# Statistic Machine Learning

# SVM with SMO

# Presented By: Nadav Gover

**Part A**

- In this part we implemented a Sequential Minimal Optimization (SMO) algorithm designed by J. C. Platt and described in *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines, 1998.*

  The strategy for picking up the multipliers to optimize was chosen with aid of this literature. Choosing these pairs is critical for the speed of the algorithm, since there are $N(N-1)/2$ possible choices and some will result in much less improvements than others.
  In order to speed convergence, SMO uses heuristics to choose which two Lagrange multipliers to jointly optimize. There are two separate choice heuristics: one for the first Lagrange multiplier and one for the second. The choice of the first heuristic provides the outer loop of the SMO algorithm. The outer loop first iterates over the entire training set, determining whether each example violates the KKT conditions. If an example violates the KKT conditions, it is then eligible for optimization. After one pass through the entire training set, the outer loop iterates over all examples whose Lagrange multipliers are neither 0 nor C (the non-bound examples). Again, each example is checked against the KKT conditions and violating examples are eligible for optimization. The outer loop makes repeated passes over the non-bound examples until all of the non-bound examples obey the KKT conditions within $\epsilon$. The outer loop then goes back and iterates over the entire training set. The outer loop keeps alternating between single passes over the entire training set and multiple passes over the non-bound subset until the entire training set obeys the KKT conditions within $\epsilon$, whereupon the algorithm terminates. The first choice heuristic concentrates the CPU time on the examples that are most likely to violate the KKT conditions: the non-bound subset. As the SMO algorithm progresses, examples that are at the bounds are likely to stay at the bounds, while examples that are not at the bounds will move as other examples are optimized. The SMO algorithm will thus iterate over the non-bound subset until that subset is self-consistent, then SMO will scan the entire data set to search for any bound examples that have become KKT violated due to optimizing the non-bound subset.
  Once a first Lagrange multiplier is chosen, SMO chooses the second Lagrange multiplier to maximize the size of the step taken during joint optimization. SMO keeps a cached error value E for every non-bound example in the training set and then chooses an error to approximately maximize the step size.
  This can be seen in the code in function *examine_example*

- The convergence criteria is epsilon $\epsilon$. In the code it is defined as $\epsilon = 10^{-3}$ as a default but it can also be altered by the user and will be discussed two bullets below. This convergence criteria is a rule of thumb convergence criteria and is not very loose. Since this is the case (not so loose epsilon) we can't use too big C's otherwise the convergence will take a long time. The C is chosen to be between

  $$2^{-6} \leq C \leq 2^{7}$$

  Which has good convergence speed with this convergence criteria.

- The threshold b is re-computed after each step, so that the KKT conditions are fulfilled for both optimized examples. The following threshold b1 is valid when the new $\alpha 1$ is not at the bounds, because it forces the output of the SVM to be y1 when the input is x1:

$$b_1 = E_1 + y_1(\alpha_1^{new} - \alpha_1)K(\overrightarrow{x_1}, \overrightarrow{x_1}) + y_2\left(\alpha_2^{new,clipped} - \alpha_2\right)K(\overrightarrow{x_1}, \overrightarrow{x_2}) + b$$

The following threshold b2 is valid when the new α2 is not at bounds, because it forces the output of the SVM to be y2 when the input is x2:

$$b_2 = E_2 + y_1(\alpha_1^{new} - \alpha_1)K(\overrightarrow{x_1}, \overrightarrow{x_2}) + y_2\left(\alpha_2^{new,clipped} - \alpha_2\right)K(\overrightarrow{x_2}, \overrightarrow{x_2}) + b$$

When both b1 and b2 are valid, they are equal. When both new Lagrange multipliers are at bound and if L is not equal to H, then the interval between b1 and b2 are all thresholds that are consistent with the KKT conditions. SMO chooses the threshold to be halfway in between b1 and b2.

This calculation can be seen in *take_step* function in the code.

- A bit about the code. In this part we built an API for SVM with SMO. I think the best way to understand the code is to see and example of usage:

  First the user needs to import the data set. The import function resides in part b

  *iris = import_data()*

  After we this, the user needs to instantiate a class of the SVM with SMO algorithm.

  This means choosing a kernel (which is either linear or RBF). Another interesting parameter to be entered is the epsilon, which is the convergence criteria. The default is $10^{-3}$ but the user can choose to loosen it or tighten it.

