Pattern Recognition & Machine Learning

Assignment 2023

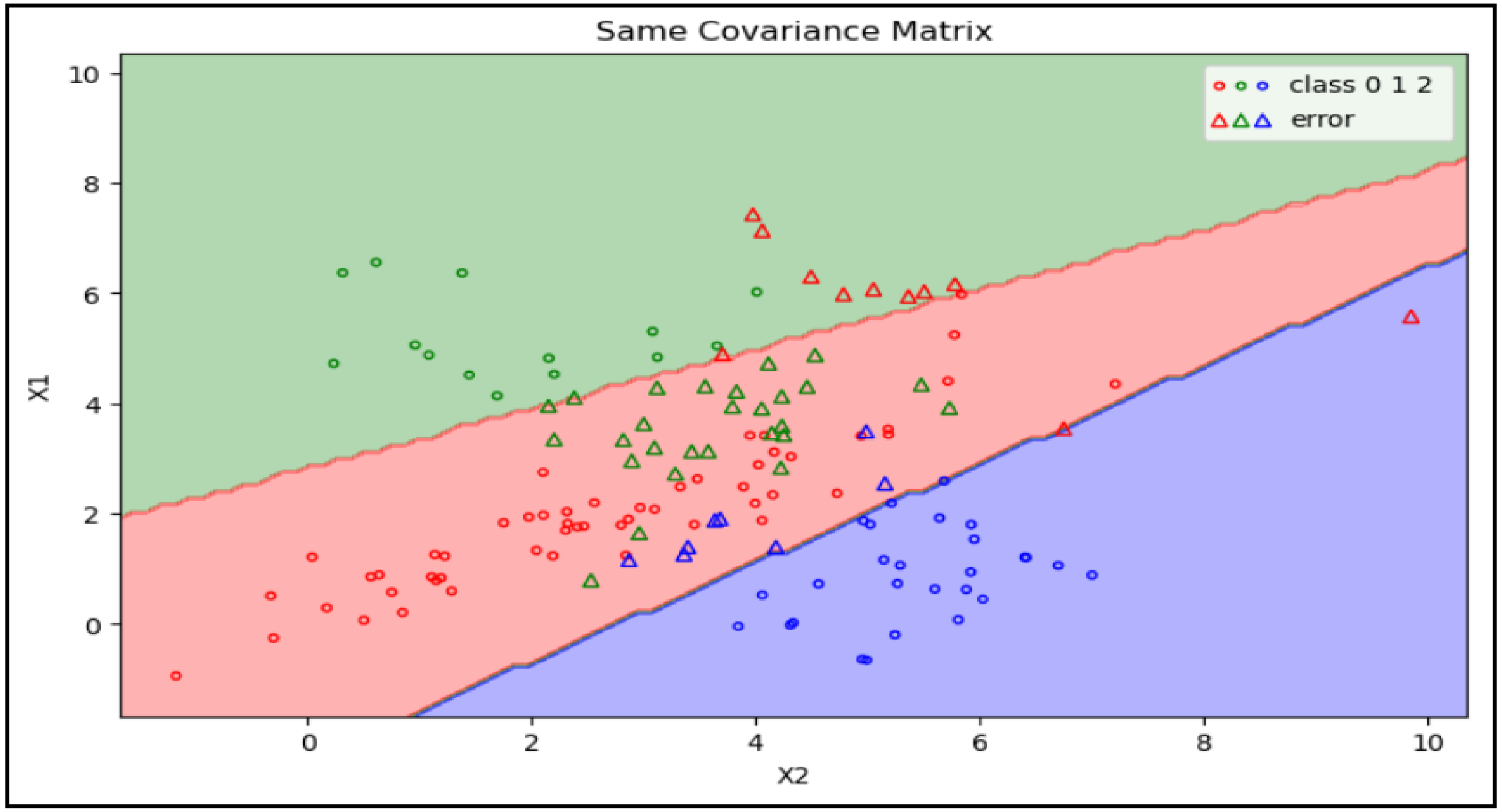
Grigoriou Stergios 9564
Zacharioudaki Danai 9418

grigster@ece.auth.gr
zachardd@ece.auth.gr

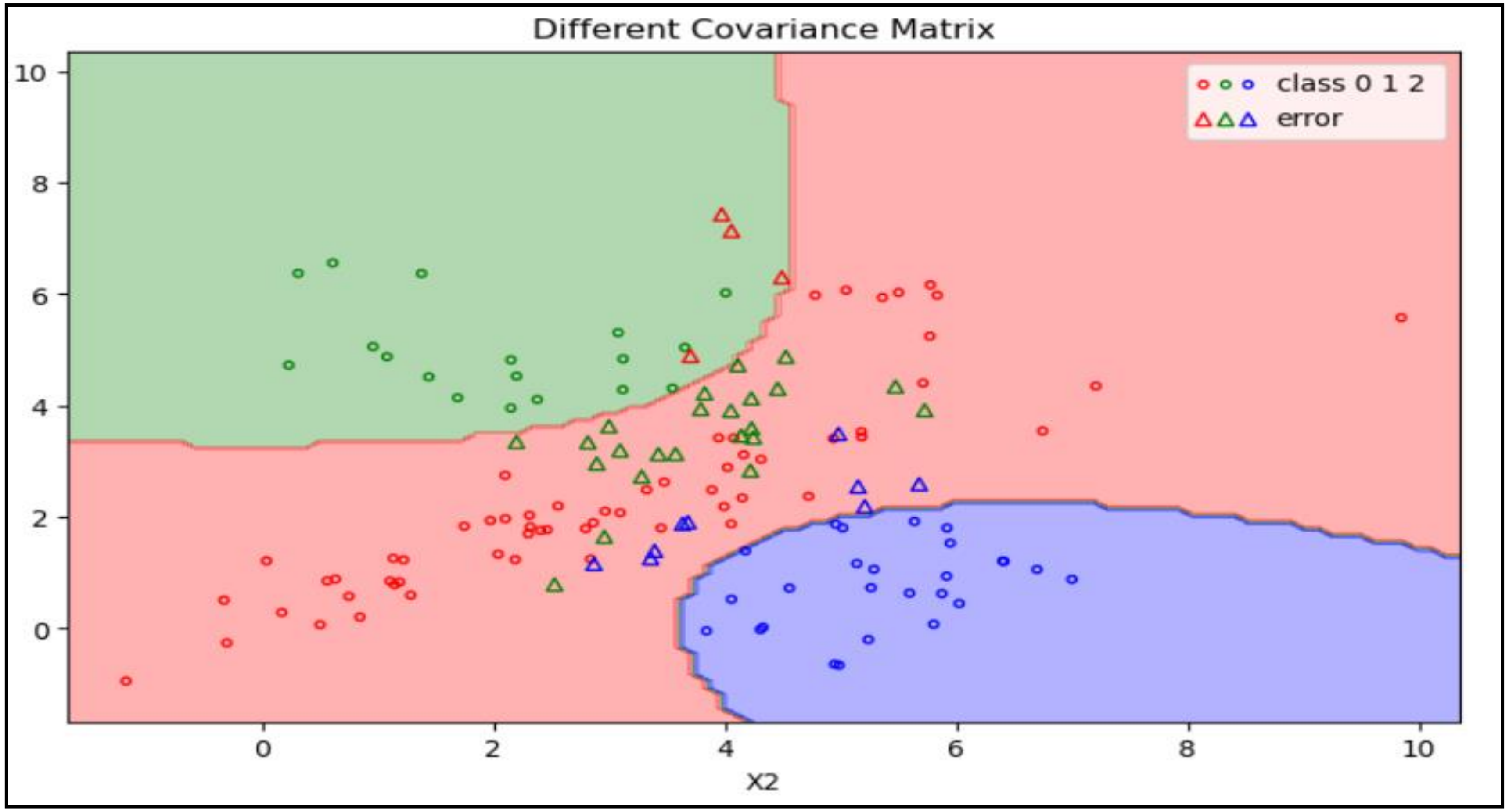
Part A: Bayes Classifier Implementation

- We created a class named *my_bayes_model()* to implement the maximum likelihood bayes estimator. We gave it a *train()* and a *predict()* method. And then we proceeded on training the two variations of the model on the train set and testing them on the test set as asked.
- We calculated the error rate of the models and kept the predicted labels to use on the plots for the markers.
- We created a function named *calc_regions()* to calculate the predicted regions of the models and plot them. It does so by predicting the labels of a mesh-grid and using the contour method.
- And for the plotting part we used matplotlib' s pyplot. Some lines of code that we also made into a function, primarily to be used on the parts to come.

