Homework 1 (KNN and SVM)

Marco Gullotto s274563

June 15, 2020

1 Introduction

The goal of this homework is to compare two different supervised machine learning models: k-nearest neighbors (**Knn**) and support vector machines (**SVM**). Knn is a hugely popular algorithm that is used only for small datasets, based solely on the proximity of the points. It is a lazy algorithm: this means that it only has a short training phase, or is not trained at all. The most important hyperparameter to set is "K", which represents the number of neighbors needed to determine which category the new sample belongs to. SVM, instead, is based on the "dual theory" and it's made to solve classification problems using both linear and non-linear kernel. It's planned to solve large scale problems, especially the " LinearSVC" class of the sklearn library, but we will deal with this later.

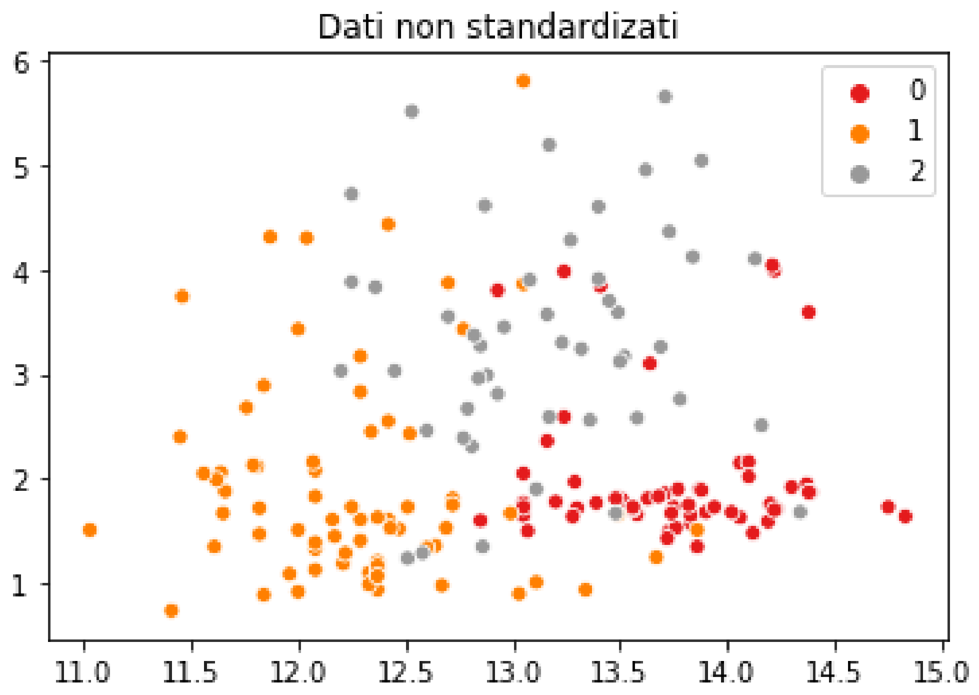
1.1 Wine dataset

```
data = load_wine()
```

The wine dataset (ds) comes from the chemical analysis of the wines. In this version of the ds, 13 features are taken into consideration, but in order to plot the distribution of our data, 2 predictors were chosen randomly: "Alcohol"(x axis in the following graph) and "Malic-acid"(y axis in the following graph).

```
X = data["data"]#Load only the data
X = X[:, :2]#Take the first 2 columns only
y = data["target"]#Load the target
sns.scatterplot(X[:, 0], X[:, 1], hue = y).set_title("Dati non standardizati");
```

From the first plot, it's clear that the problem we are dealing with can't be solved in an exactly linear way. Therefore, probably, the Knn and the SVM with a Radial Basis Function (**RBF**) kernel would work better than a linear SVM. Obviously, this is only a first impression based on what can be observed from this graph.



Therefore we can start to address our classification problem by shuffling and splitting the data into train (50%), validation (20%) and test (30%) sets.

```
X_t, X_test, y_t, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
X_train, X_val, y_train, y_val = train_test_split(X_t, y_t, test_size=0.33,␣
 ↪random_state=1)
```

SVM is extremely sensitive to the feature scales, so the datas are normalized using a Sklearn's "StandardScaler". This also applies to the Knn algorithm, since we're computing the distance to the closest K points.

```
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_val = ss.transform(X_val)

X_t = ss.fit_transform(X_t)
X_test = ss.transform(X_test)
```

2 Knn

2.1 Train and evaluation phase

Let's start by taking into consideration the Knn algorithm, which will represent the first baseline for this task. The way this method works is very simple: first, it stores the entire training dataset; then, when a new record comes, it computes the distance between the new point and all the others. The new label is decided by the nearest k-points. The gap between the records can be measured in many different ways: using 2D data, the euclidian distance is a smart choice, but there also different types of criteria, such as the Mahalanobis ones which considers the correlation between the data. Therefore, there is no deterministic way to establish the value of "K", but only the trial and error approach can be used. Assign K to low values means that we believe that the distribution of our test will be very similar to the train one. This can lead to overfitting, but using too high values of K can still cause problems, especially when the classes are not that distinct. Since the number of records in the dataset is not too high, the choice to try to assign [1,3,5,7] to K seems legitimate.

```
n_n = [1,3,5,7]
knns = []
acc_k = []
for val in n_n:
    knn = neighbors.KNeighborsClassifier(n_neighbors = val)
    knn.fit(X_train, y_train)
    knns.append(knn)
    y_pred_val = knn.predict(X_val)
    acc_k.append(accuracy_score(y_val, y_pred_val))

plotDiffSVM(knns, X_train, y_train, n_n, "KNN with K =", X_train, y_train)
```

The decision limits are very similar for K = [3, 5 7]. Vice versa, as you can easily imagine, something changes when k = 1, because this attribution allows to create more borders for isolated points (and in this case we have many of them). From the observation of the graph, we highlight the possibility that the algorithm works better with a higher value of K, because it shuold be less sensible to outlayers, to be sure, we will see the performance on the validation set.

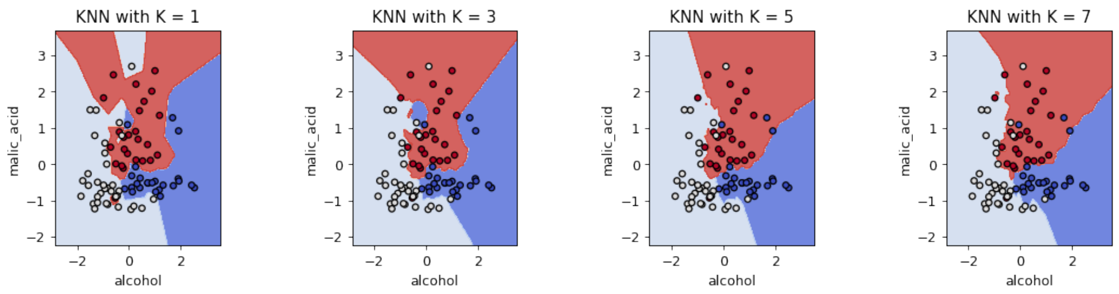
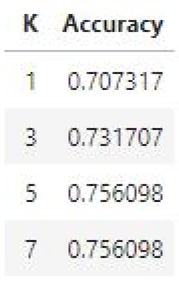


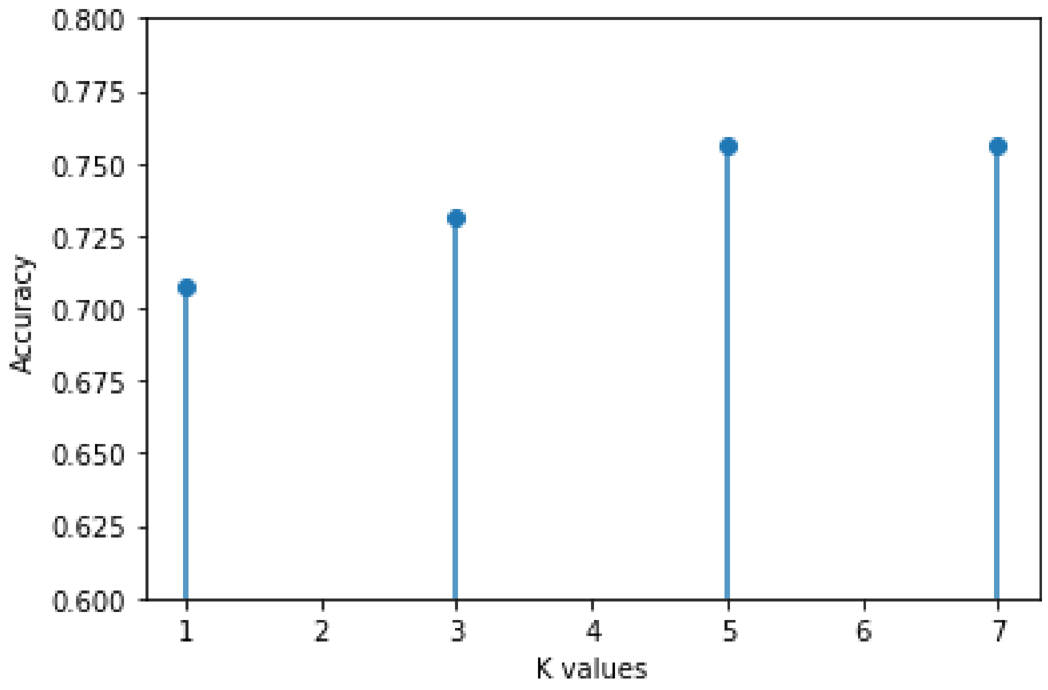
Figure 1: *KNN with various K + train data*

2.2 KNN accuracy on the validation set

The accuracy of the built models confirms the intuitive detection. If the values attributed to K are high, the performances obtained turn out to be better, but in this case the difference is not excessive. The spread between K = 1 and K = 5 differs only by 0.5%.