  *svm = SvmWithSmo(iris, kernel="rbf", epsilon=1e-3)*

  Since this is a multiclass implementation of a binary classifier, we have a training function that trains all the classes. For your convenience it is also making predictions and gets the test results which is true positive, true negative, false positive, false negative. All in matrix form.

  *predictions, test_results = svm.train_and_predict_all_classes()*

  After we have the predictions, we can now compute the confusion matrix

  *confusion_matrix = svm.test_all_classes(predictions)*

  At last we can visually see the results by plotting it

  *svm.plot_confusion_matrix_and_tables(confusion_matrix=confusion_matrix,*
  *test_results=test_results, show_plots=True)*

  The script automatically splits the data into train, test and validation sets. If the user wants to change the proportion of each set from the whole set he should alter the split_data function. In addition it finds the optimal C and gamma (in case of RBF kernel) to be used, see part b for more explanation

  There is also an example of usage in the code itself on the bottom.

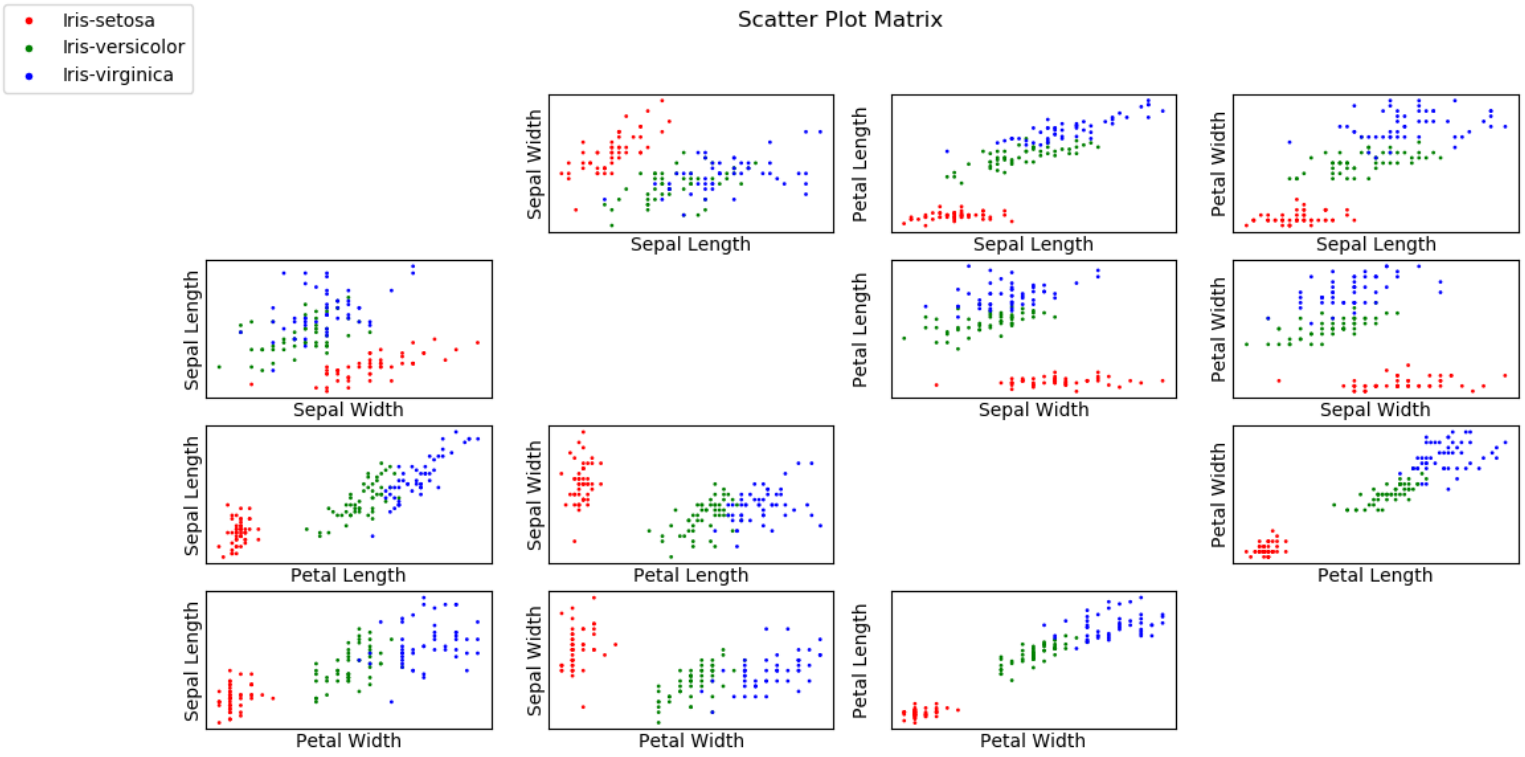
  Important note: for the import data to work you should put the iris.csv in the same folder as the script or else give it a path as a parameter (path=path_to_iris.csv)

## **Part B**

In this part we ran our code from part a on the Iris data set. It consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor), so 150 total samples. One

class is linearly separable from the other two; the latter are not linearly separable from each other. Four features were measured from each sample: the length and the width of sepals and petals, in centimeters.

