracy is just below 80%.

5 – K-FOLD

The last points of the homework requested to merge the training and validation set, resulting in the new proportion of 70% training and 30% test data.

At this point, we perform again a grid search for gamma and C, but this time using a 5-fold validation as validation technique. That means that the training set is divided into 5 folds of approximately the same size and iteratively 4 of them are used to train the model, while one is used to test it.

The final score, in terms of accuracy, is computed as the average of the 5 scores obtained using the different partitions and gives a better estimate of the goodness of the model with respect to the previously used validation technique.

>>> **Output:**

```
Fitting 5 folds for each of 35 candidates, totalling 175 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concu
rrent workers.
{'C': 1000, 'gamma': 0.01}
SVC(C=1000, break_ties=False, cache_size=200, class_weight=None, c
oef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.01, kernel='r
bf',
    max_iter=-1, probability=False, random_state=None, shrinking=T
rue,
    tol=0.001, verbose=False)

Validation accuracy (5-FOLD): [0.8  0.8  0.88 0.72 0.75]
Test-set accuracy: 0.8518518518518519
```

The resulting test accuracy is different from before, in particular it is higher than what reported in t he previous section, that was a grid search of gamma and C implemented together with a SVM with a RBF kernel.
Intuitively, performing a K-Fold validation should yield a better result, in terms of precision and cor -respondence to reality. Being a resampling technique, K-Fold is particularly effective on small data -sets and ensures a less biased result in terms of accuracy.
We can think that, owing to the small size of the dataset, by using a larger training set we can achie- ve better results.

Extra

Discuss the difference between KNN and SVM

Both KNN and SVM are machine learning algorithms that provide a classification, which is a supervised learning methodology. That means that both the algorithms aim to classify unlabeled objects after a learning phase in which a model is built based on a set of classified objects, known as training set.

The main difference between the two algorithms is that in KNN each and every point of the training set must be considered in order to compute its nearest neighbors and finally build a model, while when using SVM the model is based just on a number of points, called Support Vectors. So the points do not have the same importance, those on the margin are the ones responsible of the classifier.

The inner algorithm is indeed very different in the two cases. In KNN some sort of distance measure (e.g. Euclidean Distance) is used in order to find the K nearest objects and the classification of the current object is based on a majority voting methodology.
SVM, instead, attempts to find a hyperplane (which may be linear or not, based on the type of kernel) able to separate the different classes. As stated in the previous section SVM has two goals: find the largest minimum separating hyperplane and misclassify the fewer points as possible, the tradeoff is chosen based on the specifics of the problem by tuning the hyper parameters of the model properly.

Try also with different pairs of attributes

In this final section of the homework I repeat the previous reported methodologies with a different pair of attributes, i.e two different features of the Wine Dataset. I have decided to perform the analysis also for the features: “flavanoids” and “proanthocyanins”. The pipeline is exactly the same and the results are reported below, following the same steps as the initial analysis.

Data representation:

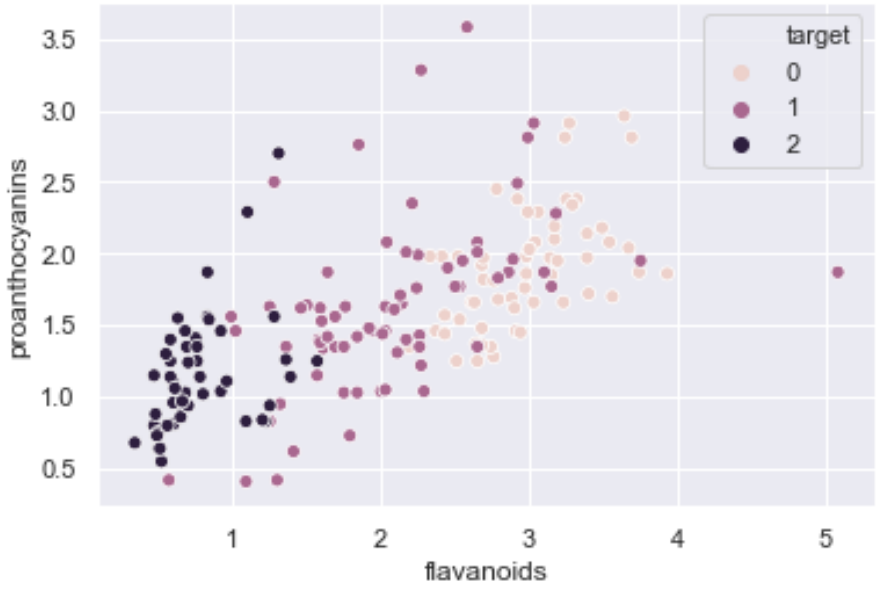


***Fig.10**: 2-d representation of the dataset*

KNN:

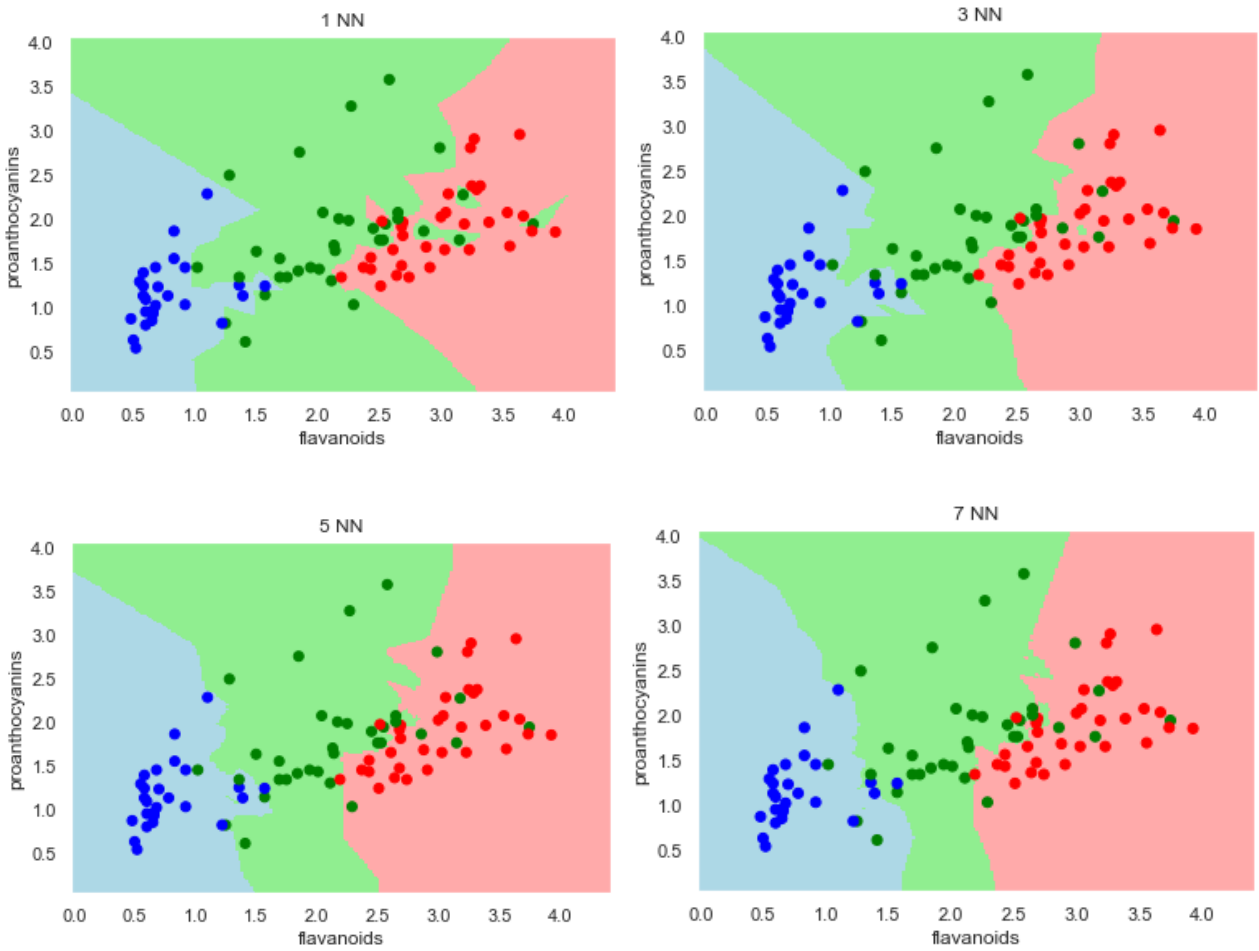


Fig. 11: *KNN decision boundaries using different values for K*