K	Accuracy
1	0.707317
3	0.731707
5	0.756098
7	0.756098



2.3 Test our best model on the test set

From the moment in which the best value for K was found, we can try to build our model on the train + validation set and evaluates its actual performance on the test set. The result is not so bad, considering the use of only two features. Approximately 83% of the test sample is correctly predicted, as illustrated by the graph. Only a few points have been incorrectly classified, and most of them are close to the borders. This means that this classifier works quite well, which leads to the conclusion that an svm could lead to an even better result.

```
k_best = n_n[np.argmax(acc_k)]
knn = neighbors.KNeighborsClassifier(n_neighbors = k_best)
knn.fit(X_t, y_t)
y_pred_test = knn.predict(X_test)
accuracy_score(y_test, y_pred_test)
```

```
Accuracy score: 0.8333333333333334
```

```
plotDiffSVM([knn], X_t, y_t, [k_best], "Knn best with k =", X_test, y_test)
```

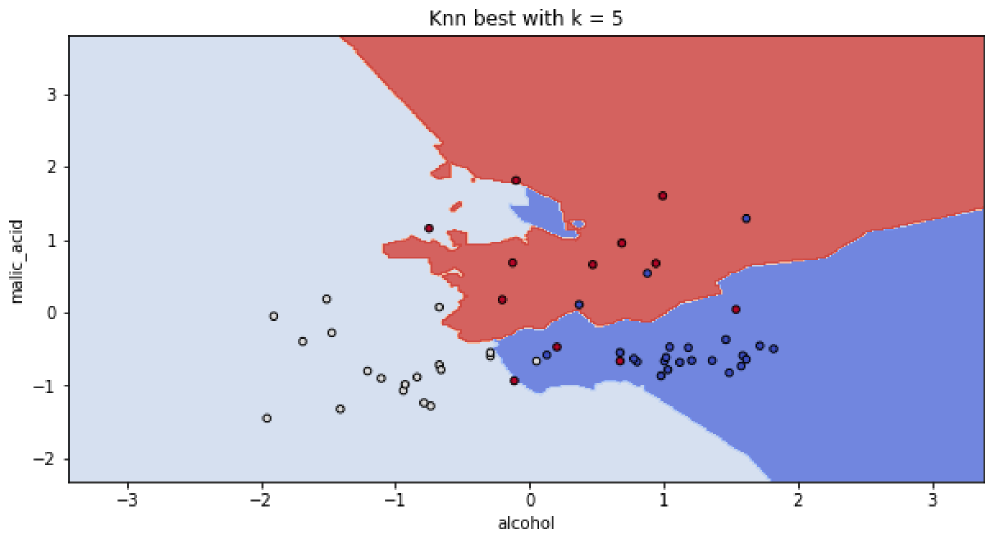


Figure 2: *KNN boundaries on test data*

3 SVM

“Linear SVC" is the sklearn class that implements linear support vector machines. The most relevant hyperparameter to be tuned in this algorithm is the value of C. The width of the SVM margin is directly proportional to 1 / C. Therefore, when we assign a high value to this parameter, the margin obtained will be narrow. This means that we start from the assumption that there are not many outliers in our training set, and consequently a large number of errors will not be allowed. This also implies that the decision limit will depend only on fewer samples. On the other hand, if the value of C is low, we would allow more errors, and this could mean that we are avoiding overfitting. The support vectors will depend on multiple samples and the margin considered will be greater, but this is not necessarily positive (as we will see with the rbf kernel).

```
C = [1e-3, 1e-2, 1e-1, 1, 10, 1e+2, 1e+3]
svcs = []
acc_s = []
for val in C:
    svc = LinearSVC(C = val, random_state=0)
    svc.fit(X_train, y_train)
    svcs.append(svc)
    y_pred_val = svc.predict(X_val)
    acc_s.append(accuracy_score(y_val, y_pred_val))

plotDiffSVM(svcs, X_train, y_train, C,"LinearSVC with C =", X_train, y_train)
```

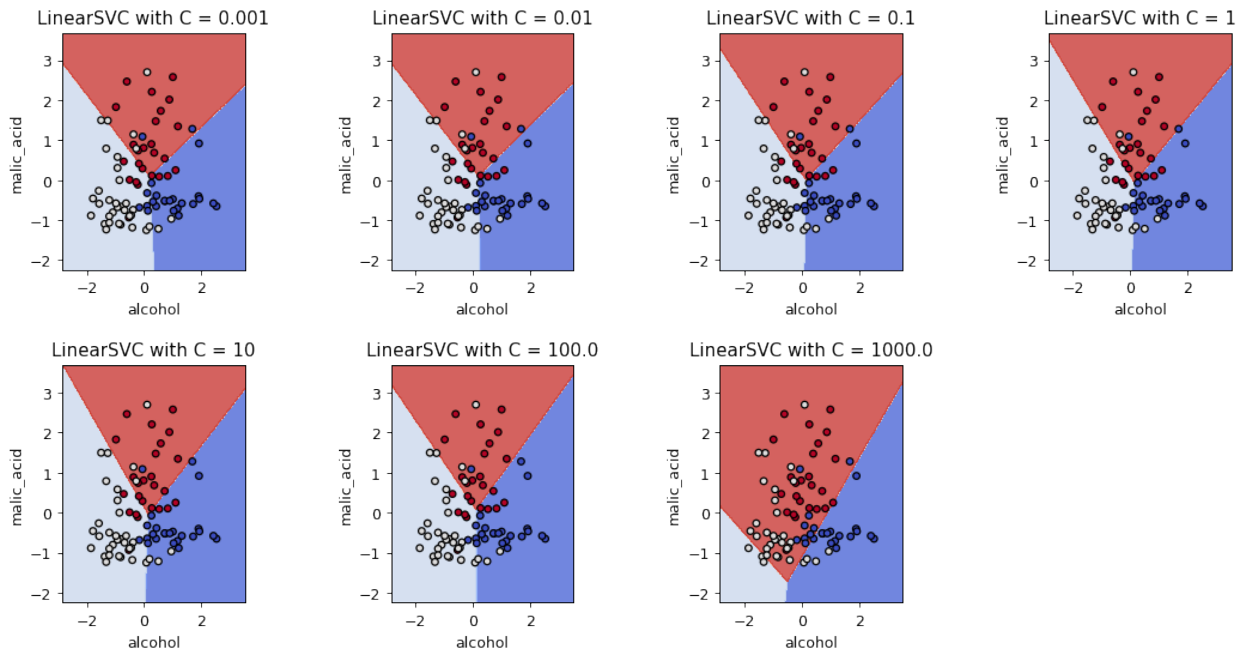
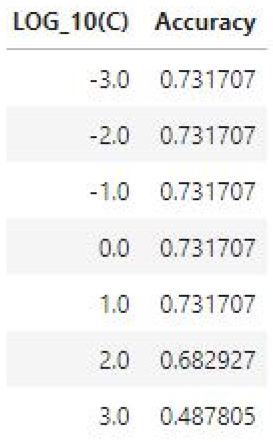
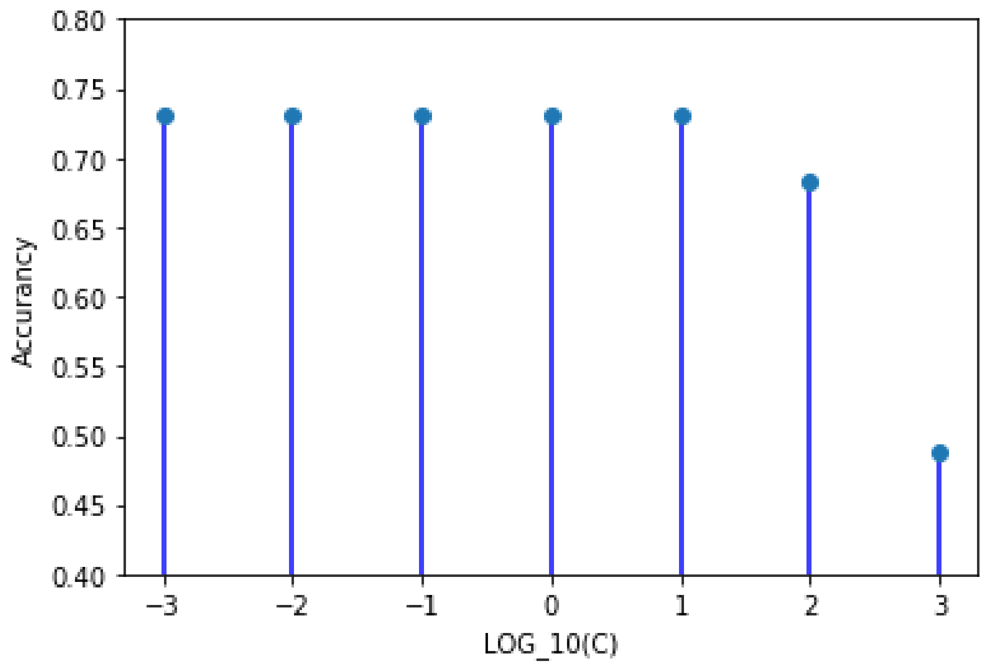