**(a)** The scatter plot matrix of the data is as follows:




From this plot we can visually see that the flower species that seems to be most separable is Iris Setosa and the worse seems to be Iris Versicolor.

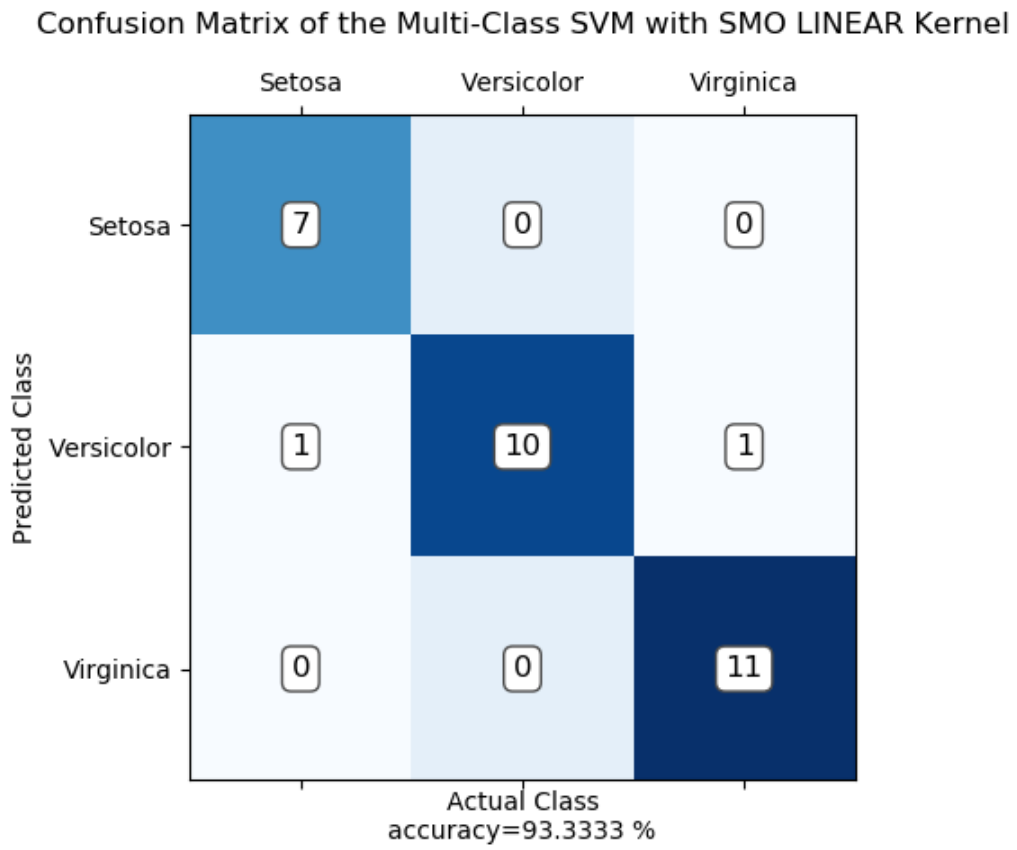
**(b)** The script automatically splits the data into train, test and validation sets by the rule of thumb of 60%, 20%, 20% Respectively.
In addition, by assigning each class to a number, we can use argmax as in PA1 to predict the class of the multiclass classifier.