Results



The predictions alongside the class regions of the Bayes model, assuming same covariance matrix among the classes for the features. (Red = 1,Green = 2, Blue = 3 and circles = correct classification, triangles = misclassification)



The predictions alongside the class regions of the Bayes model, assuming different covariance matrix for the features for each class . (Red = 1,Green = 2, Blue = 3 and circles = correct classification, triangles = misclassification)

Error Rates

- Finally calculating the classification error rate on the test set by dividing the number of incorrectly classified objects by the total number of objects

- For the Bayes model with same Covariance Matrix: error rate = 0.3286 while for the one with different Covariance Matrix: error rate = 0.2571

- The above results indicate a better performance of the Bayes Model with different Covariance Matrix for each class

Conclusions

- The assumption that each class is distributed differently seems to have a measurable impact on the model’s performance.
- We observe that assuming the same covariance matrix, the model predicts linearly, while the different covariance matrix gives non-linear predictions.
- Since the data are obviously not linearly separable, the superior performance of the model with the different covariance matrix is something reasonable.

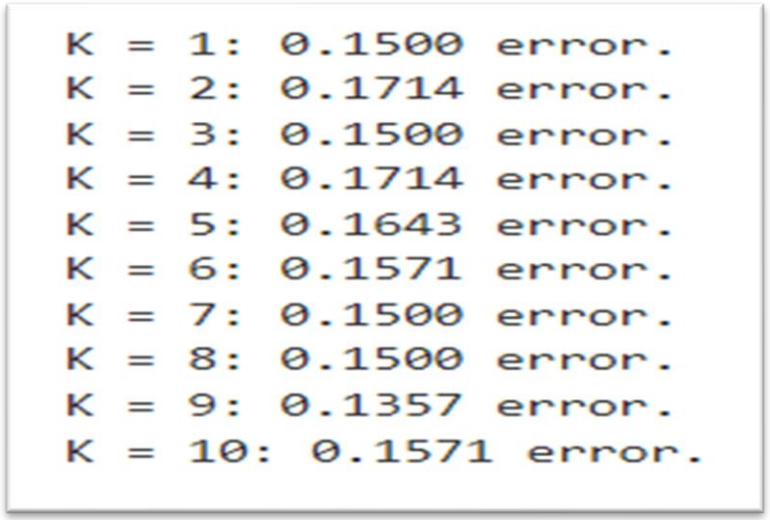
Part B: K-NN Classifier Implementation

- The implementation of the k-nearest neighbour classifier was done with the use of the KNeighborsClassifier of scikit-learn.

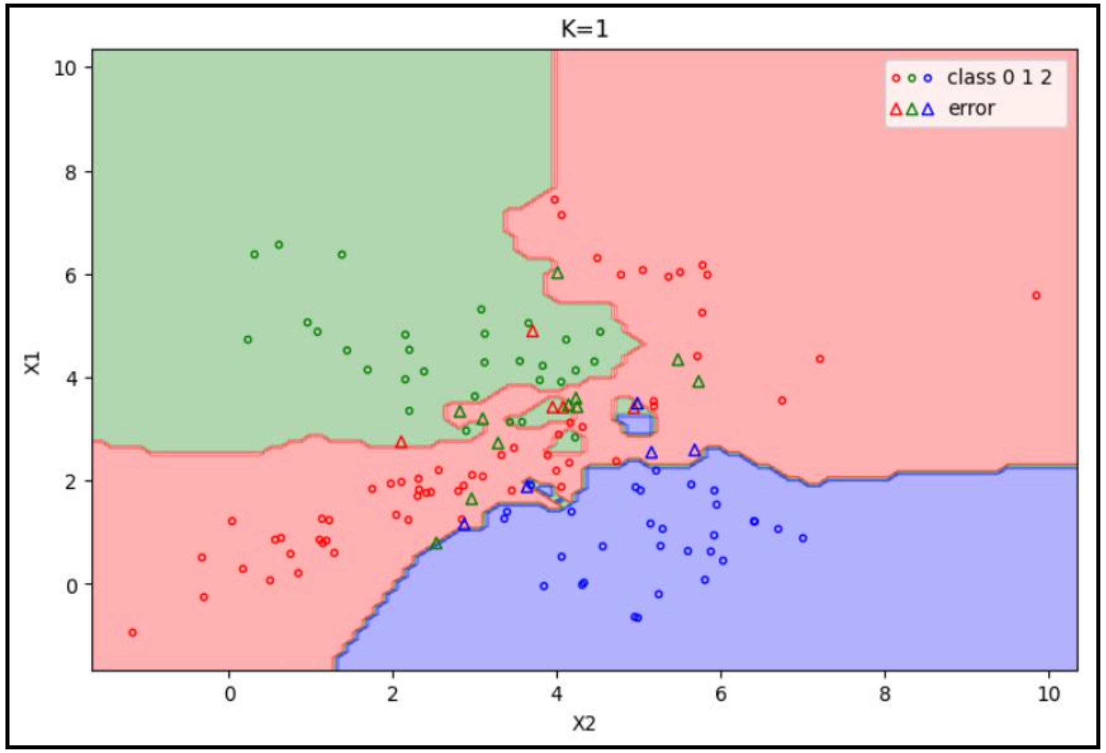
- The same 2 functions that were used for the part A were used here as well to plot the test data alongside the class regions.

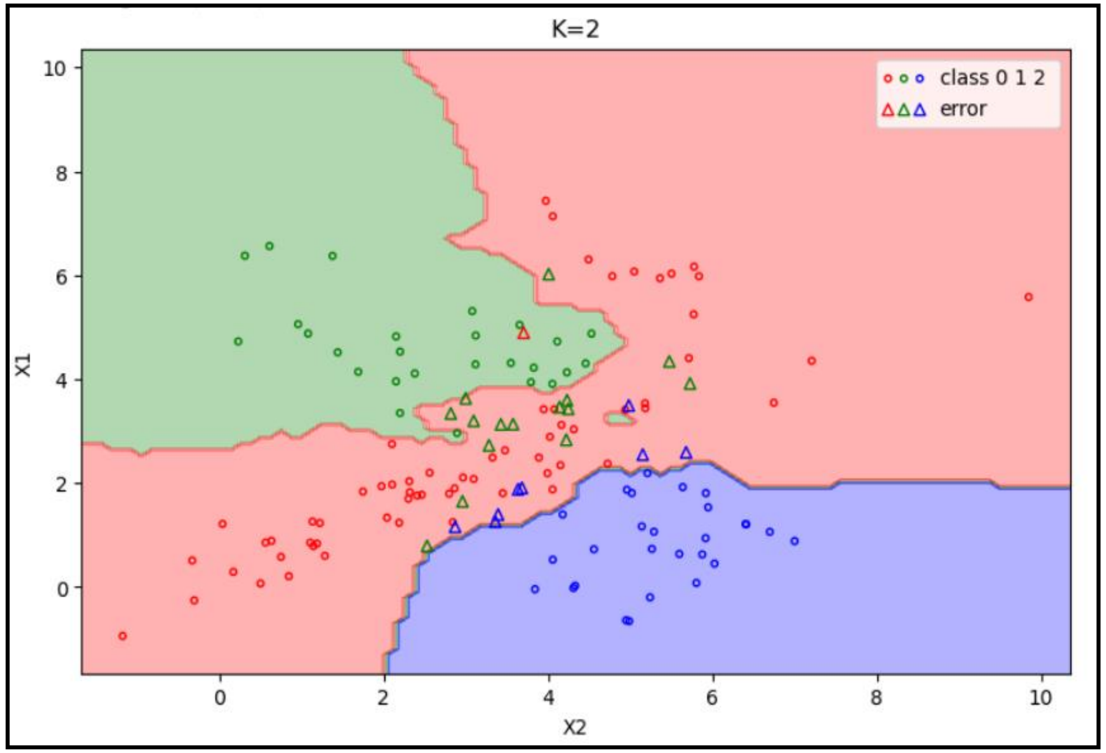
Results

- The error rates were calculated for each k from 1 to 10:



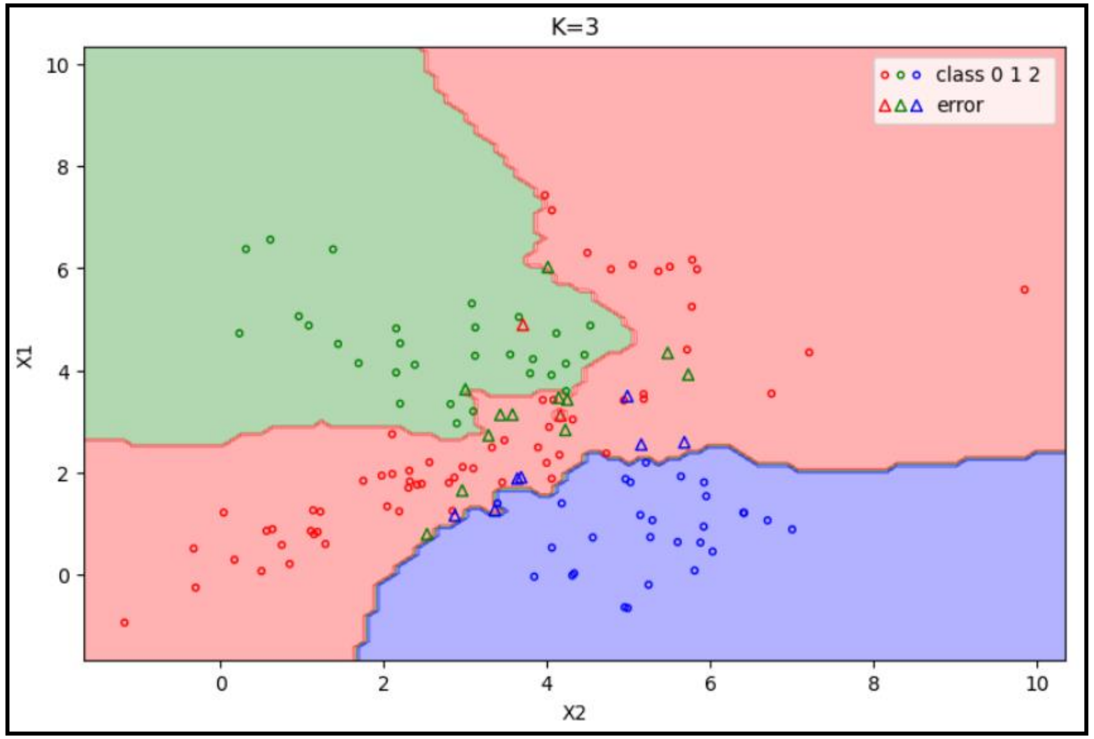
```
K = 1: 0.1500 error.
K = 2: 0.1714 error.
K = 3: 0.1500 error.
K = 4: 0.1714 error.
K = 5: 0.1643 error.
K = 6: 0.1571 error.
K = 7: 0.1500 error.
K = 8: 0.1500 error.
K = 9: 0.1357 error.
K = 10: 0.1571 error.
```

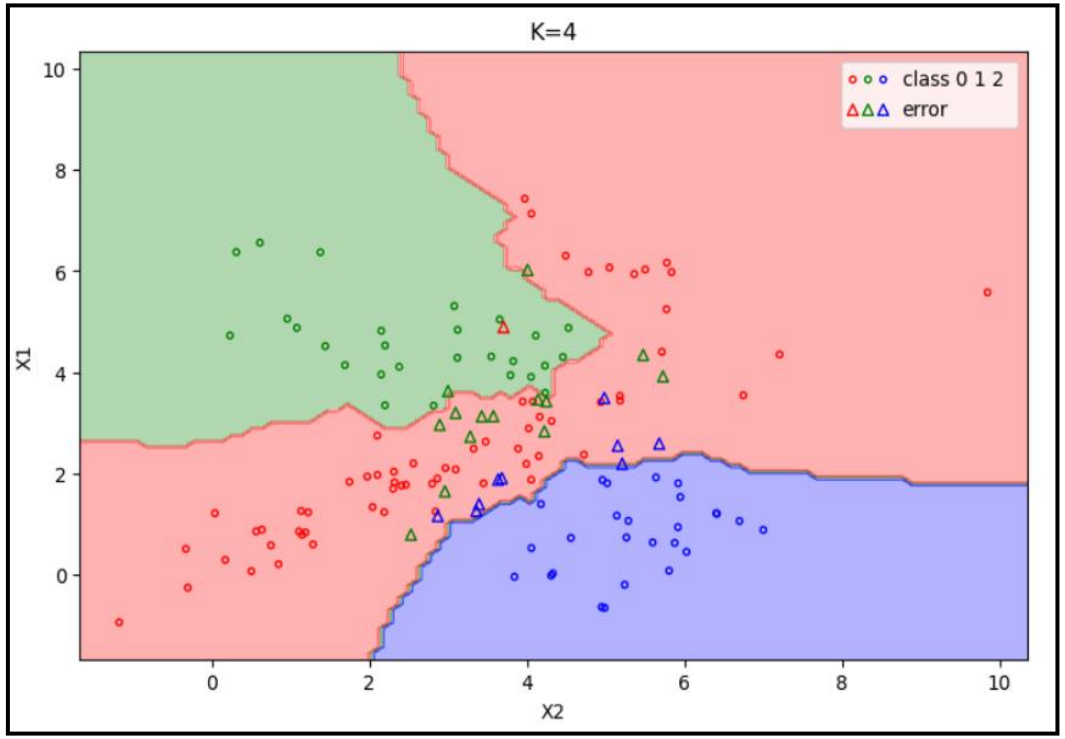




Classified test data for k=1 (left) and k=2 (right)

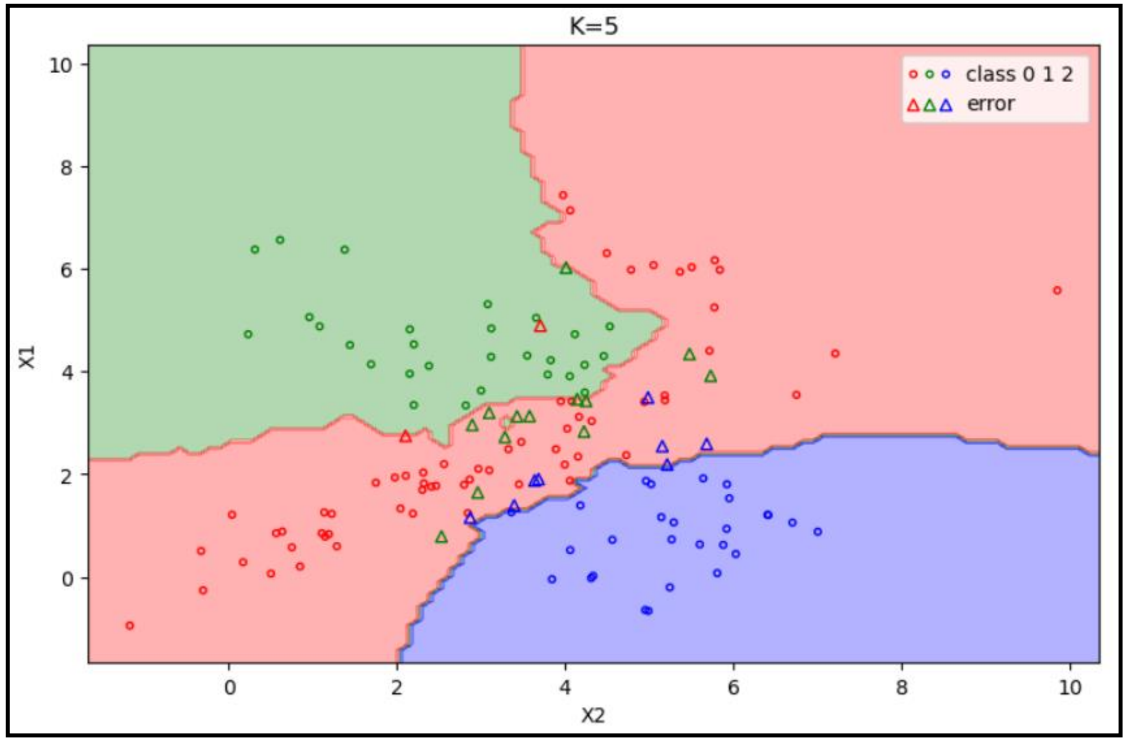
(Red = 1,Green = 2, Blue = 3 and circles = correct classification, triangles = misclassification)