Figure 3: *LinearSVC with various C + train data*

As can be seen from the graph, the behavior is very similar for the values of C in the range between 1e-3, and 1e+2, but when the margin becomes very small, the boundaries change drastically. In fact, the more the margin is reduced, the more the algorithm begins to mistakenly classify many white points: this is due to the presence of a red point very far from the others. Since we have chosen a classifier with a hard margin, we are not allowing for errors, which can lead to an overfitting problem. The following graph confirms the expectations. The accuracy is very high (about 0.75) for values of C lower than 1e+2, but when this threshold is exceeded, the result is worse. This confirms everything that has been said so far, and clearly shows that it is therefore not a linear problem.



LOG_10(C)	Accuracy
-3.0	0.731707
-2.0	0.731707
-1.0	0.731707
0.0	0.731707
1.0	0.731707
2.0	0.682927
3.0	0.487805



3.1 Try with our best model

Similarly to what was done with the Knn method, it is necessary to train the SVM in the train + validation set, and study the result obtained on the test set. So, we have trained the model using the best C value founded in the previous step. As illustrated by the accuracy (which exceeds 0.85) and the relative graph, the classifier, with a very small value of C, divides the class quite perfectly. However, problems remain on some points, but it is difficult to imagine being able to have a better result by using a linear classifier.

```
c_best = C[np.argmax(acc_s)]
svc = LinearSVC(C = c_best, random_state=0)
svc.fit(X_t, y_t)
y_pred_test = svc.predict(X_test)
accuracy_score(y_test, y_pred_test)
```

```
Accuracy score: 0.8518518518518519
```

```
plotDiffSVM([svc], X_t, y_t, [c_best], "LinearSVC best with C =", X_test, y_test)
```

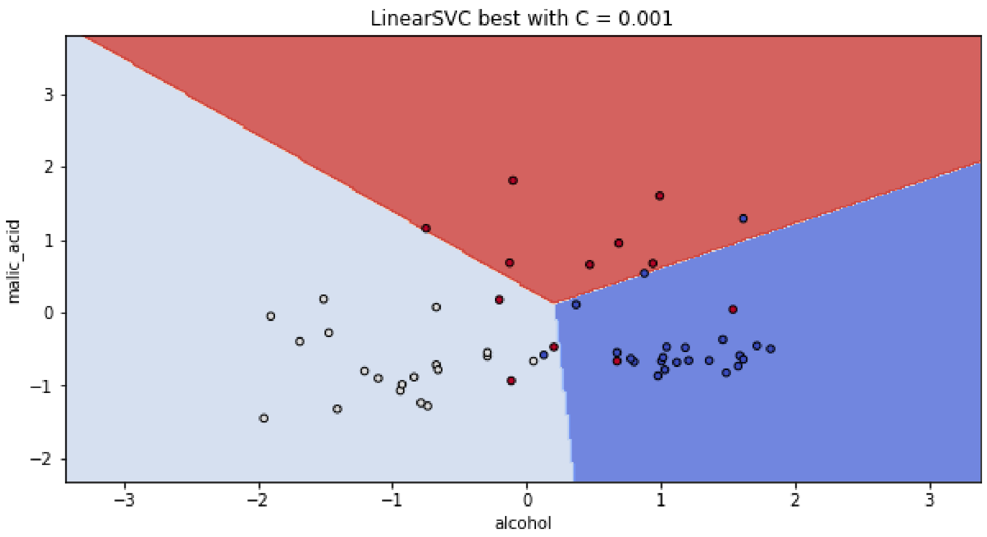


Figure 4: *LinearSVC boundaries + test data*

4 SVM with rbf kernel

So let's add some non-linearity to our SVM model, and to do this we use, as mentioned previously, the RBF kernel. It is a very common kernel, used in various algorithms which use kernel trick. If **x** and **x'**, represented our function vectors in an input space, the RBF is defined as:

$$K(x, x') = e^{-\gamma\|x-x'\|^2}$$

where $\gamma = \frac{1}{2\sigma^2}$ .This means that when we adjust the gamma value, we are actually changing the variance of our model. The gamma parameters can also be considered as the reciprocal of the radius of influence of the samples selected by the model as support vectors (as reported in the sklearn documentation). Therefore, when the gamma value is high, the influence of a single sample is reduced, while, conversely, when the gamma value is low, the opposite effect is obtained. This means that, as for the C value, when a model is overfitting we must reduce the gamma value, conversely, when the model is underfitting, we must increase the value of that parameter. The first thing to do is to optimize the C parameter: then, it is necessary to apply a "GridSearch" to find the best balance between the values of C and gamma.

```
svms = []
acc_r = []
for val in C:
    svc = SVC(C = val, random_state=0)
    svc.fit(X_train, y_train)
    svms.append(svc)
    y_pred_val = svc.predict(X_val)
    acc_r.append(accuracy_score(y_val, y_pred_val))

plotDiffSVM(svms, X_train, y_train, C, "RBF svm with C =", X_train, y_train)
```

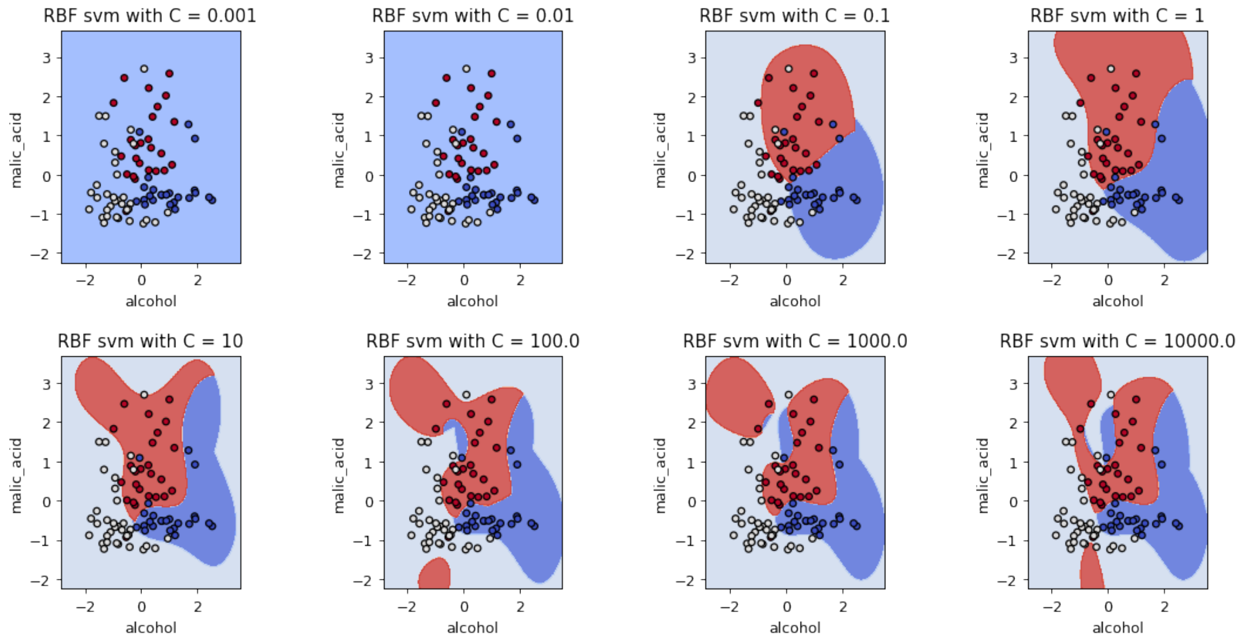
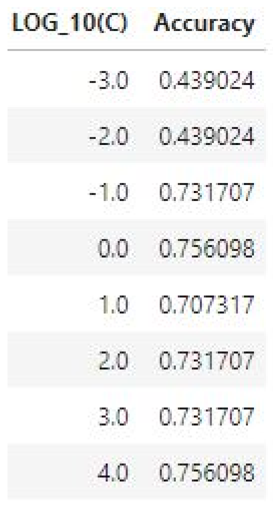
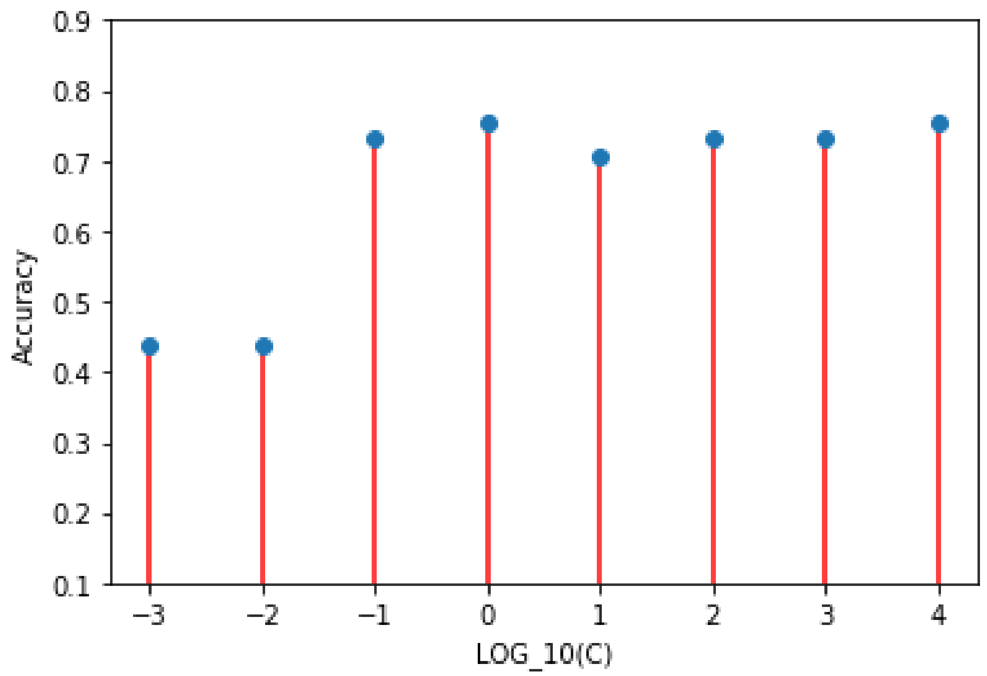