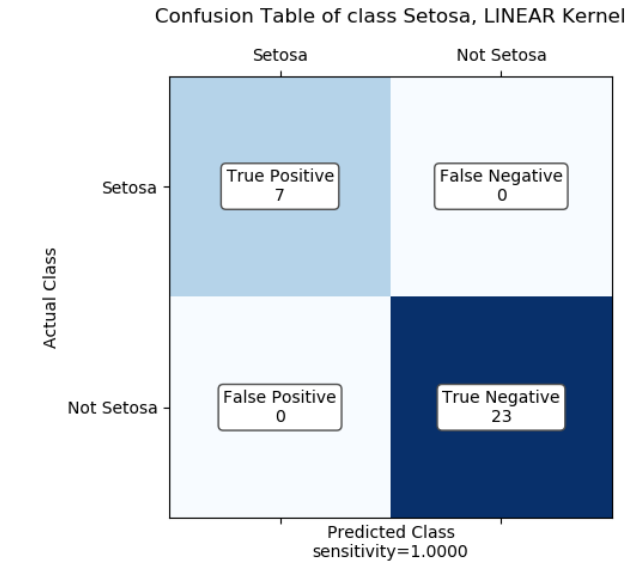
**(c+d)** In this section we used the SVM with SMO algorithm to train the three binary SVM's with a linear kernel.

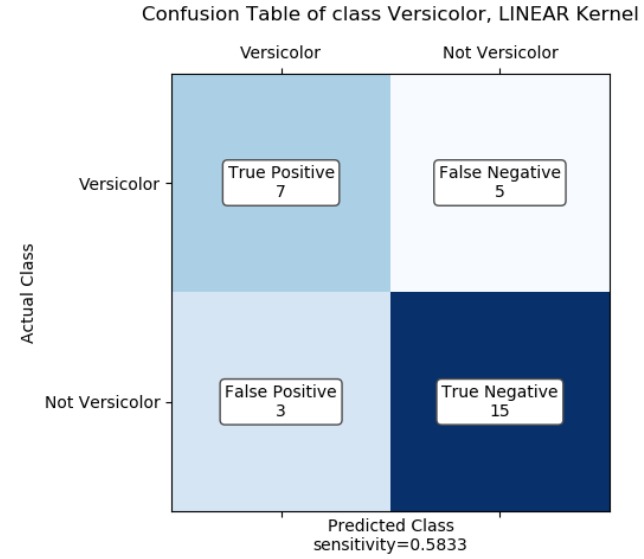
The optimal C was found using the validation set. The method that was chosen is to make a grid search. In particular, a logarithmic grid search. This means that C is each iteration $2^i$ where $i$ was

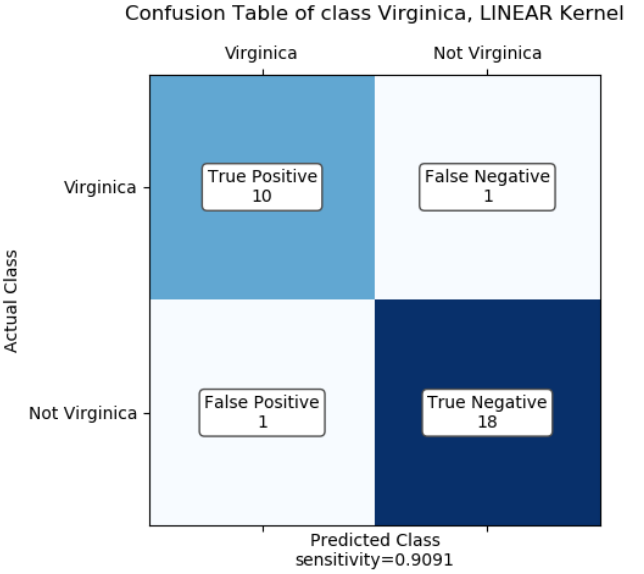
chosen to be $-5 \leq i \leq 7,\ i \in \mathbb{Z}$. This is not too big of C to make the SMO too slow and it yields good results. To make the grid search more efficient, first a coarse search was made, and after the best $i$ was found a finer search was done by increasing $i$ in steps of 0.2 to find the best C.

The results of the linear kernel are following:




Confusion Table of class Setosa, LINEAR Kernel
Setosa
Not Setosa
Setosa
Not Setosa
Actual Class
True Positive
7
False Negative
0
False Positive
0
True Negative
23
Predicted Class
sensitivity=1.0000


Confusion Table of class Versicolor, LINEAR Kernel
Versicolor
Not Versicolor
Versicolor
Not Versicolor
Actual Class
True Positive
7
False Negative
5
False Positive
3
True Negative
15
Predicted Class
sensitivity=0.5833


Confusion Table of class Virginica, LINEAR Kernel
Virginica
Not Virginica
Virginica
Not Virginica
Actual Class
True Positive
10
False Negative
1
False Positive
1
True Negative
18
Predicted Class
sensitivity=0.9091

In the above results we can see some interesting facts.