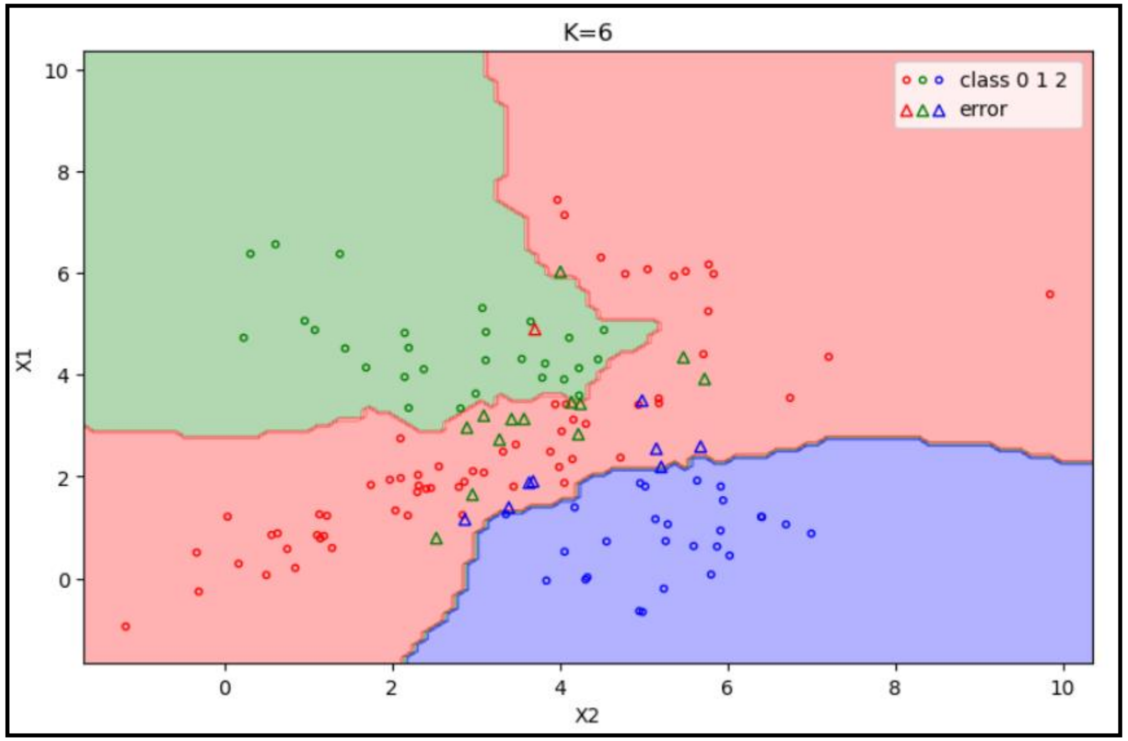




Classified test data for k=3 (left) and k=4 (right)

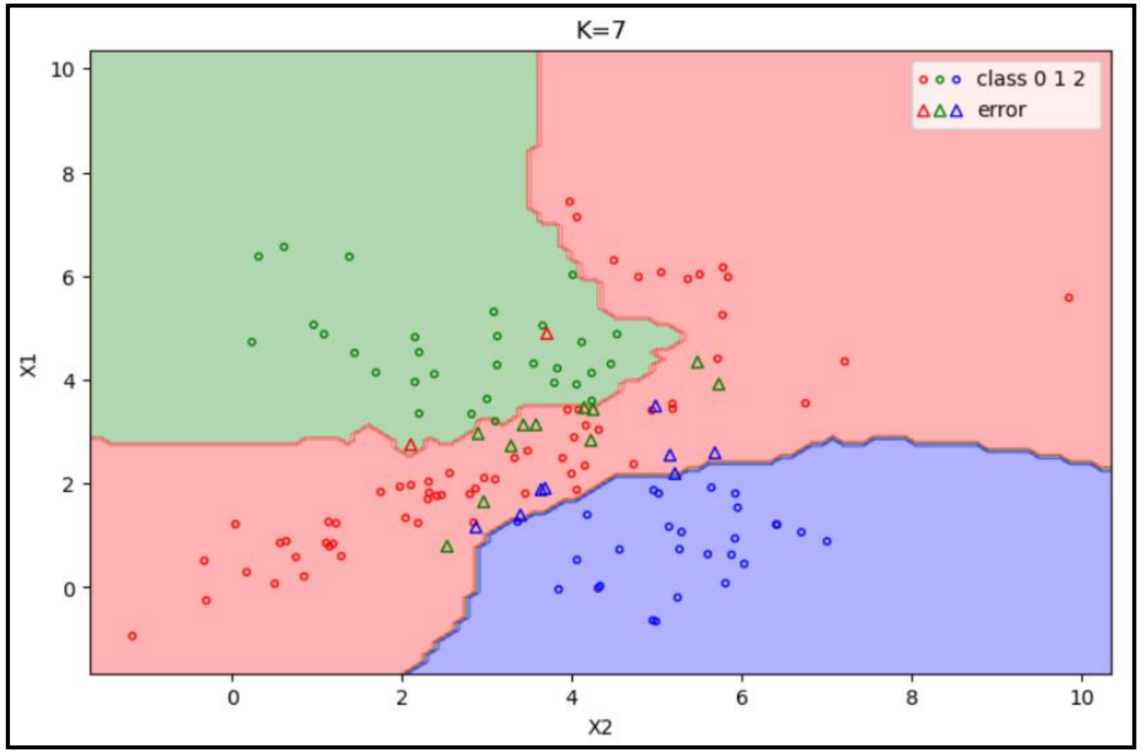
(Red = 1,Green = 2, Blue = 3 and circles = correct classification, triangles = misclassification)

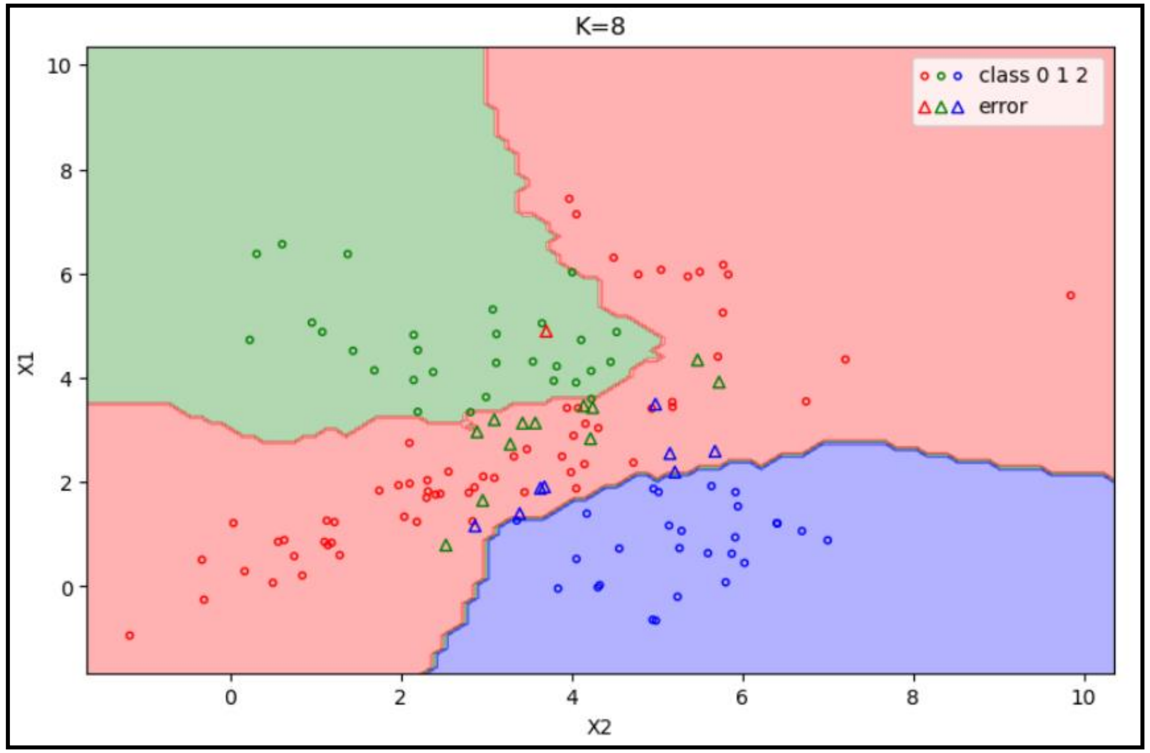




Classified test data for k=5 (left) and k=6 (right)