Figure 5: *SVM with various C + train data*

There are some fundamental differences between svm with linear kernel and RBF. With the first approach we could only get straight lines that separate the different classes, but with a nonlinear kernel the result obtained is drastically different. We have curved lines and multiple borders for the same class in different places. So, do we have a better understanding of the problem, or are we just overfitting our model? In this case, it seems that with a high C value the classifier adapts better to the data but, when applying our model on the test set, the accuracy is not better than that obtained with "LinearSVC".



LOG_10(C)	Accuracy
-3.0	0.439024
-2.0	0.439024
-1.0	0.731707
0.0	0.756098
1.0	0.707317
2.0	0.731707
3.0	0.731707
4.0	0.756098



```
c_best = C[np.argmax(acc_r)]
svc = SVC(C = c_best, random_state=0)
svc.fit(X_t, y_t)
y_pred_test = svc.predict(X_test)
accuracy_score(y_test, y_pred_test)
```

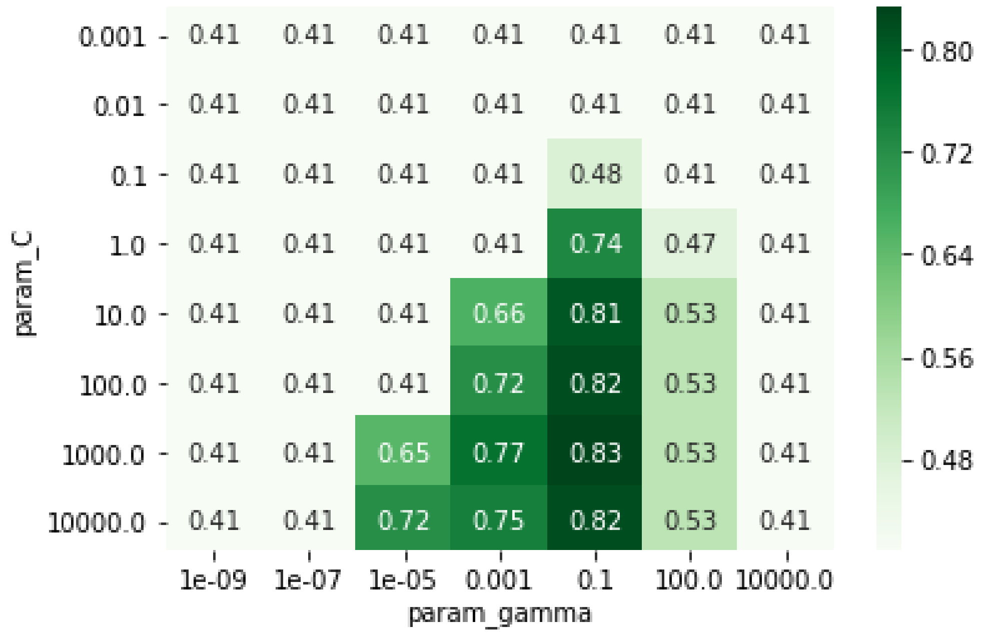
```
Accuracy score: 0.8333333333333334
```

4.1 Gamma tuning

The result obtained, perhaps, is due to the fact that our model is overfitting the data, therefore, to improve this result, we can try to optimize the gamma hyperparameter. To fix it, we expect gamma to assume a value which is not too high.

```
gamma = [1e-9, 1e-7, 1e-5, 1e-3, 0.1, 1e+2, 1e+4]
param_grid = [
  {'C': C, 'gamma': gamma},
 ]
clf = GridSearchCV(SVC(), param_grid)
clf.fit(X_train, y_train)
best = clf.best_params_
pvt = pd.pivot_table(pd.DataFrame(clf.cv_results_), values='mean_test_score',␣
 ↪index='param_C', columns='param_gamma')
sns.heatmap(pvt, cmap="Greens", annot = True)
```

To best illustrate the result, we have drawn a heat map that shows the accuracy score obtained by combining the different values of C and Gamma. As seen before, the result that can be obtained with a low C is very poor, but reaches higher scores by increasing this parameter. Not surprisingly, the algorithm works well with small gamma values. The combination C = 1000 and gamma = 0.1 produced, on average, the accuracy of 0.83. It's not an exceptional result, but as in the previous ones it's difficult to get a better result when we don't have so much data.



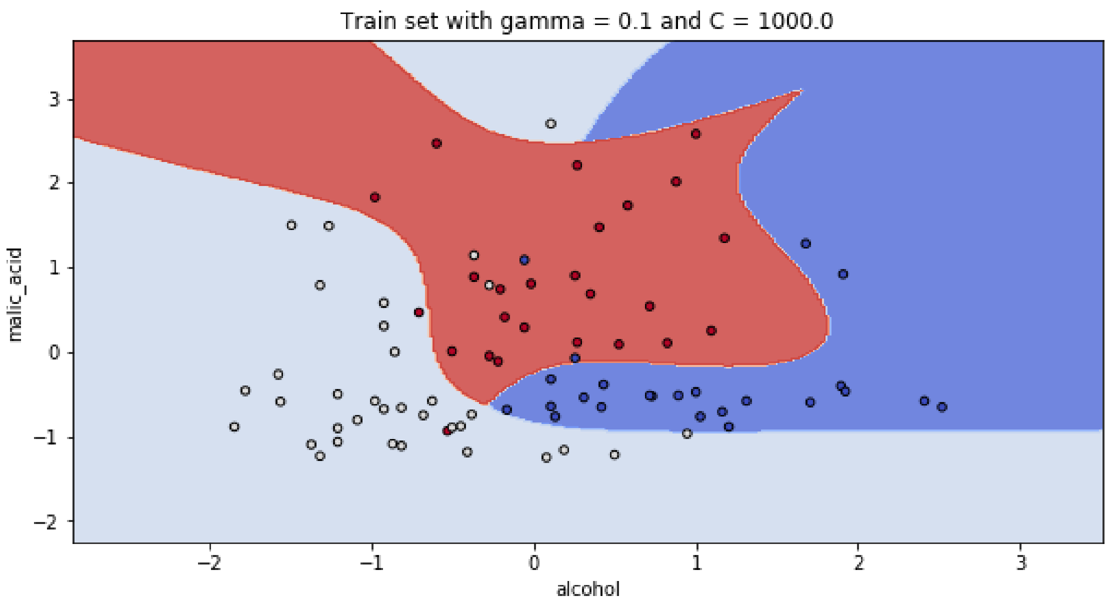
```
clf = SVC(C = best['C'], gamma = best['gamma'])
clf.fit(X_train, y_train)
y_pred_val = clf.predict(X_val)
print(accuracy_score(y_val, y_pred_val))
```