First, as we remember from part B section a, the data that is probably most linearly separable is of class Iris Setosa and the least one is Iris Versicolor. As we can see from the values of the sensitivity in the confusion tables, this theory seems to be correct. The sensitivity of Setosa is 1 while the sensitivity of Versicolor is 0.58. This fits with the theory that the linear kernel is not a good choice in the case that the data is not linearly separable.

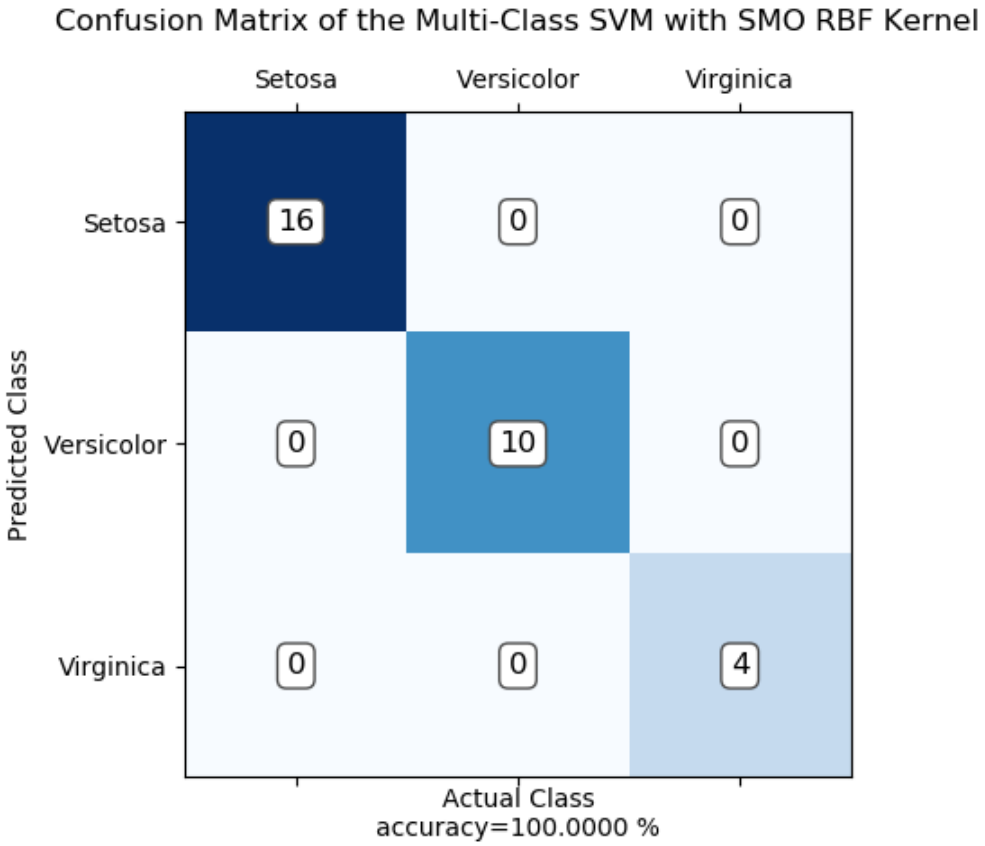
Finally we got an overall accuracy of 93.333% as can be seen in the confusion matrix. It is also worth to mention that this was achieved in a very short (~10 seconds) including optimizing C. This proves that SMO is very efficient while still getting great results. Also, the data set is considered very small (90 examples for train set, 30 for test and validation sets) and still we got great results.

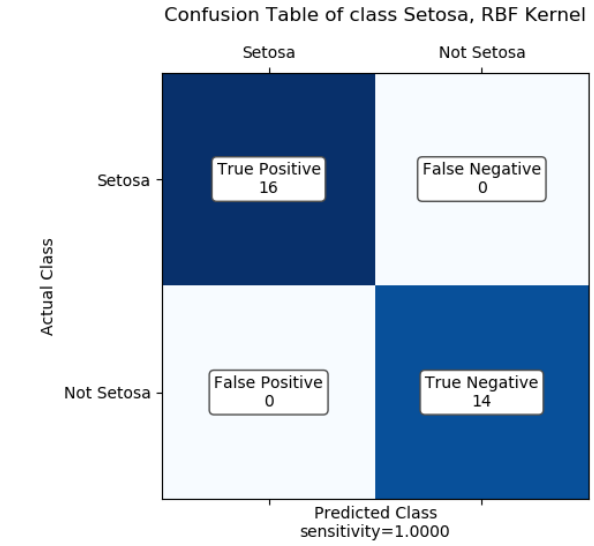
**(e)** In this section we used the SVM with SMO algorithm to train the three binary SVM's with RBF kernel.