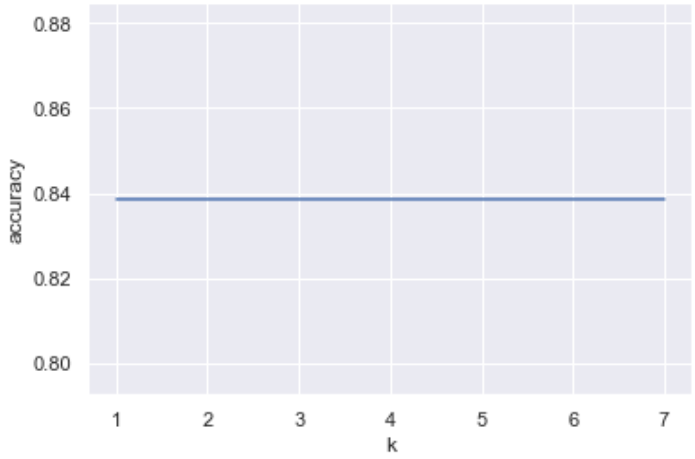
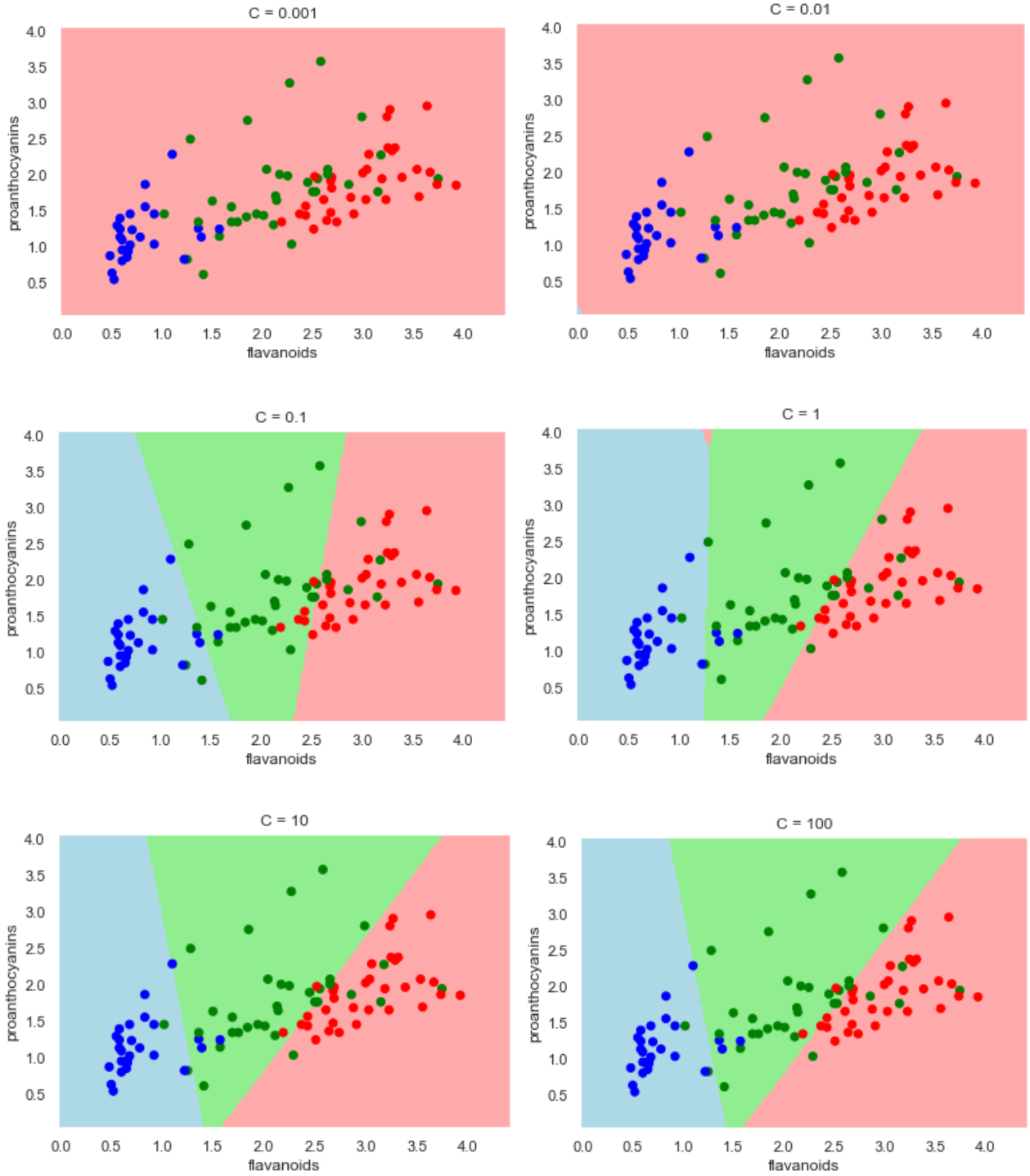