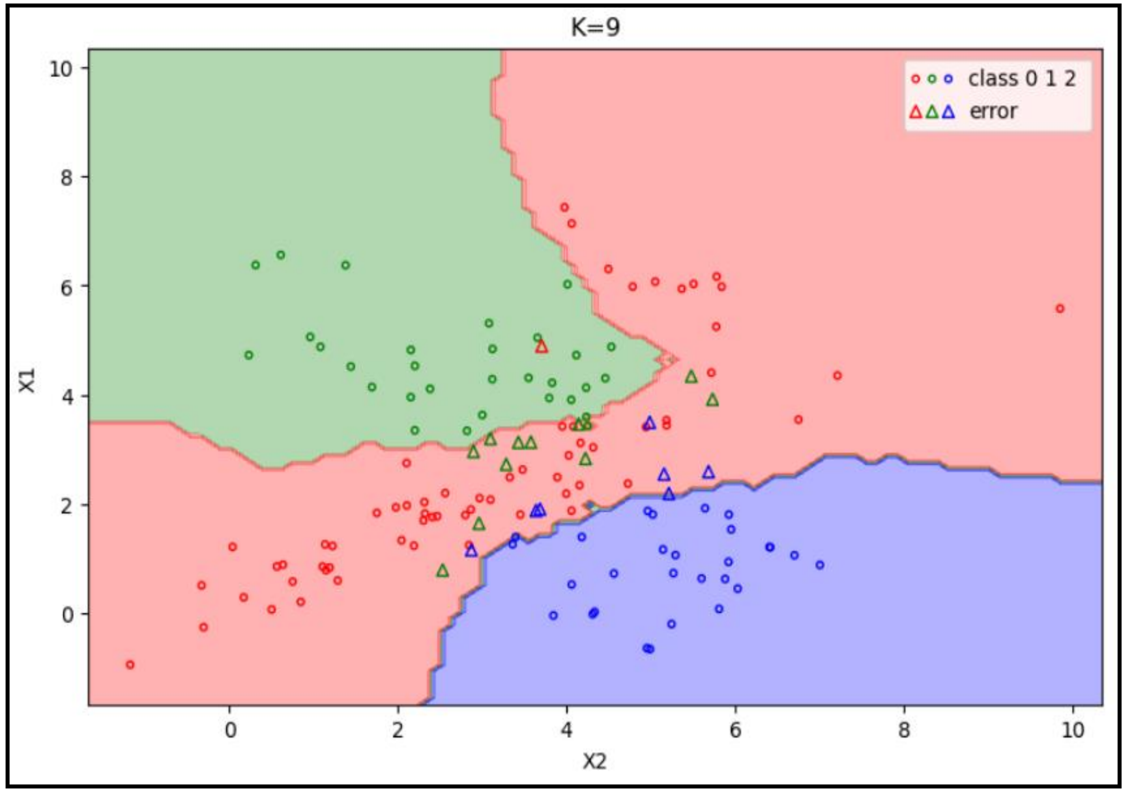
(Red = 1,Green = 2, Blue = 3 and circles = correct classification, triangles = misclassification)

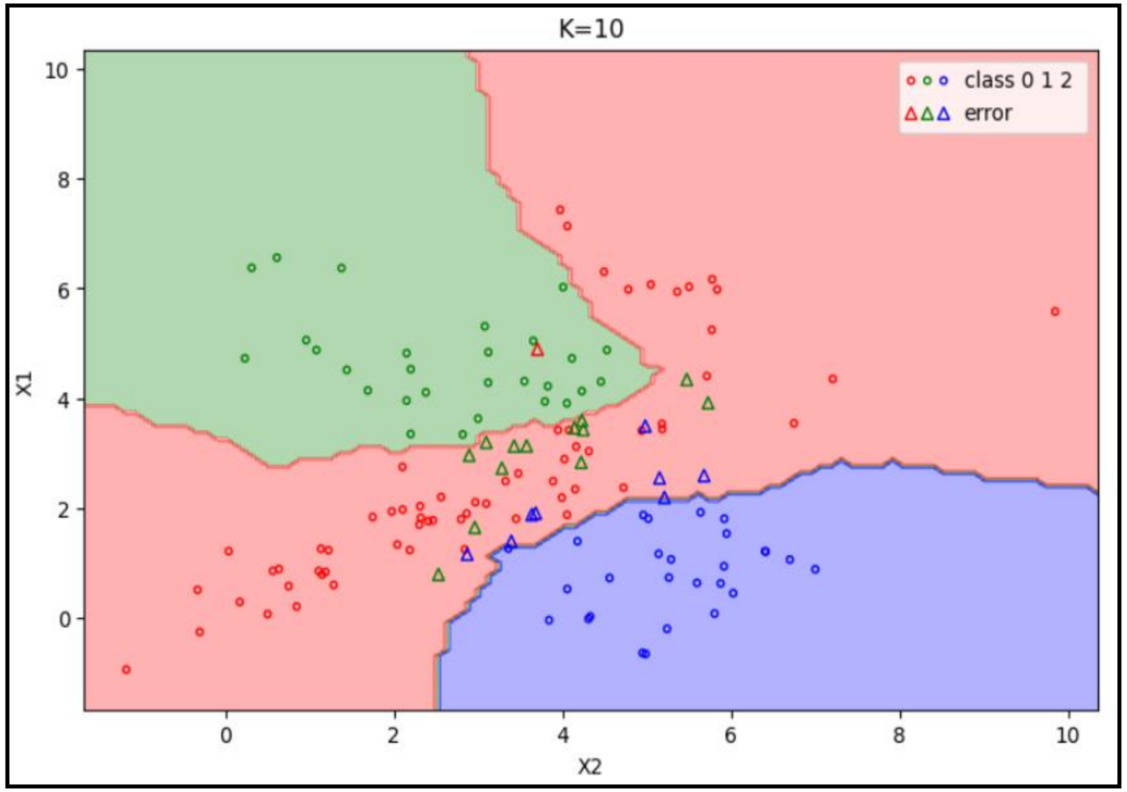




Classified test data for k=7 (left) and k=8 (right)

(Red = 1,Green = 2, Blue = 3 and circles = correct classification, triangles = misclassification)





Classified test data for k=9 (left) and k=10 (right)

(Red = 1,Green = 2, Blue = 3 and circles = correct classification, triangles = misclassification)

Conclusions

- We observe that the merrier the neighbours the smoother the predicted regions seem to be. But as long as the k parameter stays relatively low to the sample size the boundaries are always non-linear.

- Comparing the results of the K-NN Classifiers with those of the Bayes Classifiers we observe consistently lower error rates for the K-Nearest Neighbors Classifiers, with the lowest error rate being for K=9.

Part C: SVM Classifier Implementation

- For the implementation of the various SVM classifiers we used the implementation of the libSVM of scikit-learn. At first we did a relatively exhaustive grid search (since the small number of samples enabled us to do so) in order to fine tune the hyperparameters of the linear and the RBF kernel SVCs. The detailed values that we searched for can be seen on the first 2 lines of the SVM block.

- Then we proceeded on plotting the predictions of the tuned models as well as a 3 C values x 5 γ values grid of plots. To do so we modified slightly the *plot_class_data()* function in order to plot both the test and the train set predictions and mark with a cyan x the support vectors.

Results

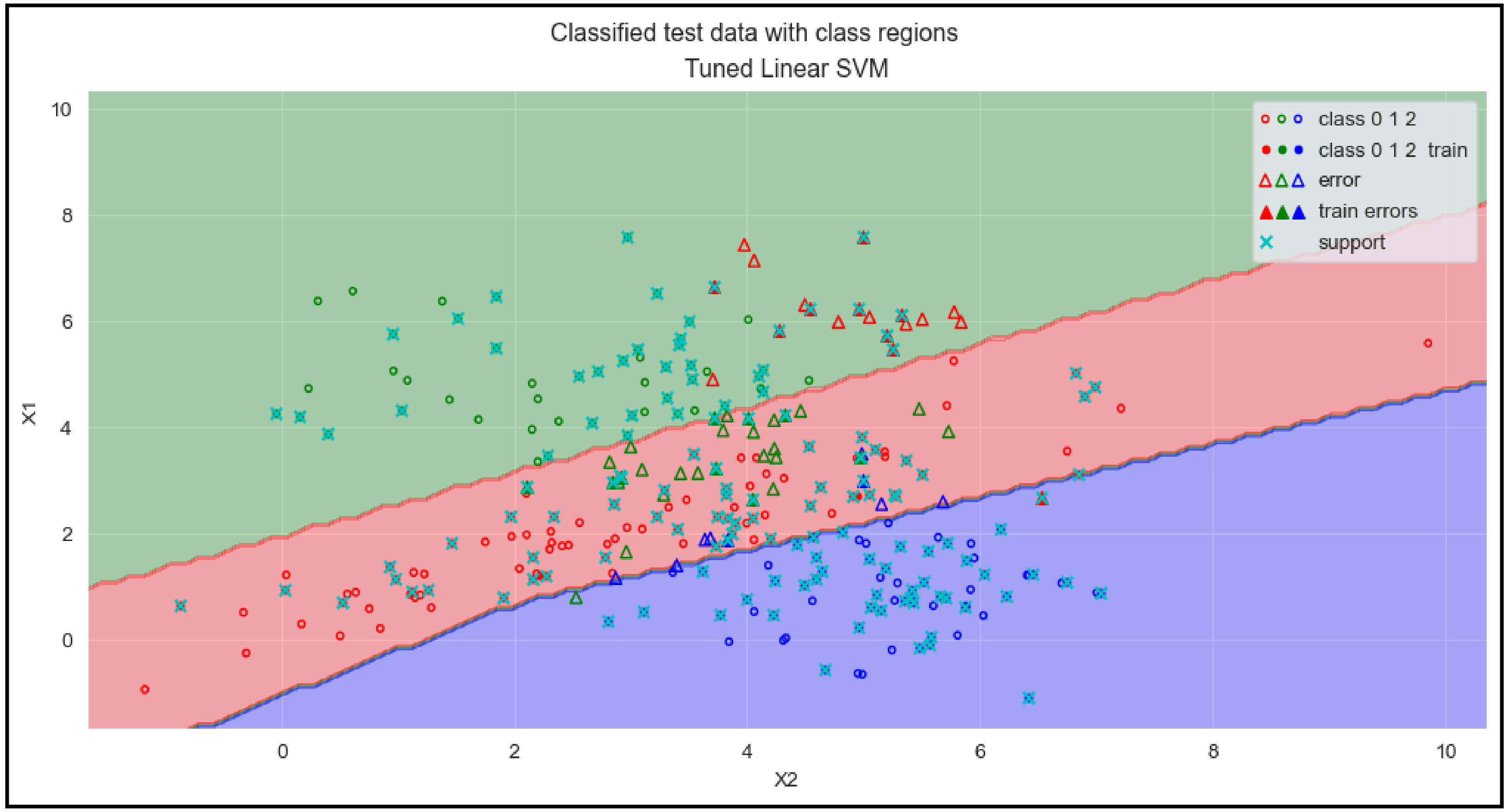
C\γ	**1e-6**	**1e-3**	**0.1**	**10**	**100**
0.01	0.5429	0.5429	0.2857	0.5429	0.15
10	0.5429	0.5429	0.2857	0.1429	0.5429
1000	0.5429	0.2857	0.1929	0.15	0.2714