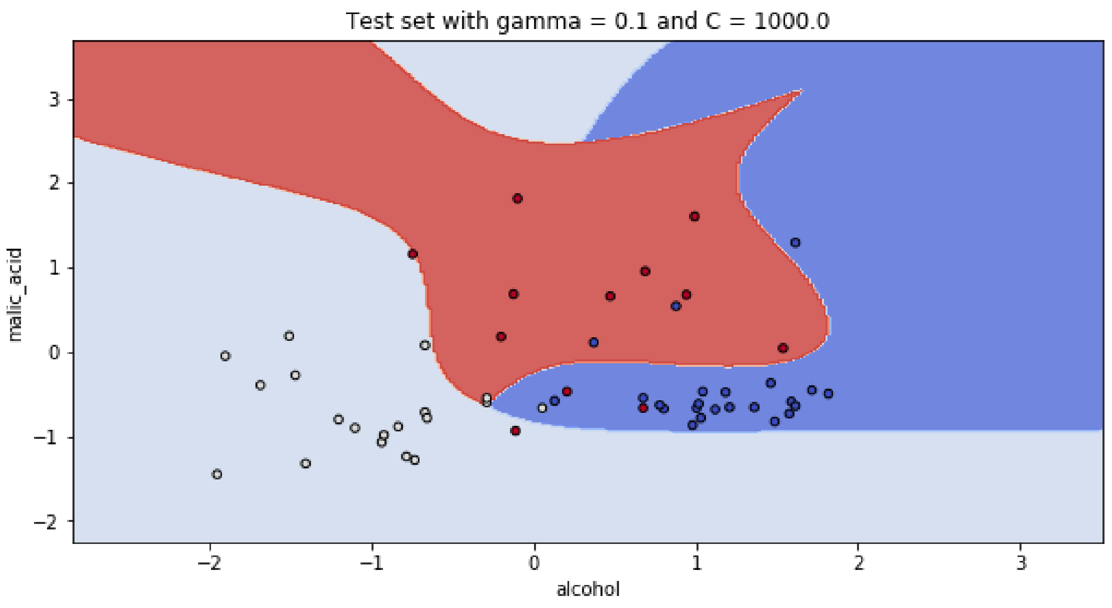
```
Accuracy score: 0.7317073170731707
```

```
clf = SVC(C = best['C'], gamma = best['gamma'])
clf.fit(X_train, y_train)
y_pred_test = clf.predict(X_test)
print(accuracy_score(y_test, y_pred_test))
```

```
Accuracy score: 0.8333333333333334
```

The result obtained in the validation set is worse than that obtained previously, but the accuracy in the test set is higher. Perhaps we have built a new model that is able to generalize better than those used so far.

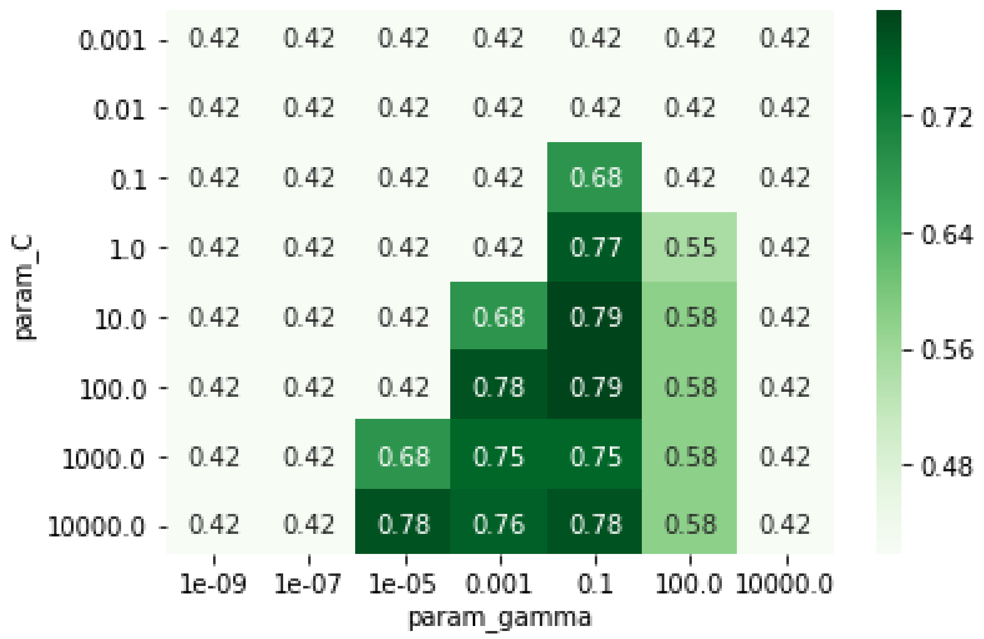




5 Cross-validation

By combining the training set and the validation set, we obtain a larger data set to train our model: this allows us to improve the result obtained. In addition, cross validation can be added. It should be noted that the "GridSearch" automatically implement the five-fold approach, therefore, to improve the result, I tried with a number of folds equal to 10.

```
param_grid = [
  {'C': C, 'gamma': gamma},
 ]
clf = GridSearchCV(SVC(), param_grid, cv=10)
clf.fit(X_t, y_t)
best = clf.best_params_
pvt = pd.pivot_table(pd.DataFrame(clf.cv_results_), values='mean_test_score',␣
 ↪index='param_C', columns='param_gamma')
ax = sns.heatmap(pvt, cmap="Greens", annot = True)
```



This heat map is, once again, very similar to the previous one. In this case the accuracy is lower, but relatively to the same area. It is not easy to understand this result, as using more data we would expect a better result. However, the new data probably have a distribution very similar to the previous one.

```
clf = SVC(C = best['C'], gamma = best['gamma'])
clf.fit(X_t, y_t)
y_pred_test = clf.predict(X_test)
print(accuracy_score(y_test, y_pred_test))
plotDiffSVM([clf], X_t, y_t, [best['C']], f"gamma = {best['gamma']} and C =",␣
 ↪X_t, y_t)
```

```
Accuracy score: 0.8333333333333334
```

Contrary to what is expected, more data does not improve performance.