Fig. 12*: Accuracy against K (number of nearest neighbors)*

Accuracy achieved on the test set: 80%.
In this case the hyper-parameter K does affect the results, we may think this is because the classes are already well separated.

Linear SVM:


C = 0.001
C = 0.01
C = 0.1
C = 1
C = 10
C = 100
proanthocyanins
flavanoids
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0

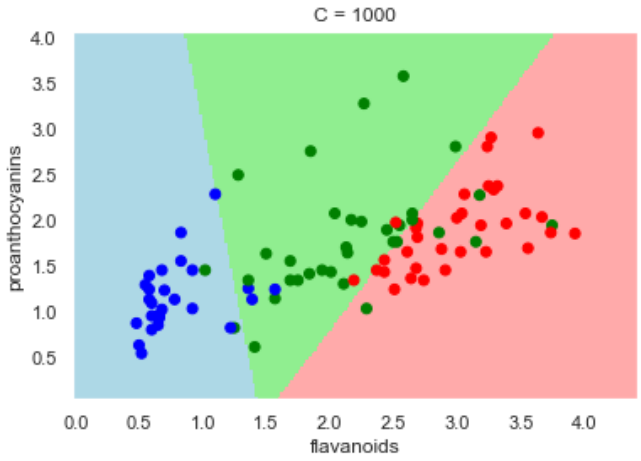


Fig. 12: *Linear SVM decision boundaries using different values of C*

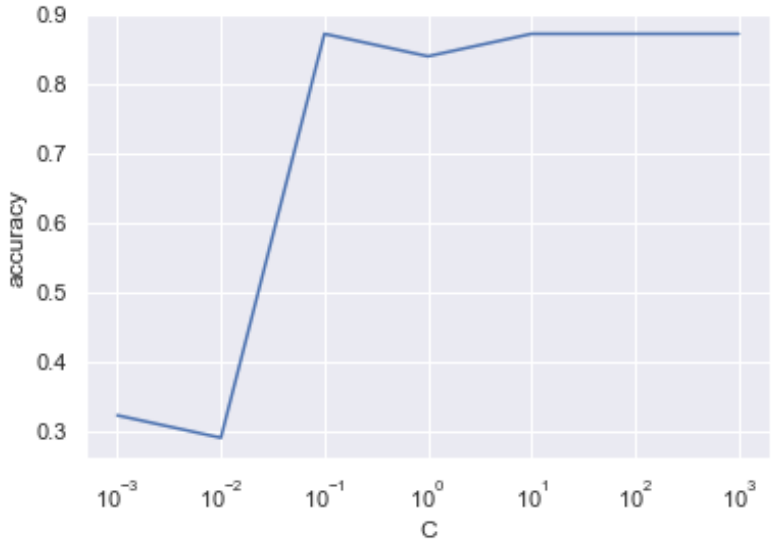
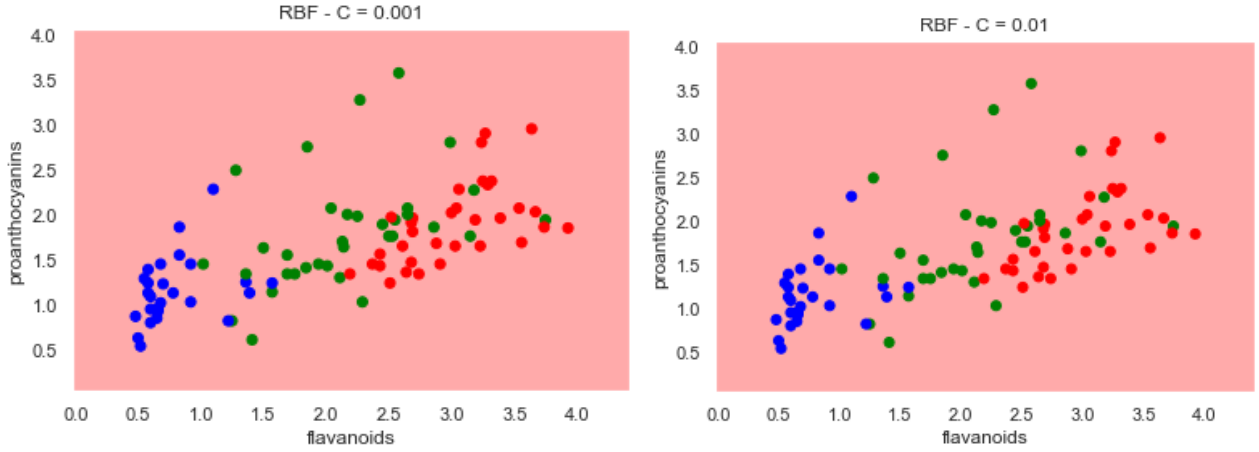