Error Rates for different combinations of C and γ parameters for RBF kernel SVM classifiers.

- Error rate for linear SVM: 0.2643 (C = $2^{8.2}$)

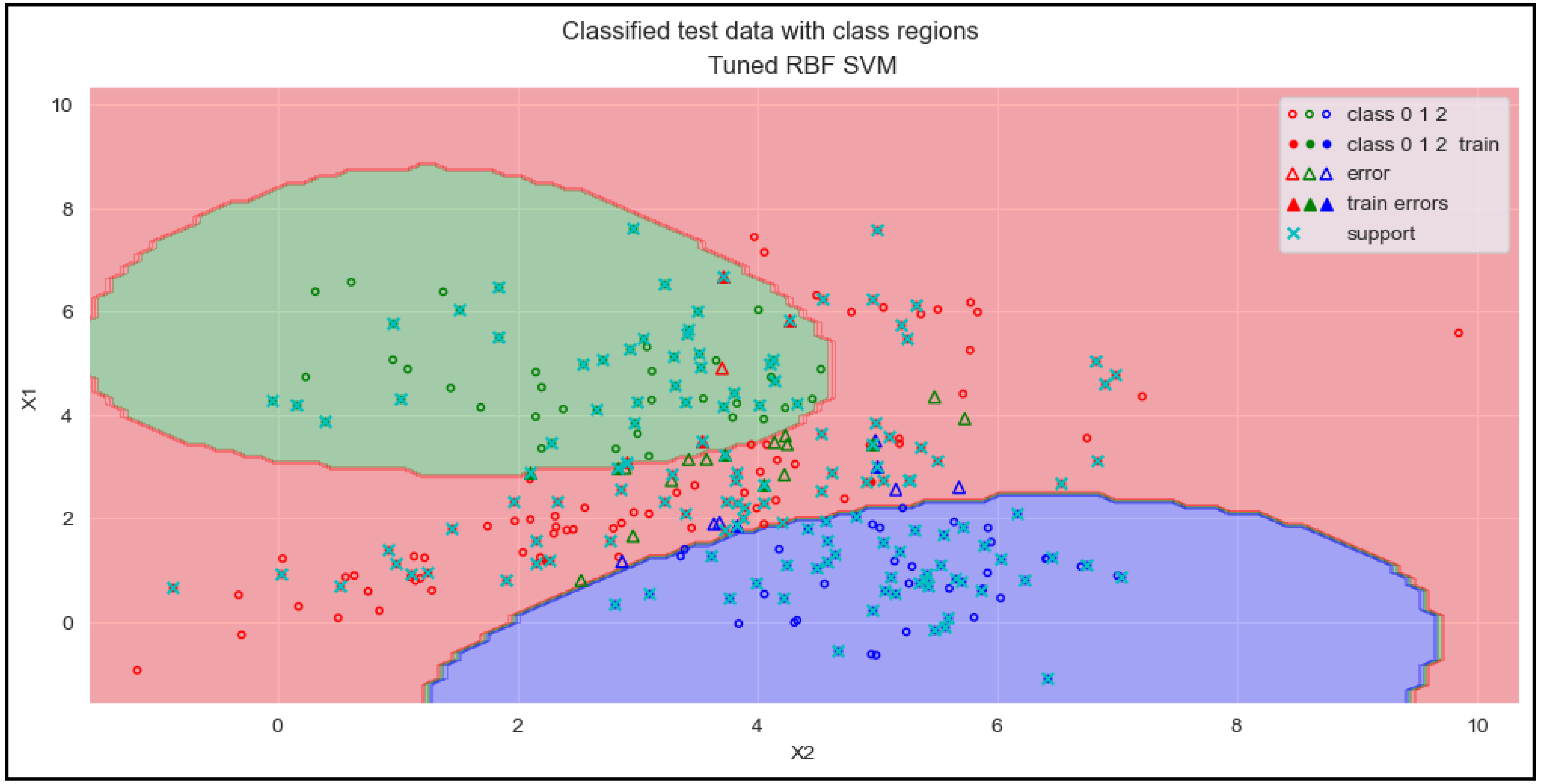
- Error rate for RBF SVM after 5-fold cross validation hyper-parameter tuning: 0.1357 (C = $2^{1.5}$, gamma = $2^{-3.8}$)

- We can now observe how hyperparameter tuning results to better performance of the classifier



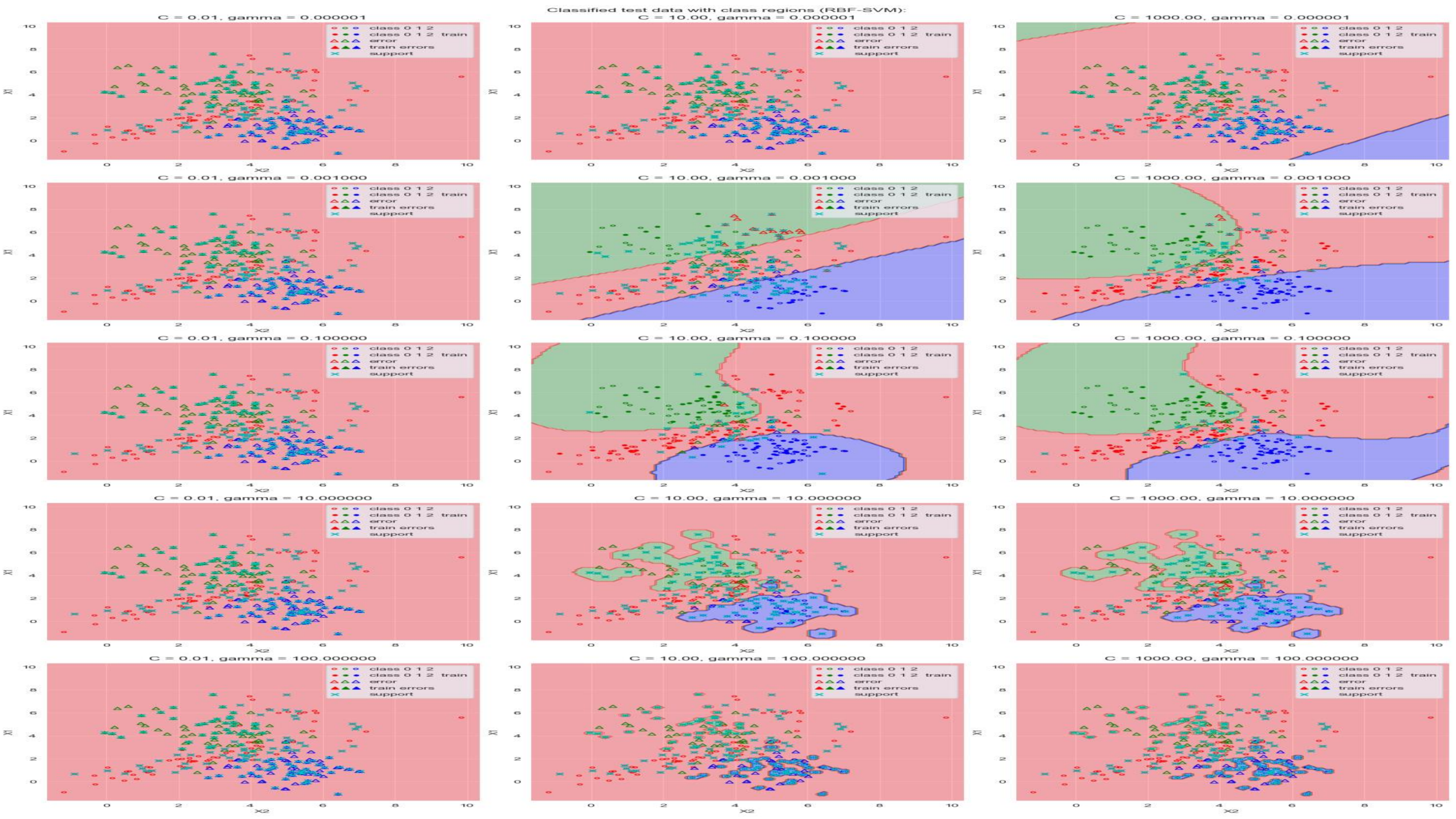
Classified Data with the tuned Linear SVM with one vs one classification schedule