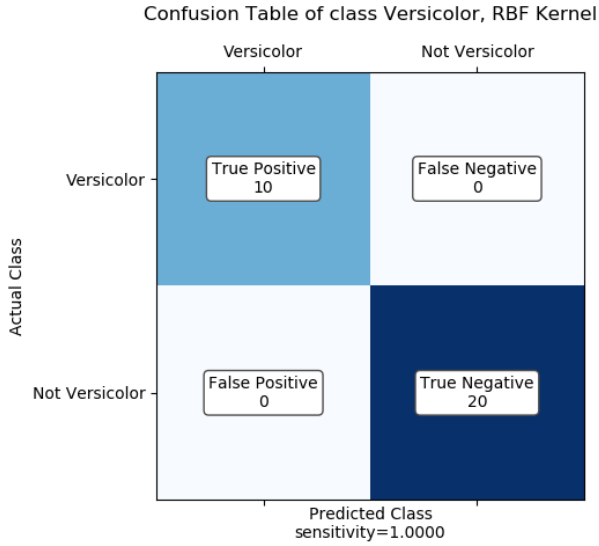
The optimal C was found using the validation set as before.

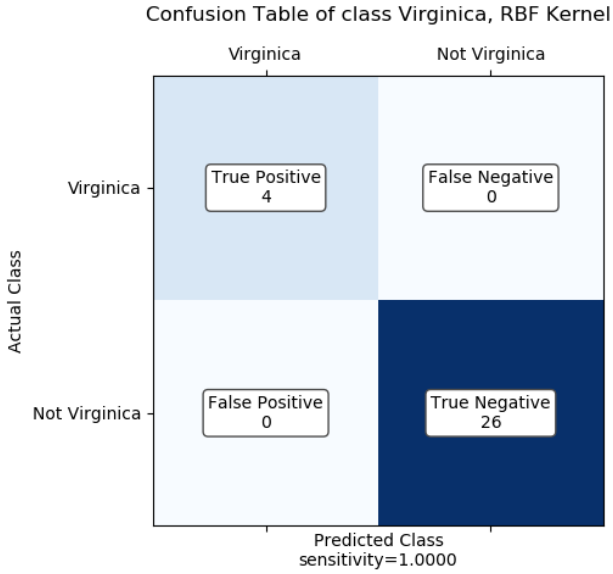
The optimal gamma was found using the validation set. And the same method for finding the optimal C was applied to find the optimal gamma

The results of the RBF kernel are following:




Confusion Table of class Setosa, RBF Kernel
Setosa
Not Setosa
Setosa
Not Setosa
Actual Class
True Positive
16
False Negative
0
False Positive
0
True Negative
14
Predicted Class
sensitivity=1.0000


Confusion Table of class Versicolor, RBF Kernel
Versicolor
Not Versicolor
Versicolor
Not Versicolor
Actual Class
True Positive
10
False Negative
0
False Positive
0
True Negative
20
Predicted Class
sensitivity=1.0000


Confusion Table of class Virginica, RBF Kernel
Virginica
Not Virginica
Virginica
Not Virginica
Actual Class
True Positive
4
False Negative
0
False Positive
0
True Negative
26
Predicted Class
sensitivity=1.0000

Well, we got a 100% on everything. Looks perfect, right?

In fact, this is not always the case. Due to randomization of the data set and which example goes to which set (train, test, validation), these results can change. But in Practice (after running the algorithm again) the accuracy is larger than 96% which is still a great result.

We can see here the difference between the linear kernel and the RBF kernel. The RBF kernel is more suitable for non-linearly separable data from its nature. While the linear kernel preformed well on linearly separable data, the performance was poorer for not separable data. But as we can see from the RBF kernel, it preformed well and yields good results for both types of data. Although another step is required to optimize gamma, it is still worth it because this process takes only several seconds thanks to the efficiency of the SMO. Overall the whole evaluation of RBF kernel took ~10 seconds.

## Conclusion

In this assignment we trained models with SVM using SMO with linear and RBF kernels.
To compare those two models, consider those two criteria: run time and performance.
The run time of both models was approximately 10 seconds. The differences between the run times can be neglected because they are very small.
Performance in the sense of accuracy the RBF kernel achieved in average 5% more than the linear kernel. The RBF kernel preformed well on both the linearly separable data and the non-linearly separable data while the linear kernel preformed well only on the linearly separable data.
This concludes this assignment.