Fig. 13*: Accuracy against C (Linear SVM)*

Accuracy achieved on the test set: 78%.

RBF Kernel SVM:



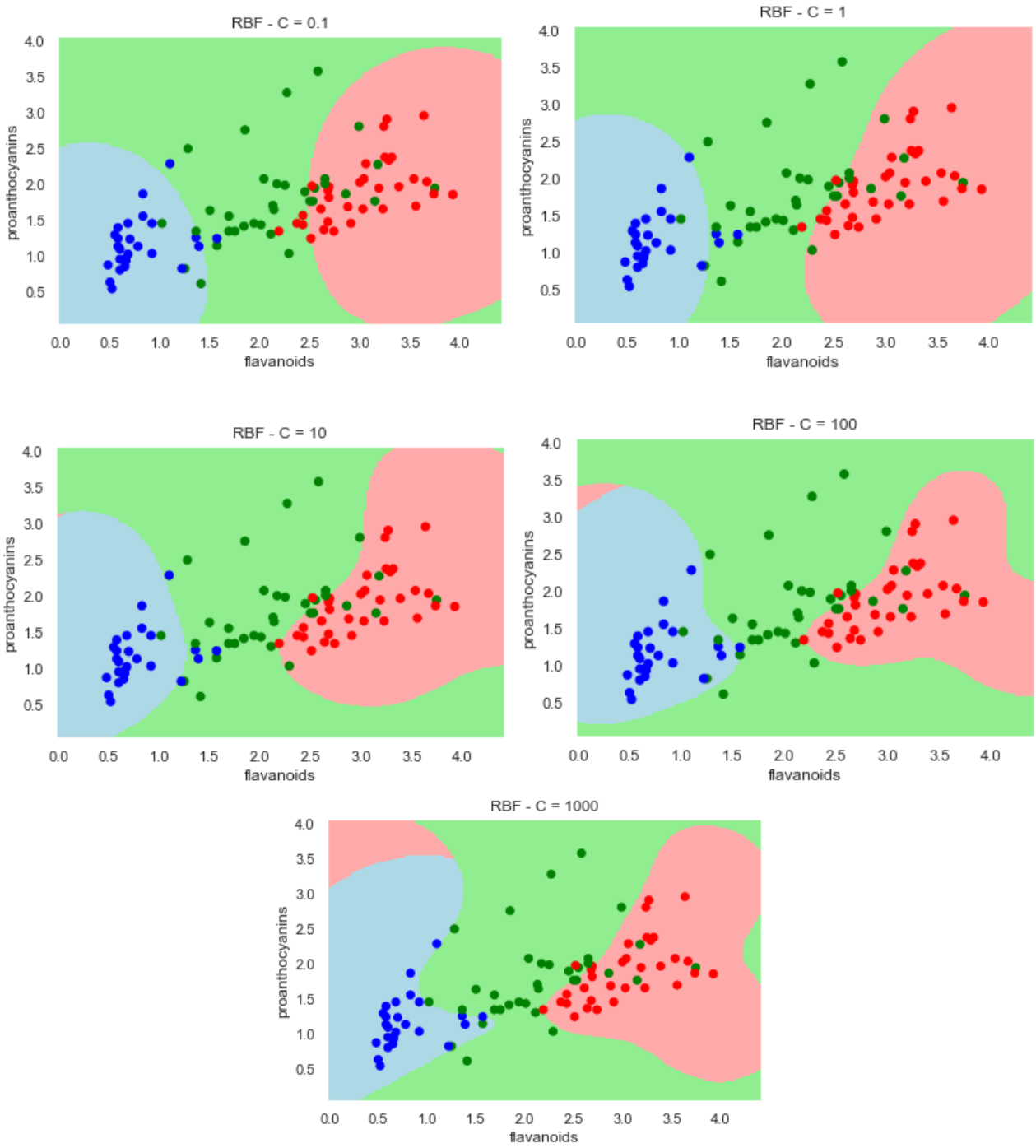


Fig. 14: *SVM with RBF kernel decision boundaries using different values of C*

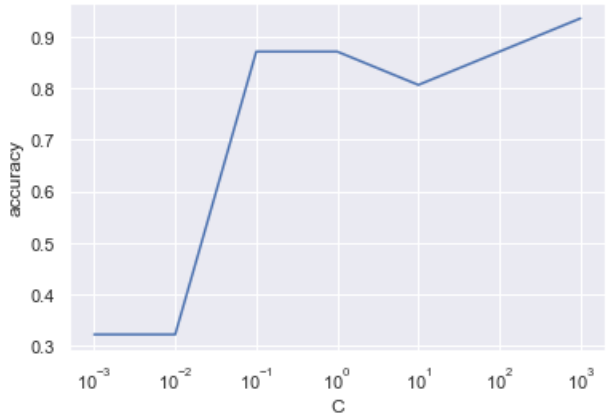


***Fig. 15**: Accuracy against C (SVM with RBF kernel)*

Accuracy achieved on the test set: 80%.

Grid search:
>>> Output:

```
Fitting 5 folds for each of 35 candidates, totalling 175 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 175 out of 175 | elapsed:    1.8s finished
{'C': 10, 'gamma': 1}
SVC(C=10, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=1, kernel='rbf', max_iter=-1,