(Red = 1,Green = 2, Blue = 3, circles = correct classification, triangles = misclassification, filled = train set, empty = test set, x = support vectors)



Classified Data with the tuned RBF SVM with one vs one classification schedule

(Red = 1,Green = 2, Blue = 3, circles = correct classification, triangles = misclassification, filled = train set, empty = test set, x = support vectors)


Classified test data with class regions (RBF-SVM):
C = 0.01, gamma = 0.000001
C = 10.00, gamma = 0.000001
C = 1000.00, gamma = 0.000001
C = 0.01, gamma = 0.001000
C = 10.00, gamma = 0.001000
C = 1000.00, gamma = 0.001000
C = 0.01, gamma = 0.100000
C = 10.00, gamma = 0.100000
C = 1000.00, gamma = 0.100000
C = 0.01, gamma = 10.000000
C = 10.00, gamma = 10.000000
C = 1000.00, gamma = 10.000000
C = 0.01, gamma = 100.000000
C = 10.00, gamma = 100.000000
C = 1000.00, gamma = 100.000000
class 0 1 2
class 0 1 2 train
error
train errors
support
X1
X2

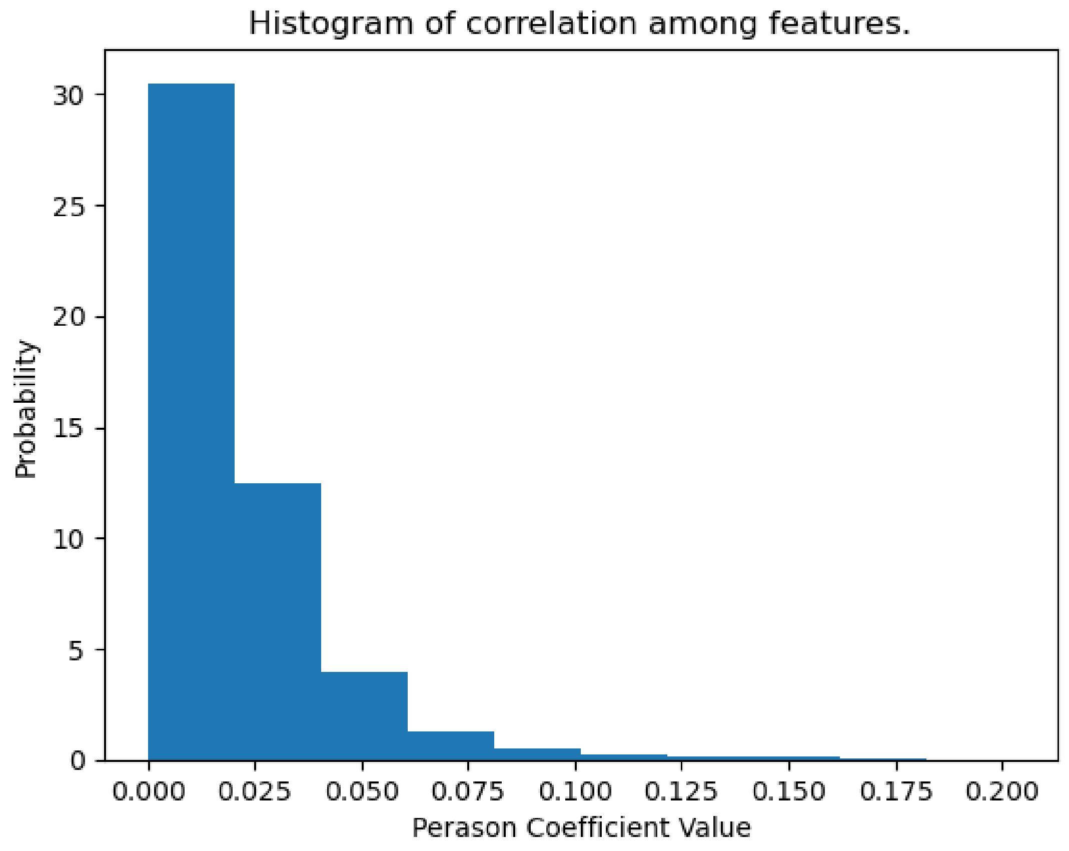
Conclusions

- First, we observe that the “tuned” models used almost all the trainset as support vectors. We are not sure if this is a bug of the implementation, but in the multiple models plotted there are models were this doesn’t happen, thus we believe its just how it works. Although we would expect the linear model to have only 9 support vectors.
- In general, we observe what we did with the previous classifiers, the data seems to be non-linearly separable and thus the linear version of the model tends to be inferior for this specific problem.
- The tuned RBF has parameter values in the middle of the grid we presented ($C= 2.8$ and $\gamma = 0.07$), and achieves the same classification error as the 9-NN model. We can see that it has purposely misclassified some of the train-data although they appear to be support vectors, and this is due to the relatively low C, that forces smoother decision boundaries.

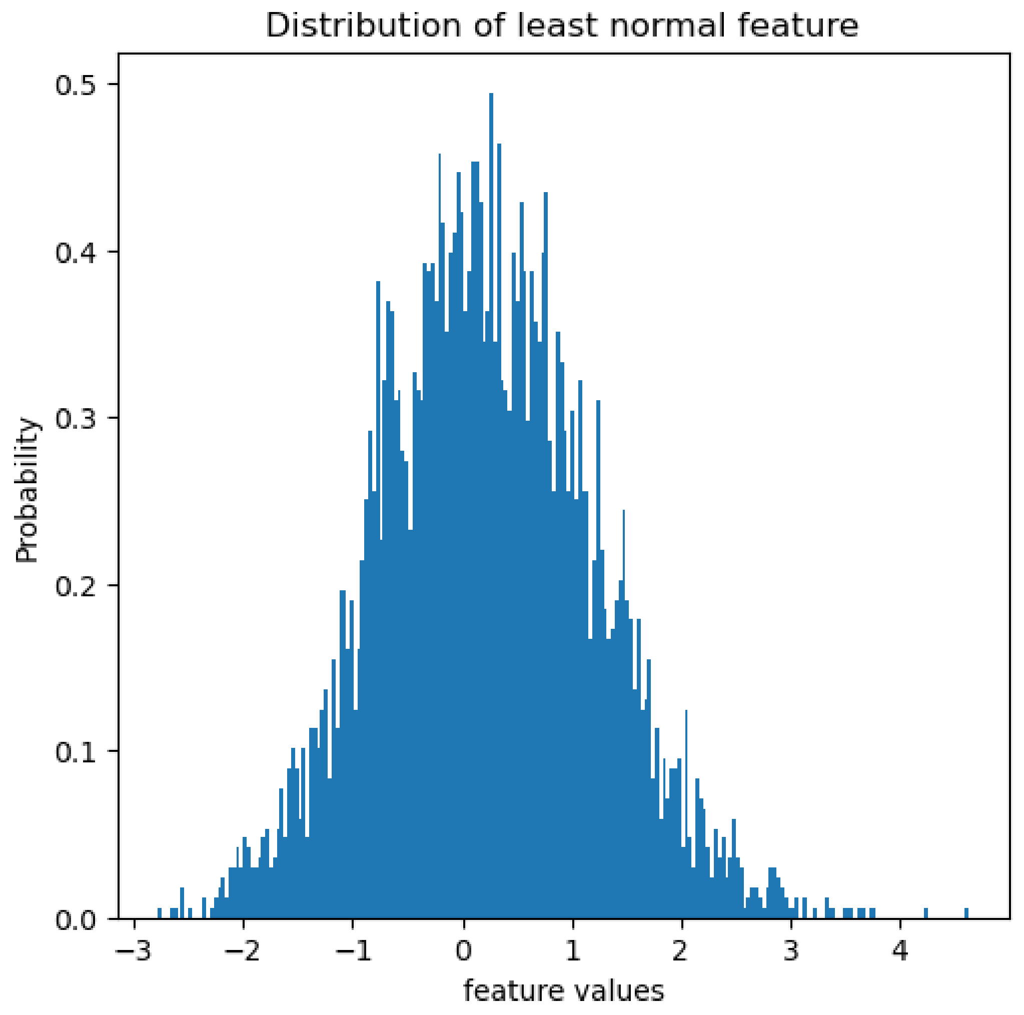
- To comment on the different parameter plots, we observe that extreme values for each of the parameters tend to shadow the influence of the other parameter. For instance, when the C value is low (0.01) , no matter how large the γ parameter becomes the decision boundary is so smooth that is out of the range of the neighborhood of the data. Similarly, when the γ parameter is extremely low even with large C (1000) we observe the same as before, because the variance of the bell is a lot bigger than what it should be.
- In the end it is just a matter of finding the right balance between regularization and how picky the kernel will be. From the table with the error rates, we can see that the best models are in a hypothetical sub-diagonal of the matrix.