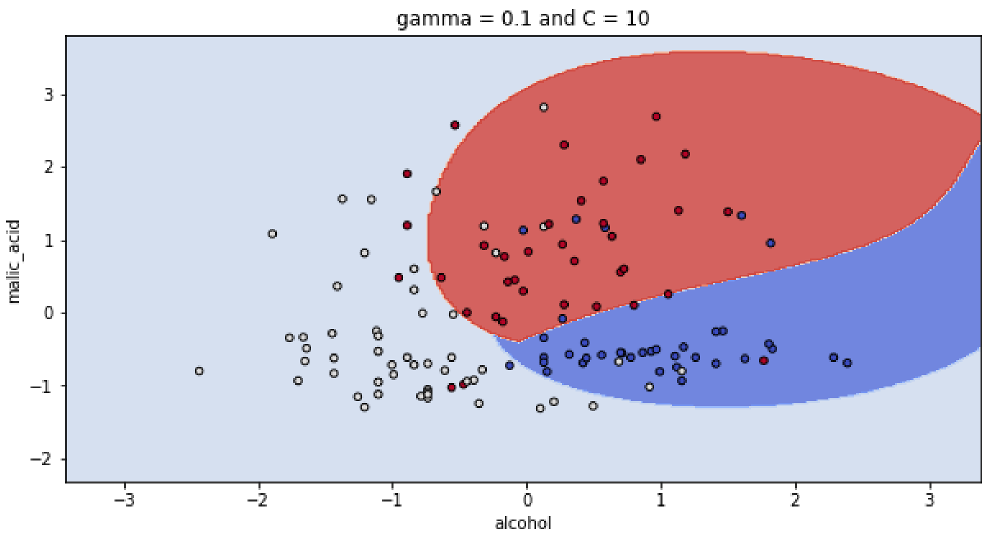


Figure 6: *SVM on train data*

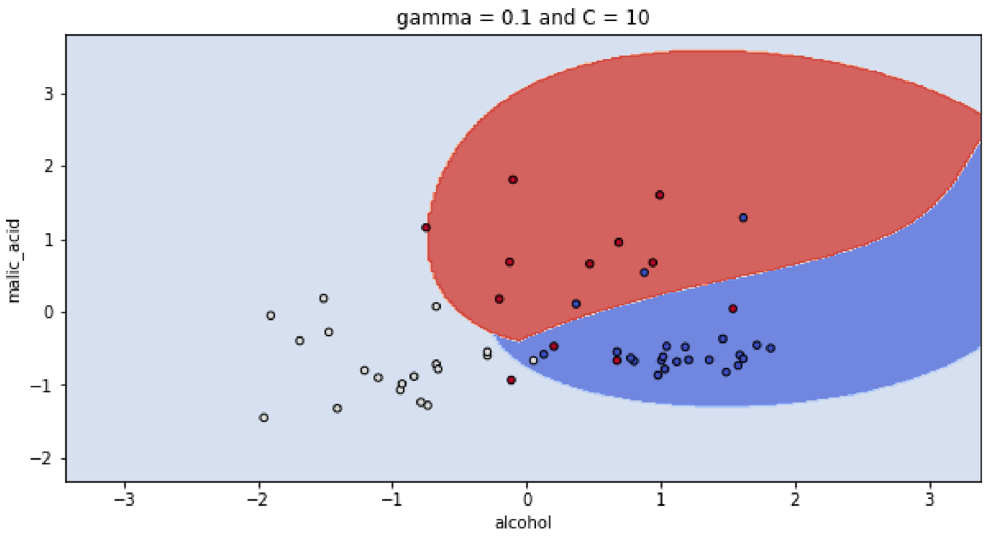


Figure 7: *SVM on test data*

6 New-features

Arbitrarily choosing two features to consider is not the best idea to get a good result in a classification problem. Therefore, in order to obtain a better score, it is possible to evaluate the correlation between the various characteristics, and then, at a later time, choose two predictors that are related to the target but that are not entirely related to each other. Considering the correlation matrix, two potential features could be "alcalinity-of-ash" and "color intensity". To check if it was a good choice, we need to trace the distribution of the data. The points thus obtained are more concentrated than before but, at the same time, the boundary between the classes is more explicit. All the previous steps are therefore repeated to confirm this first impression.

```
ds = pd.DataFrame(data=data["data"], columns=data["feature_names"])
ds["target"] = data["target"]
ax = sns.heatmap(ds.corr(),
        xticklabels=ds.corr().columns,
        yticklabels=ds.corr().columns,
        annot = True)
```

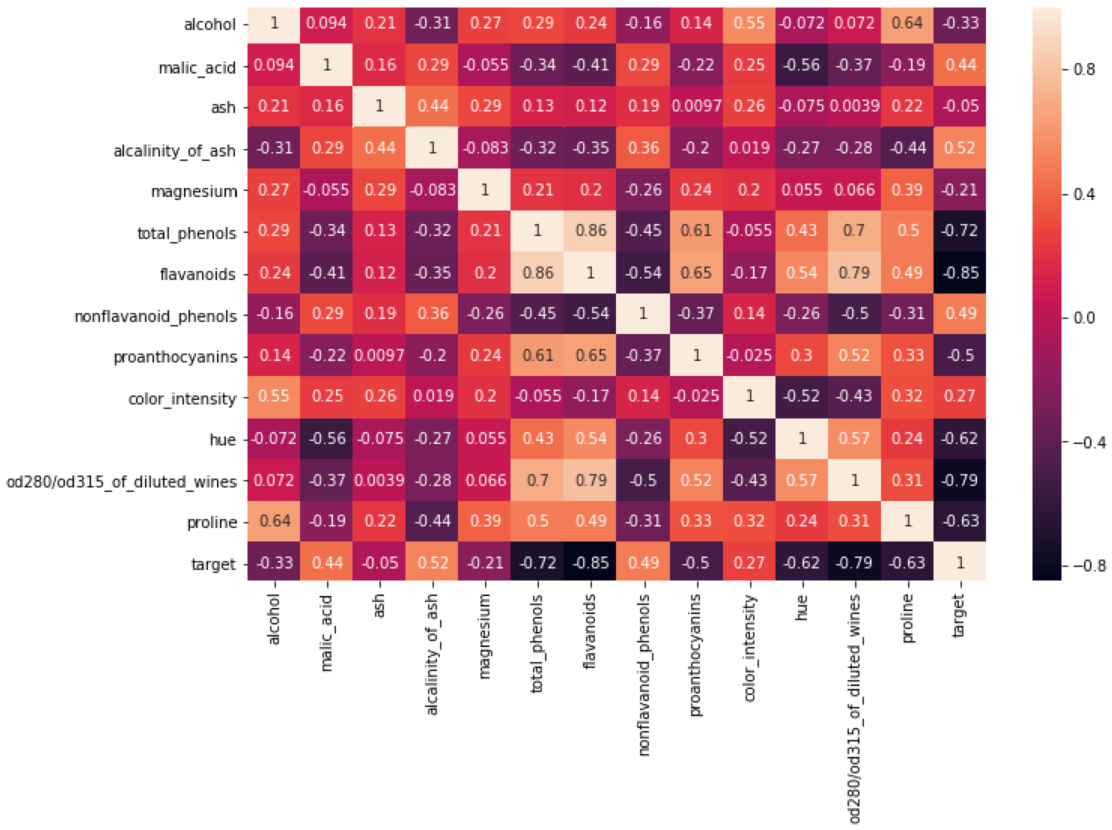
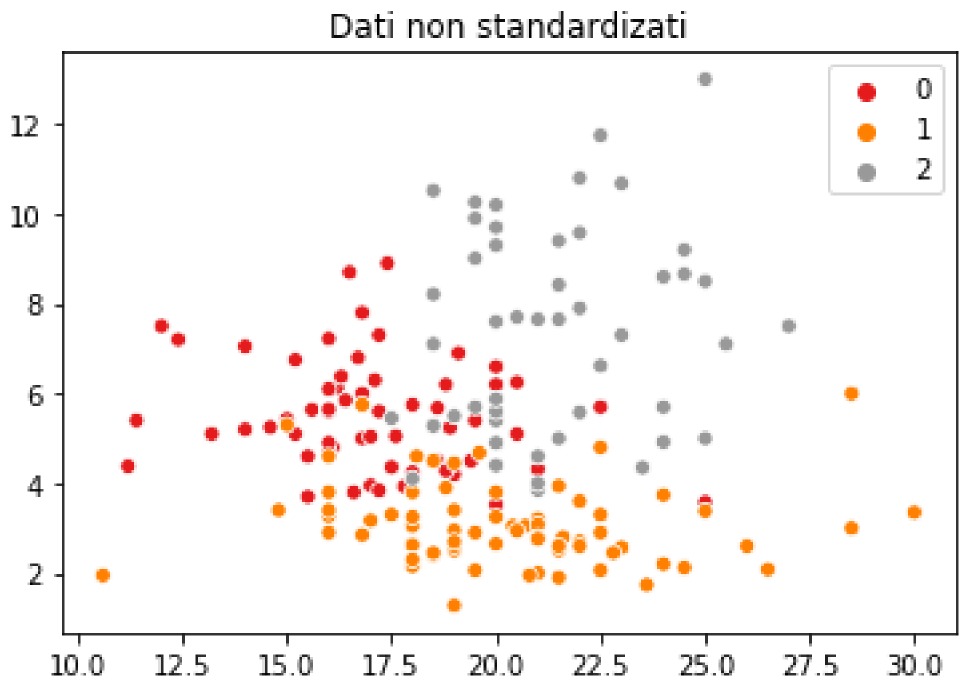


Figure 8: *Correlation matrix*



6.1 Knn again

Although the graph obtained looks better than the previous one, the accuracy achieved with the new distribution and the best K reaches only 0.7962 on the test set. This result appears strange because the graph of the points seems well separated from the borders. Perhaps this time the distribution of the train set doesn't represent the test set in a fine way.

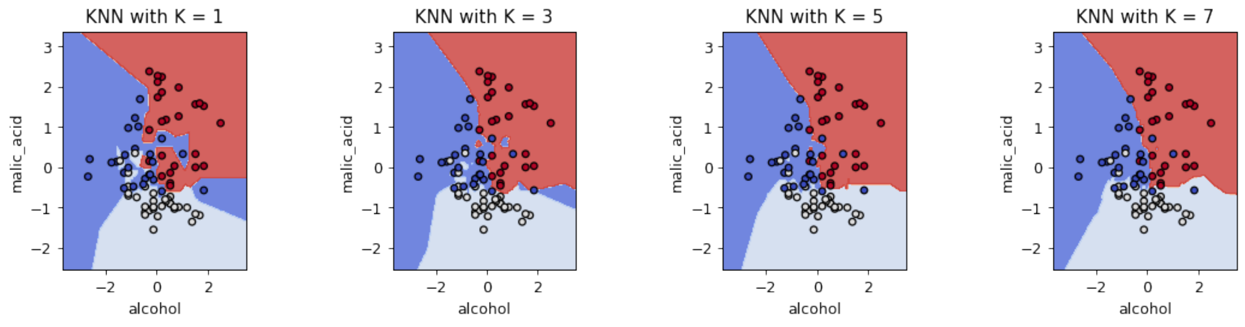


Figure 9: *Knn boundaries with various C + train data*

7 Linear svm again

Linear svm does not work properly when high values are assigned to C. In fact with C = [1e+3, 1e+4], that is when the svm works as a soft margin classifier, it has some problems in the correct recognition of the blue class. On the other hand, when C is low, the algorithm is unable to cut the white class from the other. Above all, the classifier's performance does not decrease and the score obtained with the best C value remains 0.833.