    probability=False, random_state=None, shrinking=True, tol=0.001,
    verbose=False)
0.7962962962962963
```

>>> Plot best model:

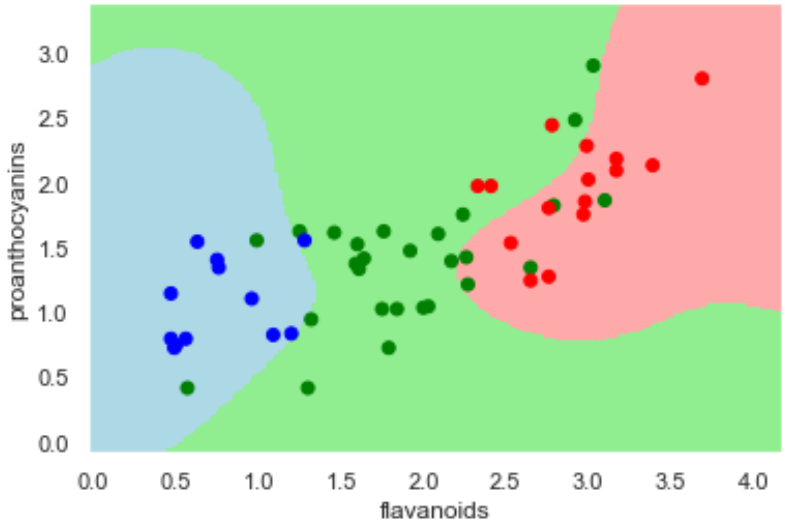


***Fig. 16**: SVM with RBF kernel decision boundaries using gridsearch on C, Gamma*

Cross validation:

>>> Output:

```
Fitting 5 folds for each of 35 candidates, totalling 175 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
{'C': 10, 'gamma': 10}
SVC(C=10, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=10, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

Validation set accuracies: [0.88 0.8 0.72 0.84 0.875]
Test set accuracy: 0.8148148148148148