Part D: Our Classifier Data

- We performed a few descriptive statistics tasks on our data before starting experimenting with statistical preprocessing.
- At first, we observed the correlations among the features. To do so we calculated the Pearson's coefficient and then plotted a histogram of the absolute correlation values. We see that even the maximum correlation between 2 features is relatively low (0.2) and doesn't imply strong dependance if it does at all.



- Next step, we searched for gaussianity. To be perfectly honest, this part we performed it with the chi2gof test of Matlab and found that only for 39 features we can be 95% certain that they are not normally distributed. Since the project is on python, we then used the Shapiro-Wilk test of sciPy, which is a stronger test and specifically designed for testing the goodness of fit to the standard normal. So before performing the test we standard scaled our features.
- Since, it's a more optimized test we demanded for 99% certainty of normality. The results, where that only 57 features can be said not to be normal at this significance level.
- All in all, we have a dataset that is mainly uncorrelated gaussian features.
- On the left we observe, feature that we can be most certain about not being normal, based on the SW test.



Class	1	2	3	4	5
Percentage	22.58	20.68	19.84	19.62	17.28

- We then experimented with PCA outlier detection alongside with PCA and plain PCA. Which seemed to slightly increase performance on selected model.

- In the end the best results were given by largely discarding the least important components and keeping only those that describe 18% of the features variation. Although this number is the one that maximized our (already chosen with 80% PCA) classifier's performance, it is possible that a most thorough and from scratch approach would give better results.

- So, to summarize, we standard scaled the data and the we performed PCA keeping only 31 features that describe 18% of the total variance. This preprocessing pipeline is justified, because the PCA demands centered data, and a large amount of our feature variance can be considered additive gaussian noise.

Looking for a Model

- We tried various models with lots of different parameters. The code can mostly be seen on the commented cells at the end of the notebook. For simpler models (Naïve Bayes, K-NN, SVM) we evaluated them and tuned them using 5-fold cross-validation. For some attempts building a simple 3-layer-NN, we just tried with simple validation to tune the hyperparameters (layer size, learning rate, loss,optimizer,etc).
- The best 2 models can be seen in a stratified 5-fold cross-validation loop on the main part of the code and they are a Gaussian Naïve Bayes and an RBF-SVM.
- The scoring that we used to determine the best classifier was accuracy and we used weights on the training of the classifiers in order to handle the slight imbalance of the dataset, between class 1 and class 5.

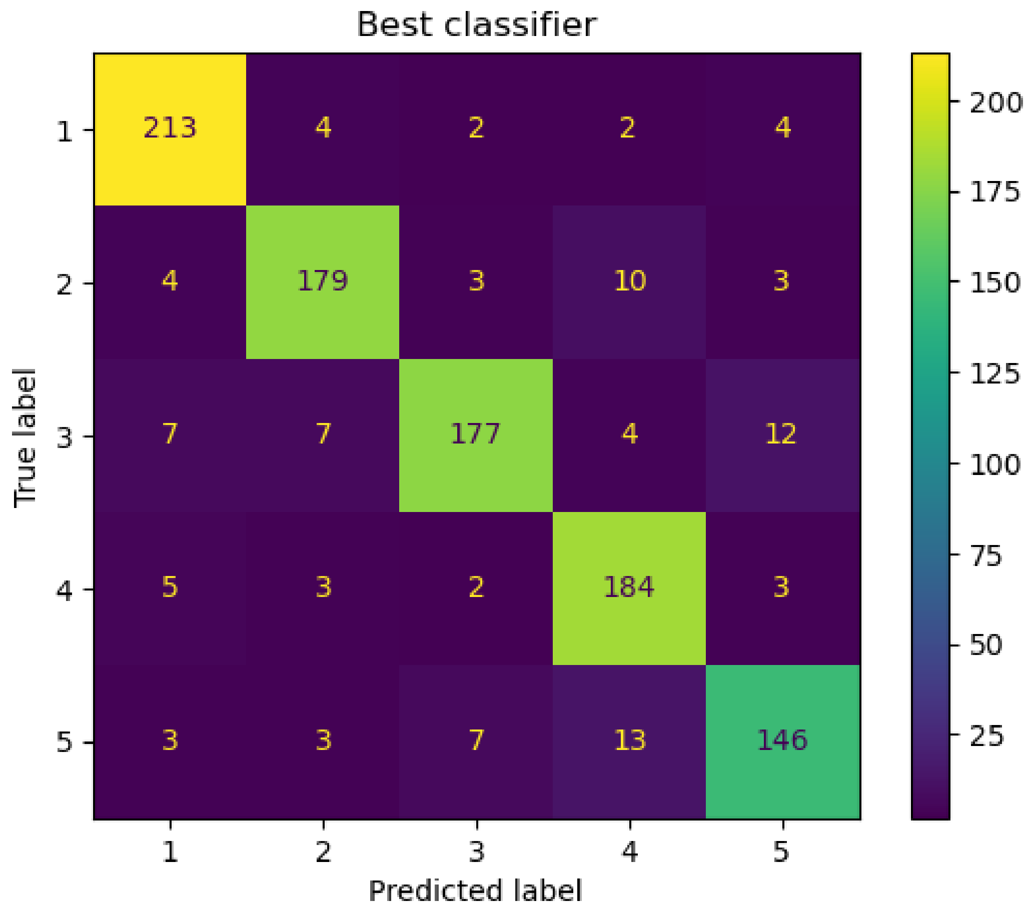
- Starting the search, we both (the authors) investigated it from a different angle, and we experimented both with linear and non-linear classifiers, but didn’t “mess” with the data.
- We found interesting that the naïve bayes outperformed non-linear classifiers like k-nn. And this was the point where we started processing the data. In the end the chosen model is the best that we came with but this out of pipeline search will likely return suboptimal results.
- Closing on the SVM, which in the end attracted our focus, we performed an exhaustive search on a grid of C and γ values, with 80% , 95% PCA and no dimension reduction. From the results it became obvious what later came to be explained by the descriptive statistics of the features, the 80% PCA was giving the best results for most of the reasonable parameter pairs.
- The C and γ values were 8 and 2^{-8} respectively. The values that we searched range from 2^{-5} to 2^{15} for C with step every 2 powers of 2 and 2^{-15} to $2^{4.75}$ with step every quarter of power of 2.

The chosen Model

- After choosing the parameters of the SVM, we started processing the data further, but since time was limited, we didn’t try to find the best combination of data processing and model fitting. So, we just tuned the data processing on the already chosen model.
- The model achieved a 5-fold cross-validation accuracy of 90.18% and then it was fitted on the worst fold where it achieved 89.9% accuracy in order to obtain further metrics and examine how it performed.

	1	2	3	4	5	Average
Recall	94.67	89.95	85.51	93.4	84.88	89.68
Precision	91.81	91.32	92.67	86.36	86.91	89.82
F1	93.21	90.63	88.95	89.76	85.88	89.69

- We observe that despite the balancing weights the classifier is both struggling to classify 5 and is wrongly classifying 5 as 3.
- Perhaps a different method of managing the imbalance in the set should have been used. Although the weights achieve at least to not overclassify 1 over the other 3.
- We also observe that it overclassifies 4 which according to relative ratios is the second least common class.
- In the end it achieves pretty good precision on the first 3 as well as recall.



True label \ Predicted label	1	2	3	4	5
1	213	4	2	2	4
2	4	179	3	10	3
3	7	7	177	4	12
4	5	3	2	184	3
5	3	3	7	13	146

Confusion Matrix of the SVC

Conclusions

- In the end not knowing the exact nature of the problem, we can only make assumptions about the adequacy of the selected classifier.

- It certainly isn’t the best to tackle this problem, since due to limited resources, time and out of order approach is most likely suboptimal, but it’s the best our process came with.

- The steps of our process are justifiable by the outcome and were chosen based on facts.