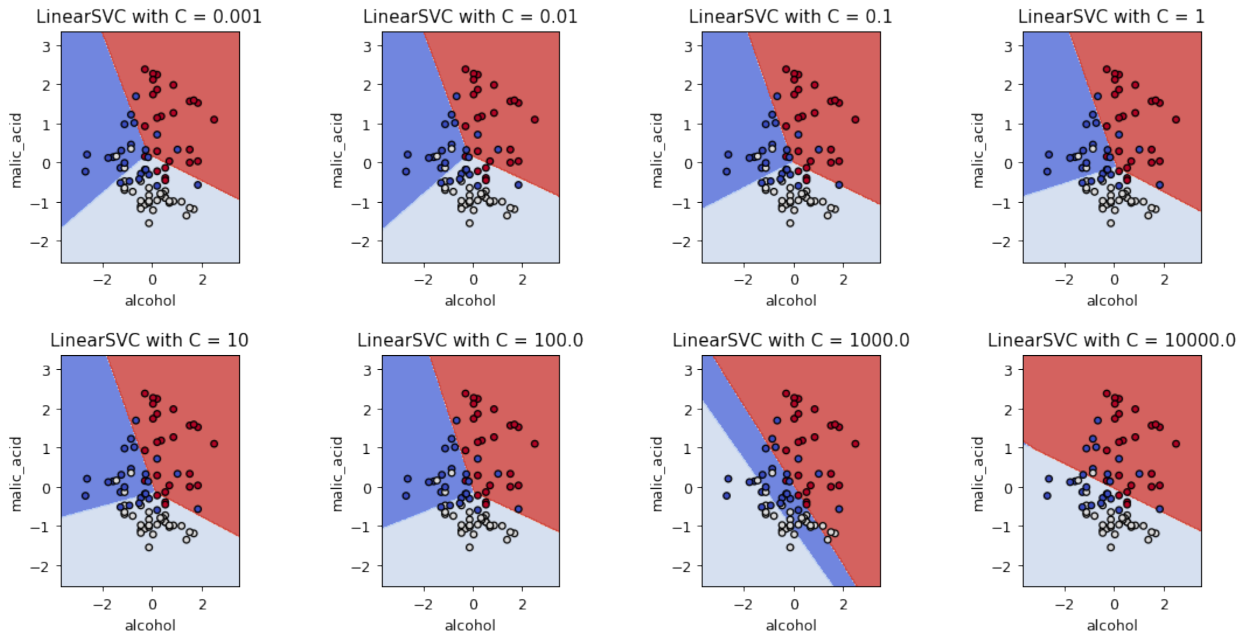
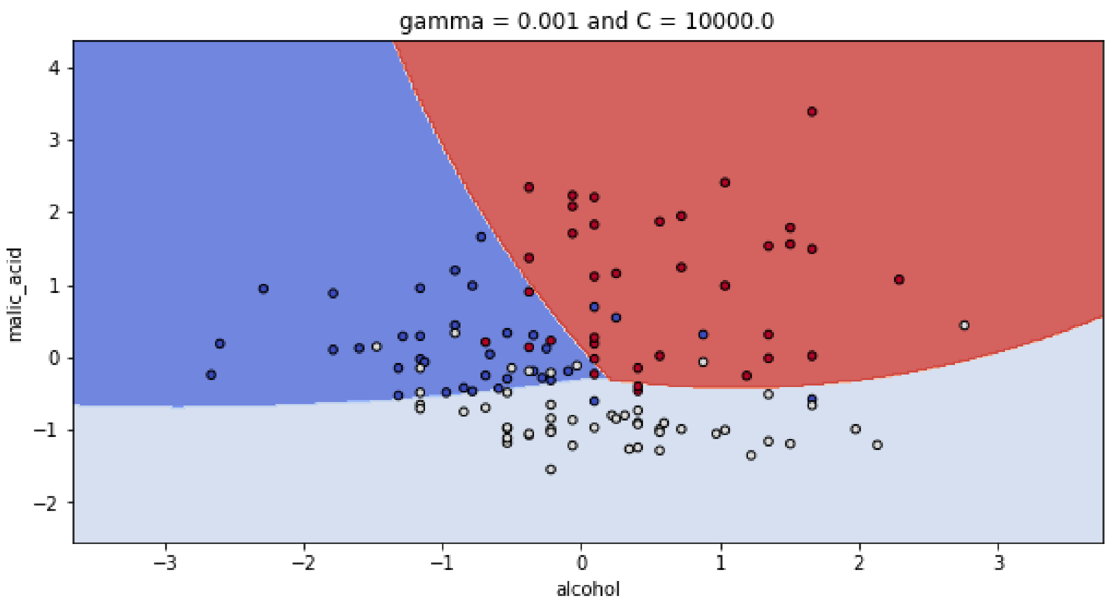


Figure 10: *LinearSVC boundaries with various C + train data*

8 SVM with rbf kernel again

To find the best possible balance between the value of C and that of Gamma, we once again use an approach based on "Grid-search". In this way, the non-linear kernel significantly improves the result. The new score on the test set has an accuracy that exceeds 0.9. At first, it may seem strange that this result has been achieved with an "extreme" value of both C and gamma. However, if we consider that with a high C value the model overfit the data and that a low gamma value instead underfit it, perhaps this result represents a good compromise between these two hyperparameters.



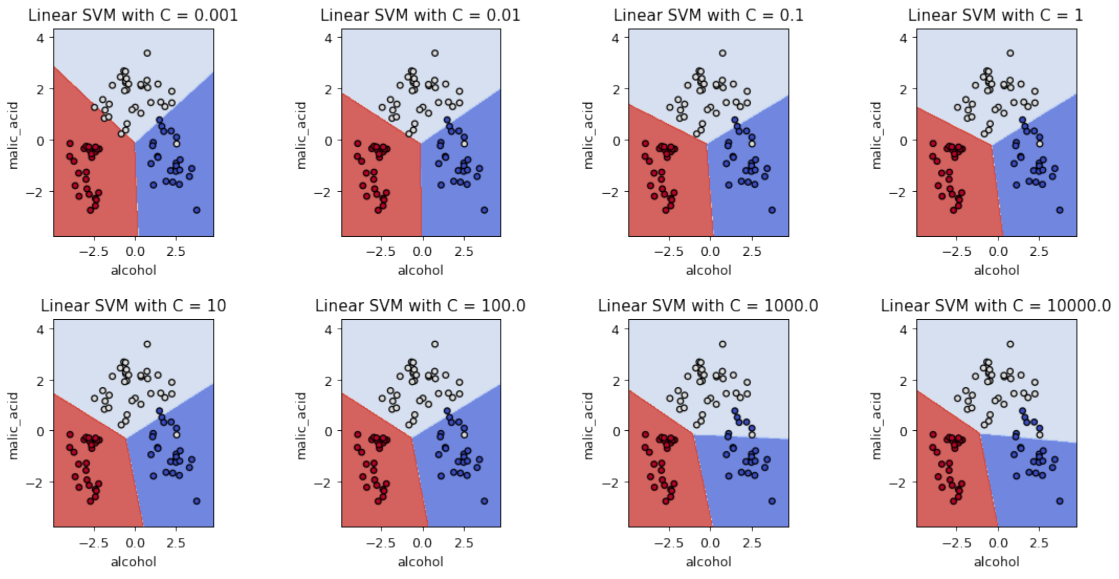
9 PCA

The last attempt to achieve a better result is not to choose two variables from the dataset but to apply a Principal component analysis. So, if $\widehat{x}$ represents our standardized samples, the first step in this method is to compute the variance-covariance matrix of $\widehat{x}$.

$$varcov(x) = \frac{< \widehat{x}^t, \widehat{x} >}{N_{sample}}$$

In this case, we get a 13X13 symmetric matrix. From this matrix, we can calculate the values of eigenvectors and eigenvalues. Eigenvectors sorted by eigenvalue values represent the directions in which the variance changes the most. To get a 2D representation of the result, only the first two main components are used. Finally, we project the samples on this 2D space. The result, highlighted by the graph below, is significantly better than before. So once you repeat all the old steps for each classifier, this time the accuracy score is always greater than 0.95.

9.1 Linear svm



```
[*]: Linear svm score on the test set with c = 0.01: 0.9814814814814815
```

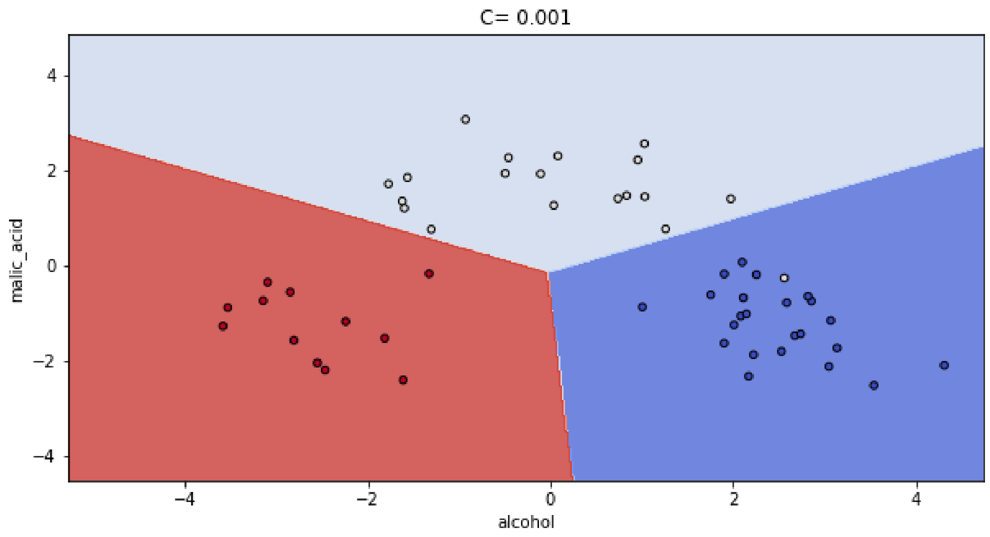
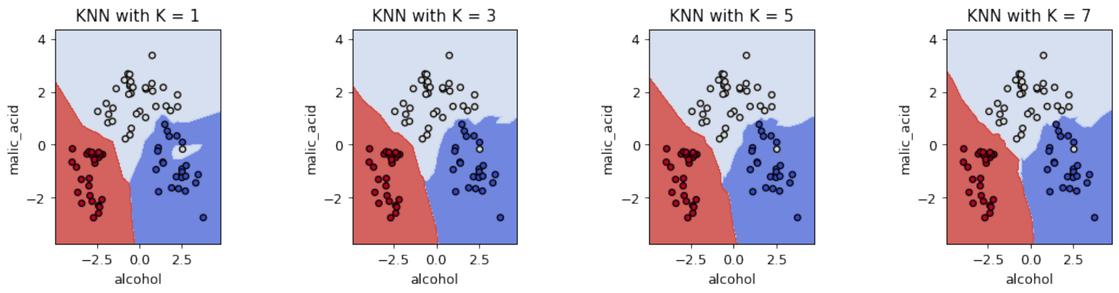


Figure 11: *LinearSVC boundaries on test data*

9.2 Knn



```
[*]: Knn score on the test set with K = 5: 0.9629629629629629
```

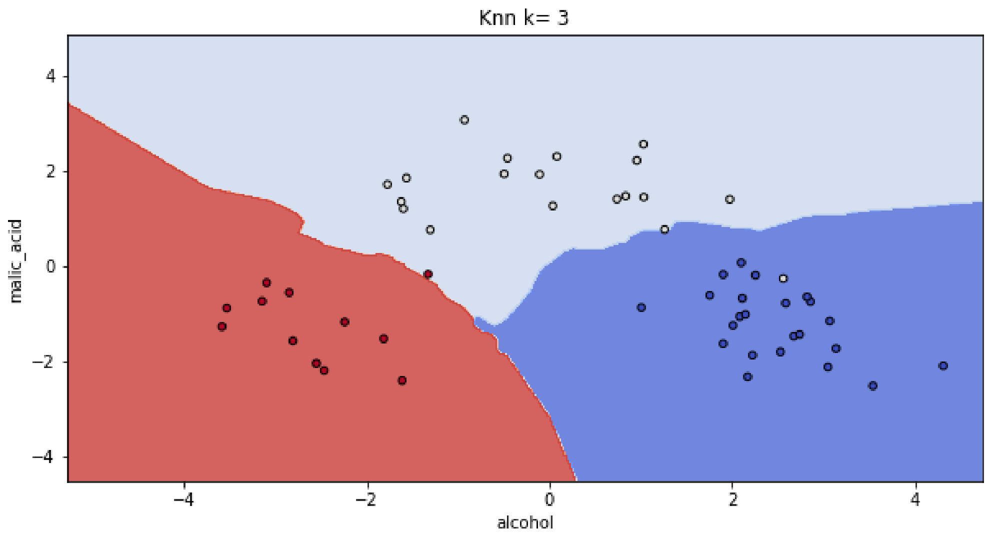


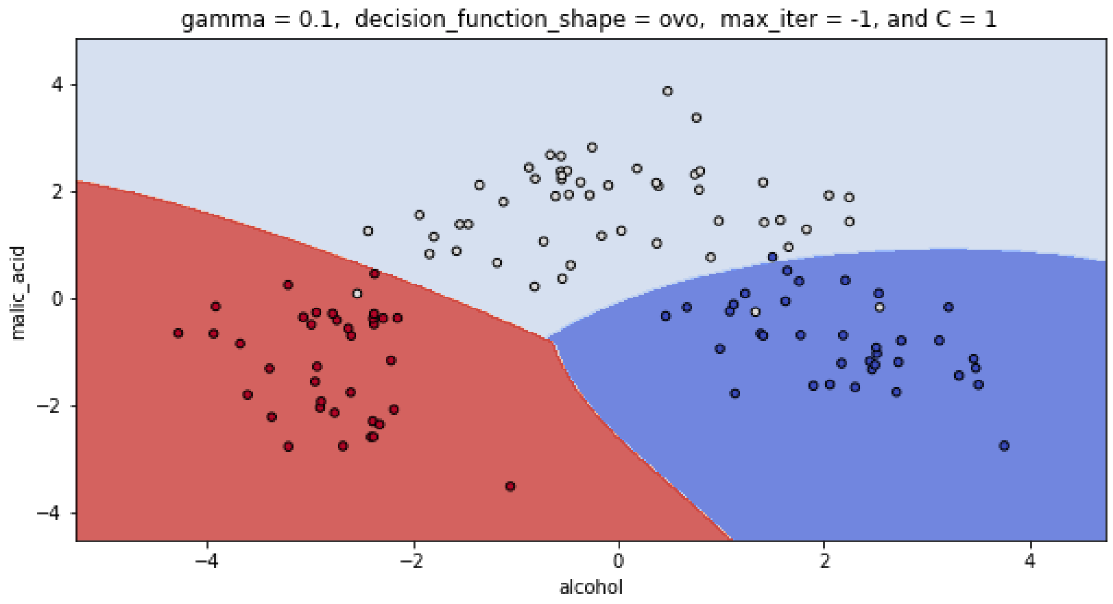
Figure 12: *Knn boundaries on test data*

9.3 Svm with rbf kernel

To try to improve the result with the RBF kernel, new hyperparameters related to the decision function and the maximum number of iterations were also tested. The result of "Grid-search", is not surprising. The classifier prefers the One-Versus-One (OVO) decision function which is always used in the multiclass problem and the algorithm prefers to have as much iterations as it can, and in fact the number of max-iter is -1.

```
decision_function_shape = ["ovo", "ovr"]
max_iter = [-1, 1e-2, 1e-3, 1e-4]
param_grid = [
  {'C': C, 'gamma': gamma, "decision_function_shape":decision_function_shape,␣
 ↪"max_iter":max_iter},
 ]
clf = GridSearchCV(SVC(), param_grid)
clf.fit(X_t_pca, y_t)
best = clf.best_params_
```

```
clf = SVC(C = best['C'], gamma = best['gamma'], decision_function_shape =␣
 ↪best["decision_function_shape"], max_iter = best["max_iter"])
clf.fit(X_t_pca, y_t)
y_pred_test = clf.predict(X_test_pca)
print(accuracy_score(y_test, y_pred_test))
plotDiffSVM([clf], X_t_pca, y_t, [best['C']], f"gamma = {best['gamma']}, ␣
 ↪decision_function_shape = {best['decision_function_shape']},  max_iter =␣
 ↪{best['max_iter']}, and C =", X_t_pca, y_t)
```



```
[*]: SVM with rbf kernel score on the test set with best parameters: 0.
 ↪9629629629629629
```

10 KNN vs SVM

Since the beginning of this report, the main differences between these two algorithms have been underlined, so the time has come to make a brief summary. Since we are dealing with a small dataset, it is difficult to notice the different speed in the training and forecasting phase of the two algorithms. Using the time class of the sklearn library, however, it becomes possible to calculate the time of the two phases. On average, svm with the RBF kernel is twice as fast as the knn algorithm. This seems perfectly reasonable because Knn, even if it doesn't have a training phase, has to calculate many distances for each point of the test set. Even considering memory usage, svm should win this comparison. Knn must remember every single point of the training dataset to calculate the distance, as already mentioned, instead svm keeps only the points that are useful for the supported vectors. This implies a huge memory saving. But we must also remember that the sklearn implementation of KNN uses the Johson-Lindenstrauss lemma. This lemma is very useful for this algorithm because it can store points from high-dimensional to low-dimensional Euclidean space with low distance distortion. But that's still not enough. It can be said that the Knn algorithm always gets a good score, comparable with both linear and non-linear SVM. This means that this algorithm can be used as a starting point for any simple exercise. SVM on the other hand, (especially if linear) is designed for large-scale problems and can be used to solve a very complex task in a reasonable amount of time.